Course notes for CSC C73

ALGORITHM DESIGN & ANALYSIS

Vassos Hadzilacos
University of Toronto

CSCC73 Course Notes © 2026 by Vassos Hadzilacos is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International. To view a copy of this license, visit CC BY-NC-ND 4.0

To the memory of my parents, $\Gamma\iota\acute{\omega}\rho\gamma o\varsigma$ and $M\alpha\acute{\iota}\rho\eta$,
in gratitude for the roots and the wings.

ACKNOWLEDGEMENTS

I became fascinated with computer science during my undergraduate studies thanks to the subject of this course and a companion course on formal languages, computability, and computational complexity. One debt of gratitude therefore goes to my teachers in these courses, Ken Steiglitz and Jacobo Valdes. Another debt of gratitude goes to my colleagues in the computer science department's theory group at the University of Toronto from whose wisdom and knowledge I benefited a great deal; especially, as regards the topics in these notes, to Allan Borodin and Charles Rackoff.

There are several excellent textbooks on algorithms from which I gained knowledge and much joy, and whichwere valuable in preparing these notes. Like many computer scientists of my generation, my first contact with this subject was through Aho, Hopcroft, and Ullman's *Design and Analysis of Computer Algorithms*. In several iterations of the course I relied on Kleinberg and Tardös's *Algorithm Design*; and Dasgupta, Papadimitriou, and Vazirani's *Algorithms*. These two books have influenced the presentation in these notes the most. Two other excellent books on the same subject are the encyclopedic *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein, and *Algorithms* by Erickson which, among its many other virtues, is *legally* available for free on the web. I recommend all of these sources for further study as they cover much more material than can fit in the twelve weeks of this course and contain many exercises that help tremendously in deepening one's mastery of the subject. (A topic that is covered in most of these books but is not covered in this course is NP-completeness; in our curriculum this topic is covered in CSCC63, Computability and Computational Complexity.)

I am grateful to my CSCC73 students over the years: Their questions and feedback helped sharpen my understanding of the material and shape the way it is presented in these notes.

While every sentence in these notes was written by me, and I am therefore solely responsible (or, perhaps I should say, irresponsible) for all errors, I found sessions with Anthropic's Claude AI system very helpful in exploring some topics and tracking down relevant literature, especially in Chapters 6 and 7. These notes are an on-going project, and I will be grateful for feedback from readers. I can be contacted by email at `vassos@cs.toronto.edu`.

Vassos Hadzilacos
Toronto, August 2026

Preface

Computer Science is a discipline with very wide scope. At its core, however, is the concept of ***algorithm***: a method for solving a problem by transforming inputs (instances of the problem) to outputs (answers to those instances) that is specified so precisely that it can be carried out mechanically by a computer.

The primary requirements for an algorithm are *correctness* and *efficiency*. Correctness means that the algorithm transforms *every* input to an *appropriate* output, as required by the problem it solves; and efficiency means that it does so quickly.[1] Thus, learning techniques one can apply when designing a new algorithm, methods for convincing oneself and others that this algorithm does its job correctly, and reasoning about its efficiency, are critical parts of a computer science education. This course aims to develop and sharpen these skills.

Of special (though not exclusive) interest to us in this course are algorithms for ***optimization problems***. In such problems we seek a solution that satisfies certain conditions, called ***constraints***, and optimizes (i.e., minimizes of maximizes, depending on the problem) an ***objective function***. Solutions that satisfy the constraints are called ***feasible***. An example of an optimization problem you have studied before is the minimum spanning tree problem. The input here consists of (a) an undirected, connected graph $G = (V, E)$ and (b) an edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$; the output is a set $T \subseteq E$ that

- (feasibility) forms a spanning tree of G, and
- (optimality) minimizes the sum of the weights of the edges in T; i.e., for every spanning tree T' of G, $\sum_{e \in T} \mathbf{wt}(e) \leq \sum_{e \in T'} \mathbf{wt}(e)$.

In this course we will study seven topics related to the design of efficient algorithms. The first three topics concern three widely applicable algorithmic techniques: ***greedy*** (Chapter 1), ***divide-and-conquer*** (Chapter 2), and ***dynamic programming*** (Chapter 3). You have encountered these techniques in prerequisite courses, but here we will study them more systematically. Then we will look at three problems: the ***shortest paths problem*** (Chapter 4), the ***maximum flow*** problem (Chapter 5), and ***linear programming*** (Chapter 6). These are not algorithmic design techniques *per se*; rather, they are specific problems whose importance lies in the fact that many other problems reduce to them. That is, we can use solutions to these problems as "black boxes" to solve other problems. The ability to reduce one problem to another is an important skill in the algorithm designer's toolbox. Finally we look briefly

[1] There are other criteria of efficiency, for example that the algorithm uses memory, or energy, sparingly, but in this course we will focus mostly on time efficiency.

at ***approximation algorithms*** (Chapter 7): These are algorithms for optimization problems that we do not know how to solve efficiently in general, and so we sometimes settle for algorithms that find suboptimal but "good enough" feasible solutions.

A third-year course on algorithms is fairly standard in computer science curricula. There are many ways one could teach such a course, with different emphases depending on one's goals. These notes are based on (the evolution of) how I have been teaching this course at the University of Toronto's Scarborough Campus for nearly two decades. The emphasis is on conceptual understanding of techniques and of individual algorithms, rather than on coding or on selection of data structures. It is not that I consider these issues unimportant, but emphasis on them can obscure the essential ideas behind an algorithm. For this reason algorithms are presented in pseudocode that should be straightforward for a third-year computer science student to turn into actual code in the programming language of their choice, using appropriate and efficient data structures learned in prerequisite courses.

Another aspect of the pedagogy reflected in the notes is insistence on correctness proofs. The reason for this is twofold: *First*, natural ideas that, at first blush, seem to work, sometimes don't. The most reliable way to convince oneself *and others* that an algorithm works, is to prove that it does. *Second*, one often gains a much deeper understanding of an algorithm by examining (and even more productively, *constructing*) its proof of correctness. It is not enough to understand *how* an algorithm works, one must also understand *why* it works. The depth of understanding one gains as a result makes it easier to remember the algorithm and, more importantly, to apply similar ideas in new contexts. For these reasons, learning how to reason about algorithms and how to present that reasoning, are central goals of this course. The manner in which such proofs are presented in these notes is consistent with these goals: I am not interested in pedantry: if an argument is straightforward, I do not dwell on it in the proof. But when you, as the reader, come across a statement such as "a straightforward argument shows that...", you should momentarily pause and convince yourself that you see, in your mind's eye, how such an argument would go.

The preceding comments highlight the centrality of this course in a computer scientist's education. Nevertheless, some students approach the course with trepidation: it has a reputation of being a tough one. I will not deny that the course has its challenges: it requires careful thinking, attention to detail and, most importantly, creativity. But these are important skills to possess, no matter where your career takes you, and few courses afford the opportunity to develop them as deeply as this course will. It's not all about challenges, however; there is also fun to be had! Discovering an efficient and clever algorithm provides a satisfaction akin to solving to a puzzle.

Background preparation

This course is required in our computer science specialist program and it is typically taken by students in their third year. Students have already taken two second-year courses with highly relevant material: A course (CSCB36) that covers, among many other topics, the use of induction in reasoning about the correctness of iterative and recursive programs; and a course

(CSCB63) that covers data structures beyond arrays and lists (such as balanced search trees and priority queues) and graph search algorithms with some applications. In the latter course students learn how to analyze the running time of algorithms at a moderately sophisticated level (in some cases, more sophisticated than what we will encounter in this course) so they are well acquainted with asymptotic notation by the time they take this course.

Contents

1 Greedy algorithms **1**
- 1.1 Interval scheduling . . . 1
- 1.2 Minimum lateness scheduling . . . 6
- 1.3 Fractional knapsack . . . 10
- 1.4 Huffman codes . . . 13
- Exercises for Chapter 1 . . . 20

2 Divide-and-conquer algorithms **27**
- 2.1 General form of divide-and-conquer algorithms . . . 27
- 2.2 Running time of divide-and-conquer algorithms . . . 28
- 2.3 Karatsuba's multiplication algorithm . . . 32
- 2.4 Strassen's matrix multplication algorithm . . . 36
- 2.5 Closest pair of points . . . 38
- 2.6 Order statistics . . . 42
  - 2.6.1 Randomized selection . . . 45
  - 2.6.2 Deterministic selection . . . 47
- 2.7 The fast Fourier transform . . . 50
- Exercises for Chapter 2 . . . 59

3 Dynamic programming algorithms **67**
- 3.1 Longest increasing subsequence . . . 67
- 3.2 Weighted-interval scheduling . . . 73
- 3.3 Edit distance . . . 75
- 3.4 Optimal binary search trees . . . 80
- 3.5 Discrete knapsack . . . 84
- 3.6 Dynamic programming and recursion . . . 87
- Exercises for Chapter 3 . . . 89

4 Shortest paths **99**
- 4.1 Introduction . . . 99
- 4.2 Dijkstra's algorithm . . . 103
- 4.3 The Bellman-Ford algorithm . . . 108
- 4.4 The Floyd-Warshall algorithm . . . 113

4.5 Johnson's algorithm . . . 117
Exercises for Chapter 4 . . . 121

5 Maximum flow and applications 125
5.1 Flow networks and the maximum flow problem . . . 125
5.2 Augmenting a flow . . . 128
5.2.1 Residual graphs and augmenting steps . . . 130
5.3 The Ford-Fulkerson algorithm . . . 133
5.3.1 Termination and running time . . . 134
5.3.2 Correctness and the max-flow min-cut theorem . . . 136
5.3.3 Speeding up the Ford-Fulkerson algorithm . . . 137
5.4 Maximum matching in bipartite graphs . . . 143
5.5 Minimum vertex cover in bipartite graphs . . . 146
5.6 Hall's Theorem . . . 149
5.7 Maximum cardinality set of disjoint paths . . . 150
5.7.1 Flow decomposition . . . 151
5.7.2 Directed graphs . . . 154
5.7.3 Undirected graphs . . . 157
5.7.4 Menger's theorem . . . 157
Exercises for Chapter 5 . . . 159

6 Linear programming 165
6.1 What is linear programming? . . . 165
6.1.1 Cost minimization — the "diet problem" . . . 165
6.1.2 Profit maximization — the "product selection problem" . . . 166
6.1.3 General form of a linear program . . . 166
6.2 The geometry of linear programs . . . 168
6.3 Linear programming algorithms . . . 175
6.4 Reductions to linear programming . . . 179
6.4.1 Maximum flow . . . 179
6.4.2 Minimum cost flow . . . 181
6.4.3 The transportation problem . . . 182
6.4.4 Shortest paths . . . 183
6.4.5 Linear regression . . . 188
6.5 Integer and 0-1 linear programming . . . 192
6.5.1 Discrete knapsack . . . 193
6.5.2 The facility location problem . . . 193
Exercises for Chapter 6 . . . 195

7 Approximation algorithms 199
7.1 What they are and why we need them . . . 199
7.2 Minimum vertex cover . . . 200
7.2.1 LP relaxation and rounding . . . 200

7.2.2 A greedy 2-approximation algorithm for minimum vertex cover 203
7.3 Minimum set cover 204
7.4 Minimum makespan scheduling 207
7.4.1 A greedy algorithm 207
7.4.2 The longest-first greedy algorithm 209
7.5 The k-centre problem 213
7.6 Discrete knapsack 218
7.7 Maximum cut 220
Exercises for Chapter 7 222

A Brief review of complex numbers **225**

Chapter 1

Greedy algorithms

A greedy algorithm constructs its answer in a series of stages. In each stage it extends *greedily* and *irrevocably* the partial answer it has constructed so far. By "greedily" I mean that, of the possible extensions to the current partial answer, it chooses one that results in what appears to be best *for the next stage*, without looking ahead to future stages. (For this reason a more apt term for this kind of algorithm would be "myopic", but the term "greedy algorithm" is now well established.) By "irrevocably" I mean that the algorithm never backtracks to undo an earlier decision; each stage strictly extends the previous partial answer. The above is not meant as a technical definition but as an informal description that will help you recognize a greedy algorithm when you see one, and consider this technique as an option when you are seeking an algorithmic solution for a new problem.

As you might expect, not all problems are amenable to this kind of solution: More often than not, to find an overall optimal solution one has to look ahead more than one step. When the greedy approach is applicable, however, it results in algorithms that are simple and very efficient, precisely because of the irrevocable nature of the decisions that such algorithms make along the way to compute the answer. The greedy approach is also very useful in the design of so-called ***approximation algorithms***. These are algorithms for optimization problems that compute a feasible solution that is not necessarily optimal but is guaranteed to be close to optimal, in a precise sense. We will examine such algorithms in Chapter 7.

1.1 Interval scheduling

Suppose we have a set of n "jobs", $n \in \mathbb{Z}^+$, which require the use of a "machine" that can only process them one-at-a-time. Each job j, has a start time $s(j)$ and a finish time $f(j) > s(j)$, where $s(j), f(j) \in \mathbb{R}$, and must be executed on the machine during the real interval $\big[s(j), f(j)\big)$. We say that two jobs ***conflict*** if their corresponding intervals intersect; note that if $f(j) = s(j')$, the jobs j, j' do not conflict. We want to find a set consisting of the maximum possible number of jobs that do not conflict and can therefore be processed by the machine.

For example, if the jobs are tasks that a real-time operating system must execute during

specific intervals, we are looking for the maximum number of tasks that can run on a single processor. Or, if the jobs are seminars that can take place in their specified time intervals in some particular room, we are looking for the maximum number of seminars that can be scheduled in that room. More precisely, the ***interval scheduling problem*** is as follows:
Input: A finite set of jobs $\mathcal{J}$, along with the start time $s(j)$ and finish time $f(j)$ of each job $j \in \mathcal{J}$, where $s(j), f(j) \in \mathbb{R}$ and $s(j) < f(j)$.
Output: A maximum cardinality subset A of $\mathcal{J}$ that contains no conflicting jobs. That is, a set $A \subseteq \mathcal{J}$ such that

- (feasibility) for all $j \neq j' \in A$, j does not conflict with j', and
- (optimality) if $B \subseteq \mathcal{J}$ has the property that for all $j \neq j' \in B$, j does not conflict with j', then $|B| \leq |A|$.

We say that a subset A of $\mathcal{J}$ is ***optimal*** if it satisfies these two requirements.

Example 1.1 Consider a set of eight jobs $\{a, b, c, d, e, f, g, h\}$ with endpoints as illustrated in Figure 1.1.

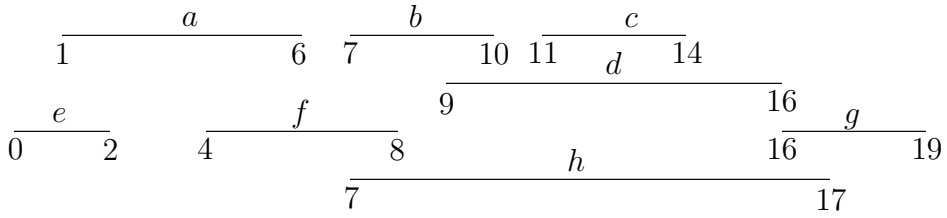


Figure 1.1: Interval scheduling example

A maximum cardinality set of non-conflicting jobs for this example is $\{a, b, c, g\}$. Note that this is not unique: other sets of non-conflicting jobs with the same cardinality are $\{c, e, f, g\}$ and $\{e, b, c, g\}$. Our task is to identify *one* such set, not to enumerate all of them. Enumerating all of them might be a very time-consuming task, as there could be an exponentially many such sets. (As an exercise, construct an example of $2n$ jobs that has 2^n maximum cardinality subsets of non-conflicting jobs.) **End of Example 1.1**

Solving this problem by a brute-force method, i.e., by considering every subset of the jobs in non-increasing size and outputting the first that contains no two jobs that conflict, is terribly inefficient because there is an exponential number of subsets we may have to consider. Following is a generic greedy algorithm for solving this problem; it becomes specific if we choose a particular criterion for ordering jobs in line 3.

```
GenericIntervalScheduling(𝒥)
1 A := ∅
2 while 𝒥 ≠ ∅ do
3     j := next job in 𝒥, according to some criterion for ordering the jobs in 𝒥
4     if j does not conflict with any job in A then A := A ∪ {j}
5     𝒥 := 𝒥 − {j}
6 return A
```

It is straightforward to argue that this algorithm returns a subset A of the jobs such that (a) A is feasible (i.e., it does not contain two conflicting jobs), and (b) no proper superset of A is feasible. (What loop invariant allows you to reach this conclusion?) Note, however, that (b) does not imply that A is optimal; all it says is that A cannot be extended to a bigger feasible subset, and this is a weaker property. Whether A is optimal very much depends on the criterion used for choosing the next job j in line 3. What criterion might we use?

One idea is to pick the job with the *smallest start time*. This is easy to dismiss by the following counterexample: The job that starts first conflicts with an arbitrarily large number of jobs that do not conflict with each other; thus this version of the greedy algorithm can produce a set that is arbitrarily worse than optimal.

In this counterexample the problem is caused by a long job that blocks many short ones, so we might consider another version of this greedy algorithm where the next job j picked in line 3 is a *shortest* one. The intuition why this might work is that a short job is involved in few conflicts and so including it in the set A will rule out few others. But it is again not hard to come up with a counterexample: A short job can conflict with two longer but non-intersecting ones. Although still incorrect, this version of the greedy algorithm is much better than the previous one: it is guaranteed to output a set that is at least half the size of the optimal one (see Exercise 1.1) and not arbitrarily smaller than it.

Inspired by the example that demonstrates that the shortest-job-next greedy algorithm is also incorrect, we might take the next job to be one with the *fewest conflicts*. Alas, this does not work either. As a first attempt at a counterexample consider the set of jobs illustrated in Figure 1.2(a). The number below each job is the number of jobs with which it conflicts. Thus, the fewest-conflicts-next greedy algorithm would first consider one of b, d, or g. If it picks b first, you can check that it will output a set of three jobs (consisting of b, one of a, d, or h, and one of c, g, or i); this is not optimal since $\{d, e, f, g\}$ is a set of four non-conflicting jobs.

This counterexample is simple but it is not entirely satisfactory: If the algorithm first picked one of the other two jobs that have two conflicts, namely d or g, it could in fact output an optimal set of jobs. So it is conceivable that the fewest-conflicts-first greedy algorithm, enhanced with a more nuanced choice in case of ties might work. To convince ourselves that there is no hope of making this idea work we need a counterexample that foils the algorithm *no matter how ties are resolved*. We can modify our unsatisfactory counterexample to do so by adding two more jobs, as shown in Figure 1.2(b). Now there is only one choice for the job with fewest conflicts, namely b, and that choice rules out the optimal answer $\{d, e, f, g\}$. Although it does not guarantee an optimal answer, the fewest-conflicts-first algorithm ensures that the answer is at least half the size of optimal, like the shortest-job-first greedy algorithm.

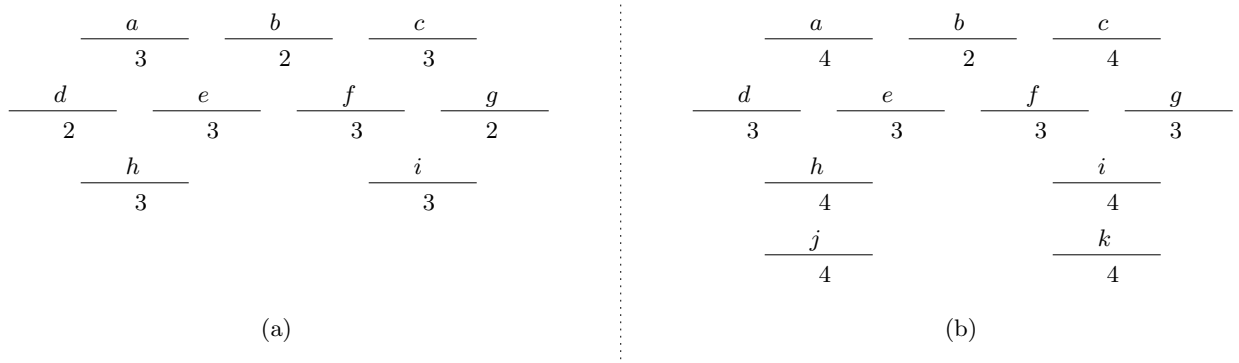


Figure 1.2: Counterexample for fewest-conflicts-next greedy algorithm for interval scheduling

Finally, here is a choice for the next job that works: choose a job with the *earliest finish time*. The intuition behind this is that this choice leaves the machine available for other jobs for the longest time. But, as we saw with the previous two choices, intuition why an idea might work is not enough: we need proof. Before doing that, let's write the algorithm as follows:

```
OptimalIntervalScheduling(𝒥)
1 sort 𝒥 by increasing finish time, ties resolved arbitrarily
2 A := ∅        ▶ partial answer so far
3 F := −∞       ▶ maximum finish time of any job in A
4 for each j ∈ 𝒥 in sorted order do
5     if s(j) ≥ F then A := A ∪ {j}; F := f(j)
6 return A
```

Throughout these notes, if x is a variable in an algorithm that is modified in a loop we will denote by x_i the value stored in that variable at the end of the i-th iteration of the loop, with the convention that x_0 denotes the value stored in x just before the loop is entered. It is straightforward to prove (and you should be sure you know how) that this algorithm satisfies the following invariants:

Lemma 1.1 *A_i contains no conflicting jobs.*

The next lemma captures the greedy nature of the algorithm. It states that the partial solution A_i is "promising", in the sense that it is contained an optimal set.

Lemma 1.2 ("Promising set lemma") *For each $i \in [1..|\mathcal{J}|]$, there is some optimal set A^* such that $A_i \subseteq A^*$.*

Proof. By induction on i. The basis $i = 0$ is obvious, since A_0 is empty.

For the induction step, assume (induction hypothesis) that A_i is a subset of some optimal set A_i^* of jobs. We will prove that A_{i+1} is a subset of some optimal set A_{i+1}^* of jobs. (Note that A_{i+1}^* may be a different optimal set than A_i^*.) This is obvious if no job is added to A in

iteration $i+1$, or if the job j added to A in iteration $i+1$ happens to be in A_i^*: In both of these cases, we take $A_{i+1}^* = A_i^*$ and we are done. So, suppose that some job j is added to A in iteration $i+1$ and $j \notin A_i^*$.

In this case, first we note that A_i^* contains some job j^* that conflicts with j: for, otherwise, we could add j to A_i^* and obtain a feasible set with more jobs than A_i^*, contradicting that A_i^* is optimal. Second, we note that A_i^* does not contain more than one job that conflicts with j: for, otherwise, all such jobs would start after all the jobs in A_i finish (because A_i is, by induction hypothesis, a subset of A_i^*, which is optimal and hence feasible), and the first of them, call it j', finishes before j finishes (otherwise j would not be conflicting with any other job in A_i^*); so the greedy algorithm would have added j', rather than j, to A in iteration $i+1$. So, there is exactly one job j^* in $A_i^* - A_i$ that conflicts with j. Let A_{i+1}^* be the set obtained from A_i^* by replacing j^* with j; i.e., $A_{i+1}^* = (A_i^* - \{j^*\}) \cup \{j\}$. The set A_{i+1}^*, (a) is feasible (since j^* is the only job in A_i^* that conflicts with j); (b) has the same number of jobs as the optimal set A_i^*; and (c) contains $A_i \cup \{j\} = A_{i+1}$ (by construction). So, A_{i+1}^* is an optimal set that contains all the jobs in A_{i+1}, as wanted. □

Theorem 1.3 *The set of jobs returned by* IntervalScheduling *is optimal.*

Proof. By Lemma 1.2, there is an optimal set $A^* \supseteq A_n$. We claim that, in fact, $A_n = A^*$. Suppose, for contradiction, that there is some job $j \in A^* - A_n$. Since A^* is optimal (and hence feasible), j does not conflict with any other job in A^*. Since $A_n \subseteq A^*$ and $j \in A^* - A_n$, j does not conflict with any job in A_n. So, when j was considered for inclusion in A by the algorithm, it would have be added to it, contradicting that $j \notin A_n$. □

We now present another proof of correctness of IntervalScheduling, which captures in a different way the intuition that the greedy algorithm was "doing the right thing" all along as it was constructing the answer. This is sometimes called the "greedy-stays-ahead" approach.

Consider the set A of jobs constructed by the algorithm. By the test in line 3 (and straightforward induction) this set is feasible: no two jobs in it conflict. Let $j_1, j_2, \ldots, j_k$ be the sequence of the jobs in A listed in left-to-right order; this is well defined since the jobs do not conflict. Note that this is also the order in which the algorithm adds these jobs to A. Let A^* be *any* optimal set of jobs and $j_1^*, j_2^*, \ldots j_m^*$ be the sequence of jobs in A^* listed left-to-right (this is also well defined, since A^* is certainly feasible and so its jobs do not conflict).

Lemma 1.4 ("Greedy-stays-ahead" lemma) *For every $t \in [1..k]$, $f(j_t) \leq f(j_t^*)$.*

Proof. By induction on t. The basis $t = 1$ is obvious by the algorithm (the first job chosen by the algorithm is one with minimum finish time).

For the induction step, suppose that $f(j_t) \leq f(j_t^*)$. We will prove that $f(j_{t+1}) \leq f(j_{t+1}^*)$. Suppose, for contradiction, that $f(j_{t+1}^*) < f(j_{t+1})$. This means that j_{t+1}^* was considered by the greedy algorithm before j_{t+1}. We have

$$f(j_t) \leq f(j_t^*) \leq s(j_{t+1}^*) < f(j_{t+1}^*) < f(j_{t+1}). \tag{1.1}$$

(The first inequality is by the induction hypothesis, the second because A^* is feasible and the jobs in A^* are indexed left-to-right, the third because every job starts before it finishes, and the fourth is by the assumption we made for the sake of contradiction.) Thus, the algorithm considered job j^*_{t+1} after j_t and before j_{t+1}, but did not add it to A (otherwise, j^*_{t+1} and not j_{t+1} would have been the $(t+1)$-th job added to A). So j^*_{t+1} starts before j_t finishes, contradicting (1.1). Therefore $f(j_{t+1}) \leq f(j^*_{t+1})$, which completes the induction step. □

Notice that, unlike Lemma 1.2, here we do not reason about the set in A as it evolves in each iteration and compare it to an optimal set that may change from each iteration to the next. Instead, we reason about the final set A as it has been computed at the end of the loop and compare it to *any fixed* optimal set. Now using *this* lemma, we can prove Theorem 1.3.

Alternative proof of Theorem 1.3. Since A is feasible, $k \leq m$. Suppose, for contradiction, that A is not optimal; i.e., $k < m$. So A^* contains a job j^*_{k+1}. By Lemma 1.4, $f(j_k) \leq f(j^*_k)$. Since A^* is feasible and its jobs are indexed left-to-right, $f(j^*_k) \leq s(j^*_{k+1}) < f(j^*_{k+1})$. So (a) the greedy algorithm considers job j^*_{k+1} after job j_k, and (b) j^*_{k+1} does not conlict with any of the jobs in A. Thus, the greedy algorithm should have added it to A, contradicting that A has only k jobs. □

Finally, we turn to the running time analysis of IntervalScheduling. Let $n = |\mathcal{J}|$. Line 1 can be done in $O(n \log n)$ time using any one of several well-known sorting algorithms, such as heapsort or mergesort. The loop in lines 4-5 is repeated n times and each iteration takes $O(1)$ time, for a total of $O(n)$ time. The remaining steps take $O(1)$ time, so the running time of the algorithm is $O(n \log n) + O(n) + O(1) = O(n \log n)$.

Instead of storing the set of jobs in an array we could store it as a heap using the finish time of each job as the key. Then line 1 would be replaced by a BuildHeap operation that takes $O(n)$ time, but obtaining each job in finish-time order in line 4 would require an ExtractMin operation, which takes $O(\log n)$ time. Thus, the overall time of the algorithm is again $O(n \log n)$. Heaps are often used in greedy algorithm implementations.

A generalization of interval scheduling is the ***weighted interval scheduling*** problem. In this version each job has a "weight", represented by a natural number, and we seek to find a set of non-conflicting jobs that has maximum total weight, where the weight of a set of jobs is the sum of the weights of the jobs in it. I do not know of an efficient *greedy* algorithm to solve this problem, but in Chapter 3 we will see an efficient algorithm for it based on a different algorithm design technique called dynamic programming.

1.2 Minimum lateness scheduling

Next we consider a different scheduling problem. We are again given a finite set $\mathcal{J}$ of jobs, but now we are not given the specific interval during which the job must be executed. Instead, for each job $j \in \mathcal{J}$, we are given its ***length*** $\ell(j) > 0$, i.e., the amount of time it takes to do job j, and its ***deadline*** $d(j)$, i.e., the time by which job j should be completed. We are also

given a ***release time*** r, which is the earliest time when the jobs can start getting scheduled. As before, we have a single machine on which to execute these jobs, and this machine can execute the jobs one-at-a-time. Once a job is started, it must execute to completion; it cannot be pre-empted by another job and completed later. A ***schedule*** is a function $S\colon \mathcal{J} \to \mathbb{R}$ that tells us the time when each job j starts; since we are given the length of j, this implicitly also tells us the time when job ends, namely $S(j) + \ell(j)$. Since all jobs must be scheduled after time r and the machine can execute at most one job at a time, we require S to satisfy the following feasibility requirements:

(a) for all $j \in \mathcal{J}$, $S(j) \geq r$, and
(b) for all $j, j' \in \mathcal{J}$, if $j \neq j'$ then the intervals $\big[S(j), S(j) + \ell(j)\big)$ and $\big[S(j'), S(j') + \ell(j')\big)$ do not intersect.

The ***lateness of job*** j ***in schedule*** S, denoted $L(j, S)$, is the amount of time by which j misses its deadline in S (0, if j finished by its deadline); i.e., $L(j, S) = \max\big(S(j)+\ell(j)-d(j), 0\big)$. The ***lateness of schedule*** S, denoted $L(S)$, is the maximum lateness of any job in S, i.e., $L(S) = \max_{j \in \mathcal{J}} L(j, S)$. Our problem is to find a schedule that minimizes the maximum lateness of the given jobs. More precisely, the ***minimum lateness problem*** is as follows:

Input: A release time $r \in \mathbb{R}$; a finite set of jobs $\mathcal{J}$; and for each $j \in \mathcal{J}$, the length $\ell(j) \in \mathbb{R}^{>0}$ and the deadline $d(j) \in \mathbb{R}$ of job j.

Output: A schedule of minimum lateness. That is, a function $S\colon \mathcal{J} \to \mathbb{R}$ that

- (feasibility) satisfies (a) and (b) above, and
- (optimality) for any function $S'\colon \mathcal{J} \to \mathbb{R}$ that satisfies (a) and (b), $L(S) \leq L(S')$.

We will call a schedule that satisfies these properties an ***optimal schedule***.

Example 1.2 Suppose the release time $r = 0$, and we have four jobs, a, b, c, and d, whose lengths and deadlines are given in the table below, which also shows the lateness of each job in each of the four schedules illustrated in Figure 1.3.

j	$\ell(j)$	$d(j)$	$L(j, S_1)$	$L(j, S_2)$	$L(j, S_3)$	$L(j, S_4)$
a	3	6	0	0	0	0
b	2	4	3	1	1	0
c	7	9	6	3	6	6
d	3	7	14	8	1	1

The lateness of schedule S_1 is 14, due to job d, which has a deadline of 7 but does not finish until 21. This schedule has some unnecessary gaps: there is no reason for the machine to be idle during the intervals $[3, 5)$, $[7, 8)$, and $[15, 18)$. We can easily remedy this by shifting all the jobs to the "left", in the same order, so as to close all the gaps. This is shown in schedule S_2, whose lateness is 8, again due to job d, which now finishes at time 15. The closing of the gaps, however, does not ensure that the resulting schedule is optimal. By swapping jobs c and d in S_2 to obtain schedule S_3, we further improve the lateness of the schedule, which is now only 6 and is due to job c. It turns out that this *is* an optimal schedule, for reasons we will see shortly. Schedule S_4, which is obtained from S_3 by swapping jobs a and b, improves the

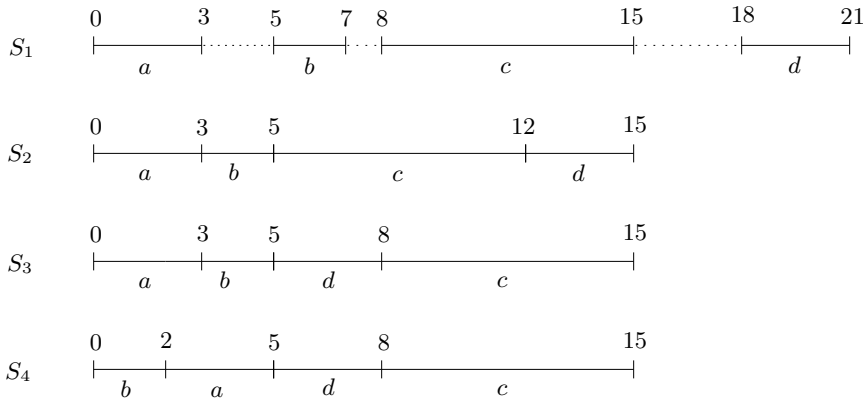


Figure 1.3: Four schedules of the jobs in Example 1.2

lateness of b without increasing the lateness of a, but does not decrease the maximum lateness, which remains 6 still due to job c.

Note that we are only interested in minimizing the maximum lateness of jobs. There are other possible optimality criteria we could have used, such as minimizing the total lateness (i.e., the sum of latenesses of jobs) or the number of jobs that miss their deadlines, but in this version of the problem the only thing that we care about is to minimize the maximum lateness. **End of Example 1.2**

From the discussion in the preceding example it is clear that there is an optimal schedule with no gaps. Therefore we can restrict our attention to schedules with no gaps. Still, if there are n jobs there are $n!$ schedules with no gaps, so a brute-force approach would be extremely inefficient. A natural idea for scheduling tasks with deadlines is to do them in the order of their deadlines: most-urgent-first. Let's apply this idea to our problem; we will schedule the jobs in increasing deadline, starting at the release time, and with no gaps. So we have the following greedy algorithm:

```
MinLatenessSchedule(J, r)
1 sort J by increasing deadline, ties resolved arbitrarily
2 F := r
3 for each j ∈ J in sorted order do
4     S(j) := F
5     F := S(j) + ℓ(j)
  return S
```

Theorem 1.5 *The schedule S returned by* MinLatenessSchedule *is optimal.*

Proof. If S_1 and S_2 are schedules of the same set of jobs $\mathcal{J}$, we define an ***inversion*** between S_1 and S_2 to be a pair of jobs $i, j \in J$ that appear in opposite orders in S_1 and S_2; i.e., $S_1(i) < S_1(j)$ if and only if $S_2(i) > S_2(j)$. We will prove that there is an optimal schedule S^*

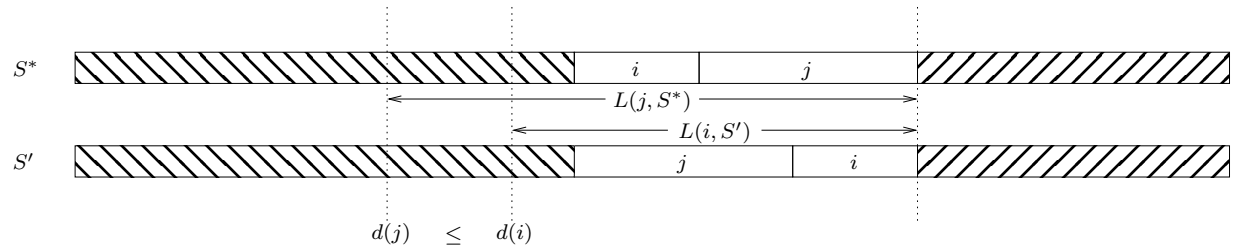


Figure 1.4: $L(i, S')$ versus $L(j, S^*)$

with no gaps in which all jobs appear in the same order as in the schedule S returned by MinLatenessSchedule — i.e., S is equal to an optimal schedule.

Let S^* be an optimal schedule *with the minimum number of inversions between S^* and S*, and let k be that number of inversions. Without loss of generality we can assume that S^* has no gaps. We claim that $k = 0$, in which case $S = S^*$ and we are done.

Suppose, for contradiction, that $k > 0$. Thus there is a pair of jobs i' and j' such that i' appears before j' in S^* but i' appears after j' in S. Then by following the sequence of jobs between i' and j' in S^*, we see that there are two jobs i and j such that i is the job immediately before j in S^*, but i appears after j in S (not necessarily immediately after). Now let S' be the schedule with no gaps obtained from S^* by swapping the order of i and j — see Figure 1.4.

Note that the number of inversions between S' and S is $k - 1$: The inversion between S^* and S caused by i and j is now fixed, with j appearing before i as in S, and no new inversions were created because i and j are next to each other in S^* and S'. Recall that in S jobs appear in order of increasing deadline; therefore $d(j) \leq d(i)$.

All jobs other than i and j have the same lateness in S' as in S^* and the lateness of j in S' is no worse than in S^* since it is moved to an earlier position. The lateness of i in S' may have increased relative to its lateness in S^* (since it is moved to a later position), but it is no greater than the lateness of j in S^*. To see this first note that i finishes in S' at the time when j finishes in S^*. Thus we have:

$$\begin{aligned} L(i, S') &= \max\big(S'(i) + \ell(i) - d(i), 0\big) \\ &= \max\big(S^*(j) + \ell(j) - d(i), 0\big) \\ &\leq \max\big(S^*(j) + \ell(j) - d(j), 0\big) \quad [\text{since } d(j) \leq d(i)] \\ &= L(j, S^*). \end{aligned}$$

Therefore, the maximum lateness in S' is no greater than the maximum lateness in S^*:

$$L(S') = \max_{m \in \mathcal{J}} L(m, S') \leq \max_{m \in \mathcal{J}} L(m, S^*) = L(S^*).$$

Since S^* is optimal and S' is feasible, it must be that S' is also optimal. But there are fewer inversions between S' and S than between S^* and S, contrary to the definition of S^*. □

This proof exemplifies another type of proof that is used to show the correctness of some greedy algorithms, called an ***exchange*** (or ***switching***) argument. In this type of proof we show that any optimal solution can be "massaged" into the solution produced by the algorithm via a sequence of *optimality-preserving tweaks* — in our case, swapping two consecutive "out-of-order" jobs.[1] The exchange argument differs from the "promising set" argument, where we show that the partial solution constructed can always be extended to some optimal solution, and from the "greedy-stays-ahead" argument, where we show that the final answer is at least as good as any optimal solution.

1.3 Fractional knapsack

A thief breaks into a store carrying a knapsack that can hold items with total weight up to $C > 0$. The store contains items $1, 2, \ldots, n$, where item t has value $v_t > 0$ and weight $w_t > 0$. The thief can steal some fraction x_t of any item t, where $0 \leq x_t \leq 1$; the value of this fraction of item t is $x_t \cdot v_t$. The thief must decide what fraction of each item to steal, so as to maximize the total value of the stolen goods, subject to the constraint that their total weight must not exceed the knapsack's capacity C.

More precisely, a ***knapsack*** is a vector $S = (x_1, \ldots, x_n)$ that satisfies the following feasibility constraints: (a) for each item t, $0 \leq x_t \leq 1$, and (b) $\sum_{t=1}^{n} w_t x_t \leq C$ — i.e., the total weight of the (fractions of) stolen items does not exceed the capacity of the knapsack. The ***value*** of S, denoted $\mathcal{V}(S)$, is defined as $\mathcal{V}(S) = \sum_{t=1}^{n} v_t x_t$; that is, the total value of the (fractions of) stolen items. S is an ***optimal*** knapsack if it is a knapsack of maximum value; i.e., for all knapsacks S', $\mathcal{V}(S) \geq \mathcal{V}(S')$. The ***fractional knapsack problem*** can now be specified as follows:

Input: $C > 0$, and $v_t > 0, w_t > 0$ for each $t \in [1..n]$.
Output: An optimal knapsack.

The ***discrete knapsack problem*** (also known as the ***0-1 knapsack problem***) is the version of this problem where we require that $x_t = 1$ or $x_t = 0$, for each item t; in other words, the thief can either steal an item in its entirety or not at all; partial items have no value. This makes sense if the items are, say, shirts (half of a shirt has no value) as opposed to flour (half a kilo of flour has half the value of one kilo of flour). The discrete knapsack problem seems to be computationally very hard. It belongs to the famous class of ***NP-complete problems***.[2] These are problems (a) *none* of which is known to be solvable by an efficient algorithm (i.e., in time that is a polynomial in the size of the input), and (b) *all* of which would have an

[1] The preceding proof obscures the sequence of tweaks that can transform any optimal solution to the greedy solution by focusing on the optimal solution that requires the minimum number of tweaks, and arguing that this number is zero.

[2] More precisely, the ***decision*** version of the 0-1 knapsack problem is NP-complete: Given the capacity C, the values and weights of the items, ***and a target value*** v, is there a knapsack whose total value is at least v? The output here is just "yes" or "no". What is defined above is the ***optimization*** version of the problem, which is obviously at least as hard. The correct terminology is that the 0-1 knapsack optimization problem in ***NP-hard***, not that it is NP-complete.

efficient solution if *any one* of them did! The class of NP-complete problems includes many optimization problems that are important in practice. You will study this class of problems in some detail in your computability and computational complexity course.

In contrast to the discrete knapsack problem, the fractional knapsack problem can be solved by means of a simple and efficient greedy algorithm. Informally, the algorithm is as follows: Consider the items in decreasing value-to-weight ratio, with ties broken arbitrarily. Add whole items to the knapsack one at a time, in this order, until we reach an item whose addition would cause the knapsack's capacity C to be exceeded; add the largest fraction of that item that fits into the knapsack, and stop. This is expressed in pseudocode below:

```
    FracKnapsack(v_1, ..., v_n; w_1, ..., w_n; C)
 1  sort the items so that v_1/w_1 ≥ v_2/w_2 ≥ ... ≥ v_n/w_n
 2  s := 0    ▶ total weight of stolen items in knapsack so far
 3  k := 1    ▶ next item to be considered
 4  while k ≤ n and s + w_k ≤ C do
 5      x_k := 1
 6      s := s + w_k
 7      k := k + 1
 8  if k ≤ n then
 9      x_k := (C − s)/w_k
10      for t = k + 1 to n do x_t := 0
11  return (x_1, ..., x_n)
```

It is straightforward to see that this algorithm can be implemented so that it runs in $O(n \log n)$ time.

Next we will prove that the knapsack returned by this algorithm is optimal. The proof has the flavour of an exchange argument: it effectively shows that we can transform any optimal solution to the solution produced by the algorithm via a sequence of optimality-preserving tweaks. In this case, each tweak amounts to transferring weight used for a higher-indexed item to a lower-indexed item.

Without loss of generality, assume that the items are indexed in the order in which they are sorted by the algorithm; thus,

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

If $\sum_{t=1}^{n} w_t \leq C$ (i.e., the knapsack can accommodate all items), then clearly the optimal knapsack is $(1, 1, \ldots, 1)$, and this is what the algorithm returns in this case. So, in the rest of the proof we assume that $\sum_{t=1}^{n} w_i > C$.

Lemma 1.6 *Let $S = (x_1, \ldots, x_n)$ be a knapsack, and i, j be items such that $i < j$, $x_i < 1$, and $x_j > 0$. Then there is a knapsack $S' = (x'_1, \ldots, x'_n)$ such that $\mathcal{V}(S') \geq \mathcal{V}(S)$, S' is identical to S in all items except i and j, and either $x'_i = 1$ or $x'_j = 0$.*

Proof. Intuitively we construct S' by transferring some of the weight occupied by item j to item i, which is at least as valuable per unit of weight. There are two cases to consider,

depending on whether we can take all of the weight occupied by item j and devote it to stealing more of item i (in which case $x'_j = 0$), or we can take only part of the weight occupied by item j and devote it to stealing all of the rest of item i (in which case $x'_i = 1$).

CASE 1. $w_j x_j \leq w_i(1 - x_i)$ (i.e., we can transfer to item i all of the weight occupied by item j). Then let $S' = (x'_1, \ldots, x'_n)$ where

$$\begin{aligned} x'_t &= x_t, \quad \text{for all } t \neq i, j \\ x'_i &= x_i + (w_j/w_i)x_j \\ x'_j &= 0, \end{aligned}$$

First we verify that this is indeed a knapsack, i.e., it satisfies the two feasibility constraints (a) and (b). For (a), it is obvious that $0 \leq x'_t \leq 1$ for all $t \neq i$, and that $x'_i \geq 0$. Furthermore, by the definition of x'_i and the hypothesis of Case 1, we have

$$x'_i = x_i + (w_j/w_i)x_j \leq x_i + w_i(1 - x_i)/w_i = 1,$$

so $x'_i \leq 1$, as wanted. For (b), by the definition of x'_i and x'_j, we have

$$\sum_{1 \leq t \leq n} w_t x'_t = \left(\sum_{t \neq i,j} w_t x'_t \right) + w_i x'_i + w_j x'_j = \left(\sum_{t \neq i,j} w_t x_t \right) + w_i\big(x_i + (w_j/w_i)x_j\big) = \sum_{1 \leq t \leq n} w_t x_t \leq C,$$

as wanted. It remains to prove that $\mathcal{V}(S') \geq \mathcal{V}(S)$. Indeed, using the definition of x'_i and x'_j we have

$$\mathcal{V}(S') - \mathcal{V}(S) = v_i x'_i + v_j x'_j - v_i x_i - v_j x_j = v_i(x_i + \frac{w_j}{w_i} x_j - x_i) - v_j x_j = \frac{v_i}{w_i} w_j x_j - v_j x_j \geq 0$$

where the last inequality follows from the fact that $v_i/w_i \geq v_j/w_j$ (since $i < j$) by multiplying both sides by $w_j x_j$. Therefore $\mathcal{V}(S') \geq \mathcal{V}(S)$.

CASE 2. $w_j x_j > w_i(1 - x_i)$ (i.e., we can only transfer part of the weight occupied by item j before exceeding the remaining weight of item i). Then let $S' = (x'_1, \ldots, x'_n)$ where

$$\begin{aligned} x'_t &= x_t, \quad \text{for all } t \neq i, j \\ x'_i &= 1, \\ x'_j &= x_j - (w_i/w_j)(1 - x_i), \end{aligned}$$

Using algebra as in Case 1, we can verify that S' is a knapsack, and that $\mathcal{V}(S') \geq \mathcal{V}(S)$. □

The next lemma states something obvious: An optimal knapsack cannot have any unused capacity.

Lemma 1.7 *If $S = (x_1, \ldots, x_n)$ is an optimal knapsack then $\sum_{t=1}^{n} w_t x_t = C$.*

PROOF. We prove the contrapositive: Suppose $\sum_{t=1}^{n} w_t x_t \neq C$. So S has some surplus capacity $c = C - \sum_{t=1}^{n} w_t x_t > 0$. Since $\sum_{t=1}^{n} w_t > C$, it cannot be that $x_t = 1$ for all items t, so there is some item k such that $x_k < 1$. By increasing x_k to $x_k + \min(c/w_k, 1 - x_k)$ (i.e., using up the surplus capacity c to steal more of item k, up to stealing all of it) we obtain a knapsack of greater value that S. (Since $c > 0$ and $x_k < 1$, $\min(c/w_k, 1 - x_k) > 0$, so the fraction of item k is strictly increased.) So S is not optimal. □

Let us define an ***anomaly*** in a knapsack $(x_1, \ldots, x_n)$ to be a pair of items i and j such that $i < j$, $x_i < 1$ and $x_j > 0$. The next lemma states that the knapsack returned by the greedy algorithm has no anomalies and no unused capacity.

Lemma 1.8 *The knapsack $G = (x_1^g, \ldots, x_n^g)$ returned by the greedy algorithm satisfies the following:*
(a) There is some item k such that $x_t^g = 1$ for all items $t < k$, and $x_t^g = 0$ for all items $t > k$.
(b) $\sum_{t=1}^{n} w_t x_t^g = C$.

PROOF. From the algorithm's description it is clear that, for some k, the algorithm assigns $x_k = 1$ to the first $k - 1$ items in line 5, assigns some part (or all) of item k's weight to x_k in line 9, and assigns 0 to all remaining items in line 10. Thus, part (a) of the lemma holds. A straightforward induction shows that, at the end of the i-th iteration of the loop in lines 4–7, $s = \sum_{j=1}^{i} w_j$. Since, by assumption, $\sum_{i=1}^{n} w_i > C$, the algorithm exits the while loop with $i \leq n$. So, by the assignments in lines 9 and 10, $\sum_{i=1}^{n} w_i x_i = C$. □

There is only one knapsack with no anomalies and no unused capacity: if there were two different ones, the 1s in one would be a proper prefix of the 1s in the other, which implies that the first has unused capacity. By Lemma 1.8, that is precisely the knapsack G returned by the greedy algorithm. By Lemmas 1.6 and 1.7, there is an optimal knapsack with no anomalies and no unused capacity.[3] But since there is only one such knapsack, namely G, it follows that G is optimal.

1.4 Huffman codes

All information stored on a computer — whether it is text, sound, image, or video — is stored as a binary string. We think of the information as a string over some alphabet Γ, $|\Gamma| \geq 2$, which could be the set of ASCII characters, the set of Chinese ideograms, the set of sound samples (24-bits per sample for high-resolution sound), or the set of pixel colours (24-bits per pixel for "true colour"). Each symbol $a \in \Gamma$ is encoded by a unique binary string, called the

[3]Note that we are not claiming here that there is only one optimal knapsack; we are claiming that there is only one optimal knapsack *with these properties*. Can you think of an input with multiple optimal knapsacks? Can you think of an input for which there is an optimal knapsack with $0 < x_i < 1$ *and* $0 < x_j < 1$ for some $i \neq j$?

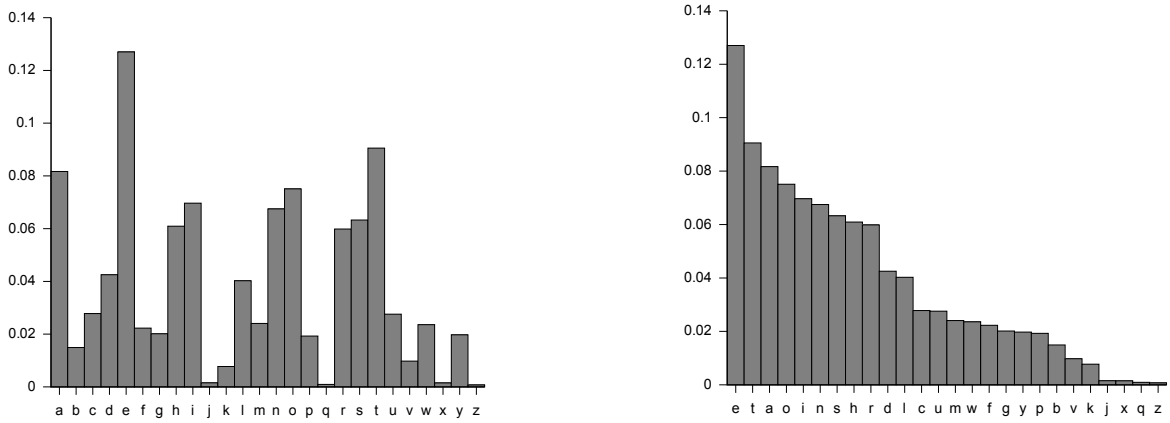


Figure 1.5: Frequency of letters in English
(source: https://en.wikipedia.org/wiki/Letter_frequency, retrieved 2013/09/13)

codeword for a, and a string over Γ is encoded by concatenating the codewords of the string's symbols.

There are two broad types of encoding schemes: ***fixed-length*** and ***variable-length***. In a fixed-length code, each symbol of the alphabet Γ is represented by a unique binary string of the same length $\lceil \log_2 |\Gamma| \rceil$. On the positive side, fixed-length codes are easy to decode and make indexing straightforward: The i-th symbol of the string over Γ is represented by the bits in positions $(i-1)\lceil \log_2 |\Gamma| \rceil + 1$ to $i\lceil \log_2 |\Gamma| \rceil$ of the encoding binary string. On the negative side, fixed-length codes are wasteful because they use the same number of bits to represent each symbol of Γ, regardless of its frequency. For example, Figure 1.5 shows the frequencies of the different letters in English text. A fixed-length code such as ASCII, uses the same number of eight bits to represent the most frequent letter E, which appears about 12.70% of the time, as to represent the least frequent letter Z, which appears only about 0.07% of the time — 180 times less frequently!

A variable-length code uses shorter codewords for more frequent symbols of Γ and longer codewords for less frequent ones, so as to minimize the number of bits used to encode strings. Our goal is to find an optimal encoding, given the frequencies of the symbols. We will explain shortly what we mean by "optimal" in this context, but first we must address a potential problem with decoding variable-length codes.

Suppose our alphabet is $\Gamma = \{A, B, C\}$. If we use the variable-length code

Symbol	Codeword
A	1
B	01
C	010

it is not possible to decode the binary string 0101 unambiguously: it could encode the string CA or the string BB. Unambiguous decoding is of course a requirement for any encoding

scheme. To avoid this problem we require that the encoding be ***prefix-free***, meaning that no codeword is a prefix of another. This ensures unique decoding of binary strings; furthermore, the decoding can be done efficiently and on-line, with no need for look-ahead: We start at the beginning of the binary string and scan it until we find a prefix that is the codeword of a symbol of Γ; we then output that symbol and continue decoding the rest of the binary string in the same manner.

A prefix-free code can be represented as a binary tree whose leaves correspond to the symbols of Γ and each edge is labeled with 0 or 1 according as it leads to a node's left or right child. Each symbol is associated with a unique leaf of the tree, and the codeword for symbol a is the sequence of labels on the edges of the path from the root to the leaf associated with a.

Example 1.3 Let $\Gamma = \{A, B, C, D, E, F\}$ be an alphabet. Figure 1.6 shows, on the left, a (prefix-free) code for Γ and, on the right, its binary tree respresentation.

Symbol	Codeword
A	10
B	001
C	0000
D	01
E	11
F	0001

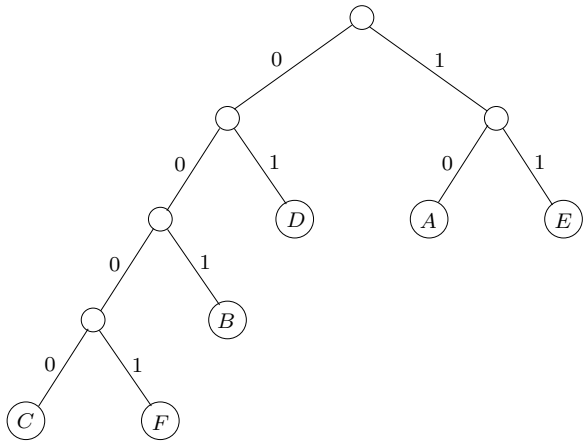


Figure 1.6: A code for $\Gamma = \{A, B, C, D, E, F\}$ and its representation as a binary tree

End of Example 1.3

Let $f\colon \Gamma \to [0, 1]$ be a function such that $f(a)$ is the frequency of symbol $a \in \Gamma$, i.e., the fraction of occurrences of a in a string over Γ or, equivalently, the probability that a symbol in a string over Γ is a. So, $\sum_{a\in\Gamma} f(a) = 1$. For example, Figure 1.5 gives this function for the letters of the Latin alphabet in English text. Given f and a binary tree T that represents a prefix-free code for Γ, we can define the ***(weighted) average depth*** of T as

$$\mathbf{ad}(T) = \sum_{a\in\Gamma} f(a) \cdot \mathbf{depth}_T(a),$$

where $\mathbf{depth}_T(a)$ is the depth of the leaf that corresponds to symbol a in T; recall that the depth of a node in a tree is the number of *edges* on the path from the root to that node. So the quantity $\mathbf{ad}(T)$ represents the average length of codewords, weighted by their frequencies.

Example 1.4 Consider the frequencies for the symbols of alphabet $\Gamma = \{A, B, C, D, E, F\}$ shown in the table on the left side of Figure 1.7.

Symbol	Frequency
A	0.38
B	0.15
C	0.1
D	0.2
E	0.12
F	0.05

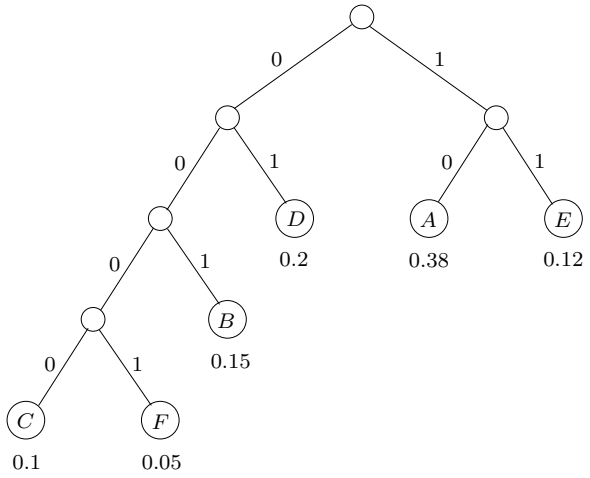


Figure 1.7: Frequencies for the alphabet and code of Example 1.3

The average depth of the code in Example 1.3 for this alphabet and these frequencies is

$$\mathbf{ad}(T) = 4 \times (0.1 + 0.05) + 3 \times 0.15 + 2 \times (0.2 + 0.38 + 0.12) = 2.45$$

If we were to use a fixed-length code for this alphabet, we need $\lceil \log_2 6 \rceil = 3$ bits for each symbol. So, the variable-length code of Example 1.3 uses $2.45/3 \approx 0.8$ of the amount of space needed by a fixed-length code for these frequencies; i.e., it achieves a better than 20% compression.

A moment's thought, however, shows that the code in Example 1.3 is not the best we can do for these frequencies: B has greater frequency but has a longer codeword (i.e., it is deeper in the tree) than E. We can certainly improve the average depth by exchanging these nodes as shown in Figure 1.8(a). This results in a code with average depth 2.42. Although this code no longer has the obvious anomaly of the code in Figure 1.7, it is not optimal. The code represented by the tree in Figure 1.8(b) further reduces the average depth to 2.39 and, as we will see, this is optimal for these frequencies. **End of Example 1.4**

We can now state the ***optimal code*** problem as follows:

Input: An alphabet Γ, and a frequency function $f\colon \Gamma \to [0, 1]$.
Output: A binary tree that minimizes $\mathbf{ad}(T) = \sum_{a \in \Gamma} f(a) \cdot \mathbf{depth}_T(a)$.

There is a famous greedy algorithm that solves this problem, known as ***Huffman's algorithm***, and the resulting codes (or trees) are called ***Huffman codes*** (or trees). These are used extensively in practice; for example, they are a crucial component in data compression used in MP3 and JPEG files.

We now note two simple but useful properties of optimal codes:

Observation 1.9 *An optimal tree must be full; i.e., every internal (non-leaf) node has two children.*

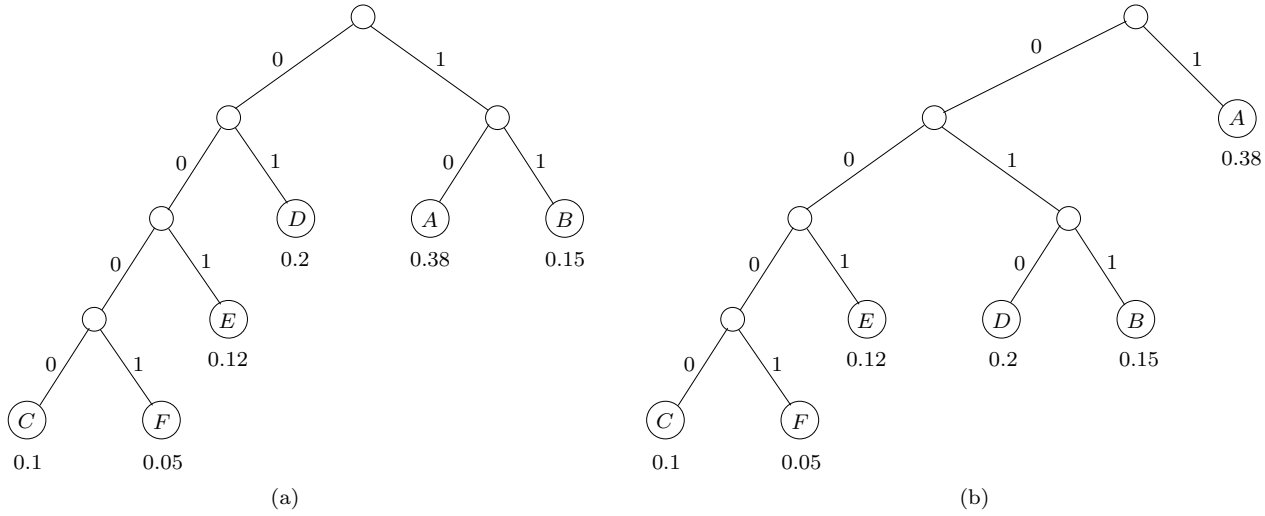


Figure 1.8: Alternative codes for the frequencies in Figure 1.7

The reason for this is that the average depth of a non-full binary tree can be improved by replacing any internal node with only one child by the subtree rooted at that child — this reduces the depth of all nodes in that subtree and therefore the average depth of the tree's leaves.

Observation 1.10 *In an optimal tree T, for any $a, b \in \Gamma$, if $f(a) > f(b)$ then $\boldsymbol{depth}_T(a) \leq \boldsymbol{depth}_T(b)$.*

We have already seen in Example 1.4 why this is true: if not, we can improve the average depth of T by swapping a and b.

Based on these properties we can now prove the following lemma, which provides the essential insight underlying Huffman's algorithm.

Lemma 1.11 *Let $a, b \in \Gamma$ be two symbols with minimum frequency such that $f(a) \leq f(b)$; that is, for all $c \in \Gamma - \{a, b\}$, $f(c) \geq f(b) \geq f(a)$. Then there is an optimal tree T such that a and b are siblings.*

Proof. Let T be any optimal tree. If a and b are siblings in T, we are done; so suppose a and b are not siblings. Since a is a symbol of minimum frequency, by Observation 1.10, it is a leaf at maximum depth in T and, by Observation 1.9, it has a sibling $c \neq b$ that is also a leaf. Let T' be the tree obtained by swapping b and c. Note that

$$\mathbf{depth}_T(a) = \mathbf{depth}_T(c) \quad \text{since } a \text{ and } c \text{ are siblings in } T \tag{1.2}$$

$$\mathbf{depth}_{T'}(b) = \mathbf{depth}_T(c) \quad \text{since } T' \text{ is obtained from } T \text{ by swapping } b \text{ and } c \tag{1.3}$$

$$\mathbf{depth}_{T'}(c) = \mathbf{depth}_T(b) \quad \text{for the same reason} \tag{1.4}$$

$$\mathbf{depth}_T(a) \geq \mathbf{depth}_T(b) \quad \text{by Observation 1.10, since } f(a) \leq f(b) \tag{1.5}$$

We have

$$\begin{aligned}
\mathbf{ad}(T') - \mathbf{ad}(T) &= f(b)\,\mathbf{depth}_{T'}(b) + f(c)\,\mathbf{depth}_{T'}(c) - f(b)\,\mathbf{depth}_T(b) - f(c)\,\mathbf{depth}_T(c) \\
&= f(b)\,\big(\mathbf{depth}_T(c) - \mathbf{depth}_T(b)\big) + f(c)\,\big(\mathbf{depth}_T(b) - \mathbf{depth}_T(c)\big) \\
&\qquad\qquad \text{[by (1.3) and (1.4)]} \\
&= \big(f(b) - f(c)\big)\,\big(\mathbf{depth}_T(c) - \mathbf{depth}_T(b)\big) \\
&= \underbrace{\big(f(b) - f(c)\big)}_{\leq 0}\,\underbrace{\big(\mathbf{depth}_T(a) - \mathbf{depth}_T(b)\big)}_{\geq 0 \text{ [by (1.5)]}} \qquad \text{[by (1.2)]} \\
&\leq 0.
\end{aligned}$$

Thus, $\mathbf{ad}(T') \leq \mathbf{ad}(T)$. Since T is optimal, this inequality must be an equality and so T' is also an optimal tree, where a and b are siblings, as wanted. □

This lemma leads us to Huffman's algorithm, which can be described informally as follows. Given the alphabet Γ with symbol frequencies f, we take two symbols of Γ *with minimum frequency*, say symbols a and b. We create a smaller alphabet Γ' by replacing a and b in Γ by a new symbol c. The frequencies f' of the symbols in Γ' are the same as the frequencies f in Γ, except that the new symbol c has frequency $f(a) + f(b)$; that is, we think of c as the combined symbol "*a or b*" in the smaller alphabet Γ'. We now recursively find an optimal tree for Γ' with frequencies f', and in that tree we make the leaf that contains c an internal node with children a and b. The recursion ends when the alphabet has two symbols, in which case the tree consists of just a root with the two symbols as children. We can express this algorithm in pseudocode as follows.

```
    HUFFMAN(Γ, f)
 1  if |Γ| = 2 then return the tree with a root and the two symbols of Γ as its children
 2  let a, b be two symbols of Γ with minimum frequency
 3  let c be a new symbol, not in Γ
 4  Γ' := (Γ − {a, b}) ∪ {c}
 5  let f': Γ' → [0, 1] be the following frequency function for Γ':
 6      for each d ∈ Γ' do
 7          if d = c then f'(d) = f(a) + f(b)
 8          else f'(d) = f(d)
 9  T' := HUFFMAN(Γ', f')
10  return the tree obtained by adding children a and b to the node in T' that contains c
```

Note that this algorithm as specified above can produce many different trees, depending on

- which minimum frequency symbols it selects in line 2, if there are more than two, and
- which of a and b becomes the left child of c.

Regardless of these choices, any tree produced by this algorithm is optimal, as we now show.

Theorem 1.12 HUFFMAN(Γ, f) *returns an optimal tree for the given input.*

PROOF. The proof is by induction on the size of Γ. More precisely, we use induction on n to prove that for all $n \geq 2$,

if $|\Gamma| = n$ *then for every frequency function* $f\colon \Gamma \to [0,1]$, HUFFMAN(Γ, f) *returns an optimal tree for* Γ *and* f. (*)

For the base case, $n = 2$, note that both full binary trees with two leaves (depending on which leaf is the left and which is the right child of the root) have the same average depth, namely 1. For the induction step, assume that (*) holds for alphabets with n symbols. We will prove that it also holds for alphabets with $n+1$ symbols. To this end, let Γ be an alphabet with $n+1$ symbols, f be any frequency function for these symbols, and consider the call HUFFMAN(Γ, f). Let a and b be the minimum frequency symbols chosen in line 2, c be the new symbol defined in line 3, $\Gamma' = \big(\Gamma - \{a,b\}\big) \cup \{c\}$, f' be the frequencies of the symbols in Γ' as defined in lines 5-8, and H be the tree returned by HUFFMAN(Γ, f).

Let H' be the tree obtained from H by removing the leaves a and b and assigning symbol c to their parent node. By the algorithm, H' is the tree returned by HUFFMAN(Γ', f'). Since Γ' has n symbols, by the induction hypothesis, H' is optimal for Γ' and f'.

Now, let T by any optimal tree for Γ and f. By Lemma 1.11 we can assume, without loss of generality, that a and b are siblings in T. Let T' be the tree contained by removing the leaves a and b from T and assigning symbol c to their parent node. So, T' is a code tree for Γ'. Since H' is optimal for Γ' and f', $\mathbf{ad}(H') \leq \mathbf{ad}(T')$. We have

$$\begin{aligned}\mathbf{ad}(H) &= \mathbf{ad}(H') + \big(f(a) + f(b)\big)\\ &\leq \mathbf{ad}(T') + \big(f(a) + f(b)\big)\\ &= \mathbf{ad}(T).\end{aligned}$$

So $\mathbf{ad}(H) \leq \mathbf{ad}(T)$. But since T is, by definition, optimal for Γ and f, this inequality must actually be an equality, and so H is also optimal for Γ and f, as wanted. □

An efficient implementation of Huffman's algorithm

We now describe an implementation of Huffman's algorithm that runs in time $O(n \log n)$, where $n = |\Gamma|$. Note that a non-empty full binary tree with n leaves has $2n-1$ nodes (this is easy to prove by complete induction). The algorithm takes its input in the form of two arrays, $\Gamma[1..n]$ and $f[1..n]$, where $\Gamma[i]$ is the i-th symbol of the alphabet and $f[i]$ is that symbol's frequency. The algorithm returns a tree T in the form of an array $T[1..2n-1]$, where the leaves (corresponding to the n symbols of the alphabet) are stored in $T[1..n]$ and the $n-1$ internal nodes are stored in $T[n+1..2n-1]$, with the root in $T[2n-1]$. Each $T[i]$ contains a pair (ℓ, r), where ℓ and r are, respectively, the indices in T of the left and right children of node i; the left and right children of leaf nodes contain 0, which is not an index in T; we denote the two fields of the pair in $T[i]$ as $T[i]$.left and $T[i]$.right.

The algorithm uses a heap H, which holds information about the nodes of the tree T as the tree is being constructed. Each element in the heap stores a pair (i, F), where i is a node of the tree being constructed, i.e., an index in T, and F is the sum of the frequencies of the leaves in the subtree rooted at $T[i]$. We denote the two fields of the pair stored in some element $H[k]$ of the heap as $H[k]$.index and $H[k]$.freq. The latter is the key used to order the elements of the heap.

Initially, for each $i \in [1..n]$ we place $\big(i, f(i)\big)$ in $H[i]$ and indicate that $T[i]$ contains a leaf by placing $(0, 0)$ into it (lines 1-3 in the pseudocde below). We then turn $H[1..n]$ into a heap (line 4). Thereafter we repeatedly extract the two mininmum-frequency nodes a and b from the heap (lines 6-7), we create a new node that will be their parent and insert the new node with the combined frequency of a and b back into the heap (line 8), and make the nodes a and b that were removed from the heap the left and right children of the new node (line 9).

```
   HUFFMAN(Γ[1..n], f[1..n])
 1 for i := 1 to n do
 2     H[i] := (i, f[i])
 3     T[i] := (0, 0)
 4 BUILDHEAP(H)
 5 for i := n + 1 to 2n − 1 do
 6     a := EXTRACTMIN(H)
 7     b := EXTRACTMIN(H)
 8     INSERT(H, (i, a.freq + b.freq))
 9     T[i] := (a.index, b.index)
10 return T
```

The loop in lines 1-3 takes $O(n)$ time. The procedure BUILDHEAP takes time linear in the size of the heap, so line 4 also takes $O(n)$ time. Each of the $n - 1$ iterations of the loop in lines 5-9 involves two EXTRACTMIN and one INSERT operation on a heap that has at most n elements, each of which takes $O(\log n)$ time; so this loop takes $O(n \log n)$ time in total. So, this implementation of Huffman algorithm runs in time $O(n) + O(n) + O(n \log n) = O(n \log n)$.

Exercises for Chapter 1

1.1 Prove that the "shortest-next" version of the greedy algorithm for interval scheduling (i.e., the version that considers jobs in order of increasing length) returns a set of jobs that contains at least half as many jobs as the optimal.

1.2 Prove that the "fewest-conflicts-next" version of the greedy algorithm for interval scheduling (i.e., the version that considers jobs in order of increasing number of conflicts) returns a set of jobs that contains at least half as many jobs as the optimal.

1.3 Consider the following variant of the interval scheduling problem. We are given again a set $\mathcal{J}$ of "jobs", each specified by its start and finish times, and we want to find the smallest

possible number m of "machines" using which we can schedule *all* the jobs. Furthermore, we want to find an assignment of all the jobs to the m machines so that no two conflicting jobs are assigned to the same machine; i.e., we want a function A: $\mathcal{J} \to [1..m]$ such that for all $j, j' \in \mathcal{J}$, if $j \neq j'$ and $A(j) = A(j')$, then j and j' do not conflict.

(a) Prove that the following algorithm does not work:

```
ScheduleAll(𝒥)
i := 0
while 𝒥 ≠ ∅ do
    i := i + 1
    B := IntervalSchedule(𝒥)      ▶ B is a max cardinality set of non-conflicting jobs
    for each j ∈ B do A(j) := i   ▶ assign to machine i the jobs in B
    𝒥 := 𝒥 − B                    ▶ remove the jobs in B from 𝒥
return A
```

(b) Give an efficient algorithm that solves this problem.

1.4 We live in an one-dimensional world where everything is a point on a straight line, and the location of each point is defined by its distance from a fixed point on that line, called the ***origin***: negative for points to the "left" and positive for points to the "right" of the origin. We are given the locations of m firefighters in an array $F[1..m]$ and the locations of n fires in an array $R[1..n]$.

The Fire Chief assigns firefighters to fires subject to the following constraints: (a) a firefighter is assigned to at most one fire, (b) at most one firefighter is assigned to a fire, and (c) a firefighter can be assigned to a fire only if the fire is located within distance $d \geq 0$ (to the left or to the right) of the firefighter's location.

Give an efficient greedy algorithm that helps the Fire Chief assign firefighters to as many fires as possible under these constraints. Your algorithm takes as input the arrays F and R and produces as output a set of pairs of the form (f, r), where $1 \leq f \leq m$ and $1 \leq r \leq n$, indicating that firefighter f is assigned to fire r. Prove the correctness of your algorithm and analyze its running time.

1.5 Let $\mathcal{I}$ be a set of closed intervals over the real numbers. A subset $\mathcal{C}$ of $\mathcal{I}$ is a ***cover of*** $\mathcal{I}$ if for every interval $I \in \mathcal{I}$ there is an interval $I' \in \mathcal{C}$ such that I and I' intersect (even at one point). We wish to find a ***minimum cardinality*** cover of $\mathcal{I}$ — i.e., a cover of $\mathcal{I}$ with as few intervals as possible.

(a) True or false? Every set of intervals $\mathcal{I}$ has some minimum cardinality cover $\mathcal{C}$ such that no two intervals in $\mathcal{C}$ intersect. Justify your answer.

(b) Describe an efficient greedy algorithm that, given a set $\mathcal{I}$ of n intervals, returns a minimum cardinality cover of $\mathcal{I}$. Prove the correctness of your algorithm and analyze its running time.

1.6 Let $\mathcal{I}$ be a sequence of closed intervals over the reals. A set of numbers C is a ***point cover*** of $\mathcal{I}$ if every interval in $\mathcal{I}$ contains at least one number in C (that is, for each $I = [s, f]$ in $\mathcal{I}$, there is some $c \in C$ such that $s \leq c \leq f$). C is a ***minimum point cover*** of $\mathcal{I}$ if it is a minimum cardinality point cover of $\mathcal{I}$.

(a) Consider the following greedy algorithm.

```
1 C := ∅
2 while I ≠ ∅ do
3     let c be a number that is contained in the largest number of intervals in I
4     C := C ∪ {c}
5     delete from I all intervals that contain c
6 return C
```

(In line 3, assume that ties are broken arbitrarily. That is, if there are multiple numbers that are contained in the same maximum number of intervals in $\mathcal{I}$, then we pick any one of these numbers as our c.)

Prove that this algorithm does not always find a minimum cover of $\mathcal{I}$. Your counterexample should be such that if in any iteration of the algorithm there are multiple choices for c, then *every* choice results in a suboptimal C at the end. (Thus, there is no hope of fixing this greedy algorithm by using a more refined choice of c as a number that is contained in the maximum number intervals *and* satisfies some additional property.)

(b) Describe an efficient greedy algorithm that given a sequence of intervals $\mathcal{I}$ (in arbitrary order) returns a minimum cover of $\mathcal{I}$. Prove the correctness of your algorithm.

1.7 (All intervals in this question are assumed to be closed real intervals with more than one element each, i.e., they are of the form $[s, f]$, where $s, f \in \mathbb{R}$ and $s < f$.) A set of intervals $\mathcal{J} = \{I_1, I_2, \ldots, I_n\}$ ***covers*** an interval I if $I \subseteq \cup_{i=1}^{n} I_i$. We are given an interval I and a finite set of intervals $\mathcal{J}$ that covers I. We wish to find a subset of $\mathcal{J}$ that covers I and has as few intervals as possible.

(Think of I as the duration of some student event, and $I_1, I_2, \ldots, I_n$ as the periods of time during which each of n volunteers can staff the Student Union booth during the event. We want to find the smallest number of volunteers so that at all times during the event at least one volunteer is staffing the Student Union booth.)

(a) Consider the following greedy strategy. Start with the empty set of intervals A; in each iteration add to A the interval in $\mathcal{I}$ that has the largest intersection with the uncovered part of I, i.e., $I - \cup_{J \in A} J$. (Note that initially the uncovered part of I is all of I, since at that time A is empty.) Repeat this until all of I is covered (i.e., until $I \subseteq \cup_{J \in A} J$). Give a counterexample showing that this strategy does not necessarily yield a desired subset of $\mathcal{I}$.

(b) Describe a greedy algorithm to find a subset of $\mathcal{I}$ that covers I and has as few intervals as possible. Prove that your algorithm is correct, and analyze its running time.

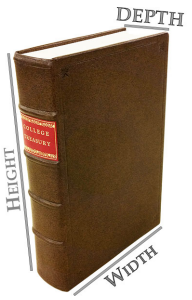
1.8 Let T be a (rooted) tree, not necessarily binary, and A be a subset of nodes of T. Given T and A, we want to find a ***maximum cardinality subset*** B of A such that the subtrees rooted at the nodes of B are pairwise disjoint.

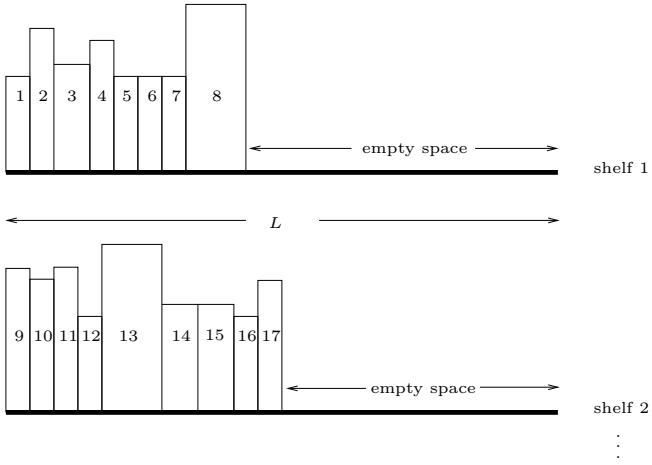
Describe a greedy algorithm that solves this problem. Prove that your algorithm is correct, and analyze its running time. For the purpose of the running-time analysis, assume that the nodes of the tree are labeled with the positive integers 1 to n; the root is the node labeled 1; and for each node $i \in [1..n]$, we are given (i) the list of its children, *children*(i) (empty, if i is a leaf), and (ii) the parent of i, *parent*(i) (0, if $i = 1$).

1.9 Consider the following process: We are given a positive integer n. We start with $x = 1$. At each step of the process we can apply to x one of two operations: INCREMENT x by 1 (i.e., $x := x + 1$) or DOUBLE x (i.e., $x := 2x$). We want to reach the given number n (i.e., make $x = n$) by applying repeatedly such steps. For example, if $n = 1024$, we can do this in 1023 increment steps, or in 10 double steps, or in some combination of these two kinds of steps.

Our problem is the following: Find a sequence of increment or double operations so that starting from 1 we reach n *with the fewest steps*. Describe an efficient greedy algorithm that solves this problem. Your algorithm should take as input the positive integer n, and should produce as output a sequence each element of which is I (increment) or D (double), so that if we apply these operations to x in the given order starting with $x = 1$, we end with $x = n$. Analyze the running time of your algorithm and prove its correctness.

1.10 We are given a sequence of n books of depths $d_1, d_2, \ldots, d_n$ (see figure on the left below). We must place these books one after the other *in the given order* on successive shelves, each of length $L \geq \max(d_1, d_2, \ldots, d_n)$, so that the sum of the depths of the books on each shelf does not exceed L.





The greedy algorithm below places books on the shelves according to these rules. Intuitively it packs as many books as possible on each shelf, and uses a new shelf if and only if the next book does not fit on the current shelf.

$i := 1$ ▶ *next book to place on a shelf*
$s := 1$ ▶ *current shelf*
$t := 0$ ▶ *total depth of books on current shelf*
while $i \leq n$ **do**
  if $t + d_i \leq L$ **then** ▶ *the next book fits on the current shelf*
    if $t = 0$ **then** place book i in the leftmost spot of shelf s
    else place book i after book $i - 1$ on shelf s
    $t := t + d_i$; $i := i + 1$
  else ▶ *the next book does not fit on the current shelf*
    $s := s + 1$; $t := 0$

(a) Does this algorithm minimize the ***sum of empty spaces on the shelves used***? Justify your answer.
(b) Does this algorithm minimize the ***maximum empty space on any shelf used***? Justify your answer.

1.11 A summer camp has n campers, labeled $1, 2, \ldots, n$, and wants to organize a canoe trip. Camper i weighs w_i. Each canoe must have exactly two campers, whose combined weight must not exceed C; naturally, no camper can be in two canoes! Our job is to pair-up as many campers as possible subject to the weight constraint.

More precisely, a set of (unordered) pairs of campers is ***feasible*** if (i) no camper belongs to two different pairs in the set; and (ii) for any pair $\{i, j\}$ in the set, $w_i + w_j \leq C$. A feasible set of pairs of campers is ***optimal*** if there is no feasible set with greater cardinality. Given $w_1, \ldots, w_n$ and C, we want to find an optimal set of pairs of campers.

(a) Prove that the following greedy algorithm does not necessarily find an optimal set: If $n < 2$, return the empty set. Otherwise, let p and q be two lightest campers (ties broken arbitrarily). If $w_p + w_q \leq C$, return the set of pairs obtained by adding the pair $\{p, q\}$ to the set returned by recursively applying the algorithm to the remaining campers; otherwise, return the empty set.
(b) Prove that the following greedy algorithm always finds an optimal set: If $n < 2$, return the empty set. Otherwise, let p be a lightest camper and q be a heaviest camper (ties broken arbitrarily). If $w_p + w_q \leq C$, return the set of pairs obtained by adding the pair $\{p, q\}$ to the set returned by recursively applying the algorithm to the remaining campers; otherwise, discard q and return the set returned by recursively applying the algorithm to the remaining campers. The recursion ends when the number of remaining campers is less than two.
(c) Suppose the camp has only four-person canoes; i.e., each canoe must carry exactly four campers. Consider the following greedy algorithm for this case: Take the two lightest and two heaviest campers (ties broken arbitrarily). If the combined weight of these four campers is at most C add this group of four campers to the set returned by recursively applying the algorithm to the remaining campers; otherwise discard the heaviest of the four campers and return the set returned by recursively applying the algorithm to the remaining campers. The recursion ends when the number of remaining campers is less than four. Does this algorithm work? Justify your answer.

1.12 A catering service receives n orders, each requiring *exactly one time unit* to complete. There is only one cook, so these orders must be completed one at a time; the cook starts work at time 0.

Order i, $1 \leq i \leq n$, has a *deadline* d_i (in time units) by which it must be completed, and a *profit* $p_i > 0$ that the catering service receives if the order is completed by the deadline; if the order is completed after the deadline there is no profit made and therefore it is not done at all. Note that several orders may have the same deadline (e.g., around lunch time). The catering service must decide on a schedule in which to do a subset of the orders, each completed by its deadline.

Describe an $O(n^2)$ greedy algorithm that takes as input the set of orders with their deadlines and profits $J = \{(i, d_i, p_i) : 1 \leq i \leq n\}$ and outputs a schedule of maximum profit. A schedule is a function S that maps each order i to a time slot j, where j is a positive integer, meaning that order i will be done during the time interval $[j-1, j)$; or to the time slot $+\infty$, meaning that order i will not be done. The schedule S cannot map two different orders to the same time slot (because there is only one cook who completes the orders one at a time); and if it maps order i to time slot $j \neq 0$, then $j \leq d_i$ (since every order that is done must be finished by its deadline). The profit of schedule S is the sum of the orders that are done, i.e., $\sum_{1 \leq i \leq n \ \&\ S(i) \neq \infty} p_i$.

Prove that your algorithm is correct and analyze its running time. (It can be done in $O(n \log n)$ time, but $\Theta(n^2)$ is good enough for the purposes of this question.)

1.13 A video production facility has n videos to process. The processing of each video i, $1 \leq i \leq n$, is done in two phases. In the first phase, the facility's editor edits the video. After the first phase of video i is finished, the edited video is processed by computer. The first phase of video i takes $f_i \geq 0$ units of time and the second phase takes $s_i \geq 0$ units of time.

The facility has a single video editor to do the first phase but it has n computers to do the second phase. So, the first phase must be done sequentially by the single editor, while the second phase can be done in parallel by the computers: As soon as the editor is finished editing a video, she feeds it into one of the free computers for the second phase, even as other computers may still be processing the videos whose first phase she completed earlier. The completion time of a video is the time by which its second phase is finished. The manager of the facility wants to find a schedule that ***minimizes the maximum completion time over all*** n ***videos*** — that is, the earliest time by which both phases of all videos have been been completed.

Describe an efficient greedy algorithm that takes as input the set of videos and the length of their first and second phases $V = \{(i, f_i, s_i) : 1 \leq i \leq n\}$ and outputs an optimal schedule. In this case, a schedule is simply a permutation of the integers from 1 to n, indicating the order in which the editor processes the videos. This determines the time by which each video finishes, and therefore the maximum completion time over all videos, which is the quantity we want to minimize.

Prove that your algorithm is correct and analyze its running time.

1.14 The optimal length of a paddle for a canoeist is his/her shoulder height. We have n canoeists with shoulder heights $h_1, h_2, \ldots, h_n$, and n paddles of length $\ell_1, \ell_2, \ldots, \ell_n$. We must assign exactly one paddle to each canoeist (and, of course, different paddles to different canoeists). Such an assignment can be conveniently represented by a permutation π of $1, 2 \ldots, n$: canoeist i is assigned paddle $\pi(i)$. The ***penalty*** of such an assignment for canoeist i is $|h_i - \ell_{\pi(i)}|$ — i.e., how far off the optimal is the length of the paddle assigned to the canoeist. The ***average penalty*** of such an assignment is $\frac{1}{n} \sum_1^n |h_i - \ell_{\pi(i)}|$. We wish to find an assignment that minimises the average penalty.

For each of the following two greedy strategies, either show that it determines a desired assignment or provide a counterexample.

(a) Best-match first strategy: Find a pair (i, j) that minimizes $|h_i - \ell_j|$ (i.e., a canoeist-paddle

pair of minimum penalty). Assign paddle j to canoeist i (i.e., $\pi(i) = j$) and apply the same strategy to the remaining canoeists and paddles, until all canoeists are assigned paddles.

(b) Increasing height strategy: Sort the canoeists in increasing shoulder height and the paddles in increasing length, and assign to each canoeist the corresponding paddle. In other words, the canoeist of minimum shoulder height gets the shortest paddle, the canoeist of second minimum shoulder height gets the second shortest paddle, and so on.

1.15 Recall that Huffman's algorithm can result in many different trees (and therefore codes) for a given alphabet and frequencies — all of them optimal — because it may have a choice in which two minimum-frequency to choose, if there are more than two, and how it orders subtrees. Are there alphabets and frequencies for which there are *optimal* trees that cannot be generated by Huffman's algorithm? Justify your answer.

1.16 Consider Huffman's algorithm.

(a) Prove that for any alphabet, if some symbol has frequency (strictly) greater than 2/5, Huffman's algorithm will necessarily produce a codeword of length 1.

(b) Give an example of a (small) set of symbols and their associated frequencies so that the maximum frequency of any symbol is equal to 2/5, and some execution of Huffman's algorithm produces a tree in which no codeword has length 1. Show such a tree that could be produced by Huffman's algorithm in your example.

(c) Prove that for any alphabet, if every symbol has frequency (strictly) less than 1/3, Huffman's algorithm cannot produce a codeword of length 1.

(d) Give an example of a (small) set of symbols and their associated frequencies so that there is exactly one symbol with frequency 1/3, all other symbols have frequency (strictly) less than 1/3, and some execution of Huffman's algorithm produces a codeword of length 1. Show such a tree that could be produced by Huffman's algorithm in your example.

Chapter 2

Divide-and-conquer algorithms

2.1 General form of divide-and-conquer algorithms

In earlier courses you have encountered a number of algorithms that fall in the "divide-and-conquer" category. Recall, for example, Merge Sort; to sort an array $A[1..n]$:

```
1 if n = 1 then return
2 else
3     recursively sort A_1 = A[1..⌊n/2⌋] and A_2 = A[⌈n/2⌉ + 1..n]
4     merge the sorted arrays A_1 and A_2
```

Also, recall binary search, which can be used to determine if a *sorted* array $A[1..n]$ contains x:

```
1 if n = 1 then return A[1] = x
2 else
3     if A[⌊n/2⌋] ≥ x then recursively search A[1..⌊n/2⌋]
4     else recursively search A[⌊n/2⌋ + 1..n]
```

The typical form of a generic divide-and-conquer algorithm that solves an instance of size n of some problem is:

```
1 if n is "small", find the answer directly
2 else
3     divide the input into a pieces, each of size ≈ n/b, where a ≥ 1, b > 1 are constants
4     recursively solve each of the a smaller instances of the problem
5     combine the solutions of the a smaller instances to obtain the solution to the original problem
```

Figure 2.1: Generic divide-and-conquer algorithm

To say that a, b are constants means that these are fixed numbers that do not depend on n. In the case of Merge Sort, $a = b = 2$; in the case of Binary Search, $a = 1$ and $b = 2$.

It is not always possible to take an instance of a problem, break it up into smaller instances of the *same* problem, and then stitch together the solution of the original problem efficiently from solutions to the smaller pieces. When this technique is applicable, however, it results in simple and efficient algorithms.

Not every recursive algorithm falls under the pattern described above; the characteristic features of divide-and-conquer algorithms are:

(i) the algorithm actively chooses how to divide its input into the smaller subproblems of the same type that it solves recursively — as opposed to the subproblems on which it operates being dictated by the input itself (as is the case, for example, in recursive algorithms such as first-depth search of a graph or preorder traversal of a tree);

(ii) the number of subproblems a is *fixed*;

(iii) the size of each subproblem is a fixed *fraction* of the original, i.e., n/b for some fixed $b > 1$, as opposed to $n - b$ for some fixed $b > 0$ — thus, the size of inputs shrinks exponentially and the depth of the recursion is logarithmic in the size of the input.

We note, however, that some algorithms that are classified as divide-and-conquer do not satisfy (iii): in such algorithms, the size of the subproblems varies. We will see some interesting examples of such algorithms in Section 2.6. Also note that the "combine" step in the generic algorithm may be trivial (a no-op), as in the case of binary search.

2.2 Running time of divide-and-conquer algorithms

Let $T(n)$ be the running time of the generic divide-and-conquer algorithm shown in Figure 2.1, where n is the size of the input. For simplicity initially we will assume that n is a power of b, so that we can keep dividing it by b and obtaining an integer value (a proper input size) until the input size is 1. This assumption turns out to be immaterial, as we will see later. $T(n)$ can be described by the following recurrence.

$$T(n) = \begin{cases} aT(n/b) + cn^d, & \text{if } n > 1 \\ c, & \text{if } n = 1 \end{cases} \tag{2.1}$$

where $c \geq 1$ and $d \geq 0$ are constants. In the general case $n > 1$, the first term, $aT(n/b)$, accounts for the time required to solve recursively the a subproblems, each of size n/b. The second term, cn^d, accounts for the time required to divide up the given input into smaller inputs (line 3 in the generic divide-and-conquer algorithm) and to compute the solution to the original instance from the solutions to the smaller instanes (line 5 in the generic divide-and-conquer algorithm). We assume that this is a polynomial of degree $d \geq 0$. For example, in the case of Merge Sort, $d = 1$, because in linear time we can split A into A_1 and A_2 and merge the sorted versions of A_1 and A_2 into the final answer. For Binary Search, $d = 0$, because in this case the splitting and combining steps can be done in constant time. For the base case, $n = 1$, the actual constant is not important; taking it to be the same as the coefficient c simplifies the algebra.

The so-called "Master Theorem" gives the solution to the above recurrence in a convenient form, and is a very useful tool that simplifies tremendously the running time analysis of many divide-and-conquer algorithms.

Theorem 2.1 *If $T(n)$ is the function defined by recurrence (2.1), where $a \geq 1$ and $b > 1$ are integer constants, $c \geq 1$ and $d \geq 0$ are real constants, and n is a power of b, then*

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n), & \text{if } a = b^d \\ \Theta(n^{\log_b a}), & \text{if } a > b^d. \end{cases} \tag{2.2}$$

(Note that the constant c in the definition of $T(n)$ does not appear explicitly in (2.2): It is "absorbed" by the constants implicit in the Θ-notation.)

PROOF. This can be easily checked by using induction. A more insightful proof, however, can be obtained by reasoning about the amount of time required by the divide-and-conquer algorithm whose running time is represented by (2.1).

Consider the tree of recursive calls made by the algorithm:

- At level 0 (the root), we have one call for input of size n. The time needed by this call, excluding the time needed by the recursive calls it makes, is cn^d — the second term in the definition of the recurrence. This is the time to divide up the input and to combine the results of the calls it makes.
- At level 1 we have the calls made by the call at level 0: there are a such calls (one for each subproblem), each working on an input of size n/b. The time needed by *each* of these calls, excluding the time needed by the calls it makes, is $c(n/b)^d$, so the total time needed by these a calls is $ac(n/b)^d$.
- At level 2 we have the calls made by the calls at level 1: there are a^2 such calls, each working on an input of size n/b^2. Reasoning as before, the total time needed by these a^2 calls is $a^2c(n/b^2)^d$.
- In general, at level i, we have a^i calls, each working on an input of size n/b^i. The total time needed for these calls is $a^ic(n/b^i)^d$.
- This continues for every integer $i = 0, 1, 2, \ldots, \log_b n$. For $i = \log_b n$, the input size is 1, and we have reached the base case of the recursion.

Thus, the total time required for all the calls at all levels is

$$T(n) = \sum_{i=0}^{\log_b n} a^i c\Big(\frac{n}{b^i}\Big)^d = cn^d \sum_{i=0}^{\log_b n} \Big(\frac{a}{b^d}\Big)^i. \tag{2.3}$$

Note that $\sum_{i=0}^{\log_b n}(a/b^d)^i$ is just a geometric series of the form $\sum_{i=0}^{k} x^i$. From basic mathematics recall that

- if $0 \leq x < 1$, then $\sum_{i=0}^{\infty} x^i = 1/(1-x)$, and
- for all $x \neq 1$, $\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x - 1)$.

So we have:

CASE 1. $a < b^d$. Then, by appealing to the formula for the infinite geometric series, we have:

$$\sum_{i=0}^{\log_b n} \Big(\frac{a}{b^d}\Big)^i < \sum_{i=0}^{\infty} \Big(\frac{a}{b^d}\Big)^i = 1/(1 - a/b^d) = \Theta(1).$$

So by (2.3), $T(n) = \Theta(n^d)$ in this case.

CASE 2. $a = b^d$. Then we have:

$$\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i = \sum_{i=0}^{\log_b n} 1^i = 1 + \log_b n = \Theta(\log n).$$

So by (2.3), $T(n) = \Theta(n^d \log n)$ in this case.

CASE 3. $a > b^d$. Then, by appealing to the formula for the finite geometric series, we have:

$$\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i = \frac{\left(\frac{a}{b^d}\right)^{1+\log_b n} - 1}{\underbrace{\frac{a}{b^d} - 1}_{\Theta(1)}} = \Theta\left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = \Theta\left(\frac{n^{\log_b a}}{n^d}\right).$$

(In the last step we use the following facts about logarithms: $a^{\log_b n} = n^{\log_b a}$ (verify by taking logarithms of both sides), and $b^{\log_b n} = n$ (by definition of logarithm).) So by (2.3), $T(n) = \Theta(n^{\log_b a})$ in this case. □

So, in the first case, $a < n^d$, the running time of the generic divide-and-conquer algorithm is dominated by the time split the input and recombine the results of the recursive calls; in the third case, $a > n^d$, the running time is dominated by the number of recursive calls made.

Arbitrary values of n

In the preceding analysis we assumed that the input size n is a positive integer that is a power of $b > 1$, and therefore that b is also an integer. It turns out that, essentially, the Master Theorem holds even if n is not necessarily a power of b, and $b > 1$ is a real number.[1] If n is not a power of b, however, (2.1) is not a legitimate recurrence: sooner or later as we keep dividing the input size by b we will end up with a non-integer and then the recurrence is not defined. Also, a non-integer as an input size does not make sense.

In general, a divide-and-conquer algorithm breaks a problem of size n into a subproblems, a_1 of which have size $\lceil n/b \rceil$ and a_2 have size $\lfloor n/b \rfloor$, for some non-negative integers a_1 and a_2 such that $a_1 + a_2 = a$. Thus, the following recurrence describes the running time of such an algorithm, when n is not necessarily a power of b, and $b > 1$ is not necessarily an integer:

$$T(n) = \begin{cases} a_1 \cdot T(\lceil n/b \rceil) + a_2 \cdot T(\lfloor n/b \rfloor) + c \cdot n^d, & \text{if } n \geq b/(b-1) \\ c, & \text{if } 1 \leq n < b/(b-1), \end{cases} \tag{2.4}$$

where a_1, a_2 are non-negative integers such that $a_1 + a_2 = a$. (The requirement that $n \geq b/(b-1)$ in the first case of the recursive definition ensures that $\lceil n/b \rceil < n$ and therefore that the

[1] We get a non-integer b if, for example, in each level of the recursion, the input size shrinks by a factor of 2/3, i.e., if $b = 3/2$.

recurrence is well-founded.) It turns out that this recurrence also satisfies (2.2). This can be verified by induction, though the algebra is somewhat daunting. The proof below uses a different approach.

Theorem 2.2 (the "Master Theorem") *If $T(n)$ is the function defined by recurrence (2.4), where a_1, a_2 are non-negative integers such that $a_1 + a_2 = a \geq 1$, and $b > 1, c > 0, d \geq 0$ are reals, then $T(n)$ satisfies (2.2).*

PROOF. Let $T' : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ be the function (over the non-negative reals) defined by:

$$T'(n) = \begin{cases} a_1 \cdot T'(\lceil n/b \rceil) + a_2 \cdot T'(\lfloor n/b \rfloor) + c \cdot n^d, & \text{if } n \geq b/(b-1) \\ c, & \text{if } n < b/(b-1), \end{cases}$$

First we have to argue that this does, in fact, define a function: recursion can be used to define a function over the natural numbers, but now our domain is the reals, and we cannot appeal directly to induction to justify that the function is properly defined. Note, however, that if n is an integer then $T'(n) = T(n)$ and since T is well-defined, T' is well defined when restricted to integers. Furthermore, for any real number n, whether integer or not, $T'(n)$ is determined uniquely by the values of function T' at integer arguments, which as we just mentioned are well-defined. Therefore T' is a well-defined function for all reals.

Using a straightforward induction it is easy to check that T (the recursively defined function over the integers) is non-decreasing. From this and the fact that the ceiling, floor, and cn^d functions are non-decreasing, it follows that T' is also non-decreasing.

Having established that T' is a well-defined non-decreasing function over the non-negative reals, we now define the function $S : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ as

$$S(n) = T'(n + \kappa)$$

for some real constant κ. For reasons that will become clear shortly, we will choose $\kappa = b/(b-1)$; for now we treat it as a constant "to-be-determined".

First we will obtain an upper bound on S. We have:

$$\begin{aligned}
S(n) &= T'(n+\kappa) && \text{[by the definition of } S\text{]} \\
&= a_1 T'\big(\lceil (n+\kappa)/b \rceil\big) + a_2 T'\big(\lfloor (n+\kappa)/b \rfloor\big) + c(n+\kappa)^d && \text{[by the definition of } T'\text{]} \\
&\leq a T'\big((n+\kappa)/b + 1\big) + c(n+\kappa)^d && \text{[since } T' \text{ is non-decreasing, and for any real } x,\ \lceil x \rceil < x+1\text{]} \\
&\leq a T'(n/b + \kappa/b + 1) + c' n^d && \text{[for some constant } c', \text{ since } \kappa \text{ is a constant]} \\
&= a T'(n/b + \kappa) + c' n^d && \text{[by choosing } \kappa = b/(b-1)\text{]} \\
&= a S(n/b) + c' n^d && \text{[by definition of } S\text{]}
\end{aligned}$$

Summarizing the above inequalities we have

$$S(n) \leq aS(n/b) + c'n^d.$$

Applying this inequality repeatedly, and using the geometric series formulas as in the proof of Theorem 2.1 we get that

$$S(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n), & \text{if } a = b^d \\ O(n^{\log_b a}), & \text{if } a > b^d. \end{cases}$$

By definition of S, $T'(n) = S(n - \kappa)$. Therefore (keeping in mind that $\kappa = b/(b-1)$ is a constant),

$$T'(n) = \begin{cases} O\big((n-\kappa)^d\big) = O(n^d) & \text{if } a < b^d \\ O\big((n-\kappa)^d \log(n-\kappa)\big) = O(n^d \log n), & \text{if } a = b^d \\ O\big((n-\kappa)^{\log_b a}\big) = O(n^{\log_b a}), & \text{if } a > b^d. \end{cases} \tag{2.5}$$

Again with $\kappa = b/(b-1)$ define

$$R(n) = \begin{cases} 0, & \text{if } n < \kappa \\ T'(n - \kappa), & \text{otherwise} \end{cases}$$

Using similar reasoning we can also obtain an asymptotic lower bound on R and thus on T':

$$T'(n) = \begin{cases} \Omega(n^d) & \text{if } a < b^d \\ \Omega(n^d \log n), & \text{if } a = b^d \\ \Omega(n^{\log_b a}), & \text{if } a > b^d. \end{cases} \tag{2.6}$$

Theorem 2.2 now follows from (2.5) and (2.6), since T coincides with T' on integer values. □

2.3 Karatsuba's multiplication algorithm

In some applications, such as public key cryptography, we need to perform arithmetic on integers whose binary representation is thousands of bits long. We cannot rely directly on the hardware instructions to perform these operations, as these operate on 64-bit integers. For such applications we must develop algorithms that operate on arrays of bits (the binary representations of the input numbers) and compute another array of bits (the binary representation of the result).

To add two n-bit numbers the standard algorithm we learned in elementary school works in $\Theta(n)$ time: We add the two numbers bit-by-bit in right-to-left order, keeping track of the carry. There is no hope of finding an algorithm that does significantly better: To compute the sum of two numbers we need to look at every bit; this means that the time complexity of any algorithm to add two n-bit numbers is $\Omega(n)$.

To multiply two n-bit numbers the elementary-school algorithm works in $\Theta(n^2)$ time, as the algorithm computes the product of n^2 pairs of bits. Can we multiply faster than that?

Let X and Y be the two n-bit numbers whose product we want to compute. For simplicity we will assume that n is a power of 2: We can ensure this by padding them out with initial 0s to the length of the closest power of 2; this at most doubles their length. Let X_1 and X_0 be the numbers represented by the high-order $n/2$ bits and low-order $n/2$ bits of X, respectively. Similarly, let Y_1 and Y_0 be the numbers represented by the high-order $n/2$ bits and low-order $n/2$ bits of Y, respectively. These are illustrated in Figure 2.2.

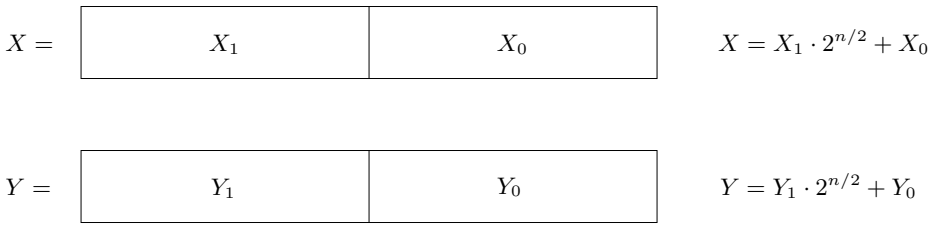


Figure 2.2: Dividing the binary representation of X and Y into two halves

We have that $X = X_1 \cdot 2^{n/2} + X_0$: The multiplication $X_1 \cdot 2^{n/2}$ appends $n/2$ 0s to the $n/2$ high-order bits of the binary representation of X, and the addition of X_0 replaces these 0s with the $n/2$ low-order bits of the binary representation of X. Similarly, $Y = Y_1 \cdot 2^{n/2} + Y_0$. Therefore

$$\begin{aligned} X \cdot Y &= (X_1 \cdot 2^{n/2} + X_0) \cdot (Y_1 \cdot 2^{n/2} + Y_0) \\ &= (X_1 \cdot Y_1) \cdot 2^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 2^{n/2} + X_0 \cdot Y_0 \end{aligned} \tag{2.7}$$

This equation suggests a divide-and-conquer algorithm to multiply two n-bit numbers: Compute this product by performing (recursively) four multiplications of $n/2$-bit numbers and some additional work consisting of

- multiplying a number by 2^n,
- multiplying a number by $2^{n/2}$, and
- adding two $2n$-bit numbers.

Multiplying a number by 2^k amounts to appending to its binary representation k 0s, and adding two k-bit numbers can be done in $\Theta(k)$ time. Therefore, the amount of work performed by this algorithm, excluding the recursive calls to perform the four $n/2$-bit multiplications, is linear in n. In terms of the parameters of the Master Theorem we have $a = 4$, $b = 2$, and $d = 1$. Since $a > b^d$, the third case of the Master Theorem applies in this case, so the running time of the algorithm is $\Theta(n^{\log_2 4}) = \Theta(n^2)$. This is not an improvement over the elementary-school algorithm for multiplication, but it motivates another divide-and-conquer idea to multiply two n-bit numbers. We can write (2.7) in the following perverse way:

$$X \cdot Y = (X_1 \cdot Y_1) \cdot 2^n + \big((X_1 + X_0) \cdot (Y_1 + Y_0) - X_1 \cdot Y_1 - X_0 \cdot Y_0\big) \cdot 2^{n/2} + X_0 \cdot Y_0 \tag{2.8}$$

The advantage of this way of computing $X \cdot Y$ over (2.7) is that it reduces the number of smaller multiplications that must be computed from four to three; the products $X_1 \cdot Y_1$ and $X_0 \cdot Y_0$ are used twice each, but need only be computed once. Thus, in this weird way of computing the product $X \cdot Y$ of two n-bit numbers, we need to recursively compute *three* products of $n/2$-bit numbers, at the expense of a constant number of additional additions and subtractions of (up to) n-bit numbers. So, the parameters of the divide-and-conquer recurrence now are $a = 3$, $b = 2$, and $d = 1$. We still have $a > b^d$, so under the third case of the Master Theorem the running time of this algorithm is $\Theta(n^{\log_2 3}) = O(n^{1.585})$. So we can multiply faster than the elementary school algorithm we were taught long ago!

Digression But how much faster? Figure 2.3 provides a striking visual demonstration of the improvement. Think of the "work" (or time) required to multiply two n-bit numbers in terms of an $n \times n$ square. In (2.7) we reduced the problem of multiplying two n-bit numbers to the problem of multiplying four pairs of $n/2$-bit numbers. This is illustrated in Figure 2.3(a); the $n \times n$ square is divided up into four $n/2 \times n/2$ squares — one for each smaller multiplication. Each of the smaller multiplications is recursively performed in a similar manner: it is reduced to multiplying four pairs of $n/4$-bit numbers. Thus, the amount of work performed in this computation covers the entire $n \times n$ square, corresponding to the $O(n^2)$ running time achieved using (2.7).

Using (2.8) we avoid one of the four multiplications, so the work required for that computation is to multiply three, instead of four, pairs of $n/2$-bit numbers, as shown in Figure 2.3(b). But each of the three smaller multiplications is done using only three multiplications of $n/4 \times n/4$-bit numbers, as shown in Figure 2.3(c). In each of the recursive calls, we repeat the same savings, as shown in Figure 2.3(d)-(g). In this figure black corresponds to work done, and white to the work avoided as a result of reducing the number of recursive calls from four to three. **End of Digression**

The preceding discussion contains an inaccuracy, which we now identify and rectify: It is not quite true that the computation of $X \cdot Y$ according to (2.8) involves three multiplications of $n/2$-bit numbers: $X_1 + Y_1$ and $X_0 + Y_0$ could have $n/2 + 1$ bits because of carries. Does this change the running time analysis? It turns out that the answer is no. Perhaps the easiest way to see this is by observing that we can multiply two $n/2 + 1$-bit numbers in the same time as multiplying two $n/2$-bit numbers plus $O(n)$ time. To see this, let $U = X_1 + X_0$ and view this as an $(n/2+1)$-bit number (by adding a leading 0, if necessary). Then let U_1 be the high-order bit of U and U_0 be the number whose binary representation consists of the $n/2$ low-order bits of U. Similarly, let $V = Y_1 + Y_0$, V_1 be the high-order bit of V, and V_0 be the number whose binary representation consists of the $n/2$ low-order bits of V. Then we have $U = U_1 \cdot 2^{n/2} + U_0$, $V = V_1 \cdot 2^{n/2} + V_0$, and so

$$U \cdot V = (U_1 \cdot V_1) \cdot 2^n + (U_0 \cdot V_1 + U_1 \cdot V_0) \cdot 2^{n/2} + U_0 \cdot V_0.$$

The multiplication $U_1 \cdot V_1$ takes $O(1)$ time (multiplying two bits) and the multiplications $U_0 \cdot V_1$ and $U_1 \cdot V_0$ take $O(n)$ time (multiplying one bit by n bits); so the above computation of $U \cdot V$ takes the amount of time to multiply the two $n/2$-bit numbers plus $O(n)$ time, as promised.

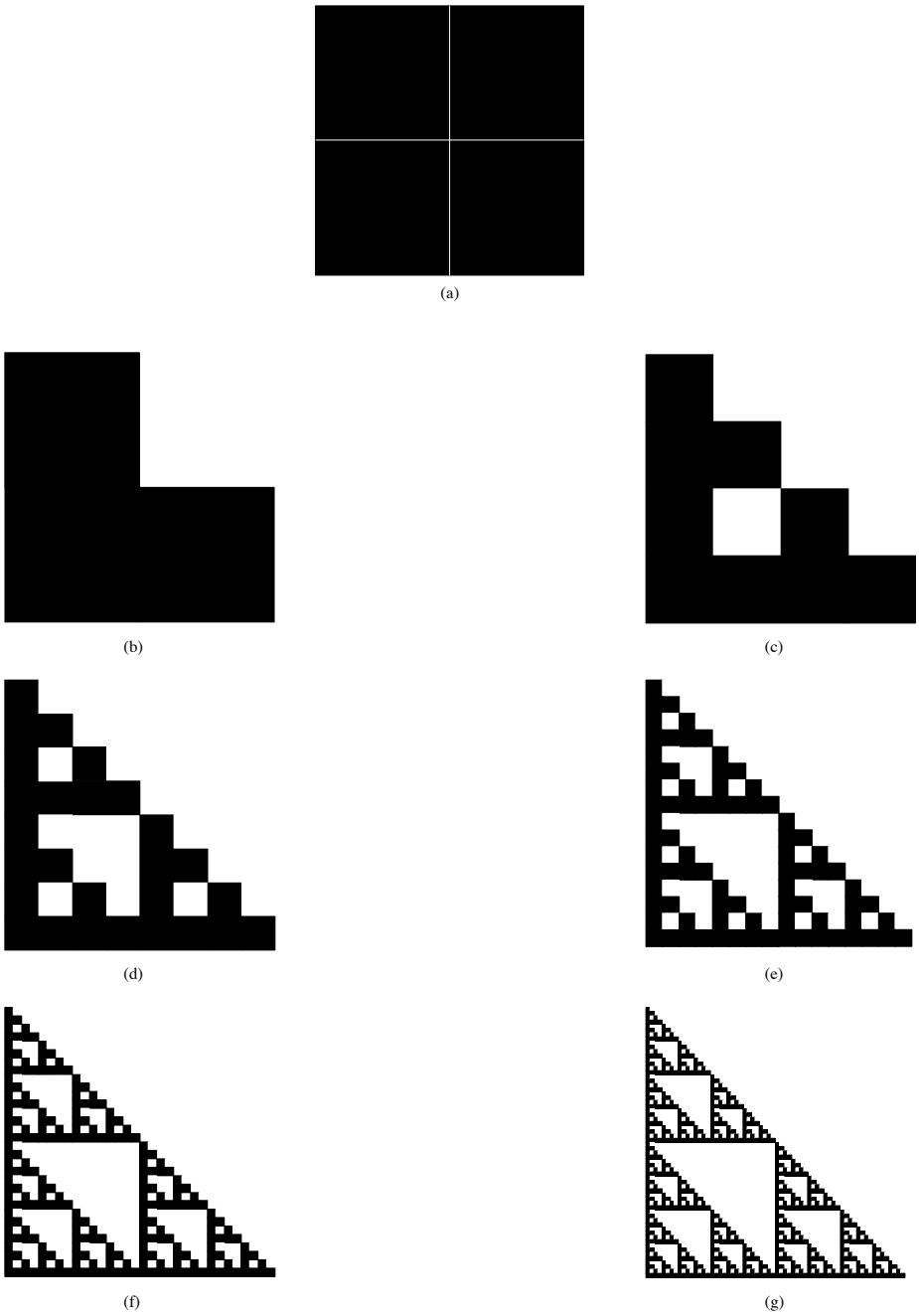


Figure 2.3: Visualization of the running time to compute $X \cdot Y$ as in (2.8)

This algorithm for multiplying was discovered by Anatoly Karatsuba. It is shown, in pseudocode, below. Its input consists of two bit-arrays x and y, storing the binary representation of two natural numbers X and Y, with the most significant bit in position 1. We assume procedures $\textsc{Add}(x, y)$ and $\textsc{Sub}(x, y)$, which return bit arrays containing the binary representations of $X + Y$ and $X - Y$, respectively. These can be implemented with running time $O(n)$, where n is the maximum of the lengths of x and y.

```
    MULT(x, y)
 1  n := max(length(x), length(y))
 2  if n = 1 then return x[1] ∧ y[1]
 3  make x and y of equal length by padding with leading 0s
 4  x1 := x[1..⌈n/2⌉]; x0 := x[⌈n/2⌉ + 1..n]
 5  y1 := y[1..⌈n/2⌉]; y0 := y[⌈n/2⌉ + 1..n]
 6  u := ADD(x1, x0); v := ADD(y1, y0)
 7  a := MULT(x1, y1); b := MULT(x0, y0); c := MULT(u, v)
 8  d := SUB(c, ADD(a, b))
 9  append 2⌊n/2⌋ 0s to a
10  append ⌊n/2⌋ 0s to d
11  return ADD(a, ADD(d, b))
```

The correctness of this follows by a straightforward induction on n, where the critical observation for the induction step is (2.8).[2] Its running time is described by the recurrence $T(n) = 3T(n/2) + cn$, where c is some constant, since the time to perform all steps other than the three recursive calls is linear in n. By the Master Theorem then, $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585})$.

This is not the asymptotically fastest multiplication algorithm known. Karatsuba's result was published in 1962. Nearly ten years later, in 1971, Schönhage and Strassen discovered an algorithm for multiplication of integers that runs in time $\Theta(n \log n \log\log n)$, and much later, in 2019, Harvey and van der Hoeven discovered a multiplication algorithm that runs in time $\Theta(n \log n)$. However, the constants being "absorbed" in the big-oh notation in these algorithms are astronomically large. It is not known if $\Theta(n \log n)$ is best possible asymptotically.

2.4 Strassen's matrix multplication algorithm

Throughout this section, A and B are $n \times n$ matrices of real numbers, and $C = A \cdot B$ is their product. Matrix multiplication is a fundamental operation: A and B represent linear transformations in n-dimensional space, and their product $C = A \cdot B$ represents the *composition* of these two transformations: first apply B and then apply A to the result. We will denote the number in the i-th row and j-th column of matrix A as a_{ij}, and similarly for matrices B and C.

[2] Not quite: Equation (2.8) was based on the assumption that n is a power of 2, and therefore even. If n is odd, the low-order "half" X_0 has $\lfloor n/2 \rfloor$ bits, and therefore the correct shift of the term $X_1 \cdot Y_1$ is by $2\lfloor n/2 \rfloor$, not n, bits. This is reflected in line 9 of the pseudocode.

Recall that c_{ij} is the dot product of the i-th row of A and the j-th column of B; that is,

$$c_{ij} = \sum_{k \in [1..n]} a_{ik} b_{kj}. \tag{2.9}$$

In this section we will describe an important algorithm for multiplying matrices, discovered by Volker Strassen. Unlike the previous section we will now assume that the numbers stored in the entries of the matrix are small enough to fit in a word of memory. So now performing basic arithmetic operations on them takes $O(1)$ time each. Thus, each entry of C can be computed in $O(n)$ time based on the definition (2.9), and we can compute the product of two $n \times n$ matrices in $O(n^3)$ time. Can we do better?

Consider the recursive decomposition of the matrices A, B, and $C = A \cdot B$ shown in Figure 2.4; for simplicity we assume that n is a power of 2. So A_{11} and A_{12} are the upper left

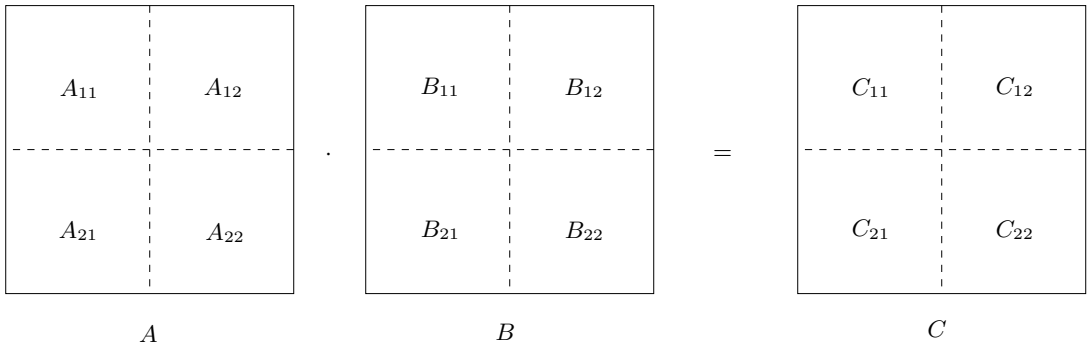


Figure 2.4: Recursive decomposition of $n \times n$ matrices A, B, and $C = A \cdot B$

and upper right quarters of A, and A_{21} and A_{22} are the lower left and lower right quarters of A; all these are $n/2 \times n/2$ matrices. Similarly for B and C.

Straight from the definition (2.9) you can verify that

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned} \tag{2.10}$$

This immediately suggests a divide-and-conquer algorithm to compute the product C of two $n \times n$ matrices A and B: First performing the eight multiplications of $n/2 \times n/2$ matrices in (2.10) and then perform the four additions of $n/2 \times n/2$ matrices in (2.10). Recall that if $D = A + B$, $d_{ij} = a_{ij} + b_{ij}$, and so we can perform each of the four additions in $O(n^2)$ time. Therefore, if $T(n)$ is the time to multiply two $n \times n$ matrices based on this divide-and-conquer algorithm we have that $T(n) = 8T(n/2) + O(n^2)$. In terms of the parameters of the Master Theorem, $a = 8$, $b = 2$, and $d = 2$. Since $a > b^d$, the third case applies, and so

$T(n) = O(n^{\log_2 8}) = O(n^3)$. This is no better than the running time of the straightforward algorithm that computes each entry of C by following the definition (2.9).

Strassen made the remarkable discovery that we can compute C by performing *seven*, instead of eight, multiplications of $n/2 \times n/2$ matrices, plus some quadratic amount of work. Here are the computations performed by his algorithm:

$$\begin{aligned}
M_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
M_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\
M_4 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_5 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_6 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_7 &= (A_{21} + A_{22}) \cdot B_{11}
\end{aligned} \tag{2.11}$$

Then:

$$\begin{aligned}
C_{11} &= M_1 + M_2 - M_4 + M_6 \\
C_{12} &= M_4 + M_5 \\
C_{21} &= M_6 + M_7 \\
C_{22} &= M_2 - M_3 + M_5 - M_7.
\end{aligned}$$

Alas, I cannot offer any insight into this stupendous choice of computations, but I have verified that they work — and so can you! This amounts to seven multiplications and 18 additions or subtractions of $n/2 \times n/2$ matrices. So, the time to perform these computations is described by the recurrence $T(n) = 7T(n/2) + O(n^2)$. In terms of the Master Theorem parameters, $n = 7$, $b = 2$, and $d = 2$, so $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$.

2.5 Closest pair of points

Let p and p' be two points on the plane, specified through their coordinates; i.e., $p = (x, y)$ and $p' = (x', y')$, where $x, y, x', y' \in \mathbb{R}$. The (Euclidean) distance between p and p', denoted $d(p, p')$ is the length of the straight segment that connects them; i.e., $d(p, p') = \sqrt{(y' - y)^2 + (x' - x)^2}$.

Consider n points on the plane. We want to find a pair among these points that are as close as possible. More precisely our problem is:
Input: A set $P = \{p_1, p_2, \ldots, p_n\}$ of n points, where $p_i = (x_i, y_i)$, $x_i, y_i \in \mathbb{R}$.
Output: Points $p, q \in P$, $p \neq q$, such that for all $p', q' \in P$, if $p' \neq q'$ then $d(p, q) \leq d(p', q')$.

The brute-force algorithm for this problem is to consider every pair of (distinct) points in P, compute the distance between each pair, and output a pair with the smallest distance. This takes $\Theta(n^2)$ time.

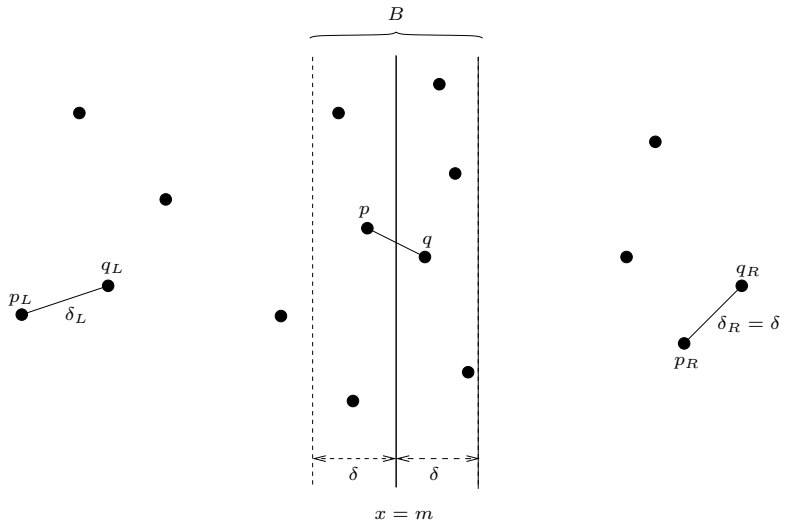


Figure 2.5: Illustration for the closest pair of points divide-and-conquer algorithm

It turns out that we can do better using divide-and-conquer. The most interesting part of this algorithm is the "conquer" step, where we combine the recursively computed solutions to the smaller subproblems to obtain the solution to the given problem. For simplicity we will assume that no two points in P have the same x-coordinate; see Exercise 2.14. Let $m \in \mathbb{R}$ be such that $\lceil n/2 \rceil$ of the points in P have x-coordinates less than m and $\lfloor n/2 \rfloor$ of them have x-coordinates greater than m. We can find m by sorting the points in P by their x-coordinates and picking m to be the average of the x-coordinates of the two middle points in this sorted list. Thus the vertical line $x = m$ bisects the plane so that (roughly) half of the points in P are on either side of it; see Figure 2.5.

We can recursively find a closest pair of points p_L and q_L among those on the left side of this bisector, and a closest pair p_R and q_R among those on the right side of it. Let $\delta_L = d(p_L, q_L)$, $\delta_R = d(p_R, q_R)$, and $\delta = \min(\delta_L, \delta_R)$. (See Figure 2.5 where it happens that $\delta = \delta_R$.) An overall closest pair of points among all in P is one of these two pairs *or* two points p and q, where p is on the left and q on the right side of the bisector $x = m$. Notice that only points in P that lie in the narrow band that is δ to the left and δ to the right of the bisector can satisfy this requirement; let B be the set of points in P that lie within this band — see the dotted vertical lines in Figure 2.5. So in searching for points on either side of the bisector whose distance from each other is less than δ, we can focus only on points in B.

Unfortunately, it is possible that *all* points in P are in B, so trying all $n^2/4$ pairs of points that lie on either side of the bisector, even within B, does not get us out of the quadratic running time we are hoping to beat. We now explain how we can reduce this to a *linear* number of comparisons.

Suppose we sort the points in B in increasing y-coordinate, with ties broken arbitrarily. If a point q is after point p in this sorted order, we will say that q is "above" p (and that p is "below" q). We have:

Lemma 2.3 *For all points $p, q \in B$, if q is above p and $d(p,q) < \delta$, there are at most six points in B above p and below q.*

PROOF. Fix any point $p \in B$. Draw a horizontal line that goes through p and another horizontal line δ above it as shown in Figure 2.6. (In this figure we have chosen p to be to

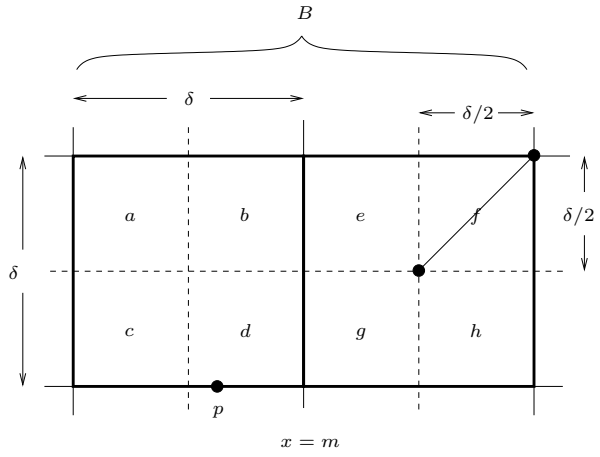


Figure 2.6: Possible location of points above p that could be closer than δ to it

the left of the bisector; the situation is completely symmetric if p is to the right.) These two horizontal lines and the three vertical lines consisting of the bisector $x = m$ and the two vertical lines δ to the left and to the right of it define two squares of side δ, shown with heavy perimeters in Figure 2.6.

Let $q \in B$ be a point that is above p so that $d(p,q) < \delta$. Note that q must be in one of these two squares: all points in B that are above p and outside these squares are at distance more than δ from p. (In fact, q must be in the opposite square than the one in which p lies, since the square in which p lies is entirely on one side of the bisector, and there are no points in P that are closer than δ in either of these sides. It will simplify the discussion and the algorithm, however, to consider points in both squares, as p could be in the other one.)

Further divide each of the two squares of side δ into four squares, each of side $\delta/2$: in Figure 2.6 these are labeled a, b, c, d for the square on the left side of the bisector and e, f, g, h for the square on the right side of it. Each of these smaller squares can contain at most one point in B: This is because the maximum distance between any two points within each of these squares is the length of its diagonal, which is $\delta\sqrt{2}/2 < \delta$. But each of the squares a-h lies entirely to the left or entirely to the right of the bisector, and we know that the distance of any two points in P in each of these regions is at least δ. (For illustration, consider the upper right and lower left corner of square f in Figure 2.6: these cannot both be in B.)

Therefore each of the squares a-h contains at most one point in P, one of them being p itself. So, if there is some point $q \in B$ that is above p and $d(p,q) < \delta$, there can be at most six points above p and below q. □

This claim implies that, for any points $p, q \in B$, if there are at least seven points in B that are above p and below q, then $d(p, q) \geq \delta$. So in seeking for points above p that care close to it than δ, we can limit our search to the next seven points after p in B (recall that B is sorted by y-coordinate). This suggests a divide-and-conquer algorithm for finding a closest pair of points in P, shown in pseudocode and explained below.

The algorithm consists of a "driver" procedure ClosestPair(P), which creates two sorted lists: P_x contains the points in P sorted by x-coordinate and P_y contains the points in P sorted by y-coordinate. P_x helps find the bisector and P_y helps to check efficiently for pairs of points on either side of the bisector that are closer to each other than δ). ClosestPair then calls the recursive procedure RCP(P_x, P_y) that uses these sorted lists to find a closest pair of points (p^*, q^*).

RCP first splits P_x into two sublists of (roughly) equal size, L_x and R_x, containing the points on the left and right of the bisector, respectively, sorted by x-coordinate (line 4). It then computes the bisector m (line 5), and creates two sorted sublists of P_y, L_y and R_y containing the points on the left and right of the bisector, respectively, but now sorted by y-coordinate. These can be constructed by scanning P_y and appending to an initially empty L_y (respectively, R_y) the points with x-coordinate less than (respectively, greater than) m. Next RCP uses L_x and L_y (respectively, R_x and R_y) to recursively find the closest pair of points (p_L, q_L) (respectively (p_R, q_R)) among those to the left (respectively, right) of the bisector (line 7). It then sets (p^*, q^*) to the closest pair between (p_L, q_L) and (p_R, q_R), δ to the distance between them, and creates a list B of the points within δ of the bisector *sorted by y-coordinate* (lines 8-10). It can do this by scanning P_y and appending to an initially empty list the points (x, y) such that $|m - x| \leq \delta$. The procedure RCP then considers each point p on B in order of appearance and checks whether each of the up to seven points after p in B is closer to p than δ; if so, it adjusts the currently closest found pair of points (p^*, q^*) to the newly found closer pair and the distance δ between them (loop in lines 11-13). At the end of this procedure, (p^*, q^*) contains a closest pair of points (line 14).

ClosestPair(P)
$P_x :=$ the list of points in P sorted by increasing x-coordinate
$P_y :=$ the list of points in P sorted by increasing y-coordinate
return RCP(P_x, P_y)

```
   RCP(P_x, P_y)
 1 if |P_x| ≤ 3 then
 2     calculate all pairwise distances and return the closest pair
 3 else
 4     L_x := first half of P_x; R_x := second half of P_x
 5     m := (max x-coordinate in L_x + min x-coordinate in R_x)/2
 6     L_y := sublist of P_y of points in L_x; R_y := sublist of P_y of points in R_x
 7     (p_L, q_L) := RCP(L_x, L_y); (p_R, q_R) := RCP(R_x, R_y)
 8     δ := min(d(p_L, q_L), d(p_R, q_R))
 9     if d(p_L, q_L) = δ then (p*, q*) := (p_L, q_L) else (p*, q*) := (p_R, q_R)
10     B := sublist of P_y of points whose x-coordinates are within δ of m
11     for each p ∈ B in order of appearence on B do
12         for each of the (up to) next seven points q after p on B do
13             if d(p, q) < d(p*, q*) then (p*, q*) := (p, q)
14     return (p*, q*)
```

Using induction on the number of points in P we can prove that, if P_x and P_y contain the points in P sorted by x-coordinate and y-coordinate respectively, the call $\mathrm{RCP}(P_x, P_y)$ returns a closest pair of points among the points in P. The critical observation needed in the induction step is Lemma 2.3.

For the running time analysis, let n be the number of points in P (and so in P_x and P_y) and $T(n)$ be the running time of $\mathrm{RCP}(P_x, P_y)$. Assume that n is a power of 2. We claim that $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + cn$, for some constant c. This is because the algorithm calls itself twice on instances of half the size (see line 7), and requires $\Theta(n)$ time to divide up the input and to combine the results of the two smaller instances into the result of the original instance. To see the latter point (i.e., that the algorithm requires only $\Theta(n)$ time for the divide and combine steps), note that lines 4, 6, and 10 can be implemented in $\Theta(n)$ time each, as explained above. The loop in lines 11–13 takes $\Theta(n)$ time since there are $\Theta(n)$ points in B and each iteration takes $\Theta(1)$ time (the inner for loop is executed only up to seven times for each point $p \in B$). All other lines only require $\Theta(1)$ time. By the Master Theorem, we conclude that $T(n) = \Theta(n \log n)$.

The main procedure ClosestPair(P) requires $\Theta(n \log n)$ time to sort the points in P by x-coordinate and by y-coordinate in order to produce the lists P_x and P_y, and an additional $\Theta(n \log n)$ time for the call $\mathrm{RCP}(P_x, P_y)$. So the entire algorithm runs in $\Theta(n \log n)$ time.

2.6 Order statistics

Let $A = a_1, a_2, \ldots, a_n$ be a sequence of numbers, in arbitrary order. There are various quantities related to the numbers in this sequence that we may be interested in, such as their maximum, their minimum, and their average. All these can be computed in $O(n)$ time in straightforward ways.

Another quantity of interest is the ***median*** value of A. Informally, this is the "middle value" of A if we were to sort its elements. More precisely, if π is a permutation of the indices

$1..n$ such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$, the median of A is $a_{\pi(\lceil n/2 \rceil)}$. So the median of $5, 7, 2, 17, 11$ is 7, and the median of $5, 7, 2, 17, 11, 3$ is 5.

We can find the median of A in $\Theta(n \log n)$ time by sorting A; but can we find it in *linear* time, like the minimum, maximum, and average of A? We will bring divide-and-conquer to bear on this question. For reasons that will become apparent soon, it is useful to solve a more general problem, namely finding the k-th smallest element in A, for *any* $k \in [1..n]$. For simplicity, we will assume that the elements of A are distinct. We can make this assumption without loss of generality by replacing each element a_i in A by the pair (a_i, i) — so if two elements of A are equal, they become distinct by virtue of their positions in the list — where we order pairs lexicographically: $(a_i, i) < (a_j, j)$ if and only if $a_i < a_j$, or $a_i = a_j$ and $i < j$. So our problem is:

Input: An array $A[1..n]$ of distinct numbers (in any order), and an integer $k \in [1..n]$.
Output: The k-th smallest element in A.

This is known as the ***selection*** or ***order statistics*** problem. Following is a divide-and-conquer algorithm to solve it, where $|B|$ denotes the length of array B.

```
SELECT(A, k)
 1  if |A| < 39 then
 2      sort A
 3      return A[k]
 4  else
 5      s := CHOOSESPLITTER(A)    ▶ returns some element of A
 6      A^< := array of elements in A that are < s (not necessarily sorted)
 7      A^> := array of elements in A that are > s (not necessarily sorted)
 8      if k ≤ |A^<| then return SELECT(A^<, k)
 9      else if k = |A| + 1 then return s
10      else return SELECT(A^>, k − (|A^<| + 1))
```

Figure 2.7: Order statistics algorithm

If the array A is small enough, we can solve the problem by brute force: sort the array and return its k-th element (lines 1-3). For the moment ignore the specific choice of $n < 39$ for when A is "small enough": We could have chosen any constant $n \geq 1$; the specific choice becomes relevant for the analysis of a version of the algorithm we will see in Section 2.6.2. In the nontrivial case when n is large enough, the algorithm selects some element s of the array A, called the ***splitter***. We have left unspecified how the splitter is selected; we will return to this soon. The algorithm uses the splitter to create two arrays $A^<$ and $A^>$, consisting of the elements less than and greater than s, respectively (lines 6-7). Then it recurses on one of these smaller arrays, depending on which of them contains the k-th smallest item of A (lines 8 or 10); if we were lucky and the k-th smallest item in A is s, we are done (line 9). Figure 2.8 may be useful.

Note the usefulness of the generality afforded by the selection parameter k: If we were trying to solve directly the problem of selecting the median (i.e., for fixed $k = n/2$) we wouldn't be

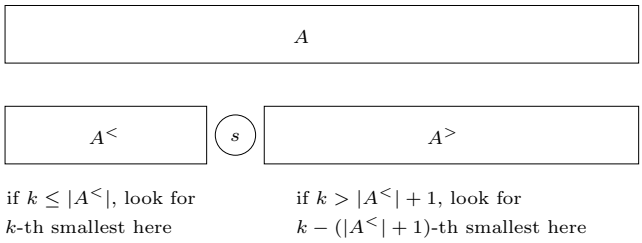


Figure 2.8: Illustration for SELECT

able to recurse on the median-finding problem, since in the recursive call we would no longer be seeking the median of the smaller array. This should remind you of something you have encountered in the past: Sometimes to prove a statement by induction it pays to *generalize* it and prove a *stronger* statement. This works because a stronger statement yields a stronger induction hypothesis that can be exploited more effectively, as is the case in this algorithm.

The correctness of this algorithm follows by an easy induction on $|A|$. Its running time depends on the chosen splitter. If s is the the i-th smallest element of A, the running time $T(n)$ is described by the recurrence

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T\big(\max(i, n-i)\big) + cn, & \text{otherwise} \end{cases}$$

for some constant c: The term $T\big(\max(i, n-i)\big)$ accounts for the time required for the recursive call and the linear term cn is for the time required to construct $A^<$ and $A^>$ from A. The minimum value for $\max(i, n-i)$, which then minimizes $T(n)$, is when $i = n/2$. This makes the recurrence $T(n) = T(n/2) + cn$, which means that $T(n) = \Theta(n)$ (by the first case of the Master Theorem since here $a = 1$, $b = 2$, $d = 1$).

But for this we need the splitter to be the median of A: this is a chicken-and-egg situation. Fortuntely, we don't need to rely on a splitter that does as perfect a job as the median. If the splitter results in the longer of $A^<$ and $A^>$ being a *fixed* fraction n/b of n, for *any* $b > 1$, however close to 1, we have that $T(n) = T(n/b) + cn$, and the solution to this is still $T(n) = \Theta(n)$. In fact, we don't even have to rely on the splitter doing this *every* time; for example, if the splitter only reduces the size of the array from n to n/b every third time, rather than every time, this at most triples the amount of time the algorithm takes — so it is still linear. This gives us hope that we could select the splitter in a fairly loose manner and still end up with a linear algorithm.

Before we explore ways to do this, let us consider the other side of the possible performance of the algorithm: What would be a *bad* choice for a splitter? Clearly the worst choice is one that reduces the size of the array as little as possible, from n to $n-1$; this happens if the splitter is the minimum or maximum element of the array. In that case, the running time of the algorithm is described by the recurrence $T(n) = T(n-1) + cn$. By "unwinding" this recurrence we get $T(n) = \Theta(n^2)$. So, the performance of the algorithm can range from linear to quadratic, depending on the choice of the splitters.

2.6.1 Randomized selection

Let's explore the simplest imaginable method for selecting the splitter: pick one *uniformly at random*! That is, the ChooseSplitter procedure used by the algorithm in Figure 2.7 is as follows, where Random(n) returns an integer in the range $1..n$, each with probability $1/n$.

ChooseSplitter(A):
$r :=$ Random($|A|$)
return $A[r]$

This seems an unusual thing for an algorithm to do: We are accustomed to algorithms making careful, deliberate choices rather than random ones. Note, however, that we are not saying "pick the splitter *arbitrarily*"; there is a method to how s is chosen, and the method is to pick each of the elements of the array with *equal probability*. We will call this version of the algorithm in Figure 2.7 ***randomized*** Select, abbreviated R-Select.

It is clear from our previous discussion that *in the worst case* the running time of this algorithm is $\Theta(n^2)$: It could be that every time the algorithm chooses a splitter, it happens to pick the minimum or maximum element in the array. We will prove, however, that *the expected* running time of the algorithm is $\Theta(n)$.

First note that if we fix the input of the algorithm, various quantities, such as the number of times the loop is executed, are no longer fixed; they depend on the random choices made by the algorithm in line 5, i.e., they are *random variables*. We will be interested in the expected value of the random variable

$$X = \text{number of steps executed by the algorithm.} \tag{2.12}$$

We will call a splitter s "good" if at least one-quarter of the array's elements are less than s *and* at least one-quarter of the array's elements are greater than s. As can be seen from Figure 2.9, which shows A as if A were sorted, half of the elements of A are good splitters. Since we are choosing the splitter uniformly at random, the probability that the splitter chosen in line 5 is good is $1/2$.

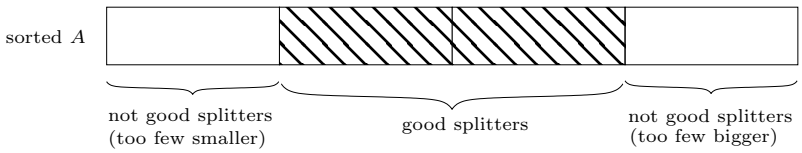


Figure 2.9: Good splitters in (sorted) A

Therefore the expected number of trials until a good splitter is chosen is 2. This makes sense intuitively, and can be verified mathematically as follows. Consider the random variable

$$Z = \text{number of trials until a good splitter is chosen.} \tag{2.13}$$

As we just saw, the probability that a good splitter is chosen in one trial is $1/2$; and, in general, the probability that a good splitter is chosen in exactly i trials is $1/2^i$: this is the probability

of the event that *each* of the first $i-1$ trials results in choosing a bad splitter and the i-th trial results in choosing a good one. Therefore,

$$\mathbf{E}[Z] = \frac{1}{2}\cdot 1 + \frac{1}{2^2}\cdot 2 + \frac{1}{2^3}\cdot 3 + \cdots = \sum_{i=1}^{\infty}\frac{i}{2^i} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2. \tag{2.14}$$

(The penultimate equality is the special case of the infinite series $\sum_{i=1}^{\infty} ix^i$ for $x = 1/2$. This infinite series converges to $x/(1-x)^2$ when $0 < x < 1$. This can be proved by differentiating both sides of the the well-known formula for the geometric series $\sum_{i=0}^{\infty} x^i = 1/(1-x)$ and then multiplying both sides of the resulting equality by x.)

Consider a call to SELECT in which the array A has n elements. If the splitter chosen in line 5 is good then the length of the array on which the algorithm recurses in lines 8 or 10, is at most $3n/4$. That is, the size of the array shrinks by a factor of at least $b = 4/3$. We will divide the sequence of recursive calls to SELECT into a sequence of *phases*: The algorithm remains in phase 0 as long as the recursive calls operate on an array of size greater than $3n/4$. The moment a recursive call is made to an array of length at most $(3/4)n$, the algorithm enters phase 1 and remains in that phase as long as the recursive calls operate on an array size greater than $(3/4)^2 n$ — i.e., the input size is reduced again by a factor of $4/3$. In general, the algorithm is in phase i during recursive calls that operate on an array of size at least $(3/4)^i n$ and less than $(3/4)^{i+1} n$. Thus,

$$\text{each time a good splitter is chosen the algorithm enters a new phase.} \tag{2.15}$$

Note that during an execution, the algorithm may skip some phases; this occurs if the random choice of splitter happens to be very favourable.

Let's define the random variable

$$X_i = \text{number of steps executed by the algorithm } \textit{while in phase } i. \tag{2.16}$$

So $X = \sum_{i=0}^{\infty} X_i$. Our goal now is to compute $\mathbf{E}[X_i]$. We have

$$X_i \le (\max \#\text{ of steps of } \textit{one} \text{ recursive call in phase } i) \times (\#\text{ of recursive calls in phase } i).$$

Since the array on which a phase i recursive call operates has size at most $(3/4)^i n$, the maximum number of steps of a recursive call in phase i (excluding the number of steps performed by the recursive call it makes, if any) is $c(3/4)^i n$, for some constant c. Also, by (2.15), the number of recursive calls in phase i is at most the number of trials before a good splitter is chosen — i.e., the random variable Z defined in (2.13). So, $X_i \le c(3/4)^i n \cdot Z$. By linearity of expectation and (2.14), $\mathbf{E}[X_i] \le c(3/4)^i n \cdot \mathbf{E}[Z] = 2cn(3/4)^i$. Also by linearity of expectation,

$$\mathbf{E}[X] = \mathbf{E}[\sum_{i=0}^{\infty} X_i] = \sum_{i=0}^{\infty}\mathbf{E}[X_i] \le 2cn\sum_{i=0}^{\infty}(3/4)^i = 8cn.$$

Therefore, the expected running time of R-SELECT is $\Theta(n)$.

This is great news, because this way of selecting a splitter is as simple as it gets: close your eyes and choose any element of A with equal probability! Unfortunately, as noted earlier, the *worst-case* running time of the algorithm is $\Theta(n^2)$.

This algorithm is an instance of a type of algorithms known as ***randomized*** or ***probabilistic*** algorithms. These are algorithms that can "flip a coin" to determine the next action. A useful way to think of such algorithms is that they operate against an "adversary" whose task is to choose the input so as to make the algorithm's job as hard as possible. An adversary who knows exactly what the algorithm does can design the input so as to cause the algorithm to take the maximum number of steps. By making random choices, the algorithm makes it difficult for the adversary to choose a "bad" input. To see this, suppose we used a different strategy for choosing the splitter; say that we "arbitrarily" choose the first element of the array. In that case the adversary could easily cause the algorithm to take a quadratic, instead of linear, number of steps by presenting the input array in sorted order.

Randomization is very powerful tool with many applications. In earlier courses you probably learned the widely used sorting algorithm QUICKSORT, which is very similar to SELECT, except that it recurses on both $A^<$ and $A^>$ in order to sort both sides — resulting in *expected* $\Theta(n \log n)$ running time. Randomization is also used in network routing (to avoid congestion), in load balancing in distributed systems (to avoid overloading), in heuristics for hard optimization problems, in data structures (e.g., hash functions, skip lists), and many other applications. It is an important algorithmic tool, but one for which unfortunately we do not have much time in this course.

2.6.2 Deterministic selection

It turns out that we can ensure linear *worst-case* running time for SELECT by choosing the splitter in a much more careful (and deterministic) way. The CHOOSESPLITTER procedure is now as follows:

CHOOSESPLITTER(A)
1 partition A into $\lfloor n/5 \rfloor$ groups of 5 elements each, with ≤ 4 (arbitrarily chosen) elements left out
2 find the median of each group, by sorting its five elements
3 $M :=$ array of the medians of the $\lfloor n/5 \rfloor$ groups
4 **return** SELECT($M, \lceil |M|/2 \rceil$)

Note that the splitter chosen is the median of the medians (of the groups) and is found using the SELECT algorithm recursively. So now, each call to SELECT results in *two* recursive calls: one to find the splitter (called within CHOOSESPLITTER in line 5 of SELECT) and one to find the element being sought in $A^<$ or $A^>$ (lines 8 or 10 of SELECT). We will refer to this algorithm as D-SELECT (for "deterministic select") to distinguish it from the randomized select R-SELECT algorithm of Section 2.6.1.

Figure 2.10 is a visualization that helps explain why the splitter selected in this manner is guaranteed to have at least one quarter of the elements less than or equal to it, and at least one quarter greater than or equal to it. The dots represent the elements of A partitioned into groups of five (the up to four elements left out in line 1 of CHOOSESPLITTER are not shown).

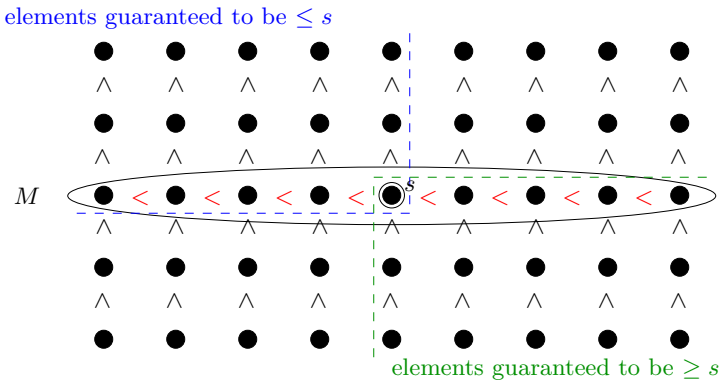


Figure 2.10: Choice of splitter by D-SELECT

The "vertical" inequalities (shown in black) are established by CHOOSESPLITTER in line 2, when it sorts each group of five elements. We emphasize that the "horizontal" inequalities (shown in red among the medians of the groups) are not actually established by the algorithm: the algorithm does *not* sort M! We *imagine* that the groups are sorted by their medians to help us see why about one quarter of the elements of A (those above and to the left of the broken line shown in blue) are guaranteed to be less than or equal to the splitter; and about one quarter of the elements of A (those below and to the right of the broken line shown in green) are guaranteed to be greater than or equal to the splitter. Therefore the number of elements in the array on which the algorithm recurses has at most three-quarters of the elements. A more careful argument follows.

Claim 2.3.1 *Each of $A^<$ and $A^>$ (lines 6–7 of* SELECT*) has at most $\lfloor 3n/4 \rfloor$ elements.*

PROOF. The number of elements less than or equal to the splitter s is at least $3 \cdot \lceil \lfloor n/5 \rfloor / 2 \rceil$. To see why, first note that there are $\lfloor n/5 \rfloor$ medians of groups, and (the ceiling of) half of them are less than or equal to their median s. So, $\lceil \lfloor n/5 \rfloor / 2 \rceil$ medians are less than or equal to s. Each of these medians is less than or equal to three of the five elements of its group. So, there are at least $3 \cdot \lceil \lfloor n/5 \rfloor / 2 \rceil$ elements less than or equal to the splitter. By the same argument there are equally many elements greater than or equal to the splitter. Therefore, the larger of the arrays $A^<$ and $A^>$ on which the algorithm may recurse has at most $n - 3 \cdot \lceil \lfloor n/5 \rfloor / 2 \rceil$ elements. (Note that in our accounting here we are *not* ignoring the up to four elements of A that may have been left out when partitioning A into $\lfloor n/5 \rfloor$ groups of five elements each.)

Since we are computing $A^<$ and $A^>$ in lines 6 and 7, we have that $n \geq 39$ (see line 1 of

SELECT). We will show that, for all $n \geq 39$, $n - 3 \cdot \lceil \lfloor n/5 \rfloor / 2 \rceil \leq \lfloor 3n/4 \rfloor$.[3] We have:

$$\begin{aligned}
n - 3 \cdot \left\lceil \frac{\lfloor n/5 \rfloor}{2} \right\rceil &\leq n - 3 \cdot \frac{\lfloor n/5 \rfloor}{2} && \text{[because } \lceil x \rceil \geq x, \forall x] \\
&\leq n - 3 \cdot \frac{(n-4)/5}{2} && \text{[because } \lfloor \frac{x}{b} \rfloor \geq \frac{x-(b-1)}{b}, \forall x, b] \\
&\leq n - \frac{3n-12}{10} \\
&= \frac{7n+12}{10} \\
&\leq \frac{3n-3}{4} && \text{[true } \forall n \geq 39] \\
&\leq \left\lfloor \frac{3n}{4} \right\rfloor. && \text{[because } \frac{x-(b-1)}{b} \leq \lfloor \frac{x}{b} \rfloor, \forall x, b]
\end{aligned}$$

□

Therefore the running time $T(n)$ of the algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} c, & \text{if } n < 39 \\ T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor) + cn, & \text{otherwise} \end{cases} \qquad (*)$$

for some constant c. In the recursive case, the term $T(\lfloor n/5 \rfloor)$ is the time used by the first recursive call to D-SELECT, which finds the median of M; the term $T(\lfloor 3n/4 \rfloor)$ is (an upper bound on) the time used by the second recursive call, made to the array $A^<$ or $A^>$; and cn is the linear term that accounts for the time to sort the $\lfloor n/5 \rfloor$ groups of five elements each and to form the array of medians M (lines 1-3 of CHOOSESPLITTER), and the time to form the arrays $A^<$ and $A^>$ (lines 6 and 7 of SELECT), once the splitter is determined.

Using induction we can prove that $T(n) \leq 20cn$. This is obvious by $(*)$ for the base cases $n < 39$. For the induction step, for any $n \geq 39$, assume that the claim holds for values less than n. Then we have

$$\begin{aligned}
T(n) &\leq T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor) + cn && \text{[by } (*)] \\
&\leq 20c\lfloor n/5 \rfloor + 20c\lfloor 3n/4 \rfloor + cn && \text{[by the induction hypothesis]} \\
&\leq 20c(n/5) + 20c(3n/4) + cn && \text{[since } \lfloor x \rfloor \leq x, \forall x] \\
&= 20cn,
\end{aligned}$$

as wanted.

[3] This bound actually holds, not just for all $n \geq 39$, but for all $n \geq 5$ with the exception of $n = 13$ and $n = 14$. At small scales, however, bounding the rounding functions (floor and ceiling) with simple inequalities as stated in the proof below has an outsize effect and does not yield the desired bound. The only way I could show that this inequality holds for all $n \geq 5$ except $n = 13$ and $n = 14$ was to consider many cases, resulting in an inelegant and laborious proof. Luckily, when $n \geq 39$, the effects of the simple bounds on the rounding functions are small enough, yielding the less laborious proof shown here.

Therefore, the (worst-case) running time of D-SELECT is $\Theta(n)$, where n is the number of elements in the array A. This is in contrast to the randomized algorithm R-SELECT, whose *expected* running time is $\Theta(n)$ but whose *worst-case* running time is $\Theta(n^2)$.

2.7 The fast Fourier transform[4]

Let $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_mx^m$ by two univariate polynomials of degree m. Their product is a polynomial of degree $2m$, $A(x) \cdot B(x) = C(x) = c_0 + c_1x + c_2x + \cdots + c_{2m}x^{2m}$, where

$$\begin{aligned}
c_0 &= a_0b_0 \\
c_1 &= a_0b_1 + a_1b_0 \\
c_2 &= a_0b_2 + a_1b_1 + a_2b_0 \\
&\vdots \\
c_m &= a_0b_m + a_1b_{m-1} + a_2b_{m-2} + \cdots + a_mb_0 \\
&\vdots \\
c_{2m-2} &= a_{m-2}b_m + a_{m-1}b_{m-1} + a_mb_{m-2} \\
c_{2m-1} &= a_{m-1}b_m + a_mb_{m-1} \\
c_{2m} &= a_mb_m;
\end{aligned}$$

that is, for each $k \in [0..2m]$,

$$c_k = \sum_{\substack{0 \le i,j \le m \\ i+j=k}} a_ib_j.$$

If we were to compute the c_k's directly from the definition, the required number of arithmetic operations (additions and multiplications) is $\Theta(n^2)$. It turns out that we can compute them in only $O(n \log n)$ arithmetic operations using a very important algorithm, the ***fast Fourier transform*** (or ***FFT***, for short).

Digression Why should we be interested in multiplying polynomials efficiently? One reason is that the asymptotically fastest known *integer* multiplication algorithms (e.g., the algorithms mentioned at the end of Section 2.3) are based on polynomial multiplication and in fact make use of the FFT.

But a far more important reason is that the FFT is indispensable for signal processing applications. Consider the two $(m + 1)$-dimensional vectors consisting of the coefficients of

[4] If your recollection of complex numbers is a bit hazy, you may want to review Appendix A before reading this section. You will also need some very basic concepts from linear algebra such as matrix multiplication, matrix invertibility, etc.

$A(x)$ and $B(x)$, i.e., $\vec{a} = (a_0, a_1, \ldots, a_m)$ and $\vec{b} = (b_0, b_1, \ldots, b_m)$. The ***convolution*** of these vectors is the $(2m+1)$-dimensional vector $\vec{c} = (c_0, c_1, \ldots, c_{2m})$ consisting of the coefficients of the product $C(x) = A(x) \cdot B(x)$. We write $\vec{c} = \vec{a} * \vec{b}$.

This may appear a strange operation, but it comes up in a surprising number of contexts. Intuitively it captures "interactions" between the two vectors $\vec{a}$ and $\vec{b}$: Suppose we reverse $\vec{b}$ and we "slide" it under $\vec{a}$ to overlap with it in every possible way, as shown in Figure 2.11 for two four-dimensional vectors. Then every c_k is the dot-product of the parts of the vectors that overlap (i.e., "interact" in some fashion). The interaction is weaker, or simpler, when the region of overlap is small and stronger, or more complex, when the region of overlap is large.

			a_0	a_1	a_2	a_3				
b_3	b_2	b_1	b_0							$c_0 = a_0b_0$
			a_0	a_1	a_2	a_3				
	b_3	b_2	b_1	b_0						$c_1 = a_0b_1 + a_1b_0$
			a_0	a_1	a_2	a_3				
		b_3	b_2	b_1	b_0					$c_2 = a_0b_2 + a_1b_1 + a_2b_0$
			a_0	a_1	a_2	a_3				
			b_3	b_2	b_1	b_0				$c_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$
			a_0	a_1	a_2	a_3				
				b_3	b_2	b_1	b_0			$c_4 = a_1b_3 + a_2b_2 + a_3b_1$
			a_0	a_1	a_2	a_3				
					b_3	b_2	b_1	b_0		$c_5 = a_2b_3 + a_3b_2$
			a_0	a_1	a_2	a_3				
						b_3	b_2	b_1	b_0	$c_6 = a_3b_3$

Figure 2.11: Convolution of two 4-dimensional vectors

Just as polynomial multiplication can be defined between polynomials of different degrees, convolution can be defined between vectors of different lengths in exactly the same way: one vector (reversed) is "sliding" under another, possibly of different length. The meaning of the result is the same: a vector that represents the possible linear interactions of the two vectors at all degrees of overlap.

What does all this have to do with signal processing? Vectors are used to represent signals. For example, an audio signal consists of a sequence of sound samples taken at high frequency (something like 96,000 samples per second for high-fidelity sound); physically, these samples are air-pressure changes converted to voltage levels by a microphone. By taking the convolution of such a signal with specially designed "filters", we can perform operations such as removing certain frequencies from the audio signal (for example, to eliminate an undesirable high-pitched sound that was picked up by the microphone). Similar filtering operations are routinely applied to all kinds of signals (image, video, economic, ...) for all kinds of purposes. The FFT is the algorithm that, by turning what appears to be an $\Omega(n^2)$ computation to a much faster $O(n \log n)$ one, makes signal processing feasible in practice. **End of Digression**

Back to multiplying polynomials. An important fact about polynomials states that a univariate polynomial of degree n is uniquely determined by its values at $n+1$ values of its variable x. More precisely,

Theorem 2.4 (Lagrange Interpolation Theorem) *Let n be any positive integer, m be any positive integer such that $m \leq n+1$, $x_1, \ldots, x_m$ be distinct values and $y_1, \ldots, y_m$ be arbitrary values.*

- *If $m = n+1$, there is exactly one polynomial $A(x)$ of degree $\leq n$ such that $y_i = A(x_i)$, for all $i \in [1..m]$.*
- *If $m \leq n$, there are infinitely many polynomials $B(x)$ of degree n such that $y_i = B(x_i)$, for all $i \in [1..m]$.*

For example, two points uniquely determine a straight line, and given only one point there are infinitely many straight lines that go through it. Similarly, three points uniquely determine a parabola (or straight line, if they happen to be co-linear), and given only two points there are infinitely many parabolas that go through them; and so on for any n.

The process of recovering the unique polynomial of degree $\leq n$ from $n+1$ pairs $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ is called ***interpolation***.

Digression This theorem is very important in mathematics and it has an interesting application to cryptography. Suppose we have a secret — say the password to a file that contains sensitive information about a company. It is too risky for only one employee to know the secret; if that employee is corrupt, the file can be leaked to the company's competitors.

Here is an idea to avoid this, called ***secret sharing***. Let k be the number of employees. *Each* of them will receive a "share" of the secret. Let $n < k$ be a parameter that we can choose. We want the k shares to have two properties:
(a) *any* $n+1$ shares suffice to reconstruct the secret; and
(b) *any* n (or fewer) shares are insufficient to reconstruct the secret.
The parameter n is chosen based on the level of trust we have in the employees: It should be such that any set of $n+1$ employees contains at least one honest person, so the remaining n, even if they are all corrupt, don't have enough shares to reconstruct the secret. (Note that we don't need to know, individually, who is honest; just that any large enough group contains at least one honest employee.)

Here is one way to do this: Let a_0 be the secret. Pick coefficients $a_1, a_2, \ldots, a_n$, and let $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$. Pick k distinct values $x_1, x_2, \ldots, x_k$ arbitrarily, and generate k shares of the secret, where each share is $\big(x_i, A(x_i)\big)$, $i \in [1..k]$. Then give share $\big(x_i, A(x_i)\big)$ to employee i.

Any subset of $n+1$ employees have the values of the polynomial $A(x)$ at enough points to interpolate it and discover the coefficients $a_0, a_1, a_2, \ldots, a_n$ — and, in particular, the secret a_0. Any subset of n or fewer individuals cannot interpolate the polynomial from their values and therefore cannot reconstruct the secret — exactly the properties we want.

This scheme is known as ***Shamir's Secret Sharing*** after its inventor Adi Shamir (the "S" in the RSA cryptosystem). It is used in applications such as digital wallets, distributed

cloud storage, and password management. **End of Digression**

By the interpolation theorem, there are two ways we can represent unambiguously a polynomial $A(x) = a_0 + a_1 x + \ldots + a_m x^m$ of degree m:

- by the list of its $m+1$ coefficients $(a_0, a_1, \ldots, a_m)$ — this is called the ***coefficient representation*** of $A(x)$; or
- by a list of $n \geq m+1$ pairs $\big(v_0, A(v_0)\big), \big(v_1, A(v_1)\big), \ldots, \big(v_{n-1}, A(v_{n-1})\big)$, where $v_0, v_1, \ldots, v_{n-1}$ are distinct values — this is called a ***value representation*** of $A(x)$.

Note that the coefficient representation of a polynomial is unique, but there are many different value representations for it: the polynomial's evaluation in any $m+1$ or more distinct points yields a different value representation for that polynomial.

Let

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m, \text{ and}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_m x^m$$

be the two degree m polynomials whose product

$$C(x) = A(x) \cdot B(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2m} x^{2m}$$

we want to compute. By the interpolation theorem, $C(x)$ is uniquely determined by its values in any $n \geq 2m+1$ distinct values of x. So suppose we are given value representations of $A(x)$ and $B(x)$ at $n \geq 2m+1$ points $v_0, v_1, \ldots, v_{n-1}$:

$$\big(v_0, A(v_0)\big), \big(v_1, A(v_1)\big), \big(v_2, A(v_2)\big), \ldots, \big(v_{n-1}, A(v_{n-1})\big), \text{ and}$$
$$\big(v_0, B(v_0)\big), \big(v_1, B(v_1)\big), \big(v_2, B(v_2)\big), \ldots, \big(v_{n-1}, B(v_{n-1})\big).$$

Then we can compute the value representation of $A(x) \cdot B(x)$ at the same n values of x

$$\big(v_0, A(v_0) \cdot B(v_0)\big), \big(v_1, A(v_1) \cdot B(v_1)\big), \big(v_2, A(v_2) \cdot B(v_2)\big), \ldots, \big(v_{n-1}, A(v_{n-1}) \cdot B(v_{n-1})\big)$$

in $O(n)$ time.

So here is our plan for polynomial multiplication. Given the coefficient representations $\vec{a} = (a_0, a_1, \ldots, a_m)$ and $\vec{b} = (b_0, b_1, \ldots, b_m)$ of two polynomials $A(x)$ and $B(x)$ of degree m, we will compute the coefficient representation $(c_0, c_1, \ldots, c_{2m})$ of their product (i.e., the convolution of the two vectors $\vec{a}$ and $\vec{b}$) in three steps:

- **Evaluate:** Evaluate $A(x)$ and $B(x)$ at $n \geq 2m+1$ values.
- **Multiply:** Compute the value representation of $C(x) = A(x) \cdot B(x)$ at those values.
- **Interpolate:** Obtain the coefficient representation of $C(x)$ from this value representation.

As we just saw, we can perform the second step in $O(n)$ time. Evaluation seems problematic, however, because evaluating a polynomial of degree n at a *single* point (i.e., value of x) already seems to require $\Omega(n)$ arithmetic operations, and to do so for n points would take $\Omega(n^2)$ time (see Exercise 2.16). By choosing the evaluation points in a very special way, however, we

can use divide-and-conquer to evaluate $A(x)$ and $B(x)$ at these chosen points in time only $O(n \log n)$. Even more surprisingly, and using the exact same method, we can interpolate the coefficients of $A(x) \cdot B(x)$ from its value representation at these n points!

Consider the problem of evaluating a polynomial $D(x) = d_0 + d_1 x + d_2 x^2 + \cdots + d_{n-1} x^{n-1}$, at n points *of our choice*, where n is a power of 2. The key insight for bringing divide-and-conquer to bear in solving this problem is to write $D(x)$ as the sum of two polynomials $D_0(x^2)$ and $D_1(x^2)$, each of degree $n/2 - 1$, by separating the odd and even terms of $D(x)$ as follows (note that $n - 1$ is odd, since n is a power of 2):

$$\begin{aligned} D(x) &= d_0 + d_1 x + d_2 x^2 + d_3 x^3 + d_4 x^4 + d_5 x^5 + \cdots + d_{n-1} x^{n-1} \\ &= \underbrace{(d_0 + d_2 (x^2) + d_4 (x^2)^2 + \cdots + d_{n-2} (x^2)^{n/2-1}}_{D_0(x^2)} \\ &\quad + x \underbrace{\left(d_1 + d_3 (x^2) + d_5 (x^2)^2 + \cdots + d_{n-1} (x^2)^{n/2-1}\right)}_{D_1(x^2)} \\ &= D_0(x^2) + x D_1(x^2). \end{aligned} \tag{2.17}$$

As a result, we can evaluate $D(x)$ at *two* points, v and $-v$, at the cost of evaluating each of $D_0(x)$ and $D_1(x)$, whose degree is *half* the degree of D, at *one* point, v^2, and one additional multiplication and addition (or subtraction):

$$\begin{aligned} D(v) &= D_0(v^2) + v D_1(v^2), \text{ and} \\ D(-v) &= D_0(v^2) - v D_1(v^2). \end{aligned}$$

So, to evaluate the polynomial $D(x)$ of degree $n - 1$ at n points, chosen as the $n/2$ pairs $\pm v_0, \pm v_1, \ldots, \pm v_{n/2-1}$, we can evaluate *two* polynomials $D_0(x)$ and $D_1(x)$ of degree $n/2 - 1$ at the $n/2$ points $v_0^2, v_1^2, \ldots, v_{n/2-1}^2$ and perform $2n$ additional arithmetic operations (multiplications and additions or subtractions).

It looks like we are on the right track for a nice divide-and-conquer evaluation algorithm. To continue with this idea recursively, however, we must get the squares $v_0^2, v_1^2, \ldots, v_{n/2-1}^2$ to be $\pm$ pairs! If we work with real numbers this is impossible because the squares of all reals are non-negative. Luckily, we can resolve this problem by expanding our horizon and working in the domain of *complex numbers*, where we can take square roots of negative numbers — and so there are negative numbers that are squares! In fact, there is a particular choice of $n/2$ pairs of $\pm$ complex numbers that makes the recursion work perfectly — namely, the n-th complex roots of unity, i.e., the n solutions (in the field of complex numbers) of the equation $z^n = 1$.

Recall the exponential representation of complex numbers: A complex number can be expressed as $z = re^{i\theta}$, where r is the ***magnitude*** and θ is the ***argument*** of z. Viewing z as a point on the plane, it is the point with polar coordinates (r, θ); that is, the point at distance r from the origin (the complex number $0 = 0 + 0i$) and at angle θ with respect to the x-axis.

Thus complex numbers on the unit circle (the circle or radius 1 centered at the origin) are of the form $z = e^{\theta i}$, and raising such a number to the j-th power yields $z^j = e^{(j\theta)i}$, i.e., another point on the unit circle with argument j times that of z. Therefore the n-th roots of unity are

the n complex numbers of the form $e^{(2\pi j/n)i}$, for $j \in [0..n-1]$: they are the n points on the unit circle equally spaced in angle increments of $2\pi/n$ (the circle divided into n equal parts) starting at angle 0.

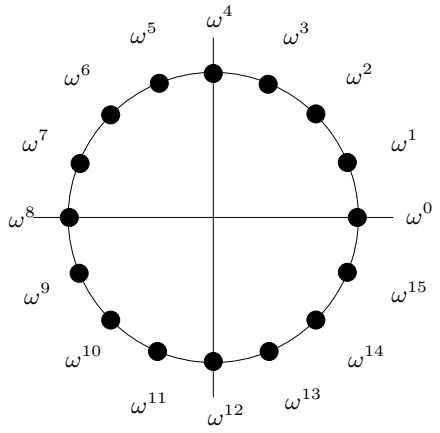


Figure 2.12: The 16-th roots of unity.

Figure 2.12 shows the 16th roots of unity, denoted $\omega^0, \omega^1, \ldots, \omega^{15}$, where $\omega^j = e^{(2\pi j/16)i}$, for $j \in [0..15]$, Note that these in fact form eight $\pm$ pairs, namely the diametrically opposed pairs of points: $\omega^8 = -\omega^0$, $\omega^9 = -\omega^1$, $\omega^{10} = -\omega^2$, and so on. Furthermore, the squares of the 16th roots of unity are the 8th roots of unity: the eight points on the unit circle that are at angles increasing by $2\pi/8$ starting from angle 0; and these also form four $\pm$ pairs. The squares of the 8th roots of unity are the 4th roots of unity (i.e., 1, -1, i, and $-i$); the squares of the 4th roots of unity are the 2nd (square) roots of unity (i.e., 1 and -1); and the squares of both of these are 1. Putting these observations in the context of polynomial evaluation:

- to evaluate a polynomial of degree 15 at the 16-th roots of unity, we would recursively
- evaluate two polynomials of degree 7 at the 8-th roots of unity, for which we would recursively
- evaluate four polynomials of degree 3 at the 4-th roots of unity, for which we would recursively
- evaluate eight polynomials of degree 1 at the square roots of unity, for which we would recursively
- evaluate 16 polynomials of degree 0 at 1.

The last of these is the base of the recursion and can be carried out trivially, since a polynomial of degree 0 is just a constant — so it does not even matter at what point we evaluate it! As promised, it all works perfectly!

We say that a complex number z is a ***primitive n-th root of unity*** if (a) $z^n = 1$ and (b) $z^0, z^1, z^2, \ldots, z^{n-1}$ are all distinct. Intuitively this means that if we take z to every power j, $j \in [0..n-1]$, we "hit" all the n-th roots of unity. Recall that if z is a complex number on the unit circle with argument θ, then z^k the complex number of the unit circle

whose argument is $k\theta$. For example, ω^1, ω^3, and ω^5 are primitive 16-th roots of 1, but ω^2 is not, since $(\omega^2)^0 = (\omega^2)^8$.

We now state some properties of n-th roots of unity that are relevant here. These are straightforward consequences of the definitions and proving them is a good "sanity check" that you have absorbed these definitions (see Exercise 2.17):

Theorem 2.5 *For any positive integer n, let ω be a primitive n-th root of unity (so the n-th roots of unity are ω^j, $j \in [0..n-1]$). Then:*
(a) ω^{-1} is also a primitive n-th root of unity.
(b) If n is even, the n-th roots of unity form $n/2 \pm$ pairs; specifically, for every $j \in [0..n/2-1]$, $\omega^{j+n/2} = -\omega^j$.
(c) If n is even, then ω^2 is a primitive $n/2$-th root of unity.

The FFT is an algorithm that solves the following problem:

Input: The coefficient representation $(d_0, d_1, \ldots, d_{n-1})$ of a polynomial $D(x)$ of degree $n-1$, where n is a power of 2; and a primitive n-th root of unity, ω. (If we are given a polynomial of degree $m < n-1$ we can pad-out the coefficients of degrees between $m+1$ and $n-1$ with 0s.)
Output: $\big(D(\omega^0), D(\omega^1), D(\omega^2), \ldots, D(\omega^{n-1})\big)$; i.e., the value representation of $D(x)$ at the n-th roots of unity.

It may appear strange that we include a primitive n-th root of unity as a parameter, since it is implicit in the length n of the input that $e^{2\pi i/n}$ is a primitive n-root of unity. As we will see shortly, this additional parameter makes it possible to use the same algorithm for interpolation, as well as evaluation.

Below is the algorithm in pseudocode. Recall from (2.17) that D_0 and D_1 are polynomials of degree $(n/2-1)$ whose coefficients are, respectively, the coefficients of the even and odd terms of the degree-n polynomial D that we wish to evaluate at the n-th roots of unity. Also recall that the n-th roots of unity form $n/2 \pm$ pairs as follows: for all $j \in [0..n/2-1]$, $\omega^{j+n/2} = -\omega^j$ (Theorem 2.5(b)).

```
    FFT((d_0, d_1, ..., d_{n-1}), ω)
 1  if n = 1 then return d_0
 2  else
 3      (s_0, s_1, ..., s_{n/2-1}) := FFT((d_0, d_2, d_4, ..., d_{n-2}), ω^2)     ▶ Evaluate D_0 at n/2-th roots of 1
 4      (t_0, t_1, ..., t_{n/2-1}) := FFT((d_1, d_3, d_5, ..., d_{n-1}), ω^2)     ▶ Evaluate D_1 at n/2-th roots of 1
 5      for j := 0 to n/2 - 1 do
 6          r_j := s_j + ω^j t_j          ▶ By (2.17), D(ω^j) = D_0((ω^j)^2) + ω^j D_1((ω^j)^2)
 7          r_{j+n/2} := s_j - ω^j t_j    ▶ and D(-ω^j) = D_0((ω^j)^2) - ω^j D_1((ω^j)^2)
 8  return (r_0, r_1, ..., r_{n-1})
```

Given the preceding discussion (specifically (2.17) and the properties of the n-th roots of unity in Theorem 2.5) the correctness of this algorithm follows by a straightforward induction.

For its time complexity, let $T(n)$ be the number of arithmetic operations (additions, subtractions, and multiplications of complex numers) performed by this algorithm when the given list of coefficients has length n. We have that $T(n) = 2T(n/2) + 2n$, since the algorithm

calls itself recursively twice on lists of coefficients of half the size (lines 3-4) and then performs n additions or subtractions and n multiplications (lines 6-7). By the Master Theorem, $T(n) = O(n \log n)$. So, the running time of this algorithm is $O(n \log n)$. (We assume, as usual, that the numbers involved are small enough to fit in a word of memory, so that each arithmetic operation on complex numbers can be performed in $O(1)$ time.)

We now know how to do the first step (evaluation) of our three-step algorithm for polynomial multiplication in $O(n \log n)$ time, and the second step (computation of the value representation of the product) in $O(n)$ time. But what about the third step, interpolation?

A linear algebra perspective on the matter provides insights that reveal (a) why interpolation is simply the inverse operation of evaluation, and (b) how the FFT can be leveraged to interpolate the coefficient representation from the value representation of a degree $n-1$ polynomial at the n-th roots of unity. Using matrix notation the evaluation of $D(x)$ at n points $v_0, v_1, v_2, \ldots, v_{n-1}$ can be written as follows:

$$\begin{pmatrix} A(v_0) \\ A(v_1) \\ A(v_2) \\ \vdots \\ A(v_{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & v_0 & v_0^2 & \ldots & v_0^{n-1} \\ 1 & v_1 & v_1^2 & \ldots & v_1^{n-1} \\ 1 & v_2 & v_2^2 & \ldots & v_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{n-1} & v_{n-1}^2 & \ldots & v_{n-1}^{n-1} \end{pmatrix}}_{\text{Vandermonde matrix } V} \cdot \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{pmatrix} \tag{2.18}$$

Let V be the $n \times n$ matrix shown above. V has a special form: It is generated by n values, $v_0, v_1, \ldots, v_{n-1}$, and each row is obtained by raising one of these values to the powers $0, 1, 2, \ldots, n-1$. Such matrices are called ***Vandermonde*** matrices and they are invertible if and only if all n values are distinct, as is the case here. Let V^{-1} be the inverse of V.

Informally speaking, (2.18) says "evaluation $= V \cdot$ coefficient representation":

$$V \cdot \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix} = \begin{pmatrix} A(v_0) \\ A(v_1) \\ \vdots \\ A(v_{n-1}) \end{pmatrix}$$

Multiplying both sides by V^{-1} and reversing the terms of the equality we get:

$$V^{-1} \cdot \begin{pmatrix} A(v_0) \\ A(v_1) \\ \vdots \\ A(v_{n-1}) \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix}$$

which, informally speaking, says "interpolation $= V^{-1} \cdot$ value representation". This is what I meant when I said earlier that interpolation is the inverse operation of evaluation.

Things get even better when we consider the Vandermonde matrix generated by the specific evaluation points we have chosen, the n-th roots of unity. Let ω be any primitive n-th root of unity and

$$V_\omega = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

So the value in row j and column k of this matrix, where $j, k \in [0..n-1]$, is ω^{jk}.

Note that multiplying an $n \times n$ matrix by an n-vector in general requires $\Theta(n^2)$ arithmetic operations. But multiplying this specific matrix V_ω by the n-vector $(d_0, d_1, \dots, d_{n-1})$ can be done using only $O(n \log n)$ operations using the FFT: It is exactly the same as evaluating the polynomial of degree $n-1$ whose coefficients are $d_0, d_1, \dots, d_{n-1}$ at the n-th roots of unity.

Now consider the matrix $V_{\omega^{-1}}$:

$$V_{\omega^{-1}} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{pmatrix}$$

Since ω is a primitive root of unity, so is ω^{-1} (Theorem 2.5(a)). Therefore, we can multiply $V_{\omega^{-1}}$ by a vector of length n in $O(n \log n)$ operations using the FFT.[5] This is great news because in a moment we will prove:

$$V_\omega^{-1} = \frac{1}{n} V_{\omega^{-1}} \tag{2.19}$$

That is, interpolation of the polynomial from the n-th roots of unity (which, as we just saw, is tantamount to multiplying V_ω^{-1} by $\big(A(\omega^0), A(\omega^1), \dots, A(\omega^{n-1})\big)^\mathsf{T}$) is the same as multiplying $V_{\omega^{-1}}$ by this vector and then dividing all the entries of the resulting vector by n. But since ω^{-1} is a primitive n-th root of unity we can perform this multiplication in $O(n \log n)$ time using the FFT, and so we can perform interpolation in $O(n \log n)$ time as well!

Proof of (2.19). Let $X = V_\omega \cdot V_{\omega^{-1}}$. The element X_{jk} in row j and column k of X is

$$X_{jk} = \sum_{\ell=0}^{n-1} \omega^{j\ell} \cdot \omega^{-\ell k} = \sum_{\ell=0}^{n-1} \omega^{(j-k)\ell} = \begin{cases} \frac{\omega^{(j-k)n}-1}{\omega^{j-k}-1}, & \text{if } j \neq k \\ n, & \text{if } j = k \end{cases}$$

[5] This is exactly why it is useful to supply the primitive n-th root of unity as a parameter to the FFT.

But since ω is an n-th root of 1, $\omega^n = 1$ and so $\omega^{(j-k)n} - 1 = 0$. Therefore,

$$V_\omega \cdot V_{\omega^{-1}} = \begin{pmatrix} n & 0 & 0 & \dots & 0 \\ 0 & n & 0 & \dots & 0 \\ 0 & 0 & n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & n \end{pmatrix} = n\mathbf{I}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. Multiplying both sides on the left by $\frac{1}{n}V_\omega^{-1}$ we get the desired equality (2.19). □

Now we have all the ingredients needed to multiply two polynomials of degree m in $O(m \log m)$ time:

Input: The coefficient representation $(a_0, a_1, \dots, a_m)$ and $(b_0, b_1, \dots, b_m)$ of two polynomials $A(x)$ and $B(x)$ of degree m. (Alternatively: two $(m+1)$-dimensional vectors).

Output: The coefficient representation $(c_0, c_1, \dots, c_{2m})$ of $C(x) = A(x) \cdot B(x)$. (Alternatively: the convolution of the two given vectors).

Note that the product $C(x)$ has degree $2m$ and so to interpolate we need to evaluate in (at least) $2m+1$ points so that we have enough points to interpolate the value representation of the product and obtain its coefficient representation. To use the FFT we need the number of evaluation points n to be a power of 2. So, we choose n to be the smallest integer power of 2 that is at least $2m+1$, i.e., $n = 2^{\lceil \log_2(2m+1) \rceil}$, and we pad the missing coefficients (for degrees $m+1..n-1$) of A and B with 0s. Note that $n \le 2(2m+1) = O(m)$.

PolyMult$\big((a_0, a_1, \dots, a_m), (b_0, b_1, \dots, b_m)\big)$

1 $n := 2^{\lceil \log_2(2m+1) \rceil}$ ▶ n is the smallest power of 2 that is $\ge 2m+1$
2 $\omega := e^{2\pi i/n}$ ▶ ω is a primitive n-th root of 1
3 **for** $j := m+1$ **to** $n-1$ **do** $a_j := b_j := 0$ ▶ pad-out the coefficients to degree $n-1$
4 $(A_0, A_1, \dots, A_{n-1}) := \text{FFT}\big((a_0, a_1, \dots, a_{n-1}), \omega\big)$ ▶ evaluate $A(x)$ at the n-the roots of 1
5 $(B_0, B_1, \dots, B_{n-1}) := \text{FFT}\big((b_0, b_1, \dots, b_{n-1}), \omega\big)$ ▶ evaluate $B(x)$ at the n-the roots of 1
6 **for** $j := 0$ **to** $n-1$ **do** $C_j := A_j \cdot B_j$ ▶ compute the value representation of the product
7 $(c_0, c_1, \dots, c_{n-1}) := \text{FFT}\big((C_0, C_1, \dots, C_{n-1}), \omega^{-1}\big)$ ▶ iterpolate
8 **return** $(c_0, c_1, \dots, c_{n-1})$ ▶ return the coefficient representation of the product

This algorithm runs in time $O(n \log n) = O(m \log m)$ (since $n = O(m)$): Lines 4, 5, and 7 take $O(n \log n)$. Lines 3 and 6 take $O(n)$ time, and the remaining lines take $O(1)$ time.

Exercises for Chapter 2

2.1 The midterm test has been graded. Each of the k TAs has sorted his or her pile of exams alphabetically. Each pile contains exactly n papers. The TAs have all gone home, and the professor is stuck with merging these k sorted piles into a single pile of kn papers, sorted alphabetically.

(a) Here is one algorithm that the professor can use to solve his problem: Merge the first two sorted piles into a (sorted) pile; then merge the resulting pile with the third sorted pile; then merge the resulting pile with the fourth sorted pile, etc. What is the running time of this algorithm in terms of k and n? (Merging two sorted piles of papers can be done in time proportional to the size of the resulting pile.)

(b) Give a more efficient divide-and-conquer algorithm that the professor can use to solve this problem. What is the running time of your algorithm? (You don't need to justify its correctness.)

2.2 Give an algorithm that takes as input two *sorted* arrays, each containing n integers, and returns the median of the union of the elements in the two arrays. Your algorithm should run in $O(\log n)$ time. You may assume that the elements in the two arrays are distinct.

2.3 Consider the algorithm LOOPYSORT(A, ℓ, r) below, where $A[1..n]$ is an array, and $1 \leq \ell \leq r \leq n$. This is a recursive sorting algorithm that (unlike Mergesort) requires no merging and (unlike Quicksort) requires no partitioning.

```
LOOPYSORT(A, ℓ, r)
 1 if r = ℓ then                          ▷ A[ℓ..r] has length 1
 2     return
 3 elsif r = ℓ + 1 then                   ▷ A[ℓ..r] has length 2
 4     if A[ℓ] > A[r] then swap A[ℓ] and A[r]
 5     return
 6 else                                   ▷ A[ℓ..r] has length at least 3
 7     t := ⌈2(r − ℓ + 1)/3⌉              ▷ t is two-thirds of the length of A[ℓ..r]
 8     LOOPYSORT(A, ℓ, ℓ + t − 1)         ▷ sort the first two-thirds of A[ℓ..r]
 9     LOOPYSORT(A, r − t + 1, r)         ▷ sort the last two-thirds of A[ℓ..r]
10     LOOPYSORT(A, ℓ, ℓ + t − 1)         ▷ re-sort the first two-thirds of A[ℓ..r]
11     return
```

(a) Prove that a call to LOOPYSORT(A, ℓ, r) sorts the subarray $A[\ell..r]$ in increasing order. To simplify matters, you may assume that the length $r - \ell + 1$ of $A[\ell..r]$ is a power of 3.

(b) Use the Master Theorem to determine the running time $T(n)$ of the call LOOPYSORT$(A, 1, n)$.

(c) How does the running time of LOOPYSORT$(A, 1, n)$ compare to the running time of Bubblesort on array $A[1..n]$? Justify your answer.

2.4 Let $A[1..n]$ be an array about which the following are true:

(i) The positions of A contain some number k of distinct elements.

(ii) One of these k elements appears in exactly one position of A; we will call this unique position ***singleton***.

(iii) Each of the remaining $k - 1$ elements appears in exactly ***two consecutive positions*** of A; we will call a pair of consecutive positions containing the same element a ***couple***.

We want to design an algorithm that, given an array A that satisfies the above properties, returns the singleton (i.e., the position of the unique element).

There is a straightforward $\Theta(n)$ algorithm that does this. Give an asymptotically faster divide-and-conquer algorithm that solves this problem.[6] Justify the correctness of your algorithm, and analyze its running time using the Master Theorem.

2.5 Karatsuba's integer multiplication algorithm takes as inputs the binary representations x and y of two non-negative integers and computes the binary representation of their product.

(a) Following is an algorithm (with line 3 left incomplete), which takes as input a positive integer n that is a power of two, and computes the binary representation of the number ten to nth power (i.e., the number that is written in decimal as a 1 followed by n 0s).

```
PowerOfTenToBinary(n)
1 if n = 1 then return "1010" (the binary string that represents the number ten)
2 else
3     x := ???
4     return FastMult(x, x)
```

Complete the missing expression in line 3, explain why this algorithm works, and analyze its running time.

(b) Following is an algorithm (with line 6 left incomplete), which takes as input the *decimal* representation $x[1..n]$ of a non-negative integer X, where n is a power of 2, and computes the binary representation of X. Assume that $x[1..\textbf{length}(x)]$ is a string of digits $0, 1, \ldots, 9$, where $x[1]$ is the most significant and $x[\textbf{length}(x)]$ is the least significant digit of the input, and that you have defined an array $\textbf{binary}[d]$, which, for any $d \in [0..9]$, gives the binary representation of d; i.e., $\textbf{binary}[0] = 0$, $\textbf{binary}[1] = 1$, $\textbf{binary}[2] = 10$, up to $\textbf{binary}[9] = 1001$.

```
DecimalToBinary(x)
1 if length(x) = 1 then return binary[x]
2 else
3     n := length(x)
4     x_L := x[1..n/2]
5     x_R := x[n/2 + 1..n]
6     return ???
```

Complete the missing expression in line 6, explain why this algorithm works, and analyze its running time. In constructing the missing expression, you may assume that, in addition to the algorithms PowerOfTenToBinary and FastMult, you are given a function $\textsc{Sum}(a, b)$, which takes as input the binary representations a and b of two non-negative integers and returns the binary representation of their sum in $O\Big(\max\big(\textbf{length}(a), \textbf{length}(b)\big)\Big)$ time.

[6] An algorithm with running time $\Theta\big(f(n)\big)$ is ***asymptotically faster*** than an algorithm with running time $\Theta\big(g(n)\big)$ if $\lim_{n\to\infty} f(n)/g(n) = 0$; this is denoted $f(n) = o\big(g(n)\big)$ or $f(n) \in o\big(g(n)\big)$, and is read as "$f(n)$ is little-oh of $g(n)$". Thus, an algorithm that runs in $\Theta(\log n)$ time is asymptotically faster than one that runs in $\Theta(\sqrt{n})$ time, which is asymptotically faster than one that runs in $\Theta(n)$ time, which is asymptotically faster than one that runs in $\Theta(n \log n)$ time.

2.6 Describe a version of Karatsuba's integer multiplication algorithm where instead of dividing the binary representation of each integer into two segments, you divide it into three segments. Justify the correctness of your algorithm, analyze its running time, and compare it to the running time of the version we saw in class. To receive credit, your algorithm should run in sub-quadratic time in the number of bits of the integers being multiplied.

2.7 We are given an array $A[1..n]$ of distinct integers so that each element is at most k positions away from its correct position in sorted order. That is, for every $i \in [1..n]$, the i-th smallest element of A is in $A[j]$, where $j \geq \max(1, i-k)$ and $j \leq \min(i+k, n)$. Describe a divide-and-conquer algorithm that sorts such an array in $O(n \log k)$ time. Justify the correctness of your algorithm and analyze its running time. For simplicity, assume that both n and k are powers of 2.

2.8 Suppose we have a two-dimensional array $A[1..n, 1..n]$ of numbers, every row and column of which is sorted. That is, $A[i, j] \leq A[i, j']$, for all $1 \leq i \leq n$ and all $1 \leq j \leq j' \leq n$; and $A[i, j] \leq A[i', j]$, for all $1 \leq j \leq n$ and all $1 \leq i \leq i' \leq n$. We want to determine whether a given number x is in A.

One way to do this is to perform binary search in each row, until we either find x is some row, or determine that x is not in any row. (Of course, we could equally well do binary search in each column instead.) This clearly takes $\Theta(n \log n)$ time, but exploits only the fact that rows are sorted, ignoring that the columns are also sorted.

In an attempt to devise an algorithm that exploits the order in both dimensions, we are led to the following divide-and-conquer idea: Divide up the square array into four roughly equal two-dimensional arrays; upper-left, upper-right, lower-right, and lower-left. Look at the "middle" element of the array (roughly the element $A[n/2, n/2]$ with appropriate rounding). Based on comparing this element with x, eliminate a part of the array where we can be certain that x cannot be, and search the remaining parts.

Describe in pseudocode a divide-and-conquer algorithm to search A based on the idea sketched in the preceding paragraph, justify its correctness, analyze its running time, and compare the running time of your algorithm to the running time of the algorithm that uses binary search for each row. You may assume that n is a power of two. As in the analysis of the first algorithm, express the running time as a function of n, the number of rows and columns of the input A. (Note that the *size* of the input, assuming $O(1)$-space for each entry of A, is n^2, not n.)

Note: There is a better algorithm than either of the above two. Can you find one?

2.9 You are given an array $A[1..n, 1..n]$ of *distinct* integers, where n is a positive integer; for simplicity assume that n is a power of 2. A ***local minimum*** of this array is an element that is smaller than all its neighbours, where the neighbours of element $A[i, j]$ are those of the elements above it, below it, to its left, and to its right that exist; in other words, those of elements $A[i-1, j]$, $A[i+1, j]$, $A[i, j-1]$, and $A[i, j+1]$ that have indices in the range $[1..n]$. (Thus, if $n = 1$, the unique element of A has no neighbours and is therefore trivially a local minimum; and if $n > 1$, the four "corners" of the array have two neighbours each, the elements

in rows 1 and n and in columns 1 and n other than the corner elements have three neighbours each, and all other elements have four neighbours each.) The minimum element of the array is obviously a local minimum, but the array may have other elements that are local minima.

Give a divide-and-conquer $O(n)$ algorithm to find a local minimum in A. Justify the correctness of your algorithm and analyze its running time. Note that the size of your input is $\Theta(n^2)$ so you don't have enough time to look at all elements of the array. Finding the minimum element of A requires $\Omega(n^2)$ time, but a *local* minimum can be found faster.

2.10 Describe an algorithm that takes as input an array $A[1..n]$ of (not necessarily distinct) integers. Your algorithm returns an integer x that appears in more than $n/5$ positions of A (i.e., $|\{i : 1 \le i \le n \text{ and } A[i] = x\}| > n/5$), if such an x exists; and returns the special value NIL if no such x exists. Note that it is possible for several such elements to exist, in which case we can return any one of them. Your algorithm should run in $O(n)$ time in the worst case. (It is trivial to solve the problem in $O(n \log n)$ time by sorting A.)

Explain why your algorithm is correct, and analyze its running time. Note that the integers in A may be enormous, and so you cannot use a direct-address table.

2.11 Let $A[1..n]$ be an array containing (possibly negative) integers, where n is a power of 2. For any i, j such that $1 \le i \le j \le n$, we define the ***value*** of subarray $A[i..j]$, denoted $\mathbf{val}(A[i..j])$, to the sum of all the integers of the subarray, i.e., $\mathbf{val}(A[i..j]) = \sum_{t=i}^{j} A[t]$. We wish to find the maximum value of any subarray of A, i.e., $\max_{i,j \text{ s.t. } 1 \le i \le j \le n} \mathbf{val}(A[i..j])$.

This can be easily done in $\Theta(n^2)$ time. Give a divide-and-conquer algorithm that solves this problem and is asymptotically faster than $\Theta(n^2)$. For full marks your algorithm should run in $\Theta(n)$ time, but an $\Theta(n \log n)$ algorithm will get you some credit. Explain why your algorithm is correct, and analyze its running time.

2.12 An ***interval*** is a pair (ℓ, r) of numbers such that $\ell \le r$, or the ***empty interval*** denoted $\varnothing$. The ***intersection*** of two intervals (ℓ, r) and (ℓ', r') is the interval $\big(\max(\ell, \ell'), \min(r, r')\big)$ if $\max(\ell, \ell') \le \min(r, r')$; otherwise, it is $\varnothing$. The ***length*** of interval (ℓ, r) is $r - \ell$, and the length of $\varnothing$ is 0.

We are given a list of $n \ge 2$ nonempty intervals $(\ell_1, r_1), (\ell_2, r_2), \ldots, (\ell_n, r_n)$. We want to find two distinct intervals in this list whose intersection is as long as possible. It is straightforward to do this in $\Theta(n^2)$ time. Describe a divide-and-conquer algorithm that solves this problem in $O(n \log n)$ time. Explain why your algorithm is correct and analyze its running time.

2.13 Let S be a set of at least two numbers, and let

$$\mathbf{ads}(S) = \frac{\max(S) - \min(S)}{|S| - 1}.$$

Intuitively, this is the average distance between pairs of successive elements of S in sorted order: there are $|S| - 1$ such pairs, and the distances between all of them add up to $\max(S) - \min(S)$. We say that $x, y \in S$ are ***close in*** S if $x \ne y$ and $|x - y| \le \mathbf{ads}(S)$ — i.e., the distance between x and y is at most the average distance between successive elements of S. Note that

any set of at least two numbers must contain a pair of close numbers: this is for the same reason that it is impossible for all of you to receive an above-average grade! Also note that a pair of numbers that are close in S need not be successive in sorted order; nor is every pair of successive numbers necessarily close in S.

Describe a divide-and-conquer algorithm that, given a set S of $n \geq 2$ numbers in arbitrary order, finds a pair of numbers in S that are close in S; your algorithm should run in $O(n)$ time. Justify the correctness and running time of your algorithm.

2.14 Explain how to remove the assumption, made for the algorithm to find the closest pair of points (Section 2.5), that the points in P have distinct x-coordinates without affecting the asymptotic running time of the algorithm.

2.15 In D-SELECT we partitioned the elements of the given array into groups of five elements each. Would the algorithm still run in linear time if we instead partitioned them into groups of seven elements each? What about groups of three elements each?

2.16 Show how to evaluate a polynomial of degree n at *one* point using n multiplications and n additions. (The challenge here is to get this exact number of operations, not more loosely $O(n)$ multiplications and additions.)

2.17 Prove Theorem 2.5.

2.18 Demonstrate the polynomial multiplication algorithm that uses the FFT by using it to multiply the polynomials $1+x+2x^2$ and $2+3x$: Use the FFT to evaluate each polynomial at an appropriate set of points, and show the resulting vectors; multiply the vectors component-wise; and use the inverse FFT to obtain the coefficients of the final result. Show your work.

2.19 In this problem we show that it is possible to get an algorithm for integer multiplication that is asymptotically faster than Karatsuba's by recursively multiplying $n/3$-bit numbers. To do this, we use ideas about fast multiplication of polynomials given their coefficients, as discussed in Section 2.7. In particular, we want to multiply two degree-2 polynomials in five (rather than the obvious nine) multiplications. Because here we'll be working with polynomials of fixed degree, we do not have to consider complex numbers as in the FFT.

(a) We first consider *polynomial evaluation*. Our input consists of the three coefficients of a degree-2 polynomial $p(x) = a_2x^2 + a_1x + a_0$, and we want to find the value of this polynomial at five integers $x = 0, 1, 2, 3, 4$.
Give a 3×5 matrix M_1 such that for every row vector $\vec{a} = (a_0, a_1, a_2)$,

$$\vec{a} \cdot M_1 = \vec{v}$$

where $\vec{v} = \big(p(0), p(1), p(2), p(3), p(4)\big)$ — i.e., the row vector consisting of the five values of $p(x)$ at $x = 0, 1, 2, 3, 4$.

(b) We next consider *polynomial interpolation*. Our input consists of the values of a degree-4 polynomial $p(x)$ at $x = 0, 1, 2, 3, 4$, and we want to find the coefficients of $p(x)$. That

is, our input is a row vector $\vec{v} = (v_0, v_1, v_2, v_3, v_4)$, and we wish to find the coefficients $\vec{a} = (a_0, a_1, a_2, a_3, a_4)$ of a polynomial $p(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ such that $(v_0, v_1, v_2, v_3, v_4) = \big(p(0), p(1), p(2), p(3), p(4)\big)$.

Give a 5×5 matrix M_2 of rational numbers such that for every $\vec{v} = (v_0, v_1, v_2, v_3, v_4)$, if we compute $\vec{a} = \vec{v} \cdot M_2$ and let $(a_0, a_1, a_2, a_3, a_4) = \vec{a}$ and $p(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$, then $(v_0, v_1, v_2, v_3, v_4) = \big(p(0), p(1), p(2), p(3), p(4)\big)$.

Hint. Consider part (a) and find the inverse of an appropriate matrix. You may use software to help you with your calculations, but be sure to credit your source.

(c) We now consider a divide-and-conquer algorithm for integer multiplication, where we multiply two n-bit numbers by recursively multiplying *five* $n/3$-bit numbers.

Suppose we are given two n-bit numbers A and B; for convenience, assume that n is a power of 3. Let a_0, a_1, a_2 be the integers comprising the rightmost third, middle third, and leftmost third of A; let b_0, b_1, b_2 be defined similarly for B. We then have

$$A = a_2(2^{n/3})^2 + a_1 2^{n/3} + a_0 \text{ and } B = b_2(2^{n/3})^2 + b_1 2^{n/3} + b_0.$$

Our goal is to compute

$$C = A \cdot B = c_4(2^{n/3})^4 + c_3(2^{n/3})^3 + c_2(2^{n/3})^2 + c_1 2^{n/3} + c_0$$

where $(c_0, c_1, c_2, c_3, c_4)$ are the coefficients of the degree-4 polynomial obtained by multiplying the two degree-2 polynomials $p(x) = a_2x^2 + a_1x + a_0$ and $q(x) = b_2x^2 + b_1x + b_0$.

Using the matrices M_1 and M_2 from parts (a) and (b), give the details of this algorithm. Give a recurrence that describes its time complexity, and explain why it does. Finally, explain why this algorithm is (asymptotically) faster than Karatsuba's algorithm discussed in class.

Chapter 3

Dynamic programming algorithms

Dynamic programming is a powerful and widely applicable algorithm design technique. The meaning of the word "programming" in this context is unrelated to its meaning when used as in "writing computer code"; rather, it is related to the meaning of the word when used as in "radio programming": i.e., scheduling or planning a series of activities.

Rather than try to describe dynamic programming in the abstract, it is best to first look at an example.

3.1 Longest increasing subsequence

Let A be a sequence of integers. A is ***increasing*** if every element other than the first is greater than its predecessor. A ***subsequence*** of A is a sequence consisting of elements of A, taken in the order in which they appear in A. Note that the elements of a subsequence of A need not be consecutive in A. For example, if $A = 5, 7, 6, 8, 10, 2, 3, 6, 4, 17, 1, 5, 9, 19, 16$, then $7, 6, 2, 17, 1, 19, 16$ is a subsequence of A. $10, 17, 19$ is an increasing subsequence of A. $5, 6, 8, 10, 17, 19$ is a ***longest increasing subsequence (LIS)*** of A: there is no increasing subsequence of A of length more than six. Our problem is:

Input: A sequence $A[1..n]$ of integers (we use array notation for the sequence).
Output: A LIS of A.

Note that a LIS is not necessarily unique; for instance in our example $2, 3, 6, 9, 17, 19$ is another increasing subsequence of length six.

A brute-force approach to solving this problem, i.e., considering all subsequences of A and returning a longest one, is hopelessly inefficient: A sequence of length n has 2^n subsequences.

To approach this problem using the methodology of dynamic programming, we try to express a *supposed* solution to the given instance of the problem as an *extension of the solution to a smaller instance of the problem.* So, suppose we have found a LIS of A, say S. S ends in some position of A, say position i; that is,

$$S = S'A_i \tag{3.1}$$

where S' is a

(a) longest
(b) increasing
(c) subsequence of A that ends in position j, for some $j < i$, such that $A[j] < A[i]$.

It is obvious that S' must satisfy (b) and (c): otherwise S would not be an increasing subsequence of A. But why must it be a *longest* subsequence of A with properties (b) and (c)? The argument is again by contradiction, and you probably see exactly what it is, but indulge my going over it in some detail. Suppose, for contradiction, that S'' is a subsequence of A that satisfies (b) and (c) and is *longer than* S'. But then $S''A[i]$ is an increasing subsequence of A that is longer than $S'A[i] = S$, which contradicts that S is a LIS of A.

This is an example of a ***cut-and-paste argument***. In our case, we "cut" S' and "paste" S'' in its place, thereby obtaining a solution that is better than a supposedly optimal one. This type of argument is used often in proving the correctness of dynamic programming algorithms. Most of the time it is as simple as in the LIS problem, and in those cases we will not bother to give it in more detail — we will simply say something like "by a cut-and-paste argument . . .". In some cases, however, the argument merits more than a passing reference (e.g., see Section 3.4).

We can summarize the above discussion in the following restatement of (3.1):

$$\underbrace{S}_{\substack{\text{LIS of } A \text{ ending} \\ \text{at position } i}} = \underbrace{S'}_{\substack{\text{LIS of } A \text{ ending} \\ \text{at some position } j < i \\ \text{s.t. } A[j] < A[i]}} A[i]. \tag{3.2}$$

This formulation shows that LIS exhibits the ***optimal substructure property***: *every optimal solution to an instance of the problem extends an optimal solution to a smaller instance of the problem.*

Digression This may appear almost like a tautology: it sounds like a property that *any* optimization problem should have, but this is not true. Recall the shortest-paths problem you studied when you learned about breadth-first search: Given a (directed or undirected) graph and two nodes s and u, find an $s \rightsquigarrow u$ path with the fewest edges. This problem clearly has the optimal substructure property: A shortest $s \rightsquigarrow u$ path p is an extension of a shortest path from s to the predecessor of u on p. This is immediate by a straightforward cut-and-paste argument. On the other hand, consider the ***longest path*** problem: Find a longest $s \rightsquigarrow u$ path that does not contain a cycle; such a path is called ***simple***. This problem does not exhibit the optimal substructure property. To see this consider the directed graph in Figure 3.1.

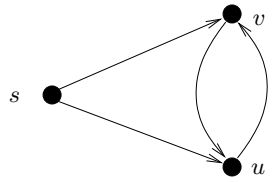


Figure 3.1: The longest path problem does not have the optimal substructure property

The longest simple $s \rightsquigarrow u$ path is s, v, u. But its prefix s, v is *not* a longest simple $s \rightsquigarrow v$ path. So a longest simple $s \rightsquigarrow u$ path p is not necessarily an extension of a longest simple path from s to the predecessor of u on p. Do you see why a cut-and-paste argument does not work here? **End of Digression**

Now that we have uncovered the optimal substructure of LIS, we are well on our way to constructing a dynamic programming algorithm to solve the problem. For now let us focus on the problem of computing the *length* of a LIS of the given sequence. Later we will see how to "retrofit" the algorithm that computes the length of a LIS to also compute an actual LIS. We define, for each $i \in [1..n]$,

$$L(i) = \text{the length of a LIS of } A \textit{ that ends in position } i. \tag{*}$$

Note that the quantity we actually want to compute is not quite (*): What we want is the length of a LIS of A; the constraint in italics, about the position where an LIS ends, is something that we added. We did so guided by the optimal substructure property we discovered in (3.2), which gives the optimal LIS that ends in some position in terms of an optimal LIS that ends in some earlier position. Nevertheless, from (*) we can easily compute the quantity we want; it is simply $\max\{L(i)\colon i \in [1..n]\}$.

In view of (3.2) we have that $L(i) = 1 + L(j)$, for *some* $j < i$ such that $A[j] < A[i]$. But *which* j? We don't know in advance, so we try out all possibilities and pick the best; since we want to maximize $L(i)$ we pick a $j < i$ such that $A[j] < A[i]$ that has maximum $L(j)$. In other words, the following recursive formula computes $L(i)$ as defined in (*):

$$L(i) = \begin{cases} 1, & \text{if } \forall\, j \in [1..i-1],\ A[j] \geq A[i] \\ 1 + \max\bigl\{L(j)\colon j \in [1..i-1] \text{ and } A[j] < A[i]\bigr\}, & \text{otherwise} \end{cases} \tag{†}$$

Note that (†) is *not* a re-definition of $L(i)$; rather, it is a recursive formula to compute the quantity defined in (*).[1] The proof that this is a correct formula to compute $L(i)$ is just the argument that led us to the optimal substructure of LIS as described by (3.2). To spell it out, there are two cases:

- For all $j \in [1..i-1]$, $A[j] \geq A[i]$. Then the longest increasing subsequence of A that ends in position i consists of $A[i]$ alone, and so $L(i) = 1$.
- For some $j \in [1..i-1]$, $A[j] < A[i]$. Then a longest increasing subsequence of A that ends in position i is $S = S'A[i]$, where S' is a longest increasing subsequence of A that ends in position j for some $j < i$ such that $A[j] < A[i]$, and (by a cut-and-paste argument) has maximum length among all such subsequences. Then $L(i) = 1 + \max\bigl\{L(j)\colon j \in [1..i-1] \text{ and } A[j] < A[i]\bigr\}$.

[1] Technically, (†) is really defining another function, and we should be using a different notation to denote it, say L'. Then it turns out that the two functions, L and L', are actually equal! This is far too formalistic and provides no useful insight. So we will stick with the more lightweight approach: we will simply say that "(†) is a recursive formula to compute (*)" and argue why this is the case, instead of defining two functions and proving that they are equal.

Following is pseudocode for the algorithm to compute the length of a LIS. For now, ignore the instruction in shaded boxes that compute **pre**(i), which we have not defined yet. The algorithm computes $L(i)$ according to (†) (lines 3-5), and keeps track of the position m where the LIS found so far ends (line 6).

```
   LIS(A[1..n])
 1 m := 1    ▶ index where the LIS found so far ends
 2 for i := 1 to n do
 3     L(i) := 1; pre[i] := 0
 4     for j := 1 to i − 1 do
 5         if A[j] < A[i] and L(i) < 1 + L(j) then L(i) := 1 + L(j); pre(i) := j
 6     if L(i) > L(m) then m := i    ▶ found index where a longer LIS ends
 7 return L(m)
```

The correctness of the algorithm is straightforward, given (†) (see Exercise 3.1). It is also easy to see that its running time is $\Theta(n^2)$. Given that (†) is a recursive formula, you may find it surprising that the algorithm to compute $L(i)$ is written iteratively rather than recursively. We will discuss why, and how to code it recursively without exploding its running time, in Section 3.6.

To retrofit this algorithm to compute an actual LIS of A, rather than just its length, we compute, alongside $L(i)$, the index **pre**(i) of the predecessor of $A[i]$ in a LIS of A that ends in position i (letting **pre**$(i) = 0$ — an invalid index in A — if no such predecessor exists). Then, we can recover a LIS of A (in reverse) as $A[m], A[\textbf{pre}(m)], A\big[\textbf{pre}[\textbf{pre}()]\big], \ldots$, until we get to an index i for which **pre**$(i) = 0$. So we can compute a LIS by replacing line 7 by the following code fragment, where the operator "$\circ$" denotes concatenation of lists:

```
i := m; S := empty list
while i ≠ 0 do S := A[i] ∘ S; i := pre(i)
return S
```

An alternative way to retrofit the algorithm is to use directly the values of L to find the elements of A that belong to a LIS of A, without computing **pre**. To do this, simply replace the statement "**return** $L(m)$" by the following code (see Exercise 3.2):

```
i := m; S := empty list
while i ≠ 0 do
    S := A[i] ∘ S
    j := i − 1
    while j ≥ 1 and (L(j) ≠ L(i) − 1 or A[j] ≥ A[i]) do j := j − 1
    i := j
return S
```

Stepping back

We now review the process through which we arrived at our algorithm for the LIS problem and generalize it so that you can apply a similar process when devising dynamic programming algorithms for other problems.

(1) We analyze the problem at hand with an eye to discovering its "optimal substructure": how an optimal solution can be constructed by extending *optimal* solutions to smaller instances of the problem. In the case of LIS, this analysis led us to (3.2).
(2) The analysis resulting in the discovery of the optimal substructure of our problem may lead us to introduce additional constraints or parameters to the problem. For example, in the case of the LIS problem we want to find the LIS of a sequence A; our analysis led us to define the problem of finding an LIS of A *that ends at a particular position* of A. Thus, the problem we end up solving recursively is not quite the original problem we are trying to solve but a slightly modified one.
(3) Often (but not always) dynamic programming is used to solve optimization problems. In this case, we first focus on computing the *value* of the optimal object we wish to find, rather than constructing the optimal object itself. In the case of LIS we focused on computing the *length* of a LIS, not the LIS itself.
(4) We define *clearly* (and I cannot overemphasize the need for this) the subproblems we want to solve. In the case of LIS this was (*): the definition of $L(i)$ as the length of an LIS of A that ends in position i.
(5) We make sure that the subproblems we defined in (4) suffice to efficiently compute the problem we actually want to solve. In the case of LIS this was indeed the case: once we had $L(i)$ for all $i \in [1..n]$, the length of an LIS is simply $\max_{i\in[1..n]} L(i)$.
(6) Based on the optimal substructure discovered in (1), we give a *recursive formula* to compute the subproblems defined in (4). In the case if LIS, this was the formula (†). This recursive formula is known as a ***Bellman equation***.[2]
(7) We write an algorithm that computes the values of the subproblems defined in (4) using the recursive formula in (6). This can be done recursively (but with some care to avoid exponential running times, as we will explain in Section 3.6) or iteratively as we did in the case of LIS. The correctness of this algorithm is usually straightforward in view of the correctness of the recursive formula in (6) to compute the subproblems defined in (4).
(8) We retrofit the algorithm in (7), which computes the *value* of the optimal object we seek, so that it computes the optimal object itself. Typically this is done in one of two ways:
    - Compute some additional information alongside the value of the subproblems defined in (4), and then use that information to compute the optimal object. In the case of LIS, we computed $\mathbf{pre}(i)$, in addition to $L(i)$, for $i \in [1..n]$.
    - Use the values of the subprolems defined in (4) directly, without any additional information, to construct the optimal object. In the case of LIS we did this by computing a LIS directly from L.

Comparison to the greedy and divide-and-conquer techniques

We now highlight the similarities and differences between dynamic programming and the algorithm design techniques we explored in the previous two chapters.

[2] After Richard Bellman, the mathematician who first identified (and named) dynamic programming as an algorithmic technique; we will encounter him again in the next chapter (Section 4.3).

Greedy: Dynamic programming is similar to the greedy method in that both apply to problems that exhibit the optimal substructure property: an optimal solution to a problem extends optimal solutions to subproblems. In greedy algorithms, however, we extend *one* partial optimal solution constructed so far to *a single* more complete optimal solution. For example, in the interval scheduling problems we extend the current "promising set", which is an optimal solution to a subset of the given set of intervals, to a larger "promising set". In contrast, in dynamic programming we construct optimal solutions to multiple subproblems and pick the best among these to extend. For example, to find a LIS the ends in position i, we had to consider the LIS that end in all positions $j < i$ that contain integers less than $A[i]$.

Divide-and-conquer: Both divide-and-conquer and dynamic programming solve an instance of the given problem by breaking it up into smaller instances. Despite this similarity, they differ in several ways:

- In divide-and-conquer the subproblems we solve are smaller versions of the same problem we are after: In Merge Sort we want to sort a big array and the subproblems are to sort smaller arrays; in Karatsuba's algorithm we want to multiply long integers and the subproblems are to multiply shorter integers. In dynamic programming we often have to modify the problem we want to solve by adding constraints or parameters. For example, in the LIS problem we added the constraint of the LIS ending in a particular position.
- In divide-and-conquer the size of the subproblems is (typically) smaller by a constant *factor*.[3] In contrast, in dynamic programming the subproblems can be smaller by a constant *amount*. For example, in Merge Sort we solve a problem of size n by solving two subproblems of size $n/2$. In the LIS algorithm, to solve a problem of size n we solve subproblems of many sizes, from $n-1$ down to 1. The fact that input size decreases geometrically in divide-and-conquer algorithms implies that the depth of the recursion is *logarithmic* in the input size and this is a source of their efficiency. In dynamic programming, the depth of the recursion (for the recursive formula used to compute the values of the subproblems — see item (6) above) is typically *linear* in the input size.
- In a divide-and-conquer algorithm each subproblem solved is used in the solution of just one larger problem. In contrast, in dynamic programming each subproblem solved may be used in the solution of multiple larger problems. For example, in Merge Sort the subproblem of sorting the subarray $A[1..n/4]$ is used only once, namely to sort the subarray $A[1..n/2]$. In contrast, in the LIS algorithm the subproblem of a LIS that ends at position j is used in all subproblems of LIS that end at positions $i > j$ with $A[i] > A[j]$. This difference is sometimes expressed by saying that in dynamic programming the subproblems overlap, while in divide-and-conquer they are independent. This is usually, but not always, the case; a counterexample is the divide-and-conquer D-Select algorithm in Section 2.6.2, where the subproblems partially overlap.

[3]There are exceptions. For example, in the randomized selection algorithm we discussed in Section 2.6, the size of the subproblems depends on the (random) choice of the splitter.

3.2 Weighted-interval scheduling

In Section 1.1 we saw a greedy algorithm to solve the interval scheduling problem: Given a set of jobs (specified as intervals in time), find a maximum-cardinality set of non-conflcting jobs (non-intersecting intervals) contained in that set. Consider a generalization of this problem where each interval has a ***weight***, i.e., a non-negative number that represents the interval's "importance". We now want to find a set of non-conflicting intervals that has maximum *weight*, where the weight of a set of jobs is the *sum* of the weights of the jobs in it: $\mathbf{wt}(A) = \sum_{j \in A} \mathbf{wt}(j)$. More precisely the ***weighted-interval scheduling*** problem is as follows.

Input: A finite set of jobs $\mathcal{J}$, along with the start time $s(j)$, finish time $f(j)$, and weight $\mathbf{wt}(j)$ of each job $j \in \mathcal{J}$, where $s(j), f(j), \mathbf{wt}(j) \in \mathbb{R}$, $s(j) < f(j)$, and $\mathbf{wt}(j) \geq 0$.

Output: A subset A of $\mathcal{J}$ that contains no conflicting jobs and has maximum weight. That is, a set $A \subseteq \mathcal{J}$ such that

- (feasibility) for all $j \neq j' \in A$, j does not conflict with j', and
- (optimality) if $B \subseteq \mathcal{J}$ has the property that for all $j \neq j' \in B$, j does not conflict with j', then $\mathbf{wt}(A) \geq \mathbf{wt}(B)$.

The problem we studied in Section 1.1 is the special case of this problem where all jobs have the same weight.

The weighted-interval scheduling problem can be solved using dynamic programming. Consider an instance of the problem, and let the set of jobs be $[1..n]$; without loss of generality, assume that $f(1) \leq f(2) \leq f(3) \leq \cdots \leq f(n)$, i.e., the jobs are sorted by finish time. Let $p(i)$ be the maximum job before i that does *not* conflict with i; or 0, if all jobs before i conflict with it. That is,

$$p(i) = \begin{cases} 0, & \text{if } \forall j < i, f(j) > s(i) \\ \max\{j\colon j < i \text{ and } f(j) \leq s(i)\}, & \text{otherwise} \end{cases}$$

Imagine we are given an optimal set A for jobs $[1..n]$. We will analyze A with an eye to discovering the optimal substructure of the problem. There are two possibilities:

- $n \notin A$. Then clearly A is an optimal set for jobs $[1..n-1]$.
- $n \in A$. Then $A = A' \cup \{n\}$, where A' is optimal for jobs $[1..p(n)]$. (This can be shown by an easy cut-and-paste argument, which will be presented, in more general form, shortly.)

But which of these two possibilities results in a better set of jobs for $[1..n]$? We don't know in advance, so we will compute the total weight of the jobs in each case and we will pick the one that has greater weight![4]

So we have expressed an optimal solution to our problem in terms of optimal solutions to two smaller problems: finding an optimal set for jobs $[1..n-1]$ and finding an optimal set for jobs $[1..p(n)]$. And how do we solve these subproblems? Recursively, of course. We have discovered the optimal substructure of our problem!

[4]Recall the difference between greedy and dynamic programming algorithms that we noted earlier: In dynamic programming, to find the optimal solution we need to consider optimal solutions to *multiple* subproblems; in the greedy method we keep extending one optimal solution.

As usual, we will first focus on computing the *value* of the subproblems. For each $i \in [0..n]$ define

$$W(i) = \text{the maximum weight of a feasible subset of jobs } [1..i] \qquad (*)$$

(When $i = 0$, this is the weight of an empty set, which we will take to be zero.) Note that if we have computed (*), the quantity we are after, i.e., the value of an optimal set for *all* jobs, is simply $W(n)$. So, the subproblems we have decided to solve suffice for finding a solution to the problem we actually want to solve.

From the analysis that led us to the optimal substructure of the problem, we conclude that the following recursive formula computes $W(i)$:

$$W(i) = \begin{cases} 0, & \text{if } i = 0 \\ \max\Big(\underbrace{W(i-1)}_{\substack{\text{case (a):}\\ \text{optimal set}\\ \text{for jobs } 1..i\\ \text{does not contain } i}}, \underbrace{W\big(p(i)\big) + \mathbf{wt}(i)}_{\substack{\text{case (b):}\\ \text{optimal set}\\ \text{for jobs } 1..i\\ \text{contains } i}}\Big), & \text{otherwise} \end{cases} \qquad (\dagger)$$

We can justify the correctness of this formula as follows. If $i = 0$, the set of jobs is empty and so its weight is (by definition) 0. If $i > 0$, let A be an optimal set for jobs $[1..i]$; there are two possibilities:

(a) $i \notin A$. Then A is optimal for jobs $[1..i-1]$: If a feasible set with greater weight existed for $[1..i-1]$, it would also be feasible for $[1..i]$, contradicting the optimality of A for $[1..i]$. So, in this case, $W(i) = W(i-1)$.

(b) $i \in A$. Then $A = A' \cup \{i\}$, where A' is optimal for $[1..p(i)]$: If a feasible set, say A'', with greater weight than A' existed for $[1..p(i)]$, then $A'' \cup \{i\}$ would be a feasible set for $[1..i]$ (since i does not conflict with any job in $[1..p(i)]$, by definition of $p(i)$) and its weight would be greater than that of $A' \cup \{i\} = A$, contradicting the optimality of A for $[1..i]$. So, in this case, $W(i) = W\big(p(i)\big) + \mathbf{wt}(i)$.

By (a) and (b), $W(i) = \max\Big(W(i-1), W\big(p(i)\big) + \mathbf{wt}(i)\Big)$.

Following is an algorithm to compute the weight of an optimal set of jobs in $\mathcal{J} = \{1, 2, \ldots, n\}$ with start time $s(i)$, finish time $f(i)$, and weight $\mathbf{wt}(i)$, for each $i \in \mathcal{J}$.

WeightedIntervalScheduling$(\mathcal{J}, s, f, \mathbf{wt})$
1 sort the jobs in $\mathcal{J}$ by finish time, so that $f(1) \le f(2) \le \cdots \le f(n)$
2 **for** $i := 1$ **to** n **do** $p(i) := \max\{j\colon j < i \text{ and } f(j) \le s(i)\}$ (or 0 if $\forall j < i,\ f(j) > s(i)$)
3 $W(0) := 0$
4 **for** $i := 1$ **to** n **do** $W(i) := \max\big(W(i-1), W\big(p(i)\big) + \mathbf{wt}(i)\big)$
5 **return** $W(n)$

The correctness of the algorithm follows easily from (†). For its running time, line 1 can be done in $O(n \log n)$ time. Taking advantage of the fact that the jobs are now sorted by finish time, line 2 can also be done in $O(n \log n)$ time (how?). Line 4 takes $O(n)$ time. So the algorithm takes $O(n \log n)$ time.

We can retrofit the algorithm to compute an optimal set of jobs (as opposed to its weight) by replacing line 5 with the following code fragment (see Exercise 3.3):

```
1 A := ∅; i := n
2 while i ≠ 0 do
3     if W(i) = W(i − 1) then i := i − 1    ▶ job i is not in the optimal set
4     else A := A ∪ {i}; i := p(i)          ▶ job i is in then optimal set
5 return A
```

3.3 Edit distance

Given two strings $x[1..m]$ and $y[1..n]$, we want to define a measure of *similarity* between them. For example, is the string "`scarp`" more similar to "`star`" or to "`escape`"? Such a measure is useful in several applications:

- In many types of text-editing software, a mis-typed word triggers a list of "similar" words from which the user can select the one they presumably meant to type.
- In computational biology strings over the alphabet {`A,C,G,T`}[5] represent genes. The "homology hypothesis" in biology holds that genes represented by similar strings are likely to have similar functions.

A useful definition of similarity between strings x and y, and the one we will use here, is *the minimum number of* ***edits*** *that can transform x to y*, where an edit is one of the following operations: insert a character, delete a character, or change a character. A useful way to visualize the process of transforming x to y via a sequence of edits is by ***aligning*** the two strings as shown below for the strings $x =$ "`banana`" and $y =$ "`plantain`".

Alignment position	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	−	a	−	−
	−	p	l	a	n	t	a	i	n

In aligning the two strings we place one above the other and we are allowed to introduce − symbols ("gaps") before or after some of the characters so that the resulting two strings have the same length. The gaps are interpreted as "insert" (if the − is in the top string) or "delete" (if the − is in the bottom string). So, for example, the above alignment tells us to transform "`banana`" to "`plantain`" using the following six edits:

- delete `b` in alignment position 1
- change `a` to `p` in alignment position 2,
- change `n` to `l` in alignment position 3,
- insert `t` in alignment position 6,
- insert `i` in alignment position 8,
- insert `n` in alignment position 9,

We say that the "cost" of this alignment is 6.

[5] for the four aminoacids that make up DNA: Adynine, Cytosine, Guanine, and Thymine

Here is another alignment of the same two strings:

Alignment position	1	2	3	4	5	6	7	8	9
	−	b	a	n	−	a	−	n	a
	p	l	a	n	t	a	i	n	−

This alignment has cost 5, because it involves 5 edits:

- insert `p` in alignment position 1
- change `b` to `l` in alignment position 2,
- insert `t` in alignment position 5,
- insert `i` in alignment position 7,
- delete `a` in alignment position 9,

And here is another alignment of these two strings that involves 5 edits:

Alignment position	1	2	3	4	5	6	7	8
	−	b	a	n	−	a	n	a
	p	l	a	n	t	a	i	n

There are two more alignments of "`banana`" and "`plantain`" that involve only 5 edits. (Can you find them?) But there is no alignment of these strings that involves fewer edits, so the edit distance between them is 5.

Let's now generalize the concept of alignment. An ***alignment*** of strings x and y is a pair of strings x' and y' so that (a) x' is x possibly with some "$-$" symbols interspersed, and similarly for y'; (b) $|x'| = |y'|$, and (c) there is no position i so that $x'[i] = y'[i] = -$. The ***cost*** of an alignment is the number of positions in which x' and y' differ. An alignment of x and y is ***optimal*** if there is no alignment of x and y with lower cost. The ***edit distance*** between x and y is the cost of an optimal alignment. Since the operations "insert" and "delete" are inverses of each other, the edit distance between x and y is the same as the edit distance between y and x. We are looking for an algorithm to solve the following problem:

Input: Two strings x and y.
Output: The edit distance between x and y.

(We may also be interested in an optimal alignment of x and y, but we will limit ourselves to computing the edit distance; see Exercise 3.4.)

Note that there is an exponential number of alignments between two strings (since for every position we have the option to add a $-$ or not). So finding the edit distance between two strings by brute force, i.e., by enumerating all possible alignments, is hopelessly inefficient. Fortunately we can solve the problem efficiently using dynamic programming.

Let $x[1..m]$ and $y[1..n]$ be the input strings. Suppose x' and y' is an optimal alignment for x and y. We analyze this optimal alignment with an eye to discovering the problem's optimal substructure. There are three possibilities for what the end of this optimal alignment looks like ($x[m]$ and $y[n]$ are the last characters of x and y, respectively):

- $x' = x''x[m]$ and $y' = y''y[n]$. By a cut-and-paste argument, x'' and y'' is an optimal alignment for $x[1..m-1]$ and $y[1..n-1]$.

- $x' = x''x[m]$ and $y' = y''-$. By a cut-and-paste argument, x'' and y'' is an optimal alignment for $x[1..m-1]$ and $y[1..n]$.
- $x' = x''-$ and $y' = y''y[n]$. By a cut-and-paste argument, x'' and y'' is an optimal alignment for $x[1..m]$ and $y[1..n-1]$.

We don't know which of these three possibilities actually applies, so we will compute the optimal alignment for $x[1..m-1]$ and $y[1..n-1]$, for $x[1..m-1]$ and $y[1..n]$, and for $x[1..m]$ and $y[1..n-1]$, and we will pick the one among them that results in the smallest edit distance.

So, we have been able to express the edit distance between $x[1..m]$ and $y[1..n]$ in terms of three smaller problems, namely, the edit distances between $x[1..m-1]$ and $y[1..n-1]$, between $x[1..m-1]$ and $y[1..n]$, and between $x[1..m]$ and $y[1..n-1]$. And to compute each of these we will apply the same idea recursively. This is the optimal substructure of our problem.

So, for each $i \in [0..m]$ and $j \in [0..n]$, we define

$$ED(i,j) = \text{the edit distance between } x[1..i] \text{ and } y[1..j]. \qquad (*)$$

(When $i = 0$, $x[1..i]$ is the empty string, and similarly for $j = 0$.) Note that once we have computed (*), the edit distance between x and y is simply $ED(m,n)$. So, the subproblems defined above suffice for finding the solution to our problem.

For all $i \in [1..m]$ and $j \in [1..n]$ let

$$\delta(i,j) = \begin{cases} 0, & \text{if } x[i] = y[j] \\ 1, & \text{otherwise.} \end{cases}$$

From the analysis that led us to the optimal substructure of the problem we conclude that the following recursive formula computes $ED(i,j)$:

$$ED(i,j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\big(ED(i-1,j-1) + \delta(i,j), ED(i-1,j) + 1, ED(i,j-1) + 1\big), & \text{otherwise.} \end{cases} \qquad (\dagger)$$

The justification of the correctness of this formula involves a case analysis along the lines described earlier in the context of discovering the optimal substructure of the problem, using straightforward cut-and-paste arguments. (For the base cases, note that the edit distance between a string z and the empty string is $|z|$.)

Following is pseudocode to compute the edit distance between two strings.

```
EditDistance(x[1..m], y[1..n])
1 for i := 0 to m do ED(i, 0) := i
2 for j := 1 to n do ED(0, j) := j
3 for i := 1 to m do
4     for j := 1 to n do
5         ED(i, j) := min(ED(i − 1, j − 1) + δ(i, j), ED(i − 1, j) + 1, ED(i, j − 1) + 1)
6 return ED(m, n)
```

The running time of this algorithm is dominated by the doubly-nested loop and is $\Theta(mn)$. After filling out the 0-th row and 0-th column, the algorithm fills out a two-dimensional table

row-by-row. We could have written it so that it fills out the table column-by-column. Any order will do, as long as before computing the value of a table entry the algorithm has computed the three values on which it depends: the one above and to the left, the one directly above, and the one immediately to the left.

The values of the table $ED(0..8, 0..6)$ computed by this algorithm for the strings $x =$ "`banana`" and $y =$ "`plantain`" are shown below:

	y:		p	l	a	n	t	a	i	n
	j:	0	1	2	3	4	5	6	7	8
x:	i:									
	0	0	1	2	3	4	5	6	7	8
b	1	1	1	2	3	4	5	6	7	8
a	2	2	2	2	2	3	4	5	6	7
n	3	3	3	3	3	2	3	4	5	6
a	4	4	4	4	3	3	3	3	4	5
n	5	5	5	5	4	3	4	4	4	4
a	6	6	6	6	5	4	4	4	5	5

Digression It is sometimes insightful to think of the operation of a dynamic programming algorithm on its input as a directed acyclic graph (dag), where nodes represent the subproblems solved by the algorithm, and an edge from subproblem A to subproblem B exists if and only if A is used in the solution of B. The edge (A, B) may have a weight, reflecting the cost or benefit (depending on whether we are solving a minimization of maximization problem, respectively) of extending an optimal solution to subproblem B in obtaining an optimal solution to subproblem A. Then the algorithm can be viewed as seeking a shortest (minimization problem) or longest (maximization problem) path in this dag.

Figure 3.2 shows the dag of the edit-distance algorithm applied to the strings "`banana`" and "`plantain`". All edges have weight 1 except those where weight 0 is explicitly shown (the edges with weight 0 are also shown in red). There are ten edges with weight 0: six for the three `a`'s of "`banana`" matching the two `a`'s of "`plantain`", and four for the two `n`'s of "`banana`" matching the two `n`'s of "`plantain`". The shortest paths from node $(0, 0)$ to node $(6, 8)$ are shown with bold edges; there are four such paths, of weight 5. Do you see how each path from $(0, 0)$ to $(6, 8)$ corresponds to an alignment of the two strings? **End of Digression**

A more realistic version of the problem

Some changes, insertions, or deletions may be more likely than others. For example, in the context of the text-editing application mentioned in the beginning of the section, typing "`s`" instead of "`a`" is much more likely that typing "`u`". Inserting an "`v`" at the end of a word is more likely than inserting "`t`". We can generalize the definition of edit distance in a way that

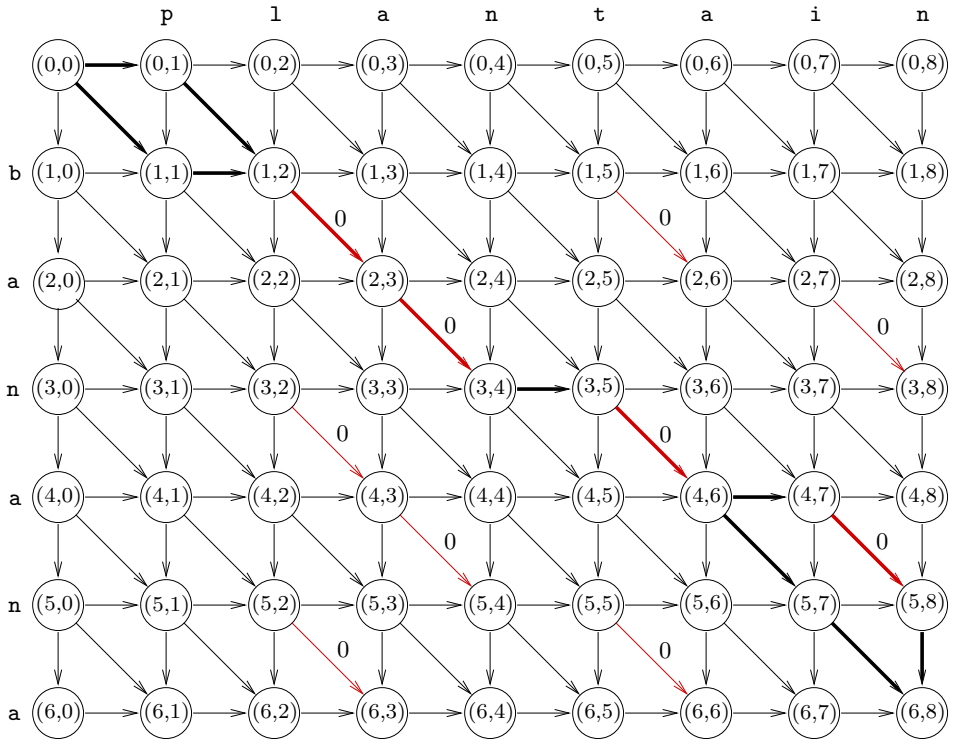


Figure 3.2: The dag of subproblems for the edit-distance algorithm

accounts for such application-specific considerations. The resulting function is just as easy to compute with essentially the same algorithm.

Let Γ be the alphabet over which the strings are defined. Without loss of generality, assume that Γ does not include the "gap" symbol $-$. Now define a function c: $\big(\Gamma \cup \{-\} \times \Gamma \cup \{-\}\big) - \big\{(-,-)\big\} \to \mathbb{Z}$, where for all $a, b \in \Gamma$, $c(a,b)$ is the cost of changing a to b, $c(a,-)$ is cost of deleting a, and $c(-,a)$ is the cost of inserting a. In many applications, we would expect that $c(a,a) = 0$, and $c(a,b) = c(b,a)$, but these restrictions are not necessary for what follows: The cost function can be completely arbitrary, reflecting the particulars of an application.

Let x, y be two strings over Γ, and x', y' be an alignment between x and y, as defined earlier. We can define the cost of this alignment as $\sum_{i \in [1..|x'|} c(x'[i], y'[i])$, an optimal alignment as one that has minimum cost, and the edit distance between x and y as the cost of an optimal alignment between the two strings. We can then compute the edit distance under this, more general, definition, using the dynamic programming algorithm presented earlier with the sole change of replacing line 5 by

$$\begin{aligned} ED(i,j) := \min\big(&ED(i-1,j-1) + c(x[i], y[j]),\\ &ED(i-1,j) + c(-, y[j]),\\ &ED(i,j-1) + c(x[i], -)\big). \end{aligned}$$

3.4 Optimal binary search trees

Suppose we are given a set of records with keys $1, 2, \ldots, n$, and the probability $P(i)$ of searching for the record with key i, for each $i \in [1..n]$. If we store the records in a linked list L, where the record with key i is placed in the node in position $\mathbf{pos}(i, L)$, the expected number of nodes accessed to find a record in L is

$$\mathbf{cost}(L) = \sum_{i=1}^{n} P(i) \cdot \mathbf{pos}(i, L). \tag{3.3}$$

There are $n!$ different ways in which to order the records in the list, but it is intuitively clear (and easy to prove by an exchange argument) that a list L that minimizes (3.3) contains the records in non-increasing probability: If $i_1, i_2, \ldots, i_n$ is a permutation of $[1..n]$ such that $P(i_1) \geq P(i_2) \geq \cdots \geq P(i_n)$, then a list L that minimizes $\mathbf{cost}(L)$ places key i_k in the k-th node, for each $k \in [1..n]$.

Now suppose we want to store the records in a binary search tree (BST) T, rather than a linked list. If $\mathbf{depth}(i, T)$ is the depth in T of the node where we store the record with key i, the expected number of nodes accessed to find a record in T is

$$\mathbf{cost}(T) = \sum_{i=1}^{n} P(i) \cdot \big(\mathbf{depth}(i, T) + 1\big) \tag{3.4}$$

(Recall that the depth of a node in a tree is the number of *edges* on the path from the root to the node, and so the number of nodes on that path is one more than the node's depth.) We wish to find a BST T that minimizes (3.4). More precisely, we want to solve the ***optimal BST problem***, specified as follows.

Input: The list of probabilities $P(1), P(2), \ldots, P(n)$ for the n keys.
Output: A BST T that minimizes (3.4).

A brute-force approach is impractical, as there are $\binom{2n}{n}/(n+1)$ different BSTs containing keys $[1..n]$. The obvious greedy approach, of putting the most frequently sought key at the root and recursing on the remaining keys — those smaller than the key at the root forming the left subtree using the same method and those greater than the key at the root forming the right subtree also using the same method — does not necessarily produce an optimal BST (construct a counterexample!).

The problem can be solved efficiently using dynamic programming. Suppose T is an optimal tree for the given probabilities. We analyze the structure of this tree with an eye to discovering the problem's optimal substructure. T's root contains some key $k \in [1..n]$, and has left and right subtrees T_L and T_R; see Figure 3.3.

It is clear that T_L is a BST that contains keys $[1..k-1]$ and T_R is a BST that contains keys $[k+1..n]$: If not, T would not be a BST that contains keys $[1..n]$ and has k in its root. But, furthermore, T_L is an *optimal* BST for keys $[1..k-1]$ and T_R is an *optimal* BST for keys $[k+1..n]$. This can be shown by a cut-and-paste argument but to use this argument we need to express the cost of a BST in terms of the costs of its left and right subtrees. The following lemma shows how to do so for any BST that contains a subset of the keys $[1..n]$.

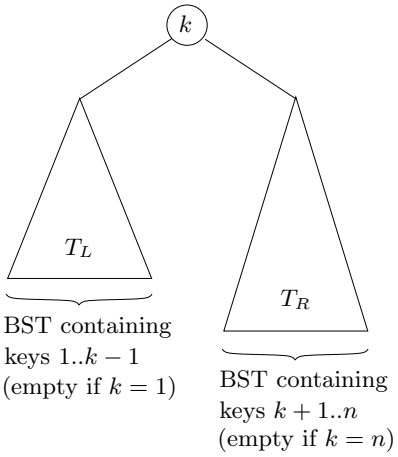


Figure 3.3: Structure of optimal BST

Lemma 3.1 *Let T' be a BST that contains some subset of the keys $[1..n]$, and let T'_L and T'_R be its left and right subtrees. Then*

$$\boldsymbol{cost}(T') = \boldsymbol{cost}(T'_L) + \boldsymbol{cost}(T'_R) + \sum_{\ell \in T'} P(\ell). \tag{3.5}$$

PROOF. By considering separately the terms of the cost of T' contributed by the keys in the nodes on the left subtree, the right subtree, and the root k (whose depth in T' is 0) we have:

$$\begin{aligned}
\mathbf{cost}(T') &= \sum_{\ell \in T'_L} P(\ell) \cdot \big(\mathbf{depth}(\ell, T') + 1\big) + \sum_{\ell \in T'_R} P(\ell) \cdot \big(\mathbf{depth}(\ell, T') + 1\big) + P(k) \\
&= \sum_{\ell \in T'_L} P(\ell) \cdot \mathbf{depth}(\ell, T') + \sum_{\ell \in T'_R} P(\ell) \cdot \mathbf{depth}(\ell, T') + \sum_{\ell \in T'} P(\ell) \\
&= \sum_{\ell \in T'_L} P(\ell) \cdot \big(\mathbf{depth}(\ell, T'_L) + 1\big) + \sum_{\ell \in T'} P(\ell) \cdot \big(\mathbf{depth}(\ell, T'_R) + 1\big) + \sum_{\ell \in T'} P(\ell) \\
&= \mathbf{cost}(T'_L) + \mathbf{cost}(T'_R) + \sum_{\ell \in T'} P(\ell).
\end{aligned}$$

□

With this lemma at hand, it is now clear that if T is optimal for keys $[1..n]$ and has root k, then T_L is optimal for keys $[1..k-1]$ and T_R is optimal for keys $[k+1..n]$: If not, cut T_L or T_R and paste in its place a better BST for the corresponding keys; by (3.5), the resulting tree is a better BST for keys $[1..n]$, contradicting that T is optimal.

Which k should we use for the root, though? We don't know in advance, so se will try out every possible $k \in [1..n]$ as the root, compute optimal subtrees for each possibility, and output the best one — i.e., the one with minimum cost. And how will we compute optimal subtrees?

Recursively, of course: T_L's root will contain some key $\ell \in [1..k-1]$ and left and right subtrees that are optimal BSTs for keys $[1..\ell-1]$ and $[\ell+1..k-1]$, respectively; and similarly for T_R. We have discovered the optimal substructure of our problem: The subproblems we want to solve are finding optimal BSTs for keys $[i..j]$ for every $i, j \in [1..n]$ such that $i \leq j$.

We need to pay some special attention to edge cases, when the left or right subtree of the optimal BST for keys $[i..j]$ is empty; i.e., the key in its root is i or j. To accommodate these we will include in our computation the (empty) tree containing keys $[i..i-1]$, for $i \in [1..n+1]$. Taking these edge cases into account, our subproblems will be to find optimal BSTs for keys $[i..j]$ for all $i \in [1..n+1]$ and $j \in [0..n]$ such that $i \leq j+1$.

As usual we first focus on finding the *value* of an optimal solution — in our case, the cost of an optimal BST. So, for all $i \in [1..n+1]$ and $j \in [0..n]$ such that $i \leq j+1$, we define

$$C(i,j) = \text{the cost of an optimal BST for keys } [i..j]. \tag{*}$$

(When $i = j+1$, $C(i,j) = 0$ since $[i..j] = \varnothing$.) Once we have computed these, the cost of the optimal BST we are seeking is simply $C(1,n)$. So, the subproblems defined in (*) suffice for finding the solution to our original problem, i.e., computing the cost of an optimal BST for all keys. Note that the number of nodes in the tree that contains keys $[i..j]$ is $j-i+1$, and so this quantity is a natural measure of the size of subproblem $C(i,j)$; it ranges from 0 (when $i = j+1$) to n (when $i = 1$ and $j = n$).

The following recursive formula computes the values $C(i,j)$ defined in (*):

$$C(i,j) = \begin{cases} 0, & \text{if } i = j+1 \\ \min\big\{C(i,k-1) + C(k+1,j) \colon k \in [i..j]\big\} + \sum_{\ell=i}^{j} P(\ell), & \text{otherwise.} \end{cases} \tag{†}$$

This is a sound recursion: The subproblem $C(i,j)$ (whose size is $j-i+1$) depends on subproblems $C(i,k-1)$ and $C(k+1,j)$ whose sizes are smaller, since $k \in [i..j]$. Thus we can compute $C(i,j)$ in order of increasing size $j-i+1$, as illustrated in Figure 3.4: First we compute the cost of empty trees (which is 0), then the cost of trees with one node, then the cost of trees with two nodes, three nodes, and so on, until finally we compute the cost $C(1,n)$ of the single tree with n nodes. The correctness of (†) follows from Lemma 3.1 and the analysis that led us to the optimal substructure of the problem.

Following is pseudocode to compute the cost of an optimal BST for keys $[1..n]$, given the number of keys n and the probability of searching for key i, for each $i \in [1..n]$. As noted earlier, the algorithm computes $C(i,j)$ in order of increasing $j-i$. We first compute the cost of the empty subtrees $C(i,i-1)$ as the base case, then the cost of all subtrees that contain one key, then the cost of all subtrees that contain two consecutive keys, three consecutive keys, and so on, based on (†). Before doing so we pre-compute $S(i) = \sum_{\ell=1}^{i} P(\ell)$ for $i \in [0..n]$; this allows us to compute in constant time the quantity $\sum_{\ell=i}^{j} P(\ell)$, for any $i, j \in [1..n]$ such that $i \leq j$, simply as $S(j) - S(i-1)$.

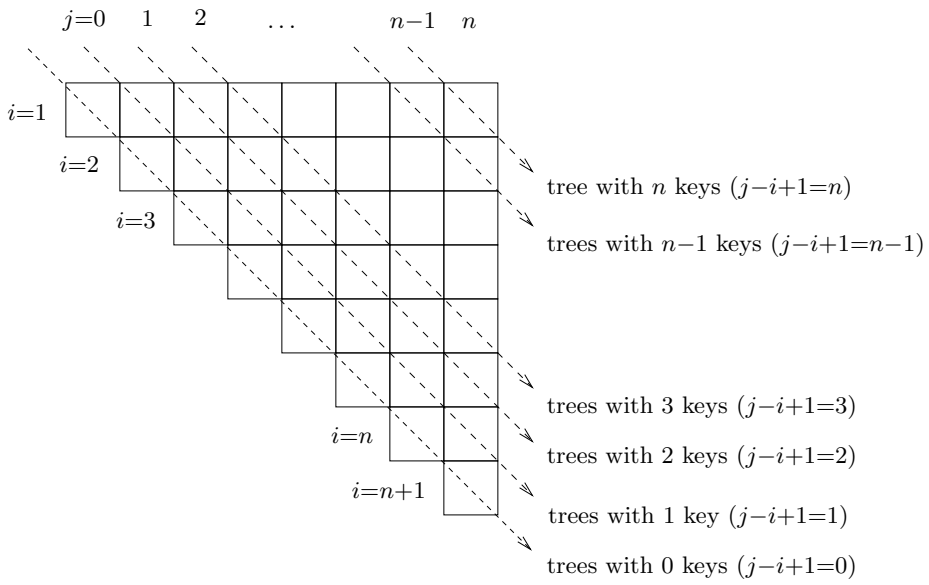


Figure 3.4: Order of evaluation of $C(i,j)$, $i \in [1..n+1]$, $j \in [0..n]$, $i \leq j+1$

```
   OptBST(n, P)
 1 S(0) := 0
 2 for i := 1 to n do S(i) := S(i − 1) + P(i)
 3 for i := 1 to n + 1 do C(i, i − 1) := 0
 4 for d := 0 to n − 1 do
 5     for i := 1 to n − d do
 6         j := i + d
 7         C(i, j) := +∞
 8         for k := i to j do
 9             if C(i, k − 1) + C(k + 1, j) + (S(j) − S(i − 1)) < C(i, j) then
10                 C(i, j) := C(i, k − 1) + C(k + 1, j) + (S(j) − S(i − 1))
11 return C(1, n)
```

The running time of the algorithm is dominated by the triply-nested loop in lines 4-10. Each loop runs for at most n iterations, and the body of the inner loop (lines 9-10) takes $O(1)$ time, so the total running time is $O(n^3)$. With a little more effort you can show that this is a tight asymptotic bound: The running time of the algorithm is $\Theta(n^3)$.

The algorithm can be retrofitted to compute an actual BST, not just its cost, without affecting its asymptotic running time (see Exercise 3.21).

Digression For any $n \in \mathbb{N}$, let $C(n)$ be the number of binary trees with n nodes. Below is a recursive formula to compute $C(n)$:

$$C(n) = \begin{cases} 1, & \text{if } n = 0 \\ \sum_{k=1}^{n} C(k-1) \cdot C(n-k), & \text{otherwise} \end{cases}$$

For the base case, there is one binary tree with no nodes, namely the empty tree. For the induction step, if the binary tree has n nodes, then its left subtree has $k-1$ nodes for some $k \in [1..n]$, so it could be any one of the $C(k-1)$ binary trees with $k-1$ nodes; and its right subtree has $n-k$ nodes, so it could be any one of the $C(n-k)$ binary trees with $n-k$ nodes.

$C(n)$ is known as the n-th ***Catalan number***.[6] There is a closed-form formula for this quantity: $C(n) = \binom{2n}{n}/(n+1)$.

In addition to counting the number of binary trees with n nodes, $C(n)$ also counts the number of a surprisingly diverse set of combinatorial objects, including:

- The number of ways of fully parenthesizing an expression that involves n binary operators (this was Catalan's motivation to study these numbers).
- The number of balanced strings of parentheses of length $2n$ (i.e., with n open and n close parentheses).
- The number of paths from the lower-left to the upper-right corner of an $n \times n$ grid that do not go above the diagonal that connects these two corners.
- The number of ways in which a regular polygon with $n+2$ vertices can be divided into triangles by non-intersecting diagonals.

If you are interested in learning more about these numbers, including a longer list of combinatorial objects they describe and a variety of proofs for the closed-form formula, check out the Wikipedia entry. **End of Digression**

3.5 Discrete knapsack

In Section 1.3 we studied a natural greedy algorithm for the *fractional* knapsack problem. To remind you, we are given a set of items $[1..n]$, the value v_i and weight w_i of each item i, and the capacity C of a "knapsack", where a thief can store the subset of items he or she steals. The thief must decide which items to steal so as to maximize the total value of the stolen items, subject to the constraint that the their total weight does not exceed C. In the fractional version of the problem, the thief can steal fractions of items, accruing the corresponding fraction of their value and incurring the corresponding fraction of their weight. This makes sense if the thief has broken into a bulk store: half a kilo of flour has half the value and half the weight of one kilo of flour; but it makes no sense if the thief has broken into a clothing store: half a jacket is worthless. In the more interesting, and (apparently) much more difficult, discrete (or 0-1) version of the problem, the thief must steal an item in its entirety or not at all. The goal remains the same. Following common practice, we will refer to this version of the problem as the ***knapsack problem*** (without the qualifier "discrete" or "0-1"), except when we want to emphasize the distinction from the fractional version. More precisely, the problem can be specified as follows.

[6] Named after the French-Belgian mathematician and revolutionary Eugène Catalan who introduced them.

Input: The value v_i and weight w_i of each item $i \in [1..n]$, and the capacity C of the knapsack. We will assume that w_i and C are *positive integers*; this is crucial for the algorithm we will see.

Output: A set $S \subseteq [1..n]$ such that

- (feasibility) $\sum_{i \in S} w_i \leq C$.
- (optimality) For every $S' \subseteq [1..n]$, if $\sum_{i \in S'} w_i \leq C$ then $\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i$.

The simple and natural greedy algorithm that worked for the fractional knapsack problem does not work for the discrete knapsack problem. (You should construct a counterexample.) We can solve this problem, however, using dynamic programming. Consider an optimal solution S; we will analyze it with an eye to discovering the problem's optimal substructure. There are two possibilities:

- Item n is not in S. Then S is an optimal set for the set of items $[1..n-1]$. So, we have a smaller problem (fewer items); we don't get any benefit from item n but we also don't incur the cost of its weight.
- Item n is in S. Then $S = S' \cup \{n\}$, and S' is an optimal set of items for $[1..n-1]$, but the knapsack now has capacity only $W - w_n$ for the remaining items. Note that this option exists only if $W \geq w_n$.

This suggests that, to express our problem as the extension of smaller subproblems we cannot limit the size of the problem to the number of items alone; we must introduce as an additional parameter the remaining capacity of the knapsack.

We do not know in advance which of these two possibilities results in a more valuable set of items: Is it better to forego item n, thereby saving its weight for other items but not accruing its value? Or is it better to steal item n, thereby accruing its value but incurring the cost of its weight so that there is less capacity remaining for other items? To find out, we will evaluate both of them and choose the one that yields higher value. And how do we evaluate each of these possibilities: Recursively, of course!

So we have discovered the optimal substructure of our problem. An optimal set of items among $[1..n]$ with available capacity C is an extension of an optimal solution to one of two subproblems: an optimal set of items among $[1..n-1]$ with available capacity the entire C (but no value from item n) and an optimal set of items among $[1..n-1]$ with available capacity only $C - w_n$ (but with the benefit of the value of item n).

As usual we first focus on computing the *value* of an optimal set, in this case the sum of the values of the items in the set. We therefore define the following subproblems, for each $i \in [0..n]$ and $c \in [0..C]$:

$$K(i, c) = \text{maximum value of a feasible subset of items } [1..i] \text{ when the available capacity is } c. \qquad (*)$$

If we compute all values of $K(i, c)$, the quantity we want, i.e., the value of an optimal subset of all items, is simply $K(n, C)$.

The following recursive formula computes the values $K(i, c)$ defined in (*).

$$K(i,c) = \begin{cases} 0, & \text{if } i = 0 \text{ or } c = 0 \\ K(i-1,c), & \text{if } i > 0,\ c > 0 \text{ and } c < w_i \\ \max\bigl(K(i-1,c),\, v_i + K(i-1,c-w_i)\bigr), & \text{otherwise.} \end{cases} \qquad (\dagger)$$

To justify the correctness of this formula, for the base cases we note that if $i = 0$, the set of items $[1..0]$ is empty, so $K(0, c) = 0$ for all c; and if $c = 0$, the knapsack has no capacity, so $K(i, 0) = 0$ for all i. For the induction step, we reason as in the analysis that led us to the optimal substructure. For the optimal set S_i for items $[1..i]$ when the capacity is c, there are two possibilities. Either

- $i \notin S_i$, in which case S_i is optimal for items $[1..i-1]$ and all of the capacity c is available, and so $K(i, c) = K(i-1, c)$; or
- $i \in S_i$, in which case it must be that $c \geq w_i$ and S_i is optimal for items $[1..i-1]$, but with only the capacity $c - w_i$ available, and so $K(i, c) = v_i + K(i-1, c-w_i)$.

Thus, $K(i, c)$ is the maximum of these two possibilities.

Following is pseudocode to compute the value of an optimal knapsack.

KNAPSACK$(v_1, \ldots, v_n; w_1, \ldots, w_n; C)$
1 **for** $c := 0$ **to** C **do** $K(0, c) := 0$
2 **for** $i := 1$ **to** n **do** $K(i, 0) := 0$
3 **for** $i := 1$ **to** n **do**
4 **for** $c := 1$ **to** C **do**
5 **if** $c < w_i$ **then** $K(i, c) := K(i-1, c)$
6 **else** $K(i, c) := \max\bigl(K(i-1, c), v_i + K(i-1, c-w_i)\bigr)$
7 **return** $K(n, C)$

The algorithm can be retrofitted to output an optimal set of items, rather than its value; see Exercise 3.23

The running time of this algorithm is $\Theta(nC)$. Despite appearances, this is *not* a polynomial-time algorithm and it is important to understand why. By definition an algorithm is polynomial time if its running time is bounded from above by a polynomial of its input *size*. To see why nC is not a polynomial of the algorithm's input size suppose, for simplicity, that all values, weights, and the knapsack capacity C are numbers represented by (roughly) the same number of bits. Then the input size of the algorithm is $\Theta(n \log C)$, since it takes $\lfloor \log_2 C \rfloor + 1$ bits to represent the number C. So, the quantity nC is *exponentially larger* than the input size $n \log C$.

Algorithms whose running time is bounded from above by a polynomial of the input's *value* (as opposed to its *size*) are called ***pseudopolynomial***. Of course this designation is meaningful only for algorithms that involve numeric input, as is the case with the knapsack algorithm.

It is not known whether a truly polynomial-time algorithm for the knapsack problem exists, and it would be a big surprise for most computer scientists (and a big advance for science) if there was one, since the discrete knapsack problem is NP-hard. (See the discussion about **NP**-complete problems on page 10 and footnote 2 on that page.)

3.6 Dynamic programming and recursion

Let's review the Bellman equations (†) for the five dynamic programming algorithms we have studied so far:

- Longest increasing subsequence: $L(i) = 1 + \max\{L(j)\colon j \in [1..i-1] \text{ and } A[j] < A[i]\}$
- Weighted-interval scheduling: $W(i) = \max\big(W(i-1), W\big(p(i)\big) + \mathbf{wt}(i)\big)$
- Edit distance: $ED(i,j) = \min\big(ED(i-1,j-1) + \delta(i,j), ED(i-1,j)+1, ED(i,j-1)+1\big)$
- Optimal BST: $C(i,j) = \sum_{\ell=i}^{j} P(\ell) + \min\{C(i,k-1) + C(k+1,j)\colon k \in [i..j]\}$
- Knapsack: $K(i,c) = \max\big(K(i-1,c), v_i + K(i-1,c-w_i)\big)$.

All of these are recursive formulas and the most natural way of coding these computations would appear to be as recursive programs. Yet we coded them iteratively. To see why, consider the recursion tree that would result from evaluating $L(5)$ when the input is an increasing sequence of numbers, shown in Figure 3.5.

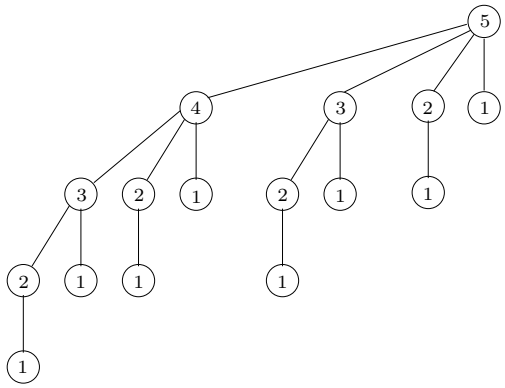


Figure 3.5: The recursion tree for the LIS Bellman equation ($i = 5$)

As you can see there are $16 = 2^{5-1}$ recursive calls made. In general, if the input is an increasing sequence of length n, there will be 2^{n-1} calls made, so the recursive computation takes exponential time! Of course, there is a tremendous amount of redundancy: In our example, the call to compute $L(5)$ results in eight calls to compute $L(1)$, four to compute $L(2)$, two to compute $L(3)$, and one to compute $L(4)$.

This behaviour is not limited to the LIS problem. If you look at any one of the above Bellman equations, you will see that (at least) for some inputs, the number of recursive calls generated is exponential. This is not surprising: The size of the subproblem reduces by a constant amount, which means that the depth of the recursion is linear and so the number of nodes in the recursion tree is exponential.[7] This means that writing the computation recursively will result in an exponential-time algorithm, unless special precautions are takes, as explained below.

[7]Recall the difference we noted between dynamic programming and divide-and-conquer (page 72) concerning the depth of the recursion in each.

Notice that although the number of recursive calls being made is exponential, the number of subproblems that are being solved is polynomial (in the case of knapsack, *pseudo*polynomial). The exponential number of recursive calls is due to the fact that the same problem is being solved over and over again. There is a trick, called ***memoization*** that solves this problem. The idea is to maintain as a global variable (accessible to all recursive calls) a table indexed by the values of the parameters of the recursion. So, for example, for $L(i)$ and $W(i)$, the table is one-dimensional; for $ED(i,j)$, $C(i,j)$, and $K(i,c)$, the table is two-dimensional. The entries of the table are initialized to some special value, say NIL. When a recursive call with a particular set of values for the parameters is made, the program first consults the table entry that corresponds to that set of values. If the entry is still NIL, no recursive call has yet been made for those values of the parameters; the program then computes the output *and stores it in that entry of the table*. Subsequent calls for the same values of the parameters simply return the value stored in that entry of the table — they do not recompute the output. (You can probably now see the origin of this method's ugly name: The first call of the recursive procedure with given values for the parameters "leaves a memo" with the output it computed for subsequent calls to use without recomputing it.)

To illustrate this idea, let us write a recursive program to compute the edit distance between $x[1..m]$ and $y[1..n]$ — first coding it naively (NED) and then using memoization (MED).

```
NED(m, n)
1 if m = 0 then return n
2 elsif n = 0 then return m
3 else
4     if x[m] = y[n] then δ := 0 else δ := 1
5     return min(NED(m − 1, n − 1) + δ, NED(m − 1, n) + 1, NED(m, n − 1) + 1)
```

The running time of this algorithm is exponential. For example, suppose $m = n$ and let $R(n)$ be the number of recursive calls made by $\mathrm{NED}(n,n)$. $R(n)$ satisfies the recurrence $R(n) \geq 3R(n-1)$, and so $R(n) = \Omega(3^n)$.

```
MED(m, n)
1 if m = 0 then return n
2 elsif n = 0 then return m
3 else
4     if x[m] = y[n] then δ := 0 else δ := 1
5     if M[m, n] = NIL then
6         M[m, n] := min(MED(m − 1, n − 1) + δ, MED(m − 1, n) + 1, MED(m, n − 1) + 1)
7     return M[m, n]
```

The running time of this algorithm is $O(mn)$. To see this observe that each recursive call takes constant time *excluding* the time required by the recursive calls it makes in line 6. Therefore, the running time is linear in the number of recursive calls. For a given value of i and j, only one call is made to $\mathrm{MED}(i,j)$, because of the conditional in line 5: after the first such call $M[i,j]$ is set to a value other than NIL. Therefore, the call $\mathrm{ED}(m,n)$ results in at most mn recursive calls.

So we can code dynamic programming algorithms recursively, as would seem quite natu-

ral, but we must remember to use memoization so as to avoid an exponential blowup of the running time. There are even some cases where the recursive formulation of the algorithm has advantages. One such example is the knapsack algorithm. For some inputs we might not need all subproblems to compute the optimal solution to the original problem. For instance suppose that all weights and the knapsack's capacity are multiples of ten. Then we only need to compute $K(n, c)$ for values of c that are multiples of ten, which means that 90% of the subproblems solved by the iterative algorithm are wasted. Had we written the algorithm recursively only the needed subproblems would have been computed, and this is done automatically: MED(m, n) calls itself *only* for the subproblems on which it depends, each of which calls the procedure *only* for the subproblems on which it depends, and so on.

Exercises for Chapter 3

The presentation of the algorithms in this chapter attempted to illustrate the process by which one might go about discovering them: Analyze the problem with an eye towards discovering its optimal substructure and use this analysis to determine the subproblems to solve, the recursive formula by which to compute these subproblems, and the reason why this formula correctly computes the subproblems. Although (hopefully) pedagogically useful, this is not necessarily the cleanest way to present an algorithm if the goal is to *describe* it (and its correctness) to someone, as opposed to illuminating how one might *discover* it. When you write up solutions to the exercises below that ask you to develop a dynamic programming algorithm it is better to use a dryer and more direct style, consisting of the steps listed below.

(1) State (very clearly!) the subproblems to solve (typically denoted (*) in this chapter).
(2) State how to compute the original problem given the solution to these subproblems.
(3) Give the recursive formula by which to compute the subproblems (typically denoted (†) in this chapter).
(4) Justify why the recursive formula correctly computes the subproblems. (This is the core argument for the algorithm's correctness; if it is done well, little else is needed as the pseudocode is usually a straightforward transcription of this formula.)
(5) Describe the algorithm in pseudocode.
(6) Analyze the algorithm's running time.

3.1 Give a careful correctness proof for the pseudocode of the LIS algorithm in Section 3.1, specifically the version that returns a LIS (not just its length) using **pre**.

3.2 State and prove an invariant which implies that the second way of retrofitting the LIS algorithm in Section 3.1 (the version that does not need to precompute **pre**) returns a LIS of the given array A.

3.3 State and prove an invariant which implies that the retrofitted algorithm for weighted-interval scheduling in Section 3.2 returns an optimal set.

3.4 Retrofit the edit distance algorithm in Section 3.3 so that it outputs an optimal alignment of the given strings.

3.5 Let $A[1..m]$ and $B[1..n]$ be sequences. The ***longest common subsequence*** of A and B is a sequence C such that

- C is a subsequence of both A and B, and
- if C' is a subsequence of both A and B, then C is at least as long as C'.

(Recall that the elements of a subsequence of A *need not* appear consecutively in A.) Give an efficient algorithm that, given A and B, computes the length of a longest subsequence of A and B. Then retrofit the algorithm to compute an actual longest subsequence of A and B. Justify the algorithm's correctness and analyze its running time.

3.6 Consider the following game played by two players, A and B. There is a sequence of "cards", each sporting a (possibly negative) integer that represents the number of points the card is worth. The cards are laid on the table face-up, i.e., their values are visible to the players. The players, starting with A, take turns picking one card from either end of the sequence of cards; i.e., the player whose turn it is can only pick the first or the last card in the sequence. The objective of each player is to accumulate as many points as possible.

(a) Give an example that demonstrates that the greedy strategy of picking the card with the highest value between the two that can be legally chosen does not work.

(b) Give an algorithm that takes as input the sequence of values of the cards and determines the maximum number of points that player A can accumulate assuming player B also plays optimally (i.e., to maximize the number of points they accumulate).

3.7 In Exercise 1.10 we explored the problem of placing books on shelves so as to minimize empty space. We saw that a simple greedy algorithm minimizes the ***sum*** of the empty spaces on the shelves used, but does not minimize the ***maximum*** empty space on any shelf used. Now we want to design an algorithm that minimizes the latter objective function.

Let n be the number of books, d_i be the depth of book $i \in [1..n]$, and L be the length of each shelf. A ***placement*** (of books on shelves) is a sequence $i_0 < i_1 < \ldots < i_k$ such that $i_0 = 0$ and $i_k = n$; intuitively i_s is the last book placed on shelf s, where for convenience we imagine a "shelf" 0 that contains "book" 0. A placement is ***feasible*** if for every $s \in [1..k]$, $\sum_{i_{s-1}<j\le i_s} d_j \le L$ (the books on shelf s fit). The ***cost*** of a feasible placement is $\max\{L - \sum_{i_{s-1}<j\le i_s} d_j : 1 \le s \le k\}$ (i.e., the maximum empty space on any shelf). A placement is ***optimal*** if it is feasible and has minimum cost. The problem we wish to solve is:

Input: The depths $d_1, d_2, \ldots, d_n$ of the books, and the length L of the shelves, where $d_i \le L$ for all $i \in [1..n]$.

Output: The cost of an optimal placement.

(a) Describe an efficient dynamic programming algorithm that solves this problem. Justify the correctness of your algorithm and analyze its running time.

(b) Retrofit your algorithm from part (a) to output the actual optimal placement, without increasing its running time by more than a constant factor. You need not justify the correctness of your retrofit or analyze its running time, but you should explain clearly the meaning of any additional information your algorithm computes.

3.8 Let $A[1..n]$ be a non-empty array of numbers, which may be negative, zero, or positive, and are not necessarily integers. We want to find the maximum number p that is the product of all the numbers in subarray $A[i..j]$ of A, over all i, j such that $1 \le i \le j \le n$. (If $i = j$, the product of all the numbers in $A[i..j]$ is, by definition, the number $A[i]$.)

Use dynamic programming to give an $O(n)$ algorithm that solves this problem. (Assume that basic arithmetic operations and comparisons take $O(1)$ time each, regardless of the size of the numbers on which they operate.) Describe your algorithm in pseudocode, explain why it is correct, and analyze its running time.

3.9 Let $A = a_1, a_2, \ldots, a_n$ be a sequence of (negative, zero, or positive) integers, $n \ge 1$. A subsequence A' of A is ***nonconsecutive*** if it does not contain consecutive elements of A; more precisely $A' = a_{i_1}, a_{i_2}, \ldots, a_{i_k}$, where $1 \le i_1 < i_2 < \ldots < i_k \le n$ and for every $t \in [2..k]$, $i_t \ne i_{t-1} + 1$. Give a linear-time dynamic programming algorithm that, given A, returns a nonconsecutive subsequence of A whose elements have the maximum possible sum. (The sum of the elements of an empty sequence is defined to be zero.)

Describe your algorithm in pseudocode, explain why it is correct, and analyze its running time.

3.10 Let $A[1..n]$ be a non-empty array of integers (they may be positive, negative, or zero) and k be a positive integer. We want to find the maximum sum of a (contiguous) subarray of A *of length at least* k. That is, we want to find $\max\{\sum_{t=i}^{j} A[t] : 1 \le i \le j \le n \text{ and } j - i + 1 \ge k\}$.

Use dynamic programming to give an $O(n)$ algorithm that solves this problem. Describe your algorithm in pseudocode, explain why it is correct, and analyze its running time.

3.11 Let x be a string. A ***division*** of x is a sequence of non-empty strings $x_1, x_2, \ldots, x_k$ such that $x = x_1 \circ x_2 \circ \cdots \circ x_k$, where $\circ$ denotes concatenation. (Note that if x is the empty string then it has a single division, namely the empty sequence — i.e., $k = 0$.) A ***legal division*** of x is a division of x such that every string in the division is a word in the English language. (The sequence of words does not have to make sense for it to be a legal division; we only require that the individual words are legitimate English words.) For example, there are exactly four legal divisions of the string "`bananagram`", and zero legal divisions of the string "`supercalifragilisticexpialidocious`".

Give a polynomial-time dynamic programming algorithm that takes as input a string $x[1..n]$ and returns the number of different legal divisions of that string. Justify the correctness of your algorithm and analyze its running time.

You may assume that you have access to a procedure $\textsc{Dict}(x, i, j)$ that, given a string $x[1..n]$ and indices i, j such that $1 \le i \le j \le n$, returns 1 if $x[i..j]$ is an English word and returns 0 otherwise. You may further assume that a call to this procedure takes $O(1)$ time.

3.12 Let x, y, and z, be strings over some alphabet Σ, where $|z| = |x| + |y|$ (here $|w|$ denotes the length of a string w). The string z is a ***shuffle*** of x and y if there are (***possibly empty***) strings $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_k$ so that $x = x_1 x_2 \ldots x_k$, $y = y_1 y_2 \ldots y_k$, and $z = x_1 y_1 x_2 y_2 \ldots x_k y_k$. Intuitively, z can be constructed by concatenating pieces of x and y, in the order in which these pieces appear in x and y, respectively. For example, `alabama` is a shuffle of `aba` and `lama`, but `amalaba` is not.

We want an algorithm that, given as input three strings x, y, and z such that $|z| = |x| + |y|$, returns **true** if z is a shuffle of x and y, and returns **false** otherwise.

(a) Consider the following greedy strategy. Find the longest nonempty prefix w of x or y that matches a prefix of z; remove the prefix w from z and from either x or y (whichever of the two it came from — only from one, if they both have the same longest prefix that matches a prefix of z). Repeat until all three of x, y, and z are empty, in which case return **true**; or there is no nonempty prefix w of either x of y that matches a prefix of z, in which case return **false**. Give a simple example that demonstrates that this greedy algorithm is not correct.

(b) Give a polynomial-time dynamic programming algorithm that solves the problem. Justify the correctness of your algorithm and analyze its running time.

3.13 Let A and B be sequences. A is a ***supersequence*** of B if B is a subsequence of A. A ***palindrome*** is a (finite) sequence that is equal to its reversal — i.e., it reads the same backwards as forwards — for example, the inscription on the fountain in the courtyard of Agia Sophia: $\nu\iota\psi o\nu\alpha\nu o\mu\eta\mu\alpha\tau\alpha\mu\eta\mu o\nu\alpha\nu o\psi\iota\nu$ which (with inter-word spaces omitted) means "wash the sins, not just the face".

(a) Give an $O(n^2)$ dynamic programming algorithm that takes as input a sequence $A[1..n]$ and returns the length of a shortest supersequence of A that is a palindrome. For example, if $A =$ `r,e,s,t,a,t,e` the algorithm should return 9 since the shortest supersequence of A that is a palindrome is `r,e,s,t,a,t,s,e,r`. Justify the correctness of your algorithm.

(b) Retrofit your algorithm in part (a) so that it returns a minimum-length supersequence of A that is a palindrome (instead of its length, as in part (a)).

3.14 Emon Lusk runs a single-person consulting service operating in Toronto, Canada and Melbourne, Australia. The consulting service has one-day projects lined up for each of the next $n \geq 2$ days in each of these locations. For each $i \in [1..n]$, $T[i]$ is the profit for working on the day i project in Toronto and $M[i]$ is the profit for working on the day i project in Melbourne; both are non-negative numbers. To work on a project in either of these locations, Emon needs to be physically present at that location. Moving from one location to the other takes one day during which Emon cannot work on any project.

A ***schedule*** is an array $S[1..n]$ such that $S[i]$ is `T`, `M`, or $\perp$ meaning, respectively, that on day i Emon is in Toronto, Melbourne, or traveling from one to the other. The ***value*** of a schedule S is the total profit that Emon makes by working on the project in the city where he is presently located according to schedule S (no profit on days when Emon is traveling). Emon can start on day 1 in either location, and wants to find a schedule of maximum value.

For example, if $n = 6$, $T = [10, 3, 5, 3, 17, 30]$ and $M = [1, 5, 10, 20, 18, 10]$ the schedule of maximum value is $S = [\mathtt{T}, \perp, \mathtt{M}, \mathtt{M}, \perp, \mathtt{T}]$ and its value is $10 + 0 + 10 + 20 + 0 + 30 = 70$.

Describe an efficient dynamic programming algorithm that, given the profit arrays $T[1..n]$ and $M[1..n]$, returns a schedule of maximum value. Justify the correctness of your algorithm and analyze its running time.

3.15 We are given an $n \times n$ array $B[1..n, 1..n]$ of bits (0s or 1s). We want to find a largest square of 1s within this array. (For example, the array could represent a map, where a 0 denotes a square parcel of private land and a 1 denotes a square parcel of public land. We want to find the largest possible square piece of public land, say to build a new school.)

More precisely, a ***square of 1s*** is a triple of integers (i, j, ℓ) such that $i, j \geq 1$, $\ell \geq 0$, $i + \ell \leq n$, $j + \ell \leq n$, and $B[i', j'] = 1$ for all i', j' such that $i \leq i' \leq i + \ell$ and $j \leq j' \leq j + \ell$. We want to find a square of 1s such that ℓ is as large as possible.

Describe and justify the correctness of a polynomial-time dynamic programming algorithm to solve this problem, and analyze its running time. For full credit, your algorithm should run in $O(n^2)$ time.

3.16 Procedure $\textsc{Split}(A, i, A_1, A_2)$, where $A[1..n]$ is an array and $i \in [1..n-1]$, copies $A[1..i]$ into array A_1 of length i and $A[i + 1..n]$ into array A_2 of length $n - i$. Thus this procedure takes n units of time.

We can split A into $k + 1$ pieces at indices $1 \leq i_1 < i_2 \cdots < i_k < n$, by calling $\textsc{Split}$ k times. The *order* in which we cut A into these pieces may affect the time needed to perform the cuts. For example, if A has length 8 and we want to cut it into three pieces at indices $i_1 = 2$ and $i_2 = 4$, the time to cut A in the order i_1, i_2 is $8 + 6 = 14$, while the time to cut it in the order i_2, i_1 is $8 + 4 = 12$. We want to find a sequence of cuts that minimizes the total time to perform them. So, we want to solve the following problem.

Input: A positive integer n (the length of the array) and a sequence of positive integers $i_1, i_2, \ldots, i_k$ such that $1 \leq i_1 < i_2 < \cdots < i_k < n$ (the indices where we want to split the array).

Output: The minimum total time to cut an array of length n at indices $i_1, i_2, \ldots, i_k$.

(a) Prove that the following greedy algorithm does not work: Cut the array at the index i_m, among $i_1, i_2, \ldots, i_k$ that is closest to $n/2$ (ties resolved arbitrarily), and recursively split each of the two subarrays using the same idea. More precisely, cut at i_m and then recursively find the best way to cut an array of size i_m in positions $i_1, i_2, \ldots, i_{m-1}$ (no recursion if $m = 1$) and recursively find the best way to cut an array of size $n - i_m$ in positions $i_{m+1} - i_m, i_{m_2} - i_m, \ldots, i_k - i_m$ (no recursion if $m = k$).

(b) Develop a polynomial-time dynamic programming algorithm that solves the problem. Justify the correctness of your algorithm and analyze its running time.

(c) Retrofit your algorithm in part (b) so that it outputs a permutation of $i_1, i_2, \ldots, i_k$ indicating an order in which to apply the cuts so as to minimize the total time to perform them.

3.17 Let $a_1\, b_1\, a_2\, b_2\, a_3\, \ldots\, a_{n-1}\, b_{n-1}\, a_n$ be a sequence where $a_i \in \{0, 1\}$ (with 0 interpreted as

"false" and 1 as "true") for each $i \in [1..n]$, and $b_i \in \{\wedge, \vee, \oplus\}$ (the Boolean operators for and, or, and exclusive-or, respectively) for each $i \in [1..n-1]$. Thus, this sequence represents an unparenthesized Boolean expression such as $1 \oplus 1 \wedge 0 \vee 0$. The actual value of such an expression depends on the order in which the logical operators are applied, i.e., on how the expression is parenthesized. For instance the above example can be parenthesized in five ways (check them out!), some evaluating to 0 and some to 1. On the other hand, the unparenthesized Boolean expression $1 \wedge 0 \oplus 0 \oplus 0$ does not evaluate to 1, no matter how it is parenthesized.

(a) Give a *polynomial-time dynamic programming algorithm* that determines whether a given unparenthesized Boolean expression can be parenthesized to evaluate to 1. More precisely, your algorithm should solve the following problem:

Input: Sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_{n-1}$, where $a_i \in \{0, 1\}$ for each $i \in [1..n]$ and $b_i \in \{\wedge, \vee, \oplus\}$ for each $i \in [1..n-1]$.
Output: 1 if $a_1 b_1 a_2 \ldots a_{n-1} b_{n-1} a_n$ can be parenthesized so that it evaluates to 1, and 0 otherwise.

Justify the correctness of your algorithm and analyze its running time.

(b) Retrofit your algorithm in part (a) so that it outputs a parenthesization that evaluates the input expression to true, if such a parenthesization exists.

3.18 A digital image is a two-dimensional array $P[1..m, 1..n]$. Each entry (i, j) of P is called a ***pixel*** and contains a number, encoding colour. For example, an 8-bit grayscale encodes 256 different intensities of light, representating variations of gray: 0 is total absence of light (black) and 255 is total presence of light (white). Suppose we want to resize our $m \times n$ image P to an $m \times (n-1)$ image with minimal loss of visual information, as perceived by the human eye. Deleting a column of pixels is not necessarily the best choice as the column might contain many "edges" of objects that are crucial in interpreting the image. Here is a better way of doing it.

A (vertical) ***seam*** of P is a sequence of pixels $S = (1, j_1), (2, j_2), \ldots, (m, j_m)$, one per row, so that the columns of successive pixels differ by at most one: for each $i \in [1..m]$, $j_i \in [1..n]$; and for each $i \in [2..m]$, $|j_i - j_{i-1}| \leq 1$. (Figure 3.6 shows in red a vertical seam, and a similarly defined horizontal seam that we will not discuss here.)

We want to reduce the number of columns of the image by one by removing a seam whose deletion is least noticeable. Pixels containing values very similar to those of their neighbours' are less noticeable when removed. Thus we define the "cost" of removing pixel (i, j) as

$$\mathbf{cost}(i,j) = \begin{cases} \infty, & \text{if } j = 1 \text{ or } j = n \\ |P[i,j] - P[i,j-1]| + |P[i,j] - P[i,j+1]|, & \text{otherwise} \end{cases}$$

(The first clause makes it prohibitively expensive to remove pixels at the vertical edges of the image.) The cost of removing the seam $S = (1, j_1), (2, j_2), \ldots, (m, j_m)$ is defined as $\mathbf{cost}(S) = \sum_{i=1}^{m} \mathbf{cost}(i, j_i)$.

The vertical seam depicted Figure 3.6 is a minimum-cost seam. Note how it tends to skirt object boundaries, such as the rock at the bottom, by seeking pixels whose horizontal

Figure 3.6: Vertical and horizontal seams of an image

neighbours are similar. (The same comment applies to the horizontal seam shown in that image, except that we now we seek pixels whose *vertical* neighbours are similar.)

Give a polynomial time dynamic programming algorithm that, given an image P, returns a minimum-cost seam of P. Note that the algorithm should return the seam itself, not its cost. Describe your algorithm in pseudocode, explain why it is correct, and analyze its running time.

Note: The idea outlined in this exercise is a simplified version of a technique used for image resizing, for example so that it can be displayed on a different device. There are various techniques used to this end, such as "letterboxing", cropping, or rescaling along the horizontal or vertical axis. Letterboxing, however, introduces unsightly black bars on the edges of the output; cropping may eliminate important elements of the image; and rescaling may create unnaturally elongated or flattened images. "Seam carving", invented by the graphics researchers Avidan and Shamir and now implemented in Photoshop, is an alternative technique for image resizing that repeatedly removes (or adds) minimum-cost (vertical or horizontal) seams. In many instances it avoids the pitfalls of letterboxing, cropping, and rescaling, and produces realistic facsimilies of the original image. At the core of this technique is the identification of a minimum-cost seam as described in this exercise, except that Avidan and Shamir use more sophisticated cost functions. In Figure 3.7 you can see, from left to right, an original image and then that image resized by cropping, rescaling, and seam carving.

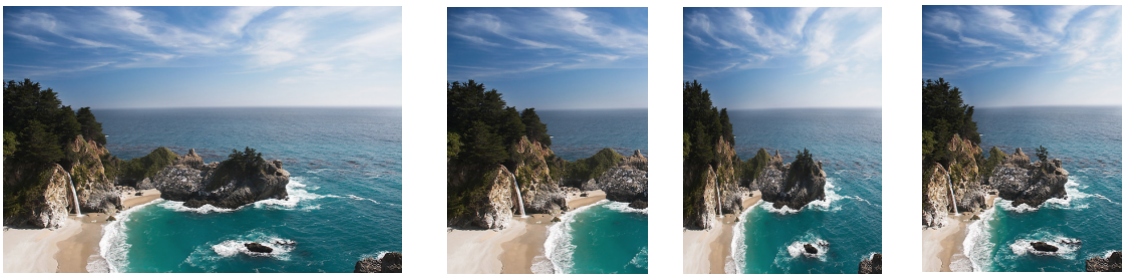


Figure 3.7: An original image resized by cropping, rescaling, and seam carving

For more information on this technique you can read the paper that introduced this resizing technique, and from which Figures 3.6 and 3.7 are taken.

3.19 Business is booming for the catering service described in Exercise 1.12. The decision is therefore made to expand the variety of orders so that each may take any number of time units, not necessarily just one as in the original setting. To ensure the high quality of the product, however, the same single cook still does all the cooking. The goal is to find a most profitable schedule for filling orders in this new and more general setting.

To be more precise, the catering service receives a set of n orders. Order i, $1 \leq i \leq n$, has a ***deadline*** d_i by which it must be completed, a ***length*** $\ell_i \leq d_i$ that it requires to be completed, and a ***profit*** $p_i > 0$ that the catering service receives if the order is completed by the deadline; if the order is completed after the deadline there is no profit made and therefore it is not filled at all. Deadlines and lengths are expressed as *positive integers*, to be thought of as multiples of an appropriate time unit.

The cook starts work at time 0. The catering service must decide on a schedule in which to fill a subset of the orders, each completed by its deadline. A schedule now is a function S that maps each order i to the finish time $S(i)$ of preparing that order, where $S(i) \geq \ell_i$ is a positive integer, meaning that order i will be prepared during the interval $\big[S(i) - \ell_i, S(i)\big)$; or to the finish "time" $+\infty$, meaning that order i will not be filled. We require that:

(1) every scheduled item must finish before its deadline: for any order i, if $S(i) \neq +\infty$ then $S(i) \leq d_i$; and

(2) no two scheduled orders conflict: for any distinct orders i and j, if $S(i) \neq +\infty$ and $S(j) \neq +\infty$, then the intervals $\big[S(i) - \ell_i, S(i)\big)$ and $\big[S(j) - \ell_j, S(j)\big)$ are disjoint.

The problem is to design an algorithm that, given as input the n triples (d_i, ℓ_i, p_i), returns as output a schedule S that maximizes the total profit: for every schedule S', $\sum_{i:\, S(i) \neq +\infty} p_i \geq \sum_{i:\, S'(i) \neq +\infty} p_i$.

(a) Give a counterexample proving that the following greedy strategy does not work: Sort and consider the orders by decreasing profit. For the current order i, if there is a time t such that $\ell_i \leq t \leq d_i$ and the interval $(t - \ell_i, t)$ is disjoint from all intervals of jobs scheduled so far, let t^* be the maximum such time and schedule order i to finish at time t^* — i.e., set $S(i) := t^*$); otherwise, don't fill order i — i.e., set $S(i) := +\infty$.

(b) Describe and justify the correctness of a dynamic programming algorithm that finds the *maximum profit* achievable by scheduling a subset of the given orders. Analyze the running time of your algorithm. Is your algorithm a polynomial-time one? Explain why or why not.

(c) Retrofit the algorithm in part (b) so that it returns a schedule that achieves the maximum profit (as opposed to returning the maximum profit).

3.20 We have two identical machines in which to execute n jobs, numbered $1, 2, \ldots, n$. Each machine can execute one job at a time, and once a job is started on a machine, it runs to completion on that machine. Each job i requires time t_i on either machine, where t_i is a positive integer. An ***assignment*** A allocates to each machine a subset of the jobs; formally A is a subset of $\{1, 2, \ldots, n\}$, interpreted as the set of jobs allocated to Machine 1 to execute,

and the remaining jobs $\overline{A} = \{1, 2, \ldots, n\} - A$ are allocated to Machine 2 to execute. Given assignment A, the **load** of each machine is the total amount of time required by the jobs allocated to that machine; i.e., the load of Machine 1 (resp. Machine 2) is $\sum_{i \in A} t_i$ (resp. $\sum_{i \in \overline{A}} t_i$). The maximum of these two quantities is the time by which all n jobs have been processed by the two machines in assignment A, and is called the ***makespan*** of A. We want to find an assignment that has minimum makespan.

Describe an $O(nT)$ dynamic programming algorithm that, given the times $t_1, t_2, \ldots, t_n$ required by the n jobs, determines an assignment with minimum makespan, where $T = \sum_{i=1}^{n} t_i$. Describe your algorithm in pseudocode, explain why it is correct, and analyze its running time.

3.21 Retrofit the algorithm for finding the cost of an optimal BST in Section 3.4 so that it returns an optimal BST, as opposed to its cost.

3.22 Assume that, in addition to the probabilities $P(i)$ for $i \in [1..n]$, we are given probabilities $Q(i)$ for $i \in [0..n]$, where $Q(0)$ is the probability of searching for keys less than key 1, $Q(n)$ is the probability of searching for keys greater than key n, and $Q(i)$, for $i \in [1..n-1]$, is the probability of searching for keys between (but not including) keys i and $i+1$. In other words, Q specifies the probability of searches for keys that are not in the tree. Define the cost of a binary search tree as the expected number of nodes accessed for a (successful or unsuccessful) search, and give a dynamic programming algorithm to construct a minimum cost BST under this cost criterion.

Hint: Add nodes to the tree to represent the between-keys ranges, and note that the number of nodes accessed during a search for a key that is not in the tree is the depth of the node that represents the appropriate between-keys range, and not the depth of that node plus 1.

3.23 Retrofit the knapsack algorithm in Section 3.4 so that it returns an optimal set of items, as opposed to its value.

3.24 Consider the following ***two-knapsack*** variation of the knapsack problem. The input is a sequence of positive integers $w_1, v_1, w_2, v_2, \ldots, w_n, v_n, C_1, C_2$. (Integers w_i and v_i are the weight and value of item i; W_1 and W_2 are the capacities of the two knapsacks.) A ***knapsack pair*** is defined to be a pair (S_1, S_2) such that

$$S_1, S_2 \subseteq \{1, 2, \ldots, n\},\ S_1 \cap S_2 = \varnothing,\ \sum_{i \in S_1} w_i \leq C_1, \text{ and } \sum_{j \in S_2} w_j \leq C_2.$$

The goal is to produce a knapsack pair so that $\sum_{i \in S_1} v_i + \sum_{j \in S_2} v_j$ is as large as possible. (Intuitively, S_1 and S_2 are the sets of items that go into knapsacks 1 and 2, respectively; the total weight of the items in each knapsack must not exceed the knapsack's capacity; and we wish to maximize the total value of the items in the two knapsacks.)

Give a dynamic programming algorithm for this problem. Explain why your algorithm is correct, and analyze its time complexity. Your algorithm should run in time polynomial in n, C_1, and C_2.

3.25 Consider the following problem.

Input: A set A of n non-negative integers (given in an array $A[1..n]$) and a non-negative integer k.
Output: Return **true** if A can be partitioned into two subsets whose sums differ by exactly k, and return **false** otherwise. More precisely, the output is **true** if there is some $B \subseteq A$ such that

$$\sum_{x \in B} x - \sum_{y \in A - B} y = k$$

and the output is **false** if no such subset of A exists. (The sum of "the integers" in the empty set is, by definition, zero.)

Give a pseudopolynomial dynamic programming algorithm for this problem. Explain why your algorithm is correct, and analyze its running time.

Chapter 4

Shortest paths

4.1 Introduction

This chapter marks a transition from studying general algorithm design paradigms, to studying some important specific problems and algorithms to solve them. These problems play a special role because many other problems can be ***reduced*** to them. For example, the ***shortest-paths problem*** that we will study in this chapter is used in mapping applications (such as Googlemaps) to find optimal (shortest, fastest) routes. Less obviously, it can be used to find optimal exchange rates between currencies (as we will discuss at the end of this section) and to determine if a system of "difference constraints" has a solution and if so to find one (as we will see in Section 4.5).

Roughly speaking, in the shortest-paths problem we are given a graph with "weights" associated with its edges, and we seek to find shortest paths, i.e., paths with minimum total weight, connecting nodes. We will study two variants. In the ***single-source shortest-paths*** problem, we are also given a node s of the graph, the ***source***, and we seek shortest paths from s to every node of the graph. In the ***all-pairs shortest-paths*** problem, we seek shortest paths between every pair of nodes. There is a third, natural version, sometimes called the ***single-pair shortest-path*** problem, where we are given both a source node s and a ***target*** node t, and we seek a shortest path from s to t. In general, however, this problem appears to be no easier than the single-source shortest-paths problem, so we will not study it separately (but see Exercise 4.3).

The problems, and the algorithms we will study, apply both to undirected and directed graphs. We will focus on the more general case of directed graphs: For the purpose of finding shortest paths we can think of an undirected graph as a directed one by replacing each undirected edge $\{u, v\}$ by a pair of directed edges (u, v) and (v, u), each having the same weight as the undirected edge $\{u, v\}$.

We now develop some notation and terminology that will be used in our discussion of shortest-paths problems and algorithms. Let $G = (V, E)$ be a directed graph and $\mathbf{wt}\colon E \to \mathbb{Z}$ be an edge-weight function that associates a number[1] with each edge. In many applications

[1] typically an integer but it could be a rational number, or a real (assuming we can do arithmetic with reals).

it is reasonable to assume that edge weights are non-negative; as we will see, for one of the algorithms we will study this is a necessary assumption. Solving the shortest-paths problem when edges can have negative weights, however, is needed in other applications. So, we also need algorithms that can handle the general case. Throughout this chapter, $n = |V|$ denotes the number of nodes of the graph G and $m = |E|$ denotes the number of edges of G.

Let $p = u_1, u_2, \ldots, u_k$ be a path in G. The ***weight*** of p, denoted $\mathbf{wt}(p)$, is the sum of the weights of the edges on p: $\mathbf{wt}(p) = \sum_{i \in [1..k-1]} \mathbf{wt}(u_i, u_{i+1})$. We will use the notation $u \rightsquigarrow v$ to indicate a path that starts at node u and ends at node v. If such a path exists we say that v is ***reachable*** from u. We say that p is a ***shortest*** $u \rightsquigarrow v$ path if (a) p is a $u \rightsquigarrow v$ path, and (b) for every $u \rightsquigarrow v$ path p', $\mathbf{wt}(p) \leq \mathbf{wt}(p')$.

If we allow edges with negative weight, we open the door to the possibility of negative-weight *cycles*. Note that if there is a $u \rightsquigarrow v$ path that includes a negative-weight cycle C, then there is no shortest $u \rightsquigarrow v$ path: The more times a path goes around C, the lower its weight, so we can create paths of arbitrarily small weight. So, algorithms that solve shortest-paths problems when edges may have negative weight must also detect negative-weight cycles (or assume for their correctness that such cycles do not exist).

Let u, v, w be nodes, p_1 be a *shortest* $u \rightsquigarrow w$ path, and p_2, p_3 be *any* $u \rightsquigarrow v$ and $v \rightsquigarrow w$ paths, respectively (see Figure 4.1). Since the path consisting of p_2 followed by p_3 is a $u \rightsquigarrow w$ path, the following ***triangle inequality*** holds:

$$\mathbf{wt}(p_1) \leq \mathbf{wt}(p_2) + \mathbf{wt}(p_3) \tag{4.1}$$

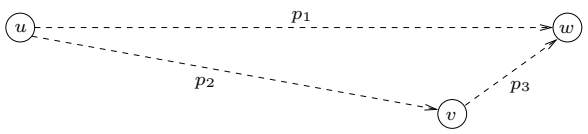


Figure 4.1: The triangle inequality

By a straightforward cut-and-paste argument, shortest paths satisfy the following property:

> If p is a shortest $u \rightsquigarrow w$ path and p_v is the prefix of p up to some node v, *then p_v is a shortest $u \rightsquigarrow v$ path.* (4.2)

That is, *shortest paths exhibit the optimal substructure property*: A shortest $u \rightsquigarrow w$ path p extends shortest $u \rightsquigarrow v$ paths for all intermediate nodes v on p. This makes shortest-paths problems ideal candidates for solutions via greedy and dynamic programming algorithms. Recall that the *longest*-path problem, which superficially looks symmetric to the shortest-path problem does *not* exhibit the optimal substructure property (see page 68).

We now state more precisely the two problems that we study in this chapter.

The single-source shortest-paths problem

Input: A directed graph $G = (V, E)$, a node $s \in V$, and an edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$.
Output: If no negative-weight cycle is reachable from s, the function δ: $V \to \mathbb{Z} \cup \{\infty\}$, where

$$\delta(u) = \begin{cases} \text{weight of a shortest } s \rightsquigarrow u \text{ path}, & \text{if there is a } s \rightsquigarrow u \text{ path in } G \\ \infty, & \text{otherwise.} \end{cases} \tag{4.3}$$

If a negative-weight cycle is reachable from s, the string "negative-weight cycle".

In addition to computing δ we typically also want to compute actual shortest $s \rightsquigarrow u$ paths, not only their weight. To do so it suffices to compute the function $\mathbf{pre}$: $V \to V \cup \{\text{NIL}\}$, where

$$\mathbf{pre}(u) = \begin{cases} s, & \text{if } u = s \\ \text{node before } u \text{ on a shortest } s \rightsquigarrow u \text{ path}, & \text{if } u \neq s \text{ and } G \text{ has a } s \rightsquigarrow u \text{ path} \\ \text{NIL}, & \text{otherwise.} \end{cases} \tag{4.4}$$

By the optimal substructure property of shortest paths (4.2), if an $s \rightsquigarrow u$ path exists, then $u, \mathbf{pre}(u), \mathbf{pre}\big(\mathbf{pre}(u)\big), \ldots, s$ is a shortest $s \rightsquigarrow u$ path in reverse. We will focus on the computation of δ, as the algorithms that compute δ can be easily retrofitted to also compute $\mathbf{pre}$, using ideas we saw in Chapter 3. You can explore these retrofits in the exercises of this chapter.

You already know how to solve the special case of this problem where all edges have the same weight: Simply use breadth-first search, starting at the source s. This works in $\Theta(m+n)$ time, which is the best we can hope for (it is the size of the input, and we have to look at the input to find a shortest path!). In this chapter we will discuss two algorithms that solve the single-source shortest-paths problem when edges may have different weights. Dijkstra's algorithm (Section 4.2) uses the greedy approach to solve the problem in $O\big((m+n)\log n\big)$ time, but works under the assumption that all edge weights are non-negative. The Bellman-Ford algorithm (Section 4.3) uses dynamic programming to solve this problem in $O(mn)$ time, but requires no assumption on the edge weights.

The all-pairs shortest-paths problem

Input: A directed graph $G = (V, E)$ and an edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$.
Output: If the graph contains no negative-weight cycle, the function Δ: $V \times V \to \mathbb{Z} \cup \{\infty\}$, where

$$\Delta(u, v) = \begin{cases} \text{weight of a shortest } u \rightsquigarrow v \text{ path}, & \text{if there is a } u \rightsquigarrow v \text{ path in } G \\ \infty, & \text{otherwise.} \end{cases} \tag{4.5}$$

If the graph contains a negative-weight cycle, the string "negative-weight cycle".

We will discuss two algorithms to solve this problem, both of which allow edge weights to be arbitrary integers. The Floyd-Warshall algorithm (Section 4.4) is a dynamic programming algorithm that solves the all-pairs shortest-paths problem in $O(n^3)$ time. Johnson's algorithm (Section 4.5) solves this problem in $O\big((m+n)n\log n\big)$ time. It uses both Dijkstra's and the Bellman-Ford algorithm as subroutines.

A reduction to the shortest-paths problem: currency trading

We now describe a problem that can be reduced to the single-source shortest paths problem, where the reduction illustrates the need to handle negative-weight edges. Suppose that the exchange rate to convert Canadian dollars to Euros is r, and the exchange rate to convert Euros to Canadian dollars in r'. Presumably $r' = 1/r$: If $r' > 1/r$, then starting with a positive amount of Canadian dollars you can make an arbitrary amount of money by converting them to Euros and then back to Canadian dollars repeatedly. If $r' < 1/r$, you can achieve the same goal starting with a positive amount of Euros.

The world currency market has many currencies, so you can convert Canadian dollars to Euros in many different ways. You can do so directly, or you can first convert your Canadian dollars to Chinese yuan, then the yuan to euros, then the euros to Indian rupees, then the rupees to Japanese yen, the yen to Swedish krona, and the krona to Euros.

Imagine now a weighted directed graph where the nodes are currencies, and there is an edge from currency c to currency c' with weight r if and only if we can exchange one unit of c to r units of c'. Note that r is a positive number, possibly a fraction. A particular way of converting currency c to currency c' corresponds to a path $c_1, c_2, \ldots, c_k$ on this graph, where $c_1 = c$ and $c_k = c'$. The effective exchange rate from c to c' along this path is the product $\Pi_{i=1}^{k-1} r_i = r_1 \cdot r_2 \cdot \ldots \cdot r_{k-1}$, where r_i is the exchange rate for converting currency c_i to currency c_{i+1}. To maximize the amount of money we receive for converting c to c', we want to find a path that *maximizes* the product of the exchange rates on the edges of the path.

This is beginning to look like a shortest-paths problem, except for two things: First, shortest paths is a minimization, not maximization, problem. But this is easy to fix: Maximizing $\Pi_{i=1}^{k-1} r_i$ is the same as minimizing $\Pi_{i=1}^{k-1} 1/r_i$. Second, in shortest paths we want to minimize the *sum* of the weights of edges on a path, not their *product*. But we can turn a product into a sum by taking logarithms: Minimizing $\Pi_{i=1}^{k-1} 1/r_i$ is the same as minimizing its logarithm, and $\log\bigl(\Pi_{i=1}^{k-1} 1/r_i\bigr) = \Sigma_{i=1}^{k-1} \log 1/r_i$. Thus, a path that maximizes $\Pi_{i=1}^{k-1} r_i$ in the currency graph is a path that minimizes $\Sigma_{i=1}^{k-1} \log 1/r_i$. And so, to find the best way of converting currency c to currency c', we look for a shortest $c \rightsquigarrow c'$ path in the currency graph where edge (a, b) has weight $\log 1/r$, where r is the exchange rate of currency a to currency b.

Note, however, that

$$\log 1/r \begin{cases} < 0, & \text{if } r > 1 \\ = 0, & \text{if } r = 1 \\ > 0, & \text{if } 0 < r < 1. \end{cases}$$

This means that the graph with nodes corresponding to the currencies and edge weights corresponding to the logarithms of the inverses of the exchange rates may contain edges with weight that is negative, zero, or positive. Thus, to solve a shortest path problem in this graph we must use an algorithm that works if edges can have negative weights: Dijkstra's algorithm is not appropriate for this application but the Bellman-Ford algorithm is.

What about negative-weight cycles? A negative-weight cycle corresponds to a sequence of currency exchanges in which starting with x units of some currency we wind up with more than x units of the same currency! Traders call this an ***arbitrage opportunity***. It is a way to

make money without engaging in any productive activity — just by moving money around. In the world of trade, graphs such as the one described above contain nodes for all exchangeable goods, not just currencies, and there are actually people who look for negative-weight cycles in such graphs — arbitrage opportunities that they hope to exploit.

Of course in real life exchange rates are dynamic and negative-weight cycles that arise last only for very short periods of time. If you are in the middle of a long sequence of trades hoping to exploit what now appears to be an arbitrage opportunity, exchange rates may change while you are executing your trades and you may end up losing money. There is no free lunch: unproductive activities that supposedly generate money can blow up and cause a great deal of damage — and, unfortunately, not always to the individuals or firms involved. But we have now entered the more important realm of ethics, and shortest-path algorithms have nothing to teach us about that.

4.2 Dijkstra's algorithm

Dijkstra's algorithm is a classic and very important algorithm for the single-source shortest-paths problem. It works under the assumption that *the graph has no negative-weight edges*. Among many other uses, it lies at the heart of mapping applications such as Google maps.[2]

Recall that the input is a directed graph $G = (V, E)$, the source node $s \in V$, and the edge-weight function $\mathbf{wt}$, under the assumption that for all $e \in E$, $\mathbf{wt}(e) \geq 0$. The goal of the algorithm is to compute the function $\delta\colon V \to \mathbb{Z} \cup \{\infty\}$, where $\delta(u) =$ weight of a shortest $s \rightsquigarrow u$ path (or ∞, if no such path exists). We begin with some terminology that will be useful in explaining how (and why) the algorithm works: If R is a set of nodes, we say that a $s \rightsquigarrow u$ is an ***R-path to*** u if all nodes on the path except possibly the last are in R. Note that if $R = \varnothing$, there is only one possible R-path, namely the path consisting of the node s and this path has weight 0.

Dijkstra's algorithm maintains two pieces of information:

- A set of nodes R that constitute the "explored" part of the graph, and
- a function $d\colon V \to \mathbb{N}$ such that for every node v:

$$d(v) = \begin{cases} \text{minimum weight of an } \underline{R\text{-path}} \text{ to } v, & \text{if an } R\text{-path to } v \text{ exists} \\ \infty, & \text{otherwise.} \end{cases} \tag{4.6}$$

$$\text{if } v \in R, \text{ then } d(v) = \delta(v). \tag{4.7}$$

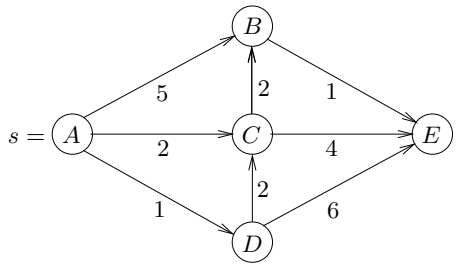
(4.6) implies that, for every node v, $d(v) \geq \delta(v)$; that is, the value of $d(v)$ is an upper bound on the quantity $\delta(v)$ we want to compute. (4.7) implies that *for those nodes that happen to be in the set R*, $d(v)$ is exactly the quantity $\delta(v)$ we want to compute.

[2]These applications also incorporate other ideas to find optimal routes faster in practice. For example, they use the so-called ***A^*-search algorithm***, which enhances Dijkstra's algorithm with application-specific, heuristic information to avoid searching parts of the graph that can be ignored in the search for a shortest path, and they use the "bidirectional" version of Dijkstra's algorithm (see Exercise 4.3).

The algorithm starts with $R = \varnothing$ and then greedily expands R one node at a time, by including into it a node u not already in R *that has minimum d-value* and then updates the d-values of the nodes adjacent to u to account for fact that there may now exist shorter R-paths to these nodes, going through u. Thus, when R has been expanded to include all nodes, by property (4.7), $d(v) = \delta(v)$ for all nodes v. Shown below is pseudocode for Dijkstra's algorithm.

DIJKSTRA$(G, s, \mathbf{wt})$
1 $R := \varnothing$
2 $d(s) := 0$
3 **for each** $v \in V - \{s\}$ **do** $d(v) := \infty$
4 **while** $R \neq V$ **do**
5 $u :=$ node not in R with minimum d-value (i.e., $u \in V - R$ and $\forall\, u' \in V - R, d(u) \leq d(u')$)
6 $R := R \cup \{u\}$
7 **for each** $v \in V$ such that $(u, v) \in E$ **do**
8 **if** $d(u) + \mathbf{wt}(u, v) < d(v)$ **then** $d(v) := d(u) + \mathbf{wt}(u, v)$
9 **return** d

Example 4.1 The table on the right side in Figure 4.2 gives the values of R and d in each iteration of Dijkstra's algorithm when the input is the graph on the left side in that figure, with start node $s = A$ and edge weights as shown. The symbol $-$ under the column of some node indicates that the node is not yet in R and ✓ indicates that it is. **End of Example 4.1**



iteration		A	B	C	D	E
0 (init)	R	$-$	$-$	$-$	$-$	$-$
	d	0	∞	∞	∞	∞
1	R	✓	$-$	$-$	$-$	$-$
	d	0	5	2	1	∞
2	R	✓	$-$	$-$	✓	$-$
	d	0	5	2	1	6
3	R	✓	$-$	✓	✓	$-$
	d	0	4	2	1	6
4	R	✓	✓	✓	✓	$-$
	d	0	4	2	1	5
5	R	✓	✓	✓	✓	✓
	d	0	4	2	1	5

Figure 4.2: Example execution of Dijkstra's algorithm

A useful way of thinking about how the algorithm works is that it maintains invariant (4.6): The d-value of each node u is always the minimum weight of an $s \rightsquigarrow u$ path *restricted to pass only through nodes in R before reaching u*. By expanding the set R in each iteration, this restriction is *relaxed*, since there are now more R-paths to the nodes. For this reason, each

iteration of the loop in the algorithm is sometimes referred to as a *relaxation step*. At the end, when $R = V$, the restriction is lifted completely.[3]

We now prove the correctness of the algorithm via a sequence of lemmas. The first lemma states that the d-value of every node does not increase in time. (Recall our convention that X_i denotes the value of a variable X at the end of iteration i of the loop, and X_0 denotes the value of X just before entering the loop.)

Lemma 4.1 *For every node v and iterations i, j, if $i \leq j$ then $d_i(v) \geq d_j(v)$.*

PROOF. After being initialized (in line 2 or 3), the value of $d(v)$ is changed only in line 8, where it is obviously assigned a smaller value than before. □

Lemma 4.2 *If node u is added to R in iteration i the value of $d(u)$ does not change in iteration i.*

PROOF. Suppose for contradiction that u is added to R in iteration i and $d(u)$ changes in iteration i. Thus, by the algorithm, (u, u) is an edge and $d_{i-1}(u) + \mathbf{wt}(u, u) < d_{i-1}(u)$. But then $\mathbf{wt}(u, u) < 0$, contradicting the assumption that weight of every edge is non-negative. □

Lemma 4.3 *For every node v and iteration i, if $d_i(v) = k \neq \infty$ then there is an R_i-path to v of weight k.*

PROOF. By induction on the iteration number $i \geq 0$. For the basis $i = 0$, i.e., just before we start the while loop, we have $R_0 = \varnothing$, $d_0(s) = 0$, and $d_0(u) = \infty$ for every node $u \neq s$. It is true that there is an R_0-path to s of weight 0.

For the induction step, let $i > 0$ and suppose the lemma holds at the end of iteration $i-1$. Let u be the node added to R in iteration i and consider any node v. If $d(v)$ does not change in iteration i, the lemma holds after iteration i by induction hypothesis. If $d(v)$ changes in iteration i and since (by Lemma 4.2) $d_{i-1}(u) = d_i(u)$, by the algorithm $d_i(v) = d_{i-1}(u) + \mathbf{wt}(u, v)$ and $d_{i-1}(u) \neq \infty$. By the induction hypothesis, there is an R_{i-1}-path to u of weight $d_{i-1}(u)$, say path p. Since $R_i = R_{i-1} \cup \{u\}$, path p followed by the edge (u, v) is an R_i-path to v, whose weight is $\mathbf{wt}(p) + \mathbf{wt}(u, v) = d_{i-1}(u) + \mathbf{wt}(u, v) = d_i(u) + \mathbf{wt}(u, v)$. So, the lemma holds after iteration i. □

[3]Dijkstra's algorithm is an instance of a class of algorithms known as ***relaxation algorithms*** that compute their output by starting with an approximation of the goal based on some restriction and obtaining better and better approximations of the goal by progressively relaxing the restriction until the restriction is completely lifted and the goal is reached. In the case of Dijkstra's algorithm $d(u)$ is the current approximation of the goal, which is $\delta(u)$. The Bellman-Ford and the Floyd-Warshall algorithms that we will study in the next two sections are also instances of relaxation algorithms.

Lemma 4.4 *For every node u, if u is added to R in iteration i and $d_i(u) = \infty$ then there is no $s \rightsquigarrow u$ path.*

PROOF. Suppose, for contradiction, that some node u is added to R in iteration i and $d_i(u) = \infty$ but there is an $s \rightsquigarrow u$ path. Without loss of generality, assume that i is the earliest iteration in which this happens. Since s is added to R in iteration 1 and $d_1(s) \neq \infty$, $i > 1$. So $s \in R_{i-1}$ and $u \notin R_{i-1}$ (since u is added to R in iteration i). Thus, there is an edge (x, y) on the $s \rightsquigarrow u$ path with $x \in R_{i-1}$ and $y \notin R_{i-1}$. Let $j < i$ be the iteration in which x was added to R; by the definition of i, $d_j(x) \neq \infty$ and so, by line 8, $d_j(y) \neq \infty$. By Lemma 4.1, $d_{i-1}(y) \neq \infty$. The facts that (a) $y \notin R_{i-1}$ and (b) $d_{i-1}(y) \neq \infty$ contradict that in iteration i the algorithm added to R the node u with $d_{i-1}(u) = \infty$. □

Lemma 4.5 *For every node u and every iteration $i \geq 1$, if u is added to R in iteration i, then $d_i(u) = \delta(u)$.*

PROOF. By complete induction on i. Suppose u is the node added to R in iteration i, and suppose the lemma holds for all nodes added to R before iteration i.

If $i = 1$, the lemma holds since (by the initialization in lines 2–3) the node added to R in iteration 1 is s and $d_1(s) = 0 = \delta(s)$.

If $i > 1$, the lemma holds by Lemma 4.4 if $d_i(u) = \infty$. So, suppose $d_i(u) \neq \infty$. Then by Lemma 4.3 there is an R_i-path to u of weight $d_i(u)$; therefore $d_i(u) \geq \delta(u)$. We will now show that $d_i(u) \leq \delta(u)$, proving that $d_i(u) = \delta(u)$, as wanted. Since there is an R_{i-1}-path to u, there is also a minimum weight $s \rightsquigarrow u$ path, say p (refer to Figure 4.3).

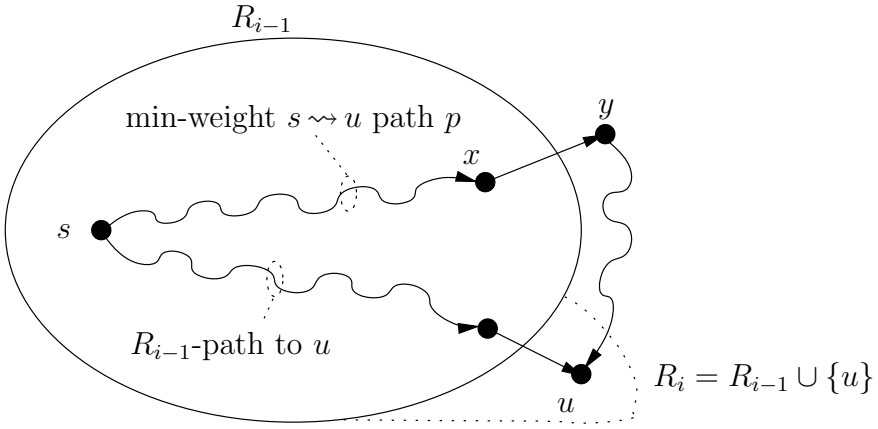


Figure 4.3: Illustration for the proof of Lemma 4.5

Since u is added to R in iteration i, $u \notin R_{i-1}$ (recall that $i > 1$, so iteration $i - 1$ exists, and R_{i-1} is well defined). Since $s \in R_{i-1}$ and $u \notin R_{i-1}$, p contains an edge (x, y) such that $x \in R_{i-1}$ and $y \notin R_{i-1}$ (it is possible that $y = u$). Let j be the iteration is which x was added

to R, so $j \leq i-1$. Since p is a minimum-weight $s \rightsquigarrow u$ path, the prefix p_x of p up to node x is a minimum-weight $s \rightsquigarrow x$ path, i.e., $\mathbf{wt}(p_x) = \delta(x)$. We have:

$$\begin{aligned}
d_i(u) &= d_{i-1}(u) && \text{by Lemma 4.2}\\
&\leq d_{i-1}(y) && \text{by definition of } u, \text{ since } u, y \notin R_{i-1}\\
&\leq d_j(y) && \text{by Lemma 4.1, since } j \leq i-1\\
&\leq d_j(x) + \mathbf{wt}(x,y) && \text{by Lemma 4.2 and line 8, since } x \text{ is added to } R \text{ in iteration } j\\
&= \delta(x) + \mathbf{wt}(x,y) && \text{by the induction hypothesis, since } j < i\\
&= \mathbf{wt}(p_x) + \mathbf{wt}(x,y) && \text{since } \mathbf{wt}(p_x) = \delta(x), \text{ as argued above}\\
&\leq \mathbf{wt}(p) && \text{since all edges have non-negative weight}\\
&= \delta(u) && \text{by definition of } p
\end{aligned}$$

So, $d_i(u) \leq \delta(u)$, as needed to complete the proof that $d_i(u) = \delta(u)$. □

The algorithm terminates, since one node is added to R in each iteration. The next theorem states that when the algorithm terminates, it has computed the weight of a minimum-weight $s \rightarrow u$ path, for every node u.

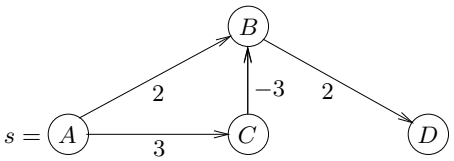
Theorem 4.6 *When the algorithm terminates, for every node u, $d(u) = \delta(u)$.*

PROOF. By Lemma 4.5, when u is added to R, $d(u) = \delta(u)$. By Lemma 4.1, $d(u)$ cannot later be assigned a larger value, and by Lemma 4.3 it cannot later be assigned a smaller value. So when the algorithm terminates, $d(u) = \delta(u)$. □

Recall that we require the edge weights to be non-negative numbers. This assumption is used in the proofs of Lemmas 4.2 and 4.5. This is not a shortcoming of these proofs: Dijkstra's algorithm may not work if some edges have negative weights. A counterexample that demonstrates this is shown in Figure 4.4. The algorithm computes the minimum weight of an $A \rightsquigarrow D$ path to be 4, when in fact there is a path of weight 2, namely A, C, B, D. Note that in this example the algorithm computes the correct minimum weight of an $A \rightsquigarrow B$ path in iteration 3, one iteration *after* B was added to R.

Running time of Dijkstra's algorithm. Let n be the number of nodes and m be the number of edges in the graph. The running time of Dijkstra's algorithm depends on the data structure used to store $d(u)$.

In the simplest implementation, we store d in an array of n elements, one per node, in no particular order. The initialization of R and d takes $O(n)$ time. The while loop is executed n times. Each iteration of the loop takes $O(n)$ time (to find the minimum element in array d of a node that is not in R, and to update the relevant entries of d). So the loop in total takes $O(n^2)$ time. This implementation then takes $O(n) + O(n^2) = O(n^2)$ time.



iteration		A	B	C	D
0 (init)	R	–	–	–	–
	d	0	∞	∞	∞
1	R	✓	–	–	–
	d	0	2	3	∞
2	R	✓	✓	–	–
	d	0	2	3	4
3	R	✓	✓	✓	–
	d	0	0	3	4
4	R	✓	✓	✓	✓
	d	0	0	3	4

Figure 4.4: Example demonstrating that Dijkstra's algorithm is incorrect if edges may have negative weight

We can also use a heap to store the d-values of the nodes that are not in R. Thus we can find a node not in R with the minimum d-value by performing an ExtractMin operation, which takes $O(\log n)$ time; and we can update the value of d for a node by performing a ChangeKey operation, which also takes $O(\log n)$ time. We perform n ExtractMin operations, one in each iteration of the while loop. We perform at most m ChangeKey operations: at most once for each edge (u, v), in the iteration of the while loop in which u is added to R. In the initialization we must also perform a BuildHeap operation, to create the initial heap; this takes $O(n)$ time. Thus, the total time required to process all these operations is $O(n)+O(n \log n)+O(m \log n) = O\big((m+n) \log n\big)$. If we assume that there is a path from s to each node, then $m \geq n-1$, and so the above expression simplifies to $O(m \log n)$.

If the graph is "dense", i.e., it has an edge between (almost) every two nodes, then $m = \Theta(n^2)$, and in that case the simple array implementation is actually faster! However, in practice often the graph is "sparse" — typically, each node has a constant or perhaps a logarithmic number of neighbours, so $m = \Theta(n)$ or $m = \Theta(n \log n)$. In this case, the heap implementation of Dijkstra's algorithm is substantially faster.

4.3 The Bellman-Ford algorithm

In this section we will discuss the Bellman-Ford algorithm, another classic and important algorithm for the single-source shortest-paths problem, which works even if edges can have negative weights. Its disadvantage compared to Dijkstra's algorithm is that it has an increased running time. It is based on dynamic programming.

As with Dijkstra's algorithm, the input is a directed graph $G = (V, E)$, the source node $s \in V$, and the edge-weight function $\mathbf{wt}$; but we no longer require that $\mathbf{wt}(e) \geq 0$ for all edges $e \in E$. For its output, the algorithm first determines if there is a negative-weight cycle (whose nodes are) reachable from s. If so the algorithm simply outputs the string "negative-

weight cycle", since shortest paths from s do not exist in this case. If no negative way cycle is reachable from s, the algorithm outputs the function δ, where $\delta(u)$ is the weight of a shortest $s \rightsquigarrow u$ path (or ∞, if no such path exists).

Before describing the algorithm we introduce some terminology and useful facts. For any $k \in \mathbb{N}$, we say that a path p is a ***k-path*** if it has *at most* k edges. We say that p is a ***shortest k-path to node*** u if (a) p is an $s \rightsquigarrow u$ k-path, and (b) for every $s \rightsquigarrow u$ k-path p', $\mathbf{wt}(p) \leq \mathbf{wt}(p')$.

The following lemma lays out the key facts that the algorithm exploits to detect if the graph as a negative-weight cycle reachable from s.

Lemma 4.7 *For every $k \geq n$, if <u>every</u> shortest $s \rightsquigarrow u$ k-path has <u>exactly</u> k edges, then every such path p:*
(a) contains a cycle; and
(b) <u>every</u> cycle that p contains has negative weight.

PROOF. Let $k \geq n$ and suppose that every shortest $s \rightsquigarrow u$ k-path has exactly k edges. Let p be any shortest $s \rightsquigarrow u$ k-path. Since $k \geq n$, p has at least $n+1$ nodes. By the pigeonhole principle some node is repeated on p, which means that p contains a cycle, proving (a).

For (b) let C be any cycle on p and suppose, for contradiction, that C has non-negative weight. If we remove from p the edges on the cycle C, we obtain an $s \rightsquigarrow u$ path p' that has fewer than k edges and weight at most $\mathbf{wt}(p)$, which contradicts that every shortest $s \rightsquigarrow u$ k-path has exactly k edges. Thus C has negative weight, as wanted. □

We are now ready to present the Bellman-Ford algorithm. We have already mentioned the optimal substructure of the shortest path problem (4.2): A shortest $s \rightsquigarrow u$ path extends a shortest $s \rightsquigarrow v$ path for some predecessor v of u. We don't know which predecessor, so we will compute the shortest $s \rightsquigarrow v$ path for all predecessors v of u and pick the one that minimizes the weight of the path to u. And how do we compute the shortest path to each predecessor of u? Recursively, hopefully. But what grounds this recursion? In what sense is the problem getting smaller? You can probably see the problem: u itself could be a predecessor of one or more of its predecessors — directly (via edges (v,u) and (u,v) forming a cycle), or indirectly (via a longer cycle that includes u and v).

As is often the case with dynamic programming we solve this problem by adding a parameter to the subproblems: the maximum number of edges we can use in the paths from s to other nodes. So we define, for each $u \in V$ and $k \in \mathbb{Z}$:

$$S(u,k) = \begin{cases} \text{the weight of a shortest } s \rightsquigarrow u \text{ } \underline{k\text{-path}}, & \text{if an } s \rightsquigarrow u \text{ } k\text{-path exists} \\ \infty, & \text{otherwise} \end{cases} \qquad (*)$$

Note that, by definition, $S(u,k) \leq S(u,k-1)$: the parameter k acts as a constraint (on the number of edges a path can use); the larger the k, the less restrictive the constraint. The Bellman-Ford algorithm is another example of a relaxation algorithm: it gradually relaxes the constraint by computing $S(u,k)$ for larger and larger values of k. Although formally we define

$S(u,k)$ for *all* $k \in \mathbb{N}$, as we will see shortly, the values for just $k \in [0..n]$ suffice to determine (a) whether a negative-weight cycle is reachable from s, and (b) if not, the weight of a shortest $s \rightsquigarrow u$ path (without any restriction on the number of edges it has), for each $u \in V$.

We claim that the following is a recursive formula for $S(u,k)$ as defined in (*):

$$S(u,k) = \begin{cases} 0, & \text{if } k = 0 \text{ and } u = s \\ \infty, & \text{if } k = 0 \text{ and } u \neq s \\ \min\big(\{\underbrace{S(u,k{-}1)}_{\text{case (a) below}}\} \cup \underbrace{\{S(v,k{-}1){+}\mathbf{wt}(v,u)\colon (v,u) \in E\}}_{\text{case (b) below}}\big), & \text{otherwise} \end{cases} \tag{†}$$

The base cases are obvious. For the inductive case, consider the weight of a shortest $s \rightsquigarrow u$ k-path. There are two possibilities:

(a) There is a shortest $s \rightsquigarrow u$ k-path p that has at most $k-1$ edges; i.e., p is a $(k{-}1)$-path. So in this case, $S(u,k) = S(u,k{-}1)$.

(b) Every shortest $s \rightsquigarrow u$ k-path p has exactly k edges. In this case, p consists of a prefix p' that is an $s \rightsquigarrow v$ $(k{-}1)$-path with exactly $k{-}1$ edges, for *some* node v, followed by the edge (v,u). By a straightforward cut-and-paste argument p' is a *shortest* $s \rightsquigarrow v$ $(k{-}1)$-path. So in this case, $S(u,k) = \mathbf{wt}(p') + \mathbf{wt}(v,u) = S(v,k{-}1) + \mathbf{wt}(v,u)$, for some predecessor v of u.

By (a) and (b), $S(u,k) = \min\big(\{S(u,k{-}1)\} \cup \{S(v,k{-}1) + \mathbf{wt}(v,u)\colon (v,u) \in E\}\big)$, as wanted.

We will refer to $S(u,k)$ as the ***S-label of node u in phase k***. The next lemma states that if the S-labels of *all* nodes do not change between two successive phases, they will not change in the next one, too.

Lemma 4.8 *If, for all $u \in V$, $S(u,k) = S(u,k{-}1)$ then, for all $u \in V$, $S(u,k{+}1) = S(u,k)$.*

Proof. Assume that for all $u \in V$, $S(u,k) = S(u,k-1)$. Then for any $u \in V$ we have

$$\begin{aligned} S(u,k{+}1) &= \min\big(\{S(u,k)\} \cup \{S(v,k) + \mathbf{wt}(v,u)\colon (v,u) \in E\}\big) && \text{[by (†)]} \\ &= \min\big(\{S(u,k{-}1)\} \cup \{S(v,k{-}1) + \mathbf{wt}(v,u)\colon (v,u) \in E\}\big) && \text{[by assumption]} \\ &= S(u,k). && \text{[by (†)]} \end{aligned}$$

□

From this lemma and induction we have that if the S-labels of *all* nodes do not change between two successive phases, they will never change again and these unchanging values are therefore the weights of shortest paths to each node.

Lemma 4.9 *If, for all $u \in V$, $S(u,k) = S(u,k{-}1)$ then:*
(a) for all $u \in V$ and all $\ell \geq k$, $S(u,\ell) = S(u,k{-}1)$;
(b) for all $u \in V$, $S(u,k{-}1)$ is the weight of a shortest $s \rightsquigarrow u$ path.

PROOF. Part (a) follows from Lemma 4.8 and straightforward induction. Part (b) follows from part (a) and the definition (*) of S. □

This lemma gives us one way of ending the computation of the S-labels for increasing phases: If in two successive phases these labels do not change for *any* node, then we can stop: we have found the weights of shortest $s \rightsquigarrow u$ paths for all nodes u. If, however, the graph contains a negative-weight cycle reachable from s, the S-labels will never stop changing; as we have seen, in that case k-paths from s get shorter and shorter, without bound, as k increases. Fortunately, Lemma 4.7 give us a way to determine if the graph contains a negative-weight cycle reachable from s: If in phase n we have that, for some $u \in V$, $S(u,n) \neq S(u,n-1)$, then every shortest $s \rightsquigarrow u$ n-path has exactly n edges (if not, $S(u,n) = S(u,n-1)$), and so by the lemma any such path contains cycles and all its cycles have negative weight.

So the Bellman-Ford algorithm is to compute S-labels as per (†) for phases $0, 1, 2, \ldots$ until one of two things happens:

(a) there has been no change in any label in two successive phases, in which case we have found the weights of shortest paths to all nodes; or

(b) we have completed the computation of phase n labels and in each phase 1 through n the label of some node has changed, in which case we have determined that there is a negative-weight cycle reachable from s.

Below is pseudocode for the algorithm.

```
   BellmanFord(G, s, wt)
 1 S(s,0) := 0
 2 for each u ∈ V − {s} do S(u,0) := ∞
 3 for k := 1 to n do
 4     S_changed := false
 5     for each u ∈ V do
 6         S(u,k) := S(u,k−1)
 7         for each v ∈ V such that (v,u) ∈ E do
 8             if S(u,k) > S(v,k−1) + wt(v,u) then
 9                 S(u,k) := S(v,k−1) + wt(v,u)
10                 S_changed := true
11     if S_changed = false then break
12 if S_changed = false then
13     for each u ∈ V do δ(u) := S(u,k)
14     return δ
15 else return "negative-weight cycle"
```

The *S_changed* flag is true at the end of an iteration of the main loop (lines 3-11) if and only if some S-label changed in that iteration (lines 9-10). So, if no S-label changes in an iteration (i.e., the S-labels are the same in two successive phases) the algorithm breaks out of the main loop in line 11 and returns the weight of a shortest $s \rightsquigarrow u$ path, for each u, on line 14. If some S-label changes in each iteration $1..n$, the algorithm completes the main loop and returns the string "negative-weight cycle" (line 15).

We have effectively justified the correctness of this algorithm in the process of describing

it, but let's do it in a more systematic way now. First, by (†) (and straightforward induction) it is clear that by the end of iteration k, $S(u,k)$ is the weight of a shortest $s \rightsquigarrow u$ k-path (i.e., the quantity defined in (*)). There are two cases.

CASE 1. No negative-weight cycle is reachable from s. We claim that

$$\text{there is some } k \in [1..n] \text{ such that for all } u \in V,\ S(u,k) = S(u,k-1). \tag{4.8}$$

Suppose, for contradiction, that (4.8) is false. Thus, for all $k \in [1..n]$ there is some $u \in V$ such that $S(u,k) \neq S(u,k-1)$. In particular, this is true for $k = n$. So, for some $u \in V$, $S(u,n) \neq S(u,n-1)$. This means that every shortest $s \rightsquigarrow u$ n-path p has exactly n edges. Then, by Lemma 4.7, there is a negative-weight cycle reachable from s, contradicting the hypothesis of the case.

Let k^* be the minimum value of k for which (4.8) holds. Then by Lemma 4.9 the S-labels do not change after iteration k^*-1 and $S(u,k^*-1)$ is the weight of a shortest $s \rightsquigarrow u$ path. The loop terminates after iteration k^* (since *S_changed* is false at the end of iteration k^*) and in line 14 the algorithm returns the weight of a shortest $s \rightsquigarrow u$ path for each u, as wanted in Case 1.

CASE 2. Some negative-weight cycle is reachable from s. Then

$$\text{for every } k \in [1..n] \text{ there is some } u \in V \text{ such that } S(u,k) \neq S(u,k-1). \tag{4.9}$$

Otherwise, by Lemma 4.9, for each node u reachable from s there would be a shortest $s \rightsquigarrow u$ path, contradicting the hypothesis of this case. Therefore, *S_changed* is true at the end of each iteration of the main loop (lines 3-11). So, the conditional on line 12 is false, and the algorithm returns "negative-weight-cycle" in line 15, as wanted in Case 2.

The running time of the algorithm is $\Theta(mn)$, if the graph is given in adjacency list form. We leave it as an exercise to work out exactly how to do this.

It would appear that the algorithm requires $\Theta(n^2)$ space, since it computes the two-dimensional table $S(u,k)$, $u \in V$ and $k \in [0..n]$. Perhaps surprisingly, we don't need the second dimension: The algorithm works correctly if we use a one-dimensional table $S(u)$, ignoring the second dimension! This appears dangerous: If there is an edge (v,u) and in some iteration we update $S(v)$ before $S(u)$, the value of $S(v)$ in lines 8 and 9 is not necessarily the same as the value of $S(v,k-1)$ in the version where the second dimension of S is used. Therefore, it is no longer true that at the end of iteration k, $S(u,k)$ is the weight of a shortest $s \rightsquigarrow u$ k-path; it may be smaller. Exercise 4.4 invites you to prove that the algorithm that uses a one-dimensional table for S is still correct. So, the Bellman-Ford algorithm can be implemented using only $\Theta(n)$ space.

The algorithm can be retrofitted to output shortest paths from s to each node u (rather than just their weights) when no negative-weight cycle is reachable from s; and to output *some* negative-weight cycle reachable from s when one exists — see Exercises 4.5 and 4.6.

4.4 The Floyd-Warshall algorithm

We now turn our attention to algorithms for the all-pairs shortest-paths problem. Recall that in this problem the input is a directed graph $G = (V, E)$ and edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$. The goal is to detect whether the graph has a negative weight cycle and, if not, to compute the function Δ: $V \times V \to \mathbb{Z} \cup \{\infty\}$, where $\Delta(u, v) =$ weight of a shortest $u \rightsquigarrow v$ path (or ∞ if no such path exists).

A straightforward way to solve this problem is to run the Bellman-Ford algorithm n times, once with each node as the source. This results in a $\Theta(mn^2)$ algorithm. Another way is to use the idea behind the Bellman-Ford algorithm and define, for all nodes $u, v \in V$ and $k \in [0..n]$:

$$S(u, v, k) = \begin{cases} \text{the weight of a shortest } u \rightsquigarrow v \text{ } \underline{k\text{-path}}, & \text{if a } u \rightsquigarrow v \text{ } k\text{-path exists} \\ \infty, & \text{otherwise} \end{cases}$$

I invite you to develop a recursive formula to compute $S(u, v, k)$, along the lines (and on the basis of similar reasoning) as in our discussion of the Bellman-Ford algorithm. When you do this and write pseudocode to compute $S(u, v, k)$ for every $u, v \in V$ and $k \in [0..n]$, you will discover that the running time of the resulting algorithm is again $\Theta(mn^2)$.

In this section we discuss the Floyd-Warshall algorithm, which solves the all-pairs shortest-paths problem in $\Theta(n^3)$ time. For now we will assume that *the graph G has no negative-weight cycles*, though it may have negative weight edges. Later we will discuss what happens if there are negative-weight cycles.

The Floyd-Warshall algorithm is a dynamic programming algorithm that defines subproblems by placing a restriction on the paths and then gradually relaxing that restriction. Instead of restricting the paths on the basis of how many edges they can use, as in the Bellman-Ford algorithm, it restricts paths on the basis of *which set of nodes they can go through*.

Without loss of generality, we will assume that the set of nodes is $[1..n]$. We say that a path $p = i_1, i_2, \ldots, i_m$ is ***an*** $i \overset{k}{\rightsquigarrow} j$ ***path***, where $i, j \in [1..n]$ and $k \in [0..n]$, if $i_1 = i$, $i_m = j$, and for every $t \in [2..m-1]$, $i_t \in [1..k]$. That is, an $i \overset{k}{\rightsquigarrow} j$ path starts at node i, ends at node j, and every *intermediate* node (i.e., every node other than the first and last on the path) is one of the nodes $[1..k]$. Note that the first and last nodes of the path may or may not be in $[1..k]$, and that we do not require all of $[1..k]$ to appear as intermediate nodes on the path. We just require that the path not "stray outside" this set, except that it may start or end outside of it. Do not confuse this restriction with the restriction of the Bellman-Ford algorithm: In that case, k placed a limit on the *number of edges* of a path; now k places a limit on the *set of intermediate nodes* of a path.

We now define $A(i, j, k)$ for all $i, j \in [1..n]$ and $k \in [0..n]$:

$$A(i, j, k) = \begin{cases} \text{weight of a shortest } i \overset{k}{\rightsquigarrow} j \text{ path}, & \text{if an } i \overset{k}{\rightsquigarrow} j \text{ path exists} \\ \infty, & \text{otherwise} \end{cases} \qquad (*)$$

Note that an $i \overset{k}{\rightsquigarrow} j$ path might not exist either because no $i \rightsquigarrow j$ path exists, or because every such path uses intermediate nodes numbered higher than k. When $k = n$, the limitation on

the intermediate nodes is gone: the path is allowed to go through any nodes. So, our desired output is simply $A(i,j,n)$ for every $i,j \in [1..n]$.

Following is a recursive formula that computes $A(i,j,k)$.

$$A(i,j,k) = \begin{cases} 0, & \text{if } k=0,\ i=j \\ \mathbf{wt}(i,j), & \text{if } k=0,\ i\neq j,\ (i,j)\in E \\ \infty, & \text{if } k=0,\ i\neq j,\ (i,j)\notin E \\ \min\big(A(i,j,k{-}1), A(i,k,k{-}1){+}A(k,j,k{-}1)\big), & \text{otherwise (i.e., if } k>0) \end{cases} \quad (\dagger)$$

We show that this formula correctly computes $A(i,j,k)$ as defined by (*) by induction on k.

For the base case $k=0$, if an $i \overset{0}{\rightsquigarrow} j$ path p exists, it cannot have *any* intermediate nodes. So p consists of a single node i, in which case $\mathbf{wt}(p)=0$; or of two nodes i and j connected by the edge (i,j), in which case $\mathbf{wt}(p)=\mathbf{wt}(i,j)$. Thus, if an $i \overset{0}{\rightsquigarrow} j$ path exists, the recursive formula (†) computes $A(i,j,0)$ consistently with the definition (*) If no $i \overset{0}{\rightsquigarrow} j$ path exists, then $i\neq j$ and there is no edge (i,j), so the recursive formula computes $A(i,j,k)$ to be ∞, again consistently with the definition (*). (Recall that, for now, we assume no negative-weight cycles, so if $(i,i)\in E$, then $\mathbf{wt}(i,i)\geq 0$.)

For the induction step, let $k>0$ and suppose the recursive formula is correct for $k-1$. There are two cases to consider.

Case 1. There is an $i \overset{k}{\rightsquigarrow} j$ path. Then, by the assumption that the graph has no negative-weight cycles, there is a *shortest* $i \overset{k}{\rightsquigarrow} j$ path. Let p be a shortest $i \overset{k}{\rightsquigarrow} j$ path *with the fewest edges* among all shortest $i \overset{k}{\rightsquigarrow} j$ paths. We must show that the recursive formula computes $A(i,j,k)=\mathbf{wt}(p)$. There are two possibilities:

(a) Path p does not use k as an intermediate node. Then p is actually an $i \overset{k-1}{\rightsquigarrow} j$ path and, by a cut-and-paste argument, a shortest such path. So, by the induction hypothesis the recursive formula computes $A(i,j,k{-}1)=\mathbf{wt}(p)$.

(b) Path p uses k as an intermediate node. Since p has the fewest edges among all shortest $i \overset{k}{\rightsquigarrow} j$ paths, and all cycles have non-negative weight, k appears only once on p. Thus, p consists of two parts: a prefix p_1 that goes from i to k with intermediate nodes in $[1..k{-}1]$, continued by a suffix p_2 that goes from k to j with intermediate nodes in $[1..k{-}1]$. Thus p_1 is an $i \overset{k-1}{\rightsquigarrow} k$ path and p_2 is an $k \overset{k-1}{\rightsquigarrow} j$ path. By a straightforward cut-and-paste argument p_1 and p_2 are actually *shortest* $i \overset{k-1}{\rightsquigarrow} k$ and $k \overset{k-1}{\rightsquigarrow} j$ paths, respectively. By the induction hypothesis, (†) computes $A(i,k,k{-}1)=\mathbf{wt}(p_1)$ and $A(k,j,k{-}1)=\mathbf{wt}(p_2)$. So $A(i,k,k{-}1)+A(k,j,k{-}1)=\mathbf{wt}(p_1)+\mathbf{wt}(p_2)=\mathbf{wt}(p)$.

Since $k>0$, (†) computes $A(i,j,k)=\min\big(A(i,j,k{-}1), A(i,k,k{-}1){+}A(k,j,k{-}1)\big)$, and so by (a) and (b), $A(i,j,k)=\mathbf{wt}(p)$, as wanted.

Case 2. There is no $i \overset{k}{\rightsquigarrow} j$ path. In this case we must show that the recursive formula computes $A(i,j,k)=\infty$. Since there is no $i \overset{k}{\rightsquigarrow} j$ path, for each $k\in[1..n]$ it must be that:

- there is no $i \overset{k-1}{\rightsquigarrow} j$ path, and

- there is no $i \overset{k-1}{\rightsquigarrow} k$ path or there is no $k \overset{k-1}{\rightsquigarrow} j$ path.

By the induction hypothesis, the recursive formula (†) computes $A(i,j,k-1)$ to be ∞ and at least one of $A(i,k,k-1)$ and $A(k,j,k-1)$ to be ∞. So (†) also computes $A(i,j,k)$ to be $\min(\infty,\infty) = \infty$, as wanted.

Following is pseudocode for the Floyd-Warshall algorithm.

```
FloydWarshall(G, wt)
 1 for i := 1 to n do
 2     for j := 1 to n do
 3         if i = j then A(i,j,0) := 0
 4         elsif (i,j) is an edge then A(i,j,0) = wt(i,j)
 5         else A(i,j,0) := ∞
 6 for k := 1 to n do
 7     for i := 1 to n do
 8         for j := 1 to n do
 9             A(i,j,k) := min(A(i,j,k−1), A(i,k,k−1) + A(k,j,k−1))
10 for i := 1 to n do
11     for j := 1 to n do
12         Δ(i,j) := A(i,j,n)
13 return Δ
```

It is clear that this algorithm computes $A(i,j,k)$ according to the recursive formula (†), and thus, by the earlier argument, according to the definition (*). A shortest $i \overset{n}{\rightsquigarrow} j$ path is a shortest $i \rightsquigarrow j$ path (since it can pass through any nodes), so the algorithm returns the weight of shortest path for each pair of nodes (or ∞ if no path exists between the two nodes).

The running time of the algorithm is dominated by the triply-nested loop in lines 5-8,so it is $\Theta(n^3)$.

The space complexity of the algorithm, as presented above, is also $\Theta(n^3)$, since we are computing a three-dimensional table $A(i,j,k)$. It turns out, however, that the third dimension of this table is unnecessary: The algorithm still works if we replace $A(i,j,k)$ by just $A(i,j)$. This may appear dangerous: To compute $A(i,j,k)$ we need $A(i,j,k-1)$, $A(i,k,k-1)$, and $A(k,j,k-1)$. So, if we eliminate the third dimension, the computation in line 8 becomes $A(i,j) := \min\big(A(i,j), A(i,k) + A(k,j)\big)$; and if, during iteration k of the main loop (lines 5-8), $A(i,k)$ or $A(k,j)$ has been updated before $A(i,j)$ is updated, the quantity stored in $A(i,j)$ might no longer be the weight of a shortest $i \overset{k}{\rightsquigarrow} j$ path. Fortunately, however, $A(i,k,k) = A(i,k,k-1)$ and $A(k,j,k) = A(k,j,k-1)$; do you see why? So any prior update of $A(i,k)$ and $A(k,j)$ in iteration k would have had no effect, and the invariant that, at the end of the k-th iteration of the main loop, $A(i,j)$ is the weight of a shortest $i \overset{k}{\rightsquigarrow} j$ path, would remain true. So, the Floyd-Warshall algorithm can be implemented using only $\Theta(n^2)$ space.

As usual, the algorithm can be retrofitted so that it computes actual shortest paths between all pairs of nodes, rather than their weight; see Exercise 4.8.

If we drop the assumption that the graph has no negative-weight cycles, the algorithm computes enough information to allow us to detect this fact. Specifically, there is a negative weight cycle that includes node i if and only if at the end of the algorithm $A(i,i,n) < 0$. If

the graph contains a negative-weight cycle, the value of $A(i,j,k)$ computed by the algorithm based on the recursive formula (†) is no longer the weight of a shortest $i \overset{k}{\rightsquigarrow} j$. See Exercise 4.9.

Computing the transitive closure of a relation

Let $G = (V, E)$ be a directed graph. The ***transitive closure*** of V is the graph $G^* = (V, E^*)$ where $(i, j) \in E^*$ if there is an $i \rightsquigarrow j$ path in G. Computing the transitive closure of a graph is a useful operation that comes up in query processing contexts such as this: Suppose we have a large software engineering project where the codebase consists of tens of thousands of procedures. An important relation between procedures is whether A calls B, so we can represent this relation as a "call-graph" $G = (V, E)$, where V is the set of procedures and there is an edge $(A, B) \in E$ if and only if procedure A calls procedure B. Now suppose some changes are made to some procedure B; this may affect the procedures that call B, the procedures that call the procedures that call B, and so on — i.e., all the procedures for which there is a path to B in the call graph. If we had G^*, the transitive closure of G, these are exactly the procedures A such that there is a direct edge (A, B) in G^*. Note that G is easy to compute by analyzing the text of every procedure. Yet, to answer a query such as "list all the procedures that might be affected if I change procedure A", we need G^*.

We can easily compute the transitive closure of a graph using the Floyd-Warshall algorithm: Given a graph G, assign weight one to each edge. An $i \rightsquigarrow j$ path exists in G if and only if at the end of the Floyd-Warshall algorithm $A(i, j, n) \neq \infty$. There is a more elegant formulation of this algorithm if all we are interested in is knowing whether an $i \rightsquigarrow j$ exists (i.e., if (i, j) is an edge in G^*), as opposed to knowing the minimum number of edges on an $i \rightsquigarrow j$ path.[4] For all $i, j \in [1..n]$ and $k \in [0..n]$ define the following Boolean function:

$$B(i,j,k) = \begin{cases} 1, & \text{if an } i \overset{k}{\rightsquigarrow} j \text{ path exists} \\ 0, & \text{otherwise} \end{cases} \qquad (*)$$

You should convince yourself that the following recursive formula correctly computes $B(i, j, k)$ as defined by (*):

$$B(i,j,k) = \begin{cases} 1, & k = 0, \text{ and } i = j \text{ or } (i,j) \in E \\ 0, & k = 0,\ i \neq j, \text{ and } (i,j) \notin E \\ B(i,j,k-1) \vee \big(B(i,k,k-1) \wedge B(k,j,k-1)\big), & \text{otherwise (i.e., if } k > 0) \end{cases} \qquad (\dagger)$$

[4] In fact, Warhsall's formulation of the algorithm that now bears his name was for the transitive-closure problem, not for all-pairs shortest paths.

```
TransitiveClosure(G)
1 for i := 1 to n do
2     for j := 1 to n do
3         if i = j or (i, j) ∈ E then B(i, j, 0) := 1
4         else B(i, j, 0) := 0
5 for k := 1 to n do
6     for i := 1 to n do
7         for j := 1 to n do
8             B(i, j, k) := B(i, j, k − 1) ∨ (B(i, k, k−1) ∧ B(k, j, k−1))
```

It is easy to see that this computes $B(i, j, k)$ according to (†) and therefore at the end of the algorithm the two-dimensional Boolean table $B(-, -, n)$ is the adjacency matrix of the transitive closure of G. It is also the case that the third dimension of the table is unnecessary.

4.5 Johnson's algorithm

If all edges have non-negative weights, we can solve the all-pairs shortest-paths problem by running Dijkstra's algorithm n times, once with each node as the source. The running time of the resulting algorithm is $\Theta\big((m+n)n \log n\big)$. If the graph G is sparse, (e.g., if $m = O(n \log n)$), this is better than the $O(n^3)$ running time of the Floyd-Warshall algorithm. Unfortunately, we cannot use this idea if G has negative-weight edges since, in that case, Dijkstra's algorithm does not work.

Johnson's algorithm uses a very clever idea to adjust the edge weights in a way that

- makes all edge weights non-negative, and
- *preserves shortest paths.*

A naive way to make all edge weights non-negative is to increase every edge weight by the minimum weight of any edge. This edge-weight adjustment, however, does not preserve shortest paths, as it favours paths that use few edges. (Construct an example to demonstrate this!) Johnson's algorithm adjusts the weights of edges by possibly different amounts so that, for any two nodes u and v, the weights of *all* $u \rightsquigarrow v$ paths change by exactly the same amount, thus preserving shortest paths between any two nodes.

Of course if the graph G contains a negative-weight cycle, such an adjustment of edge weights is impossible: If all edges have non-negative weights, shortest paths are well defined while, as we have seen, there is no such thing as a shortest path through a negative-weight cycle. So a first step in Johnson's algorithm is to determine whether the graph G has a negative weight cycle. To do so, it uses the Bellman-Ford algorithm as a subroutine.

The basic idea behind Johnson's adjustment of the edge weights is as follows. We will assign to every *node* u some weight x_u. For now we take the node weights to be arbitrary values; they will be determined by the Bellman-Ford algorithm, if it finds that the graph has no negative-weight cycle. Given the node weights, Johnson's algorithm defines a new edge-weight function:

$$\widehat{\mathbf{wt}}(u, v) = \mathbf{wt}(u, v) + x_u - x_v.$$

So, the weight of an edge is boosted by the weight assigned to its tail and reduced by the weight assigned to its head.

Let $p = u_1, u_2, \ldots, u_k$ be a path in G. We have:

$$\begin{aligned}
\widehat{\mathbf{wt}}(p) &= \widehat{\mathbf{wt}}(u_1, u_2) + \widehat{\mathbf{wt}}(u_2, u_3) + \cdots + \widehat{\mathbf{wt}}(u_{k-1}, u_k) \\
&= \mathbf{wt}(u_1, u_2) + x_{u_1} - \cancel{x_{u_2}} \\
&\quad + \mathbf{wt}(u_2, u_3) + \cancel{x_{u_2}} - \cancel{x_{u_3}} \\
&\quad + \mathbf{wt}(u_3, u_4) + \cancel{x_{u_3}} - \cancel{x_{u_4}} \\
&\quad \vdots \\
&\quad + \mathbf{wt}(u_{k-2}, u_{k-1}) + \cancel{x_{u_{k-2}}} - \cancel{x_{u_{k-1}}} \\
&\quad + \mathbf{wt}(u_{k-1}, u_k) + \cancel{x_{u_{k-1}}} - x_{u_k} \\
&= \mathbf{wt}(p) + x_{u_1} - x_{u_k}.
\end{aligned}$$

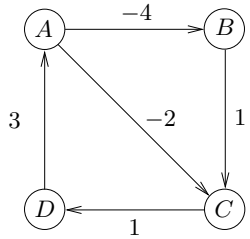
So the new weight of every path connecting two nodes is the old weight of the same path plus the difference of the weights of the path's endpoints. In other words, the weight of *every* path connecting two nodes changes *by the same amount*. This means that the new edge-weight function is shortest-path preserving, giving us one of the two properties we wanted to achieve from the re-weighing of the edges.

The other property we wanted is that all edge weights must be non-negative, so that we can use Dijkstra's algorithm. So, for every edge (u, v) we want the node weights x_u and x_v to satisfy:

$$\begin{aligned}
& \widehat{\mathbf{wt}}(u, v) \geq 0 \\
\Rightarrow \quad & \mathbf{wt}(u, v) + x_u - x_v \geq 0 \\
\Rightarrow \quad & x_v - x_u \leq \mathbf{wt}(u, v).
\end{aligned} \tag{4.10}$$

Inequality (4.10) is called a ***difference constraint***. There is one such inequality for each edge, and we call the set of all such inequalities the ***difference constraints associated with*** G ***and*** $\mathbf{wt}$. So, we are seeking values for node weights x_u, for all $u \in V$, that simultaneously satisfy the m difference constraints associated with G and $\mathbf{wt}$.

Figure 4.5 shows a graph and the associated difference constraints.



$$\begin{aligned}
x_B - x_A &\leq -4 \\
x_C - x_A &\leq -2 \\
x_C - x_B &\leq 1 \\
x_D - x_C &\leq 1 \\
x_A - x_D &\leq 3
\end{aligned} \tag{4.11}$$

Figure 4.5: Difference inequalities for node weights

In this case, there are values for the node weights that satisfy all these inequalities, for example, $x_A = 0$, $x_B = -4$, $x_C = -3$, and $x_D = -2$. This is not an accident: As the next theorem shows, values that satisfy the node-weight difference constraints exist if and only if the graph has no negative-weight cycle. Even better, if no such cycle exists, the theorem tells us how to find values for the node weights that satisfy the difference constraints. From this example you may already have an inkling as to what these values might be!

Given a directed graph $G = (V, E)$, edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$, and a new node $s \notin V$, we define a new edge-weighted graph G_s, which is simply G extended with node s and edges from s to every other node, all with weight 0. More precisely, we define graph $G_s = (V_s, E_s)$ and $\mathbf{wt}_s$: $E_s \to \mathbb{Z}$ as follows:

$$\begin{aligned} V_s &= V \cup \{s\} \\ E_s &= E \cup \{(s, u) \colon u \in V\} \\ \mathbf{wt}_s(u, v) &= \begin{cases} 0, & \text{if } u = s \\ \mathbf{wt}(u, v) & \text{otherwise (i.e., } (u, v) \in E) \end{cases} \end{aligned} \tag{4.12}$$

Theorem 4.10 *Let $G = (V, E)$ be a directed graph, $\mathbf{wt}$: $E \to \mathbb{Z}$ be an edge-weight function, and G_s, $\mathbf{wt}_s$ be as defined in (4.12).*

(a) If the difference constraints associated with G and $\mathbf{wt}$ are satisfiable, then G has no negative-weight cycle.

(b) If G has no negative-weight cycle, then taking

$$x_u = \textit{the weight of a shortest } s \rightsquigarrow u \textit{ path in } G_s$$

for each $u \in V$, satisfies the difference constraints associated with G and $\mathbf{wt}$.

Proof. For part (a), suppose the difference constraints are satisfiable, and let x^*_u, for all $u \in V$, be values for the node weights that satisfy the constraints. The constraints for the edges along any cycle $C = u_1, u_2, \ldots, u_k, u_1$ in G imply:

$$\begin{aligned} x^*_{u_2} - x^*_{u_1} &\le \mathbf{wt}(u_1, u_2) \\ x^*_{u_3} - x^*_{u_2} &\le \mathbf{wt}(u_2, u_3) \\ x^*_{u_4} - x^*_{u_3} &\le \mathbf{wt}(u_3, u_4) \\ &\vdots \\ x^*_{u_{k-1}} - x^*_{u_{k-2}} &\le \mathbf{wt}(u_{k-2}, u_{k-1}) \\ x^*_{u_k} - x^*_{u_{k-1}} &\le \mathbf{wt}(u_{k-1}, u_k) \\ x^*_{u_1} - x^*_{u_k} &\le \mathbf{wt}(u_k, u_1). \end{aligned} \tag{4.13}$$

Adding up all these inequalities, and observing that *all* the terms on the left-hand side cancel out, we get that $\mathbf{wt}(C) = \mathbf{wt}(u_1, u_2) + \mathbf{wt}(u_2, u_3) + \cdots + \mathbf{wt}(u_k, u_1) \ge 0$, as wanted.

For part (b), suppose that G has no negative-weight cycle. Then G_s also has no negative-weight cycle: no cycle in G_s can involve the new node s, since there are no edges going *into* s,

so any cycle in G_s is also present in G and, by assumption, G has no negative weight cycle. For each $w \in V$, let p_w be a shortest $s \rightsquigarrow w$ path in G_s and $x^*_w = \mathbf{wt}(p_w)$; these are well defined since (a) by construction, all nodes are reachable from s in G_s and (b) by assumption, G_s has no negative-weight cycle. We claim that setting $x_u = x^*_u$, for each $u \in V$, satisfies the difference constraints associated with G and $\mathbf{wt}$. That is, we claim that, for every edge (u, v) of G, $x^*_v - x^*_u \leq \mathbf{wt}(u, v)$ or, equivalently, that

$$x^*_v \leq x^*_u + \mathbf{wt}(u, v).$$

But this is clearly true by the triangle inequality (4.1). □

We now have all the ingredients we need for Johnson's algorithm: We have found a way to adjust the edge weights of a graph so that the new weights are non-negative and preserve shortest paths between any two nodes under the original edge weights. Thus we can then run Dijkstra's algorithm using the adjusted weights n times, once with each node as the source. The edge-weight adjustment works only if the graph has no negative-weight cycles but, as we have seen, this is a necessary condition for the all-pairs shortest-paths problem to be meaningful.

Following is a description of the algorithm in pseudocode.

JOHNSON$(G, \mathbf{wt})$
1 let G_s and $\mathbf{wt}_s$ be as defined in (4.12)
2 $r :=$ BELLMANFORD$(G_s, s, \mathbf{wt}_s)$
3 **if** $r =$ "negative-weight cycle" **then return** r
4 **else** ▶ $r(u)$ is the weight of a shortest $s \rightsquigarrow u$ path
5 **for each** node u of G **do** $x^*_u := r(u)$
6 **for each** edge (u, v) of G **do** $\widehat{\mathbf{wt}}(u, v) := \mathbf{wt}(u, v) + x^*_u - x^*_v$
7 **for each** node u of G **do**
8 $\delta :=$ DIJKSTRA$(G, u, \widehat{\mathbf{wt}})$ ▶ $\delta(v)$ is the weight of a shortest $u \rightsquigarrow v$ path
9 **for each** node v of G **do** $\Delta(u, v) := \delta(v) + x^*_v - x^*_u$
10 **return** Δ

The correctness of this algorithm follows from the preceding discussion and the properties of the Bellman-Ford and Dijkstra's algorithms. To recap: The call to the Bellman-Ford algorithm in line 2 returns "negative-weight cycle" if and only if a negative-weight cycle exists in G_s which, as we have seen, means that a negative-weight cycle exists in G. So, in this case, Johnson's algorithm also returns "negative-weight cycle", which is the correct thing to do. Otherwise, the call to the Bellman-Ford algorithm returns a function r on the nodes of G_s such that $r(u)$ is the weight of a shortest $s \rightsquigarrow u$ path. Note that this is not ∞ since every node is reachable from s in G_s. Then in line 5 Johnson's algorithm assigns weights to all nodes in a way that, by Theorem 4.10(b), satisfies all the difference constraints and therefore ensures that the new edges weights $\widehat{\mathbf{wt}}$ computed in line 6 are non-negative. As we have seen, this way of reweighing the edges (by adding to the original weight of an edge the difference between the weights of its endpoints) preserves shortest weights under the original edge-weight function and, in fact, increases the weight of a shortest $u \rightsquigarrow v$ path by exactly $x^*_u - x^*_v$. So Johnson's algorithm now runs Dijkstra's algorithm with the new non-negative and shortest-path-preserving edge weights

$\widehat{\mathbf{wt}}$ using each node u as the source, to compute the weight of shortest $u \rightsquigarrow v$ paths for each node v (lines 7-8). It then adjusts the weight of the path by $x^*_v - x^*_u$ to restore the weight of a shortest $u \rightsquigarrow v$ path under the original weight function (line 9).

For the running time of the algorithm, note that line 2 takes $\Theta(mn)$ time and lines 5 and 6 take $\Theta(n+m)$ time. Each call to Dijkstra's algorithm in line 8 takes $\Theta\big((m+n)\log n)\big)$ time and line 9 takes $O(n)$ time, so each iteration of the loop in lines 7-9 takes $\Theta\big((m+n)\log n\big)$ time. Since the loop is repeated n times, the overall time for the loop in lines7-9 is $\Theta\big((m+n)n\log n\big)$, and this dominates the other work done by the algorithm. So, the running time of Johnson's algorithm is $\Theta\big((m+n)n\log n\big)$.

Exercises for Chapter 4

4.1 Modify Dijkstra's algorithm so that it also returns the function **pre** defined in (4.4), and extend the correctness proof to show that your modification is correct. The modification should not affect the (asymtotic) running time of the algorithm.

4.2 A computer network can be described as a directed graph $G = (V, E)$ with positive edge weights $\mathbf{wt} : E \to \mathbb{Z}^+$. The nodes represent routers, the edges represent unidirectional links connecting routers, and the weight of edge (u, v) is the rate at which router u can transmit data to router v, measured in bits per second. To stream a file from node s to node t we must select a path from s to t, along which to send the file's bits in their sequence. The rate at which we can transmit the file along a path is limited by the ***slowest*** link on that path. Thus, we define the transmission rate of path $p = v_1, v_2, \ldots, v_k$ as $\mathbf{tr}(p) = \min_{1\le i<k} \mathbf{wt}(v_i, v_{i+1})$. To transmit a file from node s to node t as fast as possible we would like to use an $s \to t$ path of ***maximum*** transmission rate.

Modify Dijkstra's algorithm to compute an $s \rightsquigarrow t$ path of maximum transmission rate given the graph G, the edge weight function **wt**, and the nodes s, t. Explain the information that your algorithm maintains. You do not need to prove that your algorithm is correct, but you should re-state the five lemmas in the proof of correctness of Dijkstra's algorithm, adjusted to the information maintained by your algorithm. (You don't need to prove these lemmas: If you do things right, the proofs are essentially the same as for Dijkstra's algorithm, and you can use this fact to check if you did things right!)

4.3 The *single-pair* shortest-paths problem is to find a shortest path from a source node s to a target node t. Assuming that all edges have non-negative weights, we can solve this problem by using Dijkstra's algorithm and stopping as soon as we have found the shortest $s \rightsquigarrow t$ path. Of course this does not change the worst-case running time of the algorithm.

Another idea that also does not change the worst-case running time, but seems to work well in practice (and is used, for example, in mapping applications), is the so-called ***bidirectional*** version of Dijkstra's algorithm: Explore the graph starting from s as usual (forward exploration) and at the same time explore the *reverse* of the graph starting from t (reverse

exploration). (The reverse of $G = (V, E)$ is the graph $G^R = (V, E^R)$, where $(v, u) \in E^R$ if and only if $(u, v) \in E$.) We can alternate between forward and reverse exploration in any way; e.g., by advancing each to discover a new shortest path (from s or t, respectively) to one more node. When the two explorations "come together" (in a sense you will investigate in this exercise) we can stop and extract the desired shortest $s \rightsquigarrow t$ path.

Let's denote by $d_s(u)$ the current d-value of node u and by R_s the set of nodes for which shortest paths from s have been found so far in the *forward* exploration. Similarly, $d_t(u)$ is the current d-value of node u and R_t is the set of nodes for which shortest paths from t have been found so far in the *reverse* exploration. At what point during these explorations can we stop, having found a shortest $s \rightsquigarrow t$ path?

(a) It is perhaps tempting to think that we can stop as soon as both explorations reach the same node — i.e., as soon as $R_s \cap R_t = \{u\}$, for some node u — and that a shortest $s \rightsquigarrow t$ path then consists of the shortest $s \rightsquigarrow u$ path found by the forward exploration followed by the reverse of the shortest $t \rightsquigarrow u$ path bound by the reverse exploration. Show that this is not true by means of a counterexample.

(b) Describe an algorithm that finds a shortest $s \rightsquigarrow t$ path using Dijkstra's algorithm to perform simultaneous forward and reverse explorations of the graph. In particular, describe the condition under which your algorithm stops the explorations and how it recovers a shortest $s \rightsquigarrow t$ from the information computed by these explorations. Justify the correctness of your algorithm. (Your condition for stopping should be as weak as you can; for example, $R_s = V$ is obviously a safe condition, but it is too strong — you gain no benefit from the reverse exploration. Intuitively we want to stop as soon as we have enough information from the two explorations to determine a shortest $s \rightsquigarrow t$ path.)

4.4 Prove that the version of the Bellman-Ford algorithm where $S(u, k)$ is replaced by $S(u)$ (one-dimensional table, indexed on the nodes) is correct.

4.5 Retrofit the Bellman-Ford algorithm so that, (a) if no negative-weight cycle is reachable from s then it outputs a function **pre**: $V \to V$ so that

$$\textbf{pre}(u) = \begin{cases} s, & \text{if } u = s \\ \text{the predecessor of } u \text{ on a shortest } s \rightsquigarrow u \text{ path}, & \text{if } u \neq s \text{ and there is a } s \rightsquigarrow u \text{ path} \\ \text{NIL}, & \text{otherwise} \end{cases}$$

and (b) if a negative-weight cycle is reachable from s then it outputs such a cycle. Be sure to justify your retrofit's correctness — in particular, that if a negative-weight cycle is reachable from s then the algorithm outputs such a cycle.

4.6 The retrofit described in Exercise 4.5 finds negative-weight cycles *reachable from the source node s*. We can look for negative-weight cycles (regardless of where they are reachable from) by running the algorithm n times, once with each different node as the source, but this is quite expensive. Describe a way to solve this problem by running the Bellman-Ford algorithm only once, and justify the correctness of your solution. (Hint: Modify the graph a little bit.)

4.7 Design an efficient algorithm that takes as input a directed graph $G = (V, E)$, a node $s \in V$, a positive integer $k \leq |V|$, and an edge weight function $\mathbf{wt}$: $E \to \mathbb{Z}$, and returns, for every node u, the weight of a shortest $s \rightsquigarrow u$ path *with exactly k edges* if one exists (or ∞, if no such path exists).

4.8 Retrofit the Floyd-Warshall algorithm so that it returns a function $\sigma(i, j)$, for $i, j \in [1..n]$, where

$$\sigma(i,j) = \begin{cases} \text{the successor of } i \text{ on a shortest } i \rightsquigarrow j \text{ path}, & \text{if an } i \rightsquigarrow j \text{ path exists} \\ \text{NIL}, & \text{otherwise.} \end{cases}$$

4.9 Prove that a directed graph $G = (V, E)$ with edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$ has a negative-weight cycle that goes through node i if and only if at the end of the Floyd-Warshall algorithm $A(i, i, n) < 0$.

(Hint: As noted in Section 4.4 (page 115), the quantity $A(i, j, k)$ computed by the algorithm does not satisfy (*) if a negative-weight cycle exists. Formulate (and prove) an invariant that $A(i, j, k)$ satisfies at the end of each iteration $k \in [1..n]$, and show how to use your invariant to prove the result stated in this exercise. You may find the concept of "simple path" and "simple cycle" useful: A path $i_1, i_2, \ldots i_m$ is ***simple*** if no node on the path is repeated, except possibly the first and last; i.e., for all $p, q \in [1..m]$ such that $p < q$, if $i_p = i_q$ then $p = 1$ and $q = m$. A ***simple cycle*** is a simple path that is a cycle; i.e., a simple path in which the first and last nodes are the same.)

4.10 Let $G = (V, E)$ be an undirected graph, $\mathbf{wt} : E \to \mathbb{Z}$ be an edge weight function such that $\mathbf{wt}(e) > 0$ for every $e \in E$, $V' \subseteq V$, and $d \in \mathbb{Z}$ be a positive integer. The graph represents a road map of a rural area. Each node corresponds to a town, each edge corresponds to a road that connects two towns, and the weight of an edge indicates the length of that road. The subset V' is the subset of towns that have a gas station, and d is the maximum distance that your car can travel on a full tank of gas.

Given the above information and two nodes $s, t \in V'$, you want to find the shortest route so you can drive from s to t *without running out of gas*; that is, any portion of the route that is longer than d should pass through a town with a gas station. Note that s and t are both in V' — so you can fill up before the trip starts and when it ends.

Describe a polynomial time algorithm that takes as input G, $\mathbf{wt}$, V', d, and $s, t \in V'$, and returns the desired route (as a sequence of towns). Your algorithm should be by *reduction* to one or more problems for which we have studied efficient algorithms in this course. *Do not solve the problem "from scratch"!* You need not justify the correctness of your algorithm; this should be straightforward given your reduction. Analyze the running time of your algorithm.

4.11 Let $G = (V, E)$ be a directed graph and $\mathbf{wt} : E \to \mathbb{Z}$ be an edge-weight function, such that every cycle of G has *positive* weight (but note that the graph may have negative- or zero-weight edges). Modify the Floyd-Warshall algorithm to obtain an $O(n^3)$ algorithm that, given G and $\mathbf{wt}$, returns the array $N[u, v]$ for each pair of nodes $u, v \in V$ such that $N[u, v]$ is

the *number* of minimum-weight $u \to v$ paths. Justify the correctness of your algorithm. (You don't need to analyze its running time, but it should be $O(n^3)$.)

Note: We want all cycles to have *positive* (rather than merely non-negative) weight because with zero-weight cycles the minimum weight of $u \to v$ paths is well-defined but the number of such paths may be infinite.

Chapter 5

Maximum flow and applications

5.1 Flow networks and the maximum flow problem

A flow network is a mathematical model used to describe problems that involve the transportation of some kind of "traffic" from a site, where it is produced, to another, where it is consumed, through a network of other sites connected by channels that have limited capacity to carry traffic. This arises in a variety of situations, such as:

- The traffic could be vehicles, the sites could be junctions where roads intersect, and channels could be road segments between intersections with other roads. The capacity of a channel is the maximum number of cars per unit of time that can flow through the road segment.
- The traffic could be a manufactured good, the sites could be the locations of manufacturing plants, warehouses, and consumer outlets, and the edges could be connections between such locations. The capacity of a channel is the maximum amount of the good that can shipped through it.
- The traffic could be messages in a packet switching network, the nodes routers, and the edges links connecting routers. The capacity of a link is the maximum number of bits per second that can be transmitted through it.

The central problem in such settings, called the ***maximum flow problem***, is to determine the maximum amount of traffic that can be transported from the source to the destination through the channels, without exceeding the capacity of any channel.

In addition to answering this question in settings like the ones described above, a variety of problems that *prima facie* seem unrelated to transportation can be *reduced* to the maximum flow problem. We will see three examples later in this chapter: Finding a maximum matching in a bipartite graph (Section 5.4), finding a minimum vertex cover (a set of nodes that "touch" all the edges) in a bipartite graph (Section 5.5), and finding a maximum cardinality set of edge-disjoint paths in a graph (Section 5.7).

Formally, a ***flow network*** is a tuple $\mathcal{F} = (G, s, t, c)$ consisting of the following elements:

- A directed graph $G = (V, E)$, whose nodes are the "sites" and whose edges are the "channels" that connect the sites.

- A node $s \in V$, called the ***source***, that has *no incoming edges*. This is the site where the traffic is produced.
- A node $t \in V$, called the ***sink*** (or ***destination***, or ***target***), that has *no outgoing edges*. This is the site where the traffic is consumed.
- A function c: $E \to \mathbb{R}^+$ that assigns to each edge e a positive real number $c(e)$, called the ***capacity*** of the edge. We think of $c(e)$ as the maximum amount of traffic that the edge e can carry.

An example of a flow network is shown in Figure 5.1. The number next to each edge is the edge's capacity.

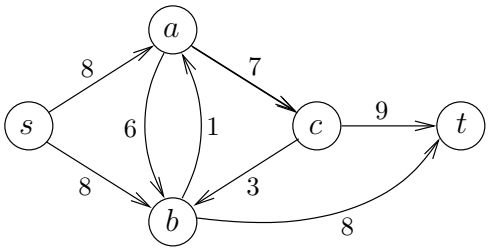


Figure 5.1: A flow network

Throughout this chapter, $\mathcal{F}$ denotes a flow network (G, s, t, c), and G is a directed graph with node set V and edge set E; $n = |V|$ is the number of nodes and $m = |E|$ is the number of edges of G. For any $v \in V$, $\text{in}(v) = \{(u, v)\text{: } (u, v) \in E\}$ is the set of edges whose head is the node v and $\text{out}(v) = \{(v, u)\text{: } (v, u) \in E\}$ is the set of edges whose tail is v.

A ***flow*** in a flow network $\mathcal{F}$ is a function f: $E \to \mathbb{R}$, subject to two constraints listed below. Intuitively, $f(e)$ is the amount of traffic that flows through edge e. The two constraints that f must satisfy are:

- **Capacity constraint:** For all $e \in E$, $0 \le f(e) \le c(e)$. That is, the amount of traffic through an edge is non-negative and cannot exceed the edge's capacity.
- **Conservation constraint:** For all $v \in V - \{s, t\}$, $\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{in}(v)} f(e)$. That is, the amount of traffic into a node (other than the source or sink) is the same as the amount of traffic out of it.

We can represent a flow f graphically by adding some information to the graphical representation of the flow network. Specifically, each edge e now has the label of the form $f(e)/c(e)$. An example of a flow in the flow network of Figure 5.1 is shown in Figure 5.2. You should verify that this is, in fact, a flow; i.e., it satisfies the capacity and conservation constraints.

The ***value*** of flow f, denoted $\mathcal{V}(f)$, is the total traffic out of the source: $\mathcal{V}(f) = \sum_{e \in \text{out}(s)} f(e)$. For example, the value of the flow f shown is Figure 5.2 is $8 + 2 = 10$. So we can now state the maximum flow problem.

Input: A flow network $\mathcal{F} = (G, s, t, c)$.
Output: A maximum flow of $\mathcal{F}$; i.e., a flow f such that for every flow f', $\mathcal{V}(f) \ge \mathcal{V}(f')$.

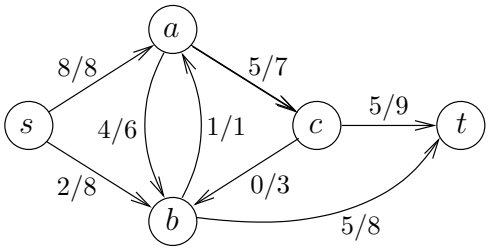


Figure 5.2: A flow f in the flow network of Figure 5.1

A ***cut*** of a flow network $\mathcal{F}$ is a pair (S,T), where S,T are sets of nodes that form a partition of the set of nodes of G such that $s \in S$ and $t \in T$.[1] If S is a set of nodes, let $\text{out}(S) = \cup_{u\in S}\text{out}(u)$, i.e., the set of edges whose tails are in S; and let $\text{in}(S) = \cup_{u\in S}\text{in}(u)$, i.e., the set of edges whose heads are in S. Thus the edges whose tails are in S and whose heads are in T are the edges in $\text{out}(S) \cap \text{in}(T)$. The ***capacity*** of cut (S,T), denoted $c(S,T)$, is the sum of capacities of edges out of S and into T: $c(S,T) = \sum_{e\in\text{out}(S)\cap\text{in}(T)} c(e)$. (S,T) is a ***minimum cut*** of $\mathcal{F}$ if it has minimum capacity; i.e., for every cut (S',T') of $\mathcal{F}$, $c(S,T) \le c(S',T')$.

For example, the capacity of the cut (S,T) of the flow network in Figure 5.1, where $S = \{s,b\}$ and $T = \{a,c,t\}$ is $8+1+8 = 17$. Note that the capacity of edges *from* T back *into* S do not contribute to the capacity of the cut. (S,T) is not a minimum cut, since the cut (S',T') with $S' = \{s,a,b\}$ and $T' = \{c,t\}$ has capacity 15. As we will see later, this *is* a minimum cut.

The following theorem tells us that we can determine the value of a flow by considering the traffic on just the edges that cross any cut, in either direction.

Theorem 5.1 *For* any *flow* f *and* any *cut* (S,T),

$$\mathcal{V}(f) = \sum_{e\in out(S)\cap in(T)} f(e) \quad - \sum_{e\in out(T)\cap in(S)} f(e).$$

Proof. Let f be an arbitrary flow and (S,T) be an arbitrary (s,t)-cut of (G,s,t,c). We have:

$$\begin{aligned}
\mathcal{V}(f) &= \Big(\underbrace{\sum_{e\in\text{out}(s)} f(e)}_{\mathcal{V}(f),\text{ by definition}} - \underbrace{\sum_{e\in\text{in}(s)} f(e)}_{0,\text{ since in}(s)=\varnothing} \Big) + \Big(\sum_{v\in S-\{s\}} \underbrace{\Big(\sum_{e\in\text{out}(v)} f(e) - \sum_{e\in\text{in}(v)} f(e) \Big)}_{0,\text{ by conservation}} \Big) \\
&= \sum_{v\in S} \Big(\sum_{e\in\text{out}(v)} f(e) - \sum_{e\in\text{in}(v)} f(e) \Big) \\
&= \sum_{v\in S} \sum_{e\in\text{out}(v)} f(e) \quad - \sum_{v\in S} \sum_{e\in\text{in}(v)} f(e)
\end{aligned}$$

[1] Recall that a partition of a set A is a collection of nonempty, disjoint subsets of A whose union is all of A.

$$
\begin{aligned}
&= \sum_{e \in \mathrm{out}(S)} f(e) \quad - \quad \sum_{e \in \mathrm{in}(S)} f(e) \\
&= \Big(\sum_{e \in \mathrm{out}(S) \cap \mathrm{in}(S)} \cancel{f(e)} + \sum_{e \in \mathrm{out}(S) \cap \mathrm{in}(T)} f(e) \Big) \quad - \quad \Big(\sum_{e \in \mathrm{in}(S) \cap \mathrm{out}(S)} \cancel{f(e)} + \sum_{e \in \mathrm{in}(S) \cap \mathrm{out}(T)} f(e) \Big) \\
&= \sum_{e \in \mathrm{out}(S) \cap \mathrm{in}(T)} f(e) \quad - \sum_{e \in \mathrm{out}(T) \cap \mathrm{in}(S)} f(e).
\end{aligned}
$$

In going from the pre-penultimate to the penultimate line in the above derivation, we use the fact that an edge out of a node in S goes either into a node in S or to a node in T but not both (because S and T are disjoint); and, similarly, an edge into a node in S goes out of either a node in S or a node in T but not both. Thus, $\mathrm{out}(S) = \big(\mathrm{out}(S) \cap \mathrm{in}(S)\big) \cup \big(\mathrm{out}(S) \cap \mathrm{in}(T)\big)$; and, similarly, $\mathrm{in}(S) = \big(\mathrm{in}(S) \cap \mathrm{out}(S)\big) \cup \big(\mathrm{in}(S) \cap \mathrm{out}(T)\big)$, where both unions are disjoint. □

From this theorem it follows immediately that the capacity of any cut is an upper bound on the value of any flow.

Corollary 5.2 *For <u>any</u> flow f and <u>any</u> cut (S, T) of a flow network $\mathcal{F}$, $\mathcal{V}(f) \leq c(S, T)$.*

And therefore

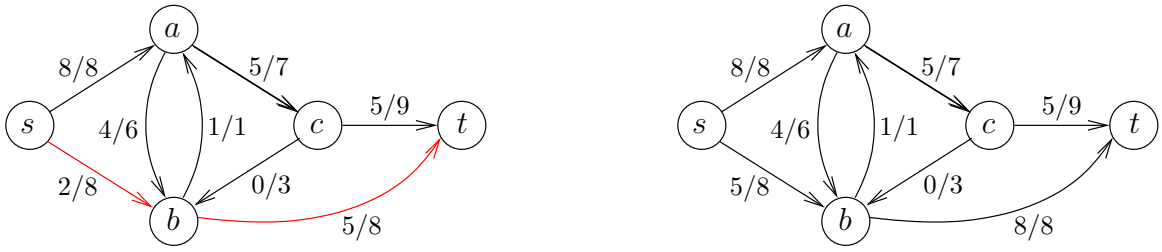
Corollary 5.3 *For <u>any</u> flow f and <u>any</u> cut (S, T) of a flow network $\mathcal{F}$, if $\mathcal{V}(f) = c(S, T)$ then f is a maximum flow and (S, T) is a minimum cut.*

5.2 Augmenting a flow

It is easy to see that the flow f in Figure 5.2, reproduced in Figure 5.3(a) for convenience, is not a maximum flow. The path s, b, t (shown in red) from the source to the sink has *unused capacity*. We can send up to 6 more units of traffic on (s, b) and 3 more units of traffic on (b, t). So we can send the *minimum* of these along this path without violating the capacity constraint and maintaining the conservation constraint, thereby increasing the flow from 10 to 13. The improved flow f' is shown in Figure 5.3(b).

You can check that in flow f' there is no longer any $s \rightsquigarrow t$ path with unused capacity: Every such path contains at least one *saturated* edge, i.e., an edge on which the flow is equal to the edge's capacity. It would be wrong, however, to conclude that f' cannot be improved. There is a more complicated way that we can accomplish this. If we redirect some of the four units of traffic that are flowing on (a, b), and are currently contributing to the traffic that goes out of b via the saturated edges (b, t) and (b, a), towards t, then we could increase the traffic on edge (s, b) by the same amount (up to the three units of unused capacity on the edge). Indeed, there is such a path: we have spare capacity on the edges (a, c) and (c, t) through which to redirect some of the flow on (a, b). Let's see what we can do with this insight. We can

- *increase* the traffic on (s, a) by *up to* 3 units;

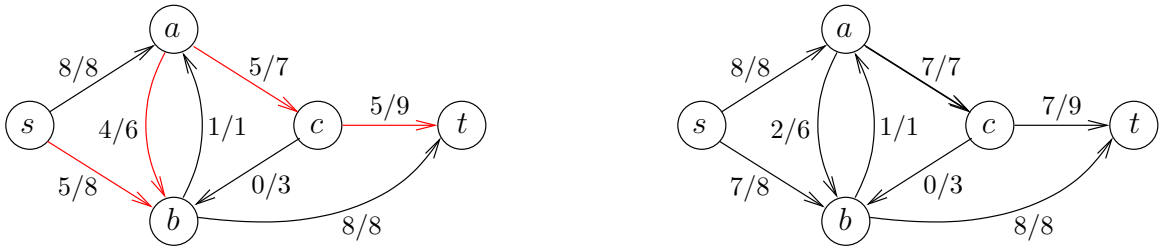


Flow f (a) Improved flow f' (b)

Figure 5.3: Improving flow f to flow f'

- *decrease* the traffic on (a, b) by *up to* 4 units;
- *increase* the traffic on (a, c) by *up to* 2 units; and
- *increase* the traffic on (c, t) by *up to* 4 units.

The affected edges are shown in red in Figure 5.4(a); these do not form a real $s \rightsquigarrow t$ path because (a, b) is traversed backward — this is the edge whose traffic can be decreased. I will call such a sequence of edges a "pseudopath". To ensure that we don't exceed the capacity of any edge whose traffic we increase and do not decrease the traffic on (a, b) below zero, we must adjust the traffic on the edges of the pseudopath by the *minimum* of these quantities, i.e., by 2. The resulting flow f'' is shown in Figure 5.4(b). This improves the value of the flow from 13 to 15.



Flow f' (a) Improved flow f'' (b)

Figure 5.4: Improving flow f' to flow f''

Is flow f'' maximum? Here is an argument to show that it is: Consider the (S, T) cut with $S = \{s, a, b\}$ and $T = \{c, f\}$. The capacity of this cut is $8 + 7 = 15$; since the value of f'' is also 15, by Corollary 5.3, f'' is a maximum flow and $(\{s, a, b\}, \{c, f\})$ is a minimum cut.

This is not an accident! It is a consequence of the memorably named "max-flow-min-cut theorem", which states that in *every* flow network the value of a maximum flow is *equal to* the capacity of a minimum cut! We will prove this important result in the next section as a

byproduct of an algorithm that solves the maximum flow problem.

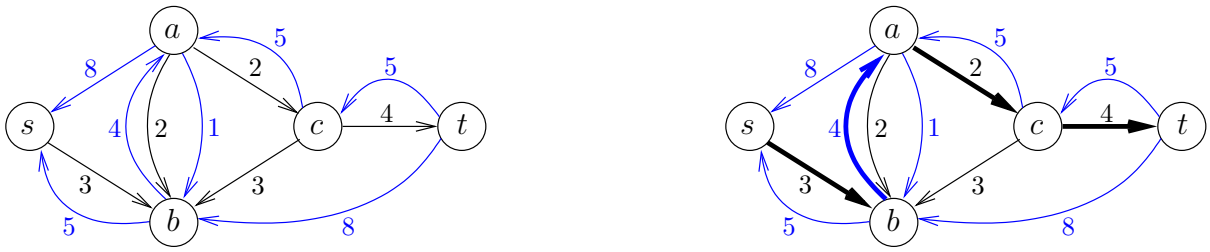
5.2.1 Residual graphs and augmenting steps

Let us now generalize the ways we saw in the previous example for improving a flow — or, to use the standard terminology, ***augmenting*** the flow. First, I note that the first improvement, from f to f' (see Figure 5.3) is a special case of the second improvement, from f' to f'' (see Figure 5.4): It increases the flow along a pseudopath that is actually a path; no edge is traversed backwards, and the traffic is increased on every edge of the path.

Given a flow f in flow network $\mathcal{F} = (G, s, t, c)$, where $G = (V, E)$, we define a labeled directed graph $\mathcal{F}_f = (V, E_f)$, called the ***residual graph of flow f (with respect to the flow network $\mathcal{F}$)***:

- The set of nodes of $\mathcal{F}_f$ is the same as the set of nodes of the graph G.
- For each edge (u, v) of G such that $f(u, v) < c(u, v)$, $\mathcal{F}_f$ has an edge (u, v) with label $c(u, v) - f(u, v)$. This is called a ***forward edge*** of the residual graph.
- For each edge (u, v) of G such that $f(u, v) > 0$, $\mathcal{F}_f$ has the edge (v, u) (*note the direction!*) with label $f(u, v)$. This is called a ***backward edge*** of the residual graph.

The labels of the edges of the residual graph are called ***residual capacities***. The residual capacity of a forward edge (u, v) is the maximum additional traffic that can be sent along (u, v). The residual capacity of a backward edge (v, u) is the maximum amount of traffic on the edge (u, v) that can be *redirected* to a node other than v, so that the amount of traffic into v from a node other than u can be increased accordingly. For example in augmenting f' to f'', we redirected traffic from (a, b) so that we could increase the traffic on (s, b). The residual graph of f' is shown in Figure 5.5(a); forward edges are shown in black and backward edges in blue.



(a) (b)

Figure 5.5: The residual graph $\mathcal{F}_{f'}$ (a) and an augmenting path (b)

So, every edge of G gives rise to at most two edges in F_f: a forward edge and a backward edge. If an edge of G is saturated by f, it gives rise only to a backward edge (such as (a, s), (a, b), and (t, b) in Figure 5.5(a)); if an edge is not used by f, i.e., it has no traffic, it gives rise only to a forward edge (such as (c, b) in the figure).

Digression There is an inconsequential detail that is worth pointing out to avoid confusion. Technically the residual graph is a *multigraph*, i.e., a mathematical structure similar to a graph except that it may have multiple edges between the same pair of nodes (running in the same direction, in case of directed graphs). This is the case in our example, where there are two edges (a, b), a forward arising from (a, b) in G and a backward arising from (b, a) in G. To avoid this, in some treatments of network flow the authors proscribe graphs that have edges connecting the same pair of nodes in opposite directions, such as (a, b) and (b, a) in our example because these give rise to multiple edges between the two nodes in the residual graph. This restriction can be made without loss of generality, because if we had a graph with such "anti-parallel edges", we could insert a node in the middle of one of the two. However, there is no real damage if the residual graph is a multigraph: the depth-first or breadth-first search algorithms that we will apply to it work equally well on multigraphs as on ordinary graphs. So we will not make this assumption in our discussion of flow networks. Following tradition we will continue referring to it as residual *graph*, rather than residual *multi*graph. **End of Digression**

Let's now review the augmentation of flow f' to f'' and rethink the steps in light of the residual graph of f'. What we did was follow what we called an $s \rightsquigarrow t$ "pseudopath" in G, consisting of *non-saturated edges* followed in the correct direction, into which we add more traffic, and *edges with non-zero flow* followed in the reverse direction from which we diverted traffic. Such a pseudopath becomes an actual path in the residual graph: an edge (u, v) traversed in the wrong direction (from v to u) in the pseudopath of G is a *backward* edge (v, u) in the residual graph. An $s \rightsquigarrow t$ path in the residual graph is called an ***augmenting path***. Furthermore, the amounts by which we can increase (respectively decrease) the traffic on the edges of the pseudopath traversed in the correct (respectively wrong) direction are precisely the residual capacities of the corresponding edges in the residual graph. And how did we improve flow f' to f''? We took the *minimum* of the residual capacities of the edges on the augmenting path, which in our example was 2, we increased by 2 the traffic on edges traversed in the correct direction (forward edges in the residual graph), and we decreased by 2 the traffic on edges traversed in the wrong direction (the *reverses* of backward edges in the residual graph). We used the *minimum* residual capacity of the edges on the path to ensure that we did not increase the traffic of any edge traversed in the correct direction beyond its capacity or decrease the traffic of any edge traversed in the wrong direct below zero. Since all residual capacities are strictly positive, the minimum residual capacity of the edge on the path is a positive quantity, so the net effect of this operation, called an ***augmenting step***, is to increase the value of the flow.

Recall that an $s \rightsquigarrow t$ path is ***simple*** if it contains no cycle. Here is how to augment flow f along an $s \rightsquigarrow t$ simple path p of the residual graph, expressed in pseudocode:

```
AUGMENT(f, p)
1 b := minimum residual capacity of any edge on p
2 for each edge (u, v) on p do
3     if (u, v) is a forward edge then    ▶ (u, v) is an edge in G
4         f(u, v) := f(u, v) + b
5     else    ▶ (v, u) is an edge in G
6         f(v, u) := f(v, u) − b
7 return f
```

We now prove that this operation improves the flow.

Theorem 5.4 *Suppose f is a flow in $\mathcal{F}$, p is a simple $s \rightsquigarrow t$ path in the residual graph of f, b is the minimum residual capacity of the edges on p, and f' is the function returned by* AUGMENT(f, p). *Then f' is a flow in $\mathcal{F}$ and $\mathcal{V}(f') = \mathcal{V}(f) + b > \mathcal{V}(f)$.*

PROOF. First we prove that f' is a flow in $\mathcal{F}$. To do so we must prove that f' satisfies the capacity and conservation constraints. Let b be the minimum residual capacity of all edges on p; since all residual capacities are strictly positive, $b > 0$.

Capacity: Let (u, v) be any edge of E. Since f is a flow, we have $0 \le f(u, v) \le c(u, v)$. We must show that $0 \le f'(u, v) \le c(u, v)$. There are three cases.

- If (u, v) is on p, then it is a forward edge with residual capacity $c(u, v) - f(u, v)$. Since b is the minimum residual capacity of all edges on p, $b \le c(u, v) - f(u, v)$. Since (u, v) is a forward edge, $f'(u, v) = f(u, v) + b$. Since $f(u, v) \ge 0$ and $b > 0$, $f'(u, v) \ge 0$. Furthermore, $f'(u, v) = f(u, v) + b \le f(u, v) + \big(c(u, v) - f(u, v)\big) \le c(u, v)$. So f' satisfies the capacity constraint at (u, v).
- If (v, u) is on p, then it is a backward edge with residual capacity $f(u, v)$. Since b is the minimum residual capacity of all edges on p, $b \le f(u, v)$. Since (v, u) is a backward edge, $f'(u, v) = f(u, v) - b$. Since $f(u, v) \le c(u, v)$ and $b > 0$, $f'(u, v) \le c(u, v)$. Furthermore, $f'(u, v) = f(u, v) - b \ge f(u, v) - f(u, v) \ge 0$. So f' satisfies the capacity constraint at (u, v).
- If neither (u, v) nor (v, u) is on p, then $f'(u, v) = f(u, v)$, so f' satisfies the capacity constraint at (u, v).

Conservation: Let v be any node other than s or t. If v is not on p then the flow into and out of v is the same in f' as in f, so conservation is satisfied at v. So suppose v is on p. Since p is a simple path, there is exactly one edge (u, v) going into v and one edge (v, w) coming out of v.

There are four cases, depending on whether each of these two edges is forward or backward. These are illustrated in Figure 5.6. On the left hand-column are shown the four cases for the two edges (u, v) and (v, w) *as they appear in the residual graph*; forward edges are indicated in black (and labeled F) and backward edge in blue (labeled B). On the right-hand column are shown the corresponding four cases for the edges *as they appear in the actual flow network graph*, labeled with the appropriate change to the traffic: $+b$ when the traffic is increased (forward edge in the residual graph) or $-b$ when the traffic is decreased (backward edge in the residual graph).

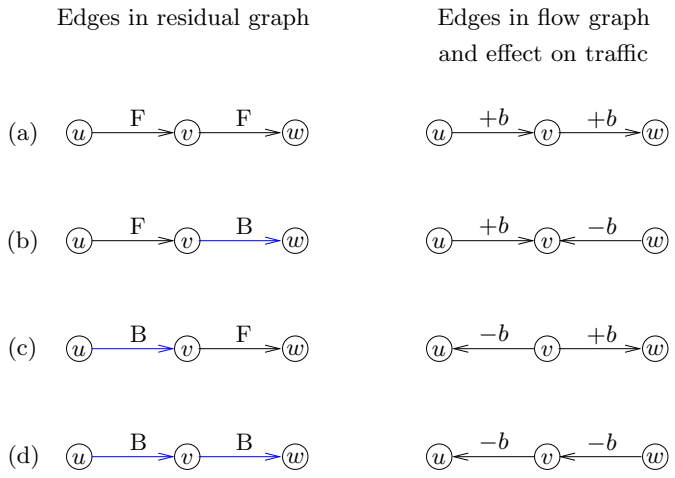


Figure 5.6: The four cases for node v on the augmenting path

It is easy to see from this figure that the net effect of the augmenting step on the traffic through v is zero: An increase in the traffic going into v is balanced by either an increase in the traffic going out of v (case (a)) or a decrease in the traffic going into v (case (b)); and a decrease in the traffic going out of v is balanced by either an increase in the traffic going out of v (case (c)) or a decrease in the traffic going into v (case (d)).

Since f satisfies the conservation property at v and, as we have seen, the changes effected by the augmenting step cancel each other, f' also satisfies the conservation property at v.

We just proved that $f' = \textsc{Augment}(f, p)$ is a flow. As a result of the augmenting step the traffic on one edge out of s, namely the edge on the path p, increases by b and the traffic on all other edges out of s does not change. So $\mathcal{V}(f') = \mathcal{V}(f) + b > \mathcal{V}(f)$, since b is positive. □

5.3 The Ford-Fulkerson algorithm

Once you have understood how (and why) augmenting steps work, the Ford-Fulkerson algorithm is easy to state: We start with the trivial flow where every edge has zero traffic; this obviously satisfies the capacity and conservation constraints. We then keep improving the flow by applying augmenting steps, until there is no augmenting step to apply: the residual graph has no $s \rightsquigarrow t$ path. Then we stop and return the final flow: it is guaranteed to be a maximum flow!

The algorithm in pseudocode is shown below.

```
FordFulkerson(F)
1 for each edge e of the flow network graph G do f(e) := 0
2 construct the residual graph F_f of flow f
3 while F_f has an s ⇝ t path do
4     p := a simple s ⇝ t path in F_f   ▶ p can be chosen in any way; more on this in Section 5.3.3
5     f := Augment(f, p)
6     update the residual graph F_f
7 return f
```

5.3.1 Termination and running time

We say that flow f is ***integral*** if $f(e)$ is an integer for every edge e. For the rest of this chapter, let $C = \sum_{e \in \text{out}(s)} c(e)$ be the total capacity of the edges out of the source. By Corollary 5.2, this is an upper bound on any flow and, in particular, of the maximum flow.

Theorem 5.5 *If the capacities of the edges are integers, then the Ford-Fulkerson algorithm terminates after at most C iterations and returns an integral flow.*

Proof. The following invariant can be shown by an easy induction and using Theorem 5.4:

> At the end of each iteration (a) the function f is an integral flow and
> (b) the residual capacities of $\mathcal{F}_f$ are positive integers.

By (b) the minimum residual capacity on the augmenting path chosen in each iteration of the loop in lines 3-5 is at least one. So the value at the end of each iteration increases by at least one. After at most $C = \sum_{e \in \text{out}(s)} c(e)$ iterations, $\mathcal{V}(f) \geq C$. By Corollary 5.2, $V(f)$ cannot exceed C, so the loop halts after at most C iterations; and by (a) the function f that the algorithm returns is a flow. (We don't yet know that it is a maximum flow; we will prove this in the next section.) □

Perhaps surprisingly, if the capacities are *irrational* real numbers, the Ford-Fulkerson algorithm may not terminate: It is possible to choose the augmenting paths in step 4 so that the algorithm does not terminate. Furthermore, although the value of the flow improves in each iteration, the sequence of values of the flows computed in this pathological choice of augmenting paths converges to a value that can be arbitrarily smaller than the maximum flow of the network; see Exercise 5.4. From now on, and unless otherwise specified, we will assume that

all edge capacities are integers

and therefore the Ford-Fulkerson algorithm terminates.

Let's now analyze the running time of the algorithm. To do so we must know something more about how the various objects it manipulates (graphs, flows, paths) are represented. We will assume that the flow network graph and the residual graph are represented in adjacency list form. We can store information about each edge, specifically its capacity and the amount of

traffic on it, with the edge itself in its adjacency list. The residual graph is likewise represented in adjacency list form. Note that, in addition to storing the residual capacity, each edge of the residual graph must have a bit to indicate whether it is a forward or backward edge; this information is needed by the AUGMENT procedure. We can represent a path p as a linked list of the edges that appear on p.

With these assumptions in mind, it is easy to see that the initialization of the flow f (line 1) takes $O(m+n)$ time, as does the construction of the initial residual graph, where all edges are forward and have residual capacity equal to the capacity of the corresponding edge (line 2). The path p (line 4) can be found in $O(m+n)$ time using breadth- or depth-first search. The algorithm AUGMENT(f,p) (line 5) can be implemented in $O(m)$ time, as does the updating of the residual graph (line 6). So, each iteration of the loop takes $O(m+n)$ time, and since the loop is repeated up to C times, the running time of the algorithm is $O\big((m+n)C\big)$. Under the reasonable assumption that every node is on some $s \rightsquigarrow t$ path (otherwise it is useless as regards carrying traffic from s to t), we have that $n \in O(m)$, and so the running time of the algorithm can be simplified to $O(mC)$.

Theorem 5.5 gives an upper bound on the number of iterations of the while loop in the Ford-Fulkerson algorithm. But is this a *tight* bound? Is there a flow network that can cause the algorithm to execute C iterations of the loop? The answer is yes, as the flow network in Figure 5.7 demonstrates.

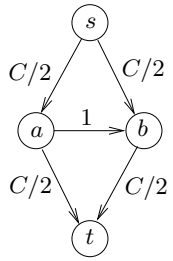


Figure 5.7: Flow network to demonstrate long sequence of augmentations

If the algorithm chooses repeatedly the augmenting paths

- s, a, b, t
- s, b, a, t

in this order, each augmenting step increases the value of the flow by just one unit, so it will take C augmenting steps before the algorithm halts. Therefore, the upper bound on the running time of the algorithm that we derived above is tight: The Ford-Fulkerson algorithm runs in $\Theta(mC)$ time, if augmenting paths can be chosen arbitrarily.

Note that this is *not* a polynomial-time algorithm! Suppose that one or more of the capacities is $\Theta(C)$ (as in the the flow network of Figure 5.7). Then the size of the input is $\Omega(m + \log C)$. Note that C is exponential in $\log C$, so the $\Theta(mC)$ running time is not polynomial. As we saw in our discussion of the running time of the dynamic programming algorithm for knapsack (Section 3.5, page 86), it is a *pseudopolynomial* algorithm: Polynomial

in the *value*, rather than the *size*, of the input.

Fortunately, this is something that can be fixed: In section 5.3.3 we will see strategies for selecting augmenting paths which ensure that the Ford-Fulkerson algorithm runs in polynomial time.

5.3.2 Correctness and the max-flow min-cut theorem

We now prove that the Ford-Fulkerson algorithm is correct. In doing so, we also prove an important result, known as the "max-flow min-cut theorem".

Theorem 5.6 *The Ford-Fulkerson algorithm returns a maximum flow.*

PROOF. Let f be the flow that the algorithm returns, and $\mathcal{F}_f$ be the residual graph of f with respect to the input flow network $\mathcal{F}$. By the exit condition of the while loop, $\mathcal{F}_f$ has no $s \rightsquigarrow t$ path. Let $S = \{u$: there is an $s \rightsquigarrow u$ path in $\mathcal{F}_f\}$ and $T = V - S$ be the rest of the nodes of $\mathcal{F}_f$. Clearly $s \in S$ and $t \in T$, so (S, T) is a cut of the flow network $\mathcal{F}$. We claim that for every edge (u, v) of the graph G of the flow network (note: the edge (u, v) is in G, not in the residual graph $\mathcal{F}_f$!)

$$\text{if } u \in S \text{ and } v \in T \text{ then } f(u,v) = c(u,v), \text{ i.e., } (u,v) \text{ is saturated by } f; \text{ and} \tag{5.1}$$

$$\text{if } u \in T \text{ and } v \in S \text{ then } f(u,v) = 0, \text{ i.e. } (u,v) \text{ is not used by } f. \tag{5.2}$$

This is because if either statement is false, we reach a contradiction:

- If (5.1) is false, i.e., $u \in S$, $v \in T$ and $f(u, v) < c(u, v)$, then the residual graph $\mathcal{F}_f$ has a forward edge (u, v) and since $u \in S$, so is v, contradicting that $v \in T$.
- If (5.2) is false, i.e., $u \in T$, $v \in S$ and $f(u, v) > 0$, then the residual graph $\mathcal{F}_f$ has a backward edge (v, u) and since $v \in S$, so is u, contradicting that $u \in T$.

Then by Theorem 5.1, which relates the value of a flow to the traffic on the edges that cross any cut, we have

$$\begin{aligned}
\mathcal{V}(f) &= \sum_{e \in \text{out}(S) \cap \text{in}(T)} \underbrace{f(e)}_{\substack{=c(e) \\ \text{by (5.1)}}} - \sum_{e \in \text{out}(T) \cap \text{in}(S)} \underbrace{f(e)}_{\substack{=0 \\ \text{by (5.2)}}} \\
&= \sum_{e \in \text{out}(S) \cap \text{in}(T)} c(e) \\
&= c(S, T).
\end{aligned}$$

So, the value of flow f is equal to the capacity of cut (S, T). By Corollary 5.3, f is a maximum flow (and (S, T) is a minimum cut). □

From this proof we immediately also get:

Corollary 5.7 (Max-flow-min-cut Theorem) *In any flow network $\mathcal{F}$, the value of a maximum flow is equal to the capacity of a minimum cut.*

Technically the proof of Theorem 5.6, which implies this corollary, relies on the fact that the Ford-Fulkerson algorithm terminates, and therefore on our assumption that the capacities of the flow network are integers. It is not hard to prove, using essentially the same argument, that the Max-Flow-Min-Cut theorem is true even if the capacities of the flow network are reals and the Ford-Fulkerson algorithm is not guaranteed to terminate. Do you see why?

Observe that the Ford-Fulkerson algorithm, in addition to giving us a maximum flow f, also gives us, essentially for free, a minimum cut (S, T). All we have to do is run a breadth- or depth-first search on the final residual graph (which was in fact already done when the algorithm ended the loop because it found that there is no $s \rightsquigarrow t$ path) to find the set S of nodes reachable from s and let T be the rest of the nodes.

5.3.3 Speeding up the Ford-Fulkerson algorithm

Recall that C is the sum of the capacities out of the source, n is the number of nodes, and m is the number of edges. We are assuming that all nodes are on an $s \rightsquigarrow t$ path, and so $n \in O(m)$

We have seen that careless choice of augmenting paths can result in the Ford-Fulkerson algorithm requiring C augmenting steps. This is not a problem if the capacities are constants, independent of the size of the network, which is the case in some applications as we will see in the next two sections. But if the capacities can be arbitrary integers, with values much larger than the size of the network, this leads to the algorithm having only *pseudo*polynomial running time. It turns out that there are ways to choose augmenting paths in ways that ensure that the number of augmenting steps is polynomial in the size $O(m)$ of the network *and* $\log C$ (the size of the representation of C, rather than its value). This results in versions of the Ford-Fulkerson algorithm that run in polynomial time.

Choosing "wide" paths

You will have noticed that in the example of Figure 5.7 we keep choosing an augmenting path with minimum residual capacity 1, even though there are augmenting paths with much larger residual capacity that would have led to a much faster termination (after only two augmenting steps). This suggests choosing augmenting paths that increase maximally the value of the flow. We define the ***width*** of an $s \rightsquigarrow t$ path p in a residual graph to be the *minimum* residual capacity of any edge on p. Note that this is the amount by which the flow f is improved if we augment it along p. Instead of picking an arbitrary augmenting path, we will choose an augmenting path of *maximum width* — i.e., a path that improves the value of the flow as much as is possible in a single augmenting step.

It turns out that if we always choose the augmenting path in this way, the Ford-Fulkerson algorithm will finish in at most $O(m \log C)$ iterations, resulting in a polynomial-time algorithm; see Exercise 5.5. But how do we find a path with maximum width? It is possible to do so using a variation of Dijkstra's algorithm; see Exercise 4.2. This, however, slows down each augmenting step: Instead of taking $O(m)$ time for a depth- or breadth-first search to find an $s \rightsquigarrow t$ simple path, we need $O(m \log n)$ time to find the one with maximum width. This makes

the running time of the algorithm $O(m^2 \log C \log n)$.

This is polynomial but we can do a bit better by doing a *less* thorough job in selecting the augmenting path: instead of looking for the widest augmenting path we will look for one that is merely "wide enough". Here is the idea. We will set a lower limit w on the width of paths we want to use for augmenting steps, and we will look for $s \rightsquigarrow t$ paths in the residual graph that have width at least w. This can be done quickly with a breadth- or depth-first search that simply ignores edges with residual capacity below w. When no such paths of width at least w exist in the residual graph we reduce w by half, and continue looking for paths of width at least (the new value of) w. We can take the initial value of w chosen to be C, an easy (and likely loose) upper bound on the width of augmenting paths. Since w decreases exponentially, after at most $\log_2 C$ halvings of w, we will be looking at augmenting paths of width 1. When we run out of such paths, there is no $s \rightsquigarrow t$ path in the residual graph and we stop.

We will show that the algorithm stops after at most $m \log C$ augmenting steps. Each augmenting step now costs only $O(m)$, giving us a version of the Ford-Fulkerson algorithm that runs in $O(m^2 \log C)$ time; we got rid of the $\log n$ factor in the running time of the version of the algorithm that attempts to find the widest path each time, rather than being satisfied with a path that is "wide enough".

Digression It is an interesting observation (and something to keep in mind because it is an idea that is useful in other instances) that sometimes it pays to do a job less thoroughly if we can do it quickly than to do a thorough job but more slowly. It is crucial to get the balance right, however. After reading the running-time analysis of the algorithm (Theorem 5.8) see Exercise 5.6 for different ways of striking the balance (not necessarily getting it right).
End of Digression

We now describe the algorithm in more detail and analyze its running time. For any integer w and flow f, let $\mathcal{F}_f^w$ be the residual graph of flow f with all edges that have capacity less than w removed. (In practice we don't actually build $\mathcal{F}_f^w$ as a separate structure; we use $\mathcal{F}_f$ and just pretend that edges with residual capacity less than w do not exist.) The algorithm is shown in pseudocode in Figure 5.8.

For the correctness of this algorithm it suffices to note that it is just the Ford-Fulkerson algorithm where we have restricted how augmenting paths are chosen. Since the Ford-Fulkerson algorithm is correct *no matter how* augmenting paths are chosen, so is this version. Next we will analyze its running time.

Theorem 5.8 *Algorithm* FF-WidePath *runs in* $O(m^2 \log C)$ *time.*

Proof. We divide the execution of the algorithm into ***phases***, based on the value of w, which starts with the value $2^{\lceil \log_2 C \rceil}$ and changes (is halved) at the end of each iteration of the outer loop (lines 4-10). We will denote by w_i the value of w during phase i, starting with phase 0. So, $w_0 = C$, and for $i > 0$, $w_i = \lceil w_{i-1}/2 \rceil$. When $i = \lceil \log_2 C \rceil$, $w_i = 1$, and so this is the last phase. Thus, the number of phases is at most $\lceil \log_2 C \rceil + 1$. In phase 0 the augmenting paths chosen have width at least w_0, and in phase $i > 0$ the augmenting paths

```
FF-WidePath(F)
 1 for each edge e of the flow network graph G do f(e) := 0
 2 construct the residual graph F_f of flow f
 3 w := Σ_{e∈out(s)} c(e)     ▶ take C as the initial lower bound on augmenting path width w
 4 repeat
 5     while F_f^w has an s ⇝ t path do     ▶ augment along paths of width at least w
 6         p := a simple s ⇝ t path in F_f^w
 7         f := Augment(f, p)
 8         update the residual graph F_f
 9     if w = 1 then return f
10     w := max(⌈w/2⌉, 1)        ▶ shrink w
```

Figure 5.8: Pseudocode for the wide-path version of Ford-Fulkerson

chosen have widths in the range $[w_i..w_{i-1}-1]$. Note that some phases may be skipped during the execution, because there are no augmenting paths whose widths are within that phase's range.

Let v^* be the value of a maximum flow. The next claim gives an upper bound on how far from v^* can be the value of the flow at the end of each phase.

Claim 5.8.1 *If f is the flow at the end of phase i, then $\mathcal{V}(f) > v^* - mw_i$.*

Proof of Claim 5.8.1. (As you read this argument, note its similarity to the proof of Theorem 5.6.) Let S be the set of nodes reachable from s in $\mathcal{F}_f^{w_i}$, and T be the rest of the nodes of $\mathcal{F}_f^{w_i}$. At the end of phase i there is no $s \rightsquigarrow t$ path in $\mathcal{F}_f^{w_i}$. Therefore, $s \in S$ and $t \in T$, so (S,T) is a cut of the flow network $\mathcal{F}$. We claim that for every edge (u,v) of the graph G of the flow network:

$$\text{if } u \in S \text{ and } v \in T \text{ then } f(u,v) \geq c(u,v) - w_i \text{ (otherwise } \mathcal{F}_f^{w_i} \text{ has a forward edge } (u,v), \text{ contradicting that } v \in T), \tag{5.3}$$

$$\text{if } u \in T \text{ and } v \in S \text{ then } f(u,v) \leq w_i \text{ (otherwise } \mathcal{F}_f^{w_i} \text{ has a backward edge } (v,u), \text{ contradicting that } u \in T). \tag{5.4}$$

By Theorem 5.1, which relates the value of a flow to the traffic on the edges that cross any cut,

$$\begin{aligned}
\mathcal{V}(f) &= \sum_{e \in \text{out}(S) \cap \text{in}(T)} \underbrace{f(e)}_{\substack{\geq c(e) - w_i \\ \text{by (5.3)}}} - \sum_{e \in \text{out}(T) \cap \text{in}(S)} \underbrace{f(e)}_{\substack{\leq w_i \\ \text{by (5.4)}}} \\
&\geq \sum_{e \in \text{out}(S) \cap \text{in}(T)} \big(c(e) - w_i\big) - \sum_{e \in \text{out}(T) \cap \text{in}(S)} w_i \\
&\geq c(S,T) - mw_i \\
&\geq v^* - mw_i
\end{aligned}$$

For the penultimate step we note that m is an upper bound on the number of edges that cross the cut (S, T); and for the last step we use the fact that the capacity of any cut is at least the value of any flow, in particular the value v^* of the maximum flow. $\square_{5.8.1}$

Claim 5.8.2 *At most $2m$ augmenting steps occur within each phase.*

PROOF. This is clearly true for phase 0, since there can be at most one phase 0 augmenting step: The minimum width in this phase is C, so if two phase 0 augmenting steps occurred, the value of the resulting flow would be at least $2C$, which is impossible since C is the value of a cut and is therefore an upper bound on the value of a maximum flow.

Now consider any phase $i > 0$. Let f', f be the flows at the beginning and the end of this phase. In phase $i-1$, the minimum width was $w_{i-1} \leq 2w_i$. So, by Claim 5.8.1, $\mathcal{V}(f') \geq v^* - (2w_i)m$. During phase i, each augmenting step increases the value of the flow by at least w_i. Suppose, for contradiction, that there are more than $2m$ augmenting steps in phase i, so the value of the flow at the end of phase i is $\mathcal{V}(f) \geq \mathcal{V}(f') + (2m+1)w_i \geq \big(v^* - (2w_i m)\big) + (2m+1)w_i \geq v^* + w_i$. Since $w_i > 0$, this contradicts the definition of v^* as the value of the maximum flow. $\square_{5.8.2}$

Since there are at most $\lceil \log_2 C \rceil + 1$ phases and each of them has at most $2m$ augmenting steps, the algorithm terminates after executing $O(m \log C)$ augmenting steps. Each augmenting step requires $O(m)$ time, so the running time of the algorithm is $O(m^2 \log C)$. $\square$

Choosing short paths

A less intuitive, but also effective, policy is to choose a *shortest* augmenting path, by which here I mean one with the fewest edges. We can find such a path in $O(m)$ time using breadth-first search (starting at s). It turns out that with this choice of augmenting paths, the Ford-Fulkerson algorithm is guaranteed to terminate after at most mn augmentations, *independent of the capacities*. (In contrast, the number of iterations of FF-WIDEPATHS depends on the capacities, hence the $\log C$ factor in its running time.) We will call this version of the Ford-Fulkerson algorithm FF-SHORTESTPATH.

Here is the basic intuition why FF-SHORTESTPATH terminates in a number of augmenting steps that is independent of the capacities. Each augmenting step causes at least one edge to be deleted from the residual graph. It may also cause some edges to be inserted. The deleted edges make shortest paths longer. And, as it turns out, re-inserted edges do not create paths that are shorter than those that already exist. Thus, shortest paths from s to the other nodes get longer and longer. Since a shortest $s \rightsquigarrow t$ path cannot have more than $n-1$ edges, after some number of augmenting steps s and t will become disconnected and the algorithm terminates. A careful counting argument shows that this must happen after no more than mn augmenting steps. This intuition is made precise in the proof of the next theorem.

Theorem 5.9 *The running time of* FF-ShortestPath *is* $O(m^2n)$.

Proof. Let G_k be the residual graph after k augmenting steps of FF-ShortestPath, where G_0 is the residual graph for the initial (empty) flow. Define the ***depth*** of a node u in G_k to be the length of a shortest $s \rightsquigarrow u$ path in G_k (i.e., the depth of u on a breadth-first search tree of G_k rooted at s); we denote this as $\mathbf{depth}_k(u)$.

Claim 5.9.1 *If an edge (u, v) is added to the residual graph during augmenting step $k + 1$ (i.e., (u, v) is not in G_k but is in G_{k+1}), then $\boldsymbol{depth}_k(u) = \boldsymbol{depth}_k(v)+1$.*

Proof. Let (u, v) be an edge that is added to the residual graph during augmenting step $k+1$ and p be the augmenting path of G_k used in that step. Because p is a *shortest* $s \rightsquigarrow t$ path in G_k, for any edge (x, y) on p, $\mathbf{depth}_k(y) = \mathbf{depth}_k(x) + 1$. So it suffices to show that (v, u) is on p. To do so, we consider the two possible ways in which (u, v) was added to the residual graph by the augmenting step $k + 1$.

Case 1. (u, v) was added as a forward edge. This means that the augmenting path p in G_k contained a backward edge (v, u) with residual capacity $c(u, v)$ and no forward edge (u, v). That is, the flow that was augmented in step $k + 1$ had saturated edge (u, v) of the flow network, and so there was no forward edge (u, v) in G_k. Augmenting step $k + 1$ reduced the traffic on the saturated edge (u, v), and this resulted in the creation of forward edge (u, v) in G_{k+1}.

Case 2. (u, v) was added as a backward edge. This means that the augmenting path p in G_k contained a forward edge (v, u) with residual capacity zero and no backward edge (u, v). That is, the flow that was augmented in step $k+1$ had no traffic on edge (v, u) of the flow network, and so there was no backward edge (u, v) in G_k. Augmenting step $k + 1$ increased the traffic on the unused edge (v, u), and this resulted in the creation of bacward edge (u, v) in G_{k+1}.

So, in both cases, (v, u) is an edge on p, as wanted. $\square_{5.9.1}$

Next we show that the depth of a node does not increase as a result of an augmenting step.

Claim 5.9.2 *For every node v, $\boldsymbol{depth}_{k+1}(v) \geq \boldsymbol{depth}_k(v)$.*

Proof. We will prove this by induction on the depth d of a node in G_{k+1}. For the base case $d = 0$, there is only one node with depth 0 in G_{k+1}, namely s, and the depth of s is 0 in all residual graphs, so the claim holds for $d = 0$.

For the induction step, let $d > 0$ and suppose the claim holds for all nodes of depth $d-1$ in G_{k+1}. We will prove that it also holds for all nodes of depth d in G_{k+1}. Consider any node v of depth d in G_{k+1} and let u be its predecessor on some breadth-first search tree of G_{k+1}; u exists because $d > 0$, and $\mathbf{depth}_{k+1}(u) = d - 1$ — so the induction hypothesis applies to u. There are two cases depending on whether the edge (u, v) is in G_k.

CASE 1. (u, v) is in G_k. In this case, we have:

$$\begin{aligned}
\mathbf{depth}_k(v) &\leq \mathbf{depth}_k(u) + 1 && \text{[since } (u,v) \text{ is in } G_k\text{]} \\
&\leq \underbrace{\mathbf{depth}_{k+1}(u)}_{=d-1} + 1 && \text{[by the IH, since } \mathbf{depth}_{k+1}(u) = d-1\text{]} \\
&= \underbrace{\mathbf{depth}_{k+1}(v)}_{=d}
\end{aligned}$$

CASE 2. (u, v) is not in G_k but is added to G_{k+1} by the $(k+1)$-th augmenting step. In this case, we have:

$$\begin{aligned}
\mathbf{depth}_k(v) &< \mathbf{depth}_k(v) + 1 \\
&= \mathbf{depth}_k(u) && \text{[by Claim 5.9.1]} \\
&\leq \underbrace{\mathbf{depth}_{k+1}(u)}_{=d-1} && \text{[by the IH, since } \mathbf{depth}_{k+1}(u) = d-1\text{]} \\
&< \underbrace{\mathbf{depth}_{k+1}(v)}_{=d} && \text{[since } \mathbf{depth}_{k+1}(v) = d\text{]}.
\end{aligned}$$

Note that in Case 2, the inequality of the claim is strict: if (u, v) is added to the residual graph in some augmenting step, its depth increases — and does so by at least 2. $\square_{5.9.2}$

In each augmenting step, at least one edge is deleted from the residual graph, namely every edge on the augmenting path that has minimum residual capacity among the edges on that path. Let (u, v) be an edge that is deleted in augmenting step ℓ and subsequently inserted in autmenting step $k+1$, for some $k \geq \ell$. We have

$$\begin{aligned}
\mathbf{depth}_{\ell-1}(v) &= \mathbf{depth}_{\ell-1}(u) + 1 && \text{[because } (u,v) \text{ is on a shortest } s \rightsquigarrow t \text{ path in } G_{\ell-1}\text{]} \\
&\leq \mathbf{depth}_k(u) + 1 && \text{[by Claim 5.9.2 and induction]} \\
&= \big(\mathbf{depth}_k(v) + 1\big) + 1 && \text{[by Claim 5.9.1]} \\
&\leq \mathbf{depth}_{k+1}(v) + 2 && \text{[by Claim 5.9.2]}
\end{aligned}$$

Thus, the depth of node v increases by at least 2 between an augmenting step ℓ that deleted (u, v) from the residual graph and the subsequent augmenting step $k+1$ that re-inserted it. Since the minimum depth of a node is 0 and the maximum is $n-1$, an edge can be added at most $n/2$ times to the residual graph. Since there are at most $2m$ edges in the residual graph, and each augmenting step deletes at least one edge, after at most $2m(n/2) = mn$ augmenting steps all edges will have been deleted and cannot be inserted (because each of them will have been re-inserted the maximum number of times that it can). Thus, FF-SHORTESTPATH terminates after at most mn augmenting steps. Each augmenting step takes $O(m)$ time, so the total running time of this algorithm is $O(m^2 n)$. $\square$

In our analysis of the running time of FF-SHORTESTPATH we assumed, as usual, that each arithmetic operation can be performed in $O(1)$ time. (The algorithm only adds or subtracts

capacities, so assuming that the capacities are numbers that fit in a word of memory, this is reasonable.) Note that we made the same assumption in our analysis of FF-WidePath, yet the running time of that algorithm depends on the capacities. This is because the number of augmenting steps in that algorithm depends (logarithmically) on C, while the number of augmenting steps of FF-ShortestPath depends only on the number of nodes and edges of the flow network and not on the capacities of the edges. Algorithms whose running time does not depend at all on the numeric values in the input (assuming $O(1)$-time arithmetic operations on numbers, regardless of their size) are called ***strongly polynomial-time algorithms***. Thus, FF-ShortestPath is a strongly-polynomial algorithm, while FF-WidePath is not.

5.4 Maximum matching in bipartite graphs

In the remainder of this chapter we will examine some problems that can be solved efficiently by reducing them to the maximum-flow problem.

A ***bipartite graph*** is an undirected graph $G = (V, E)$ whose set of nodes can be partitioned into two sets X and Y (i.e., $X \cup Y = V$ and $X \cap Y = \varnothing$), so that every edge of G connects a node in X to a node in Y (i.e., for each $\{u, v\} \in E$, $u \in X$ if and only if $v \in Y$). An example of a bipartite graph is shown in Figure 5.9(a) below.

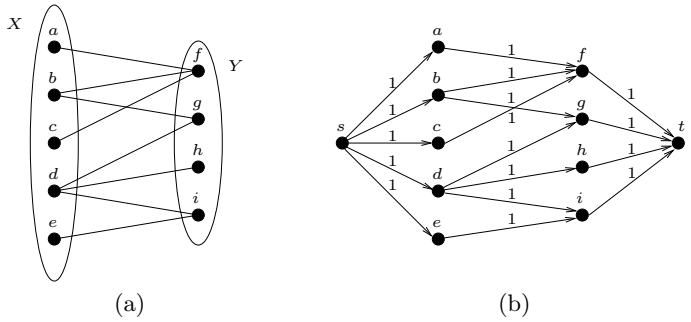


Figure 5.9: A bipartite graph and corresponding flow network

We will assume that when we are given a bipartite graph, we are also given a partition of its set of nodes into X and Y so that the edges of G connect nodes of X to nodes of Y, and we will denote this graph as $G = (\{X, Y\}, E)$. The following facts about bipartite graphs can be proved using graph searches that you learned in previous courses (see Exercise 5.8):

Theorem 5.10 *Let $G = (V, E)$ be an undirected graph.*
(a) There is an algorithm that determines in time $O(|V| + |E|)$ whether G is bipartite and, if so, partitions V into two sets X and Y so that every edge in E connects a node in X to a node in Y.
(b) G is bipartite if and only if it has no cycle of odd length.

A ***matching*** of an undirected graph $G = (V, E)$ is a subset of the edges $M \subseteq E$ so that

no two edges in M share an endpoint; i.e., for all $e, e' \in M$, if $e \neq e'$ then $e \cap e' = \varnothing$. The empty set is obviously a matching. A matching of maximum cardinality is called a ***maximum matching*** of the graph.

We can find a maximum matching of a *bipartite* graph by reducing the problem to finding a maximum flow of a network. To do so we convert the bipartite graph to the flow network defined below.

Definition 5.11 *Let $G = (\{X, Y\}, E)$ be a bipartite graph. The **flow network** $\mathcal{F} = (G', s, t, c)$ **that corresponds to** G is defined as follows:*

- *s and t are new nodes not in $X \cup Y$;*
- *$G' = (V, E')$ is the directed graph with set of nodes $V = X \cup Y \cup \{s, t\}$ and set of edges $E' = \{(s, x)\colon x \in X\} \cup \{(x, y)\colon x \in X,\ y \in Y,\ \text{and } \{x, y\} \in E\} \cup \{(y, t)\colon y \in Y\}$; and*
- *$c(e) = 1$ for each $e \in E'$.*

Figure 5.9(b) shows the flow network that corresponds to the bipartite graph in Figure 5.9(a).

The idea now is to find an integral maximum flow in $\mathcal{F}$ (for example, using the Ford-Fulkerson algorithm); this flow will put one unit of traffic on some of the edges that go from X to Y in $\mathcal{F}$. The set of these edges turns out to be a maximum matching of G. So the algorithm is as follows:

BipartiteMaxMatching(G)
▶ $G = (\{X, Y\}, E)$ is a bipartite graph
1 $\mathcal{F}$:= the flow network that corresponds to G
2 f := FordFulkerson($\mathcal{F}$)
3 $M := \big\{\{x, y\} :\ x \in X,\ y \in Y, \text{ and } f(x, y) = 1\big\}$
4 **return** M

Figure 5.10(a) shows a maximum flow in the flow network that corresponds to the bipartite graph of Figure 5.9(a), where heavy edges have traffic 1 and the other edges have traffic 0. Figure 5.10(b) shows the corresponding maximum matching of the bipartite graph.

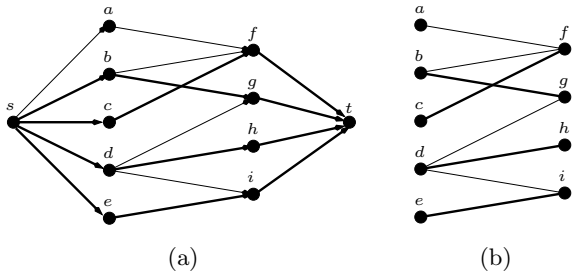


Figure 5.10: Maximum flow and corresponding maximum matching

Let n be the number of nodes and m be the number of edges of G. Line 1 takes $O(m + n)$ time; line 2 takes $O(mn)$ time (since the sum of the capacities of the edges out of s is at most n); line 3 takes $O(m)$ time, and line 4 takes $O(1)$ time. So the running time of this algorithm is $O(mn)$.

We now prove that the algorithm is correct; i.e., it returns a maximum matching of G. This follows from the following fact, which establishes a close relationship between integral flows in $\mathcal{F}$ and matchings in G:

Theorem 5.12 *Let $G = (\{X, Y\}, E)$ be a bipartite graph and $\mathcal{F}$ be the flow network that corresponds to G.*

(a) For any integral flow f of $\mathcal{F}$, the set

$$M_f = \big\{\{x, y\} : \ x \in X, \, y \in Y, \text{ and } f(x, y) = 1\big\}$$

is a matching of G with size $\mathcal{V}(f)$.

(b) For any matching M of G, the function $f_M : E' \to \{0, 1\}$ defined by

$$f_M(e) = \begin{cases} 1, & \text{if there is a path } s, x, y, t \text{ in } \mathcal{F} \text{ that contains } e \text{ and } \{x, y\} \in M \\ 0, & \text{otherwise} \end{cases}$$

is a flow of $\mathcal{F}$ with value $|M|$.

PROOF. (a) To show that M_f is a matching of G, consider any $x \in X$. We claim that there cannot be edges $\{x, y\}$ and $\{x, y'\}$ in M_f, for $y \neq y'$. Suppose, for contradiction, that we such edges exist. Then, by the definition of M_f, $f(x, y) = f(x, y') = 1$. So the traffic out of x in f is at least two, while the traffic into x is at most one (there is only one edge into x and it has capacity one). Therefore f violates the conservation constraint at x, contradicting that f is a flow. By a similar argument we can show that no $y \in Y$ can be the endpoint of two edges in M_f. Therefore M_f is indeed a matching. Finally, by considering the (S, T) cut of $\mathcal{F}$ where $S = \{s\} \cup X$ and $T = \{t\} \cup Y$ and applying the theorem that relates the value of a flow to the traffic across the edges of any cut (Theorem 5.1) we have:

$$\mathcal{V}(f) = \sum_{\text{out}(S) \cap \text{in}(T)} f(e) \ - \underbrace{\sum_{\text{out}(T) \cap \text{in}(S)} f(e)}_{\text{sum=0: no such edges!}} = |\{(x, y) : \ x \in X, \, y \in Y, \text{ and } f(x, y) = 1\}| = |M_f|.$$

(b) It is obvious that f_M satisfies the capacity constraint. For the conservation constraint, consider any node $x \in X$. If $f_M(s, x) = 0$ then there is no $s \rightsquigarrow t$ path s, x, y, t such that $\{x, y\} \in M$, and so $f_M(x, y) = 0$ for all edges (x, y) in $\mathcal{F}$; so conservation holds at x in this case. If $f_M(s, x) = 1$ then there is an $s \rightsquigarrow t$ path s, x, y, t such that $\{x, y\} \in M$; and there is only one such path (otherwise M would contain two edges with x as one endpoint, contradicting that it is a matching). Thus $\sum_{e \in \text{out}(x)} f_M(e) = 1$, and again conservation holds at x. By a similar argument we can show that conservation holds at any $y \in Y$. Therefore f_M is a flow in $\mathcal{F}$. By considering the (S, T) cut of $\mathcal{F}$ as in the proof of part (a) we have:

$$\mathcal{V}(f_M) = |\{(x, y) \text{: } x \in X, \, y \in Y, \text{ and } f_M(x, y) = 1\}| = |M|. \qquad \square$$

To see why this fact implies the correctness of algorithm BipartiteMaxMatching we first note that, by Theorem 5.12(a), the set M returned by the algorithm is a matching of G whose size is the value of the maximum flow f. By Theorem 5.12(b) M must be a maximum matching: If there was a matching M' larger than M, by part (b), there would be a flow f' of $\mathcal{F}$ whose value is $|M'| > |M| = \mathcal{V}(f)$, contradicting that f is a maximum flow of $\mathcal{F}$.

There are more efficient (but also more complicated) algorithms for bipartite matching. In particular, an algorithm by Hopcroft and Karp solves the problem in $O(m\sqrt{n})$ time; it speeds up the simpler algorithm described above by finding multiple augmenting paths at once in a series of phases. The problem of matching in a general (not necessarily bipartite) undirected graph is also solvable in polynomial time. In a landmark 1965 paper, Edmonds gave an $O(n^3)$ algorithm for this problem, which is not based on maximum flows.[2] There are now more efficient algorithms for maximum matching in general graphs. The most efficient such algorithm of which I am aware is by Micali and Vazirani, and runs in $O(m\sqrt{n})$ time.

5.5 Minimum vertex cover in bipartite graphs

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset of the nodes $R \subseteq V$ so that each edge of G has at least one endpoint in R; i.e., for each $\{u, v\} \in E$, $u \in R$ or $v \in R$ (or both). The set of all nodes is obviously a vertex cover. A vertex cover of minimum cardinality is called a ***minimum vertex cover*** of the graph. Finding a minimum vertex cover is an important optimization problem. Unfortunately, it is NP-hard, which means that it is unlikely to be solvable in polynomial time. Luckily, polynomial-time solutions for this problem exist for special types of graphs, including bipartite graphs, as we will see.

The following lemma states that the cardinality of any matching does not exceed that cardinality of any vertex cover.

Lemma 5.13 *For any undirected graph $G = (V, E)$, any matching M, and any vertex cover R of G, $|M| \leq |R|$.*

Proof. Consider any function $\phi : M \to R$ that maps each edge in M to one of its endpoints that is in R (i.e., for each $\{u, v\} \in M$, $\phi(\{u, v\}) = u$ or $\phi(\{u, v\}) = v$ and $\phi(\{u, v\}) \in R$). Such a function exists because R is a vertex cover, and so its elements "touch" every edge of G and in particular the edges in the matching M. Since M is a matching, ϕ is one-to-one. Thus, by the pigeonhole principle, $|M| \leq |R|$. □

Note the similarity of the form of this lemma to the result that the value of any flow is at most the capacity of any cut (Corollary 5.2). This is not a coincidence; both of these facts are

[2] Perhaps the most important reason why this paper is so important is because it identified the class of "efficiently solvable problems" with the problems solvable in polynomial time, contrasting such problems with those that are or appear to be solvable only via brute-force methods that require exponential time.

instances of a more general phenomenon in linear programming (the subject of the next course unit), known as "weak duality".

Lemma 5.13 is true about *all* (undirected) graphs. For bipartite graphs in particular, however, the size of a maximum vertex cover is actually *equal* to the size of a minimum vertex cover. This fact is known as König's Theorem, and its proof below also yields an efficient algorithm to find a minimum vertex cover of a bipartite graph. Note that equality does not necessarily hold for non-bipartite graphs. For example, a triangle (a graph with three nodes, every two of which are connected by an edge) has maximum matching size one, but minimum vertex cover size two.

Theorem 5.14 (König's Theorem) *In any bipartite graph, the cardinality of a maximum matching is equal to the cardinality of a minimum vertex cover.*

PROOF. Let $G = (\{X, Y\}, E)$ be a bipartite graph, $\mathcal{F}$ be the corresponding flow network (Definition 5.11), and f be an integral maximum flow in $\mathcal{F}$. Consider now the residual graph $\mathcal{F}_f$ of the maximum flow f. Since f is an integral flow and the capacity of every edge in $\mathcal{F}$ is 1,

$$f(u,v) = \begin{cases} 0, & \text{if } (u,v) \text{ is a forward edge of } \mathcal{F}_f, \\ 1, & \text{if } (u,v) \text{ is a backward edge of } \mathcal{F}_f, \end{cases} \tag{5.5}$$

Let S be the set of nodes reachable from s in $\mathcal{F}_f$, and T be the set of the remaining nodes. As we saw in the proof of correctness of the Ford-Fulkerson algorithm, (S, T) is a minimum (s,t)-cut of the flow network, and its capacity is equal to the value of f.

Figure 5.11(a) below shows the residual graph of the maximum flow in Figure 5.10(a), along with the partition of the set of nodes into those that are reachable from s in the residual graph (nodes in the green "cloud") and the rest of the nodes (in the red "cloud"). Figure 5.11(b) shows the minimum cut of the flow network in Figure 5.9(b) that is derived from this partition of the nodes of the residual graph into the "green" and "red" clouds. Next we show that there is no edge in the flow network from a "green" node in X to a "red" node in Y.

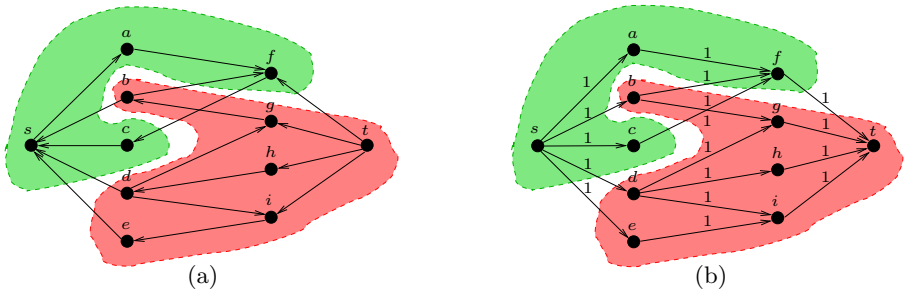


Figure 5.11: Residual graph and minimum cut of the flow network in Figure 5.9(b)

Claim 5.14.1 *There is no edge (x, y) in $\mathcal{F}$ such that $x \in (X \cap S)$ and $y \in (Y \cap T)$.*

PROOF OF CLAIM 5.14.1. Suppose, for contradiction, that there exist $x \in X \cap S$ and $y \in Y \cap T$ such that (x, y) is an edge in the flow network $\mathcal{F}$. Then (y, x) is a backward edge in $\mathcal{F}_f$ (otherwise (x, y) would be a forward edge and since x is reachable from s so would y, meaning that $y \in S$ and contradicting that $y \in T$). Thus, by (5.5),

$$f(x, y) = 1. \tag{5.6}$$

Since $x \in S$, there is an $s \rightsquigarrow x$ path p in the residual graph $\mathcal{F}_f$. There are two cases, both leading to contradiction.

CASE 1. $p = s, x$. So (s, x) is a forward edge in $\mathcal{F}_f$, and by (5.5) $f(s, x) = 0$. In view of (5.6), f violates the conservation property at x: the inflow is 0, but the outflow is at least 1.

CASE 2. $p = s, x_1, y_1, x_2, y_2, \ldots, x_k, y_k, x$, where $x_1, \ldots, x_k \in X$ and $y_1, \ldots y_k \in Y$. All these nodes are in S (since they are reachable from s in $\mathcal{F}_f$), so $y_k \neq y$. The edge (y_k, x) is a backward edge in $\mathcal{F}_f$, so by (5.5) $f(x, y_k) = 1$. But then, in view of (5.6) and the fact that $y_k \neq y$, f violates the conservation property at x: the inflow is at most 1 but the outflow is at least 2. □ Claim 5.14.1

Claim 5.14.1 implies that

$$\text{the set } R = (X \cap T) \cup (Y \cap S) \text{ is a vertex cover of the bipartite graph } G.^3 \tag{5.7}$$

This is because $S \cup T \supseteq X \cup Y$, and so every edge that connects a node in X to a node in Y in G is covered by some node in R, except for edges that connect a node in $X \cap S$ to a node in $Y \cap T$ — but, by Claim 5.14.1, no such edge exists!

Claim 5.14.1 also implies that the only edges in the flow network $\mathcal{F}$ that enter T from S are edges of the form (s, x) such that $x \in X \cap T$, and edges of the form (y, t) such that $y \in Y \cap S$. (The only other possibility are edges (x, y) with $x \in X \cap S$ and $y \in Y \cap T$ but, by Claim 5.14.1, no such edge exists.) Since all edges have capacity 1,

$$c(S, T) = |(X \cap T) \cup (Y \cap S)| = |R|. \tag{5.8}$$

By Theorem 5.12,

$$\begin{aligned} &M = \big\{\{x, y\} \colon x \in X,\ y \in Y, \text{ and } f(x, y) = 1\big\} \text{ is a maximum matching of } G, \\ &\text{and } |M| = \mathcal{V}(f). \end{aligned} \tag{5.9}$$

By the max-flow-min-cut theorem, $\mathcal{V}(f) = c(S, T)$. Together with (5.8) and (5.9), this implies that $|M| = |R|$. So we have a matching M whose size is equal to the size of vertex cover R, and so by Lemma 5.13, R is a minimum vertex cover (and M is a maximum matching). □

The preceding proof of König's Theorem gives rise to the following algorithm for finding a minimum vertex cover of a bipartite graph.

[3] In terms of Figure 5.11, these are the "red" nodes of X (i.e., b, d, e) and the "green" nodes of Y (i.e., f).

```
BipartiteMinVC(G)
▷ G = ({X, Y}, E) is a bipartite graph
1 F := the flow network that corresponds to G
2 f := FordFulkerson(F)
3 F_f := the residual graph of flow f with respect to network F
4 S := {u : u is a node reachable from s in F_f}
5 T := {u : u is a node in F_f not in S}
6 return (X ∩ T) ∪ (Y ∩ S)
```

Let n be the number of nodes and m be the number of edges of G. Line 1 takes $O(m+n)$ time; line 2 takes $O(mn)$ time; line 3 takes $O(m+n)$ time; line 4 takes $O(m+n)$ time (say, using depth-first search); and lines 5 and 6 take $O(n)$ time. Thus, we can find the minimum vertex cover of a bipartite graph in $O(mn)$ time.

As we saw in Sections 5.4 and 5.5, maximum matchings and minimum vertex covers are closely related for bipartite graphs. Both can be found efficiently (in polynomial time) by applying maximum flow techniques. This close relationship does not extend to all graphs, however: Maximum matchings in general graphs can be found in polynomial time, using algorithms that are beyond the scope of this course. In contrast, the minimum vertex cover problem for general graphs is NP-hard, and is therefore unlikely to be solvable in polynomial time.

5.6 Hall's Theorem

A matching M of an undirected graph $G = (V, E)$ is ***perfect*** if it does not leave any node unmatched; i.e., for every $u \in V$ there is some $v \in V$ such that $\{u, v\} \in M$. A perfect matching is obviously a maximum matching, but not every graph has a perfect matching — for example, a graph with an odd number of nodes cannot have a perfect matching, but of course it has a maximum matching.

We now turn our attention again specifically to bipartite graphs. When does a bipartite graph $G = (\{X, Y\}, E)$ have a perfect matching? Clearly, we must have $|X| = |Y|$: otherwise every matching must leave unmatched some node in X or Y (whichever is larger). This, however, is not enough. For example, the graph in Figure 5.12 does not have a perfect matching even though each of X and Y has three nodes: Node d can only be matched with a or b; but if it is matched with one, the other cannot be matched with any other node.

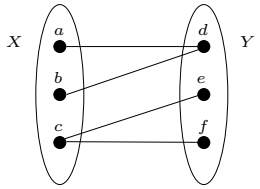


Figure 5.12: A bipartite graph with no perfect matching

This example can be generalized: Let the ***neighbours*** of a node u be the set of nodes $N(u)$ that are adjacent to it; i.e., $N(u) = \{v : \{u, v\} \in E\}$. The neigbours $N(U)$ of a set of nodes U is the union of the neighbours of the nodes in U; i.e., $N(U) = \cup_{u \in U} N(u)$. For G to have a perfect matching, there must be no $X' \subseteq X$ such that $|X'| > |N(X')|$: if such a set exists, then every matching must leave some node in X' unmatched.

It turns out that this is not only a necessary, but also sufficient, condition for a bipartite graph with $|X| = |Y|$ to have a perfect matching. This fact is known as Hall's Theorem, and one way of proving it follows from our discussion of minimum vertex covers in bipartite graphs.

Theorem 5.15 (Hall's Theorem) *A bipartite graph $G = (\{X, Y\}, E)$ such that $|X| = |Y|$ has a perfect matching if and only if there is no $X' \subseteq X$ such that $|X'| > |N(X')|$.*

Proof. [Only if] Suppose, for contradiction, that G has a perfect matching M but there is some $X' \subseteq X$ such that $|X'| > |N(X')|$. Let $\phi : X' \to N(X')$ be any function that maps $x \in X'$ to the neighbour with which it is matched in M; i.e., for every $x \in X'$, $\{x, \phi(x)\} \in M$. Such a function ϕ exists because M is a perfect matching. Since $|X'| > |N(X')|$, by the pigeonhole principle, the function is not one-to-one; this contradicts that M is a matching.

[If] We prove the contrapositive: If G has no perfect matching then there is some $X' \subseteq X$ such that $|X'| > |N(X')|$.

Suppose G has no perfect matching. Let M be a maximum matching and R be a minimum vertex cover of G. By König's Theorem, $|R| = |M|$, and because G has no perfect matching, $|M| < |X|$; so $|R| < |X|$. Note that $R = (X \cap R) \cup (Y \cap R)$ and $X = (X \cap R) \cup (X - R)$, where both unions are of disjoint sets. Therefore, $|X \cap R| + |Y \cap R| < |X \cap R| + |X - R|$. Simplifying, we get

$$|Y \cap R| < |X - R| \tag{5.10}$$

Since R is a vertex cover of G, there is no edge between a node in $X - R$ and a node in $Y - R$; thus $N(X - R) \subseteq Y \cap R$ and so $|N(X - R)| \leq |Y \cap R|$. Combining with (5.10), we get $|N(X - R)| < |X - R|$. So, the set $X - R$ is a subset X' of X with the wanted property that it has more nodes than it has neighbours. □

5.7 Maximum cardinality set of disjoint paths

In this section we will study algorithms for the following problem. We are given a graph (directed or undirected — the same basic ideas apply with some twists, as we will see in Section 5.7.3) and two nodes s and t. We want to find a set of paths connecting s and t so that (a) no two paths in the set use the same edge, and (b) the set has maximum cardinality among all sets that satisfy (a). This problem arises in various settings. For example, in fault-tolerant distributed systems we want to design algorithms that work even if some communication links fail. So we want to send a message from s to t along multiple routes *that share no edges* to ensure that the message arrives despite the failure of some communication links.

The algorithms we will study are based on a more general theorem that has other applications, called the ***flow decomposition theorem***, with which we start our discussion.

5.7.1 Flow decomposition

First some terminology that applies to both directed and undirected graphs. A ***simple path*** $u_1, u_2, \ldots u_k$ is a path in which no node is repeated, except possibly the first and last; i.e., for all $i, j \in [1..k]$ such that $i < j$, if $u_i = u_j$ then $i = 1$ and $j = k$. A ***simple cycle*** is a simple path that is a cycle; i.e., a simple path in which the first and last nodes are the same. For example, in Figure 5.13, a, b, c, d, e is a simple path; while a, c, d, e, c, b is not. The cycle a, b, c, a is a simple cycle, while a, b, c, d, e, c, b, a is not.

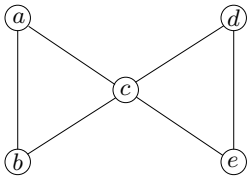


Figure 5.13: Simple and non-simple paths and cycles

Informally the flow decomposition theorem states that *any* flow can be viewed as the sum of "primitive" flows, of the following two types:

- path flows, where the edges used by each flow form a simple $s \rightsquigarrow t$ path and they all carry the same amount of traffic; or
- cyclic flows, where the edges used by each flow form a simple cycle and they all carry the same amount of traffic.

For example, the flow in Figure 5.14(a) can be decomposed into the two path flows and one cyclic flow shown in Figure 5.14(b). The edges of the first path flow carry one unit of traffic each, the edges of the second path flow carry three units of traffic each, and the edges of the cyclic flow carry two units of traffic each. Note that the traffic on the cyclic flow contributes nothing to the value of the flow; this is just traffic going around a cycle, not traffic that gets transported from s to t. Also note that the paths or cycles into which a flow can be decomposed may share edges; in our example, edge (c, d) is on both path flows and the cyclic flow of the decomposition.

A ***binary flow*** is a flow f such that for every edge e, $f(e) = 0$ or $f(e) = 1$ — i.e., every edge is either not used or carries one unit of flow. If p is a simple $s \rightsquigarrow t$ path, f_p is the binary flow such that $f_p(e) = 1$ if e is on p and $f_p(e) = 0$ otherwise. If c is a simple cycle, f_c is the binary flow such that $f_c(e) = 1$ if e is on c and $f_c(e) = 0$ otherwise.

Theorem 5.16 *For any flow f, there exist a set $\mathcal{P}$ of $s \rightsquigarrow t$ simple paths and a set $\mathcal{C}$ of simple cycles, and positive constants d_p for each $p \in \mathcal{P}$ and d_c for each $c \in \mathcal{C}$ such that*

$$f = \sum_{p \in \mathcal{P}} d_p f_p + \sum_{c \in \mathcal{C}} d_c f_c.$$

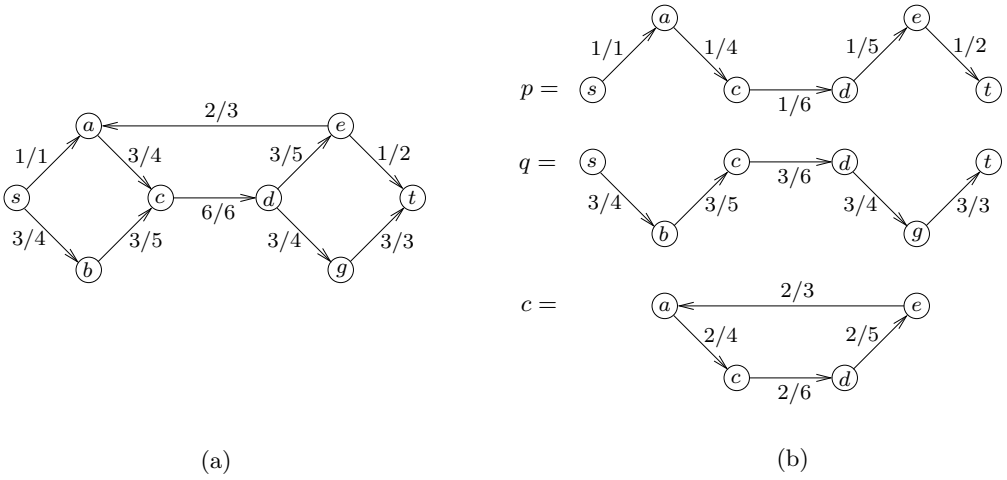


Figure 5.14: Illustration of the flow decomposition theorem

Furthermore, if f is an integral flow then there exist positive integer constants d_p and d_c that satisfy the above equality.

This statment looks a bit daunting so let's interpret Figure 5.14 in terms of this theorem. In this example we have $\mathcal{P} = \{p, q\}$ where $p = s, a, c, d, e, t$ and $q = s, b, c, d, g, t$, and $\mathcal{C} = \{r\}$, where $r = a, c, d, e, a$; $d_p = 1$, $d_q = 3$, and $d_r = 2$. You can check that $f = f_p + 3f_q + 2f_r$. Note that in this example the paths (and the cycle) share edges.

Proof of Theorem 5.16. We will prove this by complete induction *on the number k of edges used by f*; i.e., $k = |\{e\colon f(e) > 0\}|$. The base case, $k = 0$, is clearly true by taking $\mathcal{P} = \mathcal{C} = \varnothing$. For the induction step, let $k > 0$, and suppose that the theorem holds if the flow uses fewer than k edges.

Now define a path $\pi = u_0, u_1, u_2, \ldots, u_k$ as follows: If there is an edge (s, u) such that $f(s, u) > 0$, let $u_0 = s$ and $u_1 = u$; otherwise let u_0 and u_1 be any nodes such that $f(u_0, u_1) > 0$ (such nodes must exist since $k > 0$). By the conservation property of flows, since $f(u_0, u_1) > 0$, there is some edge (u_1, u_2) such that $f(u_1, u_2) > 0$, and an edge (u_2, u_3) such that $f(u_2, u_3) > 0$, and so on, until one of two things happens: either
(a) $u_k = t$, i.e., the path reaches the node t, or
(b) $u_k = u_i$ for some $i \in [0..k-1]$, i.e., the path π loops back to a node in the path.
We consider each of these cases separately.

In case (a), first we note that π is a path that starts at $u_0 = s$. This is because by Theorem 5.1 (applied to cut (S, T) with $S = V - \{t\}$ and $T = \{t\}$), $\mathcal{V}(f) = \sum_{e \in \text{in}(t)} f(e) \geq f(u_{k-1}, u_k) > 0$. Since the value of f is positive, there must be some edge out of s that carries traffic. So π is a simple $s \rightsquigarrow t$ path. Let b be the minimum traffic on any edge on path π in flow f. Note that if f is integral, b is a positive integer. Now let $f'\colon E \to \mathbb{R}$ be the function

defined by

$$f'(e) = \begin{cases} f(e) - b, & \text{if } e \text{ is on path } \pi \\ f(e), & \text{otherwise.} \end{cases}$$

It is easy to check that f' satisfies the capacity and conservation constraints (do it!). Furthermore, if f is integral, since b is an integer, f' is also integral. Clearly f' uses fewer than k edges, since all edges on π that carried traffic b now carry no traffic. So, by the induction hypothesis, f' can be decomposed: There is a set $\mathcal{P}'$ of simple $s \rightsquigarrow t$ paths and a set $\mathcal{C}'$ of simple cycles, and positive constants d_p for each $p \in \mathcal{P}'$ and d_c for each $c \in \mathcal{C}'$ so that $f' = \sum_{p \in \mathcal{P}'} d_p f_p + \sum_{c \in \mathcal{C}'} d_c f_c$. Then let $\mathcal{P} = \mathcal{P}' \cup \{\pi\}$, $d_\pi = b$, and $\mathcal{C} = \mathcal{C}'$. It is easy to see that $f = \sum_{q \in \mathcal{P}} d_q f_q + \sum_{c \in \mathcal{C}} d_c f_c$, as wanted. As we have seen, if f is integral so is f'; by the induction hypothesis in this case the constants d_p, d_c for every $p \in \mathcal{P}'$ and every $c \in \mathcal{C}'$ are integers. Since, in this case, $d_\pi = b$ is also an integer, we have that d_p and d_c for every $p \in \mathcal{P}$ and $c \in \mathcal{C}$ are integers, as wanted.

In case (b), consider the cycle $\sigma = u_i, u_{i+1}, u_{i+2}, \ldots, u_{k-1}, u_i$. We proceed in the same way as in case (a), except that now we construct the flow f' by removing b units of traffic from every edge on σ (rather than π, as in case (a)). We then leverage the induction hypothesis to get a decomposition of f' into a set of simple paths and simple cycles, and extend this to a decomposition of f by adding σ to the set of cycles. You should be able to work out the details on your own. □

The proof of this theorem immediately gives rise to a recursive algorithm that, given the flow f as input, produces as output a set of paths $\mathcal{P}$, a set of cycles $\mathcal{C}$, and constants d_p, d_c for each $p \in \mathcal{P}$ and $c \in \mathcal{C}$, that satisfy the statement of the decomposition theorem. The algorithm is shown in pseudocode in Figure 5.15. The output is in the form of two sets of pairs, *Paths* and *Cycles*, where each pair in *Paths* consists of a simple $s \rightsquigarrow t$ path p and its associated constant d_p, and each pair in *Cycles* consists of a simple cycle c and its associated constant d_c, where the collection of paths and cycles, along with their associated constants, form a decomposition of the input flow f.

The running time of FlowDecomposition is $O(mn)$. This is because, using straighforward data structures to represent the flow, each recursive call takes $O(n)$ time (exclusive of the time required by the recursive call it makes on line 14 or 20), and there are at most m recursive calls (since in each call at least one edge no longer carries traffic in the flow); see Exercise 5.9.

```
FlowDecomposition(f)
 1 if for every edge e, f(e) = 0 then return (∅, ∅)
 2 else
 3     if for some u ∈ out(s), f(s,u) > 0 then u_0 := s
 4     else u_0 := any node such that for some u ∈ out(u_0), f(u_0,u) > 0
 5     k := 0
 6     π := u_0    ▶ Start the path π at u_0
 7     while true do    ▶ Invariant: there is some u such that f(u_k,u) > 0
 8         k := k + 1
 9         u_k := any node such that f(u_{k-1},u_k) > 0
10         append u_k to π
11         if u_k = t then
12             b := min{f(u_j,u_{j+1}): j ∈ [0..k-1]}
13             for j := 0 to k-1 do f(u_j,u_{j+1}) := f(u_j,u_{j+1}) - b    ▶ reduce traffic on path π
14             (Paths, Cycles) := FlowDecomposition(f)
15             return (Paths ∪ {(π,b)}, Cycles)
16         elsif for some i ∈ [0..k-1], u_k = u_i then
17             σ := u_i, u_{i+1}, ..., u_k
18             b := min{f(u_j,u_{j+1}): j ∈ [i..k-1]}
19             for j := i to k-1 do f(u_j,u_{j+1}) := f(u_j,u_{j+1}) - b    ▶ reduce traffic on cycle σ
20             (Paths, Cycles) := FlowDecomposition(f)
21             return (Paths, Cycles ∪ {(σ,b)})
22         else continue    ▶ there is some u such that f(u_k,u) > 0
```

Figure 5.15: The flow decomposition algorithm

5.7.2 Directed graphs

Let us now turn to the problem of finding a maximum cardinality set of edge-disjoint paths between two nodes in a directed graph. More precisely the problem is:

Input: A directed graph $G = (V, E)$, and nodes $s, t \in V$.
Output: A set $\mathcal{P}$ of $s \rightsquigarrow t$ *simple* paths such that

- **Feasibility:** No two paths in $\mathcal{P}$ have an edge in common.
- **Optimality:** If $\mathcal{P}'$ is a set $s \rightsquigarrow t$ paths no two of which have an edge in common, then $|\mathcal{P}| \geq |\mathcal{P}'|$.

Note that we have restricted our interest to $s \rightsquigarrow t$ edge-disjoint *simple* paths: Allowing for non-simple paths does not increase the maximum number of edge-disjoint paths between two nodes: Given a set of $s \rightsquigarrow t$ edge-disjoint paths, we can obtain a set of equally many $s \rightsquigarrow t$ edge-disjoint *simple* paths just by deleting any cycles that may exist in the given paths.

We solve this problem as follows. Given the input, i.e., the graph $G = (V, E)$ and nodes $s, t \in V$, we construct a flow network $\mathcal{F} = (G', s, t, c)$ where

- $G' = (V, E')$ is just G with edges into s and out of t deleted, i.e., $E' := E - \big(\{(u, s) : (u, s) \in E\} \cup \{(t, u) : (t, u) \in E\}\big)$; and
- c is the capacity function where all edge capacities of G' are 1, i.e., $c(e) = 1$ for all $e \in E'$.

We then use the Ford-Fulkerson algorithm to find a maximum flow f of the flow network $\mathcal{F}$. Finally, we do a flow decomposition of f and return the $s \rightsquigarrow t$ paths found by that decomposition (and discarding the cycles found by the decomposition). Here is the algorithm in pseudocode.

```
MaxDisjointPaths(G, s, t)
1 construct the flow network F = (G', s, t, c) as described above
2 f := FordFulkerson(F)
3 (Paths, Cycles) := FlowDecomposition(f)
4 return the set of paths (first components of the pairs) in Paths
```

The running time of this algorithm is $O(mn)$: Line 1 can be done in $O(m)$ time. Line 2 takes $O(mn)$ time; this is because there are $O(n)$ edges out of s, each with capacity 1, so in this case $C \leq n$. (Note that we don't need to use any of the special ways for selecting augmenting paths discussed in Section 5.3.3, because the capacities are small.) Line 3 also takes $O(mn)$ time, as we have seen. So the running time of MaxDisjointPaths is $O(mn)$.

To prove the correctness of this algorithm we will establish a one-to-one correspondence between sets of k edge-disjoint simple $s \rightsquigarrow t$ paths in G and *integral* flows of value k in $\mathcal{F}$. (Compare this to Theorem 5.12, the key to proving the correctness of the maximum bipartite graph matching algorithm.)

Theorem 5.17 *Let $G = (V, E)$ be a directed graph, $s, t \in V$, and $\mathcal{F}$ be the flow network constructed from G, s, and t as described above.*

(a) For any integral flow f of $\mathcal{F}$, FlowDecomposition(f) returns a set of $\mathcal{V}(f)$ edge-disjoint $s \rightsquigarrow t$ simple paths in G.

(b) For any set $\mathcal{P}$ of $s \rightsquigarrow t$ edge-disjoint simple paths, the function defined by

$$f(e) = \begin{cases} 1, & \text{if } e \text{ is on one of the paths in } \mathcal{P} \\ 0, & \text{otherwise} \end{cases}$$

is a flow in $\mathcal{F}$ with value $|\mathcal{P}|$.

Proof. First we prove part (a). Let f be an integral flow in $\mathcal{F}$. Since the capacity of every edge in $\mathcal{F}$ is 1, f is a binary flow. Let (*Paths*, *Cycles*) be what FlowDecomposition returns. Recall that *Paths* is a set of pairs of the form (p, d_p), where p is a simple $s \rightsquigarrow t$ path in G' and d_p is a positive integer, and *Cycles* is a set of pairs of the form (c, d_c), where c is a simple cycle in G' and d_c is a positive integer. Let $\mathcal{P}$ be the set of paths in the pairs of *Paths* and $\mathcal{C}$ be the set of cycles in the pairs of *Cycles*. By the correctness of FlowDecomposition, $f = \sum_{p \in \mathcal{P}} d_p f_p + \sum_{c \in \mathcal{C}} d_c f_c$. Since f is a binary flow, all positive integers d_p and d_c are 1, so

$$f = \sum_{p \in \mathcal{P}} f_p + \sum_{c \in \mathcal{C}} f_c. \tag{5.11}$$

Thus no edge e can be on two paths in $\mathcal{P}$ (otherwise, $f(e) > 1$, contradicting that f is a binary flow), so $\mathcal{P}$ is a set of $s \rightsquigarrow t$ edge-disjoint simple paths in G', and therefore in G. Since f_p and

f_c are binary flows (by definition of flow decomposition), (5.11) also implies that

$$f(e) = 1 \text{ if and only if } e \text{ is on a path in } \mathcal{P} \text{ or a cycle in } \mathcal{C}. \tag{5.12}$$

Since f is a binary flow, there are exactly $\mathcal{V}(f)$ edges $e \in \text{out}(s)$ such that $f(e) = 1$. By (5.12) there are exactly $\mathcal{V}(f)$ edges out of s that appear on paths in $\mathcal{P}$ or cycles in $\mathcal{C}$. Since G' has no edges into s, these $\mathcal{V}(f)$ edges cannot be on cycles, so there are exactly $\mathcal{V}(f)$ edges out of s that are on the $s \rightsquigarrow t$ edge-disjoint simple paths in $\mathcal{P}$. Thus, $|\mathcal{P}| = \mathcal{V}(f)$, as wanted.

Next we prove part (b). Let $\mathcal{P}$ be a set of $s \rightsquigarrow t$ edge-disjoint simple paths, and $f\colon E' \to \{0, 1\}$ be the function defined by

$$f(e) = \begin{cases} 1, & \text{if } e \text{ is on one of the paths in } \mathcal{P} \\ 0, & \text{otherwise} \end{cases}$$

It is clear that f satisfies the capacity constraint. We will show that it also satisfies the conservation constraint. For any node $v \in V$ let

$$\text{in}_{\mathcal{P}}(v) = \{e \in \text{in}(v)\colon e \text{ is on a path in } \mathcal{P}\}$$
$$\text{out}_{\mathcal{P}}(v) = \{e \in \text{out}(v)\colon e \text{ is on a path in } \mathcal{P}\}.$$

By definition of f,

$$\sum_{e \in \text{in}(v)} f(e) = \sum_{e \in \text{in}_{\mathcal{P}}(v)} 1 = |\text{in}_{\mathcal{P}}(v)|, \text{and} \tag{5.13}$$

$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{out}_{\mathcal{P}}(v)} 1 = |\text{out}_{\mathcal{P}}(v)|. \tag{5.14}$$

Since the paths in $\mathcal{P}$ are edge-disjoint, each path in $\mathcal{P}$ that goes through node v must use a different edge going into v and a different edge going out of v than every other path in $\mathcal{P}$ that goes through v. So, for any $v \in V - \{s, t\}$,

$$|\text{in}_{\mathcal{P}}(v)| = |\text{out}_{\mathcal{P}}(v)| = |\{p \in \mathcal{P}\colon p \text{ goes through } v\}|.$$

Then by (5.13) and (5.14), for every $v \in V - \{s, t\}$, $\sum_{e \in \text{in}(v)} f(e) = \sum_{e \in \text{out}(v)} f(e)$, proving that the conservation constraint holds. Furthermore,

$$\begin{aligned} \mathcal{V}(f) &= \sum_{e \in \text{out}(s)} f(e) && \text{[by definition of } \mathcal{V}(f)\text{]} \\ &= |\text{out}_{\mathcal{P}}(s)| && \text{[by (5.14)]} \\ &= |\mathcal{P}| && \text{[since } \mathcal{P} \text{ is a set of edge-disjoint } s \rightsquigarrow t \text{ paths]} \end{aligned} \tag{5.15}$$

So, f is an integral flow in $\mathcal{F}$ whose value is $|\mathcal{P}|$, as wanted. □

Theorem 5.17 implies the correctness of the MaxDisjointPaths algorithm: By part (a), the algorithm outputs a set of $s \rightsquigarrow t$ edge-disjoint simple paths. By part (b), this set must have maximum cardinality among all sets of $s \rightsquigarrow t$ edge-disjoint simple paths: If a larger one existed, it would imply the existence of a flow of greater value than the flow returned in line 2, contradicting the correctness of the Ford-Fulkerson algorithm.

5.7.3 Undirected graphs

Now let's consider the same problem but when the input graph $G = (V, E)$ is *undirected*. A natural idea is to turn the undirected graph into a directed one $G_d = (V, E_d)$ where each undirected edge $\{u, v\}$ in E turns into *two* directed edges (u, v) and (v, u) in E_d. This does not quite work, because the two edges (u, v) and (v, u) could belong to different $s \rightsquigarrow t$ edge-disjoint *directed* paths in G_d which, however, share the undirected edge $\{u, v\}$ as *undirected* paths in G. This is illustrated in Figure 5.16(a): The red and green paths are edge-disjoint as directed paths in G_d, but the underlying undirected paths share the edge $\{u, v\}$.

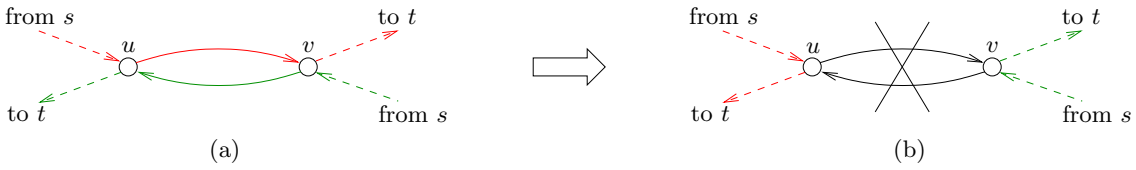


Figure 5.16: Edge-disjoint paths in G_d that are not edge-disjoint in G

Fortunately there is an easy way to fix this. This problem arises when there are two paths that contain ***anti-parallel edges***, i.e., edges that connect the same pair of nodes in opposite directions, say (u, v) and (v, u). This means that these edges carry traffic in the flow f computed in line 2. But the traffic on these anti-parallel edges is a cyclic flow that contributes nothing to the value of the flow — and therefore to the number of edge-disjoint paths in G_d that the algorithm computes in line 3. So, we can remove the traffic from both edges and get another maximum flow f' that does not use (u, v) or (v, u). Applying flow decomposition on f' replaces the red and green paths that contain anti-parallel edges as shown in Figure 5.16(a) by the red and green paths that do not contain such edges as shown in Figure 5.16(b).

The algorithm for finding a maximum cardinality set of edge-disjoint simple paths that connect nodes s and t of an undirected graph G is to run MaxDisjointPaths(G_d, s, t) with the following modification: After computing a maximum flow f, obtain another maximum flow f' by removing the traffic (one unit each) on all pairs of anti-parallel edges used by f. Then run flow decomposition on f'. Exercise 5.10 invites you to prove that this algorithm is correct and to analyze its running time.

5.7.4 Menger's theorem

In our discussion of the problem of finding a maximum cardinality set of edge-disjoint $s \rightsquigarrow t$ paths in a graph we developed results that we can leverage to prove Menger's Theorem, a well-known result in graph theory. It is one of those max-this-equals-min-that results, like max-flow-equals-min-cut and bipartite-max-matching-equals-min-vertex-cover. All these results are instances of a general phenomenon known as "linear programming duality".

Theorem 5.18 (Menger's Theorem) *Let $G = (V, E)$ be a digraph and $s, t \in V$. The*

maximum number of edge-disjoint $s \rightsquigarrow t$ paths in G is equal to the minimum number of edges whose removal from G disconnects t from s.

PROOF. Let $\mathcal{P}$ be any set of edge-disjoint $s \rightsquigarrow t$ paths, and D be any set of edges whose removal from G disconnects t from s. By the pigeonhole principle, $|\mathcal{P}| \leq |D|$: Every path in $\mathcal{P}$ (pigeon) must use an edge in D (pigeonhole); otherwise, the removal of D from G does not disconnect t from s. So, there is a function $\phi : \mathcal{P} \to D$ such that path $p \in \mathcal{P}$ uses edge $\phi(p) \in D$. Since $\mathcal{P}$ is edge-disjoint, ϕ must be one-to-one. So, by the pigeonhole principle $|\mathcal{P}| \leq |D|$. This immediately implies the following:

If $|\mathcal{P}| = |D|$ then
(a) *$\mathcal{P}$ is a* maximum *cardinality set of edge-disjoint $s \rightsquigarrow t$ paths, and* (5.16)
(b) *D is a* minimum *cardinality set of edges whose removal disconnects t from s.*

Let $\mathcal{P}$ be a maximum cardinality set of edge-disjoint $s \rightsquigarrow t$ paths. Let $\mathcal{F}$ be the flow network obtained from G by removing all edges into s and from t, and assigning capacity 1 to every edge. Let f be a maximum flow of $\mathcal{F}$, (S, T) be a minimum (s, t)-cut of $\mathcal{F}$, and $D = \text{out}(S) \cap \text{in}(T)$; i.e., D is the set of edges that cross the cut from S to T. By definition, the removal of D from the graph of $\mathcal{F}$ disconnects t from s, and so the removal of D from G also disconnects t from s (why?). We have:

$$\begin{aligned} |\mathcal{P}| &= \mathcal{V}(f) && \text{[by (5.15)]} \\ &= c(S, T) && \text{[by max-flow-min-cut]} \\ &= |D| && \text{[by definition of } D \text{ and the fact that all edges have capacity 1]} \end{aligned}$$

By part (b) of (5.16), D is a minimum cardinality set of edges whose removal from G disconnects t from s.

So, we proved that the maximum number of edge-disjoint $s \rightsquigarrow t$ paths in G is equal to the minimum number of edges whose deletion from G disconnects t from s. □

This proof of Menger's theorem immediately suggests an algorithm that, given a directed graph G and nodes s, t, finds a minimum cardinality set of edges whose deletion from G disconnects t from s:

(1) Construct $\mathcal{F}$ from G, s, and t
(2) Find a maximum flow f of $\mathcal{F}$
(3) Using f, find a minimum cut (S, T) of $\mathcal{F}$
(4) Return the set of edges $\text{out}(S) \cap \text{in}(T)$

This takes $O(mn)$ time, where m is the number of edges of G and n is the number of nodes of G.

Exercises for Chapter 5

5.1 (a) Give an example of a flow network $\mathcal{F}$ with integer capacities,and a *maximum* flow f in $\mathcal{F}$ such that for some edge e, $f(e)$ is *not* an integer. Prove that your flow f is a maximum flow in G.

(b) Let $\mathcal{F}$ be any flow network with integer capacities, and f be any *maximum* flow in $\mathcal{F}$. Assume that the value of f is positive (i.e., the source and sink are connected through at least one path). Prove that there is an edge e such that $f(e)$ is a positive integer. (In other words, it is not possible to construct an example as in part (a) where *every* edge used by the flow carries a fractional amount of traffic.)

(c) Let $\mathcal{F}$ be any flow network with integer capacities, and m be the value of the maximum flow in G. Prove that, for each integer k such that $0 \leq k \leq m$, there is a flow f in $\mathcal{F}$ that has value k.

5.2 Let $\mathcal{F} = (G, s, t, c)$ be a flow network with integer capacities, f be an *integral* maximum flow in $\mathcal{F}$, and e be an edge of the graph G. Let $\mathcal{F}^+ = (G, s, t, c^+)$ be the flow graph obtained from $\mathcal{F}$ by increasing the capacity of e by one unit, and $\mathcal{F}^- = (G, s, t, c^-)$ be the flow graph obtained from $\mathcal{F}$ by decreasing the capacity of e by one unit. (That is, for every edge $e' \neq e$ of G, $c^+(e') = c^-(e') = c(e')$, $c^+(e) = c(e) + 1$, and $c^-(e) = c(e) - 1$.)

(a) Prove that the value of a maximum flow in $\mathcal{F}^+$ is $\mathcal{V}(f)$ or $\mathcal{V}(f) + 1$, and the value of a maximum flow in $\mathcal{F}^-$ is $\mathcal{V}(f)$ or $\mathcal{V}(f) - 1$.

(b) Give an algorithm that, given $\mathcal{F}$, f, and e, finds a maximum flow in $\mathcal{F}^+$. Your algorithm should be *significantly faster* than recomputing a maximum flow of $\mathcal{F}^+$ from scratch. Justify the correctness of your algorithm and analyze its running time.

(c) Do as in part (b), but for $\mathcal{F}^-$.

5.3 Given a flow network in which all edges have capacity 1 and a positive integer k, we want to remove k edges from the flow network so as to reduce as much as possible the maximum flow in the resulting flow network.

Give a polynomial-time algorithm that does this. That is, your algorithm takes as input (a) a flow graph (G, s, t, c), where $G = (V, E)$ and $c(e) = 1$ for each $e \in E$, and (b) a positive integer $k \leq |E|$, and produces as output a set $E' \subseteq E$ such that (i) $|E'| = k$, and (ii) the maximum flow in the flow graph (G', s, t, c') is as small as possible, where $G' = (V, E - E')$ and $c'(e) = 1$ for each $e \in E - E'$.

Prove that your algorithm is correct and analyze its running time.

5.4 In this exercise you will investigate how the Ford-Fulkerson algorithm may fail to terminate if the capacities are irrational numbers. An elegant flow network used to demonstrate this, shown in Figure 5.17, is due to Uri Zwick. The quantity ϕ is the inverse of the famous ***golden ratio*** $\Phi = (\sqrt{5}+1)/2$, so $\phi = 1/\Phi = (\sqrt{5}-1)/2 \approx 0.618$. Note that $\phi^2 = 1 - \phi$. The capacity k is an arbitrary positive integer that is at least 3, but you should be thinking of it as a large integer.

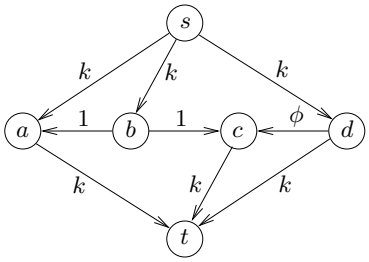


Figure 5.17: Zwick's flow network illustrating non-termination of Ford-Fulkerson's algorithm

(a) What is the maximum flow in this nework (as a function of k)? Prove that your answer is correct!

(b) Suppose that we run the Ford-Fulkerson algorithm where the first augmenting path is s, b, c, t. Prove that thereafter we can then repeatedly select the following four augmenting paths in the indicated order:
(1) s, d, c, b, a, t,
(2) s, b, c, d, t,
(3) s, d, c, b, a, t (same as (1)), and
(4) s, a, b, c, t.
Show that in the i-th repetition of this group of augmenting steps, the flow is increased by ϕ^{2i-1} in paths (1) and (2) and by ϕ^{2i} in paths (3) and (4). Give a closed-form formula for the value $v(i)$ of the flow computed at the end of the i-th repetition of these augmenting steps.

(c) Show that the sequence $v(1), v(2), v(3), \ldots$ converges. What is its limit? How does this limit compare to the value of the maximum flow you found in part (a)?

5.5 Let FF-WidestPath be the version of the Ford-Fulkerson algorithm that always chooses an augmenting path with maximum "width" (minimum residual capacity over the edges of the path). We have noted that we can find a widest path using a variant of Dijkstra's algorithm, but this is slower than using a depth- or breadth-first search, and for this reason we used the FF-WidePath algorithm that finds a "wide enough" path. In this exercise you will show that FF-WidestPath terminates after $O(m \log C)$ augmenting steps. Let $\mathcal{F} = (G, s, t, c)$ be a flow network and v^* be the value of the maximum flow in $\mathcal{F}$.

(a) Prove that if f is a flow in $\mathcal{F}$ and w is the maximum width of any augmenting path in $\mathcal{F}_f$, then $v^* - \mathcal{V}(f) < mw$. **Hint:** This is essentially Claim 5.8.1 and proved similarly.

(b) Prove that if f is a flow in $\mathcal{F}$ and f' is the flow resulting by augmenting f using a widest augmenting path in $\mathcal{F}_f$, then $v^* - \mathcal{V}(f') < \big(v^* - \mathcal{V}(f)\big)(1 - 1/m)$. **Hint:** Use part (a).

(c) Prove that if f_i is a flow in $\mathcal{F}$ after i augmenting steps of FF-WidestPath (where f_0 is the trivial flow, in which every edge has zero traffic), then $v^* - \mathcal{V}(f_i) < v^*(1 - 1/m)^i$. **Hint:** Use part (b).

(d) Prove that FF-WidePath halts after at most $m \ln C$ augmenting steps. **Hint:** Use

part (c) and the inequality $(1 - x) \leq e^{-x}$, which holds for all $x \in \mathbb{R}$ (and is reasonably tight for x close to 0).

5.6 In this exercise we will consider some variants of the FF-WidePath algorithm to experiment with different ways of balancing doing things sloppily but fast, versus doing things carefully but more slowly.

(a) Suppose that instead of halving the minimum width w of augmenting paths during a phase, we decrease it by a fixed amount, say 5. More precisely, in line 10 of the algorithm where we shrink the lower bound w on the width to start the next phase, r instead of halving w, we we decrease it by 5 ($w := w - 5$). First conjecture what you think the effect of this will be on the running time of the algorithm, and then analyze its running time. How does it compare (asymptotically) with the running time of the version discussed in the notes? Is this a polynomial-time algorithm?

(b) Suppose that instead of halving the minimum width w of augmenting paths during a phase, we divide it by 3. More precisely, in line 10 of the algorithm where we shrink the lower bound w on the width to start the next phase, instead of halving w, we divide it by 3 ($w := \lceil w/3 \rceil$). First conjecture what you think the effect of this will be on the running time of the algorithm, and then analyze its running time. How does it compare (asymptotically) with the running time of the version discussed in the notes? Is this a polynomial-time algorithm?

(c) Suppose that, to reduce the number of phases faster, we reduce w *doubly*-exponentially. We can do this as follows. We define a "decay factor" r, initially set to 2. After each iteration of the outer loop (lines 4-8), we divide w by r, and set $r := r^2$. Now the number of phases is smaller: $\Theta(\log \log C)$, instead of $\Theta(\log C)$. What is the running time of the resulting algorithm? Is this a polynomial-time algorithm?

5.7 Let $\mathcal{F} = (G, s, t, c)$ be a flow graph with integer capacities. A flow f in $\mathcal{F}$ is ***cyclic*** if there is a cycle in G such that every edge on the cycle has positive traffic in f.

(a) Suppose we use the shortest paths version of the Ford-Fulkerson algorithm. Give an example flow network in which this algorithm *necessarily* produces a cyclic flow. Your example should be as simple as you can. You should identify the augmenting paths that the algorithm chooses in your example flow network and the cycle of the maximum flow that it creates.

(b) Prove that for any flow network $\mathcal{F}$ and any flow f in $\mathcal{F}$, there is an acyclic flow of $\mathcal{F}$ with the same value as f. Do *not* assume that f is an integral flow.

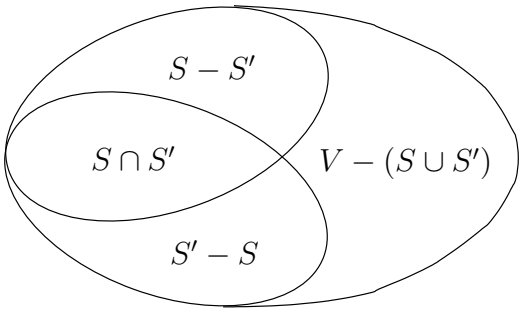
5.8 Prove Theorem 5.10.

5.9 Describe the data structures that can be used to achieve $O(mn)$ running time for the FlowDecomposition algorithm. Assume that the flow f is given as the adjacency list for the flow network graph G, where the node that corresponds to edge e contains a field that stores the traffic $f(e)$.

5.10 Prove the correctness and analyze the running time of the algorithm described in Section 5.7.3 for finding a maximum cardinality set of edge-disjoint simple paths that connect nodes s and t of an undirected graph G.

5.11 Let $\mathcal{F} = (G, s, t, c)$ be a flow network and (S, T), (S', T') be ***minimum*** cuts of $\mathcal{F}$. Prove that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are also minimum cuts of $\mathcal{F}$.

Hint: Prove both facts together. In thinking about this problem you may find the following diagram useful:



5.12 Given a flow network in which all edges have capacity *one* and a positive integer k, we want to remove k edges from the flow network so as to reduce as much as possible the maximum flow in the resulting flow network.

Give a polynomial-time algorithm that does this. That is, your algorithm takes as input (a) a flow network $\mathcal{F} = (G, s, t, c)$, where $G = (V, E)$ and $c(e) = 1$ for each $e \in E$, and (b) a positive integer $k \leq |E|$, and produces as output a set $E' \subseteq E$ such that (i) $|E'| = k$, and (ii) the maximum flow in the flow graph (G', s, t, c') is as small as possible, where $G' = (V, E - E')$ and $c'(e) = 1$ for each $e \in E - E'$.

Prove that your algorithm is correct and analyze its running time.

5.13 Let $\mathcal{F} = (G, s, t, c)$ be a flow network with integer capacities, p be an $s \rightsquigarrow t$ simple path of G, and ℓ be the length of p. (Recall that the length of a path is the number of *edges* on it.) Let $\mathcal{F}^+ = (G, s, t, c^+)$ be the flow graph obtained from $\mathcal{F}$ by increasing the capacity of every edge on p by one unit. That is, for every edge e of G,

$$c^+(e) = \begin{cases} c(e) + 1, & \text{if } e \text{ is on } p \\ c(e), & \text{otherwise.} \end{cases}$$

Let V be the value of a maximum flow of $\mathcal{F}$ and V^+ be the value of a maximum flow of $\mathcal{F}^+$. In this question we explore the effect of increasing the capacities of the edges on p: What are the possible values of $V^+ - V$?

(a) Prove that $V^+ \geq V + 1$. That is, the increase of the capacities of the edges on p *necessarily* increases the value of the maximum flow by at least 1.

(b) Prove that $\lceil \ell/2 \rceil$ is a *tight* bound on $V^+ - V$. That is, prove that (i) for *any* flow network $\mathcal{F}$, $V^+ \leq V + \lceil \ell/2 \rceil$, and (ii) there is *some* flow network $\mathcal{F}$ such that $V^+ = V + \lceil \ell/2 \rceil$.

(c) For each $d \in [1..\lceil \ell/2 \rceil]$, can you design a flow network whose maximum flow value increases by exactly d if you increase by 1 the capacity of every edge on some $s \rightsquigarrow t$ path?

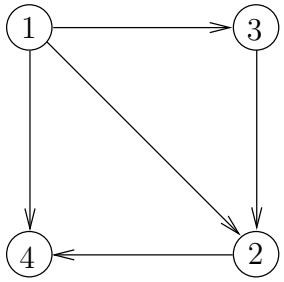
5.14 A ***bottleneck*** of a flow network $\mathcal{F}$ is an edge of $\mathcal{F}$ such that increasing the capacity of that edge (and of no other edge) increases the value of the maximum flow. Describe an algorithm that, given a flow network $\mathcal{F}$ *and a maximum flow* f *of* $\mathcal{F}$, returns the set of *all* bottleneck edges of $\mathcal{F}$. Your algorithm should run in $O(n+m)$ time, where n is the number of nodes and m is the number of edges of $\mathcal{F}$. Prove that your algorithm is correct and analyze its running time.

5.15 Let $G = (V, E)$ be a directed graph. For each node u, the ***in-degree*** (respectively, ***out-degree***) of u in G is the number of edges in E that go into (respectively, out of) u. For the remainder of this question we consider only directed graphs with no self-loops, i.e., no edge from a node to itself.

Suppose that we are given a set of nodes V (but no edges), and for each $u \in V$, a pair of non-negative integers, $in(u)$ and $out(u)$; we call this the ***degree-pair*** of u. Is there a directed graph $G = (V, E)$ such that for each $u \in V$, the in-degree of u in G is $in(u)$ and the out-degree of u in G is $out(u)$? If so, we say that the given degree-pairs are ***realizable***, and that the graph G ***realizes*** them.

For example, the degree pairs given below for $V = \{1, 2, 3, 4\}$ is realizable, as confirmed by the graph to the right.

u	$in(u)$	$out(u)$
1	0	3
2	2	1
3	1	1
4	2	0



On the other hand, the degree pairs given below for $V = \{1, 2, 3\}$ is not realizable: each of nodes 2 and 3 has out-degree 2, so each of them must have an edge to node 1, whose in-degree, however, is only 1.

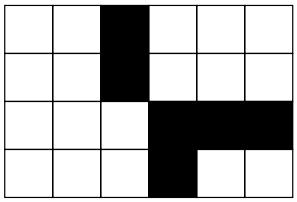
u	$in(u)$	$out(u)$
1	1	1
2	2	2
3	2	2

Give a polynomial-time algorithm that, given a set of nodes V and a pair of non-negative integers $\big(in(u), out(u)\big)$ for each $u \in V$, determines whether these degree-pairs are realizable. If they are, the algorithm should also output a directed graph that realizes them. Prove that your algorithm is correct and analyze its running time.

Hint. Reduce this problem to a max-flow problem: Define a flow network $\mathcal{F} = (G, s, t, c)$ that, in addition to source and target nodes, has two nodes, v_{out} and v_{in}, for each node $v \in V$. It is up to you to determine the edges of G and their capacities. G should be defined in such a manner that the given degree pairs are realizable if and only if the maximum flow of $\mathcal{F}$ satisfies a certain (easy-to-check) property.

5.16 Consider an $m \times n$ grid (m rows and n columns). Some squares of the grid contain ***tokens*** and others are empty. Each token is identified by the coordinates (i, j) of the square that contains it, where $1 \leq i \leq m$ and $1 \leq j \leq n$. A subset of the set of tokens is called ***irreducible*** if it contains no two tokens on the same row or column. A set consisting of rows or columns is called ***full*** (with respect to the set of tokens on the grid) if each token is on some row or some column in the set.

For example, in the 4×6 grid below, squares containing tokens are shown in black. The following sets of tokens are irreducible: $\varnothing$, $\{(2,3),(4,4)\}$, and $\{(1,3),(3,6),(4,4)\}$. The last of these is an irreducible set of maximum cardinality: no irreducible set contains four tokens. The following sets of rows or columns are full, where Ri denotes row i and Cj denotes column j: $\{\text{R1}, \text{C2}, \text{C3}, \text{R3}, \text{C4}\}$, $\{\text{R1}, \text{C3}, \text{R3}, \text{C4}\}$, $\{\text{C3}, \text{R3}, \text{C4}\}$. The last of these is a full set of minimum cardinality: no cover contains only two rows or columns.



Prove that the maximum cardinality of an irreducible set of tokens is equal to the minimum cardinality of a full set of rows or columns.

Design polynomial-time algorithms that, given the dimensions of the grid and the positions of the tokens, find (i) a cover of minimum cardinality, and (ii) an irreducible set of maximum cardinality.

Chapter 6

Linear programming

This chapter is a brief and, frankly, superficial treatment of a very important topic, known as ***linear programming***, which is properly the subject of an entire course. If you have the opportunity, and you are interested in optimization, you should take such a course. The treatment of the topic here will be very different from the other chapters. My goal is not to describe algorithms (except in very-high level terms), prove their correctness, or analyze their running times. Rather, I want to give a sense of the type of problems that can be expressed as linear programs, provide some geometric intuition, and give examples of how we can efficiently reduce other problems to linear programming.

6.1 What is linear programming?

We begin with two typical examples of problems that can be formulated as linear programs. We then describe the general form of linear programs. By the way, the word "programming" in this context has the meaning of "planning" or "scheduling", as it does in the term "dynamic programming". Like that term, it dates to the late 1940s and early 1950s.

6.1.1 Cost minimization — the "diet problem"

A food aid agency can buy n different foods $F_1, F_2, \ldots, F_n$ supplying m nutrients $N_1, N_2, \ldots, N_m$ that are vital in human diet. We know that each unit of food F_j, $j \in [1..n]$, has cost c_j and provides a_{ij} units of nutrient N_i, $i \in [1..m]$. We also know that a person needs at least b_i units of nutrient N_i per day. The question is: What is the cheapest diet (i.e., combination of foods) that satisfies the daily nutrient requirements?

We can express this as the following ***linear program***:

- **Variables:** $x_1, x_2, \ldots, x_n$.
  Quantity x_j represents the amount of food F_j in the chosen diet; the x_js range over the reals (or rationals).
- **Objective function:** Minimize $\underbrace{c_1x_1 + c_2x_2 + \ldots + c_nx_n}_{\text{the cost of the chosen diet}}$.

- **Constraints:**
  - For each nutrient N_i, $i \in [1..m]$, the constraint $\underbrace{a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n}_{\text{amount of nutrient } N_i \text{ in chosen diet}} \geq b_i$.
  - For each variable x_j, $j \in [1..n]$, the constraint $x_j \geq 0$; we cannot buy a negative amount of food F_j.

We emphasize that the quantities we seek for the variables are reals (or rationals), and *not integers*. The additional requirement that the variables take on integer values makes the problem very different; more on this in Section 6.5

6.1.2 Profit maximization — the "product selection problem"

A company can make n different products $P_1, P_2, \ldots, P_n$ using m resources $R_1, R_2, \ldots, R_m$ (labour, different types of materials, warehouse space, etc). We know that each unit of product P_j, $j \in [1..n]$, can be sold at a profit of c_j and making it requires a_{ij} units of resource R_i, $i \in [1..m]$. We also know that the company has only b_i units of resource R_i. The question is: What is the most profitable product lineup (i.e., combination of products) for the company to make without exceeding the available resources?

We can express this as the following linear program:

- **Variables:** $x_1, x_2, \ldots, x_n$.
  Quantity x_j represents the amount of product P_j in the chosen lineup; the x_js range over the reals (or rationals).
- **Objective function:** Maximize $\underbrace{c_1x_1 + c_2x_2 + \ldots + c_nx_n}_{\text{profit that the chosen lineup yields}}$.
- **Constraints:**
  - For each resource R_i, $i \in [1..m]$, $\underbrace{a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n}_{\text{amount of resource } R_i \text{ in chosen lineup}} \leq b_i$.
  - For each variable x_j, $j \in [1..n]$, the constraint $x_j \geq 0$; we cannot make a negative amount of product P_j.

6.1.3 General form of a linear program

In general, a linear program (abbreviated ***LP***) consists of three ingredients:

- A set of variables $x_1, x_2, \ldots, x_n$, that take on values over the reals (or rationals, but from now on we will stop repeating this possibility).
- A *linear* objective function $\sum_{j=1}^{n} c_j x_j$, where $c_j \in \mathbb{R}$ for all $j \in [1..n]$; this function is to be minimized or maximized.
- A list of m *linear equality or inequality* constraints, i.e., constraints of the form

$$\sum_{j=1}^{n} a_{ij}x_j \left\{\begin{matrix}=\\ \leq \\ \geq\end{matrix}\right\} b_i,$$

where $a_{ij}, b_i \in \mathbb{R}$ for all $i \in [1..m]$ and $j \in [1..n]$.

Definition 6.1 *A linear program is in* ***standard max form*** *if*

- *the objective function is to be maximized,*
- *the constraints require all variables to be non-negative, and*
- *all remaining constraints are $\leq$-inequalities.*

That is, a linear program in standard max form looks like this:

- **Variables:** $x_1, x_2, \ldots, x_n$
- **Objective function:** maximize $c_1x_1 + c_2x_2 + \cdots + c_nx_n$
- **Constraints:**

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ &\vdots \\ x_n &\geq 0 \end{aligned}$$

An LP in max standard form can be expressed in a very compact and convenient way using linear-algebraic notation. Define the column vectors

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

and the $m \times n$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

Then we can express the objective function simply as the dot product $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$, and the constraints as $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$, where inequality between vectors indicates component-wise inequalities in all dimensions. That is, the LP is: maximize $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ subject to $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$.

There is a corresponding ***canonical min form*** for linear programs: The objective function is now to be *minimized*, the constraints require the variables to be non-negative, and all other constraints are $\geq$-inequalities. Using our handy linear algebra notation, a canonical min form LP is: minimize $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ subject to $\mathbf{Ax} \geq \mathbf{b}$ and $\mathbf{x} \geq 0$.

You can check that the diet problem we discussed earlier is in standard min form, and the product selection problem is in standard max form.

It is not hard to see that we can put *any* linear program to standard max or standard min form:

- We can turn a minimization problem to a maximization problem (and vice versa) by negating the coefficients of the objective function: instead of minimizing $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ we maximize $-\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$.
- We can replace any equality constraint $\sum_{j=1}^{n} a_{ij}x_j = b_i$ by the two inequality constraints $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ and $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$.
- If an inequality constraint is pointing in the wrong direction ($\geq$ instead of $\leq$ or vice-versa) we can multiply both sides by -1 and reverse the inequality sign.
- For every variable x_i we can introduce two variables x_i^+ and x_i^-, add the constraints $x_i^+ \geq 0$ and $x_i^- \geq 0$, and replace every occurrence of x_i by the difference $x_i^+ - x_i^-$. This works because every real number can be written as the difference of two non-negative real numbers.

Note that putting an arbitrary LP to standard max (or min) form using these transformations does not change the size of the program significantly: At most it doubles the number of variables and the number of constraints. So, the ***linear programming problem*** can be stated as follows:

Input: Column vectors $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $m \times n$ matrix $\mathbf{A}$ of reals.
Output: Column vector $\mathbf{x} \in \mathbb{R}^n$ that maximizes $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ subject to the requirements that $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$.

The input *size* is expressed as a function of the number of variables n, the number of constraints m, *and the total number B of bits required to represent the input* $\boldsymbol{c}$, $\boldsymbol{b}$, *and* $\boldsymbol{A}$.[1]

6.2 The geometry of linear programs

Our goal in this section is to gain some geometric insight into linear programs. We will develop our geometric interpretation of a linear program using a simple instance of the product selection problem.

Imagine a farmer who can grow two types of corn, a "high grade" type C1 (say for human consumption) and a "low grade" type C2 (say for industrial use to make corn syrup). C1 yields a profit of 3 while C2 yields a profit of 2 (in thousands of dollars per hectare). The farmer has unlimited land, but is constrained by three resources: the amount of labour needed (40 hours are available), the amount of seed (100 bags are available), and the amount of fertilizer (10 bags are available). The resource needs of the two types of corn are as follows, all for each hectare of land planted:

- C1 requires 2 hours of labour, 1 bag of seed, and 1 bag of fertilizer.
- C2 requires 1 hour of labour, 3 bags of seed, and no fertilizer.

The farmer wants to decide how many hectares of C1 and how many hectares of C2 to plant so as to maximize her profit. We can set this up as the following linear program:

[1] You may be wondering how to represent an arbitrary real number with a finite number of bits. When analyzing the running time of algorithms for the linear programming problem, it is more precise to think of the input as *rationals*, rather than reals.

- **Variables:** x_1, x_2.
  x_1 and x_2 represent the number of hectares of corn type C1 and C2, respectively, that the farmer plants.
- **Objective function:** maximize $3x_1 + 2x_2$.
- **Constraints:**

$$\begin{aligned} 2x_1 + \ x_2 &\leq 40 \\ x_1 + 3x_2 &\leq 100 \\ x_1 \qquad &\leq 10 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

We have limited ourselves to just two variables so as to make drawing on a flat surface possible. Note that in this case the ***decision space*** i.e., the set of vectors that assign values to the variables, is simply $\mathbb{R}^2$. We will visualize this as the Cartesian plane, with the horizontal axis for x_1 and the vertical axis for x_2. The solutions to the linear program are therefore points on this plane defined by a pair $(x_1, x_2) \in \mathbb{R}^2$.

The geometry of constraints

Some points on the plane satisfy the constraints and some do not. The set of points that satisfy all constraints is called the ***feasible set*** or, since we are visualizing the decision space as a plane, the ***feasible region***. Let's see what the feasible region of a two-dimensional (two-variable) linear program looks like.

Each constraint defines a half-plane. For example, the constraint $2x_1 + x_2 \leq 40$ is satisfied by the points on or below the line $2x_1 + x_2 = 40$, which has slope -2 and x_2-intercept 40, as shown in Figure 6.1(a). Similarly, the constraint $x_1 + 3x_2 \leq 100$ is satisfied by the points on or below the line $x_1 + 3x_2 = 100$, as shown in Figure 6.1(b); and the constraint $x_1 \leq 10$ is satisfied by the points on or to the left the line $x_1 = 10$, as shown in Figure 6.1(c). There are also the two non-negativity constraints $x_1 \geq 0$ and $x_2 \geq 0$; these are satisfied by the points to the right of the vertical axis $x_1 = 0$ and above the horizontal axis $x_2 = 0$, respectively — these are not shown in Figure 6.1.

We are interested in the points that satisfy *all* constraints; that is, the points in the *intersection* of the five half-planes defined by the five constraints. This is the feasible region of our linear program and is the region highlighted in Figure 6.1(d).

Note that the feasible region is a ***convex polygon***. This means that the straight line segment that connects *any* two points inside the polygon is entirely inside the polygon. See Figure 6.2 for examples of convex and non-convex polygons. The convexity of the feasible region is a direct consequence of the fact that the feasible region is an intersection of half-planes. To see why the non-convex polygon in the middle of Figure 6.2 cannot possibly be the feasible region of a linear program consider the dashed line that corresponds to the linear constraint that defines one of the polygon's sides as shown in the right of that figure. If this constraint is a $\leq$-inequality then the half-plane it defines is the plane *below* that line and so the feasible region has no business containing points above it; and similarly if the constraint

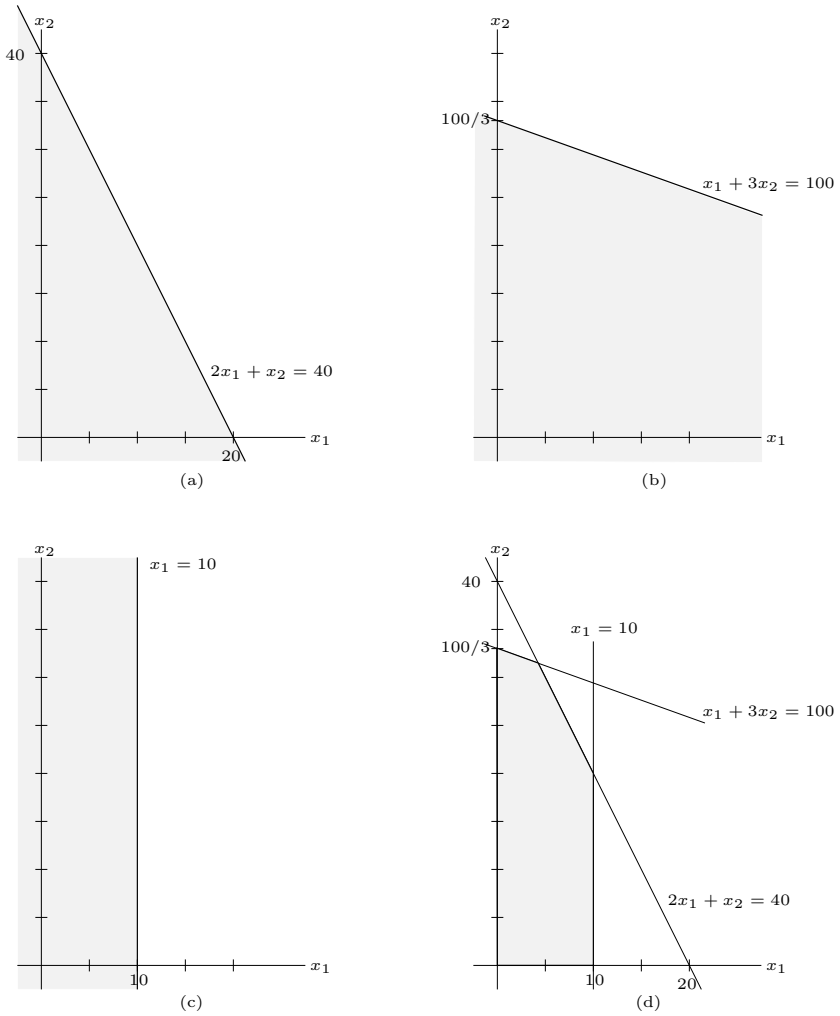


Figure 6.1: The geometry of a linear program's constraints

is a $\geq$-inequality. The fact that the feasible region is convex is an important property that is exploited by algorithms that solve linear programs.

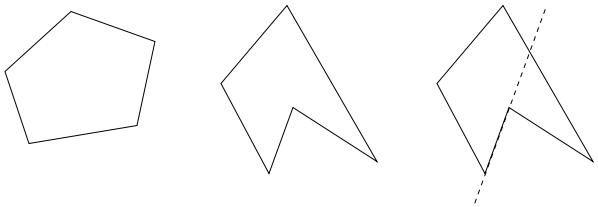


Figure 6.2: A convex (left) and non-convex (middle) polygon

The geometry of the objective function

The objective function in our example linear program is to maximize $3x_1 + 2x_2$. Consider the line $3x_1 + 2x_2 = p$, for some value of p. This line has slope $-3/2$ and x_2-intercept $p/2$; it represents the set of points that result in a profit of p for the farmer; see Figure 6.3(a), which shows in green the line $3x_1 + 2x_2 = 40$, corresponding to points (x_1, x_2) that yield a profit of 20. By varying the value of p we get different lines, each corresponding to the set of points that yield a specific level of profit. All these lines differ only in the value of p; they have the same slope and are therefore parallel to each other. This is illustrated in Figure 6.3(b), which shows lines that correspond to different profit levels, shown in green and one shown in red.

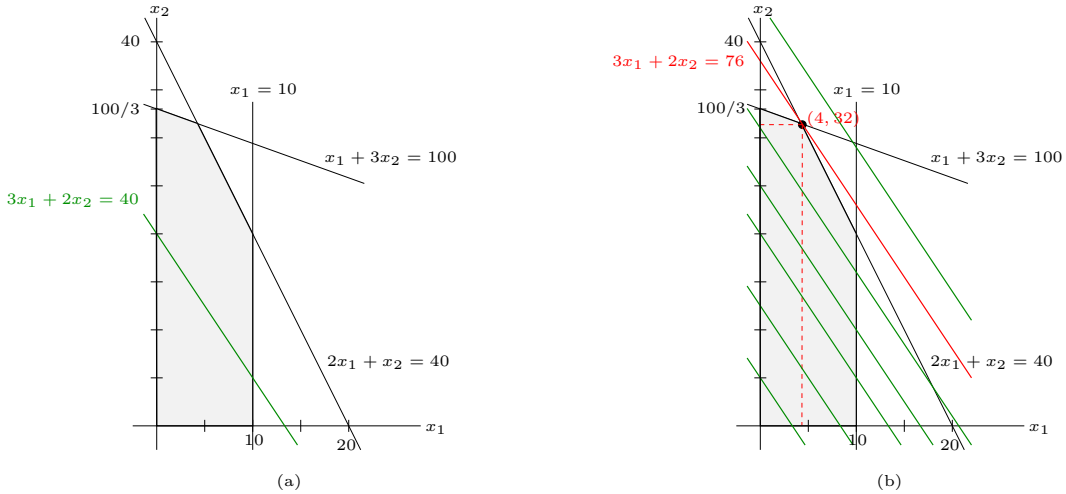


Figure 6.3: Lines representing different profit levels for the objective function

Imagine now the line $3x_1 + 2x_2 = p$ for increasing values of p, ranging from $-\infty$ to $+\infty$. While $p < 0$ this line has empty intersection with the feasible region since, for these values of p, one of the non-negativity constraints $x_1 \geq 0$ and $x_2 \geq 0$ must be violated. Then, for a while,

the line intersects the feasible region; these values of p correspond to profit levels that are feasible within the resource constraints. Finally, for large enough values of p, the line no longer intersects the feasible region: these are profit levels that cannot be attained with the available resources. We are interested in the precise value of p just before the line "loses contact" with the feasible region as we increase p. In our case, this is the line $3x_1 + 2x_2 = 76$, shown in red in Figure 6.3(b). It "touches" the feasible region at $(4, 32)$, which is the optimal solution yielding a profit of $3 \cdot 4 + 2 \cdot 32 = 76$.

In this example there is a unique optimal solution, namely $x_1 = 4$ and $x_2 = 32$. This is not necessarily the case. For example, if the objective function had been to maximize $2x_1 + x_2$, then there would be an infinite number of solutions: the slope of this objective function is the same as that of the line that corresponds to the labour constraint $2x_1 + x_2 \leq 40$, and then every point on the line $2x_1 + x_2 = 40$ between (and including) the points $(4, 32)$ and $(10, 20)$ would be an optimal solution yielding a maximum profit of 40 (for *that* objective function).

We have seen that a linear program could have a unique optimal solution or an infinite number of them. It is also possible that a linear program has no solution at all; and indeed there are two different reasons why that might be the case:

- The linear program might be "overconstrained" so that the feasible region is empty. For instance if we have two constraints requiring $x_i \leq -2$ and $x_i \geq 3$, it is impossible to satisfy both of them. In this case we say that the linear program is ***infeasible***.
- The linear program might be "undercontrained", i.e., the constraints might be too loose for the objective function to obtain a maximum or minimum. For example, if the linear program is: maximize x_1 subject to $x_1 \geq 0$, there is no optimal solution. Or in our example, if our linear program was missing the labour and seed constraints, the farmer could make an arbitrary profit by planting an arbitrary amount of corn type C1, since we have not constrained the amount of land available. In this case we say that the linear program is ***unbounded***.

The vertices of the feasible region are called ***basic feasible solutions***. In our example, where the linear program has a unique optimal solution, that solution is in fact a *basic* feasible solution. In the modified linear program that had infinitely many solutions (when we changed the objective function from maximizing $3x_1 + 2x_2$ to maximizing $2x_1 + x_2$), included among them were two basic feasible solutions. This is not an accident:

Theorem 6.2 *If a linear program has an optimal solution then at least one optimal solution is a basic feasible solution.*

This fact is very useful, as it narrows the search for an optimal solution to a well-defined finite subset: the "corners" of the feasible region; it is exploited by some algorithms for solving linear programs.

A linear algebraic view of a linear program

Recall that the value of the objective function is the dot product $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$, where $\mathbf{x}$ is a solution vector and $\mathbf{c}^\mathsf{T}$ is the constant vector of the objective function's coefficients. We are seeking a

vector $\mathbf{x}$ that lies in the feasible region and *maximizes* (or minimizes, depending on the kind of problem we are solving) this dot product. But recall from your linear algebra course the geometric meaning of the dot product: it is the length of the projection of $\mathbf{x}$ on $\mathbf{c}^\mathsf{T}$, scaled by the length of $\mathbf{c}$:

$$\mathbf{c}^\mathsf{T} \cdot \mathbf{x} = |\mathbf{c}^\mathsf{T}| \cdot \text{the length of projection of } \mathbf{x} \text{ in the direction of vector } \mathbf{c}\mathsf{T}$$

Since $\mathbf{c}^\mathsf{T}$ (and therefore $|\mathbf{c}^\mathsf{T}|$) is fixed, we are seeking a vector $\mathbf{x}$ in the feasible region that maximizes (or minimizes) the projection of $\mathbf{x}$ in the direction of $\mathbf{c}^\mathsf{T}$. This is illustrated in Figure 6.4 for our example. The vector $\mathbf{c}^\mathsf{T} = (3, 2)$ is shown in blue, and three feasible solutions (points in the feasible region) are shown as heavy black dots, along with their projections on the (direction of the) vector $\mathbf{c}^\mathsf{T}$. As you can see the point, not only among these three but among all points in the feasible region, that "casts the longest shadow" in the direction of the vector $(3, 2)$ is the optimal solution $(4, 32)$.

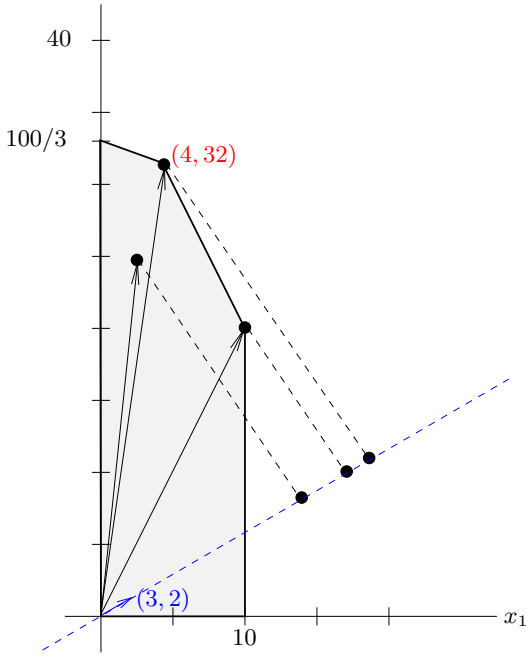


Figure 6.4: The linear algebra view of our linear program example

Higher dimensional linear programs

What about more variables? Building on our example, suppose the farmer has the option for a third type of corn, C3, say intended for animal feed. So, we introduce a third variable, x_3, representing the amount of hectares to plant corn of type C3. If each hectare planted with C3 yields a profit of 2.5 (thousands of dollars), the objective function is now to maximize

$3x_1 + 2x_2 + 2.5x_3$. If each hectare planted with C3 requires 1.5 hours of labour, 2 bags of seed, and 0.5 bags of fertilizer, the set of constraints becomes:

$$\begin{aligned} 2x_1 + x_2 + 1.5x_3 &\leq 40 \\ x_1 + 3x_2 + 2x_3 &\leq 100 \\ x_1 + 0.5x_3 &\leq 10 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ x_3 &\geq 0 \end{aligned}$$

So, now the decision space is 3-dimensional instead of 2-dimensional. Each constraint defines a *half-space*, instead of a half-plane: the equation $a_1x_1 + a_2x_2 + a_3x_3 = b$ defines a plane in $\mathbb{R}^3$, and the equality $a_1x_1 + a_2x_2 + a_3x_3 \leq b$ defines the half space "below" and to the "left" of that plane. The feasible region is the intersection of these half-spaces, the shape of which is a *convex polyhedron*. The objective function $c_1x_1 + c_2x_2 + c_3x_3$ defines a set of parallel planes, and if we are maximizing this function, the maximum value corresponds to the "highest" of these planes that has a non-empty intersection with the convex polyhedron formed by the constraints. This is not too hard to visualize, but it is much harder for me to render on a flat page, so no illustration for this — use your imagination!

And what if we had four or more variables? At this point it is no longer possible (at least for me) to "visualize" the geometry of linear programs. Fortunately, all the relevant geometric concepts generalize algebraically to an arbitrary number of variables, i.e., dimensions. And that's a good thing because the kinds of linear programs that arise in modern optimization problems can involve hundreds of thousands of variables and constraints! Space now become "hyperspace", planes become "hyperplanes", and a closed part of hyperspace is called a "polytope" (generalizing the two-dimensional polygon and the three-dimensional polyhedron). So the intersection of half-hyperspaces is a convex polytope, which is the "shape" of the feasible region in an n-dimensional decision space.

Terminology

Following are terms associated with linear programs; these are self-explanatory and not really to be memorized, but you should be familiar with them because they are fairly standard in the literature.

- **Objective function:** The function to be maximized or minimized.
- **Solution**: An assignment of real values to the variables of the linear program.
- **Feasible solution**: A solution that satisfies all constraints.
- **Feasible region**: The set of feasible solutions; a convex polytope.
- **Basic feasible solution**: A vertex of the feasible region.
- **Optimal solution**: A feasible solution that maximizes or minimizes the objective function.

- **Value of a solution**: The value of the objective function at the solution. If the objective function is to be minimized, this is sometimes called the ***cost*** of the solution.
- **Optimal value (or cost) of the linear program**: The value (or cost) of the objective function at an optimal solution.
- **Feasible linear program**: A linear program with non-empty feasible region.
- **Infeasible linear program**: A linear program with empty feasible region.
- **Unbounded linear program**: A feasible linear program that has no optimal solution.

6.3 Linear programming algorithms

A linear programming algorithm takes as input a linear program, i.e., the (column) vectors $\mathbf{b} \in \mathbb{Q}^m$, $\mathbf{c} \in \mathbb{Q}^n$ and the $m \times n$ matrix $\mathbf{A}$ with rational-valued entries,[2] and outputs the following:

- special flags to indicate that the linear program is infeasible or unbounded; or
- if the linear program is feasible and bounded, a vector $\mathbf{x} \in \mathbb{Q}^n$ that is an optimal solution of the linear program; that is,
  (a) $\mathbf{Ax} \leq \mathbf{b}$ (i.e., $\mathbf{x}$ is a feasible solution), and
  (b) for all $\mathbf{x}' \in \mathbb{Q}^n$ such that $\mathbf{Ax}' \leq \mathbf{b}$, $\mathbf{c}^\mathsf{T} \cdot \mathbf{x} \geq \mathbf{c}^\mathsf{T} \cdot \mathbf{x}'$ (i.e., $\mathbf{x}$ maximizes the objective function over all feasible solutions).

Historically, the first person to recognize the general problem of linear programming was the Soviet mathematician Leonid Kantorovich. In 1939 he published a booklet where, among other things, he presented an algorithm he called the ***method of resolving multipliers*** for solving linear programs. This algorithm is now pretty much abandoned as there are better ones, but some of the underlying principles of linear programming, in particular an important property called ***duality***, were first presented in this work.

In this section we will briefly discuss three algorithms for solving linear programs: the ***simplex method***, the ***ellipsoid method***, and the ***interior-point method***. The first and last of these are important because they are the ones used in practice; the second was an intellectual breakthrough that opened the way to the third. The descriptions of these algorithms are given here at a very high level, omitting many important aspects and relying heavily on the geometric intuition about linear programming we developed in the previous section. If you are having a hard time following these descriptions, don't worry. They are meant as starting points for those of you who are interested in exploring fuller explanations in other sources. Knowing how these algorithms work is *not* a learning objective for this course; knowing how to *use* them to solve other problems, as discussed in the next section, *is* a learning objective.

The simplex method

The simplex method was devised by the American mathematician George Dantzig in 1947. The basic idea can be explained using the geometric intuition we developed in the previous

[2]The algorithms deal with rationals, rather than reals, since the inputs must be represented in a finite number of bits.

section: The algorithm starts at some arbitrary *vertex* of the feasible region. It then examines the vertices of the feasible region adjacent to it; if there is one in which the value of the objective function improves relative to its value in the current vertex, the algorithm moves to it and continues in the same way. The algorithm stops when it reaches a vertex that has no neighbour that improves the value of the objective function. That vertex is an optimal solution.

For example, in the linear program depicted in Figure 6.1(d), the algorithm might start at the basic feasible solution $(0, 0)$, where the objective function $3x_1 + 2x_2$ attains the value 0. It might then go to the neighbour $(10, 0)$, where the objective function attains the improved value 30. From there it would move to the neighbour $(10, 20)$, where the objective function attains the improved value 70. From there it would move to the neighbour $(4, 32)$, where the objective function attains the improved value 76. Now we are at a vertex in all neighbours of which the objective function attains a lower value: 70 at $(10, 20)$ and $200/3 \approx 66.6$ at $(0, 100/3)$. The algorithm then returns the optimal solution $(4, 32)$.

The correctness of the simplex method relies on two properties of linear programs we have seen: First, that if an optimal solution exists then there is one that is a vertex of the feasible region. And, second, that the feasible region is *convex*. This implies that if a vertex is a *local* optimum (i.e., better than all its neighbours), it is guaranteed to be a *global* optimum (i.e., better than *all* points in the feasible region). This is because convexity ensures that if all neighbours of a vertex lie on the same half-space defined by the objective function hyperplane that goes through the vertex, then the entire feasible region is on that same half-space and the vertex is an optimum. (Consider the non-convex polygon in Figure 6.2(c). Can you see why in this case a local optimum of a linear objective function is not necessarily a global one?)

The simplex method has been used for solving linear programs for a very long time and usually works very well in practice. It is, however, an exponential-time algorithm in the worst case: a convex polytope can have many vertices (for example, an n-dimensional cube has 2^n vertices). Although in practice the algorithm quickly finds a path to an optimal solution, there is no known way to ensure that it always does so; there are cases where the number of vertices it visits is exponential.

The ellipsoid method

For a long time, it was an important open question whether a polynomial-time algorithm for linear programming exists. In 1979 the Soviet (later also American) mathematician Leonid Khachiyan devised the first such algorithm, known as the ellipsoid method. An ellipsoid is a shape that generalizes in $\mathbb{R}^n$ what in two dimensions is an ellipse and in three dimensions a shape akin to a rugby ball, i.e., a symmetrically deformed sphere.

The ellipsoid method relies on a completely different idea than the simplex method. Roughly speaking it works as follows. The algorithm starts with an ellipsoid that contains the entire feasible region and therefore an optimal solution (in fact, this initial ellipsoid is a (hyper)sphere). It then checks if the centre c of the current ellipsoid is inside the feasible region.

- If it is, by examining the gradient of the objective function, the algorithm can determine

a hyperplane through the centre of the current ellispoid that separates the "good half" of the ellipsoid, i.e., the half that contains an optimal solution, from the "bad half". The algorithm then computes the smallest ellipsoid that encompasses the good half of the current ellipsoid, which becomes the current ellipsoid.

- If the centre c of the current ellipsoid is not in the feasible region the algorithm can again shrink the current ellipsoid as follows. At least one of the constraints is violated, and the hyperplane that corresponds to this constraint divides the current ellipsoid into two parts, a "good part" that contains the feasible region (the one that does *not* contain the centre) and a "bad part" that does not. It then computes the smallest ellipsoid that contains the good part, which becomes the current ellipsoid.

The algorithm continues this way by defining a sequence of shrinking ellipsoids until it gets to one with sufficiently small volume. At that point the algorithm has enough information to be able to identify an optimal solution within that ellipsoid.

It turns out that this algorithm finds an optimal solution in time polynomial in the relevant parameters: The number of variables n, the number of constraints m, and the number B of bits needed to represent the input of the linear program (the vectors $\mathbf{c}$ and $\mathbf{b}$, and the matrix $\mathbf{A}$).

The discovery of this algorithm was a major theoretical breakthrough. Despite its polynomial running time, however, the algorithm is not practical. The rate of shrinkage of the ellipsoid volume is quite slow and the constants hidden behind the asymptotic notation quite large. So this algorithm never displaced the simplex method as a practical algorithm for solving linear programs. It opened a window for the possibility of other polynomial-time linear programming algorithms, however, and it did not take long for another to emerge.

The interior-point method

Indeed in 1985, the Indian computer scientist Narendra Karmarkar devised another polynomial-time algorithm for linear programming. This was the starting point of a series of improved algorithms now known collectively as "interior-point methods". Like the ellipsoid method, the interior-point method has a numerical flavour to it, unlike the combinatorial algorithms we study in this course.

Suppose the linear program is given in standard max form, i.e., to maximize $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b}$, and let $\mathbf{a}_i$ be the i-th row of $\mathbf{A}$. We are assuming implicitly that $\mathbf{A}$ incorporates the non-negativity constraints $\mathbf{x} \geq 0$ (or, to put the inequality in the correct direction, $-\mathbf{x} \leq 0$). For each constraint $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$, define the function $\ln(b_i - \mathbf{a}_i \cdot \mathbf{x})$. Note that this function plummets precipitously towards $-\infty$ as $\mathbf{x}$ takes values for which $\mathbf{a}_i \cdot \mathbf{x}$ approaches b_i (see Figure 6.5 for the graph of the function $\ln(5 - x)$ to get a sense for this). Now define the so-called ***barrier function*** $B(\mathbf{x}) = \sum_{i=1}^{m} \ln(b_i - \mathbf{a}_i \cdot \mathbf{x})$. Thus values of $\mathbf{x}$ that *maximize* the barrier function keep $\mathbf{x}$ within the feasible region but away from the boundaries defined by the constraints; the barrier function "repels" $\mathbf{x}$ from the surface of the feasible region — hence its name.

Now replace our linear program of maximizing $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ subject to $\mathbf{A}x \leq \mathbf{b}$ with the problem of maximizing the objective function $\mathbf{c}^\mathsf{T} \cdot \mathbf{x} + \mu \cdot B(\mathbf{x})$, where μ is a parameter. This new

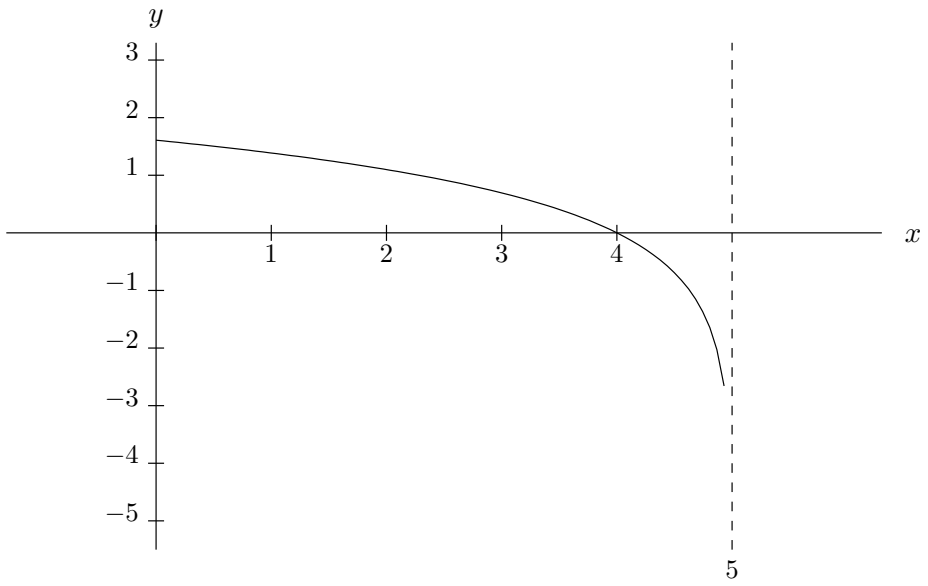


Figure 6.5: The graph of the function $\ln(5 - x)$ when $x \geq 0$

objective function is made of two terms: our original objective function $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ and the barrier function multiplied by μ. The parameter μ determines the contribution of the barrier function towards the new objective function. Note that the new objective function is no longer linear, but our problem now is an optimization problem with no explicit constraints.

For each value of the parameter μ our new objective function has an optimal solution $\mathbf{x}^*(\mu)$. When μ is large, the barrier function is the dominant term. In this case, maximizing the objective function basically amounts to maximizing the barrier, which makes $\mathbf{x}^*(\mu)$ a feasible solution that, intuitively, satisfies the constraints loosely — i.e., it stays away from all constraint boundaries. When μ is small, the contribution of the barrier function is minute and so by maximizing the new objective function we are effectively maximizing the original objective function $\mathbf{c}^\mathsf{T} \cdot \mathbf{x}$ of the linear program. Now think of varying μ from large to small values continuously: The optimal solution $\mathbf{x}^*(\mu)$ of the new objective function traces a path inside the feasible region from some initial position away from all constraint boundaries towards the optimal solution of the original linear program, always remaining inside the feasible region (hence the name "interior-point method").

The idea of the algorithm then is to start with a large μ and iteratively make it smaller and smaller. In each iteration the algorithm uses a few iterations of Newton's method to find (a value close to) the maximum of the new objective function $\mathbf{c}^\mathsf{T} \cdot \mathbf{x} + \mu \cdot B(\mathbf{x})$. It then decreases μ by a fixed factor and repeats. The iterations end when μ is sufficiently small that the algorithm can determine the constraint boundaries that determine the point (or face) of the feasible region where the optimal solution(s) of the original linear program lie, and "jumps" to one of these optimal solutions.

The values of μ decay exponentially and so the number of iterations is logarithmic in the

input size (which, you will recall, is the number n of variables, the number m of constraints, and the number B of bits required to represent the linear program). Each iteration requires polynomial time (a few iterations of Newton's method) and so the overall running time is polynomial in the input size. These days interior-point methods are competitive with the the simplex method in terms of efficiency. Both are considered practical algorithms for solving even the extremely large-scale linear programs that arise in today's applications.

* * * * *

The exact polynomial expressions for the running times of the ellipsoid and interior-point methods are rather complicated and not particularly relevant for our purposes. The key take-away point here is that there are polynomial-time algorithms for solving linear programs. We will think of these algorithms as black boxes that take as input a linear program and return as output an optimal solution (or an indication that the given linear program is infeasible or unbounded). In the next section we will see how to use such black boxes to obtain polynomial-time solutions for other problems.

Note that the running time of all known polynomial-time linear programming algorithms depends not only on the number n of variables and the number m of constraints, but also on the number B of bits needed to represent the numbers involved in the linear program. It is an open question whether a *strongly* polynomial-time algorithm for linear programming exists (see page 143).

6.4 Reductions to linear programming

As we saw with the diet problem and the product selection problem in Section 6.1, many optimization problems can be naturally expressed as linear programs. Linear programming is a very rich and expressive "language" in which one can formulate optimization problems. In this section we will see more examples of reductions of various problems to linear programming.

6.4.1 Maximum flow

Our familiar maximum flow problem, which we studied in some detail in Chapter 5, can be formulated as a linear program. Before reading on, give it a try!

Given a flow network $\mathcal{F} = (G, s, t, c)$, where $G = (V, E)$, we construct the following linear program:

- **Variables:** x_e, for each edge $e \in E$. (Intuitively, x_e is the traffic on edge e.)
- **Objective function:** Maximize $\sum_{e \in \text{out}(s)} x_e$.
- **Constraints:**
  (1) $x_e \geq 0$, for every $e \in E$;
  (2) $x_e \leq c(e)$, for every $e \in E$; and
  (3) $\sum_{e \in \text{in}(v)} x_e - \sum_{e \in \text{out}(v)} x_e = 0$, for every $v \in V - \{s, t\}$.

Constraints (1) and (2) are the capacity constraints, and constraints (3) are the conservation constraint in the maximum flow problem. Thus, if the graph has n nodes and m edges, the linear program has m variables, and $2m + (n - 2)$ constraints.

It should be clear that an optimal solution $\mathbf{x}^* = (x_e^*: e \in E)$ to the above linear program immediately translates into a maximum flow f: $E \to \mathbb{R}$ simply by taking $f(e) = x_e^*$: The fact that $\mathbf{x}^*$ is feasible (since it is in fact optimal) implies that f satisfies the capacity and conservation constraints, and the fact that $\mathbf{x}^*$ maximizes the objective function implies that there cannot be a flow of greater value than f.

The parenthetical comment that follows the definition of the variables in the above linear program is not part of the linear program itself. It is similar to a comment in a piece of code: Its function is to help the reader understand the intended role of the variables. It does not *cause* the variables to magically assume that role; it is the constraints that do that.

In designing the linear program it is important to get the constraints exactly right: They must say *everything* that is required to ensure that the variables function as intended; and they must say *nothing more* so as not to prevent the objective function from attaining its optimal value.

Note that we do not have a single linear program that solves all instances of the maximum flow problem the way we have a single algorithm (say the Ford-Fulkerson algorithm) that solves all such instances. Instead, each instance of the maximum flow problem, i.e., each flow network $\mathcal{F} = (G, s, t, c)$, results in a particular linear program whose optimal solution yields a maximum flow for that network. However, the way the linear program that solves the maximum flow for $\mathcal{F}$ is *constructed* is completely algorithmic. That is, given the input $\mathcal{F}$, there is an algorithm that transforms $\mathcal{F}$ into the linear program described above, whose optimal solution can be turned (also algorithmically) into the outuput of maximum flow on input $\mathcal{F}$. This is precisely the process of reduction of problem A into problem B, as illustrated in the diagram of Figure 6.6.

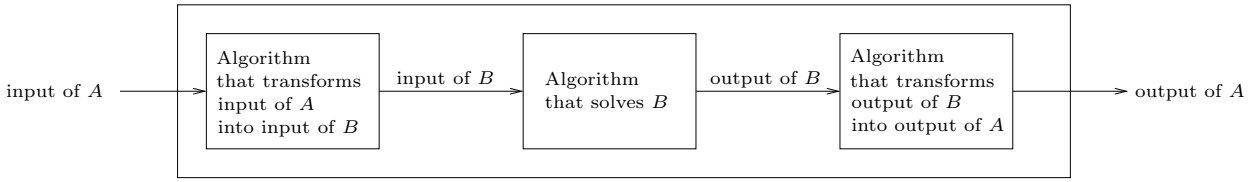


Figure 6.6: Reduction of problem A to problem B, obtaining algorithm for A

In our case, A is the maximum flow problem and B is linear programming. The algorithm that transforms the input of A to an input of B is an algorithm that takes as input a flow network $\mathcal{F}$ and outputs the corresponding linear program. The algorithm that solves B is a linear programming solver, such as the simplex or the interior-point method. The algorithm that transforms the output of B to the output of A is the (trivial) algorithm that takes as input the optimal solution $\mathbf{x}^*$ of the linear program and outputs the maximum flow f. If our goal is to obtain a polynomial-time algorithm for A we must make sure that all three "boxes" in the

diagram are polynomial-time algorithms: The leftmost box must translate the input of A to an input of B in polynomial time (and therefore the input of B produced must be of polynomial size), the B-solver in the middle box must run in polynomial time, and the translation of the output of B into the correct output of A must also be done in polynomial time. In our example this is the case if we choose the interior-point method as our linear programming solver; the transformers of inputs of A (flow network) to inputs of B (linear program) and outputs of B (optimal solution to linear program) to outputs of A (maximum flow) are easy to implement in linear time.

In Chapter 5 we saw specialized algorithms for solving the maximum flow problem. These are much more efficient than using linear programming solvers. The purpose of the reduction to linear programming that we saw here was not as a practical way to solve the problem but as an illustration of the process of reduction in a simple and familiar setting.

6.4.2 Minimum cost flow

Closely related to the maximum flow problem is the so-called ***minimum-cost flow problem***. Imagine a flow network where "traffic" is generated at the source s and d units of that traffic must be transported to the destination t. Each edge e has, in addition to its capacity $c(e)$, a cost $\kappa(e) \geq 0$, which is the cost *per unit* of traffic transported along e; that is, if we have a flow f, the cost of the traffic transported on edge e is $\kappa(e) \cdot f(e)$, and the ***cost of the flow*** f is $\sum_{e \in E} \kappa(e) \cdot f(e)$. The goal is to determine if it is possible to transport d units of the product from s to t via the edges of G and, if so, to find the cheapest way to do it. So, our problem is:

Input: A flow network $\mathcal{F} = (G, s, t, c)$, a cost function κ: $E \to \mathbb{R}_0^+$ (the non-negative reals) and a demand $d \in \mathbb{R}_0^+$.

Output: A minimum-cost flow of value d, if such a flow exists; otherwise a flag to indicate that no flow can have value d.

This can be formulated as a linear program very much along the lines of the maximum flow linear program. Given the flow network $\mathcal{F}$, cost function κ, and demand d, construct the following linear program.

- **Variables:** x_e, for each edge $e \in E$. (Intuitively, x_e is the traffic on edge e.)
- **Objective function:** Minimize $\sum_{e \in E} \kappa(e) \cdot x_e$.
- **Constraints:**
  (1) $x_e \geq 0$, for every $e \in E$;
  (2) $x_e \leq c(e)$, for every $e \in E$;
  (3) $\sum_{e \in \text{in}(v)} x_e - \sum_{e \in \text{out}(v)} x_e = 0$, for every $v \in V - \{s, t\}$; and
  (4) $\sum_{e \in \text{out}(s)} x_e = d$.

Constraints (1)-(3) are exactly as in the linear program for maximum flow. Constraint (4) ensures that a feasible solution defines a flow of value d.

An optimal solution $\mathbf{x}^* = (x_e \colon e \in E)$ of this linear program, if one exists (see Exercise 6.1) gives rise to a flow f: $E \to \mathbb{R}_0^+$ defined by $f(e) = x_e$ that is exactly what we are looking for: A flow of value d that has minimum cost among all flows with value d.

The minimum-cost flow problem can be solved by specialized algorithms that are more efficient than solving this linear program. Notice, however, how the two problems, maximum flow and minimum-cost flow, are closely related and their linear programs are accordingly similar: We obtained our linear program for minimum-cost flow from the maximum flow linear program simply by adding one constraint and adjusting the cost function! In contrast, the specialized algorithms for solving minimum-cost flow are not all that closely related to the specialized algorithms for solving maximum flow. This demonstrates the power and flexibility of linear programming as a language for expressing different problems.

6.4.3 The transportation problem

Here is another classical optimization problem that arises in supply-chain management and can be formulated naturally as a linear program. Suppose we have m suppliers, labeled $1, 2, \ldots, m$, that make some product and n consumers of that product, labeled $1, 2, \ldots, n$. For our current purposes this is a product that can be made, consumed and transported in arbitrary fractional quantities. For example, the suppliers could be fuel depots, the consumers could be gas stations, and the product could be gasoline. Supplier i can produce s_i units of the good and consumer j requires d_j units of the good. Transporting one unit of the good from supplier i to consumer j costs c_{ij}. The problem is to determine the *cheapest* way of transporting the product from suppliers to consumers so as to satisfy every consumer's demand without exceeding any supplier's production capacity. We can formulate this as the following linear program:

- **Variables:** x_{ij}, for each supplier $i \in [1..m]$ and consumer $j \in [1..n]$. (Intuitively, x_{ij} is the amount of the good that supplier i ships to consumer j.)
- **Objective function:** Minimize $\sum_{\substack{i\in[1..m]\\ j\in[1..n]}} c_{ij} \cdot x_{ij}$.
- **Constraints:**
  (1) $x_{ij} \geq 0$, for every $i \in [1..m]$ and $j \in [1..n]$;
  (2) $\sum_{j=1}^{n} x_{ij} \leq s_i$, for each $i \in [1..m]$; and
  (3) $\sum_{i=1}^{m} x_{ij} \geq d_j$, for each $j \in [1..n]$.

Constraints (1) state that we cannot ship a negative amount of the good between any supplier and consumer. Constraints (2) state that the total shipments out of supplier i cannot exceed the supplier's production capacity. Constraints (3) state that the total shipments sent to consumer j must meet the consumer's demand.

You can check that this linear program is feasible if and only if the total supply is sufficient to meet the total demand, i.e., $\sum_{i=1}^{m} s_i \geq \sum_{j=1}^{n} d_j$. If the linear program is feasible then it is bounded (since the objective function cannot be negative at any feasible solution), and its optimal solution $\mathbf{x}^* = (x^*_{ij}: i \in [1..m], j \in [1..n])$ gives rise to a minimum-cost schedule for shipping the good from suppliers to consumers: send x^*_{ij} units from supplier i to consumer j.

6.4.4 Shortest paths

Here we will formulate the single-pair version of the shortest paths problem as a linear program; in fact we will do so in two different ways. The specialized algorithms for shortest-paths that we studied in Chapter 4 are much more efficient than translating the problem into a linear program, and then solving that linear program. Nevertheless the reductions we will see in this subsection are interesting and instructive.

Recall that in the single-pair shortest path problem we are given as input a directed graph $G = (V, E)$, an edge-weight function $\mathbf{wt}$: $E \to \mathbb{R}$, and two nodes $s, t \in V$, and the goal is to find a shortest $s \rightsquigarrow t$ path or determine that no such path exists. There are two reasons why a shortest $s \rightsquigarrow t$ path might not exist: (a) there is no $s \rightsquigarrow t$ path at all in the graph or (b) there is an $s \rightsquigarrow t$ path that contains a negative-weight cycle. As we will see, the three possibilities (no $s \rightsquigarrow t$ path, no *shortest* $s \rightsquigarrow t$ path, and well-defined shortest $s \rightsquigarrow t$ path) line up neatly with the three possible outcomes of a linear program (infeasible, unbounded, or feasible and bounded).

Nodes that are not on any $s \rightsquigarrow t$ path are not relevant for finding a shortest $s \rightsquigarrow t$ path. so we will assume that the graph G given to us does not have such nodes — except possibly s and t in case G has no $s \rightsquigarrow t$ path. That is, we will assume that for every node $u \in V - \{s, t\}$, there is a $s \rightsquigarrow u$ and a $u \rightsquigarrow t$ path in G. (Note that, as a consequence, if there is no $s \rightsquigarrow t$ path, G consists of the two nodes s and t, and no edges.)

Alternatively we could construct the linear program that corresponds to finding a shortest $s \rightsquigarrow t$ path in G by defining the graph $G' = (V', E')$ where

$$V' = \{u\text{: } u \in V \text{ and } u \text{ is on some } s \rightsquigarrow t \text{ path}\} \cup \{s, t\}, \text{and}$$
$$E' = \{(u, v)\text{: } u, v \in V' \text{ and } (u, v) \in E\}.$$

Note that p is an $s \rightsquigarrow t$ path in G if and only if p is an $s \rightsquigarrow t$ path in G'. We avoid the needless complication of introducing this new graph and simply assume that the given graph has no nodes and edges that are unnecessary for $s \rightsquigarrow t$ paths.

First reduction

Given the graph $G = (V, E)$, the weight function $\mathbf{wt}$: $E \to \mathbb{R}$, and the nodes $s, t \in V$, we construct the following linear program, which we will call SPLP1 (for "shortest path linear program 1" — there will be a SPLP2).

- **Variables:** x_u, for each $u \in V$. (Intuitively x_u is a *lower bound* on the length of a shortest $s \rightsquigarrow u$ path.)
- **Objective function:** Maximize x_t.
- **Constraints:**
  (1) $x_s = 0$; and
  (2) $x_v - x_u \leq \mathbf{wt}(u, v)$, for each $(u, v) \in E$.

Constraint (1) states that zero is a lower (in fact, exact) bound on the length of a shortest path from s to itself. Constraints (2) state that the length of a shortest $s \rightsquigarrow v$ path cannot

exceed the length of a shortest path to a predecessor u of v plus the weight of the edge (u, v). This is a special case of the triangle inequality (4.1).

It may appear paradoxical that a linear program formulation of the *shortest*-path problem has a maximization goal! Recall, however, that x_u is intended to be a *lower* bound on a shortest $s \rightsquigarrow u$ path, so it is maximized when it is *equal* to that.

Here is a visualization that may help expain how this maximization linear program correctly captures the single-pair shortest-path problem (this visualization is meaningful only when the edge weights are non-negative). Imagine a physical model of the edge-weighted graph where each edge (u, v) is made of (flexible but inelastic) string connecting u and v and has length equal to the edge's weight. Now take the nodes s and t and move them away from each other. When s and t are stretched as far apart as the pieces of string permit, the string will be taut along all shortest $s \rightsquigarrow t$ paths, with the remaining nodes hanging loose. The maximum value of x_t is precisely the length of the taut string, i.e., the length of the shortest $s \rightsquigarrow t$ paths.

The following two lemmas relate paths in G to feasible solutions of SPLP1.

Lemma 6.3 *If $p = u_1, u_2, \ldots, u_k$ is a path in G and $\mathbf{x} = (x_u\colon u \in V)$ is a feasible solution of SPLP1, then $x_{u_k} \leq x_{u_1} + \boldsymbol{wt}(p)$.*

PROOF. Apply repeatedly constraint (2) on edges $(u_{k-1}, u_k), (u_{k-2}, u_{k-1}), \ldots, (u_1, u_2)$. □

Lemma 6.4 *Suppose G has no negative-weight cycle. For each $u \in V$ let $\hat{x}_u$ be the length of a shortest $s \rightsquigarrow u$ path in G and, if t is not reachable from s, then let $\hat{x}_t$ be <u>any</u> real number. Then $\hat{\mathbf{x}} = (\hat{x}_u\colon u \in V)$ is a feasible solution of SPLP1.*

PROOF. Since G has no negative-weight cycle, there is a shortest $s \rightsquigarrow u$ path for every $u \in V$ except possibly t. Therefore, $\hat{\mathbf{x}}$ is well-defined. Clearly $\hat{x}_s = 0$, so $\hat{\mathbf{x}} = (\hat{x}_u\colon u \in V)$ satisfies constraint (1). It also satisfies constraints (2): For every node u reachable from s this follows by the triangle inequality (4.1); and if t is not reachable from s it is true because t has no predecessor in G, by our assumption that every node other than s and t is on an $s \rightsquigarrow t$ path. □

We can now state and prove the theorem that relates SPLP1 to shortest $s \rightsquigarrow t$ paths in G.

Theorem 6.5 *The following are true:*
(a) SPLP1 is infeasible if and only if G has a negative-weight cycle.
(b) SPLP1 is unbounded if and only if G has no negative-weight cycle and no $s \rightsquigarrow t$ path.
(c) If G has an $s \rightsquigarrow t$ path and no negative-weight cycle then the optimal value of SPLP1 is the length of a shortest $s \rightsquigarrow t$ path.

PROOF. <u>Part (a)</u>: [IF] We prove the contrapositive. Let $\mathbf{x} = (x_u\colon u \in V)$ be any feasible solution of SPLP1 and $c = u_1, u_2, \ldots, u_{k-1}, u_1$ be any cycle in G. By Lemma 6.3, $x_{u_1} \leq x_{u_1} + \mathbf{wt}(c)$, therefore $\mathbf{wt}(c) \geq 0$. So, G has no negative-weight cycle.

[ONLY IF] By (the contrapositive of) Lemma 6.4.

Part (b): [IF] Suppose G has no negative-weight cycle and no $s \rightsquigarrow t$ path. Then the feasible solution $\hat{\mathbf{x}}$ in Lemma 6.4 can assign to x_t an arbitrary real number, so SPLP1 is unbounded.

[ONLY IF] Suppose SPLP1 is unbounded. Therefore it is feasible and, by part (a), there is no negative-weight cycle reachable from s. Suppose, for contradiction, that there is an $s \rightsquigarrow t$ path p. By Lemma 6.3, for any feasible solution $\mathbf{x} = (x_u\colon u \in V)$, $x_t \leq x_s + \mathbf{wt}(p)$. By constraint (1), $x_s = 0$, so $x_t \leq \mathbf{wt}(p)$, which contradicts that SPLP1 is unbounded.

Part (c): Suppose G has an $s \rightsquigarrow t$ path and no negative-weight cycle. Then there is a shortest $s \rightsquigarrow t$ path, say $\hat{p}$. By parts (a) and (b), SPLP1 has an optimal solution, say $\mathbf{x}^* = (x_u^*\colon u \in V)$. Since $\mathbf{x}^*$ is optimal, it is feasible and so, by Lemma 6.3 and constraint (1),

$$x_t^* \leq x_s^* + \mathbf{wt}(\hat{p}) = \mathbf{wt}(\hat{p}) \tag{6.1}$$

Since G has an $s \rightsquigarrow t$ path and no negative-weight cycle, the feasible solution $\hat{\mathbf{x}}$ defined in Lemma 6.3 assigns to $\hat{x}_t$ the length of a shortest $s \rightsquigarrow t$ path, i.e., $\hat{x}_t = \mathbf{wt}(\hat{p})$. Since $\mathbf{x}^*$ is optimal and $\hat{\mathbf{x}}$ is feasible,

$$x_t^* \geq \hat{x}_t = \mathbf{wt}(\hat{p}) \tag{6.2}$$

From (6.1) and (6.2) it follows that $x_t^* = \mathbf{wt}(\hat{p})$, so the optimal value of SPLP1 is the length of a shortest $s \rightsquigarrow t$ path, as wanted. □

Second reduction

Our second reduction turns the single-pair shortest paths problem into an instance of the minimum-cost flow problem which, as we saw in Section 6.4.2, can be formulated as a linear program.

Let $G = (V, E)$ be a directed graph, $\mathbf{wt}\colon E \to \mathbb{R}$ be an edge-weight function, and $s, t \in V$ be nodes of G. As before, we will assume that every node of G other than s and t is on some $s \rightsquigarrow t$ path. In addition, because we now want to think of G as the graph of a flow network, we will assume that G has no edge into s or out of t. Note that such edges cannot contribute to a shortest $s \rightsquigarrow t$ path in G, if such a path exists at all.

Consider now the flow network $\mathcal{F} = (G, s, t, c)$, where $c(e) = \infty$ for every $e \in E$ — i.e., all edges have unbounded capacity.[3] Given a simple $s \rightsquigarrow t$ path p in G, the function

$$f(e) = \begin{cases} 1, & \text{if } e \text{ is on } p \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

is a binary flow in $\mathcal{F}$. If we interpret the weight $\mathbf{wt}(e)$ of edge e as the cost of putting a unit of traffic through e, the length of path p is precisely the cost of the corresponding binary flow f. So, seeking a shortest path amounts to seeking a minimum-cost flow. This is the intuition underlying the following linear program SPLP2 to which we will reduce the problem of finding a shortest $s \rightsquigarrow t$ path.

[3]Such flow networks are called ***uncapacitated***.

- **Variables:** x_e, for each $e \in E$. (Intuitively x_e is the traffic on edge e of a flow in $\mathcal{F}$.)
- **Objective function:** Minimize $\sum_{e \in E} \mathbf{wt}(e) \cdot x_e$.
- **Constraints:**
  (1) $x_e \geq 0$, for each $e \in E$;
  (2) $\sum_{e \in \text{in}(u)} x_e - \sum_{e \in \text{out}(u)} x_e = 0$, for each $u \in V - \{s, t\}$; and
  (3) $\sum_{e \in \text{out}(s)} x_e = 1$.

Let $\mathbf{x} = (x_e\colon e \in E)$ be a solution of SPLP2. Think of x_e as the amount of traffic on edge e. Constraints (1) say that each edge carries a non-negative amount of traffic, and so the capacity constraint is satisfied, since the edges in $\mathcal{F}$ have unbounded capacity. Constraints (2) say that, for any node u other than s and t, the total traffic into u equals the total traffic out of u; i.e., the conservation constraint is satisfied. Constraint (3) says that the amount of traffic out of s is one. So, if $f_{\mathbf{x}}\colon E \to \mathbb{R}$ is the function defined by $f_{\mathbf{x}}(e) = x_e$, we have:

Lemma 6.6 *$\boldsymbol{x}$ is a feasible solution of SPLP2 if and only if $f_{\boldsymbol{x}}$ is a flow of value 1 in $\mathcal{F}$.*

In motivating the linear program SPLP2 as a formulation of the shortest-path problem, we saw how the length of an $s \rightsquigarrow t$ path is the cost of the corresponding *binary* flow as defined in (6.4); therefore, the length of a *shortest* path is the *minimum* cost of a binary flow. We are not quite done, however: The optimal solution $\mathbf{x}^*$ to SPLP2 might be fractional, so it does not correspond to a binary flow and therefore to a single $s \rightsquigarrow t$ path. Could the optimal solution to SPLP2 (which may be *fractional* and therefore consist of *multiple* $s \rightsquigarrow t$ paths) correspond to a flow whose cost is *less* than the length of a shortest path (which corresponds to a *binary* flow consisting of a *single* $s \rightsquigarrow t$ path)? It turns out that this is not possible, and we can bring to bear the flow decomposition theorem to show why. Let $F(\mathbf{x})$ be the objective function of SPLP2; i.e., $F(\mathbf{x}) = \sum_{e \in E} \mathbf{wt}(e) \cdot x_e$.

Lemma 6.7 *If G has an $s \rightsquigarrow t$ path and no negative-weight cycle then for any feasible solution $\boldsymbol{x}$ of SPLP2, $F(\mathbf{x}) \geq \mathbf{wt}(\hat{p})$, where $\hat{p}$ is a shortest $s \rightsquigarrow t$ path.*

Proof. Suppose G has an $s \rightsquigarrow t$ path and no negative-weight cycle. Thus, there is a shortest $s \rightsquigarrow t$ path, say $\hat{p}$. Let $\mathbf{x}$ be any feasible solution of SPLP2. By Lemma 6.6, $f_{\mathbf{x}}$ is a flow and $\mathcal{V}(f_{\mathbf{x}}) = 1$. By the flow decomposition theorem (Theorem 5.16), $f_{\mathbf{x}} = \sum_{p \in \mathcal{P}} d_p \cdot f_p + \sum_{c \in \mathcal{C}} d_c \cdot f_c$, for some set $\mathcal{P}$ of $s \rightsquigarrow t$ simple paths, set $\mathcal{C}$ of simple cycles, and positive constants d_p for every $p \in \mathcal{P}$ and d_c for every $c \in \mathcal{C}$, where f_p and f_c are the binary flows

$$f_p(e) = \begin{cases} 1, & \text{if } e \text{ is on path } p \\ 0, & \text{otherwise} \end{cases} \quad \text{and} \quad f_c(e) = \begin{cases} 1, & \text{if } e \text{ is on cycle } c \\ 0, & \text{otherwise.} \end{cases}$$

So,

$$\begin{aligned}
F(\mathbf{x}) &= \sum_{e\in E} \mathbf{wt}(e)\cdot x_e \\
&= \sum_{e\in E} \mathbf{wt}(e)\cdot f_{\mathbf{x}}(e) \\
&= \sum_{e\in E}\Big(\mathbf{wt}(e)\cdot\big(\sum_{\substack{p\in\mathcal{P}\text{ and}\\ e\text{ is on }p}} d_p + \sum_{\substack{c\in\mathcal{C}\text{ and}\\ e\text{ is on }c}} d_c\big)\Big) && \text{[where a sum over an empty set of terms is zero]} \\
&= \sum_{p\in\mathcal{P}}\big(d_p\cdot\mathbf{wt}(p)\big) + \sum_{c\in\mathcal{C}}\big(d_c\cdot\mathbf{wt}(c)\big) && \text{[by grouping the terms in the above sum that correspond to the edges of each } p\in\mathcal{P} \text{ and } c\in\mathcal{C}] \\
&\geq \sum_{p\in\mathcal{P}}\big(d_p\cdot\mathbf{wt}(p)\big) && \text{[since } G \text{ has no negative-weight cycle]} \\
&\geq \sum_{p\in\mathcal{P}}\big(d_p\cdot\mathbf{wt}(\hat{p})\big) && \text{[since } \hat{p} \text{ is a shortest } s\rightsquigarrow t \text{ path]} \\
&= \mathbf{wt}(\hat{p}) && \text{[since } \sum_{p\in\mathcal{P}} d_p = \mathcal{V}(f_{\mathbf{x}}) = 1] \qquad \square
\end{aligned}$$

We can now state and prove the theorem that relates SPLP2 to shortest $s\rightsquigarrow t$ paths in G.

Theorem 6.8 *The following are true:*
(a) SPLP2 is infeasible if and only if there is no $s\rightsquigarrow t$ path in G.
(b) SPLP2 is unbounded if and only if G has an $s\rightsquigarrow t$ path and a negative-weight cycle.
(c) If G has an $s\rightsquigarrow t$ path and no negative-weight cycle, then the optimal value of SPLP2 is the length of a shortest $s\rightsquigarrow t$ path.

PROOF. Part (a): [IF] We prove the contrapositive: Suppose $\mathbf{x}$ is a feasible solution. By Lemma 6.6, $f_{\mathbf{x}}$ is a flow of value one. Consider a decomposition of $f_{\mathbf{x}}$. The set of simple $s\rightsquigarrow t$ paths in this decomposition is nonempty, because $\mathcal{V}(f_{\mathbf{x}}) = 1$. Therefore there is an $s\rightsquigarrow t$ path. [ONLY IF] We prove the contrapositive. Suppose p is an $s\rightsquigarrow t$ path; without loss of generality, we assume that p is simple.[4] For each $e\in E$, let

$$x_e = \begin{cases} 1, & \text{if } e \text{ is on } p \\ 0, & \text{otherwise.} \end{cases}$$

You can check that $\mathbf{x} = (x_e\colon e\in E)$ satisfies the constraints (1)-(4), so $\mathbf{x}$ is a feasible solution. Part (b): [IF] Suppose G has an $s\rightsquigarrow t$ path and a negative-weight cycle c. By part (a), SPLP2 has a feasible solution, say $\mathbf{x} = (x_e\colon e\in E)$. We will prove that SPLP2 is unbounded; that is, for *any* bound B, however small, there is a feasible solution $\mathbf{x}'$ such that $F(\mathbf{x}')\leq B$. We will

[4] If there is an $s\rightsquigarrow t$ path, there is a simple such path, by deleting any cycles.

obtain $\mathbf{x}'$ by modifying $\mathbf{x}$ by putting an arbitrary amount of additional traffic on every edge of the cycle c, causing the value of the objective function to decrease by an arbitrary amount.

More specifically, let B be any real number, however small. If $F(\mathbf{x}) \leq B$, take $\mathbf{x}' = \mathbf{x}$ and we are done: $F(\mathbf{x}') \leq B$. Otherwise, let $k = \big(B - F(\mathbf{x})\big)/\mathbf{wt}(c)$; note that, since $F(\mathbf{x}) > B$ and $\mathbf{wt}(c) < 0$, $k > 0$. Intuitively, k is the amount of additional traffic we will put in each edge on the cycle c to obtain the feasible solution $\mathbf{x}'$ the makes the value of the objective function B or less. So let $\mathbf{x}' = (x'_e\colon e \in E)$ be the solution of SPLP2 defined by

$$x'_e = \begin{cases} x_e + k, & \text{if } e \text{ is on cycle } c \\ x_e, & \text{otherwise.} \end{cases}$$

You can check that $\mathbf{x}'$ is a feasible solution, given that $\mathbf{x}$ is. Furthermore,

$$F(\mathbf{x}') = F(\mathbf{x}) + k \cdot \mathbf{wt}(c) = F(\mathbf{x}) + \Big(\frac{B - F(\mathbf{x})}{\mathbf{wt}(c)}\Big) \cdot \mathbf{wt}(c) = B.$$

So, $F(\mathbf{x}') \leq B$, as wanted.

[Only if] Suppose SPLP2 is unbounded and therefore, in particular, it is feasible. So, by part (a), G has a $s \rightsquigarrow t$ path. It remains to show that G has a negative-weight cycle. Suppose, for contradiction, that it does not. By Lemma 6.7, for any feasible solution $\mathbf{x}$, $F(\mathbf{x}) \geq \mathbf{wt}(\hat{p})$, contradicting that SPLP2 is unbounded.

Part (c): Suppose G has an $s \rightsquigarrow t$ path and no negative-weight cycle. Therefore G has a shortest $s \rightsquigarrow t$ path, say $\hat{p}$. Without loss of generality, we assume that $\hat{p}$ is a simple path. By parts (a) and (b), SPLP2 is feasible and bounded, so it has an optimal solution, say $\mathbf{x}^*$. In particular, $\mathbf{x}^*$ is feasible and so, by Lemma 6.7,

$$F(\mathbf{x}^*) \geq \mathbf{wt}(\hat{p}) \tag{6.4}$$

Let $\hat{\mathbf{x}} = (\hat{x}_e\colon e \in E)$, where

$$\hat{x}_e = \begin{cases} 1, & \text{if } e \text{ is on } \hat{p} \\ 0, & \text{otherwise} \end{cases}$$

so $F(\hat{\mathbf{x}}) = \mathbf{wt}(\hat{p})$. It is easy to check that $\hat{\mathbf{x}}$ is feasible. By the optimality of $\mathbf{x}^*$,

$$F(\mathbf{x}^*) \leq F(\hat{\mathbf{x}}) = \mathbf{wt}(\hat{p}) \tag{6.5}$$

By (6.4) and (6.5), $F(\mathbf{x}^*) = \mathbf{wt}(\hat{p})$. Thus, the value of an optimal solution of SPLP2 is the length of a shortest $s \rightsquigarrow t$ path in G, as wanted. □

6.4.5 Linear regression

Linear regression is a statistical technique used for predicting an *unknown outcome* for an individual (person, entity, or event) based on *known attributes* of that individual, using *past observations* of individuals where both the attributes and the outcomes are known.

For example, to assess property taxes a municipality must assign a value to each house. The value of the house is not known (until it is sold), but some attributes that determine its value are known — e.g., the area of the house's living space, the number of bedrooms it contains, its distance from the city centre, and its age. The municipality has data on the actual sale prices of houses sold in its jurisdiction in recent years as well as their attributes, so it uses this information to come up with a linear function that describes how the price of a house depends on these attributes and uses this function to assign values to all houses. In this example, the outcome is the value of the house, the known attributes are the area, number of bedrooms, distance from city centre, and age of the house, and the past observations are the recent house sales. Note that there is a big assumption here, namely that the dependence of the outcome on the attributes is described by a linear function. This turns out to be a reasonably good approximation in many cases, but we should not forget that it is an assumption we are imposing on the world in our attempt to understand it; it is not necessarily a fact of the world.

As another example, to develop the policy for admission to a restricted program like computer science, a university might want to predict each student's potential for success in the program as a function of the student's performance in certain first-year courses. In this example the outcome we want to predict might be graduation GPA, and the known attributes might be the scores attained in the selected first-year courses. The past observations are graduated students' performance in those first-year courses and their actual graduation GPA.

In a machine-learning application of linear regression, we might want to perform so-called "sentiment analysis" of texts, say reviews on a web site. In this case the attributes are described by a huge vector that gives, for each word of the English language, the number of occurrences of that word in the text. The outcome might be a rating between 0 (terrible review) to 5 (excellent review). Someone (or a group of individuals, if we want to get a more diverse set of opinions) can manually rate a sample of the reviews so that we have a set of observations. Using these observations we can develop a function of how ratings depend on the word frequencies in texts and use this function to automatically score all reviews.

In linear regression the input is the set of observations, where both the attributes and the outcome are known, and the output is a linear function that describes how the outcome depends on the attribute values; we want this function to "best fit" the given observations. There are several ways of defining what this means mathematically; more on this shortly.

The concept is illustrated in Figure 6.7(a) for a very simple case where we have a single attribute x_1 (the horizontal axis) on which the outcome y (the vertical axis) depends. The dots represent the observations, where for certain individuals we know both the attribute and the associated outcome. The line on the plane shown in green represents the linear function that "best fits" the data. So you can think of the input of linear regression as the set of dots and the output as a pair of numbers (β_0, β_1), which specifiy the linear function $y = \beta_0 + \beta_1 x_1$. We can now use this line to predict the outcome (y-value) for an individual not previously encountered whose attribute (x_1-value) we know.

If the outcome y depended on two attributes x_1 and x_2, the input would be represented by a set of points in three-dimensional space, and the output would be a triple $(\beta_0, \beta_1, \beta_2)$ representing the linear function $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$. This now defines a plane in space, rather

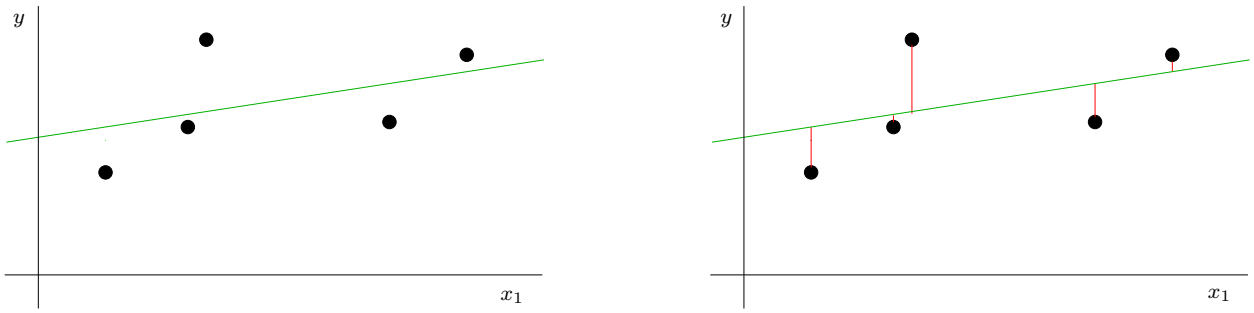


Figure 6.7: Illustration of linear regression

than a line on the plane. In some applications, such as the text sentiment analysis application described above, the outcome (score) might depend on tens of thousands of attributes (word frequencies), and the linear function defines a hyperplane in the attributes-outcome space.

A side note about terminology. In the preceding informal description I spoke of unknown "outcomes", known "attributes", and "past observations". In statistics, the discipline in which linear regression originated, attributes are called ***independent variables***, the outcome is called the ***dependent variable***, and the linear function is called the ***regression equation***. As computer scientists we have a bad habit of invading other disciplines' territories and redefining their terminology. So, in machine learning, which uses regression extensively, the observations are called ***training data***, attributes are called ***features***, outcomes are called ***labels***, and the linear function is called the ***model***.

Let's now set up linear regression as a computational problem. We assume that the independent variables and the dependent variable take on real values. Let d be the number of independent variables and m be the number of observations. We are given as input m observations. The i-th observation, $i \in [1..m]$, is described as a pair $(\mathbf{a}_i, y_i)$, where $\mathbf{a}_i = (a_{i1}, a_{i2}, \ldots, a_{id}) \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. The interpretation of this is that, in the i-th observation, a_{ij} is the value of the j-th independent variable, $j \in [1..d]$, and y_i is the value of the dependent variable. The linear function we want to determine based on these observations is a function of the form

$$f(x_1, \ldots, x_d) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_d x_d = \beta_0 + \sum_{j=1}^{d} \beta_j x_j \tag{6.6}$$

for some coefficients $\beta_0, \beta_1, \ldots, \beta_d$. Our goal is to determine these coefficients. We want the function f that they define to have the property that it is "close" to the observations. That is, if we plug-in $x_j = a_{ij}$ for each independent variable $j \in [1..d]$, we want $f(a_{i1}, \ldots, a_{id})$ to be "close" to the observed value y_i of the dependent variable, and we want this to be the case for *all* observations: we want a linear function of the form (6.6) that "best fits" all observations.

A natural and useful way to define what "best fits" means in this context is that our linear function should minimize the so-called ***absolute error***, as we now explain. Given a linear

function $f(x_1, \ldots, x_d)$, the ***error of f for the i-th observation*** $(\boldsymbol{a}_i, y_i)$ is defined as

$$E_i(\mathbf{a}_i, y_i) = |f(a_{i1}, \ldots, a_{id}) - y_i| = \left|\left(\beta_0 + \sum_{j=1}^{d} \beta_j a_{ij}\right) - y_i\right|$$

i.e., how much off (in absolute terms) $f(a_{i1}, \ldots, a_{id})$ is from the value y_i of the dependent variable in the i-th observation. In terms of the illustration in Figure 6.7(b), these are the lengths of the red segments — the vertical distance of each observation from the line. The ***absolute error of*** f (with respect to the m observations) is then defined as $\sum_{i=1}^{m} E_i(\mathbf{a}_i, y_i)$. We want to find the coefficients $\beta_0, \ldots, \beta_d$ of a linear function $f(x_1, \ldots, x_d)$ that minimizes the absolute error. In terms of Figure 6.7(b), this is the sum of the lengths of the red segments. We take absolute values to avoid positive and negative errors from canceling each other out.[5]

To summarize, the ***linear regression problem*** can be stated as follows:

Input: Observations $(\mathbf{a}_i, y_i)$, where $\mathbf{a}_i = (a_{i1}, \ldots, a_{id}) \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, for $i \in [1..m]$.
Output: Coefficients $\beta_0, \beta_1, \ldots, \beta_d \in \mathbb{R}$ of a linear function $f(x_1, \ldots, x_n) = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$ such that $\sum_{i=1}^{m} E_i(\mathbf{a}_i, y_i)$ is minimized.

This is *almost* a linear program:

- **Variables:** $\beta_0, \beta_1, \ldots, \beta_d$. (These are the coefficients of the desired linear function.)
- **Objective function:** Minimize $\sum_{i=1}^{m} E_i(\mathbf{a}_i, y_i)$.
- **Constraints:** None!

The only problem is that our objective function is not linear: it contains terms with absolute values and absolute value is not a linear operator. Fortunately we can use a trick to replace the non-linear objective function with a linear one by introducing new variables and some constraints.

Suppose F is some function and we want to minimize its absolute value $|F|$. Since $|F| = \max(F, -F)$, *minimizing* $|F|$ amounts to minimizing a new quantity Z subject to the constraints that $Z \geq F$ and $Z \geq -F$. The constraints ensure that Z takes on non-negative values and so its *minimum* value is the minimum value of $|F|$. We can therefore turn our minimization problem involving absolute values into a *bona fide* linear program as follows:

- **Variables:** $\beta_0, \beta_1, \ldots \beta_d$, and $Z_1, Z_2, \ldots Z_m$. (The β_js are the coefficients of the linear function we want to find, as before; the Z_is are new variables introduced to eliminate the absolute values from the objective function.)
- **Objective function:** Minimize $\sum_{i=1}^{m} Z_i$.

[5]Another way to avoid positive and negative errors canceling each other is to *square* them, leading to ***least-squares regression***, which seeks to find a linear function that minimizes the sum of the squares (rather than the sum of the absolute values) of the errors. Both definitions of error functions have advantages and disadvantages. The technique we will describe here applies only to minimizing the absolute error.

- **Constraints:** For each $i \in [1..m]$,

$$Z_i \geq \left(\beta_0 + \sum_{j=1}^{d} \beta_j a_{ij}\right) - y_i, \quad \text{and}$$

$$Z_i \geq -\left(\left(\beta_0 + \sum_{j=1}^{d} \beta_j a_{ij}\right) - y_i\right).$$

So, we can reduce any instance of the linear regression problem to linear programming as follows: Given the m observations $(\mathbf{a}_i, y_i)$, where $\mathbf{a}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, we construct the above linear program. The values of variables $\beta_0, \beta_1, \ldots, \beta_d$ in the optimal solution of this linear program are the coefficients of our linear regression function; the values of variables $Z_1, \ldots, Z_m$ in the optimal solution are ignored; they were only introduced to formulate the linear program properly, with a linear objective function. Note that the linear program we constructed has only $m+d+1$ variables and $2m$ constraints, and the coefficients of its constraints (the a_{ij}s and y_is) are numbers also contained in the input. So the size of the linear program is polynomial in the size of the input to the linear regression problem we wanted to solve.

6.5 Integer and 0-1 linear programming

Recall that the variables in a linear program range over the reals (or rationals). Sometimes, however, we seek an optimal solution that requires the variables to take on *integer* values. For instance, it is perfectly reasonable for an optimal solution to be $x_1 = 7.3$ and $x_2 = 11.7$ if x_1 and x_2 represent hectares planted with wheat and soybeans, respectively, for an instance of the product selection problem faced by a farmer; but not if they represent the number of tables and chairs for an instance of the product selection problem faced by a carpenter whose products are only useful in discrete quantities.

Linear programming with the additional constraint that all variables must be integers is called ***integer linear programming*** (abbreviated ***ILP***) and the important special case in which the variables can take on only the values 0 and 1 is called ***0-1 linear programming*** (abbreviated ***0-1 LP***). Note that this additional constraint is *not* a linear constraint; it cannot be expressed as a set of linear inequalities involving the variables.

It turns out that this additional requirement makes the problem *very* different and apparently computationally more difficult than linear programming: Integer linear programming and 0-1 linear programming are both NP-hard problems. They belong to this important class of computational problems that we do not know how to solve in polynomial time, but which are polynomial-time reducible to each other. So, if *any one* of them could be solved in polynomial time, all of them would!

Many of the nice properties that make linear programming (over the reals) computationally tractable are no longer true: Optimal solutions are not necessarily vertices of the feasible region; they might not be even be on the border of that region. Rounding (up or down) fractional solutions of the linear program to the closest integer might yield an infeasible solution, or

a solution that is far from optimal — though this trick sometimes leads to reasonably good approximate solutions, as we will see in the next chapter.

Nevertheless, integer and 0-1 linear programming are central problems because many other problems reduce efficiently and naturally to them: Like linear programming, ILP and 0-1 LP is a powerful "language" in which we can express other problems. For this reason there are specialized ILP or 0-1 LP solvers that use heuristics to avoid searching large portions of the feasible region and can find optimal solutions reasonably efficiently for many practical applications, even though they take exponential time in the worst case. We will look at two examples of reductions to 0-1 LP.

6.5.1 Discrete knapsack

Recall the discrete knapsack problem that we previously studied in Section 3.5, where we saw a pseudopolynomial-time algorithm for this problem based on dynamic programming:

Input: The value v_i and weight w_i of each item $i \in [1..n]$, and the capacity C of the knapsack.
Output: A set of items $S \subseteq [1..n]$ that maximizes $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq C$.

Given an instance of this problem we can construct the following 0-1 linear program:

- **Variables:** x_i, for each $i \in [1..n]$. The intention is that $x_i = 1$ means that item i is included in the knapsack and $x_i = 0$ means that item i is not included in the knapsack.
- **Objective function:** Maximize $\sum_{i=1}^{n} x_i v_i$.
- **Constraints:**
  (1) $\sum_{i=1}^{n} x_i w_i \leq C$.
  (2) $x_i \in \{0, 1\}$, for each $i \in [1..n]$.

Note that constraints (2) are *not* linear. It is clear that from an optimal solution $\mathbf{x}^* = (x_i^*\colon i \in [1..n])$ to this 0-1 LP, we can obtain an optimal solution to the knapsack problem $S^* = \{i\colon x_i^* = 1\}$. Constraint (1) ensures the feasibility of S^*, and the fact that we are maximizing the objective function implies that there is no feasible set that has greater value: If such a set existed, we could obtain a better feasible solution to the 0-1 LP.

If we replace each constraint (2) by the pair of constraints

(2a) $x_i \geq 0$
(2b) $x_i \leq 1$

we get a *bona fide* linear program that can be used to solve the *fractional* knapsack problem. Of course, the greedy algorithm for fractional knapsack that we saw in Section 1.3 is much faster than solving this linear program.

6.5.2 The facility location problem

A new city is under construction and the planners must decide where to place the schools. There are certain sites that could be used to that end, and constructing a school at a site has a certain cost (depending on its location). Each school serves a certain district, and the location of the school affects how inconvenient it is for each resident of the district, perhaps measured by the distance of the resident's home from the site chosen for that district's school. The

planners would like to choose the sites where the schools will be built so as to minimize the total cost of constructing schools in the chosen sites *and* the total inconvenience to the city's residents. This is an instance of an optimization problem known as the ***facility location problem***. We can formulate this generally as follows:

- There is a set of n "sites" $i \in [1..n]$, in each of which we can potentially locate a facility (e.g., construct a school). We must select some subset of these sites in which facilities will actually be located.
- There are m "clients" (e.g., households), $j \in [1..m]$. We must assign each of them to exactly one of the sites selected to locate facilities.
- The cost of opening a facility in site i is f_i.
- The cost of assigning client j to site i is c_{ji}.
- The goal is the decide in which sites to open facilities, and to which opened facility to assign each client so as to minimize the sum of two quantities: (a) the total cost of opening facilities in the chosen sites (***facility cost***), and (b) the total cost of client-facility assignment (***service cost***).

We can formulate this as a 0-1 LP as follows.

- **Variables:** x_i, for each $i \in [1..n]$; and y_{ji}, for each $j \in [1..m]$ and $i \in [1..n]$. The intention is that $x_i = 1$ means that site i is selected as the location of a facility and $x_i = 0$ otherwise; and $y_{ji} = 1$ means that client j is assigned to the facility in site i and $y_{ji} = 0$ otherwise.
- **Objective function:** Minimize $\underbrace{\sum_{i=1}^{n} f_i x_i}_{\text{facility cost}} + \underbrace{\sum_{j=1}^{m}\sum_{i=1}^{n} c_{ji} y_{ji}}_{\text{service cost}}$.
- **Constraints:**
  (1) (Client j is assigned to exactly one site)
  $\sum_{i=1}^{n} y_{ji} = 1$, for each $j \in [1..m]$.
  (2) (If client j is assigned to site i then a facility is opened at site i)
  $y_{ji} \leq x_i$, for each $j \in [1..m]$ and $i \in [1..n]$.
  (3a) $x_i \in \{0, 1\}$, for each $i \in [1..n]$; and
  (3b) $y_{ji} \in \{0, 1\}$, for $j \in [1..m]$ and $i \in [1..n]$.

Similar to the 0-1 LP for knapsack, constraints (3a) and (3b) are non-linear. The list of constraints (2) is worthy of special mention. Note how we turned a *Boolean* constraint into a *linear* one that can be a legitimate constraint of a 0-1 linear program. A rookie (but common, and fatal) error in constructing a linear program would be to express this constraint as a logical formula, such as $y_{ji} \rightarrow x_i$ or (equivalently) $\neg y_{ji} \vee x_i$. These capture the intended constraint literally and directly, but they are not legal expressions in the language of linear programming, as they are not linear inequalities. Some Boolean constraints can be expressed as linear constraints and can be incorporated into a 0-1 LP in *that* form.

The facility location problem is an NP-hard problem. No polynomial-time algorithm for it is known, and there are strong reasons to believe that none exists. So translating the problem into a 0-1 LP and feeding the resulting program into a 0-1 LP solver is as good a way we have

for solving this problem.

It is a powerful demonstration of the expressiveness of 0-1 LP that we can solve many variations of the facility location problem by simple and natural adjustments to the list of constraints. For example, we can easily incorporate to the problem requirements such as: the facility cost must not exceed a given budget b, or the number of clients assigned to a facility in site i must not exceed a specified capacity r_i, or the difference in the number of clients assigned to two facilities must not exceed a specified bound k, and so on.

Exercises for Chapter 6

6.1 Consider the linear programs for the maximum flow and the minimum cost flow problems presented in Sections 6.4.1 and 6.4.2. For each of these, state whether there are conditions under which the linear program is (a) infeasible or (b) unbounded — and, if so, what these conditions are. Justify your answer.

6.2 A temporary employment agency has a set of workers $W = \{w_1, w_2, \ldots, w_n\}$. Each worker w_i is qualified for a subset Q_i of a set of skills $S = \{s_1, s_2, \ldots, s_k\}$ (e.g., answering phones, filing, word-processing, Excel, etc). In addition, each worker w_i is available for a certain number h_i of hours per day, where h_i is an integer in the range 0 to 8.

Every day, the agency receives a set of m tasks $T = \{t_1, t_2, \ldots, t_m\}$ to be carried out. Each task t_j requires *exactly one* skill $\ell_j \in S$, and a non-negative integer number of hours r_j.

An ***assignment*** (of workers to tasks) is a set A of triples of the form (w, t, d) where w is a worker, t is a task, and $d > 0$ is a number of hours (not necessarily an integer), subject to the constraints (i)-(iii) listed below. Such a triple indicates that worker w is assigned to task t for d hours, Note that the assignment A may contain several triples with the same first component (when a worker is assigned to multiple tasks), or with the same second component (when several workers are assigned to the same task). The constraints that an assignment must satisfy are:

(i) If a worker w_i is assigned to task t_j, then w_i is qualified for the skill ℓ_j that t_j requires.
(ii) The total number of hours that worker w_i is assigned to tasks does not exceed his or her availability h_i.
(iii) The total number of hours that workers are assigned to task t_j does not exceed the task's time requirement r_j.

Note that an assignment is not required to fill all of a worker's available hours or to provide all the hours required to complete a task.

(a) Consider the following problem:
Input: The set of workers W, the set of tasks T, the set of skills S, the qualifications Q_i and availability h_i of every worker $w_i \in W$, and the skill ℓ_j and time requirement r_j of every task $t_j \in T$.

Output: An assignment A that *maximizes the total number of hours assigned to the workers.*

Describe an algorithm to solve this problem *by reduction to maximum flow.* Justify the correctness of your algorithm, and analyze its running time.

(b) Suppose that for every hour that a worker is assigned to task t_j, the agency makes a profit p_j. The agency wants to find an assignment that ***maximizes its total profit***. Express this optimization problem as a linear program. Explain the meaning of your variables and of each of your constraints. You are not asked to justify anything here; only to provide a linear program and the required explanation.

6.3 A new semester is about to dawn and the Department of Commuter and Metaphysical Sciences (CMS) has to deal once again with the problem of teaching assistant (TA) assignments.

A set S of students have applied for TA positions in the set C of courses offered by CMS next semester. Each course $c \in C$ has a set of tutorials T_c. Let $T = \cup_{c \in C} T_c$ be the set of all tutorials of CMS courses; the sets of tutorials of different courses are disjoint. For each applicant $s \in S$ there is a known set of courses Q_s for which s is qualified to serve as a TA. A ***TA assignment*** A is a set of pairs (s, t), where $s \in S$ and $t \in T$, that satisfies requirements (i)-(iii) below. The pair (s, t) indicates that applicant s has been assigned to tutorial t.

(i) Each tutorial has at most one applicant assigned to it.
(ii) Each applicant $s \in S$ is assigned to at most 3 tutorials, possibly of different courses.
(iii) If an applicant s is assigned to a tutorial of course c, then s is qualified for c (i.e., $c \in Q_s$).

CMS hires you to develop algorithms to find TA assignments that satisfy certain optimality criteria. The input to your algorithms consists of:

- The set of applicants S.
- The set of courses C.
- For each applicant $s \in S$, the set of courses Q_s for which s is qualified; this could be empty.
- For each course $c \in C$, the set of tutorials T_c of course c; this could be empty and the sets of tutorials of different courses are disjoint.

(a) The first algorithm you must develop finds a TA assignment that maximizes the number of tutorials that have a TA assigned to them.

By *reduction to the maximum flow problem* give a polynomial-time algorithm that determines such an assignment. The polynomial depends on $\ell = |S|$, $m = |C|$, and $n = |T|$. Justify the correctness of your algorithm and analyze its running time.

(b) We say that a course c is ***fulfilled*** by a TA assignment A if *every* tutorial of c has a TA assigned to it in A. The second algorithm you must develop finds a TA assignment that *maximizes the number of courses that are fulfilled.*

Formulate this problem as a 0-1 LP, and explain how from an optimal solution to this program (i.e., an assignment of 0 or 1 to its variables that satisfies the constraints and optimizes the objective function) you can obtain the desired TA assignment. Explain the role of your program's variables, and how the constraints capture the above requirements. State the number of variables and constraints of your 0-1 LP; these, and the coefficients

involved in the constraints and the objective function of your 0-1 LP, must be bounded by a polynomial of the input size (the relevant parameters are $\ell = |S|$, $m = |C|$, and $n = |T|$).

6.4 Let $S = \{s_1, \ldots, s_n\}$ be the set of Master's students in the Department of Computer Science (DCS), and $C = \{c_1, \ldots, c_m\}$ be the set of graduate courses offered by DCS. C is partitioned into two sets T_1 and T_2, the set of courses offered in the fall and winter term, respectively.

Each course c_i has an enrolment limit of ℓ_i. Each student s_j chooses a set $C_j \subseteq C$ of courses that s_j is interested in taking. (C_j may be any subset of C, including the empty set!)

DCS must assign each Master's student to a set of courses that the student is interested in taking. A student may be assigned to at most five courses, at most three of which may be in the same term. Furthermore, no course should be assigned more students than its enrolment limit stipulates.

(a) DCS receives from the university \$1,000 for each student in each course. By *reduction to maximum flow*, design an algorithm that the department can use to assign students to courses so as to maximise its revenue. Justify the correctness of your algorithm and analyze its running time.

(b) A student can graduate if he/she takes 5 courses. By efficient reduction to 0-1 LP, design an algorithm that the department can use to assign students to courses so as to maximise the number of students who can graduate? Explain the role of variables and the constraints of your 0-1 LP. Why is your reduction efficient? (Note that we only require the reduction — i.e., the algorithm that transforms an instance of the problem to a 0-1 LP — to be efficient. We don't (presently) know of an efficient and general way to solve 0-1 LPs!)

6.5 Another important NP-hard optimization problem is ***graph colouring***. A "colouring" of an undirected graph $G = (V, E)$ with k "colours" is an assignment of a colour to each node of the graph from a set of k colours such that any two nodes that are connected by an edge are assigned different colours. We want to find a colouring of a graph that uses the fewest colours. More precisely:

Input: An undirected graph $G = (V, E)$.

Output: A function $f\colon V \to [1..k]$, where k is an integer, such that

- **Feasibility:** for all $\{u, v\} \in E$, $f(u) \neq f(v)$; and
- **Optimality:** if $f'\colon V \to [1..k']$ is a function with the property that for all $\{u, v\} \in E$, $f'(u) \neq f'(v)$, then $k' \geq k$.

Give an efficient reduction of the graph colouring problem to 0-1 LP. Explain the role of variables and the constraints of your 0-1 linear program, and why your reduction is efficient.

Chapter 7

Approximation algorithms

7.1 What they are and why we need them

Throughout this course we encountered so-called "NP-hard" problems. At present we don't know how to solve these problems efficiently — by which we mean solving them in polynomial time, in the worst case. There are strong reasons to believe that this is because, in fact, it is impossible to solve these problems efficiently, but we cannot prove that this is the case; this is the famous "P versus NP problem".

Unfortunately, we cannot simply turn our backs to these NP-hard problems and ignore them: many of them are very important in practice. There are various approaches that computer scientists have devised for dealing with them.

- We can look for efficient algorithms that work only in special cases, but ones that suffice for certain applications. For example, we don't know how to solve the vertex cover problem for arbitrary graphs in polynomial time, but we can do so for *bipartite* graphs, as we saw in Section 5.5.
- We can look for *heuristics* that do not achieve polynomial time in the worst case, but find optimal, or near-optimal, solutions in many cases of interest. These are important algorithmic techniques that we have not studied in this course. You can seek more information about them in other sources.
  In passing, and to give a high-level idea of how they work, we mention three important examples of such techniques: *branch-and-bound*, *backtracking*, and *local search* algorithms. Roughly speaking these are clever ways of searching the solution space, which is usually exponential in the size of the input. Branch-and-bound and backtracking algorithms try to avoid detailed examination of certain (hopefully large) parts of the solution space by determining that they do not contain solutions of interest; they give up guaranteed polynomial time but always find an optimal solution (in the case of brach-and bound) or a feasible solution (in the case of backtracking, which is used for feasibility problems).[1]

[1] A *feasibility* (or *constraint satisfaction*) problem seeks to find a feasible solution without regard to optimizing an objective function. For example, finding a path between two nodes is a feasibility problem; finding a *shortest* path between two nodes is an optimization problem.

Local search algorithms typically examine only part of the solution space and construct what is (or appears to be) a "good enough" solution; they stop when they find a local optimum which is not guaranteed to be a global optimum.

- We can look for *approximation algorithms*, i.e., algorithms that run in polynomial time and find a feasible solution that is *guaranteed* to be close to optimal.

By "close to optimal" we mean that the approximation algorithm finds a solution within a factor $c \geq 1$ of the optimal solution. More precisely, consider an optimization problem P and let $OPT(x)$ be the value of an optimal solution of P on input x. Let A be an algorithm that returns a feasible (not necessarily optimal) solution for P, and let $ALG(x)$ be the value of the solution that A finds on input x. We say that A is a ***c*-approximation algorithm** for P if

- P is a minimization problem and, for *any* input x, $ALG(x) \leq c \cdot OPT(x)$; in this case $c \geq 1$.
- P is a maximization problem and, for *any* input x, $ALG(x) \geq c \cdot OPT(x)$; in this case $c \leq 1$.

The quantity c is called an ***approximation ratio*** of algorithm A. It is a maximum "degradation factor": it indicates how worse than optimal is the solution produced by A; the closer to 1, the better.

For some problems we can find efficient c-approximation algorithms for constant c, regardless of the size of the input x. In other cases, the only efficient c-approximation algorithms we know are for a quantity c that depends on the size n of the input; that is, the larger the input, the worse the degradation (with respect of optimal) of the guaranteed quality of the solution found by the algorithm. Of course in this case we would like c to be a "small" function of the input size n such as $\log n$. In yet other cases we can choose $c = 1 + \epsilon$ for minimization problems or $c = 1 - \epsilon$ for maximization problems for *arbitrarily* small $\epsilon > 0$, and achieve a c-approximation but at the cost of a time complexity that, while polynomial, increases as ϵ approaches 0. An approximation algorithm of this last type, which is the "gold standard" for approximation algorithms, is called a ***polynomial-time approximation scheme*** — or ***PTAS*** for short. The word "scheme" refers to the fact that it is not a single algorithm, but a whole collection of algorithms, parameterized by the choice of ϵ.

In this chapter we will see approximation algorithms of all these types for NP-hard problems by leveraging many of the algorithmic techniques we learned in this course.

7.2 Minimum vertex cover

Recall the minimum vertex cover problem: Given an undirected graph, find a minimum cardinality subset R of its nodes that "cover" all its edges — i.e., every edge has at least one endpoint in R.

7.2.1 LP relaxation and rounding

We can express the minimum vertex cover problem for graph $G = (V, E)$ as the following 0-1 linear program P:

- **Variables:** x_u, for each $u \in V$. The intention is to have $x_u = 1$ mean that node u is in the vertex cover and $x_u = 0$ otherwise.
- **Objective function:** Minimize $\sum_{u \in V} x_u$.
- **Constraints:**
  (1) $x_u + x_v \geq 1$, for each $\{u, v\} \in E$ (at least one endpoint of each edge must be in the vertex cover).
  (2) $x_u \in \{0, 1\}$, for each $u \in V$ (the variables take on binary values).

To obtain an approximation algorithm for vertex cover we proceed as follows.

Step 1: Replace the non-linear constraint (2) with the pair of inequality constraints
(2a) $x_u \geq 0$, for each $u \in V$, and
(2b) $x_u \leq 1$, for each $u \in V$.

This way we obtain from the 0-1 LP P an ordinary linear program P'. This step is called ***LP relaxation***, because we relax the stricter 0-1 LP constraint (2) to the weaker constraints (2a) and (2b). We can solve this linear program in polynomial time (e.g., using the interior point method).

Step 2: The intention behind the variables of the 0-1 LP gives meaning to the variables only when their values are 0 or 1. The optimal solution $\mathbf{x}^* = (x_u^*\colon u \in V)$ to the relaxed linear program, however, could very well assign fractional values to some of the variables. We will turn these values into 0s or 1s by rounding: For each $u \in V$, let

$$\hat{x}_u = \begin{cases} 1, & \text{if } x_u^* \geq 1/2 \\ 0, & \text{otherwise.} \end{cases} \tag{7.1}$$

Let $\hat{R} = \{u\colon \hat{x}_u = 1\}$ and R be a minimum vertex cover of G. We claim that (1) $\hat{R}$ is a vertex cover of G and (2) $|\hat{R}| \leq 2|R|$.

Proof of (1): Suppose, for contradiction, that there is some edge $\{u, v\} \in E$ such that $u \notin \hat{R}$ *and* $v \notin \hat{R}$. Thus, $\hat{x}_u = \hat{x}_v = 0$ and therefore $x_u^* < 1/2$ and $x_v^* < 1/2$, and so $x_u^* + x_v^* < 1$, contradicting that $\mathbf{x}^*$ is feasible.

Proof of (2): Let $\overline{\mathbf{x}} = (\overline{x}_u\colon u \in V)$ be an optimal solution to the 0-1 LP P. Thus, a minimum vertex cover R of G has size $\sum_{u \in V} \overline{x}_u$. Since P and P' have the same (minimization) objective function and the constraints of P' are looser than those of P, we have that $\sum_{u \in V} x_u^* \leq \sum_{u \in V} \overline{x}_u$. Finally, by (7.1), for every $u \in V$, $\hat{x}_u \leq 2x_u^*$. Putting all these together, we have

$$|\hat{R}| = \sum_{u \in V} \hat{x}_u \leq 2 \sum_{u \in V} x_u^* \leq 2 \sum_{u \in V} \overline{x}_u = 2|R|$$

as wanted.

In summary we have a polynomial-time, 2-approximation algorithm for vertex cover in general graphs. The algorithm consists of the following steps. Given an undirected graph G:
(1) construct a 0-1 LP P whose solution yields a minimum vertex cover;
(2) relax this 0-1 LP to an ordinary LP P';
(3) solve P';

(4) round the optimal solution to P' as in (7.1); and
(5) extract the vertex cover $\hat{R}$.

If n is the number of nodes and m the number of edges of G, the relaxed LP P' constructed in steps (1) and (2) has n variables and $m + 2n$ constraints, so its size in linear in the size of G and it can be constructed in linear time give G. By using, say, the interior point method to solve P', step (3) can be done in time polynomial in m and n. The remaining two steps are straightforward. Thus, the algorithm described above runs in polynomial time in the size of the input, and produces an answer within (at most) a factor of 2 of optimal, as we proved.

Is our approximation bound of 2 *tight*? That is, are there examples of graphs for which the algorithm could produce a vertex cover that is exactly twice as large as the minimum vertex cover? This is not obvious — it could be that our argument for proving the approximation bound was sloppy and in fact the algorithm performs better. It is not hard to come up with an example that demonstrates that this is not the case.

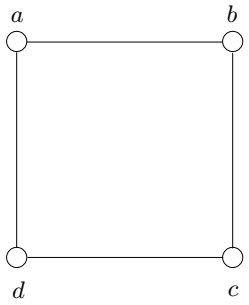


Figure 7.1: A graph with minimum vertex cover of size 2

Consider the graph in Figure 7.1. The associated relaxed LP P' is to minimize $x_a + x_b + x_c + x_d$ subject to the constraints

$$\begin{aligned} x_a + x_b &\geq 1 \\ x_b + x_c &\geq 1 \\ x_c + x_d &\geq 1 \\ x_d + x_a &\geq 1 \\ x_a &\geq 0 \\ x_a &\leq 1 \\ x_b &\geq 0 \\ x_b &\leq 1 \\ x_c &\geq 0 \\ x_c &\leq 1 \\ x_d &\geq 0 \\ x_d &\leq 1 \end{aligned}$$

It turns out that an optimal solution to this LP is $x_a = x_b = x_c = x_d = 1/2$.[2]The rounded solution is $\hat{x}_a = \hat{x}_b = \hat{x}_c = \hat{x}_d = 1$, giving the trivial vertex cover $\{a, b, c, d\}$ consisting of all nodes. This has size twice the size of a minimum cover, i.e., $\{a, c\}$ or $\{b, d\}$.

This result (or better, its generalization to graphs of arbitrarily large size — see Exercise 7.1) shows that the approximation that *this* algorithm achieves cannot be improved; our approximation analysis is as good as the algorithm truly achieves. This raises the question whether a *different* algorithm can achieve a better approximation ratio than 2 in polynomial time. Such algorithms exist, but they are based on more advanced techniques and, frankly, they do not achieve a much better approximation ratio. The best that I am aware of achieves an approximation ratio of $2 - \Theta(1/\sqrt{\log n})$, where n is the number of nodes of the graph. This is a technically masterful achievement, but the improvement in the approximation ratio deteriorates with increasing n, so the algorithm is probably not useful in practice.

There is another type of results about approximation algorithms, known as "inapproximability" or "hardness of approximation" results. These are also beyond the scope of this course, but it is worth knowing what they are about: they seek to establish bounds on how well we can approximate various NP-hard problems. Concerning minimum vertex cover, it is known that not only is the problem NP-hard, but even approximating it within a factor of 1.3606 is NP-hard! Note that a consequence of this result is that, unless the minimum vertex cover problem itself can be solved in polynomial time, there is no PTAS for vertex cover.

There is a variant of the minimum vertex cover problem known as the ***weighted minimum vertex cover problem***. In this we are given as input, in addition to the undirected graph $G = (V, E)$, a node-weight function $\mathbf{wt}$: $V \to \mathbb{Z}$. We are seeking a vertex cover of minimum weight, rather than of minimum cardinality, where the weight of a vertex cover is the sum of the weights of the nodes in it. The (unweighted) minimum vertex cover is the special case of this problem where all nodes have the same weight. We can use essentially the same LP-relaxation and rounding idea to get a polynomial-time 2-approximation algorithm for the weighted vertex cover problem. (See Exercise 7.2.)

7.2.2 A greedy 2-approximation algorithm for minimum vertex cover

We now describe another very simple polynomial-time 2-approximation algorithm for the minimum vertex cover problem. Recall that a matching of an undirected graph G is a subset M of its edges so that no two edges in M share an endpoint. A matching M of G is ***maximal*** (as opposed to maximum) if no superset of M is a matching of G.

Using an obvious greedy algorithm, we can find a maximal matching of G in time linear in the size of G. (What is this algorithm?) The set of nodes that are endpoints of the edges of *any* maximal matching of G is a vertex cover of G and its size is *at most* twice the size of a minimum

[2]To see this, first note that this solution is feasible. Then add the first four constraints to conclude that *any* feasible solution must satisfy that $x_a + x_b + x_c + x_d \geq 2$. Note that the left-hand side is the objective function, so this tells us that the objective function is *at least* 2 for all feasible solutions. Since for the feasible solution at hand, i.e., $x_a = x_b = x_c = x_d = 1/2$, the objective function is equal to 2, it follows that this solution is, in fact, optimal. (This argument exploits an important property of linear programs called *duality*.)

vertex cover of G. (See Exercise 7.3.) This gives us a simple, linear-time 2-approximation algorithm for vertex cover. Unfortunately, unlike the LP-relaxation based algorithm we saw earlier, this does not generalize to the weighted minimum vertex cover problem.

7.3 Minimum set cover

The minimum set cover problem is defined as follows.

Input: A finite set $U = \{a_1, a_2, \ldots, a_n\}$ and a collection $\mathcal{S}$ of subsets of U, whose union is all of U; that is, $\mathcal{S} = \{A_1, A_2, \ldots, A_m\}$, where $A_j \subseteq U$ for each $j \in [1..m]$ and $\cup_{j=1}^{m} A_j = U$. We say that $\mathcal{C} \subseteq \mathcal{S}$ is a ***cover*** (of $\mathcal{S}$) if every element of U is in a set that $\mathcal{C}$ contains; that is, $\cup_{A \in \mathcal{C}} A = U$.

Output: A minimum cardinality cover of $\mathcal{S}$.

The minimum set cover problem is a generalization of the minimum vertex cover problem and is therefore also NP-hard (see Exercise 7.4).

A natural greedy algorithm to find a "small" set cover is to start with selecting a set in $\mathcal{S}$ that contains as many elements as possible. Now some elements of U are covered and some remain uncovered. We continue by selecting a set in $\mathcal{S}$ that contains as many uncovered elements as possible, and we continue in the same manner until no element of U remains uncovered. Shown below is this algorithm in pseudocode.

```
GreedySetCover(U, S)
1 C := ∅      ▶ C is the partial set cover so far
2 R := U      ▶ R is the set of elements of U that remain uncovered by C
3 while R ≠ ∅ do
4     let A ∈ S be such that |A ∩ R| is maximum
5     C := C ∪ {A}
6     R := R − A
7 return C
```

It is clear that this is a polynomial-time algorithm: With just elementary data structures, we can implement it in $O(mn^2)$ time (the while loop is executed at most n times, and finding the set with the most uncovered elements can be done in $O(mn)$ time). It is also clear that it returns a cover $\mathcal{C}$ of $\mathcal{S}$. The remaining question is how good an approximation $\mathcal{C}$ is relative to the minimum set cover $\mathcal{C}^*$. We will show that $|\mathcal{C}| \leq \ln n \cdot |\mathcal{C}^*|$, so the algorithm achieves an approximation ratio of $\ln n$. In a nutshell, the reason is that the number of uncovered elements decreases exponentially with each iteration, so after at most $\ln n \cdot |\mathcal{C}^*|$ of iterations (i.e., after adding at most $\ln n \cdot |\mathcal{C}^*|$ sets of $\mathcal{S}$ to $\mathcal{C}$), the algorithm terminates. We will now make this argument more precise.

As usual we subscript a program variable with index i to indicate the value of the variable at the end of iteration i, with "iteration 0" denoting the moment just before the loop starts. Let $\ell = |\mathcal{C}^*|$ be the number of sets in a minimum set cover. Consider the i-th iteration of the loop, $i > 0$, in which the algorithm adds a set A_i to the partial cover $\mathcal{C}_{i-1}$ constructed at the end of the previous iteration, and shrinks the set of uncovered elements from R_{i-1} to $R_i = R_{i-1} - A_i$.

We claim that $|R_i| \leq |R_{i-1}|(1 - 1/\ell)$. To see why, note that the elements in R_{i-1} must be covered by the minimum set cover $\mathcal{C}^*$. Therefore $\mathcal{C}^*$ contains a set A that covers at least $|R_{i-1}|/\ell$ of the elements of R_{i-1}: otherwise, all ℓ sets of $\mathcal{C}^*$ together would not cover all the elements of R_{i-1}, contradicting that $\mathcal{C}^*$ is a cover — let alone a minimum one. That set A is available to be chosen in iteration i (since it contains elements in R_{i-1}), and therefore the set A_i that the algorithm adds to $\mathcal{C}$ in iteration i covers at least as many uncovered elements as A does; that is, A_i covers at least $|R_{i-1}|/\ell$ elements of R_{i-1}. So, $|R_i| \leq |R_{i-1}| - |R_{i-1}|/\ell = |R_{i-1}|(1 - 1/\ell)$. Applying this repeatedly and recalling that $|R_0| = n$ (or, more formally, using induction) we conclude that

$$|R_i| \leq n(1 - 1/\ell)^i \tag{7.2}$$

We will now use the inequality $1 - x \leq e^{-x}$, for all $x \in \mathbb{R}$, where the equality holds only for $x = 1$. (The easiest and most memorable way to see why this is true is to look at the graphs of the two functions $1 - x$ and e^{-x} — see Figure 7.2. For a more rigorous proof, consider the difference of the two functions, $D(x) = e^{-x} - (1 - x)$. Taking the first and second derivatives of $D(x)$, we conclude that its minimum value is 0 and is obtained only when $x = 1$.)

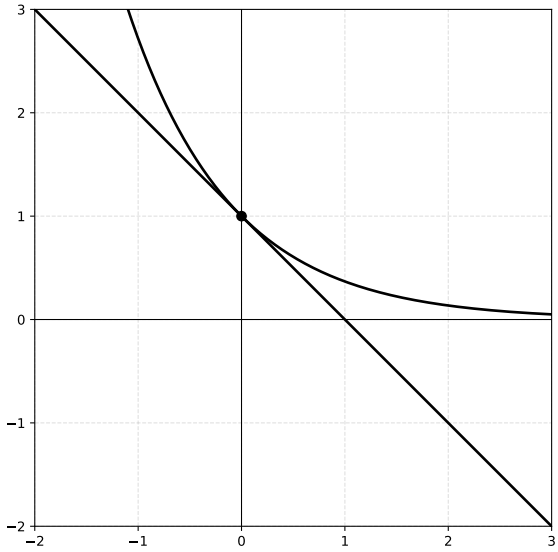


Figure 7.2: Graphs of $1 - x$ and e^{-x}

If we assume that $\mathcal{S}$ does not contain a set that is the entire U (so that the minimum set cover has at least two sets), $\ell > 1$, so $1/\ell \neq 1$ and $(1 - 1/\ell) < e^{-1/\ell}$. Applying this inequality to (7.2), we conclude that $|R_i| < ne^{-i/\ell}$. So, if the loop has not terminated before $i = \ell \ln n$ iterations, $R_i < ne^{-\ell \ln n/\ell} = 1$; thus, after iteration $\ell \ln n$, all elements are covered. So, the loop terminates after at most $\ell \ln n$ iterations, which means that the set cover returned by the algorithm has size $|\mathcal{C}| \leq \ln n \cdot |\mathcal{C}^*|$, as wanted.

So we have a $\ln n$-approximation algorithm for the minimum set cover problem. This is not nearly as good as what we achieved for minimum vertex cover, where the approximation ratio

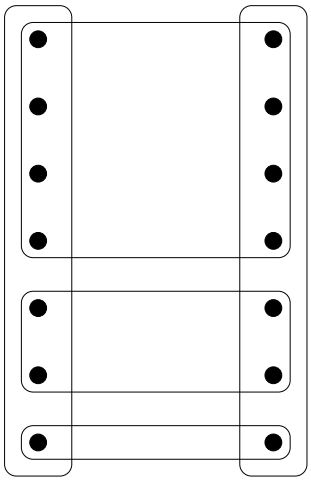


Figure 7.3: Example illustrating the approximation ratio of the greedy set cover algorithm

was 2, constant regardless of the size of the input. With the greedy approximation algorithm for minimum set cover, our approximation ratio gets worse as the size of the input increases. So, one question is whether the ratio we obtained is actually tight: Could it be that our algorithm achieves a better ratio in the worst case but our analysis was too crude to prove it? As the example illustrated in Figure 7.3, our analysis is tight, within a constant *factor*.

What we see in that figure is an instance of the set cover with $n = 14 = 2^4 - 2$ elements and a collection of five sets. The greedy algorithm would pick a set cover consisting of the three "middle" sets (of sizes 8, 4, and 2) while the optimal set cover consists of only the two "side" sets (each of size 7). You can easily generalize this to obtain, for any positive integer $k \geq 3$, an instance with $n = 2^k - 2$ elements, $k - 1$ "middle" sets, and 2 "side" sets, where the minimum set cover has two sets, each of size $n/2 = 2^{k-1} - 1$; while the set cover returned by the greedy set cover algorithm has $k - 1 = \log_2 n - 1$ sets of size $2^{k-1}, 2^{k-2}, \ldots, 2^1$. Thus, the approximation ratio for this instance is $(\log_2 n - 1)/2 = \Theta(\ln n)$.

Incidentally, we could have used the same idea for a greedy approximation algorithm for vertex cover: Build a vertex cover greedily, always choosing the vertex that "covers" the most still-uncovered edges. This would give us an algorithm with approximation ratio $\ln n$, worse than the ratio 2 we achieved with the algorithms described in Section 7.2 — see Exercise 7.5.

Is there a better approximation algorithm for set cover that runs in polynomial time? Not likely: It has been shown that the problem of finding a set cover with approximation ratio $\big(1 - o(1)\big) \ln n$ is NP-hard.[3]

As with vertex cover, there is a weighted version of the minimum set cover problem. We are now given the set of elements U, the collection $\mathcal{S}$ of subsets of U, and a weight function $\mathbf{wt}\colon \mathcal{S} \to \mathbb{R}$. We seek a cover of minimum weight, where the weight of a cover is the sum of the weights of the sets in it. There is a greedy algorithm for this problem that generalizes the

[3]A function $f(n)$ is $o(1)$ if $\lim_{n\to\infty} f(n) = 0$; for example, $1/n = o(1)$.

one we just analyzed: We now extend the partial set cover $\mathcal{C}$ constructed so far with the set A in $\mathcal{S}$ that maximizes the ratio $|R \cap A|/\mathbf{wt}(A)$, where R is the set of elements still-uncovered by the current partial set cover $\mathcal{C}$. This achieves a similar approximation ratio, but the analysis is now much subtler and beyond the scope of our course.

7.4 Minimum makespan scheduling

Consider the following setting: There are m "machines", labeled $1, 2, \ldots, m$, and n "jobs", labeled $1, 2, \ldots, n$, that must be performed by these machines. Each job j has a specified "length" (or "processing time") t_j, which is the amount of time that j requires on any machine. Each job j can be performed by any machine, and each machine performs one job at a time.

An ***assignment*** A specifies which jobs each machines performs: for each $i \in 1..m$, $A(i)$ is the set of jobs to be performed by machine i; we require that $\cup_{i=1}^{m} A(i) = \{1, 2, \ldots, n\}$ (every job is assigned to some machine) and $A(i) \cap A(i') = \varnothing$ if $i \neq i'$ (no job is assigned to two machines). Given an assignment A, the ***load*** L_i of machine i is the sum of the lengths of the jobs assigned to i: $L_i = \sum_{j \in A(i)} t_j$. The ***makespan*** L of assignment A is the maximum load of any machine: $L = \max_{1 \leq i \leq m} L_i$; thus, L is the earliest time by which all machines have finished processing all the jobs assigned to them. We want to assign jobs to machines so as to minimize the makespan. So, the ***minimum makespan problem*** can be specified as follows:

Input: $m \in \mathbb{Z}^+$, $n \in \mathbb{Z}^+$, $t_1, \ldots, t_n \in \mathbb{Z}^+$.
Output: An assignment A with minimum makespan.

This is an NP-hard problem, so to solve it it makes sense to look for a polynomial-time approximation algorithm.

7.4.1 A greedy algorithm

The first algorithm uses a natural greedy approach. We consider the jobs in *any* order and assign each job to the *currently least loaded machine*:

```
GreedyAssignment(m, n, t_1, ..., t_n)
1 for i := 1 to m do A(i) = ∅
2 for each job j in any order do
3     let i be a least loaded machine (ties broken arbitrarily)
4     A(i) := A(i) ∪ {j}
5 return A
```

The running time of this algorithm is clearly polynomial; in fact we can implement it in $O(n \log m)$ time by keeping the loads of the machines in a heap. It is easy to construct examples where this algorithm does not produce an optimal assignment. For example, suppose $m = 2$, $n = 3$, and $t_1 = t_2 = 1$ and $t_3 = 2$. If the greedy algorithm considers the three jobs in the order 1, 2, 3, it produces the assignment $A(1) = \{1, 3\}$ and $A(2) = \{2\}$, which has makespan 3. The assignment $A'(1) = \{1, 2\}$ and $A'(2) = \{3\}$ has makespan 2, and is clearly optimal.

We will now show that although not optimal, the greedy algorithm has a constant approximation ratio. Consider any input and let

- L^g be the makespan of the assignment produced by the greedy algorithm for that input, and
- L^* be the minimum makespan for that input.

Theorem 7.1 $L^g \leq (2 - \frac{1}{m})L^*$.

PROOF. We start with three simple observations:

(1) *For each job j, $t_j \leq L^*$.*
This is because any assignment, and in particular an optimal assignment, must assign job j to some machine.

(2) $\sum_{j=1}^{n} t_j \leq mL^*$.
This is because in any assignment, and in particular in an optimal assignment, the average load $\big(\sum_{j=1}^{n} t_j\big)/m$ cannot exceed the maximum load L^*.

(3) *Just before the greedy algorithm assigns job j, the load of the least loaded machine at that time (which is the machine to which j is assigned) is at most $L^* - t_j/m$.*
This is because, just before the greedy algorithm assigns job j, the total load on all machines is $\sum_{k=1}^{j-1} t_k$ and therefore the minimum load on any machine so far is at most $\big(\sum_{k=1}^{j-1} t_k\big)/m$ (the minimum load cannot exceed the average load). But $\sum_{k=1}^{j-1} t_k \leq \big(\sum_{k=1}^{n} t_k\big) - t_j \leq mL^* - t_j$ (by Observation (2)), and so the minimum load on any machine is at most $(mL^* - t_j)/m = L^* - t_j/m$.

Now consider a machine that ends up being most loaded at the end of the greedy algorithm, and let ℓ be the last job assigned to that machine. We will call this the ***critical*** job — intuitively, it is one that "causes" the maximum load. The makespan of the assignment produced by the greedy algorithm is the load of that machine after ℓ is assigned to it. So we have

$$\begin{aligned} L^g &\leq \big(L^* - \frac{1}{m}t_\ell\big) + t_\ell && \text{[by Observation (3)]} \\ &= L^* + t_\ell\big(1 - \frac{1}{m}\big) \\ &\leq L^* + L^*\big(1 - \frac{1}{m}\big) && \text{[by Observation (1)]} \\ &= L^*\big(2 - \frac{1}{m}\big). \end{aligned}$$

□

The bound of Theorem 7.1 is tight, in the sense that there are inputs for which $L^g = (2 - \frac{1}{m})L^*$. We have already seen an example where this is the case when $m = 2$. We can generalize this for any number of machines m. Suppose there are $m(m-1)$ jobs each of length 1, and one job of length m, which is considered last by the greedy algorithm. Then the greedy algorithm produces an assignment with makespan $2m-1$ but there is an assignment (which is

optimal) that assigns m short jobs to each of $m-1$ machines and the long job to the remaining machine, resulting in a makespan of m. So in this case $L^g/L^* = (2m-1)/m = 2 - \frac{1}{m}$.

7.4.2 The longest-first greedy algorithm

The example proving that the approximation rate of Theorem 7.1 is tight shows exactly what can go wrong with the greedy algorithm: If the long jobs arrive at the end, we may have already filled all the machines with short jobs, forcing a sudden large increase in the makespan. This observation suggests the following refinement of the greedy algorithm: Instead of considering the jobs in any order, we will consider them in order of *decreasing* length. This way we deal with the long jobs first, putting them in different machines as long as possible, and then fill-in the machines with shorter jobs. This adds to the algorithm a preliminary step to sort the jobs, which takes $O(n \log n)$ time, so the overall running time is $O(n \log n + n \log m)$.

This refinement of the greedy algorithm is known as the ***longest-first***, or ***longest-processing-time-first (LPTF)***, algorithm. As we will see shortly (Theorem 7.2), it achieves a better approximation ratio than the greedy algorithm, but it is still not optimal — not surprisingly, since the minimum makespan problem is NP-hard! For example, if $m = 3$, $n = 7$, and $t_1 = t_2 = 5$, $t_3 = t_4 = 4$, and $t_5 = t_6 = t_7 = 3$, the longest-first algorithm will construct the assignment $A(1) = \{1, 6, 7\}$, $A(2) = \{2, 5\}$, and $A(3) = \{3, 4\}$ with makespan 11. A better assignment — in fact, an optimal one — is $A'(1) = \{1, 3\}$, $A'(2) = \{2, 4\}$, and $A'(3) = \{5, 6, 7\}$, with makespan 9.

Consider any input and let

- A^{lf} be the assignment produced by the longest-first algorithm for that input, and L^{lf} be its makespan, and
- A^* be an optimal assignment and L^* be its makespan for that input.

Theorem 7.2 $L^{lf} \leq (\frac{4}{3} - \frac{1}{3m})L^*$.

Proof. Note that Observations (1)-(3) hold, since the longest-first algorithm is a special case of the greedy algorithm. Without loss of generality assume that the jobs are indexed in non-increasing length, so $t_1 \geq t_2 \geq \cdots \geq t_n$.

We will also assume that the critical job (the one that "causes" the maximum load), i.e., the job we called ℓ in the proof of the previous theorem, is actually the last (and shortest) job; that is, we will assume that $\ell = n$. The reason we can make this assumption without loss of generality is that ignoring jobs after the critical one does not change the makespan of the assignment produced by the longest-first algorithm, and it does not increase the makespan of an optimal assignment (since there are fewer jobs to assign); so, the assumption does not artificially reduce the approximation ratio of the longest-first algorithm.

There are two cases.

Case 1. $t_n \leq L^*/3$. Recalling that the critical job is n and by reasoning as in the previous

proof we have:

$$\begin{aligned} L^{\text{lf}} &\le \big(L^* - \frac{1}{m}t_n\big) + t_n && \text{[by Observation (3)]} \\ &= L^* + t_n\big(1 - \frac{1}{m}\big) \\ &\le L^* + \frac{L^*}{3}\big(1 - \frac{1}{m}\big) && \text{[by hypothesis of Case 1]} \\ &= L^*\Big(\frac{4}{3} - \frac{1}{3m}\Big), \text{ as wanted.} \end{aligned}$$

CASE 2. $t_n > L^*/3$. In this case we will prove that the longest-first algorithm is actually optimal! This is obvious if $n \le m$ (there are no more jobs than machines), so we will also assume that $n > m$. The essence of the argument will be to show that there is an optimal assignment that looks exactly like the assignment produced by the longest-first algorithm (up to renaming the machines).

First consider an optimal assignment A^* of the given input. We will argue that we can assume, without loss of generality, that A^* has certain properties. Since the shortest job has length more than $L^*/3$, no machine is assigned more than two jobs, so $n \le 2m$. Since $n > m$, in A^* at least one machine is assigned two jobs; furthermore, we can assume that every machine is assigned at least one job (if a machine is assigned no jobs, we can move to it a job from a machine that is assigned two jobs with the resulting assignment remaining optimal, and we can keep doing this until every machine is assigned a job). We will call a machine with only one job a ***singleton*** and a machine with two jobs a ***double***. Let $i1$ be the longer of the up to two jobs assigned to machine $i \in [1..m]$, and $i2$ be the shorter one, if machine i is a double (ties resolved arbitrarily). Without loss of generality (by renaming the machines, if needed) we can assume that $t_{11} \ge t_{21} \ge \cdots \ge t_{m1}$. As a result we can also assume that if machine i is a double, so are all machines with a higher index: if not, we can move the second job of machine i to one of the singleton machines with higher index *without increasing the makespan of the assignment*. Finally, we can assume that the lengths of the *shorter* jobs in the double machines are non-*decreasing* with the index; that is, if $i \in [1..m]$ is the minimum index of a double machine, $t_{i2} \le t_{i+1,2} \le \cdots \le t_{m2}$. (Before proceeding with the explanation why we can make this assumption, consult Figure 7.4, which might help you visualize what we claim we can assume about the lengths of the jobs in some optimal assignment.)

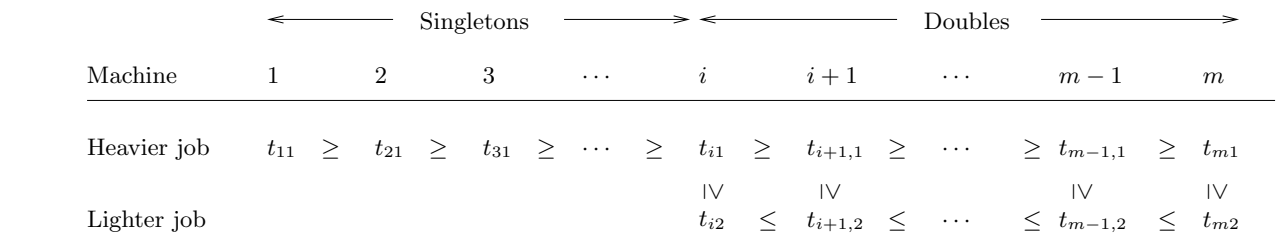


Figure 7.4: The lengths of the jobs in some optimal assignment when $t_n > L^*/3$

We justify this assumption by using a standard exchange argument: Let us call a violation of the assumption by a pair of machines an ***inversion***; if there is an inversion, there must be one between two consecutively-indexed machines, say k and $k+1$, where $k \in [i..m-1]$. That is, we have $t_{k1} \geq t_{k+1,1}$ but $t_{k2} > t_{k+1,2}$ (instead of the wanted $t_{k2} \leq t_{k+1,2}$), where we know that $t_{k1} \geq t_{k2}$, $t_{k+1,1} \geq t_{k+1,2}$, and that $t_{k1} + t_{k2} \leq L^*$ and $t_{k+1,1} + t_{k+1,2} \leq L^*$. You can readily check that if we swap jobs $k2$ and $k+1, 2$ (placing the former in machine $k+1$ and the latter in machine k), we obtain an optimal assignment, since the sums of the loads on these two machines remain at most L^* (do it!). Thus we can obtain an optimal assignment with fewer inversions and so, by induction, an optimal assignment with no inversions, as wanted.

Now we consider the assignment A^{lf} produced by the longest-first algorithm on the given input. Our goal is to show that it is the same as A^*. As in A^*, we can assume that if some machine is not a singleton, than neither is any higher-indexed machine. To see why, suppose that in A^{lf} machine i is a not singleton but machine $i' > i$ is. At the time A^{lf} assigned the second job to machine i, the load on machine i' was no greater than on machine i, so A^{lf} should (or at least could, if the load was the same on both machines) assign the second job to machine i', rather than to machine i.

For the first m jobs A^{lf} makes the same assignment as A^*, placing job $j \in [1..m]$ on machine j so that the load of the machines is non-increasing with their index. For job $m+1$ job, the algorithm continues like A^* by assigning it to the least loaded machine, which is machine m, and continues by assigning every remaining job to the currently least loaded machine. The only potential difference from A^* is that, by following this rule, A^{lf} might assign *three* jobs to some machine, because the total load of the first two jobs is the minimum one, even less than the load of a machine that has only one job assigned to it. (Note that we don't know yet that A^{lf} is optimal, and therefore it is conceivable that its makespan is more than L^*.) We will show that this cannot happen.

To this end suppose, for contradiction, that A^{lf} assigns three jobs, say p, q, and r in that order, to the same machine, say i. First we claim that in A^{lf} there is a singleton machine that is a double in A^*. To see this, let s^* be the maximum index of a singleton machine in A^* and s be the maximum index of a singleton machine in A^{lf} (either of these could be 0, if no machine is singleton). By considering A^* we see that $n = s^* + 2(m - s^*) = 2m - s^*$. And since A^{lf} assigns three jobs to at least one machine, $n > s + 2(m - s) = 2m - s$. Therefore $2m - s^* > 2m - s$ and so $s > s^*$. This means that machine s is a singleton in A^{lf} but a double in A^*. Let j be the job assigned to machine s by A^{lf}. So, j is one of the first m jobs and, as we have seen, it is assigned to machine s by A^* as well. Let j' be the other job assigned to machine s by A^*. At the time that A^{lf} assigned the third job r to machine i, j was already assigned to machine s and therefore $t_j \geq t_p + t_q$ (otherwise, A^{lf} would have assigned job r to machine s, rather than to i). So, $t_j > 2L^*/3$ (since $t_p \geq t_q \geq t_n > L^*/3$). Since A^* assigns j and j' to machine s, the load of s in A^* is $t_j + t_{j'} \geq t_j + t_n > 2L^*/3 + L^*/3 = L^*$, contradicting that L^* is the makespan of A^*.

So, we have shown that A^{lf} is the same assignment as A^*, which means that the longest-first algorithm is optimal in Case 2, as we claimed.

In both cases the approximation ratio is at most $\frac{4}{3} - \frac{1}{3m}$, as wanted. □

The bound of Theorem 7.2 is tight. The example we saw earlier demonstrates this for the case $m = 3$: In that example, the optimal assignment has makespan 9, while the makespan of the assignment produced by the longest-first algorithm is $11 = \left(\frac{4}{3} - \frac{1}{3 \cdot 3}\right) \cdot 9$. This example can be generalized for any odd number of machines m as follows. The input consists of the following jobs, listed in decreasing length:

- 2 jobs of length $2m - 1$
- 2 jobs of length $2m - 2$
- 2 jobs of length $2m - 3$
- $\vdots$
- 2 jobs of length $m + 1$
- 3 jobs of length m.

On this input, the longest-first algorithm produces an assignment with makespan $4m-1$ while an optimal assignment has makespan $3m$ (one machine has the three jobs of length m and every other machine has two jobs, one of length $2m - k$ and one of length $m + k$, for each $k \in [1..m - 1]$), yielding an approximation ratio of $\frac{4m-1}{3m} = \frac{4}{3} - \frac{1}{3m}$ for this input.

* * * * *

Although the longest-first algorithm has the advantage of achieving a better approximation, the first greedy algorithm has the advantage of being an ***online*** algorithm. Online algorithms do not receive all their input at the start, but rather receive the input as a stream over the course of the computation. There are many settings in which we need online algorithms:

- A multiprocessor operating system must assign tasks to processors as the tasks arrive, without knowing in advance all the tasks to be scheduled.
- An operating system must decide which cache line to replace when the cache is full and a memory reference is made to a location that is not presently cached, without knowing in advance the future memory references.
- A trading algorithm must make decisions about which stocks to buy and sell, without knowing in advance the future prices of stocks.
- An algorithm that manages a dictionary that is being searched repeatedly and is stored as a list or a tree should keep the elements being sought frequently near the beginning of the list or the root of the tree, without knowing in advance the sequence of searches.

The gold standard for an online algorithm is how it performs when compared to the best that is achievable by an ***offline*** algorithm, i.e., an algorithm that has access to the entire input in advance. The ratio between the value of the solution obtained by an online algorithm and the value of the optimal solution that can be obtained by any offline algorithm is called the online algorithm's ***competitive ratio***. We want this to be as close to 1 as possible.

Theorem 7.1 shows that the competitive ratio of the greedy algorithm for makespan has competitive ratio $2 - \frac{1}{m}$. Historically, the analysis of online algorithms started with the study of the makespan problem by Ronald Graham in his seminal paper "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal on Applied Mathematics*, 17(2):416429 (March 1969). Theorems 7.1 and 7.2 are from this paper.

7.5 The k-centre problem

Consider the following scenario. In a rural district that contains several villages the regional authority must decide where to locate k fire stations that will serve all the villages; the number k is dictated by budget constraints and is typically much smaller than the number of villages in the district. There is plenty of room, so these fire stations can be located anywhere in the district, not necessarily inside villages. If we fix the locations of the k fire stations, the distance from each village to its *closest* fire station is a measure of the "inconvenience" experienced by that village as a result of the chosen locations. We want to choose the locations so as to *minimize the maximum inconvenience* faced by the villages.

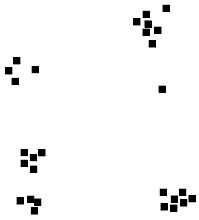
This is illustrated in Figure 7.5. Figure 7.5a shows the location of the villages (shown as black squares) on a map. Figure 7.5b shows the villages and an optimal location of the fire station if we could only have one ($k = 1$). The circle shown is the smallest circle that contains all villages; its centre is the location of the fire station and its radius is the maximum distance from a village to the fire station. The most inconvenienced village lies on the circumference of this circle. Figure 7.5c shows the villages and optimal locations for two fire stations ($k = 2$). The two circles shown have radius equal to the maximum distance of any village from its *closest* fire station. If we draw circles with that radius centred at each fire station, *all* villages are within one (or more) of these circles, and the most inconvenienced village is on the circumference of these cicles. Similarly, Figures 7.5d, e, and f show optimal locations of the fire stations if $k = 3$, 4, and 5, respectively.

This is an example of an optimization problem known as the ***k-centre problem***. We will define this more precisely shortly, but the general idea is that we have a set of "sites" (villages) located in some space, and we must choose the locations of k "centres" (fire stations) in that space, with the objective of minimizing the maximum distance between sites and their closest centre, according to some measure of "distance" between locations. In addition to operations research, this type of problem arises in statistics and machine learning where the sites are data points in a multidimensional space and we want to group these data points into some number k of clusters, where similar data points are grouped into the same cluster. In this case, the measure of distance is a measure of similarity: the "closer" two data points are, the more similar they are.

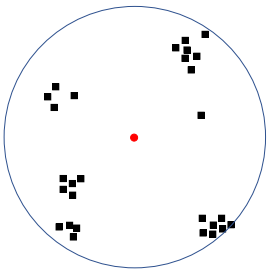
Let us now formulate the problem in more general terms. We will need some terminology. A ***metric space*** is a pair (M, d), where M is a set of points and d: $M \times M \to \mathbb{R}$ is a distance function that satisfies the following three properties:

- (non-negativity) for all $x, y \in M$, $d(x, y) \geq 0$, and $d(x, y) = 0$ if and only if $x = y$;
- (symmetry) for all $x, y \in M$, $d(x, y) = d(y, x)$; and
- (triangle inequality) for all $x, y, z \in M$, $d(x, z) \leq d(x, y) + d(y, z)$.

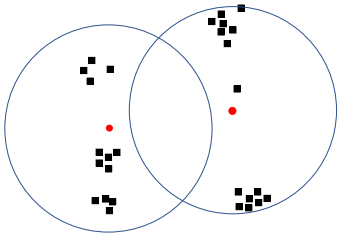
As an example of a metric space, M could be $\mathbb{R}^k$, and the distance function could be the usual Euclidean distance in k-dimensional real space: If $x = (x_1, \ldots, x_k)$ and $y = (y_1, \ldots, y_k)$, then $d(x, y) = \sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$. Other distance functions when $M = \mathbb{R}^k$ include the ***Manhattan distance*** $d(x, y) = \sum_{i=1}^{k} |x_i - y_i|$ and the ***Chebyshev distance*** $d(x, y) = \max_{i=1}^{k} |x_i - y_i|$. As another example of a metric space, M could be the set of all strings of length k over some



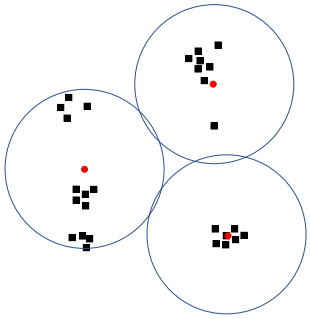
(a) Map of villages (black squares)



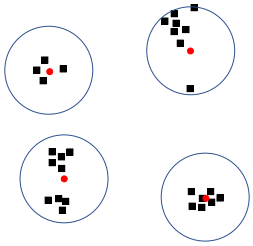
(b) Optimal location for one fire station (red dot)



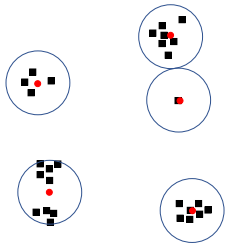
(c) Optimal locations for $k = 2$ fire stations



(d) Optimal locations for $k = 3$ fire stations



(e) Optimal locations for $k = 4$ fire stations



(f) Optimal locations for $k = 5$ fire stations

Figure 7.5: A map of the villages in a region, with optimal locations for k fire stations, $k \in [1..5]$

alphabet, and the distance function could be the ***Hamming distance*** defined as $d(x, y) =$ the number of positions in which x and y differ. You can check that all these ways of defining "distance" satisfy the three properties listed above.

In the k-centre problem we are given a metric space (M, d), and a set of ***sites*** $S \subseteq M$. A set of ***centres*** is a set $C \subseteq M$. For any site $s \in S$ and set of centres C, we define $\delta(s, C)$ to be the distance between s and the closest centre $c \in C$; i.e., $\delta(s, C) = \min_{c \in C} d(s, c)$. The ***radius*** $r(C)$ of the set of centres C is the maximum distance of a site in S from its closest centre; i.e., $r(C) = \max_{s \in S} \delta(s, C)$. As we noted before, this is the *smallest* radius so that if we draw a circle (or, in more dimensions, a sphere or hypersphere) of that radius around each centre in C, then every site is inside one (or more) of these circles. We can now specify the k-centre problem as follows:

Input: A metric space (M, d), a set of n sites $S \subseteq M$, and $k \in \mathbb{Z}^+$.
Output: A set $C \subseteq M$ such that $|C| = k$ and $r(C)$ is minimum.

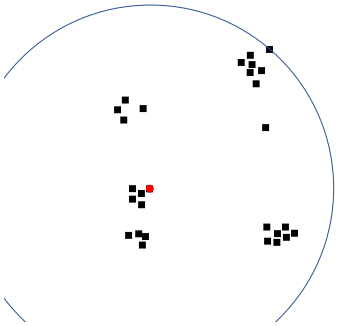
This is an NP-hard problem, so it makes sense to look for a polynomial-time approximation algorithm. Following is a simple greedy 2-approximation algorithm for the k-centre problem. The algorithm starts by selecting *arbitrarily* some site as the first centre. It then repeatedly adds to the set of centres a site that is most inconvenienced by the present set of centres (i.e., a site that is as far as any from its closest centre), ties resolved arbitrarily, until it has chosen k centres. Figure 7.6 shows a possible sequence of sites selected by this algorithm for the example in Figure 7.5.
Following is the algorithm described in pseudocode.

```
    GreedyCentre((M, d), S, k)
 1  s := any element of S
 2  C := {s}
 3  while |C| < k do
 4      s := element of S such that δ(s, C) is maximum (ties resolved arbitrarily)
 5      C := C ∪ {s}
 6  return C
```

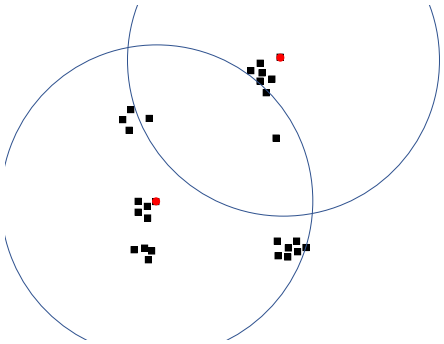
It is clear that the algorithm runs in polynomial time: We can maintain the value of $\delta(s', C)$ for every $s' \in S$ in an array indexed on S. Then in line 4 we can find an element $s \in S$ such that $\delta(s, C)$ is maximum in $O(n)$ time. Under the reasonable assumption that computing $d(p, p')$ for all $p, p' \in M$ can be done in $O(1)$ time, we can update $\delta(s, C)$ to reflect the addition of a new site in C in line 5 also in $O(n)$ time. So, each iteration of the loop takes $O(n)$ time and the entire algorithm runs in $O(nk)$ time.

Note that this algorithm returns a set of k centres that are sites, i.e., elements of S. Recall that this is not required by the problem, which allows the k centres to be elements of M, not necessarily elements of S. (In terms of our example application of the k-centre problem, this algorithm selects locations for the fire stations that are villages.) Despite this additional constraint, we will show that this algorithm returns a set of centres whose radius is no greater than twice the radius of an optimal set of centres (which may contain points in M that are not in S).

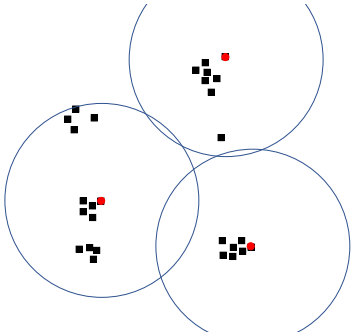
As usual, a variable name subscripted with index i indicates the value of that variable at



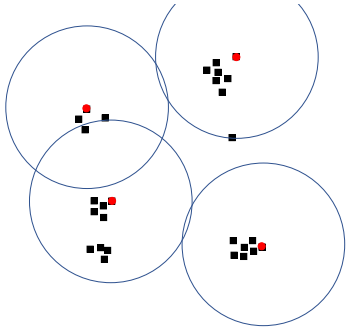
(a) Centre 1: any site



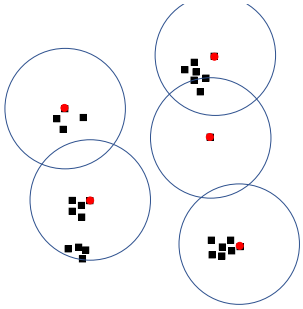
(b) Centre 2: most inconvenienced by centre 1



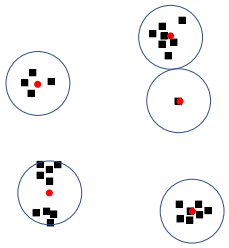
(c) Centre 3: Most inconvenienced by centres 1-2



(d) Centre 4: most inconvenienced by centres 1-3



(e) Centre 5: most inconvenienced by centres 1-4



(f) Compare to optimal locations for five centres

Figure 7.6: Five centres chosen by the greedy algorithm for the example in Figure 7.5

the end of the i-th iteration of the loop, where we take "iteration 0" to be the point before the loop starts (in our case, just after line 2). Thus, $|C_i| = i + 1$, the while loop terminates after $k - 1$ iterations, and the algorithm returns C_{k-1} as its set of k-centres. Let C^* be an optimal set of k centres, i.e., one that minimizes $r(C^*)$.

Theorem 7.3 $r(C_{k-1}) \leq 2r(C^*)$.

PROOF. We first prove the following:

Claim 7.3.1 *For all $i \geq 1$ and all $s, s' \in C_i$ such that $s \neq s'$, $r(C_{i-1}) \leq d(s, s')$; i.e., the radius of C_{i-1} is at most the distance between any two centres in C_i.*

PROOF OF CLAIM 7.3.1. In each iteration i of the loop, the site s_i that the algorithm adds to C is a site that is most disadvantaged by the centres in C_{i-1}; the distance of s_i from its closest centre in C_{i-1} is the radius $r(C_{i-1})$. Thus, $r(C_{i-1}) = \delta(s_i, C_{i-1})$. It remains to prove that, for any $s, s' \in C_i$, $\delta(s_i, C_{i-1}) \leq d(s, s')$. There are two cases.

CASE 1. One of s, s' is s_i. Without loss of generality, assume $s = s_i$; then $s' \in C_{i-1}$. By definition, $\delta(s_i, C_{i-1}) = \min_{s'' \in C_{i-1}} d(s_i, s'') \leq d(s_i, s') = d(s, s')$.

CASE 2. Both $s, s' \in C_{i-1}$. Then s and s' are added to C in two different iterations j and j', both before iteration i. Without loss of generality, suppose $j' < j < i$. We have:

$$\begin{aligned}
\delta(s_i, C_{i-1}) &\leq \delta(s_i, C_{j-1}) && [\text{since } j < i \text{ and so } C_{j-1} \subseteq C_{i-1}] \\
&\leq \delta(s_j, C_{j-1}) && [\text{otherwise } s_j \text{ would not have been added to } C \text{ in iteration } j] \\
&\leq d(s_j, s_{j'}) && [\text{since } j' < j \text{ and } s_{j'} \in C_{j-1}] \\
&= d(s, s').
\end{aligned}$$

□

Imagine now that we run the algorithm for one more iteration, namely iteration k, at the end of which it it has computed a set $C_k = \{s_1, \ldots, s_{k+1}\}$ of $k + 1$ centres, all of which are sites. Let C^* be an optimal set of k-centres. Thus, every site is within a distance of $r(C^*)$ of one of the centres of C^*. In particular each of the $k + 1$ centres of C_k is within a distance of $r(C^*)$ of the k centres of C^*. By the pigeonhole principle then, there are two sites in $s, s' \in C_k$, $s \neq s'$, that are within a distance of $r(C^*)$ of the *same* centre $c^* \in C^*$. By Claim 7.3.1,

$$\begin{aligned}
r(C_{k-1}) &\leq d(s, s') \\
&\leq d(s, c^*) + d(c^*, s') && [\text{by the triangle inequality}] \\
&\leq r(C^*) + r(C^*) && [\text{since } s, s' \text{ are within } r(C^*) \text{ of centre } c^* \in C^*] \\
&= 2r(C^*), \text{ as wanted.}
\end{aligned}$$

□

The approximation ratio 2 for this algorithm is tight — see Exercise 7.6. And there is little hope for a polynomial-time algorithm that achieves a better approximation ratio for this problem: It has been shown that the problem of approximating the k-centre problem within *any* ratio less than 2 is NP-hard.

7.6 Discrete knapsack

Recall that in the discrete knapsack problem we are given the value v_i and weight w_i of each item $i \in [1..n]$, and a capacity C; and we want to find a subset of $[1..n]$ that has maximum value and weight at most C. We will assume that all the numbers involved (values, weights, and the capacity) are positive integers.

In this section we will see a polynomial-time approximation *scheme* (PTAS) for the discrete knapsack problem: An algorithm with an approximation parameter ϵ, for *any* real $\epsilon > 0$ no matter how small, that finds a subset of the items whose total weight does not exceed the capacity C and whose value is *at least* $(1 - \epsilon)$ the optimal value. So, if we want to be guaranted that the set of items returned by the algorithm has value at least 99% of optimal, we can run this algorithm with $\epsilon = 0.01$. Of course, there is a price to be paid for achieving close approximation: the running time of the algorithm is $O(n^3/\epsilon)$. So, the better the desired approximation (i.e., the smaller the ϵ), the slower the algorithm: If we want 99% approximation, the algorithm runs 100 times more slowly.

In Section 3.5 we saw a pseudopolynomial-time dynamic programming algorithm for this problem. The subproblems that this algorithm computed were, for every $i \in [0..n]$ and every $w \in [0..C]$:

$$K(i, w) = \textit{maximum value} \text{ of a subset of items } [1..i] \text{ whose weight is } \textit{at most } w.$$

There is a similar dynamic programming algorithm that computes subproblems based on the minimum weight of a set that is needed to achieve a certain value (rather than the maximum value of a set that can be achieved with a certain capacity). So, our subproblems now are:

$$L(i, v) = \textit{minimum weight} \text{ of a subset of items } [1..i] \text{ whose value is } \textit{at least } v.$$

for all $i \in [0..n]$ and $v \in [0..\sum_{j=1}^{i} v_j]$.

The algorithm computes these subproblems $L(i, v)$, using dynamic programming and returns the maximum value v such that $L(n, v) \leq C$ — i.e., the maximum value achievable using items $[1..n]$ without exceeding the capacity. Here is this algorithm, in pseudocode. It starts by computing $V[i]$, $i \in [0..n]$, to be the sum of the values of items $1..i$.

```
KNAPSACK2(v_1, ..., v_n; w_1, ..., w_n; C)
 1 V[0] := 0
 2 for i := 1 to n do V[i] := V[i − 1] + v_i
 3 for i := 0 to n do L(i, 0) := 0
 4 for i := 1 to n do
 5     for v := 1 to V[i] do
 6         if v > V[i − 1] then L(i, v) := L(i − 1, max(0, v − v_i)) + w_i
 7         else L(i, v) := min(L(i − 1, v), L(i − 1, max(0, v − v_i)) + w_i)
 8 return max{v: L(n, v) ≤ C}
```

Using standard dynamic programming tricks you can modify the algorithm so that it returns a feasible set of items that achieve the maximum value. Since the discrete knapsack problem

is NP-hard, it is not surprising that this algorithm is also only pseudopolynomial: its running time is $O(n \sum_{i=1}^{n} v_i)$.

The high-level idea of the approximation algorithm we will describe is simple. Suppose our input consisted of three items with values 86,302,117, 105,735,763, and 25,201,101. The optimal knapsack would not be very different if we thought of these as 86, 106, and 25 — i.e., eliminating six orders of magnitude of the least significant digits (and rounding up or down). But doing so dramatically reduces (by six orders of magnitude) the number of iterations executed to find that set. Of course there might be a small difference arising from rounding errors during the scaling down of the values, but with proper scaling the error would be small and the approximation achieved good. We will explain shortly how to do "proper scaling". The algorithm consists of the following steps:

(1) Discard all items with weight $w_i > C$ — such items cannot be part of a feasible set. So we will assume, without loss of generality, that $w_i \leq C$ for all $i \in [1..n]$.
(2) Let $v_{\max} = \max\{v_i \colon i \in [1..n]\}$.
(3) Obtain new, scaled values for each item, $\widehat{v}_i = \left\lfloor \frac{v_i}{v_{\max}} \cdot \sigma \right\rfloor$. The first factor $v_i/v_{\max}$ scales the value down to the range $[0, 1]$. The second factor σ rescales the value up; for now we will treat it as an unknown quantity, whose value we will determine through our analysis to meet our approximation goal.
(4) Run the above algorithm KNAPSACK2 using the *scaled* values $\widehat{v}_i$ (and the original weights w_i and capacity C).
(5) Return the set $\widehat{S}$ of items that the algorithm finds for the scaled values.

Let's see now what is the value of σ we need to use to achieve our goal. Let

- $\widehat{S}$ be an optimal set for scaled values $\widehat{v}_i$ (what our algorithm computes), and
- S^* be an optimal set for the original values v_i (what we wish to compute).

We want

$$\sum_{i \in \widehat{S}} v_i \geq (1 - \epsilon) \cdot \sum_{i \in S^*} v_i. \tag{7.3}$$

The sum on the left-hand side is the value of the set returned by our algorithm and the sum on the right-hand side is the value of the optimal set. We have

$$\begin{aligned}
\sum_{i \in \widehat{S}} v_i &\geq \sum_{i \in \widehat{S}} \widehat{v}_i \cdot \frac{v_{\max}}{\sigma} && \left[\text{because } \widehat{v}_i = \left\lfloor \frac{v_i}{v_{\max}} \cdot \sigma \right\rfloor \leq \frac{v_i}{v_{\max}} \cdot \sigma\right] \\
&= \frac{v_{\max}}{\sigma} \sum_{i \in \widehat{S}} \widehat{v}_i \\
&\geq \frac{v_{\max}}{\sigma} \sum_{i \in S^*} \widehat{v}_i && [\text{because } \widehat{S} \text{ is optimal for the scaled values}] \\
&\geq \frac{v_{\max}}{\sigma} \sum_{i \in S^*} \left\lfloor \frac{v_i}{v_{\max}} \cdot \sigma \right\rfloor && [\text{by definition of } \widehat{v}_i] \\
&\geq \frac{v_{\max}}{\sigma} \sum_{i \in S^*} \left(\frac{v_i}{v_{\max}} \cdot \sigma - 1 \right) && [\text{since } \lfloor x \rfloor \geq x - 1, \text{ for any } x]
\end{aligned}$$

$$\begin{aligned}
&\geq \sum_{i \in S^*} v_i - \frac{v_{\max}}{\sigma} \cdot \sum_{i \in S^*} 1 \\
&\geq \sum_{i \in S^*} v_i - \frac{n}{\sigma} \cdot \sum_{i \in S^*} v_i && \text{[because (a) } \textstyle\sum_{i \in S^*} 1 \leq n \text{ and (b) } v_{\max} \leq \sum_{i \in S^*} v_i \text{ (since} \\
& && \text{all items of weight } > C \text{ were removed in step (1))]} \\
&= \Big(1 - \frac{n}{\sigma}\Big) \sum_{i \in S^*} v_i.
\end{aligned}$$

Therefore, if we take $\frac{n}{\sigma} = \epsilon$ we have the desired result (7.3). This means that the rescaling factor σ that gives us the desired approximation is $\sigma = \frac{n}{\epsilon}$.

Regarding the running time of the algorithm sketched above in steps (1)-(5), it is clear that the dominating step is (4). This takes time $O(n \sum_{i=1}^{n} \widehat{v}_i) = O\big(n \sum_{i=1}^{n} \big\lfloor \frac{v_i}{v_{\max}} \cdot \frac{n}{\epsilon} \big\rfloor\big) = O(n^3/\epsilon)$ (since $v_i/v_{\max} \leq 1$).

A noteworthy feature of this PTAS is that the dependence of the running time on ϵ^{-1} is (only) polynomial — in this case actually only linear. There are polynomial-time approximation schemes for other problems, where the dependence is much worse. For instance the running time could be of the form $O(n^3 \cdot 2^{-\epsilon})$; in this case the degradation of the running time needed to achieve a better approximation is much more severe. PTAS like the one we described for knapsack, where the dependence of the running time on ϵ^{-1} is polynomial are called ***fully polynomial-time approximation schemes*** (abbreviated ***FPTAS***). Problems with known FPTAS solutions are few and far between.

7.7 Maximum cut

A ***cut*** of an undirected graph $G = (V, E)$ is a pair (X, Y), where X and Y form a partition of the set of nodes; i.e., they are nonempty, disjoint sets whose union is V. The set $\mathit{cross}(X, Y)$ of ***cross edges*** (with respect to a cut (X, Y)) is the set of edges with one endpoint in X and one endpoint in Y; i.e., $\mathit{cross}(X, Y) = \big\{\{x, y\} \in E\colon x \in X \text{ and } y \in Y\big\}$. The ***maximum cut problem*** is specified as follows:
Input: An undirected graph $G = (V, E)$.
Output: A cut (X, Y) that maximizes $|\mathit{cross}(X, Y)|$.

Maximum cut is an NP-hard problem. In this section we will discuss a simple $\frac{1}{2}$-approximation algorithm for this problem, based on an algorithmic paradigm for solving optimization problems called ***local search***. Local search algorithms have the following structure:

```
1 start with some feasible solution S
2 while there is a better feasible solution S' "near" S do S := S'
3 return S
```

Here "better feasible solution" means a feasible solution that improves the objective function (decreases it, if we are minimizing; or increases it, if we are maximizing). The meaning of "S' is near S" depends on the problem we are trying to solve.

In our case, of maximum cut, we will take the cuts "near" (X, Y) to be those resulting by moving a single node from X to Y or from Y to X; i.e., the cuts (X', Y') such that $Y' = Y - \{u\}$

for some $u \in Y$, or $X' = X - \{u\}$ for some $u \in X$. Note that moving a node u from one side of the cut to the other increases the number of cross edges if and only if u has more neighbours in its side of the cut than in the opposite side. So our local-search approximation algorithm for maximum cut is as follows:

```
    ApproxMaxCut(G)
1   (X, Y) := arbitrary cut of G
2   while there is a node u with more neighbours in its side of (X, Y) than the other side do
3       (X, Y) := cut resulting from (X, Y) by moving u to the other side of the cut
4   return (X, Y)
```

Let n be the number of nodes and m be the number of edges of G. Since the number of cross edges of the current cut increases in each iteration, the loop ends after at most m iterations. With straightforward data structures each iteration takes $O(n+m)$ time, so the algorithm runs in $O\big(m(n+m)\big)$ time. It is not hard to see that it does not necessarily find a maximum cut: it can be stuck in a *local* maximum, where moving a single node from one side of the cut to the other does not increase the number of cross edges, but that cut is not a *global* maximum. Fortunately, as the next theorem shows, there is a lower bound on how far off the global maximum the local maximum found by the algorithm can be: The algorithm is, in fact, a $\frac{1}{2}$-approximation algorithm for maximum cut.

Theorem 7.4 *Let (X, Y) be the cut returned by the algorithm for graph G, and (X^*, Y^*) be the maximum cut of G. Then $|\mathit{cross}(X, Y)| \geq \frac{1}{2} \cdot |\mathit{cross}(X^*, Y^*)|$.*

Proof. For each node u of $G = (V, E)$, let n_u be the number of its neighbours; i.e., $n_u = |\{v \in V\colon \{u, v\} \in E\}|$. Let n_u^s be the number of u's neighbours that are in the same side of the cut (X, Y) as u, and n_u^o be the other neighbours of u, so $n_u = n_u^s + n_u^o$. By the termination condition of the algorithm, $n_u^o \geq n_u^s$, for every node u. Therefore, $n_u \leq 2n_u^o$ for every node u. Summing over all nodes,

$$\sum_{u \in V} n_u \leq 2 \cdot \sum_{u \in V} n_u^o \tag{7.4}$$

Also,

$$2 \cdot |\mathit{cross}(X, Y)| = \sum_{u \in V} n_u^o \tag{7.5}$$

since the sum on the right-hand side counts every cross edge twice — once from each endpoint. For the same reason,

$$\sum_{u \in V} n_u = 2 \cdot |E| \tag{7.6}$$

(This fact is known by the colourful name "the Handshake Lemma" — with each edge imagined as a "handshake" between its two endpoints.) Obviously, the number of cross edges in any cut, and in particular in the maximum cut (X^*, Y^*), cannot exceed the total number of edges in the graph! Using this and (7.4)-(7.6), we have

$$|\mathit{cross}(X^*, Y^*)| \leq |E| = \frac{1}{2} \cdot \sum_{u \in V} n_u \leq \frac{1}{2} \cdot \Big(2 \cdot \sum_{u \in V} n_u^o\Big) = 2 \cdot |\mathit{cross}(X, Y)| \qquad \square$$

The approximation ratio 1/2 for this algorithm is tight (see Exercise 7.8). There are polynomial-time approximation algorithms for maximum cut that achieve a better approximation ratio. The best I am aware of achieves an approximation ratio of 0.87 — but it is based on techniques beyond the scope of this course. There is also an inapproximability result for this problem: It has been shown that it is NP-hard to approximate maximum cut with an approximation ratio better than $16/17 \approx 0.94$.

Not surprisingly, there is also a weighted version of the maximum cut problem where, in addition to the undirected graph $G = (V, E)$, we are given an edge-weight function $\mathbf{wt}$: $E \to \mathbb{Z}$, and we seek a cut (X, Y) so that the set of cross edges $cross(X, Y)$ has maximum *weight* (as opposed to maximum cardinality). As usual, the weight of a set of edges is the sum of the weights of the edges in the set. This problem can also be approximated by a variant of the ApproxMaxCut algorithm discussed in this section (see Exercise 7.9).

Exercises for Chapter 7

7.1 In arguing that the approximation ratio 2 for the LP-relaxation-based approximation algorithm for minimum vertex cover, we presented one example of a graph in which the algorithm produces a vertex cover that is twice the size of the minimum. This is not entirely convincing: It is conceivable that the case $n = 4$ is a special one and that, for larger values of n, the approximation ratio of the algorithm is actually better. It turns out that this is not the case. Show that there are arbitrarily large n for which the algorithm returns a vertex cover twice the size of the minimum vertex cover.

7.2 Apply the ideas behind LP-relaxation and rounding to obtain a polynomial-time 2-approximation algorithm for the *weighted* minimum vertex cover problem.

7.3 Prove that the simple greedy approximation algorithm for minimum vertex cover sketched in Section 7.2.2 achieves an approximation ratio of 2. Furthermore, prove that this approximation bound is tight for this algorithm.

7.4 In what sense is the set cover problem a generalization of vertex cover? Or, put it in reverse: What special case of set cover is the vertex cover problem; i.e., what restrictions on an instance of set cover actually makes it an instance of vertex cover?

Use your answer to describe an efficient *reduction* of the vertex cover problem to the set cover problem. That is, an algorithm that solves the vertex cover problem in polynomial time using a "black box" subroutine that can instantaneously solve the set cover problem.

7.5 Prove that the following greedy algorithm for finding a vertex cover has approximation ratio $\ln n$: Start with the empty partial vertex cover R, and in each iteration of a loop add to

R the node that covers the largest number of not-yet-covered nodes; repeat until all nodes are covered. Prove that this ratio is tight within a constant factor *for this algorithm*.

7.6 Prove that the approximation ratio of 2 for the greedy algorithm GreedyCentre that finds a set of k centres for a given set of sites is tight. That is, show that for arbitrarily large k there is a set of n points and distances between them so that the algorithm returns a set of k centres with radius d but the radius of an optimal set of k centres is $d/2$.

7.7 Recall the optimal greedy algorithm for *fractional* knapsack in Section 1.3: Sort the items in non-increasing value/weight ratio, and take whole items until the next item doesn't fit, at which point, take the fraction of that item that fits and halt.

(a) Consider the same algorithm adapted for the *discrete* knapsack problem: Delete all items that don't fit. Then sort the remaining items in non-increasing value/weight ratio and take whole items until the next item doesn't fit, at which point halt.
Prove that this is *not* a c-approximation algorithm for *any* constant c.

(b) Consider now the following variation of the algorithm in part (a): First run the algorithm in part (a); let S be the set of items it returns. Then let i^* be the item of maximum value (after the items of weight more than the knapsack's capacity have been deleted). If $\sum_{i \in S} v_i \geq v_{i^*}$, return S; else return $\{i^*\}$.
Prove that this is a 1/2-approximation algorithm for the discrete knapsack problem.
Hint: Note that, for *any* input, the value of the optimal solution for fractional knapsack for that input is at least as large as the value of the optimal solution for discrete knapsack for the same input. (Why?)
(Of course for discrete knapsack we can do much better than this algorithm using the FPTAS we saw in Section 7.6 to achieve *any* approximation ratio we desire.)

7.8 Prove that the approximation ratio of 2 for the greedy algorithm GreedyCentre that finds a maximum cut is tight. That is, show that for arbitrarily large k there is a graph and a choice for the initial cut in line 1 so that the algorithm returns a cut with k cross edges but the maximum cut has $2k$ cross edges.

7.9 Describe a variation of the maximum cut 2-approximation algorithm that solves the *weighted* maximum cut problem. What is the approximation ratio achieved by your algorithm? What is its runnng time? Is it a polynomial-time algorithm?

7.10 We are given the sequence of "weights" of n items, $w_1, w_2, \ldots, w_n$, where $0 < w_i \leq 1$ is the weight of item i, $1 \leq i \leq n$. We want to place these n items into the smallest possible number of identical containers, each of capacity 1. That is, the sum of the weights of the items placed on any container must not exceed 1. For example, suppose the weights are $0.5, 0.25, 0.75, 0.5, 0.75$. Then the minimum number of containers in which to place these items is three.

Finding the minimum number of containers is a well-known NP-hard problem.

(a) Reduce this problem to 0-1 linear programming.

(b) Consider the following greedy algorithm for this problem: Suppose the containers are numbered $1, 2, \ldots$; these are initially empty. Consider the items in the order given. When considering item i, place it in the first container that (together with the items previously placed in it) can accommodate the item without exceeding its capacity; that is, place i in container j, where j is the smallest positive integer such that the weight of the items presently in container j plus w_i is at most 1. Applied to the example above, this heuristic places items 1 and 2 in the first container, and each of the remaining three items in a separate container — i.e., it uses four containers instead of the optimal three.
Prove that this is a 2-approximation algorithm.

Note: Two is not a tight bound for this approximation algorithm: It can be shown that the algorithm always does better than twice the optimal, but the proofs of better bounds are more complicated.

7.11 Consider the following problem.

INPUT: A set $A = \{a_1, \ldots, a_n\}$ of positive integers, and a positive integer b.
OUTPUT: A subset X of A such that the sum of the elements in X is maximum but does not exceed b.

This is an NP-hard problem.

(a) Reduce this problem to 0-1 LP.

(b) Consider the following greedy algorithm.

```
GreedySum(A, b)
    X := ∅
    total := 0
    for i := 1..n do
        if total + a_i ≤ b then
            X := X ∪ {a_i}
            total := total + a_i
    return X
```

Prove that this is *not* a c-approximation algorithm for the problem, for any constant $c > 1$. That is, for every $c > 1$, there is an input for which the algorithm outputs X, the optimal output is Y, and the sum of the elements in X is less than $1/c$ of the sum of the elements in Y.

(c) Give a $\frac{1}{2}$-approximation greedy algorithm for the problem. Prove that your algorithm is, indeed, a $\frac{1}{2}$-approximation algorithm for the problem, and analyze its running time.

Appendix A

Brief review of complex numbers

We review briefly complex numbers to provide the relevant background that students need to follow the material on the fast Fourier transform (Section 2.7). Typically students in this course have previously encountered this material in their linear algebra courses.

- ***The "imaginary unit"*** $i = \sqrt{-1}$***.*** This is a quantity which, multiplied by itself, yields -1; i.e., $i^2 = -1$. It is called "imaginary" because no real number has this property.
- ***Standard representation of complex numbers.*** A complex number has the form $z = a+bi$, where a and b are real numbers; a is the "real part" of z and b is the "imaginary part" of z. We can think of z as the pair (a, b), and so complex numbers can be mapped to the Cartesian plane. Just as we can geometrically view the set of real numbers as the set of points along a straight line (the "real line"), we can view complex numbers as the set of points on a plane (the "complex plane"). So in a sense complex numbers are a way of "packaging" a pair of real numbers into a single quantity.
- ***Equality between complex numbers.*** We define two complex numbers to be equal to each other if and only if they have the same real parts and the same imaginary parts. In other words, if viewed as points on the plane, the two numbers coincide.
- ***Operations on complex numbers.*** We can perform addition and multiplication on complex numbers. Let $z = a + bi$ and $z' = a' + b'i$.
  - *Addition:* The sum of z and z' is, by definition, $z + z' = (a + a') + (b + b')i$. This can be "justified" algebraically simply by noting that $z + z' = (a + bi) + (a' + b'i) = (a + a') + (b + b')i$. (The word "justified" is in quotes above, because here we are abusing notation horribly, but very usefully. We use the symbol + in three different ways: to separate the real and imaginary part of the complex numbers, to represent the addition operation on real numbers, and to represent the addition operation on complex numbers that we are defining!) The geometric intuition for this operation is clear if we view z and z' as points on the plain; or, equivalently, as two-dimensional position vectors. Then the sum of the two complex numbers is the sum of the two vectors.

  - *Multiplication:* The product of z and z' is, by definition, the complex number $z \cdot z' = (aa' - bb') + (ab' + a'b)i$. This can also be "justified" algebraically (through the same abuse of notation): $z \cdot z' = (a + bi) \cdot (a' + b'i) = aa' + ab'i + a'bi + bb'(i^2) = (aa' - bb') + (ab' + a'b)i$. There is no obvious geometric interpretation of this operation in this form, but we will see a nice interpretation below (see Euler's formula).

- ***The field of complex numbers.*** Addition and multiplication of complex numbers defined in the above manner behave like addition and multiplication of real numbers in the sense that they satisfy the following properties (where z, z', z'' are complex numbers):

  - *Commutativity of addition and multiplication:* $z + z' = z' + z$, and $z \cdot z' = z' \cdot z$.
  - *Associativity of addition and multiplication:* $z + (z' + z'') = (z + z') + z''$, and $z \cdot (z' \cdot z'') = (z \cdot z') \cdot z''$.
  - *Additive and multiplicative identities:* There is a "zero" complex number, namely $0+0i$, which added to any complex number z yields z; and a "one" complex number, $1 + 0i$, which multiplied by any complex number z yields z. (Verify this!)
  - *Additive inverse:* For each complex number z there is a complex number z', called the ***additive inverse*** of z (and denoted $-z$), such that $z + z'$ is the complex "zero". The additive inverse of $a + bi$ is $(-a) + (-b)i$. (Verify this!)
  - *Multiplicative inverse:* For each complex number z *other than* the complex number "zero" $(0 + 0i)$, there is a complex number z', called the ***multiplicative inverse*** of z, and denoted z^{-1}, such that $z \cdot z'$ is the complex "one". The multiplicative inverse of $a + bi$ (when at least one of a, b is non-zero) is the complex number $a/(a^2 + b^2) + \big(-b/(a^2 + b^2)\big)i$. (Verify this!)

  Any algebraic structure, i.e., any set of objects with two operators + ("plus") and $\cdot$ ("times"), that satisfy the above properties is called a ***field***. The rational numbers, the real numbers, and the complex numbers are examples of fields. (Is the set of natural numbers with ordinary addition and multiplication a field? Is the set of all integers with ordinary addition and multiplication a field?) In general, we can define a vector space — and therefore do linear algebra — over any field, not just the field of real numbers. So we can do linear algebra over the field of complex numbers: We can define matrices of complex numbers and add or multiply such matrices as we do matrices of reals. We can define vectors of complex numbers, spaces of such vectors, linear transformations between such spaces, and so on.

- ***Polar coordinate representation of complex numbers:*** Consider the complex number $z = a + bi$ as the point (a, b) in two-dimensional space. Recall that a point in two-dimensional space can be fixed either by its Cartesian coordinates (in our case the pair (a, b)), or by its ***polar coordinates*** (r, θ), where r is the distance of (a, b) from the origin $(0, 0)$, i.e., $r = \sqrt{a^2 + b^2}$; and θ is the angle between the horizontal axis and the line segment that starts at the origin and ends at point (a, b). (The distance r from the origin

determines a circle centered at the origin with radius r, and the angle θ determines the point on that circle where (a, b) lies.) By its definition the angle θ satisfies the property that $\cos\theta = a/r$ and $\sin\theta = b/r$. Thus, the complex number $z = a + bi$ can also be written as $z = r(\cos\theta + i\sin\theta)$; this is sometimes called the ***polar form*** of z. So, the complex number z can be viewed as the pair (a, b) (in Cartesian coordinates), or as the pair (r, θ) (in polar coordinates).

- ***Euler's formula:*** Euler's formula states that for any real number θ, $e^{i\theta} = \cos\theta + i\sin\theta$, where the arguments to sin and cos are expressed in radians. It describes a fundamental relationship between the basic trigonometric functions and the exponential function. (We will take the formula for granted, without proof. It can be proved by considering the Maclaurin expansions of $e^{i\theta}$, $\cos\theta$, and $\sin\theta$. Formally speaking, $e^{i\theta}$ — an exponential to a complex number! — is *defined* as the Maclaurin expansion.)

  Plugging $\theta = \pi$ into Euler's formula we get $e^{i\pi} = \cos\pi + i\sin\pi = -1 + i0 = -1$, i.e. $e^{i\pi} + 1 = 0$. (Pause here to marvel at the awesomeness of this: The four most ubiquitous constants in mathematics (0, 1, π, e) all tied up in one amazing equality.)

  As we have seen, the complex number $z = a + bi$, in polar form, is $z = r(\cos\theta + i\sin\theta)$; by Euler's formula, $z = re^{i\theta}$. This representation is called the ***exponential form*** for complex numbers and it is extremely useful. (Recall that $r = \sqrt{a^2 + b^2}$; this quantity is called the ***magnitude*** of z. The angle θ is such that $\cos\theta = a/r$ and $\sin\theta = b/r$; this quantity is called the ***argument*** of z or, if you are an electrical engineer, the ***phase*** of z.)

  The exponential form gives a useful geometric interpretation for multiplication of complex numbers. Let $z = re^{i\theta}$ and $z' = r'e^{i\theta'}$ be two complex numbers. Then $z \cdot z' = (re^{i\theta}) \cdot (r'e^{i\theta'}) = rr'e^{i(\theta+\theta')}$. In other words, to multiply two complex numbers in exponential form, we multiply their magnitudes and add their angles!

- ***The unit circle:*** Consider the complex numbers on the complex plane. The ***unit circle*** consists of all complex numbers with magnitude 1, i.e., the numbers that are at distance 1 from the origin. Obviously these are precisely the points along the circle whose centre is the origin and whose radius is 1. That is, these are the complex numbers of the form $e^{i\theta}$, where $0 \leq \theta < 2\pi$.

  Consider any two such numbers, $z = e^{i\theta}$ and $z' = e^{i\theta'}$. Their product is $z \cdot z' = e^{i(\theta+\theta')}$, which is another point on the unit circle. For any $n \in \mathbb{N}$, the nth power of z is $z^n = (e^{i\theta})^n = e^{in\theta}$ — i.e., the point on the unit circle whose angle is n times the angle of z.

- ***The n-th roots of unity:*** The complex n-th roots of unity are the solutions to the equation $z^n = 1$ over the complex numbers. (More precisely, the solutions to the equation $z^n = 1 + 0i$.) Note that the number 1, as a complex number, has magnitude 1 and angle that is any integer multiple of 2π, i.e., is of the form $2\pi k$, where k is an integer; all these are simply the angle 0. Thus, if $z = re^{i\theta}$ is a complex root of unity, we must have $z^n = r^n e^{in\theta} = e^{i2\pi k}$, for some integer k. Thus, it must be the case that $r = 1$ and

$n\theta = 2\pi k$, i.e., $\theta = (2\pi/n)k$, for some integer k. Note that there are exactly n distinct angles θ in the range $0 \leq \theta < 2\pi$ that satisfy $\theta = (2\pi/n)k$, for some integer k, namely those obtained by setting $k = 0, 1, \ldots n-1$. (Setting k to other integer values yields one of these n angles, when reduced to the range between 0 and 2π.) Therefore, for each n, there are exactly n distinct complex n-th roots of unity. They all lie on the unit circle and they go around the circle, starting from the point $(1, 0)$ (the complex number 1 that has magntitude 1 and angle 0), and proceeding counterclockwise in angle increments of $2\pi/n$.