Algorithms: An Intuitive Approach

Gopal Pandurangan
Department of Computer Science
University of Houston

January 19, 2024

With figures by Khalid Hourani

To
Sumithra, Iniya, and Sathyan

Contents

Appendices

List of Figures

LIST OF ALGORITHMS

1 INTRODUCTION

This book is for a course on algorithms. It focuses on the *fundamentals of algorithms* and their *design* and *mathematical analysis*. Many *real-world* applications give rise to algorithmic *problems*. In fact, the modern computer revolution was and continues to be fueled by algorithms. Thus, we will emphasize the *real-world significance of algorithms*.

One can design many *algorithms* for *computing* a solution to a given problem. Various important questions arise:

- How do we *design* a good algorithm?
- What do we mean by a *good* algorithm?
- How do we *analyze* an algorithm?
- How do we know that an algorithm is *good* or *efficient*?
- How do we *efficiently implement* an algorithm (say, using suitable *data structures*) in a high-level programming language such as C++, Java, or Python?

Algorithms are everywhere. They occur and play a key role in many areas not limited to computer science alone, but also to other sciences (such as biology, economics, physics, and chemistry), and engineering (such as electrical and electronics engineering and mechanical engineering). Listed below are some applications, predominantly from a computer science perspective.

- Scientific Computation, Numerical Analysis
- Computational Biology, Bioinformatics
- Communication Networks
- Computer Systems, Operating Systems, Programming Languages and Compilers
- Typesetting and Text Editors
- Internet/Web Applications
- e-Commerce, Cryptography

- Games
- Optimization
- Data Processing: “Big Data” Analysis
- Computer Graphics, Animation
- Signal processing, Communications Technology
- Machine Learning and Artificial Intelligence

In fact, it is hard to think of an area in science and engineering in which algorithms do not play a major role. Thus, it is appropriate to say that algorithms (and computer science) have come to play a central and enabling role in other areas of science and engineering as well.

1.1 What this book teaches

This book teaches how:

- To solve various problems that arise in a wide range of applications using *algorithmic* thinking and techniques.
- To learn and *apply* various *algorithmic design techniques.*
- To *design* and *analyze* **efficient** algorithms for a variety of problems. The emphasis is on *algorithmic thinking* to solve problems and not just *programming.*
- To understand, appreciate, and explore how algorithms are applied in the real world.

The book endeavors to give *intuition* on the main ideas behind algorithms and their analyses. While we learn algorithms by studying specific problems, the book emphasizes on technique and patterns in algorithm design and analysis that could be applicable in general.

1.2 Topics

The book covers the following topics (not necessarily in this order). The goal is to expose the reader to a variety of algorithmic problems that are related through one or more of the following themes: problem domain, algorithmic technique, and analysis technique.

- Algorithm design and algorithmic thinking (especially recursion).
- Mathematics of algorithm analysis.
- Algorithms for numbers.
- Recursion, Divide and Conquer.
- Randomized algorithms and techniques

- Algorithms for searching, sorting, and various other problems.
- Hashing and its applications.
- Data Structures.
- Greedy method.
- Dynamic programming.
- Graph algorithms.
- Matrix algorithms.
- Approximation Algorithms.

1.3 Some of the problems that we will solve (and their real-world applications)

Throughout the course, we will study algorithms for fundamental problems that also play an important role in real-world applications. Some of the problems that we will study are:

- *Prime number checking* — a basic tool in cryptosystems
- *Searching a large database* — How does a university retrieve a record quickly given a student ID? How does Google retrieve webpages given a search keyword?
- *Signal processing* — How to compute a Fourier transform — which is at the heart of many applications — quickly?
- *Sorting* — How to arrange a given set of elements in some desired order very fast?
- *Searching a graph* — robot motion planning, graph exploration
- *Finding the shortest route* — in a map (Google maps)
- *DNA similarity checking.* — How do you check for a DNA match?
- *Compressing* — a large file. How does UNIX `compress` work?
- *Ranking* — Web pages (How does Google rank Webpages?)

1.4 Fundamental Algorithm Design Techniques

We will learn various algorithm design techniques. These techniques are useful in solving a variety of problems. Mastering these techniques is the key to designing an algorithm for a new problem.

- *Search* techniques — exhaustive search, backtracking, binary search, hashing, fingerprinting

- *Divide/Decrease and Conquer* — a *Recursive* technique
- *Randomization and probabilistic analysis* — randomness in algorithms and analysis of algorithms under a probabilistic input. *Hashing/fingerprinting* is an important randomized technique used in search, cryptography, and other applications
- *Greedy technique* — used in optimization
- *Data Structures* — organizing data efficiently (binary search tree, hash tables, heaps, etc.)
- *Dynamic Programming* — another technique used in optimization
- *Linear Algebraic techniques* — used heavily in matrix analysis
- *Optimization techniques* — find optimal or near-optimal solutions to optimization problems
- *Machine learning* — used widely for data analysis

1.5 Background needed

For this course, it is important to have a strong background in discrete mathematics, taught typically in the first or second year of undergraduate computer science. You will find some background material in the appendix of this book. It is highly recommended that you refresh your background by reviewing a textbook on discrete mathematics (e.g., Rosen [5]). Alternatively, an online reference is the first year computer science undergraduate course on discrete structures at Stanford [3]. The algorithms textbook [2] by Cormen, Leiserson, Rivest, and Stein has an appendix that reviews the basic discrete math concepts that we need. This book reviews some basic background in discrete mathematics and data structures in the appendices.

You also must have taken an undergraduate course in algorithms and data structures and should have good programming experience in at least one high-level programming language, e.g., C++, Java, Python, etc.

1.6 Worked Exercises and Exercises

Each chapter has numerous exercises which complement the material covered in the chapters. Solving the exercises is a good exposure to solving algorithmic problems. Each chapter also has several "worked exercises" — these are important to further understand and apply the concepts covered. The solutions to the worked exercises can provide insights in solving the exercises. However, trying to solve the worked exercises *before* seeing their solution is a good learning technique.

1.7 Advanced Material

The heading of some chapters and sections of the book are marked by a star (*). These are (somewhat) advanced materials for undergraduate students and can be skipped during a first reading. Exercises and Worked Exercises with stars are problems with above average difficulty for undergraduate students.

1.8 References

- There are lots of great books on algorithms that can serve as useful references. Some of these books have influenced this book as well. These can give you a different perspective or cover topics not covered in this book. Some other references and texts are:
  - Cormen, Leiserson, Rivest and Stein: Introduction to Algorithms.
  - Udi Manber: Introduction to Algorithms: A Creative Approach.
  - Kleinberg and Tardos: Algorithm Design.
  - Dasgupta, Papadimitriou, and Vazirani: Algorithms.
  - Horowitz, Sahni, Rajasekaran: Computer Algorithms.
  - Algorithms by Jeff Erickson (https://jeffe.cs.illinois.edu/teaching/algorithms/)

2 PROBLEM SOLVING AND ALGORITHMS

The major goal of algorithms is to solve problems. Problems arise in the real world in virtually every domain, and many of them are algorithmic in nature. Some examples that we are familiar with are:

(1) Given two numbers x and y, and a binary operation $\circ$, evaluate $x \circ y$

(2) Given a number, check whether it is prime or composite

(3) Given an integer n, find a prime number that is n digits long

(4) Given two cities on a road map, find the shortest route between them

(5) Given a chess position, determine the player's *best* move

(6) Given a body of text and a search term, determine whether the search term occurs within the text

(7) Given a search query, find the web pages that are *most relevant* to the query

One can list tens of thousands of such problems that occur in our every day world (see also the problems in Section 1.3).

One may ask why we cannot solve them manually, that is, why we cannot solve them "by hand". The main reasons for not doing that are: (1) it is too cumbersome or difficult, even if the problems involve "small" inputs, (2) it does not usually scale when the problem size becomes large, and (3) (perhaps the most important and obvious) it is too slow. A better alternative, in the computer age, is to design algorithms to solve them. The main goal of this book is to show how one can correctly and efficiently design algorithms to solve problems. To argue about correctness and efficiency, we need mathematics. Understanding the mathematics behind algorithms is another main goal.

In this chapter, we begin with some basic concepts in algorithm design and analysis.

What is an Algorithm?

An algorithm is a sequence of instructions that

- are *precise* — unambiguous and mathematically well-defined

- *terminate* — after a finite number of steps, the algorithm will finish.[1]

2.1 Two Warm-up Problems

Let us warm up by looking at some algorithmic puzzle problems that also illustrate a key algorithmic technique. Our solutions also illustrate efficiency and analysis of algorithms — two things that are part and parcel of every algorithm that we will study. An efficient solution is one which uses as few resources (such as time, space, etc.) as possible, which typically translates to using as few *costly* operations as possible. Mathematical analysis is necessary to show precise bounds on the performance of algorithms.

2.1.1 Find the Celebrity

Problem 2.1 ▶ The Celebrity Problem

A *celebrity* among a group of n people is a person who *knows no one* but is *known by everyone else.* Identify a celebrity by only asking questions to people of the following form: “Do you know this person?”.

The goal is to design an efficient algorithm to identify a celebrity or determine that a group has no such person. By *efficient* we mean asking as few questions as possible.

Solution

This problem can be solved by reducing the problem size by one — a strategy called *decrease and conquer*, a *recursive* technique that is useful in solving many problems. In the decrease and conquer strategy, we reduce the problem size by one (or sometimes by more than one) and then recursively apply the same strategy to the reduced problem until the problem size becomes sufficiently small (e.g., 1). Let us illustrate this by solving the celebrity problem.

We begin with n people. Choose 2 people, say A and B, and ask whether A knows B. If yes, then A cannot be a celebrity and can be eliminated. If no, then B cannot be a celebrity and can be eliminated. This reduces the problem size by one. We repeat the same strategy among the remaining $n-1$ people.

After $n-1$ questions, one person (say X) will remain. If there is a celebrity then it must be this person.[2] To check whether this person is a celebrity, we ask whether X knows everyone else and whether everyone else knows X. The total number of questions is $n-1+n-1+n-1 = 3n-3$. (Exercise 2.2 asks you to improve this to $3n-4$.) An even better algorithm can be given! It can be shown that the celebrity can be found using $3n - \lfloor \log n \rfloor - 3$ questions and this is the best possible (see Worked Exercise 2.3)! That is, in general, one cannot solve this problem using fewer than $3n - \lfloor \log n \rfloor - 3$ questions.

Note that the performance or *complexity* in the above problem is naturally determined by the number of questions. The complexity of an algorithm is typically determined by the costliest operation — this operation often dominates the overall cost of the algorithm.

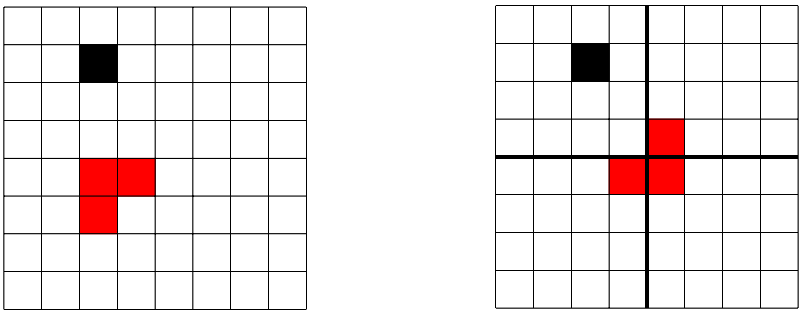
[1] Although most algorithms terminate in a finite number of steps, there are useful algorithms that go on forever (can you think of an example?). We will generally deal with algorithms that have a finite running time.

[2] A formal proof by mathematical induction on why this strategy is correct is given in Chapter 3.

2.1.2 Tiling a Board

Problem 2.2 ▶ The Tiling Problem

Given a $2^n \times 2^n$ board with one missing square, cover all squares (except the one that is missing) using L-shaped tiles with no overlap. (The missing square can be any of the board squares. See Figure 2.1.)



(a) Tiling Problem (b) Solution

Figure 2.1: Tiling problem and its solution.

Solution

There are many ways to solve this problem. This can be viewed as a search problem (we will discuss such problems in Section 2.3) where we want to search for a solution, i.e., a placement of L-shaped tiles to cover the $2^n \times 2^n$ grid. Such problems can be solved by using a *backtracking* strategy, which is essentially an exhaustive strategy in which we place the L-shaped tiles one-by-one, starting from some position, and continue until we are *stuck*, that is, we can no longer place an additional tile, at which point, we backtrack to the previously placed tile and continue the search by trying a different placement. Backtracking is discussed in Section 2.3. By exhaustively searching, backtracking will *always* find a solution, if it exists, but may (and often does) take a long time. Additionally, it is not clear how to establish an upper bound on the number of steps taken during backtracking, making analysis of this technique difficult.

Instead, let us approach this problem by a very different strategy. We will decompose the problem into smaller *subproblems* (unlike backtracking, which works directly on the original problem). This important algorithmic strategy is called *Divide and Conquer*. In this strategy, the bigger problem is decomposed into one or more smaller *subproblems of the same kind*, that is, these subproblems are identical to the original problem but on a *strictly smaller* input size. Then we solve the subproblems by invoking the same strategy. We repeat this strategy till the sizes of the subproblems become small which we can then solve directly without further decomposition. Divide and conquer is a very powerful technique. It can be naturally implemented using *recursion* (we discuss recursion in Chapter 4), a concept central to algorithmic thinking.

We will show that the original problem can be decomposed into 4 subproblems of the *same kind* (see Figure 2.1). That is, each of the subproblems is exactly the same type as the original problem — its dimensions are a power of 2 (in fact, reduced by half) and it has one missing square.

It is important to note that in both decrease and conquer *and* divide and conquer strategies, the resulting subproblems should be of the *same kind*, and also *strictly smaller* in size. It is even better if the subproblem sizes are reduced by a constant factor, in which case the number of levels of the recursion (explained in more detail in Chapter 4) will be *logarithmic* in the problem size.

In the tiling problem, to apply the divide and conquer strategy, we need to determine how to reduce a bigger problem into a smaller problem of the *same* kind. First, it is always instructive to look at *small instances of the problem*: this is very useful in understanding the problem and, in many cases, also in designing an algorithm. When $n = 1$, we have a 2×2 grid with one missing square. It is easy to see that this can be solved, no matter where the missing square is, by placing one L-shaped tile. Thus, if we can reduce the problem size repeatedly, we have a way to solve the problem.

The larger problem has size $2^n \times 2^n$ and we would like our subproblems to have size $2^{n-1} \times 2^{n-1}$. Note that each of the smaller problem should have a missing square, otherwise they would not be the same kind of problem as the original! The key to reducing our problem of size $2^n \times 2^n$ to subproblems of size $2^{n-1} \times 2^{n-1}$ is in placing the L-shaped tile in the right location (convince yourself that putting it in the upper left corner, for example, does not work).

Noting that $4 \times (2^{n-1} \times 2^{n-1}) = 2^n$, let us try to divide the $2^n \times 2^n$ grid into 4 smaller grids each of size $2^{n-1} \times 2^{n-1}$. To get a missing square, we place an L-shaped tile in the center as shown in Figure 2.1b. This creates four subproblems, each of size $2^{n-1} \times 2^{n-1}$, with one missing square (by removing the 3 squares that are covered by the L-shaped tile). If we repeat this $n - 1$ times, we arrive at a 2×2 instance, which we can solve easily.

The above strategy demonstrates not only that a solution exists (wherever the missing square may be), but it also yields a *method* to find a solution using a *divide and conquer* strategy, which is a naturally recursive strategy. Note that this is a *top-down* strategy — reducing the larger problem to a smaller one and finding a solution (the placement of tiles) in the process.

2.2 Problem: Prime Number (Primality) Checking

Let us now look at a classical real-world problem, namely, primality checking, whose complete solution was given relatively recently, though the problem itself has existed for centuries.

Problem 2.3 ▶ Primality Checking

Given a natural number, n, return `True` if it is prime and `False` if it is composite.

This is a problem that requires only a boolean answer: `True` or `False`. It is central to number theory *and* to many algorithms in crytography. Later in the book we will give an efficient (randomized) algorithm for this problem. But first, let us look at some straightforward algorithms. Before we do, we will look at how we can generally approach solving any problem.

2.2.1 Problem Solving Process

Let us explicitly list the major steps in solving an algorithmic problem. Usually, we solve problems by going through these steps implicitly. Sometimes these steps are skipped, a mistake that usually leads to incorrect (or inefficient) solution.

- *Understanding* — what is the *input* and *output* for the problem? A very useful tip in understanding a problem is to see what the output is for a *small-sized* input. Often, this can easily be computed by hand and helps immensely with both understanding the problem and with determining a solution. Additionally, it can help ensure that the solution is correct for certain edge cases.[3] This also helps to ensure that your solution works for trivial (degenerate) cases, e.g., $n = 1$.

- *Problem solving* — Construct an *algorithm*.

  - Many times, this involves understanding how to solve small instances of the problem. This usually suggests an algorithm, although it may not be the most efficient.
  - Attempt to apply an algorithmic design technique. For example, if you want to apply Divide and Conquer — break the problems into smaller subproblems and combine their solutions.
  - Write the algorithm in pseudocode. More importantly, it is *mandatory* to write a description of your algorithm in plain language. In fact, if the description is precise and unambiguous, pseudocode may not even be *necessary*. However, pseudocode is always recommended, since it is precise and makes the transition to a computer program straightforward.

- *Analysis* — Analyze the *correctness* and the *performance* of the algorithm.

  - This usually involves exploiting some mathematics behind the problem.
  - The first thing to do is to argue that the algorithm is correct, i.e., that it produces the correct output for every possible input.[4] Generally speaking, *correctness trumps performance*, since there is no point in having an algorithm that is extremely efficient but incorrect. [5] The second is to analyze the running time of the algorithm — this is called *time complexity analysis*. Sometimes, we also analyze the memory used by the algorithm — this is called *space complexity analysis*.

- *Implementation* — Convert the pseudocode into a computer program.

  - This involves converting the pseudocode in some programming language, e.g. C++, Java, Python, Lisp.

Do not short-circuit the process by omitting the problem solving phase — a typical mistake of beginners — as, in many cases, skipping the above steps leads to erroneous results.

[3] An *edge case* for a problem is one which typically occurs for trivial or extreme (typically small) instances of the problem. For example, what should our solution for Primality checking be for 0 and 1? Similarly, in the celebrity problem, check what happens when n is small, say 2 or 3.

[4] Interestingly, there is a class of algorithms called *randomized algorithms*, where this need not be true always, but instead has to be correct with "high probability."

[5] For some problems, it turns out it is possible to design "efficient" algorithms if we allow some approximation to the correct solution; these are called *approximation algorithms*. We will see such examples later.

2.2.2 Understanding the Primality Checking Problem

It is easy to understand the input and output for this problem.

- *Input:* A natural number n.
- *Output:* `True` (if n is prime) or `False` (if n is composite).

2.2.3 Problem Solving Phase

How can we construct an algorithm? For a start, let us look at the mathematical definition of a prime number.

Definition 2.1 ▶ Prime number

A natural number greater than 1 is prime if it is divisible *only* by itself and 1. A number that is not prime is called *composite*.

The definition suggests an algorithm to check for *primality*:

- Given a number n, check whether any number between 2 and $n-1$ divides n
  - If any of these divides n, output `False`
  - If none of them divides n, output `True`

2.2.4 Pseudocode for Algorithm IsPrime

The pseudocode is given in Algorithm 1. It is quite straightforward: the **for**-loop exhaustively checks whether each of the numbers between 2 and $n-1$ divides n. If any of them divides n, then the number is composite. If none of them divide n, then the number is prime.[6]

Algorithm 1 IsPrime — Naive Primality Checker
Input: A Natural number $n > 1$
Output: `True` if n is prime, else `False`

```
1: func IsPrime($n$):
2:     for $i = 2$ to $n - 1$:
3:         if $i$ divides $n$:
4:             return False
5:     return True
```

2.2.5 Correctness of the Algorithm IsPrime

The algorithm IsPrime *faithfully* implements the mathematical definition of a prime number. In the **for**-loop (line 2), we check for divisibility of n by *every* number between 2 and $n-1$. If n is prime, the condition on line 3 will *always* be `False`, hence line 4 will never execute, and our algorithm will execute line 5 and return `True`. If n is composite, there will be some factor m such that $2 \leqslant m \leqslant n-1$. When $i = m$, the condition on line 3 will be `True`, hence line 4 will execute and return `False`. Thus, the algorithm is correct.

[6] By convention, if $n = 2$, then the set of numbers between 2 and $n-1$ is $\emptyset$. Thus, our algorithm will correctly output that 2 is prime.

2.2.6 Speeding up the algorithm

Algorithm IsPrime is rather slow, especially if the input number n is *prime*. In this case, the **for**-loop runs $n - 2$ times. Thus, the running time of the algorithm is *approximately proportional* to n, the input number.

We can speed up IsPrime by making use of the following theorem.

Theorem 2.1

If i is a divisor of a number n, then n/i is also a divisor of n.

Proof. Write $n = ki$. Then $k = n/i$ divides n by definition. □

If we check a divisor i, then we do not need to check divisor n/i. Thus, the largest value we need to check is when $n/i = i$, or when $n = i^2$. It is therefore enough to check for *divisors up to* $\sqrt{n}$.

2.2.7 FastIsPrime Algorithm

Algorithm 2 FastIsPrime – Faster Primality Checking
Input: A natural number $n > 1$
Output: `True` if n is prime, `False` if n is composite

```
func FastIsPrime($n$):
    for $i = 2$ to $\lfloor\sqrt{n}\rfloor$:
        if $i$ divides $n$:
            return False
    return True
```

The pseudocode for FastIsPrime is given in Algorithm 2. The only change from IsPrime is that the **for** loop runs from 2 to $\lfloor\sqrt{n}\rfloor$. The run time of FastIsPrime is proportional (approximately) to $\sqrt{n}$ which is significantly smaller than n.

However, note that FastIsPrime is still quite slow, as its running time is an *exponential* function in the *size of the input* (See Section 2.6). An important aspect of *efficient* algorithms is that their run time scales *polynomially* in the input size. Even then, we would like to keep the degree of the polynomial as small as possible. In particular, linear time algorithms are preferred, since their run time is proportional to the size of the input. Such algorithms are particularly useful in *Big Data* applications, where the input sizes are extremely large.

2.3 Problem: Search

Search is a fundamental theme that appears often when solving various kinds of problems. A problem can be cast as[7] a *search problem* if it involves searching for elements that satisfy some desired property (or properties) *from* an appropriate *set* S. The set S is called the *search space*.

[7] We say a problem A can be *cast as* another problem B if the solution to problem B can be used to yield a solution to problem A.

Many problems indeed are (or *can be* cast as) search problems, e.g., finding the median in a given set of numbers, finding a particular word (if it exists) in a given text, finding non-trivial factors of a given number (if they exist), finding the *best* move in a given Chess or Go position, finding the shortest route between two points in a map.

Search Method

A straightforward method is *brute-force* or *exhaustive* search:

- examine each element in the set S and see if it satisfies the desired property
- if it does, then output the element as the solution

An exhaustive search examines all possible candidate solutions until a correct solution is found. Exhaustive search is costly (it is often *exponential* in the input size) if the search space is large, which is frequently the case. In such cases, more efficient search algorithms are needed.

2.3.1 n-Queens Problem

Problem 2.4 ▶ n-Queens

Given an $n \times n$ chessboard, place n queens so that no two queens can attack each other by being on the same row, column, or diagonal.

An exhaustive search is to try all possible positions. For the n-Queens problem, the number of possible positions is very large — there are n^2 squares, from which we choose n, for a total of

$$\binom{n^2}{n} \geqslant n^n$$

which grows exponentially in n.[8] A very powerful algorithmic idea that can drastically reduce the number of positions to be searched is called *backtracking*. The backtracking method can be summarized as follows:

- Construct candidate solutions *incrementally*, i.e., in a *step-by-step* fashion.
- Evaluate a partially constructed solution to see if it can be extended by one more *step* without violating the problem's constraints.
- If no extension is possible from this partial solution, backtrack to replace the previous step with another step.
- Repeat until a solution is found.

Let us apply the backtracking method to the n-Queens problem.

[8] We use this combinatorial inequality from Appendix G.2.

n-Queens problem: Backtracking solution

A backtracking process can be represented as a *tree* (see Figure 2.2). We build the solution *one column at a time.* Initially the root of the tree starts with the empty solution (no queens). The *depth* of a node will represent the number of queens placed up to this point. A node's *children* will represent all possible one-column extensions to consider: there will be n children per node — representing the n possible row positions corresponding to that particular column.

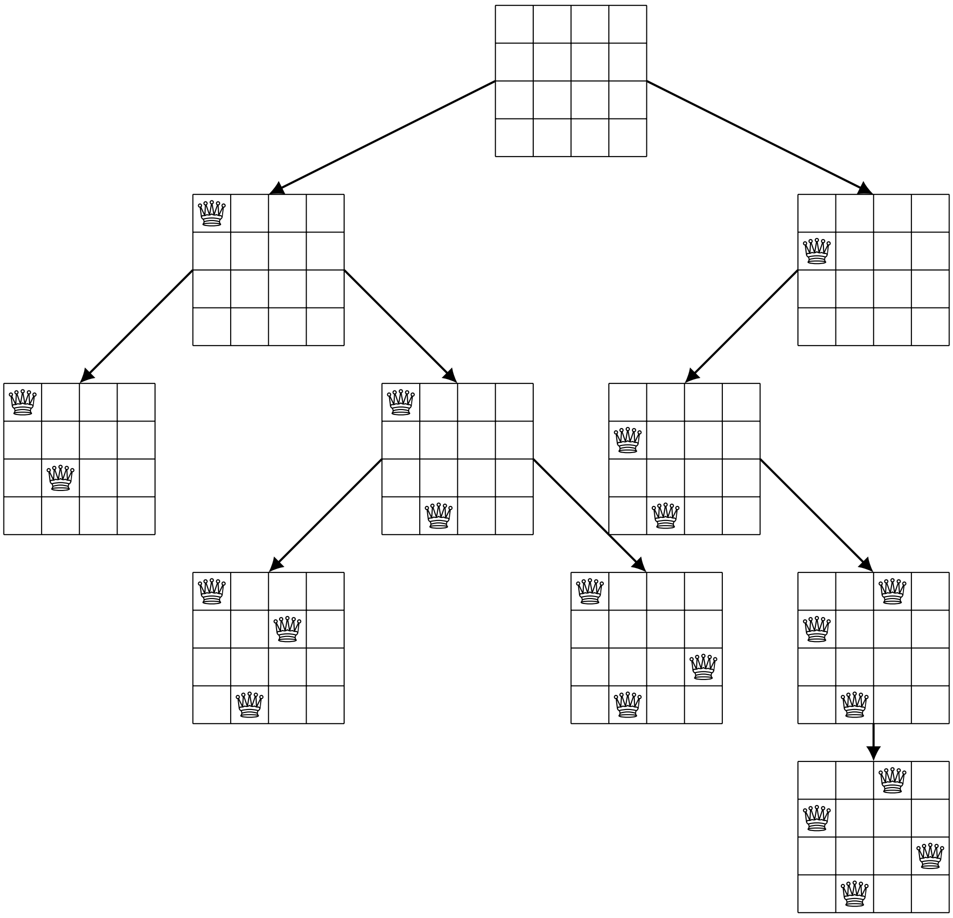


Figure 2.2: Backtracking solution to the 4-queens problem. Each node in the tree represents the placement of a queen. When the search is unable to progress further, we backtrack. One can continue the backtracking until all solutions are found.

How fast is backtracking compared to exhaustive search?

Exhaustive search examines $\binom{n^2}{n}$ positions. On the other hand, backtracking examines only those positions where there is one queen in each column and row. Since queen 1 can be placed in any one of the n rows in the first column, queen 2 can be placed in any one of the $n-1$ remaining rows in the second column (notice that she cannot be in the same row as the queen 2), queen 3 can be placed in any one of the $n-2$ remaining rows in the third column, and so on, for a total of $n \times (n-1) \times \cdots \times 2 \times 1 = n!$ positions. In fact, we could have reduced this number of positions even further by considering the diagonal squares, which become unavailable when a queen occupies a square. This is difficult to compute, however, and so was ignored for simplicity.

We note that $n!$ is significantly smaller than $\binom{n^2}{n}$. For example, for $n = 4$,

$$\binom{4^2}{4} = \binom{16}{4} = \frac{16!}{4!(16-4)!} = \frac{16 \cdot 15 \cdot 14 \cdot 13}{4 \cdot 3 \cdot 2} = 1820$$

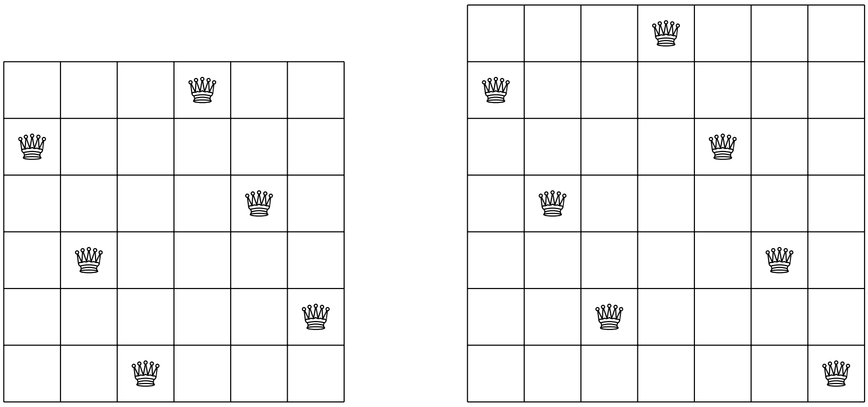
whereas, $4! = 24$. However, $n!$ still increases extremely fast. For example,

$$20! = 2432902008176640000 \approx 2.4 \times 10^{18}$$

How fast can you solve n-Queens?

Both exhaustive and backtracking search can take exponential time. This is too slow. Even if n is as small as 100, this can be beyond the scope of a typical laptop computer. It turns out that one can find a solution to the n-queens problem much faster — in *linear* time, i.e., in time proportional to n. This can be done *analytically*, i.e., without performing any search. Here, we give only a partial solution, which works only for certain values of n, leaving the full solution as an exercise.

The idea is to generalize the solution position for $n = 4$ — see Figure 2.3. Note that there are no solutions for $n = 2$ and $n = 3$. Since there is no queen on the diagonal, we can easily obtain a solution for $n = 5$ by placing the 5th queen on the last square of the 5th column (in the diagonal). For $n = 6$, we can construct the following solution, using the idea from $n = 4$: place the first three queens in the second, fourth and sixth rows of the first 3 columns, respectively, then place the remaining three queens in the first, third, and fifth rows of the last three columns, respectively (see Figure 2.3). This, again, can be easily extended for $n = 7$ by placing the queen on the diagonal as shown in Figure 2.3. Note, however, that this does not work for $n = 8$. On the other hand, you can verify that it does work for all values of n that are of the form $6t$ as well as $6t + 4$. By extension, this also works for values of $n = 6t + 1$ and $n = 6t + 5$. Thus this solution works for all values of n that leave a remainder 0, 1, 4, and 5 when divided by 6. For other values of n (i.e., those that leave a remainder of 2 and 3, we must modify the above solution somewhat. We leave this as an exercise (Exercise 2.9).



$n = 6$

$n = 7$

Figure 2.3: Solution for n-queens.

It is thus interesting to note that we can always find *a* solution to the n-Queens problem, and moreover, find it very quickly. But to find *all* solutions to the n-Queens problem is a very difficult exercise. To do so requires time that is *at least* exponential in n. For example, even today's fastest computers struggle to find all solutions to the 1000-Queens problem!

2.3.2 The Integer Square Root Problem

We next consider a number-theoretic problem, like primality checking, which is, however, quite a bit simpler. Namely, we wish to find the square root of a given number. More precisely, we have the following problem.

Problem 2.5 ▶ The Integer Square Root Problem

Given a natural number n, find the *largest integer* that is less than or equal to $\sqrt{n}$.

Recall that we needed to solve the above problem in Algorithm FastIsPrime.

Finding the Square Root: A search Problem

This can be cast as a search problem where the search space S is the set $\{1, \ldots, n\}$, and the number desired is $\lfloor\sqrt{n}\rfloor$, i.e., the largest integer that is less than or equal to $\sqrt{n}$.

A naïve (brute-force) algorithm

A naïve algorithm (Algorithm ISqrt) for The Integer Square Root Problem is as follows: simply search all numbers from 1 to n, one by one, and check if they satisfy the desired property.

Algorithm 3 ISqrt — Brute Force square root function
Input: A natural number n
Output: $\lfloor\sqrt{n}\rfloor$

```
func ISqrt($n$):
    for $i = 1$ to $n$:
        if $i^2 \leqslant n$ and $(i+1)^2 > n$:
            return $i$
```

This algorithm is slow: it takes about $\sqrt{n}$ iterations of the *for* loop since the number of operations per loop is four – two multiplications and two comparisons.

A Significantly Faster Algorithm: Binary Search

The main algorithmic idea behind the faster algorithm (Algorithm BinaryISqrt)[9] is the technique of *binary (or bisection)* search, a fundamental technique that is useful in many algorithmic applications (arguably, the most used algorithmic technique in computer science). Binary search reduces the search space by approximately *half* in each iteration. That is, if the search space is originally of size k, then in the first iteration it becomes at most $k/2$, and, in the next iteration, it becomes at most $k/4 = k/2^2$, and in the next, $k/8 = k/2^3$, and so on. Hence, after $\log_2 k$ iterations, the size is at most $k/2^{\log k} = 1$.

[9] Note that this algorithm is an iterative algorithm. However, it is easy to modify to make it recursive — see Exercise 2.3.

Thus, after $\log_2 k$ iterations, the search space has reduced to at most 1, and we have found the answer. Thus, the convergence to the desired answer happens in at most $\log k$ steps.

Thus, in The Integer Square Root Problem, the algorithm finishes in time at most $\log n$, since the initial search space is of size at most n. Figure 2.4 illustrates the algorithm on an input of 31. The algorithm maintains two values, `low` and `high`, which are initialized to 1 and n, respectively. The algorithm checks if the mid point value, $\texttt{mid} = \lfloor(\texttt{low} + \texttt{high})/2\rfloor$, is the desired answer by checking whether $\texttt{mid}^2 \leqslant n$ **and** $(\texttt{mid} + 1)^2 > n$. If `mid` is not the answer, then the desired answer is either less than `mid` or greater than `mid`; depending on which, the value `low` or `high` is set to `mid` accordingly. It is important to note that the following *loop invariant*[10] is maintained in every iteration of the **while** loop: the desired answer, $\lfloor\sqrt{n}\rfloor$, is *always between* `low` and `high` at the beginning of the iteration. The algorithm ends when `mid` reaches $\lfloor\sqrt{n}\rfloor$, as desired. Note that the loop invariant guarantees correctness at the end: since the size of the search space is reduced (roughly by a factor of 2), the desired value, which is always between `low` and `high`, will eventually be returned. We can formally prove that the loop invariant is maintained by *mathematical induction*, which we will see in Chapter 3 (see Worked Exercise 3.1).

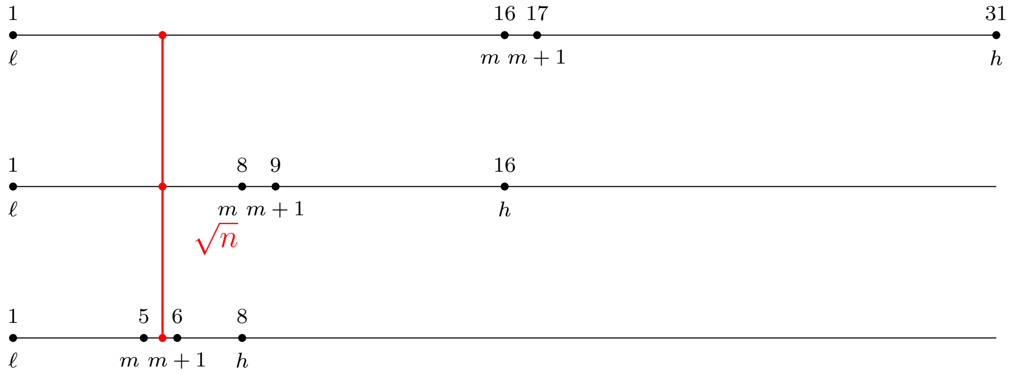


Figure 2.4: Illustrating the Algorithm BinaryISqrt on $n = 31$.

Algorithm 4 BinaryISqrt – Binary Search Square Root
Input: A natural number n
Output: $\lfloor\sqrt{n}\rfloor$

```
func BinaryISqrt($n$):
    lo $= 1$    ▷ Low value of guess
    hi $= n$    ▷ High value of guess
    loop:
        mid $= \lfloor(\texttt{lo} + \texttt{hi})/2\rfloor$    ▷ current estimate
        if mid$^2 \leqslant n$ and $(\texttt{mid} + 1)^2 > n$:
            return mid
        else if mid$^2 < n$:    ▷ mid $< \lfloor\sqrt{n}\rfloor$
            lo $=$ mid
        else:    ▷ mid $> \lfloor\sqrt{n}\rfloor$
            hi $=$ mid
```

[10] A loop invariant is a property that holds in every iteration of a for or while loop.

2.4 A Model for Computation: Random Access Machine (RAM)

We describe a simple computation model used to *analyze the running time of algorithms*. This model captures computation that takes place in a single processor machine, where instructions are executed one after the other in a sequential manner.[11]

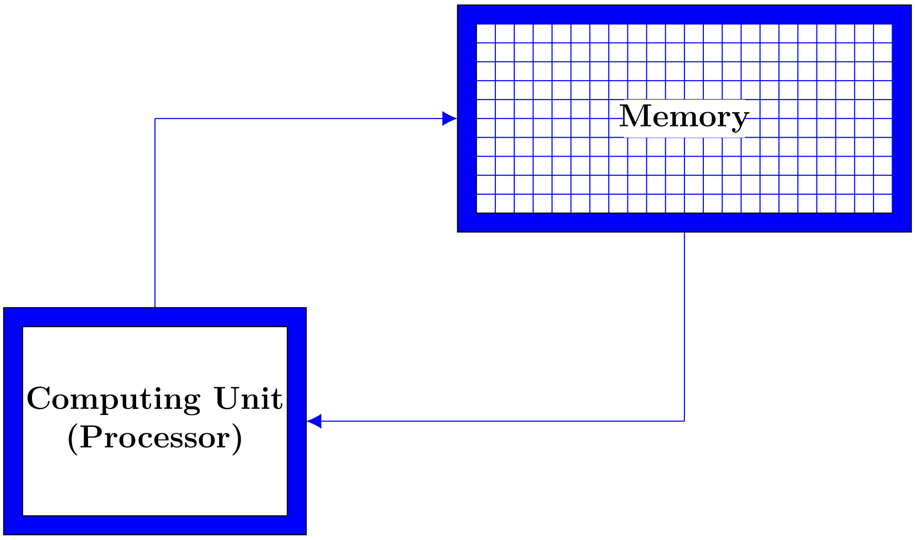


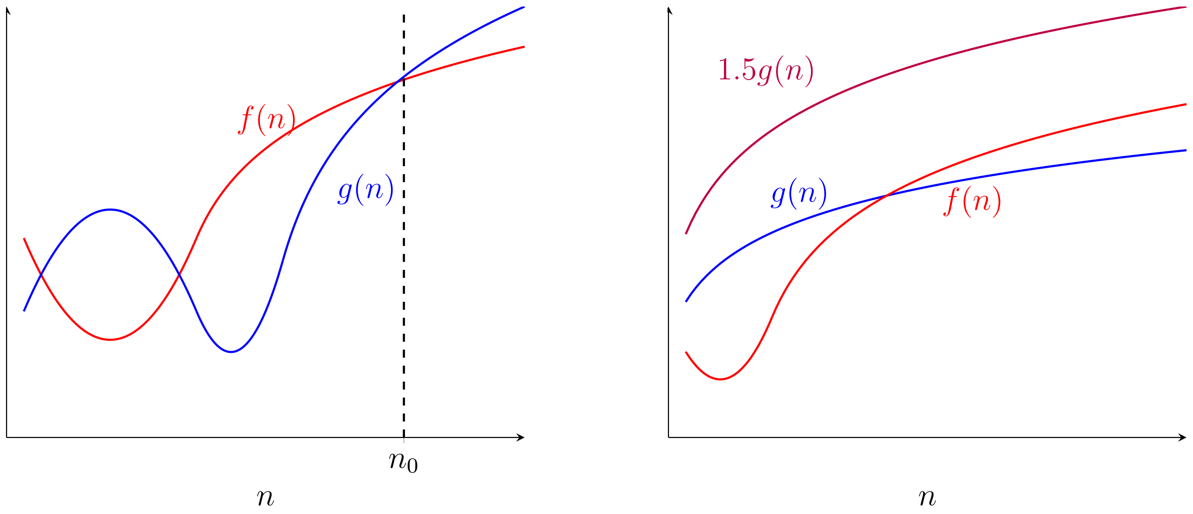
Figure 2.5: Random Access Machine (RAM) model. The RAM model consists of a computing unit plus a memory unit that allows random access. The memory is assumed to consist of an unlimited number of cells, each having a unique address. Each cell can be accessed in constant time in any step.

The Random Access Machine model, popularly called the RAM model (see Figure 2.5), has the following key features:

1. A computing unit plus memory.
2. Instructions are executed sequentially (i.e., one after the other, according to the order of the program).
3. Accessing (reading/writing) any one memory cell takes one step. This is why it is called *random* access even though there is no randomness. That is, regardless of its location, any memory cell can be accessed in one step.
4. Any simple operation takes one step.[12]

[11] Many modern computers are actually *parallel* machines in which more than one processor operates in parallel, for which we need to modify the RAM model.

[12] We must clarify which operations are simple. Basic arithmetic on two integers that are smaller than the *word size* (in many machines, this is 64 bits) can be assumed to take a single step. However, this is not the case for n-bit numbers — for arbitrary n, these operations can take n steps or more — and so care must be taken when accounting for the cost of certain operations. However, in many algorithms, this abstraction of assuming a unit cost for primitive operations simplifies analysis tremendously, without which it is hard to quantify the performance of the algorithm in a clean way. This is indeed a powerful abstraction!



(a) $f(n) \leqslant g(n)$ for all $n \geqslant n_0$. Note that behavior for $n < n_0$ is irrelevant.

(b) $f(n) \geqslant g(n)$, but $f(n) \leqslant 1.5g(n)$ for all n.

Figure 2.6: Illustration of Big-O notation. In both figures, $f(n) = \mathcal{O}(g(n))$.

We note that, in many cases, we even further simplify the assumptions of the RAM model. Typically, we only bound the number of *costly* operations — those that take the greatest number of time steps — that the algorithm performs. Usually, bounding the number of costly operations will bound the total running time as well (up to some constant factor). For example, in analyzing sorting algorithms, we typically only focus on the total number of comparisons made by the algorithm, while ignoring other operations. This is usually a good metric by which to measure the performance of comparison-based sorting algorithms (more on this in Chapter 5).

2.5 Asymptotic Notation: Big-$\mathcal{O}$, Big-Ω, and Big-Θ

Algorithmic analysis would not be as elegant were it not for the emphasis placed on asymptotic and approximate analysis, rather than exact analysis, for a given input size. In other words, emphasis is placed primarily on how the algorithm's performance *scales* with the problem size in a *reasonably approximate* manner. This allows the algorithm analyzer to focus on quantifying the dominant function that determines the algorithm's performance, while avoiding cumbersome details. We will see examples of this philosophy throughout the course. As an example, we say that IsPrime takes $\mathcal{O}(n)$ time and FastIsPrime takes $\mathcal{O}(\sqrt{n})$ time.

2.5.1 Big-$\mathcal{O}$

Big-$\mathcal{O}$ notation captures two key aspects: (1) the behavior of a function of n as n grows *large* (i.e., the value of the function when n is large and ignoring its behavior for small values of n) and (2) *ignoring constant (multiplicative) factors* in the growth of the function (constant factors do not matter).

Formally, a function $f(n)$ is Big-$\mathcal{O}$ of $g(n)$, written $f(n) = \mathcal{O}(g(n))$, if there is some positive constant $c > 0$ (independent of n), such that $cg(n)$ grows asymptotically (i.e., as n grows) at least as *fast* as $f(n)$ (see Figure 2.6). In other words, $f(n) \leqslant cg(n)$ for sufficiently large n that is, for all $n \geqslant n_0$ for some constant n_0.

Informally, we will say that if $f(n) = \mathcal{O}(g(n))$, then $g(n)$ (possibly, multiplied with a large-enough constant factor) *dominates* or *upper bounds* $f(n)$. Formally, we have the following definition.

Definition 2.2 ▶ Big-$\mathcal{O}$

Let $f(n)$ and $g(n)$ be positive functions. Then $f(n)$ is Big-$\mathcal{O}$ of $g(n)$, written $f(n) = \mathcal{O}(g(n))$, if there exist positive constants c and n_0, independent of n, such that, for all $n \geqslant n_0$, $f(n) \leqslant cg(n)$.

The above definition is usually cumbersome for demonstrating the relationship between two functions. An equivalent, but easier to apply, definition, uses limits:

Definition 2.3 ▶ Big-$\mathcal{O}$ (Limit Definition)

Given two positive functions $f(n)$ and $g(n)$, we say $f(n) = \mathcal{O}(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leqslant c$$

where $c \geqslant 0$ is a fixed constant (assuming the limit exists[1]). In other words, the limit is bounded above by a fixed constant.

[1] If the limit does not exist, we can replace the limit with the Limit Superior, or $\limsup$ (e.g., see Example 2.1). However, for most of the functions that we will encounter, the limit exists and we do not need to worry about this technicality.

2.5.2 Big-Ω

Big-Ω notation can be considered the complement to Big-$\mathcal{O}$ notation. If $g(n) = \mathcal{O}(f(n))$ then we can say that $f(n) = \Omega(g(n))$. Informally, we will say that if $f(n) = \Omega(g(n))$, then $g(n)$ (multiplied by a small-enough constant factor) is *dominated by* or *lower bounds* $f(n)$.

Formally, we define Big-Ω as we did for Big-$\mathcal{O}$.

Definition 2.4 ▶ Big-Ω

Let $f(n)$ and $g(n)$ be positive functions. Then $f(n)$ is Big-Ω of $g(n)$, written $f(n) = \Omega(g(n))$, if there exist positive constants c and n_0, independent of n, such that, for all $n \geqslant n_0$, $f(n) \geqslant cg(n)$.

Alternatively, using limits we have the definition:

Definition 2.5 ▶ Big-Ω (Limit Definition)

Given two positive functions $f(n)$ and $g(n)$, we say $f(n) = \Omega(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geqslant c$$

where $c > 0$ is a fixed constant (assuming the limit exists[1].). In other words, the limit is bounded below by a fixed constant.

[1] Otherwise, we can modify the definition to make us of the lim inf, rather than the limit, similar to the definition for Big-$\mathcal{O}$

2.5.3 Big-Θ

Big-Θ notation is used to compare functions that are both Big-$\mathcal{O}$ and Big-Ω of each other, i.e., functions that have the same asymptotic behavior. More precisely, two positive functions $f(n)$ and $g(n)$ are said to be *Big-Theta* of each other if their growth is asymptotically *within a constant factor of each other*. We define Big-Θ as follows.

Definition 2.6 ▶ Big-Θ

Let $f(n)$ and $g(n)$ be positive functions. Then $f(n)$ is Big-Θ of $g(n)$, written $f(n) = \Theta(g(n))$, if there exist positive constants c_1, c_2, and n_0, independent of n, such that for any $n \geqslant n_0$, $c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$. Equivalently, $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ *and* $f(n) = \Omega(g(n))$.

Informally, we will say that if $f(n) = \Theta(g(n))$, then both $f(n)$ and $g(n)$ are of the *same order*, i.e., they are essentially the same function (as far as asymptotic growth is concerned).

Alternatively, using limits we have the definition:

Definition 2.7 ▶ Big-Θ (Limit Definition)

Given two positive functions $f(n)$ and $g(n)$, we say $f(n) = \Theta(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

where $c > 0$ is a fixed constant (assuming the limit exists[1]).

[1] Otherwise, we can modify the definition to make us of both lim sup and lim inf, rather than the limit, similar to the definition for Big-$\mathcal{O}$ and Big-Ω

We next illustrate with some examples. More examples can be found in Section 2.7.

Example 2.1.

1. $2n = \mathcal{O}(n) = \Theta(n)$. Constants are ignored! Big-O, Big-Omega, and Big-Theta notation "hides" *constant factors.*

2. $n^3 + 2^{1000}n^2 + 2^{100000}n = \mathcal{O}(n^3) = \Theta(n^3)$.

3. In general for any polynomial

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where a_i are constants and $a_d > 0$, we have $p(n) = \Theta\left(n^d\right)$.

4. $n = \mathcal{O}(n^2)$, but $n \neq \Theta(n^2)$. That is, n^2 grows *strictly* faster than n.

5. $\lg n = \mathcal{O}(n)$. (By $\lg n$ we mean $\log_2 n$; also unless otherwise stated, we assume base 2 to logarithmic functions.) There are several ways to show this, but let us use the limit definition: $\lim_{n\to\infty} \frac{\lg n}{n} = \infty/\infty$, an indeterminate form, by allowing both the numerator and denominator to go to ∞. To get a finite limit, we use L'Hôpital's rule (see Worked Exercises) by taking derivatives of the numerator and denominator and then applying the limit:

$$\lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{\frac{d}{dn}\lg n}{\frac{d}{dn} n} = \lim_{n\to\infty} \frac{1}{n \lg 2} = 0$$

6. $n^{1000} = \mathcal{O}(2^n)$, but $n^{1000} \neq \Theta(2^n)$.

7. Algorithm IsPrime runs in $\mathcal{O}(n)$ time, where n is the input number.

8. Consider f and g given by

$$f(n) = \begin{cases} n^2 & \text{n is even} \\ n^3 & \text{n is odd} \end{cases}$$
$$g(n) = n^3$$

Note that $\lim_{n\to\infty} f(n)/g(n)$ does not exist (i.e., it does not converge — it fluctuates between 0 and 1 depending whether n is even or odd). However, the ratio $f(n)/g(n)$ is upper-bounded by 1, hence $f(n) = \mathcal{O}(g(n))$. Formally, this "limit upper bound" is captured by lim sup:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

2.5.4 Little-o and Little-ω

Sometimes, we would like asymptotic notation to capture a function that grows *strictly slower* than some other function. For this we use Little-o notation which can be considered a stronger form of Big-$\mathcal{O}$ notation. If $f(n) = o(g(n))$ then $f(n) = \mathcal{O}(g(n))$, but the converse is not true in general.

Definition 2.8 ▶ Little-o

Let $f(n)$ and $g(n)$ be positive functions. Then $f(n)$ is Little-o of $g(n)$, written $f(n) = o(g(n))$, if for *any* positive constant c there is a constant n_c such that, for all $n \geqslant n_c$, $f(n) < cg(n)$.

The above definition says that if $f(n)$ is $o(g(n))$, i.e., $f(n)$ grows *strictly* slower than $g(n)$, then for any arbitrarily small c, for sufficiently large n, $f(n)$ is dominated by $cg(n)$. In other words, regardless of how small c is, $cg(n)$ will *eventually* dominate $f(n)$.

Alternatively, using limits we have the definition:

Definition 2.9 ▶ Little-o (Limit Definition)

Given two positive functions $f(n)$ and $g(n)$, we say $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Similarly, we would like asymptotic notation to capture a function that grows *strictly faster* than some other function. This is the Little-ω notation, and can be considered a complement to Little-o notation. That is, if $g(n) = o(f(n))$, then $f(n) = \omega(g(n))$.

Definition 2.10 ▶ Little-ω

Let $f(n)$ and $g(n)$ be positive functions. Then $f(n)$ is Little-ω of $g(n)$, written $f(n) = \omega(g(n))$, if for *any* positive constant c there is a constant n_c such that, for all $n \geqslant n_c$, $f(n) > cg(n)$.

Alternatively, the above can be expressed using limits as:

Definition 2.11 ▶ Little-ω (Limit Definition)

Given two positive functions $f(n)$ and $g(n)$, we say $f(n) = \omega()g(n)$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

We note that Little-o does not require the use of $\limsup$. Notice that the property that, for all $c > 0$, there exists an n_0 such that $f(n) \leqslant cg(n)$ for all $n > n_0$, implies $|f(n)/g(n)| \leqslant c$, which is, in fact, the very *definition* of $\lim_{n \to \infty} f(n)/g(n) = 0$. A similar argument applies to Little-ω and $\liminf$.

Example 2.2.

- $1/n = o(1)$
- $n^{1000} = o(1.1^n)$
- $n^2 = \omega(n)$
- Let $f(n) = n$ and $g(n) = n^{1+\sin n}$. Is $f(n) = o(g(n))$ or $g(n) = o(f(n))$?

  Notice that
  $$\lim_{n \to \infty} \frac{n}{n^{1+\sin n}} = \lim_{n \to \infty} n^{-\sin n}$$
  does not exist. Hence the two functions are not asymptotically comparable via Little-o or Little-ω notation. In fact,
  $$\limsup_{n \to \infty} n^{-\sin n} = \infty$$
  $$\liminf_{n \to \infty} n^{-\sin n} = 0$$

so the two functions cannot even be compared via Big-$\mathcal{O}$ or Big-Ω! Intuitively, this is because the function $g(n) = n^{1+\sin n}$ oscillates *around* $f(n) = n$.

2.5.5 Summary

Below is a summary of the definitions for different asymptotic notations and their equivalent limit definitions. We assume the limit of $\frac{f(n)}{g(n)}$ exists.

Notation	Definition	Limit Definition
$f(n) = \mathcal{O}(g(n))$	$f(n) \leqslant cg(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) = \Omega(g(n))$	$f(n) \geqslant cg(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) = \Theta(g(n))$	$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) = o(g(n))$	$f(n) < cg(n)$ for all $c > 0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \omega(g(n))$	$f(n) > cg(n)$ for all $c > 0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Table 2.1: Asymptotic Notation Summary

Finally, we present a theorem that can be useful in the application of the limit definition.

Theorem 2.2

Suppose $\log f(n) \leqslant c \log g(n)$ for some $0 \leqslant c \leqslant 1$. Then $f(n) = \mathcal{O}(g(n))$. In particular, if $\log f(n) = o(\log g(n))$, then $f(n) = \mathcal{O}(g(n))$.

Proof. Simply note that, if $\log f(n) \leqslant c \log g(n)$, then

$$\begin{aligned} 2^{\log f(n)} &\leqslant 2^{c \log g(n)} \\ f(n) &\leqslant \left(2^{\log g(n)}\right)^c \\ f(n) &\leqslant g(n)^c \end{aligned}$$

The righthand side above is clearly less than $g(n)$ for $c \leqslant 1$. □

2.5.6 Some common runtimes of algorithms

We use the following terminology for some common run times of algorithms.

Runtime	Terminology
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log n)$	Linearithmic
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^c)$ for $c > 1$	Polynomial
$\mathcal{O}(c^n)$ for $c > 1$	Exponential

Polynomial time algorithms are considered *efficient* — typically preferred compared to exponential time algorithms, although this should be carefully interpreted, especially in practice. For example, a n^{10} algorithm may not be preferable over a $(1.1)^n$ algorithm (although the latter is exponential), for even reasonably large values of n. However, a phenomenon that is often seen in the theory of algorithms is that, once polynomial time solvability is established, the degree of the polynomial can be reduced, leading to more efficient algorithms. Thus, breaching the exponential barrier has important consequences both in theory and in practice. In particular, the famous *P vs NP* problem [1] (considered one of the most important open problems in mathematics and computer science[13]) is very much related to this. For many important algorithmic problems in the real world, the best known algorithms take exponential time and no polynomial time algorithms are known. However, *verifying* the solution to these algorithmic problems can be done in polynomial time. The P versus NP problem asks whether for such problems polynomial time algorithms exist for *computing* a solution, not only for verifying.

Even among polynomial time algorithms, ones with small exponents are preferred. For many *big data* problems with large input size, linear time algorithms are not only preferred, but *required*. Even a small exponent greater than 1 can have a huge impact on performance for extremely large data sets.

Another key point one should note about asymptotic notation is that it *hides* constants. It is quite possible that a $\mathcal{O}(n)$ algorithm can be very impractical since the constant factor masked by the big-O notation can be very large. Such an algorithm might be slower in practice than even an $\mathcal{O}(n^2)$ algorithm with a small hidden constant factor. These issues are important when implementing an algorithm.

2.6 Runtimes of Algorithms with Numerical Inputs

A common misunderstanding in the analysis of algorithms is associated to those with *numerical* inputs. Specifically, there is a distinction between the numerical *input* and the *input size*. We want to emphasize that the complexity of an algorithm is a function of the input size. In particular, for those with numerical inputs, the input size is assumed to be the number of *bits* needed to encode the input.

For example, the algorithm IsPrime has runtime $\mathcal{O}(n)$ — one may erroneously conclude that this is a *linear* time algorithm. *In fact*, it is an exponential time algorithm! The *input size* of a number n is the number of *bits* used to represent it. In the case of IsPrime,

[13] In fact, the P versus NP problem is listed as one of the 7 Millennium Prize Problems of the Clay Mathematics Institute that carry a million dollar prize!

this is $b = \lceil \log n \rceil = \mathcal{O}(\log n)$. Thus, the input size of IsPrime is b and its runtime is $\mathcal{O}(n) = \mathcal{O}\left(2^b\right)$, so this algorithm has a runtime that is, in fact, *exponential* in its input size. Similarly, for FastIsPrime, the runtime is $\mathcal{O}(\sqrt{n}) = \mathcal{O}\left(\sqrt{2^b}\right) = \mathcal{O}\left(\sqrt{2}^b\right)$, and is also exponential.

Because these algorithms are polynomial in the input value but not in *the size of the input*, they are called *pseudo-polynomial time* algorithms. Going forward, we will typically consider the input-size of an algorithm with numerical input n as $\mathcal{O}(\log n)$, without explicitly writing $b = \lceil \log n \rceil$. As demonstrated above, care must be taken with analysis of such algorithms. Later, we will encounter the 0/1 Knapsack Problem (see Chapter 6), in which this distinction is *hugely* important. This problem, which is NP-hard, admits a pseudo-polynomial time algorithm, but has no known polynomial time algorithm.

2.7 Worked Exercises

Worked Exercise 2.1. Rank the following 8 functions by order of growth. That is, find an arrangement $f_1, f_2, \ldots, f_8$ of the functions satisfying

$$\begin{gathered} f_1 = \mathcal{O}(f_2) \\ f_2 = \mathcal{O}(f_3) \\ \vdots \\ f_7 = \mathcal{O}(f_8) \end{gathered}$$

Justify your ordering.

$$\log^3 n,\ n^{\log\log n},\ n^{1+1/\log n},\ 4^{\log n},\ n!,\ 2^{\sqrt{2\log n}},\ n^2,\ (1.1)^n$$

Note that $\log^3 n$ is the usual way of writing $(\log n)^3$.

Solution. The ordering (in increasing order of growth) is:

$$\log^3 n,\ 2^{\sqrt{2\log n}},\ n^{1+1/\log n},\ 4^{\log n} = n^{\log 4} = n^2,\ n^{\log\log n},\ (1.1)^n,\ n!$$

We use two helpful ideas in comparing functions:

1. When comparing functions of the form f^g it might be helpful to compare the logarithm of those functions, i.e., $\log f^g = g \log f$ and apply Theorem 2.2.

   The inequality $a^n < n! < n^n$ (for $a > 1$ and sufficiently large n) can be helpful in bounding $n! = n(n-1)\ldots(2)(1)$. For example, if $a = 2$, we have $2^n < n! < n^n$ for $n > 4$.

2. Apply the limit definition of Big-$\mathcal{O}$. In particular, it is helpful to recall L'Hôpital's rule for computing the limit of the ratio of two functions: if

   $$\lim_{x\to a} \frac{f(x)}{g(x)}$$

   evaluates to an indeterminate form, such as $0/0$ or ∞/∞, then

   $$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Here, a can be any number (or $\pm\infty$) and f' and g' denote the derivatives of f and g, respectively.

To compare $\log^3 n$ and $2^{\sqrt{2\log n}}$, compare the log of both the functions — the first is $\log\log^3 n = 3\log\log n$ and the second $\log 2^{\sqrt{2\log n}} = \sqrt{2\log n}$. The latter grows faster: this can be verified by substituting $x = \log n$: the first then becomes $3\log x$ and the second becomes $\sqrt{2x}$. Since a polynomial function grows faster than a logarithmic function, the latter grows faster.

Let $f(n) = n^{1+1/\log n}$. Taking the logarithm of both sides, we have,

$$\log f(n) = \left(1 + \frac{1}{\log n}\right)\log n = \log n + 1$$

Comparing this with $\log 2^{\sqrt{2\log n}} = \sqrt{2\log n}$, we see $\log n$ grows faster than $\sqrt{2\log n}$. Also note that $f(n) = 2^{\log n + 1} = 2n$. Clearly n^2 grows faster than $2n$. To compare n^2 and $n^{\log\log n}$, clearly the latter is growing faster, as the exponent is growing with n, as opposed to being fixed (as in the case of n^2).

Similarly, by comparing the logarithms of $n^{\log\log n}$ and $(1.1)^n$, we see that the latter is faster.

Finally, $n! > (1.1)^n$, for large n, because of the following: $n! = n(n-1)(n-2)\ldots 1 \geqslant 3 \times 3 \times 3 \ldots 3 \times 2 \times 1 \geqslant 3^{n-2}$. It is easy to see that $3^{n-2} > (1.1)^n$ for all $n \geqslant 3$. □

Worked Exercise 2.2. You are given a **sorted** array $S[1,\ldots,n]$ of n *distinct* numbers, i.e., the numbers in the array are arranged in **increasing** order: $S[1] < S[2] < \cdots < S[n]$.

Give a recursive algorithm to **search** for a given element (say x) in the array. If x is present in the array, the algorithm should return the index of x, otherwise it should return `Null`.

Your algorithm should take $\mathcal{O}(\log n)$ comparisons. Give pseudocode of your algorithm and argue that your algorithm indeed takes $\mathcal{O}(\log n)$ comparisons (in the worst case).

Solution. The algorithm is very similar to the BinaryISqrt for The Integer Square Root Problem. However, care must be taken to avoid off-by-one errors in setting the termination condition and the `low` and `high` values. The pseudocode is given in Algorithm 5.

Algorithm 5 BinarySearch – Find position of value within sorted array
Input: Sorted array $S[1,\ldots,n]$ and an element x
Output: If x is in S, then i such that $S[i] = x$, otherwise `Null`

```
func BinarySearch($S$, $x$):
    lo = 1    ▷ Low value of guess
    hi = $n$    ▷ High value of guess
    while lo $\leqslant$ hi:
        mid = $\lfloor$(lo + hi)/2$\rfloor$
        if $S$[mid] == $x$:
            return mid
        else if $S$[mid] < $x$:
            lo = mid + 1
        else:
            hi = mid − 1
    return Null
```

The analysis is again very similar to that of The Integer Square Root Problem. The search space decreases by at least a factor of 2, hence the number of iterations of the **while** loop is at most $\log n$. A formal correctness proof follows by mathematical induction — refer to the argument given in the book for The Integer Square Root Problem given in Section 2.3.2. □

Worked Exercise 2.3. Show that the celebrity problem can be solved by using $3n - \log n - 3$ questions (assume that n is a power of 2). (Hint: Group people into pairs — similar to a knockout tournament, e.g., Wimbledon tennis tournament.)

Solution. As in the hint, first group the n people in pairs of two — i.e., we will have $n/2$ pairs. This will be the first "round". For each pair, say (x, y), ask one question — if x knows y. This question will eliminate one person — i.e., they cannot be a celebrity. The winner from this pair will be one who is not eliminated. All the winners from this round — there will be $n/2$ of them — will advance to the next round. In the next round, again repeat the process — group into $n/4$ pairs and question — $n/4$ will advance to the next round. This is exactly how a knockout tournament is organized. Finally, there will be one champion (the winner in the final round) — after $\log n$ rounds; because each round eliminates half the number of people. How many questions were asked before the champion is found? This can be computed by summing up the questions asked in each round:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 = n\left(\frac{1}{2} + \frac{1}{2^2} + \frac{n}{2^3} + \cdots + \frac{1}{2^{\log n}}\right)$$

The series on the right hand side is a geometric sum (see Appendix G): hence that can be evaluated as

$$\frac{1}{2} + \frac{1}{2^2} + \frac{n}{2^3} + \cdots + \frac{1}{2^{\log n}} = \frac{\frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{\log n}\right)}{\frac{1}{2}} = 1 - \frac{1}{n}$$

Therefore,

$$\begin{aligned}\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 &= n\left(\frac{1}{2} + \frac{1}{2^2} + \frac{n}{2^3} + \cdots + \frac{1}{2^{\log n}}\right)\\ &= n\left(1 - \frac{1}{n}\right)\\ &= n - 1\end{aligned}$$

If there is a celebrity, they must be the champion. This has to be checked by asking whether all others know the champion and whether the champion knows anybody else. This takes no more than $2n - 2$ questions. However, note that we do not have to ask the same question (in one way) with the person who *lost* to the winner. This saves $\log n$ questions. Hence, the total number of questions is $n-1+2n-2-\log n = 3n-\log n-3$. □

Worked Exercise 2.4. There are n coins placed in a row (see Figure 2.7 for an example). The goal is to form $n/2$ pairs of them by a sequence of moves. On each move a single coin can jump left or right over *two coins adjacent* to it (either over two single coins or one previously formed pair) to land on the next single coin — no triples are allowed. Note that the coin cannot land on an empty space (it has to land on exactly one coin). Empty space between adjacent coins is ignored.

Find out all values of n for which the puzzle has a solution and devise an algorithm that solves the puzzle in the minimum number of moves for such n. Explain why your solution works in the minimum number of moves.

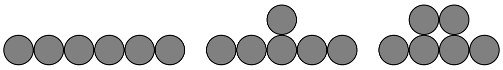


Figure 2.7: Figure for Worked Exercise 2.4. Note that there is no solution for $n = 6$.

(Hint: Clearly, there is no solution when n is odd. Find the smallest n for which there is a solution. For larger n use a decrease and conquer strategy).

Solution. Clearly, there is no solution for odd n. For even n, let us first see if there is a solution for small values (this is always a good idea!). It is easy to see that $n = 2$, 4, 6 have no solution. For $n = 8$, there is a solution in 4 steps (see Figure 2.8), which can be found either by exhaustive search or by backtracking. Clearly this is minimal, since 4 moves are needed.

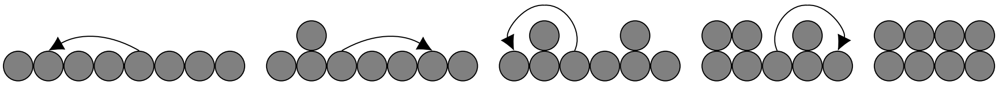


Figure 2.8: Solution to the Coin Jump exercise for $n = 8$

For a general even number, say $n = 8 + 2k$, we reduce the problem by decrease-and-conquer. In particular, we decrease by 2. Assume the n coins are arranged in increasing order from 1 to n (as above). The the first move is: $4 \rightarrow 1$. This reduces the problem to the *same* problem with size $n - 2$ which we solve again recursively.

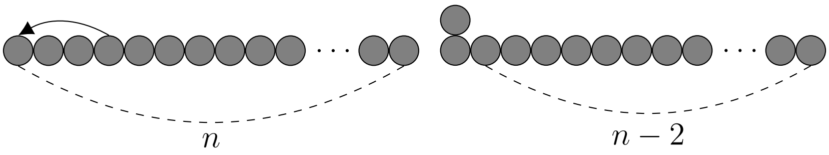


Figure 2.9: Solution to the Coin Jump exercise for general n

The solution clearly uses the minimum (i.e., $n/2$) number of moves, since each move pairs up two coins. □

2.8 Exercises

Exercise 2.1. Does the Algorithm Foo terminate? Justify your answer.

Algorithm 6 Foo

Input: A natural number n

```
proc Foo($n$):
    $i = 1$
    while $i \leqslant n$:
        if $i == 5$:
            $i = 0$
        $i = i + 1$
```

Exercise 2.2. Show how to reduce the number of questions to $3n - 4$ from $3n - 3$ in the celebrity problem as discussed in Section 2.1.1.

Exercise 2.3. The algorithm BinaryISqrt is iterative. Modify it to make it a recursive algorithm.

Exercise 2.4. Consider the following algorithm for determining x^n for non-negative n.

```
func POW($x$, $n$):
    if $n == 0$:
        return 1
    else:
        result = 1
        do $n$ times:
            result = result × $x$
        return result
```

(a) Give a recursive implementation of the above algorithm. Give pseudocode.

(b) What is the runtime of this algorithm? Justify your answer.

(c) Give a recursive algorithm that computes x^n with $\mathcal{O}(\log n)$ runtime.

(d) Give an iterative version of the $\mathcal{O}(\log n)$ algorithm.

Exercise 2.5. For each problem, rank the listed functions by order of growth. That is, find an arrangement $f_1, f_2, \ldots, f_k$ of the functions satisfying

$$\begin{gathered} f_1 = \mathcal{O}(f_2) \\ f_2 = \mathcal{O}(f_3) \\ \vdots \\ f_{k-1} = \mathcal{O}(f_k) \end{gathered}$$

Justify your ordering. Note that $\log^k n$ is the usual way of writing $(\log n)^k$.

(a) n^2, $\dfrac{n}{\log n}$, $n \log n$, $(1.001)^n$, $\dfrac{1}{n^2}$, $\log^{100} n$, $\dfrac{1}{\log n}$, $4^{\lg n}$, $n!$, $n^{\lg \lg n}$, $n^{1.6}$, $2^{\sqrt{\log n}}$

(b) $n^{\log \log n}$, n^n, $\sqrt{n}^{\frac{1}{\log n}}$, $e^{\log n}$, $\log(n!)$, $3^{(2 \log n)^{\frac{1}{3}}}$, $(\log n)!$, $\log^{10} n$

(c) n^3, $\dfrac{n}{\log^2 n}$, $n \log n, 1.1^n$, $\dfrac{1}{n^3}$, $\log^6 n$, $\dfrac{1}{n}$, $2^{\lg n}$, $n!$, $n^{\lg \lg n}$, $2^{\sqrt{\log n}}$, $n^{\frac{1}{\log n}}$

(d) $n^{1.5} + n$, $\dfrac{3n}{\log^3 n}$, $n \log^2 n$, 1.01^n, $\log^5 n$, $\dfrac{1}{n^2}$, $4^{\lg n}$, $n!$, $(\lg n)^{\lg n}$

Exercise 2.6.

(a) Show that given any polynomial

$$p(n) = \sum_{i=0}^{d} a_i n^{d-i} = a_0 n^d + a_1 n^{d-1} + \cdots + a_1 n + a_0$$

where a_i are constants and $a_0 > 0$, we have $p(n) = \Theta\left(n^d\right)$.

(b) Show that $n^{1000} = \mathcal{O}(2^n)$, but $n^{1000} \neq \Theta(2^n)$.
(Hint: Apply Theorem 2.2 and use Table 2.1)

Exercise 2.7. Prove or disprove each of the following statements:

i. If $f(n) = \Theta(g(n))$, then $n^{f(n)} = \Theta\left(n^{g(n)}\right)$

ii. For any positive function f, $f(n) = \Omega(f(n/2))$

Exercise 2.8. Show that the number of L-shaped tiles needed to cover a $2^n \times 2^n$ board is $(4^n - 1)/3$.

Exercise 2.9. Complete the solution (proposed in Section 2.3.1) to n-Queens by giving a solution for values of n that leave a reminder of 2 or 3 when divided by 6. (Hint: Solve for $n = 8$ and try to generalize.)

Exercise 2.10. The *Hamming Weight* of a number n is the number of 1s in its binary representation. Give a decrease-and-conquer algorithm (give pseudocode) for determining the hamming weight of n.

3 MATHEMATICAL INDUCTION AND ALGORITHMS

Before we delve deeply into the design and analysis of algorithms, we focus on a fundamental mathematical concept that is perhaps the single most useful mathematical tool that we will in use in this book to *analyze* algorithms and, in many cases, will also help in understanding the *design* of algorithms. That concept is *mathematical induction*. Mathematical induction is a powerful *proof technique* that is used to prove many mathematical theorems. However, as you will see throughout this book, mathematical induction is used many times to prove *correctness of algorithms* as well as to *analyze the performance of algorithms*. Mathematical induction is also useful in *designing algorithms*, and although this will not be immediately clear in some cases, it will become apparent after seeing the correctness proof and will lead to an appreciation of how designing algorithms and proving their correctness are intimately connected. If you are not familiar with mathematical induction, see Appendix A for a brief introduction.

3.1 Celebrity Problem Revisited

Let us recall The Celebrity Problem and the *decrease and conquer* algorithm proposed for its solution.

We start with n people. Choose any 2 people, say A and B, and ask whether A knows B. If yes, then A cannot be a celebrity and can be eliminated. If no, then B cannot be a celebrity and can be eliminated. This reduces the problem size by one. We repeat the same strategy among the remaining $n-1$ people. After $n-1$ questions, we will be left with one person (the *potential* celebrity), and if there is a celebrity, it must be this person.

Although it is intuitively clear that this strategy is correct, we will now formally *prove* the correctness of the algorithm using mathematical induction.

Theorem 3.1

The above strategy correctly finds a celebrity (assuming one exists) when applied to a group of n people.

Proof. As usual, the proof consists of showing the following two steps.

Base case: $n = 1$. Clearly, this is *vacuosly* true, since there is only one person.

Induction step: Assume that the strategy is correct for k people (this is the *induction hypothesis*). We will show that the strategy is correct for $k+1$ people. Applying the above strategy for $k+1$ people, we eliminate one person. It is clear that the eliminated person cannot be a celebrity. Thus, if there is a celebrity, then it has to be among the remaining k people. Hence the strategy is correct for $k+1$ people. □

Note that the above proof is fairly simple, but it crucially formalizes the correctness of the decrease and conquer strategy: showing the base case and showing the correctness of the decrease-and-conquer step is enough to show the correctness of the whole algorithm.

3.2 Tiling Problem Revisited

Let us recall the tiling problem and its divide and conquer strategy (Section 2.1.2). On lines very similar to the celebrity problem, it is easy to show the correctness of the strategy using mathematical induction. We leave this to the reader (see Exercise 3.3).

What is the complexity of this strategy? We can measure the complexity by the number of steps needed to cover the entire grid. The number of steps needed is equal to the number of tiles placed in this strategy.[1]

The number of tiles needed is the number of L-shaped tiles needed to cover a $2^n \times 2^n$ grid. How many such tiles are needed? Since we know that we can tile all the squares (except the missing square) using L-shaped tiles (each of which covers 3 squares), the number needed is $\frac{(2^n)(2^n)-1}{3} = \frac{4^n-1}{3}$. Let us prove this fact, using an *alternative* technique, namely using a *recurrence*, which is a powerful, general, and natural way to analyze divide and conquer (and more generally, recursive) algorithms This is discussed in more detail in Chapter 4.

We first define a function that captures the number of tiles needed. This function, obviously, depends on n, the size parameter. Let $T(n)$ denote the number of tiles needed to cover a $2^n \times 2^n$ grid (with one missing square). By the divide and conquer strategy, we reduced the problem to four subproblems each of size $2^{n-1} \times 2^{n-1}$ by adding one L-shaped tile. Hence we can write a equation for $T(n)$ in terms of $T(n-1)$ — this is called a recurrence:[2]

$$T(n) = 4T(n-1) + 1$$

Note that $T(n-1)$ denotes the number of tiles needed to cover a $2^{n-1} \times 2^{n-1}$ grid (with one missing square). The above recurrence says that the number of tiles needed to cover a $2^n \times 2^n$ grid is 4 times the number of tiles needed to cover a $2^{n-1} \times 2^{n-1}$ grid (because of the four subproblems of this size) plus one (because of the one tile placed in the center). The recurrence is similar to the induction step. Just like the base step of induction, recurrences need a base case; otherwise, they are not defined. We have $T(1) = 1$ which says that for a 2×2 grid we need only one L-shaped tile to cover it. Note how closely the recurrence mirrors the induction argument.

How to solve the above recurrence? One way to solve is to "unwind" the recurrence, i.e., write $T(n-1)$ in terms of $T(n-2)$ and write $T(n-2)$ in terms of $T(n-3)$ and so on and solve the sum directly (the next chapter has some examples of this techique).

[1] This is unlike backtracking strategy where we might place a tile and then later undo that placing, as discussed in Section 2.1.2.

[2] More generally, in a recurrence, $T(n)$ can be written in term of $T(k)$ where k is strictly smaller than n.

Another way is to use induction to solve the recurrence. For this we need to "guess" the solution first and then use induction to verify its correctness. We know that $\frac{4^n-1}{3}$ tiles are required, just by a divisibility argument. We next show that this is indeed the case.

Theorem 3.2

The solution to the recurrence $T(n) = 4T(n-1)+1$, with $T(1) = 1$ is $T(n) = \frac{4^n-1}{3}$.

Proof. We will prove this by induction on n.

Base case: $n = 1$. $T(1) = (4-1)/3 = 1$ which is true.

Induction step: assume that the statement is true for k. We will prove that it holds for $k+1$.

By the recurrence and the induction hypothesis,

$$\begin{aligned} T(k+1) &= 4T(k)+1 \\ &= 4\left(\frac{4^k-1}{3}\right)+1 \\ &= \frac{4^{k+1}-4+3}{3} \\ &= \frac{4^{k+1}-1}{3} \end{aligned}$$

which is exactly what we need to show. □

3.3 Worked Exercises

Worked Exercise 3.1. Consider the algorithm BinaryISqrt in Chapter 2. Prove that the following loop invariant holds at the beginning of every iteration of the **while** loop: $\texttt{low} \leqslant \lfloor\sqrt{n}\rfloor \leqslant \texttt{high}$.

Proof. The proof is via mathematical induction on the number of iterations. We first show that the invariant is true at the beginning of the first iteration — this is the *base case*. This is obviously true, since $\texttt{low} = 1$ and $\texttt{high} = n$ at the beginning.

We assume that the invariant is true for some iteration i (this is the *induction hypothesis*). *Under this assumption*, we show that it remains true for iteration $i+1$ (assuming, of course, the algorithm has not halted in iteration i). This is again easy to establish. If the algorithm has not terminated in iteration i, then there are two cases:

Case 1: $\texttt{mid}^2 \leqslant n$. In this case, clearly $\lfloor\sqrt{n}\rfloor$ should be larger than `mid`. Hence `low` is set to `mid`. By the induction hypothesis, $\lfloor\sqrt{n}\rfloor \leqslant \texttt{high}$, hence the invariant is true at the end of the iteration.

Case 2: $\texttt{mid}^2 > n$. In this case, clearly $\lfloor\sqrt{n}\rfloor$ must be smaller than `mid`. Hence `high` is set to `mid`. By the induction hypothesis, $\lfloor\sqrt{n}\rfloor \geqslant \texttt{low}$, thus the invariant is true at the end of the iteration.

Therefore the invariant is true at the end of iteration i, i.e., at the beginning of iteration $i+1$. □

Worked Exercise 3.2. This problem is a variant of Worked Exercise 2.2.

You are given an array $S[1, \dots, n]$ of n (distinct) numbers, where the numbers in the array are arranged in **increasing-increasing** order, i.e., $S[i] < S[i+1] < \cdots < S[n] < S[1] < S[2] < \dots S[i-1]$, for some *unknown* index i.

Give a recursive algorithm to **search** for a given element (say x) in the array S. If x is present in the array, the algorithm should return the index of the position where it is present, otherwise it should return 0.

Your algorithm should take $\mathcal{O}(\log n)$ comparisons. Give pseudocode of your algorithm, argue its correctness using mathematical induction, and show that your algorithm takes $\mathcal{O}(\log n)$ comparisons (in the worst case).

Solution. The algorithm is similar to the binary search algorithm but with modifications to account for the fact that the array is in increasing-increasing order which makes it a bit tricky. Another way to look at this order is that it is *circularly sorted*, i.e., take a sorted array and shift the elements in a circular fashion. The pseudocode is given in Algorithm 5. There are a couple of different approaches to search this order: one is to find the split, i.e., the maximum (or minimum) element in the array and search both the parts separately by the usual binary search, i.e., if $S[i]$ is the maximum element in the array, then search $S[1] < S[2] < \cdots < S[i]$ (first part) and $S[i+1] < S[i+2] < \cdots < S[n]$ (second part) separately since they are both sorted in the usual sense. This is conceptually simpler, but requires first finding where the split is, i.e., the index of the maximum element. Another approach is to search directly without finding the split.

Algorithm 7 CircularSearch
Input: An Increasing-Increasing Array S and an element x
Output: If x is in S, its index, else Null

```
func CircularSearch($S, x$):
▷ First find the index of the maximum element
    lo $= 1$    ▷ Low value of maximum
    hi $= n$    ▷ High value of maximum
    while lo $\neq$ hi:
        if lo $==$ hi $- 1$:    ▷ there are only two elements
            if $S$[lo] $>$ $S$[hi]:
                max $=$ lo
            else:
                max $=$ hi
        else:
            mid $= \lfloor$(lo $+$ hi)$/2\rfloor$
            if $S$[mid] $>$ $S$[lo]:    ▷ maximum is on the right of mid
                lo $=$ mid
            else:    ▷ maximum on the left of mid
                hi $=$ mid
    max $=$ lo    ▷ this is the index of max
    first $=$ BinarySearch($S[1,$ max$], x$)
    if first $\neq$ Null:
        return first
    if max $< n$:
        second $=$ BinarySearch($S[$max $+ 1, n], x$)
        if second $\neq$ Null:
            return second
    return Null
```

The correctness proof and analysis is very similar to the Binary-Search problem (Worked Exercise 2.2), except that there are 2 calls to Binary-Search each of which takes $\mathcal{O}(\log n)$ comparisons. The number of comparisons needed to find the *max* is $\mathcal{O}(\log n)$ as it also is like Binary-Search. Hence overall, $\mathcal{O}(\log n)$ comparisons.

□

3.4 Exercises

Exercise 3.1. Consider Worked Exercise 2.2. Prove the correctness of your algorithm by using mathematical induction.

Exercise 3.2. Consider the problem of *sorting* n numbers (see Section 5.1 for a formal definition): Given array S of n (distinct) numbers (in some arbitrary order), we want to arrange them in *increasing* order, i.e., at the end, we want to rearrange S such that $S[1] < S[2] < S[3] < \cdots < S[n]$. The only operation that we will allow to distinguish numbers is *comparison*, i.e., given two distinct array elements $S[i]$ and $S[j]$, we can ask whether $S[i] < S[j]$ or not. The goal is to design a sorting algorithm using as few comparisons as possible. Consider the following sorting algorithm known as *insertion sort*, which is described *inductively*. If $n = 1$ then the array is trivially sorted. Now consider an array of size $n > 1$. We sort the first $n-1$ elements of the array and then insert the $S[n]$ element in the array in the appropriate place in the sorted order of the first $n-1$ elements, by comparing $S[n]$ with the rest of the elements. Prove the correctness of this algorithm by using mathematical induction and analyze the number of comparisons needed in the worst case. Show how to use binary search to improve over the number of comparisons.

Exercise 3.3. Prove the correctness of the divide and conquer algorithm for The Tiling Problem given in Section 2.1.2 using mathematical induction.

Exercise 3.4. Prove the following by induction.

a) $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

b) $n^2 < 2^n$ for $n > 4$

c) $1 + 4 + 9 + \cdots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$

d) $3^n < n!$ for $n \geqslant 7$

Exercise 3.5. Suppose figurine A costs \$7 and figurine B costs \$10. You have n dollars and want to spend *all* of your money on figurines. Prove by mathematical induction that there are non-negative integers x and y such that, by purchasing x copies of figurine A and y copies of figurine B, all n dollars will be spent for any $n \geqslant 54$.

Hint: unlike a typical induction problem, there are many base cases here, and the first base case is $n = 54$, *not* $n = 0$.

Exercise 3.6. Consider the following proof by mathematical induction that all horses are the same color:

Base Case: one horse	Clearly, this is *vacuosly* true, since there is only one horse.
Induction hypothesis:	Assume this is true for $n-1$ horses.
Induction step:	Consider n horses. Let us number the horses 1 through n. By our induction hypothesis, horses 1 through $n-1$ must be the same color. But similarly, by our induction hypothesis, horses 2 through n must be the same color. Both of these sets of horses include horse 2, hence all n horses are of the same color.

What is wrong with the above proof?

Exercise 3.7. Using strong mathematical induction, prove that every integer can be expressed in binary. That is, show that, for any positive integer n, there exist integers a_0, a_1, …, a_k in $\{0, 1\}$ such that

$$n = \sum_{i=0}^{k} a_i 2^{k-i} = a_0 2^k + a_1 2^{k-1} + \cdots + a_{k-1} \cdot 2^1 + a_k \cdot 2^0$$

Exercise 3.8. Consider the following two-player game. There are stones placed on a table. Players alternate turns. On a player's turn, they must remove k stones for some $0 < k < n$ such that k divides n, where n is the number of remaining stones on the table. If there are no available moves for the player, that player loses.

Let us say the players are Alice and Bob and that Alice goes first. An example game that starts with 9 stones is:

n	Player	Move
9	Alice	3
6	Bob	2
4	Alice	1
3	Bob	1
2	Alice	1
1	Bob	$\emptyset$

In this example game, Bob loses. However, neither player played optimally!

Assuming both players play optimally, provide a rule that, given n, determines whether Alice wins. Prove the correctness of this rule via mathematical induction.

Hint: a move is always available except when $n = 1$.

4 RECURSION

Recursion is a fundamental paradigm in algorithm design which embodies a basic algorithmic strategy for solving problems — break a problem into smaller problems and use the *solutions from those subproblems* to solve the original. We saw several examples (such as the Celebrity problem) in the last chapter. Consider a problem of certain size. If the size is *small* (say a small constant, such as 1), then it is typically easy to solve directly. If, on the other hand, the problem size is large, then the strategy is to break the problem into smaller-sized *subproblem(s)* of the *same kind*. To solve these smaller subproblems, we *again* break them down into further subproblems of even smaller size, and so on. When this strategy is repeatedly applied, eventually the sizes of the problems will become small and then the problems can be solved directly. This can be naturally implemented as a recursive strategy, i.e., solving a problem by decomposing it into one or more subproblems each of smaller size and then recursively solving the subproblems. Recursion ensures that we only have to worry about the decomposition part and the part where the solutions of the subproblems need to be combined to get the solution to the original problem. Recursion "automatically" takes care of solving the subproblems. This is indeed a powerful strategy that works for many problems as we will see in this chapter and subsequent chapters, especially in implementing *divide and conquer* (Chapter 5) and *dynamic programming* (Chapter 6) techniques.

A *recursive* algorithm solves a problem by solving *one or more smaller instances* of the *same* problem. A *recursive function* is one that invokes *itself.* A recursive algorithm or function will always have a *termination condition*, otherwise the function will keep calling itself forever.

Recursion can be considered as a natural "algorithmic counterpart" to mathematical induction in the sense that their correctness and analysis can (usually) be shown straightforwardly by induction. Furthermore, recursive algorithms are naturally the preferred way to implement divide (or decrease) and conquer algorithmic strategy. Recursive algorithms are generally compact, elegant, and easier to show correctness, compared to non-recursive (i.e., iterative) algorithms.[1] Generally, all in all, it is a good idea to think first on a recursive strategy to solve a problem.

In principle, any algorithm can be expressed using recursive functions — in many

[1] Iterative algorithms, unlike recursive algorithms, typically have "loops" that do the non-trivial computation, e.g., **for** and **while** loops. In recursive algorithms, there will be typically no loops, but only recursive calls (and **if** statements).

cases, the recursive version is much simpler to write. In particular, divide/decrease and conquer algorithms can be naturally implemented using recursion. We will next look at several problems that illustrate recursion and divide and conquer technique.

4.1 Problem: Greatest Common Divisor (gcd)

Consider the gcd computation problem which is central to many algorithmic applications involving numbers.

Problem 4.1

Given two integers a and b find the greatest common divisor (gcd) of a and b, denoted by $\gcd(a, b)$.

We will design a *recursive* algorithm for finding a gcd of two given numbers. We use the following mathematical fact.

Theorem 4.1

The gcd of two integers a and b with $a > b$ is equal to the gcd of b and $a \bmod b$ (the remainder when a is divided by b). In other words, $\gcd(a, b) = \gcd(b, a \bmod b)$.

Proof. Assuming $a > b$, we can write:

$$a = bd + (a \bmod b)$$

where d is an integer (the quotient). That is, in other words,

$$a \bmod b = a - bd$$

Consider any number x that divides both a and b. x divides the right hand side of the above equation, since both a/x and bd/x are both integers. Hence, $(a \bmod b)/x$ is also an integer; thus x divides $a \bmod b$ as well. Thus any common divisor of a and b, including the $\gcd(a, b)$, also divides $a \bmod b$. Also, any common divisor y of b and $a \bmod b$ is a divisor of a. Hence the theorem follows. □

4.1.1 The Euclidean Algorithm

Theorem 4.1 suggests an algorithm to calculate gcd *recursively*. This algorithm, attributed to Euclid, dates back over 2000 years and is one of the the oldest algorithms known!

Algorithm 8 gcd – Greatest Common Divisor
Input: Integers a and b
Output: $\gcd(a, b)$

```
func GCD($a, b$):
    if $b == 0$:    ▷ Termination condition
        return $a$
    else:
        return GCD($b, a \bmod b$)    ▷ Recursive call
```

4.1.2 Recursive functions: Two key features

1. *Termination condition*: A recursive algorithm should have one or more termination conditions.

2. *Recursive call*: It should call *itself* on a *strictly smaller* value of its arguments.

The recursion flow diagram

Consider invoking Algorithm gcd on inputs $a = 8$ and $b = 5$. The sequence of recursive calls can be represented as a recursion flow diagram which, in this case, will be just a *path* (Figure 4.1).

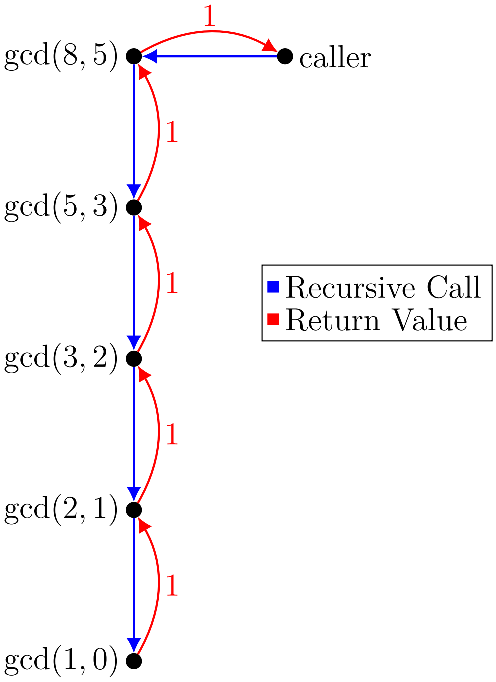


Figure 4.1: Recursion flow diagram for gcd

4.1.3 Termination, Correctness, and Running Time

Note that without a termination condition, the recursive algorithm may go on forever. The termination condition is sometimes called a "base (or basis) case" whose correctness is easy to check. In Algorithm gcd, it is clear that if $b = 0$, then a is the gcd (trivially). The algorithm's correctness follows from repeated application of Theorem 4.1 and by the termination condition. Formally, this can be shown by a straightforward induction proof, which we leave as an exercise for the reader.

How long does Algorithm gcd take? As discussed earlier, we measure the running time by the number of "costly" operations in the algorithm. The costly operation in the algorithm is the mod operation; it is easy to see that the overall running time is proportional to the number of such operations. The number of mod operations is also equal to the *number of recursive calls*. Hence we bound the number of recursive calls.

Given inputs a, b (assume $a \geqslant b > 0$), it can be shown that the *number of recursive calls* is at most $1+2\lceil\log_2(b)\rceil$. Thus the algorithm's running time is $\mathcal{O}(\log b)$ (the logarithm of the smaller of the two numbers). That is, the running time is essentially proportional to the *number of bits* of b.

Theorem 4.2

The number of recursive calls generated by Algorithm gcd when invoked on inputs a and b ($a \geqslant b$) is at most $1 + 2\lceil \log_2(b) \rceil$, i.e., $\mathcal{O}(\log b)$.

Proof: The first call is $\gcd(a, b)$. Henceforth, we show that the second argument in the call decreases by at least a factor of 2 in *at most two* recursive calls.

The subsequent recursive call is $\gcd(b, a \bmod b)$. Consider 2 cases:

- Case 1: $a \bmod b \leqslant b/2$. In this case, the second argument decreases by a factor of 2 by assumption.
- Case 2: $a \bmod b > b/2$. The next recursive call will be $\gcd(a \bmod b, b \bmod (a \bmod b))$. The second argument $b \bmod (a \bmod b) < b/2$, since $a \bmod b > b/2$. Thus in two calls the second argument has reduced by at least a factor of 2.

Thus, after at most every two calls we have the following changes to the second argument:

$$b \to (\leqslant b/2) \to (\leqslant b/4) \to (\leqslant b/8) \to \cdots \to 0$$

How many "steps" before the second argument (which is an integer) reaches 1? Let the number of steps be t. Then $b/2^t \leqslant 1$, i.e., $2^t \geqslant b$, and hence $t \geqslant \log_2(b)$. Thus, the number of steps is $\lceil \log_2(b) \rceil$. One more step is needed to reach 0. Hence the overall number of recursive calls is at most $1 + 2\lceil \log_2(b) \rceil = \mathcal{O}(\log b)$.

4.2 Problem: Searching for the maximum element

Let us consider another classic problem to illustrate the idea of recursion and its analysis.

Problem 4.2 ▶ Maximum

Given an array S of n elements, return the maximum value in S.

We can think of an array as an *ordered set (or list)* of elements.

4.2.1 A non-recursive algorithm

This algorithm simply compares each element with the current maximum in a sequential manner. This is called *sequential search*.

Algorithm 9 Max
Input: An array S
Output: The maximum value in S

```
func Max($S$):
    max $= -\infty$
    for $x$ in $S$:
        if max $< x$:
            max $= x$
    return max
```

In each step of the **for**-loop the current max is compared with the i-th element of S; if it is less then variable *max* is updated. Clearly, at the end of the **for**-loop, `max` contains the maximum number in S. Thus, Algorithm 9 takes $\mathcal{O}(n)$ comparisons(we will assume that comparison is the costly operation for this algorithm and hence this is a good measure of the running time. Nevertheless, in this example, even if you take the other operations into account, e.g., assignment etc., the overall time is $\mathcal{O}(n)$).

4.2.2 A Recursive Algorithm: the Divide and Conquer Strategy

There are many ways to design a recursive algorithm for finding the maximum element. One can adopt a decrease and conquer strategy whereby the size of the array (in the subproblem) is reduced by one in iteration. This strategy gives rise to one recursive subproblem (Can you see how?).

Alternatively, we adopt a divide and conquer strategy has typically 3 parts: *Split*, *Solve (recursively)*, and *Combine*.

- *Split* the problem into one or more **smaller independent** subproblems. Note that each subproblem should be of the same kind as the original problem, but of strictly smaller size.
- *Recursively solve* the subproblems.
- *Combine* the solutions of the subproblems to get the solution for the original problem.

Divide and conquer strategy for finding max:

- *Split*: Split the given set S into two (almost) *equal* parts (first half and second half).[2]
- *Solve*: Recursively find the maximum of the first part and the second part — let these be $\texttt{max}_1$ and $\texttt{max}_2$ respectively.
- *Combine*: Output the maximum of S as the maximum of $\texttt{max}_1$ and $\texttt{max}_2$.

[2] In many divide and conquer problems, the input is typically split into two (almost) equal *disjoint* parts giving rise to two independent problems; thus this is a good strategy to try for divide and conquer problems.

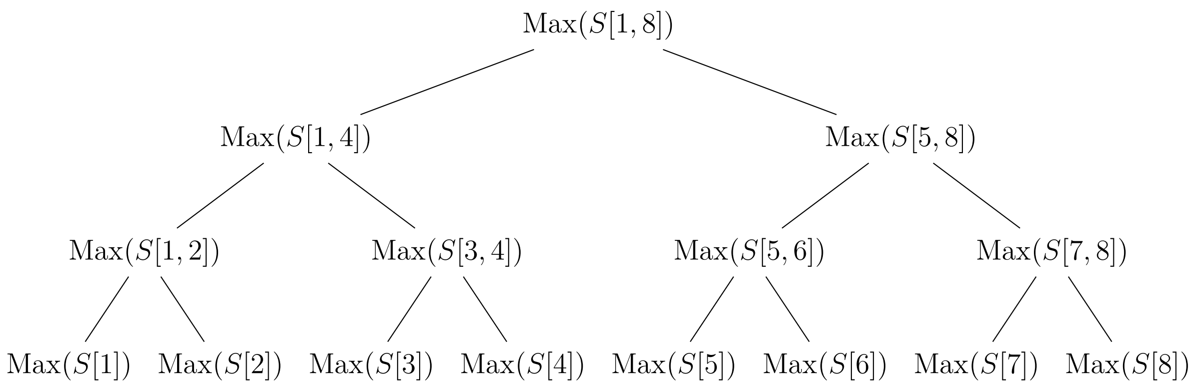


Figure 4.2: Recursion flow diagram for Maxr

A Divide and Conquer Algorithm

Algorithm 10 Recursive-Max – Recursive Max Function
Input: An array S and indices a and b
Output: Max value in S between indices a and b

```
func RECURSIVE-MAX($S$, $a$, $b$):
    if $a == b$:
        return $S[a]$
    $m = \lfloor (a+b)/2 \rfloor$    ▷ middle point
    max$_1$ = RECURSIVE-MAX($S, 1, m$)    ▷ first part
    max$_2$ = RECURSIVE-MAX($S, m+1, b$)    ▷ second part
    if max$_1$ > max$_2$:
        return max$_1$
    else:
        return max$_2$
```

To solve the original problem of finding the maximum element in $S[1 \dots n]$, we call RECURSIVE-MAX$(S, 1, n)$.

The recursion "flow diagram"

Consider invoking Algorithm Recursive-Max on input S of size 8.

The sequence of recursive calls can be represented as a "flow diagram" which will be a *tree*. This is usually called a *Recursion Tree*. See Figure 4.2.

4.2.3 Correctness of Algorithm Maxr

It is easy to establish the correctness of the above algorithm by using *mathematical induction*. The induction is on the *size* of the input set S.

This is done in two steps.

Proof by Mathematical Induction

1. *Base case*: This is the trivial case when the size of S is 1. Clearly, the algorithm is correct, since it outputs this single element.

2. *Induction step*: Assume that the algorithm is correct for *all* sets of size less than n. This is called the Induction hypothesis.

Establishing the Induction Step

Assuming the induction hypothesis, we will show that the algorithm is also correct for an input of n elements, i.e., that Recursive-Max$(S, 1, n)$ is the maximum of S when $|S| = n$. By the induction hypothesis, $\texttt{max}_1$ and $\texttt{max}_2$ are the correct maximum values of the sets $S[a \dots m]$ and $S[m+1, b]$ respectively (since they are of size less than n). Hence the maximum of $\texttt{max}_1$ and $\texttt{max}_2$ must be the maximum of $S[1 \dots n]$.

Running Time

How long does Algorithm Recursive-Max take? As usual, we will measure the running time by the total number of costly operations. The costly operation for this algorithm is the *comparison* operation. We can see that all other operations in the algorithm, e.g., finding the middle point etc., are proportional to the number of comparison operations. There is one comparison operation per recursive call, hence the running time is determined by the *total number of recursive calls.* The number of recursive calls is equal to the *size of the recursion tree*, i.e., the *number of nodes in the tree.*

In our example, when input is $S[1 \dots 8]$, the size of the tree is $15 = 1 + 2 + 4 + 8 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$. Generalizing the above: when the input is $S[1 \dots 2^k]$, the size of the tree $= 2^0 + 2^1 + 2^2 + \dots 2^k = 2^{k+1} - 1 \leqslant 2(2^k) = 2n = \mathcal{O}(n)$, where $n = 2^k$ is the size of the array. Thus the running time is *proportional to the size of the array*, i.e., $\mathcal{O}(n)$.

Analyzing using a Recurrence

Recurrence relations are a powerful way to analyze recursive algorithms. This is especially the case with divide/decrease and conquer algorithms. Let us analyze Algorithm Recursive-Max using a recurrence. Instead of counting recursive calls, we will count the number of comparisons. Both are equivalent ways (asymptotically) to bound the run time.

Let the function $T(n)$ denote the number of comparisons required by Recursive-Max on an input array of size n.

Then we can write the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

We also need a “base” case:

$$T(1) = 0$$

How to solve such a recurrence?

Unwinding the recurrence

For convenience, we will assume that n is a power of 2, i.e., $n = 2^k$, for some k. This assumption will not change the asymptotic result.[3] Then:

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + 1 = 2(2T(2^{k-2}) + 1) + 1 \\ &= 2^2T(2^{k-2}) + 2 + 1 \\ &= 2^3T(2^{k-3}) + 2^2 + 2 + 1 = 2^kT(1) + 2^{k-1} + \cdots + 1 \\ &= 2^{k-1} + 2^{k-2} + \cdots + 2 + 1 = \sum_{i=0}^{k-1} 2^i \end{aligned}$$

The above sum is obtained by using the formula for the sum of a *geometric* series:

$$\sum_{i=0}^{k-1} x^i = 1 + x + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}$$

where, x is called the (common) ratio.

Hence,

$$T(n) = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

Hence,

$$T(2^k) = 2^k - 1 = n - 1 = \mathcal{O}(n).$$

4.3 Techniques for Solving Recurrences

4.3.1 Guess and Verify

Alternative to the “unwinding” technique discussed above, another powerful way to solve recurrences is the *“guess and verify”* method. This uses *mathematical induction* to guess the correct solution to the recurrence and then prove that it is indeed correct, by using induction. Let us solve the recurrence for Algorithm Recursive-Max using this method.

The “real” recurrence is:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$$

Guess $T(n) = \mathcal{O}(n)$. Let us use **induction** to show that $T(n) \leqslant cn$, for some (suitable) constant c.

Induction hypothesis: Assume that $T(k) \leqslant ck$ for all $k < n$. Plugging the hypothesis in the recurrence gives:

$$\begin{aligned} T(n) &\leqslant c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\ &= cn + 1 \end{aligned}$$

which does not prove our hypothesis (note that it should be $T(n) \leqslant cn$).

[3] The reason is as follows. If n is not a power of 2, one can sandwich n between two successive powers of 2, i.e., $2^k < n < 2^{k+1}$ and then apply the argument for 2^k and 2^{k+1} which are away by a factor of 2 from n. Since $T(n)$ is an increasing function of n, we have $T(2^k) \leqslant T(n) \leqslant T(2^{k+1})$ and thus: $2^k - 1 \leqslant T(n) \leqslant 2^{k+1} - 1$.

Revising our Guess

We revise our guess to: $T(n) \leqslant cn - b$ and show this by induction. Not that this is called *strengthening* the guess (or the hypothesis), i.e., we now try to show a stronger bound. Strengthening (as opposed to weakening) the hypothesis might seem counterintuitive; but, in fact, this strengthened hypothesis allows the induction proof to go through.

(New) Induction hypothesis: Assume that $T(k) \leqslant ck - b$ for all $k < n$. Then

$$\begin{aligned} T(n) &\leqslant \left(c\left\lfloor\frac{n}{2}\right\rfloor - b\right) + \left(c\left\lceil\frac{n}{2}\right\rceil - b\right) + 1 \\ &= cn - 2b + 1 \\ &\leqslant cn - b \end{aligned}$$

if $b \geqslant 1$.

How large should c and b be? This depends on the “base” case of the recurrence.

In Recursive-Max, $T(1) = 0$. Hence, we can choose $c = 1$ and $b = 1$, since it satisfies the base case. Thus, we have shown $T(n) \leqslant n - 1$ for all n.

Another Example

Example 4.1. Solve the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Guess and verify that $T(n) \leqslant cn \log n$, for some constant c. *Induction hypothesis*: Assume that $T(k) \leqslant ck \log k$ for all $k < n$.

Hence we have:

$$\begin{aligned} T(n) &\leqslant 2\frac{cn}{2} \log\left(\frac{n}{2}\right) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leqslant cn \log n \end{aligned}$$

if $c \geqslant 1$. Note that c should be chosen large enough to satisfy the base condition.

A Wrong Argument using Induction

If you try to show $T(n) \leqslant cn$ by arguing:

$$T(n) \leqslant 2\frac{cn}{2} + n = (c+1)n = \mathcal{O}(n).$$

This is incorrect! The reason is, as mentioned before, we have to show $T(n) \leqslant cn$, *not*, $\leqslant (c+1)n$.

Solving Recurrences via Change of Variables

Example 4.2. Solve

$$T(n) = 2T(\sqrt{n}) + 1$$

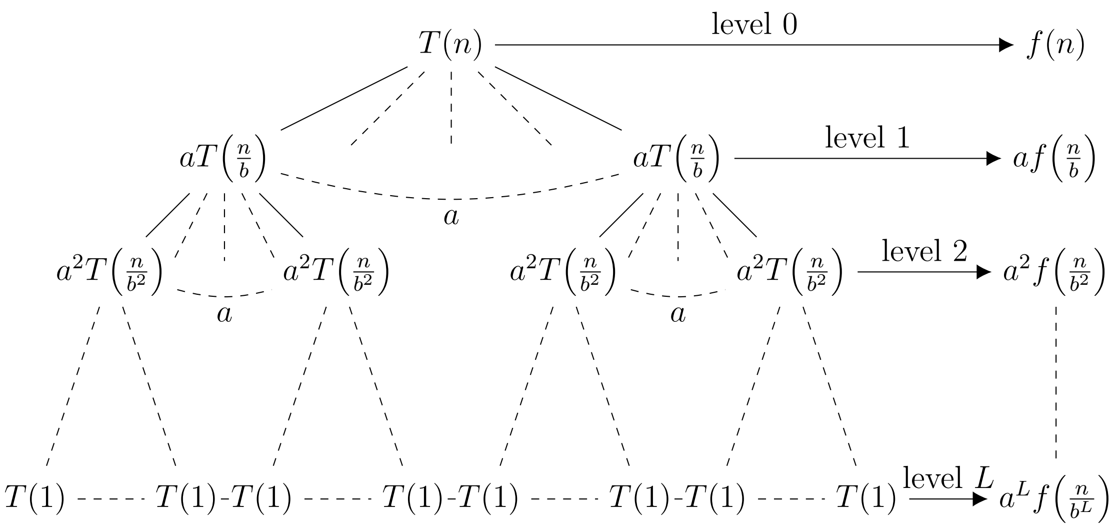

$T(n)$
level 0
$f(n)$
$aT\left(\frac{n}{b}\right)$
$aT\left(\frac{n}{b}\right)$
level 1
$af\left(\frac{n}{b}\right)$
a
$a^2T\left(\frac{n}{b^2}\right)$
$a^2T\left(\frac{n}{b^2}\right)$
$a^2T\left(\frac{n}{b^2}\right)$
$a^2T\left(\frac{n}{b^2}\right)$
level 2
$a^2f\left(\frac{n}{b^2}\right)$
a
a
$T(1)$
$T(1)$-$T(1)$
$T(1)$-$T(1)$
$T(1)$-$T(1)$
$T(1)$
level L
$a^Lf\left(\frac{n}{b^L}\right)$

Figure 4.3: Illustration for proof of The DC Recurrence Theorem

Renaming $m = \log n$, we have:

$$T(2^m) = 2T(2^{m/2}) + 1$$

Again, renaming $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + 1$$

Thus, $S(m) = \mathcal{O}(m)$ and $T(n) = T(2^m) = S(m) = \mathcal{O}(m) = \mathcal{O}(\log n)$.

4.3.2 A General Theorem for "Divide and Conquer" Recurrences

We now give a formula that can quickly solve certain types of recurrences. This type occurs most commonly in analysis of divide and conquer type algorithms and thus very useful.

Consider recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geqslant 1$ and $b > 1$ are constants and $f(n)$ is some function. Assume, without loss of generality, that n is a power of b, i.e., $n = b^L$ for some $L \geqslant 0$. (It can be shown that the asymptotic bounds as shown in The DC Recurrence Theorem below will hold, even when n is not a power of b. Exercise 4.3 asks you to show it.) Assume that $T(1) = \mathcal{O}(1)$, i.e., the time for solving an instance of size 1 is a constant. Under these assumptions, we prove the following theorem.

Theorem 4.3 ▶ The DC Recurrence Theorem

Let $T(n) = aT(n/b) + f(n)$ where $a \geqslant 1$ and $b > 1$ are constants and $f(n)$ is some function. Let $T(1) = \Theta(1)$. Suppose there exists a constant c such that $af(n/b) = cf(n)$. Then the solution for the above recurrence $T(n)$ is:

1. If $c < 1$ then $T(n) = \Theta(f(n))$.
2. If $c > 1$ then $T(n) = \Theta\left(n^{\log_b a}\right)$.
3. If $c = 1$ then $T(n) = \Theta(f(n) \log_b n)$.

Proof. Draw a "recursion tree" for $T(n)$: $T(n)$ stands for the root node (level 0) and it has a children each of which is a root of the recursion tree for $T(n/b)$. One can associate a value with each node: the value is the cost that is paid at that level to go to the next recursive level. The value associated with the root node $T(n)$ is $f(n)$ (this is the cost paid for going to the next level, i.e., level 1). A recursion tree is a complete a-ary tree (i.e., all levels are full — this follows from the assumption that n is a power of b), where each node at level (depth) i has the value $f(n/b^i)$. The total value of all nodes in level i is $a^i f(n/b^i)$. See Figure 4.3 for an illustration. The leaves of the tree contains the "base cases" of the recursion. By our assumption, $T(1) = f(1) = \Theta(1)$. Thus we have the "total cost" of the recursion which is the sum of all the values (costs) in all levels:

$$T(n) = f(n) + af(n/b) + a^2 f(n/b^2) + \cdots + a^L f(n/b^L)$$

where L is the depth of the recursion tree. Note that $n/b^L = 1$.

Substituting, $af(n/b) = cf(n)$ in the above formula for $T(n)$,[4] we have

$$T(n) = f(n) + cf(n) + c^2 f(n) + \cdots + c^L f(n)$$

Note that $L = \log_b n$ and since $f(1) = \Theta(1)$, $a^L f(n/b^L) = \Theta\left(a^{\log_b n}\right) = \Theta\left(n^{\log_b a}\right)$.

1) If $c < 1$, then $T(n)$ is a geometric series (see Appendix G) with largest term $f(n)$. Hence $T(n) = \Theta(f(n))$.

2) If $c > 1$, then $T(n)$ is a geometric series with largest term $a^L f(n/b^L) = \Theta\left(n^{\log_b a}\right)$.

3) If $c = 1$, then there are $L + 1$ levels each level summing to $f(n)$ and hence $\Theta(f(n) \log_b n)$.

□

Example 4.3.

1. $T(n) = T(3n/4) + 2n$. Here $af(n/b) = 2(3n/4) = (3/4)f(n)$. Hence $T(n) = \Theta(n)$.
2. $T(n) = 7T(n/2) + \Theta(n^2)$. That is, $T(n) = 7T(n/2) + c_1 n^2$, for some positive constant c_1. $af(n/b) = 7c_1(n/2)^2 = (7/4)c_1 n^2 = (7/4)f(n)$. Hence, $T(n) = \Theta\left(n^{\log_2 7}\right) = \mathcal{O}(n^{2.81})$.
3. MergeSort: $T(n) = 2T(n/2) + n$. Here $af(n/b) = f(n)$ and hence $T(n) = \Theta(n \log n)$.

[4] Note that, we can write $a^2 f(n/b^2) = a(af(n/b^2)) = a(cf(n/b)) = c(af(n/b)) = c^2 f(n)$. Similarly, we can infer that $a^i f(n/b^i) = c^i f(n)$, for any i.

Other Examples

Some recurrences do not exactly fit The DC Recurrence Theorem.

Example 4.4. $T(n) = 2T(n/2) + n/\log n$

We cannot apply The DC Recurrence Theorem directly since $af(n/b) = n/(\log n - 1)$ is not equal to a constant factor times $n/\log n$. We compute the recurrence directly.

The sum of the nodes in the ith level is $n/(\log n - i)$. Thus, the depth of recursion is at most $\log n - 1$.

$$T(n) = \sum_{i=0}^{\log n-1} n/(\log n - i) = \sum_{j=1}^{\log n} n/j = nH_{\log n} = \Theta(n \log\log n)$$

where $H_k = \sum_{i=1}^{k} 1/i$ is the *harmonic function* and $H_k = \Theta(\log k)$.

Example 4.5. $T(n) = T(3n/4) + T(n/4) + n$.

We can solve this by directly looking at the recursion tree for this recurrence. Each complete level in the tree adds up to n and each leaf has depth between $\log_4 n$ and $\log_{4/3} n$. Hence, $T(n) = \Theta(n \log n)$.

4.3.3 Handling Floors and Ceilings in Recurrences*

Suppose we have the following recurrence (this arises in MergeSort, as we will see later)

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

We want to show that $T(n) = \Theta(n \log n)$.

We show the upper bound, lower bound is similar.

$$T(n) \leqslant 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \leqslant 2T\left(\frac{n}{2} + 1\right) + n$$

To solve for $T(n)$ we do a "domain transformation" as follows. Let $S(n) = T(n+\alpha)$ where α is a constant chosen such that

$$S(n) \leqslant 2S\left(\frac{n}{2}\right) + \Theta(n)$$

To find α, we compare the two recurrences — that of $T(n)$ and $S(n)$.

$$T(n+\alpha) \leqslant 2T\left(\frac{n+\alpha}{2} + 1\right) + n$$

$$T(n+\alpha) \leqslant 2T\left(\frac{n}{2} + \alpha\right) + \Theta(n)$$

which gives $\alpha = 2$.

By The DC Recurrence Theorem, $S(n) = \mathcal{O}(n \log n)$, and hence $T(n) = S(n - \alpha) = \mathcal{O}(n \log n)$.

4.4 Worked Exercises

Worked Exercise 4.1. A light bulb is controlled by n switches. Each switch can be in one of two possible states — “OFF” and “ON”. We cannot tell the status of a switch (i.e., whether it is ON or OFF) at any given time. We can only toggle it between the two states by pressing the switch. Only when **all** switches are in the ON state, the light bulb will glow, otherwise it will not glow. Given that the light bulb is not initially glowing, give a **recursive** algorithm to make it glow. What is the number of operations (as a function of n) needed by your algorithm in the worst case? (Count each toggle of a switch as one operation.)

Solution. As usual, we will think recursively. We look at the base case, i.e., $n = 1$, of glowing the bulb with one switch. In this case, it takes one operation, simply toggle the switch if the bulb is not glowing. Now consider the recursive step where we have $n > 1$ switches. We can reduce this to a problem glowing $n - 1$ switches; with the nth switch taking two possibilities. The recursive function *glow*(.) is as follows. The pseudocode is given in Algorithm 11. Note the recursive calls — there are two recursive calls to glow($S[1, \ldots, n - 1]$) which is needed; the first one for one position of switch $S[n]$ and other after toggling $S[n]$ (if it was not glowing after the first recursive call). Intuitively what this is doing is that fixing one position of $S[n]$ it explores all possibilities of the first $n - 1$ switches.

The number of operations (toggling) can be computed by a recurrence. Let $T(n)$ be the number of operations needed to glow a bulb with n switches. Then from the algorithm, we have: $T(1) = 1$ and for $n > 1$, $T(n) = 2T(n - 1) + 1$. This can be solved by unwinding the recurrence as follows:

$T(n) = 2T(n-1)+1 = 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1 = 2^3T(n-3)+2^2+2+1$ and so. Thus, continuing the pattern:

$$\begin{aligned} T(n) &= 2^{n-1}T(n - (n - 1)) + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1 \end{aligned}$$

by using the formula for the geometric series (with common ratio 2).

Algorithm 11 Glow – Recursively toggle switches to make lightbulb glow
Input: An array S of n switches

```
proc GLOW($S$):
    if $n == 1$:
        if bulb is not glowing:
            Toggle switch $S[1]$
    else:
        GLOW($S[1, \ldots, n-1]$)    ▷ recursive call
        if bulb is not glowing:
            Toggle switch $S[n]$    ▷ one toggle operation
            GLOW($S[1, \ldots, n-1]$)    ▷ recursive call
```

□

Worked Exercise 4.2. Consider the problem of finding the largest and second largest elements from a set of n elements.

a) Give a simple iterative algorithm that takes no more than $2n - 3$ comparisons.

b) Give a divide and conquer algorithm and show that it takes $\frac{3n}{2} - 2$ comparisons when n is a power of 2.

c) Show how you can find both these elements in $n + \log n - 2$ comparisons in the worst case (assume n is a power of 2). (Hint: See hint in Worked Exercise 2.3.)

Solution.

a) Find the max by a simple iterative algorithm (see Section 4.2.1 of the book). Then repeat the same, by finding the max among the rest of the elements — this will be the second max. The total number of comparisons is $n - 1 + n - 2 = 2n - 3$.

b) A divide and conquer algorithm: Divide the elements into two groups. Find the largest and second largest elements from each group. Then find the largest and second largest from these four elements. To find the largest and second largest from each group, proceed recursively.

   A divide and conquer algorithm is given in Algorithm Find-Two-Largest.

Algorithm 12 Find-Two-Largest – Find 2 largest elements in array
Input: An array A of length n
Output: Two largest numbers in A

```
func FIND-TWO-LARGEST($A$, left, right):
    if left == right:    ▷ Only one element
        return ($A$[left], Null)
    if left + 1 == right:    ▷ two elements
        if $A$[left] > $A$[right]:
            largest = $A$[left]
            second_largest = $A$[right]
        else:
            largest = $A$[right]
            second_largest = $A$[left]
        return (largest, second_largest)
    middle = $\lfloor$(left + right)/2$\rfloor$
    ▷ Divide the input and solve the problem for each subset.
    $(p_1, p_2)$ = FIND-TWO-LARGEST($A$, left, middle)
    $(q_1, q_2)$ = FIND-TWO-LARGEST($A$, middle + 1, right)
    ▷ Combine the results
    if $p_1 > q_1$:
        largest = $p_1$
        if $p_2 > q_1$:
            second_largest = $p_2$
        else:
            second_largest = $q_1$
    else:
        largest = $q_1$
        if $q_2 > p_1$:
            second_largest = $q_2$
        else:
            second_largest = $p_1$
    return (largest, second_largest)
```

Analysis: Let $T(n)$ be the number of comparisons needed for Find-Two-Largest, when there are n elements. Assuming n is a power of 2, number of comparisons to solve each sub-problem is $T(n/2)$ and exactly 2 comparisons for combining the result. Thus

$$T(n) = 2T(n/2) + 2$$

Boundary condition: with two elements number of comparisons is $T(2) = 1$.
By induction on n, we show that $T(n) = 3n/2 - 2$.

1) Base case: for $n = 2$, $T(n) = 3(2/2) - 2 = 1$.

2) Induction step: for $n > 2$

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(3(n/2)/2 - 2) + 2 \\ &= 3n/2 - 2 \end{aligned}$$

Alternate way to do this is to simply "unwind" the recurrence, i.e., use the recursion tree method.

c) The algorithm consists of two phases. In the first phase, we find the largest element by comparing all n elements. In the second phase, we find the second largest element, by comparing a particular subset of the elements (as described below).

For the first phase, compare the elements pairwise. From each pair, eliminate the smaller element and keep the larger element. If the number of elements is odd, there is an unpaired element; keep the unpaired element. Thus we are left with $\lceil n/2 \rceil$ elements. Repeat the process on the remaining $\lceil n/2 \rceil$ elements that leaves us with $\lceil n/4 \rceil$ elements and so on. After exactly $\lceil \lg n \rceil$ steps, we are left with exactly one element, which is the largest of the n elements. In these $\lceil \lg n \rceil$ steps, we make exactly $n - 1$ comparisons. Because, each comparison eliminates one element and exactly $n-1$ elements are eliminated. Notice that the second largest element can be eliminated only when it is compared to the largest element. Thus to find the second largest element, we need to check only the elements, say the *candidate elements*, that are compared with the largest element in the first phase.

To facilitate tracking of the candidate elements, in addition to the pairwise comparison in the first phase, build a binary tree. Initially, each element constitutes a leaf of the tree. When two elements a and b are compared, create an internal node having children a and b, and label it with the larger of a and b. For an unpaired element c, create a node having only one child c and label it also with c. The root of the tree is labeled with the largest element. The candidates for the second largest can be found by tracking the path of the largest element back from the root to the leaf labeled by the largest element. There are at most $\lceil \lg n \rceil$ candidates requiring at most $\lceil \lg n \rceil - 1$ comparisons to get the second largest element. The total number of comparisons $\leqslant (n-1) + (\lceil \lg n \rceil - 1) = n + \lceil \lg n \rceil - 2$.

□

Worked Exercise 4.3. Solve the following recurrences. Give the answer in terms of *Big-Theta* notation. Solve up to constant factors, i.e., your answer must give the correct function for $T(n)$, up to constant factors. You can assume constant base cases, i.e., $T(1) = T(0) = c$, where c is a positive constant. You can ignore floors and ceilings. You can use The DC Recurrence Theorem if it applies.

a) $T(n) = 3T(n/2) + n$

b) $T(n) = 2T(n-1) + 2$

c) $T(n) = 4T(n/2) + n^3$

d) $T(n) = T(3n/4) + T(n/4) + n$

e) $T(n) = T(7n/8) + n$

Solution.

a)

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + n \\ 3\left(\frac{n}{2}\right) &= \frac{3}{2}n \end{aligned}$$

hence $c = \frac{3}{2} > 1$. Thus, $T(n) = \Theta\left(n^{\log_2 3}\right)$, by The DC Recurrence Theorem.

b)

$$T(n) = 2T(n-1) + 2$$

Unwinding the recurrence, we have

$$\begin{aligned} T(n) &= 2(2T(n-2) + 2) + 2 \\ &= 2^2T(n-2) + 2^2 + 2 \\ &= 2^2(2T(n-3) + 2) + 2^2 + 2 \\ &= 2^3T(n-3) + 2^3 + 2^2 + 2 \end{aligned}$$

and so on.

Continuing the pattern, we see that

$$T(n) = 2^{n-1}T(n-(n-1)) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2$$

Assuming $T(1) = c$, where c is a positive constant, we have

$$T(n) = 2^{n-1}c + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2$$

Applying the formula for the geometric series:

$$\begin{aligned} 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 &= 2(1 + 2 + \cdots + 2^{n-2}) \\ &= 2(2^{n-1} - 1) \\ &= 2^n - 2 \end{aligned}$$

Hence $T(n) = 2^{n-1}c + 2^n - 2 = \Theta(2^n)$.

c) $T(n) = 4T(n/2) + n^3$.

Applying The DC Recurrence Theorem:

$4(n/2)^3 = (4/2^3)n^3$, hence $c = 4/2^3 = 4/8 < 1$.

Thus $T(n) = \Theta(n^3)$.

d) $T(n) = T(3n/4) + T(n/4) + n$.

There are a couple of different ways to do this:

First way: Build a recursion tree for $T(n)$. The root has two children $T(3n/4)$ and $T(n/4)$ with a total cost of n. Continuing this, it is easy to see that at every level the total cost is n. The number of levels is at most $\log_{4/3} n$ since the longest path from root to leaf is via the leftmost path, i.e., path with 3/4-fraction — this is because that this is the path where the size of the subproblem reduces the least. Hence total cost, $T(n)$, is $\Theta(n \log n)$.

Second way: Guess the solution is $\Theta(n \log n)$ and then verify this by induction.

First, we show the upper bound. Assume that, for some constant $a > 0$, $T(k) \leqslant ak \log k$ for all $k < n$ (induction hypothesis). Note that the base case is easily true, by choosing a large enough (compared to $T(1)$ and $T(0)$). Next we show that the statement is true for $k = n$, i.e., $T(n) \leqslant an \log n$. Then

$$\begin{aligned} T(n) &\leqslant a(3n/4)\log(3n/4) + a(n/4)\log(n/4) + n \\ &= a(3n/4)(\log n + \log(3/4)) + a(n/4)(\log n + \log(1/4)) + n \\ &= \log(n)(3an/4 + an/4) - a(3n/4)\log(4/3) - a(n/4)\log(4) + n \\ &= an\log n - a(3n/4)\log(4/3) - a(n/4)\log(4) + n \\ &= an\log n - an(3/4\log(4/3) + 1/4\log(4)) + n \leqslant an\log n \end{aligned}$$

if a is chosen sufficiently large.

Now we show the lower bound. Assume that, for some constant $a > 0$, $T(k) \geqslant ak \log k$ for all $k < n$ (induction hypothesis). Note that the base case is easily true, by choosing a small enough. Next we show that the statement is true for $k = n$, i.e., $T(n) \geqslant an \log n$. Then, we have:

$$\begin{aligned} T(n) &= T(3n/4) + T(n/4) + n \geqslant a(3n/4)\log(3n/4) + a(n/4)\log(n/4) + n \\ &= a(3n/4)(\log n + \log(3/4)) + a(n/4)(\log n + \log(1/4)) + n \\ &= \log(n)(3an/4 + an/4) - an(3/4)\log(4/3) - an(/4)\log(4) + n \\ &= an\log n - an((3/4)\log(4/3) + (1/4)\log(4)) + n \geqslant an\log n \end{aligned}$$

if a is chosen sufficiently small.

e) (5 points) $T(n) = T(7n/8) + n$

Applying The DC Recurrence Theorem:

$7n/8 = (7/8)(n)$, hence $c = 7/8 < 1$.

Thus $T(n) = \mathcal{O}(n)$.

□

4.5 Exercises

Exercise 4.1. Consider the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Assume that $T(n) \leqslant c$ for all $n < n_0$ for some constants c and $n_0 > 0$.

(a) Show by mathematical induction that $T(n) = \mathcal{O}(n^2 \log n)$.

(b) Show by unwinding the recurrence that $T(n) = \mathcal{O}(n^2 \log n)$.

(c) Finally, show by the Theorem 4.3 that $T(n) = \Theta(n^2 \log n)$.

Exercise 4.2. Prove the asymptotic bound for the following recurrences by using induction. Assume that base cases of all the recurrences are constants i.e., $T(n) = \Theta(1)$, for $n < c$ where c is some constant.

(a) $T(n) \leqslant 2T(n/2) + n^2$. Then, $T(n) = \mathcal{O}(n^2 \log n)$.

(b) $T(n) \leqslant T(5n/6) + \mathcal{O}(n)$. Then, $T(n) = \mathcal{O}(n)$.

(c) $T(n) = T(5n/6) + T(n/6) + n$. Then, $T(n) = \mathcal{O}(n \log n)$.

Exercise 4.3. Show the asymptotic bounds in The DC Recurrence Theorem hold even when n is not a power of b. (Hint: Sandwich n between two successive powers of b, i.e., let $b^k \leqslant n \leqslant b^{k+1}$. Then proceed as in the proof of The DC Recurrence Theorem.)

Exercise 4.4. Solve the following recurrences. Give the answer in terms of *Big-Theta* notation. Solve up to constant factors, i.e., your answer must give the correct function for $T(n)$, up to constant factors. You can assume constant base cases, i.e., $T(1) = T(0) = c$, where c is a positive constant. You can ignore floors and ceilings. You can use The DC Recurrence Theorem if it applies.

1. $T(n) = 3T(n/2) + n$
2. $T(n) = 4T(n/2) + n \log n$
3. $T(n) = \sqrt{n}T(\sqrt{n}) + n$
4. $T(n) = T(n/2) + T(n/3) + 1$
5. $T(n) = 2T(n-1) + 2$
6. $T(n) = 4T(n/2) + n^3$
7. $T(n) = T(n-1) + 2^n$
8. $T(n) = 2T(n/2) + n/\log n$
9. $T(n) = 4T(n/3) + n^2$
10. $T(n) = T(3n/4) + T(n/4) + n$
11. $T(n) = T(7n/8) + n$
12. $T(n) = T(3n/4) + \mathcal{O}(n)$

Exercise 4.5. You are given an array consisting of n numbers. A popular element is an element that occurs (strictly) **more than** $n/2$ times in the array. Give an algorithm that finds the popular element in the array if it exists, otherwise it should output “NO”. Your algorithm should take no more than $2n$ comparisons. (As usual, we only count the comparisons between array elements.) Give pseudocode, argue its correctness, and show that your algorithm indeed takes no more than $2n$ comparisons. (Hint: Use a decrease and conquer strategy, similar to the celebrity problem.)

Exercise 4.6. There are n marbles placed in a line. A marble's color is either red or blue. The colors can appear in any arbitrary manner (e.g., they can all be red or blue, or some may be red and the others blue). The goal is to remove all the marbles by a sequence of moves. In each move, a red marble can be removed; at the same time, we replace its neighbors (if any) by marbles of *opposite* color (i.e., Red becomes Blue and vice versa). A marble is a neighbor of another if it is adjacent to it in the original line (two marbles are not neighbors if there are “gaps' between them, which can occur when in-between marbles are removed). To illustrate, see the following example.

Let $\texttt{R B B B R B R B B}$ be the initial arrangement of marbles. Then the successive steps are shown as follows (a removed marble is shown as $\texttt{X}$):

$$\begin{aligned} &\texttt{X R B B R B R B B} \rightarrow \texttt{X R B B R R X R B} \rightarrow \texttt{X R B B R R X X R} \\ \rightarrow &\texttt{X R B B R R X X X} \rightarrow \texttt{X R B B B X X X X} \rightarrow \texttt{X X R B B X X X X} \\ \rightarrow &\texttt{X X X R B X X X X} \rightarrow \texttt{X X X X R X X X X} \rightarrow \texttt{X X X X X X X X X} \end{aligned}$$

1. It is clear that problem does not always have a solution (i.e., all marbles cannot be removed); for example, if all marbles are blue then nothing can be removed. Give a *necessary and sufficient* condition for the problem to have a solution. Prove that your condition is indeed necessary and also sufficient. “Necessary” means that without the condition, there is no solution; “Sufficient” means that with the condition there is a solution. (Hint: Consider small problem sizes to understand; red marbles are the key, in particular the number of red marbles.)

2. Assuming that there is a solution, give an algorithm to remove all the marbles. Prove that your algorithm indeed works. (Hint: It is important to remove the red marbles in a proper manner. Consider $\texttt{R R R}$, which clearly has a solution:

   $$\texttt{X B R} \rightarrow \texttt{X R X} \rightarrow \texttt{X X X}$$

   However, if you remove the middle red first, then you will be stuck. Consider small problem sizes to understand what happens. For large sizes, use decrease/divide and conquer. Use induction in your proof.)

Exercise 4.7. Given two integers a and b, we want to compute $d = \gcd(a, b)$ and two integers x and y such that $ax + by = d$.

Modify The Euclidean Algorithm to return x and y as well as d.

Exercise 4.8. There are n adults in town A and they all need to go to town B. There is only a single motorbike available which is owned by two boys. The motorbike can carry *exactly* one adult *or* one or two boys at a time (note that at one least person is needed to ride the motorbike). Show how all adults can be moved from town A to town B while, at the end, leaving the motorbike with the two boys in town A.

Show the correctness of your algorithm by using mathematical induction and analyze the number of trips needed. (Hint: Use decrease and conquer to reduce the problem size.)

Exercise 4.9. You are given n stones (assume that n is a power of 2) each having a distinct weight. You are also given a two-pan balance scale (no weights are given). For example, given two stones, you can use the scale to compare which one is lighter by placing the two stones on the two different pans. The goal is the find the heaviest and the lightest stone by using as few weighings as possible. Give a divide and conquer strategy that uses only $\frac{3n}{2} - 2$ weighings. (Hint: See Worked Exercise 4.2.)

Exercise 4.10. Give a recursive algorithm to compute b^n for a given integer n. Your algorithm should perform only $\mathcal{O}(\log n)$ integer multiplications.

Exercise 4.11. You have the task of heating up n buns in a pan. A bun has two sides and each side has to be heated up separately in the pan. The pan is small and can hold only (at most) two buns at a time. Heating one side of a bun takes 1 minute regardless of whether you heat up one or two buns at the same time. The goal is to heat up both sides of all the n buns in the *minimum* amount of time.

Suppose you use the following recursive algorithm for heating up (both sides) of all n buns. If $n = 1$, then heat up the bun on both sides; if $n = 2$, then heat the two buns together on each side; if $n > 2$, then heat up any two buns together on each side and recursively apply the algorithm to the remaining $n - 2$ buns.

- Set up a recurrence for the amount of time needed by the above algorithm. Solve the recurrence.
- Show that the above algorithm does not solve the problem in the minimum amount of time for all $n > 0$.
- Give a correct recursive algorithm that solves the problem in the minimum amount of time.
- Prove the correctness of your algorithm (use induction) and also find the time taken by the algorithm.

Exercise 4.12. Consider the following algorithm for generating all $n!$ permutations of $\{1, 2, \ldots, n\}$. For example, if $n = 3$, then there 6 possible permutations: 123, 132, 213, 231, 312, 321.

For simplicity, assume the numbers are stored in a global array A in increasing order, i.e., $A = [1, 2, \ldots, n]$.

Algorithm 13 Permutations — Permutations of first n natural numbers
Input: An integer greater than or equal to 1
Output: Prints all $n!$ permutations of global array $A = [1, 2, \ldots, n]$

```
func PERMUTATIONS($n$):
    if $n == 1$:
        PRINT($A$)
    else:
        for $i = 1$ to $n$:
            PERMUTATIONS($n - 1$)
            if $n$ is odd:
                SWAP($A[1], A[n]$)
            else:
                SWAP($A[i], A[n]$)
```

1. Run the algorithm by hand for $n = 2, 3$, and 4.
2. Prove the correctness of the algorithm.
3. Analyze the running time of the algorithm.

5 MORE DIVIDE AND CONQUER

In this chapter, we will see more problems which can be solved using the divide and conquer strategy. The problems that we will study are fundamental with variety of applications in the real world.

5.1 Problem: Sorting

Problem 5.1 ▶ Sorting

Given an array A of n elements that can be ordered[1](i.e., we can write $a \leqslant b$ or $b \leqslant a$), return an ordering A' of A such that $A'[1] \leqslant A'[2] \leqslant \ldots \leqslant A'[n]$.

Sorting is one of the most fundamental problems in computing, perhaps the most well-studied. It arises as a basic subroutine in other problems. There are many different algorithms devised for sorting, but here will focus on two Divide and Conquer algorithms namely, MergeSort and Quicksort. Note that we assume that the only way to infer the ordering between two elements (say a and b) is by *comparing* them, i.e., asking the question "Is $a \leqslant b$?". The goal is to devise an algorithm that does as few comparison operations as possible.

5.1.1 A Simple Sorting Algorithm

Here is an "obvious" algorithm for sorting. (This is not a divide and conquer algorithm.) First, find the *smallest* element in the array and put it in the first position. Then, find the second *smallest* element and place it in the second position. In general, find the ith smallest element and place it in the ith position. To find the ith smallest element we have to only compare with elements that are in the positions $i + 1$ or greater. We can call

[1] Formally, this is called a non-strict total order, in which some binary relation $\leqslant$ is defined and satisfies the following properties:

Connexity – for any x, y, $x \leqslant y$ or $y \leqslant x$.

Antisymmetry – if $x \leqslant y$ and $y \leqslant x$ then $x = y$.

Transitivity – if $x \leqslant y$ and $y \leqslant z$ then $x \leqslant z$.

this algorithm SimpleSort — see Algorithm 14 for the pseudocode. See Figure 5.1 for an example run of this algorithm.

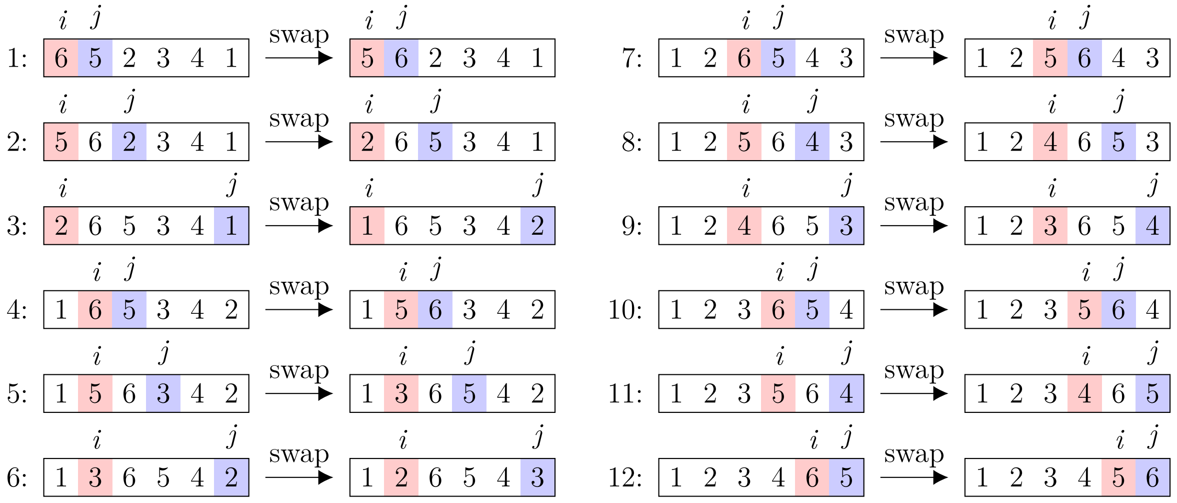

1: 6 5 2 3 4 1 swap 5 6 2 3 4 1
2: 5 6 2 3 4 1 swap 2 6 5 3 4 1
3: 2 6 5 3 4 1 swap 1 6 5 3 4 2
4: 1 6 5 3 4 2 swap 1 5 6 3 4 2
5: 1 5 6 3 4 2 swap 1 3 6 5 4 2
6: 1 3 6 5 4 2 swap 1 2 6 5 4 3
7: 1 2 6 5 4 3 swap 1 2 5 6 4 3
8: 1 2 5 6 4 3 swap 1 2 4 6 5 3
9: 1 2 4 6 5 3 swap 1 2 3 6 5 4
10: 1 2 3 6 5 4 swap 1 2 3 5 6 4
11: 1 2 3 5 6 4 swap 1 2 3 4 6 5
12: 1 2 3 4 6 5 swap 1 2 3 4 5 6
i
j

Figure 5.1: An example to illustrate the swaps of Algorithm SimpleSort on $\{6, 5, 2, 3, 4, 1\}$.

Algorithm 14 SimpleSort – Obvious Sorting Algorithm
Input: An array A of length n
Output: None. Sorts the array in place.

```
1: proc SimpleSort($A$):
2:     for $i = 1$ to $n$:
3:         for $j = i + 1$ to $n$:
4:             if $A[j] < A[i]$:
5:                 swap $A[i]$ and $A[j]$
```

Correctness of SimpleSort

The correctness follows from the loop invariant of the *outer for* loop: after the ith iteration of the *outer for* loop, the ith smallest element is placed in the ith index of the array. It is then never moved again. Formally, we can show the loop invariant by induction.

Running time of SimpleSort

How to measure the running time of SimpleSort? One reasonable measure is to count the *number of comparisons* that the algorithm performs. Note that a "comparison" is an operation that compares one element with another. In Algorithm SimpleSort, there is comparison operation in line 3. Measuring number of comparisons is a good way to measure the *efficiency of any sorting algorithm*. It typically dominates (or is at least of the same order compared to) other operations performed by a sorting algorithm.

We calculate the number of comparisons in SimpleSort. Let the array A contain n elements. In the first iteration of the outer *for* loop, the first element of A is compared with the rest of the elements — this costs $n - 1$ comparisons. In the second iteration of the outer *for* loop, the (current) second element of A is compared with elements in positions 3 onwards until the last — costs $n-2$ comparisons. Thus, in the ith iteration of the outer *for* loop, the (current) ith element of A is compared with elements in positions $i+1$ onwards until the last — costs $n-i$ comparisons.

Thus the total number of comparisons performed by SimpleSort is

$$\begin{aligned}(n-1)+(n-2)+\cdots+2+1 &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)(n)}{2}\end{aligned}$$

since this is the sum of the first $n-1$ numbers.

We can write $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$. The dominating term is $n^2/2$. Hence the number of comparisons of SimpleSort is $\Theta(n^2)$, i.e., it has *quadratic* run time.

SimpleSort took $(n-1)n/2$ comparison operations. Note that $(n-1)n/2 = \binom{n}{2}$, i.e., the total number of different pairs of elements. SimpleSort compares *every distinct pair* of elements, i.e., it compares every element with every other element once. Can we get a "better" sorting algorithm — one that takes substantially less number of comparisons? We next give an algorithm called MergeSort that takes only $\mathcal{O}(n\log n)$ comparisons (assume log is to the base 2). Note that if $n = 2^{20}$, then $n^2 = 2^{40}$, whereas $n\log n$ is only 20×2^{20}. Hence an $\mathcal{O}(n\log n)$ algorithm gives us substantial savings compared to an $\Theta(n^2)$ algorithm.

5.1.2 MergeSort: A Divide and Conquer Sorting Algorithm

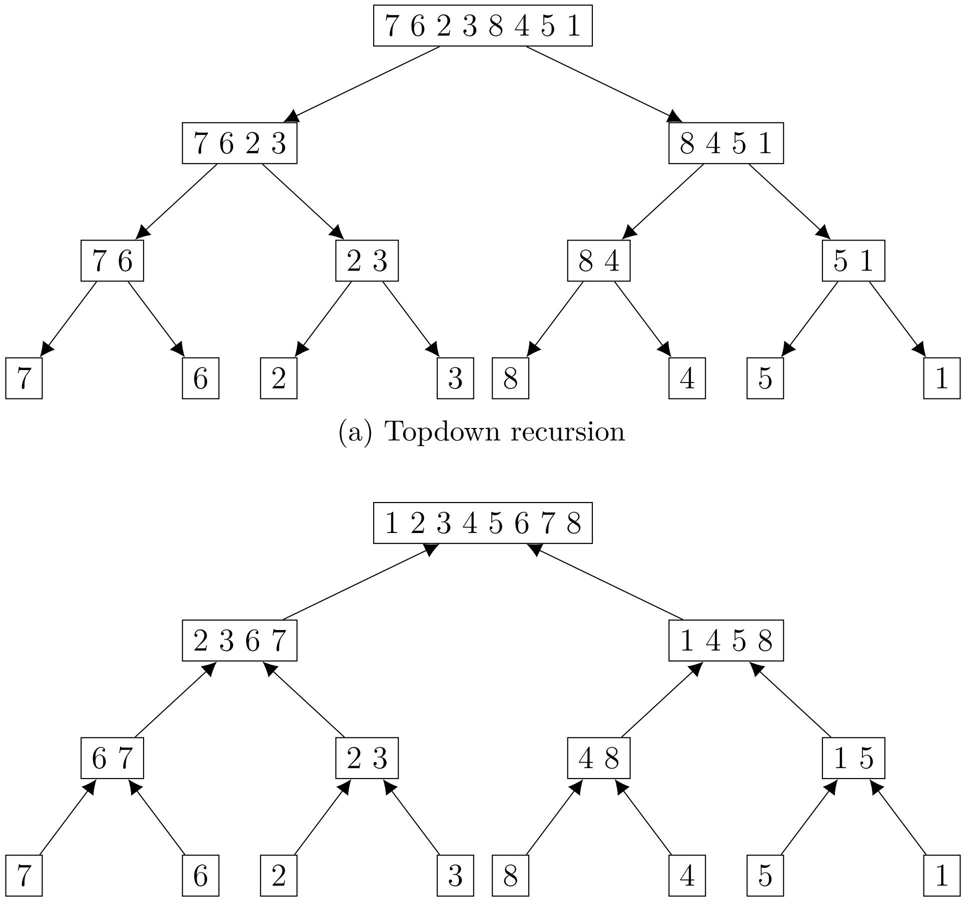
We follow a divide and conquer strategy to design a sorting algorithm. As outlined in Chapter 2, the strategy consists of three parts.

- *Split*: Split the array into two (approximately) equal halves.
- *Solve*: Recursively sort the two subarrays independently.
- *Combine*: Combine the two sorted subarrays (by merging) to get an overall sorted array.

Using the above strategy, it is easy to write a high-level procedure for MergeSort as follows.

```
MergeSort($A[1,\ldots,n]$):
1. If $n = 1$ then output $A$.
2. $A_1$ = MergeSort($A[1,\ldots,n/2]$): (Sort the first half).
3. $A_2$ = MergeSort($A[n/2+1,\ldots,n]$): (Sort the second half).
4. Return Merge($A_1, A_2$) (Merge the two halves into an overall sorted array and
   return it. Merge takes two sorted arrays and combines them into one overall sorted
   array.)
```

See Figure 5.2 for the MergeSort recursion tree which illustrates the algorithm in action on an input of size 8.



(a) Topdown recursion

(b) Bottom-up merging

Figure 5.2: MergeSort example. The top figure shows the recursion as it unwinds in a *top-down* fashion. The bottom figure shows how the array gets sorted in a *bottom-up* fashion.

Merge Procedure

The only thing that is really non-trivial in MergeSort is the Merge procedure which we describe next. This procedure takes as input two *sorted* arrays A and B and outputs an overall sorted array X consisting of the elements of A and B.

Merge procedure uses an auxiliary array X. X will be initially empty. In each iteration one element from either A or B will be added to X, depending on whichever is the smallest; it then considers the next element in the set from which the element was just added and repeats the process. Since both A and B are *individually sorted themselves* it is enough to consider the elements of both in increasing order (from left to right). The total time is linear in the total number of elements in A and B. At the end, all the elements will be added and they will also be sorted. See Figure 5.3 that illustrates how the Merge procedure works on a example. Pseudocode of Merge is given in Section Section 5.1.2.

Running time of Merge

We need to bound the running time of Merge to bound the running time of MergeSort. As done earlier (for SimpleSort), we will use the number of *comparisons* as an estimate of the running time. How many comparisons does MergeSort procedure take? Suppose the input arrays A and B are each of size k. Then, Merge takes no more than $2k$ comparisons, since at most one comparison is done to add one new element to the X array. More generally, if A and B are of sizes k_1 and k_2 respectively, then Merge takes no more than $k_1 + k_2$ comparisons.

Running time of MergeSort

We write a recurrence. Let $T(n)$ be the number of comparisons taken by MergeSort on input size n. Assume n is a power of 2 (without loss of generality). Then

$$T(n) \leqslant 2T(n/2) + n$$

with the base case: $T(1) = 0$.

Using The DC Recurrence Theorem, we obtain the solution as $T(n) = \mathcal{O}(n \log n)$. Hence MergeSort takes $\mathcal{O}(n \log n)$ time.

Correctness of MergeSort

It is easy to establish the correctness of the above algorithm by using *mathematical induction*. This is similar to the proof of the correctness of Recursive-Max, another divide and conquer algorithm we saw earlier. The induction is on the *size* of the input set S. This is done in two steps.

1. *Base case*: This is the trivial case when the size of S is 1. Clearly, the algorithm is correct, since it outputs this single element.

2. *Induction step*: Assume that the algorithm is correct for *all* sets of size $< n$. This is called the Induction hypothesis.

The proof of establishing the induction step is left as an exercise for you (it is similar to what we did for Recursive-Max in Chapter 2).

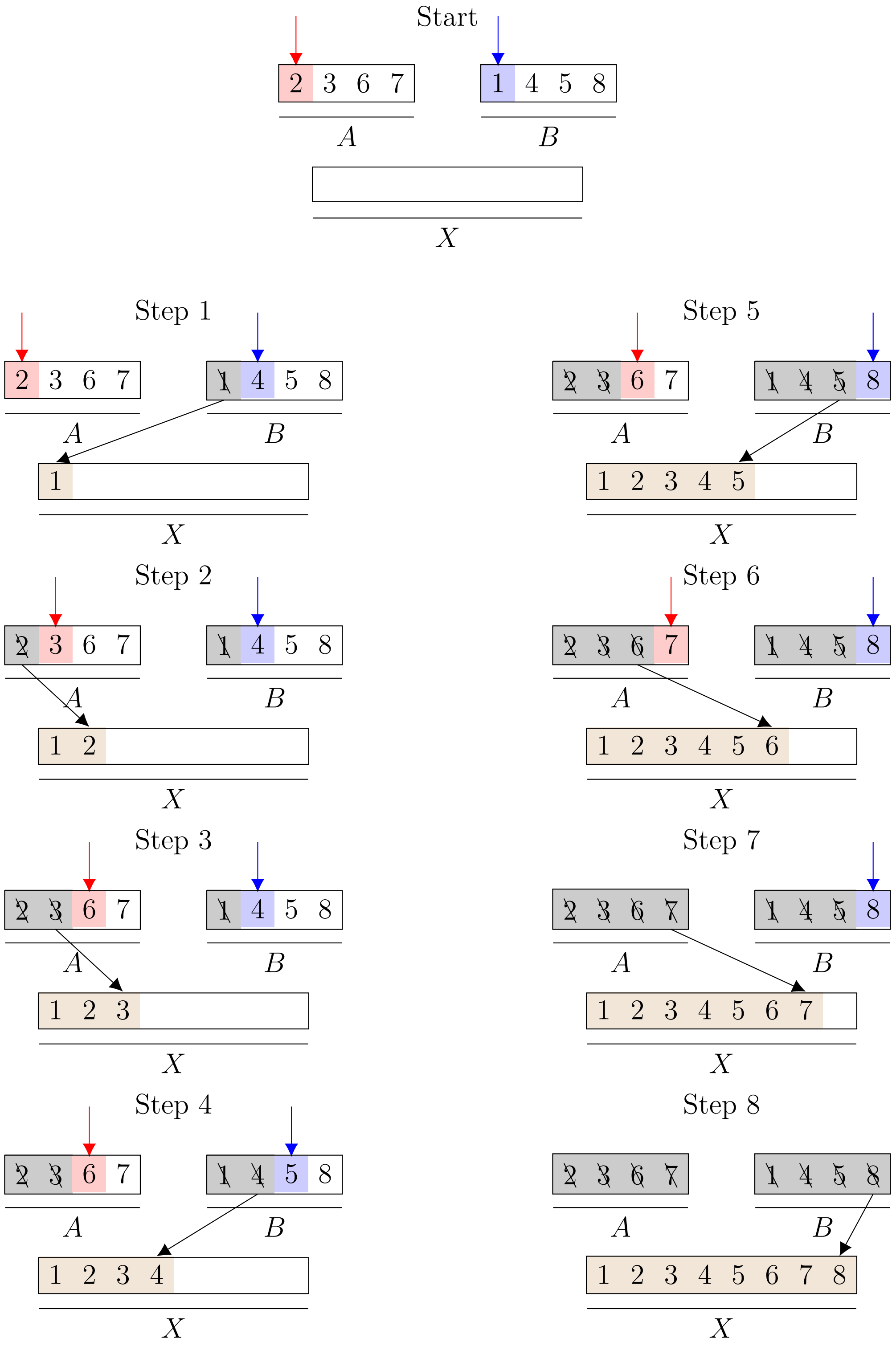


Figure 5.3: Merge Procedure. A and B are the sorted input arrays and X is the output array that is sorted.

Algorithm MergeSort: Pseudocode

We present more detailed pseudocode for MergeSort.

Algorithm 15 MergeSort
Input: An array X of length n

```
func MergeSort($X$):
    MergeSortHelper($A, 1, n$)
```

Algorithm 16 MergeSortHelper – Helper for recursive MergeSort
Input: An array of length n and left and right indices

```
func MergeSortHelper($X$, left, right):
    if left $<$ right:
        middle $= \lceil (\texttt{left} + \texttt{right})/2 \rceil$
        MergeSortHelper($X$, left, middle $- 1$)
        MergeSortHelper($X$, middle, right)
        Merge($X$, left, right)
```

Merge: Pseudocode

We assume the following: *Input:* Array X and indices `left` and `right`, such that the subarrays $X[\texttt{left} \ldots \texttt{middle} - 1]$ and $X[\texttt{middle} \ldots \texttt{right}]$ are already sorted, where $\texttt{middle} = \lceil (\texttt{left} + \texttt{right})/2 \rceil$ *Output:* Sorted array X.

Note that a temporary array `Temp` is used to store the overall sorted array.

Algorithm 17 Merge – Merge two sorted arrays
Input: An array X and left and right indices. Assume $X[\texttt{left} \ldots \texttt{middle} - 1]$ and $X[\texttt{middle} \ldots \texttt{right}]$ are already sorted, where $\texttt{middle} = \lceil (\texttt{left} + \texttt{right})/2 \rceil$.

```
proc Merge($X$, left, right):
    $i$ = left    ▷ i points to index in the first half
    middle $= \lceil (\texttt{left} + \texttt{right})/2 \rceil$
    $j$ = middle    ▷ j points to index in the second half
    $k = 0$
    while $i <$ middle and $j \leqslant$ right:
        $k = k + 1$
        if $X[i] \leqslant X[j]$:
            Temp[$k$] $= X[i]$    ▷ Copy from first half in Temp
            $i \mathrel{+}= 1$
        else:
            Temp[$k$] $= X[j]$    ▷ Copy from second half in Temp
            $j \mathrel{+}= 1$
    if $j >$ right:    ▷ j reaches the right end
        for $t = 0$ to middle $- 1$:
        ▷ Shift elements from first half of X to second half
            $X[\texttt{right} - t] = X[\texttt{middle} - 1 - t]$
    for $t = 0$ to $k - 1$:
        $X[\texttt{left} + t]$ = Temp[$t$]
```

5.1.3 QuickSort: Another Divide and Conquer Algorithm

QuickSort is a popular sorting algorithm that is usually fast in practice. It is implemented as function "qsort" in the UNIX operating system and in C++.

Idea of QuickSort

Let S be the input (an ordered set of n elements) to be sorted. (S is usually represented as an array as in MergeSort, but here we will be a bit more abstract and represent the input as a set.) For convenience, we will assume that all elements in S are distinct, although it is easy to modify the algorithm to handle if there are duplicates. Choose the first element of S, say p — called the *pivot* element. The whole set is then *partitioned* based on p into two parts — S_1 and S_2. All elements of S_1 are less than p and all elements of S_2 are greater than p. We then recursively sort S_1 and S_2. The output is the sorted list of $S_1 + [p] + S_2$, where $+$ here indicates the concatentation of ordered sets (i.e., $(a_1, a_2, \ldots, a_n) + (b_1, b_2, \ldots b_m) = (a_1, a_2, \ldots a_n, b_1, b_2, \ldots, b_m)$). The pseudocode is given in Algorithm 5.

Note that QuickSort is a "Conquer and Divide" algorithm in the sense that (unlike MergeSort), it first partitions the array around the pivot (the left part being less than or equal to the pivot and the right part greater than the pivot) and then recursively sorts the left part and the right part separately. It is easy to implement this partitioning "in place", i.e., if S is stored as an array, then the partitioning around the pivot can be implemented by exchanging elements in the array itself (see Exercise 5.21), without the need for an auxiliary array (as was in the case of MergeSort). This makes QuickSort code save space leading to better running time, although this is not an asymptotic improvement. As is clear from the pseudocode, the partitioning can be done in time linear in the size of the set, i.e., $\mathcal{O}(n)$.

Algorithm 18 QuickSort
Input: A set S
Output: S in sorted order

```
func QuickSort($S$):
    if $|S| \leqslant 1$:
        return $S$
    else:
        Let $p$ be the first element of $S$ — called the pivot
        $S_1 = \{x \in S - \{p\} \mid x < p\}$
        $S_2 = \{x \in S - \{p\} \mid x > p\}$
        ▷ Elements in $S_1$ and $S_2$ are in the same relative order as in $S$
        return QuickSort($S_1$) + $[p]$ + QuickSort($S_2$)
```

Correctness

The correctness of QuickSort can be shown by mathematical induction and is left as an exercise (Exercise 5.4).

Time Analysis: Worst-case

Let $T(n)$ be the worst-case running time (number of comparisons) of Algorithm QuickSort on a set S of size n. By "worst-case running time" we mean the running time that is the maximum among all possible inputs. Note that worst-case running time is usually the *default* notion of running time used in the analysis of algorithms — a single "bad" input can substantially increase the running time of an algorithm. However, for some problems and algorithms, it might be the case that all inputs take approximately the same amount of time. In particular, in the case of QuickSort, it is instructive to understand the "worst-case", "best-case", and "average-case" performance of the algorithm. We will analyse the worst-case and best-case time below, but defer the analysis of average-case later (this involves use of probability). Note that we assumed that all elements in the input set are *distinct*.

Theorem 5.1

The worst-case time, $T(n)$, of QuickSort is $\Theta(n^2)$.

Proof. We can write the recurrence for QuickSort as the following:

$$T(n) \leqslant \max_{0 \leqslant k \leqslant n-1} \{T(k) + T(n-k-1)\} + n - 1$$

The reason is as follows. The pivot partitions the array of size n into two disjoint parts each of size k (the size of set S_1) and $n-k-1$ (the size of set S_2), where k depends on the pivot chosen. (Note that the pivot itself is excluded from the two parts). For example, if the pivot is the median (i.e., the middle element in the sorted order), then k is close to $n/2$. On the other hand, if the pivot is (say) the largest element, then k is $n-1$ (i.e., all the elements are smaller than the pivot) and if the pivot is the smallest element, then $k = 0$ (i.e., all the elements are larger than the pivot). We take the maximum over all k since we want an upper bound on $T(n)$. The cost for partitioning is $n-1$ comparisons (regardless of k), since the pivot has to be compared with the rest of the $n-1$ elements.

The worst-case is when the partitioning produces one subproblem of size $n-1$ and another of size 0 (Exercise 5.5 asks you to show this). That is, the partitioning that leads to the *maximum imbalance* (See Figure 5.4 (top)).

Since partitioning costs $n-1$ comparisons,

$$T(n) = T(0) + T(n-1) + n - 1 = T(n-1) + n - 1$$

since $T(0) = 0$. Solving the above recurrence gives $T(n) = \Theta(n^2)$ (this is left as an exercise — See Exercise 5.5). □

Time Analysis: Best-case

Theorem 5.2

Let $T(n)$ be the best-case running time of QuickSort on a set S of size n. Then $T(n) = \mathcal{O}(n \log n)$.

Proof. We can write the recurrence for best case performance of QuickSort as follows:

$$T(n) \geqslant \min_{0 \leqslant k \leqslant n-1} \{T(k) + T(n-k-1)\} + n - 1$$

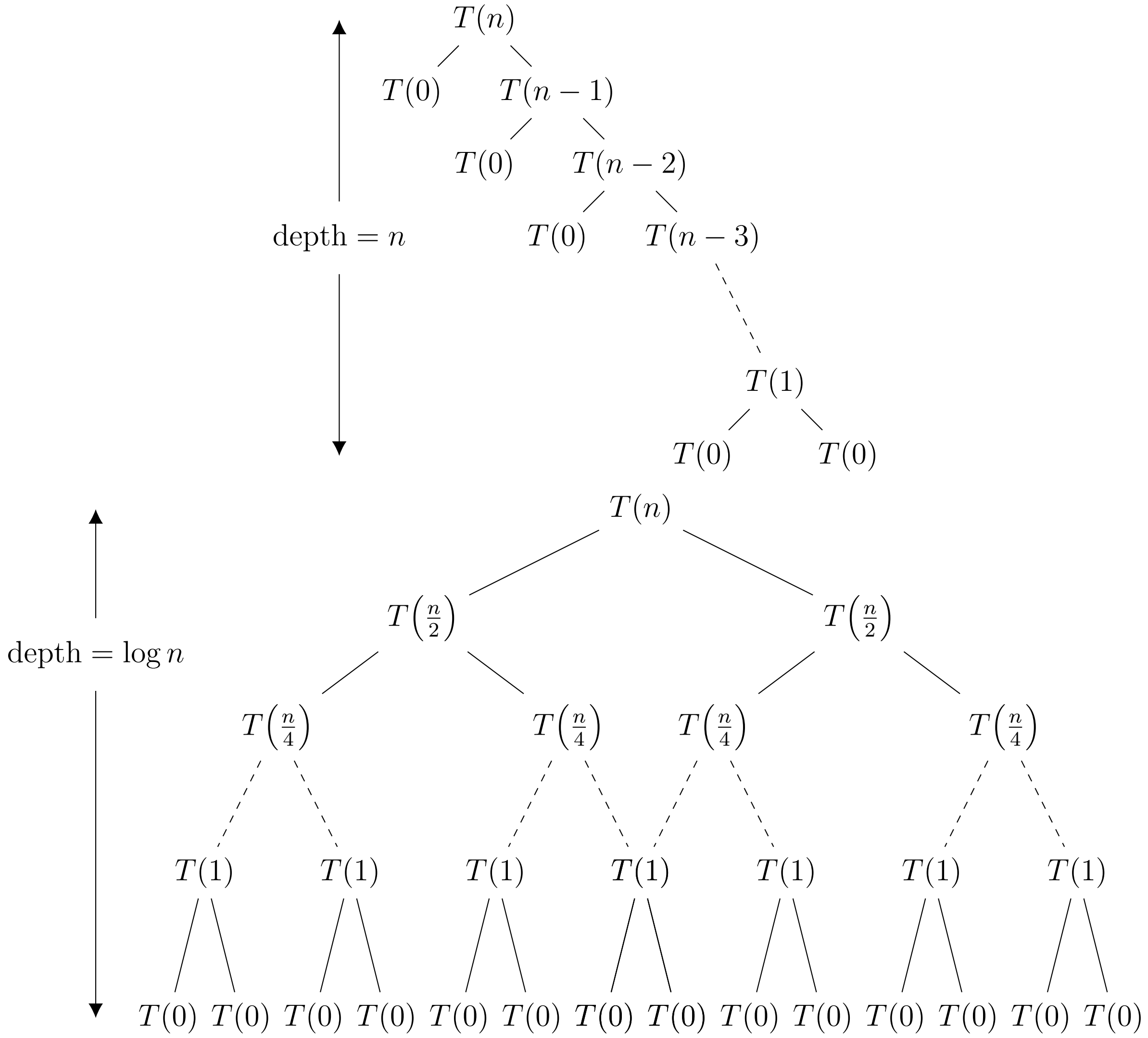


Figure 5.4: QuickSort recursion tree: worst-case (top) and best-case (bottom). The worst-case tree is very skewed (to the right). This happens when the input is already sorted; thus the pivot is always the smallest element in each recursive call and all elements are to the right of the pivot; hence one subproblem is empty and all the remaining elements are in the other subproblem. The depth of the tree is n. The best-case is when the pivot is the middle element, i.e., the median, in *every* recursive call when the two subproblems of almost equal size. The depth of the tree is at most $\log n$.

The best case is when the partitioning produces two subproblems each of size about $n/2$. (Exercise 5.6 asks you to prove that no matter what the partition is, the best-case is $\mathcal{O}(n \log n)$ by showing that the solution of the above recurrence is $\Omega(n \log n)$.) In other words, when we partition the subproblems into (almost) equal size. The reason is that this is the partition that keeps the tree balanced and leads to a recursion tree of depth $\log n$ (See Figure 5.4 (bottom)). Note that this is the smallest depth possible for a binary tree of n nodes. Smaller the depth of the recursion tree, smaller the total number of comparisons, since the total number of comparisons in each level (except at the leaf level) is proportional to the total number of elements in that level (summing across all the nodes). The recurrence is $T(n) \leqslant 2T(n/2) + n$, which gives $T(n) = \mathcal{O}(n \log n)$ by The DC Recurrence Theorem.

(Note that the $\mathcal{O}(n \log n)$ bound holds as long as the two subproblem sizes are within a constant factor of each other, e.g., see Example 2.7.)

□

From the above, it is clear that although the best case performance of QuickSort is $\Theta(n \log n)$ (this is the best possible for any algorithm due to the existence of a lower bound of $\Omega(n \log n)$ for any comparison-based sorting algorithm — See Exercise 5.7), its worst-case time is much worse compared to that of MergeSort (which has worst-case running time of $\mathcal{O}(n \log n)$). Despite this, QuickSort is used widely in practice! We will examine this issue more closely when we analyze the average-case performance of QuickSort.

5.1.4 Average-Case Analysis of QuickSort*

In the last section we discussed the best-case and the worst-case analysis of QuickSort. Here we discuss the average-case analysis of QuickSort. But before that, we need to understand what "average-case" analysis of an algorithm means. This literally means: the performance of the algorithm *averaged over all possible inputs* for the algorithm. First let us elaborate on this meaning for algorithms with a finite number of inputs which is the case for many algorithms, including QuickSort (as we will see below). For each input, find the running time of the algorithm, sum the running time over all the (finite) inputs and compute the average by dividing the sum by the total number of inputs — this gives the average-case performance of the algorithm.

Let us explain the above with respect to QuickSort. First, it might appear that the number of inputs for QuickSort is not finite. Indeed it is, because the only thing that matters for comparison-based algorithms[2] such as QuickSort is the relative ordering of the elements, not their values. How many possible orderings are there for an input size of n elements: it is $n!$, which is the total number of inputs for comparison-based algorithms such as QuickSort. Thus the average-case analysis of QuickSort is the average performance of QuickSort which is the average runtime (as usual, measured by the number of comparisons) over all the $n!$ different inputs. Mathematically, we can write it as follows. Order the $n!$ permutations of the input in some order and let $i = 1, \ldots, n!$ denote the ith permutation of the input. Let T_i be the number of comparisons that QuickSort does on the ith permutation. Then the average run time of QuickSort, i.e., the average-case

[2] Note that one can have non-comparison based algorithms for sorting which can make use of the actual values of the elements for sorting.

performance of QuickSort is defined as

$$T_{\text{avg}} = \frac{1}{n!}\sum_{i=1}^{n!} T_i$$

How can we compute T_{avg} of QuickSort. This seems to be a non-trivial task as it seems to involve determining the number of comparisons of QuickSort on each of the $n!$ permutations which is a huge number. Even if this is possible in principle, it seems difficult to compute this quantity, especially, analytically.

Surprisingly, this can be done by using some facts about probability and averages (see Chapter 8). The first is, computing T_{avg} is equivalent to choosing a random permutation, i.e., choosing a permutation uniformly at random, and computing the performance of QuickSort on that random permutation. This follows from the definition of "average" value of a set of values which is defined as simply as a weighted sum of the values, weighted by the respective probabilities of choosing the values. That is, let set $S = \{s_1, s_2, \ldots, s_n\}$ of n values, and each element s_i be chosen with probability p_i, then the average value is

$$\text{Average} = \sum_{i=1}^{n} p_i s_i$$

Note that the sum of all the p_i's should equal 1, i.e., $\sum_{i=1}^{n} p_i = 1$, since p_i's form a probability distribution.

Coming back to QuickSort, the set S consists of $n!$ values, where the ith value corresponds to the number of comparisons of QuickSort on the ith permutation. If each value (i.e., permutation) is chosen randomly, i.e., uniformly at random (in other words, each permutation is equally likely), then $p_i = 1/n!$. Thus

$$T_{\text{avg}} = \frac{1}{n!}\sum_{i=1}^{n!} T_i = \sum_{i=1}^{n!} \frac{1}{n!} T_i$$

Hence, to compute T_{avg}, it is enough the compute the (average or expected) performance of QuickSort on a *random* permutation.

Analyzing the performance of QuickSort on a random input permutation (i.e., each permutation is equally likely to be the input), is called as *average-case analysis* or *probabilistic analysis* of QuickSort. This is discussed in Section 8.4.1. Here we simply state the result and refer the reader to the proof in Section 8.4.1.

Theorem 5.3

The average-case performance of QuickSort is $\mathcal{O}(n \log n)$.

The above theorem says that on an average input QuickSort takes $\mathcal{O}(n \log n)$ comparisons. This is significantly better than the worst-case performance of $\Theta(n^2)$. Thus, on average, i.e., for a significant number of inputs, QuickSort takes $\mathcal{O}(n \log n)$ comparisons. In Chapter 8, we show that, in fact, for most inputs QuickSort takes $\mathcal{O}(n \log n)$ time and only for a tiny fraction of inputs, the number of comparisons is much larger. In other words, "bad" inputs for QuickSort (e.g., input in increasing or decreasing order) constitute only a tiny fraction of the total possible number of all $n!$ inputs. In practice, it turns out that such bad inputs show up much less often. Thus, in the real-world (where most inputs are likely to be good), QuickSort performs well.

5.2 Problem: Selection

Problem 5.2 ▶ Selection

Given an array S of n elements that can be ordered (i.e., we can write $a \leqslant b$ or $b \leqslant a$), return the kth-smallest element, called the element of *rank* k.

Selection is also a basic problem. In particular, selecting the "middle" element, i.e., the *median*, arises in many applications. Note that the median is the $\lceil n/2 \rceil$ smallest element (if n is even, the $\lfloor n/2 \rfloor$ smallest element is also a median). Clearly, we can find the median, or any ith smallest element, in $\mathcal{O}(n \log n)$ time via sorting. Thus the non-trivial question is whether one can do better, in particular, if one can solve the problem in *linear*, i.e., $\mathcal{O}(n)$ time. Note that, as in sorting, we only use comparisons to infer the order between elements.

A Linear Time Selection Algorithm

Main idea behind the algorithm.

Assume that we are looking for the kth smallest element. The idea is "Conquer and Divide" and is very similar to the partitioning paradigm of the QuickSort algorithm. The input set S is partitioned into two parts with respect to a pivot element: all elements less than the pivot go to the left (call this set S_1) and all elements greater than the pivot go to the right (call this S_2). (We assume that all elements in the input set are distinct.) If the size of S_1 is equal to $k-1$, then clearly the pivot is the desired kth element. On the other hand, if the size of S_1 is bigger than $k-1$, then clearly the desired element is in the set S_1; otherwise it will be in S_2, but its rank will be $k - |S_1| - 1$ (why?). To achieve linear time, what property do we need regarding the size of the subproblem generated? Note that there will be only one subproblem (on set S_1 or S_2). To achieve linear time, we require subproblem size to go down by a *constant* factor. This will imply that the number of comparisons in each recursive level goes down by a constant factor leading to a linear time (number of comparisons) overall.

To ensure that the subproblem size goes down by a constant factor, the pivot has to be chosen carefully — it has to be chosen such that its rank is "somewhere in the middle", i.e., within a constant factor of $n/2$ (the rank of the median). The algorithm uses a clever idea: it finds such a pivot by recursively solving another selection problem as follows. Partition the set S into groups of 5 elements (the last group might have less than 5), e.g., the first 5 elements, the next 5 elements as so on. Find the median of each of these groups — there will be about $n/5$ medians; this is easy to find in linear time, since each group has only 5 elements.[3] the median to be We now *recursively* find the median of these $n/5$ medians. It turns out that this "median of medians" is a good pivot, that partitions the overall set into approximately balanced sets. This can be shown by showing that a constant fraction of the number of the elements in the set are smaller *and* larger than this pivot (see proof of Theorem 5.5).

Algorithm 19 gives the pseudocode.

[3] The last group can have less than 5 elements; in this case, if there are even number of elements in the group we take the median to be second smallest element in the group.

Algorithm 19 Select
Input: An array S and the desired order statistic, k
Output: The value of the kth order statistic

```
1:  func SELECT($S$, $k$):
2:      if $|S| == 1$:
3:          return $S[1]$
4:      Partition $S$ into $\lceil n/5 \rceil$ groups of 5 elements each and a leftover group of up to 4 elements.
5:      Find the median of each group
6:      $R$ = the set of these $\left\lceil \frac{n}{5} \right\rceil$ medians
7:      $m = |R|$
8:      $p = $ SELECT($R$, $\lceil m/2 \rceil$)
9:      $S_1 = \{x \in S \mid x < p\}$
10:     $S_2 = \{x \in S \mid x > p\}$
11:     if $|S_1| = k - 1$:
12:         return $p$
13:     else if $|S_1| > k - 1$:
14:         return SELECT($S_1, k$)
15:     else:
16:         return SELECT($S_2, k - |S_1| - 1$)
```

Correctness

Theorem 5.4

Algorithm Select returns the correct value.

Proof. The correctness can shown by induction. We leave it as an exercise for the reader (Exercise 5.8). □

Run-time

Theorem 5.5

The run-time of Algorithm Select is $\mathcal{O}(n)$.

Proof. We first analyze the following: How many elements in S are larger than p, the "median of medians" value computed in line 8 of the algorithm?

Excluding the leftover group, and the group that includes p, in at least half of the remaining groups, there are at least three elements that are greater than p. Thus, at least

$$3\left(\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil - 2\right) \geqslant \frac{3n}{10} - 6$$

in S are greater than p.

Similarly, at least $\frac{3n}{10} - 6$ elements in S are $\leqslant p$. Thus, Select is called in line 8 with at most $\frac{7n}{10} + 6$ elements.

Let $T(n)$ denote the run-time on a set of size n. We note that the number of comparisons in the top level of the recursion (i.e., in lines 4, 5, 9 and 10) is bounded by αn, for some constant $\alpha > 0$. This is because one can find each of the $\lceil n/5 \rceil$ medians in $O(n)$ time (e.g., simply sort each group of 5 elements which takes $O(1)$ time) and partitioning the set S by the pivot takes also $O(n)$ time. Thus we can write the recurrence:

$$T(n) \leqslant T\Big(\Big\lceil \frac{n}{5} \Big\rceil\Big) + T\Big(\frac{7n}{10} + 6\Big) + \alpha n$$

We show that $T(n) \leqslant cn$ for some constant $c > 0$ by using induction. Assume $T(k) \leqslant ck$ for all $k < n$. Then, by induction hypothesis:

$$\begin{aligned} T(n) &\leqslant c\Big(\frac{n}{5} + 1\Big) + c\Big(\frac{7n}{10} + 6\Big) + \alpha n \\ &\leqslant \frac{9cn}{10} + 7c + \alpha n \\ &\leqslant cn \end{aligned}$$

if $n > 70$ and for a sufficiently large constant c (compared to α). (How large should c be?) Also, the base case (say, $n \leqslant 70$) can be satisfied by choosing a constant c sufficiently large, i.e., $T(k) = \Theta(1)$, for $k \leqslant 70$. □

5.3 Problem: Multiplying two numbers

We consider a very basic problem: Given two numbers x and y (say, in decimal or in binary), how fast can we multiply them? Let us assume that x and y both have n digits. We will assume that multiplying or adding two single digit numbers takes constant time. We want to minimize the total time to multiply two n-digit numbers. We know an algorithm for this problem that is taught in schools, namely "school multiplication." This algorithm, multiplies each digit of y with x, one by one, and then adds all the respective sums. Clearly this algorithm takes $\Theta(n^2)$ time, since it involves n^2 single digit mutiplications and about n^2 single digit additions. Incidentally, the 9th century mathematician Al-Khwarizmi (the term "algorithms" is coming from his name) who systematically developed algorithms for the four arithmetic operations (in decimal system), had a different algorithm for multiplication which also needed $\Theta(n^2)$ operations. On the other hand, note that, crucially, adding two n-digit numbers takes only $\mathcal{O}(n)$ time. We exploit this to design an improved algorithm using divide and conquer.

Main Idea of the Algorithm

Let x and y be two n-bit numbers. How can we use divide and conquer to multiply? One obvious way is to split the two numbers into two parts — the first (most significant) $n/2$ bits and the next (least significant) $n/2$ bits. As usual, we will assume n to be a power of 2. Then we can write:

$$x = 2^{n/2}x_1 + x_2$$

and

$$y = 2^{n/2}y_1 + y_2$$

where $x_1(y_1)$ and $x_2(y_2)$ are the most significant bits and least significant bits of $x(y)$ respectively. Hence,

$$xy = 2^n(x_1y_1) + 2^{n/2}(x_1y_2 + x_2y_1) + x_2y_2$$

Thus, we have reduced the problem of multiplying two n-bit numbers to multiplying *four* $n/2$-bit numbers (x_1y_1, x_1y_2, x_2y_1, and x_2y_2) plus some constant number of additions and shifts (multiplying by power of 2). Note that additions and shifts take only $\mathcal{O}(n)$ time. If $T(n)$ represents the number of bit operations for multiplying two n-bit numbers, then based on the above formula, we can write the recurrence:

$$T(n) = 4T(n/2) + \mathcal{O}(n)$$

As usual, the base case is $T(1) = \mathcal{O}(1)$. The solution to the above recurrence via the DC Recurrence Theorem is $\Theta(n^2)$, which is no better than the school multiplication. How to improve this?

The key idea is to reduce the number of multiplications at the cost of a few more additions and subtractions (costing $\mathcal{O}(n)$ time total). The trick is to compute $x_1y_2 + x_2y_1$ using an idea that can be traced to Gauss:

$$(x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$$

Hence,

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

Thus, we can compute $x_1y_2 + x_2y_1$ using only three multiplications (two of which x_1y_1 and x_2y_2 are needed anyway!). Now we can write a better recurrence:

$$T(n) = 3T(n/2) + \mathcal{O}(n)$$

(Strictly speaking, the above recurrence should be $T(n) \leqslant 3T(n/2 + 1) + \mathcal{O}(n)$, since $(x_1+x_2)(y_1+y_2)$ can involve multiplying two $(n/2+1)$-bit numbers. See Exercise 5.12.) The solution to the above recurrence via The DC Recurrence Theorem is $\Theta\left(n^{\log_2 3}\right) = \mathcal{O}(n^{1.585})$, which is significantly better than $\Theta(n^2)$.

There is an interesting story behind the above algorithm which is due to Karatsuba who was a student with Kolmogorov, one of the prominent mathematicians of the last century. Kolmogorov had conjectured that the school mulpilication algorithm that had a complexity of $\Theta(n^2)$ is optimal and had called a seminar to discuss the computational complexity of multiplication among other problems. Soon after this, Karatsuba disproved the conjecture by discovering his algorithm which runs in $\mathcal{O}(n^{1.585})$. Kolmogorov was excited by this result and he gave many lectures on this. He also wrote a paper attributing the result to Karatsuba who came to know about the published paper much later!

Karatsuba's algorithm raises the question whether there are faster algorithms for integer multiplication. Indeed, we will see that one can design substantially faster algorithms using Fast Fourier Transform (FFT) that we will see in Section 5.6. The currently fastest known algorithm (after a long line of research) for multiplying two integers is $\mathcal{O}(n \log n)$. One can achieve close to this bound using the FFT technique. It is conjectured that $\Omega(n \log n)$ is a *lower bound* for the complexity of multiplying two integers.

5.4 Problem: Matrix Multiplication*

We consider another fundamental problem, this time involving matrices (which is a fundamental object we will see throughout the course — See Appendix F for basic facts on matrices): Given two $n \times n$ matrices A and B we want to multiply them, i.e., we want to compute $C = A \cdot B$. Note that C is another $n \times n$ matrix whose elements are defined as:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

In other words, the (i, j)th element of C is computed by elementwise multiplication of the ith row of A and the j column of B and adding the result of the n multiplications. Thus $c_{i,j}$ can be computed using $\mathcal{O}(n)$ operations (where one operation is a either an addition or multiplication of two elements of the matrix), a straightforward algorithm takes $\mathcal{O}(n^3)$ time to compute all elements of C. Note that, on the other hand, adding two $n \times n$ matrices, takes $\mathcal{O}(n^2)$ time. We will exploit this crucially in designing a faster divide and conquer algorithm.

5.4.1 A Divide and Conquer Algorithm

Suppose we are given two $n \times n$ matrices A and B. Our approach is similar to the idea we used in multiplying two integers in Section 5.3. There we exploited that adding two integers is cheaper than multiplying; here we do the same. Let us partition each matrix into two *four* submatrices of dimension $n/2 \times n/2$ (as usual, we will assume n is a power of 2) as follows.

Let

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$B = \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Then the product $C = AB$ can be similarly be partitioned into four $n/2 \times n/2$ submatrices as:

$$C = AB = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

where, the submatrices of C can be computed as follows (why?):

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned}$$

Thus to compute r, s, t, and u, we need to perform a total of 8 matrix products each of which is a product of two $n/2 \times n/2$ matrices. Hence, a simple recursive algorithm gives

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta\left(n^2\right) = \Theta\left(n^3\right)$$

using The DC Recurrence Theorem. Can we improve this?

To get a better running time, we use an idea that is very similar in spirit to the idea that was used to improve the performance of multiplying two n-bit numbers in Section 5.3. The observation is that adding (or subtracting) two matrices is cheaper than multiplying. Hence if we can reduce the number of multiplications from 8 to something smaller (say 7), at the cost of a few more (a constant number) of additions/subtractions, then we can get a faster algorithm. This is exactly what Strassen's algorithm does.

5.4.2 Strassen's Algorithm

Strassen found a clever (and highly non-trivial) way to compute r, s, t and u using 7 matrix multiplications.

First we define 14 $\frac{n}{2} \times \frac{n}{2}$ matrices $A_1, A_2, \ldots, A_7$ and $B_1, B_2, \ldots, B_7$ as shown below.

$$\begin{aligned}
A_1 &= a & B_1 &= g - h \\
A_2 &= a + b & B_2 &= h \\
A_3 &= c + d & B_3 &= e \\
A_4 &= d & B_4 &= f - e \\
A_5 &= a + d & B_5 &= e + h \\
A_6 &= b - d & B_6 &= f + h \\
A_7 &= a - c & B_7 &= e + g
\end{aligned}$$

Next we define the 7 matrix products $P_i = A_i \cdot B_i$ as follows:

$$\begin{aligned}
P_1 = A_1B_1 = &\quad a(g-h) &&= ag - ah \\
P_2 = A_2B_2 = &\quad (a+b)h &&= ah + bh \\
P_3 = A_3B_3 = &\quad (c+d)e &&= ce + de \\
P_4 = A_4B_4 = &\quad d(f-e) &&= df - de \\
P_5 = A_5B_5 = &\quad (a+d)(e+h) &&= ae + ah + de + dh \\
P_6 = A_6B_6 = &\quad (b-d)(f+h) &&= bf + bh - df - dh \\
P_7 = A_7B_7 = &\quad (a-c)(e+g) &&= ae + ag - ce - cg
\end{aligned}$$

We can compute r, s, t and u by linear combinations (additions and subtractions) of $P_1, \ldots, P_7$ as follows.

- Computing s: $P_1 + P_2 = ag - ah + ah + bh = ag + bh = s$.
- Computing t: $P_3 + P_4 = ce + de + df - de = ce + df = t$.
- Computing r: $P_5 + P_4 - P_2 + P_6 = ae + ah + de + dh + df - de - ah - bh + bf + bh - df - dh = ae + bf = r$.
- Computing u: $P_5 + P_1 - P_3 - P_7 = ae + ah + de + dh + ag - ah - ce - de - ae - ag + ce + cg = cg + dh = u$.

Thus, Strassen's recursive algorithm to compute the product of 2 $n \times n$ matrices is:

- Compute recursively the product of 7 $\frac{n}{2} \times \frac{n}{2}$ $P_1, \ldots, P_7$ matrices.
- Perform $\mathcal{O}(1)$ matrix additions and subtractions to compute r, s, t, u.

This leads to the recurrence:

$$T(n) = 7T(\frac{n}{2}) + \Theta\Big(n^2\Big) = \Theta\Big(n^{\log_2 7}\Big) = \mathcal{O}\Big(n^{2.81}\Big)$$

using The DC Recurrence Theorem.

5.5 Problem: Closest Pair of Points

In this section we address a fundamental problem in geometry:

Problem 5.3 ▶ Closest Pair of Points

Given a set S of $n > 1$ points, output a pair of points that are closest among all pairs.

We will assume distances are measured by the usual Euclidean distance and "closest" means according to this distance measure. Without loss of generality, we will assume that all the points are distinct.

There is a naive algorithm that computes the distances between every pair of points and outputs the pair with the minimum pairwise distance. This takes $\Theta(n^2)$ time, since we have to compute the distances between $\binom{n}{2}$ points. Note that the costly operation in this problem is finding the distance (or comparing the coordinates) between two given points; we want to minimize the number of such operations.

The main question is whether we can solve this problem significantly faster. Indeed, this is possible if the points are in one dimension, i.e., all points are on a line, say the x-axis. In this case, one can simply sort the points and go through the sorted order and output the *consecutive* pair of points that are closest; see Worked Exercise 5.2. It is easy to see that the closest pair will be consecutively listed in the sorted order. Hence this takes $\mathcal{O}(n \log n)$ time.

We now to turn to higher dimensions. Can we still solve the problem in $\mathcal{O}(n \log n)$ time? This is more challenging in higher dimensions, say in 2 dimensions, i.e., the points are in a plane. In this case, sorting alone does not work. However, a divide and conquer strategy works. This is a generalization of the divide and conquer algorithm that can be applied to even the one-dimensional case; see Worked Exercise 5.2.

Main ideas behind the algorithm

We consider a set $S = \{p_1, \ldots, p_n\}$ of points in two dimensions, where each point p_i has coordinates (x_i, y_i). We use a natural divide and conquer strategy. We partition the set S of n points into (nearly) equal disjoint subsets S_1 and S_2 and recursively find the closest pair in S_1 and S_2 respectively and then combine the results efficiently to find the overall closest pair. What is a good way to partition? We would like to partition in such a way so that each subset contains points that are close in at least one dimension. To do this efficiently, we will *sort the points according to their x coordinates* (say using MergeSort) and let S_1 be the subset of $\lfloor n/2 \rfloor$ points with the smallest x coordinates and S_2, the rest of the points. Equivalently, let m be the median of the S points, i.e., the $\lfloor n/2 \rfloor$ ranked point in the sorted order of the x coordinates. Then we include all points having rank less than or equal to m to S_1 and the rest to S_2. (See Figure 5.5) We find the closest pair of points in S_1 and S_2 using recursion. The recursion bottom out when there are only two points (or one point) in which case the answer is trivial.

Note that the recursion takes care of all pairs of points that belong to S_1 as well as all pairs that belong to S_2, but does not look at pairs where one point is in S_1 and the other in S_2. This is addressed when we combine the results of the recursion. Note that one cannot naively compare each point in S_1 with each point in S_2, as this involves $(n/2)^2 = \Theta(n^2)$ comparisons. Instead, we use the information returned from the recursive calls to efficiently find the closest pair between points in S_1 and S_2.

Let points a and b be the closest pair of points returned from the recursive call on S_1; similarly let c and d be the closest pair of points returned from the recursive call on S_2. Also, let the distances between the respective pairs of points be: $d_1 = \text{dist}(a, b)$ and $d_2 = \text{dist}(c, d)$. Let $\delta = \min\{d_1, d_2\}$. Hence δ is the closest pair distance among pairs of points that both belong to either S_1 or S_2. *So, the main insight for efficiently checking pairs where one point is in S_1 and the other in S_2 is to check only those pairs that have distance closer than δ.* To check only those pairs, it is enough to look at the strip that is of length δ on either side of a vertical line that divides S_1 and S_2. Specifically, consider the vertical line L that goes through the median m; note that all points in S_1 are to the left of L (including m itself which is on the line) and all points in S_2 are to the right of L. Draw two vertical lines L_1 and L_2 that is at distance δ on either side of L. See Figure 5.5.

Since we need to check for point pairs that are at a closer distance than δ, it is clear that we need to check points that are between L_1 and L_2, i.e., on the strip of length 2δ. This is because, any point that is to the left of L_1 cannot be closer than δ to any point in S_2 and similarly any point to the right of L_2 cannot be closer than δ to any point in S_1. Note that it is easy to identify all points in the strip, they are the ones that have their x-coordinates within δ of the median m's x-coordinate.

Algorithm 20 ClosestPair

Input: A set S of n points in 2 dimensions stored in X and Y. Points in X are sorted by x-coordinate, points in Y by y-coordinate.

Output: The closest pair of points in S

```
func ClosestPair($X$, $Y$):
    if $|X| \leqslant 3$:
        return Brute-Force Solution by comparing all pairs
    else:
        $m = $ Median($X$)
        ▷ Partition $X$ around the median
        $X_1 = \{(x, y) \in X \mid x \leqslant m\}$
        $X_2 = \{(x, y) \in X \mid x > m\}$
        $(a, b) = $ ClosestPair($X_1, Y$)
        $(c, d) = $ ClosestPair($X_2, Y$)
        if $\text{dist}(a, b) \leqslant \text{dist}(c, d)$:
            $\delta = \text{dist}(a, b)$
            closest_pair $= (a, b)$
        else:
            $\delta = \text{dist}(c, d)$
            closest_pair $= (c, d)$
        $Q = \{(x, y) \in Y \mid |x - m| \leqslant \delta\}$    ▷ Points in the strip sorted by y-coordinate.
        for each $p \in Q$ with index $i$:
            for each $q \in Q[i + 1..i + 8]$:
            ▷ Compare p with the next 8 points in the order
                if $\text{dist}(p, q) < \delta$:
                    $\delta = \text{dist}(p, q)$
                    closest_pair $= (p, q)$
        return closest_pair
```

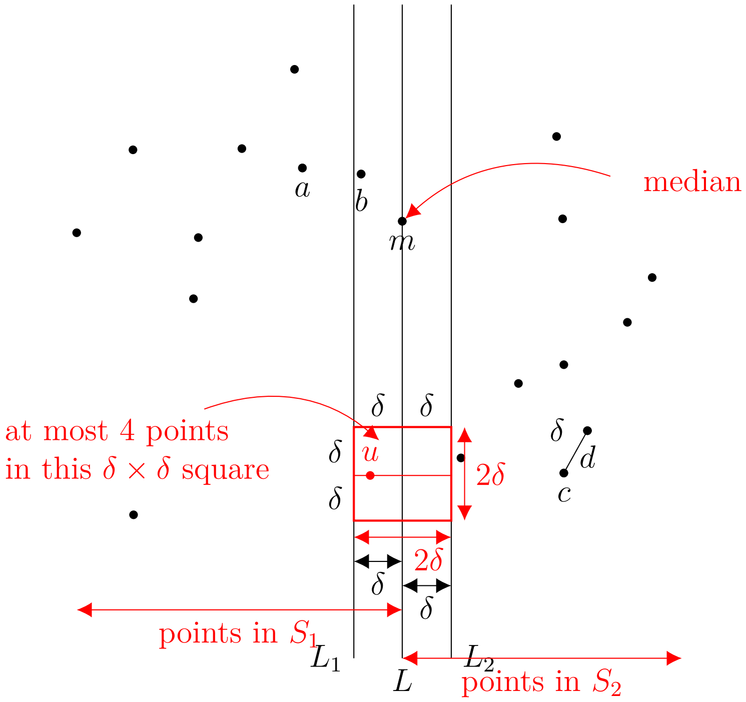


Figure 5.5: Demonstration of ClosestPair. m is the median of the x-coordinates of points in S. S_1 consists of all points whose x-coordinate is less than or equal to the median and S_2 the rest. (a, b) is the closest pair among points in S_1 and (c, d) is the closest pair among points in S_2. We assume that among these two pairs (c, d) has the minimum distance and has value δ. L_1 and L_2 are vertical lines that are of distance δ from the vertical line passing through the median. If you consider any point u in the strip, then one has to compare it with points only in the $2\delta \times 2\delta$ square as shown (in red). This square can have at most 16 points in it, since any $\delta \times \delta$ square can have at most 4 points in it, because if there were more points, then two of them will be closer than δ.

Note that although, we need to compare only points falling within this strip 2δ, this strip can contain most of the n points. Hence again one cannot compare every pair of points falling within this strip. *The key idea that saves the day is that each point in this strip need to be compared with only a constant (at most 16) number of other points in the strip!* How can we do this? Consider a point u in the strip. We need to compare u with points that fall within the square R of dimensions $2\delta \times 2\delta$ (see Figure 5.5). Note that any point outside this square will be at distance larger than δ. How many points can be within this square R? Since no two points in S_1 (or S_2) can be closer than δ, there can be at most 4 points in a $\delta \times \delta$ square and hence at most 16 points in square R (see Figure 5.5). In fact, it can be shown that there can be at most 9 points in square R (but any constant will work for us).

The final idea to compare the points efficiently is to sort the points according to their *y-coordinates* (using MergeSort). Then we go through the sorted order of points in the *strip* (starting from the point with the smallest y coordinate) and compare each point with 8 other points that come *next to it*, i.e, have larger y-coordinates. Why eight? Note that there can be at most 8 other points that can be in the square R that have a larger y coordinates (in fact, it can be shown can be at most 5 points).

Thus the "combining" process that checks pairs of points in the strip takes only $O(n)$ comparisons (at most 8 comparisons per point). Hence we can write the recurrence for $T(n)$, the total number of comparisons for a set of n points. (As usual, we will assume that n is a power of 2.)

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

which gives $T(n) = \mathcal{O}(n \log n)$ (by the DC Recurrence Theorem).

The algorithm implementation and its analysis

It is straightforward to implement the algorithm based on the above description. See Algorithm 20 for the pseudocode. There is one key point to notice. We assume that the points are input in sorted order of their x and y coordinates and stored respectively in the arrays X and Y. This preprocessing step costs $O(n \log n)$ time (using MergeSort). If points are not initially sorted, we first sort them (once based on x-coordinates and stored in X and another based on y-coordinates and stored in Y) and then then call the recursive algorithm. The reason for this presorting is as follows.

Since the algorithm is recursive, to get $O(n \log n)$ run time, one should sort the points in the strip (after the recursive calls on S_1 and S_2 return). Since there can be $\Theta(n)$ points in the strip, sorting these many points takes $O(n \log n)$ time and that dominates the cost. Furthermore, even to partition one should not sort the points in every recursive call. This also costs $O(n \log n)$ time. Hence the recurrence will be

$$T(n) = 2T(n/2) + O(n \log n)$$

which gives $T(n) = O(n \log^2 n)$ (Exercise 5.22 asks you to show this.)

The way to avoid this is to *presort* all the n points at the beginning of the algorithm, once based on x-coordinates (which is anyway needed for partitioning S into S_1 and S_2) and then based on y-coordinates and store them in arrays X and Y respectively. Array X can be used to partition the points in $O(n)$ time in each recursive call, since all the points are already sorted according to their x-coordinates (we simply look at the median of the current set of points and partition according to the median). Similarly, since array Y is presorted according to the y-coordinates, after the recursive calls on S_1 and S_2 returns, we simply go through Y in increasing order and process only the points that belong to the strip (this can be easily checked by looking at the distance of a point's x coordinate from the median m's x-coordinate as mentioned earlier). Hence we can process all the points in the strip in $O(n)$ time. Hence the overall cost of partitioning and combining is only $O(n)$ giving a overall time of $O(n \log n)$ (including the time for presorting the points).

5.6 Problem: Fourier Transform (FT)*

We study a problem of great importance in science and engineering, namely, computing the Fourier transform. Fourier transform is named after Joseph Fourier who used it solve a physics problem in early 19th century. Fourier transform is an amazing insight that transforms (rather decomposes) functions in terms of much simpler functions (read sine and cosine) which are much easier to manipulate than the original function — this has tremendous applications in many areas such as signal processing, data compression, partial differential equations, multiplying large polynomials and numbers etc.

The main idea behind the Fourier transform is that it is used to transform a problem from a given (input) domain (in which it is not very efficient to solve the problem) into another domain where it is significantly easier to solve the same problem. We solve the problem in the transformed domain and then convert the solution (which is in the transformed domain) back to the original domain. Fast Fourier Transform allows one to to do this transformation (in both ways) efficiently.

To illustrate the Fourier transform, we will focus on the application of multiplying two polynomials (which also arises a lot in signal processing).

Problem 5.4

Given two degree d polynomials $A(x) = a_0 + a_1x + \cdots + a_dx^d$ and $B(x) = b_0 + b_1x + \cdots + b_dx^d$, compute the product $C(x) = A(x)B(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$, where the coefficients

$$c_k = a_0b_k + a_1b_{k-1} + \cdots + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}$$

(for $i > d$, set $a_i = b_i = 0$).

Coefficient representation

The polynomial $A(x) = \sum_{i=0}^{d} a_ix^i$ is represented by the coefficient vector $a = (a_0, a_1, \ldots, a_d)$.

Sum

As in the case of multiplying two integers, summing two polynomials is easier than multiplication; we will leverage this to get a faster way to multiplying two polynomials (as we did for multiplying integers and matrices).

Given two polynomials $A(x) = \sum_{i=0}^{d} a_ix^i$ and $B(x) = \sum_{i=0}^{d} b_ix^i$, their sum is

$$C(x) = A(x) + B(x) = \sum_{i=0}^{d} (a_i + b_i)x^i$$

The degree of $C(x)$ is the max degree of $A(x)$ and $B(x)$. The sum of two degree d polynomials, given in a coefficient representation, can thus be computed in $\mathcal{O}(d)$ time.

Product

Given two polynomials $A(x) = \sum_{i=0}^{d} a_i x^i$ and $B(x) = \sum_{i=0}^{d} b_i x^i$, their product is

$$D(x) = A(x)B(x) = \sum_{i=0}^{2d} d_i x^i$$

where, as mentioned earlier,

$$d_k = \sum_{i=0}^{k} a_i b_{k-i}$$

Thus, the coefficient vector d is the *convolution* of vectors a and b. The degree of $D(x)$ is the sum of the degrees of $A(x)$ and $B(x)$. If we compute d_i from the above formula it takes $\mathcal{O}(i)$ steps. Thus finding all $2d+1$ coefficients takes $\Theta(d^2)$ steps. We will show how using the fast Fourier transform (FFT), we can find all the coefficients in $\mathcal{O}(d \log d)$ time.

Point-value representation

To multiply faster, we use a transformation from the "coefficient" representation of a polynomial (the traditional representation) to a "point-value" representation as follows. A degree d polynomial $A(x) = \sum_{i=0}^{d} a_i x^i$ can be represented by a set of $d+1$ pairs

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_d, y_d)\}$$

such that

- for all $i \neq j$, $x_i \neq x_j$ (i.e., all points are *distinct*).
- for every k, $y_k = A(x_k)$;

A key fact (Theorem 5.6) that facilitates the point-value representation is that a degree d polynomial is uniquely determined (i.e., all its $d+1$ coefficients are fixed uniquely) by its values at *any* $d+1$ *distinct* points. Thus we can specify polynomial $A(x)$ either by its coefficient vector $a = (a_0, a_1, \ldots, a_d)$ or by its values $A(x_0), A(x_1), \ldots, A(x_d)$ at $d+1$ distinct points.

Note that a polynomial can be evaluated at any x_i, i.e., value $A(x_i)$ can be computed in $\mathcal{O}(d)$ time by using the so-called *Horner's rule*:

$$A(x_i) = a_0 + x_i(a_1 + x_i(a_2 + \cdots + x_i(a_{d-1} + x_i a_d)))$$

Theorem 5.6

For any set of d point value pairs (x_i, y_i) (where x_is are all distinct) there is a unique degree $d-1$ polynomial $A(x)$ such that $A(x_i) = y_i$ for all pairs.

Proof. We need to solve

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{d-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{d-1} & x_{d-1}^2 & \ldots & x_{d-1}^{d-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{pmatrix}$$

The $d \times d$ matrix (call it M) that appears on the left side of the above equation is called the Vandermonde matrix and has nice properties. In particular, the determinant of the Vandermonde matrix is $\prod_{j<k}(x_k - x_j)$ (Exercise 5.13). Thus, if all the x_i's are distinct, then the determinant is non-zero, and hence the matrix is *nonsingular* (i.e., it is *invertible*) and the linear system has a unique solution. That is, the (unique) coefficient vector is given by: $a^T = M^{-1}y^T$, where a^T and y^T are the column vectors of the row vectors $(a_0, \ldots, a_{d-1})$ and $(y_0, \ldots, y_{d-1})$ respectively. □

Multiplication in the point-value representation

Given two polynomials in (same) point value representation

$$\left\{\left(x_0, y_0^1\right), \left(x_1, y_1^1\right), \ldots, \left(x_d, y_d^1\right)\right\}$$

and

$$\left\{\left(x_0, y_0^2\right), \left(x_1, y_1^2\right), \ldots, \left(x_d, y_d^2\right)\right\}$$

the sum of two degree d polynomials in point value representation is computed in $\mathcal{O}(d)$ time:

$$\left\{\left(x_0, y_0^1 + y_0^2\right), \left(x_1, y_1^1 + y_1^2\right), \ldots, \left(x_d, y_d^1 + y_d^2\right)\right\}$$

To compute the product of two degree d polynomials we need an "extended" point value representation of $2d+1$ points (since the product polynomial has degree $2d$ which is uniquely determined by evaluating it at $2d+1$ distinct points). Given such a representation, the product of two polynomials in point value representation can again be easily computed in $\mathcal{O}(d)$ time:

$$\left\{\left(x_0, y_0^1 y_0^2\right), \left(x_1, y_1^1 y_1^2\right), \ldots, \left(x_{2d+1}, y_{2d+1}^1 y_{2d+1}^2\right)\right\}$$

5.6.1 Fast Polynomial Multiplication via FFT

The idea to compute the product of two degree d polynomials in coefficient representation is as follows:

1. Evaluate the polynomials in $2d+1$ (distinct) points to create an extended $2d+1$ point value representation of the polynomials.
2. Compute the product of the two polynomials in $\mathcal{O}(d)$ time.
3. Convert the point value representation of the product back to coefficient representation.

We show that using the FFT method (1) and (3) can be done in $\mathcal{O}(d \log d)$ time. Henceforth we will focus on solving the following problem (called *Evaluation*) using FFT:

Problem 5.5 ▶ Evaluation of a polynomial

Given a degree $n-1$ polynomial (in coefficient form), evaluate its value at n distinct points.

Later we will focus on the "opposite" problem (called *Interpolation*):

Problem 5.6 ▸ Interpolation of a polynomial

Given the values of a degree $n-1$ polynomial at n distinct points, find the coefficient vector of the polynomial.

Key idea of FFT

The key idea of FFT is to evaluate a degree $n-1$ polynomial, say,

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_ix^i + \cdots + a_{n-1}x^{n-1}$$

not at some arbitrary set of n distinct points, but rather at a very special set of n distinct points which have a peculiar property. For the following discussion, we will assume throughout that n is a power of 2 (this assumption can be easily removed – how?). To understand the idea, let us consider evaluating $A(x)$ at two points, say x_0 and $-x_0$. We will first split $A(x)$ into $A^{[0]}(x)$ and $A^{[1]}(x)$ which are polynomials with even and odd coefficients of $A(x)$ respectively:

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$

Then

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

Thus, $A(x_0) = A^{[0]}(x_0^2) + xA^{[1]}(x_0^2)$ and $A(-x_0) = A^{[0]}(x_0^2) - xA^{[1]}(x_0^2)$. Note that the above two values involve lot of overlapping computations: we need to perform evaluations of two polynomials, namely $A^{[0]}(x')$ and $A^{[1]}(x')$ at the point $x' = x_0^2$. This is exactly a divide and conquer type operation, assuming that the n distinct points come in pairs, i.e., $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$. Note that we need this trend to continue in *each level of the recursion*, and not just at the topmost level: in other words, we need the $n/2$ points in the next level, namely, $x_0^2, x_1^2, \ldots, x_{n/2}^2$ to also come in ± pairs. What special numbers can guarantee this property: enter complex numbers, in particular, the n complex roots of unity! See Figure 5.9.

Complex roots of unity

A complex number w is an nth root of unity if

$$w^n = 1$$

There are n complex nth roots of unity given by

$$e^{\frac{2\pi ik}{n}} \text{ for } k = 0, 1, \ldots, n-1$$

were $e^{iu} = \cos(u) + i\sin(u)$ and $i = \sqrt{-1}$. See Figure 5.8.
The **principal** nth root of unity is

$$w_n = e^{2\pi i/n}$$

the other roots are *powers* of w_n, i.e., $w_n^2, w_n^3, \ldots, w_n^{n-1}, w_n^n = 1$. Exercise 5.14 asks you to show that that that $1, w_n, w_n^2, w_n^3, \ldots, w_n^{n-1}$ are the n distinct complex nth roots of unity.

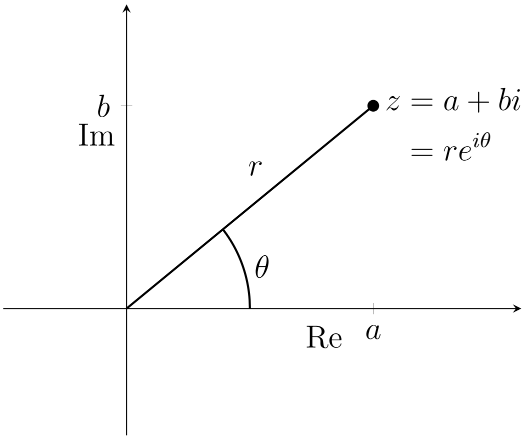


Figure 5.6: Complex plane. A complex number $z = a + bi$ is represented. It can be represented in two equivalent ways: (1) as coordinates (a, b) where a denotes the real part and b the imaginary part (2) as polar coordinates (r, θ), where r is the modulus (distance from origin) and θ is the amplitude (angle made with the real axis). $r = \sqrt{a^2 + b^2}$ and $\theta \in [0, 2\pi)$, where $cos(\theta) = a/r$ and $\sin(\theta) = b/r$. Thus $z = r(\cos(\theta) + i\sin(\theta)) = re^{i\theta}$ by Euler's formula.

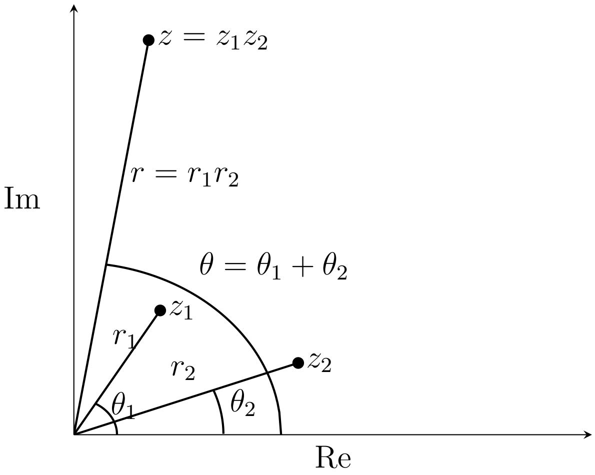


Figure 5.7: One can multiply complex numbers easily in polar form: $r_1e^{i\theta_1}r_2e^{i\theta_2} = r_1r_2e^{i(\theta_1+\theta_2)}$, i.e., by multiplying the moduli and rotating the first angle by the second angle. Notice that $z = z_1z_2$ has modulus r_1r2 and angle $\theta_1 + \theta_2$

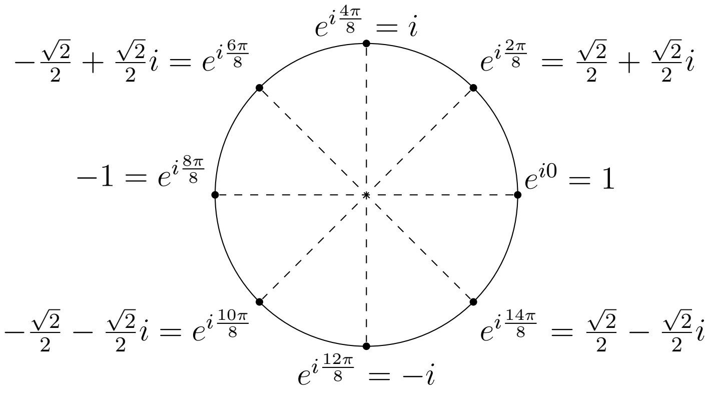

$e^{i\frac{4\pi}{8}} = i$
$-\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i = e^{i\frac{6\pi}{8}}$
$e^{i\frac{2\pi}{8}} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i$
$-1 = e^{i\frac{8\pi}{8}}$
$e^{i0} = 1$
$-\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i = e^{i\frac{10\pi}{8}}$
$e^{i\frac{14\pi}{8}} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i$
$e^{i\frac{12\pi}{8}} = -i$

Figure 5.8: The complex 8th roots of unity represented in the complex plane. They are all solutions to $z^8 = 1$. Letting $z = re^{i\theta}$, the solutions to this equation are $e^{2k\pi/8}$, for $k = 0, \dots, 7$. Note that the solutions come in positive-negative pairs — these are separated by angle $\pi(180°)$. Note that $e^{i(\theta+\pi)} = e^{i\theta}e^{i\pi} = -e^{i\theta}$. The squares of the 8th roots of unity are the 4th roots of unity $(+1, i, -1, -i)$ which also come in ± pairs — this is the crucial property exploited by the divide and conquer FFT algorithm.

Operations on the roots of unity

For any j and k:

$$w_n^k w_n^j = w_n^{j+k}$$

Since $w_n^n = 1$

$$w_n^k w_n^j = w_n^{j+k} = w_n^{(j+k) \bmod n}$$

and

$$w_n^{-k} = w_n^{n-k}$$

5.6.2 Discrete Fourier Transform (DFT)

The **Discrete Fourier Transform (DFT)** of a coefficient vector $\boldsymbol{a} = (a_0, a_1, \dots, a_{n-1})$ is a vector $\boldsymbol{y} = (y_0, y_1, \dots, y_{n-1})$ such that

$$y_k = A(w_n^k) = \sum_{j=0}^{n-1} a_j w_n^{kj}$$

and we can write the vector $\boldsymbol{y}$ as:

$$\boldsymbol{y} = DFT_n(\boldsymbol{a})$$

Using **Fast Fourier Transform (FFT)** we can compute $DFT_n(\boldsymbol{a})$ in $\mathcal{O}(n \log n)$ steps, instead of $\mathcal{O}(n^2)$.

FFT

Assume that n is a power of 2 (otherwise complete to the nearest power of 2).

Given the polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ we define two polynomials as done earlier:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \dots + a_{n-2} x^{n/2-1}$$

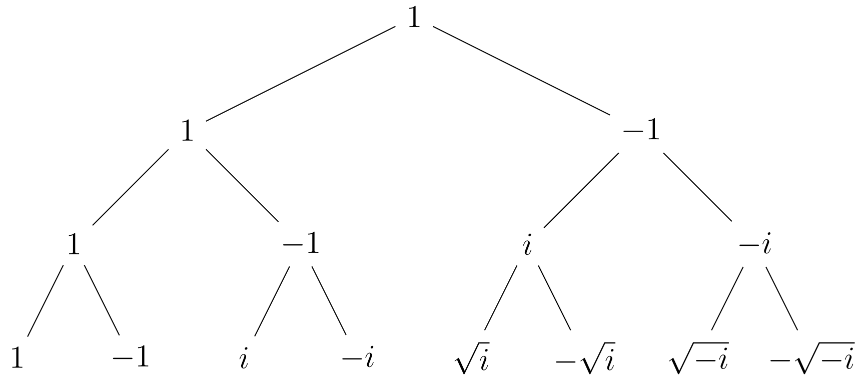


Figure 5.9: Showing the 8th-roots of unity and how the recurrence proceeds on them. The second level represents the square roots of unity, the third level the 4th-roots of unity, and the fourth level the 8th roots of unity. At each level, the roots are grouped into positive-negative pairs; for each pair only one of them need to be solved in the next level of recursion. Each level is obtained by taking successive square roots. At the top level we evaluate the polynomial at the 8 points corresponding to the 8th-roots of unity. This is accomplished by evaluating at the polynomial (recursively) at the 4 points corresponding to the 4th-roots of unity and so on.

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

Then

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

To compute $DFT_n(\boldsymbol{a})$ (on n points) we need to compute the polynomials $A^{[0]}(y)$ and $A^{[1]}(y)$ on $n/2$ points

$$(w_n^0)^2, (w_n^1)^2, \ldots, (w_n^{n-1})^2$$

Theorem 5.7

The set $\left(w_n^0\right)^2, \left(w_n^1\right)^2, \ldots, \left(w_n^{n-1}\right)^2$ contains only $n/2$ distinct points. In other words, the n complex nth roots of unity come in positive-negative pairs. In particular, $w_n^{k+n/2} = -w_n^k$ and $\left(w_n^{k+n/2}\right)^2 = \left(w_n^k\right)^2 = w_{n/2}^k$.

Proof. We will show that the squares of n complex nth roots of unity are the $n/2$ complex $n/2$-th roots of unity. Assume that $k \leqslant \frac{n}{2}$.

$$\left(w_n^k\right)^2 = (e^{2\pi ik/n})^2 = e^{(2\pi ik)/(n/2)} = w_{n/2}^k$$

$$\begin{aligned}\left(w_n^{k+n/2}\right)^2 &= (e^{2\pi i(k+n/2)/n})^2 \\ &= e^{2\pi in/n} e^{(2\pi ik)/(n/2)} \\ &= \left(w_n^1\right)^n w_{n/2}^k \\ &= w_{n/2}^k\end{aligned}$$

Thus, $w_n^{k+n/2} = -w_n^k$. □

Computing the $DFT_n(a)$ is reduced to (see Algorithm 21 for pseudocode):

1. Computing two $DFT_{n/2}$.
2. Combining the results:

   Given $y_k^{[0]} = A^{[0]}\left(w_{n/2}^k\right) = A^{[0]}\left(\left(w_n^k\right)^2\right)$ and $y_k^{[1]} = A^{[1]}\left(w_{n/2}^k\right) = A^{[1]}\left(\left(w_n^k\right)^2\right)$, for $k \leqslant n/2$. For $k < n/2$, by Theorem 5.7, $w_n^{k+n/2} = -w_n^k$, $y_{k+n/2}^{[0]} = y_k^{[0]}$, and $y_{k+n/2}^{[1]} = y_k^{[1]}$. Thus,

$$\begin{aligned} y_k &= y_k^{[0]} + w_n^k y_k^{[1]} \\ y_{k+n/2} &= y_k^{[0]} + w_n^{k+n/2} y_k^{[1]} \\ &= y_k^{[0]} - w_n^k y_k^{[1]} \end{aligned}$$

Algorithm 21 DFT – Discrete Fourier Transform
Input: A vector $\boldsymbol{a}$ of length n (power of 2)
Output: A vector $\boldsymbol{y}$ such that $y_k = \sum_{i=0}^{n-1} a_i (w_n^k)^i$

```
func DFT($\boldsymbol{a}$):
    if $n == 1$:
        return $a_0$
    else:
        $w_n = e^{2\pi i/n}$
        $\boldsymbol{a}_{\text{even}} = (a_0, a_2, \dots)$
        $\boldsymbol{a}_{\text{odd}} = (a_1, a_3, \dots)$
        $\boldsymbol{b} = \text{DFT}(\boldsymbol{a}_{\text{even}})$
        $\boldsymbol{c} = \text{DFT}(\boldsymbol{a}_{\text{odd}})$
        for $k = 0$ to $n/2 - 1$:
            $y_k = b_k + w_n^k c_k$
            $y_{k+n/2} = b_k - w_n^k c_k$
        return $\boldsymbol{y}$
```

Hence, we have the recurrence

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

which yields the solution $T(n) = \mathcal{O}(n \log n)$.

Theorem 5.8

A point value representation of a nth degree polynomial given in a coefficient representation can be generated in $\mathcal{O}(n \log n)$ time.

5.6.3 From point-value to coefficient representation: Interpolation

Given the DFT $y = (y_0, \dots, y_{n-1})$ of a degree n polynomial we want to generate the coefficient representation $a = (a_0, \dots, a_{n-1})$ of the polynomial.

We need to solve

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \dots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

or $y = V_n a$. We solve for a: $a = V_n^{-1} y$. It turns out that the entries of matrix V_n^{-1} can be easily calculated as shown by the following theorem.

Theorem 5.9

The (i, j) entry in V_n^{-1} is $\frac{w_n^{-ij}}{n}$.

Proof. We show that $V_n^{-1} V_n = I_n$:

The (j, j') entry of $V_n^{-1} V_n$

$$\begin{aligned} [V_n^{-1} V_n]_{j,j'} &= \sum_{k=0}^{n-1} \frac{w_n^{-kj}}{n} (w_n^{kj'}) \\ &= \frac{1}{n} \sum_{k=0}^{n-1} w_n^{-k(j-j')} \end{aligned}$$

If $j = j'$ the summation is 1.

If $j \neq j'$

$$\begin{aligned} \sum_{k=0}^{n-1} w_n^{-k(j-j')} &= \sum_{k=0}^{n-1} (w_n^{j'-j})^k \\ &= \frac{(w_n^{j'-j})^n - 1}{w_n^{j'-j} - 1} \\ &= \frac{(w_n^n)^{j'-j} - 1}{w_n^{j'-j} - 1} \\ &= \frac{(1)^{j'-j} - 1}{w_n^{j'-j} - 1} \\ &= 0 \end{aligned}$$

□

Thus, we need to compute

$$a_i = \frac{1}{n} \sum_{k=0}^{n-1} y_k w_n^{-ki}$$

Note that this is similar to the evaluation algorithm, but instead of using $w_n = e^{2\pi i/n}$, we use $w_n^{-1} = e^{-2\pi i/n}$ as the "principal" root of unity. This can again be *computed by using the FFT algorithm* in $\mathcal{O}(n \log n)$ time. Note that we have to divide by a factor of $1/n$ to get the final answer.

Theorem 5.10

Given a point value representation of an n degree polynomial in nth roots of unity, the coefficient representation of that polynomial can be computed in $\mathcal{O}(n \log n)$ time.

Since we can do both evaluation and interpolation in $\mathcal{O}(n \log n)$ time, we have the following theorem.

Theorem 5.11

The product of two n-degree polynomials can be computed in $\mathcal{O}(n \log n)$ time.

5.6.4 A Matrix-based View of Fourier Transform

There is a natural way to view FFT using matrix operations and linear algebra.

The FFT is evaluating a given degree $n-1$ polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_i x^i + \cdots + a_{n-1}x^{n-1}$, at the n complex roots of unity $1, w_n, w_n^2, \ldots, w_n^{n-1}$, respectively, yielding $y_i = A(w_n^i)$. As we saw in Section 5.6.3, this operation can be cast as a multiplying a matrix V_n with the (column) vector $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$, yielding the evaluation vector $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$. We already saw that V_n is a Vandermonde matrix and hence invertible. So $y = V_n a$ and $a = V_n^{-1} y$, in other words, evaluation is multiplication by V_n and interpolation is multiplication by V_n^{-1}. It is easy to describe V_n: its (i,j)th entry is w^{jk}.

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n & \omega_n^2 & \ldots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \ldots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \ldots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_n^{2(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

The crucial fact of the matrix V_n is that its columns are *orthogonal* to each other (we will show this momentarily). In other words, the columns are at right angles to each other and thus they form axes in a new basis called the *Fourier basis*. Recall that multiplying a vector by an orthogonal matrix is equivalent to *rotating* it. Thus multiplying a vector by V_n is rotating it from the standard basis with the usual set of axes (i.e., the $x, y, z, \ldots,$ axes) into the Fourier basis. Thus FFT (evaluation), which is multiplying by V_n, is *just a change of basis* — a *rigid rotation* of the coefficient vector a in standard basis to the point value vector y in the Fourier basis. Similarly, inverse FFT (interpolation), which is multiplying by V_n^{-1} is just changing the basis of the point value vector from the Fourier basis back to the standard basis to get the coefficient vector.

We now show that the columns of V_n are orthogonal to each other. To see this, consider any two column vectors of V_n: column j vector $(1, w^j, w^{2j}, \ldots, w^{nj})$ and column $k (\neq j)$ vector $(1, w^k, w^{2k}, \ldots, w^{nk})$. Since these are vectors in complex space, to show that two such vectors are orthogonal we have to show that their *inner product* is zero. The inner product of two complex vectors $u = (u_0, u_1, \ldots, u_{n-1})$ and $v = (v_0, v_1, \ldots, v_{n-1})$ is defined

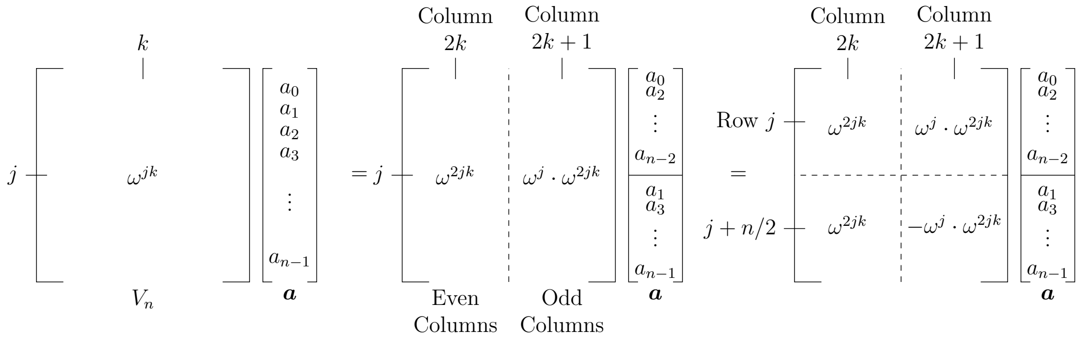

k
j
ω^{jk}
V_n
a_0 a_1 a_2 a_3 ⋮ a_{n-1}
a
= j
Column 2k
Column 2k + 1
ω^{2jk}
$\omega^j \cdot \omega^{2jk}$
Even Columns
Odd Columns
a_0 a_2 ⋮ a_{n-2} a_1 a_3 ⋮ a_{n-1}
a
=
Row j
j + n/2
Column 2k
Column 2k + 1
ω^{2jk}
$\omega^j \cdot \omega^{2jk}$
ω^{2jk}
$-\omega^j \cdot \omega^{2jk}$
a_0 a_2 ⋮ a_{n-2} a_1 a_3 ⋮ a_{n-1}
a

Figure 5.10: FFT as a matrix-based divide and conquer algorithm.

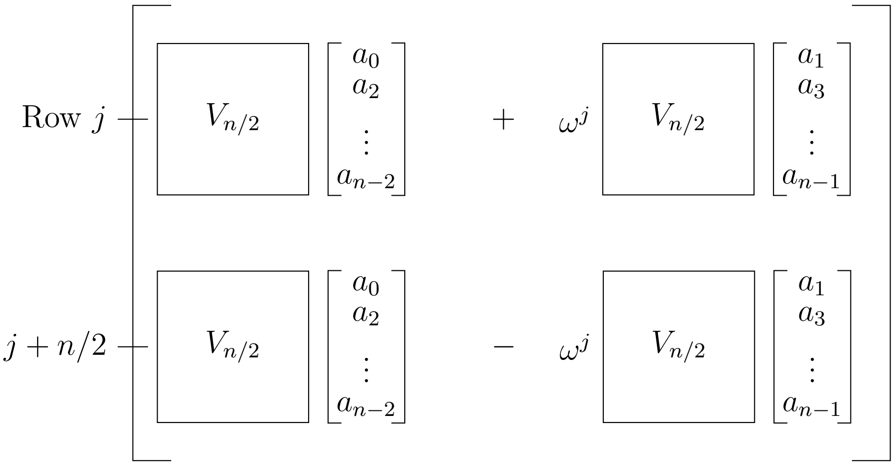


Figure 5.11: FFT gives rise to two matrix multiplication subproblems.

as $u_0v_0^* + u_1v_1^*, \ldots, u_{n-1}v_{n-1}^*$, where v_i^* is the *complex conjugate* of v_i.[4] In particular, the complex conjugate of w^{ik} is w^{-ik}. Thus we have to show that the inner product of columns j and k is zero if $j \neq k$, i.e.,

$$\sum_{i=0}^{n-1} w^{i(j-k)} = 0.$$

This is precisely what we have already showed in Theorem 5.9. Moreover, it showed that if $j = k$, the inner product is n. Thus we can write that $V_nV_n^* = nI$, where V_n is the complex conjugate of V_n which is obtained by taking the complex conjugates of each of its entries (note that the (i, j)th entry of V_n is w^{ij} and the corresponding entry of V_n^* is simply w^{-ij}). Hence the inverse of V_n, i.e., $V_n^{-1} = V_n^*/n$.[5] Thus interpolation is just multiplying by V_n^*/n.

We can also view the divide and conquer FFT algorithm from a matrix standpoint. As mentioned earlier FFT is multiplying a by V_n. This matrix multiplication can be decomposed into two matrix multiplications subproblems — operating on the even entries of a and the other on the odd entries of a, each being multiplied by $V_{n/2}$. See Figures 5.10 and 5.11.

[4] The complex conjugate of complex number $a + ib$ is defined as $a - ib$ or in polar notation, the complex conjugate of $re^{i\theta}$ is $re^{-i\theta}$.

[5] An orthogonal matrix or an orthonormal matrix (i.e., an orthogonal matrix where all the columns are unit vectors) A has the property that $AA^T = I$, i.e., the inverse is its transpose. If A is complex, then its inverse is its conjugate transpose which is what we see here.

5.7 Worked Exercises

Worked Exercise 5.1. Given two SORTED (in ascending order) arrays $A[1..n]$ and $B[1..n]$, each containing n numbers, give an $\mathcal{O}(\log n)$-time algorithm (counting the number of comparisons) to find the median of all the $2n$ elements in arrays A and B.

Solution:

We need to find the median of the combined arrays A and B, in other words, we want to find the nth smallest of the $2n$ combined elements.

The main idea is to compare the median element of A to the median element of B: If the former is greater, then the elements in the larger half of A can be thrown away (for being too big — their ranks exceed n), and the elements in the smaller half of B can also be thrown away (for being too small — their ranks are surely smaller than n); in other words we recurse on the smaller half of A and the larger half of B. If it is the median of B that is greater than the median of A then we simply interchange the roles of A and B in the previous sentence. The reason it is correct to recurse in this way is that the element we are searching for has, in the recursive call, "lost" $n/2$ elements smaller than it and $n/2$ elements greater than it (the halves of A and B that were "thrown away"), and therefore its rank in the reduced-size problem is $n - (n/2) = n/2$. A more formal exposition is given below.

We denote the ith element in set A as $A[i]$ and the ith element in set B as $B[i]$. We will assume that n is a power of 2 (the algorithm can be extended if this is not the case).

We define $A_1 = \{A[1], A[2], \ldots, A[n/2-1]\}$ and $A_2 = \{A[n/2+1], \ldots, A[n]\}$ as A_2. Notice that $A = A_1 \cup \{A[n/2]\} \cup A_2$. Similarly, we can define $B_1 = \{B[1], B[2], \ldots, B[n/2-1]\}$ and $B_2 = \{B[n/2+1], \ldots, B[n]\}$.

We first compare the median number of the two sets, i.e., $A[n/2]$ and $B[n/2]$. There are three possible results from the comparison:

1. $A[n/2] = B[n/2]$. In this case, the nth smallest element of all $2n$ elements is $A[n/2]$. This is based on the observation that $n-1$ elements (i.e., any element in A_1 or B_1, and $B[n/2]$) are no greater than $A[n/2]$ and all other elements (i.e., any element in A_2 or B_2) are no less than $A[n/2]$.

2. $A[n/2] > B[n/2]$. In this case, since any element in A_2 is no less than at least n elements (i.e., any element in A_1 or B_1, $A[n/2]$, and $B[n/2]$), it cannot be the nth smallest element. Similarly, since any element in B_1 and $B[n/2]$ is no greater than at least $n+1$ elements (i.e., any element in A_2 or B_2, and $A[n/2]$), it cannot be the nth smallest element. Therefore, the nth smallest element in all $2n$ elements should be the $n/2$-th smallest element in $A_1 \cup \{A[n/2]\}$ and B_2, both of which are of size $n/2$.

3. $A[n/2] < B[n/2]$. Using similar argument, we can prove that the nth smallest element in all $2n$ elements should be the $n/2$-th smallest element in A_2 and $B_1 \cup \{B[n/2]\}$, both of which are of size $n/2$.

In case 1, the nth smallest element can be found in constant time. In either case 2 or case 3, we reduced the problem to the **same problem** with 1/2 size. There is constant work for the transformation (only one comparison of the median elements of the two sets). Let the time for solving the problem to be $T(n)$. Then, based on the above argument, we have

$$T(n) = T(n/2) + 1$$

Applying The DC Recurrence Theorem, $T(n)$ is thus $\mathcal{O}(\log n)$.

Worked Exercise 5.2. Given n points on the x-axis line (in some arbitrary order), find the closest pair of points in $\mathcal{O}(n \log n)$ time.

(a) Show that the problem can be solved using sorting.

(b) Give a divide and conquer algorithm that does not use sorting. (The advantage of this approach is that it generalizes to points in higher dimensions, e.g., 2 dimensions). (Hint: Partition the points into two subproblems of equal size, recursively solve the subproblems, and efficiently combine them to solve the original problem; exploit the fact that the recursion returns the closest pair in the respective subproblems.)

Solution.

(a) The problem can be solved by sorting the points and by examining the distance between successive pairs of points and taking the pair with the minimum distance. Clearly, the closest pair will occur consecutively in the sorted order.

(b) The main idea is to use the Selection algorithm that we learnt in class to partition the problem into two equal parts. In particular, we call Select algorithm (Algorithm 19) and find the median of the points. The median will lie (approximately) in the middle and we partition the points about the median, i.e., all points smaller than the median will be to the left of the median and the rest of the points to the right. Let L be the set of points to the left of the median and R be the set of points to the right of the median. (Note that the median point is not included in R and S.) Recursively solve the closest pair on two subproblems of (almost) equal size. When the recursion returns, the respective closest pair of points in both R and S are returned. (The base/termination case is when there are just one or two points, in which case, we just return the points.) Compare the distances between these two pairs, i.e., the closest pair of the left and the closest pair of the right and take the minimum among them. Of course, this may not be the closest pair of the entire set, because, we have not compared the median with the rest. Compare the distance between the median and each of the other points to find if there is a closer pair. Note that we do not have to compare any other pair (i.e., between points in the left and right of median) — Why?

Note that the above algorithm does not involve sorting.

The recurrence is $T(n) \leqslant 2T(n/2) + \mathcal{O}(n)$ since the total cost of finding the median, partitioning, and the final checking of all points with the median is $\mathcal{O}(n)$.

The solution to the above recurrence is $\mathcal{O}(n \log n)$, by the recurrence theorem.

It is easy to write a pseudocode based on the above description (left to you to do).

□

Worked Exercise 5.3. We are given a *sequence* of n positive numbers $a_1, \ldots, a_n$ and a fixed number $k > 0$. We want to find a pair of numbers a_i and a_j such that $j > i$ and $a_j - a_i = k$. If there is no such pair we want to output "false". This can be solved easily in $\Theta(n^2)$ time by examining all pairs of numbers. Your task is to give a divide and conquer algorithm that runs in $\mathcal{O}(n \log n)$ time. Show the correctness of your algorithm and the analysis of its running time.

Solution. The main idea is to use the divide and conquer strategy that is used in MergeSort. Let the time to solve this problem and at the same time **sort** the sequence S of numbers to be $T(n)$, where n is the number of numbers in the sequence. Note that we do the "extra work" of sorting, although this is not required for the solution; however, sorting is important to get $\mathcal{O}(n \log n)$ run time.

Using divide and conquer, we divide the sequence into two sub-sequences S_1 (the first half) and S_2 (the second half), each of which is of length $n/2$ (assume, without loss of generality, that n is even).

We solve recursively the two subsequences. Note by our assumption, when the algorithm returns, it also leaves the sequence sorted (besides the answer to the subproblem). If such a pair exists in either S_1 or S_2 we return it. If not, we have to find whether their exists a pair x, y, such that $y - x = k$, and $x \in S_1$ and $y \in S_2$ (note that the index of all elements in S_2 will be greater than S_1). We also note that when the recursion returns S_1 and S_2 are sorted. Using the fact that S_1 and S_2 are sorted, we can find whether such a pair x, y exists in $\mathcal{O}(n)$ time.

Let $S_1 = x_1, x_2, \ldots, x_{n/2}$ and $S_2 = y_1, y_2, \ldots, y_{n/2}$. We start by examining the pairs x_1 and y_1. Let x_i and y_j be the currently examined pair.

- If $(y_j - x_i) = k$, then we return the pair.
- If $(y_j - x_i) > k$, then we examine the next pair x_{i+1} and y_j. (If $i = n/2$ then return "false".)
- If $(y_j - x_i) < k$, then we examine the next pair x_i and y_{j+1}. (If $j = n/2$ then return "false").

Note that the above process is very similar to the merging process in MergeSort. In fact, at the same time, we also merge S_1 and S_2 to get a sorted sequence S (as in mergesort). Since each time we advance to the next number in at least one sub-sequence and the total length is n, the time to search for the pair is $\mathcal{O}(n)$. Merging takes $\mathcal{O}(n)$ time as well.

Therefore, we have

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Based on The DC Recurrence Theorem, $T(n) = \mathcal{O}(n \log n)$.

Note that the formal pseudocode is very similar to that of MergeSort as well (please do it!). □

5.8 Exercises

Exercise 5.1. Determine the total number of comparisons that each of the following algorithms takes on $S = [8, 2, 6, 7, 5, 1, 4, 3]$.

- SimpleSort
- MergeSort
- QuickSort

Show the steps of the algorithm when calculating the number of comparisons.

Exercise 5.2. Show the correctness of MergeSort using mathematical induction. Assume that Merge is correct.

Exercise 5.3. Show that the pseudocode Merge correctly implements the Merge procedure. (Hint: Use Induction to argue that each element, starting from the first element, is added in the Temp array in the correct order.)

Exercise 5.4. Show the correctness of QuickSort by using mathematical induction.

Exercise 5.5. Show that QuickSort can take $\Theta(n^2)$ time for sorting a set of n elements and this is the *worst* possible.

Exercise 5.6. Show that every input for the QuickSort algorithm takes $\Omega(n \log n)$ time. (Hint: Show that the solution of the recurrence for the best case of QuickSort given in proof of Theorem 5.2 is $\Omega(n \log n)$ by using induction.)

Exercise 5.7. Show that any comparison-based sorting algorithm (i.e., an algorithm that uses only comparisons between pairs of elements to infer the ordering) needs $\Omega(n \log n)$ comparisons (in the worst-case). (Hint: Represent the execution of a comparison-based sorting algorithm as a *decision* tree. A decision tree is a binary tree, where each node represents a comparison between two elements, say x_i and x_j; the left child is taken by the algorithm if $x_i < x_j$, otherwise the right child is taken. Argue that the number of leaves of this decision tree should be $n!$. Calculate a lower bound on the the depth of this tree (hence the number of comparisons needed).

Exercise 5.8. Show the correctness of Algorithm Select.

Exercise 5.9. Given three SORTED (in ascending order) arrays $A[1..n], B[1..n]$, and $C[1..n]$, each containing n numbers, give an $\mathcal{O}(\log n)$-time algorithm (again, counting the number of comparisons) to find the nth smallest number of all $3n$ elements in arrays A, B, and C.

Exercise 5.10. Show how to modify QuickSort so that it runs in worst case $\mathcal{O}(n \log n)$ time. Write the pseudocode of the modified algorithm and analyze its worst-case run time. (Hint: The modification is choosing the pivot in a better way. How?)

Exercise 5.11. Write the complete pseudocode for the divide and conquer algorithm discussed in Section 5.3 for multiplying two integers (do not forget the base case!).

Exercise 5.12. Show that the solution to the recurrence $T(n) \leqslant 3T(n/2 + 1) + \mathcal{O}(n)$ (with constant base case) is $T(n) = \mathcal{O}\left(n^{\log_2 3}\right)$.

Exercise 5.13. Show that the determinant of the Vandermonde matrix

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{d-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{d-1} & x_{d-1}^2 & \dots & x_{d-1}^{d-1} \end{pmatrix}$$

is

$$\prod_{j<k} (x_k - x_j)$$

(Hint: Use induction.)

Exercise 5.14. Show that $1, w_n, w_n^2, w_n^3, \dots, w_n^{n-1}$ are the n distinct complex nth roots of unity.

Exercise 5.15. Suppose you are working for a government organization that enforces copyright laws. You are given a set of n files and your task is to determine whether at least $n/2$ of them are identical (this is flagged as a potential copyright infringement). The only tool you have is a machine that can answer the following type of query: given two files, are they the same or not? Your goal is to design a divide and conquer algorithm that takes $\mathcal{O}(n \log n)$ queries. (Hint: (1) To have at least $n/2$ identical files in the set, at least one of the halves of the set should contain at least $n/4$ of them. (2) When you solve a subproblem, you may want to return more than just a boolean indicating whether that subproblem should be flagged as a potential copyright infringement.)

Exercise 5.16. You are given an $n \times n$ matrix represented as a 2-dimensional array $A[][]$ (there are totally n^2 elements). Each row of A is sorted in increasing order and each column of A is sorted in increasing order. (Other than these properties, the elements of A are arbitrary.) Your goal is to find whether some given element x is in A or not. Note that you can do only comparisons between elements.

- Give an algorithm that takes $\mathcal{O}(n)$ comparisons.
- Can we solve the problem using asymptotically lesser number of comparisons, i.e., in $o(n)$ comparisons?

Exercise 5.17. We are given an array A containing n numbers, and an integer k that divides n. We want to find k numbers x_1,x_2, …,x_k of A such that x_i is the $(\frac{n}{k}i)$th smallest element of A. Develop a divide and conquer algorithm that finds the x_i's that runs in time $\mathcal{O}(n \lg k)$. Prove its correctness and running time.

Exercise 5.18. Consider the *max-sum* problem: Given an array A of size n containing positive and negative integers, determine indices i and j, $1 \leqslant i \leqslant j \leqslant n$, such that $A[i] + A[i+1] + \cdots + A[j]$ is a maximum.

(a) Consider the array $A = [5, 10, -15, 20, -4, 6, 4, 8, -10, 20]$. Find the indices i and j that give the maximum sum and state the maximum sum.

(b) Give a *divide and conquer* algorithm that runs in $\mathcal{O}(n \log n)$ time. Assume, that adding or comparing two numbers takes constant time.

(Hint: Split the array into two (almost) equal parts and recursively solve the problem on the two parts. The non-trivial part is combining the solutions — note that it is possible that the indices i and j that give the optimal sum might be on opposite parts.)

Exercise 5.19. Recall that the Fibonacci numbers are defined by the following recurrence: $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geqslant 2$.

1. Using the above recurrence, give a simple algorithm to compute F_n, given n. The complexity of the algorithm in terms of the number of *arithmetic operations* should be $\mathcal{O}(n^2)$. Show that the algorithm indeed has this complexity. Recall that an arithmetic operation is the cost of adding or multiplying two single digit numbers. (Hint: How many digits are there in F_n? Use the fact that F_n is $\Theta\Big((\frac{1+\sqrt{5}}{2})^n\Big)$.
2. Our goal is to use the following matrix identity to compute the nth Fibonacci number more efficiently.

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

(a) Show that

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

(b) Show that $\mathcal{O}(\log n)$ matrix multiplications are enough to compute F_n.

(c) Using the fast algorithm for multiplying two integers (Karatsuba's algorithm) show that F_n can be computed in $\mathcal{O}\left(n^{\log_2 3} \log n\right)$ arithmetic operations.

(d) Can you improve your analysis of (c) and show a cost of $\mathcal{O}\left(n^{\log_2 3}\right)$ arithmetic operations?

Exercise 5.20.

(a) Suppose we have an integer array A of length n. Our goal is to sort A so that negative numbers appear before non-negative numbers. For example, if $A = [-2, 4, 7, 0, 6, -1]$, one valid sort would be $[-2, -1, 7, 4, 0, 6]$. Devise a linear time algorithm that can accomplish this using $\mathcal{O}(1)$ additional space.

(b) Consider a generalization of the above problem. Instead of only negative and positive numbers (i.e., numbers in 2 categories), suppose we have k categories of numbers ($2 \leqslant k < n$) in array A of length n: category 1 consists of numbers less than c_1, category 2 of numbers belong to interval $[c_1, c_2)$, and, in general, category i ($i < k$) of numbers belong to interval $[c_{i-1}, c_i]$, and category k of numbers bigger than c_{k-1} ($c_1 < c_2 < \cdots < c_{k-1}$ are arbitrary fixed constants). The goal is to sort the numbers so that all numbers in a smaller category come before numbers in a higher category (numbers within a category can come in any order in the sorted array).

Give an efficient divide and conquer algorithm that sorts the array; assume that k and the constants $c_1, \ldots, c_{k-1}$ are known. What is the running time of your algorithm? Your algorithm's running time should depend on both n (the size of the array) and k (the number of categories). When k is constant, the run time should be $O(n)$ and if $k = \Theta(n)$, it should be $O(n \log n)$. Prove the correctness of your algorithm, analyze the run time, and provide a brief explanation of the space complexity.

Exercise 5.21. Given an unsorted array $A[1, \ldots, n]$ and an element p (that may or may not belong to the array), give an $O(n)$ time algorithm to partition A into two subarrays $A[1, \ldots, m]$ and $A[m+1, \ldots, n]$ such that all elements in the first subarray are less than p and all elements in the second subarray are greater than or equal to p. Your algorithms should do the partition *in place*, i.e., without using any auxiliary array (using only $O(1)$ additional memory). Your algorithm should output the index m.

Exercise 5.22. Show that the solution of the recurrence $T(n) = 2T(n/2) + O(n \log n)$ is $T(n) = O(n \log^2 n)$. Assume constant base cases.

Exercise 5.23. You are given an unsorted array of $n \geqslant 1$ integers. Give an efficient divide and conquer algorithm to find if an element in the array occurs more than 3 times, and if so, output the element. Show the correctness of your algorithm and analyze its run time.

Exercise 5.24. You are given 2 sorted arrays of size n_1 and n_2 respectively. Give an $O(\log n_1 + \log n_2)$ time algorithm for computing the median element in the union of the two arrays.

6 DYNAMIC PROGRAMMING

In the previous chapters, we studied the technique of divide and conquer which gives efficient algorithms for many problems. In this chapter, we focus on another powerful technique for solving problems called *dynamic programming (DP)*.

6.1 Divide and Conquer vs. DP

As we saw in divide and conquer, the basic strategy in dynamic programming is again breaking problems into subproblems and combining their solutions to obtain the solution to the original problem. However, there are differences between the two strategies. While in divide and conquer, the problem is decomposed into *independent* and *disjoint* subproblems, in dynamic programming, it is almost always decomposed into *independent* and *overlapping* subproblems. Independent means the solution to one subproblem does not affect the solution to another subproblem. Thus the solution to each subproblem can be computed independently of one and another. Disjoint means that each subproblem is different; in many examples, the subproblems operate on different parts of the input and hence are naturally disjoint. For example, in mergesort, a divide and conquer algorithm for sorting (Section 5.1.2), the input array is partitioned into two (almost) equal and disjoint subarrays and the subproblem is solved in each subarray recursively. Since the subarrays are disjoint, the subproblems are disjoint. In other cases, the subproblems may share some input, but they are different subproblems. For example, in the linear-time selection algorithm (Section 5.2), the subproblem of finding the "median of medians" shares some input values with the other two subproblems one of either side of the pivot (note that only one of these will actually execute); however, the subproblems are different and hence disjoint.

On the other hand, a key ingredient of dynamic programming is that the subproblems are overlapping, i.e., the *same subproblem occurs in the solution of other subproblems*. A main point of dynamic programming is to solve each subproblem *only once* (there is no point in solving the same subproblem again and again which is inefficient!). We will illustrate the technique with various examples throughout this chapter.

Dynamic programming (DP) is generally a more difficult technique than divide and conquer. The main reason is that, in many applications of DP, the difficulty is in figuring out how to decompose a given problem into subproblems. Thus the nontrivial work in a

DP solution is figuring out this decomposition.

The best way to state the decomposition is by using a recursive formula. The recursive formula gives the solution of the problem in terms of the solutions of its subproblems. It is easy to show the correctness of the recursive formula by using induction. Translating the recursive formula into a pseudocode is also fairly straightforward — the only care that has to be taken is that each (distinct) subproblem has to be solved *exactly once* since the same subproblem can arise in the solution of other subproblems, i.e., overlapping subproblems. Again, we will see how to do this translation with the help of various examples. The good news is that once we have the correct recursive formulation, an efficient implementation (that solves each subproblem only once) is rather standard and "mechanical" to write. However, coming with a correct recursive formulation (that leads to a correct and efficient algorithm) is usually an "art". Nevertheless, there are a few "patterns" in coming with this recursive formulation which occurs repeatedly in many problems. Studying these patterns will help in designing DP algorithms. We will see these patterns in this chapter.

6.2 Optimization problems

Dynamic programming is especially a powerful technique for solving *optimization* problems. In optimization problems (very much like in search problems that we saw in Chapter 2) the goal is to find (search for) an optimum (e.g., maximum or minimum) solution to a problem specified on an input domain. Usually the problem comes with some constraints and the goal is to find an optimal solution that satisfies the problem's constraints. For example, consider the *max-sum* problem (see Exercise 5.18) stated as follows:

Problem 6.1 ▶ Maxsum

Given an array A of size n containing positive and negative integers, determine indices i and j, $1 \leqslant i \leqslant j \leqslant n$, such that $A[i] + A[i+1] + \cdots + A[j]$ is a maximum.

In other words, the goal is to find *contiguous* array values $A[i], A[i+1], \ldots, A[j]$ such that their sum — $A[i] + A[i+1] + \cdots + A[j]$ — is maximized. This is an optimization problem since we want to optimize, i.e., maximize, some value (i.e., the sum) over an input domain. The input domain for this is the set of all pair of indices i and j, $1 \leqslant i \leqslant j \leqslant n$, with each pair giving rise to a sum. Thus there are n^2 such sums. Note that all these sums are "potential" solutions for the problem (the optimum is the one which is the maximum sum). These are called *feasible* solutions. Feasible solutions are those that satisfy the constraints of the problem. Here the constraint is that the array sum has to be over a contiguous subarray, i.e., it includes all array values between indices i and j (including $A[i]$ and $A[j]$). It is easy to see that the problem can be solved by calculating all the possible n^2 array sums and taking the maximum among those. This takes at least n^2 operations; actually, even more since to compute a particular sum $A[i] + \cdots + A[j]$ requires $j - i$ addition operations. It is not difficult to calculate that the total number of addition operations over all the $\Theta(n^2)$ array sums is $\Theta(n^3)$. Using divide and conquer (Exercise 5.18) one can design an (somewhat complicated) algorithm that takes $\mathcal{O}(n \log n)$ operations. We will see that using DP we can come with a simple and elegant $\mathcal{O}(n)$ *time* algorithm which is also easy to implement. Thus DP generally gives rise to efficient algorithms for problems that otherwise are inefficient to solve.

6.3 Illustrating the DP Technique using the Maxsum Problem

We will illustrate the key steps involved in solving a problem using DP by using the maxsum problem as an example.

Step 1: Understand the problem.

As discussed in Section 2.2.1, this is crucial for solving any problem. In this step, we figure out the input and output of the problem and understand the constraints (if any). In the maxsum problem, the input is an array of values and the output is a contiguous subarray, i.e., $A[i], A[i+1], \dots, A[j]$ of values such that the sum $A[i] + \cdots + A[j]$ is maximized (among all possible such subarrays). How many such subarrays are there? There is a subarray for every pair of distinct indices i and j, which can be chosen in $\binom{n}{2}$ different ways, hence there are so many subproblems. The only constraint is that the subarray is contiguous.

Step 2: Focus first on the value of the solution.

It is helpful in many cases (especially optimization problems) to focus on the value (maximum or minimum) of the solution rather than the solution itself. This simplifies the next steps. For example, in the maxsum problem, we will first focus on finding the maxsum value rather than the subarray that gives that maximum sum. Typically, it is easy to extend the DP solution that gives the maximum sum to also give the solution that has this maximum value.

Step 3: Formulate subproblems.

As mentioned in Step 2, we will first focus on finding the value of the maximum sum, rather than the subarray which gives that maximum sum. Later, we will discuss how to extend this to finding the subarray as well.

What are the subproblems? This is an important step in DP. Figuring out the right subproblems usually leads to the correct recursive formulation which captures the DP solution. What are the subproblems for the maxsum problem? Of course, a straightforward way is to have a subproblem for each subarray, i.e., for each pair for indices i and j ($1 \leqslant i \leqslant j \leqslant n$), we have a subproblem of computing the maxsum restricted to that subarray. While this is a valid decomposition, this is not efficient, because there are n^2 subproblems. In a DP solution, the total time is at least proportional to the number of subproblems (since it takes at least constant time to solve a subproblem) and hence the running time with this decomposition is at least n^2. If we are shooting for an $\mathcal{O}(n)$ time algorithm, the number of subproblems have to be $\mathcal{O}(n)$ as well. Generally, the *smaller the total number of subproblems, the better.* How to come with such a decomposition?

This is the tricky part. One helpful idea in finding subproblems is to introduce *auxiliary variables*. Here is a first attempt keeping in mind that we are shooting for $\mathcal{O}(n)$ subproblems. Let $M[i]$ be the solution to the max-sum problem considering only the *first i elements* of the array A. That is, we introduce the auxiliary variable i that restricts the array to only the first i elements. Clearly, the original problem is one of the subproblems, i.e., $M[n]$. Is this a good decomposition? To answer this, we need to check if we can write a recursive formulation for $M[i]$ in terms of subproblems of smaller size, i.e., $M[i-1]$ and below. Can we write a recursive solution to $M[i]$ based on $M[i-1]$. Note that it is not enough to know just the value of $M[i-1]$, but we also need to know if *the solution to* $M[i-1]$ *includes the last element, i.e.,* $A[i-1]$*, or not.* This is because, if $A[i-1]$ is not in the solution of $M[i-1]$, then $A[i]$ cannot be included in the solution of $M[i]$ if elements of A with index less than $i-1$ are included; on the other hand, if $A[i-1]$ is included, then $A[i]$ can be included in the solution of $M[i]$. Thus, in general, the solution of a subproblem $M[i]$ depends on whether the last element, i.e., $A[i]$ is in the solution or not. Hence, to write a recursion for $M[i]$ we need to know whether $A[i]$ is in the solution or not. This motivates the following subproblems.[1]

We consider the following two subproblems based on the above fact.[2]

Let $M^{\text{in}}[i]$ be the solution to the max-sum problem considering only the *first i elements* of the array A, i.e., the maximum sum considering the array $A[1,\ldots,i]$, with the *condition that the solution includes the ith element*, i.e., the last element $A[i]$ is in the max-sum.

Similarly, let $M^{\text{out}}[i]$ be the solution to the max-sum problem considering the *first i elements* of the array A, i.e., the maximum sum considering the array $A[1,\ldots,i]$, with the condition that the solution *does not include the ith element*, i.e., the last element $A[i]$ is not in the max-sum. .

The solution to the max-sum problem in the array $A[1,\ldots,i]$ is

$$M[i] = \max\Big(M^{\text{in}}[i], M^{\text{out}}[i]\Big)$$

i.e., the maximum of the (only) two possibilities — $M^{\text{in}}[i]$ and $M^{\text{out}}[i]$ — where the first includes $A[i]$ and the second excludes it.

The important point about the subproblems is that it includes the original problem as well which is simply $M[n]$. This is typical of DP, the decomposed subproblems typically include the original problem as one of the subproblems. Although we are not interested in the solution of other subproblems, it turns out their solutions will help in building the solution to the original problem.

The total number of subproblems is $\mathcal{O}(n)$, or $2n$ to be precise — $M^{\text{in}}[i]$ and $M^{\text{out}}[i]$ for $1 \leqslant i \leqslant n$.

Step 4: Recursive formulation.

We now come to the key step of expressing the solution of the problem *recursively* in terms of the solutions to its subproblems. This is called *recursive formulation.* Once we have the correct recursive formulation, it turns out it is easy to write an algorithm that implements the formulation (see next step).

Recursive formulation is like writing a recurrence. There are one or more base cases which show the solution to base-case subproblems which are usually those that are of small

[1] Note that writing the recursive formulation is the next step in DP. However, it is quite obvious that these two steps go hand in hand; if we cannot easily write a recursive formula for our subproblems, then we try to modify it.

[2] A very naive way of formulating subproblems based on the fact that each element is in the solution or not is to look at all possible subsets of the array. There are 2^n such subsets. This is exponentially large and does not constitute an efficient decomposition since it ignores the key constraint that a *feasible* solution subset has to be *contiguous.* Even if we allow only for contiguous subsets to be contiguous, there are n^2 of them which is polynomial but still much bigger than $\mathcal{O}(n)$ that we are shooting for.

size. Then the recurrence shows how to compute the solution of larger-sized subproblems in terms of smaller-sized subproblems. The advantage of recursive formulation is that its correctness can be quickly established by (as usual) mathematical induction.

Let us illustrate recursive formulation by using the maxsum problem. As, mentioned above, $M[i] = \max\{M^{\text{in}}[i], M^{\text{out}}[i]\}$, since there are only two possibilities, either the last element ($A[i]$) is in the optimal solution or not. Note that $M[n]$ gives the maxsum value for the whole array (which is the answer we want to find).

The *recursive formulation* is writing a recurrence to compute $M^{\text{in}}[i]$ and $M^{\text{out}}[i]$ recursively (for $i > 1$):

$$\begin{aligned} M^{\text{in}}[i] &= \max\Big(M^{\text{in}}[i-1] + A[i], A[i]\Big) \\ M^{\text{out}}[i] &= M[i-1] \end{aligned}$$

The base cases are $M^{\text{in}}[1] = A[1]$ and $M^{\text{out}}[1] = -\infty$.[3]

Why is the above recursive formulation correct? Before we formally prove correctness by using mathematical induction, we give the intuition for why the above formulation is correct.

As mentioned earlier, the optimal solution to $M[i]$ will either include or exclude $A[i]$, so naturally there are two cases:

Case 1: *Exclude* $A[i]$. In this case, the optimal solution for the subarray $A[1 \ldots i]$, i.e., $M[i]$ will simply be the optimal solution to the subarray $A[1 \ldots i-1]$ which is $M[i-1]$.

Case 2: *Include* $A[i]$. In this case, there are only two possibilities, either the optimal solution is $A[i]$ alone or is $M^{\text{in}}[i-1]+A[i]$ (we take the max of the two). Note that the solution determining $M^{\text{in}}[i-1]$ *will include* element $A[i-1]$, and so we have contiguity of elements (this is the reason why we needed to decompose like this in the first place!).

Step 5: Correctness proof of the recursive formulation.

We can show formal correctness of the recursive formulation by using *mathematical induction*. As usual, the base cases, $M^{\text{in}}[1] = A[1]$ and $M^{\text{out}}[1] = -\infty$, are trivially true.

To show the induction step, we assume the induction hypothesis that the formulation is correct for $M^{\text{in}}[j]$ and $M^{\text{out}}[j]$, for all $1 \leqslant j < i$. We show that it is true for $j = i$ as well, under the above assumption. From the recurrence, we have $M^{\text{out}}[i] = M[i-1]$. This is true, because since we exclude $A[i]$, and $M[i-1]$ is the *optimal* solution that corresponds to the subarray $A[1, \ldots, i-1]$. Note that the optimality of $M[i-1]$ is guaranteed by the induction hypothesis. Similarly, from the recurrence, we have $M^{in}[i] = \max\{M^{\text{in}}[i-1] + A[i], A[i]\}$. This is also true, since the optimal value of the subarray $A[1, \ldots, i]$ that includes $A[i]$ can either be $A[i]$ itself or can be a *continuation* of the *optimal* solution of subarray $A[1, \ldots, i-1]$, which, of course should include $A[i-1]$ — such a solution is precisely $M^{\text{in}}[i-1]$. This completes the proof.

The proof shows an important property of problems that have DP solutions sometimes referred to as the *optimal substructure property* which says that the optimal solution of problems contains within itself optimal solution of its subproblems. This is really applying the proof “in reverse.” Suppose $M^{\text{out}}[i]$ is the optimal solution to the subproblem $A[1, \ldots, i]$, then this implies, by the correctness of the recurrence, that $M[i-1]$ should

[3] We set $M^{\text{out}}[1] = -\infty$, since we assume that at least one element of A should be in the optimal solution. If the empty array is a valid solution, then we could have set $M^{\text{out}}[1] = 0$.

be the optimal value of the subproblem $A[1, \ldots, i-1]$. Suppose not. Then by replacing $M[i-1]$ with the optimal value, we get a better optimal value for $M^{\text{out}}[i]$ which contradicts the optimality of $M^{\text{out}}[i]$. Similar argument can be made with the recurrence for $M^{\text{in}}[i]$.

Step 6: Computing run time of DP problems.

We outline the key points to upper bound the run time of a DP problem.

- Bound the total number of subproblems generated — let it be K.
- Bound the (worst-case) time take to solve each subproblem — let it be T.
- Then the total time of DP is upper bounded by $\mathcal{O}(KT)$ because, there are at most K subproblems and each takes at most T time. Sometimes a better bound can be obtained by looking at the time taken by individual subproblems. Let the time taken to solve subproblem S_i be T_i, $1 \leqslant i \leqslant K$. Then the overall time is $\sum_{i=1}^{K} T_i$.

Let us calculate the running time of the DP algorithm for the maxsum problem. The number of subproblems is $2n$ — $M^{\text{in}}[i]$ and $M^{\text{out}}[i]$ for each i. The time to compute each is just constant, which follows from the recurrence. Hence, the total run time is $\mathcal{O}(n)$.

Algorithm 22 MaxsumDP – Iterative Implementation of Maxsum
Input: An array A of length n
Output: Maximum subarray sum

```
func MaxsumDP($A$):
    $M[1] = A[1]$
    $M^{\text{in}}[1] = A[1]$
    $M^{\text{out}}[1] = -\infty$
    for $i = 2$ to $n$:
        $M^{\text{out}}[i] = M[i-1]$
        $M^{\text{in}}[i] = \max(M^{\text{in}}[i-1] + A[i], A[i])$
        $M[i] = \max(M^{\text{in}}[i], M^{\text{out}}[i])$
    return $M[n]$
```

Algorithm 23 MaxsumRec – Naive Recursive Implementation of Maxsum
Input: An array A of length n
Output: Maximum subarray sum

```
func MaxsumRec($A$):
    return max(M^IN Rec($A$), M^OUT Rec($A$))

func M^IN Rec($A$):
▷ Maxsum including the ith element
    if $i == 1$:
        return $A[1]$
    else:
        return max(M^IN Rec($A[1, \ldots, i-1]$) $+ A[i]$), $A[i]$)

func M^OUT Rec($A$):
▷ Maxsum excluding the ith element
    if $i == 1$:
        return $-\infty$
    else:
        return MaxsumRec($A[1, \ldots, i-1]$)
```

Algorithm 24 MaxsumRecLookup – Efficient implementation of the recursive formulation of the maxsum problem that uses the table lookup strategy.
Input: An array A of length n
Output: Maxsum value

```
func MaxsumRecLookup($A$):
    for $i = 1$ to $n$:
        $M[i] = M^{in}[i] = M^{out}[i] = -\infty$    ▷ initialization of arrays
    $M^{in}[1] = A[1]$
    return max(M^IN RecLookup($n$), M^OUT RecLookup($n$))

func M^IN RecLookup($i$):
    if $M^{in}[i] == -\infty$:    ▷ Value has not yet been computed
        include = M^IN RecLookup($A[1, \ldots, i-1]$) $+ A[i]$)
        $M^{in}[i]$ = max(include, $A[i]$)
    return $M^{in}[i]$

func M^OUT RecLookup($i$):
    if $M^{out}[i] == -\infty$:    ▷ Value has not yet been computed
        $M^{out}[i]$ = MaxsumRecLookup($A[1, \ldots, i-1]$)
    return $M^{out}[i]$
```

Step 7: Converting the recursive formula into an algorithm.

It is relatively straightforward to turn the above recursive formula into a pseudocode — either iterative or recursive. However, care has to be taken, especially in a recursive implementation so that each subproblem is executed exactly once. Otherwise, the running time of the implementation can be large (even exponential in the size of the problem). We will next show two different approaches for implementing DP algorithms:

1. *Iterative algorithm*: Implementing the recursive formulation in an interative manner is what is known as dynamic programming. This is a bottom-up strategy, where we start from the solution of the base cases and then compute the solutions to larger-sized problems using smaller-sized problems. Let us illustrate this in the case of the maxsum problem. We first compute $M^{\text{in}}[1]$ and $M[1]$ which are both $A[1]$. Note that $M^{\text{in}}[2]$ and $M^{\text{out}}[2]$ depend only on $M^{\text{in}}[1]$ and $M[1]$ and hence can be computed. Similarly $M^{\text{in}}[3]$ and $M^{\text{out}}[3]$ can be computed and so on. Finally, we compute $M^{\text{in}}[n]$ and $M^{\text{out}}[n]$. The final answer is $M[n]$ which is equal to the maximum of $M^{\text{in}}[n]$ and $M^{\text{out}}[n]$. This can be implemented by a **for** loop. The iterative DP algorithm is given in Algorithm 22. The running time of the iterative algorithm is clearly $\mathcal{O}(n)$ since there are are $n-1$ iterations of the **for** loop and each iteration takes constant time. This is clearly an efficient implementation of the DP formulation as each subproblem is computed only once leading to an $\mathcal{O}(n)$ run time as postulated in Step 6.

2. *Recursive algorithm*: A more straightforward way is to implement the recursive formula in a recursive way. There is an almost a direct translation of the recursive formula into a recursive algorithm. See algorithm in Algorithm 23. However, this translation is bad because it does not take into account *overlapping subproblems*, i.e., the same subproblems occur in different recursive calls and will be calculated over and over again leading to a very inefficient algorithm.

   In the maxsum problem, the subproblem $M^{\text{in}}[i]$ occurs in the solution of two different subproblems $M^{\text{in}}[i+1]$ and in $M^{\text{out}}[i+1] = M[i] = \max\{M^{\text{in}}[i], M^{\text{out}}[i]\}$.

   Since Algorithm 23 solves the same subproblem more than once, its running time turns out to be significantly more than $\mathcal{O}(n)$, in fact it will be $\Theta(n^2)$. Exercise 6.1 asks you to show that the running time of algorithm in Algorithm 23 is $\Theta(n^2)$.

   To get an implementation of the recursive algorithm that runs in $\mathcal{O}(n)$ time, we make sure that we compute each subproblem only once. To ensure this, as soon as we compute the solution to a subproblem for the first time, we store its solution (in a suitable data structure like an array); subsequently, if the same subproblem is encountered we simply "look up" this stored solution without the need to recompute the solution from scratch. This is called as the *table lookup strategy* or *memoized strategy*. This only requires a simple modification to the inefficient Algorithm 23. The modified algorithm is given in Algorithm 24. In the modified algorithm we store the values of the computed subproblems — $M^{\text{in}}[i]$ and $M^{\text{out}}[i]$ — in an array and then use the stored values to avoid recomputing the solution to a subproblem that was already computed.

Step 8: Computing the solution that gives the optimum value.

Typically it is easy to modify the algorithm that computes the optimum value to also output the solution that gives the optimum value. This usually involves doing some *additional bookkeeping*. Let us illustrate this with the maxsum problem. How do you output the indices that give the maximum sum? We do this by keeping an additional array called $S[i]$ which is either "in" or "out." If it is "in" then the $A[i]$ is included in the optimal solution of the subproblem with value $M[i]$. $S[i]$ array values are computed as we compute the optimal $M[i]$ values. This can be done by simply checking how $M[i] = \max(M^{\text{in}}[i], M^{\text{out}}[i])$ is computed. If $M[i] = M^{\text{in}}[i]$, then it means that $A[i]$ is part of the optimal solution and hence we set $S[i] =$ "in"; on the other hand if $M[i] = M^{\text{out}}[i]$, then it means that $A[i]$ is not part of the optimal solution and hence we set $S[i] =$ "out." Similarly, we also keep an additional array $S^{\text{in}}[i]$ which records which of the two values $M^{\text{in}}[i]$ takes, i.e., whether it is $M^{\text{in}}[i-1] + A[i]$ or just $A[i]$. In the case of the former, we set $S^{\text{in}}[i] =$ `True`, otherwise we set $S^{\text{in}}[i] =$ `False`. Algorithm 25 shows how to modify the DP algorithm of Algorithm 22 to compute the additional array $A[i]$.

To output the optimal solution we usually *trace back* the steps that we took to compute the optimum value, which in the case of the maxsum problem is $M[n]$. Note that $M[n]$ is the maximum of $M^{\text{in}}[n]$ and $M^{\text{out}}[n]$. If the former is the maximum of the two, then $A[n]$ is in the solution, otherwise, it is not. This information is stored in $S[n]$. If $A[n]$ is in the solution, we look at whether $A[n-1]$ is in the solution — this can be inferred by looking at the $S^{\text{in}}[n]$. We can then continue whether $A[n-2]$ is in the solution or not and so on. This is easier to do recursively, since we do this in a top-down fashion starting from the optimal value $M[n]$. The algorithm is given in Algorithm 26.

Algorithm 25 MaxsumDPMod – This is the modified DP (iterative) implementation of the recursive formulation of the maxsum problem that also computes the S and S^{in} arrays which are used in outputting the optimal solution.

Input: An array A

Output: Maxsum and arrays S and S^{in}

```
func MaxsumDPMod($A$):
    $M[1] = A[1]$
    $M^{\text{in}}[1] = A[1]$
    $M^{\text{out}}[i] = -\infty$
    $S[1] =$ in
    for $i = 2$ to $n$:
        $M^{\text{out}}[i] = M[i-1]$
        $M^{\text{in}}[i] = \max\{M^{\text{in}}[i-1] + A[i], A[i]\}$
        $M[i] = \max(M^{\text{in}}[i], M^{\text{out}}[i])$
        if $M[i] == M^{\text{in}}[i]$:
            $S[i] =$ in
        else:
            $S[i] =$ out
        if $M^{\text{in}}[i] == M^{\text{in}}[i-1] + A[i]$:
            $S^{\text{in}}[i] =$ True
        else:
            $S^{\text{in}}[i] =$ False
    return $M[n]$
```

Algorithm 26 MaxsumSol
Input: Array A, S, and S^{in} as in MaxsumDPMod
Output: None. The optimal solution is printed backwards.

```
func MaxsumSol($A$, $S$, $S^{in}$):
    if $n \geqslant 1$:
        if $S[n]$ == in:
            print($A[n]$)    ▷ $A[n]$ is included in the optimal solution
            if $S^{in}[n]$:    ▷ $A[n-1]$ is also included in the optimal solution
                MaxsumSol($A[1,\ldots,n-1], S[1,\ldots,n-1], S^{in}[1,\ldots,n-1]$)
        else:
            ▷ $S[n]$ = out
            MaxsumSol($A[1,\ldots,n-1], S[1,\ldots,n-1], S^{in}[1,\ldots,n-1]$)
```

We next further illustrate these ideas by looking at various problems.

6.4 Problem: Matrix Chain Multiplication

We are given a chain of n matrices: $< A_1, A_2, \ldots, A_n >$ where matrix A_i has dimension $p_{i-1} \times p_i$, for $i = 1, 2, \ldots, n$. We are interested in finding the "best" way to compute the product of the n matrices:

Problem 6.2 ▶ Matrix Chain Multiplication

Find a way to *parenthesize* the product $A_1 A_2 \ldots A_n$ which *minimizes* the number of multiplications.

To understand the problem, we recall some basic facts about matrix multiplication:

- The number of columns of A_i is equal to the number of rows of A_{i+1} (for $1 \leqslant i \leqslant n-1$). This property is needed for a valid multiplication.
- Given 2 matrices A, B, with dimension $m \times n$ and $n \times p$ respectively, the number of element-wise multiplications needed to multiply the 2 matrices is mnp. Note that here we assume that multiplications are done in the standard way, i.e., the i, jth entry of the product matrix $C = AB$ is computed as: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ which takes n multiplications. Since the product matrix C has mp elements, the total number of multiplications is mnp.
- Matrix multiplication is *associative*, i.e., $(AB)C = A(BC)$. Thus the order of multiplying the matrices does not matter for the final answer. However, order might matter for minimizing the total number of multiplications (cost).

Example

Suppose we are given 3 matrices A_1, A_2 and A_3 with dimensions 10×1000, 1000×100 and 100×200 respectively. If we multiply according to parenthesization $(A_1 A_2) A_3$ then the cost is: $10 \times 1000 \times 100 + 10 \times 100 \times 200 = 1200000$. If we multiply according to parenthesization $A_1(A_2 A_3)$ then the cost is: $1000 \times 100 \times 200 + 10 \times 1000 \times 200 = 22000000$.

Number of ways of parenthesizing n multiplications

Given a chain of n matrices, the number of different ways to parenthesize is *exponential* in n. Each parenthesization gives a different way to multiply the chain with different costs. Let $P(n)$ be the number of different ways of parenthesizing a product of n matrices. Then we can obtain the following recurrence:

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k), \quad n \geqslant 2$$

$$P(1) = 1$$

We can show that $P(n)$ is *exponential* in n, in fact, $P(n) = \Omega(2^n)$. (See Exercise 6.2.) Hence a naive strategy of checking all possible parenthesizations is *very inefficient*.

6.4.1 DP solution

We first write a *recursive formulation* for computing the minimum cost parenthesization. For $1 \leqslant i \leqslant j \leqslant n$, let $m[i,j]$ be the *minimum cost* of paranthesizing the product $A_iA_{i+1}\ldots A_j$. Then we can write a recursive formula to compute $m[i,j]$.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{otherwise} \end{cases}$$

Writing the above *recursive formula* is the *key step* in solving a dynamic programming problem. Using the recursive formula, we can develop an algorithm. As we mentioned earlier there are 2 possible algorithmic strategies: (1) top-down / recursive/ table look up /memoization or (2) bottom-up / iterative / DP.

Key step in the recursive formulation

In many DP problems, the key step in writing a correct recursive formulation is identifying the *right subproblem(s)*. The subproblems may not be obvious. A useful idea is to introduce auxiliary parameters in the problem. For example, though we are interested in finding out the best way to paranthesize the entire matrix chain, by introducing the two auxiliary variables i and j, we identify the subproblem(s) of parenthesizing the subchain $A_i \ldots A_j$ which proves useful in solving the desired problem. Note that the original problem is one of the subproblems, i.e., computing $m[1,n]$.

Correctness of the recursive formulation

To show the correctness of the recursive formulation, as usual we we turn to mathematical induction.

- Base case $(i = j)$: Trivially true.
- Recursive step $(i < j)$: The induction hypothesis is that the recursive formulation is correct for all ℓ such $j - i < \ell$. That is, the induction is on the *length* of the chain. Assuming the induction hypothesis, we prove that the formulation is correct for $j - i = \ell$, i.e., $m[i,j]$ is the optimal cost of multiplying $A_i \ldots A_j$.

Take the minimum of the $j - i = \ell$ possible parenthesizations between A_i and A_j depending on where you split the product, i.e.,

$$(A_i \dots A_j) = (A_i \dots A_k)(A_{k+1} \dots A_j)$$

We note that $m[i, k]$ is the minimum cost of multiplying $(A_i \dots A_k)$ (by induction hypothesis), $m[k + 1, j]$ is the minimum cost of multiplying $A(A_{k+1} \dots A_j)$ (again by induction hypothesis), and $p_{i-1}p_kp_j$ is the cost of combining the two partial products. Hence, since the split is minimum, the overall cost $m[i, j]$ is also optimal.

Note that the above argument shows the *optimal substructure property* which is nothing but saying that the optimal solution to a problem consists of optimal solution to its subproblems. Here computing $m[i, j]$ is the problem and that depends on $m[i, k]$ and $m[k + 1, j]$ (for all $i \leqslant k < j$), which are optimal solutions to subproblems. As we saw above, the proof is straightforward by using induction. The optimal substructure property tells us that there is *no need* to consider *suboptimal* solutions to subproblems when solving a problem; only optimal solutions to subproblems are relevant. This drastically reduces the number of subproblem solutions to consider — leading to efficient algorithms.

A Naive Algorithm

The following is a naive algorithm that directly converts the recursive formula into a recursive algorithm. Exercise 6.3 asks you to show that the above algorithm takes exponential time.

Algorithm 27 Cost

Input: A sequence of matrix dimensions: $p = p_0, p_1, \dots, p_n$ and indices i and j

Output: The optimal cost to multiply $A_i \dots A_j$

```
func Cost($p$, $i$, $j$):
    if $i == j$:
        return 0
    $m[i, j] = \infty$
    for $k = i$ to $j - 1$:
        $q = \text{Cost}(p, i, k) + \text{Cost}(p, k + 1, j) + p_{i-1}p_kp_j$
        if $q < m[i, j]$:
            $m[i, j] = q$
    return $m[i, j]$
```

Bottom-up Algorithm (DP/iterative algorithm)

Here is the DP algorithm which is an iterative algorithm. Note that the DP algorithm builds the solution bottom up: it first computes 1 length chains (which is 0) and then successively computes chains of increasing length, i.e., lengths $2, 3, \dots,$ up to n. Each chain of length l depends on chains of length $l - 1$ or smaller. Hence computing bottom up works.

Algorithm 28 MatrixCostDP – DP Matrix Cost
Input: A sequence of matrix dimensions p
Output: The optimal cost to multiply $A_1 \dots A_n$

```
func MatrixCostDP($p$):
    for $i = 1$ to $n$:
        $m[i,i] = 0$
    for $\ell = 2$ to $n$:
        for $i = 1$ to $n - \ell + 1$:
            $j = i + \ell - 1$
            $m[i,j] = \infty$
            for $k = i$ to $j - 1$:
                $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
                if $q < m[i,j]$:
                    $m[i,j] = q$
    return $m[1,n]$
```

Memoization: Top-down Algorithm (recursive/table look up)

While the naive recursive algorithm is inefficient, the following recursive algorithm that uses table lookup to *avoid computation of the same subproblem more than once* takes polynomial time.

Algorithm 29 MatrixCostRec – Recursive Matrix Cost
Input: A sequence of matrix dimensions p
Output: The optimal cost to multiply $A_1 \dots A_n$

```
func MatrixCostRec($p$):
    return MatrixCostHelper($p, 1, n$)

func MatrixCostHelper($p$, $i$, $j$):
    if $m[i,j] == \infty$:
        if $i == j$:
            $m[i,j] = 0$
        else:
            for $k = i$ to $j - 1$:
                left = MatrixCostHelper($p, i, k$)
                right = MatrixCostHelper($p, k+1, j$)
                total = left + right + $p_{i-1}p_kp_j$
                if total $< m[i,j]$:
                    $m[i,j]$ = total
    return $m[i,j]$
```

Run time of Matrix Multiplication

The number of subproblems is $\Theta(n^2)$ since there is a distinct subproblem for every pair of indices i and j, $1 \leqslant i \leqslant j \leqslant n$. Each subproblem can be solved in $\mathcal{O}(n)$ time. Hence total time is $\mathcal{O}(n^3)$.

Getting the actual parenthesization

We only focused on getting the *minimum cost.* It is easy to modify the code to obtain the actual paranthesization that achieves the minimum cost as well, in the same time bounds (see Exercise 6.6).

6.5 Problem: Aligning two Sequences

We consider a problem that is fundamental to computational molecular biology: Given two (e.g., DNA or protein) sequences (strings) s of length m and t of length n we want the "best" *alignment* between the two sequences.

Suppose we are given two DNA strings s and t as follows.

$$s = \texttt{GACGGATTAG}$$
$$t = \texttt{GATCGGAATAG}$$

The goal is to find the "best" way to "align" the two strings. We define what is meant by alignment next.

Alignment

An alignment is an insertion of spaces in arbitrary locations along the sequences so that they end up with the same size. A space in one sequence should not align with a space in the other. But spaces can be inserted at the beginning or at the end. An alignment is **scored** by a scoring scheme. We assume a given scoring matrix score where the entry score$[x, y]$ gives the alignment score for characters x and y. For simplicity, we will assume that the score of any letter with space is -1. The *score* for an alignment is the *sum of the scores* of its aligned characters. A best alignment is one which receives the *maximum score* called the *similarity* — sim(s, t).

Example

We want to find the best way to align $s = \texttt{GACGGATTAG}$ and $t = \texttt{GATCGGAATAG}$. Assume that the score for aligning two characters which are the same is 1, but aligning two different characters has score 0, while aligning a letter with a space has score -1. Then the best alignment of s and t is:

$$s = \texttt{GA CGGATTAG}$$
$$t = \texttt{GATCGGAATAG}$$

Note that a space was introduced in s after the first two characters. The score for this alignment is $1 + 1 - 1 + 1 + 1 + 1 + 1 + 0 + 1 + 1 + 1 = 8$. This is the maximum possible alignment score and hence this is a best alignment (there can be more than one best alignment).

A Naive solution

A naive approach is to look at all possible alignments of the two input strings, compute the score for each alignment and then take the maximum. This is very inefficient as the number of possible alignments is at least exponential in the size of the strings. For example, if one string has $2n$ characters and the other string has n characters, one has to insert at least n gaps to align the two strings; the number of ways that one can insert n gaps among the n characters is at least $\binom{2n}{n} \geqslant 2^n$.[4] DP leads to a significantly more efficient solution as described below.

Recursive formulation

As in the matrix multiplication problem, to compute the optimal alignment score we first write the recursive formulation. Again the key is to identify the right subproblems.

Let string s be represented as an array (list) of characters $s[1 \dots m]$ and t as $t[1 \dots n]$. We will use the subproblem(s) of finding the best alignment of the *substrings* $s[1 \dots i]$ and $t[1 \dots j]$, where $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$. Note that we only allow the right end to vary. One could have allowed "both" ends to vary, i.e., consider subproblems of aligning strings $s[i \dots j]$ and $t[k \dots \ell]$. However this is not necessary and leads to an increased run time (though still polynomial) since the number of subproblems is more. We note that $s[]$ and $t[]$ define the empty string.

The key idea behind the recursive formulation is that to compute $\text{sim}(s[1 \dots i], t[1 \dots j])$, we need to only look at aligning the last character of each of the two strings. There are only three possibilities: (1) aligning $s[i]$ with $t[j]$ (2) aligning gap with $t[j]$ (3) aligning $s[i]$ with gap. Thus we have the following recursive formula:

$$\text{sim}(s[1 \dots i], t[1 \dots j]) = \max \begin{cases} \text{sim}(s[1 \dots i-1], t[1 \dots j-1]) + \text{score}(s[i], t[j]) \\ \text{sim}(s[1 \dots i], t[1 \dots j-1]) + \text{score}(-, t[j]) \\ \text{sim}(s[1 \dots i-1], t[1 \dots j]) + \text{score}(s[i], -) \end{cases}$$

The base cases are:

$$\text{sim}(s[1 \dots i], t[]) = -i$$

$$\text{sim}(s[], t[1 \dots j]) = -j$$

$$\text{sim}(s[], t[]) = 0$$

The above base cases capture the three base case situations that arise from the three possibilities: (1) when t becomes empty, and s is not empty, in which case we have to align all the characters left in s with gaps (2) when s becomes empty, and t is not empty, in which case we have to align all the characters left in t with gaps (3) both become empty. Note that in the above base cases, we have used that aligning a gap with a character has score -1.

The correctness of the recursive formula follows from mathematical induction which is left as an exercise.

Dynamic Programming (Bottom-up) Algorithm

We can convert the recursive formulation into a DP algorithm as follows in a fairly straightforward fashion.

[4] See Appendix G for the inequality used to infer this lower bound.

Algorithm 30 SimScoreDP
Input: sequences s and t of lengths m and n
Output: $\mathrm{sim}(s,t)$

```
func SimScoreDP($s$, $t$):
    for $i = 0$ to $m$:
        $a[i,0] = -i$    ▷ $a[i,j]$ holds $\mathrm{sim}(s[1 \dots i], t[1 \dots j])$
    for $j = 0$ to $n$:
        $a[0,j] = -j$
    for $i = 1$ to $m$:
        for $j = 1$ to $n$:
            align_t $= a[i,j-1] +$ score$(-, t[j])$
            align_s $= a[i-1,j] +$ score$(s[i], -)$
            align_both $= a[i-1,j-1] +$ score$(s[i], t[j])$
            $a[i,j] = \max($align_t, align_s, align_both$)$
    return $a[m,n]$
```

It is easy to see that the run time of the above algorithm is $\mathcal{O}(mn)$.

6.6 Problem: 0/1 Knapsack

The knapsack is a classic optimization problem which has been studied extensively for more than a century and arises in many real-world applications. Here we consider the integral (or 0/1) knapsack problem which is a modification of the fractional knapsack problem considered in Chapter 7. The problem is defined as follows.

Problem 6.3 ▶ 0/1 Knapsack

We are given a set $A = \{a_1, a_2, \dots, a_n\}$ of n objects and a knapsack. Object a_i has an integer weight w_i and the knapsack has integer capacity m, i.e., the maximum weight of all items placed in the knapsack cannot exceed m. An object a_i is either placed into the knapsack or entirely omitted (no fractional part allowed), i.e., $x_i = 1$ if object is placed in the knapsack, and 0 otherwise. If the object a_i is placed, then a profit of p_i is earned. The objective is to *maximize* the total profit earned subject to the knapsack constraint.

We want to find the optimal solution using Dynamic Programming. That is we want to find the solution vector $x_1, x_2, \dots x_n$ which is 0-1 vector that maximizes the total profit earned subject to the knapsack constraint.

A Naive Solution

As usual, a simple, but naive solution is to consider all possible subsets of objects (since each object can be either in the optimal solution or not) and take the subset that satisfies the knapsack capacity and maximizes the total profit. Since the number of subsets is 2^n, this leads to exponential running time. DP, again, leads to a significantly improved running time (that depends linearly on m) as explained below.

Subproblems

The key, again, for a correct recursive formulation is identifying the right subproblems. For this, as mentioned earlier, introducing auxiliary variables help. Let us introduce two new variables j, y as follows. Let $KNAP(j, y)$ represent the problem: maximize $\sum_{1 \leqslant i \leqslant j} p_i x_i$; subject to the constraint $\sum_{1 \leqslant i \leqslant j} w_i x_i \leqslant y$ with the (integral) requirement that $x_i \in \{0, 1\}$, $1 \leqslant i \leqslant j$. The above is simply a restatement of the knapsack problem as an explicit *maximization* problem. In words, it is the subproblem that considers only the first j items with a knapsack capacity y. The Knapsack problem is simply $KNAP(n, m)$. Note that we assume all weights and m are *integers*.

Recursive formulation

Let $P_j(y)$ be the optimal solution to $KNAP(j, y)$. Then, we can write:

$$P_j(y) = \max\{P_{j-1}(y), P_{j-1}(y - w_j) + p_j\}$$

The base cases are: $P_0(y) = 0$ for all $y \geqslant 0$ and $P_i(y) = -\infty$ if $y < 0$.

Correctness of the formulation

The above formulation is correct because to compute $P_j(y)$ we need to look at only *two* choices: either the a_jth object is in the knapsack or not. If it is not included, then $P_j(y) = P_{j-1}(y)$, since no weight has been added. If it is included, then $P_j(y) = P_{j-1}(y - w_j) + p_j$, since we have to subtract the weight w_j from the knapsack capacity and include its profit. The correctness of the formulation, as usual, can be established by induction.

Again induction implies that the optimal substructure property holds. $P_j(y)$ should contain within itself optimal solution to its subproblem(s) — which are $P_{j-1}(y)$ (in the first case) and $P_{j-1}(y - w_j)$ (in the second case). Of course, we take the maximum value of the two cases.

DP algorithm

A DP algorithm is straightforward from the above formulation. The desired optimal value is $P_n(m)$ which we compute in a bottom up manner. That is, we first start from the base cases and compute $P_1(.), P_2(.), \ldots$ in this order.

The time complexity is $\mathcal{O}(nm)$. Note that the running time depends on the number of objects as well as the (magnitude) of the (integer) capacity. Hence this is *not* a polynomial time algorithm, since the running time is not a polynomial function of the input size which is $\log m$ (since m can be represented using $\log m$ bits), but rather a polynomial function of m. Such a running time is referred to as *pseudo-polynomial* and the algorithm is called a *pseudo-polynomial* algorithm. Such algorithms are fast when m is small; more importantly they can be converted to yield efficient polynomial time algorithms that give an *approximately* optimal solution. This is discussed in the next section.

6.7 A Fully Polynomial-Time Approximation Scheme for Knapsack*

We note that Knapsack is an *NP-complete* problem, i.e., there is unlikely to be a polynomial time algorithm for Knapsack. It is important to note that by "polynomial" we mean polynomial in the *input size*, i.e., the number of items n and the sizes of the integers used in the input (represented in binary). In other words, the runtime should be a polynomial function of n and $\log(m)$ (the capacity), and $\log(w_i)$ (for all weights w_i of all items) and $\log(p_i)$ (for all profits). The DP-based pseudo-polynomial algorithm presented in the previous section is a linear function of m and is thus not a truly polynomial time algorithm, it is a *pseudo-polynomial* one. As mentioned there, it is fast (i.e., polynomial) when the capacity, weights and profits are small, i.e., polynomial function of n. However, one can obtain a truly polynomial *approximation* algorithm for Knapsack that can achieve as close to the optimal as possible (however, the run time will increase, the closer we get to the optimum). In other words, we can achieve what is called as a **fully polynomial-time approximation (FPTAS) scheme**, which is essentially the best possible that one can hope for an NP-complete problem.

Before we proceed, we will define formally an approximation algorithm and an FPTAS, which is a special type of approximation algorithm.

6.7.1 Approximation Algorithm and FPTAS

An approximation algorithm for an optimization problem is (typically) a polynomial time algorithm that computes a solution whose value is "close" to the optimal value. The closeness is defined as a ratio between optimal and approximate solution values.

Definition 6.1

An approximation algorithm has a **ratio bound** $\rho(n)$, if for any input of size n, the optimal solution $C^*(n)$ and the algorithm solution $C(n)$ satisfy the relation:

$$\frac{C(n)}{C^*(n)} \leqslant \rho(n) \text{ for minimization problems}$$
$$\frac{C(n)}{C^*(n)} \geqslant \rho(n) \text{ for maximization problems}$$

To establish an approximation ratio, we need to compare the cost of the solution of the algorithm (ALG) to the cost of an optimal solution (OPT). Typically, this means that we have to find "good" polynomial-time computable bounds for OPT.

Definition 6.2 ▶ FPTAS

A minimization (resp. maximization) problem has a **fully polynomial-time approximation (FPTAS)** scheme if for any $\epsilon > 0$ there is a $1 + \epsilon$ (resp. $1 - \epsilon$)-approximation algorithm for the problem that runs in time which is polynomial in both the problem input size and $1/\epsilon$.

Note that an FPTAS is actually a family of algorithms, parameterized by ϵ. Given an ϵ, the algorithm guarantees that the solution is within a factor of $1 - \epsilon$ of the optimal solution; the algorithm runs in time polynomial in the input size and $1/\epsilon$. Thus it gives a *tradeoff* between accuracy of the solution and the running time; smaller the ϵ, closer is the solution to the optimal, but larger is the run time.

A problem which has a pseudo-polynomial time algorithm can be typically converted to yield an FPTAS, as we will illustrate next.

6.7.2 Alternate DP algorithm for Knapsack

We will use DP to design a pseudo-polynomial time algorithm for knapsack; this DP algorithm, called KNAPSACK, is a bit different from one proposed in Section 6.6. In the earlier formulation, we fixed a bound on the capacity and sought to maximize the profit subject to the capacity bound. In the alternate formulation, we fix a particular profit and seek to minimize the total size that achieves (if possible) the particular profit. In this formulation, the algorithm's run time is a function of the profit values (unlike the earlier formulation which was a function of the maximum capacity). This is readily suitable for designing a FPTAS.

Again, we are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n objects, with specified weights and profits (all values are positive integers), and a knapsack capacity m, a positive integer. The problem is to find a subset of objects whose total weight is bounded by m and the total profit is *maximized*. (Without loss of generality, we will assume that every object has weight at most m.)

Let P be the profit of the most profitable object, i.e., $P = \max_{a \in S} \text{profit}(a)$. Note that nP is a trivial upper bound for any solution.

For each $i \in \{1, \ldots, n\}$ and $p \in \{1, \ldots, nP\}$, let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i\}$ whose *total profit is exactly* p and whose *total weight is minimized*. Let $W(i,p)$ denote the sum of the weights of the objects in the set $S_{i,p}$. Note that $W(i,p) = \infty$ if no such set exists.

The recursive formulation is as follows. There are two cases:

1. If $\text{profit}(a_{i+1}) \leqslant p$, then we have two choices: either include object a_{i+1} or not. Thus we have:

$$W(i+1,p) = \min\{W(i,p), \text{size}(a_{i+1}) + W(i, p - \text{profit}(a_{i+1}))\}$$

   To explain the above formula, we note that $W(i+1,p)$ is the same as $W(i,p)$ if we don't include a_{i+1}; otherwise, if we include a_{i+1} we have to include its weight plus $W(i, p - \text{profit}(a_{i+1}))$. Note that in the latter case, we get the subproblem of computing the minimum weight from a subset of the first i items that gives a profit of $p - \text{profit}(a_{i+1})$ (since when we add the profit of a_{i+1}, we get a total profit of p).

2. If $\text{profit}(a_{i+1}) > p$, then

$$W(i+1,p) = W(i,p)$$

The base case $W(1,p)$ can be trivially computed for every $p \in \{1, \ldots nP\}$. Hence all values $W(i,p)$ can be computed in $\mathcal{O}(n^2P)$ time (see Exercise 6.12).

The maximum profit achievable by objects of total weight bounded by m is $\max\{p \mid W(n,p) \leqslant m\}$.

6.7.3 FPTAS for Knapsack

The idea used to design an $(1-\epsilon)$-approximation algorithm is simple. If P is "small" (say polynomial in n), then the DP algorithm gives a polynomial time algorithm. However, P can be large (e.g., it can be exponential in n). So the high-level idea behind the approximation algorithm is to scale down the value of P. If we are interested in only approximating the optimal, we can round the numerical values by ignoring a certain number of least significant bits of the profit values. The number of bits we will ignore will depend on the error parameter ϵ. Intuitively, the optimal profit will also be rounded down; however, there will be some loss of accuracy because of rounding.

We will simply run the DP algorithm on the rounded profit values which returns the optimal (exact) solution for the rounded values. Note that the run time of the DP solution is $\mathcal{O}(n^2P)$ where P is the value of the most profitable object. The idea of rounding is simple: we round down all profits by a factor of $K = \frac{\epsilon P}{n}$, i.e., the rounded profit $p_i' = \lfloor p_i/K \rfloor$. Because of the rounding down, the largest profit value is also rounded down to n/ϵ and the running time becomes $\mathcal{O}(n^3/\epsilon)$ which is a polynomial in n and ϵ. But what about the optimal value? We show next that the optimal value can be obtained within an factor of $(1-\epsilon)$; in other words, the effect of rounding can be bounded.

The algorithm.

The FPTAS algorithm is given in Algorithm 31. As mentioned earlier, it is simple: we simple round down the profit of each item by a factor of $K = \epsilon P/n$ and run the DP algorithm on the rounded profits and output the solution. We will show that the output solution is a $(1-\epsilon)$-approximation to the given problem.

Algorithm 31 FPTASKnapsack – FPTAS Algorithm for Knapsack
Input: An array X of objects with weights and profits and an $\epsilon > 0$
Output: An approximate solution to Knapsack

```
func FPTASKnapsack($X$, $m$, $\epsilon$):
    $P = \max\{\text{profit}(x) \mid x \in X\}$
    $K = \epsilon P/n$
    $X' = \emptyset$   ▷ Initialize an empty array $X'$
    for $w, p \in X$:   ▷ Iterate through weights and profits of objects in $X$
        $p' = \lfloor p/K \rfloor$
        append $(w, p')$ to $X'$
    return Knapsack$(X', m)$   ▷ Run the above DP algorithm on the rounded profits
and return the optimal solution
```

Analysis

We now formalize the intuition discussed earlier.

Lemma 6.1

Let ALG denote the set output by algorithm FPTASKnapsack. Then

$$p(ALG) \geqslant (1-\epsilon)p(OPT)$$

where OPT and ALG are the set of objects output by the optimal algorithm (with optimal profit value $p(OPT)$) and the algorithm FPTASKnapsack (with profit $p(ALG)$) respectively.

Proof. To bound the approximation ratio, we compare the optimal solution with the solution output by the algorithm. Since we cannot exactly compute the optimal solution, we obtain a good, *polynomially computable upper bound* on the optimal solution and compare it with the solution obtained by DP.

Let O denote the optimal set of objects. For an object a, let p_a and $p'_a = \lfloor \frac{p_a}{K} \rfloor$ denote the original profit and the rounded down profit respectively. For any set A of objects associated with the original profits, let $p(A)$ denote the sum of the profits of the objects in A. Analogously, let $p'(A)$ denote the sum of the rounded version of the profits of the objects in A.

Note that we don't know how to compute OPT (or $p(OPT)$) in polynomial time, but we would like to upper bound $p(OPT)$ and relate it to $p(ALG)$.

We note that for any object a,

$$K\lfloor p_a/K \rfloor = Kp'_a \leqslant p_a \leqslant K(\lfloor p_a/K \rfloor + 1) = K(p'_a + 1). \tag{6.1}$$

Thus,

$$p(OPT) = \sum_{a \in OPT} p_a \leqslant \sum_{a \in OPT} K(p'_a + 1) = K \sum_{a \in OPT} p'_a + K \sum_{a \in OPT} 1 \leqslant Kp'(OPT) + nK. \tag{6.2}$$

This is because $\sum_{a \in OPT} p'_a = p'(OPT)$ which is the sum of the rounded version of the profits of the objects in OPT; and there can be at most n items in OPT.

The key observation is that the set ALG returned by the dynamic programming algorithm must be at least as good as OPT *under the rounded profits*. This is because dynamic programming computes the optimal solution with respect to the rounded profits.

Hence,

$$p'(OPT) \leqslant p'(ALG).$$

Substituting the above in the Equation 6.2, we have

$$p(OPT) \leqslant Kp'(ALG) + nK = K \sum_{a \in ALG} p'_a + nK = \sum_{a \in ALG} Kp'_a + nK$$

$$\leqslant \sum_{a \in ALG} p_a + nK = p(ALG) + nK.$$

The bound $Kp'_a \leqslant p_a$ is from inequality 6.1.

Thus, we have,

$$p(ALG) \geqslant p(OPT) - nK = p(OPT) - n\epsilon P/n = p(OPT) - \epsilon P$$

$$\geqslant p(OPT) - \epsilon p(OPT) = (1 - \epsilon)p(OPT),$$

as desired (in the last term, we used the fact that $OPT \geqslant P$, since the optimum is at least the profit of the most profitable object, since we assume all objects have weight bounded by the capacity). □

Theorem 6.2

The above algorithm is an FPTAS for Knapsack.

Proof. The solution is within $(1 - \epsilon)OPT$.

The running time of the algorithm is

$$\mathcal{O}\left(n^2 \left\lfloor \frac{P}{K} \right\rfloor\right) = \mathcal{O}\left(n^2 \left\lfloor \frac{n}{\epsilon} \right\rfloor\right) = \mathcal{O}\left(n^3/\epsilon\right)$$

which is polynomial in n and $1/\epsilon$. □

6.8 DP Summary

We summarize the common themes underlying the application of DP to various problems that we studied in this chapter.

Key ingredients of DP

For DP to apply for a problem — typically, an optimization problem — it should have two key ingredients:

1. *Optimal substructure property*: The optimal solution is computed from optimal solutions to *independent* subproblems. Hence, instead of looking at all feasible solutions for the subproblems (which can be very inefficient), it is enough to look at *only optimal solutions* for the subproblems.

2. *Overlapping* subproblems. Overlapping means the same subproblem occurs in the solution of many different subproblems. We compute the solution of each subproblem *only once.*

Two algorithmic implementations

As was illustrated by various examples, once we formulate the all-important recursive formulation of the problem, there are two algorithmic ways for solving DP-based problems.

1. *Iterative algorithm:* Based on the recursive formula we write an *iterative* algorithm that solves the recursion in a *bottom-up* manner. That is, it first solves the base-case subproblems and then progressively solves subproblems of increasing size based on the the solutions to smaller-sized subproblems. Note that recursive formula shows how to compute a subproblem of size k from subproblems of smaller size. This iterative strategy is called *dynamic programming* which is a *bottom-up* strategy. (The term "programming" is actually a misnomer and has been used for historical reasons.)

2. *Recursive algorithm:* An alternate way is to solve the problem recursively using the recursive formula directly, i.e., by writing a *recusrive* code. However, as we saw in the problems we studied, a naive recursive implementation can solve the same subproblems repeatedly and can be very inefficient. Hence we have to the strategy of *table look up*, also known as *memoization.* Compute the solution of each subproblem *only once* and store the result in a table. When the same subproblem is encountered subsequently, get the value by looking up the table. This strategy is called *memoization* or *table look up* — which is a *top-down strategy*, since it is a recursive stategy.

6.9 Worked Exercises

Worked Exercise 6.1. Suppose you want to dial a telephone number consisting of n digits on a standard 12-button telephone keypad, using the index fingers of both hands. Assuming both fingers start over the 1 button, devise an $\mathcal{O}(n)$ time algorithm using dynamic programming (in particular, give the recursive formulation) which determines which finger to use for each of the n digits so as to minimize the sum of the total distance traveled by both fingers. Assume that the distance between two buttons i and j is given by the function $dist(i, j)$. You should first give the recursive formulation, clearly explaining what the subproblems are. Then devise two efficient algorithms that implements your recursive formulation — a recursive algorithm (using top-down, lookup strategy) and an iterative (DP, bottom-up strategy). Analyze the running time of your algorithms.

Solution. Let S be the n-digit number to be dialed. ($S[i]$ contains the ith digit). Let the array $D_i(k)$ be the minimum distance needed to dial (starting from both fingers resting on button 1) the first i digits of the phone number such that one of the fingers rests on the button $S[i]$ (after dialing $S[i]$) and the other rests on the button k. (Without loss of generality we will assume that $S[i-1] \neq S[i]$, otherwise $D_i(k) = D_{i-1}(k)$).

Note that computing $D_i(k)$ for different is and ks are the **subproblems** that we need to solve. The number of subproblems is at most $12n$, since there are n possible values for i and there are at most 12 values (buttons) for k.

Then for $i \geqslant 1$ we have the following recursive formulation:

$$D_i(k) = \begin{cases} D_{i-1}(k) + dist(S[i-1], S[i]) & \text{if } k \neq S[i-1] \\ \min\limits_{1 \leqslant j \leqslant 12} \{D_{i-1}(j) + dist(j, S[i])\} & \text{if } k = S[i-1] \end{cases}$$

and the base cases are: $D_0(k)$ is 0 if $k = 1$, and ∞ otherwise, and we assume that $S[0] = 1$. Note that the $dist$ function gives the distance between two buttons. Why is (1) and (2) correct? The first case, i.e., when $k \neq S[i-1]$ is the case where the other button (i.e., k) after dialing $S[i]$ should not be $S[i-1]$; of course, this can happen only by having the other button at k after dialing $S[i-1]$ and then moving the finger on $S[i-1]$ to dial $S[i]$. In this case, the cost is $dist(S[i-1], S[i])$ and one should move the finger that was on $S[i-1]$ to dial $S[i]$ (leaving the other finger sitting on k).

The second case is when $k = S[i-1]$ is the case where the other button (i.e., k) after dialing $S[i]$ should be $S[i-1]$; in this case, there are many possibilities. In particular, we look at possibilities $D_{i-1}(j)$ where the other button after dialing $S[i-1]$ is j and we move this other button to k (leaving one finger on $S[i-1]$ as desired in this case).

The original problem that we are looking to solve is $D_n(k)$ and the minimum cost is the minimum of $D_n(k)$ over all k.

It is straightforward to transform the above formulation into pseudocode which runs in time $\mathcal{O}(n)$: each entry in D_i array takes $\mathcal{O}(1)$ time to fill and there are only 12 entries to fill, and there are n such arrays to construct. It is also easy to recover the finger movements from the above arrays: look at the minimum entry in the D_n array and trace backwards.

□

Worked Exercise 6.2. Consider the 0/1 Knapsack problem with two knapsacks having capacities m_1 and m_2 respectively. We have n items $x_1, \ldots, x_n$. Item x_i has an associated weight w_i and profit p_i. The goal is to fill as many items as possible in the two knapsacks so that we maximize the total profit. An item can be chosen to be put in either of the two knapsacks (if possible) or not chosen at all. Assume that weights, profits, m_1, and m_2 are all positive integers.

(i) Show that the optimal substructure property holds for the problem.

(ii) Let $f_i(y_1, y_2)$ be the optimal solution for the subproblem restricted to the first i items $x_1, \ldots, x_i$ such that the two knapsacks have capacities y_1 and y_2 respectively. Give a recursive formulation for computing $f_i(y_1, y_2)$.

(iii) Give the pseudo-code of the resulting algorithm. You can give either a DP algorithm or a memoized algorithm. Analyze the running time of your algorithm.

(iv) Describe how to change the algorithm so that it also generates the optimum solution (i.e., which items are chosen and in which knapsack).

Solution.

(i) Let i be an item in an optimal solution S with knapsacks have capacities m_1 and m_2, respectively. Assume first that i is in knapsack 1. Let S' is the solution derived from S for capacities $m_1 - w_i$ and w_2. If S' would not be an optimal solution, we could generate a better solution containing item i and S would not be optimal, a contradiction. A similar argument holds when i is in knapsack 2. Hence, the principle of optimality holds and we can use dynamic programming for generating an optimal solution.

(ii) The recursive formulation is similar to the Knapsack problem in the textbook. However, since we now have two knapsacks, there exist two possibilities on where to place item i, in addition to the possibility of not using it at all. Hence:

$$f_i(y_1, y_2) = \max\{f_{i-1}(y_1, y_2), f_{i-1}(y_1 - w_i, y_2) + p_i, f_{i-1}(y_1, y_2 - w_i) + p_i\}$$

Base cases:

$$\begin{aligned} f_0(y_1, y_2) &= 0 \text{ for all } y_1, y_2 \geqslant 0 \\ f_i(y_1, y_2) &= -\infty \text{ if } y_1 < 0 \text{ or } y_2 < 0 \text{ for all } i > 0 \end{aligned}$$

(iii) We use n matrices T_1, T_2, ..., T_n, each of size $m_1 \times m_2$. We also use a matrix $m_1 \times m_2$ matrix T_0 which has all of its entries initialized to 0. $T_i(r, c)$ contains the value of the optimal solution using elements 1, ..., i and filling the knapsacks to r and c, respectively. We determine T_i from T_{i-1} by using the recursive formulation to compute an entry in constant time. This means all the entries in matrix T_i is computed in $\mathcal{O}(m_1 \cdot m_2)$ time. The matrices are generated in increasing index order of i and the entries in T_{i-1} are used to compute the entries in T_i. As there are n matrices, the total time is $\mathcal{O}(n \cdot m_1 \cdot m_2)$ time. $T_n(m_1, m_2)$ contains the required optimal solution.

(iv) To generate the items to be included into knapsack 1 and 2, the T matrices need to contain an additional backtrack entry which records whether item i was used to generate $T_i(r, c)$, and if it was used, whether it was placed into knapsack 1 or 2. (A variable with values 0, 1, 2 will handle this). To backtrack and generate the sets, start at $T_n(w1, w2)$.

- If the backtrack entry is 0, continue in matrix $T_{n-1}(w1, w2)$.
- If the backtrack entry is 1, add item n to knapsack 1 and look at the entry $T_{n-1}(w1 - p_1, w2)$.
- If the backtrack entry is 2, add item n to knapsack 2 and look at the entry $T_{n-1}(w1, w2 - p2)$.

The process continues in this fashion, costing $\mathcal{O}(1)$ time per step, until the knapsacks are filled or all matrices have been visited. This gives $\mathcal{O}(n)$ total time.

□

6.10 Exercises

Exercise 6.1. Write a recurrence for computing the running time of Algorithm 23. Solve the recurrence and show that it takes $\Theta(n^2)$ time.

Exercise 6.2. Let $P(n)$ be the number of different ways of parenthesizing a product of n matrices. Use mathematical induction to prove that $P(n)$ is $\Omega(2^n)$.

Exercise 6.3. Show that the naive algorithm Cost takes exponential time.

Exercise 6.4. Show that the run time of algorithm MatrixCostDP is $\mathcal{O}(n^3)$. Show that it correctly computes the optimal cost.

Exercise 6.5. What is the run time of the algorithm MatrixCostRec?

Exercise 6.6. Modify the recursive formulation that enables computation of the optimal paranthesization. Modify the DP (iterative) algorithm so as to incorporate this additional computation. Give the algorithm and the pseudocode for the same.

Exercise 6.7. Consider the following *variant* of the max-sum problem.

Given an array A of size n containing positive and negative integers, the goal is to determine indices i and j, $1 \leqslant i \leqslant j \leqslant n$, such that $A[i] + A[i+1] + \cdots + A[j] - A[k]$ is a **maximum**, where k can be any index between i and j, i.e., $i \leqslant k \leqslant j$. In other words, we are allowed to exclude **up to** one element (any one) in the contiguous subarray $A[i], \ldots, A[j]$, including $A[i]$ or $A[j]$.

Give a dynamic programming (DP) algorithm that computes the optimum value of the modified max-sum problem, i.e., we are interested in just computing the value of the optimum solution.

1. Specify the subproblems, clearly explaining your notation.
2. Give a recursive formulation that relates how you can solve larger-sized subproblems into smaller-sized subproblems. Also mention the base cases.
3. Implement your formulation by a DP algorithm and a memoized recursive algorithm.
4. Analyze the runtime of your algorithm.

Exercise 6.8. Let x and y be strings of symbols from some alphabet set of length m and n respectively (m need not be equal to n). Consider the operations of deleting a symbol from x, inserting a symbol into x and replacing a symbol in x by another symbol (belonging to the alphabet set). Your goal is to design an efficient algorithm using dynamic programming to find the minimum number of such operations (as well as to find the operations that transform x to y that correspond to the minimum number) needed to transform x into y. Your algorithm should run in time $\mathcal{O}(mn)$. (Hint: This problem is similar to the string alignment problem.)

1. Suppose the two strings are $x =$ "*algorithmic*" and $y =$ "*agonistic*" and the alphabet set is the English alphabet, what is the minimum number of operations needed to transform x into y.
2. Specify the subproblems for computing the minimum *number* of operations, clearly explaining the notations you use.

3. Give a recursive formulation that relates how you can solve larger-sized subproblems into smaller-sized subproblems. Also mention the base cases.

4. Implement your formulation by a DP (iterative) algorithm.

5. Analyze the running time of your DP algorithm.

6. Modify your algorithm to output the optimal sequence of operations as well.

Exercise 6.9. You start on a long road trip at hotel a_0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point a_0. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n) which is your destination.

You would ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you stop at a hotel during a day, you stay there for the rest of that day and continue your trip the next day. If you travel x miles a day, the penalty for that day is $|200 - x|$ (i.e., the absolute value of the difference between x and 200). You want to plan your trip so as to minimize the total penalty — that is the sum, over all travel days, of the daily penalties. Your goal is to give an efficient dynamic programming algorithm that determines the optimal sequence of hotels at which to stop.

(a) Give a recursive formulation for this problem. Clearly explain your notation, in particular, what the subproblems are. Do not forget to mention the base case(s). (Hint: This problem has a similar approach to the matrix chain multiplication problem.)

(b) Give an efficient algorithm (give pseudocode) based on your recursive formulation. What is the running time of your algorithm and why? Note that your algorithm should not only output the minimum penalty value, but also the optimal sequence of hotels that gives this minimum penalty.

Exercise 6.10. There are n programs that are to be stored in some order on two computer tapes of length ℓ. Associated with each program i is a length $\ell(i)$, $1 \leqslant i \leqslant n$. Assume that $\sum_{i=1}^{n} \ell(i) \leqslant \ell$, where ℓ is the length of each tape. A program can be stored in either of the two tapes. If S_1 is the set of programs on tape 1, then the worst-case access time for a program is proportional to $\max\{\sum_{i\in S_1} \ell(i), \sum_{i\notin S_1} \ell(i)\}$. An **optimal** assignment of programs to tapes **minimizes** the worst-case access time. Design an efficient dynamic programming algorithm (first give the recursive formulation, clearly explaining your notation) to determine the worst-case access time of an optimal assignment, and analyze its running time. Assume that all the lengths of the programs and the lengths of the tapes are integers.

Exercise 6.11. Given a sequence of n distinct numbers, $a_1, a_2, \ldots, a_n$, our goal is to find a *longest increasing subsequence (LIS)* of the given sequence. For example, if the given sequence is $10, 15, 2, 13, 20, 25, 5, 7, 9$, a LIS is $10, 15, 20, 25$ (another LIS is $2, 5, 7, 9$) of length 4 (no other increasing subsequence has bigger length).

1. Give a dynamic programming algorithm to find the LIS of given sequence of n numbers in $\mathcal{O}(n^2)$ time. (Hint: Subproblems are similar in style to the maxsum problem.)

2. Give an $\mathcal{O}(n \log n)$ algorithm for the problem. (Hint: Using the fact that the LIS is increasing, can you efficiently add a new element to the existing LIS of the subsequences)?

Exercise 6.12. Show that all values $A(i, p)$ in the Knapsack problem (cf. Section 6.7.2) can be computed in $\mathcal{O}(n^2 P)$ time.

Exercise 6.13. This exercise focuses on obtaining a space-efficient algorithm for the sequence alignment problem (Section 6.5). Our goal is to obtain an $\mathcal{O}(mn)$ time algorithm for sequence alignment problem that *also outputs the alignment* using $\mathcal{O}(m+n)$ space.

1. Show how to modify the DP algorithm (Algorithm SimScoreDP) to output the optimal score (i.e., the similarity score) of the optimal alignment using only $\mathcal{O}(m+n)$ space. Argue that this modification does not help in outputting the optimal alignment itself in the same amount of space, except for outputting the alignment of the last character of each sequence.

2. The key idea is to find out the middle alignment point that *optimally* aligns the first half of string s with string t. That is we would like to find out the index $d = \text{midindex}(m, n)$ such that the optimal alignment of $s[1 \ldots m]$ and $t[1 \ldots n]$ can be split into an optimal alignment of $s[1 \ldots \lfloor m/2 \rfloor]$ and $t[1 \ldots d]$ and an optimal alignment of $s[\lfloor m/2 \rfloor + 1 \ldots m]$ and $t[d+1 \ldots n]$. Clearly, such a middle index d exists in t. The algorithm combines DP with divide and conquer. The idea is to use DP to find d first and then use divide and conquer to optimally align $s[1 \ldots \lfloor m/2 \rfloor]$ with $t[1 \ldots d]$ and $[s\lfloor m/2 \rfloor + 1 \ldots m]$ with $t[d+1 \ldots n]$.

   (a) To find $\text{midindex}(m, n)$, as usual we consider all subproblems of the form $\text{midindex}(i, j)$ ($1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$) as the middle alignment point that aligns $s[1 \ldots i]$ with $t[1 \ldots j]$, i.e., if $d' = \text{midindex}(i, j)$, then an optimal alignment of $s[1 \ldots i]$ with $t[1 \ldots j]$ (for $i \geqslant \lfloor m/2 \rfloor$) can be split into an optimal alignment of $s[1 \ldots \lfloor m/2 \rfloor]$ with $t[1 \ldots d']$ and an optimal alignment of $s[\lfloor m/2 \rfloor + 1 \ldots i]$ and $t[d'+1 \ldots j]$. Write the recursive formula for computing $\text{midindex}(i, j)$ ($1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$).

   (b) Show how to compute $\text{midindex}(m, n)$ by modifying the DP algorithm for computing the optimal alignment. Your algorithm should run in $\mathcal{O}(mn)$ time and $\mathcal{O}(m+n)$ space.

   (c) To output the optimal alignment we recursively output the optimal alignment between $s[1 \ldots \lfloor m/2 \rfloor]$ and $t[1 \ldots d]$ and between $[s\lfloor m/2 \rfloor + 1 \ldots m]$ and $t[d+1 \ldots n]$, where $d = \text{midindex}(m, n)$. (Crucially to save space, we *reuse* the space between the recursive calls.) The recursion bottoms out when one string has constant length when we can determine the optimal alignment in linear time and space using DP. Write a recurrence for the running time of the algorithm and show that the overall running time is $\mathcal{O}(mn)$.

   (d) Write a recurrence for the space complexity of the algorithm and show that the overall space used is $\mathcal{O}(m+n)$.

Exercise 6.14. Give a memoized recursive algorithm for the alignment problem.

Modify the recursive formulation to output the optimal alignment as well.

Exercise 6.15. The **subset-sum** optimization problem is as follows. Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of positive integers and an integer t, find a subset of S with the largest sum less than or equal to t. The problem is known to be NP-hard and has no known polynomial time algorithm.

Our goal is to design an FPTAS algorithm for this problem, i.e., an approximation algorithm that outputs a solution that is within $(1-\epsilon)$ factor of the optimal solution and whose running time is polynomial in n and $1/\epsilon$.

(a) Consider the following algorithm

1. For $i = 0$ to n do
   - Compute all the sums bounded by t from subsets of up to i elements of S.

Show that the algorithm takes time that is exponential in n.

(b) Consider the following approximation algorithm.

1. For $i = 0$ to n do
   - Compute all the sums bounded by t from subsets of up to i elements of S.
   - Remove sums that are within $(1 + \frac{\epsilon}{2n})$ factor of other sums.

Show that the run time of the above algorithm is polynomial in n and $1/\epsilon$. (Hint: Let L_i be the collection of sums after the i-th iteration; bound the number of elements in L_i.)

(c) Show that the solution output by the algorithm is within a factor of $1-\epsilon$ of the optimal solution.

Exercise 6.16. Give pseudocode of the iterative DP algorithm for computing the optimal profit for the 0/1 Knapsack Problem. Give also pseudocode of the recursive algorithm (with table lookup) for computing the optimal profit. Analyze and show that the running time of both the algorithms is $\mathcal{O}(mn)$. Give an algorithm to output the optimal solution, i.e., the subset of objects that gives the optimal profit.

Exercise 6.17. You are given a stick of length n units. Your goal is to cut the stick into different pieces (each piece should be of integer length only) so that the total value of all the pieces is *maximized*. A piece of length i units has value v_i, for $i = 1, 2, \ldots, n$. You should design a dynamic programming algorithm that determines the maximum total value that is possible.

(i) Consider the sequence of 7 numbers: $2, 3, 4, 5, 6, 7, 9$. Let the i^{th} number in the sequence represents the value of piece of length i, i.e., the value of piece of length 1 is 2, of length 2 is 3, length 3 is 4 and so on. What is the best way to cut the pieces for a stick of length 7? What is the maximum value ?

(ii) Define the subproblems for your DP solution on a general instance of the problem where the stick is of length n and a piece of length i has value v_i.

(iii) Give a recursive formulation to solve the subproblems. Don't forget the base cases.

(iv) What is the running time of your solution?

(v) Write a DP algorithm (give pseudocode) that outputs the maximum value.

(vi) Describe an algorithm to output the sizes of the pieces of the cut that corresponds to the maximum value.

Exercise 6.18. Assume that you have a machine that cleans a floor modeled as a $m \times n$ grid consisting of mn cells. Each cell may have some dirt. Assume that the machine starts its cleaning from the upper left cell of the grid and in each step it can move either one cell to its right or one cell down from its current location. When it is in a cell, it can pick up the dirt in that cell and clean it. The goal is to clean as many dirty cells as possible. You should design a DP algorithm for the machine to clean the maximum number of dirty cells in the grid and the path it has to follow to achieve this. Assume that you know which cells have dirt in the entire grid.

1. Define the subproblems for your DP solution.
2. Give a recursive formulation to solve the subproblems. Don't forget the base cases.
3. What is the running time of your solution?
4. Write a DP algorithm (give pseudocode) that outputs the maximum number.
5. Describe an algorithm to output the path that achieves the maximum number.

7 GREEDY ALGORITHMS

Greedy algorithms (or the greedy technique), like Dynamic Programming (DP), are often used in solving **optimization** problems. As described in Chapter 6, in an optimization problem, typically, we are given a set of inputs and the goal is to find an output that satisfies some constraints. Any output that satisfies these constraints is called a *feasible* solution. In an optimization problem, we need to find a feasible solution that *maximizes* or *minimizes* a given objective function. A feasible solution that does this is called an *optimal solution*.

Greedy algorithms or the greedy technique can be considered as a special case of DP in the following sense. Recall that in DP, we decompose a problem into one or more subproblems. Most often the case, there will be more than one subproblem arising from the decomposition and one has to solve all the subproblems (recursively) to find the solution to the original problem. In greedy, the situation is special — typically we get only one subproblem to solve in the decomposition. This makes the algorithm particularly efficient, since in each "step" we solve only one subproblem. The key in greedy is choosing the correct (one) subproblem to solve; the non-trivial part is to show that it is *enough* to consider just this particular subproblem and ignore the other subproblems since they do not lead to the (optimal) solution. Unlike DP, problems should have a special structure that allows greedy algorithms to work.

The greedy technique works in stages. At each stage, a decision is made depending on whether it is "best" at this stage (hence the name "greedy"). For example, a simple criterion can be whether the decision made at the current stage will lead to a feasible solution or not. Thus, greedy choice (that looks good currently) is made in the hope that it will lead to a **globally optimal** solution. And if greedy succeeds, there will be no need for backtracking.

Some important problems admit greedy solutions that are *provably optimal*, i.e., the greedy algorithm indeed produces the optimal solution. We will see some examples this chapter and in later chapters as well. While greedy does not help in solving a lot of optimization problems, when they do work they are usually very efficient and simple. In some cases, even if they do not give the optimal solution, they can give a provably "good" solution (one that has a value close to the optimal).

7.1 Problem: Fractional Knapsack

The knapsack problem is a classical problem in optimization.

Problem 7.1 ▶ Fractional Knapsack

Given n objects with weights and profits $w_1, w_2, \ldots, w_n$ and $p_1, p_2, \ldots, p_n$, respectively, and and a knapsack with capacity m, choose $x_1, x_2, \ldots, x_m$ between 0 and 1 that *maximizes*

$$\sum_{i=1}^{n} x_i p_i$$

subject to the constraint

$$\sum_{i=1}^{n} x_i w_i \leqslant m.$$

Note that, unlike Problem 0/1 Knapsack considered in Section 6.6, in Problem Fractional Knapsack, we are allowed to choose a fraction of an objects. This change allows one to design a simple and efficient algorithm that runs in polynomial time and returns the optimal solution.

Before we outline an algorithm for the problem, we make the following observations:

- If the sum of all the weights is less than or equal to m, then $x_i = 1$, $1 \leqslant i \leqslant n$ is an optimal solution.
- All optimal solutions will fill the knapsack exactly.

7.1.1 Greedy Strategies

Consider the following instance of the Fractional Knapsack problem with $n = 3$ and $m = 20$:

Table 7.1: Fractional Knapsack with 3 items and maximum capacity 20

Item	Weight	Profit
I_1	18	25
I_2	15	24
I_3	10	15

The following are four different feasible solutions each with with a different profit: Many simple "greedy" strategies are possible:

1. Fill the knapsack by including next the object with largest profit. If an object under consideration does not fit, then a fraction of it is included to fill the knapsack (solution 2 in the above instance)
2. Fill the knapsack by including objects in nondecreasing order of weights (solution 3)
3. Consider objects in nonincreasing order of p_i/w_i (solution 4)

Different solutions for Table 7.1

Quantity			Total	
I_1	I_2	I_3	Weight	Profit
1/2	1/3	1/4	16.5	24.25
1	2/15	0	20.0	28.20
0	2/3	1	20.0	31.00
0	1	1/2	20.0	31.50

Table 7.2: Different solutions for Table 7.1

We can see that greedy strategy number 3, i.e., choosing objects in nonincreasing of p_i/w_i gives the highest value (31.25) among the other the four different solutions in the example above. Is this the best possible, i.e., the optimal solution? We will next prove that it is indeed so. Note that this greedy strategy can be implemented in $\mathcal{O}(n \log n)$ time via sorting the elements in nonincreasing order of p_i/w_i.

Note that in a greedy strategy, we consider only one subproblem at a time. For example, in the greedy strategy number 3, at each step, we choose the object with the highest profit to weight ratio among the items that are *not yet* chosen. The main point in greedy is to show that it is *enough* to consider just this ordering to get the optimal solution.

7.1.2 Proof of Optimality

Theorem 7.1

If objects are included in the nonincreasing order of p_i/w_i then this gives an optimal solution to the knapsack problem.

Before we prove this theorem, we outline the following *general proof technique* which is typically useful in proving optimality of greedy algorithms. Suppose we want to show that the solution output by a greedy algorithm is indeed the optimal solution. Assume that greedy outputs the solution $(x_1, x_2, \ldots, x_n)$ and the optimal solution is $(o_1, o_2, \ldots, o_n)$. We compare the greedy solution with the optimal solution. If the two solutions differ, then find the first index where they differ (say i). Then show how to make o_i in the optimal solution equal to x_i (of the greedy solution) without loss of the total value of the optimal solution. Repeated use of this transformation, transforms the optimal solution to greedy without losing optimality and thereby shows that the greedy solution is (also) optimal. This proof technique can be called as an "exchange" or "transform" argument.

We first give a simple and intuitive proof that clearly illustrates the above exchange argument under the assumption that all the weights, profits, and the capacity are integers (if they are rational numbers, then the proof can still be made to work by scaling the values appropriately). Later we give a proof without these assumptions using algebra.

Proof of Theorem 7.1 assuming integer values. Assume that all the weights, profits, and the capacity are integers. We consider the greedy solution which chooses objects in nonincreasing order of profit per unit weight. Let the objects chosen by the greedy solution, in the above order be $g_1, g_2, \ldots, g_i$ and the corresponding weights be $w(g_1), w(g_2), \ldots, w(g_i)$.

All the objects chosen by greedy are chosen *fully*, except, possibly, the last object g_i which could be fractional.

Let $o_1, o_2 \ldots, o_j$ be the objects chosen by the optimal solution and *arranged in nonincreasing order of profit per unit weight*; the corresponding weights are $w(o_1), w(o_2), \ldots, w(o_j)$. Note that some or all of the objects may be chosen in a fractional manner in the optimal solution.

Consider partitioning the knapsack into m unit weights. We note that both greedy and optimal solutions will fill up the knapsack fully. The greedy solution will fill the first $w(g_1)$ weight units with object g_1, the second $w(g_2)$ units with object g_2, and so on. Similarly the optimal solution will fill the first $w(o_1)$ weight units by object o_1 and so on.

We examine the ordering of the unit weights in an incremental manner, starting from the first unit weight in the knapsack. If the first unit weight in both greedy and optimal ordering belong to the same object, we continue to the second unit weight and so on. Let k be the first unit weight $(k \geqslant 1)$ where greedy and optimal disagrees. We now transform the optimal solution to agree with greedy in the kth unit weight as follows. Let the kth unit weight of greedy and optimal belong to objects g and o respectively. We simply *replace* the kth unit weight belonging to object o by a unit weight belonging to object g. The crucial point is that since we are examining objects in nonincreasing order of profit per unit weight, g's unit weight profit is at least as large as o's. Hence the value of the optimal solution does not decrease. Thus we have shown how to transform optimal to greedy so that one additional unit weight identifies with greedy. By continuing this argument, we will transform optimal to identify with greedy without loss of optimality. This implies that greedy solution is optimal.

□

We next give a proof that works for all weight values.

Proof of Theorem 7.1. Let $x = (x_1, \ldots, x_n)$ be the solution generated by the greedy algorithm. Since greedy packs the knapsack fully, we have $\sum w_i x_i = m$. If $x_i = 1$, for all i, then clearly the solution is optimal. Let j be the first index such that $x_j \neq 1$. Then, $x_i = 1$ for $1 \leqslant i < j$, $x_i = 0$ for $j < i \leqslant n$, and $0 \leqslant x_j < 1$.

Let $y = (y_1, \ldots, y_n)$ be an optimal solution. Note that crucially, we consider the optimal solution in the *greedy order*; this is crucial for the analysis below. Since y is optimal, we can assume that $\sum w_i y_i = m$. Let k be the least index such that $y_k \neq x_k$. Then $y_k < x_k$. To see this consider three possibilities:

- If $k < j$, then $x_k = 1$. Since $y_k \neq x_k$, and so $y_k < x_k$.
- If $k = j$, then since $w_i x_i = m$ and $y_i = x_i$ for all $1 \leqslant i < j$, either $y_k < x_k$ or $\sum w_i y_i > m$.
- If $k > j$, then $\sum w_i y_i > m$.

Suppose we increase y_k to x_k and decrease as many of $(y_{k+1}, \ldots, y_n)$ as necessary. This results in a new solution $(z_1, \ldots, z_n)$ with $z_i = x_i$, for $1 \leqslant i \leqslant k$ and

$$\sum_{i=k+1}^{n} w_i(y_i - z_i) = w_k(z_k - y_k)$$

We calculate the total profit for z:

$$\begin{aligned}\sum_{i=1}^{n} p_i z_i &= \sum_{i=1}^{n} p_i y_i + p_k(z_k - y_k) - \sum_{i=k+1}^{n} p_i(y_i - z_i)\\ &= \sum_{i=1}^{n} p_i y_i + (p_k/w_k)(z_k - y_k)w_k - \sum_{i=k+1}^{n} (p_i/w_i)(y_i - z_i)w_i\end{aligned}$$

Since we considered the items in greedy order, we have $p_k/w_k \geqslant p_i/w_i$, for all $i > k$. Hence the above expression is lower bounded by:

$$\begin{aligned}&\geqslant \sum_{i=1}^{n} p_i y_i + \left[(z_k - y_k)w_k - \sum_{i=k+1}^{n} (y_i - z_i)w_i\right](p_k/w_k)\\ &= \sum_{i=1}^{n} p_i y_i\end{aligned}$$

If $\sum_{i=1}^{n} p_i z_i > \sum_{i=1}^{n} p_i y_i$, then y cannot be optimal. Otherwise, z is optimal and either $z = x$ or $z \neq x$; in the case of the latter, repeat the argument with z. □

Exercise 7.1. Show that the optimal greedy algorithm for Fractional Knapsack is not optimal for 0/1 Knapsack.

7.2 Problem: Caching or Demand Paging

We consider another classical problem, namely caching or demand paging that is central to modern computer architectures that use cache memory. Assume that a processor has a fast memory device called **cache** that can store k items (pages), and a slower, much larger, secondary memory. If a processor requests an item that is in the cache (a **hit**) it has no cost. If the processor requests an item that is not in the cache (a **miss**) it costs one unit of time to **fetch** it to the cache from the secondary memory. When an item is fetched to the cache, another item is **evicted**. An item is fetched only when it is requested (hence demand paging). Given a sequence of item requests, the goal is to **minimize** the number of page misses. This is the optimization problem that we will study.

Possible eviction strategies

We can study several different eviction strategies as given below. These are popular strategies that are used in practice.

- Least Recently Used (LRU): Evict the page that was least recently used (i.e., was requested the farthest in the past in the sequence of requests).

- First-in First-out (FIFO): The page that entered the first also is evicted first.

- Least Frequently Used (LFU): The page that has been least frequently accessed (among the current set of pages in the cache) is evicted.

What is the best eviction policy? Suppose we know the complete sequence of page requests in advance. Then the following greedy strategy called the **Longest-Forward-Distance (LFD)** strategy is the best: If the requested page is not in the cache then evict the page whose next request is the farthest (in the sequence of requests).

Example

Consider the request sequence $a, b, c, d, a, d, e, a, d, b, c$ and a cache of size $k = 3$. LFD strategy will evict c on the 4th request and b on the 7th request. However, evicting b on the 4th step and c on the 7th step will also give the same misses.

7.2.1 Proof of Optimality of LFD

Theorem 7.2

LFD is an optimal eviction strategy for caching.

Proof. We will use the proof technique outlined earlier: take any optimal solution and modify it step by step until it becomes equal to the greedy solution, while at the same time not losing optimality. We will show how any optimal algorithm can be modified to behave like LFD without degrading its performance. The proof depends on the following claim.

Theorem 7.3 ▶ Caching Construction

Let ALG be an optimal caching algorithm. Let σ be any request sequence. For any $i, i = 1, 2, \ldots, |\sigma|$, it is possible to construct an algorithm ALG_i that satisfies the following 3 properties:

- ALG_i processes the first $i - 1$ requests exactly as ALG does.
- If the ith request results in a page fault, ALG_i uses the LFD strategy to evict a page.
- $\text{ALG}_i(\sigma) = \text{ALG}(\sigma)$, i.e., ALG_i has the same number of page misses as ALG on the entire sequence σ.

We can prove the Theorem by applying the Claim $n = |\sigma|$ times (starting from $i = 1$). □

Proof of Caching Construction. Given ALG we construct ALG_i. Given a set of pages X and a page p, let $X + p$ denote the set $X \cup \{p\}$. Without loss of generality, assume that the ith request resulted in a page fault. Assume that after processing the ith request, let the caches of ALG and ALG_i contain the page sets $X + v$ and $X + u$ respectively, where X is some set of $k - 1$ common pages and v and u are any pages. Assume that $u \neq v$, otherwise, ALG_i simply mimics ALG, i.e., makes the same evictions as ALG.

We consider two cases:

Case 1: Until v is requested: ALG_i mimics ALG to serve subsequent requests. If ALG evicts v then ALG_i evicts u, unless u itself has been requested. The number of common pages become k and ALG_i mimics ALG from now on. Also, number of page misses of ALG_i is at most that of ALG.

Case 2: If v is requested and assume that ALG and ALG_i have not yet identified. ALG_i will evict u and the two algorithms identify. But, ALG_i will incur a page fault and ALG will not. However, by the time v was requested, there must have been at least one request for u after the ith page fault. The first such request incurs a page

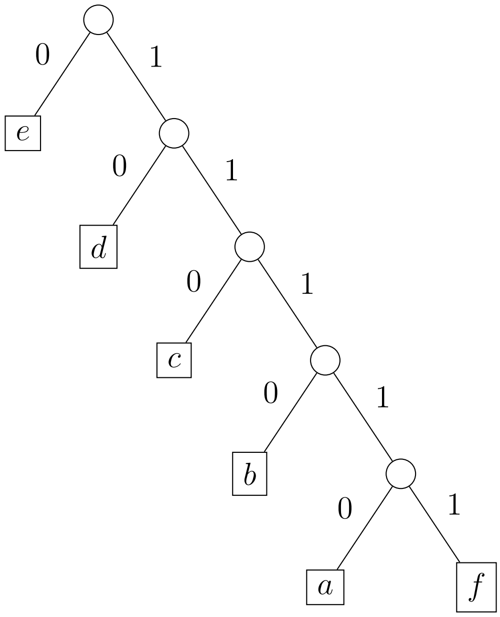


Figure 7.1: Binary tree representation of the prefix code of Example 7.1.

fault to ALG but not to ALG$_i$. Thus the total number of page faults for ALG$_i$ after servicing v is equal to that for ALG. □

7.3 Problem: Data Compression

Data compression is a fundamental problem where the goal is to compress a given data as much as possible while still being able to uniquely recover the original data from the compressed data. To compress data we use *codes.* We encode each character (sometimes even blocks of characters) using a alphabet set (typically binary). The encoded data should be *uniquely decodable* so as to get back the original data.

Given a set of characters belonging to an alphabet set C, a variable length code assigns to each character $\alpha \in C$, a (binary) code (string of 0s and 1s). Different characters can have different code lengths. Let $d(\alpha)$ be the length of the code for character α. Assume that the frequency of character α in the data is $f(\alpha)$. The **cost** of the code, called the *average code length*, is

$$B(C) = \sum_{\alpha \in C} f(\alpha) \cdot d(\alpha)$$

Example 7.1. Suppose your data consists of only 6 characters — a, b, c, d, e, f — and they occur with frequencies $0.05, 0.1, 0.2, 0.2, 0.4, 0.05$ respectively. Let the encoding be as follows. $e = 0$, $d = 10$, $c = 110$, $b = 1110$, $a = 11110$, $f = 11111$. Why is this code uniquely decodable? It is because no code is a prefix of any other code. So by scanning the encoded text from left to right, one can uniquely decode it. For example, the encoded text 110010 can be uniquely decoded to *ced.*

The cost of this code is $\sum_{\alpha \in C} f(\alpha) \cdot d(\alpha) = 0.05 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.2 \times 2 + 0.4 \times 1 + 0.05 \times 5 = 2.3$. Thus the *average length* of this encoding is 2.3.

Shannon, in his pathbreaking work, showed that a quantity called *entropy* is a fundamental lower bound on the *average* length of the code. In other words,

$$B(C) = \sum_{\alpha \in C} f(\alpha) \cdot d(\alpha) \geqslant H$$

where H is the entropy of the data set. The entropy is defined as

$$H = \sum_{\alpha \in C} f(\alpha) \log_2(1/f(\alpha))$$

Example 7.2. The entropy of the data in the above example is: $0.05 \log(1/0.05) + 0.1 \log(1/0.1) + 0.2 \log(1/0.2) + 0.2 \log(1/0.2) + 0.4 \log(1/0.4) + 0.05 \log(1/0.05) = 0.432 + 0.332 + 0.93 + 0.528 = 2.22$.

Prefix Codes

In a **prefix code** no codeword is a prefix of another code word. As mentioned in the above example, this allows for easy encoding and decoding. The prefix code can be conveniently represented as a binary tree where the characters are at the leaves and the path taken from the root to a leaf gives the code for that character. We use the usual interpretation of bit 0 for the left child and 1 for the right child. In an optimal code each non-leaf node has two children. Figure 7.1 gives the binary tree representation of the prefix code of Example 7.1. Note that there can be more than one prefix code for an alphabet set.

Exercise 7.2. We are given a text consisting of k characters and their respective frequencies. Show that *every* prefix code, with *binary code lengths* $\ell_1, \ell_2, \ldots, \ell_k$ (corresponding to k characters in the data) should satisfy the following inequality:

$$\sum_{i=1}^{k} \frac{1}{2^{\ell_i}} \leqslant 1.$$

7.3.1 Huffman Coding Algorithm

We now present a greedy algorithm — called *Huffman coding* algorithm — that generates an *optimal* prefix code. The algorithm is quite simple. It builds the prefix tree bottom up. In the first step, it combines the the least two frequent characters and makes them siblings of an internal node. Then it creates a new character whose frequency is the sum of the frequencies of these two. Then the process is again repeated — least two frequent characters are combined and made siblings of a new internal node and so on. In each step the number of characters is reduced by 1. Thus the algorithm finishes in $n-1$ steps.

Example 7.3. In Example 7.1, the Huffman code is constructed as follows. First, the least two frequent characters are combined — a and f — to give a new character af which has frequency 0.1. In the next step, af and b are combined to obtain afb with frequency 0.2. In the next step, c and d are combined to obtain cd with frequency 0.4. Then afb and cd are combined to obtain $afbcd$ with frequency 0.6. This is then combined with e to obtain the root of the tree. Thus the Huffman code obtained from this tree is: $e = 0, c = 100, d = 101, a = 1100, f = 1101, b = 111$. See Figure 7.2. The average code length of this code is 2.3. We will show next that this is the optimal average length attainable.

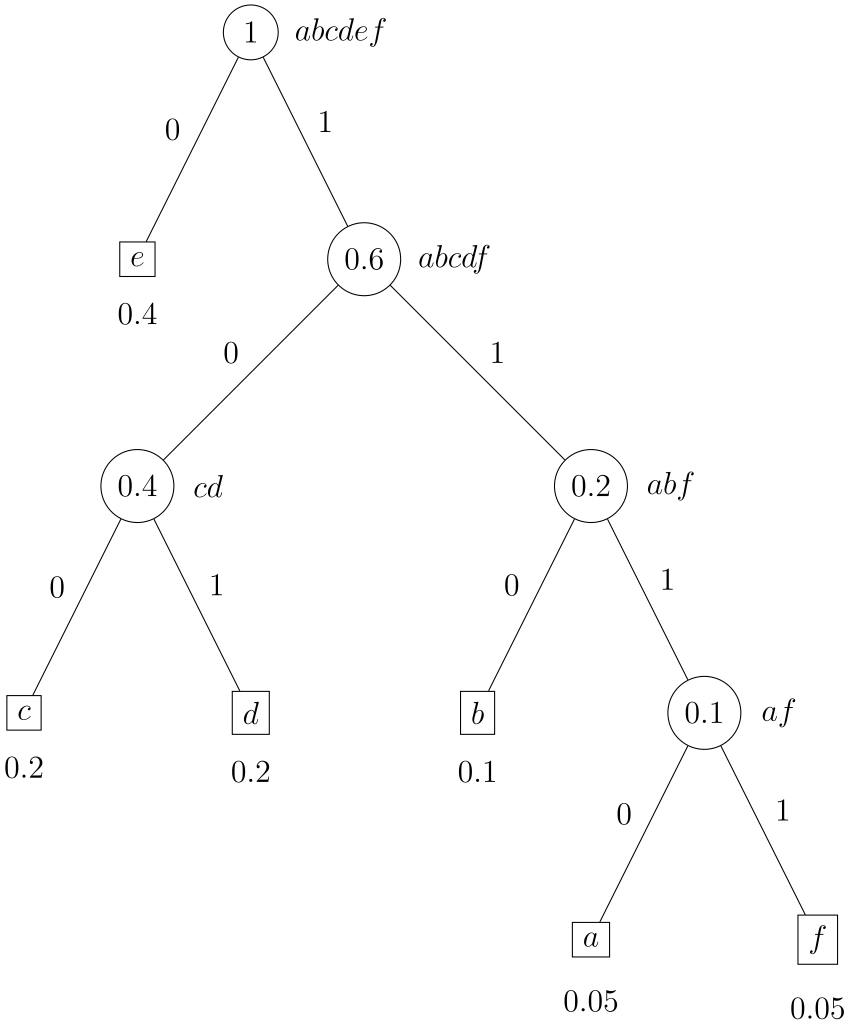


Figure 7.2: Binary tree representation of the Huffman code of Example 7.1.

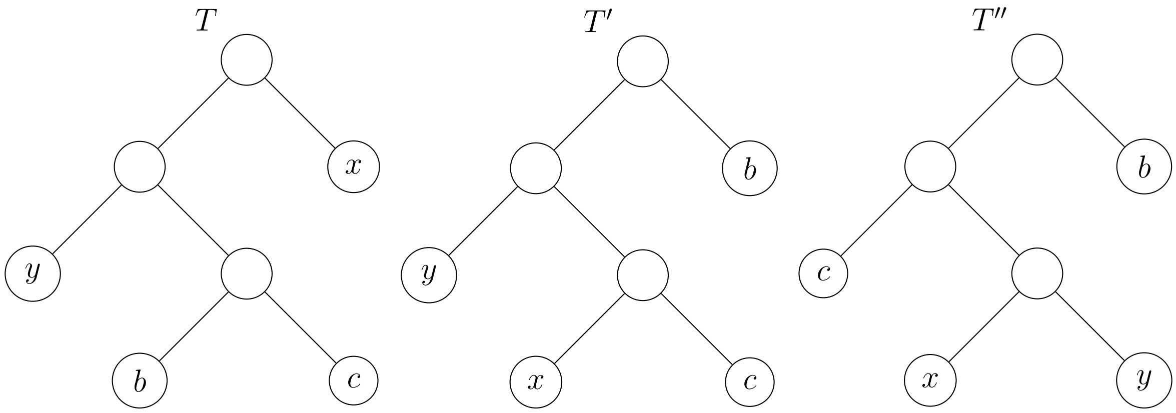


Figure 7.3: Illustration of proof of Lemma 7.5. x and y are the two lowest frequency characters. They can be made siblings at the lowest depth without increasing the cost of the code.

Implementing Huffman's algorithm efficiently.

To implement Huffman's algorithm efficiently, the key operation is to efficiently choose two characters with the lowest frequencies in each step. This can be accomplished by storing the characters in a *min heap* data structure (see Appendix B) with their frequencies as key values. Then the two lowest frequencies can be found by performing two ExtractMin operations each of which costs $\mathcal{O}(\log n)$ time. Adding a character whose frequency is the sum of the frequencies of the two lowest frequencies can be done by an Insert operation which again takes $\mathcal{O}(\log n)$ time. Hence the overall time to perform one step of Huffman's algorithm is $\mathcal{O}(\log n)$ and since there are $n-1$ steps, the overall time is $\mathcal{O}(n \log n)$. We leave the detailed implementation as an exercise (Exercise 7.3).

Thus we have the following theorem.

Theorem 7.4

The Huffman algorithm encodes n characters in $\mathcal{O}(n \log n)$ time.

Exercise 7.3. Prove Theorem 7.4.

7.3.2 Optimality of Huffman Coding

Lemma 7.5

Let x and y be the two characters in C with the lowest frequencies. There is an optimal prefix tree for C in which the nodes x and y are siblings at maximum depth and thus have the same length and differ only in the last bit.

Proof. Given a tree T of optimal prefix code of C we generate a new tree T'' by moving x and y to be siblings with maximum depth. Let b and c be two characters that are encoded by two sibling leaves of maximum depth. Assume $f(b) \leqslant f(c)$ and $f(x) \leqslant f(y)$, which implies $f(x) < f(b)$ and $f(y) < f(c)$. Exchange x and b to generate a tree T' and then y and c to generate tree T''. See Figure 7.3. We show that the exchange can only reduce the cost.

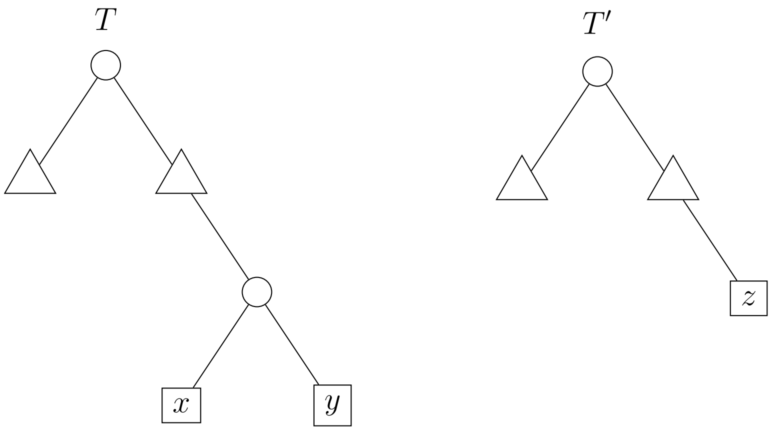


Figure 7.4: Illustration of proof of Lemma 7.6.

We first show that exchanging x and b cannot increase the cost.

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\ &= (f(b) - f(x))(d_T(b) - d_T(x)) \geqslant 0. \end{aligned}$$

We can give a similar argument for the move from T' to T''. □

Lemma 7.6

Let T be a tree representing an optimal prefix code for C. Consider two characters x and y that appears as siblings in the tree. Let z be their parent in the tree. Consider z a character with frequency $f(z) = f(x) + f(y)$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for the alphabet $C' = C - \{x, y\} \cup \{z\}$.

Proof. See Figure 7.4.

$$B(T) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y)$$

Since, $c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$, we can write:

$$\begin{aligned} B(T') &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(z) d'_T(z) \\ &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + (f(x) + f(y)) d'_T(z) \end{aligned}$$

Since, $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have:

$$\begin{aligned} B(T') &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x)(d_T(x) - 1) + f(y)(d_T(y) - 1) \\ &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) - f(x) - f(y) \\ &= B(T) - f(x) - f(y) \end{aligned}$$

Hence,

$$B(T) = B(T') + f(x) + f(y)$$

If T' is not optimal, there is a tree T'' such that $B(T'') < B(T')$. Replacing z with x and y in T'' will give a code with cost

$$B(T'') + f(x) + f(y) < B(T)$$

This contradicts the fact that T was optimal. Hence T' is optimal. □

Theorem 7.7

The procedure Huffman generates an optimal prefix code.

Proof. Lemma 7.5 shows that at every step, to get an optimal cost tree you can take the two lowest frequency characters and make them siblings at the lowest depth. Lemma 7.6 shows that you can merge the two lowest frequency characters and continue with the remaining set of characters; the resulting tree will still be optimal with respect to the new characters. Hence applying both lemmas repeatedly in each step, shows that the final tree is optimal. □

7.4 Problem: Set Cover

The set cover problem is a fundamental optimization problem that arises in many applications. Many other optimization problems can be reduced to solving set cover. The set cover problem is defined as follows:

Problem 7.2 ▶ Set Cover

Given a ground set U and a collection of non-empty subsets S of U, find the *minimum* subset of S (i.e., containing as few subsets as possible) whose union equals U.

Consider the following example.

Example 7.4. Let the ground set $U = \{1, 2, 3, \ldots, 10\}$ and the collection of subsets $S = \{\{1, 2, 3\}, \{1, 2, 3, 4, 5\}, \{3, 4, 5\}, \{6, 7, 8\}, \{8, 9, 10\}\}$. One way to cover the ground set is by using the first, third, fourth and the fifth sets. This takes four sets. Alternatively, one can cover using only three sets in the collection, namely, the second, fourth, and the fifth. This is the optimal solution.

The set cover problem is a classic NP-hard optimization problem, and hence it is unlikely to have a polynomial time algorithm that finds the optimal solution. In fact, it is NP-hard to even *approximate* the solution to within a *logarithmic factor* of the optimal. Recall (from Section 6.7) that the approximation factor for a minimization problem such as set cover is the ratio of the solution output by the algorithm to the optimal solution. We show that a simple greedy algorithm gives a logarithmic factor approximation, which is the best possible.

7.4.1 A Greedy Algorithm

Assume that we are given a ground set U and a collection S of subsets of U. Assume that every member of U is contained in some subset in the collection (hence a solution always exists).

We will show that the following *greedy algorithm* gives an $\mathcal{O}(\log \Delta)$ approximation, where Δ is the size of the largest set. The algorithm proceeds in iterations. In each iteration, it includes the set (which is chosen from the set collection) that covers the most *uncovered* elements.

We leave an efficient implementation of the greedy set cover algorithm as an Exercise (see Exercise 7.11).

Algorithm 32 GreedySetCover
Input: A ground set U and collection of subsets S
Output: A collection of sets C that covers U

```
func GreedySetCover($U$, $S$):
    $C = \emptyset$
    while $C \neq U$:
        $A$ = an element of $S$ which maximizes the number of uncovered elements
        $C = C \cup A$
    return $C$
```

7.4.2 Performance of Greedy Algorithm

We show that the greedy algorithm gives an $\mathcal{O}(\log \Delta)$ approximation to the optimal.

Theorem 7.8

The greedy algorithm gives a set cover of size that is within $\mathcal{O}(\log \Delta)$ factor of the optimal.

Proof. Let r_i be the number of elements that are uncovered at the *beginning* of iteration i by the greedy algorithm. This means that $r_1 = n$. Let t be the number of subsets in the optimal cover. At the start of iteration i, since the optimum covers all the elements with t number of sets, there is at least one set which has at least r_i/t uncovered elements. Suppose not. Indeed, if each set covers less than r_i/t uncovered elements in this iteration, then this would imply that one cannot cover all the elements using t sets (which the optimal does). Note that this argument applies regardless of some of the sets chosen by greedy being in the optimal cover, before the beginning of iteration i.

Hence, greedy will cover at least r_i/t uncovered elements in iteration i. Thus the number of elements that remain uncovered at the beginning of iteration $i + 1$ is

$$r_{i+1} \leqslant r_i - r_i/t = r_i\left(1 - \frac{1}{t}\right)$$

From the above recurrence, we have

$$r_{i+1} = r_1\left(1 - \frac{1}{t}\right)^i = n\left(1 - \frac{1}{t}\right)^i$$

Setting $i = t\ln(n/t)$, we have

$$\begin{aligned} r_{i+1} &\leqslant n\left(1-\frac{1}{t}\right)^{t\ln\left(\frac{n}{t}\right)} \\ &\leqslant ne^{-\frac{1}{t}t\ln\left(\frac{n}{t}\right)} \\ &= t \end{aligned}$$

Thus, after picking $t\ln(n/t)$ sets, there are only t elements left that are (still) uncovered; this can be covered by picking at most t more sets. Hence the number of sets picked by greedy is at most

$$t\ln\left(\frac{n}{t}\right) + t = \mathcal{O}\left(t\log\left(\frac{n}{t}\right)\right)$$

Since $t \geqslant n/\Delta$ (see Exercise 7.9), we have that the approximation ratio of greedy is $\mathcal{O}(\log(n/t)) = \mathcal{O}(\log\Delta)$. □

7.5 Worked Exercises

Worked Exercise 7.1. Consider a highway that is M miles long consisting of L lanes. At every mile, there exists a lane that is blocked from mile i to (just before) mile $i+1$. This information is recorded in an array $LC[0,1,\ldots,M-1]$ of length M, where $LC(i) = j$ means lane j is blocked from mile i to $i+1$. For example, $LC = [1,1,3,4,1,2,2,5,4,2,2]$ describes the blockages on a 5-line highway of length 11. The objective is to find a route making the minimum number of lane changes. Changing from one lane into another one has a cost of 1. For the example given above, a solution with one lane change exists: start in lane 5 and change into lane 1 at mile 6 or later. It is not hard to see that more than one optimal solution can exist.

(i) For $M = 12$ and $L = 4$ give an example of blocked positions that maximizes the number of lane changes. State the bound in terms of M and L and explain your answer.

(ii) Consider now a greedy algorithm that starts in the lane having the farthest occurrence of a blockage and, whenever forced to switch lanes, it always switches into the lane having the farthest occurrence of a blockage. In the above example, greedy will start in lane 5 (since lane 5 has the farthest occurrence of the blockage) and at mile 6 change into lane 1 or lane 3 (these two lanes have no further blockages). Describe this greedy approach in pseudo-code and analyze its time performance in terms of M and L. Do not assume that either of these variables is a constant. Make sure to describe what data structures are used to select the next lane after a lane change.

(iii) Prove that this greedy approach always generates a solution minimizing the number of lane changes.

Solution.

(i) $M = 12$ and $L = 4$. Example of blocked position: $LC = [1,2,3,4,1,2,3,4,1,2,3,4]$ In the above example, every lane is blocked every L miles. Thus, every solution must change lane at least once every L miles, in general. This is the upper bound of the maximum number of lane changes in an optimal solution is M/L time.

(ii) The greedy algorithm uses the following idea: The path starts in the lane that has its first obstacles the latest. Then, when a lane encounters an obstacle, change into the lane which has the next obstacle furthest down the road. This approach can be implemented in $\mathcal{O}(M + L)$ time by scanning highway of length M and using an array of size L to assist in determining the next lane to switch into. The switch happens at most every M/L positions.

We give more detailed as customary in a detailed algorithmic description as some had problems with the details. We use an array `lane[i]` to store the lanes which are blocked since the last change of lane.

Algorithm 33 FindRoute
Input: Number of lanes, length of highway, lane blockings
Output: Route with minimum number of lane changes

```
 1: func FindRoute($L$, $M$, $LC$):
 2:     count = 0   ▷ Number of lanes that has been blocked
 3:     changes = 0   ▷ Number of lane changes that has been made
 4:     for $j = 0$ to $L - 1$:
 5:         lane[$j$] = False
 6:     for $i = 0$ to $M - 1$:
 7:         if not lane[$LC[i]$]:
 8:             lane[$LC[i]$] = True
 9:             count += 1
10:             if count == $L$:
11:                 solution[changes] = ($LC[i], i - 1$)
                    ▷ Use lane $LC[i]$ until mile $i - 1$
12:                 changes += 1
13:                 for $j = 0$ to $L - 1$:
14:                     lane[$j$] = False
15:                 lane[$LC[i]$] = True
                    ▷ The same lane could not be chosen successively
16:                 count = 1
17:     for $j = 0$ to $L - 1$:
18:         if lane[$j$] == False:
19:             solution[changes] = ($j, M$)   ▷ Choose the last lane
20:             break
21:     return solution
```

Time Complexity Analysis

The for-loops at lines 4 and 17 will take $\mathcal{O}(L)$ time each. The for-loop at line 6 will take $\mathcal{O}(m)$ time. However, since the if-statement at line 10 will run at most $\mathcal{O}(M/L)$ time by the result of part (i), the inner for-loop at line 13 will run at most $\mathcal{O}(M/L) \cdot \mathcal{O}(L) = \mathcal{O}(M)$ time. As a result, the total running time is $\mathcal{O}(M) + \mathcal{O}(L) = \mathcal{O}(M + L)$.

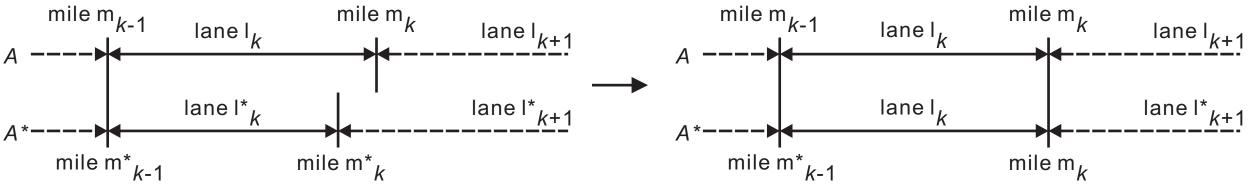
(iii) Suppose our greedy solution is different from the optimal solution. Let

$$A = \{(l_1, m_1), (l_2, m_2), \dots, (l_M, m_M)\}$$

be our greedy solution and

$$A^* = \{(l_1^*, m_1^*), (l_2^*, m_2^*), \ldots, (l_N^*, m_N^*)\}$$

be an optimal solution. Since the optimal solution should make the fewest number of lane changes, $N \leqslant M$. Suppose our greedy solution and the optimal solution first disagree at the kth lane change, that is $(l_i, m_i) = (l_i^*, m_i^*)$ for $i < k$. Now, since we know that our greedy solution will always choose the furthest unblocked lane, $m_k^* < m_k$. From the greedy solution, we know that lane l_k is unblocked from mile $m_{k-1} + 1$ to m_k. Therefore, we could change the optimal solution by replacing (l_k^*, m_k^*) with (l_k, m_k) without changing the total number of lane changes. This means that the optimal solution now use lane l_k until mile m_k and change to l_{k+1}^* afterwards. Repeating this procedure, eventually we will get $A = A^*$. We could then conclude that our greedy algorithm gives the optimal solution.



□

7.6 Exercises

Exercise 7.4. Consider the Huffman algorithm. Assume that the set of characters and their frequencies are given in sorted order, i.e., in increasing order of frequencies. Then show that the Huffman algorithm can be implemented in $\mathcal{O}(n)$ time.

Exercise 7.5. You are given a set of n jobs. Associated with job i is an integer is an integer $d_i \geqslant 0$ and a profit $p_i > 0$. For any job i, the profit p_i is earned if and only if the job is completed by its deadline. To complete a job, one has to process the job on a machine for exactly one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset F of the jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution F is $\sum_{i \in F} p_i$. An optimal solution is a feasible solution with maximum value.

Now consider the following greedy algorithm to find an optimal solution F of the above job sequencing problem.

Algorithm 34 GreedyJobSequencing – Greedy algorithm to find an optimal solution F of the above job sequencing problem.

```
Sort the jobs in decreasing order of the $p_i$'s, say $a_1, a_2, \ldots, a_n$.
$F = \{a_1\}$
for $k = 2$ to $n$:
    if $F \cup \{a_k\}$ is a feasible solution:
        $F = F \cup \{a_k\}$
```

a) Show that the above greedy method obtains an optimal solution to the job sequencing problem.

b) Give an $\mathcal{O}(n \log n)$ time algorithm to check feasibility of a solution.

Exercise 7.6. Assume that there is a set of n distinct points on the x-axis: $x_1 < x_2 < \cdots < x_n$ (you can take $x_1 = 0$). You do not know the coordinates of these n points, but you know all the $\binom{n}{2}$ pairwise distances between them, i.e., you know $|x_i - x_j|$, for all $1 \leqslant i < j \leqslant n$. You also know which of these pairwise distances have at least one of the endpoints as either x_1 or x_n, i.e., you know the distances of the form $|x_1 - x_j|$ and $|x_n - x_j|$, for all j. But you do not know which one of the two (whether the distance is from x_1 or x_n). Give an algorithm to output the coordinates of n points *that are consistent* with all the given pairwise inter-distances. (Hint: First determine all the "complementary" pairs of distances, i.e., pairs of distances (x_i, y_i) such that $x_i + y_i = |x_1 - x_n|$. Each such pair corresponds to a point. Then use these to determine the points in a greedy way.)

Exercise 7.7. The bin packing problem is as follows: we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$, and an unlimited supply of **unit-sized** bins. We wish to pack all the objects into the **minimum** number of bins. Each bin can hold any subset of the objects as long as the total size of the objects **does not exceed 1**.

(a) Consider the following greedy algorithm for assigning objects to bins: Assume you have already placed objects $1, 2, \ldots, k-1$ into bins 1, 2, …, i. To place object k, consider the bins with increasing indices and place object k into the first bin that can accommodate it. If none of the i bins can, place object k into a new bin. Show that this greedy algorithm does not always return the optimal solution to the bin packing problem. Note that you do not have to give an implementation of the algorithm.

(b) Consider another greedy algorithm for assigning objects to bins: Assume you have already placed objects $1, 2, \ldots, k-1$. Place object k in a **used** bin that has the **maximum** amount of **available** space. If no such bin exists, put the object in an unused (new) bin.

1. Give an example showing that the above greedy algorithm does not always generate the optimal solution (i.e., it may not necessarily pack all the objects in the smallest number of bins).
2. Describe how to efficiently implement the above algorithm using a heap data structure. Analyze the running time of your algorithm. (For your analysis, you can assume an implementation of the heap operations with the appropriate run time.)

Exercise 7.8. There are n programs that are to be stored in some order on a computer tape of length ℓ. Associated with each program i is a length $\ell(i)$, $1 \leqslant i \leqslant n$. Assume that all programs can be stored in the tape (i.e., $\sum_{i=1}^{n} \ell(i) \leqslant \ell$). To retrieve a particular program from the tape the time needed is proportional to the distance from the beginning of the tape to the position where the program begins plus the length of the program. That is, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leqslant k \leqslant j} \ell(i_k)$.

Assume that all programs are retrieved equally often and that we want to reduce the *mean retrieval time* (MRT)

$$\frac{1}{n}\left(\sum_{i=1}^{n}\sum_{j=1}^{i} t_j\right) = \frac{1}{n}\left(\underbrace{t_1}_{1} + \underbrace{t_1 + t_2}_{2} + \underbrace{t_1 + t_2 + t_3}_{3} + \cdots + \underbrace{t_1 + t_2 + \cdots + t_n}_{n}\right)$$

Our **goal** is to find a **permutation** for the n programs so that when they are stored on the tape in this order the MRT is **minimized**. Note that minimizing MRT is equivalent to minimizing $d(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} \ell(i_k)$.

(a) Consider the following example where we have 3 programs and $(l_1, l_2, l_3) = (5, 10, 3)$. What is the (optimal) ordering of these 3 programs that will minimize the MRT and what is the optimal value of MRT?
(b) Give a greedy algorithm to find the optimal order assuming that we have n programs and length of program i is ℓ_i, $1 \leqslant i \leqslant n$. What is the running time of your algorithm?
(c) Prove that your greedy algorithm always gives the optimal ordering.

Exercise 7.9. Show that in the set cover problem, $|\mathrm{OPT}| \geqslant n/\Delta$.

Exercise 7.10. Consider the weighted version of the set cover problem defined as follows. Given a universe U of n elements and a collection of subsets of U, $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$, find a **minimum** cost subcollection of $\mathcal{S}$ that covers all elements of U. (Assume that every member of U is contained in some subset.)

Note that this is generalization of the problem we studied in Section 7.4, where we assumed that all subsets in the collection had weight 1.

Consider the following greedy algorithm which can be considered as a generalization of Algorithm 32.

```
1. $C = \phi$
2. while $C \neq U$ do
        Pick a set $S$ such that $c(S)/|S - C|$ is minimized
        $C = C \cup S$
3. Output the picked sets.
```

Show that the above algorithm is a H_n-factor approximation algorithm, where $H_n = \sum_{i=1}^{n} 1/i = \Theta(\ln n)$.

(Hint: Distribute the cost of a set picked in an iteration among the *new* elements it covers. During an iteration, define the *cost-effectiveness* of a set S to be the average cost at which it covers new elements: $c(S)/|S - C|$. The greedy algorithm picks the most cost-effective set in the current iteration. Define the *price* of an element to be the average cost at which it is covered. Analyze the prices of elements in the order in which greedy picks them.)

Exercise 7.11. Give an efficient implementation of the greedy set cover algorithm. Assume that the number of elements in the ground set is n, the total number of subsets S in the collection is s, and each subset has at most Δ elements.

Exercise 7.12. Suppose you are the "algorithmatician" of your company and the manager comes to you with the following problem. The company has to buy n different software products. Due to various constraints, the company can only buy at most one software product per month. Each software is currently selling for a price of \$100. However, they become expensive each month according to the following formula: the cost of software j increases by a factor $r_j > 1$ each month, where r_j is a known parameter. This means that if software j is purchased t months from now, it will cost $100(r_j)^t$. It is given that all price growth rates are distinct: that is, $r_i \neq r_j$ for software $i \neq j$ (even though at the start they all have the same price of \$100).

The question the manager poses for you is this: Given that the company can only buy at most one software product a month, in which order should it buy the products so that the *total amount of money spent is as small as possible*? In particular he has the following questions:

(i) Consider the following example to get started. Suppose $n = 3$ and that $r_1 = 2$, $r_2 = 3$, and $r_3 = 4$. What is the best order to buy the three products and why?

(ii) Give a greedy algorithm to find the optimal order assuming that we have n products to buy and the growth rate of product i is r_i, $1 \leqslant i \leqslant n$. What is the time complexity of your algorithm?

(iii) Prove that your greedy algorithm always gives the optimal ordering. (Hint: Use the exchange argument.)

Exercise 7.13. A shop sells n items whose respective costs are $c_1, c_2, \ldots, c_n$. A group of k friends want to buy all the n items. The shop has the following policy. If a person buys an item, say item i, for the first time, he/she gets the original price of that item, i.e., c_i. If the same person buys another item, say item j, for the second time, then the price is $2c_j$. So in general, if a person buys an item for the tth time, then the price to be paid is t times the (original) cost of that item.

The goal of the k friends is to devise a strategy to buy all the n items as cheap as possible.

1. Let there be 7 items and let the costs of the items be the respective digits in your ID. For example, if your ID is 1348919, then the costs of the 7 items are respectively, 1,3,4,8,9, 1, and 9. Assume that $k = 3$. What is the best possible way for the 3 friends to buy all the items? What is the minimum cost needed?

2. Give a greedy algorithm to solve the problem. Explain your algorithm.

3. Show how to implement your greedy algorithm and analyze its running time.

4. Prove that your greedy algorithm always outputs the optimal solution.

Exercise 7.14. Consider the following optimization problem arising in various applications and stated here in terms of a coin changing problem. We have access to an unlimited supply of coins with k distinct denominations, $1 = c_1 < c_2 < \cdots < c_k$, each being a positive integer. We are given an integer quantity $T \geqslant 1$ and are to determine the minimum number of coins that make up T. Since $c_1 = 1$, a solution always exists. We point out that for the denominations 1, 5, 10, 25, a greedy solution can be shown to always generate the optimal solution. For other denominations, greedy may not produce the optimal solution.

1. Show that the optimal substructure property holds for the coin changing problem.

2. Show how to determine the minimum number of coins making up T by computing the entries $C(i)$, $1 \leqslant i \leqslant T$, where $C(i)$ is the minimum number of coins making up quantity i. Give the recursive formulation for computing $C(i)$ and give the pseudo-code of the resulting algorithm.

3. Analyze the running time of your algorithm.

4. Describe how to change the algorithm so that it also generates the optimum solution (i.e., the number of coins of each type to be used).

8 RANDOMIZED ALGORITHMS AND PROBABILISTIC ANALYSIS*

"It is remarkable that this science, which originated in the consideration of games and chances, should have become the most important object of human knowledge... The most important questions of life are, for the most part, really only problems of probability."

– Pierre-Simon, Marquis de Laplace (1749–1827)

"If somebody would ask me, what in the last 10 years, what was the most important change in the study of algorithms I would have to say that people getting really familiar with randomized algorithms had to be the winner."

– Donald Knuth, *Randomization and Religion, 1999*

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of it operation is at least 2^{-100}*".*

– Christos Papadimitriou, *Computational Complexity*

The influence of probability theory in algorithm design and analysis has been profound in the last two decades or so. This chapter will focus on the interplay between algorithms and probability, in particular, randomized algorithms and probabilistic analysis of algorithms. Randomized algorithms are typically faster, simpler, and easier to implement, compared to purely deterministic (i.e., algorithms that do not involve any randomness) algorithms. (We will see many examples of this throughout the course in many domains.) This chapter will introduce some basic probabilistic tools and techniques that help us in the design of randomized algorithms and their analysis. We will see throughout the course that randomization and probabilistic analysis are immensely used in various algorithmic areas ranging from fundamental algorithms and data structures (such as sorting, fingerprinting, hashing), to network/graph algorithms, communication networks, distributed/parallel computing, data compression, reliable communication, approximation algorithms etc.

This chapter will use basic concepts in probability. It will be useful to refer to the Appendix C if needed.

8.1 Randomization

Randomization in algorithm design and analysis comes in two flavors:

- *Randomized algorithms:* These are algorithms that perform random steps, i.e., one or more steps of the algorithm will involve random coin tosses. Depending on the outcome of the toss (whether HEADS or TAILS, each of which occurs with equal probability), the algorithm will act differently. Randomized algorithms are typically faster and simpler to implement than their deterministic counterparts. However, the correctness of the algorithm or its performance or both will come with probabilistic guarantees. Note that in a randomized algorithm, usually, the input to the algorithm is arbitrary, i.e., no probability distribution on the inputs is assumed.

- *Probabilistic analysis of algorithms:* In probabilistic analysis, performance of the algorithm (which is typically deterministic, but can be randomized as well) is analyzed assuming that the input to the algorithm is chosen according to some probability distribution on the input domain. Probabilistic analysis is useful in analyzing the performance of an algorithm on a "typical" input; often this is called as *average case analysis*. Probabilistic analysis is very useful in situations where the worst-case performance is much worse compared to the average-case performance. This can explain why certain problems are easier to solve in practice than their theoretical worst-case guarantees. We will see such examples throughout the course.

8.2 Introducing a Randomized Algorithm

Problem 8.1 ▶ Finding a repeated element

Consider an array of n numbers (assume n is even) that has $n/2$ distinct elements and $n/2$ copies of another element. We want an algorithm to identify the repeated element.

We note that any deterministic algorithm for the above will need at least $n/2+2$ steps in the worst case. Consider an adversary who has full knowledge of the algorithm. The adversary can make sure that the first $n/2+1$ elements examined by the algorithm are all distinct.

A simple randomized algorithm

Now, consider the following randomized algorithm:

- Randomly pick two array elements and check whether they come from different cells and have the same value.

- If they do, output **true** else output **false**.

There are generally two questions that one can ask regarding a randomized algorithm: (1) Does the algorithm always give a correct answer? If not, what is the probability that the answer output is correct? (2) What is the running time of the algorithm? Usually the running time is a *random variable*, and hence one is interested in the average running time of the algorithm.

In the case of the repeated element problem, the questions are: (1) What is the probability that the algorithm will find the repeated element? (2) What is the running time of the algorithm?

Analysis of the randomized algorithm

There are n^2 possible combinations of picking the two elements. Out of these, $\frac{n}{2}\left(\frac{n}{2}-1\right)$ combinations will result in success (why?). Since every combination is equally likely, the probability that the algorithm will succeed is $\frac{n/2(n/2-1)}{n^2} = \frac{1}{2}(\frac{1}{2}-\frac{1}{n}) = \frac{1}{4} - \frac{1}{2n} \geqslant \frac{1}{5}$ for $n \geqslant 10$. Thus the probability that the algorithm fails is at most $4/5$.

Reducing the failure probability

How to reduce the failure probability? Repeat the algorithm again, if we do not succeed. Each time we make **independent** random choices. If we repeat the algorithm 100 times, the probability of failure is $(4/5)^{100} < 10^{-9}$. If we repeat the algorithm $c \log n$ times the probability that the algorithm fails is $< (4/5)^{c \log n} = \frac{1}{n^{c \log(5/4)}} = 1/n$ if $c = 1/\log(5/4)$. Thus with $\mathcal{O}(\log n)$ repetitions, the algorithm succeeds **with high probability (whp)**, i.e., with probability at least $1 - 1/n^c$, where $c > 0$ is some constant. Thus the algorithm succeeds in $\mathcal{O}(\log n)$ repetitions whp.

Note that the notion of "with high probability" is used often in the analysis of randomized algorithms, where the "high probability" is with respect to n, the size of the input.

Definition 8.1

In probabilistic analysis, "with high probability (whp)" (with respect to a parameter n), means with probability at least $1 - 1/n^c$, where $c > 0$ is some fixed constant.

Analyzing the running time

We augment the standard RAM computation model (Section 2.4) with a new operation: **Choose a random number uniformly from the set** $\{a_1, a_2, \ldots, a_k\}$. We assume that this operation takes 1 step (or, constant time). This is reasonable to assume if the size of the set, i.e., k, is not too large. In any case, one can show that if tossing a fair coin takes constant time, then choosing a random element from a set of size k can be done in $\mathcal{O}(\log k)$ time (Exercise 8.1).

By our model, each random choice of picking an element randomly can be implemented in 1 step, thus the algorithm (one repetition) takes only constant time. Thus, in $\mathcal{O}(\log n)$ time, we can find the repeated element whp.

8.3 Problem: Verifying Matrix Multiplication

We give a simple randomized algorithm that checks the correctness of multiplying two matrices. Remarkably, using randomization, we show that verification can be done in $\Theta(n^2)$ time, which is significantly faster than the best known algorithms for multiplying two matrices. Strassen's algorithm in Chapter 5 was one such algorithm, but faster algorithms are known, but these algorithms take time asymptotically larger than n^2, i.e., they take $\Omega(n^{2+\epsilon})$ for some $\epsilon > 0$. No deterministic algorithms that match the $\Theta(n^2)$ bound are known. We will see later in Chapter 9 that randomization is also very useful (faster) in verifying other operations such as comparing strings.

Problem 8.2 ▶ Matrix Multiplication Verification

Given three $n \times n$ integer matrices A, B, and C, verify whether $AB = C$.

One can accomplish this by just multiplying A and B and checking whether the product matches C element by element. Simple matrix multiplication takes $\Theta(n^3)$ time, while the best known algorithm takes at least $\Omega(n^{2.37})$ time.

8.3.1 A Randomized Algorithm

The randomized algorithm is extremely simple and is as follows.

1. Choose a random column vector $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)^T \in \{0, 1\}^n$, i.e., each entry is chosen with probability 1/2.
2. Compute $A(B\boldsymbol{r})$ and $C\boldsymbol{r}$.
3. If $A(B\boldsymbol{r}) \neq C\boldsymbol{r}$, then output $AB \neq C$; Else output $AB = C$.

The above algorithm takes $\Theta(n^2)$ time because it just involves multiplying a matrix with a vector.

8.3.2 Bounding the Error Probability

Clearly, the algorithm can output a wrong answer, since the algorithm checks using only one vector, e.g., if the vector chosen is the all-zeros vector ($(0, 0, \ldots, 0)$), then the check is true, but AB may not be equal to C. We now bound the probability that a bad vector choice is made.

Theorem 8.1

If $AB \neq C$, and $\boldsymbol{r}$ is chosen uniformly at random from $\{0, 1\}$, then

$$\mathbb{P}(AB\boldsymbol{r} = C\boldsymbol{r}) \leqslant 1/2$$

Proof. Let $D = AB - C \neq 0$. Hence, D has some non-zero entry. Without loss of generality, let it be d_{11}. $AB\boldsymbol{r} = C\boldsymbol{r}$ implies that $(AB - C)\boldsymbol{r} = 0$, i.e., $D\boldsymbol{r} = 0$. In particular, this means that we should have $\sum_{j=1}^{n} d_{1j} r_j = 0$, i.e.,

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$$

What is the probability of the above equality to hold? We show that the probability of the above happening is at most 1/2.

It appears difficult to compute the probability of the above event hapenning under the assumption that all the r_is are chosen independently and uniformly at random, *all at once*. We instead use the *principle of deferred decisions* (See Appendix). Instead of choosing the $r_1, r_2, \ldots, r_n$ all at once, we set $r_n, \ldots, r_2, r_1$ one by one in that order. Once we have set all values from r_n to r_2, the right hand of the above equality is fixed (it is some particular value). Now r_1, the left hand side, can take two values, depending on the random choice. The equality holds in at most one choice of r_1. Hence the probability is at most 1/2. (It can be zero if the right hand side value is not 0 or 1.) □

We can get a high probability algorithm, by repeating the above algorithm a few times, say $\mathcal{O}(\log n)$ times. It is easy to show (as we did in the case of the "Repeated element") that the error probability is reduced to $1/n$ or less.

8.4 Randomized Quicksort

In Section 5.1.3, we presented QuickSort. This was a deterministic algorithm (one which had no random choices) and its worst case performance can be $\Theta(n^2)$. Note that this worst-case performance can happen if the input is sorted (or reverse-sorted) or nearly sorted. Is it possible to make Quicksort perform well on all inputs? That is, can we make Quicksort run in $\mathcal{O}(n \log n)$ time (i.e., comparisons) for *all* inputs? Enter *Randomization.* By making use of randomization, we can modify QuickSort to perform well on all inputs (with high probability). More precisely, we will first show that the "average case" or "expected" performance of QuickSort is $\mathcal{O}(n \log n)$. Then we will show the same bound holds not just on average, but also with high probability, i.e., with probability at least $1 - 1/n$.

Main idea and intuition of the algorithm.

The pseudocode of randomized Quicksort is given in Algorithm 35. The only change, compared to the (deterministic) QuickSort, is that the pivot is chosen *randomly* in Step 4 of the algorithm (as opposed to choosing the first element). Thus, the pivot is equally likely to be any one of the elements in S. The main intuition as to why this is better than the deterministic algorithm is that regardless of the input, the algorithm has a good probability of choosing a pivot whose rank is in the "middle". More precisely, the probability that the pivot belongs to the middle one-third of the elements when considered in sorted order (i.e., the elements whose ranks are in the interval $[|S|/3, 2|S|/3])$ — see Figure Figure 8.1 — is 1/3. Why is it good to choose the pivot among the middle one-third of the elements? This is because, in this case, the set S is partitioned in an approximately balanced fashion, i.e., $\max\{|S_1|, |S_2|\} \leqslant \frac{2}{3}|S|$. If the sets are partitioned in this fashion, in every level of the recursion tree, then it is easy to see that the depth of the recurrence tree is $\mathcal{O}(\log n)$; since $\mathcal{O}(n)$ comparisons are done in every level of the recursion tree, the total cost is $\mathcal{O}(n \log n)$. However, we should note that in randomized Quicksort, this kind of balanced partitioning does not happen in every level; however, it happens with enough frequency, which we will show gives us the same $\mathcal{O}(n \log n)$ behaviour.

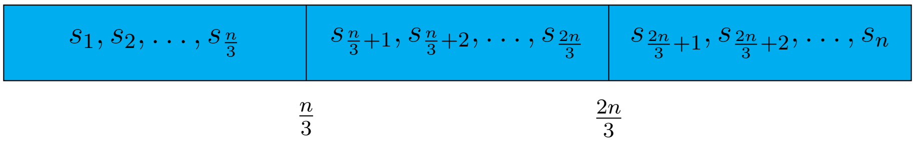

$s_1, s_2, \ldots, s_{\frac{n}{3}}$
$s_{\frac{n}{3}+1}, s_{\frac{n}{3}+2}, \ldots, s_{\frac{2n}{3}}$
$s_{\frac{2n}{3}+1}, s_{\frac{2n}{3}+2}, \ldots, s_n$
$\frac{n}{3}$
$\frac{2n}{3}$

Figure 8.1: Elements are listed in sorted order. The probability that a randomly chosen pivot lands in the middle one-third of the set is 1/3.

Algorithm 35 RandQuickSort
Input: An array S
Output: S in sorted order

```
func RandQuickSort($S$):
    if $|S| \leqslant 1$:
        return $S$
    else:
        $p$ = RandomChoice($S$)
        ▷ $p$ is chosen uniformly at random from $S$
        $S_1 = \{x \in S - \{p\} \mid x < p\}$
        $S_2 = \{x \in S - \{p\} \mid x > p\}$
        ▷ Elements in $S_1$ and $S_2$ are in the same order as in $S$
        return RandQuickSort($S_1$) + $[p]$ + RandQuickSort($S_2$)
```

Analysis of randomized Quicksort

Since Quicksort is a randomized algorithm, the number of comparisons performed by the algorithm is a *random variable.* This random variable takes values in the range $[cn \log n, c'n^2]$, where c, c' are some constants (why?). Thus the parameter of most interest is the *expected* value of this random variable, which is the most basic statistic associated with the random variable (and thus characterizes it by a single value). We will show that the expected number of comparisons is $\mathcal{O}(n \log n)$. Later, we will show that this bound holds also with high probability.

Theorem 8.2

Let the random variable T denote number of comparisons in a run of QuickSort. Then,

$$\mathbb{E}[T] = \mathcal{O}(n \log n)$$

Proof. As is typical in probabilistic analysis, we will write T as a sum of simpler, i.e., 0-1 random variables; then the expectation of T can be computed using linearity of expectation.

Let $s_1, \ldots, s_n$ be the elements of S in sorted order. For $i = 1, \ldots, n$, and $j > i$, define 0-1 (or indicator) random variable $X_{i,j}$, such that $X_{i,j} = 1$ iff s_i is compared to s_j in the run of the algorithm. Note that X_{ij} "indicates" occurrence of the event denoting the comparison of elements s_i and s_j.

The number of comparisons in running the algorithm is

$$T = \sum_{i=1}^{n} \sum_{j>i} X_{i,j}$$

We are interested in $\mathbb{E}[T]$. Using linearity of expectation, we write

$$\mathbb{E}[T] = \mathbb{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{i,j}\right] = \sum_{i=1}^{n}\sum_{j>i}\mathbb{E}[X_{i,j}] = \sum_{i=1}^{n}\sum_{j>i}\mathbb{P}(X_{ij} = 1)$$

What is the probability that $X_{i,j} = 1$? To evaluate this, we use the following crucial argument. s_i is compared to s_j if and only if either s_i or s_j is chosen as the pivot *before any* of the $j - i - 1$ elements between s_i and s_j are chosen as pivots. First we note, that the choosing elements s_k, where $k < i$ or $k > j$ will not have any influence of whether s_i and s_j are compared or not. On the other hand, choosing an element s_k, where $i < k < j$, will result in s_i and s_j going to *different* sets in the next level of the recursion and hence will not be compared again.

A pivot is chosen uniformly at random from the elements in the set $\{s_i, s_{i+1}, \ldots, s_j\}$. Thus,

$$\mathbb{P}(X_{i,j} = 1) = \frac{2}{j - i + 1}$$

Hence,

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&= \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k+1} \\
&\leqslant \sum_{i=1}^{n}\sum_{k=1}^{n}\frac{2}{k} \\
&= 2\sum_{k=1}^{n}\sum_{i=1}^{n}\frac{1}{k} \\
&= 2\sum_{k=1}^{n}\frac{1}{k}\sum_{i=1}^{n}1 \\
&\leqslant 2n\sum_{k=1}^{n}\frac{1}{k} \\
&= 2nH_n \\
&= 2n\log n + \mathcal{O}(n)
\end{aligned}$$

In the above, we used the fact that $\sum_{k=1}^{n} 1/k = H_n$, where H_n is called the Harmonic sum and it can be shown that $H_n = \Theta(\log n)$. □

8.4.1 Probabilistic Analysis of Deterministic Quicksort

There is an entirely different way to probabilistically analyze Quicksort. Consider the deterministic Quicksort (where the pivot is always the first element), but assume a probability distribution on the *input*. Note that there are $n!$ different possible inputs to Quicksort (for a input size n) — corresponding to $n!$ different orderings of the input set (note that, since we only do comparisons, only the ordering matters). Some permutations are "bad" for Quicksort, i.e., they can result in $\Theta(n^2)$ comparisons (e.g., sorted order). On the other hand, there are clearly "good" permutations, i.e., they result in $\mathcal{O}(n\log n)$ comparisons. This raises a basic question: What fraction of the $n!$ permutations are good? It will turn out that except for a tiny fraction, the vast majority of permutations are good. In particular, we will show that if one chooses a random permutation (i.e., a permutation that is equally likely to be any one of the $n!$ permutations), then the expected performance of deterministic Quicksort is $\mathcal{O}(n \log n)$. (It is crucial to note that the expectation is with respect to the random choice of the input and not due to the randomness in the algorithm.) This analysis sheds some light on why Quicksort does very well in practice, despite its worst-case performance: on a “typical” input (in fact for most inputs), Quicksort’s performance is similar to the best-case, i.e., $\mathcal{O}(n \log n)$.

Theorem 8.3

The expected run time of (deterministic) Quicksort on a random input, uniformly chosen from all possible permutations of S is $\mathcal{O}(n \log n)$.

Proof. The proof is almost identical to that of Theorem 8.2. Set $X_{i,j}$ as before. If all permutations have equal probability, all permutations of $s_i, \ldots, s_j$ have equal probability, thus

$$\mathbb{P}(X_{i,j}) = \frac{2}{j-i+1}.$$

Thus, as before, we obtain:

$$\mathbb{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{i,j}\right] = \mathcal{O}(n \log n).$$

□

8.5 Randomized Algorithms vs. Probabilistic Analysis

We compare the two flavors below:

Randomized Algorithms:

- Analysis is true for **any** input.
- The sample space is the space of random choices made by the algorithm.
- Repeated runs are independent.

Probabilistic Analysis:

- The sample space is the space of all possible inputs.
- If the algorithm is **deterministic** – repeated runs on the same input yield the same output.

8.5.1 Randomized Algorithm classification

A **Monte Carlo Algorithm** is a randomized algorithm that may produce an incorrect solution. Thus, in a Monte-Carlo algorithm, one has to bound the probability of error. The algorithm for finding the repeated element (Section 8.2) is a Monte-Carlo randomized algorithm, since the output can be incorrect. The same is the case with the randomized algorithm for verifying the product of two matrices. A **Las Vegas** algorithm is a randomized algorithm that **always** produces the correct output. Randomized Quicksort is a Las Vegas randomized algorithm, since the output is always sorted. In both types of algorithms, the run-time is a random variable.

8.6 Randomized Selection

Just like Randomized Quicksort is the randomized counterpart for the (deterministic) Quicksort algorithm, randomized selection (Algorithm 36 gives the pseudocode) is the randomized counterpart of the linear time selection algorithm of Section 8.6. The randomized variant is surprisingly simple and similar to that of randomized Quicksort: the pivot is chosen randomly among the set of elements. Note that the randomized selection algorithm is significantly simpler compared to the deterministic linear time selection algorithm presented in Section 5.2. We will show that this randomized algorithm, though its worst case performance can be $\Theta(n^2)$, its expected performance is $\mathcal{O}(n)$.

Algorithm 36 RandomizedQuickSelect
Input: An array S and the desired order statistic, k
Output: The value of the kth order statistic

```
func RandomizedQuickSelect($S$, $k$):
    if $|S| \leqslant 1$:
        return $S[1]$
    else:
        $p = \text{RandomChoice}(S)$    ▷ $p$ is chosen uniformly at random from $S$
        $S_1 = \{x \in S - \{p\} \mid x < p\}$
        $S_2 = \{x \in S - \{p\} \mid x > p\}$
        if $|S_1| == k - 1$:
            return $p$
        else if $|S_1| > k$:
            return $\text{RandomizedQuickSelect}(S_1, k)$
        else:
            return $\text{RandomizedQuickSelect}(S_2, k - |S_1| - 1)$
```

Correctness and Worst-case runtime

Theorem 8.4

The algorithm RandomizedQuickSelect returns a singleton with the correct value.

The above proof is left as an exercise.

Run time

Theorem 8.5

The worst-case run-time of algorithm RandomizedQuickSelect is $\Theta(n^2)$.

The above proof is left as an exercise (Exercise 8.3).

Expected run-time analysis

As explained in RandQuickSort, intuitively, choosing a random pivot and partitioning around it creates an *approximately balanced* partition (see Figure 8.1). There is a constant probability of creating a partition wherein either of the two subarrays has *at most*, say, $\frac{2}{3}$ of the elements. The probability of this occurring is exactly the probability that a pivot is chosen in the *middle one-third* of the elements (considered in sorted order).

Theorem 8.6

The expected run-time of RandomizedQuickSelect is $\mathcal{O}(n)$, where n is the size of the input set.

Let random variable $T(n)$ be the run time of the algorithm on an input size of n elements.

Consider the elements of the set in *sorted* order (just for analysis sake). If the pivot is one of the elements belonging to the middle one-third, then both the partitions will have size at most $2n/3$. (Why?) The probability that this happens is $= \frac{n/3}{n} = 1/3$.

We can write a recurrence relation for $T(n)$:[1]

$$T(n) \leqslant (1/3)T(2n/3) + (2/3)T(n) + n$$

Explanation for the above recurrence:

- With probability *1/3* the pivot is chosen in the middle part: The set is partitioned into two parts. In this case, one is of size at most $2n/3$ and the other is of size at least $n/3$. Since $T(n)$ is increasing function of n (why?), we will assume the worst-case situation where the kth smallest element (the desired element) is in the *larger* of the two partitions which can be of size at most $2n/3$.

- With probability *2/3* the pivot is chosen in the first or the third part: We assume there is no reduction in the size of the subproblem (again the worst situation).

We take expectations on both sides of the above recurrence:

$$\mathbb{E}[T(n)] \leqslant \mathbb{E}[(1/3)T(2n/3) + (2/3)T(n) + n]$$

By linearity of expectation, we have:

$$\mathbb{E}[T(n)] \leqslant (1/3)\,\mathbb{E}[T(2n/3)] + (2/3)\,\mathbb{E}[T(n)] + n$$

[1] Strictly speaking, this inequality is a stochastic dominance inequality, since it relates random variables. Note that, unlike the deterministic setting, $T(n)$ is not a value, but rather a random variable. Hence, as such one cannot write inequalities (or equalities) with respect to random variables. In this case, the interpretation is that the random variable on the left is *stochastically dominated* by the random variable on the right. In other words, if X and Y are two random variables, and we write $X \leqslant Y$, we mean that for any value a, $\mathbb{P}(X \geqslant a) \leqslant \mathbb{P}(Y \geqslant a)$.

Simplifying the above recurrence, we have:

$$\mathbb{E}[T(n)] \leqslant \mathbb{E}[T(2n/3)] + 3n$$

We can use the DC recurrence theorem (or induction) to show that the solution to the above recurrence is: $\mathbb{E}[T(n)] = \mathcal{O}(n)$.

8.7 Problem: Maximum Cut

The maximum cut problem is a graph optimization problem defined as follows.

Problem 8.3 ▶ Max Cut

Given a (undirected) graph $G = (V, E)$ find a cut of S of V such that the number of edges crossing the cut is maximized.

The max cut problem is known to be *NP-hard*, i.e., there exists no known polynomial time algorithm (see Section **??**). However, one can come up with approximation algorithms that yield a cut such that the number of edges crossing the cut is a *constant* factor of the optimum. In other words, one can design a *constant factor* approximation algorithm for the problem. Note that the max cut problem is very different in complexity compared to its *minimization* counterpart — the *minimum cut* problem which we will study in Chapter 13. In the minimum cut problem the goal is to find a cut such that the number of edges crossing the cut is minimized. This problem has known *polynomial* time algorithms. Both these problems are well-studied graph optimization problems with several applications.

We now present a very simple randomized approximation algorithm that runs in polynomial time and that gives an *expected* approximation ratio of 1/2, i.e., the ratio of the solution given by the approximation algorithm has at least half the number of edges (in expectation) of the best possible solution. We note that the approximation factor guarantee is in terms of expectation, but we will show later that one can convert this algorithm into a Las Vegas algorithm, which always finds a cut with the approximation factor guarantee of 1/2, while still running in polynomial time with high probability.

Another important consequence of the randomized approximation algorithm is that it can be *derandomized*! That is, the algorithm can be made deterministic by eliminating the randomness to construct a *deterministic* algorithm that has the same approximation factor guarantee of 1/2. In fact, the deterministic algorithm turns out to be pretty simple. This is accomplished by a technique known as "method of conditional probabilities." This is a powerful method that is useful in derandomizing some randomized algorithms. In general, there have been several problems where directly designing efficient deterministic algorithms has not been easy. Instead, we first design a randomized algorithm and then derandomize it.

8.7.1 A Simple Randomized Algorithm

A simple randomized algorithm is as follows. Start with two empty sets A and B which will eventually be the cut. For each node v, independently, with probability 1/2, add it to A and, with probability 1/2, add it to B. Output the cut (A, B).

We show below that the number of edges crossing this cut is $m/2$, where m is the number of edges in the graph. Since the number of edges in the optimum cut is at most m, this means that the approximation factor given by this cut is at least 1/2.

Theorem 8.7

Given any undirected graph $G = (V, E)$ with n vertices and m edges, the above algorithm outputs a cut (A, B) such that the expected number of edges crossing the cut is at least $m/2$ edges.

Proof. Let X be the random variable that counts the total number of edges crossing the cut. We will compute $\mathbb{E}[X]$.

Computing $\mathbb{E}[X]$ directly is difficult. Instead we will define m indicator random variables X_1, X_2, ..., X_m, where

$$X_i = \begin{cases} 1 & \text{if edge } i \text{ crosses the cut} \\ 0 & \text{otherwise} \end{cases}$$

Then $X = \sum_{i=1}^{m} X_i$.

Note that $X_i = 1$ if one endpoint of the edge lands in A and the other lands in B. Let the endpoints of the ith edge be u and v. Then the probability of this happening is

$$\mathbb{P}(u \in A)\,\mathbb{P}(v \in B) + \mathbb{P}(u \in B)\,\mathbb{P}(v \in A) = \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{1}{2}.$$

Hence $\mathbb{P}(X_i = 1) = 1/2$.

Since $X = \sum_{i=1}^{m} X_i$, we have, by linearity of expectation

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} \mathbb{E}[X_i] = \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2}.$$

Thus, the expected number of edges crossing the cut is $m/2$. □

8.7.2 A High Probability Algorithm

Note that the above randomized algorithm has the approximation guarantee of $1/2$ only in expectation and not with high probability.

We desire a Monte Carlo algorithm that outputs a cut with at least $m/2$ edges crossing the cut, with high probability.

To accomplish this, we lower bound the probability that the randomized algorithm outputs a cut that has at least the expected number of edges, i.e., $m/2$.

Theorem 8.8

Let X denote the cut size constructed by the randomized algorithm. Then $p = \mathbb{P}(X \geqslant m/2) \geqslant \frac{1}{m/2+1}$.

Proof. By Theorem 8.7, $\mathbb{E}[X] = m/2$. Let $p = \mathbb{P}(X \geqslant m/2)$ denote the probability that X takes at least its expected value. To lower bound p, we simply write $\mathbb{E}[X]$ using the definition of expectation as follows.

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{k=0}^{m} k\,\mathbb{P}(X=k) \\
&= \sum_{k<\frac{m}{2}} k\,\mathbb{P}(X=k) + \sum_{k\geqslant\frac{m}{2}} k\,\mathbb{P}(X=k) \\
&\leqslant \sum_{k<\frac{m}{2}} \left(\frac{m}{2}-1\right)\mathbb{P}(X=k) + \sum_{k\geqslant\frac{m}{2}} m\,\mathbb{P}(X=k) \\
&= \left(\frac{m}{2}-1\right)\sum_{k<\frac{m}{2}} \mathbb{P}(X=k) + m\sum_{k\geqslant\frac{m}{2}} \mathbb{P}(X=k) \\
&= \left(\frac{m}{2}-1\right)\mathbb{P}\left(X\leqslant\frac{m}{2}-1\right) + m\,\mathbb{P}\left(X\geqslant\frac{m}{2}\right) \\
&\leqslant \left(\frac{m}{2}-1\right)(1-p) + mp
\end{aligned}$$

which yields

$$p \geqslant \frac{1}{m/2+1}$$ □

We note that by the above theorem, the probability that the algorithm outputs a cut of desired value (i.e., at least $m/2$ edges crossing the cut) is $\Theta(1/m)$, where m is the number of edges in the graph. Although this probability is small, it can be boosted up to high probability by repeating the algorithm $(m/2+1)\ln n$ times. This is similar to what was done for the Matrix Multiplication Verification problem of verifying matrix multiplication (Section 8.3).

The probability of not outputting a desired cut is at most $1-\frac{1}{m/2+1}$. If we repeat the algorithm $(m/2+1)\ln n$ times, the probability of failing in each of those times is at most

$$\left(1-\frac{1}{m/2+1}\right)^{(m/2+1)\ln n} \leqslant e^{\frac{-(m/2+1)\ln n}{m/2+1}} = e^{-\ln n} = 1/n$$

Hence the probability of success is at least $1-1/n$.

One can also get a Las Vegas algorithm that always outputs a cut of size at least $m/2$. We repeatedly run the algorithm until it outputs a cut of size at least $m/2$; this can be easily checked by counting the number of edges crossing the cut which takes $\mathcal{O}(m)$ time. The expected number of times the algorithm has to be run is at most $O(m)$, since the probability of "success" is at least $\Omega(1/m)$. We have used the geometric distribution (See Appendix C) to conclude that the expected number of times one has to repeat the algorithm is $O(m)$.

Note that the Monte Carlo algorithm still has polynomial run time (this will be in expectation, as in the case of Las Vegas). We leave it the reader to analyze and show that the algorithm takes $O((m+n)m\log n) = O((m^2+mn)\log n)$ time.

8.7.3 A Deterministic Algorithm via Derandomization

It turns out that the simple randomized algorithm can be turned into a simple deterministic algorithm that always outputs a cut with value at least $m/2$. This is accomplished by using a technique known as the "the method of conditional probabilities." We use this technique to *derandomize* the randomized algorithm and construct a deterministic algorithm. We describe this technique by applying it our problem.

Intuition behind the method

Let $C(A, B)$ be the cut generated by a randomized algorithm and let the random variable X denote the number of the edges crossing the cut.

Note that initially both A and B are empty. The randomized algorithm places each vertex, one by one, randomly in one of the two sets with equal probability. After placing all the vertices, the random variable X is fully determined. Hence we can view X as a function of the placings of each of the vertices which themselves can be viewed as random variables. Concretely, let random variable X_i determine the placing of the ith vertex for $1 \leqslant i \leqslant n$. As per our algorithm, X_i takes two values, A or B, with equal probability. Hence $X = f(X_1, X_2, \ldots, X_n)$, i.e., X is some function of the placings of all the vertices and is fully determined by setting values to $X_1, \ldots, X_n$.

Before we place any vertex, X is fully undetermined and as we showed earlier, $\mathbb{E}[X] = m/2$. One important consequence of the expected value is that a cut of size at least $m/2$ *must exist* in the graph![2] Our goal is to deterministically find such a cut.

After we place all the vertices, X is fully determined (there is no more "randomness"). Our goal is to determine values for $X_1, \ldots, X_n$ such that *after* we fix these values, the fully determined value of X is at least its expectation, i.e., $m/2$. This will yield a *deterministic method* of choosing the placings of all the vertices that guarantees a cut size of at least $m/2$.

Using the method of conditional probabilities, we will deterministically determine, for each vertex, which set it should go depending on the choices made by the previous placings. Suppose we place the vertices one by one deterministically in some order, say $v_1, \ldots v_n$ according to the following rule. For this, we consider, conditional expectation (Appendix C.10). We start with $\mathbb{E}[X] = m/2$ and condition on the random variables X_1, X_2, $\ldots$, X_n, taking values one by one. Let the value taken by X_k be x_k (A or B) after v_k is placed. We look at a sequence of values,

$$\begin{aligned}
&\mathbb{E}[X] = m/2\\
&\mathbb{E}[X \mid X_1 = x_1]\\
&\mathbb{E}[X \mid X_1 = x_1, X_2 = x_2]\\
&\qquad\vdots\\
&\mathbb{E}[X \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n] = X
\end{aligned}$$

Note that $\mathbb{E}[X \mid X_1 = x_1, X_2 = x_2 \ldots X_k = x_k]$ is the expected value of the random variable after placing the first k vertices. Its value can be computed based on the vertices already placed and the random choices made by the rest of the vertices (using linearity of expectation, as before — See Exercise 8.20). After all vertices have been placed, X is determined completely.

Assume, without loss of generality, that we place v_1 in set A. We assume inductively that we have already placed v_1, v_2, $\ldots$, v_{k-1}.

For $k \geqslant 2$, place v_k in the set x_k (either A or B) such that

$$\mathbb{E}[X \mid X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k] \geqslant \mathbb{E}[X \mid X_1 = x_1, X_2 = x_2, \ldots, X_k = x_{k-1}].$$

[2] This inference is known as the *probabilistic method*. More generally it says that if any random variable has an expected value a, then the random variable will (with non-zero probability) take a value at least equal to a. (Exercise 8.19 asks you to prove this.)

In other words, place v_k in the set such that the expected value given the choices already made for v_1, v_2, ..., v_k is at least the expected value given the choices made for v_1, v_2, ..., v_{k-1}.

Before we show that the above rule guarantees a cut of size at least $m/2$, we point out that it yields a deterministic algorithm. It is easy to compute $\mathbb{E}[X \mid x_1, \dots, x_k = A]$ and $\mathbb{E}[X \mid x_1, \dots, x_k = B]$ for any k since they depend only on the placement of the first k vertices and the rest of the vertices which are placed randomly. The latter can be computed by using linearity of expectation (Exercise 8.20 asks you to show this.)

Lemma 8.9

If we choose the placement of the vertices using the above rule, then we have $X = \mathbb{E}[X \mid X_1 = x_1, \dots, X_n = x_n] \geqslant \mathbb{E}[X] = m/2$. Thus, the final cut will have at least $m/2$ edges crossing the cut.

Proof. Suppose we place v_k with equal probability in A or B (as done by our randomized algorithm). Let x_k denote where it is placed. Then, we compute $\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}]$ by conditioning on the placement of v_k:

$$\mathbb{E}[X \mid X_1 = x_1, \dots, X_k = x_{k-1}] = \mathbb{E}[\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k]]$$

where the r.v. X_k denotes the outcome of the random choice of the placement of v_k. By the property of conditional expectation (Equation (C.1)) we have:

$$\begin{aligned}\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}] &= \mathbb{E}[\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k]]\\ &= \mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = A]\,\mathbb{P}(X_k = A)\\ &+ \mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = B]\,\mathbb{P}(X_k = B)\\ &= \frac{1}{2}\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = A]\\ &+ \frac{1}{2}\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = B]\end{aligned}$$

Thus,

$$\max(\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = A], \mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = B])$$

$$\geqslant \mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}].$$

Hence if we choose the set x_k which gives a larger conditional expectation, we have $\mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}, X_k = x_k] \geqslant \mathbb{E}[X \mid X_1 = x_1, \dots, X_{k-1} = x_{k-1}]$. Since this holds for every k, we have inductively: $X = \mathbb{E}[X \mid X_1 = x_1, \dots, X_n = x_n] \geqslant \mathbb{E}[X] = m/2$. □

The above rule suggests a simple greedy deterministic algorithm:

Exercise 8.20 asks you to show the correctness of the above algorithm and also determine its running time.

Algorithm 37 GreedyMaxCut
Input: A graph G with n nodes and m edges
Output: A cut of G of size at least $m/2$ edges crossing the cut.

```
func GreedyMaxCut($G$):
    $A$, $B$ = $\emptyset$, $\emptyset$
    for $v$ in $G$:
        $n_a$, $n_b$ are the number of neighbors of $v$ in $A$ and $B$, respectively
        if $n_a \leqslant n_b$:
            $A$.append($v$)
        else:
            $B$.append($v$)
    return $A$, $B$
```

8.8 High Probability Bounds: Concentration near Expectation

While expectation of a random variable is a very useful parameter, in many applications, it is important to understand how the random variable takes values *away* from its expected value. In particular, a key question is: What is the probability that the random variable takes a value that is far away from the expectation? If the above probability is large (say a constant), then although the expected value is a desired value, it means that there is a large probability that it will *deviate* from its expected value. Thus a fundamental task in probabilistic analysis is to compute a probability bound on the deviation from expectation. In particular, in many applications, we would like to show that the random variable is concentrated *near* the expectation, i.e., the probability of the random variable taking values significantly away from the expected value is small. We will see examples of this paradigm in many applications.

Probabilistic tools

We will use three major probabilistic tools to bound deviation from expectation: Markov's inequality, Chebyshev's inequality, and Chernoff's bound. We refer to Appendix A (Appendix C.8) for the statement of these inequalities. Here we will focus on applying these inequalities in algorithmic applications.

8.8.1 Load Balancing: Balls into Bins Paradigm

A basic paradigm in probabilistic analysis is the "balls into bins" paradigm. We are given m balls and n bins. Assume that each ball is thrown randomly among the bins, i.e., the location of each ball is independently and uniformly chosen at random from the n possibilities. A common question that pops up in many applications is to bound the maximum number of balls in any bin. We will use this setting to illustrate the application of Chernoff bound in a direct way.

Theorem 8.10

Assume that m balls are thrown independently and uniformly at random into n bins. Let L be the maximum number of balls in any bin.

1. If $m \geqslant n \log n$, then $L = \mathcal{O}(m/n)$ whp.
2. If $m = n$, then $L = \mathcal{O}(\log n)$ whp.

Proof. Define the indicator random variable X_i^k as: 1 if the ith ball falls into the kth bin and 0 otherwise. Let L^k the number of balls that end up in bin k. Then, $L^k = \sum_{i=1}^m X_i^k$. As usual, we first calculate the expectation of L^k:

$$\mathbb{E}\Big[L^k\Big] = \mathbb{E}\Bigg[\sum_{i=1}^m X_i^k\Bigg] = \sum_{i=1}^m \mathbb{E}\Big[X_i^k\Big] = \sum_{i=1}^m \mathbb{P}\Big(X_i^k = 1\Big)$$

Since $\mathbb{P}\Big(X_i^k = 1\Big) = 1/n$, we have $\mathbb{E}\Big[L^k\Big] = m/n$.

We now apply the Chernoff bound (Theorem C.9, part 1c) to the random variable L^k:

1. If $m \geqslant n \log n$, then $\mu = m/n \geqslant \log n$ and
$$\mathbb{P}\Big(L^k \geqslant 6\mu\Big) = \mathbb{P}\Big(L^k \geqslant 6 \log n\Big) \leqslant 2^{-6\log n} = 1/n^6.$$
2. If $m = n$, then $\mu = m/n = 1$ and
$$\mathbb{P}\Big(L^k \geqslant 6 \log n \mu\Big) \leqslant 2^{-6\log n} = 1/n^6.$$

Note that to get a high probability bound, we need to use $\Theta(\log n)$ factor deviation.

Let random variable L denote the maximum number of balls in any bin. Then $L = \max_{k=1}^n L^k$. Note that the random variables L^k are not independent. Also the max function is not linear, so we cannot even compute expectation of L via linearity of expectation. However, we can appeal to the union bound (Theorem C.2), to bound the probability that there are more than $6 \log n$ balls in any bin as follows. Let the event B_k denote the (bad) event that more than $6 \log n$ balls landed in bin k. We just showed that this probability is at most $1/n^6$. By union bound,

$$\mathbb{P}(\cup_{1 \leqslant k \leqslant n} B_k) \leqslant \sum_{k=1}^n \mathbb{P}(B_k) \leqslant n/n^6 = 1/n^5$$

Thus the probability that no bin receives more than $6 \log n$ balls is at most $1/n^5$. Note that this bound is $\mathcal{O}(m/n)$, when $m \geqslant n \log n$. The theorem follows. □

8.9 High Probability Analysis of Randomized Quicksort

We showed in Section 5.1.3 that randomized Quicksort makes $\mathcal{O}(n \log n)$ comparisons on average. Here, we would like to show a much stronger result: The number of comparisons made by Quicksort is $\mathcal{O}(n \log n)$ with high probability (whp). Thus, in other words, we are showing that the performance of randomized Quicksort is "concentrated" near its average, almost always.

One way to show the high probability bound is to use the approach of the proof of Theorem 8.2. As in Theorem 8.2, let T be the random variable that denotes the number of comparisons in a run of randomized Quicksort. Then, as in the proof of Theorem 8.2, we write

$$T = \sum_{i=1}^{n} \sum_{j>i} X_{i,j}.$$

Unfortunately, one cannot use Chernoff bound directly to show concentration of T, since it is the sum of *dependent* random variables. We use a different approach.

We can view an execution of the randomized Quicksort algorithm (a divide and conquer algorithm) using the following (familiar) recursion tree:

- The root of the recursion tree is the (initial) problem of sorting a given set of n distinct numbers; and a leaf is a subproblem of sorting a singleton set.
- An internal node of this tree is a subproblem of sorting a set S (of size greater than 1).
- Its left child (if any) is the subproblem of sorting a set $S_1 \subset S$ consisting of elements smaller than (or equal to) the pivot.
- Its right child (if any) is the subproblem of sorting a set $S_2 \subset S$ consisting of elements larger than the pivot.

One can analyze the number of comparisons made by QuickSort using the above recursion tree. The number of comparisons in each level of the recursion is bounded by $\mathcal{O}(n)$. Thus the total number of comparisons in (nh), where h is the depth of the recursion tree. To show a $\mathcal{O}(n \log n)$ high probability bound on the total number of comparisons, it is enough to show that the *height* of the tree is bounded by $\mathcal{O}(\log n)$ with high probability.

High Probability Analysis

Theorem 8.11

Randomized Quicksort runs in $\mathcal{O}(n \log n)$ time with high probability, i.e., with probability at least $1 - 1/n^b$, for some constant $b > 0$.

Proof. Let the size of the set to be sorted at a particular node be S. A node in the execution tree is labeled **good** if the pivot element divides the set into two parts, each of size not exceeding $2|S|/3$. Otherwise the node is called **bad**.

We now have:

1. The probability of a node being labeled good is 1/3. Why? As explained in Section 8.4, if the pivot is selected among the middle third of the elements, then the size of each of the two subproblems does not exceed $2|S|/3$.
2. The number of good nodes in any root to leaf path is bounded by $\log_{3/2} n = c \log n$ for some (fixed) constant c. This is because each good node reduces the sizes of both of the subproblems by a factor of $2/3$. Hence, this can happen for only $\log_{3/2} n$ times — and thus only so many good nodes are possible.

The main idea of the proof is to first bound the length of a root to leaf path in the recursion tree. Let us fix one root to leaf — call it P. Each node in this path is labelled either good or bad. There can be at most $\log_{3/2} n$ good nodes in the path. We now bound the probability that the path cannot be very long. More precisely, it cannot be much longer than the average path length of $3c \log n$— the reason being that since each node has a probability $1/3$ of being good, on average we expect a path length of $3c \log n$.

We next bound the probability that a path of length $ac \log n$ (for some constant $a > 1$) will have at most $c \log n$ good nodes. The mean $\mu = \frac{1}{3}(ac \log n)$. Using the Chernoff bound (lower tail, Theorem C.9, part 1c)

$$\mathbb{P}(X < c \log n) = \mathbb{P}\Big(X < (1 - (1 - \frac{3}{a}))\mu\Big) \leqslant e^{-\mu(1-3/a)^2(1/2)} \leqslant 1/n^2$$

for a suitably large (but fixed) constant a.

We have thus shown that a root to leaf path cannot be longer than $ac \log n$ with probability at least $1 - 1/n^2$. To bound the height of the recursion tree, we have to show that *all* root to leaf paths are not longer than $ac \log n$ with high probability. Let ℓ be the number of root to leaf paths ($\ell \leqslant n$ there are at most only n elements). For $1 \leqslant i \leqslant \ell$, let B_i be the (bad) event denoting the probability that path i has length longer than $ac \log n$. By union bound,

$$\mathbb{P}(\cup_{1 \leqslant i \leqslant \ell} B_i) \leqslant \sum_{1 \leqslant i \leqslant \ell} \mathbb{P}(B_i) \leqslant n/n^2 = 1/n$$

Thus with probability at most $1/n$ no root to leaf path has length more than $ac \log n$. Since the total work done at each level of the tree is $\mathcal{O}(n)$, the running time is bounded by $\mathcal{O}(n \log n)$ with high probability. □

8.10 A Randomized Data Structure: Skip Lists

Data structures are needed for *efficient implementation* of algorithms. Choosing the right data structure can also help in *simplifying* the implementation of an algorithm. Thus an important aim in algorithm design is to *organize* data in a suitable way to efficiently support certain kinds of operations on the data stored.

Consider the problem of storing and searching a database. For example, an university that wants to store and retrieve student records identified by their IDs — typically called *keys*. We want a data structure that can INSERT, SEARCH and DELETE, given any key. What is the best way to organize the keys?

Dictionary Data Structure

A *dictionary* data structure supports operations:

- *INSERT*(D, k) — Insert key k (along with its record) into the dictionary D.
- *DELETE*(D, k) — Remove key k (and its record) from the dictionary D.
- *SEARCH*(D, k) — Return (the record associated with) key k if it is present in D.

Array	Insert	Delete	Search
Unsorted	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sorted	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Table 8.1: Performance of Operations on a Dictionary implemented as a sorted or unsorted array

Implementing a Dictionary

How to implement a dictionary data structure? One simple way is using an array: store the keys in an array. There are two ways to store the keys in an array — *sorted* or *unsorted* which determines the time needed for the 3 operations. Assume that there are n keys currently in the array A — $A[1], A[2], \ldots, A[n]$. We perform INSERT, DELETE, and SEARCH operations. The costs for each of these operations are given in Table 8.1. We note that at one of these operations take linear time in an array, whether it is sorted or unsorted.

Static vs. Dynamic Dictionary

If the dictionary is *static* — i.e., all the insertions take place in the *beginning* at once, and there no changes after that — then a *sorted* array is a good option — SEARCH takes *logarithmic* time. If the dictionary is *dynamic* — insertions and/or deletions happen all the time — then array performs poorly. At least one operation takes *linear* time. We need a better way to implement a dynamic dictionary.

Skip List Data Structure

Skip list is a randomized data structure that efficiently supports the DICTIONARY operations of INSERTION, DELETION, and SEARCH of an element. It is a simpler alternative to deterministic data structures such as AVL trees and red-black trees which have a $\mathcal{O}(\log n)$ worst-case time for all the dictionary operations but are complicated to specify and implement. Skip lists, on the other hand, are conceptually much simpler and also easier to implement. However, since it is randomized, the $\mathcal{O}(\log n)$ time for all dictionary operations is guaranteed with high probability.

In a skip list, many linked lists are joined together. Each linked list is associated with a level as follows. We order these elements (in increasing order) and arrange them in a linked list. This is level 0 and call this list L_0. Assume that the first element in the list is always $-\infty$. Recursively, we define level i of the list as follows. An element in level i is obtained by selecting each element in level $i-1$ with probability $1/2$ *independently* of other elements. Copies of the same element in different levels are joined by a doubly linked list (so one can go up and down levels easily). See Figure 8.2.

Search, Insertion, and Deletion

To search for an element e we start from the left most element $(-\infty)$ of the *top* list and keep going right until we hit the largest element of the list that is *smaller than or equal* to e. We then go to that element and go one level down and do the same, until we reach level 0. (We will assume that we always go to level 0 even if we find the element in a higher level — in practice, anyway, the record associated with the key might be stored

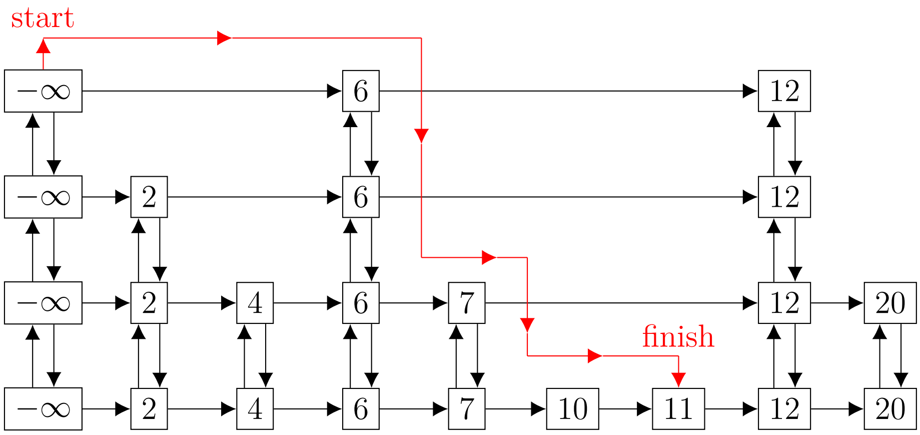


Figure 8.2: A skip list data structure. The path taken to search for element 11 is shown in red.

only on level 0). Search takes time proportional to the number of levels and the total number of right links traversed.

Insertion of an element e is done by searching first and going down to level 0. Suppose we reach an element f in level 0. We insert e between f and its successor. We toss a fair coin. If we get Heads, we insert e one level up (above the copy in level 0), otherwise we stop. We keep inserting e in higher levels, until we get Tails. Deletion involves search and deleting all occurrences of the element. For deletion, one needs to visit the node in level 0 that points to the node being deleted; this can be easily accomplished by going down the level on the element that occurs just before the element being deleted. For example, if we wanted to delete 7 in the skip list of Figure 8.2, we search for 7. When we encounter 7 in level 1 we go back to 6 (the predecessor of 7) and and go down the level, till we land on 6 at level 0. This makes it easier to delete 7 by making 6 point to 10 (the successor of 7 in the list).

Insertion and Deletion takes time proportional to Search plus the height of the element. We next show that both (Search time and height) are bounded by $\mathcal{O}(\log n)$ whp.

8.10.1 Analysis

Height

Let H_i denote the height of element x_i — the highest level to which it belongs. Define the height H of the Skip list to be the maximum of over all the elements' heights.

Lemma 8.12

$\mathbb{P}(H \geqslant 3\log n) \leqslant 1/n^2$.

Proof. Each H_i is *geometrically distributed* (see Appendix C.7.2) with parameter 1/2. In other words, the probability for H_i to be at least $3\log n$, there at least the first $3\log n$

tosses should be Heads and the last should be Tails. The probability of this happening is:

$$\begin{aligned}\mathbb{P}(H_i \geqslant 3\log n) &= \sum_{t\geqslant 3\log n} \mathbb{P}(H_i = t) = \sum_{t\geqslant 3\log n} (1/2)^{t+1}\\ &= (1/2)^{3\log n+1} \sum_{t\geqslant 0} (1/2)^t = (1/2)^{3\log n}\\ &= 1/n^3.\end{aligned}$$

Since there are n elements in total, by union bound, $\mathbb{P}(H \geqslant 3\log n) \leqslant n/n^3 = 1/n^2$. □

Search

The cost of a search is equal to the number of "down" traversals plus number of "right" traversals. The number of down traversals is equal to the height of the Skip list (note: we assume that we always go to level 0). We start from $-\infty$ at the top list and in every step, we either go down or right. The important thing to note is that each down traversal is due to "Heads" outcome of the corresponding element, i.e., the element that we are traversing as we go down (note that, if we are going down on the $-\infty$ element, the corresponding element is the one that is to the right of $-\infty$). Each right traversal is due to a "Tails" outcome of the corresponding element, i.e., the element that was reached by going right in the current list. This is easy to see if we *reverse* the search path from the bottom level (L_0) — starting from where the search ended — to the $-\infty$ element at the top level. Then, each time we go up, it is because of "Heads" outcome at this element; and each time we go left it is because of a "Tails" outcome of this element (hence this element terminates at this level). Thus the total number of traversals is the *number of times needed to toss a fair coin in order to obtain H heads*, where H is the height of the Skip list.

Lemma 8.13

Let T be the random variable denoting the total number of traversals for a *single* search. Then $T = \mathcal{O}(\log n)$ whp.

Proof. Since $H < 3\log n$ whp (by Lemma 8.12), we condition as,

$$\begin{aligned}\mathbb{P}(T > m) &= \mathbb{P}(T > m \mid H < 3\log n)\,\mathbb{P}(H < 3\log n) + \mathbb{P}(T > m \mid H \geqslant 3\log n)\,\mathbb{P}(H \geqslant 3\log n)\\ &\leqslant \mathbb{P}(T > m \mid H < 3\log n) + \mathbb{P}(H \geqslant 3\log n)\end{aligned}$$

For the above we applied the identity:

$$\begin{aligned}\mathbb{P}(A) &= \mathbb{P}(A \mid B)\,\mathbb{P}(B) + \mathbb{P}\left(A \mid \overline{B}\right)\mathbb{P}\left(\overline{B}\right)\\ &\leqslant \mathbb{P}(A \mid B) + \mathbb{P}\left(\overline{B}\right)\end{aligned}$$

for any events A and B, where $\overline{B}$ denotes the complement of B.

Fix $m = 72\log n$. To bound the first term, we bound the probability that there will be less than $3\log n$ heads when tossing a fair coin $72\log n$ times. We expect $36\log n$ Heads. Let Z denote the number of heads. By Chernoff bound (lower tail),

$$\mathbb{P}(Z < 3\log n) = \mathbb{P}(Z \leqslant (1 - 11/12)(36\log n)) \leqslant e^{-3\log n(11/12)^2(1/2)} \leqslant 1/n^2$$

Hence, $\mathbb{P}(T > m) \leqslant n^{-2} + n^{-2} \leqslant 2n^{-2}$. □

By union bound, the probability that any search (i.e., of any of the n elements) takes more than $\mathcal{O}(\log n)$ time is at most $2/n$.

8.11 Probably Approximately Correct (PAC) Learning

The notion of probably approximately correct (PAC) learning was introduced by Valiant in the early 1980s which started the field of computational learning theory. The goal of PAC learning is to provide a framework for learning algorithms using probability theory. In particular, Chernoff bounds play an important role in the analysis of PAC learning.

Learning algorithms vs. conventional algorithms

Learning algorithms are, in some ways, quite different from other algorithms in this book, and in fact, different from the traditional notion of algorithms. The algorithms we have seen so far conform to the traditional notion, i.e., the algorithm — designed for a particular problem — takes an input instance of the problem and outputs a solution to the particular input. For example, for the problem of sorting, the MergeSort algorithm, takes a set of elements and outputs them in sorted order. Learning algorithms are different. They can be considered as "meta" algorithms, since their output is an "algorithm." Learning algorithms typically (not always) take as input "training" data (i.e., data with known solutions) and output an *algorithm (or rule)* that performs well on "test" data (i.e., new data or data with unknown solutions).

Classification problem

To illustrate the above concepts we will consider a fundamental problem underlying many machine learning applications, namely *classification*. Here the goal is to efficiently "learn" a good and simple classifying algorithm (or rule) from a given set of training data. The concepts of "learn", "efficient", "good" and "simple" have to be formalized. To do this we introduce a very general paradigm of learning called *probably approximately correct (PAC)* learning. We will then discuss applications to the classification problem.

Suppose we have to classify emails into two categories: *spam* or *not spam*. This is a real-world classification problem used everyday in email filters. How to design an algorithm for achieving this? This is actually a difficult problem for a traditional algorithmic application, as the input email can be pretty much anything. However, this set up is ideal for a learning algorithm. A learning algorithm will first be fed a certain number of "training" emails with known classification. If the number of training emails are large, then the learning algorithm may "learn" enough about the setting, i.e., able to figure out a *general* rule which can be applied to classify new emails. (This is how a spam filter actually works; it gets better as it sees more email.)

PAC learning concepts

PAC learning is a simple but robust framework to address the above problem and gives a formal way to define learning and its complexity. To formalize the concept of learning, Valiant introduced a simple but key assumption, called the *stationarity assumption* which is as follows. We will assume that the training and test instances of a learning problem are coming from the same domain, and, more importantly, is drawn under the *same* probability distribution.

The problem domain is typically called the *instance space*, denoted by U. For example, in the case of emails, the instance space consists of all email messages. The goal is to classify each email into two categories — spam or not spam, i.e., *binary* classification.[3] To be concrete, we will assume our instance space to be points in d dimensions (i.e., R^d) or in $\{0,1\}^d$. This corresponds to data that is described by d real-valued or Boolean "features." Boolean features for email messages could be the presence or absence of certain words, for example.

Thus the learning algorithm is given a set of labeled training examples, which are points in instance space U along with their correct classification. This training data could be a collection of email messages, each labeled as spam or not spam. The input to the learning algorithm is the set of training examples and its output will be a classification algorithm that will perform well on new data. Note a trivial learning algorithm simply stores the training data and replies the correct answer on only the training data and replies arbitrarily (say randomly) on new data. This algorithm is useless for learning. A key feature of learning algorithm is *generalization*, i.e., able to come up with a "simple" rule that performs well not only on the training data but on the test data as well. The notion of "simple" can also be formalized.

8.11.1 Learning Framework

We assume that data — both training and test data — is drawn from the instance space U according to a fixed (but unknown) probability distribution D. This is how we formalize the notion that training and test data are "similar." Of course, if test and training data have very different classifications (which is possible if they are drawn from different distributions), then it is not possible for a learning algorithm to predict correctly test data based on training data.

Let us assume that the instance space is *finite* throughout this discussion, e.g., as mentioned earlier they can be points in $\{0,1\}^d$. A classification function (rule) maps each point (example) of the instance to *yes (positive)* or *no (negative)*. For example, if the instance space consists of set of all emails (defined by presence or absence of d features), then a classification function maps each instance to "spam" or "not spam." Let F_c be the correct classification rule, i.e., the rule that correctly classifies all the instances. The learning algorithm should try to output a classification rule F_a such that under the distribution D, the *error probability*, defined as $err(F_a) = \mathbb{P}(F_c \neq F_a)$ is minimized. This error probability called as *true error* is the probability that an example is not correctly classified by rule output by the learning algorithm assuming that the examples are drawn randomly according to probability distribution D. Thus, we can define true error as follows.

Definition 8.2 ▶ True error

The true error of a classification rule F_a is simply the probability that it errs on a randomly drawn example from the instance space under distribution D. In other words, it is $err(F_a) = \mathbb{P}_{x\sim D}(F_c(x) \neq F_a(x))$, where F_c is the correct classification rule.

On the other hand, the *training error* (also called empirical error) of F_a is the fraction

[3] We will henceforth deal with binary classification, but the discussion can be generalized to multi-valued classification as well.

of training examples in which the function F_a is incorrect. In other words, we have the following definition.

Definition 8.3 ▶ Training error

Let $Tr \subseteq U$ be the set of training examples, and $E = \{u \in Tr | F_a(u) \neq F_c(u)\}$ be the set of training examples which are classified incorrectly by a classification rule F_a. Then the training error of F_a is defined as $trerr(F_a) = |E|/|Tr|$.

The above notion of errors allow us to define a notion of learning a "general" classification rule (later as we will see it will also allow us to characterize 'simple" rules), i.e., a rule that learns something general about the underlying instance space, rather than just "memorizing" the training examples. Indeed, a simple classification rule that simply stores the training examples will return the correct answer on the training examples, but can answer wrongly on new test examples. This is not really "learning"; it is called "overfitting" the training data. On the other hand, we want the learning algorithm to output a classification rule that has small *true error probability* (over all examples).

Hypothesis and hypothesis class

To formally handle the above issues such as overfitting, generalization, simplicity etc., a convenient notion is that of a *hypothesis* and a *hypothesis class*. A hypothesis is simply a subset of the instance space U. There is a one-to-one correspondence between a hypothesis and a classification rule. Given a classification rule f, the examples that are classified as positive belong to the hypothesis and those that are classified as negative do not. A hypothesis class over the instance space U is a collection of subsets of U. In other words, it is a set of classification rules. We will limit our learning only from a given hypothesis class, i.e., our goal is to learn the best classification rule from the hypothesis class that fits the training data. By "best", we mean a rule that has small true error compared to F_c, the correct classification rule.

An example of an hypothesis class for the email classification problem might be set of all *disjunctions* (i.e., clauses) over d features represented by boolean variables $x_1, \dots, x_d$ which take values 1 or 0 depending on whether the feature is present or absent. For example, a classification rule for spam can be the disjunction $x_2 \vee \bar{x_3} \vee x_7$, consisting of presence of one of the features x_2 or x_7 or absence of feature x_3. Other features are irrelevant. That is an email is classified as "spam" if it has x_2 or x_7 or does not have x_3. The number of all such disjunctions is 3^d (see Worked Exercise 8.3).

In this way, by choosing a suitable hypothesis class, say by choosing a class where all classifications can be expressed as "simple" functions, one can capture simplicity.[4] Generalization is also captured since we are interested in learning only such functions, not arbitrary functions. Using "simple" rules to classify data (with reasonably low true error) is considered preferable to "complicated" rules (which can be captured by arbitrary functions). This is known as *Occam's razor*, which informally means that "in general, simpler rules or explanations are preferable compared to complicated ones."

[4] Formally, simpler functions are those that can be described using a lesser number of bits. This is because, the number of such functions are less. For example, the set of all disjunctions over d Boolean variables is at most 3^d, since each variable can be either positive, negative, or don't care. Hence such functions can be described by using $O(\log d)$ bits. On the other hand, the set of all Boolean functions over d variables is 2^{2^d} and hence needs 2^d bits to describe. See Section 8.11.3.

The main goal in PAC learning is to obtain bounds on the training set size (i.e., how large should Tr be) for a given hypothesis class, so that one can find an hypothesis (i.e., a classification rule) so that the true error is low. The main idea is to choose training set of suitable size so that that if an hypothesis has low training error, then it also has low true error. Hence *any* hypothesis in the hypothesis class that has low training error can be chosen.

8.11.2 PAC Learning Theorems

We will show two main theorems of PAC learning. Both relate the size of the training set to the size of the hypothesis class. For these results, we will assume that the hypothesis class is of *finite* size.

The first result gives a lower bound on the number of training examples needed so that if there is an hypothesis in the hypothesis class that has zero training error, then it will have a small true error, i.e., the hypothesis is *probably approximately correct.*

Theorem 8.14 ▶ Zero training error

Let HC be a hypothesis class and let $0 < \epsilon \leqslant 1$ and $0 < \delta \leqslant 1$ be fixed constants. Assume that the training set examples and the test set examples are drawn independently and randomly from the instance space U under (the same) distribution D. If the training set Tr is of size $n \geqslant \frac{1}{\epsilon}(\ln|\mathrm{HC}| + \ln(1/\delta))$, then with probability at least $1-\delta$ every hypothesis $h \in \mathrm{HC}$ that has training error zero has true error at most ϵ.

Proof. We will prove that with probability at least $1-\delta$, every hypothesis in HC that has true error more than ϵ will have training error greater than zero. This will prove the theorem.

Consider any hypothesis $h \in \mathrm{HC}$ that has true error at least ϵ. We will upper bound the probability that h will have a training error 0. The probability that h will have zero training error is the probability that on all the n training examples, h is correct. Since the training examples are drawn independently and randomly according to D, and since h has a true error of at least ϵ, the probability that h will not err on any of the n training examples is at most

$$(1-\epsilon)^n$$

The above bound is for one hypothesis h. By using union bound, the probability that any hypothesis in HC will not err on any of the $n \geqslant \frac{1}{\epsilon}(\ln|\mathrm{HC}| + \ln(1/\delta))$ training examples, is at most

$$|\mathrm{HC}|(1-\epsilon)^n \leqslant |\mathrm{HC}|e^{-\epsilon n} \leqslant |\mathrm{HC}|e^{-(\ln|\mathrm{HC}|+\ln(1/\delta))} = \delta$$

Hence with probability at least $1-\delta$, the training error will be non-zero. □

The above bound on the number of training samples needed for PAC learning is called the *sample complexity* of the hypothesis class. If $\ln|\mathrm{HC}|$ is small then the class is called *efficiently PAC-learnable.*

We note that the previous theorem is applicable if there is an hypothesis in the hypothesis class that has zero training error. The next theorem can be considered as a generalization of the above result and is applicable in a more general setting where there is no such hypothesis. In particular, suppose the best hypothesis in the hypothesis class has small, but non-zero, training error. Still, we would like to say that for such an hypothesis that has small training error, it has small true error. This is called as the *agnostic* setting, i.e., a setting where there is no hypothesis in the class that is "perfect" (with respect to the training set). More precisely, we want to show that every hypothesis in the class that has a small training error will have a small true error as well. This is called *uniform convergence*, i.e., the training errors and true errors are close uniformly over all hypotheses in the class.

Theorem 8.15 ▶ Agnostic setting

Let HC be a hypothesis class and let $0 < \epsilon \leqslant 1$ and $0 < \delta \leqslant 1$ be fixed constants. Assume that the training set examples and the test set examples are drawn independently and randomly from the instance space U under (the same) distribution D. If the training set Tr is of size $n \geqslant \frac{1}{2\epsilon^2}(\ln|\mathrm{HC}| + \ln(2/\delta))$, then with probability at least $1-\delta$ every hypothesis $h \in \mathrm{HC}$ satisfies $|err(h) - trerr(h)| \leqslant \epsilon$, where $err(h)$ and $trerr(h)$ are the true error and training error of h respectively.

Proof. Fix an hypothesis $h \in \mathrm{HC}$. Let X_i be the indicator random variable that h errs on the ith training example, $1 \leqslant i \leqslant n$, where n is the number of training examples. Since the examples are drawn randomly and independently from distribution D, the X_is are independent. Furthermore, the true error of h, $err(h)$, is, by definition, $\mathbb{P}(X_i = 1) = \mathbb{E}[X_i]$. On the other hand, the training error of h, $trerr(h)$ is the fraction of training examples that h errs, i.e., $\frac{\sum_{i=1}^n X_i}{n}$.

Let $err(h) = p = \mathbb{P}(X_i = 1) = \mathbb{E}[X_i]$. Let $trerr(h) = X = \frac{\sum_{i=1}^n X_i}{n}$. By linearity of expectation, $\mathbb{E}[X] = \frac{\sum_{i=1}^n \mathbb{E}[X_i]}{n} = np/n = p$. That is, the probability that the training error X (i.e., the error on a random sample) will deviate from its expected (true) error $\mathbb{E}[X]$ can be bounded by using a Chernoff bound. We use a convenient variant of the Chernoff bound called the Chernoff-Hoeffding bounds (see Theorem C.10 in Appendix C) to bound the probability that $|err(h) - trerr(h)| > \epsilon$:

$$\mathbb{P}(|X - \mathbb{E}[X]| > \epsilon) \leqslant 2e^{-2n\epsilon^2}$$

The above bound is for one hypothesis h. Applying the union bound over all hypothesis in HC, we have the probability that there is any hypothesis $h \in \mathrm{HC}$ such that $|err(h) - trerr(h)| > \epsilon$ is bounded above by

$$|\mathrm{HC}|2e^{-2n\epsilon^2}$$

Plugging $n = \frac{1}{2\epsilon^2}(\ln|\mathrm{HC}| + \ln(2/\delta))$ in the above bound, shows that the above probability is at most δ as required.

□

8.11.3 Examples

Consider the email classification where the instance space is $\{0,1\}^d$ consisting of d boolean features. Consider an hypothesis class HC_1 that consists of all possible classification rules, i.e., all possible boolean functions over the d features represented by boolean variables $x_1, \ldots, x_d$ which take values 1 or 0 depending on whether the feature is present or absent. The number of different boolean functions on d boolean variables is 2^{2^d}. Hence by the PAC learning theorems, the sample complexity of learning an (arbitrary) boolean function is $\mathcal{O}(\ln|\mathrm{HC}_1|) = \mathcal{O}\Big(2^d\Big)$. This means that one has to see an exponential many examples (exponential in the number of features) to learn this hypothesis class, i.e., one has to see essentially all the possible examples! Hence this hypothesis class is not efficiently learnable.

Consider an hypothesis class HC_2 that is the set of all *disjunctions* (i.e., clauses) over d features represented by boolean variables $x_1, \ldots, x_d$ which take values 1 or 0 depending on whether the feature is present or absent. The number of all such disjunctions is 3^d (see Worked Exercise 8.3). Hence by the PAC learning theorems, the sample complexity of learning HC_2 is $\mathcal{O}(\ln|\mathrm{HC}_2|) = \mathcal{O}(d)$. This is efficiently learnable, since the number of examples needed is proportional to the number of features.

Worked Exercise 8.3 shows how one can learn a disjunction that is consistent with a given set of training examples, assuming such a consistent disjunction exists.

8.12 Worked Exercises

Worked Exercise 8.1. Consider the following insertion sort algorithm to sort a given array of n numbers. The first and second numbers are compared; if they are out of order, they are swapped so that they are in sorted order. Then the third number is compared with the second; if it is not in the proper order, it is swapped with the second and then compared with the first. Iteratively, the kth number is handled by swapping it downward until the first k numbers are in sorted order. Determine the expected (i.e., average) number of swaps that need to be made with insertion sort when the input is a random permutation of n distinct numbers, i.e., the input is equally likely to be any of the $n!$ permutations.

Solution. Let $s_1, s_2, \ldots, s_n$ be the sequence of n numbers in sorted order. Let X be the random variable denoting the total number of swaps performed in the insertion sort algorithm. Let X_{ij} be the indicator random variable for the event that the two elements s_i and s_j are swapped, that is,

$$X_{ij} = \begin{cases} 1 & \text{if } s_i \text{ and } s_j \text{ are swapped} \\ 0 & \text{otherwise} \end{cases}$$

Then we have $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

We observe that $\mathbb{E}[X_{ij}] = \frac{1}{2}$ since s_j will appear before s_i in exactly half of random permutations. Therefore,

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} \\ &= \frac{1}{2} \frac{(n-1)(n)}{2} \frac{n^2-n}{4} \end{aligned}$$

□

Worked Exercise 8.2. In RandQuickSort, given an element x in the input array:

1. What is the maximum number of comparisons that x is involved (i.e., the number of times x is compared with other elements) during the course of the algorithm?

2. What is the minimum number of comparisons that x is involved during the course of the algorithm?

3. What is the average number of comparisons that x is involved during the course of the algorithm?

Justify your answers.

Solution. Note that in QuickSort (whether deterministic or randomized), only the pivot is compared with some other element.

1. The maximum number of comparisons any given element is involved is at most $n-1$, if it chosen as a pivot in the very first iteration; after that is it never involved in any other comparison.

2. The minimum number is 1. This can happen, say, if x is the smallest element in the array and in the very first partition, the pivot is chosen as the second smallest element. The smallest element is compared once with this pivot and then never compared with any other element again.

3. The average number of comparisons for a given element x is a bit tricky to compute directly. But we can use the following reasoning. The total average number of comparisons of randomized QuickSort is $\Theta(n \log n)$ (cf. Section 8.4). Since there are n elements the average number of comparisons per element is $\Theta(\log n)$. Strictly speaking this argument is not correct, since this is the average over all elements, while we are *fixing* an element x and asking the average number of comparisons it is involved in (note that in both cases, the average is with respect to the random choices of the QuickSort algorithm). However, we will now argue that the bound of $\Theta(\log n)$ is the correct answer for the latter average as well by a different (correct) argument.

   Fix an element x and let its rank be i (i.e., it is the ith smallest element). Fix any other element y and let its rank be j. Assume without loss of generality $i < j$ (otherwise, the argument below is identical with roles of i and j interchanged). We compute the probability that x and y are compared. Note that this will happen only if one of the two is chosen as a pivot. Furthermore it will happen only if x or y is chosen as a pivot **before** any of the elements whose ranks are in-between i and j (i.e., ranks $i+1, i+2, \ldots, j-1, j$) are chosen as pivots. Why? Because, if a pivot is chosen from the in-between elements first, then x and y will go to different subproblems and will never be compared. Hence the probability that x and y are compared is the probability of choosing x or y from the set of $j-i+1$ elements (x and y and all the in-between elements) which is $\frac{2}{j-i+1}$.

   Hence the average number of comparisons that x is involved is simply the weighted sum of values, where the values are either 0 or 1 (0 means no comparison, and 1 means no comparison) and the weights are the probabilities. We do not have to worry about 0 values, since they contribute 0. Thus the average is simply the sum of the probabilities of x being compared with another element, summed over all other elements:

$$\begin{aligned}\sum_{j\neq i}\frac{2}{j-i+1} &= 2\sum_{k=1}^{n-1} 1/k = 2(1/2+1/3+\cdots+1/n)\\ &\leqslant 2(1/2+1/3+\cdots+1/n)\\ &= 2H_n = \Theta(\log n)\end{aligned}$$

□

Worked Exercise 8.3. Assume that we have an instance space of d Boolean variables, and the target concept we are trying to learn can be represented by a *disjunction* of features. For example, if $F_c = x_1 \vee x_3 \vee \hat{x_5}$, this means that an example is positive if it has either x_1 or x_3 or does not have x_5. Equivalently, any example that is negative should not have x_1, x_3, and should have x_5. In the case of email classification, this would mean that a positive example (i.e., not spam) would correspond to having any x_1 or x_3 or not having x_5. A negative example (spam) will not have x_1 and x_3 and will have x_5.

(1) Show that the number of different disjunctions on a set of d boolean variables is 3^d.

(2) Show how to learn a disjunction that is consistent with a given set of training examples, assuming a disjunction exists that agrees with all the training examples. State and prove your learning algorithm.

Solution.

(1) Each disjunction can either have the variable, or its complement, or not have the variable at all. Thus there are three possibilities for each variable. Hence the total number of different disjunctions is 3^d.

(2) Start with the disjunction $F = x_1 \vee \hat{x_1} \vee x_2 \vee \hat{x_2} \cdots \vee x_d \vee \hat{x_d}$ (this evaluates to 1). Consider every negative example (if any). Any feature that appears in a negative example, cannot appear in disjunction. Hence remove the feature variable that appears in any negative example from F. For example, if feature x_2 appears in a negative example, then remove x_2 from F (but leave $\hat{x_2}$ in F). On the other hand, if a feature does not appear in a negative example, keep it in F and remove its complement. That is, if x_1 does not appears in a negative example, keep x_1 and remove $\hat{x_1}$. Clearly, such a disjunction satisfies every positive example (if any). Also every negative example, will not be satisfied, since every feature that is present in the negative example only its complement is included. Hence the resultant disjunction satisfies all training examples.

□

8.13 Exercises

Exercise 8.1. Show that if tossing a fair coin takes constant time (i.e., bit 0 or 1 can be chosen randomly, each with probability 1/2), then choosing a random element from a set of size k can be done in $\mathcal{O}(\log k)$ time.

Exercise 8.2. Let X denote the runtime of RandQuickSort on an input of size n. Show that $c_1 n \log_n \leqslant X \leqslant c_2 n^2$ for some constants c_1 and c_2.

Exercise 8.3. Prove the correctness and worst-case run time of Random-Select (i.e., in the worst case, it runs in $\Theta(n^2)$ time).

Exercise 8.4. There are three empty boxes. A magician places a prize in one box chosen uniformly at random. You are asked to pick a box (of course you do not know which box contains the prize). The magician then opens an empty box (different than the one you have picked). He asks you whether you would like to change your pick (to the other unopened box) or keep your pick. What is the best strategy? Justify with correct probabilities.

Exercise 8.5. Consider the following game. We start with one blue ball and one red ball in a bin. We repeatedly do the following: choose one ball from the bin uniformly at random, and then put the ball back in the bin with another ball of the same color. We repeat until there are n balls in the bin. Show that the number of red balls is equally likely to be any number between 1 and $n-1$.

Exercise 8.6. Give an example of three random events A, B, and C, such that any two of these events are pairwise independent, but all three together are not independent (i.e., $\mathbb{P}(A \cap B \cap C) \neq \mathbb{P}(A)\,\mathbb{P}(B)\,\mathbb{P}(C)$).

Exercise 8.7. Given an n-element set S, let X and Y be two random subsets of S (each chosen independently and uniformly at random from all possible subsets of S). Determine the probability of the following events:

1. $\mathbb{P}(X \supseteq Y)$.
2. $\mathbb{P}(X \cup Y = S)$.

Exercise 8.8. The following program finds the maximum value in a nonempty set of numbers S of size n.

Algorithm 38 RandMax – Randomized maximum of a set.
Input: A set S of n elements
Output: The maximum in S

```
1: func RandMax($S$):
2:     max = $-\infty$
3:     for $i = 1$ to $n$:
4:         $x$ = RandomChoice($S$)
5:         $S$.Remove($x$)
6:         if $x >$ max:
7:             max = $x$
8:     return max
```

Show that the expected number of times the assignment statement in line 7 is executed is $\mathcal{O}(\lg n)$.

Exercise 8.9. We are given a set of n points in 2 dimensions. Let $p[0]$ and $p[1]$ be the first and second coordinates (respectively) of point p. Each coordinate of each point is an integer. For any two distinct points p_i and $p_{i'}$, we say that $p_{i'}$ is *worse* than p_i if, for $j \in \{0,1\}$, $p_i[j] \geqslant p_{i'}[j]$, and there exists some $j' \in \{0,1\}$ such that $p_i[j'] > p_{i'}[j']$. We say that a point p_i is *interesting* if it is not worse than any other point.

Assume that each coordinate of of each point is chosen *independently and uniformly at random* from the set $[1, \ldots, n^2]$, i.e., each component is a random integer chosen from the first n^2 integers (all choices are made independently).

What is the expected number of interesting points among the n points?

(Hint: Use principle of deferred decisions. See also Problem C.2.)

Exercise 8.10. A permutation $\pi : [1, n] \to [1, n]$ can be represented as a set of cycles as follows. Let there be one vertex for each number i, $i = 1, \ldots, n$. If the permutation maps the number i to the number $\pi(i)$, then a directed arc is drawn from vertex i to vertex $\pi(i)$. This gives a directed graph that is a set of disjoint cycles. Notice that some of the cycles could be self-loops. What is the expected number of cycles in a random permutation of n numbers?

Exercise 8.11. Give an alternate analysis to the **randomized select** algorithm using an approach that is similar to the one we used to analyze **randomized quicksort** in Section 8.4. That is let X be a random variable denoting the total number of comparisons performed during an execution of the randomized select algorithm. Show by a direct analysis (a la Quicksort) that $\mathbb{E}[X] \leqslant cn$ for some constant c (of course, n is the number of elements in the input). You should not assume anything about the rank of the element that you are trying to find, i.e., k in $Random - Select(S, k)$ can be arbitrary.

Exercise 8.12. Assume m balls are thrown randomly in n bins.

1. Assume $m = n$. What is the expected number of empty bins (bins that do not have any ball)?
2. What should be the minimum m if every bin should contain at least one ball with high probability, i.e., with probability at least $1 - 1/n$?

Exercise 8.13. For any two events A and B, show that

$$\mathbb{P}(A) = \mathbb{P}(A \mid B)\,\mathbb{P}(B) + \mathbb{P}\Big(A \mid \overline{B}\Big)\,\mathbb{P}\Big(\overline{B}\Big)$$

where $\overline{B}$ is the complement of the event B.

Exercise 8.14. There is some wastage of space in a Skip list — elements can be repeated and stored multiple times. Show that, if there are n distinct elements, the total space used over all elements is $\mathcal{O}(n)$ in expectation and with high probability. (Hint: For the high probability bound, bound the number of elements, level by level.)

Exercise 8.15. You need a new staff assistant, and you have n people to interview. You want to hire the best candidate for the position. When you interview a candidate, you give them a score, with the highest score being the best and no ties being possible (in other words, there is a strict ordering among the candidates based on the quality from best to worst.). You interview the candidates one by one. Because of your company's hiring practices, after you interview the k^{th} candidate, you either offer the candidate the job before the next interview or you forever, lose the chance to hire that candidate. We suppose candidates are interviewed in a random order, chosen uniformly at random from all $n!$ possible orderings.

We consider the following strategy. First interview m candidates and reject them all; these candidates give you an idea of how strong the field is. After the m^{th} candidate, hire the first candidate you interview who is better than all of the previous candidates you have interviewed (note that, you may end up not hiring anyone under this strategy).

(a) Let E be the event that we hire the best assistant, and let E_i be the event that the ith candidate is the best and we hire him. Determine $\mathbb{P}(E_i)$, and show that

$$\mathbb{P}(E) = \frac{m}{n} \sum_{j=m+1}^{n} \frac{1}{j-1}$$

(b) Bound $\sum_{j=m+1}^{n} \frac{1}{j-1}$ to obtain

$$\frac{m}{n}(\ln n - \ln m) \leqslant \mathbb{P}(E) \leqslant \frac{m}{n}(\ln(n-1) - \ln(m-1))$$

(c) Show that $\frac{m}{n}(\ln n - \ln m)$ is maximized when $m = n/e$, and explain why this means $\mathbb{P}(E) \geqslant 1/e$ for this choice of m.

Exercise 8.16. Suppose we have a Monte Carlo algorithm for a problem that runs in time $T(n)$ for any input size of size n and produces the correct answer (assume that the output is of size at most $\mathcal{O}(n)$) with probability $p(n)$. We are also given a verification algorithm for the problem, that takes an input and its solution and verifies whether the given solution is indeed a correct solution; it takes time $V(n)$ to verify (n, as given above is the input size of the problem). Using the Monte Carlo algorithm and the verification algorithm, show how to obtain a Las Vegas algorithm that always returns the correct answer. What is the expected running time of the Las Vegas Algorithm?

Exercise 8.17. We are given two *unsorted* sets of numbers S_1 and S_2, each of size n (the numbers in each set are distinct). For each number in S_1, there exists (exactly) one corresponding equivalent number in S_2. (For example, think of S_1 and S_2 as consisting of numbers that denote weights; weights in S_1 are in kilograms, whereas weights in S_2 are in pounds, and for every weight in S_1, there is an equivalent weight in S_2.). The goal is to group the equivalent numbers in S_1 and S_2 in pairs. The only operation allowed is comparing any number (say a) in S_1 with any number (say b) in S_2; the comparison will tell whether a is equivalent to b, or whether $a < b$ or $a > b$. Two numbers belonging to the *same* set cannot be compared. Give a randomized algorithm for the problem that takes $\mathcal{O}(n \log n)$ comparisons in expectation and with high probability.

Exercise 8.18. We showed that the Random-Select algorithm of Section 8.6 runs in $\mathcal{O}(n)$ time in expectation. Can we show a high probability analysis for Random Select as we did for randomized Quicksort? That is, can we show that Random-Select runs in $\mathcal{O}(n)$ time with high probability (i.e., probability at least $1 - 1/n$)?

Exercise 8.19. For a random variable X, show that if $\mathbb{E}[X] = C$, then there are values $c_1 \leqslant C$ and $c_2 \geqslant C$ such that $\mathbb{P}(X = c_1) > 0$ and $\mathbb{P}(X = c_2) > 0$.

Exercise 8.20.

1. Show how to compute $\mathbb{E}[X \mid X_1 = x_1, \ldots, X_k = x_k]$ in the deterministic algorithm for maximum cut via derandomization.

2. Show the correctness of the greedy deterministic algorithm — Algorithm 37 — and analyze its running time.

Exercise 8.21. Consider the following algorithm called GoodSelect for finding the kth smallest element which is a modification of the Algorithm Random-Select. We showed that algorithm Random-Select had $\mathcal{O}(n)$ expected number of comparisons. Our motivation for GoodSelect is to design an algorithm that has $\mathcal{O}(n)$ comparisons, not just in expectation, but with high probability.

The main change in GoodSelect is how the pivot is chosen (Step 4 of Algorithm 12). Instead of choosing a random pivot, we do the following: we choose $400 \log n$ elements independently and uniformly at random from the set (the elements are sampled with replacement — so the same element can be selected more than once). Then we sort these $400 \log n$ elements (say, using MergeSort) and take the middle element of this sorted set, i.e., the median. This median element is taken as the pivot. The rest of the algorithm proceeds as usual — see full pseudocode in Algorithm 39.

Algorithm 39 GoodSelect
Input: An array S and the desired order statistic, k
Output: The value of the kth order statistic

```
func GoodSelect($S$, $k$):
    if $|S| \leqslant 400 \log n$:
        MergeSort($S$)
        return $S[k]$
    else:
        $P$ = RandomSample($S, 400 \log n$)
        ▷ $P$ is a random sample (with replacement) of $400 \log n$ elements
        MergeSort($P$)
        $p$ = Median($P$)
        $S_1 = \{x \in S - \{p\} \mid x < p\}$
        $S_2 = \{x \in S - \{p\} \mid x > p\}$
        if $|S_1| == k - 1$:
            return $p$
        else if $|S_1| > k$:
            return GoodSelect($S_1, k$)
        else:
            return GoodSelect($S_2, k - |S_1| - 1$)
```

1. Show that, with high probability, p is a "good" pivot — i.e., with high probability (say with probability at least $1 - 1/n^c$, for some constant $c > 0$), the sets S_1 and S_2 are approximately balanced.

2. Show that the running time (i.e., number of comparisons) of GoodSelect is $\mathcal{O}(n)$ with high probability.

Exercise 8.22. In a casino, there is a machine where you can play the following game: if you put \$1, you will get either \$2 with probability 1/10, or \$4 with probability 1/40 or \$100 with probability 1/1000 or nothing with the remaining probability. Each game is independent of the others. The casino lost \$1000 in the first 10000 games. Obtain a Chernoff bound for the probability of this happening.

Hint: Use the following Chernoff bound that works even for non-Bernoulli random variables, i.e., random variables that can take values other than 0 or 1.

Let $X_1, \ldots, X_n$ be independent random variables such that $X_i \in [a, b]$ for $1 \leqslant i \leqslant n$, i.e., the random variables take values in the range $[a, b]$, where a, b $(a < b)$ are some numbers (can be positive or negative). Let $X = \sum_{i=1}^{n} X_i$. Then for any $t > 0$,

$$\mathbb{P}(|X - \mathbb{E}[X]| \geqslant t) \leqslant 2e^{\frac{-2t^2}{n(b-a)^2}}$$

Exercise 8.23. Let $G = (V, E)$ be an undirected graph with m edges and n vertices. For any subset $U \subseteq V$, let $G[U]$ denote the subgraph induced on U, i.e., the graph with vertex set U and whose edge set consists of all edges of G with both ends in U. Given a number k $(1 \leqslant k \leqslant n)$, your task is to design a polynomial time algorithm that produces a set $U \subseteq V$ of k nodes with the property that the induced subgraph $G[U]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

(a) Give a deterministic polynomial time algorithm for the problem. Show that your algorithm indeed produces the desired set. What is the running time of your algorithm? (Hint: Try a greedy strategy.)

(b) Give a Las Vegas randomized algorithm that has expected polynomial running time. Show correctness and running time analysis.

Exercise 8.24. Consider the instance space $\{0, 1\}^d$ for the classification problem (consisting of d boolean features) and let the hypothesis class HC be the set of all 3-CNF formulas. That is, each hypothesis can be described by a conjunction (AND) of clauses and each clause is a disjunction (OR) of exactly three literals. An example $h \in$ HC is $(x_1 \vee x_2 \vee \hat{x_5}) \wedge (x_9 \vee \hat{x_1} \vee \hat{x_2})$. Note that there is no limit on the number of clauses in a 3-CNF formula. Assume that the correct classification rule is given by some (unknown) 3-CNF formula F_c and we want to PAC-learn it.

1. Give a lower bound on the number of training examples needed so that with probability at least $1 - \delta$, all 3-CNF formulas consistent with all the training examples have true error at most ϵ. Note that $0 < \delta, \epsilon < 1$ are fixed constants. As assumed in the PAC setting, the training and test examples are drawn independently and randomly from the same distribution D (unknown).

2. Give a polynomial time algorithm (polynomial in d) to output a 3-CNF formula that is consistent with all the training examples.

Exercise 8.25. Assume that you throw n balls independently and randomly in n bins (i.e., each ball has a probability of $1/n$ of landing in any bin and the probability is independent of any other ball). Let random variable X denote the number of bins that are *empty* (did not receive any ball).

1. Compute $\mathbb{E}[X]$.

2. Show that X does not exceed $\mathbb{E}[X] + \mathcal{O}\left(\sqrt{n \log n}\right)$ with high probability.

   Hint: Use the fact that the usual Chernoff bound (upper tail) for sum of independent indicator random variables holds even under the condition given below.

Let $X_1, X_2, \ldots, X_l \in \{0,1\}$ be random variables such that for all i, and for any $S \subseteq \{X_1, \ldots, X_{i-1}\}$, $\mathbb{P}[X_i = 1 | \bigwedge_{j \in S} X_j = 1] \leqslant \mathbb{P}[X_i = 1]$. (Such random variables, can be called *negatively correlated*, since the probability that $X_i = 1$ given that $X_j = 1$ $(j < i)$ decreases compared to the unconditioned probability that $X_i = 1$.) Then for any $0 < \delta < 1$, $\mathbb{P}[\sum_i X_i \geqslant (1+\delta)\mu] \leqslant e^{-\mu\delta^2/3}$, where $\mu = \sum_i \mathbb{E}[X_i]$.

Exercise 8.26. Assume that there are n distinct coupons that you need to collect to win a prize. Every time you go to shop you collect a random coupon i.e., every coupon is equally likely to show up. Our goal is to determine the total number of coupons you have to collect till you have all the n distinct coupons.

Show each of the following statements.

1. The expected number of coupons needed to collect all the n distinct coupons is $n \ln n + O(n)$.

2. With high probability (whp) i.e., with probability at least $1 - 1/n^b$, for some $b > 1$, the number of coupons needed is $O(n \log n)$.

3. Furthermore, whp, no coupon is collected more than $O(\log n)$ number of times.

Exercise 8.27. In the bin packing problem, we are given n objects each of size between 0 and 1 (both included). The goal is to pack all the objects using as few *unit-sized* bins as possible. The bin packing problem is a classic NP-hard optimization problem and has no known polynomial time algorithm. However, if the sizes of the objects are assumed to be uniformly distributed, then one can design a simple algorithm that achieves an expected approximation ratio of $1 + o(1)$.

Assume the size of each object is uniformly distributed over $[0,1]$. The goal is to pack all objects in unit-sized bins, such that the total number of bins is minimized.

Consider the following algorithm: Let $\alpha = 1 - \frac{6 \log n}{\sqrt{n}}$.

1. Place each element $x_i \geqslant \alpha$ into a bin on its own. Suppose there are B_1 such elements.

2. Let $N = n - B_1$ be the number of elements yet to be packed.

3. Order the items so that $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_N \leqslant \alpha$.

4. **For** $i = 1, 2, \ldots, \lfloor N/2 \rfloor$

   (a) **if** $x_i + x_{N-i+1} \leqslant 1$ put x_i, x_{N-i+1} into one bin

   (b) **else** put into separate bins.

   Put item $\lceil N/2 \rceil$ into a separate bin if N is odd.

- Show that the expected number of bins required by any solution is at least $n/2$.

- Show that for n sufficiently large, the expected number of bins used by the algorithm is at most $\frac{n}{2} + O(\sqrt{n} \log n)$. (Hint: Show that for $i = 1, 2, \ldots, \lfloor N/2 \rfloor$: $\mathbb{P}(x_i + x_{N+i-1} > 1) \leqslant 1/n$.)

- Conclude that the approximation ratio of the algorithm is $1 + o(1)$.

Exercise 8.28. We are given an instance of a boolean formula in 3-CNF form, i.e., it is a formula which is conjunction (AND) of clauses, where each clause is a disjunction (OR) of 3 literals (each literal is a variable or its complement and each of the 3 variables are different). An example of such a formula is $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4 \vee x_1)$ which is a conjunction of two clauses (each clause has exactly 3 literals). The optimization problem we want to solve is, given a Boolean formula F in 3-CNF form consisting of m clauses and n variables, find a truth assignment (i.e., TRUE or FALSE) to the variables so as to *maximize* the number of satisfied clauses. (In the above example, choosing x_1 as TRUE and x_3 to be FALSE and x_2 to be TRUE will satisfy both the clauses, i.e., make both of them TRUE.)

1. Show using an expectation argument that there is always a truth assignment of the boolean variables that satisfies at least 7/8 fraction of the clauses, i.e., $7m/8$, where m is the total number of clauses. (Hint: Give a random truth assignment to the variables.)

2. Give a Las Vegas algorithm that finds an assignment satisfying at least $7m/8$ clauses and analyze its expected time.

3. Derandomize the algorithm using the method of conditional expectation.

9 HASHING AND FINGERPRINTING

Hashing and fingerprinting are related (and essentially similar) techniques that map numbers and objects from a large domain to a much smaller set of values (typically, positive integers). Both are very efficient and used in lots of applications.

9.1 Hashing

Hashing is a simple but powerful technique with a lot of algorithmic applications. It also can be used to support **Dynamic Dictionary** operations: INSERT, SEARCH, and DELETE (see Section 8.10).

Here is a simple way to motivate the idea of hashing. Assume keys are positive integers: $k_1, k_2, \ldots,$. Assume that you have an array A whose size is *at least as large* as the largest key. A fast way to insert/delete/search is to store key k_i as a boolean value at $A[k_i]$, i.e., $A[k_i] = 1$, k_i is inserted, otherwise 0. This simple strategy gives a $\mathcal{O}(1)$ (i.e., constant) time for INSERT, DELETE, and SEARCH operations. What is the (big) drawback of this strategy? The drawback is that the array's size depends on the *key values* and not on the *number of keys*. Ideally, if we have m keys, we want to store it using an array of size $\mathcal{O}(m)$, regardless of the values of the keys.

Hashing is a clever way to implement the above strategy without the above drawback. Assume that the keys are drawn from a Universe set U. U is typically large, say $|U| = 10^9$ — e.g., the set of all social security numbers. Also, the *number* of keys — n — is usually much smaller than $|U|$, e.g., consider $|U|$ to be the population of USA, whereas n is the number of students in a University.

Hash function

Let the array — called as *hash table* — be of size m — used to store the keys. (Usually m will be somewhat larger than n.) A *hash function* h is simply a function from U to $\{0, \ldots, m-1\}$. A hash function maps keys from a Universe (typically a large set) to a much smaller set of numbers $\{0, \ldots, m-1\}$. See Figure 9.1. Note that m is the size of the hash table.

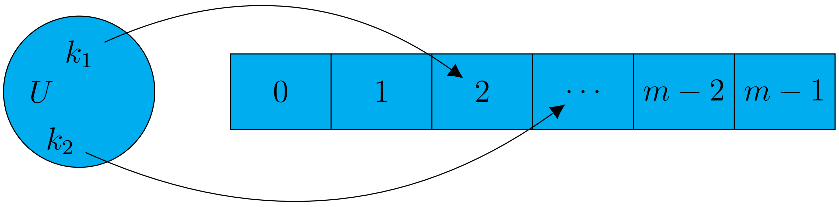


Figure 9.1: A hash function h maps elements from a large set U to a smaller set $\{0, \ldots, m-1\}$.

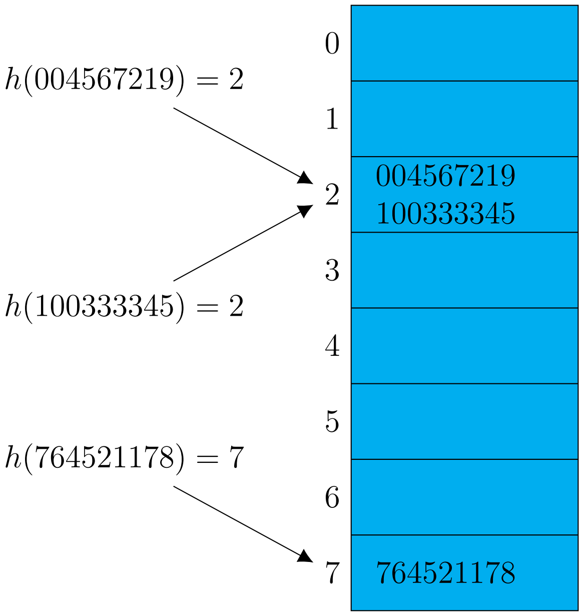


Figure 9.2: Two elements are hashed to the same value (2), causing collision.

Example

Figure 9.2 gives a simple example of hashing values to a hash table.

Hash function properties

What properties should a hash function satisfy? We would like it have the following properties.

- Should be a *deterministic* function, i.e., $h(k)$ is a fixed hash value for a fixed input k. (Why?)
- Maps values from a *large* universe U to a much *smaller* valued set $\{1, \ldots, m\}$, where $|U| >> m$.
- Easy to compute.
- Distribute the keys more or less evenly among the m slots — *balanced* key distribution. That is, there should not be too many collisions: A **collision** occurs when $h(x) = h(y)$, for two keys $x \neq y$.

Choosing a Hash Function

The following are some examples of hash functions. Note that they all satisfy that the range of the function is the set $\{0, \ldots, m-1\}$ and hence, the result of the hash provides an index to a hash table with size m. This is the reason why these functions have a "mod m" at the end.

- $h(k) = k \bmod m$, where k is the key and m is the size of the hash table. Typically m is chosen to be a prime (why?). In practice, it has been observed that a prime not close to a power of 2 is generally a good choice for m.

- $h(k) = (ak + b) \bmod m$, where $a > 0$ and $b >= 0$ are some constants. If m not a prime, let $p > m$ be a prime: $h(k) = ((ak+b) \bmod p) \bmod m$. Typically, p is chosen larger than the size of the largest key in the universe.

Note that it is hard to avoid collisions. Indeed, since we are mapping $|U|$ elements to m values, for *every* hash function, there is a subset B of size at least $|U|/m$ elements ("bad" set) that map to the *same* value (because of pigeon hole principle). Thus *whatever* hash function we use, if an "adversary" chooses a subset of elements from the bad set, then all of them hash to the same value! We will use randomization to get around this seemingly impossible task.

Exercise 9.1. A hash function $h : U \to M$ is **perfect** for a set S if it causes no collisions for pairs in S.

- For any given set S such that $|S| \leqslant m$, show that there is always a perfect hash function for S.

- For any S such that $|S| > m$, show that there is **no** perfect hash function.

- If $|U| > m$, show that there is no perfect hash function for all $S \subset U$, such that $1 < |S| \leqslant m$.

9.1.1 Hashing by Chaining

Even with the "best" choice of an hash function, there will be some collisions. We store all the elements hashed to the same slot as a list, e.g., a linked list or container (vector). Given a hash table $T[1..m]$, $T[k]$ points to a container of all the elements hashed to $T[k]$. See Figure 9.3.

9.1.1.1 INSERT, SEARCH, and DELETE

All the above operations are now easy under chaining.

- INSERT k: add k to the container $T[h(k)]$.

- SEARCH k: search in $T[h(k)]$.

- DELETE k: search in $T[h(k)]$ and delete the element.

The time for each of the above operations is proportional to the *length* of the respective lists.

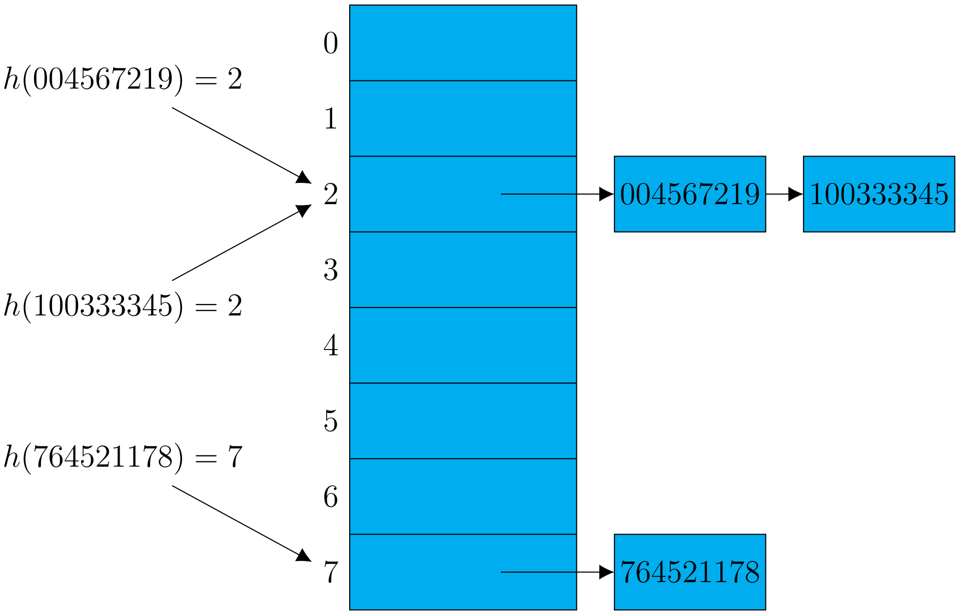


Figure 9.3: Hashing with chaining. The elements that hash to the same value (2) are stored in a linked list.

9.1.1.2 Analysis of Hashing with Chaining

Let n be the number of keys stored in the table and m be the number of slots. The worst case insert time is clearly, $\mathcal{O}(1)$ whereas the worst case search/delete time is $\mathcal{O}(n)$, since all the keys can be hashed to the same slot.

What is the search/delete time for a "good" hash function (defined below)? Define **load factor** $\alpha = \frac{n}{m}$. Assume that the hash function is *good*, i.e., it distributes keys (approximately) *evenly* among the slots. In particular, consider a hash function that is fully *random*, i.e., each of the n elements is equally likely to be mapped to any of the m slots, independent of other elements.[1] Then we can show the following theorem.

Theorem 9.1

Assuming a random hash function (whose hash value can be computed in time $\mathcal{O}(1)$), the expected time for search (and delete) is $O(1)$ if $\alpha = n/m = O(1)$.

Proof. Since the hash function maps each key to a particular slot with probability $1/m$, the expected number of keys hashed to a particular slot is $n/m = \alpha$.

Taking into the time to compute the hash value, the overall time for search (and deletion) is $\mathcal{O}(1) + \mathcal{O}(\alpha) = \mathcal{O}(\alpha)$. Thus, if we choose $m = \Theta(n)$, then search/delete takes expected $\mathcal{O}(1)$, i.e., *constant* time. □

We note that to keep the load factor constant (say between 1/2 and 1), one can adapt the size of the hash table to the number of elements currently in the table. When the load factor goes above 1, we double the size of the table and rebuild it by hashing all the items with a new appropriately chosen hash function. Similarly, if the load factor goes

[1] For example, a hash function chosen randomly among all possible hash functions will have this property (Exercise 9.2). However, such an hash function will be inefficient to represent. In Section 9.2, we will show how to construct a function that is very efficient to represent, while still having the property of distributing elements uniformly on *average*.

below half, we decrease the hash table by a factor of 2 and rebuild. It is easy to establish that the *amortized* cost of rebuilding per insertion or deletion is constant (Exercise 9.3).

9.1.2 Cuckoo Hashing

As we saw in the previous section, hashing by chaining is a simple data structure that gives expected constant search time (assuming a good hash function), but can have linear worst-case time. There are many other hashing schemes known that have similar properties. A popular hashing scheme that does very well in practice is *linear probing.* In linear probing, instead of using a linked list to handle collisions, we handle all collisions *within* the hash table itself. If we hash an element x to a slot, say k, and find that slot k has already been taken (due to collision), we simply check slots $k+1$, $k+2$ and so on (we cycle to slot 1, if we reach the last slot in the array) and insert x to the first empty slot that we find. This kind of hashing technique is called *open addressing*, where elements can be inserted into slots other than the values they hash to. One can show if we use a random hash function and if the load factor is (say) 1/2, then one can do search, insert and delete in constant *expected* time. Exercise 9.12 explores the complexity of linear probing as well as handling deletions.

Unlike, hashing with chaining and linear probing, which can guarantee only constant *expected* search time, in this section we will see a hashing technique called *cuckoo hashing* which guarantees constant *worst-case* search time!

Cuckoo hashing technique

In cuckoo hashing, the main idea is to use two hash functions h_1 and h_2. To insert an element x into the hash table, we hash x using h_1 to obtain the slot $a = h_1(x)$ to insert x. If slot a is occupied, then we still insert x in a and "kick off" the current occupant of the slot a (say y) to the alternate slot of y.[2] That is, if a was the slot for y determined by hash function h_1 (say), then we insert y into the slot $b = h_2(y)$. If slot b was vacant, then we are done. Otherwise, we eject the current occupant from slot b, say z, and repeat the process with z, i.e., insert z in its alternate slot and so on. If the process finishes (i.e., when there are no more evictions), then we are done with our insertion procedure. Otherwise, if we find that the process starts repeating — i.e., we encounter the eviction of the same element again — then we abandon the hash table and rebuild the hash table from scratch with the current set of elements using *new* hash functions. Note that the insertion procedure will take at most n steps before it starts repeating, since there are only so many elements. Figure 9.4 shows a cuckoo hash table and illustrates a few insertions. The pseudocode for the insertion procedure is given in Algorithm Cuckoo-Insert.

[2] This is similar to some species of the cuckoo bird including the koels which exhibit brood parasitism — the eggs of a cuckoo bird are dropped into the nest of a host bird (of different species) and the cuckoo chick kicks off eggs or youngs of the host bird from the nest.

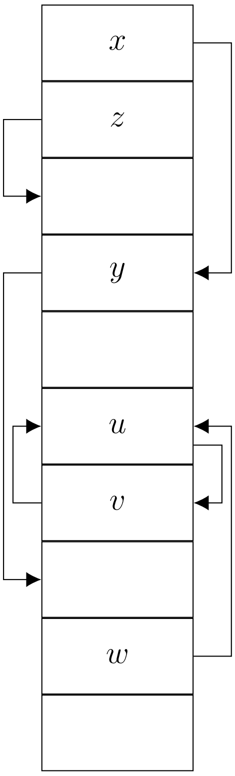


Figure 9.4: A cuckoo hash table. The arrow on the edges connecting two slots show the alternate position that the element can be inserted. For example, the alternate position for element x is the slot occupied by element y. Hence if a new element t is inserted such that slot $h_1(t)$ points to the slot occupied by x, then x is evicted to its alternate slot $(h_2(x))$ which is the slot currently occupied by y; y is then evicted to the alternate slot which is currently empty and the insertion succeeds. On the other hand, if another new element p is inserted and $h_1(p)$ points to the slot occupied by w, and $h_2(p)$ be the slot occupied by u. Then inserting p in $h_1(p)$ leads to w being evicted to its alternate slot, currently occupied by u. u's alternate slot is v and hence v will be evicted and put in v's alternate slot which is now occupied by w. We insert w in its original slot which leads to eviction of p which repeats the whole process, since there is a cycle. In this case the insertion fails.

Algorithm 40 Cuckoo-Insert — Insert an element x in the hash table T
Input: An element x
Output: None

```
func CUCKOO-INSERT($x$):
    if $T[h_1(x)] == x$ or $T[h_2(x)] == x$:    ▷ $x$ is already present
        return
    $a = h_1(x)$
    for $i = 1$ to $n$:
        if $T[a]$ == Null:
            $T[a] = x$
            return
        SWAP($T[a], x$)
        if $a = h_1(x)$:
            $a = h_2(x)$
        else:
            $a = h_1(x)$
    Rebuild $T$    ▷ Rebuild the table using new hash functions
    CUCKOO-INSERT($x$)
```

Analysis

It is clear that time to search is *constant*, since an element x if present in the hash table, will be in one of the two slots: $h_1(x)$ or $h_2(x)$. Similarly the time to delete is also a constant.[3]

[3] To maintain a constant load factor, say, $> 1/2$, then one may need to appropriately reduce the size of the table if deletions decrease the number of elements to less than half the table size.

The main issue to analyze in cuckoo hashing is the cost of insertion. We show that the *expected* cost of inserting an element is constant, assuming that the insertion succeeds, i.e., there is no rebuilding the table by rehashing. Later we will show that we can bound the *amortized* cost of rebuilding.

Cuckoo graph

We need the concept of a *cuckoo graph* for the analysis. The cuckoo graph has m nodes corresponding to the m slots of the hash table and two slots a and b are connected by an edge if there is an item, say x, such that $h_1(x) = a$ and $h_2(x) = b$. The cuckoo graph is undirected; Figure 9.4 shows an example of the cuckoo graph (ignore the directions of the edges). Based on the cuckoo graph, we can make the following statement which is easy to show.

Lemma 9.2

When we insert an item, say x, in the hash table, then the insertion procedure will only visit slots that are connected to the slots $h_1(x)$ or $h_2(x)$ by a path in the cuckoo graph.

The above lemma means that if a node x is inserted in the hash table, then it will lead to insertion (by possibly another node) in a slot only if that slot can be reached via a path from either $h_1(x)$ or $h_2(x)$. The following key lemma bounds the probability that there exists a path between any two nodes (slots) in the cuckoo graph. It shows that if the load factor of the hash table is less than $1/2$, then the probability that there exists a path between any two slots in the cuckoo graph is small. In particular, it shows that the probability that there is a constant length path is $O(1/m)$ and the probability decreases exponentially as the path length increases.

For our analysis, as we assumed in Section 9.1.1.2, we assume that hash functions h_1 and h_2 are fully *random*, i.e., for any element x, $h_i(x)$ is equally likely to be any one of the m slots, independent of other elements. In other words, $\mathbb{P}(h_i(x) = j) = 1/m$, for any slot j, $1 \leqslant j \leqslant m$. As usual, we assume that the cost of computing a hash value is constant.

Lemma 9.3

We are given a set S of n elements and a hash table of size m; assume that the load factor $\alpha = n/m < 1/2$. Then, for any two slots a and b, the probability that there is a path of length $k \geqslant 1$ (and no shorter path) between a and b in the cuckoo graph is at most $\frac{(2\alpha)^k}{m}$.

Proof. We prove by induction on k. We first prove the base case, i.e., $k = 1$. For any two slots a and b, to be connected by an edge (i.e., path of length 1), at least for one of the n items (say x) it should be the case that $h_1(x) = a$ and $h_2(x) = b$ or vice versa. Assuming that h_1 and h_2 are fully random hash functions, the probability of this happening for a particular item is $2/m^2$. Hence, over all n items, the probability is bounded above by

$$\sum_{x \in S} \frac{2}{m^2} = \frac{2n}{m^2} \leqslant \frac{2\alpha}{m}$$

Next we consider the inductive step, where we bound the probability that there is a path of length $k > 1$ (but no shorter path) between any two slots a and b. This can happen only if there is some slot c, such that there is a path of length $k-1$ between a and c (event A) and an edge between c and b (event B). By induction hypothesis, the probability of event A is at most $\frac{(2\alpha)^{k-1}}{m}$. The probability of event B is bounded by $\frac{2\alpha}{m}$, since there are a set of at most n items that could have created an edge between these two slots (as computed in the base case). Hence the probability that both events happen is at most $\frac{(2\alpha)^{k-1}}{m} \times \frac{2\alpha}{m} = \frac{(2\alpha)^k}{m^2}$.[4] The above bound is for a particular choice of c. Summing over all m possible choices for c, the probability of having of path of length k (and no shorter path) is at most $\frac{m(2\alpha)^k}{m^2} = \frac{(2\alpha)^k}{m}$, as desired. □

Now we are ready to prove a bound for the cost of insertion.

Theorem 9.4

The expected cost of inserting an element, assuming that the insertion is successful (without rehashing), is constant, assuming that the load factor α is $< 1/2$.

Proof. Suppose we insert an element x in the hash table. We bound the probability that it may "collide" with another element y in the table. By "collide" we mean that one may lead to the eviction of the other. This will happen only if there is a path between slots $h_i(x)$ $(i = 1, 2)$ and $h_j(y)$ $(j = 1, 2)$. By Lemma 9.3, the probability that there is such a path of length k is at most $4\frac{(2\alpha)^k}{m}$ (we need a factor of 4, since there are two pairs of slots each for x and y). Hence the probability there is some path between x and another element y is bounded above by

$$\sum_{k=1}^{\infty} 4\frac{(2\alpha)^k}{m} \leqslant \frac{4}{m}\frac{2\alpha}{1-2\alpha} = O(1/m).$$

Now we finish the proof, by doing an analysis similar to that of hashing with chaining (Theorem 9.1). Since there are at most n elements, the expected number of elements that collide with a particular element x is at most $O(n/m)$, which follows from the above bound and linearity of expectation. Since the load factor is constant, this is bounded by $O(1)$.

Hence the expected time to insert an element is constant.

□

Cost of rebuilding the hash table

The above analysis of insertion cost assumes that the insertion succeeds. On the other hand, if the insertion fails, the whole hash table is rebuilt. This costs $O(n)$ time, assuming, as usual, that cost of hashing an item is constant. Since rebuilding is costly, we want to show that it does not occur often. More precisely, we would like to show that the expected *amortized* cost of rebuilding per insertion is constant. In other words, we can show that over the course of inserting $\Theta(n)$ elements, the expected number of rebuilds is $O(1)$. Since each rebuild takes $O(n)$ time, the amortized cost per insertion is constant.

[4] Note that we use the fact that $\mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B \mid A)$. Here, we have that $\mathbb{P}(B \mid A)$ is at most $\mathbb{P}(B)$, since the hash functions are fully random, and that there are at most n elements (may be less) under consideration for event B, since some elements could have already been used for establishing event A.

Note that rebuilding can occur only if there is a cycle in the cuckoo graph. Using an analysis similar to the proof of Lemma 9.3, we can show that over the course of $\Theta(n)$ insertions, the expected number of rebuilds is $O(1)$. We leave this as an exercise (Exercise 9.11).

Using more advanced techniques, one can show that the cost of inserting an item (assuming the insertion was successful) is $O(\log n)$ with high probability, assuming that the load factor is less than 1/2. Furthermore, one can show that the expected amortized cost of rebuilding per insertion is only $O(1/n)$.

9.2 Universal Hashing*

Universal hashing mitigates the problem of designing a hash function that works well for all subsets of the domain U. Although (as mentioned earlier), no one hash function is good for all subsets of U, choosing a random hash function from a family of hash functions allows us to avoid the worst case situation of having too many collisions. Thus randomizing the choice of hash function allows us to foil the adversary — there is no "worst-case" instance for a random function from the family. Of course, one should note that once an hash function is chosen, it is not changed henceforth. The point is that the adversary is oblivious to the random choice of the hash function and hence it is hard for it to come up with a "bad" set without this knowledge.

What property should the family of hash functions satisfy? This is captured in the following definition.

Definition 9.1 ▶ Universal family of hash functions

A family H of hash functions from U to M is **universal** if for all $x, y \in U$ such that $x \neq y$, and for a *randomly chosen* function h from H

$$\mathbb{P}(h(x) = h(y)) \leqslant \frac{1}{m}.$$

There are two questions that arise: (1) Why is the above property useful (or what does it buy us)? and (2) Can one design a family of functions that satisfy the above property which says that the collision probability between any two distinct elements is small, i.e., at most $1/m$. We answer both these questions next. The next theorem answers the first question.

Theorem 9.5

Assume that we hash n keys to a table of size m, $n \leqslant m$, using a hash function h chosen at random from a universal family of hash functions, and we resolve collisions by chaining. Then searching for a key takes expected time $\mathcal{O}(\alpha)$. Similarly, INSERT and DELETE take expected time $\mathcal{O}(\alpha)$.

Proof. For every pair of distinct keys k and l, define the indicator random variable (r.v.) $X_{kl} = 1$ if and only if $h(k) = h(l)$, else $X_{kl} = 0$. Then $\mathbb{E}[X_{kl}] = \mathbb{P}(X_{kl} = 1) \leqslant 1/m$.

For each key k, define the r.v. Y_k that equals the number of keys *other than* k that hash to the same slot as k in the table T: $Y_k = \sum_{l \in T, l \neq k} X_{kl}$. Thus

$$\mathbb{E}[Y_k] = \sum_{l \in T, l \neq k} \mathbb{E}[X_{kl}] \leqslant \sum_{l \in T, l \neq k} 1/m$$

We have two cases:

(1) If key $k \notin T$: $\mathbb{E}[Y_k] \leqslant n/m = \alpha$.
(2) If key $k \in T$: $\mathbb{E}[Y_k] \leqslant (n-1)/m < \alpha$.
Thus search takes $\mathcal{O}(\alpha)$ time. □

Corollary 9.6

Using universal hashing and collision resolution by chaining in a table with m slots, it takes expected time $\mathcal{O}(s)$ to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $\mathcal{O}(m)$ INSERT operations.

Proof. There are at most $\mathcal{O}(m)$ insert operations, and hence at most so many elements. Thus the load factor $\alpha = \mathcal{O}(1)$. Thus each operation takes $\mathcal{O}(1)$ expected time. Using linearity of expectation, s operations take $\mathcal{O}(s)$ time. □

9.2.1 Constructing universal hash functions

It is clear that universal families exists because we can show that the set of all functions from U to M is universal (this is left as an exercise). Let $|U| = u$ and $|M| = m$. Then, there are m^u functions from U to M — requires $u \log m$ bits to choose, represent and store as a table — which is as large as the size of the universe.

We next show how to construct a universal family of hash functions that is of small size.

Choose a prime number p such that $0 \leqslant k \leqslant p-1$, i.e., p is larger than the largest key in U. Define the following. Let $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ and $\mathbb{Z}^*{}_p = \{1, 2, \ldots, p-1\}$.

Note that since p is a *prime number* $\mathbb{Z}_p$ and $\mathbb{Z}^*{}_p$ are groups — the first being the *additive group modulo* p and the second is the *multiplicative group modulo* p (See Appendix D). These two groups simply mean that in the former the group operation is addition and the latter it is multiplication. Since these are groups we can essentially do arithmetic like that we are used to when doing arithmetic using integers. The only difference being that this group is finite — arithmetic operations are done modulo p. That is, we can add and subtract (in $\mathbb{Z}_p$) and multiply and divide (in $\mathbb{Z}^*{}_p$). The fact that these are groups, in $\mathbb{Z}_p$ every element has an (unique) additive inverse (and hence subtraction can be done) and in $\mathbb{Z}^*{}_p$ every element has a (unique) multiplicative inverse (and hence division which is equivalent to multiplying by the inverse is possible).

Exercise 9.2. Show that the set of all functions from U to M is a universal hash family.

A universal family of hash functions

Definition 9.2 ▶ $\mathcal{H}_{p,m}$ family of hash functions

Let

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

be a hash function for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$. Then we define

$$\mathcal{H}_{p,m} = \left\{ h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p \right\}$$

We will show next that the above family is universal. We note that the above family can be efficiently stored, evaluated, and chosen in $\mathcal{O}(\log p)$ space and time. Indeed, choosing a random hash function is easy from this set: simply choose a and b uniformly at random from their respective ranges. Note that $\mathcal{O}(\log p) = \mathcal{O}(\log u)$ if p is chosen such that $u < p < 2u$ (A prime always exists between any number n and $2n$ — this is known as *Bertrand's postulate*).

Theorem 9.7

$\mathcal{H}_{p,m}$ is universal.

Proof. We show that $\mathcal{H}_{p,m}$ satisfies the definition of universal hash function. That is given any two distinct elements x and y, the probability that they collide on randomly chosen hash function from the family is at most $1/m$.

Consider two distinct keys k and ℓ from $\mathbb{Z}_p$. For a given hash function $h_{a,b}$ we let

$$r = (ak + b) \bmod p$$
$$s = (a\ell + b) \bmod p$$

$r \neq s$ because

$$r - s = a(k - \ell) \bmod p$$

and both a and $k - \ell$ are nonzero $(\text{mod} p)$. Thus there is no collision at the "mod p" level if the two keys are distinct.

Moreover, each distinct pair (a, b) with $a \neq 0$ yields a distinct pair (r, s) with $r \neq s$ because we can solve for a and b uniquely given r and s:

$$a = (r - s)((k - \ell)^{-1} \bmod p) \bmod p$$

and

$$b = (r - ak) \bmod p$$

Thus, for given pair of distinct keys k and ℓ, if we pick (a, b) randomly then (r, s) is also picked randomly. Thus the probability that keys k and ℓ collide is equal to the probability that $r = s \bmod m$ with $r \neq s$. For a given r, the number of values of s such that $r \neq s$ and $r = s \bmod m$ is at most $\lceil p/m \rceil - 1$. The reason for this is that there are at most $\lceil p/m \rceil$ values between 0 and $p - 1$ that leave the same remainder when divided by m. We subtract 1 since we are looking for such values that does not include r itself. Now we have,

$$\left\lceil \frac{p}{m} \right\rceil - 1 \leqslant \frac{p + m - 1}{m} - 1 = \frac{p - 1}{m}$$

(We use the fact that $\lceil a/b \rceil \leqslant (a + b - 1)/b$).

Thus probability that s collides with r is $\leqslant 1/m$. □

9.3 Applications of Universal Hashing*

Universal hashing has many algorithmic applications. Here we discuss two different applications: (1) *perfect hashing* which implements a static dictionary in constant worst-case search time; and (2) *Heavy hitters algorithm* which is a *streaming* algorithm for finding most frequent items in a data stream.

The two key advantages of universal hashing that are useful in many applications is that: (1) universal hashing is a usually good substitute of “fully random” hash functions which involve use of substantially more randomness (random bits) and hence computationally more expensive and take more space; (2) universal hash functions are simple, and are fast and easy to compute, and still give provable performance guarantees.

9.3.1 Perfect Hashing

By perfect hashing we mean that the **worst case** time to search is $\mathcal{O}(1)$. We will design a hashing scheme that gives a worst-case search time of $\mathcal{O}(1)$ and takes *linear* space, assuming that the set of keys are static. That is, the keys are inserted once into the hash table in such a way that the time to search for any key is constant.

A two-level hashing scheme

Clearly, any one hash function chosen randomly from a universal family will cause collision with non-zero probability. Note that if the size of the hash table m is at least the number of elements (denoted by set S) in the dictionary, then there exists an hash function (among all possible hash functions from U to S) that is perfect for this set of elements (Exercise 9.1). However, such a function may not exist in $\mathcal{H}_{p,m}$, which is a much smaller set of functions. Perfect hashing gives an efficient scheme to choose a small *set* of hash functions from the universal family such that they cause no collisions, while still using *linear* space.

The approach is to use the following two-level hashing scheme. We will use hashing with chaining. For elements that hash to the same slot in the first level, we will use a secondary hash table that will further hash these elements to another table. To search for an element, use the first level hash function to find the slot in the primary hash table and then use another hash function (corresponding to this slot) to hash again to find the slot in the secondary hash table.

1. Use universal hashing (from the class $\mathcal{H}_{p,m}$) to hash a given set of n keys to a table of $m = n$ slots.

2. For $0 \leqslant j \leqslant m - 1$:
   let n_j be the number of elements hashed to slot j;
   use universal hashing again (from the class $\mathcal{H}_{p,n_j^2}$) to hash these in a secondary table of size n_j^2.

We show the following theorem.

Theorem 9.8

There exists a two-level hashing scheme that uses $\mathcal{O}(n)$ space (for a set of n keys) and guarantees a worst case search time of $\mathcal{O}(1)$ assuming that the set of keys is **static** (i.e., fixed in advance).

To prove the theorem we need the following lemma which shows the main idea behind the scheme, namely, if we use n^2 slots for n keys, then there exists a hash function which does not cause any collisions.

Lemma 9.9

If we store k keys in a hash table of size k^2 using universal hashing (from the class $\mathcal{H}_{p,k^2}$), then the probability of there being any collision is $< 1/2$. Thus, there **exists** an hash function that that will not produce any collision. Moreover, such a function can be found in expected constant number of trials.

Proof. Let random variable X denote the number of collisions. Then we can write $X = \sum_{i \neq j} X_{ij}$, where X_{ij} is the indicator r.v. for the event that two *distinct* keys i and j collide. In other $X_{ij} = 1$ if they collide, otherwise 0. By the property of universal hashing, $\mathbb{E}[X_{ij}] = \mathbb{P}(X_{ij} = 1) \leqslant 1/k^2$, since the number of slots in the hash table $m = k^2$. Then, by linearity of expectation:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i \neq j} X_{ij}\right] = \sum_{i \neq j} \mathbb{E}[X_{ij}] = \binom{k}{2} 1/k^2 < 1/2$$

Using Markov's inequality (see Appendix C), we have

$$\mathbb{P}(X \geqslant 1) \leqslant \mathbb{E}[X] < 1/2.$$

Thus if we choose a random hash function from the family, with probability at least 1/2, there will be no collision. We hash all the elements and check and see if there is any collision. If there is a collision, we try again with another random hash function choice. Note that the probability of "success" (i.e., getting no collision) is at least 1/2. Thus we have a geometric distribution (see Appendix A) with probability of success at least 1/2. Thus the expected number of times we repeat before we get success (i.e., find a hash function that has no collisions) is at most 2. □

While the above lemma shows that if we use k^2 slots for k keys then there is choice of a hash function that causes no collision. However, k^2 is quadratic in the number of keys; we desire $\mathcal{O}(k)$ space. This is the reason for using a secondary level hash function. The first level hash (using a randomly chosen hash function from the family $\mathcal{H}_{p,m}$) reduces the number of collision per slot and then we use the "quadratic" blow up on the number of elements hashing to the same slot to eliminate collisions. Let n_j be the number of elements hashing to slot j in the primary table. Then the amount of space in the secondary table is n_j^2. We bound the total space as follows.

Lemma 9.10

In the two-level hashing scheme the expected amount of storage required for all the secondary hash tables is (assuming $m = n$):

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

Proof. Using the identity, $a^2 = a + 2\binom{a}{2}$:

$$\begin{aligned}
\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &= \mathbb{E}\left[\sum_{j=0}^{m-1} (n_j + 2\binom{n_j}{2})\right] \\
&= \mathbb{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathbb{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \\
&\leqslant n + 2\binom{n}{2}1/m = n + 2n(n-1)/2n < 2n
\end{aligned}$$

Note that $\sum_{j=0}^{m-1} n_j$ is simply the sum of all elements which is n and $\sum_{j=0}^{m-1} \binom{n_j}{2}$ is the same as the total number of (pairwise) collisions. Hence, $\mathbb{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = \binom{n}{2}1/m$, by the property of universal hashing and linearity of expectation. □

Thus, if we choose universal hashing in the secondary level (i.e., for a random choice from the class $\mathcal{H}_{p,n_j^2}$, for each primary slot j), we expect to use no more than $2n$ space. Using Markov's inequality gives the following corollary.

Corollary 9.11

In the two-level hashing scheme the probability that the total storage used for secondary hash tables exceeds $4n$ is $< 1/2$. Hence the expected number of trials needed to have at most $4n$ secondary space is constant.

Thus to get perfect hashing, we first try a few (constant on average) choices of the primary hash function to get at most $4n$ secondary table space. Then, we hash all the elements that hash to the same primary slot using the quadratic blow up idea — we need to repeat only a constant number of times until we find a secondary hash function that causes no collision.

9.3.2 Heavy Hitters: Tracking Frequent Items in Data Streams

In a data stream setting, one deals with a massive amount of "streaming" data, where a large amount of data has to be processed in a short amount of time with, typically, small amount of memory (due to limitations of the processing device). The data items is assumed to arrive continuously (one by one) over time and the goal for a "streaming" algorithm is to compute some function on the data (e.g., the most frequent items, or other statistics) without storing all the data items (since there is very limited storage space available). For example, consider a sensor device which has only a limited computational power and very limited memory. The sensor is continuously recording various events over time which occur one after the other. For example, the sensor might be recording the frequencies of different kinds of particles in an experiment. The number of different items (particles) are very large and the goal is to maintain the number of times (i.e., the frequency) of occurrence of the most frequent items — these are called the *heavy hitters* — at any point in time. That is, at any time T (from the start of the experiment), we need to maintain a list of the heavy hitters (where the frequencies are counted until time T). (Assume one item arrives in each time step.) To specify what qualifies as a heavy hitter, we define a *threshold* r such that any item that occurs more than r times is defined as a heavy hitter. The goal for the sensor to output the list of heavy hitters at any (required) time T.

Of course, if we record the frequency of *every* item, then this will solve our problem. But a sensor has only a small amount memory (say, just a constant) and hence cannot afford to store the frequency of every item, as there a large number of items. How to use only constant memory to efficiently and accurately count the heavy hitters?

Description of the Data Structure

The data structure uses universal hashing to hash items to a (small) hash table which stores the frequencies of the hashed items. Each entry of the table will be a *counter* and will store the counts of items hashed to it. Since the table should be small, we cannot afford to store each item in the table with a small number of collisions. The key idea of count-min filter is to use multiple hash functions (each chosen independently and uniformly at random from a universal family) to separately hash the item in different tables.[5] As explained below, this allows accurate counting with good approximation of heavy hitting items.

A count-min filter has m counters and uses k hash functions chosen randomly from a family of universal hash functions. The hash functions are H_j, $1 \leqslant j \leqslant k$, where each H_j is chosen independently and uniformly at random from a 2-universal family $\mathcal{H}_{p,m/k}$, i.e., range of the hash functions is $[0, m/k - 1]$. The m counters are further split into k groups each of size m/k (we assume that k divides m evenly). So we can think of the counters being arranged in a 2-dimensional table with m/k rows and k columns.

Hashing items to counters

Initially all counters (hash table entries) are set to 0. We label the counters in the 2-dimensional table by $C(\ell, j)$, $0 \leqslant \ell \leqslant m/k - 1$ and $1 \leqslant j \leqslant k$. See Figure 9.5. When an item i_t arrives at time t in the data stream (we assume one item per time step), it is hashed by each H_j, for $1 \leqslant j \leqslant k$, i.e., $H_j(i_t)$ is computed for each j, and the value of the counter $C(H_j(i_t), j)$ is incremented by 1.[6] Thus each item is hashed to k counters. Let $C^T(\ell, j)$ be the value of the counter $C(\ell, j)$ after processing the first T items. We take the *smallest* counter (among the k counters) associated with the item as the (approximate) frequency of the item. We next prove that this smallest counter will always be an *upper bound* on the correct count (frequency) of the item and will be not more than ϵ times T (T is the total count of all items seen until T, since we assume one item per time step), where ϵ is a user-chosen accuracy parameter (can be chosen to be small).

Main idea behind the Count-min filter

The main idea behind the count-min filter is to store the count of an item using k (for a suitably small k) different randomly chosen hash functions. That is we hash an element to k different counters using the respective hash functions. Since the hash functions have

[5] This idea is similar to the *Bloom filter* data structure (Exercise 9.8) which answers membership queries using a small amount of space. In the Bloom filter exercise, we assumed fully-random random hash functions, whereas here we assume 2-universal hashing which is efficient to implement — it needs only small number of random bits and hence small storage.

[6] Here we are assuming that we simply count the number of times an item occurs. But, in general, we could also count any weighted value associated with the item and increment the counter by that weight.

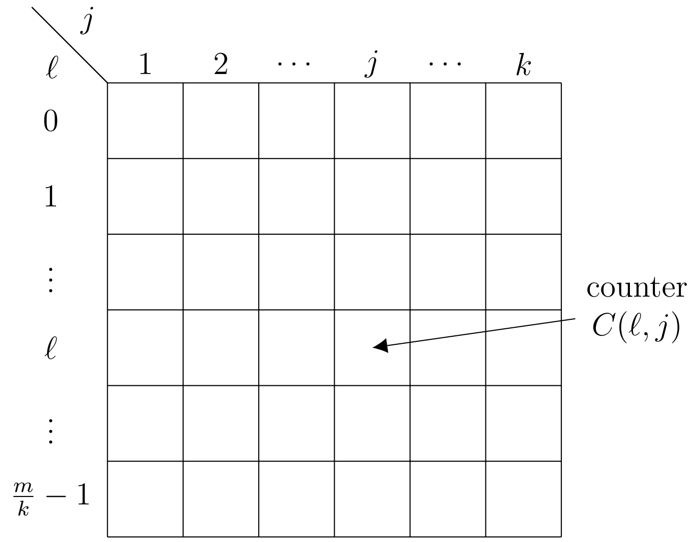


Figure 9.5: The 2-dimensional counter array. There is a set of m/k counters for each of the k hash functions. Each item is hashed by each of the k hash functions and inserted into the respective, appropriate counters.

bounds on collision probabilities, it is likely that two different items may not collide for all k hash functions. Hence one can take the counter that has the *smallest* value among all the k hash functions; this is likely to have a count value that is close to the true frequency of the item.

We next formalize the above idea by the following theorem.

Theorem 9.12

For any item g, and for any sequence of T items $i_1, i_2, \ldots, i_T$, let C_g be the minimum value among all the counters that g hashes to, i.e., $C_g = \min_{a=H_j(g), 1 \leqslant j \leqslant k} C(a, j)$. Then

$$C_g \geqslant \text{Freq}(g, T)$$

where $\text{Freq}(g, T)$ is the total frequency (count) of item g up to time step T. Furthermore, with probability $1 - (\frac{k}{m\epsilon})^k$ over the choice of the randomly chosen hash functions,

$$C_g \leqslant \text{Freq}(g, T) + \epsilon T$$

Proof. The lower bound of C_g is easy since each counter associated with any item g is incremented whenever item g appears in the sequence. Hence, the value of each counter, including the counter with minimum value, is at least $Freq(g, T)$.

For the upper bound of C_g, consider a particular item g at a particular time step T. Consider a specific counter, say the counter of the first hash function: $C(H_1(g), 1)$. This counter is at least $Freq(g, T)$ at time T, since every time item g occurs this counter is incremented. We want to bound the counter increments due to collisions by other items. Let r.v. Z_1 denote the total counter increments due to elements other than g. Define r.v. $X_t = 1$ if $i_t \neq g$ and $H_1(i_t) = H_1(g)$; otherwise, $X_t = 0$. Then $Z_1 = \sum_{i=1}^{T} X_t$.

Since we choose H_1 randomly from a 2-universal hash family of functions with range

$[0, m/k - 1]$, for any $i_t \neq g$, we have

$$\mathbb{P}(H_1(i_t) = H_1(g)) \leqslant \frac{k}{m}$$

Thus, we have $\mathbb{E}[X_t] \leqslant \frac{k}{m}$ and hence,

$$\mathbb{E}[Z_1] = \mathbb{E}\left[\sum_{i=1}^{T} X_t\right] = \sum_{i=1}^{T} \mathbb{E}[X_t] \leqslant \frac{k}{m} T$$

Using Markov's inequality (Appendix C.8),

$$\mathbb{P}(Z_1 \geqslant \epsilon T) \leqslant \frac{k/m}{\epsilon} = \frac{k}{m\epsilon}$$

Let $Z_2, Z_3, \ldots, Z_k$ be the corresponding random variables for each of the other hash functions. Applying the above argument to each of the above random variables, we have that all the Z_is satisfy the above probabilistic bound. We note that the Z_is are independent, since the hash functions are chosen independently and randomly. Hence

$$\mathbb{P}\left(\min_{1 \leqslant j \leqslant k} Z_j \geqslant \epsilon T\right) = \prod_{1 \leqslant j \leqslant k} \mathbb{P}(Z_j \geqslant \epsilon T) \leqslant (\frac{k}{m\epsilon})^k$$

This completes the proof of the upper bound.

□

Suppose we want to use the count-min filter to guarantee, for a given threshold r, an accuracy of ϵT, i.e., any item that has frequency at most $r - \epsilon T$ is not listed as heavy hitter with probability at least $1 - \delta$, for some user specified parameters ϵ and δ. (Note that all heavy hitters are always listed.) For example, one can take $\epsilon = \delta = 0.01$. Then, from Theorem 9.12, it follows if we choose $k = \left\lceil \ln \frac{1}{\delta} \right\rceil$ and $m = \left\lceil \ln \frac{1}{\delta} \right\rceil \left\lceil \frac{e}{\epsilon} \right\rceil$, then the probability that no item with frequency at most $r - \epsilon T$ is listed as heavy hitter is at most

$$\left(\frac{k}{m\epsilon}\right)^k \leqslant e^{-\ln(1/\delta)} = \delta$$

Thus the count-min filter uses only $\mathcal{O}\left(\frac{1}{\epsilon} \ln(\frac{1}{\delta})\right)$ counters and only $\mathcal{O}\left(\ln \frac{1}{\delta})\right)$ hash functions (and hence only so many computations) to process each item. (Note that if ϵ and δ are constants, so are k and m.)

Regarding outputting the heavy hitters, we can use additional data structures to efficiently store the heavy hitters at any point. For example, if after processing an item g at some time step, we find it is a heavy hitter (since its minimum counter value exceeds the threshold r), then we can store in a dictionary data structure (say, a hash table or binary search tree). Typically, the number of heavy hitters are not too large, and hence this will not take too much space.

9.4 Hashing Strings and Fingerprinting

In many applications, we want to hash strings, e.g., storing words in an English dictionary or hashing a name to a database. The idea is simple: convert the string into a

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
000	000	Null	032	020	Space	064	040	@	096	060	‘
001	001	Start of Heading	033	021	!	065	041	A	097	061	a
002	002	Start of Text	034	022	"	066	042	B	098	062	b
003	003	End of Text	035	023	#	067	043	C	099	063	c
004	004	End of Transmission	036	024	$	068	044	D	100	064	d
005	005	Enquiry	037	025	%	069	045	E	101	065	e
006	006	Acknowledgement	038	026	&	070	046	F	102	066	f
007	007	Bell	039	027	’	071	047	G	103	067	g
008	008	Backspace	040	028	(	072	048	H	104	068	h
009	009	Horizontal Tab	041	029	)	073	049	I	105	069	i
010	00a	Line Feed	042	02a	*	074	04a	J	106	06a	j
011	00b	Vertical Tab	043	02b	+	075	04b	K	107	06b	k
012	00c	Form Feed	044	02c	,	076	04c	L	108	06c	l
013	00d	Carriage Return	045	02d	-	077	04d	M	109	06d	m
014	00e	Shift Out	046	02e	.	078	04e	N	110	06e	n
015	00f	Shift In	047	02f	/	079	04f	O	111	06f	o
016	010	Data Link Escape	048	030	0	080	050	P	112	070	p
017	011	Device Control 1	049	031	1	081	051	Q	113	071	q
018	012	Device Control 2	050	032	2	082	052	R	114	072	r
019	013	Device Control 3	051	033	3	083	053	S	115	073	s
020	014	Device Control 4	052	034	4	084	054	T	116	074	t
021	015	Negative	053	035	5	085	055	U	117	075	u
022	016	Synchronous Idle	054	036	6	086	056	V	118	076	v
023	017	End of Trans. Block	055	037	7	087	057	W	119	077	w
024	018	Cancel	056	038	8	088	058	X	120	078	x
025	019	End of Medium	057	039	9	089	059	Y	121	079	y
026	01a	Substitute	058	03a	:	090	05a	Z	122	07a	z
027	01b	Escape	059	03b	;	091	05b	[	123	07b	{
028	01c	File Separator	060	03c	<	092	05c	\	124	07c	|
029	01d	Group Separator	061	03d	=	093	05d	]	125	07d	}
030	01e	Record Separator	062	03e	>	094	05e	^	126	07e	~
031	01f	Unit Separator	063	03f	?	095	05f	_	127	07f	-

Figure 9.6: ASCII code for characters.

number and then hash the number. Let us associate a number with each character, e.g.: $a \to 1$, $b \to 2$, $c \to 3$, $\ldots$, $z \to 26$. A more convenient way is to associate the *ASCII* value of the character. Characters are represented using the American Standard Code for Information Interchange (ASCII). Each character is represented using a number between 0 and 127 (8 bits). Figure 9.6 shows the ASCII table.

Hashing a string

Given a string $x = x_{n-1}x_{n-2}\ldots x_0$ consisting of the characters $x_0, x_1, \ldots, x_{n-1}$, define a hash function of x:

$$h_p(x) = \left(\sum_{i=0}^{n-1} a(x_i)r^i\right) \bmod p$$

where $a(x_i)$ is the ascii value of character x_i, $r > 1$ is a radix, and p is a (suitably chosen) prime. We will call $h_p(x)$ as the *"fingerprint"* of x.

Since there are at most 128 characters, we can choose $r = 128$ — then we can interpret the number $\sum_{i=0}^{n} a(x_i)r^i$ as equivalent to the string represented in base 128. But as we will see later, we can choose r *to be other values* as well, e.g., we can choose $r = 2$ as well. p should be chosen larger than 128 and also somewhat larger than n — say, $100n$, where

n is the number of characters in the string. We will see shortly why such a p is a good choice.

Example

Let us hash the string $x = \mathsf{ALGO}$. In ASCII, $A = 65$, $L = 76$, $G = 71$, $O = 79$. Let us choose $r = 128$ and $p = 131$.

$$h(x) = \Big(65(128)^3 + 76(128)^2 + 71(128) + 79\Big) \bmod 131 = 137569231 \bmod 131 = 105$$

Computing the above sum involves heavy calculation, as the sum becomes large. Can we do this better?

Mod properties

We use the following properties of mod function in conjunction with Horner's rule (see below) to reduce the sizes of the intermediate sums.

- *sum rule*: $(a+b) \bmod p = (a \bmod p + b \bmod\; p) \bmod p$.
- *product rule*: $(a \times b) \bmod p = (a \bmod p \times b \bmod p) \bmod p$.
- *exponentiation rule*: $a^y \bmod p = (a \bmod p)^y \bmod p$. The exponentiation rule is very useful in avoiding large integers.

Horner's rule

Suppose we want to calculate:

$$h(x) = \Big(65(128)^3 + 76(128)^2 + 71(128) + 79\Big) \bmod 131$$

Rewrite the sum as follows — called *Horner's rule*:

$$\Big(65(128)^3 + 76(128)^2 + 71(128) + 79\Big) = 128(\mathit{128}\ (128\ (65)\ +\ 76)\ +\ \mathit{71}) + 79$$

We compute the above sum by repeated modding — mod using the product and sum rule repeatedly.

In the above example, we proceed as:

$$\begin{aligned}
(128 \times 65) \bmod 131 &= 67\\
(67 + 76) \bmod 131 &= 12\\
128\mathit{(12)} \bmod 131 &= 95\\
95 + 71 \bmod 131 &= 35\\
128\mathit{(35)} \bmod 131 &= 26\\
26 + 79 = 105 \bmod 131 &= 105
\end{aligned}$$

Thus we get the final answer is 105, but more importantly, all intermediate sums in the computation are less than $(128)(130) = 16640$.

The pseudocode for computing hash function using Horner's rule is shown in Algorithm 41. From the pseudocode, it is clear that the running time is $\mathcal{O}(n)$, where n is the *length* of the string.

Algorithm 41 Fingerprint – $\textsc{Ord}(c)$ is the ASCII value of c, p is prime, and r is a radix.
Input: A string $x = x_{n-1}, x_{n-2}, \ldots, x_0$
Output: Fingerprint Checksum $h_p(x) = \sum_{i=0}^{n-1} \textsc{Ord}(x_i) r^i \bmod p$

```
func FINGERPRINT($x$):
    sum = 0
    for $i = n - 1$ downto 0:
        sum = $(r \cdot$ sum $\bmod p + \textsc{Ord}(x_i)) \bmod p$
    return sum
```

9.4.1 Problem: Comparing Strings

Problem 9.1

Given two strings $a = a_{n-1}, \ldots, a_0$ and $b = b_{n-1}, \ldots, b_0$ we want to check for equality, i.e., whether the two strings are the same or not.

The straightforward way to solve the above problem is to just compare the two strings character by character, i.e., if $a_i = b_i$, for all $0 \leqslant i \leqslant n - 1$. This takes $\mathcal{O}(n)$ character comparisons. A better way is to use algorithm Fingerprint to solve this. We compare the fingerprint of the two strings:

- If they are *not* the same, then output "No".
- If they are the same, then output "Yes".

Why is this better? Is this always correct? We answer these two questions next.

Why is this better?

Fingerprint is *much smaller* than the string itself. Suppose we are trying to check the equality of two long strings, e.g., DNA sequences (which can have billions of characters). Assume that one sequence is in Singapore and the other in USA. The naive way is to send *all* n characters of the string from Singapore to the USA. However, we can send just the fingerprint of the string in Singapore to the USA and compare it with the fingerprint of the string in the USA. For this, only $\mathcal{O}(\log p)$ bits need to be communicated, where p is the prime chosen in the algorithm, since the fingerprint is less than p and thus can be represented in $\mathcal{O}(\log p)$ bits. When p is chosen to be $\Theta(n)$, only $\mathcal{O}(\log n)$ bits need to the communicated! Thus equality of two n length strings can be checked by comparing only $\mathcal{O}(\log n)$ bits.

Correctness

There is a chance that fingerprint gives the wrong answer. Compare the fingerprint of the two strings:

- If they are *not* the same, then output "No". *No error here.*
- If they are the same, then output "Yes". *Error is possible — false positive.*

The false positive case occurs when there is a *collision*: $h_p(x) = h_p(y)$, for $x \neq y$. We will show that the probability that this happens is *small*, if p and r are chosen appropriately in the fingerprint function.

9.4.2 Fingerprinting Theorem

Theorem 9.13

Let the length of the string be n. Let p be a prime number that is larger than 127 (the largest ascii value) and also larger than λn, where λ is some parameter (which determines the probability of error). Let r be the radix chosen uniformly at random in $[0, p-1]$. Then the probability of having a collision, i.e., for any two *different* strings x and y (each of length n),

$$\mathbb{P}(h_p(x) = h_p(y)) = n/p \leqslant 1/\lambda.$$

Proof. Assume that $x \neq y$, but $h_p(x) = h_p(y)$, i.e.,

$$\left(\sum_{i=0}^{n-1} a(x_i)r^i\right) \bmod p = \left(\sum_{i=0}^{n-1} a(y_i)r^i\right) \bmod p$$

That is, in terms of congruence:

$$\sum_{i=0}^{n-1} a(x_i)r^i \equiv \sum_{i=0}^{n-1} a(y_i)r^i \bmod p$$

Thus we can write a congruence equation (consider r as the unknown variable):

$$\sum_{i=0}^{n-1}(a(x_i)r^i) - \sum_{i=0}^{n-1} a(y_i)r^i = \sum_{i=0}^{n-1}(a(x_i) - a(y_i))r^i \equiv 0 \bmod p$$

Note that r and the coefficients of r are in $[0, p-1]$ (and not all coefficients are zero, since $x \neq y$). Thus the above is a polynomial congruence (mod p) equation of degree (at most) $n-1$. *Lagrange's Theorem* says that it has at most $n-1$ solutions, i.e., only so many choices of r for which the equation is satisfied. If we choose r randomly in $[0, p-1]$, then the probability that one of the above choices is chosen is at most n/p. □

9.4.3 Application to Pattern Matching

Problem 9.2

We are given a long text string $X = x_1, \ldots, x_n$ and a shorter pattern string $Y = y_1, \ldots, y_m$. The **pattern matching** problem is to find an occurrence of Y in X or return `Null` if no such occurrence exists.

Pattern matching is a fundamental problem with lots of applications. For example, when you search for a word in a webpage or try to do a "find" in your editor, you are solving the pattern matching problem. There is an easy algorithm that takes $\mathcal{O}(mn)$ time: simply compare the pattern string with the text string starting at every position i, $1 \leqslant i \leqslant n-m$. We design a simple algorithm using fingerprinting that takes $\mathcal{O}(m+n)$ time with high probability.

An $\mathcal{O}(m+n)$ time algorithm

We are given as *input:* $X = x_1, \ldots, x_n$ and $Y = y_1, \ldots, y_m$. The goal is to *output:* a position in X where Y occurs, or output `Null`.

Let $X(i) = x_i, \ldots, x_{i+m-1}$, i.e., $X(i)$ denotes the substring of length m starting at position i.

The idea is as follows:

- Compare the fingerprint of the substrings $X(i)$ and Y at every i, $1 \leqslant i \leqslant m$.
- Fingerprint of $X(i+1)$ can be computed from fingerprint of $X(i)$ in *constant* time as follows. The fingerprint of $X(i)$ is:
$$h_p(X(i)) = \left(a(x_i)r^{m-1} + a(x_{i+1})r^{m-2} + \cdots + a(x_{i+m-1})\right) \bmod p$$
The fingerprint of $X(i+1)$ can be computed from $X(i)$ as follows:
$$h_p(X(i+1)) = \left(a(x_{i+1})r^{m-1} + \cdots + a(x_{i+m-1})r + a(x_{i+m})\right) \bmod p$$
Thus,
$$h_p(X(i+1)) = \left((h_p(X(i)) - a(x_i)r^{m-1})r + a(x_{i+m})\right) \bmod p$$
We can compute $h_p(X(i+1))$ in an "incremental" fashion using the value of $h_p(X(i))$ without the need to compute from scratch.

The pseudocode of Algorithm Match is given in Algorithm 42. This algorithm is called *Rabin-Karp*'s algorithm. Algorithm Match works even when the entirety of X does not fit in memory or when X is given in an "online" fashion. This is because we need to see only m consecutive positions of X at any given time. It is best to compute r^{m-1} using the exponentiation rule and *repeated squaring* to avoid large integers.

Algorithm 42 Match
Input: Sequences X and Y of lengths n and m
Output: An occurrence of Y in X

```
func Match($X$, $Y$):
▷ $p$ is a prime greater than $mn^2$
    $i = 1$
    while $i \leqslant n - m$:
        if $h_p(Y) \neq h_p(X(i))$:
            $h_p(X(i+1)) = ((h_p(X(i)) - a(x_{i+1})r^{m-1})r + a(x_{i+m})) \bmod p$
        else if $Y == X(i)$:
            return $i$
        $i$ += 1
    return Null
```

Analysis of Algorithm Match

Theorem 9.14

Algorithm Match runs in $\mathcal{O}(m+n)$ time and outputs the correct answer with high probability.

Proof. We first note that, assuming *no false positives*, the algorithm terminates in $\mathcal{O}(m+n)$ time. We next upper bound the probability of getting a false positive.

For a given i, assuming $Y \neq X(i)$:

$$\mathbb{P}(h_p(Y) = h_p(X(i))) \leqslant \frac{m}{mn^2} = \frac{1}{n^2}$$

by our choice of p and the fingerprinting theorem. By the union bound (Appendix A), for every $1 \leqslant i \leqslant n-m+1$, assuming that $Y \neq X(i)$,

$$\begin{aligned}\mathbb{P}(\cup_i(h_p(Y) = h_p(X(i)))) &\leqslant \mathbb{P}(h_p(Y) = h_p(X_1)) + \cdots + \mathbb{P}(h_p(Y) = h_p(X_{n-m+1}))\\ &\leqslant \frac{1}{n^2} + \cdots + \frac{1}{n^2}\\ &= (n-m+1)\frac{1}{n^2}\\ &\leqslant 1/n\end{aligned}$$

Thus, if there is a false positive, then that happens with probability $\leqslant 1/n$. Thus the algorithm Match runs in time $\mathcal{O}(m+n)$ with probability at least $1-1/n$ (high probability). □

9.5 Worked Exercises

Worked Exercise 9.1. Assume that have a hash function that hashes keys to uniformly random locations. That is each key is equally like to hash to any particular slot in the hash table. Assume that the hash can be computed in $\mathcal{O}(1)$ time. In the hash table collisions are resolved by chaining. There are n keys and the size of the hash table is m.

After hashing all the keys into the table, we search for the keys. Assuming that each of the n keys are equally likely to be searched, show that any search takes $\Theta(1+\alpha)$ expected time, where $\alpha = n/m$ is the load factor.

Solution. As mentioned, we assume that all keys are equally likely to be searched and compute the average time to search a key.

Assume that a key is inserted at the head of the link list.

Let k_i denote the ith key inserted into the table. For keys k_i and k_j, we define the indicator r.v. X_{ij} to indicate whether $h(k_i) = h(k_j)$.

$$\mathbb{E}[X_{ij}] = 1/m$$

The expected number of elements examined is:

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

The reason for the above is that the term $1 + \sum_{j=i+1}^{n} X_{ij}$ computes the number of comparisons needed to find k_i, ith inserted element: it is equal to 1 plus the number of elements that are inserted subsequently in the same slot. We then take the sum of over all n keys and divide by $1/n$ since the probability of searching for any key is uniform.

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}]\right)\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} 1/m\right)\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}(n^2-n(n+1)/2)\\
&= 1+\frac{n-1}{2m}\\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}$$

Thus total time needed is $\Theta(1+\alpha)$. □

9.6 Exercises

Exercise 9.3. Consider the following simple scheme for adaptively maintaining the load factor of a hash table to be between 1/2 and 1 at all times. If the number of elements in the table equals the number of slots, then the table size is doubled and if the number of elements becomes less than half the number of slots, then the table size is decreased by a factor of two. Elements are rehashed appropriately in the new table and, as usual, we assume that the cost of rehashing n elements is $\Theta(n)$.

Show that the amortized cost of rebuilding the table per insertion or deletion is constant.

Exercise 9.4. Suppose we want to output *all* occurrences of pattern Y in text X. Consider the Algorithm Match of Algorithm 42 and instead of stopping at the first occurrence, we continue and output all occurrences. Suppose the pattern Y occurs in the text X in many locations, say in $\Theta(n)$ locations (e.g., the pattern can occur in every location). In this case, the Algorithm Match takes $\Theta(mn)$ time. Can you modify Algorithm Match so that in all cases, it outputs correctly (with high probability) *all* occurrences of the pattern in $\mathcal{O}(m+n)$ time.

Exercise 9.5. We have a hash table of size m to store a set of n keys (from a universe U), where $n \leqslant m/2$. Let h be a hash function from $U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$. We use the following technique called **open addressing** for inserting an element: given a key j we successively examine a *probe sequence* $< h(j,0), h(j,1), \ldots, h(j,m-1) >$ and insert the key at the first empty slot.

For the purpose of analysis we will make the assumption of uniform hashing i.e., each key is equally likely to have any of the $m!$ permutations of $< 0,1,\ldots,m-1 >$ as its probe sequence.

1. Show that for $i = 1,2,\ldots,n$ the probability that the ith insertion requires strictly more than k probes is at most $1/2^k$.

2. Let the random variable X_i denote the number of probes required by the ith insertion. Show that $\mathbb{P}\{X_i > 2\lg n\} \leqslant 1/n^2$.

3. Let $X = max_{1 \leqslant i \leqslant n} X_i$ denote the maximum number of probes required by any of the n insertions. Show that $\mathbb{E}[X] = \mathcal{O}(\lg n)$.

Exercise 9.6. We are given an array A of size n, such that each element is a positive number less than 1. Consider the following algorithm to sort A. We assume a hash table B where we handle collisions by chaining (using a linked list).

```
1. For $i = 1$ to $n$
        1.1 Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
2. For $i = 0$ to $n - 1$
        2.1 Sort list $B[i]$ with Insertion Sort (See Exercise 3.2)
3. Output the concatenated list $B[0], B[1], \ldots B[n-1]$.
```

Assume that the input numbers are drawn from a *uniform distribution* over the interval $[0, 1)$. Show that the expected running time of the above sorting algorithm is $O(n)$.

Exercise 9.7. A "good" hash function satisfies the property of uniform hashing i.e., if keys are drawn uniformly at random from the set $\{0, 1, \ldots n-1\}$ the probability of hashing into any particular slot is $1/m$. (Assume that $n(> 0)$ is a multiple of m, the number of slots.) Is the hash function $h(k) = k^2 \bmod m$ good? Justify your answer.

Exercise 9.8. We will implement an approximate and efficient membership tester called *Bloom filter* as follows. We have a set S of n elements (numbers). We store it in the Bloom filter which is simply a bit array of size m (each slot in the array can store one bit) as follows. (Initially all bits are set to 0.) Choose k random hash functions — $h_1, h_2, \ldots, h_k$ — assume each hash function is fully independent of others and hashes each item to a randomly chosen slot in the array. To insert a number x in the Bloom filter, simply compute $h_1(x), h_2(x), \ldots h_k(x)$ and set the corresponding indices in the bit array to be 1 (if they are not already set to 1). To check for membership, given an element x, we similarly, compute the k hashes and check and see whether *all* of them are 1; if so we say that x is in the filter, otherwise not.

- Argue that the membership query using a Bloom filter can only give a false positive, but never a false negative.

- Given n, m, and k, what should be k (in terms of n and m), so that the probability of getting a false positive is minimized? What is this minimum probability?

Exercise 9.9. We are given two *unsorted* sets of numbers S_1 and S_2, each of size n (the numbers in each set are distinct). The two sets are equal, i.e., $S_1 = S_2$; however, the numbers are ordered differently in the two sets. The goal is to group the equivalent numbers in S_1 and S_2 in pairs. Two numbers belonging to the *same* set cannot be compared. Using universal hashing, give a randomized algorithm for the problem that takes $\mathcal{O}(n)$ time in expectation and $\mathcal{O}(n)$ space. (Note that this problem, is similar to Exercise 8.17, but there we required $\mathcal{O}(n \log n)$ time.)

Exercise 9.10. A binary search tree (BST) is a data structure for implementing a dynamic dictionary. Recall that in a BST, the keys are stored in the nodes a binary tree which satisfies the search property: for any node x, the left child (if any) of x has key value less than x and the right child (if any) has key value greater than x. A key is inserted in a binary search tree by simply following the path from the root to a leaf that is given by search property. Suppose we want to insert n keys in a BST.

(1) Show that the worst-case height of a binary search tree is $\Theta(n)$.

(2) Suppose instead of inserting based on the key value, we use a fully random hash function to hash the value and then insert the hashed value. Assume that the random hash function hashes each element to a value between 1 and m (where $m \geqslant n$) with uniform probability, independently. What is the expected height of the BST?

Exercise 9.11. In cuckoo hashing, show that the expected amortized cost of rebuilding is $O(1)$ per insertion. How small should be the load factor for your argument to work?

Exercise 9.12. Consider the linear probing scheme described in the beginning of Section 9.1.2.

1. Assuming a fully random hash function and a hash table with load factor at most $1/2$, show that the expected cost of inserting or searching for an element is constant.

2. Note that care has to be taken while deleting an item. Since we stop searching when we reach an empty slot, if we just delete an item, this can lead to incorrect search. Describe a simple way to handle deletions so that search can be done correctly.

Exercise 9.13. Given a rectangular text T of size $m \times n$, and a rectangular pattern P of size $m_1 \times n_1$ (where $m_1 \leqslant m$ and $n_1 \leqslant n$), the problem is to find all occurrences of P in T. Give a randomized algorithm for this problem that runs in $O(mn)$ expected time. Assume that modular arithmetic with integers of value at most polynomial in m and n can be done in $O(1)$ time.

10 NUMBER-THEORETIC ALGORITHMS

In this chapter, we will study two important algorithms of tremendous real-world importance that involve numbers and extensively make use of number-theoretic concepts. The two applications are: primality testing and public key cryptography. Both are crucial ingredients in today's digital economy. We will also see two interesting applications of cryptography: zero-knowledge proofs and secure multi-party computation.

This chapter needs basic number-theoretic concepts for which we refer to Appendix D.

10.1 Problem: Primality Testing

We will study a randomized algorithm for testing whether an integer is prime or not. As mentioned earlier, primality testing is one of the most fundamental problems in mathematics and an efficient algorithm for the same has been sought for centuries. Until about 2002 or so, the most efficient algorithm known, i.e., that works in polynomial time, was a randomized algorithm. Indeed, it was unclear for a long time whether there was a *determinisitic* polynomial time algorithm for primality testing. Today, thanks to the result of Agrawal, Kayal, and Saxena, we know that primality testing can be accomplished in deterministic polynomial time. However, the randomized algorithm that we are going to see in this chapter is simpler and faster than the deterministic algorithm and is typically used in practice. The downside of the randomized algorithm is that it can err with a small probability, but this probability can be made as small as desired.

The randomization idea used in the primality testing is quite simple and can be explained as follows. Using number theory, it is first established that if a number, say n, is not prime (i.e., composite) then there are a lot of "witnesses" to this fact. Witnesses are numbers that are less than n; a witness satisfies a simple test (will be discussed below) that shows conclusively that n is not prime. The crucial point established is that the number of such witnesses are abundant: at least a constant fraction of n. This suggests a simple randomization idea that is very similar to the one that we saw in Section 8.2: Choose a random number between 1 and n and check whether it is a witness or not. If it is a witness, then we immediately know that n is not prime. If n is indeed not prime, then the probability of choosing a witness is at least a constant (because of the abundance of witnesses). Repeating the process a few times (say $\mathcal{O}(\log n)$ times) — each time, independently choosing a random number between 1 and n — the probability that we fail to find a witness is very small.

10.1.1 An Almost Correct Algorithm: Pseudoprimality Testing

Our randomized algorithm (like many other primality testing algorithms) is based on a simple theorem due to Fermat, *Fermat's Little Theorem*, which is proved in Appendix D.

Theorem 10.1 ▶ Fermat's Little Theorem (FLT)

If n is a prime then, for every a $(1 \leqslant a < n)$,

$$a^{n-1} \equiv 1 \pmod{n}$$

Using Fermat's Little Theorem, we can design a simple primality test called the *Pseudo-Prime Test* (Algorithm IsPseudoPrime) which is "almost" correct, in a sense which will become clear. A consequence of FLT is that, if a number is prime, it will pass this test for all values of a belonging to the set $\{1, \ldots, n-1\}$. FLT provides a *necessary*, but not a *sufficient*, condition for primality. That is, it does not guarantee the converse: that if n is composite, it will fail the test for at least one value of a. However, it turns out that the converse is true for essentially all composite numbers, except for a very tiny fraction of composite numbers called *Carmichael Numbers*.

Definition 10.1 ▶ Carmichael number

An integer $n > 1$ is a Carmichael number if $a^{n-1} \equiv 1 \bmod n$ for all $a \in \{1, 2, \ldots, n-1\}$, i.e., if a satisfies FLT for all $1 \leqslant a < n$.

Thus, one can consider Carmichael numbers "Pseudo-primes", i.e., like prime numbers, they satisfy FLT. Carmichael numbers are very rare – for example, there are only 646 Carmichael Numbers in the first billion integers, less that 0.0001%. We now show that for non-Carmichael composite numbers the test fails with probability at least 1/2. Hence, by repeating the Pseudo-prime test a certain number of times (say $\mathcal{O}(\log n)$ times) then, with high probability, i.e., with probability at least $1 - 1/n$, we can determine whether a number is a pseudo-prime. We will see that this provides an effective way to choose a large *random* prime number (this has many applications, e.g., in cryptography, as discussed later).

Algorithm 43 IsPseudoPrime
Input: A positive integer n
Output: `True` if n is a pseudoprime, else `False`

```
func IsPseudoPrime($n$):
    $a = \textsc{RandomChoice}(\{2, 3, \ldots, n-1\})$
    if $a^{n-1} \equiv 1 \pmod{n}$:
        return True
    else:
        return False
```

The following is the crucial lemma.

Lemma 10.2

Fix a number n. If n fails to satisfy FLT for some positive integer a in $\{1, \ldots, n-1\}$, then it fails to satisfy FLT for at least half of the values in the set $\{1, \ldots, n-1\}$.

Proof. Consider the set A of values in $\mathbb{Z}_n^* = \{1, \ldots, n-1\}$ that *satisfy* the FLT, i.e., $A = \{a \in \mathbb{Z}_n^* \mid a^{n-1} \equiv 1 \pmod{n}\}$. Clearly, A is nonempty, since 1 belongs to A. Also A is closed under multiplication modulo n: let $a_1, a_2 \in A$, then $a_1^{n-1} a_2^{n-1} \equiv 1 \pmod{n}$; hence $(a_1 a_2)^{n-1} \equiv 1 \pmod{n}$ as well. Hence A is a subgroup of $\mathbb{Z}_n^*$ by the closure property of subgroup (Lemma D.3). By the lemma's premise, there is an $a \in \mathbb{Z}_n^*$, that fails to satisfy FLT. Hence $a \notin A$. Hence A is a proper subgroup and hence by Lagrange's theorem, $|A| \leqslant |\mathbb{Z}_n^*|/2 = (n-1)/2$. Hence at most half of the values in the set $\{1, \ldots, n-1\}$ satisfy the FLT, implying the lemma. □

The above lemma, implies the following theorem.

Theorem 10.3

Algorithm Pseudoprime always returns `True` if the input number is prime or a Carmichael number. Otherwise, with probability at least 1/2, it returns `False`.

It is important to note that in the FLT test, we need to compute $a^{n-1} \pmod{n}$. If done naively, this will take $\mathcal{O}(n)$ multiplications which is not polynomial in the input size. The following exercise (Exercise 10.1) asks you to show that $a^{n-1} \pmod{n}$ can be computed in $\mathcal{O}(\log n)$ multiplications (See also Hashing Strings and Fingerprinting).

Exercise 10.1. Show how to compute $a^{n-1} \pmod{n}$ by using only $\mathcal{O}(\log n)$ multiplications (each multiplication involves only numbers of size $\mathcal{O}(\log n)$ bits). (Hint: You can use divide and conquer and the exponentiation rule for *mod*: $(a^y \bmod p) = (a \bmod p)^y \bmod p$. See Hashing Strings and Fingerprinting.)

Run time analysis

We analyze the running time of Algorithm Pseudo-prime Test. Choosing a random integer from the set $\{2, \ldots, n-1\}$ can be accomplished in $\mathcal{O}(\log n)$ time (Exercise 8.1). Computing $a^{n-1} \equiv 1 \pmod{n}$ can be done in $\mathcal{O}(\log n)$ multiplications (Exercise 10.1). If we apply the standard method to multiply two b-bit numbers in time $\Theta(b^2)$ time, then the modular exponentiation can be done in $\mathcal{O}\left(\log^3 n\right)$ time. Hence the overall time is $\mathcal{O}\left(\log^3 n\right)$ (i.e., cubic in the input size).

In practical applications, e.g., in RSA cryptography (Section 10.2), one needs to choose a large prime number. A fast way of accomplishing this is as follows. Choose a large random number and apply the Pseudo-prime test a few times (e.g., $2 \log n$ times, where n is the number chosen). If the test succeeds in all times, then we output the number as prime, otherwise not. One can argue that the number output is prime with high probability. The reason is that there are far more primes than Carmichael numbers. There is an important theorem in number theory called the *Prime number theorem* that says that the number of primes less than n is approximately $n/\ln n$. Hence, finding a prime near n (say within a factor of 2) by randomly choosing a number near n will succeed with probability about $\Theta(1/\ln n)$. On the other hand, the probability of choosing a Carmichael number near n is asymptotically much smaller.

10.1.2 A Correct Primality Testing Algorithm

Somewhat surprisingly, it turns out that we can modify the Algorithm Pseudoprime only a bit to get a primality testing algorithm that with high probability returns `False` when the input number is composite, even if it is a Carmichael number (see Algorithm PrimeTest). If it is a prime, then it always will return `True`. The main modification is that given that $a^{n-1} \equiv 1 \pmod{n}$ (where a is randomly chosen as before), we compute the *square root* of a^{n-1} modulo n, i.e., the number $\pm a^{(n-1)/2}$ (modulo n). A result in number theory is the following (which we state without proof):

Theorem 10.4

If n is a prime number then the square root of 1 (modulo n) is always *trivial*, i.e., it is either $1 \pmod{n} = 1$ or $-1 \pmod{n} = n - 1$.

The above theorem says that if there exists a *non-trivial* square root of 1 modulo n, i.e., some number other than $\pm 1 \pmod{n}$, then n is composite. On the other hand, if the algorithm does not find a non-trivial square root of 1 in the **for** loop and if n passes the FLT (i.e., n is a Carmichael number), then we show the following theorem (which we state without proof).

Theorem 10.5

If n is a Carmichael number, then for at least half the values of a, i.e., at least for half the values in the set $\{1, \ldots, n-1\}$, the algorithm finds a non-trivial square root of 1 modulo n. Hence if a is chosen randomly, then the probability that the algorithm returns "no" (i.e., composite) is at least $1/2$.

From the above theorem, it follows that if we repeat the Algorithm PrimeTest a few times (say $\mathcal{O}(\log n)$ times) then, with high probability, we will be able to detect a Carmichael number. Hence, this test "takes care" of Carmichael numbers as well.

Algorithm 44 PrimeTest
Input: An odd positive integer n
Output: `True` if n is prime else `False`

```
func PrimeTest($n$):
    ▷ Choose $k$ and odd $b$ such that $n - 1 = 2^k b$
    $a$ = RandomChoice($\{2, 3, \ldots, n-1\}$)
    $x_0 = a^b \bmod n$.
    for $i = 1$ to $k$:
        $x_i = (x_{i-1})^2 \bmod n$
        if $x_i == 1$ and $x_{i-1} \neq 1$ and $x_{i-1} \neq n - 1$:
            return False
    if $x_k \neq 1$:
        return False
    return True
```

10.2 Public Key Cryptography

In cryptography, two parties want to send a message securely, i.e., Alice wants to send a message to Bob and wants to be sure that no eavesdropper can learn anything about the message being sent.

One way to do this is for Alice to encode the message using a key and then send the encoded message. Bob can decode the message using the same key. This is pretty secure. However, the catch is both Alice and Bob need to know the key a priori. Thus the key itself has to be shared in a secure way.

Public-key cryptography is an ingenious method around the above problem. In this system, each party has a public and a private key. The private key is known only to the party, but the public key is advertised to everybody. Suppose Alice wants to send a message to Bob. She will first encode the message using Bob's public key and send it. Bob can then decode the message using his private key. Can we construct such a cryptosystem?

10.2.1 RSA Cryptosystem

RSA is exactly such a system and is widely used today for secure transactions. The RSA system is as follows. Suppose Bob wants to create a RSA-based public-key crypto system. He will first choose two (large) *random* prime numbers — p and q — and compute their product $n = pq$.[1] Let d be an integer that is relatively prime to $(p-1)(q-1)$, i.e., $gcd(d,(p-1)(q-1)) = 1$. Let e be the inverse of d modulo $(p-1)(q-1)$ (since d is relatively prime to $(p-1)(q-1)$, the inverse exists via Corollary D.2). Bob's *private* key is d and his *public* key is e. He will keep d private and announce the numbers e and n to the world.

Suppose Alice wants to send a message $x \in \{1,\ldots,n-1\}$ to Bob. She will then use Bob's public key to encrypt message x using the following *encryption* function:

$$E(x,n,e) = x^e \pmod n = y$$

She then sends y to Bob.

Bob uses his private key to decrypt the message using the *decryption* function:

$$D(y,n,d) = y^d \pmod n$$

The following theorem shows that the decryption function used by Bob correctly decrypts the message, yielding the original message sent by Alice.

Theorem 10.6

If

$$y = E(x,n,e) = x^e \bmod n$$

then

$$D(y,n,d) = y^d \bmod n = x$$

Proof. We have:

$$y^d \bmod n = x^{ed} \bmod n$$

[1] In today's applications, p and q will be (about) 1000-bit numbers.

Since $ed \equiv 1 \bmod (p-1)(q-1)$, we can write

$$ed = 1 + c(p-1)(q-1)$$

for some integer c. Hence, we have

$$x^{ed} \bmod n = x^{1+c(p-1)(q-1)} \bmod n = x \cdot x^{(p-1)(q-1)c} \bmod n$$

We want to show that $xx^{(p-1)(q-1)c} = x(\mathrm{mod} n)$, i.e., n divides

$$x\Big(x^{(p-1)(q-1)c} - 1\Big)$$

By applying FLT for the primes p and q, and using the modular exponentiation rule (Chapter 10), we have:

$$\begin{aligned} x^{(p-1)(q-1)c} - 1 \bmod p &= (x^{p-1} \bmod p)^{(q-1)c} - 1 \bmod p \\ &= 1 - 1 \bmod p \\ &= 0 \bmod p \end{aligned}$$

and similarly,

$$x^{(p-1)(q-1)c} - 1 = 0 \bmod q$$

Hence p divides $x(x^{(p-1)(q-1)c} - 1)$ and q divides $x\Big(x^{(p-1)(q-1)c} - 1\Big)$ and since $n = pq$ (product of two primes), it divides $x\Big(x^{(p-1)(q-1)c} - 1\Big)$. □

10.2.2 Why is RSA hard to break?

The reason RSA is hard to break is because of two hard computational problems that RSA uses. The first is factoring large numbers. If an eavesdropper can factor pq into primes p and q, then it is easy for him to decode the message, since if he knows p and q he can compute $(p-1)(q-1)$ and then using e, he can easily find d using Euclid's gcd algorithm.

The second is inverting an exponentiation modulo a prime – solving the equation $r^x = y \bmod p$ given a prime p, *primitive root* r of p, and y. A solution x to this equation is called a base-r *discrete logarithm* of y.

Currently, no polynomial time algorithms are known for either problem. Hence the security of RSA depends on the presumed hardness of the above two number-theoretic problems.[2]

10.2.3 Randomized RSA

Note that RSA is a purely deterministic protocol (no randomness involved). This can be a weakness, as it leads to attacks by seeing repetitions of messages, or by using known messages and encoding them.

Another glaring problem is encoding small messages, e.g., a message of just 1 bit. Since $x^e \bmod n = x$ if x is a single bit message, there is no encoding at all.

The way to avoid these problems is by using randomization. Suppose you want to encode a single bit. Choose a random number $x < n/2$ and then transmit to Bob

[2] However, it is now known that, using Shor's Algorithm [6], a *quantum computer* can break RSA encryption in polynomial time.

$y = (2x+b)^e \bmod n$. Bob receives y and uses his private key to recover the bit b from the message $2x+b$ (i.e., b will be the last bit in the message). This method can be shown to be as secure as the original RSA, but in addition due to randomization it is immune to attacks such as detecting repetitions etc. In fact, note that every message can be sent like this if we can encode each bit individually; however, this will blow up the size of the messages.

10.3 Zero Knowledge Protocols

Zero knowledge proofs are a nice application of public key cryptography. Suppose Alice has some knowledge and she wants to convince Bob of the same. A zero knowledge protocol (ZKP) will accomplish that: at the end of the ZKP, Bob will be *convinced with high probability*, that Alice has *the knowledge that she claims*, but Bob himself *will not learn* anything about Alice's knowledge (hence zero knowledge is transferred from Alice to Bob).

We now give a ZKP for a problem in NP, namely the 3-coloring problem. Since 3-coloring is NP-complete, it is possible to show by reduction that all problems in NP admit a ZKP.

First we define the 3-coloring problem.

Problem 10.1 ▶ 3-coloring

Given a graph G, the goal is to color the graph using 3 colors only, if such a coloring is possible. That is, each vertex should be colored using one of the three colors and no adjacent vertices (i.e., that share an edge) should get the same color.

It is easy to show that the decision version of this problem (i.e., given a graph G whether it is 3-colorable) is in NP (see Chapter **??**). Furthermore, this problem can be shown to be NP-Complete, i.e., it is among the hardest problems in NP.

In the setting of ZKP, we have two parties — Alice and Bob — and each is given the same (undirected) graph G. Alice also knows a 3-coloring of G which she wants to convince Bob of. Let G has n nodes and m edges. We present a ZKP for the problem.

Zero Knowlege Protocol for 3-coloring. *One round* of the ZKP consists of the following four phases of back and forth communication between Alice and Bob:

Phase 1 (Alice) Alice generates a *random* permutation of her colors. Then she generates n RSA public-private key pairs $(p_i q_i, d_i, e_i)$ one for each node $i \in V$. For each node i she encodes the two bits of its *permuted* color, by using the public key (e_i). (Recall she will encode each bit by using randomised RSA cryptography described in Section 10.2.3). She will send the encrypted colours and $(p_i q_i, e_i)$ to Bob for all nodes i.

Phase 2 (Bob) Bob will a pick a *random* edge and ask Alice for the private keys of two endpoints of the edge.

Phase 3 (Alice) Alice in turn will respond with the private keys of the two endpoints.

Phase 4 (Bob) Bob in turn will use these private keys to decrypt the colours of the two endpoints and check whether they are different. This ends one round of the protocol.

The overall protocol consists of $m \log n$ rounds, where m and n are the number of edges and nodes of G respectively. It is important to note that in each round (in Phase 1), Alice generates a *new* random permutation of her colors.

Theorem 10.7

The above protocol is a ZKP for 3-coloring.

Proof. First we show that Bob will be convinced that Alice has a correct 3-coloring, with high probability, if in each of the $m \log n$ rounds, Bob always see two different colours of the randomly chosen edge.

Assume that Alice does not have a correct 3-coloring. Then some edge will always be wrongly colored, i.e., both endpoints will have the same color. The probably that Bob will choose such a wrongly colored edge is at least $1/m$ in a round. The probability that he will find out in $m \log n$ rounds (if Alice is cheating) is very high: $1 - (1 - 1/m)^{m \log n} = 1 - 1/n$. Note that Alice cannot cheat, when Bob asks for the private keys of the random edge. She has already given the public keys for all the nodes, so the private keys have to match.

Now we claim that protocol is zero knowledge. This is because in each round, Bob learns nothing new. In other words, he hasn't learnt anything that he himself cannot generate. He simply sees some RSA keys, and some randomly permuted colors of the endpoints of an edge. (Note that that's why it is important for Alice to choose a new random permutation of the colours in each round). □

10.4 Exercises

Exercise 10.2. The *lcm (least common multiple)* of two integers a and b is the smallest number divisible by both a and b. Given two n-bit numbers a and b, give an efficient algorithm to compute the lcm of a and b. Analyze the running time of your algorithm.

Exercise 10.3. Show that if a is a non-trivial square root of 1 (modulo n), i.e., $x^2 \equiv 1 \pmod{n}$, but $x \not\equiv \pm 1 \pmod{n}$, then n is composite.

Exercise 10.4. Instead of using $N = pq$ (where p and q are primes) in the RSA, suppose we use just p as the modulus. Similar to RSA, we have public and private keys e and d respectively (chosen as before), and a message $m \in \{1, \ldots, p-1\}$ is encrypted as $y = m^e \bmod p$. Prove that this cryptosystem is not secure by showing that an adversary can efficiently recover m given p, e, and y. Analyze the running time of the recovery algorithm.

Exercise 10.5. Show that $(n-1)! \equiv -1 \pmod{n}$ if and only if n is a prime. Why can/cannot we use this as a test for checking primality?

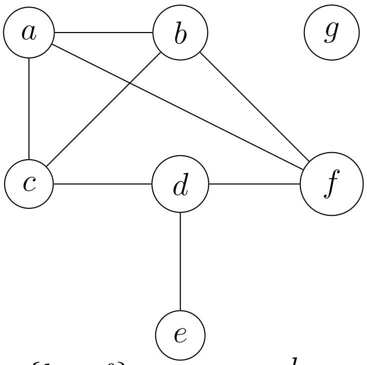
11 FUNDAMENTAL GRAPH ALGORITHMS

Graphs or networks are fundamental objects that can be used to represent various kinds of relationships in the real world. A graph (also called as a network) consists of *nodes/vertices* and *edges/links* where nodes are elements from some set and edges represent connections (relationships) between pairs of nodes. Edges can be *directed* or *undirected*. There can be *weights* on the edges which can denote the "strength" of the connections. A graph is usually denoted by the notation $G = (V, E)$, where V is the set of nodes and E is the set of edges. V is typically a set such as $\{1, 2, \ldots, n\}$ denoting n vertices labelled $1, 2, \ldots, n$ and E consists of elements (2-tuples) of the form (u, v) where (u, v) represents an edge connecting node u with node v. We usually denote the number of nodes by n and the number of edges by m. We refer to Appendix E for basic graph concepts and examples.

Graphs naturally model many real-world data, especially data with relationships. Some prominent examples of real-world graphs are:

- A *communication network* (e.g., the Internet) can be represented as a graph where nodes are end-hosts (computers, routers etc.) and edges are communication links between end-hosts.
- Nodes can represent people and two people are connected if they are mutually friends (*Social network*).
- Nodes can represent webpages and there is a (directed) link between a webpage A and a webpage B if there is a hyperlink from page A pointing to page B (*Web graph*).
- Nodes can represent points (places) in a map and edges can represent roads between the points. Edges can have weights on them, which can represent the road distance. (*Road network*).

In this chapter, we will focus on two fundamental graph search algorithms, namely *depth-first search* and *breadth-first search* and their applications. Both search algorithms are useful in numerous other graph algorithms and applications. Both algorithms can be implemented in linear time, i.e., $\mathcal{O}(m + n)$ time, where, m denotes the number of edges and n the number of vertices. Thus linear means "linear in the size of the graph", which is $m + n$. These linear time algorithms are the basis of fast algorithms for several other fundamental graph problems.



$$\begin{aligned} \text{adj}[a] &= \{b, c, f\} \\ \text{adj}[b] &= \{a, c, f\} \\ \text{adj}[c] &= \{b, a, d\} \\ \text{adj}[d] &= \{c, f, e\} \\ \text{adj}[e] &= \{d\} \\ \text{adj}[f] &= \{a, b, d\} \\ \text{adj}[g] &= \{\} \end{aligned}$$

$$\begin{array}{c|ccccccc|} & a & b & c & d & e & f & g \\ a & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ b & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ c & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ d & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ e & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ f & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Figure 11.1: A graph and its representation. The left shows the adjacency list representation and the right the adjacency matrix representation.

11.1 Representing a Graph

Before we turn to graph algorithms, we first discuss how to represent a graph in a computer, namely the data structures needed to store a graph. There are typically two ways to represent a graph $G = (V, E)$, where V is the set of vertices (nodes) and E is the set of edges (links). Throughout, we will use $|V| = n$ (denoting the number of vertices) and $|E| = m$ (denoting the number of edges). We will also assume that the vertices are numbered from 1 to n, unless otherwise stated.

11.1.1 Adjacency List (Adj) Representation

The adjacency list representation consists of an array of vectors (or linked lists), i.e., $A[i]$ is a vector (linked-list) corresponding to vertex i, $1 \leqslant i \leqslant n$. There is a vector (or a linked list) for each vertex. The vector contains a list of *all neighbors* of the vertex. If it is a weighted graph, the weight is also stored along with the neighbour. If (i, j) is an (undirected) edge in G, then i appears in j's list and vice-versa. If the edge (i, j) has a weight $w(i, j)$ then it is stored alongside i in j's list and vice-versa. In an undirected graph, the size of the adjacency list of v is $\deg v$.

Figure 11.1 shows a graph and its representation.

Space and Time for Adjacency List

The total space needed for an adjacency list is $\mathcal{O}(m + n)$. It gives a sequential way to access (or check) a particular neighbor of a vertex, as one has to go through the adjacency list of the vertex. This takes time that is *linear* in the size of the adjacency list.

11.1.2 Adjacency Matrix Representation

Alternatively, one can use an adjacency matrix representation, where the graph is stored as a 2-dimensional $|V| \times |V|$ array, where $A[i, j] = 1$ iff (i, j) is an edge in the graph, otherwise $A[i, j] = 0$. For a weighted graph, $A[i, j] = w(i, j)$.

Space and Time for Adjacency Matrix

Total space needed for an adjacency matrix is $\Theta(n^2)$. The advantage of matrix representation is that it provides random access to any edge — takes $\mathcal{O}(1)$ time.

Note that the two representations have different space and access time bounds. For sparse graphs, i.e., graphs with few edges (say $\mathcal{O}(n)$ edges), it might be a good idea to store them as an adjacency list rather than as an adjacency matrix, since it saves space. On the other hand, if the graph is dense, say, has $\Theta(n^2)$ edges, then adjacency matrix representation is more suitable, since it gives random access to an edge with essentially optimal space.

We will next study two fundamental algorithms to search (explore) a graph: *Depth-first search (DFS)* and *Breadth-first search (BFS)*. While DFS explores graph by always exploring the most recently visited vertex, BFS explores nodes in increasing distances from a given source node. We first focus on DFS which can be naturally implemented as a recursive algorithm.

11.2 Depth-first Search (DFS)

In DFS, the search starts from a vertex s, and edges are explored out of the *most recently* visited vertex. DFS yields a spanning tree called *depth-first spanning (DFS) tree* (see Appendix E for definition of spanning tree and its properties). DFS can be implemented in $\mathcal{O}(|V| + |E|)$ (i.e., linear) time.

Figure 11.2 shows a graph and its DFS tree.

We will next explore in detail DFS and its algorithmic applications.

11.2.1 Depth-first Search (DFS) Algorithm in an Undirected graph

We will first explore DFS in an *undirected* graph. We are given an *undirected and connected* graph $G = (V, E)$ represented by its adjacency list; and a vertex $s \in V$. The goal is to output a *DFS tree* T. As a byproduct, it will also be useful to output the DFS numbering of vertices, i.e., the order in which the vertices are visited in the graph during the DFS search. As we will see, this numbering is very useful for various applications.

DFS starts at a given source vertex s and it explores edges in a depth-first manner, i.e., an edge is explored from the currently (most recently) visited vertex — call it v (initially v is s). If the explored edge leads to an already visited vertex, then another edge out of v is explored. If all edges out of v leads to vertices that were already visited (i.e., all neighbors of v have already been visited), then the search *backtracks* to the vertex that was visited *prior* to v (i.e., the vertex from which v was visited). The algorithm that we have described can be written in an iterative (non-recursive) way; to implement

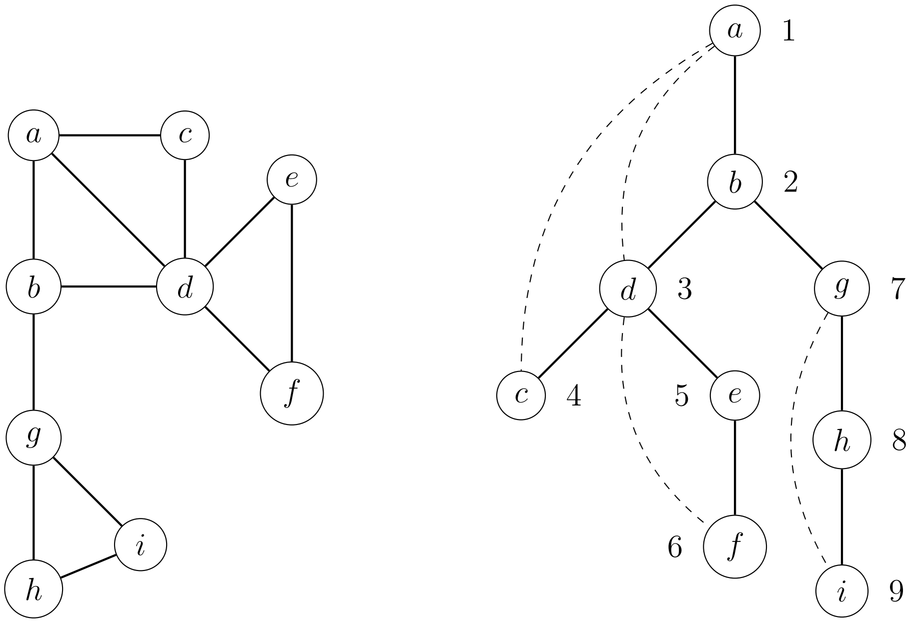


Figure 11.2: An undirected graph and its DFS tree rooted at vertex a where the DFS is started. Thick lines show the tree edges and the dotted lines show the back edges. The *num* values of the vertices — which shows the order in which the vertices are visited — are also shown.

backtracking the most natural data structure is a *stack*. This implementation is a bit cumbersome, as we have to explicitly maintain the stack (see Exercise 11.10).

Alternatively, it is easiest to write DFS as a *recursive* algorithm. The recursive algorithm does not need an explicit data structure and is compact and elegant. The pseudocode is given in Algorithm 45. The pseudocode has two auxiliary data structures:

1. An array `visited` that is initialized to false for every vertex $v \in V$, i.e., $\texttt{visited}[v] = false$. This array maintains whether the vertex has already been visited or not.

2. An array `num` — $\texttt{num}[v]$ gives the DFS number of v, i.e., the order in which vertex v was visited in the graph. This DFS numbering is very useful in understanding the structure of the graph.

3. A tree T (which is maintained as a list of edges and is initially empty) which stores the DFS tree edges. Note that T will have exactly $n - 1$ edges at the end of the algorithm since it is a spanning tree of G and G is assumed to be connected.

Let us describe the recursive algorithm which is quite simple and short: when a vertex v is visited, the DFS is called recursively on all unvisited neighbors of v one by one. $\texttt{visited}[v]$ is updated to `True` when DFS was invoked on v. Note that it is invoked only if $\texttt{visited}[v] =$ `False` prior to that — this is checked in the **for** loop before DFS is called. When a vertex is visited for the first time, that means the edge that is used to visit it belongs to the DFS tree; this is added to T (line 7). Also, when a vertex is visited for the first time, its DFS numbering is set (line 4). Note that variable `count` is initialized to zero before invoking the DFS algorithm.

In this algorithm, we maintain the DFS tree edges as a list (or an array). However, the algorithm can be easily modified to maintain the tree structure, by maintaining parent and child relationships for every vertex. It is clear that if an edge (a, b) is added to the tree (line 7), then a is the parent of b and b is the child of a in the DFS tree.

Algorithm 45 DFS
Input: An undirected graph G and a source vertex s
Output: DFS Tree T

```
1: func DFS($G$, $s$):
2:     visited[$s$] = True
3:     num[$s$] = count
4:     count += 1
5:     for vertex $v \in$ adj[$s$]:
6:         if not visited[$v$]:
7:             add edge $(s, v)$ to $T$
8:             DFS($G, v$)
```

Running time of DFS

The run time of DFS can be upper bounded by the number of recursive calls made plus the number of times the **if** statement is executed. First, we bound the number of recursive calls. A recursive call is made each time a vertex is visited for the first time, i.e., visited value for that vertex is false. Note that once a vertex is visited its *visited* value will be set to true and hence will not be visited again. Thus each vertex is visited exactly *once*. Thus the number of recursive calls made is $\mathcal{O}(n)$. Next, we bound the number of times the **if** statement is executed. Fix a vertex v. The **if** statement is executed as many times the **for** loop is run. The **for** loop runs $\deg v$ times (i.e., the degree of v, the number of neighbors of v), since that is the size of the adjacency list. Thus the total number of iterations overall is: $\sum_{v \in V} \deg v = 2m$. Hence overall run time of DFS is $\mathcal{O}(n + m)$, i.e., *linear* in the size of the graph.

11.2.2 A key property of DFS

DFS is very useful in understanding the structure of the graph. An important byproduct of DFS is that it allows the edges of graph G to be classified into two types based on the output DFS tree T.

1. *tree edges*: edges that belong to T.

2. *back (non-tree) edges*: edges that do not belong to T.

The next lemma shows an important property of back edges.

Lemma 11.1 ▶ Property of back edges

If (v, w) is a back edge, then in a DFS spanning tree T, v is an *ancestor* of w or vice versa.

Proof. Let (v, w) be a back edge in a DFS spanning tree T. Assume that v is visited before w. Then we will show that v will be an ancestor of w. $DFS(v)$ is called before $DFS(w)$. Thus, when v is visited w is still not visited. On the other hand, when $DFS(v)$ has finished, $DFS(w)$ would have been finished, since all recursive calls made by neighbors of v will have to finish before the call $DFS(v)$ finishes. Thus all previously unvisited vertices visited by $DFS(v)$ will become *descendants* of v in T. This includes w as well, since w is on the Adjacency list of v. Hence v will be an ancestor of w. □

Back edges are useful in many ways. For example, a back edge indicates the presence of a cycle. It is also useful in identifying cut vertices and edges as described in Section 11.2.4. Before that we first describe a fundamental application of DFS, namely, checking *connectivity* of a graph.

11.2.3 DFS Application: Finding connected components of a graph

We are given a graph $G = (V, E)$ (possibly disconnected) and we want the check if the graph is *connected* or not. More generally, in many applications, we want to output the *connected components* of the graph, if the graph is *disconnected*. DFS can be used to find all the connected components of the graph in *linear* time. The pseudocode is given in Algorithm 46. This pseudocode is just simply running DFS starting from every node in the graph, thus it calls $DFS(G, v)$ for every node v. Each call will return a connected component of G. If the graph is connected, then calling DFS from any node will explore all nodes in the graph and thus all nodes will be visited in the first DFS call itself. Hence only one component (the whole graph) is returned. If the graph is disconnected, each call will return the component containing the vertex where the DFS is called.

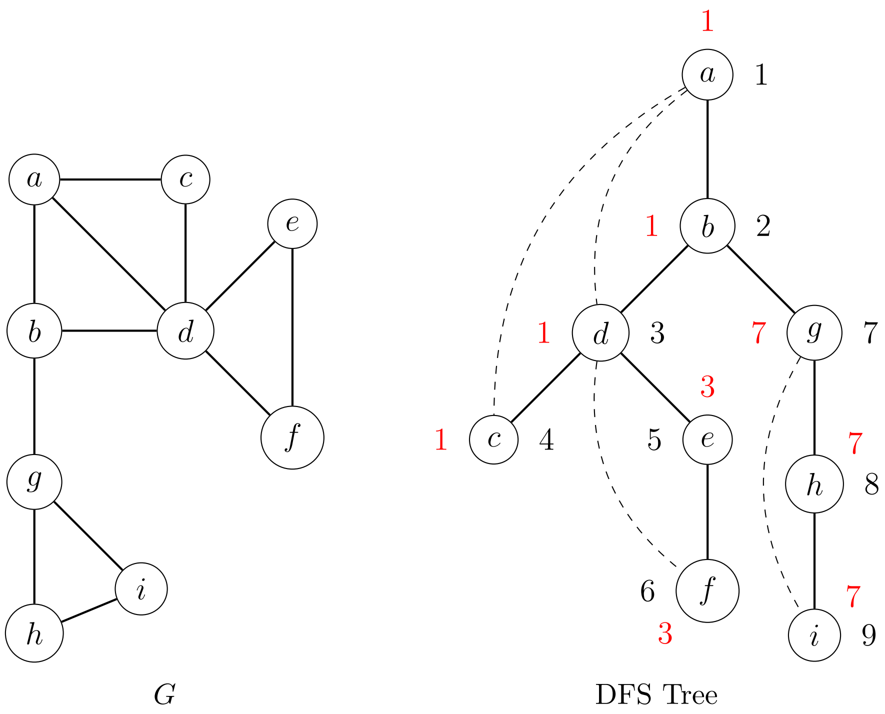
Algorithm 46 DFSConnectedComponents
Input: A graph G
Output: Spanning tree of connected components of G

```
func DFSConnectedComponents($G$):
    for $v \in V$:
        visited[$v$] = False
    for $v \in V$:
        if not visited[$v$]:
            DFS($G, v$)
```

11.2.4 DFS Application: Finding cut vertices and cut edges of a graph

Let $G = (V, E)$ be a connected undirected graph. A vertex x is said to be the *cut-vertex* of G if removing x from G *disconnects* the graph. Equivalently, x is a cut vertex if there exists two other vertices u and v such that *every path* between u and v contains the vertex x. Note that that above property means that x, u, and v cannot lie in a common *cycle*. (Otherwise, removing x will not disconnect u and v, since there is an alternate path via the cycle.) A graph is said to be *biconnected* if it has *no cut vertices* (See Worked Exercise 11.1). See Figure 11.3.

Similarly, an edge $e \in E$ is a *cut-edge* if removing e disconnects the graph. It is easy to see that a cut-edge cannot lie in any cycle of the graph.



G

DFS Tree

Figure 11.3: A graph G (left) and its DFS tree (right). The *num* values of the vertices are shown in black and the *low* values are shown in red. The cut vertices of G are b, d and g. Note that each of their *num* values is less than or equal to the *low* values of at least one their respective children. For example, d is a cut vertex because one of its children, e, has its *low* value greater than or equal to d's num value. The DFS tree of G can be used to find all its cut vertices. Note that a cut vertex will not have a back edge that goes from its descendent to its ancestor. For example, there is a back edge from d's descendent f, but it does not go to d's ancestor, just d. Hence d is a cut-vertex, since it is the only vertex that connects its descendant f to its ancestors.

Using DFS to find all cut-vertices

A straightforward algorithm for finding a cut-vertex based on its definition can take $\mathcal{O}(mn)$ time (see Exercise 11.4). Using DFS we can give an elegant algorithm that runs in $\mathcal{O}(n+m)$ time. We use the back edge property of the DFS spanning tree to help us characterize whether a vertex is cut vertex or not. The following lemma shows exactly that.

Lemma 11.2

Let T be a DFS spanning tree of G. A vertex x is a cut vertex of G if and only if:

1. x is the *root* of T and x has *more than one child*.
2. x is not the root, and for some child of y of x, there is *no back edge* between any descendant of y (including y itself) and a (proper) ancestor of x.

Proof. We first show the "if" part. If condition (1) is satisfied then clearly removing the root disconnects the graph. If condition (2) is satisfied, then removing x will disconnect y from the root. This is because, since there is no back edge from y to any proper ancestor of x (i.e., any ancestor of x other than x itself), removing x disconnects y and any proper ancestor of x (say the root). Thus, we have shown that x is a cut vertex under both conditions.

Next we show the "only if" part. That is we have to show that if the two conditions are false, then x is not a cut-vertex (this is contrapositive to showing that x is a cut vertex only if one of the two conditions hold). If x is a root and condition (1) does not hold, then x has only one child, and, clearly, removing x does not disconnect the graph (it does not disconnect any two nodes u and v since they are all descendants of a single child). If x is not a root, then there are two cases. If x is a leaf (i.e., has no children), then clearly it cannot be a cut vertex. On the other hand, if x has at least one child, and every child of x has a back edge to a proper ancestor of x, then removing x will not disconnect any child, since they are part of a cycle created by the back edge. □

To design an efficient algorithm we have to check the two conditions of the above lemma efficiently. It is easy to check condition 1. To get an efficient algorithm we have to check condition 2 efficiently.

Checking condition 2 in DFS

We can check for condition 2 while doing DFS itself by slightly modifying the DFS algorithm to compute an array called *low*[] defined as follows. Note that the DFS number of each vertex v is given by the *num* array. *low*[v] is the highest ancestor (i.e., the one with the lowest DFS number) v can reach (*including* v) either by a *back edge from* v or by a back edge from one of the *descendants* of v.

Defining *low* value

For a vertex v, we first define a quantity called back-edge-num[v] as follows. In words, back-edge-num[v] is simply the highest ancestor (i.e., the one with the lowest DFS number) that any descendant of v (including v) can reach via a back edge.

back-edge-num$[v]$ = $\min\{num[w] | \exists$ a back edge (x, w) such that x is a descendant of v(including v)$\}$

Then we define $low[v]$ as:

$$low[v] = \min\{num[v], \text{back-edge-num}[v]\}$$

In other words, $low[v]$ is the minimum of $num[v]$ and back-edge-num$[v]$. Note that *low* value of a node depends only on its *num* value and the *low* values of its descendants, in particular the low values of its children.

We note that Condition 2 is satisfied for vertex x if it has a child y such that $low[y] \geqslant num[x]$. This is because $low[y]$ is highest ancestor that y can reach via back edge and that is not a proper ancestor of x, since $low[y] \geqslant num[x]$. Hence, we can use *low* values to check for Condition 2 efficiently.

Computing *low* values

We would like to compute *low* values during DFS. To do this we need to compute *low* values recursively. This can be done, since the *low* value of a vertex x depends on its *num* value, the *low* value of its children and whether x has any back edge.

We can compute the *low* values recursively:

$$low[x] = \min\{num[x], \{low[y] \mid y \text{ is a child of } x\}, \{num[w] \mid (x, w) \text{ is a back edge }\}\}$$

When the DFS call finishes at vertex x it has *already finished* exploring all the children of x. Hence x has all the information needed to compute its *low* value.

DFS-based algorithm for finding cut vertices

The algorithm takes as an input an undirected and connected graph $G = (V, E)$ represented by its adjacency list; and a vertex $s \in V$ and it outputs the cut vertices of G (if any). The algorithm's pseudocode is presented in Algorithm 47. The algorithm invokes DFS on a source vertex s (any arbitrary vertex of the graph). It uses the following auxiliary data structures: *num* array, *visited* array, *low* array, and *parent* array. $parent[v]$ stores the parent of v in the tree T, it is not defined for root of the tree. Initially, $visited[v] = false$, and $parent[v] = null$ for all v. Initially $T = \emptyset$ and $count = 1$.

The algorithm implements the recursive way of computing *low* value as defined earlier. When $DFS(v)$ finishes and returns, its low value has already been computed, since they depend on itself and its descendants (which are already finished). Hence if $low[v] \geqslant num[s]$ (line 11), Condition 2 is satisfied and we can output s as a cut vertex. Note that the algorithm is invoked at vertex s and hence s is the root. It will be a cut vertex if it has more than one child — this is checked at the end, when the call at s finishes. We will leave the formal proof of correctness (using induction) as an exercise.

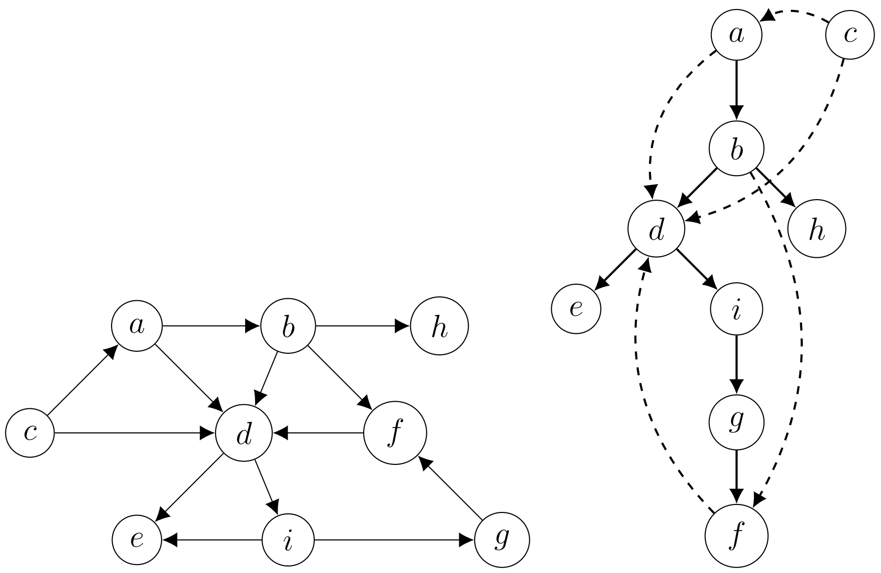


Figure 11.4: Example: A directed graph and its DFS tree. The tree edges are shown in bold and non-tree edges are shown in dotted lines. (f, d) is a back edge, (a, d) and (b, f) are forward edges, (c, a) and (c, d) are cross edges.

Algorithm 47 DFSCutVertices

Input: A connected, undirected graph G and a vertex s

Output: Prints any nodes (other than s) which are cut vertices

```
func DFSCutVertices($G$, $s$):
    visited[$s$] = True
    num[$s$] = count
    count += 1
    low[$s$] = num[$s$]
    for $v \in Adj[s]$:
        if not visited[$v$]:
            Add edge $(s, v)$ to $T$
            parent[$v$] = $s$
            DFSCutVertices($v$)
            if low[$v$] $\geqslant$ num[$s$]:
                Output $s$ is a cut-vertex
            low[$s$] = min(low[$s$], low[$v$])
        else if $v \neq$ parent[$s$]:        ▷ checks whether $(s, v)$ is a back edge
            low[$s$] = min(low[$s$], num[$v$])
```

11.2.5 Depth-first search in a directed graph

DFS in a directed graph is similar to that of an undirected graph. The pseudocode is the same as that of DFS in an undirected graph; however, note that a vertex v is adjacent to u only if there is a directed edge from u to v. Figure 11.4 shows a directed graph and a DFS tree of the graph. In an undirected graph, DFS classifies the edges into two types: tree and back edges. In the directed case, it classifies into four types described below.

Classification of edges in a directed graph

DFS in a directed graph G classifies the edges of G into 4 types:

1. *Tree edges*: Edges that belong to the DFS forest/tree. An edge (u, v) is a tree edge if v is visited for the first time when exploring edge (u, v).
2. *Back edges*: Edges that connecting a vertex to its *ancestor* in the DFS tree.
3. *Forward edges*: Edges that connect a vertex to its *descendent* in the DFS tree.
4. *Cross edges*: All other edges of G. They can go between vertices in the same DFS tree or between vertices in different trees.

The above classification is useful in many applications involving DFS in a directed graph. For example, they are useful in understanding the connectivity structure in directed graphs. Directed graphs have a different notion of connectivity compared to undirected graphs. A directed graph is said to be *strongly connected* if for every pair of nodes u and v, there is a *directed* path from u to v and from v to u. It is called *weakly connected* if the graph is connected by ignoring the directions (i.e., the underlying undirected graph is connected). The edge classification of DFS helps in finding strongly connected components in directed graphs. Worked Exercise 11.2 solves the problem of checking whether a directed graph is strongly connected or not. This approach can be modified to find all the strongly connected components of a directed graph as well (see Exercise 11.9).

11.2.6 Topological Sort

We will consider an important application of DFS in a directed graph, called *topological (topo) sort*. Topo sort can be used to find a *linear ordering* of the vertices of a *directed acyclic graph (DAG)*. A DAG is a directed graph with *no (directed) cycle*. See Figure 11.5 for an example.

DAG

DAGs are used in real-world applications to model precedences among events. For example, you have to complete a set of n operations (e.g., these might be operations to assemble a car in a factory). There are precedences among the operations which are represented as directed edges: (u, v) denotes that operation u has to be performed *before* operation v. The above edges form a DAG, if it is valid set of operations. Topological sorting of a DAG gives a valid ordering of performing operations such that all the precedence constraints are obeyed. Note that there may be more than one valid ordering — topo sort returns one such ordering.

Topological ordering can also be viewed as an ordering of the vertices of the DAG along a horizontal line so that all directed edges go from *left to right*. Note that if the graph is not a DAG, then there is no topological ordering possible (we leave it to the reader to prove this).

Using DFS for Topo sort

We need to modify DFS code a little for doing topo sort as follows. The modified DFS code computes *finishing* order for vertices (defined below) as well. Recall that $num[v]$

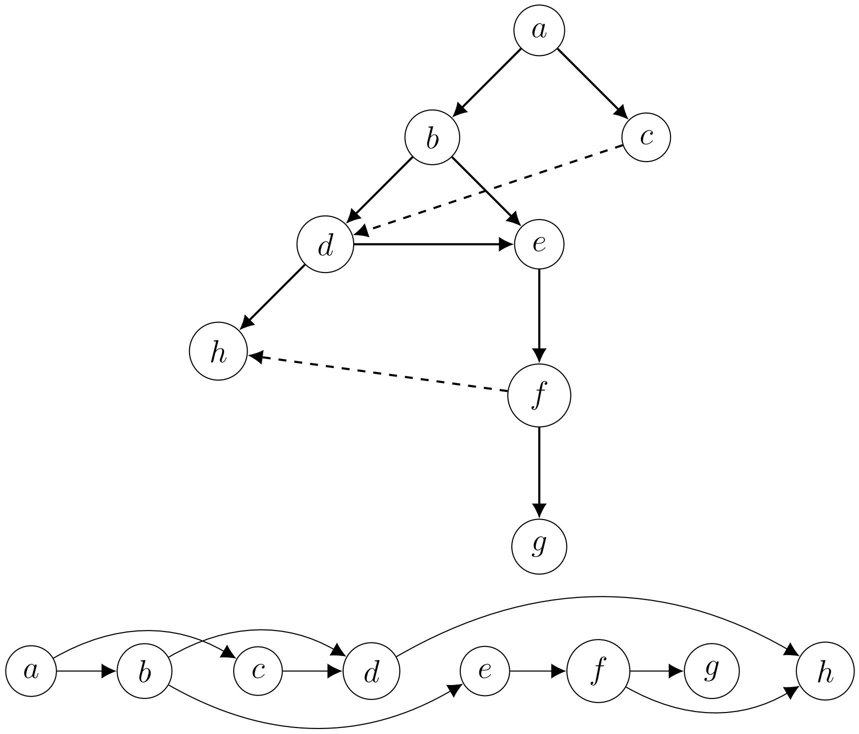


Figure 11.5: Example: A DAG. The top shows a topological ordering of the vertices of the DAG so that all edges are directed from left to right. The bottom shows the DFS tree of the DAG. Note that it has no back edges.

represents the order in which the vertices were visited, i.e., the time at which the vertex was first visited. Let $fin[v]$ represent the order in which v was finished processing, i.e., the time at which the recursive call $DFS(v)$ returns. Algorithm 48 gives the DFS algorithm that also outputs the finishing times of vertices.

Algorithm 48 DFSFinishTimes – DFS with Finishing Times
Input: A directed graph G
Output: The finishing times of vertices

```
func DFSFinishTimes($G$):
    for $v \in V$:
        visited[$v$] = False
    time = 0
    for $v \in V$:
        if not visited[$v$]:
            DFSFinishTimesHelper($v$)

    func DFSFinishTimesHelper($s$):
        visited[$s$] = True
        time += 1
        num[$s$] = time
        for $v \in Adj[s]$:
            if not visited[$v$]:
                DFSFinishTimesHelper($v$)
        time += 1
        fin[$s$] = time
```

Algorithm Topo Sort

The Topo Sort algorithm takes as input a DAG $G = (V, E)$ represented by its adjacency list. It outputs a topological ordering of vertices of G. The algorithm's pseudocode is given in Algorithm 49. The algorithm simply calls DFS on G to compute the finish times and list the vertices in decreasing order of their finish times. We will show that this order is a valid topological ordering.

Algorithm 49 TopoSort
Input: A DAG G
Output: Topological ordering of the vertices of G

```
func TopoSort($G$):
    DFSFinishTimes($G$)
    Output the vertices by decreasing order of their finish times
```

One way of implementing the Step 2 of the Topo Sort algorithm is to store vertices as they finish in a vector (or array) and then returning the vertices in *reverse* order.

It is clear that the running time of Topo Sort is the same as DFS, i.e., $\mathcal{O}(|V| + |E|)$.

Correctness of Topo Sort

The correctness of Topo Sort uses the following key property of a DAG.

Lemma 11.3

In a DAG G, DFS yields no back edges.

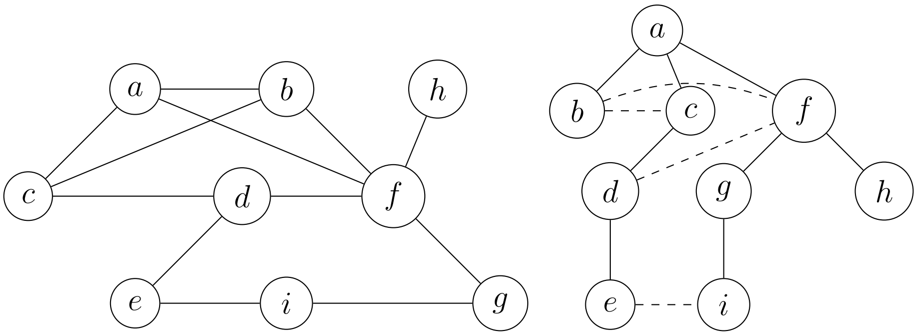


Figure 11.6: An undirected graph and its BFS tree.

Proof. Suppose DFS on G yields a back edge (u, v). Then v is an ancestor of u in the DFS (directed) tree. Thus there is a path from v to u in G (using the tree edges), and the back edge completes a directed cycle. This contradicts the definition of a DAG. □

Theorem 11.4

Algorithm TopoSort produces a topological ordering of a DAG.

Proof. It is enough to show that if there is a directed edge (u, v) then $fin[v] < fin[u]$. DFS will explore edge (u, v) when it is exploring vertex u. There are 2 cases:

1. v is not yet visited: Then v will be visited for the first time and $DFS(v)$ will be called which will finish first before $DFS(u)$ finishes. Hence $fin[v] < fin[u]$.

2. v is already visited: Note that (u, v) cannot be a back edge, i.e., v cannot be an ancestor of u. The only possibility is that v has already finished exploring. Since u is not yet finished, $fin[u] > fin[v]$.

□

11.3 Breadth-first search (BFS)

In BFS, the search starts from a vertex s, and visits vertices in *increasing order of distance* from s. BFS yields a spanning tree called *breadth-first spanning (BFS) tree* in an undirected connected graph. An important application of breadth-first search is finding the *(shortest path) distance* (as well as *shortest path*) from s to all other vertices. Recall that the distance between two vertices is the number of edges in a shortest path between them. BFS can be implemented in $\mathcal{O}(|V| + |E|)$ time.

Figure 11.6 shows an undirected graph and its BFS tree.

11.3.1 BFS Algorithm

The pseudocode for the BFS algorithm is given in Algorithm 50. It takes as input an *undirected and connected* graph $G = (V, E)$ represented by its adjacency list; and a source vertex $s \in V$. It outputs the following:

1. BFS tree T rooted at s.

2. *dist* array — $dist[v]$ gives the distance of s to v. (the *dist* array is initialized to ∞ for all nodes.)

3. *parent* array — $parent[v]$ stores the parent of v in the tree T.

The algorithm uses the following auxiliary data structures:

1. A queue Q that stores the list of vertices that will be explored.

2. An array $visited[]$ that is initialized to false for every vertex $v \in V$, i.e., $visited[v] = false$. This array maintains whether the vertex has already been visited or not.

The algorithm simply explores all vertices starting from s in a "layered" fashion. That is, first all neighbors of s (i.e., nodes at distance 1 from s) are explored; then all nodes at distance 2 from s are explored, and so on. Thus BFS tree is a layered graph with *levels*: s is at level 0 (distance 0 from itself), neighbors of s are at level 1, nodes at distance 2 from s are at level 2 and so on. In other words, nodes are explored in increasing order of distances from s, with a higher distance node will be explored only when all lower distance nodes have been finished exploring. Note that as in DFS, a node is explored only once (enforced by the *visited* array). The most natural way to keep track of the order of the nodes is a queue data structure (See Appendix B), where the neighbors of a node that is currently explored is added to the back of the queue. These neighbors include nodes at the next level and they will explored when all nodes in the current level are finished.

Algorithm 50 BFS – Breadth-First Search

Input: A connected, undirected graph G and a source node s

Output: The BFS tree rooted at s, an array containing the parent of each node and an array of distances to s

```
func BFS($G$, $s$):
    dist[$s$] $= 0$
    visited[$s$] $=$ True
    $Q =$ Queue    ▷ Initially Empty
    $Q$.ENQUEUE($s$)
    while $Q$ is not empty:
        $u = Q$.DEQUEUE()
        for $v \in Adj[u]$:
            if not visited[$v$]:
                visited[$v$] $=$ True
                dist[$v$] $=$ dist[$u$] $+ 1$
                parent[$v$] $= u$
                add edge $(u, v)$ to $T$
                $Q$.ENQUEUE($v$)
```

Run Time of BFS

The time for number of operations *before* the WHILE loop is $\mathcal{O}(n)$. We now focus on bounding the time taken to finish the WHILE loop. The number of iterations of the WHILE loop is $\mathcal{O}(n)$ since each vertex is inserted into the queue exactly once and each iteration dequeues one vertex. (Note that each ENQUEUE and DEQUEUE operation can be implemented in $\mathcal{O}(1)$ time.)

The FOR loop (inside the WHILE loop) takes time $\mathcal{O}(\deg v)$. Thus total time spent in the FOR loop is $\mathcal{O}(\sum_{v \in V} \deg v) = \mathcal{O}(m)$. Hence overall running time of BFS is $\mathcal{O}(n + m)$.

Correctness of BFS

Theorem 11.5

BFS correctly computes the (shortest path) distance from s to every node $v \in V$. When the algorithm ends, $dist[v]$ has the correct distance from s to v. Moreover, the path from s to v in the BFS tree T gives a shortest path from s to v.

Proof. Let the (correct) distance from s to a node v be denoted by $d[v]$. We show that BFS correctly computes $d[v]$ for all v.

Let $L[v]$ denote the *level* of node v in BFS tree T. $L[v]$ is defined as the *number of tree edges* from s to v. $L[s] = 0$.

We show the theorem by induction on the level of v (i.e., distance from s).

Induction hypothesis: BFS correctly computes $d[v]$ for all nodes v, such that $0 \leqslant L[v] \leqslant k$. Moreover, $d[v] = dist[v] = L[v]$ and the path from s to v in tree T is a shortest path from s to v.

Base case: Holds trivially for level 0 (i.e., vertex s), since $d[s] = dist[s] = L[s] = 0$ and the shortest path has no edges.

Induction step: Assume that the hypothesis holds for all nodes up to level k. We will show that it holds for level $k + 1$ as well.

Consider a node v at level $k + 1$, i.e., $L[v] = k + 1$. Note that $d[v] \geqslant k + 1$, since otherwise if $d[v] \leqslant k$ the induction hypothesis for v will be contradicted.

Consider the tree path $P = s, \ldots, u, v$. Since u is the parent of v in the tree, u belongs to level k and by induction hypothesis: $d[u] = dist[u] = L[u] = k$. Hence there is a path (P) from s to v having length $k + 1$. This should be a shortest path between s and v. Note that $dist[v]$ is set to $dist[u] + 1 = k + 1$. Hence, $d[v] = dist[v] = L[v] = k + 1$.

□

11.4 Worked Exercises

Worked Exercise 11.1. A graph is called *biconnected* if it does not contain any cut-vertex. The biconnected components of a graph $G = (V, E)$ is a partition of the edges into sets such that the (sub)graph formed by each set of edges is biconnected.

(a) Modify the algorithm that finds cut vertices to find biconnected components also. Show how you will modify it so that it will output the biconnected components also. Note that your algorithm should run in $\mathcal{O}(|V| + |E|)$ time. Give a clear description of your algorithm and the pseudocode and argue the running time bound.

(b) Run your algorithm on the undirected graph given in Figure 11.3. Show the main steps and output the cut vertices and the biconnected components of the graph.

Solution.

(a) Note that the algorithm for finding cut-vertices is recursive and outputs cut-vertices in a *bottom-up* manner. Refer to Algorithm 47. The main observation is when we make the recursive call FINDBICONNCOMPONENTS(w) (assume that v is the parent of w in the dfs tree, i.e, w is a neighbor of v and w is not visited yet) then (v, w) together with all edges (both tree and back edges) encountered during this FindBiConnComponents(w) call (except for edges in other biconnected components contained in subtree rooted at w) forms a biconnected component. When we find a cut-vertex v (say, after returning from one of its children, w) then all the edges that we explored from (v, w) onwards form a biconnected component (except those in other biconnected components contained in the subtree of w — notice the recursive nature of this statement). Thus we get to know the edges of a biconnected component only when after we discover a cut vertex. Hence to keep track of edges that will be explored we use a stack. We keep pushing edges on to the stack until a cut vertex is found, then we pop edges of the stack (until (v, w)). These set of edges will form a biconnected component.

The modified algorithm is given below:

(Initially, time = 0 and `visited`$[v]$ = `False` for all v. Array `num` is also initialized to zero for all v. Initially Stack S is empty.)

Algorithm 51 FindBiConnComponents – Finds biconnected components of source node.
Input: A graph G and a source node v
Output: None. Prints edges that form a biconnected component of v.

```
 1: func FindBiConnComponents($G$, $v$):
 2:     visited[$v$] = True
 3:     time += 1
 4:     low[$v$] = num[$v$] = time
 5:     for $w \in$ Adj[$v$]:
 6:         if parent[$v$] $\neq w$ and num[$w$] < num[$v$]:
 7:             $S$.Push(($v, w$))     ▷ ($v, w$) not on stack before
 8:         if not visited[$w$]:
 9:             parent[$w$] = $v$
10:             FindBiConnComponents($w$)
11:             if low[$v$] $\geqslant$ num[$v$]:     ▷ cut vertex
12:                 Print($v$)
13:                 while $y$ = $S$.ExtractMin() $\neq (v, w)$:
14:                     Print($y$)
15:             low[$v$] = min{low[$v$], low[$w$]}
16:         else:   ▷ ($v, w$) is a back edge
17:             low[$v$] = min{low[$v$], Num[$w$]}
```

Note that the above algorithm will output the biconnected components correctly regardless whether the root has one or more than one children (why?). However to output root as a cut vertex or not we have to test separately (see Section 11.2.4). In line 3 of the algorithm we are checking whether the edge (v, w) is not already in the stack before we push it. For this, we check whether w is not a parent of v and `num`$[w]$ < `num`$[v]$. (If w is a parent of v then we pushed (w, v) into the stack when we visited w.) If both these are true then (v, w) is either a back edge or a tree edge (that we are just going to explore — `num`$[w]$ will be zero). So this edge has not been explored before and we push it to the Stack.

(b) The cut vertices of the graph in Figure 11.3 are b, g, d. The 4 biconnected components are the *edges* induced by the following sets of vertices:

$$\{a, b, c, d\}, \{g, h, i\}, \{d, e, f\}, \{b, g\}$$

□

Worked Exercise 11.2. Using two applications of DFS in a directed graph, give an algorithm to check if a given directed graph is strongly connected or not. Argue the correctness of your algorithm.

Solution. As defined in Section 11.2.5, a directed graph is strongly connected, if for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u in the graph. The algorithm for testing whether a directed graph G is strongly connected is as follows.

1. Run $DFS(G, v)$ (from any source vertex v). Check if all vertices of G are in the DFS spanning tree. If there are less than n vertices in the spanning tree, then output "G is not strongly connected."

2. Reverse the directions of edges in G. That is (u, v) is an edge, then delete it and add (v, u) as an edge (note that if both (u, v) and (v, u) are in G, then nothing is really added.) Call the reversed graph G'. Run $DFS(G', v)$. Check if all vertices of G are in the DFS spanning tree. If there are less than n vertices in the DFS spanning tree, then output "G is not strongly connected."

3. Output "G is strongly connected."

The running time of the above algorithm is clearly $\mathcal{O}(n + m)$, since DFS runs in linear time. Reversing edges, can be done in linear time, by simply visiting each edge in the adjacency list.

Correctness

We show that the algorithm outputs that the graph is strongly connected if and only if the graph is indeed so. If, in Step 1, all vertices are not in the spanning tree, then clearly some vertex is not reachable from v in G and hence not visited when doing DFS from v. Thus, G is not strongly connected. If, in Step 2, all vertices are not in the DFS spanning tree, that means that some vertex (say u) is not reachable from v in G'. Since edges in G' are reversed, this means that there is no directed path from u to v in G. Thus the graph G is not strongly connected.

On the other hand, if the spanning trees in both steps include all the vertices, that means all vertices in G are reachable from v and all other vertices can reach v. Hence this means every pair of vertex can reach other — through v. For example, consider vertices x and y. There is a path from v to x and v to y. Also there is a path from x to v and y to v. Hence there is a path from x to y and from y to x. Hence the graph G is strongly connected.

□

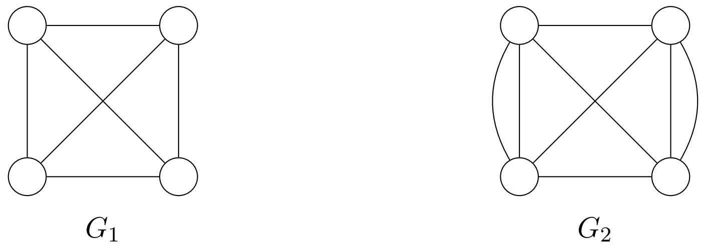


Figure 11.7: The goal is to start from some vertex s and go through each edge exactly once, returning to s. This is impossible in G_1 but is possible in G_2.

11.5 Exercises

Exercise 11.1. Consider the graph in Figure 11.1. It has two connected components, one containing the vertex g by itself and the rest of the vertices. We focus only on the latter connected component with 6 vertices — a, b, c, d, e, f — call it H.

(a) Draw the DFS tree of H starting from vertex a. What are its tree edges and back edges? Give also the DFS numbering of the vertices.

(b) List the cut vertices of H (if any). Find the *low* values of all its vertices.

(c) List the biconnected components of H.

(d) List the cut edges of H (if any).

Exercise 11.2. Consider the graph G in Figure 11.6.

(a) Draw the *DFS* tree of G starting from vertex a. What are its tree edges and back edges? Give also the DFS numbering of the vertices.

(b) List the cut vertices of G (if any). Find the *low* values of all its vertices.

(c) List the biconnected components of G.

(d) List the cut edges of G (if any).

(e) Draw the *BFS* tree of G rooted at i.

Exercise 11.3. You are given a connected undirected graph. Starting from some vertex s, the goal is to go through each edge of the graph **exactly once** and come back to the starting vertex s. This is not possible for all graphs, e.g., in the Figure 11.7 it is not possible in graph $G1$ for any starting vertex. However it is possible in graph $G2$.

(a) Prove a condition that if satisified allows one, starting from some vertex s, the goal is to go through each edge of the graph **exactly once** and come back to the starting vertex s.

(b) Give an algorithm to accomplish the above.

Exercise 11.4. Give a simple $\mathcal{O}(mn)$ time algorithm to output all the cut vertices of a graph.

Exercise 11.5. (a) A cut-edge of a graph $G = (V, E)$ is an edge whose removal paritions the graph into two or more connected components (note that some of these components can be single vertex). Give a $\mathcal{O}(|V| + |E|)$ time algorithm to find all the cut-edges of G. Give a clear description of your algorithm and the pseudocode and argue the running time bound. (Hint: Again this is a modification of the algorithm to find cut-vertices. What is the condition for an edge to be a cut-edge? Can it lie on a simple cycle of G?)

(b) Run your algorithm on the undirected graph given in Figure 11.3 of the textbook. Show the main steps and output the cut edges of the graph.

Exercise 11.6. Given an undirected connected graph $G = (V, E)$ with $|V| = n$ nodes, let u and v be two nodes in G such that their distance is greater than $n/2$, i.e., the shortest path between u and v has at least $\frac{n}{2} + 1$ edges.

(a) Show that there is at least one vertex x in the graph whose removal disconnects u and v. (Hint: Can there be a cycle in G that includes u and v?)

(b) Give an $\mathcal{O}(|V| + |E|)$-time (i.e., linear time) algorithm to find such a vertex x.

Exercise 11.7. Let $G = (V, E)$ be a directed graph. We call G to be *mildly-connected* if for all pairs of vertices $u, v \in V$, there is a path either from u to v **or** from v to u (or possibly both). Give an $\mathcal{O}(|V| + |E|)$ (i.e., linear) time algorithm to determine if a given directed graph is mildly-connected.

(Hint: First find the component graph $G' = (V', E')$ of G (i.e., the vertices of G' correspond to the strongly connected components and there is a directed edge between two such vertices $u', v' \in V'$ if there a directed edge between some vertex of the component (corresponding to) u' and some vertex of component (corresponding to) v'). (You should show how to construct the component graph in linear time.) Note that the component graph is a DAG (directed acyclic graph) (why?). What property should this DAG have for G to be mildly connected? Topological sort will prove useful now.)

Exercise 11.8. The diameter of a graph $G = (V, E)$ is the *maximum* distance among all pairs of vertices. That is if $d(u, v)$ is the distance between vertices u and v in G, $diam(G) = \max_{u,v \in V} d(u, v)$. The goal is to design an efficient algorithm to compute the diameter of a **tree** (i.e., the graph is a tree) and analyze the running time of the algorithm. Assume that you are given a rooted tree with parent-child relationships, i.e., each node, except the root, has a pointer to its parent and also pointers to its children (if any). Assume that the tree is rooted at a node with degree at least two (i.e., the root has at least two children). Note that, one does not need to assume this, since given any tree, you can choose a node with degree at least two, and do a BFS from that node and create a tree with parent-child relationships.

(i) Show that the distance determining the diameter will always be between two leaves.

(ii) Describe and analyze an $\mathcal{O}(n)$ time algorithm to compute the diameter of tree T.

Exercise 11.9. A strongly connected component of a directed graph $G = (V, E)$ is a maximal subset of vertices $S \subseteq V$ such that for every pair of vertices u and v in S, there is a directed path from u to v as well as a directed path from v to u, i.e., u and v are mutually reachable from each other. Note that "maximal" means that the set S is maximal, i.e., no more vertices can be added to S and still guarantee the mutual reachability property.

(1) Find the strongly connected components of the directed graph in Figure 11.4.

(2) Let $S_1, S_2, \ldots$ be the strongly connected components of G. Construct the "component graph" G' of the directed graph G as follows: Each node of the component graph G' represents a strongly connected component of G (i.e., there are as many nodes in G' as there are strongly connected components in G) and there is a directed

edge between two nodes (representing, say, strongly connected components S_i and S_j) if a vertex in component S_i has a directed edge to a vertex in component S_j.

Draw the component graph of the directed graph in Figure 11.4.

(3) Show that the component graph G' of any directed graph G is always a DAG. (Hint: What if there is a directed cycle in the component graph?)

(4) Consider the following algorithm for finding the strongly connected components of a given directed graph G:

1. Run the Algorithm DFSFinishTimes (Algorithm 48) on G and compute the finish times $fin[v]$ of every vertex.
2. Compute the reverse graph G^r by reversing the directions of edges in G.
3. Run the Algorithm DFSFinishTimes (Algorithm 48) on G^r, but in the for loop consider the vertices in the decreasing order of $fin[v]$ computed in Step 1.
4. Output the vertices of each tree in the depth-first forest formed in Step 3 as a separate strongly connected component (i.e., each time $DFS(G, v)$ returns, all the vertices it explores form a strongly connected component).

Run the above algorithm on the the directed graph in Figure 11.4 and show that it indeed outputs the strongly connected components. Show the steps 1-4 of the algorithm on this graph.

(5) Prove that the algorithm given above correctly outputs all the strongly connected components in linear time.

Exercise 11.10. Implement DFS as an *iterative* algorithm using the stack data structure. (Hint: This is a modification of the BFS algorithm using the stack (instead of a queue), but with a crucial difference on what to push and pop).

Exercise 11.11. Implement BFS as a *recursive* algorithm. Give pseudocode.

Exercise 11.12. An undirected graph $G = (V, E)$ is called *bipartite* if it contains no *odd* cycle, i.e., no cycle has an odd number of edges. Note that a graph with no cycles is also bipartite. We are going to show an important property of bipartite graphs: its vertex set can be partitioned into two disjoint sets A and B, i.e., $A \cup B = V$ and $A \cap B = \phi$, such that all the edges are between vertices in A and vertices in B — in other words, there are no edges between any two vertices in A and any two vertices in B.

The goal of this problem is that, given a *connected bipartite* graph $G = (V, E)$, to find a partition of the vertex set into A and B such that all edges are between some vertex in A and some vertex in B.

(a) Do Breadth-First Search (BFS) on G from some starting vertex, say a, and compute the distance (i.e., the level in the BFS tree) of every vertex v to a. Show that if G is bipartite then there are no edges between vertices in the same level. (Hint: Use the odd cycle property of bipartite graph).

(b) Show that you can output the sets A and B in linear time, i.e., $\mathcal{O}(|V| + |E|)$ time.

12 GRAPH ALGORITHMS: MST AND SHORTEST PATHS

In this chapter, we study two fundamental graph optimization problems, namely the minimum spanning tree (MST) and the shortest paths problem. Both problems have numerous real-world applications and also arise as subroutines in solving other problems (including non-graph problems that can be cast as graph problems). This chapter uses techniques from both greedy algorithms (Chapter 7) and dynamic programming (Chapter 6). We use greedy technique to solve MST efficiently, while for shortest paths, both greedy and dynamic programming are used. It will be helpful to review the techniques of both greedy and dynamic programming while reading this chapter. MST and shortest paths are two further examples of applying the above algorithmic techniques in the context of graph problems.

12.1 Problem: Minimum Spanning Tree (MST)

The MST problem is defined as follows. Given a weighted undirected connected graph $G = (V, E, w)$ (w is the weight function assigning weights to edges), the goal is to find a spanning tree of G (see Appendix E.7) with *minimum total weight* (i.e., sum of the weights of its edges is minimum). MST is a fundamental graph optimization problem with applications in many settings. For example, consider a communication network modelled as a weighted graph. Edge weight may represent the cost of sending a message through the edge. Suppose we want to repeatedly "broadcast" messages from one node to all other nodes in the network so that the total cost (i.e., the sum of the costs of the edges used in the broadcast) is minimized. A MST serves as a backbone for efficient broadcasting in a communication network. Broadcasting via a MST minimizes the total cost.

In this section, we will design two *greedy* algorithms for this problem.

Figure 12.1 shows a weighted graph and its MST.

12.1.1 Kruskal's Algorithm

We first consider a classical algorithm called Kruskal's algorithm which is a simple greedy algorithm. We are given a weighted (undirected) connected graph G with n nodes and m edges. The idea of the algorithm is as follows. consider the edges of G in order of non-decreasing weight; this makes sense since we want to add edges with as small weight as possible. We add edges one at a time to the MST. Note that we need to add

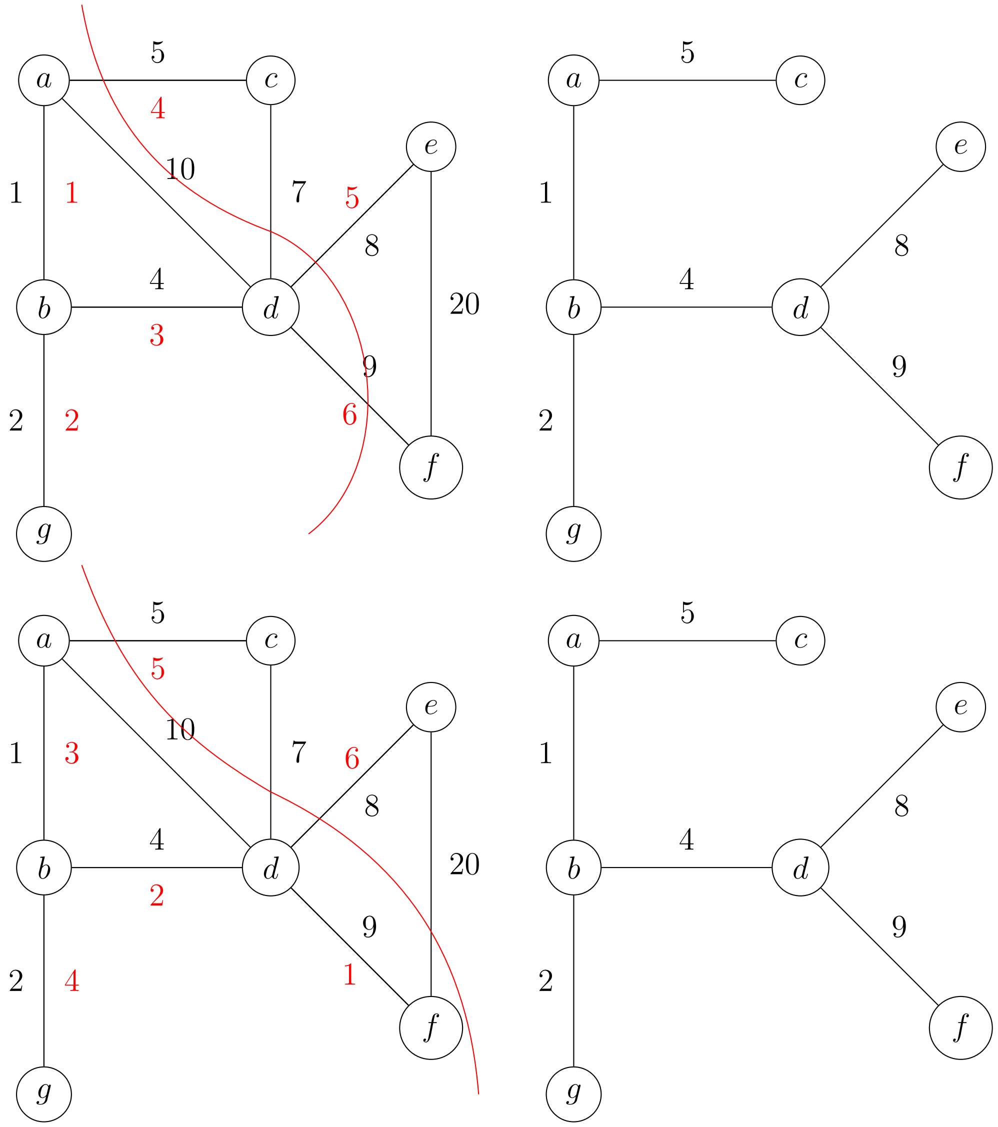


Figure 12.1: A connected weighted graph (left) and its MST (right). Note that MST is a spanning tree and hence will have exactly $n-1$ edges on a connected graph of n nodes. (Top) Kruskal's algorithm, starting from an empty tree, adds edges in the order of increasing weight, starting from the smallest weighted edge, unless the added edge closes a cycle with edges already added. In this example, Kruskal adds edges in the following order (shown in red): $(a,b),(b,g),(b,d),(a,c),(d,e),(d,f)$. The cut that validates the addition of edge (a,c) is the partition (shown by red line) — $(\{a,b,d,g\},\{c,e,f\})$, because (a,c) is the lightest edge crossing this cut. (Bottom) Prim's algorithm, starting from a tree with one vertex, adds edges so that the tree is extended by one additional vertex at a time. The edge that connects a non-tree vertex to a vertex in the tree via the least-weighted edge is added. In this example, Prim adds edges in the following order (shown in red) starting from a tree with the singleton vertex f: $(f,d),(b,d),(a,b),(b,g),(a,c),(d,e)$. The cut that validates the addition of edge (a,c) is the partition (shown by red line) — $(\{f,a,b,d,g\},\{c,e\})$ because (a,c) is the lightest edge crossing this cut.

exactly $n - 1$ edges (since any spanning tree should have exactly those many edges — See Theorem E.2 in Appendix E). Initially the set of tree edges is empty. The algorithm simply adds edges one by one to the tree in the above order if it does not close a cycle with edges already added to the tree. Figure 12.1 shows a weighted graph and the order in which edges are added by the Kruskal's algorithm.

The above algorithm is greedy because, once an edge is added to the tree it is never removed (no backtracking). We will show next that the above algorithm indeed produces a MST, i.e., the added edges induces a MST of the input graph. The algorithm's pseudocode is given below.

Pseudocode for Kruskal's Algorithm

Algorithm 52 KruskalHighLevel – A High Level Overview of Kruskal's MST Algorithm
Input: A weighted graph G
Output: The MST of G

```
func KruskalHighLevel($G$):
    $T = \emptyset$
    Sort the edges in increasing order of weight.
    for $e$ in edges:
        if $e$ does not form a cycle with edges in $T$:
            $T$.Add($e$)
        if $|T| == n - 1$:    ▷ $T$ is spanning after adding $n - 1$ edges
            return $T$
```

Correctness of Kruskal's algorithm

It is clear that Kruskal's algorithm produces a spanning tree (why?). We next show that it produces a spanning tree whose weight is minimum.

Recall the following technique which was used earlier (see Chapter 7) in proving optimality of greedy algorithms. Compare the greedy solution with the optimal. If the two solutions differ, then find the first x_i where they differ. Then show how to make x_i in the optimal solution equal to that of the greedy solution *without losing optimality*. Repeated use of this transformation shows that the greedy solution is optimal. This proof idea can be called as an "exchange" or "transform" argument. We will use an "exchange" argument to show that the greedy algorithm produces the optimal.

Correctness Proof of Kruskal's Algorithm

Theorem 12.1

Kruskal's algorithm produces a MST.

We need the following definition for the proof. A *cut* $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V. An edge *crosses* the cut if one of its endvertices is in S and the other is in $V - S$.

Proof. We will show that the tree T output by Kruskal's algorithm has minimum weight among all spanning trees in G.

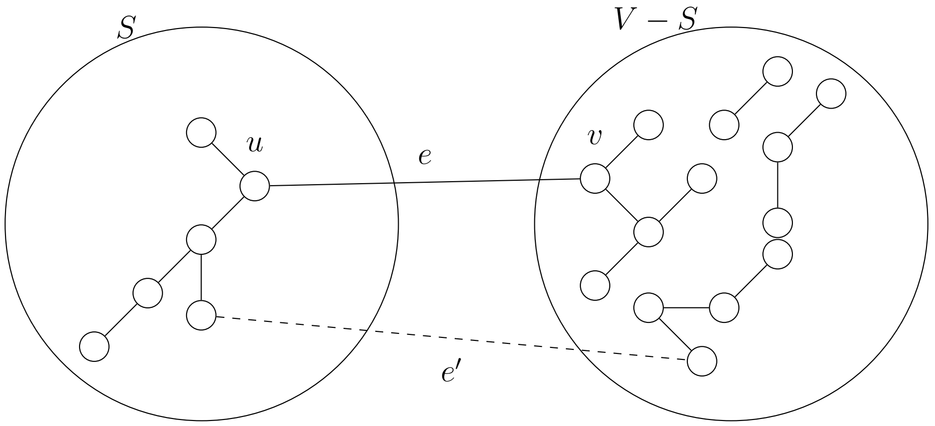


Figure 12.2: Illustration of proof of Kruskal's algorithm. $e = (u, v) \in T$ crosses the cut $(S, V - S)$. Note that S is the set of all nodes connected to u using edges already added in the tree T until now. If $e \notin T'$, then the optimal tree T' should have another edge e' crossing the cut. This edge e' should have weight no less than the weight of e, since Kruskal adds edges in increasing order of weight.

Let T' be a MST of G. We will show either $T' = T$ or we can transform T' to T without losing optimality. Sort edges of T' in increasing order of weight. Let $e = (u, v)$ be the *first* edge added by Kruskal that is not in T'. Let U be the set of edges in T *just before* e is added. Just *before* e is added consider the cut $(S, V - S)$, where S is the set of all vertices that are connected to vertex u using (only) edges of U. Since e is added by the algorithm, this means that $v \in V - S$ (otherwise, if $v \in S$, this means by our construction of S, v will close a cycle with u). That is, e crosses the cut $(S, V - S)$. See Figure 12.2.

The optimal tree T' also should have at least one tree edge crossing the cut. (If no edges of T' cross the cut, then the tree is not connected.) Let $e' \in T'$ cross the cut. We note that $w(e') \geqslant w(e)$. This is because, e' cannot be e or any of the other edges added before e, since all such edges do not cross the cut (by construction of the cut $(S, V - S)$). Remove e' from T' and add e to get a spanning tree T''.

Why is T'' a spanning tree? Removing e' from T' disconnects T' into two connected components — one containing u and the other containing v. Adding $e = (u, v)$ connects them back.

Now: $w(T'') = w(T') - w(e') + w(e) \leqslant w(T')$, since $w(e') \geqslant w(e)$. Thus T'' is also a MST, but it has one more edge (e) which is in common with T. Repeat the argument by comparing T and T''. □

Algorithm 53 Kruskal – Kruskal's MST Algorithm
Input: A weighted graph $G = (V, E)$
Output: The MST of G

```
func Kruskal($G$):
    $T = \emptyset$
    for $v \in V$:
        MakeSet($v$)    ▷ create a singleton set for each vertex
    for edges $e_i = (u_i, v_i)$ in increasing order of weight:
        $\ell_1 =$ Find($u_i$)
        $\ell_2 =$ Find($v_i$)
        if $\ell_1 \neq \ell_2$:
            ▷ Since $u_i$ and $v_i$ belong to different trees, $e_i$ does not close a cycle with edges already in $T$
            $T$.add($e_i$)
            Union($\ell_1, \ell_2$)    ▷ combine the two trees
        if $|T| == n - 1$:
            return $T$
```

Implementing Kruskal's Algorithm Efficiently

Assume that we take the graph as an *adjacency* list. Since we process the edges in increasing order of weight, we can first sort all the edges. This takes $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$ time, where, m is the number of edges and n is the number of nodes of G. The main non-trivial operation is the condition in line 4.1 — checking for a cycle.

We can implement line 5 of KruskalHighLevel in $\mathcal{O}(|V|)$ time using a suitable data structure as explained below. Thus, the algorithm can be implemented in $\mathcal{O}(|E| \log |V|)$ time.

Note that Kruskal's algorithm can be viewed as follows. It maintains a dynamic collection of connected components (each component is a tree) at every step. Initially, every vertex is a tree by itself (i.e., singleton nodes). In each step, Kruskal's algorithm adds an edge that does not close a cycle with the edges added, i.e., it adds an edge between two *different* trees and *joins them into one tree*. Each edge reduces the number of trees by one — thus starting with n trees, after adding $n - 1$ edges we are left with one tree which is the MST.

To efficiently check whether an edge (u, v) closes a cycle in each step, we need to efficiently check if u and v belong to the same tree or not — in the former case, it closes a cycle, and in the latter case it does not. Further, in the latter case, it joins two trees and makes them as one tree and thus the number of trees is reduced by one. We need a data structure to maintain the different trees dynamically as they change so that we can efficiently check whether an edge (u, v) connects two different trees or not. We accomplish this using the following general data structure which is known as a *union-find* data structure (see Appendix B.4), since it implements two operations in a collection of disjoint sets (here, each set is a tree):

- *find*: Given an element u, find(u) finds the set, i.e., the representative of the set, where u belongs to. Note that each set has a unique representative (or leader).

- *union*: Given two elements u and v belonging to two different sets, union(u, v) joins the two sets into one set.

A union-find data structure also supports a third operation (for initializing the sets) called Make-Set(v) which creates a singleton set with element v. We initially call Make-Set on each vertex v to create n singleton sets — each representing a tree by itself.

Appendix B.4 shows a way to implement the union-find operations so that any find or union operation takes $\mathcal{O}(\log n)$ time, where $n = |V|$ is the total number of elements in the set. In the case of Kruskal, n is the number of nodes.

Note that we can implement Kruskal's algorithm efficiently by using a union-find data structure to maintain the disjoint trees in every step. To check whether an edge (u, v) closes a cycle, we check whether find(u) = find(v), i.e., whether u and v belong to the same set (i.e., same tree), by checking whether their representatives are the same. If they are different, then u and v belong to different trees and hence does not close a cycle. In that case, we can do union(u, v) to join the two trees into one tree. Both operations take $\mathcal{O}(\log n)$ time with the implementation given in Appendix B.4. Hence the whole algorithm can be implemented in $\mathcal{O}(m \log n)$ time. Algorithm 53 gives a pseudocode for Kruskal's algorithm using the union-find data structure.

12.1.2 Prim's Algorithm

Prim's algorithm (which is essentially the same as Dijkstra's algorithm for shortest paths — see Section 12.2.1) builds the MST one edge at a time starting from some vertex which will be the root of the tree. Let G be a connected (undirected) graph and let r be some vertex in G. The algorithm builds a MST A rooted at r (initially A has just one vertex, the root). Unlike Kruskal, it will maintain only one tree and in every step will add one new vertex to the tree — in other words, an edge that connects a vertex in the tree to a vertex outside the tree. The edge that is chosen to be added will be *the minimum weight edge connecting the tree A to a vertex not in the tree.* Worked Exercise 12.1 shows that it is safe to add this edge to the MST. Figure 12.1 shows a weighted graph and the order in which edges are added by the Prim's algorithm.

During the algorithm, all vertices that are **not** in the tree are kept in a min-heap data structure Q based on a *key* defined as: *key*$[v]$ is the *minimum weight of any edge connecting v to a vertex in the tree.* If there is no such edge then *key*$[v] = \infty$. $\pi[v]$ names the parent of v in the tree.

Note that *key*$[v]$ has to be updated after every iteration. Suppose u is the vertex added in the current iteration. Then clearly, the only key values that need to be updated are the neighbors of u. That's what Prim's algorithm does.

The tree is kept implicitly as

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}.$$

When the algorithm terminates Q is empty and the MST is given by

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

At every step of the Prim's algorithm, one new vertex is added to the tree rooted at r. The new vertex u that is added has the smallest *key* value that is defined as above. The key values are kept in a heap data structure and hence the smallest key value can be obtained by doing an ExtractMin operation on the heap data structure. Note that once a vertex u is added to the tree, the key values of its neighbors may have to be updated (i.e., decreased), since their key values can decrease because u is now part of the tree. For a neighbor v of u, if $w(u, v)$ is smaller than its current key value, i.e., if $w(u, v) < key[v]$, then $key[v]$ is set to $w(u, v)$, since v is connected to a tree vertex, i.e., u, via smaller weight. The decrease of key value of v which is in the heap can be accomplished using a DecreaseKey operation, which decreases the key value of a given key to a new value (See Appendix B.1).

Pseudocode for Prim's Algorithm

Algorithm 54 Prim – Prim's MST Algorithm
Input: A graph G and source vertex r
Output: The MST of G

```
1:  func Prim($G$, $r$):
2:      for $v \in G$:
3:          key$[v] = \infty$
4:          $\pi[v] =$ Null
5:      key$[r] = 0$
6:      $Q =$ BuildHeap($G$)  ▷ Create a min-heap of nodes in G
7:      while $Q$ is not empty:
8:          $u = Q$.ExtractMin()
9:          for $v \in$ Adj$[u]$:
10:             if $v \in Q$ and $w(u, v) <$ key$[v]$:
11:                 $Q$.DecreaseKey($v, w(u, v)$)
12:                 $\pi[v] = u$
13:     return $\{(v, \pi[v]) \mid v \in V - \{r\}\}$
```

Running time

The running time of Prim's algorithm depends on the heap implementation. We observe that the algorithm uses $\mathcal{O}(n)$ ExtractMin operations and $\mathcal{O}(m)$ DecreaseKey operations. Since each node is extracted exactly once (once a node is removed from the heap, it is never inserted again in the heap), there are n ExtractMin operations. For each node u, there are $deg(u)$ DecreaseKey operations, where $deg(u)$ is the number of neighbors of u. This is because, when a node u is extracted from the heap, all of its neighbors are examined exactly once in the for loop (Line 8 of the Prim's algorithm). Hence the total number of DecreaseKey operations is $\sum_{u \in V} deg(u) = \mathcal{O}(m)$.

If we use a binary heap implementation for the heap data structure (Appendix B.1), then each ExtractMin and DecreaseKey operation takes $\mathcal{O}(\log n)$ time. Thus overall, this yields a $\mathcal{O}(n \log n + m \log n) = \mathcal{O}(m \log n)$ time (since $m \geqslant n - 1$, as the graph is connected). Note that we assume that we are given the location of the key in the heap to implement the heap operations. This can be accomplished in constant time by maintaining an index (or pointer) to every vertex in the binary heap. Thus locating a key in the heap takes only $\mathcal{O}(1)$ time.

12.2 Problem: Shortest Paths

The shortest paths problem is one of the most important graph problems. In this problem, we are given a *directed* graph $G = (V, E)$ and a *weight* function $w : E \to Q$. The weight of a path $p = v_0, v_1, v_2, \ldots, v_k$ is defined as the sum of the weights of the edges in the path:

$$w(p) \;=\; \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The weight of the *shortest path* from u to v, $\delta(u, v)$ is the minimum of $w(p)$ for all p connecting u to v, and ∞ if there is no such path in G.

Various Shortest Path Problems

There are many types of shortest path problems:

- *Single Source Shortest Paths (SSSP)*: Compute shortest paths from a given *source* to all vertices in the graph.
- *Single Destination Shortest Paths (SDSP)*: Compute shortest paths to a given *destination* from all vertices in the graph.
- *Single Pair Shortest Path (SPSP)*: Compute shortest path for a given *pair* of vertices.
- *All Pairs Shortest Paths (APSP)*: Compute the shortest paths for *all pairs* of vertices in the graph.

Negative edge weights are allowed, but the problem is not well defined when a **negative weight cycle** is present. A negative weight cycle is one which is a cycle in the graph and whose sum of weights of the edges of the cycle is less than zero. If the negative cycle is reachable from the source vertex, then by using the negative cycle path many times, one can reduce the shortest path distance to $-\infty$. Hence we assume that there is no negative weight cycle, or at least it is not reachable from the source vertex from which we need to find shortest paths, as in the SSSP or SPSP problems.

We present various important algorithms for all the above problems. For the SSSP problem, we present two classical algorithms, namely *Dijkstra's* and *Bellman-Ford's* algorithms. Note that the SDSP problem can be solved by reversing the directions of the edges and solving the SSSP problem with the given destination as the source.

For the SPSP problem, where the goal is to find a shortest path from a given source to a given destination, we present an algorithm called A^* that is a variant of Dijkstra's algorithm. Although A^* performance may not be better than Dijkstra's algorithm in the worst-case, in many instances, especially in real-world problems, it can perform significantly better than Dijkstra's.

One can solve the APSP problem, by solving n instances of the SSSP problem, one from each node as a source. While this can be faster in some cases, it can be much slower in others. We present a well-known dynamic programming algorithm called *Floyd-Warshall* algorithm for this problem.

An important property of shortest path

Shortest path has the following important property which shows that a shortest path has the following *optimal substructure property* that we studied in Dynamic Programming (Chapter 6). The property says that a shortest path contains within it other shortest paths. This property enables both greedy and DP algorithms for shortest paths.

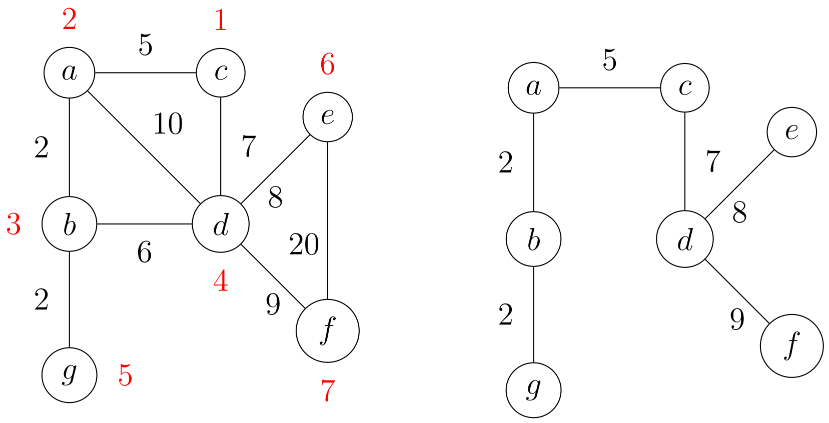


Figure 12.3: A weighted graph and its shortest path tree (SPT), rooted at source node c. Dijkstra's algorithm adds vertices in increasing order of distances from the source. In this example, the order (shown in red) in which vertices are added to the SPT is: c, a, b, d, g, e, f.

Lemma 12.2 ▶ Optimal Substructure Property

If $p = v_0, v_1, \ldots, v_j, \ldots, v_k$ is a shortest path from v_0 to v_k, then $p' = v_i, v_1, \ldots, v_j$ is a shortest path from v_i to v_j.

Proof. Assume that p'' is a shorter path from v_i to v_j, then $v_0, \ldots, v_{i-1}$ followed by p'' and then $v_{j+1}, \ldots, v_k$ would be a shorter path from v_0 to v_k which contradicts the optimality of p. □

12.2.1 Single Source Shortest Paths Problem

The Single Source Shortest Path Problem (SSSP) is a classical and important problem. The problem is to compute shortest paths from a given source vertex s to all vertices in the graph. This problem has many applications, e.g., finding the shortest route in a map, or finding routes in the Internet. Dijkstra's algorithm is used in the Internet to find short paths for routing.

Figure 12.3 shows an example of a weighted graph and its shortest path tree (SPT), defined next.

12.2.1.1 Dijkstra's Algorithm

The algorithm constructs a *tree of shortest paths* (rooted at s) in a *greedy* manner. This is called a *shortest paths tree (SPT)*. The root of the tree is s. A path on the tree corresponds to shortest paths to s for all vertices on that path. The shortest paths tree is constructed in $|V|$ iterations.

Dijkstra's algorithm computes the distances from source s to other vertices in *increasing order of their distances* from s. Starting from $S = \emptyset$ and extending S by one vertex in each iteration, the algorithm computes a shortest paths tree restricted to vertices only from S. When $S = V$ we get the shortest paths tree for the graph.

As mentioned earlier, Dijkstra's algorithm has the *same structure* as Prim's algorithm. The main difference is in the interpretation of the key values — here we call them d (distance) values, since at the end of the algorithm they hold the correct shortest path distances to the source. (In fact, by changing the interpretation of key values, one can modify Dijkstra to compute solutions of other problems as well, e.g., see Worked Exercise 12.4.) In Dijkstra's algorithm, the distance value of a non-tree node (i.e., belonging to $V - S$) is the shortest distance to the source vertex using only vertices of S (i.e., tree vertices). It is easy to see that at the start of the algorithm the above interpretation of distance values holds since the d values of all nodes is ∞, except the source which is 0. Dijkstra's algorithm efficiently updates the distance values as a vertex is added (one at a time); if a vertex is added to the tree, only the distance values of its neighbors (outside the tree) are affected and these are updated. Dijkstra makes the greedy choice of adding the vertex outside the tree with the smallest distance value (i.e., one that has the shortest distance to the source using only the tree vertices) as the vertex that has to be added next. We show that this greedy choice is correct by showing a proof using mathematical induction.

Data structures used in Dijkstra's Algorithm

This is similar to the Prim's algorithm but, of course, with a different interpretation of key values — here these are the distance values.

For all $v \in V$, and in each iteration:

- $d[v]$ — the distance from s to v by a path that uses only vertices of S.
- $\pi[v]$ — the predecessor of v on the tree.
- Q — min heap that maintains a list of vertices *not in* S.
- ExtractMin(Q) — the vertex with smallest $d[v]$ among the vertices in Q.

Pseudocode of Dijkstra's Algorithm

Algorithm 55 Dijkstra
Input: A weighted graph G and starting node s
Output: The Shortest Path Tree for G

```
func Dijkstra($G$, $s$):
    for $v \in G$:
        $d[v] = \infty$
        $\pi[v] =$ Null
    $d[s] = 0$
    $S = \emptyset$
    $Q =$ BuildHeap($G$)
    while $Q$ is not empty:
        $u = Q$.ExtractMin()
        $S$.Add($u$)
        for $v \in$ Adj[$u$]:
            newDist $= d[u] + w(u, v)$    ▷ Distance of $v$ to $s$ via $u$
            if newDist $< d[v]$:
                $Q$.DecreaseKey($v$, newDist)
                $\pi[v] = u$
```

Correctness of Dijkstra's Algorithm

Theorem 12.3

Dijkstra's algorithm computes a correct shortest paths tree when applied to a graph G with *no negative weight* edges.

Proof. We will show by induction on the size of S that for every $1 \leqslant k \leqslant n$, at the end of the while loop with $|S| = k$, the functions $\pi[]$ and $d[]$ satisfy:

1. If $v \in S$ then $d[v]$ is the shortest path distance of v from s, and $\pi[v]$ encodes the last edge of that path.
2. If $v \notin S$ then $d[v]$ is the shortest path of s from v using only vertices of S, and $\pi[v]$ encodes the last edge of that path.

The induction hypothesis holds for $|S| = 1$ since in that case $S = \{s\}$, and for all v either $d[v]$ is ∞ or is the weight of the edge (s, v).

Assume that the induction hypothesis holds for $|S| = j - 1$. Consider the j-th iteration of the while loop. We next prove of the part 1 of the induction step for the j-th iteration.

Lemma 12.4

The shortest path from s to u uses only vertices of S. Note that u is the vertex selected in line 9 of the algorithm.

Proof. Assume that a shorter path from s to u contains a vertex in Q. Let v be the first such vertex, then $d[v] < d[u]$ (since all weights are non-negative — this is where we make use of the assumption that all weights are non-negative). Thus, v would have been selected instead of u (in line 9) of the jth iteration which is a contradiction. □

The above lemma, in conjunction with the second part of the induction hypothesis of the $(j-1)$th iteration implies the truth of the part 1 of the induction hypothesis of the jth iteration. This is because, by the hypothesis, at the end of the $(j-1)$th iteration, u has the correct $d[u]$ distance value to s considering only vertices in S. Lemma 1 says that just considering the vertices of S is enough.

Next we prove part 2 of the induction step (for the j-th iteration).

Lemma 12.5

If vertex $v \in Q$ had a correct value $d[v]$ at the beginning of the while iteration (which is guaranteed by the induction hypothesis), it has a correct value at the end of the iteration.

Proof. By the lemma's assumption we need only to check paths from s to v that contain u, since the induction hypothesis guarantees that paths don't include u (which was outside S before the beginning of this iteration) are correct.

The algorithm checks only paths in which v is adjacent to u and updates their distance values. See Figure 12.4. How about the remaining paths?

If there is a shorter path

$$P = s, v_1, v_2, \ldots, u, v_k, \ldots, v$$

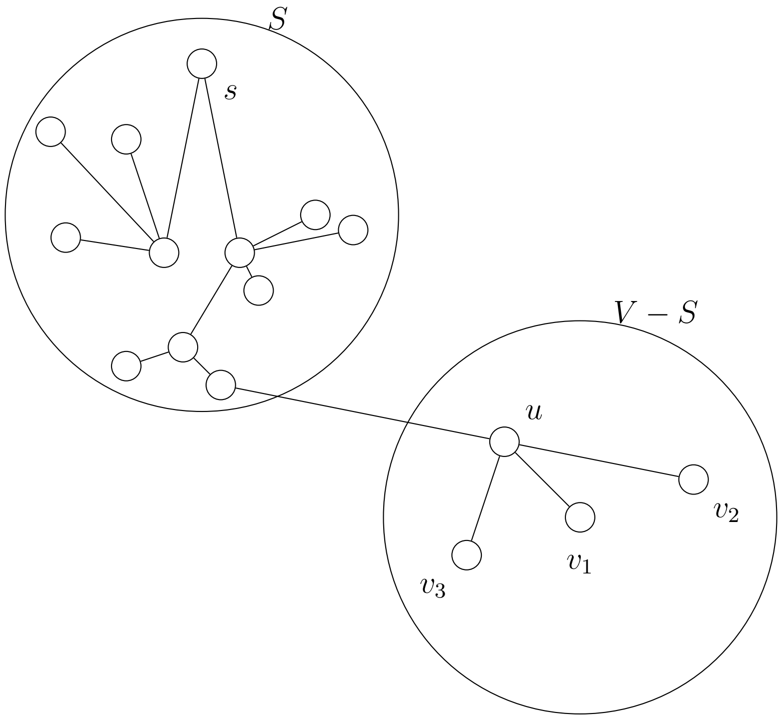


Figure 12.4: Illustration of proof of Dijkstra's algorithm. u is the node that has the smallest d value. Hence it is the one that is extracted from the heap and added to the tree (i.e., made part of S). The algorithm updates the d values of only the non-tree neighbors of u (i.e., part of $V - S$).

with u not adjacent to v, then since v_k *joined* S *before* u, there must be a path from s to v_k that does not use u and is not longer. Thus, $d[v]$ has the correct value (by the premise of the Lemma) without considering paths that use u. □

Thus, part 2 of the induction step is satisfied. □

Running time

The running time of Dijkstra is exactly the same as that of Prim.

Hence Dijkstra's algorithm can be implemented in $\mathcal{O}(m \log n + n \log n) = \mathcal{O}(m \log n)$ time using a binary heap.

12.2.1.2 A^* algorithm

The A^* algorithm is a heuristic variant of the Dijkstra's algorithm that is especially well-suited for the Single Pair Shortest Paths (SPSP) problem where the goal is, given a weighted directed graph, and a pair of vertices s and t, to find a shortest path between s and t. This, in principle, can be solved faster than the SSSP problem since instead of finding shortest paths from a source s to all vertices, the goal is to find a shortest path to a single (target) vertex t. The A^* algorithm is a good candidate for this problem and has many real-world applications, for example, finding the shortest route between two places in a map or finding the shortest route for a robot from a given source to a destination. In fact, the A^* algorithm was invented for robot motion planning.

The only modification of the A^* algorithm compared to the Dijkstra's algorithm is simply changing the key value of a (non-tree) node v to incorporate *an estimate* of the shortest path distance from v to the target t. For the A^* algorithm to work (i.e., find the shortest path from source to target), the estimate should be a *lower bound* on the true distance from v to t. Let $h(v)$ be such an (heuristic) estimate of the true distance from v to t. This estimate may depend on the underlying graph and other factors. For example, in road networks, such an estimate can be obtained by using the Euclidean distance between two points which will be always be a lower bound on the shortest path distance between the two points.

The key value in the A^* algorithm is the sum $d[v] + h[v]$, where $d[v]$ is the same as in the Dijkstra's algorithm, the shortest distance from source s to v using only vertices in the tree. The rest of the algorithm is the same as before. At every step, the node with the smallest key value is chosen and added to the tree. The algorithm ends when the target is added to the tree. It can be shown that if the function $h(v)$ is a lower bound on the true distance between a node to the target (such a function is called *admissible*), then the correct shortest path distance to the target is found. For example, $h[v] = 0$ is a trivial lower bound which is simply the Dijkstra's algorithm.

It is important to note that unlike Dijkstra's algorithm, when a node (other than t) is added to the tree, its key value may not be the true estimate of the shortest distance of the source s to the node. However, if $h[v]$ is admissible, i.e., never overestimates the true distance of v to target t, it is guaranteed that when t is included in the tree, its key value will be the correct shortest path distance and the path from s to t is the shortest path from s to t (Exercise 12.14 asks you to show this). The advantage of A^* is that, depending on how good $h[v]$ estimates the true distance to the target, substantially less nodes may need to be explored before a shortest path to t is found. Dijkstra, on the other hand, may explore more nodes before it finds the shortest path to t. Recall that, Dijkstra find the shortest paths in increasing order of distances from s, *without considering* distance to target. Since A^* makes use of additional information in the form of $h[v]$ it can find the shortest path to t quicker than Dijkstra (by exploring less number of nodes).

12.2.1.3 Bellman-Ford Algorithm

Bellman-Ford algorithm is a *dynamic programming* based algorithm for single-source shortest paths. It computes single source shortest paths even when some edges have negative weight. The algorithm detects if there are negative cycles reachable from s. If there are no such negative cycles, it returns the shortest paths. Like Dijkstra, the Bellman-Ford algorithm is also used in the *Internet* to find short paths.

DP formulation

As is typical in DP algorithms (see Chapter 6), we first need to identify the subproblems and then write the recursive formulation.

Let $w(u, v)$ denote the weight of edge (u, v). Let $d^k[v]$ to be the weight of the shortest path between source vertex s and vertex v using paths of *up to* k edges. The subproblems are computing $d^k[v]$ for $1 \leqslant m \leqslant n - 1$ and for all $v \in V$. If there are no negative weight cycles then no shortest path has more than $n - 1$ edges, i.e., k can be at most $n - 1$.

When $k = 0$, we have

$$d^{(0)}[v] = \begin{cases} 0 & v = s \\ \infty & v \neq s \end{cases}$$

For $k > 0$, we can write a recursive formula:

$$d^{(k)}[v] = \min_{u \in \text{Neighbor}(v)} \left\{ d^{(k-1)}[u] + w(u, v) \right\}$$

where Neighbor(v) is the set of all neighboring vertices of v.

The idea of the above formula is as follows. To compute $d^{(k)}[v]$ we look at v's neighbors and its $d^{(k-1)}[.]$ values. In particular, for a neighbor u, its $d^{(k-1)}[u]$ value holds the shortest distance from the source to u using at most $k - 1$ edges. Hence adding the weight of the edge, $w(u, v)$, connecting u and v will give the shortest distance to the source from v using at most k edges. Since we do not know which neighbor of v through which the shortest path to the source occurs, we consider all neighbors of v and take the minimum.

Intuition behind the correctness of Bellman-Ford

As can be seen from the formulation of the subproblems and the recursive formula, the main idea behind Bellman-Ford is to build the shortest path incrementally one edge at a time. Note that any shortest path can have at most $n - 1$ edges. Otherwise, a vertex will be repeated in the path, implying a cycle in the path, which contradicts the optimality of the shortest path, since, one can remove the cycle (assuming the cycle is not of negative weight) from the path and get a shorter path. Hence, unless there is a negative weight cycle, one need not consider paths longer than $n - 1$ edges. Bellman-Ford as shown in the recursive formulation builds the shortest path incrementally in terms of the length of the path.

How does Bellman-Ford detect a negative-weight cycle that is reachable from the source? The idea is unless there is a negative weight cycle, since all paths will have length at most $n-1$ edges, the correct distances will not change once the path length has reached $n - 1$ (the correct distance could have been reached even earlier, if the shortest path has lesser number of edges). Hence, if one tries to find a shortest path of length n (i.e., having n edges), this should not change the distance, unless there is a negative weight cycle. So if $d^n[v]$ is strictly smaller than $d^{n-1}[v]$, for any v, then this implies that there is a negative weight cycle.

The formal correctness of the above recursive formula can be shown by induction — we refer to Theorem 12.7.

The Algorithm

The algorithm has two parts:

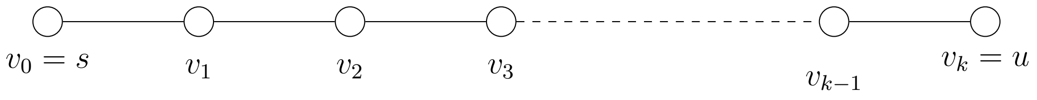
- **Part 1:** Computing Shortest Paths Tree (SPT): $|V| - 1$ iterations, iteration i computes the shortest path from s using paths of up to i edges.

- **Part 2:** Checking for Negative Cycles.

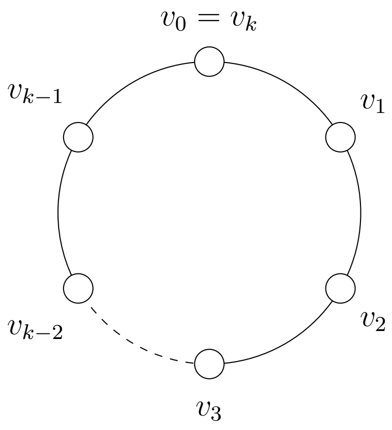
Pseudocode of the Bellman-Ford Algorithm

Algorithm 56 BellmanFord – Bellman-Ford Algorithm
Input: A weighted graph $G = (V, E)$ and source node s
Output: If the graph contains a negative cycle, `False`, else the SPT rooted at s

```
func BellmanFord($G$, $s$):
    for $v \in G$:
        $d[v] = \infty$    ▷ $d[v] = d^{(k)}[v]$ in iteration $k$
        $\pi[v] =$ Null    ▷ $\pi[v]$ is the parent of $v$ in SPT
    $d[s] = 0$
    for $i = 1$ to $|V| - 1$:
        for $(u, v) \in E$:
            newDist $= d[u] + w(u, v)$
            if newDist $< d[v]$:
                $d[v] =$ newDist
                $\pi[v] = u$
    for $(u, v) \in E$:
        if $d[v] > d[u] + w(u, v)$:
            return False    ▷ Negative weight cycle has been found
    return $d$
```



(a) $P = v_0, v_1, \ldots, v_k$, where $v_0 = s$ and $v_k = u$ is a shortest path from s to u.



(b) $v_0, \ldots, v_k$ is a negative weight cycle, i.e., a cycle whose sum of weights is negative.

Figure 12.5: Illustration of proof of Bellman-Ford algorithm.

Run Time

Theorem 12.6

The run time of the algorithm is $\mathcal{O}(mn)$.

Proof. The initialization (2-4) takes $\mathcal{O}(n)$. The path creation (6-11) takes $\mathcal{O}(mn)$. The negative cycle detection (12-14) takes $\mathcal{O}(m)$. Hence, overall the run time is $\mathcal{O}(mn)$. □

Correctness of Bellman-Ford Algorithm

Theorem 12.7

Assuming that G contains no negative cycles reachable from s, the algorithm computes the shortest paths for all vertices of G.

Proof. Fix a vertex $u \in V$, we first prove that the algorithm computes the correct (shortest path) distance from s to u. The correctness of the shortest path that achieves this distance follows easily.

Let $P = v_0, v_1, \ldots, v_k$, where $v_0 = s$ and $v_k = u$ be a shortest path from s to u. See Figure 12.5. Since there are no negative cycles P is a simple path, $k \leqslant |V| - 1$.

We prove by induction on i that after the ith iteration of the (3) loop, the algorithm computes the shortest path for v_i.

The hypothesis holds for $v_0 = s$. Assume that it holds for $j \leqslant i - 1$. After the ith iteration, we have:

$$d[v_i] \leqslant d[v_{i-1}] + w(v_{i-1}, v_i)$$

Since P is a shortest path from s to v_i, and the RHS is the distance between s to v_i on that path, it follows that $d[v_i]$ is the shortest path distance from s to v_i.

□

Theorem 12.8

The algorithm returns `True` if there are no negative cycles reachable from s, otherwise it returns `False`.

Proof. Assume that there are no negative cycles reachable from s, then by the previous theorem, the algorithm returns a shortest path tree, and $d[v]$ is the weight of the shortest path to s. Thus, all inequalities in 4(a) do not hold.

Assume that there is a negative weight cycle $v_0, \ldots, v_k$ reachable from s ($v_0 = v_k$). See Figure 12.5.

Since the path is reachable from s the values $d[v_i]$ are defined.

$$\sum_{i=1}^{k} d[v_{i-1}] = \sum_{i=1}^{k} d[v_i]$$

and

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

Thus,

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

So there must be an i such that

$$d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$$

and the algorithm returns False. □

12.2.2 All-Pairs Shortest Paths

The All-Pairs Shortest Paths (APSP) problem is as follows: Given a graph $G = (V, E)$, $|V| = n$, and a weight function w on the edges, compute the shortest paths between all pairs of vertices.

As mentioned earlier in the context of SSSP as well, the problem is not well defined in the presence of a **negative weight cycle** in the graph.

We can solve an all-pairs shortest-paths problem by running a single source shortest-paths algorithm n times, once for each vertex as a source. Then the running time is:

- $\mathcal{O}(n|E|\log n)$ if we use the Dijkstra algorithm (assuming no negative edge weights);
- $\mathcal{O}(n^2|E|)$ if we use Bellman-Ford algorithm - i.e., $\mathcal{O}(n^4)$ if the graph is dense.

Instead of using SSSP, we will give a direct solution to the APSP problem using a dynamic programming-based algorithm known as Floyd-Warshall algorithm that runs in $\mathcal{O}(n^3)$ time. Note that using n applications of Dijkstra's algorithm is more efficient than Floyd-Warshall if the graph is sparse, i.e., if $|E|$ is significantly less than $\Theta(n^2)$, say, for example, when $|E| = \Theta(n)$. However, Dijkstra's algorithm will apply only if there are no negative edge weights.

Input and Output

Input: an adjacency matrix W where $w_{i,j}$ is the weight of the edge (i, j):

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$$

Output:
A matrix D where $d_{i,j}$ is the shortest path distance from i to j. As usual, we will focus on outputting the distances, the paths that correspond to the distances can be output easily as well.

12.2.2.1 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm gives a dynamic programming algorithm for computing all-pairs shortest paths in a given directed weighted graph (negative weights are allowed).

For a path $P = v_1, v_2, \ldots, v_{k-1}, v_k$, the vertices $v_2, \ldots, v_{k-1}$ are intermediate vertices. Let the vertices be numbered from 1 to n, i.e., $V = \{1, \ldots, n\}$. The main idea of the

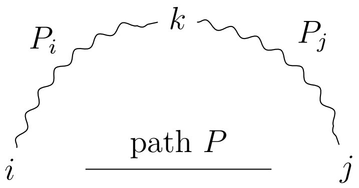


Figure 12.6: Illustration of proof of Floyd-Warshall algorithm. P is a shortest path from i to j going through vertex k. It consists of a path P_1 from i to k, followed by a path P_2 from k to j. Since P is a shortest path from i to j, P_1 is a shortest path from i to k using only intermediate vertices in $\{1, \ldots, k-1\}$ and P_2 is a shortest path from k to j using only intermediate vertices in $\{1, \ldots, k-1\}$.

Floyd-Warshall algorithm is that the algorithm, in iteration k, computes all pairs shortest paths with intermediate vertices in $\{1, \ldots, k\}$, i.e., the intermediate vertices are restricted to belong to the set $\{1, \ldots, k\}$. This uses a different recursive formulation compared to the Bellman-Ford algorithm. (Recall that in the Bellman-Ford algorithm, the algorithm, in iteration k, computes the shortest paths that use up to k edges.)

Let $d_{i,j}^{(k)}$ be the distance of a shortest path from i to j using *only intermediate vertices* from the set $\{1, \ldots, k\}$, i.e., using intermediate vertices only from the first k (numbered) vertices. Thus the subproblems are computing $d_{i,j}^{(k)}$, for $1 \leqslant k \leqslant n$ and for every pair of vertices i and j. Note that the required shortest path distance is $d_{i,j}^{(n)}$, i.e., all intermediate vertices are allowed without any restriction.

We can write a recursive formulation for $d_{i,j}^{(k)}$ as follows:

Lemma 12.9

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0 \\ \min\Big\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\Big\} & \text{if } k \geqslant 1 \end{cases}$$

Proof. Let P be a shortest path from i to j using vertices in $\{1, \ldots, k\}$. For any two vertices i and j, if $k = 0$, then clearly, $d_{i,j}^{(0)} = w_{i,j}$ (note that $w_{i,j} = \infty$, if i and j are not connected by an edge). That is, unless vertices i and j are connected directly by an edge, there cannot be any path which does not use any intermediate vertex. This shows the base case.

We now argue the recursive case, i.e., $k \geqslant 1$, i.e., there is at least one intermediate vertex. There are two cases. If P does not use k then $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$.

Otherwise, P uses k, i.e., P consists of a path P_1 from i to k, followed by a path P_2 from k to j. See Figure 12.6. By the optimal substructure property, since P is a shortest path from i to j, P_1 is a shortest path from i to k using only intermediate vertices in $\{1, \ldots, k-1\}$ and P_2 is a shortest path from k to j using only intermediate vertices in $\{1, \ldots, k-1\}$. Note that vertex k cannot occur (as an intermediate vertex) in P_1 and P_2; otherwise k will occur twice in path P, which makes P not optimal (there is a cycle in P if k occurs twice). The distance of P_1 is $d_{i,k}^{(k-1)}$ and the distance of P_2 is $d_{k,j}^{(k-1)}$ by definition. Hence, in this case, $d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$. We take the minimum of the two cases. □

Run time and Implementation

Floyd-Warshall algorithm can be implemented in $\mathcal{O}(n^3)$ time. This follows from the dynamic programming formulation. The total number of subproblems is $\mathcal{O}(n^3)$ since, for for every pair of vertices i and j, there are n subproblems, $1 \leqslant k \leqslant n$. The time needed to solve a subproblem is constant, since it involves taking minimum of two values. Hence overall time is $\mathcal{O}(n^3)$.

As usual, the algorithm can be implemented recursively using memoization or iteratively using dynamic programming. The latter is usually the choice, since it avoids the overhead of recursion. The pseudocode for the dynamic programming algorithm is given in Algorithm 57. It simply uses three for loops to compute $d_{i,j}^{(k)}$ for all $1 \leqslant k, i, j \leqslant n$. As mentioned earlier, the pseudocode assumes that the graph is input as an adjacency matrix which is convenient for the algorithm. Then adjacency matrix W represents matrix $D^{(0)}$ which is the matrix of distances in the base case, i.e., $d_{i,j}^{(0)} = w_{i,j}$. Subsequent iterations compute $D^{(k)}$ which represents the matrix of distances in iteration k, i.e., $d_{i,j}^{(k)}$. The pseudocode computes only the shortest distance, but not the paths; the algorithm can be easily modified to do that as well.

The pseudocode given in Algorithm 57 uses n matrices and hence uses $\mathcal{O}(n^3)$ space. However, since the matrix $D^{(k)}$ depends only on matrix $D^{(k-1)}$ (see the recursive formulation), we can easily modify the pseudocode to use only two matrices (see Exercise 12.15). This saves space and thus the total space needed is only the space required for 2 matrices which is $\mathcal{O}(n^2)$.

Algorithm 57 FloydWarshall – Floyd Warshall Algorithm
Input: The adjacency matrix of a directed, weighted graph
Output: Matrix $D^{(n)}$ of all-pairs shortest distances

```
func FloydWarshall($G$):
    $D^{(0)} = G$    ▷ $D^{(0)}$ is set to the adjacency matrix of $G$
    for $k = 1$ to $n$:
    ▷ Matrix $D^{(k)}$ represents the distances $d_{i,j}^{(k)}$
        for $i = 1$ to $n$:
            for $j = 1$ to $n$:
                $d_{i,j}^{(k)} = \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$
    return $D^{(n)}$
```

12.3 Worked Exercises

Worked Exercise 12.1. A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. An edge **crosses** the cut if one of its endpoints is in S and the other is in $V - S$.

The **cut property** of MST can be stated as follows: For any proper nonempty subset X of the vertices, the lightest edge with exactly one endpoint in X belongs to a minimum spanning tree. In other words, we have the following theorem.

Theorem 12.10 ▶ cut property

Let $G = (V, E, w)$ be a connected undirected graph with weight function w defined on E, i.e., $w(e)$ gives the weight of edge e. Let $(S, V - S)$ be any cut of G and let (u, v) be a minimum weight edge crossing the cut. Then (u, v) belongs to a minimum spanning tree of G.

Prove the cut property.

Solution. Proof of the cut property:

Let T be a MST of G. If T contains (u, v), we are done. Otherwise, we will construct another MST T' that includes edge (u, v) which will prove the theorem.

Since (u, v) does not belong to T, adding edge (u, v) to T completes a cycle with the edges on the (unique) path (call it p) from u to v in T.

Since T is a spanning tree, there is at least one edge (x, y) in p that crosses the cut $(S, V - S)$, since u and v are on different sides of the cut. Note that since $w(u, v)$ is the lightest edge crossing the cut, $w(u, v) \leqslant w(x, y)$.

Remove (x, y) from T and add (u, v) to T to get a new spanning tree T'. The weight of T' is:

$$w(T') = w(T) - w(x, y) + w(u, v) \leqslant w(T)$$

Thus T' is also a MST. Thus there is a MST T' that contains edge (u, v).

In a MST algorithm, the cut property validates the addition of the edge (u, v) and the cut that validates the edge is the cut $(S, V - S)$ that is mentioned in the theorem.

□

Worked Exercise 12.2. The algorithm of this problem is called *Boruvka's* algorithm, the first algorithm proposed for MST (in 1926).

We are given a *connected undirected* graph $G = (V, E)$ with positive edge weights. Assume that all the edge weights are *distinct*. The operation of *contracting* an edge $e = (u, v)$ in G involves the following operations: (1) remove the vertices u and v and introduce a new vertex x; (2) for each edge e' having an endpoint in either u or v do: replace the endpoint u or v by x, i.e., if $e' = (w, u)$ (or $e' = (w, v)$) it is replaced by the edge $e' = (w, x)$. (For this problem, we can assume that each edge comes with a distinct label. When we contract edges, we use the same labels on the edges, as they were before contraction, for easy identification. For example, in the above example, the edge label for e' remains the same after contraction as well, despite the endpoints changing.)

Consider the following algorithm for finding a minimum spanning tree:

a) Show that the minimum spanning tree of G is unique if all the edge weights are distinct. (Hint: One way to show this is proof by contradiction: assume two distinct minimum spanning trees and get a contradiction.)

b) Argue that the above algorithm correctly outputs the edges of the minimum spanning tree. (Hint: Use the cut property of Worked Exercise 12.1.)

c) Give a tight (up to a constant factor) upper bound on the number of iterations (one iteration is execution of Steps 2-6) of the algorithm.

d) Show how to implement one iteration of the algorithm in $\mathcal{O}(|V| + |E|)$ time. (Hint: Use depth-first search.) Using your answer to part (d), what is the (overall) running time of the algorithm?

Algorithm 58 Boruvka
Input: A connected undirected graph $G = (V, E)$
Output: The MST of G

```
1: func Boruvka($G$):
2:     while $|G| > 1$:
3:         for $v \in G$:
4:             Select the minimum-weight edge incident on $v$
5:         Contract all edges chosen on line 4
6:         Eliminate self-loops
7:         Eliminate all but the lowest-weight edge among each set of multiple edges
           ▷ edges that go across the same pair of vertices
8:     return edges selected on line 4 as MST edges.
```

e) Suppose the edge weights are not distinct. Will the algorithm still work? If your answer is “no”, then show how to fix it.

Solution.

a) Assume that there are two minimum spanning trees (MSTs), T and T'.

Let e be the minimum weight edge that is included in only one MST.

WLOG, let $e \in T$ and $e \notin T'$. Then $T' \cup \{e\}$ has a cycle.

The cycle has an edge $e' \notin T$ because if not, T has a cycle. And the weight of e' is greater than that of e because all weights are distinct and by the choice of e.

Thus $T' \cup \{e\} - \{e'\}$ is a spanning tree with weight smaller than T', that leads to a contradiction.

b) Let in some iteration i, we are selecting the minimum-weight edge, say e, incident on w. w can be a composite (“super”) node (meaning a collection of nodes due to contraction). Let $l(w)$ be the set of nodes in G that are collected into w. e is the minimum-weight edge across the cut $(l(w), V - l(w))$ of G. Thus, by the cut property, e must be an edge of some MST. Since the MST is unique, the algorithm only picks the edges of the same MST.

Contraction of an edge reduces the number of nodes by 1. Thus there will be exactly a total of $n-1$ contractions, since finally we are left with 1 vertex. Each contraction adds an edge to the MST. Thus by the end of the last iteration, exactly $n-1$ edges are selected.

c) In each iteration, at least half of the vertices are removed. This is because, if there are k vertices at the beginning of the iteration, then at least $k/2$ edges are selected (it can happen that vertex u selects edge (u, v) and v selects (v, u)). Each selected edge leads to removal of one vertex. After the first iteration, for example, the number of vertices is at most $\lfloor |V|/2 \rfloor$. Thus, after kth iteration, the number of vertices is at most $\lfloor |V|/2^k \rfloor$. And the iterations are executed until G has one vertex. If t is the maximum number of iterations, $1 \leqslant \frac{|V|}{2^t}$, i.e., $t \leqslant \log_2 |V|$.

d) We will next show how to implement one iteration in $\mathcal{O}(|E|)$ time, thus giving a total time of $\mathcal{O}(|E| \log |V|)$.

Note that since the given graph is connected $\mathcal{O}(|V|+|E|)=\mathcal{O}(|E|)$.

Implement the graph G using adjacency (adj) list. Represent each vertex by a unique number.

Search the adjacency list to find minimum-weight edge incident on each node (or "super" node) v. This takes $2|E|$ time for all nodes.

Using BFS find the connected components of the graph defined by the current nodes and the edges selected in this iteration. Call each component a fragment — this will be treated as a "super" vertex for the next iteration. Let the root of the BFS tree of a fragment be numbered r. Simply propagate this value r to the all nodes in the fragment (can be done during BFS itself — takes $\mathcal{O}(|E|)$ time). Thus each node in the fragment will be renumbered with the value r. This will be done to every fragment. Thus at the end of this operation each node will know the fragment number of its fragment. Finally, the adjacency list of the fragment will be updated: if there is an edge connecting a node in this fragment to a different fragment then this will be added to the adjacency list of the fragment. This will take totally $\mathcal{O}(|E|)$ time, since only so many edges will be examined. Note that if there are two edges connecting the same two fragments then this can be eliminated (taking care of multiple edges).

e) The algorithm will not work if the edge weights are not distinct. For example, imagine a triangle of 3 nodes with 3 edges of weight 1 each. It is possible that the algorithm will choose all 3 edges to contract in the first step.

To fix the problem, we create a total ordering of the edge weights by breaking ties between same weight edges based on the labels of the incident nodes (note that each node has a unique label or number). If a node has more than one minimum-weight edge incident on it, select the edge that has the lowest numbered vertex at the other end.

□

Worked Exercise 12.3. Let $G=(V,E,w)$ be a directed, weighted graph, where $w(e)$ denotes the weight of edge e. The weights can be negative, but G contains no negative cycles. Let s be a distinguished vertex in V. The goal is to compute the shortest paths from s to all vertices.

(i) Given G as input, show that Dijkstra's algorithm does not always generate the correct solution.

(ii) Consider the following algorithm that modifies the weights of G before running Dijkstra.

Does the above algorithm always generate the shortest paths for G? Note we want to generate the paths, not the value of the shortest paths. Justify your answer.

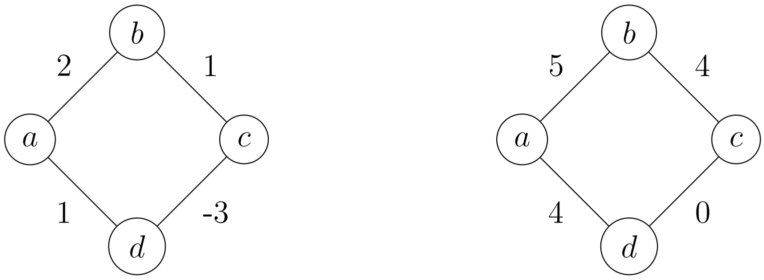
Solution. **Dijkstra's algorithm for negative edge graph**

(i) Consider the graph in Figure 12.7a. The Dijkstra's algorithm solution and correct solution with the source vertex a would be:

Algorithm 59 DijkstraNew

Input: A weighted graph $G = (V, E)$

```
func DijkstraNew($G$):
    $G' = G$
    $w_{\max} = \max\{|w(e)| \mid e \in E\}$
    ▷ $W_{\max}$ is the maximum absolute-value edge weight.
    for edge $e' \in G'$:
        $w(e') = w(e) + w_{\max}$    ▷ Update the weight of $e'$
    return Dijkstra($G'$)
```



(a) Graph G (b) Graph G'

Figure 12.7: Negative edge graph

destination	Dijkstra's algorithm	Correct solution
b	$(a, d) \to (d, c) \to (c, b)$	$(a, d) \to (d, c) \to (c, b)$
c	$(a, d) \to (d, c)$	$(a, d) \to (d, c)$
d	(a, d)	$(a, b) \to (b, c) \to (c, d)$

It can be seen that Dijkstra's algorithm fails to generate the correct solution in this example.

(ii) Consider the same graph in part (i), the graph G' generated is shown in figure Figure 12.7b. Then the solution of Dijkstra's algorithm on graph G' would be

destination	Dijkstra's algorithm	Correct solution
b	(a, b)	$(a, d) \to (d, c) \to (c, b)$
c	$(a, d) \to (d, c)$	$(a, d) \to (d, c)$
d	(a, d)	$(a, b) \to (b, c) \to (c, d)$

It could be observed that this solution is not the correct solution for graph G provided in part (i). Thus, **Dijkstra-new** does not generate the shortest paths for G.

□

Worked Exercise 12.4. A communications provider has a number of computers that are connected using a network. Assume that the network is an undirected graph $G = (V, E)$ where the vertices represent the computers and the edges represent the links between them. Each edge $(u, v) \in E$ has an associated weight $b(u, v)$ (a positive number) which represents the bandwidth capacity of that edge. Given a path between two nodes, the **bottleneck value** of the path is the *minimum* of the weights of the edges in the path. The **maximum** bottleneck path between two nodes is the path that has the maximum bottleneck value among all paths between them.

(a) In the graph of Figure 12.3, compute the respective maximum bottleneck paths between node a and all other nodes.

(b) Develop an algorithm that given two nodes a and b computes the **maximum** bottleneck value path from a to b. What is the running time of your algorithm. (Hint: Modify Dijkstra's algorithm to solve this problem.)

Solution.

(a) The following are the maximum bottleneck paths between a and all other nodes:

1. $a \to b$: $a \to d \to b$
2. $a \to c$: $a \to d \to c$
3. $a \to d$: $a \to d$
4. $a \to e$: $a \to d \to f \to e$
5. $a \to f$: $a \to d \to f$
6. $a \to g$: $a \to b \to g$

(b) Although this problem appears to be very different from the shortest paths problem, there is a uniform way of looking at both the problems. In the shortest path problem, the "value" of a path is the sum of weights in the path, and we want to find (among all possible paths) the one with minimum value. In the maximum bottleneck problem, the "value" of a path is the minimum weight edge of that path and we want to find the path with the maximum value. Thus, the definition of "value" differs, and instead of finding a path of minimum value (as in shortest paths), we are finding a path with maximum value.

A modification of Dijkstra's algorithm can be used to compute the maximum bottleneck value path between a (the source) and all other vertices. In particular, this will yield the **maximum** bottleneck value path from a to b.

Let

$b(u, v)$ — the weight (bandwidth) of edge $(u, v) \in E$.

$key[v]$ — the value of the maximum bottleneck path from s to v.

$prev[v]$ — the predecessor of v in the path.

The Algorithm MaxBottleneck computes the maximum bottleneck path from s to all vertices v.

Algorithm 60 MaxBottleneck – Maximum Bottleneck Path using Dijkstra's
Input: A graph G and source node s
Output: Maximum Bottleneck Path from s to all nodes

```
func MaxBottleneck($G$, $s$):
    for $v \in G$:
        key[$v$] = 0
        prev[$v$] = Null
    key[$s$] = $\infty$
    $S = \emptyset$
    $H$ = BuildHeap($G$)    ▷ here, $H$ is a max-heap
    while $H$ is not empty:
        $u = H.ExtractMax()$
        $S$.Add($u$)
        for $v \in$ Adj[$u$]:
            newKey = $\min\{$key[$u$]$, b(u, v)\}$    ▷ Distance through $v$
            if dist[$v$] < newKey:
                $H.IncreaseKey(v,$ newKey$)$
                prev[$v$] = $u$
```

Note that we can use a binary max-heap to implement the heap operations — extract-max and increase-key. □

Worked Exercise 12.5. A **vertex-cover** of a tree T is a subset V' of its vertices such that, for each edge in T, at least one of its endpoints belongs to V'. For example, the set containing all the vertices of T is a vertex-cover. The **minimum** vertex-cover of T is a vertex-cover that contains the fewest vertices.

We are given a rooted tree T of n vertices. (If you want you can assume the tree to be binary, i.e., each node has at most two children.) Design a dynamic programming algorithm to output the size of the minimum vertex-cover of T, and analyze its running time. Note that you should first give a recursive formulation of the dynamic programming solution, clearly explaining your notation. Then give the pseudocode and analyze its running time. (Hint: Each vertex can either be in the vertex cover or not; this will be useful in setting up your subproblems.)

Solution. Let v be any node in the tree T. We will recursively compute the size of the minimum vertex cover of the *subtree rooted* at v — call this $\mathrm{VC}(v)$. If root of T is r, then $\mathrm{VC}(r)$ is the answer.

To compute $\mathrm{VC}(v)$ we define two subproblems: $\mathrm{VC}_{\mathrm{in}}(v)$ and $\mathrm{VC}_{\mathrm{out}}(v)$ — these denote the size of the minimum vertex cover of the subtree rooted at v that *includes* v and the size of the minimum vertex cover of the subtree rooted at v that *excludes* v, respectively (there are only two possibilities). Note that for each edge, at least one of its endpoints should be in the vertex cover to cover the edge.

Note that $\mathrm{VC}(v) = \min\{\mathrm{VC}_{\mathrm{in}}(v), \mathrm{VC}_{\mathrm{out}}(v)\}$.

Now we can write the recursive formulation to compute $\mathrm{VC}_{\mathrm{in}}(v)$ and $\mathrm{VC}_{\mathrm{out}}(v)$.

Trees (as oppopsed to general graphs) naturally admit a recursive formulation: to compute a quantity for a tree rooted at a node v, we recursively compute the quantity at each of its subtrees (rooted at its respective children) and then combine the solution.

Since the subtrees do not have any edges between them (since it is a tree), this usually works.

If v is a leaf (i.e., it has no children), then:

$\mathrm{VC}_{\mathrm{in}}(v) = 1$ and $\mathrm{VC}_{\mathrm{out}}(v) = 0$.

If v has k children (say, $v_1, v_2, \ldots, v_k$) then:

$$\mathrm{VC}_{\mathrm{in}}(v) = 1 + \mathrm{VC}(v_1) + \mathrm{VC}(v_2) + \cdots + \mathrm{VC}(v_k)$$

The above says that if v is included in the vertex cover, then each of its children may or may not be included.

$$\mathrm{VC}_{\mathrm{out}}(v) = \mathrm{VC}_{\mathrm{in}}(v_1) + \mathrm{VC}_{\mathrm{in}}(v_2) + \cdots + \mathrm{VC}_{\mathrm{in}}(v_k)$$

The above says that if v is excluded in the vertex cover, then each of its children has to be included to cover the edge between v and its children.

It is straightforward to write the pseudocode from the above formulation. For example, the bottom-up solution will compute $\mathrm{VC}(v)$ starting from the leaves (the base case) and going up the tree.

The number of subproblems is $2n$ — two corresponding to each vertex of the tree. The total time to compute a subproblem $\mathrm{VC}(v)$ is proportional to the number of children of v which is bounded by the degree of v. Thus, the total time is proportional to the total degree of all nodes which is $\mathcal{O}(n)$ (since it is a tree, there are only $n-1$ edges). Hence, the running time is $\mathcal{O}(n)$. □

12.4 Exercises

Exercise 12.1. Consider the weighted graph in Figure E.1 (right) in Appendix E. Run the following MST algorithms on this graph. Show the edges added step by step in each algorithm. Also, mention the cut that validates the addition of each edge for Kruskal's and Prim's algorithm.

(a) Kruskal's algorithm

(b) Prim's algorithm

(c) Boruvka's algorithm (see Worked Exercise 12.2).

Exercise 12.2. Consider the weighted graph in Figure E.1 (right) in Appendix E. Run the following algorithms for finding the shortest paths from node a to all other nodes. Show the edges added step by step in each algorithm to construct the SPT (Shortest Paths Tree) rooted at a.

(a) Dijkstra's algorithm

(b) Bellman-Ford algorithm

Exercise 12.3. Give an example of a complete graph on 5 nodes — a, b, c, d, e — (i.e., all pairs of edges between the 5 nodes are present) with suitable positive edge weights (all unique) so that:

(1) the MST of the graph is the same as the SPT (shortest path tree) rooted at a.

(2) the MST of the graph is different from the SPT rooted at a.

Show how Prim's algorithm and Dijkstra's algorithm compute the MST and SPT respectively on the two examples in parts (1) and (2).

Exercise 12.4. Give an example of a complete graph on 6 nodes — a, b, c, d, e, f — (i.e., all pairs of edges between the 6 nodes are present) with suitable positive edge weights (all unique) so that:

(1) the MST of the graph is the same as the SPT (shortest path tree) rooted at a.

(2) the MST of the graph is different from the SPT rooted at a.

Show how Prim's algorithm and Dijkstra's algorithm compute the MST and SPT respectively on the two examples in parts (1) and (2).

Exercise 12.5. Give an efficient algorithm to find the length (number of edges) of a minimum length negative-weight cycle in a graph.

Exercise 12.6. Let $G = (V, E)$ be a weighted, directed graph with positive integer weights (all weights greater than 0); and the maximum weight is W.
(a) Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $\mathcal{O}(W|V| + |E|)$ time.
(b) Give an $\mathcal{O}(|E| \lg W)$ time algorithm for the problem by improving your algorithm of part (a).

Exercise 12.7. Give an $\mathcal{O}(|V||E|)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

Exercise 12.8. You are given an undirected weighted graph $G = (V, E)$, where each node $v \in V$ is assigned an unique number called *rank*. Ranks are given as part of the input and are integers chosen from the interval $[1, n]$, where n is the number of nodes in the graph (thus, the rank of a node can be simply taken to be its label, which is an integer from $[1, n]$). Note that the minimum ranked node in the graph has rank 1 and the maximum ranked node has rank n. All weights are positive. For each node v, we would like to compute the minimum ranked node within *every* distance d. We want to represent this information as a list of pairs called as $L(v)$ for each node v, defined as follows: $(a_v(i), u_v(i))$ $(1 \leqslant i \leqslant |L(v)|$, where $|L(v)|$ is the number of pairs in the list $L(v)$), where $a_v(i)$ is a rank and $u_v(i)$ is a distance, such that:

(a) the list of ranks is in decreasing order, i.e., $a_v(1) > \cdots > a_v(|L(v)|) = 1$; and the list of corresponding distances is in increasing order, i.e., $u_v(1) < \cdots < u_v(|L(v)|) = \infty$.

(b) for all $1 \leqslant i \leqslant |L(v)|$, $a_v(i)$ is the minimum ranked node *strictly within* distance $u_v(i)$ of node v (i.e., the set of all nodes within a distance d of v, for every $d < u_v(i)$).

It follows from the above definition that, for all v, the first rank in the list $L(v)$ will be $a_v(1) = rank(v)$ and the last rank will be 1.

(i) Consider a line graph on n nodes, i.e., the graph is just a path of length $n - 1$. Let all edge weights be 1. Let the nodes be numbered from 1 to n, left to right. Let rank of node k be k. Give $L(v)$ for all v.

(ii) In an arbitrary graph G, give an efficient algorithm to compute $L(v)$ for all v. Analyze the running time. (Hint: Modify Dijkstra).

Exercise 12.9. Given a weighted (connected) undirected graph $G = (V, E, w)$, where $w(u, v)$ is the weight of edge (u, v). Assume that all weights are positive. The goal is to partition the vertex set V into k disjoint non-empty subsets $C_1, C_2, \ldots, C_k$ (i.e., for all $1 \leqslant i < j \leqslant k$, $C_i \cap C_i = \phi$, and $C_1 \cup C_2 \cdots \cup C_k = V$) such that the *separation* between the subsets is *maximized*. The separation is defined as the minimum distance between any pair of vertices lying in different subsets. (The distance d_{ij} between two subsets C_i and C_j is defined as $d_{ij} = \min_{u \in C_i, v \in C_j} w(u, v)$ and separation $d = \min_{1 \leqslant i < j \leqslant k} d_{ij}$.) Give an efficient algorithm that outputs the k disjoint non-empty subsets. Prove the correctness of your algorithm and analyze its running time. (Hint: Consider Kruskal's algorithm.)

Exercise 12.10. Given a weighted (connected) undirected graph $G = (V, E, w)$, with $m = n + \mathcal{O}(1)$ edges (i.e., the number of edges is at most the number of vertices plus some constant). Give an $\mathcal{O}(n)$-time algorithm to compute the minimum spanning tree of G. Prove the correctness and the running time of your algorithm. (Hint: Think of how one can eliminate an edge from being in the MST.)

Exercise 12.11. Dijkstra's single-source shortest-paths (SSSP) algorithm on a directed graph uses a set S that initially contains only the source s and that eventually includes all the vertices of the graph. Vertices are added to S one at a time. Let $N(v)$ denote the number of times that the $d[v]$ value of a vertex v in $V - S$ changes due to an update (line 14 of the Dijkstra's algorithm pseudocode). Answer each of the following questions (provide a brief justification for each of your answers).

- Can the $d[v]$ value of a vertex v in $V - S$ ever get smaller than the cost of a shortest s-to-v path in the graph?
- Can $N(v)$ exceed the in-degree of v? (Recall that the in-degree of a vertex is the number of edges going into it.)
- Can $N(v)$ be less than the in-degree of v?
- If (v, w) is the lowest weighted edge in the graph, is it true that v must be added to S before w is added to S?
- Let the vertex v have the ith longest shortest-path from s to it, and assume that there is no tie for that ith rank (i.e., no vertex other than v has that particular length of a shortest path from s to it). Suppose you are told that, at a given point during the execution of Dijkstra's algorithm, the size of S is j. Is this information sufficient to determine whether, at that point, v is in S or in $V - S$? Why?

Exercise 12.12. We are given a *directed* graph $G = (V, E)$ with n vertices, labelled $1, 2, \ldots, n$. All (directed) edges in G are of the form (i, j) where $i < j$ (i.e., an edge is always directed from a lower labelled vertex to a higher labelled vertex). Also, every vertex except the vertex labelled n has at least one edge leaving it, i.e., for every vertex $i \in \{1, \ldots, n-1\}$, there is at least one edge of the form (i, j) $(j > i)$. The length of a path is the number of edges in it. Give a dynamic programming algorithm to find the *longest* path (and its length) that begins at vertex labelled 1 and ends at vertex labelled n. Prove the correctness and analyze your running time. (As usual give the recursive formulation, clearly stating your notation).

Exercise 12.13. Give an $\mathcal{O}(mn)$ time algorithm to output the vertices of a negative weight cycle (if one exists) of a given graph $G(V, E)$.

Exercise 12.14. Show that in the A^* algorithm, if the heuristic function h that estimates the distance of a node to the target is admissible (i.e., it never overestimates the true shortest path distance), then A^* is guaranteed to find the shortest path from the source to the target.

Exercise 12.15. Give an implementation of Floyd-Warshall algorithm that uses $\mathcal{O}(n^2)$ space and runs in $\mathcal{O}(n^3)$ time.

Exercise 12.16. Unlike the shortest paths problem which admits efficient polynomial time algorithms, the *longest paths* problem in graphs is NP-hard with no known polynomial time algorithm. The longest paths problem comes in many variants: Find the longest path (1) between two nodes (2) between all pairs of nodes (3) in the graph. By longest path we mean a *simple* path (i.e., a path with no repeated nodes) with largest total weight.

If we restrict our graphs to directed acyclic graphs (DAGs), then the longest paths problem can be solved in polynomial time. Consider a DAG $G = (V, E)$ with weights on its edges. Given two vertices s and t in V, give an efficient algorithm to find the longest path (and its weight) between s and t in G. If there is no path between s and t, then output null.

The idea for the algorithm is as follows. We saw in Chapter 11 that DAGs admit a linear (topological) ordering. Using the ordering, apply dynamic programming technique to build longest paths starting from the source node.

1. What are the subproblems?
2. Give a recursive formulation for solving a problem in terms of smaller subproblems.
3. Give a DP algorithm for the problem.
4. What is the running time of the algorithm?
5. Why does this algorithm not work on arbitrary directed graphs?

Exercise 12.17. Given a directed graph G, the **transitive closure** of G is a directed graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) \mid \text{there is a path from } i \text{ to } j \text{ in } G\}$$

Given G, give a dynamic programming algorithm to compute G^*.

Exercise 12.18. Let $G = (V, E)$ be a weighted, directed graph. (Weights may be positive or negative). Assume that the graph G is input as a weighted adjacency matrix W, i.e.

- $W_{i,j} = w_{i,j}$ if there is an edge between i and j and $w_{i,j}$ is the weight of this edge
- $W_{i,j} = \infty$ if there is no edge between i and j
- $W_{i,i} = 0$ for all $i \in V$

Given a path between two vertices in G, the **bottleneck (value)** of the path is the *minimum* of the weights of its edges. The *maximum bottleneck* path between two vertices is a path with the maximum bottleneck value (among all possible paths between the two vertices). The goal of this problem is to find the *value of the maximum bottleneck path* between *all pairs* of vertices. The goal is to design an $\mathcal{O}(n^3)$ time dynamic programming algorithm that will output the *value of the maximum bottleneck path* between *all pairs* of vertices.

(a) Give a recursive formulation for this problem. Clearly explain your notation, in particular, what the subproblems are. Do not forget to mention the base case(s).

(b) Give the pseudocode for efficient algorithm based on your recursive formulation. What is the running time of your algorithm and why?

13 GRAPH ALGORITHMS: MAXIMUM FLOW AND MINIMUM CUT

This chapter focuses on network flow problems which are fundamental graph optimization problems with a lot of applications. Network flow is also related to a number of other fundamental graph problems such as minimum cut and matching. In a network flow problem, we are given a graph with weighted edges where the weights stand for capacities of the edges. For example, one can think of a network of pipes with each pipe having a capacity. The goal is to route flow (say water or oil) from one or more source nodes to one or more destination nodes (called sinks) obeying the capacity constraints. Depending on the setting, we have different types of flow problems. Such problems arise in a wide variety of applications such as routing, transportation, reliability etc.

13.1 Flow Network

A **flow network** is a directed graph $G = (V, E)$, and a capacity function

$$c : V \times V \to R$$

such that:

1. For each $(u, v) \in E$ the capacity $c(u, v) \geqslant 0$.
2. If $(u, v) \notin E$ then $c(u, v) = 0$.
3. There is a source s and a sink t.
4. Each vertex is on a directed path from s to t.

We will assume without loss of generality, that in a flow network, if $(u, v) \in E$, then $(v, u) \notin E$. Exercise 13.1 asks you to show why this assumption is without loss of generality.

Flow

A flow in a flow network G with capacity function c is a function $f : V \times V \to R$ such that:

1. For all $(u, v) \in E$ (capacity constraint)

$$f(u, v) \leqslant c(u, v)$$

2. For all $u, v \in V$, (symmetry)

$$f(u, v) = -f(v, u)$$

3. For all $u \in V - \{s, t\}$, (flow conservation)

$$\sum_{v \in V} f(u, v) = 0$$

The **value** of the flow is

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$$

Flow Problems

There are many fundamental problems associated with network flow:

- **Maximum flow**: Given a flow network G with source s and sink t, find the **maximum** flow.
- **Minimum-cost flow**: Ship d units from s to t such that the total cost is minimized (each edge charges a certain amount for a unit of flow through it).
- **Multi-commodity flow**: ship k commodities, each with its own sink, source, and flow value (demand).

In this chapter, we will focus on the maximum flow problem, perhaps the most fundamental of the flow problems.

Exercise 13.1. Show we can assume, without loss of generality, that in a flow network, if $(u, v) \in E$, then $(v, u) \notin E$. In particular, show that this assumption does not affect the solvability of the max flow problem in a flow network (which may have edges such as (u, v) and (v, u)).

13.2 Maximum Flow: The Ford-Fulkerson Method

Given a flow network, our goal is to design an efficient algorithm that finds the maximum flow. A very natural algorithm for such a flow is due to Ford and Fulkerson, which is essentially a greedy algorithm. The algorithm consists of phases. In every phase, it pushes more flow, until no more flow is possible, in which case the algorithm terminates. Ford-Fulkerson showed that when the algorithm terminates, the total flow pushed across all the phases constitutes a maximum flow.

Ford Fulkerson Approach: Pseudocode

```
Ford_Fulkerson_Method($G, c, s, t$)
  1. Initialize flow to 0;
  2. While "possible"
     (a) Improve flow (push more flow from $s$ to $t$);
```

Residual Network

Given a flow f on a flow network G the **residual** capacity of an edge (u, v), $c_f(u, v)$, is the additional amount of flow that can be send from u to v before exceeding the capacity $c(u, v)$, i.e.,

$$c_f(u, v) = c(u, v) - f(u, v).$$

We note that flow is defined on pairs of vertices and not edges. *We will assume that edges with zero residual capacity will be removed from the network (since no flow can be pushed through these edges).* We next define the concept of a *residual network* which plays an important role in network flow algorithms.

Definition 13.1 ▶ Residual Network

Given a flow f on a flow network G with capacity function c, the **residual network** G_f of G induced by f is a flow graph on G with capacity function defined by $c_f = c - f$.

It is important to note that the residual graph might have edges that are not in G, but if (u, v) is an edge either (u, v) or (v, u) is an edge of G.

The main idea of having the "reverse" direction edges is to allow the algorithm to push flow "back" — this has the effect of cancelling the flow that was pushed in the forward direction. Thus if there is a flow of 10 from u to v, pushing a flow of 3 from v to u is equivalent to pushing a (net) flow of 7 from u to v. As we will see, this is very useful in designing algorithms for network flow.

Figure 13.1 shows a flow network and a residual graph.

Improving Flow

We show that if one can push additional flow in a residual graph, then this flow can be combined with an already existing flow.

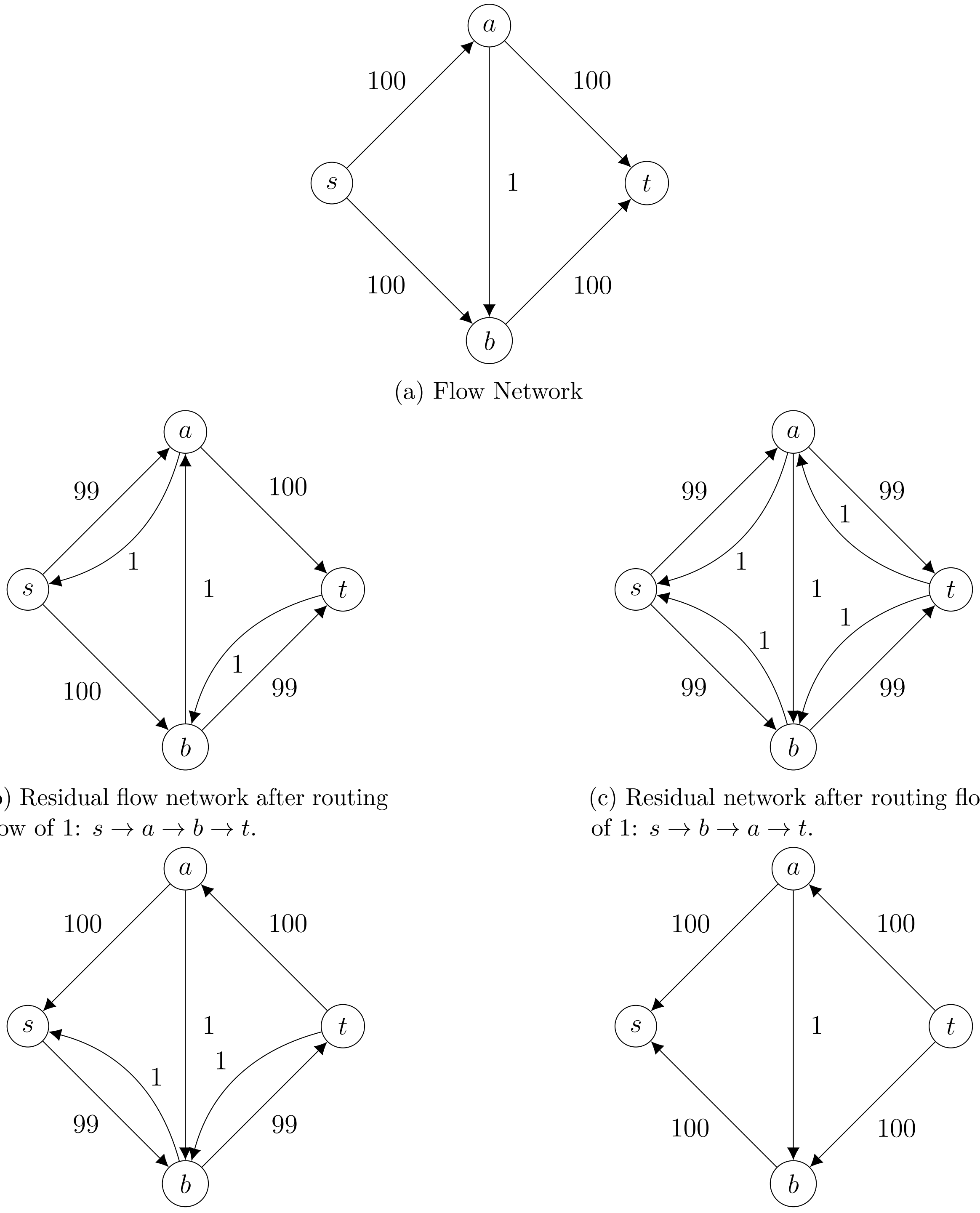
Let f be a flow function on G. Let f' be a flow function on the residual graph of G_f induced by f. Define the "flow" function $f + f'$ as

$$(f + f')(u, v) = f(u, v) + f'(u, v)$$

Lemma 13.1

The function $f + f'$ is a flow function in G, with flow value

$$|f + f'| = |f| + |f'|$$



(a) Flow Network

(b) Residual flow network after routing flow of 1: $s \to a \to b \to t$.

(c) Residual network after routing flow of 1: $s \to b \to a \to t$.

(d) Residual flow network after routing flow of 100.

(e) Residual network after routing flow of 200.

Figure 13.1: Figure a shows a flow network. Figure b shows the residual network after routing a flow of 1: $s \to a \to b \to t$. Note how new edges $(a, s), (b, a), (t, b)$ appear in the residual graph. For example, edge (b, a) appears with a capacity of 1 because, originally, (b, a) had capacity 0 and we pushed a flow of -1 from b to a (i.e., a flow of 1 from a to b). Thus, the residual capacity of this edge is $0 - (-1) = 1$. Figure c shows the residual network after routing a flow of 1: $s \to b \to a \to t$. Figures d and e show residual graphs after routing flows of 100 and 200, respectively. There is no directed (augmenting) path from s to t in the network of Figure e, so no more flow can be pushed. The total flow pushed overall is 200 which is the maximum flow.

Proof. We show that $f + f'$ satisfies all the three flow function properties:

1. *Capacity constraint:* For all $(u, v) \in E$,

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v)\\ &\leqslant f(u, v) + (c(u, v) - f(u, v))\\ &= c(u, v)\end{aligned}$$

2. *Symmetry:* For all $u, v \in V$,

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v)\\ &= -f(v, u) - f'(v, u)\\ &= -(f + f')(v, u)\end{aligned}$$

3. *Flow conservation:* For all $u \in V - \{s, t\}$,

$$\begin{aligned}\sum_{v\in V}(f + f')(u, v) &= \sum_{v\in V}(f(u, v) + f'(u, v))\\ &= \sum_{v\in V} f(u, v) + \sum_{v\in V} f'(u, v))\\ &= 0 + 0\end{aligned}$$

$$\begin{aligned}|f + f'| &= \sum_{v\in V}(f + f')(s, v)\\ &= \sum_{v\in V}(f(s, v) + f'(s, v))\\ &= \sum_{v\in V} f(s, v) + \sum_{v\in V} f'(s, v))\\ &= |f| + |f'|\end{aligned}$$

□

Augmenting Path

We next define the concept of agumenting path which is used repeatedly in algorithms for network flow (as well as related problems such as matching).

Definition 13.2 ▶ Augmenting Path

Given a graph $G = (V, E)$ and a flow f in G, an **augmenting path** p is a simple (directed) path connecting s to t in the residual graph G_f.

The following lemma (which is easy to prove and left as an exercise) is useful in identifying the amount of flow that can be pushed in an augmenting path. It simply says that a flow equal to the minimum capacity among all the edges of the augmenting path can be pushed from s to t. Since we assumed that zero capacity edges are removed from the flow network, this means a non-zero amount of flow can be pushed from s to t along the augmenting path. One can call this minimum capacity edge of the path as the *bottleneck* edge, as it determines the maximum flow that can be pushed using only the edges of the path.

Lemma 13.2

Given a graph $G = (V, E)$ and a flow f in G, and an augmenting path p in the residual graph G_f, let

$$c_f(p) = Min\{c_f(u, v) \mid (u, v) \in p\}.$$

Define function f_p as follows:

$$f_p(u, v) = \begin{cases} c_f(p) & (u, v) \in p \\ -c_f(p) & (v, u) \in p \\ 0 & \text{otherwise.} \end{cases}$$

Then f_p is a flow function in G_f with value $|f_p| = c_f(p) > 0$.

The above lemma says that if we have an augmenting path in G_f, then we can push additional flow in G. Thus, we have the following theorem.

Theorem 13.3

If there is an augmenting path p in G_f then then there is a flow $|(f + f_p)| > |f|$ in G.

Now we are ready the state the Ford-Fulkerson algorithm in detail. This algorithm is important since it forms the basis of several other more efficient algorithms.

Ford-Fulkerson-Method(G, c, s, t)

```
1. Initialize $f(u, v) = 0$ for every edge in $E$.
2. While there exists an augmenting path $p$ in $G_f$
    (a) Augment the flow $f$ with $f_p$.
```

Figure 13.1 shows applying the Ford-Fulkerson algorithm for four iterations (Figures B,C, D, and E), each iteration increasing the flow pushed until we reach a residual graph (Figure E) where is no more augmenting path.

The proof of correctness of Ford-Fulkerson follows from a fundamental duality theorem called the Max-Flow Min-Cut Theorem.

13.2.1 The Max-Flow Min-Cut Theorem

We need the following definitions to state the theorem.

An (S, T)-**cut** in a flow network $G = (V, E)$ is a partition of V to S and $T = V - S$ such that $s \in S$ and $t \in T$. Given a flow f in G, the **net flow** across a cut (S, T) is

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v).$$

The **capacity** of the cut is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

Theorem 13.4 ▶ Max-flow Min-Cut Theorem

Maximum $s - t$ flow in G equals the minimum $c(S, T)$ over all (S, T) cuts.

We first prove the following lemma which will imply that any flow cannot be larger than the minimum cut capacity.

Lemma 13.5

Let f be a flow in G. For any (S, T) cut

$$f(S, T) = |f|$$

Proof.

$$\begin{aligned} f(S,T) = f(S, V - S) = &= f(S,V) - f(S,S) && (13.1)\\ &= f(S,V) && (13.2)\\ &= f(s,V) + f(S-s,V) && (13.3)\\ &= f(s,V) && (13.4)\\ &= |f| && (13.5) \end{aligned}$$

The first equality follows from the fact that $T = V - S$ and from flow conservation. Note that $f(S, S) = 0$. Similarly we can write $S = s \cup S - s$ and use flow conservation. □

Since any flow across any cut has to be bounded by the capacity of that cut, it is bounded by the capacity of the minimum cut that separated s and t. Thus we have the following corollary.

Corollary 13.6

The flow in G is bounded by the minimum (S, T)-cut capacity in G.

To prove the Max-flow Min-cut theorem we prove an intermediate result (statement 2 below).

Theorem 13.7 ▶ Max-flow Min-cut Theorem

Let f be a flow from s to t in G. The following conditions are equivalent:

1. f is a maximum flow.
2. The residual graph G_f allows no augmenting paths.
3. $|f| = c(S, T)$ for some (S, T)-cut.

Proof. We first show that

- (1) ⇒ (2):

  This is true because the contrapositive is true, i.e., if G_f has an augmenting path the flow f can be improved.

- Next we show that (2) ⇒ (3):

  We construct an (S, T) cut as follows: Let

  $$S = \{v \in V \mid \text{there is a path from } s \text{ to } v \text{ in } G_f\}$$

  and let $T = V - S$.

  This partition defines an (S, T)-cut where $s \in S$ and $t \in T$; otherwise there is an augmenting path in G_f.

  For each $u \in S$ and $v \in T$ we have $f(u, v) = c(u, v)$, otherwise $(u, v) \in G_f$ and $v \in S$. In other words, all edges of the form (u, v) with $u \in S$ and $v \in T$ are *saturated*, i.e., the flow is equal to the capacity.

  Thus $f(S, T) = c(S, T)$ and we proved that $|f| = f(S, T)$. Hence

  $$|f| = f(S, T) = c(S, T)$$

- Finally, to complete the proof, we show that (3) ⇒ (1):

  We proved that for any (S, T) cut, $|f| \leqslant c(S, T)$ (Corollary 13.6), thus if $|f| = c(S, T)$ for some cut, it is a maximum flow.

□

13.2.2 Analysis of the Ford-Fulkerson Algorithm

Theorem 13.8

Assume that all the capacities are integral then the run-time of the Ford-Fulkerson algorithm is $\mathcal{O}(|E| \cdot |f|)$ where $|f|$ is the maximum flow value.

Proof. Steps 1, 2, and 2a each take $\mathcal{O}(|E|)$ time. The while loop is executed $\mathcal{O}(|f|)$ times, since each iteration improves the flow by at least 1. An augmenting path in the residual graph can be found by breadth-first or depth-first search, thus each iteration of the while loop takes $\mathcal{O}(|E|)$ time. □

13.3 Faster Maximum Flow Algorithms

We note that Ford-Fulkerson is a *pseudo-polynomial* time algorithm, since its running time is proportional to the flow value (which depends on the capacity values) which can be quite large compared to n and m, the graph parameters. A (strongly) polynomial algorithm's running time will depend only on n and m. In this section we present three strongly polynomial-time algorithms, each faster than the previous one.

13.3.1 Edmonds-Karp Algorithm

We present an algorithm due to Edmonds and Karp, which is a simple modification of Ford-Fulkerson, that is strongly polynomial.

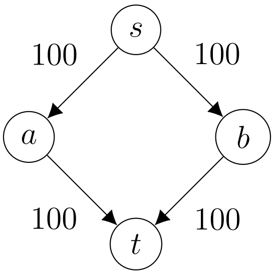


Figure 13.2: Level graph of the flow graph of Figure A of Figure 13.1. Note that edge (a, b) is deleted in the level graph.

Theorem 13.9

If the augmenting path in the Ford-Fulkerson algorithm is found by a breadth-first search (i.e. the algorithm uses a path of minimum number of edges), the run time of the algorithm is $\mathcal{O}(m^2n)$, where m is the number of edges and n is the number of vertices.

To prove this theorem, we use the concept of a **level graph** which is a useful tool for analyzing other maximum flow algorithms as well.

Definition 13.3 ▶ Level Graph

The **level graph** L_G of G is the directed breath-first graph of G (this is considered as an unweighted graph, where the capacities are ignored) with root s with sideways and back edges **deleted**. The **level** of a vertex u is the length of the shortest path from s to u in G.

In Figure 13.1, the level graph of the flow network of Figure A is the same graph with edge (a, b) removed (see Figure 13.2). The level graph can be obtained by doing a BFS starting from s on the flow graph (ignoring capacities).

Observations: The following observations are easy to show which follows directly from the properties of a bfs tree (Section 11.3):

1. The level graph has no edges from level i to level j for $j \geqslant i + 2$.
2. Any shortest path from s to any other vertex is a path in the level graph.
3. Any path with either a back or a sideways edge of the level graph will be strictly longer.

Any path from s to t in the level graph will be a shortest path. We use such a path to augment the flow.

The following is a key lemma.

Lemma 13.10

(a) Let p be an augmenting path of minimum length in G. Let G' be the residual graph obtained by augmenting along p and let q be an augmenting path of minimum length in G'. Then $|q| \geqslant |p|$. Thus the length of the shortest augmenting path cannot decrease by applying the above heuristic.
(b) We can augment along shortest paths of the same length at most $m = |E|$ times before the length of the shortest augmenting path must increase strictly.

Proof. (a) p is some path in the level graph L_G. After augmenting along p, at least one edge will disappear (due to saturation) and at most $|p|$ new edges will appear in the residual graph G'. These new edges are back edges and cannot contribute to a shortest path from s to t as long as t is reachable from s in the level graph. Then any augmenting path in the *residual graph* must use a back or side edge and must be strictly longer.

(b) Every time we augment using a path in L_G we lose an edge in L_G. Thus we can augment at most m times, before t is no longer reachable in L_G. Since we lose at least one edge (due to saturation) every time we augment, we can augment only $\mathcal{O}(m)$ times. □

Edmonds-Karp Algorithm

We are now ready to describe an implementation of Edmonds-Karp algorithm in terms of the level graph framework.

1. Find the level graph L_G.
2. Repeatedly augment along paths in L_G, updating residual capacities and deleting *saturated* edges (zero capacity edges) until t is no longer reachable from s.
3. Calculate a new level graph from the residual graph at that point and repeat.
4. Continue as long as t is reachable from s (in the residual graph).

The correctness of the algorithm is the same as that of Ford-Fulkerson.

Running time

After each level graph calculation, the distance between s and t increases by 1. Thus, there are at most n level graph calculations. We need $\mathcal{O}(m)$ time to calculate a level graph (using breadth-first search which takes $\mathcal{O}(m)$ time). The total number of augmentations in one level graph is at most $\mathcal{O}(m)$ and each augmentation takes $\mathcal{O}(n)$. Thus the total amount of time per level graph is $O(m + mn) = O(mn)$. Since there are at most n level graph calculations, the total time is $\mathcal{O}(m^2 n)$.

13.3.2 Dinitz Algorithm

Dinitz max flow algorithm is similar to Edmonds-Karp, but we find augmenting paths in the level graph L_G more carefully. It is similar to Edmonds-Karp as we compute the level graph many times (at most n times), but differs in the way augmenting paths are found in a level graph.

Define a **blocking flow** as the flow when there is no path from s to t in L_G. That is, this is the total flow that has been pushed until there is no more possible using the current L_G. We need to recompute a new level graph at this point.

Instead of constructing a blocking flow path by path (as done in Edmonds-Karp), we construct a blocking flow all at once by finding a maximal set of minimum-length augmenting paths. Each such construction is a *phase*.

In each phase we will construct augmenting paths in L_G in a *depth-first* fashion.

13.3.2.1 Dinitz Algorithm: One Phase

1. Initialize: Construct a new level graph L_G. Let $u = s$ and $p = s$. (u will denote the vertex currently being visited and p is a path from s to u.) Go to **Advance**.

2. Advance: If there is no edge out of u, go to **Retreat**. Otherwise, let (u, v) be such an edge. Set $p = p.[v]$ and $u = v$. If $v \neq t$ then go to **Advance** else go to **Augment**.

3. Retreat: If $u = s$ then **Stop**. Otherwise, delete u and all adjacent edges from L_G and remove u from the end of p. Set $u =$ last vertex on p. Go to **Advance**.

4. Augment: Let Δ be the bottleneck capacity along p. Augment path flow along p and adjust residual capacities along p. Delete newly saturated edges. Set $u =$ the last vertex on the path p reachable from s along unsaturated edges of p. Set $p =$ the portion of p up to and including u. Go to **Advance**.

Runtime

We analyze the time taken for each for the different types of operations in *one* phase.
1. **Initialize:** Takes $\mathcal{O}(m)$ time, since this can be done by a breadth-first search which takes $\mathcal{O}(m + n)$ time.
2. **Advance:** The time spent in all Advance steps is $\mathcal{O}(mn)$ since there are at most $2mn$ advances in each phase. This is because there at most n advances before an augment or retreat and there are at most m augments (since at least one edge is deleted in each augment) and n retreats (at least one vertex is deleted in each retreat).
3. **Retreat:** Takes $\mathcal{O}(m + n)$ time. Beause there are most n retreats in each phase and each retreat takes $\mathcal{O}(1)$ time plus time to delete edges which is $\mathcal{O}(m)$.
4. **Augment:** The time spent in all augment steps is $\mathcal{O}(mn)$ time. This is because there are at most m augments in each phase and each augment takes $\mathcal{O}(n)$ time.

Thus each phase requires $\mathcal{O}(mn)$ time and there are at most n phases. Hence total running time of Dinitz algorithm is $\mathcal{O}(mn^2)$.

13.3.3 MPM Algorithm

MPM is similar to Edmonds-Karp and Dinitz (i.e., the algorithm consists of phases, and each phase finds a blocking flow using the current level graph), but blocking flows are found for level graphs in $\mathcal{O}(n^2)$ time. We consider the capacity of a vertex as opposed to the capacity of an edge.

Definition 13.4

The *capacity* $c(v)$ of a vertex v in a flow network is the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

$$c(v) = \min\left\{\sum_{u\in V} c(u,v), \sum_{u\in V} c(v,u)\right\}$$

The main idea of the MPM algorithm is to push a flow equal to the minimum capacity among all vertices. Note that this is always possible, since every vertex can support this much amount of flow going through it.

The MPM algorithm proceeds in phases. In each phase, we compute the residual graph for the current flow and also compute the level graph. If t does not appear we are done. Otherwise, all vertices not on a path from s to t in the level graph are deleted. We now a find a blocking flow as follows.

MPM Algorithm: One Phase

Step 1: Find a vertex v of minimum capacity d. If $d = 0$ goto step 2 else Go to **Step 3**.

Step 2: Delete v and all its incident edges and update the capacities of the neighboring vertices. Go to **Step 1.**

Step 3: **Push** d units of flow from v to sink and **pull** d units of flow from the source to v to increase the flow through v by d:

Push to sink: Saturate the outgoing edges of v in order, leaving *at most one partially saturated edge*. (In other words, send flow in a greedy fashion, saturating each edge fully, with possibly only one edge that might be partially saturated.) Repeat this process on each vertex (all the way up to t) that received flow during the saturation of the edges out of v. Delete all saturated edges. Update capacities of all nodes.

Pull from source: Saturate the incoming edges of v in order, leaving at most one partially saturated edge. Repeat this process on each vertex (all the way back to s) from which flow was taken during the saturation of the edges out of v. Delete all saturated edges. Delete v and all its remaining incident edges from the level graph. Update the capacities of all the nodes. Go to **Step 1**.

Running time

We compute the time needed for one phase.

- $\mathcal{O}(m)$ time is needed to compute the residual graph and level graph using BFS.
- $\mathcal{O}(n \log n)$ time to find and delete a vertex of minimum capacity using Fibonacci (or binary) heaps.
- $\mathcal{O}(m)$ time to delete all the (fully) saturated edges.
- $\mathcal{O}(n^2)$ time to do partial saturations, because it is done at most once in Step 3 at each vertex for each choice of v in step 1.

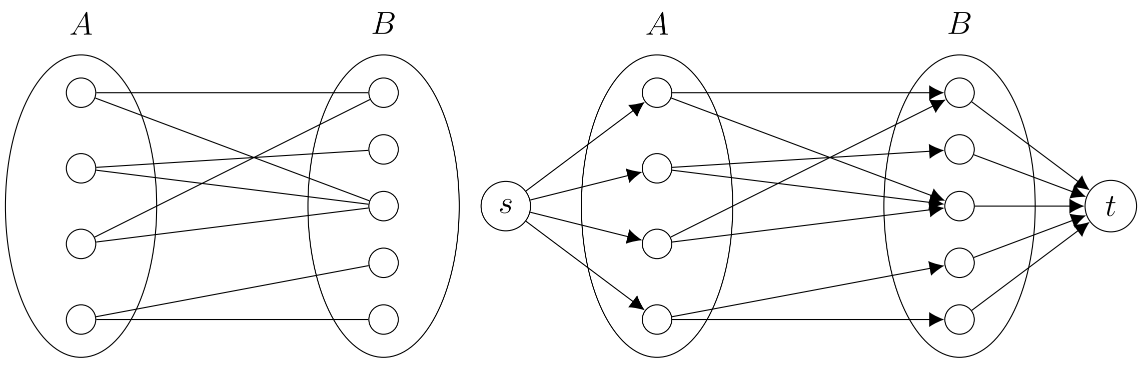


Figure 13.3: A bipartite graph G (top) and the directed graph G' (bottom) which is a flow graph obtained from G. All capacities in G' are 1.

- $\mathcal{O}(n^2)$ time to decrement capacities of incident vertices using Fibonacci heaps. (If binary heaps are used, then this will take $\mathcal{O}(n^2 \log n)$ time, since each DecreaseKey operation takes $\mathcal{O}(\log n)$ time in a binary heap.)

Thus, one phase takes $\mathcal{O}(n^2)$ time and there are at most n phases, thus the overall running time of MPM algorithm is $\mathcal{O}(n^3)$.

13.4 Matching

Matching is another fundamental graph problem. A matching is defined as follows.

Definition 13.5

Given a graph $G = (V, E)$ a set of edges M is a **matching** in G if

1. $M \subseteq E$;

2. no two edges of M share the same node;

A **maximum** matching in (unweighted graph) G is a matching with maximal cardinality, i.e., the maximum number of edges.
A graph has a **perfect** matching if it has a matching of size $|V|/2$ (i.e. every vertex is covered by the matching).

In this section, we show an algorithm to solve the matching problem in bipartite graphs. This is done by giving a simple *reduction* to the maximum flow problem.

13.4.1 Bipartite Matching Through Maximum Flow

A graph $G = (V, E)$ is **bipartite** if the set of vertices V can be partitioned to two sets A and B, ($A \cup B = V$ and $A \cap B = \emptyset$), such that every edge in E has one adjacent vertex in A and one in B (i.e., there are no edges between two vertices in A or two vertices in B).

Given a bi-partite graph $G = (A, B, E)$ we can use a maximum flow algorithm to find maximum matching in G as follows (see Figure 13.3).

Define a directed graph $G' = (V'E')$:

- $V' = A \cup B \cup \{s, t\}$

- Connect s to all vertices in A. Direct all edges in E from A to B. Connect all vertices in B to t.
- All edges have capacity 1.

A flow is **integer value** if the flow through each edge is an integer value.

Theorem 13.11

The Ford-Fulkerson algorithm generates an integer value flow in G'.

Proof. Note that each augmentation can be used to push a flow of 1. □

Theorem 13.12

The cardinality of the maximum matching in G equals the value of the maximum flow in G'.

Proof. 1. We show that if f is the size of the maximum matching in G, then we can push a flow of f in G'. This is clear, since the matching edges do not share a common endpoint, one can push a flow of 1 through each edge from s to t.

2. We show that if one can push an integer flow of f in G', then there is matching of size f in G. Since each node in A and B can be adjacent to only one vertex with integer flow > 0, a flow f in G' corresponds to a matching M in G with $|f| = |M|$. □

Running time

The running time of the maximum flow, using the Ford-Fulkerson algorithm, is $\mathcal{O}(|V||E|)$, since the Ford-Fulkerson runs in $\mathcal{O}(|E||f|)$ time, where $|f|$ is the maximum flow, which is $\mathcal{O}(|V|)$ here.

13.5 Minimum Cut

We study the minimum (min) cut problem, another fundamental graph optimization problem. Recall, that given a cut $(S, T = V - S)$, the size of the cut is the number of edges crossing the cut. (In the weighted case, it is the sum of the weights of the edges crossing the cut.) The min cut problem is to find the minimum sized cut in the graph. There are two variants of the problem: (1) $s - t$ min cut which finds a min cut where s and t are on opposite sides of the cut and (2) global min cut, which is the min cut over all cuts.

Another way to define the global min cut is the minimum set of edges needed to *disconnect* the graph into two or more pieces. Thus clearly, the global min cut cannot be larger than the *minimum degree* of the graph. (Why?)

One obvious way to find the $s - t$ min cut is to use the max flow-min cut duality theorem that we showed. Find the maximum flow between s and t also gives the size of the min cut. (One can also recover the edges also from the max flow. How?) To find the global min cut, one can find the $s - t$ min cut over all pairs of vertices and take the minimum. However, it can be shown that the global min cut can be computed using one max flow computation. Even this takes $\mathcal{O}(mn)$ time, since the best max flow algorithm

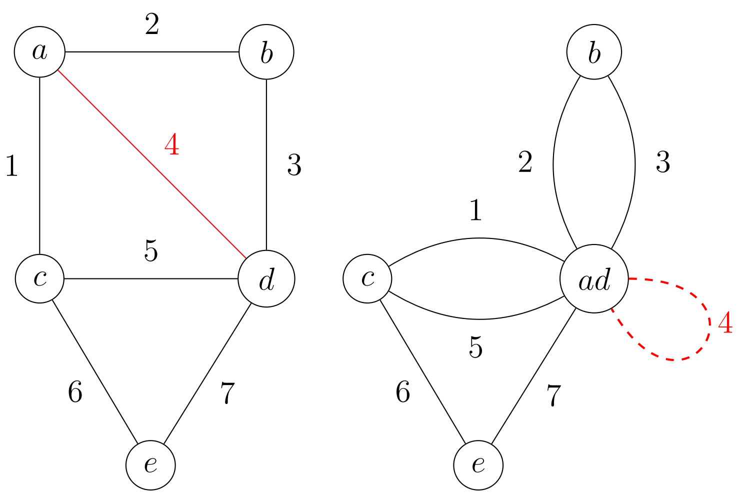


Figure 13.4: A graph G (left) and the contracted graph G' (right) obtained by contracting edge (a, d) (labeled 4). Note that self-loop (shown in dotted line) is not present in the contracted graph, but multi-edges are retained.

takes that much time (here we saw an $\mathcal{O}(n^3)$ algorithm). All these are deterministic algorithms.

In this section, we will describe a randomized min cut algorithm that finds the global min cut in time $\mathcal{O}\left(n^2 \log^3 n\right)$ algorithm which is significantly faster than the best known deterministic algorithm of $\mathcal{O}(mn)$ in dense graphs. The randomized algorithm is conceptually quite simple and does not use any flow computation.

13.5.1 A Randomized Min Cut Algorithm

The algorithm (due to Karger) is as follows. Starting from a n-node graph, it repeatedly chooses a random edge and contracts the edge. A contraction operation is defined as follows: contacting an edge (u, v) is equivalent to coalescing u and v into a single vertex (say z) and making the neighbors of u and v as neighbors of z; any self loops created are deleted, while *multi-edges are retained*. See Figure 13.4 for an example. Note that each contraction operation reduces the number of vertices by one. The algorithm stops when the number of vertices are reduced to 2; in that situation the edge between the two vertices are output. The pseudocode is as follows.

Algorithm 61 Karger – Contract Until t nodes remain
Input: A graph G of n and an integer $t \geqslant 2$
Output: A minimum set of edges that disconnects the graph

```
func KARGER($G$, $t$):
    for $i = 1$ to $n - t$:
        sample an edge uniformly at random
        contract the edge
    return the set of edges connecting the $t$ remaining vertices
```

Analysis of the Algorithm

The following is the key theorem that lower bounds the success probability of the algorithm.

Theorem 13.13

For $t = 2$, algorithm Karger outputs a min-cut set of edges with probability at least $\frac{2}{n(n-1)}$.

We need the following crucial lemma.

Lemma 13.14

For $t \geqslant 2$, the contraction operation (61.4) does not reduce the size of the min-cut set.

Proof. Every cut set in the new graph is a cut set in the original graph. This is because deleting the cut set in the contracted graph also disconnects the original graph (before contractions). □

Assume that the graph has a min-cut set of k edges. We compute the probability of finding *one such set* C. The proof of the following lemma is easy and left as an exercise (see Exercise 13.10).

Lemma 13.15

Let C be a min-cut set of G. If the run of the algorithm did not contract any edge of C, it also did not eliminate any edge of C.

Now we are ready to prove Theorem 13.13.

Proof. Since the minimum cut-set has k edges, all vertices have degree $\geqslant k$, and the graph has $\geqslant nk/2$ edges.

Let the event E_i denote the event that "the edge contracted in iteration i is not in C". We want

$$\mathbb{P}\left(\cap_{i=1}^{n-2} E_i\right) = \mathbb{P}(E_1)\,\mathbb{P}(E_2 \mid E_1)\,\mathbb{P}(E_3 \mid E_2 \cap E_1) \ldots \mathbb{P}\left(E_{n-2} \mid \cap_{i=1}^{n-3} E_i\right)$$

$$\mathbb{P}(E_1) \geqslant 1 - \frac{k}{nk/2} = 1 - \frac{2}{n}$$

$$\mathbb{P}(E_2 \mid E_1) \geqslant 1 - \frac{k}{(n-1)k/2} = 1 - \frac{2}{n-1}$$

$$\mathbb{P}\left(E_i \mid \cap_{j=1}^{i-1} E_j\right) \geqslant 1 - \frac{2}{n-i+1}$$

Thus,

$$\mathbb{P}\left(\cap_{i=1}^{n-2} E_i\right) \geqslant \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)}$$

□

The following lemma generalizes the success probability (of outputting a minimum cut) given by the above lemma to the situation where we stop the Karger's algorithm when we reach $t \geqslant 2$ vertices. The proof is similar to the previous lemma which we leave as an exercise.

Lemma 13.16

Show that, for any $t \geqslant 2$, a min cut will survive in the t-vertex graph output by Algorithm *Karger* with probability at least $\frac{t(t-1)}{n(n-1)}$.

A High Probability Algorithm

The success probability of Karger's algorithm is very small, i.e., $\frac{2}{n(n-1)}$. One can easily boost up the success probability to be close to 1 by repeating the algorithm about $n^2 \ln n$ times.

Algorithm 62 HighProbabilityKarger
Input: A Graph G of n vertices
Output: The minimum cut of G

```
func HighProbabilityKarger($G$):
    for $i = 1$ to $n^2 \ln n$:
        $C_i$ = Karger($G, 2$)
    return the set with minimum size among the $C_i$s
```

Theorem 13.17

Algorithm HighProbabilityKarger outputs a min-cut set with probability at least $1 - \frac{1}{n}$.

Proof. The probability that the algorithm fails to output the min cut in one iteration of the for loop is $1 - \frac{2}{n(n-1)} \leqslant 1 - 1/n^2$. Since all the iterations are independent of each other (in terms of the random choices), the probability that the min cut is not output in all the $n^2 \ln n$ iterations is at most

$$(1 - 1/n^2)^{n^2 \ln n} \leqslant e^{-n^2 \ln n/n^2} \leqslant e^{-\ln n} = 1/n$$

(Note that we applied the inequality $1 - x \leqslant e^{-x}$, which holds for all $|x| \leqslant 1$.) □

Running time

The running time of the algorithm depends on the time needed to implement one iteration of Karger's algorithm. Exercise 13.2 asks you to show that one step of the iteration — choosing a random edge and contracting it — can be implemented in $\mathcal{O}(n)$ time; hence Karger's algorithm (one iteration) can be implemented in $\mathcal{O}(n^2)$ time.

Thus the high probability algorithm, which requires $\mathcal{O}(n^2 \ln n)$ iterations of Karger's algorithm, requires $\mathcal{O}(n^4 \ln n)$ time which is significantly more than the best known $\mathcal{O}(mn)$ deterministic algorithm. How can we improve Karger's algorithm?

Exercise 13.2. Show that one step of Karger's algorithm — choosing a random edge and contracting it (61.3 and 61.4) can be implemented in $\mathcal{O}(n)$ time.

13.5.2 Improving Karger's Algorithm*

We notice that the main weakness of the Karger's algorithm is towards the end, when there are only a few vertices remaining, when the probability of contracting an edge belonging to the min cut increases. In particular in the last step (when there are 3 vertices), the probability of contracting such an edge becomes $2/3$ which is large. Note that, on the other hand, when there are a large number of vertices, the probability of contracting an edge belonging to the min cut is small. This suggests an idea. Use Karger's algorithm to contract the graph until t vertices are remaining (for a suitable t) and then use a slower deterministic algorithm to find the min cut in the remaining graph which always succeeds. Exercise 13.7 asks you to show that this idea does gives you a better algorithm with a running time of $\mathcal{O}(n^{2+\epsilon})$ for some constant ϵ, but it does not give a $\mathcal{O}(n^2 \operatorname{polylog} n)$ algorithm.

The key idea of the improved algorithm is to perform Karger's iteration *two* times (each independently) on G until the number of vertices is reduced by a constant factor. This gives two contracted graphs G_1 and G_2. Find the min cut in G_1 and G_2 and output the minimum of the two cuts. To find the min cut in G_1 and G_2, we apply the algorithm recursively. The pseudocode is as follows:

Algorithm 63 ImprovedCut – Improvement on Karger's Algorithm
Input: A graph G of n nodes
Output: A min-cut of G

```
func ImprovedCut($G$):
    if $n \leqslant 10$:
        return MinCut($G$)    ▷ Brute Force Search
    $t = \frac{n}{\sqrt{2}}$
    $G_1$ = Karger($G, n - t$)
    $G_2$ = Karger($G, n - t$)
    $C_1$ = ImprovedCut($G_1$)
    $C_2$ = ImprovedCut($G_2$)
    return min($C_1, C_2$)
```

Intuition. The main intuition of the choosing the value of t as in the algorithm is as follows. First, consider the recursion tree of the ImprovedCut algorithm. It is a binary tree with G as the root which has two subtrees — roots of the subtrees are G_1 and G_2 respectively. The size of G goes down by a factor of $\sqrt{2}$. So the depth of the recursion tree is about $2 \log n$ (why?). So the number of leaves is about $2^{2 \log n} = n^2$. Now consider Karger's algorithm, which has a (small) success probability of $\Theta(1/n^2)$. We repeated Karger's algorithm about n^2 times (actually $n^2 \ln n$ times to be precise) to boost the success probability (Algorithm High-Probability-Karger). The High-Probability-Karger can be represented as a tree of depth 1, with G as the root, and the n^2 repetitions forming the n^2 leaves. Thus both trees have about the same number of leaves (which is required since the $1/n^2$ success probability has to be boosted up by so many repetitions), but the ImprovedCut Algorithm shares a lot of work in the initial steps (where the success probability is large), leading to an improved algorithm, unlike the High-Probability-Karger, which repeats all the iterations from scratch.

Run time analysis

We can analyze the run time by writing a recurrence (which is natural, since we are dealing with a recursive algorithm).

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + \mathcal{O}\left(n^2\right)$$

The $\mathcal{O}(n^2)$ term comes from the fact that we need the run $n - n/\sqrt{2}$ steps of the Karger algorithm which takes $\mathcal{O}(n^2)$ time (actually the whole Karger algorithm — running for $n-2$ steps — takes $\mathcal{O}(n^2)$ time). The base case is $T(n) = \mathcal{O}(1)$, for $n \leqslant 10$. Using the DC recurrence theorem (Theorem 2.2, Chapter 2), we can show that $T(n) = \mathcal{O}(n^2 \ln n)$.

Success probability analysis

The crucial thing is to show that the success probability, i.e., the probability of outputting a min cut, of the ImprovedCut algorithm is large. Again, we write a recurrence and solve. Let $P(n)$ denote the probability of the Improved Cut algorithm to output a min cut on an input graph of n vertices. Then we can write

$$P(n) \geqslant 1 - (1 - 1/3(P(n/\sqrt{2})))^2 \tag{13.6}$$

The reason for the above is as follows. Consider the ImprovedCut algorithm operating on G with n nodes. The probability that the min cut will survive $n - n/\sqrt{2}$ contractions is at least $1/3$ (by applying result of Lemma 13.16): $\frac{(n/\sqrt{2})(n/\sqrt{2}-1)}{n(n-1)} \geqslant 1/3$, for $n \geqslant 10$. The algorithm will output a min cut if the min cut survives the contraction and the min cut is found successfully in the contracted graph of $n/\sqrt{2}$ nodes. Since we do two independent contractions, the algorithm will succeed if min cut is found in one of the two cases. The base case of the recurrence is $P(n) = 1$, for $n \leqslant 10$.

Exercise 13.11 asks you to show that the solution of the above recurrence is $\Omega(1/\log n)$. Hence ImprovedCut algorithm finds a min cut with probability at least $\Omega(1/\log n)$. Note that, this is significantly better than Karger's algorithm (one iteration) $\Theta(1/n^2)$ success probability.

Thus, by repeating the ImprovedCut algorithm $\Theta\left(\log^2 n\right)$ times, we get a high probability algorithm.

13.6 Worked Exercises

Worked Exercise 13.1. A *path flow* in G is a flow that takes nonzero values only on some simple path from s to t. In other words, there exist a number d and a simple path $u_0, u_1, \ldots u_k$ with $s = u_0$, $t = u_k$, and such that

$f(u_i, u_{i+1}) = d$, $0 \leqslant i \leqslant k-1$
$f(u_{i+1}, u_i) = -d$, $0 \leqslant i \leqslant k-1$
$f(u,v) = 0$ for all other (u,v), $u, v \in V$.

Recall that the smallest (residual) capacity along an (augmenting) path is called the *bottleneck* capacity of that path.

(a) Show that any flow in G can be expressed as a sum of at most m path flows in G and a flow of value 0, where m is the number of edges of G. (Hint: Let f be a flow in G and let $|f| > 0$. Consider G', a graph with the new capacity function $c'(e) = \max\{f(e), 0\}$. Consider the flow with value 0 in G'.)

(b) Suppose we use the Ford-Fulkerson algorithm with the following heuristic for finding an augmenting path: always choose an augmenting path of *maximum* bottleneck capacity.

Show that if the edge capacities are integers, then the above heuristic results in a maximum flow f^* in at most $\mathcal{O}(m \log |f^*|)$ augmenting steps, where m is the number of edges. (Hint: How can you bound the flow pushed after each augmenting step? Use the result of part (a). You might find the following estimate useful in your analysis: $\log(m/(m-1)) = \Theta(1/m)$.)

(c) Give an algorithm to find a maximum bottleneck capacity augmenting path in a flow graph. (Hint: Modify Dijkstra's algorithm.)

Solution.

(a) Intuitively, the decomposition of a flow into simple paths can be understood from the Ford-Fulkerson (FF), where at each step we augment by a simple path (there is positive flow only along the augmenting path and zero flow elsewhere). For this question, however, we have to show this, since the flow could have been constructed by any means (not necessarily by using FF). We do this by decomposing the flow using augmenting paths as follows.

Let f be a flow in G. If $|f| = 0$, we are done. Otherwise, then $|f| > 0$. As given in the hint, define a new capacity function $c'(e) = \max\{f(e), 0\}$ and let G' be the graph with these capacities. Then f is also a flow in G', and since $c'(.) \leqslant c(.)$, any flow in G' is also a flow in G. By the Max flow-Min Cut Theorem, the zero flow in G' must have an augmenting path, which is a (simple) path from s to t with positive capacities. Note that by construction, every edge of G' is saturated by flow f. Take p to the path flow on that path whose value is the bottleneck capacity, i.e., set every flow on that path equal to the bottleneck capacity of that path. Then the two flows p and $f - p$ are both flows in G' and at least one edge on the path is saturated by p. Remove the saturated edge (but do not add the reverse edge — here we are not computing the residual graph, but simply subtracting the flow p from flow f). Now we repeat this process with $f - p$ to get $c''(.) < c'(.)$ and so on. Note that G'' has strictly fewer edges than G', since the bottleneck edge has disappeared. This process can be repeated at most m times before the flow value becomes zero. The original flow f is then the sum of the flow of value zero and the paths flows found in each step.

(b) By part (a) any flow f can be decomposed into at most m path flows. Thus the maximum value of these (at most) m flows is at least $|f|/m$. Thus if we augment along the path with the maximum bottleneck capacity, we push a flow of at least $|f|/m$. Then the flow (in the residual graph) that remains to be pushed will be at most $|f| - |f|/m = |f|(1 - 1/m)$. We then repeat the process. Since we reduce the remaining flow by a factor of at least $1 - 1/m$, the number of augmenting path iterations is at most $\frac{\log |f|}{\log(1-1/m)} = m \log |f|)$ (using the fact that $\log(1 - 1/m) = \Theta(1/m)$).

(c) Although finding the path with the maximum bottleneck capacity appears to be very different from the shortest paths problem, there is a uniform way of looking at both the problems. In the shortest path problem, the “value” of a path is the sum of weights in the path, and we want to find (among all possible paths) the one with minimum value. In the maximum bottleneck problem, the “value” of a path is the minimum weight edge of that path and we want to find the path with the maximum value. Thus, the definition of “value” differs, and instead of finding a path of minimum value (as in shortest paths), we are finding a path with maximum value.

A modification of Dijkstra’s algorithm can be used to compute the maximum bottleneck value path between s (the source) and all other vertices, see Algorithm MaxBottleneck.

We can prove the correctness of this algorithm by showing that the following two invariants hold by induction on the number of iterations (or the size of S). The proof is very similar to the proof of Dijkstra’s algorithm for computing shortest paths in a graph.

1. if $v \in S$ then $key[v]$ is the value of the maximum bottleneck value path of v from s, and $parent[v]$ encodes the last edge of that path.
2. if $v \notin S$ then $key[v]$ is the value of the maximum bottleneck path of v from s using only internal vertices of S, and $parent[v]$ encodes the last edge of that path.

Note that we can use a binary max-heap to implement the heap operations — extract-max and increase-key. The running time is the same as Dijkstra’s algorithm.

□

13.7 Exercises

Exercise 13.3. Given an undirected graph $G = (V, E)$ and elements $s, t \in V, s \neq t$, decide whether there exist k edge-disjoint paths from s to t, and find them if so. Provide an efficient polynomial time algorithm.

Exercise 13.4. A unit network $G = (V, E)$ is a network in which all edge capacities are integers and each vertex $v \in V$ other than the source s and sink t has either a single entering edge, of capacity one, or a single outgoing edge, of capacity one.

a) Show that on a unit network, Dinitz algorithm halts after at most $\mathcal{O}\left(\sqrt{|V|}\right)$ phases (each phase finds a blocking flow).

b) Show that the above result can be used to obtain an $\mathcal{O}\left(\sqrt{|V|}|E|\right)$-time algorithm for finding a maximum matching in unweighted bipartite graphs.

Exercise 13.5. Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. We know the maximum flow value in G. Suppose that the capacity of a single edge $(u, v) \in E$ is increased by 1. Give an $\mathcal{O}(|V| + |E|)$ algorithm to update the maximum flow.

Exercise 13.6. Modify Karger's algorithm so that it finds a minimum cut in a weighted graph where all weights are integers. (The min cut is the set of edges with the smallest total weight whose removal disconnects the graph.)

Exercise 13.7. Suppose you are given a deterministic algorithm for finding min cut that runs in $\mathcal{O}(mn)$ time. Show how you can combine this algorithm with Karger's algorithm (that runs until t vertices remain, for some suitably chosen t) to obtain an algorithm that runs in time $\mathcal{O}(n^{2+\epsilon})$ time. How small can ϵ be?

Exercise 13.8. Generalizing on the notion of a cut-set, we define an $r-$way cut-set in a graph as a set of edges whose removal breaks the graph into r or more connected components. Explain how the randomized min-cut algorithm can be used to find minimum r-way cut-sets, and bound the probability that it succeeds in one iteration.

Exercise 13.9. Let $G = (V, E)$ be an undirected graph with m edges and n vertices. For any subset $U \subseteq V$, let $G[U]$ denote the subgraph induced on U, i.e., the graph with vertex set U and whose edge set consists of all edges of G with both ends in U. Given a number k ($1 \leqslant k \leqslant n$), your task is to design a polynomial time algorithm that produces a set $U \subseteq V$ of k nodes with the property that the induced subgraph $G[U]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

(a) Give a deterministic polynomial time algorithm for the problem. Show that your algorithm indeed produces the desired set. What is the running time of your algorithm? (Hint: Try a greedy strategy.)

(b) Give a Las Vegas randomized algorithm that has expected polynomial running time. Show correctness and running time analysis.

Exercise 13.10. Prove Lemma 13.15.

Exercise 13.11. Consider the recurrence $P(n)$ defined in Equation 13.6. Show that its solution is $\frac{c}{\log n+0.5}$ for some (large) constant $c > 0$. (Hint: Use induction).

Appendices

APPENDIX A
MATHEMATICAL INDUCTION

Mathematical induction is a simple but very powerful technique to prove mathematical theorems. It is also extremely useful in proving correctness of algorithms as well as in understanding the insight behind their design; it is also useful in analyzing the performance of algorithms. Before we illustrate mathematical induction with some examples, a few words about the general technique. Suppose we want to prove some mathematical statement that depends on some parameter n which can be any natural number. Examples are: (1) the sum of the first n numbers is $n(n+1)/2$; (2) $\binom{2n}{n} \geqslant 2^n$; (3) the running time of an algorithm on input size n is at most $2n$ etc. The goal is to show that the mathematical statement is true, i.e., it is true for all natural numbers. Note that it is generally easy, at least in principle, to verify the statement for a particular value of n, e.g., if $n = 100$, the first statement can be checked by simply summing up the first 100 numbers (may take a while) and verifying that it equals (100)(101)/2. However, this is not a valid approach to proving the statement *for every natural number*. Mathematical induction is a technique for precisely doing this. (Note that in many cases, there are alternative ways to prove such statements, but induction is the most straightforward and "automatic" way to do that; indeed there are important mathematical theorems where the only known proof uses induction.)

A.1 The General Technique

How does mathematical induction work? Suppose we want to show that a mathematical statement is true for all natural numbers $n \geqslant 0$. Mathematical induction consists of two steps: (1) Base step and (2) Induction Step. The base step is verifying whether the statement is true for $n = 0$. This is, of course, easy, since we have to verify for just one value. The induction step is the crucial step: *assuming that the statement is true for natural number k*, we have to *prove* that the statement is true for $k + 1$. It is important to note that k is a variable and not any fixed number, and thus proving the induction step involves proving the statement for any (general) k. The assumption that the statement is true for natural number k is usually called the *induction hypothesis*. If we show these two steps — the base step and the induction step — then by mathematical induction it follows that *the statement is true for all natural numbers*. Why is this the case? This is not difficult to see if we iterate through the numbers. The base step shows that the statement is true for $n = 0$. Applying the induction step for $k = 0$ (since we show it for any number k), it holds for $k = 1$. Since it holds for $k = 1$, we can again use the induction step to show the statement for $k = 2$. Again, since it holds for $k = 2$, we can again use the induction step to show the statement for $k = 3$ and so on. Repeating this argument, it is clear that the statement holds for every natural number. This is the essence of a proof by mathematical induction.

The power of mathematical induction — which naturally underlies recursive algorithms (and iterative algorithms as well) — comes from the induction hypothesis which simply assumes that the statement is true for k (somehow by "magic" !). This gives us a lot of "ammunition" to prove that the statement is true for $k + 1$.

A.1.1 Sum of first n natural numbers

Let us illustrate the technique of mathematical induction by proving a basic mathematical statement (which also arises frequently in analysis of algorithms): the sum of the first n natural numbers is $n(n + 1)/2$.

Theorem A.1

$$0 + 1 + 2 + \cdots + n = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Proof. We prove by using mathematical induction on the variable n. (Note that it is a good idea to say the variable that we are applying induction to. In this example, it is obvious since there is only one variable, but in others, there may be more than one variable in the statement, and it better to be explicit about it. We will see more examples throughout this book.)

We have to show the following two steps:

Base step: $n = 0$: The statement is trivially true, since $0(1)/2 = 0$ which is indeed the sum of the first number (0).

Induction step: Assume that the statement is true for k (induction hypothesis). We will prove that the statement is true for $k + 1$.

By the induction hypothesis, we have

$$0 + 1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

Using the above, we have to *prove* that the following statement holds:

$$0 + 1 + 2 + \cdots + k + k + 1 = \frac{(k+1)(k+2)}{2} \tag{A.1}$$

How can we show that? We use the induction hypothesis that already tells us the value of the sum up to k. Thus the left hand side of Equation (A.1) becomes

$$\frac{k(k+1)}{2} + k + 1$$

which simplifies to $\frac{(k+1)(k+2)}{2}$ which is the right hand side of Equation (A.1) as desired.

Thus we have proved the induction step and hence the statement is proved by mathematical induction. □

A.1.2 Coloring regions formed by lines

We will prove a simple, but somewhat surprising result on coloring regions formed by lines. Consider n distinct (infinite) lines drawn in a plane. A region is an area of the plane that is enclosed by lines; note that a region can be finite (closed) or infinite (open). The goal is to assign colors to regions so that no two adjacent regions (i.e., regions that share a line as a boundary) have the *same* color — such a coloring is called *valid.* For example, if there is only one line, it partitions the plane into two (adjacent) regions and if there are two intersecting lines then it partitions the plane into four regions where each region is adjacent to two other regions. How many colors are needed to validly color regions formed by n lines? The following theorem shows that only two colors are needed.

Theorem A.2

Two colors are needed to validly color all regions formed by any number of lines.

Proof. The proof is by induction on n, the number of lines.

Base step: $n = 1$. Then the line partitions the plane into two regions; color one region by blue and the other by red. Clearly this coloring is valid.

Induction step: Assume that the statement is valid for k lines. We will prove the statement when there are $k + 1$ lines. Consider $k + 1$ distinct lines drawn in a plane. We will demonstrate the existence of a valid coloring of the regions formed by these $k + 1$ lines using two colors. Consider the set of $k + 1$ lines and remove one line from the set. This leaves us with k lines and by the induction hypothesis, there is a valid coloring of the regions formed by it using 2 colors — red and blue. Assume this valid coloring. Now let us add the $(k + 1)$th line and we will show how to modify the colors of the regions to still have a valid coloring. We leave the colors of all regions that are on one side of the $(k + 1)$th line unchanged and *reverse* the colors of the regions on the other side. We claim that this is a valid coloring. Consider the regions formed by k lines. When the $(k + 1)$th line was added, it cuts some regions formed by the k lines into two parts — one on each side. Since we reverse the color on one side of the $(k + 1)$th line, these "cut" regions have a valid coloring. Now consider the regions that are *not* cut by the $(k + 1)$th line, i.e., those that are entirely on either side of the line. These are (already) validly colored by the induction hypothesis, since these are regions formed by k lines. □

Algorithm via Induction Proof

The above induction proof also suggests a natural algorithm that gives a valid coloring of the regions formed by n lines. The algorithm is naturally incremental. Start with one line and use two colors to color the two regions on either side — this is the base case. Then add one line at a time. After adding a line, simply reverse the colors of the regions on one side of the added line, leaving the other side unchanged — this is the strategy used in the induction step. Clearly this algorithm gives a valid coloring and the proof of its correctness is the indeed the induction proof!

This example illustrates how algorithms can be designed by using induction proofs; as an added bonus the proof of correctness of the algorithm follows from the induction proof.

Weak Induction

The above form of mathematical induction is sometimes called the "weak form of induction" as the induction hypothesis assumes only the validity of the statement for value k (and not for values less than k — as done in the "strong form" as explained in the next section). This is already useful in proving correctness of many algorithms, especially *iterative* algorithms, which operate incrementally on the input size. In particular, it is useful in proving "loop invariant" in a *for* or *while* loop. A loop invariant is a property that holds in every iteration of the for loop; if it fails to hold the loop terminates. Correctness of many algorithms with for and while loops involve showing statements on loop invariants. Such examples are shown throughout the book.

A.2 Strong Mathematical Induction

In many cases, the induction hypothesis can be *strengthened* for "free." Instead of assuming that the statement is true for k, we can assume that the statement is true for all natural numbers $n \leqslant k$ and then proceeding to show, under this assumption, that the statement is true for $k + 1$. The strengthened hypothesis, called *strong inductin*, is extremely useful, leading to simpler proofs. It is especially useful in proving correctness (and understanding) of *recursive* algorithms, e.g., see Chapter 4.

A.2.1 Fundamental Theorem of Arithmetic

We will illustrate the strong form of induction using the following example, which is an important theorem in mathematics: *the fundamental theorem of arithmetic* which states that every natural number greater than 1 can be expressed as a product of prime numbers. Recall that a prime number is a positive integer (greater than 1) that is divisible only by itself and by 1.

Theorem A.3

Every natural number $n > 1$ can be expressed as a product of prime numbers.

Proof. We will prove by induction on n.

Base case: $n = 2$ (note that this is the base case, as it is the smallest natural number for which the theorem is supposed to hold). Since 2 is prime, the statement is trivially true.

Induction step: Assume that the statement is true for all natural numbers from 2 to k (induction hypothesis). Note that we are assuming the strong form of the induction which is needed for this proof. We will prove that the statement is true for $k + 1$.

There are two cases to consider. Suppose $k + 1$ is prime — in this case, the statement is trivially true. Suppose $k+1$ is not a prime number. Then $k+1$ can be decomposed into a product of two numbers a and b, i.e., $k + 1 = ab$, where $1 < a < k+1$ and $1 < b < k+1$ (in fact they are both strictly smaller than k). Since a and b are strictly smaller than $k+1$ and strictly greater than 1, induction hypothesis applies to both of them, and hence each of them can be written as a product of prime numbers. Hence $k + 1$ which is a product of a and b can be written as a product of prime numbers. □

Note that the strong form was needed for the above proof, since a and b are numbers that are smaller than k.

A.3 Using Induction for Algorithm Analysis

One weakness of induction is that one needs to know the exact statement that one needs to prove beforehand. For example, we need to know that the sum of the first n numbers is $n(n+1)/2$ and then induction can be used to prove this as demonstrated in Theorem A.1. Induction does not seem to help to *discover* the fact that the sum of the first n numbers is $n(n+1)/2$; it merely serves to *verify* it. Can induction do better? In many cases, it can, as we show as follows. Suppose we want to solve a slightly easier problem that occurs very often in analysis of algorithms: we want a good bound (not the exact value) on some sum or, more generally, some function of n. To be concrete, suppose we want to show an upper bound on the sum of the first n numbers (which actually arises in analysis of algorithms for sorting and selection) and assume that we do not know the exact formula. We want to show that the bound is $\mathcal{O}(n^a)$ which usually suffices for algorithm analysis. We want a *tight* upper bound as possible, i.e., we want a to be as small as possible. We know from Theorem A.1 that it is true for $a = 2$ (since $n(n+1)/2 \leqslant n^2$), but let us use induction to *find* this.

Let us set up the statement to be shown:

$$0 + 1 + 2 + \cdots + n \leqslant cn^a$$

where, c and a are fixed constants. We want to find the the smallest a for which the above statement is true (c can be any fixed constant).

Base step: $n = 0$. The statement is clearly true for any fixed a and c.

Induction step: Assume the statement is true for k. We will prove that it is true for $k+1$ for a suitable a.

By induction hypothesis, $\sum_{i=0}^{k} i \leqslant ck^a$ and thus,

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + k + 1 \leqslant ck^a + k + 1$$

For the induction step, we need to show that

$$ck^a + k + 1 \leqslant c(k+1)^a. \tag{A.2}$$

The above inequality will not be satisfied for $a = 1$, since that will imply that $k + 1 \leqslant c$ which is clearly not true (the left hand side increases with k, while the right hand side is a constant). Let us try $a = 2$. Plugging $a = 2$ into Inequality (A.2) and simplifying yields $\frac{k+1}{2k+1} \leqslant c$, which is true for all k if $c = 1$. Hence if we take $c = 1$ and $a = 2$, the induction step is satisfied. Thus, we have shown, *in fact, derived*, that $\sum_{i=0}^{n} i \leqslant n^2$.

A.4 Worked Exercises

Worked Exercise A.1. Show using mathematical induction that any integer whose sum of digits is divisible by 3 is divisible by 3 as well.

Solution. Base Case: $n = 0$: Trivially true.

Induction hypothesis: Assume the statement is true for all integers less than k.

Induction step: We prove the statement that it is true for k.

Consider the number k and say it has decimal expanstion $x_m x_{m-1} \dots x_1 x_0$ — consisting of $m+1$ digits with x_0 being the least significant digit and x_m being the most significant. In other words, $k = 10^m x_m + 10^{m-1} x_{m-1} + \cdots + 10x_1 + x_0$. Let us denote the sum of the digits of number k to be $s_k = x_m + x_{m-1} + \cdots + x_1 + x_0$. Let us assume that s_k is divisible by 3. Then we have to show that k is divisible by 3 as well.

Consider the integer $k-3$. We have two cases:

Case 1: The last digit of k, i.e., x_0 is greater than or equal to 3. Then the decimal representation of $k-3$ is $x_m x_{m-1} \dots x_1 x_0 - 3$ and the sum of its digits is $s_{k-3} = x_m + x_{m-1} + \cdots + x_1 + x_0 - 3 = s_k - 3$. Since s_k is divisible by 3 (by the above assumption), $s_k - 3$ is divisible by 3 as well. Hence the sum of the digits of number $k-3$ is divisible by 3. By induction hypothesis (which applies to numbers less than k), $k-3$ is divisible by 3. Hence $k - 3 + 3 = k$ is divisible by 3 as well.

Case 2: The last digit of k, i.e., x_0 is less than 3. Then the decimal representation of $k-3$ can be computed as follows. The idea is to figure out the digits of $k-3$. Let x_i ($i > 0$) be the least significant non-zero digit, i.e., it is the first digit starting from the left (except, x_0) that is non-zero.

(Example: if $k = 83001$, then $x_i = 3$, since that is the first non-zero digit from the left (apart from the leftmost digit). If we subtract 3 from 83001, then we get 82998, where x_i will be subtracted by 1, and then all 9s and then the last digit is gotten by subtracting 11 (10+1) from 3. This is because of the usual subtraction where we borrow 1 from x_i and then repeat until we reach x_0; that why have 9's in between. Note that since the sum of the digits of 83001 is divisible by 3, the sum of the digits of 82998 is divisible by 3 as well. Next we generalize the above example.)

Then the sum of its digits of $k-3$ is $s_{k-3} = x_m + x_{m-1} + \cdots + x_i - 1 + 9 + \cdots + 9 + 10 + x_0 - 3 = x_m + x_{m-1} + \cdots + x_i + \dots x_0 + -1 + 9 + \cdots + 9 + 10 - 3 = s_k + 9 + \cdots + 9 + 9 - 3$ which is divisible by 3 since s_k is divisible by 3 and the sum $9 + \cdots + 9 + 9 - 3$ is divisible by 3 (since each term is). The rest of the proof is now the same as Case 1.

Hence the sum of the digits of number $k-3$ is divisible by 3. By induction hypothesis (which applies to numbers less than k), $k-3$ is divisible by 3. Hence $k - 3 + 3 = k$ is divisible by 3 as well.

□

A.5 Exercises

Exercise A.1. Prove that the sum of the squares of the first n natural numbers is $\frac{n(n+1)(2n+1)}{6}$.

Exercise A.2. Using induction, give a tight bound on the following sum: $\sum_{i=1}^{n} i^d$, where d is a fixed constant.

Exercise A.3. Prove the following statement by mathematical induction:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

where $a \neq 1$ is some fixed real number.

By the way, this is the sum of a geometric series, a useful formula that one comes across often in algorithmic analysis.

Exercise A.4. Use mathematical induction to prove that $x^n - y^n$ is divisible by $x - y$ for all natural numbers x, y $(x \neq y)$, and n. (Hint: You will need strong induction.)

Exercise A.5. Show that for all natural numbers $n \geqslant 1$, $\binom{2n}{n} \geqslant 2^n$. Note that $\binom{2n}{n} = \frac{(2n)!}{n!n!}$.

Exercise A.6. Consider the set of first $2n$ positive integers, i.e., $A = \{1, 2, \ldots, 2n\}$. Take any subset S of $n + 1$ distinct numbers from set A. Show that there two numbers in S such that one divides the other.

APPENDIX B

BASIC DATA STRUCTURES: HEAP, STACK, QUEUE, AND UNION-FIND

We discuss three basic data structures used in algorithm design: heap, stack, and queue. We also discuss the union-find data structure which is useful in certain algorithms such as Kruskal's MST algorithm (Section 12.1.1).

B.1 The Heap Data Structure

In many applications, we want to maintain the highest (or lowest) priority item among a set of items that are dynamically changing — i.e., items are added and deleted. A *Heap* is a data structure[1] that supports four operations:

- Insert(P, k): Insert key k (along with its data) into priority queue P.
- Min(P): Return the smallest key (along with its data) in P.
- ExtractMin(P): Return and *remove* the smallest key (along with its data) from P.
- DecreaseKey(P, k, v): Decrease the value of key k to value v.

The above is called a MIN heap. Alternatively, we can define a MAX heap which has INSERT, MAX, EXTRACT-MAX, INCREASE-KEY operations.

B.1.1 Implementing a Heap

One simple way is using an array: store the keys in an array. There are two ways to store the keys in an array — *sorted* or *unsorted* — which determine the time needed for the 3 operations. Assume that there are n keys currently in the array A — $A[1], A[2], \dots, A[n]$. (We will assume that keys are all *distinct.*)

We now perform an Insert, Min, or ExtractMin operation. The costs are:

[1] Sometimes a Heap is referred to as a Priority Queue. However, a Priority Queue is an abstract data type in which higher priority element will be served before a lower priority one. A Heap is a tree-based data structure that can be used to implement a Priority Queue.

Array	Insert	Min	ExtractMin
Unsorted	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sorted	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table B.1: Performance of Operations on a Heap implemented as a sorted or unsorted array

As we will see next, a heap gives a much better way to perform the above operations.

B.1.2 Binary heap

A binary heap stores the keys in a *complete binary tree*, where each node stores one key. We recall that a binary tree is a tree where each node has *at most two* children; it is *complete* if all levels are "full" except (possibly) the last level, i.e., nodes in all levels, (except possibly the last) will have 2 children. The last level will be filled from the *left*. Figure B.1 shows a example of a complete binary tree.

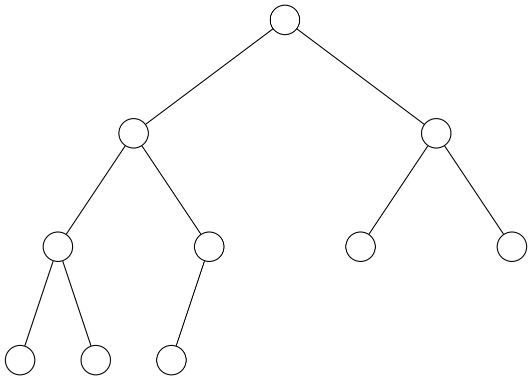


Figure B.1: A complete binary tree.

The main property of the binary (min) heap is that the keys are stored so that they satisfy the following (min) *heap* property: The key stored in any node will be *smaller* than the keys stored in its *children* (if any).

Figure B.2 shows an example of a (min) binary heap.

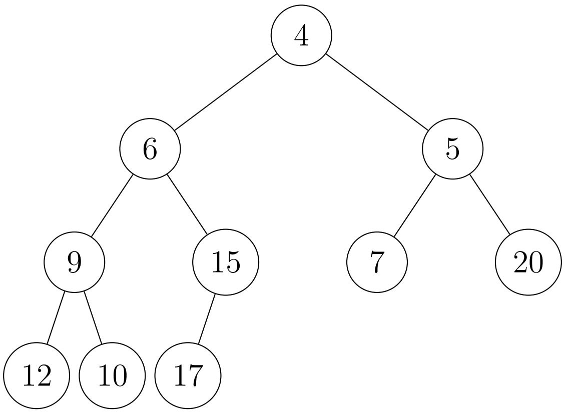


Figure B.2: A binary heap.

B.1.3 Representing a binary heap

A binary heap can be conveniently represented using an array. The indices of the vector capture the parent-child relationship. Let A be a vector. The number of the elements in the vector is given by $A.\text{SIZE}()$. The root of the binary heap is stored in $A[0]$. Given element $A[i]$, the children of this element are stored in $A[2i+1]$ and $A[2i+2]$, if they exist. The left child of i denoted as $left(i) = A[2i+1]$. The right child of i denoted as $right(i) = A[2i+2]$. The parent of $A[i]$ is stored in $A[\lfloor \frac{i-1}{2} \rfloor]$. To navigate the tree we follow the corresponding indices of the vector. See Figure B.3.

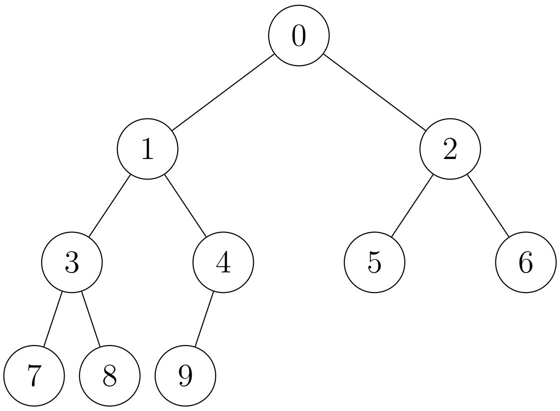


Figure B.3: The indices of an array corresponding to the nodes of a binary heap.

B.1.4 Maintaining the heap property

We use a recursive algorithm called Heapify to maintain the heap property. It takes as input a vector A and an index i into the vector. Heapify assumes that the binary trees rooted at $left(i)$ and $right(i)$ are min-heaps. But $A[i]$ might be larger than the values at its children — violating the min-heap property. Heapify moves the value of $A[i]$ *down* in the min-heap so that the subtree rooted at index i obeys the min-heap property. Heapify will be used to build a heap and also implement the priority queue operations.

The pseudocode for Heapify is given in Algorithm 64. For an illustration see Figure B.4.

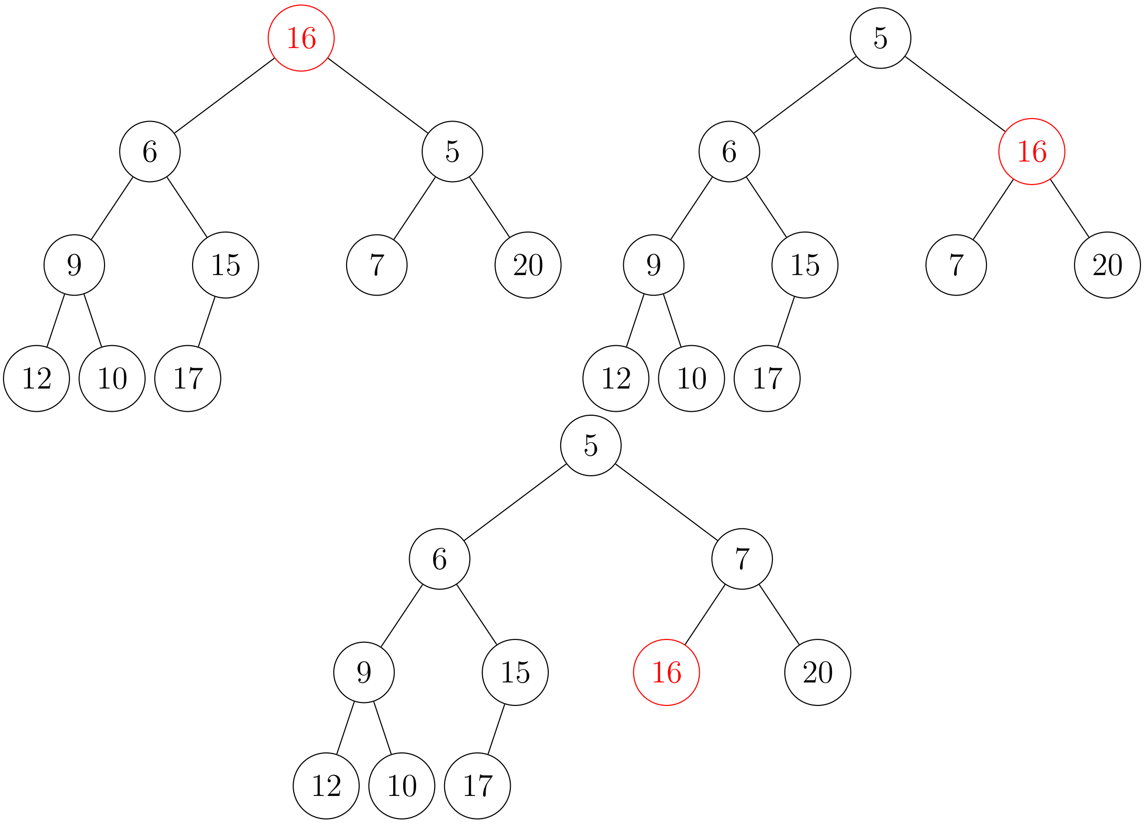


Figure B.4: Ilustrating Heapify The algorithm fixes the heap property at a node if it is violated. In this example, there is a violation at node 16 which is fixed.

Algorithm 64 Heapify

Input: An array A and an index i

Output: None. A is heapified from index i.

```
func HEAPIFY($A$, $i$):
    $\ell$ = left($i$)
    $r$ = right($i$)
    if $\ell \leqslant |A|$ and $A[i] > A[\ell]$:
        smallest = $\ell$
    else:
        smallest = $i$
    if $r \leqslant |A|$ and $A[$smallest$] > A[r]$:
        smallest = $r$
    if smallest $\neq i$:
        Swap $A[i]$ and $A[$smallest$]$
        HEAPIFY($A$, smallest)
```

Running time of Heapify

It is easy to show that the time taken by Heapify(A, i) is proportional to the *height* of node i in the binary tree. If there are n elements in the heap, the height of the *complete* binary tree is $\mathcal{O}(\log n)$. Hence Heapify takes $\mathcal{O}(\log n)$ time in the worst case.

B.1.5 Building a heap

Suppose we are given n elements in an array and we want to make it into a binary heap. We can do this by calling Heapify in a bottom-up manner. The pseudocode is given in Algorithm 65. The main idea of the algorithm is as follows. Represent the elements of the array as a binary tree (Appendix B.1.3). That is, the root of the binary heap is $A[0]$ and for $i \geqslant 0$, the children of node $A[i]$ are stored in $A[2i+1]$ and $A[2i+2]$, if they exist. Initally, the values in the binary tree need not be organized as a heap. The algorithm Heapify, will organize the elements to satisfy the (min) heap property starting from the leaves. The leaves trivially satisfy the heap property. As the algorithm works on nodes at higher levels, the nodes at the lower levels — and hence the subtree rooted at these nodes — already satisfy the heap property. Thus the conditions for Heapify are satisfied. The formal correctness is argued next.

Algorithm 65 BuildHeap
Input: An array A of n elements
Output: A heap H of the values in A

```
func BuildHeap($A$):
    $H = A$
    for $i = \lfloor n/2 \rfloor$ downto 1:
        Heapify($H, i$)
    return $H$
```

Correctness

Consider the complete binary tree representation of the array. Heapify fixes the heap property *level by level* starting from the last level and going up. Thus when Heapify is called at a node i, the subtrees rooted at $left(i)$ and $right(i)$ already satisfy the heap property. When Heapify is finally called at the root, the whole array becomes a min heap.

Running time of Algorithm BuildHeap

It is easy to see to upper bound the run time of Algorithm BuildHeap as $\mathcal{O}(n \log n)$. Actually, by a more careful analysis, it can be shown to be faster — takes $\mathcal{O}(n)$ time. We will bound the total time taken by *all* the Heapify calls. The time taken by Heapify(A, i) is proportional to the height of node i (note that leaves are at height 0). Hence the total time is bounded by:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h)$$

since there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h. Hence,

$$n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{\mathcal{O}(h)}{2^h} \leqslant n \sum_{h=0}^{\infty} \frac{\mathcal{O}(h)}{2^h}$$

The sum $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$, hence,

$$n \sum_{h=0}^{\infty} \frac{\mathcal{O}(h)}{2^h} = \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = \mathcal{O}(n)$$

B.1.6 Implementing Heap Operations

The operations Insert, Min, ExtractMin, and DecreaseKey be implemented using a binary heap. Given a binary heap A, we implement the above operations as follows.

Min

Simply return $A[0]$. This takes constant time.

Algorithm 66 Min – Return the root value of the heap

```
func Min($A$):
    return $A[0]$
```

ExtractMin

Algorithm 67 ExtractMin – Remove and return the root value of the heap while maintaining heap invariant

```
func ExtractMin($A$):
    min = $A[0]$
    $n = A$.Size()
    $A[0] = A[n-1]$
    $A$.Delete($n-1$)    ▷ Delete last element of A
    Heapify($A, 0$)
    return min
```

This algorithm takes $\log n$ time.

Insert

Algorithm 68 Insert – Insert key k into A.

```
func Insert($A$):
```

Exercise B.1 asks how this can be done in $\mathcal{O}(\log n)$ time.

DecreaseKey

Algorithm 69 DecreaseKey – Decrease the value of key k to v

```
func DecreaseKey($A$, $k$, $v$):
```

Exercise B.1 asks how this can be done in $\mathcal{O}(\log n)$ time. Note that for this operation, we assume that pointer to key k in the binary heap (i.e., its index in the array) is given as part of the input (one need not search for the key which can take linear time).

Exercise B.1. Show how an element can be inserted in a binary heap in $\mathcal{O}(\log n)$ time, where n is the number of elements in the heap. The heap property should be satisfied after the insertion. Show also how to perform DecreaseKey in $\mathcal{O}(\log n)$ time. Assume that pointer to key k in the binary heap (i.e., its index in the array) is given as part of the input of the DecreaseKey operation.

B.2 Stack Data Structure

Stack is a data structure that supports the following operations:

1. INSERT(S, k) or PUSH(S, k): Insert or push key k into stack S.

2. DELETE(S) or POP(S): Return and delete the *most recently* inserted key.

3. EMPTY(S): Return true if S is empty, otherwise return false.

A stack can be easily implemented using an array so that all the above operations take $\mathcal{O}(1)$ time. A stack is a *Last In First Out (LIFO)* data structure, i.e., the item that is inserted last is the first one to be output (popped).

A stack is useful in a variety of algorithmic applications — a notable application is that they are useful in iteratively implementing recursive algorithms. In fact, a compiler typically implements a recursive code by converting it into a iterative code — stack is the data structure that is used. In particular, a recursive algorithm can be coverted into an iterative algorithm and the stack data structure usually plays a useful role in this conversion. A good illustration of this *depth first search* which is naturally stated as a recursive algorithm. This can be converted into an iterative algorithm using a stack. In practice, iterative algorithms are faster, since there is overhead involved in implementing recursive algorithms (since the compiler has to convert them into iterative algorithms).

Exercise B.2. Show how to implement all the stack data structure operations in $\mathcal{O}(1)$ time.

B.3 Queue Data Structure

A queue is data structure that supports the following operations:

1. INSERT(Q, k) or ENQUEUE(Q, k): Insert or push key k into Queue Q.

2. DELETE(Q) or DEQUEUE(Q): Return and delete the *least recently* inserted key, i.e, the key that was inserted the earliest.

3. EMPTY(Q): Return true if Q is empty, otherwise return false.

A queue is a *First-in-First-Out (FIFO)* data structure, i.e., the item that is inserted first is the first one to be output (dequeued).

B.3.1 Implementing a Queue

A Queue can be implemented using a vector (or an array) so that all the above operations take $\mathcal{O}(1)$ time. However, one has to be careful about space used. Let us say we use a vector A to implement a Queue: simply keep the elements in the order of insertion and *keep track* of the indices of the first and last element. ENQUEUE will insert the element as the last element in the vector. DEQUEUE will remove and return the first element in the vector; update the index of the first element. The problem with the above implementation is that space is wasted at the beginning of the array when elements are dequeued. Exercise B.4 asks you a way to address the problem. Exercise B.5 asks you to give an alternate implementation of a Queue.

B.4 Union-Find Data Structure for Disjoint Sets

The union-find data structure is used to maintain a **dynamic** collection of disjoint sets. Each set has a unique representative (an arbitrary member of the set). The data structure supports the following three efficient operations:

- **Make-Set**(x) - Create a new set with one member x.
- **Union**(x, y) - Combine the two sets, represented by x and y into one set.
- **Find-Set**(x) - Find the representative of the set containing x.

We maintain each disjoint set as a *directed rooted tree*, rooted at some node belonging to the tree. See Figure B.5. Each node in the directed tree points to its parent (the root points to itself). The root of the tree is the *representative* of the set. We implement the data structure operations as follows:

- MAKE-SET: creates a rooted tree with just one node which points to itself (Algorithm 70).
- FIND-SET: Given a pointer to an element u, $find(u)$, simply follows parent pointers to the root, and returns the root which is the representative of the set that u belongs to (Algorithm 71).
- UNION: Given two representative elements u and v (i.e., these are the roots of the respective trees), *union*(u, v), makes the root of the tree with a fewer number of nodes point to the root of the tree with the larger number of nodes (if the two trees have the same number of nodes, then one of the roots points to the other). The new root will be the representative of the combined set. This strategy of making the root of the smaller tree point to the larger is called *union-by-rank* heuristic (Algorithm 72). Note that instead of maintaining the size of the tree, the union-by-rank heuristic uses a parameter called *rank* which is *equal to the height of the tree*. Hence, the root of the tree of smaller height is made a child of the root of the tree with larger (or equal) height.

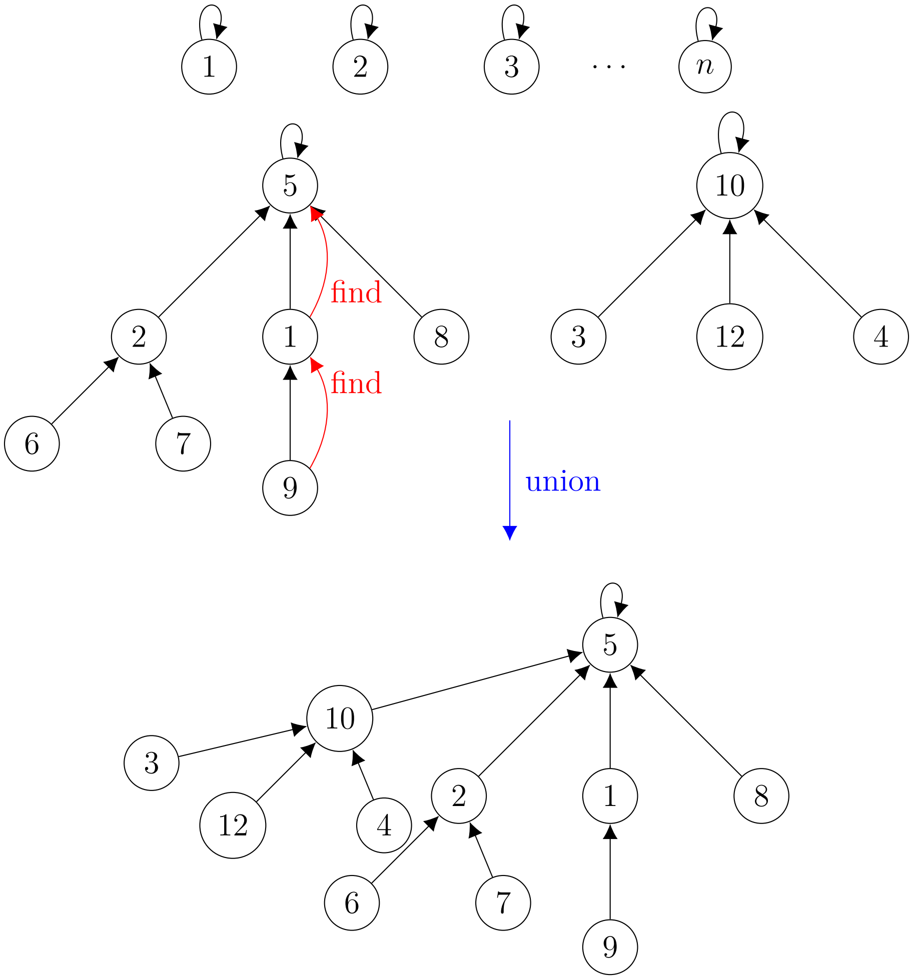


Figure B.5: Union-Find data structure as a collection of disjoint rooted trees. (Top) Make-Set creates n singleton trees, where n is the total number of elements in the set. (Middle) Two trees representing two sets after some union operations. Find operation finds the root of the tree that the node belongs to by parent pointer chasing to the root. (Bottom) Combing the two trees via union-by-rank heuristic attaches the tree with smaller height to a tree of larger height.

Algorithm 70 MakeSet

Input: An element x

Output: A singleton Union-Find with x as the root

```
func MakeSet($x$):
    $U = (\texttt{parent}, \texttt{rank})$
    ▷ Initialize parent and rank of each element
    $\texttt{parent}[x] = x$
    $\texttt{rank}[x] = 0$
    return $U$
```

Algorithm 71 Find

Input: An element x

Output: A representative of the set containing x

```
func Find($x$):
    if $x \neq \texttt{parent}[x]$:
        return Find($\texttt{parent}[x]$)
    else:
        return $x$
```

Algorithm 72 Union

Input: Elements x and y belonging to different sets maintained as disjoint directed rooted trees.

Output: A union of the two sets containing x and y

```
func Union($x$, $y$):
    $r(x) = $ Find($x$)
    $r(y) = $ Find($y$)
    if $r(x) \neq r(y)$:   ▷ if $x$ and $y$ belong to different sets
        if $\texttt{rank}[x] > \texttt{rank}[y]$:
        ▷ union-by-rank heuristic
        ▷ make the tree with smaller rank the child of the one with larger rank
            $\texttt{parent}[y] = x$
        else:
            $\texttt{parent}[x] = y$
        if $\texttt{rank}[x] == \texttt{rank}[y]$:
             $\texttt{rank}[y] \mathrel{+}= 1$     ▷ increment rank if both trees are of same rank
```

B.4.1 Analysis of Union and Find operations

We show that implementations of the union and find operations as given in Algorithms 71 and 72 respectively are efficient. In particular, any union or find operation on a set of a total of n elements takes $\mathcal{O}(\log n)$ time. To show this we need to only show that any find operation takes $\mathcal{O}(\log n)$ time. This is because, apart from the two find operations in the union algorithm, the rest of the algorithm (attaching the pointer of one root to the other) takes only constant time.

To show that any find operation takes only $\mathcal{O}(\log n)$ time, we show that the height (i.e., the rank) of every rooted tree does not exceed $\log n$. The main idea for this is as follows. Note that initially the height of all trees is 0, since each element is a singleton tree. The height of a tree increases only when it is combined with another tree — note that this happens during the union operation and this happens only when the two trees that are combined have the *same* rank (Union line 10). We show that whenever the height increases the size of the tree, i.e., the number of nodes in the tree doubles. Since there are only n nodes, and each tree starts with one node, the doubling can happen only $\log n$ times until we include all n nodes. Hence the height of any tree is bounded by $\log n$. The following lemma shows the formal proof by induction.

Lemma B.1

Let $\mathit{size}(x)$ be the number of nodes in the tree rooted at x. Then, for any tree with root x, $\mathit{size}(x) \geqslant 2^{\mathit{rank}[x]}$. Furthermore, the height of a tree is the same as its rank.

Proof. Proof is by induction on number of union operations, the only operation which changes the rank (height) or size of its tree.

- *Base case:* True for singleton trees, since ranks are initialized to 0 and size is 1.
- *Inductive step:* Assume that the lemma holds before performing union(x, y). Let rank denote the rank just before the union and let rank' denote the rank immediately following the union. Similarly, let size and size' be the size before and after the union, respectively.

  We first address the case, $\mathit{rank}[x] \neq \mathit{rank}[y]$. Without loss of generality, assume $\mathit{rank}[x] < \mathit{rank}[y]$. Node y would be the root of the resulting tree and:

$$\begin{aligned} \mathit{size}'(y) &= \mathit{size}(x) + \mathit{size}(y) \\ &\geqslant 2^{\mathit{rank}[x]} + 2^{\mathit{rank}[y]} \\ &\geqslant 2^{\mathit{rank}[y]} \\ &= 2^{\mathit{rank}'[y]} \end{aligned}$$

  The other case is that $\mathit{rank}[x] = \mathit{rank}[y]$. Node y would be the root of the new tree, and:

$$\begin{aligned} \mathit{size}'(y) &= \mathit{size}(x) + \mathit{size}(y) \\ &\geqslant 2^{\mathit{rank}[x]} + 2^{\mathit{rank}[y]} \\ &\geqslant 2^{\mathit{rank}[y]+1} \\ &= 2^{\mathit{rank}'[y]} \end{aligned}$$

□

Theorem B.2

The height of any tree in the collection of rooted trees representing a collection of disjoint sets of n elements is $\mathcal{O}(\log n)$. Thus any find or union operation takes $\mathcal{O}(\log n)$ time.

Proof. By Lemma B.1, since the number of nodes in a tree rooted at x, $size(x)$ is at least $2^{rank[x]} = 2^{height[x]}$, where $height[x]$ is the height of the tree rooted at x. Hence $height[x] \leqslant \log(size[x]) \leqslant \log n$, since the maximum size can be at most n, the total number of elements.

Since the find operations involve pointer chasing to the root, it is bounded by the height of the tree which is $\mathcal{O}(\log n)$. Union operations involve two find operations and other operations which take constant time, hence union takes also $\mathcal{O}(\log n)$ time. □

B.5 Exercises

Exercise B.3. Given a list of n numbers, the median is the number of rank $\lceil n/2 \rceil$, i.e., it is the $\lceil n/2 \rceil$th smallest element. Suppose the list is changing dynamically. That is at every step, a number is added or deleted from the list. The goal is to output the median in the current list as fast as possible. In other words, at any point in time, we would like to answer the query: "What is the median of the current list of numbers?". How fast can you answer this query?

Exercise B.4. Show how to efficiently implement a queue using a single array. Your array should not waste significant space. In particular, at any time, the total space used should be $\mathcal{O}(n)$ where n is the (current) number of elements in the queue. Also, the time needed for all operations should be $\mathcal{O}(1)$ on the *average*, i.e., the total time time for k operations (which can be any of the three operations — INSERT, DELETE, EMPTY) divided by k should be $\mathcal{O}(1)$.

Exercise B.5. Show how to implement a (FIFO) Queue using two stacks. Analyze the running time of your Queue operations as follows:

1. What is the worst case time for ENQUEUE, DEQUEUE, and EMPTY?

2. If there are a total of m ENQUEUE and DEQUEUE operations (the number of DEQUEUE operations is not more than the number of ENQUEUE operations), what is the total time for all the m operations. For full credit, your total time should be $\mathcal{O}(m)$.

Exercise B.6. A *strange* number is one whose only prime factors are in the set $\{2, 3, 5\}$. Give an efficient algorithm (give pseudocode) that uses a **binary heap** data structure to output the nth strange number. Explain why your algorithm is correct and analyze the run time of your algorithm. (Hint: Consider generating the strange numbers in increasing order, i.e., 2,3,4,5,6,8,9,10,12,15, etc. Show how to generate efficiently the next strange number using a heap.)

Exercise B.7. Consider the following geometric problem. You are given n points in the plane, $(x_1, y_1), \ldots, (x_n, y_n)$ and you want to determine whether there exist four points that are the vertices of a square. If there are no such four points, report that no square exists. Otherwise report four points belonging to a square. If there exist more than one square, report the four points corresponding to the square of maximum area. Your algorithm should solve this problem in $\mathcal{O}(n^2 \log n)$ time. Explain your algorithm and its implementation (explain what data structure operations will be used) and analyze the run time. You can assume that computing the distance between any two given points takes constant time. (Hint: Comparing all possible quadruplets of points will result in an $\Theta(n^4)$ time algorithm. To obtain the desired running time, you can only consider pairs of points.)

Exercise B.8. Show how you can sort n numbers using a binary heap. Your algorithm should use the heap operations. What is the worst-case running time of your algorithm? You should show that your time bound is *tight*, by giving an example where your algorithm will require that much time (asymptotically). In other words, if you say that the running time of your algorithm is $\mathcal{O}(T(n))$, then you should give an n-size input where the algorithm actually takes $\Theta(T(n))$ time.

APPENDIX C PROBABILITY

We recall basic definitions and concepts of probability theory. We also present various probabilistic inequalities and bounds that are used throughout the book. These prove very useful in the analysis of randomized algorithms. We refer to [7, 4] for more details on the concepts presented here.

C.1 Events, Probability, Probability Space

Consider an experiment with a finite (or countably infinite) number of outcomes. Each outcome is a simple event (or a sample point). The *sample space* is the set of all possible simple events. An *event* $\mathcal{E}$ is a union of simple events — a subset of the sample space.

Two events are said to be **mutually exclusive** if $A \cap B = 0$.

With each simple event s we associate a number $\mathbb{P}(s)$ which is called as the *probability* of s.

Definition C.1 ▶ Probability Space

A probability distribution $\mathbb{P}$ on a discrete sample space S is a function from events of S to $\mathbb{R}$ such that it satisfies the following three *axioms*:

1. $\mathbb{P}(A) \geqslant 0$ for any event A.
2. $\mathbb{P}(S) = 1$.
3. For any (finite or countably infinite) sequence of pairwise mutually exclusive events $A_1, A_2, \ldots$:
$$\mathbb{P}(\cup_i A_i) = \sum_i \mathbb{P}(A_i)$$

The pair $(S, \mathbb{P})$ is called a discrete **probability space**.

The probability of any event $\mathcal{E}$ can be computed as the sum of the probabilities of the simple events that it is composed of; this follows from the third axiom:

$$\mathbb{P}(\mathcal{E}) = \sum_{s \in \mathcal{E}} \mathbb{P}(s)$$

C.2 Principle of Inclusion-Exclusion

The *inclusion-exclusion* principle gives a formula for computing the probability of the union of a set of events in terms of the probabilities of the individual events and their intersections.

Theorem C.1 ▶ Inclusion-Exclusion Principle

Let $E_1, E_2, \ldots, E_n$ be arbitrary events. Then

$$\begin{aligned}\mathbb{P}(\cup_{i=1}^{n} E_i) &= \sum_{i=1}^{n} \mathbb{P}(E_i) - \sum_{1 \leqslant i<j \leqslant n} \mathbb{P}(E_i \cap E_j) + \sum_{1 \leqslant i<j<k \leqslant n} \mathbb{P}(E_i \cap E_j \cap E_k) \\ &\quad - \cdots + (-1)^n \, \mathbb{P}(E_1 \cap E_2 \cdots \cap E_b) \\ &= \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \leqslant i_1 < \cdots < i_k \leqslant n} \mathbb{P}(E_{i_1} \cap E_{i_2} \cap \cdots \cap E_{i_k}) \right)\end{aligned}$$

An important consequence of the inclusion-exclusion principle is a simple upper bound on the probability of the union of events known as *Boole's inequality or union bound.*

Theorem C.2 ▶ Boole's inequality (union bound)

For any arbitrary sequence of events $E_1, E_2, \ldots, E_n$:

$$\mathbb{P}(\cup_{i=1}^{n} E_i) \leqslant \sum_{i} \mathbb{P}(E_i)$$

The union bound is used often in upper bounding the occurrence of the union of a set of "bad" events. If this upper bound is small, then it implies that none of the bad events occur with high probability.

C.3 Conditional Probability

Definition C.2 ▶ Conditional Probability

Given two events A and B, the conditional probability of A given B is defined as follows:

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

assuming $\mathbb{P}(B) \neq 0$.

In the above definition, by conditioning on B we restrict the sample space to the set B. Thus the conditional probability can be considered as $\mathbb{P}(A \cap B)$ "normalized" by $\mathbb{P}(B)$.

Given two events E_1 and E_2, Bayes' rule relates the conditional probability of the first given the second, to the conditional probability of the second given the first. This is useful to infer one conditional probability from the other.

Theorem C.3 ▶ Bayes' rule

$$\mathbb{P}(E_1 \mid E_2) = \frac{\mathbb{P}(E_1 \cap E_2)}{\mathbb{P}(E_2)} = \frac{\mathbb{P}(E_2 \mid E_1)\,\mathbb{P}(E_1)}{\mathbb{P}(E_2)}$$

The following are some useful identities that involve computing the probability of the intersection of many events.

When you compile this theorem looks ugly. Not clear why?

Theorem C.4 ▶ Chain rule of Conditional Probabilities - $\mathbb{P}(A \cap B) = \mathbb{P}(A \mid B)\,\mathbb{P}(B)$.

- $\mathbb{P}(A \cap B \cap C) = \mathbb{P}(A \mid B \cap C)\,\mathbb{P}(B \cap C) = \mathbb{P}(A \mid B \cap C)\,\mathbb{P}(B \mid C)\,\mathbb{P}(C)$.
- The following is a generalization of the above identity. Let $A_1, \ldots, A_n$ be a sequence of events. Let $E_i = \bigcap_{j=1}^{i} A_i$. Then

$$\begin{aligned}\mathbb{P}(E_n) &= \mathbb{P}(A_n \mid E_{n-1})\,\mathbb{P}(E_{n-1})\\ &= \mathbb{P}(A_n \mid E_{n-1})\,\mathbb{P}(A_{n-1} \mid E_{n-2}) \ldots \mathbb{P}(A_2 \mid E_1)\,\mathbb{P}(A_1)\end{aligned}$$

A fundamental concept that follows from conditional probability is that of *independence*.

Definition C.3 ▶ Independence of events

Two events A and B are said to be *independent* if

$$\mathbb{P}(A \cap B) = \mathbb{P}(A) \times \mathbb{P}(B)$$

or (when $\mathbb{P}(B) > 0$)

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \mathbb{P}(A)$$

C.4 The Birthday Paradox

We study a problem called the "Birthday Paradox", that arises often in algorithm design and analysis. The problem also serves to illustrate basic concepts in probability theory.

Question: What is the probability that among m people no two have the same birthday? We make the following two assumptions: (1) All birthdays are equally likely and (2) Birthdays are independent events.

To compute the above probability, we first define the sample space of the experiment. The sample space is the set of all vectors $S = \{(b_1, \ldots, b_m) | b_i \in [1, \ldots, N]\}$, where b_i denotes the birthday of the ith person, and N is the total number of different birthdays ($N = 365$ in Earth). We need to compute $\mathbb{P}(E)$ where the event $E = \{(b_1, \ldots, b_m | b_i \neq b_j \text{ for all } i \neq j\}$, i.e., the event E is the set of all events where no two birthdays are the same.

How many different atomic events are counted in E? The number of possible m

different birthdays is $N.(N-1).(N-2)\dots(N-m+1)$. Hence,

$$\begin{aligned}\mathbb{P}(E) &= \frac{N\cdot(N-1)\cdot(N-2)\dots(N-m+1)}{N^m}\\ &= \prod_{i=0}^{m-1}(1-i/N)\\ &\leqslant \prod_{i=0}^{m-1} e^{-i/N}\\ &= e^{-\sum_{i=0}^{m-1} i/N}\\ &= e^{-m(m-1)/2N}\end{aligned}$$

For $m=\sqrt{2N}+1 \leqslant 28$, $\mathbb{P}(E) < 1/e < 1/2$.

The apparent "paradox" in this problem is that significantly less number of people, i.e., only about $\sqrt{N}$ people (which is much smaller than N), are needed to have a good chance (about 50%) to have two people to have a common birthday.

Alternate Analysis

Assume that we choose one birthday after the other independently and uniformly at random from $[1,2,\dots,N]$. Let the event E_i denote: "the ith choice is different from the first $i-1$ choices". Then, we compute:

$$\begin{aligned}\mathbb{P}(E) &= \mathbb{P}(\cap_{i=1}^m E_i)\\ &= \mathbb{P}(E_1)\,\mathbb{P}(E_2 \mid E_1)\,\mathbb{P}(E_3 \mid E_2 \cap E_1)\dots\mathbb{P}\Big(E_m \mid \cap_{i=1}^{m-1} E_i\Big)\\ &= 1(1-1/N)(1-2/N)\dots(1-(m-1)/N)\\ &= \prod_{i=1}^{m}(1-\frac{i-1}{N})\end{aligned}$$

This gives the same result as before.

This analysis uses a very useful principle called the **Principle of Deferred Decisions**. In this principle, the idea is to defer the fixing of choices of events to "when they are needed" and not a priori (i.e., all at once). Typically, this type of analysis will use conditional probabilities. In the above analysis, to compute $\mathbb{P}(E)$, we first computed $\mathbb{P}(E_1)$, then $\mathbb{P}(E_2 \mid E_1)$ and so on. At each stage, we only fixed the event that determined the respective conditional probabilities, i.e., first E_1 which fixes b_1, and then the event E_2 conditioned on E_1, which fixes $b_2 \neq b_1$, and so on.

C.5 Random Variable

Random variable is a very useful concept in probability; it allows one to quantify the parameters associated with the events in a probability space as we see fit to model.

Let $(\mathcal{S},\mathbb{P})$ be a discrete probability space. Let V be a set of values. A random variable X defined on $(\mathcal{S},\mathbb{P})$ is a *function*

$$X : \mathcal{S} \to V.$$

The random variable X allows one to "transfer" the probability function that is defined on the set of events of $\mathcal{S}$ to one that is defined on the set of values V, the range of X, in the following manner. Let $\mathcal{E}(r) = \{s \in \mathcal{S} \mid \mathrm{X}(s) = r\}$. Then we can define:

$$\mathbb{P}(X = r) = \mathbb{P}(\mathcal{E}(r)) = \sum_{s \in \mathcal{E}} \mathbb{P}(s)$$

Two random variables X and Y (defined on the same sample space) are called *independent* if for all x and y

$$\mathbb{P}(X = x \text{ and } Y = y) = \mathbb{P}(X = x)\,\mathbb{P}(Y = y)$$

C.6 Expectation of a random variable

Random variable is a function (thus "variable" is actually a misnomer!), and thus it can take any value in its range. The probability function $\mathbb{P}$ gives the probability that each such value is taken, i.e., it gives the *probability distribution* over the values in V. In many applications, we are less interested in the entire distribution of the probability over the values, but rather than a single (or few) parameter(s) that characterizes the distribution. A standard such parameter is the *expectation*, also called as the *mean* or the *average* of the random variable.

Definition C.4 ▶ Expectation of a random variable

The *expectation* of a discrete random variable X with outcomes $x_1, x_2, \ldots, x_n$ is

$$\mathbb{E}[X] = \sum_{i=1}^{n} x_i\,\mathbb{P}(X = x_i)$$

Thus, the expectation is a weighted sum over all possible values of the random variable, weighted by the corresponding probabilities of the values.

A very useful property that comes in handy in computing the expected value of a random variable is the linearity property of expectation stated as follows.

Theorem C.5 ▶ Linearity of Expectation

Given a sequence of random variables, $X_1, X_2, \ldots, X_n$, we have

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

The linearity property says that the expectation of the sum of random variables is equal to the sum of their individual expectations. *Note that this is true, regardless whether the random variables are independent or not; in fact, it is the latter case that makes this property so useful in many applications.* In many applications, this is commonly used to compute the expectation of a random variable (which is not directly easy to compute by applying Definition C.4) by decomposing it into a sum of random variables whose expectations are easier to compute. A typical example is decomposing the random variable into a set of *indicator* (also called *Bernoulli or 0-1)*) random variables, which take only two values 0 and 1. The following example illustrates this paradigm.

Problem C.1

Assume that N customers checked their jackets in a restaurant. The jackets get mixed and while leaving the restaurant, each customer gets a random jacket, i.e., probability of getting any particular jacket is $1/N$. The goal is to compute the expected number of customers who get their own jacket.

Solution: Let random variable X denote the number of customers who get their own jacket. The range of X is $\{0, \dots, n\}$. It is difficult to compute $\mathbb{E}[X] = \sum_{k=0}^{N} k \, \mathbb{P}(X = k)$. Instead, we define N indicator random variables X_i:

$$X_i = \begin{cases} 1 & \text{if customer } i \text{ received jacket } i \\ 0 & \text{otherwise} \end{cases}$$

That is, X_i "indicates" whether customer i receives their own jacket. It is much simpler to compute $\mathbb{E}[X_i]$ for any i, since it takes only two values 0 and 1.

$$\begin{aligned} \mathbb{E}[X_i] &= 1 \cdot \mathbb{P}(X_i = 1) + 0 \cdot \mathbb{P}(X_i = 0) \\ &= \mathbb{P}(X_i = 1) \\ &= \frac{1}{N} \end{aligned}$$

Now, note that $X = \sum_{i=1}^{N} X_i$. The X_is are not independent, but we can still apply linearity of expectation:

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{N} X_i\right] \\ &= \sum_{i=1}^{N} \mathbb{E}[X_i] \\ &= \sum_{i=1}^{N} \frac{1}{N} \\ &= 1 \end{aligned}$$

Problem C.2

Let $a_1, a_2, \dots, a_n$ be a sequence of n values. Each value a_i is independently and randomly chosen from a fixed distribution $\mathcal{D}$ (e.g., uniform distribution from the real interval $[0, 1]$). Let $m_i = \min\{a_1, a_2, \dots, a_i\}$, i.e., the minimum of the first i values. Let r.v. Y denote the number of times the minimum value is updated, i.e., the number of times $m_i \neq m_{i+1}$. Then $\mathbb{E}[Y] = \mathcal{O}(\log n)$.

Solution: First, without loss of generality we can assume that all the values are distinct, since we are upper bounding Y. Indeed, when a value repeats, there will not be any update.

Let the indicator random variable Y_i denote the event that m_i is updated. The value m_i will be updated if a_{i+1} is the minimum value among the first $i + 1$ values. The probability that the above event happens is $\frac{1}{i+1}$. This is because each of the first $i + 1$

values are chosen independently from the same distribution, and assuming that values are distinct, the probability that a particular value, (i.e., a_{i+1}) is the minimum is $\frac{1}{i+1}$. Hence $\mathbb{E}[Y_i] = \mathbb{P}(Y_i = 1) = \frac{1}{i+1}$.

We have $Y = \sum_{i=1}^{n-1} Y_i$. Thus, $\mathbb{E}[Y] = \sum_{i=1}^{n-1} \mathbb{E}[Y_i] = \sum_{i=1}^{n-1} \frac{1}{i+1} \leqslant H_n = \Theta(\log n)$.

Note that H_n is the harmonic mean, defined as $\sum_{i=1}^{n} 1/i$ and is $\Theta(\log n)$.

C.7 Useful Types of Random Variables

We define two types of random variables that arise often in probabilistic analysis of algorithms — the binomial and the geometric random variables.

C.7.1 Binomial Random variables

Let X be a 0-1 random variable such that

$$\mathbb{P}(X = 1) = p$$
$$\mathbb{P}(X = 0) = 1 - p$$

Then:

$$\mathbb{E}[X] = 1 \cdot p + 0 \cdot (1 - p) = p$$

Consider a sequence of n independent Bernoulli trials $X_1, \ldots, X_n$. Let $X = \sum_{i=1}^{n} X_i$.

Then we say that the random variable X has a **Binomial** distribution with parameters n and p denoted as $X \sim B(n, p)$. Equivalently, X denotes the number of successes obtained when doing n independent trials where each trial has a probability of success p. The probability of getting k successes is given by:

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

and the mean is:

$$\mathbb{E}[X] = np$$

C.7.2 The Geometric Distribution

Assume that an experiment has probability p for success $1 - p$ for failure. How many trials do we need until the first success? The number of trails needed is a random variable and is said to have the geometric distribution. Geometric random variable arises often in probabilistic analysis of algorithms, typically in the following fashion. A randomized step will be shown to succeed with some probability p. The quantity that will be interest will be number of such steps needed to get one (or more) success(es).

Thus a random variable X is said to be geometrically distributed if:

$$\mathbb{P}(X = i) = (1 - p)^{i-1} p$$

We say X has a geometric distribution with parameter p $X \sim G(p)$. Clearly, this is a probability distribution, as the probabilities sum to 1:

$$\sum_{i=1}^{\infty} \mathbb{P}(X = i) = \sum_{i=1}^{\infty} (1 - p)^{i-1} p = p \frac{1}{1 - (1 - p)} = 1$$

Assume that X get values in $\mathbb{N}$.

$$\mathbb{E}[X] = \sum_{i=0}^{\infty} i\,\mathbb{P}(X = i) = \sum_{i=1}^{\infty} \mathbb{P}(X \geqslant i)$$

Let $X \sim G(p)$. Then, the expectation of X is:

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} \mathbb{P}(X \geqslant i) = \sum_{i=1}^{\infty} (1-p)^{i-1} = \frac{1}{1-(1-p)} = \frac{1}{p}$$

The geometric distribution is said to be **memoryless**: For any $k > r$,

$$\begin{aligned}\mathbb{P}(X > k \mid X > r) = \frac{(1-p)^k}{(1-p)^r} &= (1-p)^{(k-r)} \\ &= \mathbb{P}(X > (k-r))\end{aligned}$$

C.8 Bounding Deviation of a Random Variable from its Expectation*

In many applications, we are interested in bounding the deviation of a random variable from its expectation. This is useful in bounding the probability of success (or failure) of an algorithm. The typical way to do this is to first compute the expectation, and then showing that with good probability the random variable does not deviate “too far” from its expectation, i.e., it is “concentrated” around its mean.

C.8.1 Markov Inequality

There are two basic deviation inequalities: Markov and Chebyshev. Markov’s inequality uses only the value of the expectation (also called “first moment”). Hence it can be quite weak.

Theorem C.6 ▶ Markov Inequality

For any non-negative random variable and for any $a > 0$

$$\mathbb{P}(X \geqslant a) \leqslant \frac{\mathbb{E}[X]}{a}$$

Proof. For any $a > 0$, let I be an indicator r.v. for the event $X \geqslant a$. Then $I \leqslant X/a$. Taking expectations on both sides, we get $\mathbb{E}[I] = \mathbb{P}(X \geqslant a) \leqslant \frac{\mathbb{E}[X]}{a}$. □

Chebyshev’s inequality gives a stronger bound, which assumes that the *variance* (also called the “second moment”) or *standard deviation* of the random variable is known. These are defined below.

Definition C.5

The **variance** of a random variable X is

$$\mathbb{V}[X] = \mathbb{E}\left[(X - \mathbb{E}[X])^2\right]$$

The **standard deviation** of a random variable X is

$$\sigma(X) = \sqrt{\mathbb{V}[X]}$$

Theorem C.7

Chebyshev's Inequality For **any** random variable

$$\mathbb{P}(|X - \mathbb{E}[X]| \geqslant a) \leqslant \frac{\mathbb{V}[X]}{a^2}$$

Proof.

$$\mathbb{P}(|X - \mathbb{E}[X]| \geqslant a) = \mathbb{P}\left((X - \mathbb{E}[X])^2 \geqslant a^2\right)$$

By Markov inequality,

$$\mathbb{P}\left((X - \mathbb{E}[X])^2 \geqslant a^2\right) \leqslant \frac{\mathbb{E}\left[(X - \mathbb{E}[X])^2\right]}{a^2} = \frac{\mathbb{V}[X]}{a^2}$$

□

A useful special case of the Markov and Chebyshev's inequalities is given by the following theorem. This is typically useful in showing whether a non-negative integer-valued random variable takes 0 value or otherwise.

Theorem C.8 ▶ First and Second Moment Inequalities

Let X be a non-negative integer-valued r.v. Then

$$\mathbb{P}(X \geqslant 1) \leqslant \mathbb{E}[X]$$

$$\mathbb{P}(X = 0) \leqslant \frac{\mathbb{V}[X]}{\mathbb{E}[X]^2} = \frac{\mathbb{E}[X^2]}{\mathbb{E}[X]^2} - 1$$

C.8.2 Chernoff Bound for Sum of Indicator R.V's

Chernoff bound is a standard tool to show deviation bounds for a random variable that can be expressed as a sum of independent indicator (or Bernoulli) random variables. Such a random variable typically comes in many algorithmic applications. We will state these bounds here and then prove them in Appendix C.9.1. Typically, in many algorithmic applications, the proofs are not needed to use these bounds.

One way to understand Chernoff bounds is from the point of view of the Binomial random variable $B(n, p)$. As seen from Appendix C.7.1, the mean of the binomial distribution is $\mu = np$. What is the probability that $B(n, p)$ takes values that are far away from its mean? The probability that $B(n,p)$ takes a value $k > \alpha\mu$, where $\alpha > 1$ is some parameter is:

$$\mathbb{P}(B(n,p) > \alpha\mu) = \sum_{k=\lceil \alpha\mu \rceil}^{n} \mathbb{P}(B(n,p) = k) = \sum_{k=\lceil \alpha\mu \rceil}^{n} \binom{n}{k} p^k (1-p)^{n-k}$$

This probability is called the "upper tail", i.e., the probability of deviating above the mean. Analogously, one can write an expression for calculating the "lower tail", i.e., the probability of deviating below the mean. The above formula does not directly give a useful way of quantifying the tail probabilities; as such it is quite a cumbersome sum to manipulate. On the other hand, Chernoff bound approximates the above sum in a way that is very useful to interpret and apply in applications. Moreover, the Binomial tail bounds are a special case of Chernoff, where the success probabilities of all trials are the same (equal to p); more generally, in Chernoff, each trial can have a different success probability — see Theorem C.9.

Theorem C.9 ▶ Chernoff bounds

Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that, for $1 \leqslant i \leqslant n$, $\mathbb{P}[X_i = 1] = p_i$, where $0 < p_i < 1$.
Define $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} p_i$.

1. Upper Tail Bounds:

   a. For any $\delta > 0$,

   $$\mathbb{P}(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}.$$

   b. For $0 < \delta < 1$,

   $$\mathbb{P}(X > (1+\delta)\mu) \leqslant e^{-\mu\delta^2/3}.$$

   c. For $R \geqslant 6\mu$,

   $$\mathbb{P}(X \geqslant R) \leqslant 2^{-R}.$$

2. Lower Tail Bound:

   For $0 < \delta < 1$,

   $$\mathbb{P}(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}.$$

C.8.3 Chernoff-Hoeffding Bound for Sum of Indicator R.V's

Sometimes the following variant of Chernoff bound is useful.

Theorem C.10 ▶ Chernoff-Hoeffding bounds

Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that, for $1 \leqslant i \leqslant n$, $\mathbb{P}[X_i = 1] = p$, where $0 < p < 1$.
Define $X = \frac{\sum_{i=1}^{n} X_i}{n}$ and let

$$\mathbb{E}[X] = \mathbb{E}\left[\frac{\sum_{i=1}^{n} X_i}{n}\right] = \frac{\sum_{i=1}^{n} \mathbb{E}[X_i]}{n} = \sum_{i=1}^{n} np/n = p.$$

Then we have the following tail bounds:
Upper tail:

$$\mathbb{P}(X > \mathbb{E}[X] + \epsilon) \leqslant e^{-2n\epsilon^2}.$$

Lower tail:

$$\mathbb{P}(X < \mathbb{E}[X] - \epsilon) \leqslant e^{-2n\epsilon^2}.$$

Both tails:

$$\mathbb{P}(|X - \mathbb{E}[X]| > \epsilon) \leqslant 2e^{-2n\epsilon^2}.$$

C.9 Moment Generating Function*

The moment generating function $M(t)$ of (discrete) random variable X is defined for all real values t by

$$M(t) = \mathbb{E}\left[e^{tX}\right] = \sum_{x} e^{tx}\, \mathbb{P}(X = x)$$

All moments of X can be obtained from $M(t)$ and then evaluating the result at $t = 0$, hence the name.

$$M'(t) = \frac{d}{dt}\mathbb{E}\left[e^{tX}\right] = \mathbb{E}\left[\frac{d}{dt}(e^{tX})\right] = \mathbb{E}\left[Xe^{tX}\right]$$

Thus, $M'(0) = \mathbb{E}[X]$ and the nth derivative of $M(t)$ is given by:

$$M^n(t) = \mathbb{E}\left[X^n e^{tX}\right] \; n \geqslant 1$$

and

$$M^n(0) = \mathbb{E}[X^n] \; n \geqslant 1$$

Example: Binomial Distribution $B(n, p)$

We calculate the Moment generating function of the Binomial random variable.

$$\begin{aligned} M(t) &= \mathbb{E}\left[e^{tX}\right] \\ &= \sum_{k=0}^{n} e^{tk} \binom{n}{k} p^k (1-p)^{n-k} \\ &= \sum_{k=0}^{n} \binom{n}{k} (pe^t)^k (1-p)^{n-k} \\ &= (pe^t + 1 - p)^n \end{aligned}$$

Example: Unit Normal Distribution $N(0,1)$

We calculate the Moment generating function of the unit normal distribution.

$$\begin{aligned} M(t) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{tx} e^{-x^2/2} dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-(x-t)^2/2+t^2/2} dx \\ &= e^{t^2/2} \end{aligned}$$

Important Properties of MGFs

1. The moment generating function of the sum of independent random variables equals the product of the individual moment generating functions.

2. MGF uniquely determines the distribution.

Chernoff Bounds via MGF

Let X be a random variable and M be its MGF. Then, for all $t > 0$, via Markov's inequality, we have:

$$\begin{aligned} \mathbb{P}(X \geqslant a) &= \mathbb{P}\Big(e^{tX} \geqslant e^{ta}\Big) \\ &\leqslant \frac{\mathbb{E}\Big[e^{tX}\Big]}{e^{ta}} \\ &= e^{-ta} M(t) \end{aligned}$$

Similarly, for all $t < 0$,

$$\mathbb{P}(X \leqslant a) \leqslant e^{ta} M(t)$$

The above bounds are called *Chernoff bounds.* We obtain the best bound by using the t that minimizes the right hand side.

Example

Let X be the standard normal r.v. $\mathbb{P}(X \geqslant a) \leqslant e^{-ta} e^{t^2/2}$ for all $t > 0$. The right hand side is minimized for $t = a$. Thus, for $a > 0$

$$\mathbb{P}(X \geqslant a) \leqslant e^{-a^2/2}$$

Similarly, for $a < 0$,

$$\mathbb{P}(X \leqslant a) \leqslant e^{-a^2/2}$$

C.9.1 Proof of Chernoff Bound for Sum of Indicator R.V's

Theorem C.11

Let $X_1, X_2, \dots, X_n$ be independent indicator random variables such that, for $1 \leqslant i \leqslant n$, $\mathbb{P}[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = \mathbb{E}[X] = \sum_{i=1}^n p_i$, and any $\delta > 0$,

$$\mathbb{P}(X > (1+\delta)\mu) < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$$

For $0 < \delta < 1$,

$$\mathbb{P}(X > (1+\delta)\mu) \leqslant e^{-\mu\delta^2/3}$$

For $R \geqslant 6\mu$,

$$\mathbb{P}(X \geqslant R) \leqslant 2^{-R}$$

For $0 < \delta < 1$,

$$\mathbb{P}(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}$$

Proof. **Upper tail:** For any positive real t,

$$\mathbb{P}(X > (1+\delta)\mu) = \mathbb{P}\Big(e^{tX} > e^{t(1+\delta)\mu}\Big)$$

By Markov's inequality,

$$\begin{aligned}
\mathbb{P}(X > (1+\delta)\mu) &< \frac{\mathbb{E}\Big[e^{tX}\Big]}{e^{t(1+\delta)\mu}} \\
&= \frac{\mathbb{E}\Big[e^{t\sum_{i=1}^n X_i}\Big]}{e^{t(1+\delta)\mu}} \\
&= \frac{\mathbb{E}\Big[\prod_{i=1}^n e^{tX_i}\Big]}{e^{t(1+\delta)\mu}} \\
&= \frac{\prod_{i=1}^n \mathbb{E}\Big[e^{tX_i}\Big]}{e^{t(1+\delta)\mu}} \\
&= \frac{\prod_{i=1}^n (p_i e^t + 1 - p_i)}{e^{t(1+\delta)\mu)}} \\
&= \frac{\prod_{i=1}^n (1 + p_i(e^t - 1))}{e^{t(1+\delta)\mu}} \\
&< \frac{\prod_{i=1}^n e^{p_i(e^t-1)}}{e^{t(1+\delta)\mu}} \\
&= \frac{e^{\sum_{i=1}^n p_i(e^t-1)}}{e^{t(1+\delta)\mu}} \\
&= \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}} \\
&\leqslant \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu
\end{aligned}$$

for $t = \ln(1+\delta)$ Using $\delta - (1+\delta)\ln(1+\delta) \leqslant -\delta^2/3$ for $0 < \delta < 1$ we get

$$\mathbb{P}(X > (1+\delta)\mu) \leqslant e^{-\mu\delta^2/3}$$

For $R \geqslant 6\mu$, $\delta \geqslant 5$.

$$\begin{aligned}\mathbb{P}(X \geqslant (1+\delta)\mu) &\leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}\\ &\leqslant \left(\frac{e}{6}\right)^{R}\\ &\leqslant 2^{-R}\end{aligned}$$

Lower tail:

$$\mathbb{P}(X < (1-\delta)\mu) = \mathbb{P}\Big(e^{-tX} > e^{-t(1-\delta)\mu}\Big)$$

By Markov's inequality,

$$\mathbb{P}(X < (1-\delta)\mu) < \frac{\mathbb{E}\left[e^{-tX}\right]}{e^{-t(1-\delta)\mu}}$$

Similar calculations yield

$$\frac{\mathbb{E}\left[e^{-tX}\right]}{e^{-t(1-\delta)\mu}} < \frac{e^{(e^{-t}-1)\mu}}{e^{-t(1-\delta)\mu}}$$

for $t = \ln\frac{1}{1-\delta}$, this becomes

$$\leqslant \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$$

Since $(1-\delta)^{(1-\delta)} > e^{-\delta+\delta^2/2}$, we have

$$\mathbb{P}(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}$$

□

C.9.2 Example application of Chernoff bound

Theorem C.12

Consider n coin flips and let X be the number of heads. Then

$$\mathbb{P}\left(\left|X - \frac{n}{2}\right| > \frac{1}{2}\sqrt{6n\log n}\right) \leqslant \frac{2}{n}$$

Proof. Let X_i denote an indicator random variable such that $X = \sum_{i=1}^{n} X_i$. Then

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n}\mathbb{E}[X_i] = \sum_{i=1}^{n}\frac{1}{2} = \frac{n}{2}$$

Set

$$\delta = \sqrt{\frac{6\log n}{n}}$$

Then

$$\begin{aligned}\mathbb{P}\Big(X - \frac{n}{2} > \frac{1}{2}\sqrt{6n\log n}\Big) &= \mathbb{P}\Big(x > \frac{n}{2} + \frac{1}{2}\sqrt{6n\log n}\Big)\\ &= \mathbb{P}\Big(x > (1+\delta)\frac{n}{2}\Big)\\ &\leqslant e^{-\frac{n}{2}\frac{\delta^2}{3}}\\ &\leqslant \frac{1}{n}\end{aligned}$$

Similarly,

$$\begin{aligned}\mathbb{P}\Big(X - \frac{n}{2} < -\frac{1}{2}\sqrt{6n\log n}\Big) &= \mathbb{P}\Big(x < \frac{n}{2} - \frac{1}{2}\sqrt{6n\log n}\Big)\\ &= \mathbb{P}\Big(x < (1-\delta)\frac{n}{2}\Big)\\ &\leqslant e^{-\frac{n}{2}\frac{\delta^2}{2}}\\ &\leqslant \frac{1}{n}\end{aligned}$$

We combine these results to achieve our final bound

$$\begin{aligned}\mathbb{P}\Big(\Big|X - \frac{n}{2}\Big| > \frac{1}{2}\sqrt{6n\log n}\Big) &= \mathbb{P}\Big(X - \frac{n}{2} > \frac{1}{2}\sqrt{6n\log n} \text{ and } X - \frac{n}{2} < -\frac{1}{2}\sqrt{6n\log n}\Big)\\ &\leqslant \mathbb{P}\Big(X - \frac{n}{2} > \frac{1}{2}\sqrt{6n\log n}\Big) + \mathbb{P}\Big(X - \frac{n}{2} < -\frac{1}{2}\sqrt{6n\log n}\Big)\\ &\leqslant \frac{1}{n} + \frac{1}{n}\\ &= \frac{2}{n}\end{aligned}$$

□

Comparing Markov, Chebyshev, and Chernoff

What is the probability of more than $\frac{3N}{4}$ heads in N coin flips?

1. Using Markov Inequality:
$$\mathbb{P}(X \geqslant 3N/4) \leqslant 2/3$$

2. Using Chebyshev's Inequality:
$$\mathbb{P}(X \geqslant 3N/4) \leqslant 4/N$$

3. Using the Chernoff bound:
$$\mathbb{P}(X \geqslant 3N/4) \leqslant e^{-\frac{N}{2}\frac{1}{4}\frac{1}{3}}$$

We see that Chernoff gives a stronger bound compared to Chebyshev which in turn is stronger compared to Markov. The reason is Markov's inequality uses only the first moment (i.e., the expectation) of the random variable in its bound, whereas, Chebyshev uses the first and second moment (equivalently, the variance) into account, while Chernoff uses all the moments (via the MGF).

C.10 Conditional Expectation

The conditional expectation of a random variable is defined as follows.

Definition C.6

Given two random variables X and Y, the random variable $\mathbb{E}[X \mid Y]$ is defined to be the random variable $f(Y)$ such that $f(y) = \mathbb{E}[X \mid Y = y]$ which is the conditional expectation of X given that $Y = y$:

$$\begin{aligned}\mathbb{E}[X \mid Y = y] &= \sum_x \mathbb{P}(X = x \mid Y = y) \\ &= \frac{\sum_x xp(x,y)}{\sum_x p(x,y)}\end{aligned}$$

where the summation is over all x in the range of X.

Note that $\mathbb{E}[X \mid Y]$ is itself a random variable that is a function of the random variable Y that takes the value $\mathbb{E}[X \mid Y = y]$.

Example C.1. Consider independent throws of an unbiased 6-sided die. For $1 \leqslant i \leqslant 6$, let X_i denote the number of times the value i appears in n throws of the die. Then

$$\begin{aligned}\mathbb{E}[X_1 \mid X_2] &= \frac{n - X_2}{5} \\ \mathbb{E}[X_1 \mid X_2, X_3] &= \frac{n - X_2 - X_3}{4}\end{aligned}$$

Sometimes it could be helpful to compute the expectation of a random variable by conditioning it with another random variable. The following identity proves useful:

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]] = \sum_y \mathbb{P}(Y = y)\, \mathbb{E}[X \mid Y = y]. \tag{C.1}$$

Thus we can state the following theorem (the second is a generalization of the first).

Theorem C.13

$$\begin{aligned}\mathbb{E}[X] &= \mathbb{E}[\mathbb{E}[X \mid Y]] \\ \mathbb{E}[\mathbb{E}[X \mid Y, Z] \mid Y] &= \mathbb{E}[X \mid Y]\end{aligned}$$

APPENDIX D

NUMBER-THEORETIC CONCEPTS

This appendix reviews basic concepts in number theory.

D.1 Modular Arithmetic

The algorithms in Chapters 9 and 10 heavily use modular arithmetic, i.e., arithmetic modulo some number n. The following properties of modular arithmetic are useful.

- *sum rule*: $(a+b) \bmod n = (a \bmod n + b \bmod n) \bmod n$.
- *product rule*: $(a \times b) \bmod n = (a \bmod n \times b \bmod n) \bmod n$.
- *exponentiation rule*: $(a^y \bmod n) = (a \bmod n)^y \bmod n$. The exponentiation rule is very useful in avoiding large integers.

D.2 Group Theory

Group theory is needed to understand and prove the correctness of primality testing algorithms, and once we have the basic concepts in place, it gives a powerful, yet easier way, to make our arguments.

A group $(G, \cdot)$ consists of a set G and a binary operation $\cdot$ (called the *group law* of G), where the following four properties hold:

- Closure: For all $a, b \in G$, $a \cdot b$ also belongs to S.
- Identity: there exists an *identity* element $e \in S$ such that $a \cdot e = e \cdot a = a$, for all $a \in S$.
- Associativity: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$, for all $a, b, c \in S$.
- Inverses: For each element $a \in S$, there exists an *inverse* $b \in S$, such that $a \cdot b = b \cdot a = e$.

For any $g \in G$, we write

$$g^k = \underbrace{g \cdot g \cdot \cdots \cdot g}_{k \text{ times}}$$

for any $k \geqslant 1$, and we define $g^0 = e$ and g^{-1} to be the unique inverse of g. Finally, define $g^{-k} = (g^{-1})^k$.

The cardinality of a group G, $|G|$, is called the *order* of G. A group with finite order is called a *finite group*. An element $g \in G$ is said to have order k if k is the smallest positive integer such that $g^k = e$.

Common examples of groups that we use everyday are the set of (all) integers with the binary operation of addition, i.e., the group $(\mathbb{Z}, +)$. It is easy to check that all four properties hold. In particular, the identity element is 0, and the inverse of an integer a is $-a$. Another common example is the group $(\mathbb{Q}^+, \times)$, where $\mathbb{Q}^+$ is the set of positive rational numbers and $\times$ is multiplication. Here, the identity element is 1, and the inverse element of a rational number a is $1/a$. (Note that, however, the set of rational numbers that includes 0 is not a group under multiplication. Why?)

Unless otherwise specified, we proceed informally and refer to groups only by the set G. When the choice of binary operation is not clear from context, it is stated explicitly. When it is not provided explicitly, we will frequently refer to it as multiplication, though it need not have any relation to multiplication of real numbers. Similarly, we will commonly refer to the identity element as 1, but it need not have any relation to the number 1.

Exercise D.1. Show that, for any group, the identity element is unique.

Exercise D.2. Show that the inverse of an element $g \in G$ is unique.

D.2.1 Additive group modulo n

Given a positive number n, this group consists of the set $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ with the binary operation being modular addition, i.e., *addition modulo* n. This is the same as the usual addition, but the sum is always taken modulo n. Hence, the set consists of elements which are all possible remainders of n, i.e., from 0 to $n-1$. The identity element is 0, as usual. The inverse of an element $a \in \mathbb{Z}_n$ is $-a \bmod n$ which is the same as $n - a \bmod n$.

D.2.2 Multiplicative group modulo n

The binary operation that we use here is modular multiplication. This is the same as the usual multiplication, but the product is always taken modulo n. Given a positive number n, this group consists of the set of all numbers in $\mathbb{Z}_n$ that are *relatively prime* to n (two numbers are relatively prime to each other if they have no common factor other than 1, i.e., their gcd is 1) — this set is denoted by $\mathbb{Z}_n^*$. In other words,

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$$

Why only numbers that are relatively prime to n belong to $\mathbb{Z}_n^*$? As we will see below, only such numbers will have a (unique) inverse under modular multiplication. It is easy to show that the set satisfies the closure and associativity properties. It is also clear that the identity element is 1 and this obviously belongs to $\mathbb{Z}_n^*$, since it is relatively prime to n. What about inverses? We would like to show that for every $a \in \mathbb{Z}_n^*$, a^{-1} exists: that is $aa^{-1} = a^{-1}a = 1$. This follows from the following lemma and its corollary (see e.g., the Discrete Math Notes [3] — Section 6 for a proof).

Lemma D.1

The equation $ax \equiv b \pmod{n}$ has a solution if and only if $\gcd(a, n)$ divides b.

By putting $b = 1$ in the above lemma, we have the following corollary.

Corollary D.2

The equation $ax \equiv 1 \pmod{n}$ has a unique solution if $\gcd(a, n) = 1$.

Hence there is a unique inverse for every $a \in \mathbb{Z}_n^*$, since $\gcd(a, n) = 1$. Thus $\mathbb{Z}_n^*$ is a group under multiplication modulo n. In particular, if n is prime, then $gcd(a, n) = 1$ for all $1 \leqslant a \leqslant n - 1$. Hence $\mathbb{Z}_n^* = \{1, 2, \ldots, n - 1\}$. In fact, in many applications, a prime n is the common case. In this case, $|\mathbb{Z}_n^*| = n - 1$.

Calculating the solution to the modular equation $ax \equiv b \pmod{n}$, and in particular, calculating the inverse of $a \bmod n$ can be done by using the Euclidean algorithm (using Exercise 4.7).

Exercise D.3. Show how to find the inverse of $a \bmod n$ (assuming that $\gcd(a, n) = 1$) using Euclid's algorithm (see Exercise 4.7).

D.2.3 Subgroups and generator of a group

Given a group G and set $G \subseteq G$, if G is also a group, then G' is a *subgroup* of G (written $G' \leqslant G$). If G' is a proper subset of G, then the subgroup is called a *proper subgroup* and we write $G' < G$. For example, given the group of integers $\mathbb{Z}$ under addition, the group of *even* integers forms a subgroup of. It is easy to show the following property of a subgroup:

Lemma D.3 ▶ Closure property of a subgroup

If G is a group and G' is a non-empty subset of G that is closed under the group operation, then G' is a subgroup of G'.

There is an important property of the size of a subgroup given by *Lagrange's Theorem* which we state without proof.

Theorem D.4 ▶ Lagrange's Theorem

If G is a finite group of order n and $G' < G$ is a subgroup of order m, then m divides n. In particular, if G' is a proper subgroup, then the order of $|G'| \leqslant n/2$.

We next introduce a very useful way to create subgroups of a given group, that uses the closure property of a subgroup. Let G be a group. Consider the set of all elements "generated" by g, i.e., repeatedly multiplying g with itself. We denote this set by $\langle g \rangle$, i.e.,

$$\langle g \rangle = \{g^k \mid k \geqslant 1\}$$

By the closure property, the group $\langle g \rangle$ is a subgroup (why?). For example, in the group $\mathbb{Z}_n$ under addition, $\langle a \rangle = \{ak \bmod n \mid k \geqslant 1\}$ and in the group $\mathbb{Z}_n^*$ under multiplication, $\langle a \rangle = \left\{a^k \bmod n \mid k \geqslant 1\right\}$. We say $\langle g \rangle$ is the *cyclic subgroup generated by* g, and g is called the generator of this subgroup. Since G is finite, it is clear that the size of the subgroup, i.e., $|\langle a\rangle|$ is finite and will divide $|G|$, by Lagrange's theorem. Furthermore, we can show the following lemma.

Lemma D.5

Let g be an element of order k. Then, the elements $g, g^2, \ldots, g^k = 1$ are all distinct and $k = |\langle g\rangle|$. In particular, the sequence $g, g^2, g^3, \ldots,$ is periodic with period k. Furthermore, $g^i = g^j$ if and only if $i \equiv j \pmod{k}$.

Proof. Suppose $g^i = g^j$ for $i < j < k$. Then $g^i g^{j-i} = g^j = g^i$. This means that $g^{j-i} = 1$ and $j - i < k$ which contradicts the minimality of k. Hence, $g, g^2, \ldots, g^k = e$ are all distinct. Furthermore, $g^{k+i} = g^k g^i = 1g^i = g^i$, hence there are only k distinct elements. This also implies that they are periodic with period k and $g^i = g^j$ if and only if $i \equiv j \pmod{k}$. □

We next state an important result that immediately proves Fermat's Little Theorem (stated next) that we use in primality testing and in the proof of the RSA cryptosystem.

Theorem D.6

Let G be a finite group of order n. Then for all $g \in g$,

$$g^n = 1$$

Proof. Consider any element $g \in G$ and suppose g has order k. By Lagrange's Theorem, $k \mid n$, hence $n = k\ell$ for some ℓ. Then $g^n = g^{k\ell} = \left(g^k\right)^\ell = 1^\ell = 1$. □

D.3 Fermat's Little Theorem

Fermat's Little Theorem (FLT) is stated as follows.

Theorem D.7 ▶ Fermat's Little Theorem (FLT)

If p is a prime, then for every $1 \leqslant a < p$,

$$a^{p-1} \equiv 1 \pmod{p}$$

Proof. Apply Theorem D.6 to the group $\mathbb{Z}_p^*$. Since p is prime, $\left|\mathbb{Z}_p^*\right| = p - 1$. Hence for $a \in \mathbb{Z}_p^*$, $a^{p-1} \equiv 1 \pmod{p}$. □

APPENDIX E

GRAPH-THEORETIC CONCEPTS

This appendix reviews basic concepts in graph theory.

E.1 Definition

We formally define a graph as follows.

Definition E.1

An *undirected* graph G is a pair (V, E) where

- V is the set of vertices.
- $E \subseteq V^2$ is the set of edges (*unordered pairs*) $E = \{(u, v) \mid u, v \in V\}$.

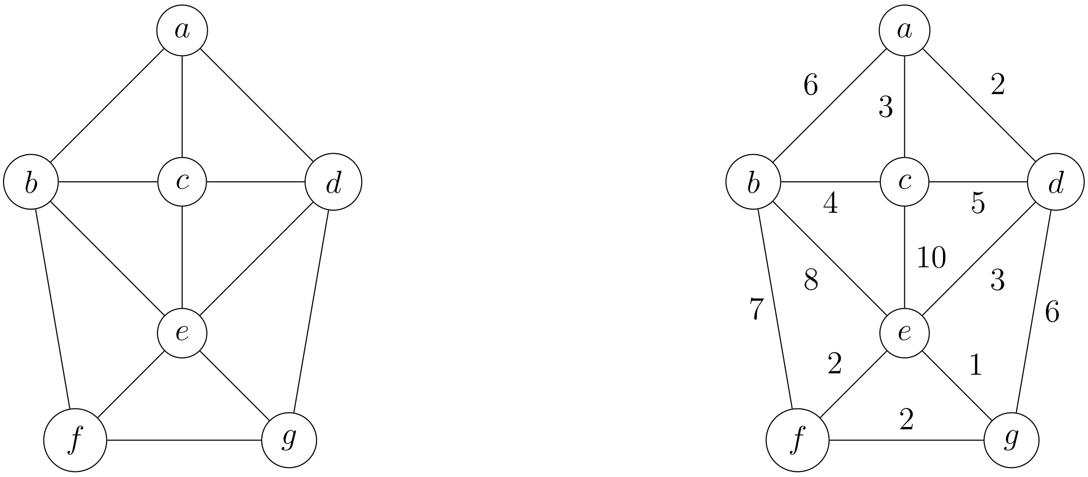
In a *directed* graph the edges have directions, i.e., *ordered pairs* — (u, v) means that u points to v and (v, u) means that v points to u. A *weighted* graph includes a weight function $w : E \to Q$ attaching a (rational) number (weight) to each edge.

Running time of a graph algorithm on a graph $G = (V, E)$ is usually in terms of $|V| = n$ and $|E| = m$. (We will use this notation throughout.) Sometimes, the running time can be a function of weights on the edges.

Figure E.1 shows an example of an undirected graph and an undirected weighted graph. Note that there can be at most $\binom{n}{2}$ edges in an undirected graph with n nodes.

E.2 Path, Cycle, and Tree

These are fundamental structures in graphs and are defined as follows.



(a) Graph without weights

(b) Graph with weights

Figure E.1: A connected undirected graph and the same graph with weights

Definition E.2

A *path* in a graph $G = (V, E)$ is a sequence of vertices v_1, v_2, $\ldots$, v_k such that for $1 \leqslant i \leqslant k-1$, $(v_i, v_{i+1}) \in E$.
A *cycle* in a graph $G = (V, E)$ is a path v_1, v_2, $\ldots$, v_k such that $(v_k, v_1) \in E$.
A *tree* is a graph with *no cycles*. A disjoint (i.e., the vertices are disjoint) collection of trees is a *forest*.

E.3 Distance

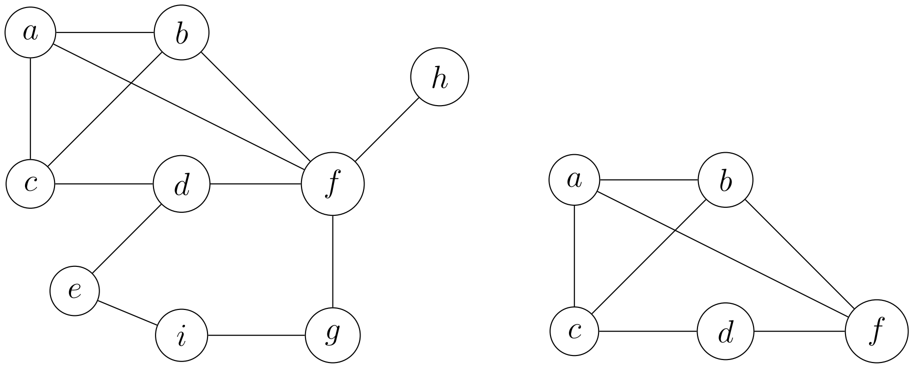
Given a graph $G = (V, E)$, and two vertices u and v belonging to V, and a path P between u and v, the *length* of the path is the *number of edges* between u and v in P. The *distance* between u and v is the length of the *shortest path* between u and v. The shortest path between u and v will have *minimum length* among all other paths between the two vertices.

E.4 Subgraph and Induced Subgraph

Definition E.3

A graph $H = (V', E')$ is a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.
If $V' = V$, then H is a *spanning* subgraph.
Given a subset $U \subseteq V$ of vertices, the subgraph *induced* by U in G is the subgraph consisting of all vertices in U and all the edges between them in G.

Figure E.2 shows an example of a graph and an induced subgraph.



(a) Graph (b) Induced Subgraph

Figure E.2: A graph and an induced subgraph of the graph.

E.5 Degree of a vertex

Given an undirected graph $G(V, E)$, the degree of a vertex is the number of its *neighbors*, i.e., the number of vertices that it shares an edge with. The degree of v is denoted by $deg(v)$.

Lemma E.1

In an undirected graph $G = (V, E)$: $\sum_{v \in V} deg(v) = 2|E| = 2m$.

Proof. Consider an edge $(u, v) \in E$. This contributes one to $deg(u)$ and one to $deg(v)$. □

E.6 Connectivity

Connectivity is a basic concept in graph theory.

Definition E.4

An undirected graph is *connected* if there is a path between *every pair* of vertices. A directed graph is *connected* if there is a directed path from u to v *or* a directed path from v to u, for every pair of vertices u, v.
A directed graph is *strongly connected* if there is a directed path from u to v *and* a directed path from v to u, for every pair of vertices u, v.
A directed graph is *weakly connected* if the undirected graph obtained by ignoring the directions of the edges is connected.

A *connected component* of an undirected graph G is a *maximal* induced subgraph of G that is connected. It is maximal in the sense that no more vertices can be added to the subgraph, while still keeping the subgraph connected. Figure E.3 shows the example of a disconnected graph.

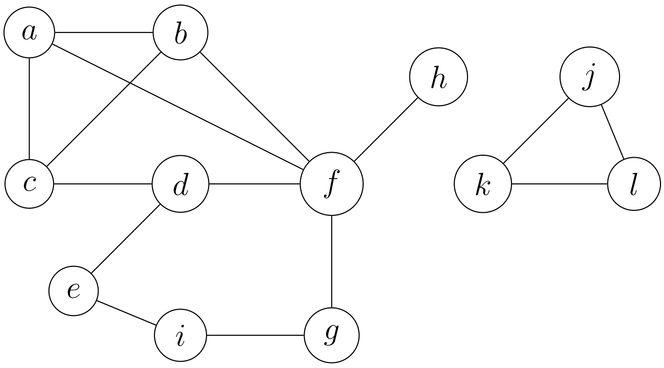


Figure E.3: A disconnected graph.

E.7 Spanning Tree (ST)

Definition E.5

Given a *connected* graph $G = (V, E)$ a **spanning tree** T in G is a spanning subgraph of G which is a tree.

If the graph is not connected, then we can define a spanning tree on each *connected component*. A (disjoint) collection of spanning trees is called a spanning *forest*.

Maximality of a Spanning Tree

Note that a spanning tree T is "maximal" in the sense that you cannot add additional edges — any edge added to T will close a cycle.

The reason of the above is as follows. Assume that T is a spanning tree in G, and we add add an edge (v, u) to T. There is a path p from u to v using only edges of T (since it's a spanning tree). Thus, the path p and the edge (v, u) must include a cycle.

E.7.1 The Size of a Spanning Tree

Theorem E.2

A spanning tree of a connected graph $G = (V, E)$ has $|V| - 1 = n - 1$ edges.

Proof. The proof is by induction on $|V|$. Base case ($|V| = 1$) is trivial. Assume that the lemma is true for all graphs of size $< |V|$. Consider a connected graph G of size $|V|$ and a spanning tree T of G. Remove an edge (u, v) of T. This breaks T into two components — T_1 and T_2. T_1 (T_2) is a spanning tree of the subgraph induced by the vertices of T_1 (T_2). The number of edges in T is $|T_1| - 1 + |T_2| - 1 + 1 = |T| - 1 = |V| - 1$. □

APPENDIX F

MATRICES

This appendix reviews basic concepts in theory of matrices.

F.1 Basic Facts

We recall some basic facts about matrices. If the entries of the matrices are numbers then the matrix operations $\{+, *\}$ define a **Ring** as seen below.

1. There is an *identity* element I, such that for any matrix A

$$AI = IA = A$$

2. There is a *zero* element O, s.t. for any matrix A

$$AO = OA = O$$

3. Every item has an additive inverse, i.e., every matrix A has an inverse $-A$. Hence subtraction is simply adding one matrix with the additive inverse of the other.

4. Matrix multiplication is **associative**:

$$A(BC) = (AB)C$$

5. Matrix multiplication is **distributive** over addition:

$$A(B + C) = AB + AC$$

$$(B + C)D = BD + CD$$

However, one should note that matrix multiplication is NOT commutative:

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

APPENDIX G

COMMONLY USED MATH FORMULAS IN ALGORITHM ANALYSIS

We collect commonly used math formulas in algorithm analysis. For a elaborate list see the theoretical computer cheat sheet by Steve Seiden.

G.1 Geometric series

Summing a geometric series occurs very often in algorithmic analysis. The formula for the sum of a *geometric* series:

$$\sum_{i=0}^{n-1} x^i = 1 + x + \cdots + x^{n-1} = \frac{x^n - 1}{x - 1}$$

where $x \neq 1$ is called the *common ratio.*

Thus:

$$\sum_{i=0}^{n} x^i = \begin{cases} \Theta(x^n) & \text{if } x > 1 \text{ (increasing series)} \\ \Theta(1) & \text{if } 0 < x < 1 \text{ (decreasing series)} \end{cases}$$

If the series is infinite and $|x| < 1$, then

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

G.2 Combinatorial Inequalities

1. $$\left(\frac{n}{k}\right)^k \leqslant \binom{n}{k} \leqslant \left(\frac{ne}{k}\right)^k$$

2. For all $x \in \mathbb{R}$, $1 + x \leqslant e^x$.

3. For all $0 \leqslant x < 1$, $1 - x \leqslant e^{-x} \leqslant 1 - x/2$.

4. For all $x, n \in \mathbb{R}$, such that $n \geqslant 1$ and $|x| \leqslant n$,

$$e^x\left(1 - \frac{x^2}{n}\right) \leqslant \left(1 + \frac{x}{n}\right)^n \leqslant e^x$$

   The above holds even for negative values of x.

5. For all $x, n \in \mathbb{R}^+$,

$$\left(1 + \frac{x}{n}\right)^n \leqslant e^x \leqslant \left(1 + \frac{x}{n}\right)^{n+x/2}$$

BIBLIOGRAPHY

[1] S. A. Cook. The complexity of theorem-proving procedures. In M. A. Harrison, R. B. Banerji, and J. D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[2] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2009.

[3] V. Koltun. Discrete structures. `https://web.stanford.edu/class/cs103x/cs103x-notes.pdf`, 2008.

[4] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[5] K. H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill Higher Education, 5th edition, 2002.

[6] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[7] E. Upfal and M. Mitzenmacher. *Probability and Computing: Randomized algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.