Lecture notes for Math182: Algorithms
Draft: Last revised July 28, 2020

Allen Gehret

Author address:

Department of Mathematics, University of California, Los Angeles, Los Angeles, CA 90095
E-mail address: allen@math.ucla.edu

Contents

Abstract

The objectives of this class is are as follows:

(1) Survey various well-known algorithms which solve a variety of common problems which arise in computer science. This includes reviewing the mathematical background involved in the algorithms and when possible characterizing the algorithms into one of several algorithm paradigms (greedy, divide-and-conquer, dynamic,...).
(2) Use mathematics to analyze and determine the efficiency of an algorithm (i.e., does the running time scale linearly, scale quadratically, scale exponentially, etc. with the size of the input, etc.).
(3) Use mathematics to prove the correctness of an algorithm (i.e., prove that it correctly does what it is supposed to do).

Portions of these notes are based on [**6**], [**1**], and [**5**]. Any and all questions, comments, typos, suggestions concerning these notes are enthusiastically welcome and greatly appreciated.

Last revised July 28, 2020.
2010 *Mathematics Subject Classification.* Primary .
The first author is supported by the National Science Foundation under Award No. 1703709.

List of Figures

Introduction

What is an algorithm?

This is a tough question to provide a definitive answer to, but since it is the subject of this course, we will attempt to say a few words about it. Generally speaking, an *algorithm* is an unambiguous sequence of instructions which accomplishes something. As human beings, we are constantly following algorithms in our everyday life. For instance, here is an algorithm for eating a banana:

(Step 1) Peel banana.
(Step 2) Eat banana.
(Step 3) Dispose of banana peel.

Many other routine tasks can also be construed as algorithms: tying your shoes, driving a car, scheduling a Zoom meeting, etc.

Of course, since this is a mathematics class, we will narrow our focus to algorithms which accomplish objectives of a more mathematical nature[1]. For instance:

(1) Given a very large list of numbers, how do you sort that list so the numbers are in increasing order?
(2) Given two very large integers a and b, how do you compute the greatest common divisor of a and b?
(3) Given a very large weighted graph, how do you find the shortest path between two nodes?

Furthermore, as human beings with busy lives, we have no interest in actually carrying out these tasks ourselves by hand. Instead, we will be interested in having a computer do these things for us. This brings us to one of the main themes of this class:

How can we leverage the decision-making facilities of a computer to efficiently accomplish tasks for us of a mathematical nature?

In this context, an *algorithm* might as well be synonymous with *computer program*, and indeed, this is essentially what we will be studying. With that said, this is not a *programming class* and our goal will not be to develop competence with any one particular programming language. Instead, we will be more concerned with the *logical essence* of various computer programs and we will study to what extent they efficiently solve the problem at hand.

Before we proceed any further, we will expand our answer to the original question: what is an algorithm? An algorithm (specifically, an algorithm for a computer) is typically characterized by the following five features:

[1] Here we take major liberties with what types of problems we consider to be of a *mathematical nature*.

(1) *Finiteness.* An algorithm must terminate after a finite number of steps. After all, what good is an algorithm if it runs forever and never accomplishes the task it is meant to do? All of the algorithms we will study in this class have this feature. One caveat: occasionally you may encounter algorithms which run indefinitely and continually interact with their environment (for instance, various operating system or server algorithms). We will ignore such algorithms in this class.
(2) *Definiteness.* Each step in the algorithm must be precisely and unambiguously defined. This means that any two people reading the step will carry out the instruction in exactly the same way. Generally speaking, instructions for the computer are written in a *programming language* (e.g., Python, C++, Java) which has the effect of providing unambiguous instructions. For us, we will often write our algorithms in *pseudocode* (see Chapter A) or even sometimes in *plain English*.
(3) *Input.* An algorithm accepts zero or more inputs, either at the beginning of the algorithm, or while the algorithm is running. The inputs are either provided by the user (a human being), or some other algorithm. This is analogous to saying that a *function* (in the mathematical sense) can take as input any element from its specified *domain*.
(4) *Output.* An algorithm has one or more outputs. An output can be a number, an answer to a question, or an indication that the algorithm has accomplished some task. This is analogous to the elements in the *range* of a (mathematical) function.
(5) *Effectiveness.* The instructions of an algorithm should be sufficiently basic and concrete enough that they can, in principle and with enough time, be carried out with paper and pencil by any well-trained clerical assistant who otherwise has no insight into what task they are ultimately performing.

Finally, we conclude with a list of what we will *not* do or care about in this course:

(1) We will not be concerned with issues of *software engineering*. In particular, we will disregard issues of memory management, error/exception handling, garbage collection, testing, debugging, etc.
(2) We will not be concerned with the idiosyncrasies of any one particular programming language, or of what hardware we are working with. In fact, the issues we will deal with are by-and-large both *language independent* and *hardware independent*.
(3) We will not be concerned with the practical limitations of computers as they exist in the year 2020. Indeed, computers are much faster and hold more memory now than they did 50 years ago, and in 50 years from now we expect them to be faster and better still. Nevertheless, we consider the ideas we will be studying to be *timeless*, i.e., they are equally valid and useful regardless of the particular era of computing we are living in. As funny as it might sound, we will not be bothered if we find that an algorithm may take 10^{100} years to run, or if it requires more bits of memory than there are atoms in the entire universe (although our sympathies will always be with faster algorithms which take up less space).
(4) We will not be concerned with *numerical* problems. I.e., using the computational power of a computer to approximate the roots of a polynomial, the value of a definite integral, or the solution to a differential equation,

etc. This is a very important subject, but not one we will pursue in this class[2]. Instead we will be focused more on the logical abilities of a computer to solve "nonnumerical" problems (i.e., sorting a list of numbers, analyzing a graph, etc.). In fact, we will probably at no point in our algorithms use numbers other than integers, and we will have no need to use functions such as $\sin, \cos, \tan$, etc. We will use functions such as e^x and $\log x$ in our *analysis* of algorithms, but that is a different story.

(5) We will primarily be interested in the *worst-case* running times of algorithms. The *average-case* running times of algorithms are also important in the analysis of algorithms, however this requires knowledge of probability theory which we are not assuming as a prerequisite. However, there is no major harm in ignoring average running and focusing on the worst-case running times since in practice the worst-case will happen quite frequently. Furthermore, since we will not be doing any probability, we will focus our attention to *deterministic* algorithms (as opposed to *randomized* algorithms).

(6) We will only deal with with single-processor *sequential* algorithms, i.e., algorithms which execute in a sequential manner one step at a time with a single flow of control. We will not be interested in *parallel* algorithms (algorithms where multiple tasks can be done concurrently across different processors) or *distributed* algorithms (algorithms run concurrently on many computers communicating with each other distributed in a complex graph-like network).

Prerequisites

The formal prerequisites for this class are Math 61 and one of Math 3C or 32A.

Math 61 is *Introduction to Discrete Structures*. While it is assumed you are generally familiar with the topics in Math 61, we will recall anything of particular relevance and importance. You can refer to [**3**] to refresh and review topics from that course. From Math 61 we will need: proofs, induction, recursion, summations, sequences, functions, relations, graphs, counting, and perhaps a few other things.

Math 3C is *Ordinary Differential Equations with Linear Algebra for Life Sciences Students* and Math 32A is *Calculus of Several Variables*. We will not have any need for differential equations or calculus of several variables in this course. However, these prerequisites ensure that you have a sufficient command of the basics of calculus and pre-calculus to the extent we will use such things. From calculus we will primarily need: limits of functions, properties of exponentials and logarithms, and the occasional derivative, integral, and infinite summation.

Conventions and notation

In this section we establish various mathematical and expository conventions. For pseudocode conventions, see Chapter A.

In this class the natural numbers is the set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ of nonnegative integers. In particular, we will consider 0 to be a natural number.

[2]This is the subject studied in Math151A/B.

Unless stated otherwise, the following convention will be in force throughout the entire course:

Global Convention 0.0.1. Throughout, m and n range over $\mathbb{N} = \{0, 1, 2, \dots\}$.

In a mathematical setting, when we write "$X := Y$", we mean that the object X does not have any meaning or definition yet, and we are defining X to be the same thing as Y. When we write "$X = Y$" we typically mean that the objects X and Y both already are defined and are the same. In other words, when writing "$X := Y$" we are performing an action (giving meaning to X) and when we write "$X = Y$" we are making an assertion of sameness.

In making definitions, we will often use the word "if" in the form "We say that ... if ..." or "If ..., then we say that ...". When the word "if" is used in this way in *definitions*, it has the meaning of "if and only if" (but only in definitions!). For example:

Definition 0.0.2. Given integer $d, n \in \mathbb{Z}$, we say that d **divides** n if there exists an integer $k \in \mathbb{Z}$ such that $n = dk$.

This convention is followed in accordance with mathematical tradition. Also, we shall often write "iff" or "$\Leftrightarrow$" to abbreviate "if and only if."

Acknowledgements

I am grateful to Julian Ziegler Hunts for his valuable feedback on these notes and his help with this course in general.

CHAPTER 1

Discrete mathematics and basic algorithms

In this chapter we review various ideas from discrete mathematics which will be relevant to our implementation and study of algorithms. We also take this chapter as an opportunity to ease ourselves into a more rigorous understanding and analysis of algorithms.

1.1. Induction

Our story starts with *mathematical induction*. Of course, you should already be familiar with induction from Math 61, however we are choosing to review it for several reasons:

(1) Induction is one of the main proof methods used in discrete mathematics.
(2) Induction is very *algorithmic* by nature.
(3) In fact, our understanding of a proof by induction often mirrors our understanding of how certain algorithms work. Indeed, induction will usually be our go-to method for proving the correctness of an algorithm.

Before we get to induction, we need to state a more primitive and more important property of the natural numbers which we will take for granted:

Well-Ordering Principle 1.1.1. *Suppose $S \subseteq \mathbb{N}$ is such that $S \neq \emptyset$. Then S has a least element, i.e., there is some $a \in S$ such that for all $b \in S$, $a \leq b$.*

We will not give a proof of 1.1.1. In fact, usually it is something that can't be proved as it is typically built in to the *definition*[1] of the natural numbers. Of course, given our intuition for the natural numbers, there should be no issue with accepting 1.1.1 as true.

One immediate practical consequence of the Well-Ordering Principle is the so-called *Division Algorithm*. It is not an "algorithm" in the same sense that we will later use this word, although it is typically the first result in the mathematical curriculum with the moniker *algorithm*. It also serves as the basis for many other facts in elementary number theory:

Division Algorithm 1.1.2. *Given integers $a, b \in \mathbb{Z}$, with $b > 0$, there exist unique integers $q, r \in \mathbb{Z}$ satisfying*

(1) $a = bq + r$
(2) $0 \leq r < b$.

The integer q is called the **quotient** and the integer r is called the **remainder** in the division of a by b.

[1] See [**7**] for a careful construction of the natural numbers.

PROOF SKETCH. Consider the following set of natural numbers:

$$S \;:=\; \{a - bq : q \in \mathbb{Z} \text{ and } a - bq \geq 0\}$$

One can show that $S \neq \emptyset$ and that $r := \min S$ (which exists by 1.1.1) satisfies $0 \leq r < b$. Furthermore, the $q \in \mathbb{Z}$ for which $a - bq = r$ has the property $a = bq + r$. For full technical details, see [**3**, 1.4.2]. □

Principle of Induction 1.1.3. *Suppose $P(n)$ is a property that a natural number n may or may not have. Suppose that*

(1) $P(0)$ holds (this is called the "base case for the induction"), and
(2) for every $n \in \mathbb{N}$, if $P(0), \ldots, P(n)$ holds, then $P(n+1)$ holds (this is called the "inductive step").

Then $P(n)$ holds for every natural number $n \in \mathbb{N}$.

PROOF. Define the set:

$$S \;:=\; \big\{n \in \mathbb{N} : P(n) \text{ is false}\big\} \;\subseteq\; \mathbb{N}.$$

Assume towards a contradiction that $P(n)$ does not hold for every natural number $n \in \mathbb{N}$. Thus $S \neq \emptyset$. By the Well-Ordering Principle, the set S has a least element $a := \min S$. Since $P(0)$ holds by assumption, we know that $0 < a$ (so $a - 1 \in \mathbb{N}$). By minimality of a, we also know that $P(0), \ldots, P(a-1)$ all hold. Thus by assumption (2) we conclude that $P(a)$ holds. This is a contradiction and so it must be the case that $P(n)$ is true for all $n \in \mathbb{N}$. □

1.2. Summations

We will often be in a situation where we want to add a bunch of numbers together. For instance, suppose $a_1, a_2, \ldots$ is a sequence of numbers and we are interested in the sum

$$a_1 + a_2 + \cdots + a_n.$$

This sum may be more compactly written in **summation notation** as

$$\sum_{k=1}^{n} a_k \quad \text{or} \quad \sum_{1 \leq k \leq n} a_k.$$

By definition, if n above is zero (corresponding to the *empty sum*), then we define the resulting summation to be 0. The letter k in our summations above is referred to as a **dummy variable** or **index variable**. In general, the specific letter used to denote the index variable doesn't matter as long as it is not being used for something which already has meaning. Thus the following sums are all equal:

$$\sum_{k=1}^{n} a_k \;=\; \sum_{i=1}^{n} a_i \;=\; \sum_{j=1}^{n} a_j \;=\; \cdots$$

For $m, n \in \mathbb{Z}$, the notation $\sum_{k=m}^{n} a_k$ is often called the **delimited summation notation** and this notation tells us to include in the sum every number a_k for which $m \leq k \leq n$, e.g.,

$$\sum_{k=5}^{10} a_k \;=\; a_5 + a_6 + a_7 + a_8 + a_9 + a_{10}.$$

The notation $\sum_{1\leq k\leq n} a_k$ is an example of **generalized summation notation**. Generalized summation notation allows us to consider summations of the form:

$$\sum_{P(k)} a_k$$

where $P(k)$ is some relation[2] involving the integer k. This notation tells us to include in the sum every number a_k for which $P(k)$ is true. For instance, in $\sum_{1\leq k\leq n} a_k$, the relation $P(k)$ is "$1 \leq k \leq n$". Most of the summation formulas we will consider are written with delimited notation, however it is often more convenient to work with the generalized notation. For example, the sum of all odd natural numbers below 100 can be written in both ways:

$$\sum_{k=0}^{49}(2k+1)^2 \;=\; \sum_{\substack{1\leq k<100\\ k \text{ odd}}} k^2$$

In this situation, the delimited form on the left might be easier to evaluate to a final answer, but the generalized form on the right is easier to understand intuitively. In some cases, there may be no good delimited form for a summation of interest, for instance:

$$\sum_{\substack{0\leq p\leq 20\\ p \text{ prime}}} p \;=\; 2+3+5+7+11+13+17+19$$

Another use for the generalized notation is that it makes it easy to shift indices while avoiding errors:

$$\sum_{k=1}^{n} a_k \;=\; \sum_{1\leq k\leq n} a_k \;=\; \sum_{1\leq k+1\leq n} a_{k+1} \;=\; \sum_{0\leq k\leq n-1} a_{k+1} \;=\; \sum_{k=0}^{n-1} a_{k+1}$$

and it also is helpful in interchanging the summations in certain "triangular" double-sums:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} a_{ij} \;=\; \sum_{1\leq i\leq j\leq n} a_{ij} \;=\; \sum_{j=1}^{n}\sum_{i=1}^{j} a_{ij}.$$

Basic summation operations. In this subsection, we state without proof some general operations which are allowed with summations. The conventional wisdom with summations says that summation identities are proved using mathematical induction. This is true, however, as it turns out you can accomplish quite a lot with summations by judiciously using rules 1.2.1, 1.2.2, 1.2.3, and 1.2.4 below.

Distributive Law 1.2.1. *Suppose $S(i)$ and $R(j)$ are relations which may or may not be true for integers i and j. Then*

$$\left(\sum_{R(i)} a_i\right)\left(\sum_{S(j)} b_j\right) \;=\; \sum_{R(i)}\left(\sum_{S(j)} a_i b_j\right).$$

As a special case of 1.2.1 where $R(i)$ is "$i = 0$" and $a_0 = a$, we get

$$a\sum_{S(j)} b_j \;=\; \sum_{S(j)} ab_j,$$

[2]At the moment, we are assuming that $P(k)$ is only true for finitely many integers, as we wish to only consider finite sums.

i.e., summations commute with multiplication by scalars.

Change of Variable 1.2.2. *Suppose $R(i)$ is a relation and $\pi : \mathbb{Z} \to \mathbb{Z}$ is a bijection. Then*

$$\sum_{R(i)} a_i \;=\; \sum_{R(\pi(i))} a_{\pi(i)}.$$

As a special case of 1.2.2 where $\pi(i) := i + 1$ for $i \in \mathbb{Z}$, we get

$$\sum_{R(i)} a_i \;=\; \sum_{R(i+1)} a_{i+1}.$$

Interchanging Order of Summation 1.2.3. *Suppose $R(i)$ and $S(j)$ are relations on integers. Then*

$$\sum_{R(i)} \sum_{S(j)} a_{ij} \;=\; \sum_{S(j)} \sum_{R(i)} a_{ij}.$$

As a special case of 1.2.3, we can derive

$$\sum_{R(i)} b_i + \sum_{R(i)} c_i \;=\; \sum_{R(i)} (b_i + c_i)$$

i.e., the sum of two summations over the same index set can be combined. The following shows how to combine sums over different index sets:

Manipulating the Domain 1.2.4. *Suppose $R(i)$ and $S(i)$ are relations on integers. Then*

$$\sum_{R(i)} a_i + \sum_{S(i)} a_i \;=\; \sum_{R(i) \text{ or } S(i)} a_i + \sum_{R(i) \text{ and } S(i)} a_i.$$

Common summation formulas.

Geometric Sum 1.2.5. *Suppose $x \neq 1$. Then*

$$\sum_{0 \le j \le n} x^j \;=\; \frac{1 - x^{n+1}}{1 - x}.$$

Proof. Note that

$$\begin{aligned}
\sum_{0 \le j \le n} x^j \;&=\; 1 + \sum_{1 \le j \le n} x^j \quad \text{by 1.2.4} \\
&=\; 1 + x \sum_{1 \le j \le n} x^{j-1} \quad \text{by 1.2.1} \\
&=\; 1 + x \sum_{0 \le j \le n-1} x^j \quad \text{by 1.2.2} \\
&=\; 1 + x \sum_{0 \le j \le n} x^j - x^n \quad \text{by 1.2.4.}
\end{aligned}$$

Comparing the first term with the last and solving for $\sum_{0 \le j \le n} x^j$ (which involves dividing by $1 - x$, which is possible by assumption), yields the desired formula. □

Triangular Numbers 1.2.6. *Suppose $n \geq 0$. Then*

$$\sum_{0 \le j \le n} j \;=\; \frac{n(n+1)}{2}.$$

PROOF. Note that

$$\begin{aligned}\sum_{0\le j\le n} j &= \sum_{0\le n-j\le n} (n-j) \quad \text{by 1.2.2}\\ &= \sum_{0\le j\le n} (n-j) \quad \text{by simplifying the domain}\\ &= \sum_{0\le j\le n} n - \sum_{0\le j\le n} j \quad \text{by 1.2.3}\\ &= n(n+1) - \sum_{0\le j\le n} j.\end{aligned}$$

By comparing the first term with the last and solving for $\sum_{0\le j\le n} j$, we get the desired formula. □

Infinite summations. We won't have too much need for infinite sums (i.e., series), but here is the definition (in the delimited notation):

$$\sum_{k=0}^{\infty} a_k \;:=\; \lim_{n\to\infty} \sum_{k=0}^{n} a_k \quad \text{(if this limit exists)}$$

The primary infinite series we will need is the *geometric series*:

Geometric Series 1.2.7. *Suppose* $|x| < 1$. *Then*

$$\sum_{k=0}^{\infty} x^k \;=\; \frac{1}{1-x}.$$

PROOF. By 1.2.5 we know that for each $n \ge 0$ that

$$\sum_{k=0}^{n} x^k \;=\; \frac{1-x^{n+1}}{1-x}.$$

Since $|x| < 1$, it follows that $\lim_{n\to\infty} x^{n+1} = 0$. Thus

$$\sum_{k=0}^{\infty} x^k \;=\; \lim_{n\to\infty} \frac{1-x^{n+1}}{1-x} \;=\; \frac{1}{1-x}.$$ □

1.3. Triangular number algorithms

In this section we encounter our first algorithms. As a follow-up to the previous section on summation, here we discuss two algorithms for computing the nth triangular number:

$$\sum_{j=0}^{n} j$$

Of course, there is nothing inherently difficult with writing an algorithm to compute this summation. However, this simple example will allow us to introduce two very important themes for this class: proving the correctness of an algorithm, and analyzing the running time of an algorithm. To compute this summation by hand (without using 1.2.6), you would need to start with 0. Then continually add numbers to your running partial sum until you add the last number n. The number you end up with is the value you want. The following algorithm does exactly this:

```
TRIANGLE(n)
1  Sum = 0
2  // Initializes Sum to 0
3  for j = 0 to n
4        Sum = Sum + j
5        // Replaces the current value of Sum with Sum + j
6        // This has the effect of adding j to Sum
7  return Sum
```

Since this is our first algorithm example, let's talk through what happens when we run the algorithm TRIANGLE for $n = 2$.

(1) We would first call TRIANGLE(2), so $n = 2$ as we run through the algorithm.
(2) In Line 1 we introduce a variable *Sum* and it gets assigned the value 0. Thus $n = 2$ and $Sum = 0$.
(3) Line 2 is a *comment*. It has no official meaning except to provide readers of the algorithm some commentary as to what is going on.
(4) Lines 3-6 is a **for** loop. Since $n = 2$, this means we will run Lines 4-6 (the *body* of the **for** loop) three times: first with $j = 0$, again with $j = 1$, and again with $j = 2$.
(5) The first time we run Line 4, we have $j = 0$ and currently $Sum = 0$. Thus the expression $Sum = Sum + 0$ means we compute $Sum + 0$ (which equals $0 + 0 = 0$), and then reassign *Sum* to this value. Thus our variable values are $Sum = 0$, $j = 0$, $n = 2$. Lines 5-6 are comments, so they don't do anything.
(6) The second time we run Line 4, we have $j = 1$ and currently $Sum = 0$. Thus we compute $Sum + j = 0 + 1 = 1$, and reassign *Sum* to be 1. Now our variables are $Sum = 1, j = 1, n = 2$.
(7) The third and final time we run Line 4, we have $j = 2$ and currently $Sum = 1$. Thus we compute $Sum + j = 1 + 2 = 3$, and reassign *Sum* to be 3. Now our variables are $Sum = 3, j = 2, n = 2$.
(8) Technically, the variable j in the **for** loop gets increased one last time to $j = 3$, and since $3 > 2$, the **for** loop body does not run again and we proceed to Line 7. (This feature is important for proving the correctness of the algorithm below).
(9) Now our **for** loop is finished, so we run Line 7. The pseudocode

    return *Sum*

    tells us to output the current value of the variable *Sum*, which is 3.
(10) To summarize, if we run TRIANGLE(2), the algorithm outputs the value 3. This indeed is the correct value, since $\sum_{j=0}^{2} j = 0 + 1 + 2 = 3$.

Algorithm correctness. At this point, there should be no doubt that TRIANGLE does what it is intended to do. However, we will illustrate how to formally *prove* that it is correct. The first thing to do is ask: what do we claim that this algorithm does? Answer: TRIANGLE(n) returns the sum $\sum_{j=0}^{n} j$. This tells us what statement we should prove:

Theorem 1.3.1. *The algorithm* TRIANGLE(n) *outputs the summation* $\sum_{j=0}^{n} j$.

Next, we need to ask: how do we prove such a statement? In terms of the pseudocode, this theorem says that the value of the variable *sum* is $\sum_{j=0}^{n} j$ once line 7 is run. This is what we *actually* need to show. Since this is a statement of the form "The value of variable __ is __ after line __", this suggests that if we prove something about the value of *Sum* after each line of code is run in order, then in particular we will know the state of *Sum* at the very end, which is what we really want. It is clear that *Sum* takes the value 0 after lines 1 and 2. What is the value of *Sum* after line 3?

This introduces the important idea of a *loop invariant*. The value of *Sum* after line 3 (i.e., immediately before the **for** loop on lines 4-6 run) does change depending on which iteration of the **for** loop we are on (more specifically: depending on the value of j). Thus we cannot make a definitive statement about the value of *Sum* after line 3 like we did after lines 1-2. However, we still can claim *something*. To get some intuition for the value of *Sum*, let's see what the value of *Sum* is immediately after line 3 when $n = 4$. In this case j will go from 0 to 5:

$$j = 0 \ : \ Sum = 0 = \sum_{k=0}^{0} k$$

$$j = 1 \ : \ Sum = 0 = \sum_{k=0}^{0} k$$

$$j = 2 \ : \ Sum = 1 = \sum_{k=0}^{1} k$$

$$j = 3 \ : \ Sum = 3 = \sum_{k=0}^{2} k$$

$$j = 4 \ : \ Sum = 6 = \sum_{k=0}^{3} k$$

$$j = 5 \ : \ Sum = 10 = \sum_{k=0}^{4} k$$

We notice two things: (1) the final value of *Sum* is what we want, and (2) even though *Sum* changes, there still is a fixed relationship between the value of j and the value of *Sum*, namely, every time j takes a certain value, then the value of *Sum* at that spot is $\sum_{k=0}^{\max(j-1,0)} k$. This in fact tells us what we should prove about the state of *Sum* every time line 3 is run. We state this formally as a **loop invariant**:

> (Loop invariant for TRIANGLE) At the start of each iteration of the **for** loop on lines 3-6 (i.e., immediately after each time line 3 is run, but before proceeding to line 4 or line 7) the value of the variable *Sum* is $\sum_{i=0}^{\max(j-1,0)} i$.

Ultimately we want to know the loop invariant is true once the **for** loop is finished (i.e., after the last time line 3 is run, when $j = n + 1$). For this we need to know the loop invariant is true every step along the way. To do this we need to prove three things about the loop invariant:

(1) **Initializiation:** We need to show that the loop invariant is true prior to the first iteration of the loop. What this means is that at the moment we just finished line 3 for the first time and $j = 0$, but prior to the first time we run line 4 we need to show that the loop invariant is true. Usually this step is easy and is analogous to the base case of an induction proof.
(2) **Maintenance:** We need to show that if the loop invariant is true after a certain time line 3 is run (except the last time), it remains true the next time line 3 is run (after running one more iteration of the body of the **for** loop and then running line 3 one more time). Usually this step is analogous to the inductive step of an induction proof.
(3) **Termination:** We need to show that after the very last time line 3 is run (here when $j = n + 1$ and before we proceed to line 7), the loop invariant gives us some useful property that helps us show the algorithm is correct.

We now take these ideas and package them into a proof of the correctness of TRIANGLE:

PROOF OF 1.3.1. The algorithm begins by assigning $Sum = 0$ on line 1. Line 2 does nothing since it is a comment. We claim that the state of the variables after each time line 3 is run is the following:

> (Loop invariant for TRIANGLE) At the start of each iteration of the **for** loop on lines 3-6 (i.e., immediately after each time line 3 is run, but before proceeding to line 4 or line 7) the value of the variable Sum is $\sum_{i=0}^{\max(j-1,0)} i$.

(Initialization) Just prior to the first iteration of the **for** loop, the variable $j = 0$ and $Sum = 0 = \sum_{i=0}^{\max(0,-1)} i$.

(Maintenance) Suppose the loop invariant is true after an iteration in which $j = j_0$ for some $0 \leq j_0 \leq n$. This means that the current value of Sum is $\sum_{i=0}^{\max(j_0-1,0)} i$. Next we run line 4 so the new value of Sum is $\sum_{i=0}^{\max(j_0-1,0)} i + j_0 = \sum_{i=0}^{\max(j_0,0)}$ (do a case distinction on whether $j_0 = 0$ or $j_0 > 0$). Then we return to line 3 and the value of j becomes $j = j_0 + 1$. The current value of Sum is still $\sum_{i=0}^{\max(j_0,0)} i = \sum_{i=0}^{\max(j-1,0)} i$. Thus the loop invariant is still true.

By Initialization and Maintenance, we now know that the loop invariant is always true. In the next step (Termination) we will exploit this loop invariant to tell us what the final value of Sum is, which will help us to finish the proof.

(Termination) After the **for** loop terminates, i.e., after the final time line 3 is run, we have $j = n + 1$ and our loop invariant is true. Thus in line 7 the value of Sum is $\sum_{i=0}^{\max(j-1,0)} i = \sum_{i=0}^{n} i$. Since the program outputs this value, the program is correct. □

Running time analysis. The next thing we are interested in is determining how long it takes TRIANGLE to run, as a function of n. This will foreshadow various concepts we will make precise in Chapter 2. Ultimately we will *count* the number of primitive computational steps that the computer does whenever we call TRIANGLE(n), as a function of the *input size*, which in this case is the number n. For the pseudocode we've used so far, we will use the following rules for counting:

(1) Each line of code can be done in a constant number of steps each time it is executed.

(2) Since we don't actually know (or care) what this constant number of steps is, and it may differ depending on what the instruction is or specifics of the computer architecture, each line will receive a different constant c_i.
(3) **for** loop tests (i.e., line 3 in TRIANGLE) get executed one additional time than the body of the **for** loop gets executed.
(4) Comments are not actual instructions, so they count as zero time.

Applying these rules to the pseudocode of TRIANGLE yields:

TRIANGLE(n)

1	$Sum = 0$	*cost:* c_1 *times:* 1
2	// Initializes...	*cost:* 0 *times:* 1
3	**for** $j = 0$ **to** n	*cost:* c_2 *times:* $n + 2$
4	$Sum = Sum + j$	*cost:* c_3 *times:* $n + 1$
5	// Replaces the...	*cost:* 0 *times:* $n + 1$
6	// This has ...	*cost:* 0 *times:* $n + 1$
7	**return** Sum	*cost:* c_4 *times:* 1

Now to determine the running time of our algorithm, we sum up the costs of each line times the number of times that line is run. The running time for TRIANGLE(n) is therefore:

$$c_1 + c_2(n + 2) + c_3(n + 1) + c_4 \;=\; (c_2 + c_3)n + (c_1 + 2c_2 + c_3 + c_4)$$

This expression is a little nasty. The good news is that since we don't care about particular constants, we might as well write the running time as

$$an + b$$

where a and b are constants which depend on c_1, c_2, c_3, c_4. Moreover, the only thing we *actually* care about is that this is a *linear* function and that the dominating term in this expression (as n gets very large) is n. In the parlance of Chapter 2 we summarize our analysis with three statements:

(1) The running time is $\Theta(n)$. Informally: the running time is bounded above and below by some linear function.
(2) The running time is $O(n)$. Informally: the running time is bounded *above* by some linear function (in this example, this is implied by (1)).
(3) The running time is $\Omega(n)$. Informally: the running time is bounded *below* by some linear function (also implied by (1)).

In a nutshell, this is the game we play when it comes to analyzing the running time of algorithms. We don't care about constants or lower-order terms, just whether the running time is *linear*, *quadratic*, *exponential*, etc. We will make all of these notions precise in our discussion of *asymptotics* in the next chapter.

A faster triangular number algorithm. Before we end our discussion of triangular numbers, it would be very remiss to not point out the obvious fact that 1.2.6 tells us

$$\sum_{j=0}^{n} j \;=\; \frac{n(n+1)}{2}$$

which we can use to write a much faster algorithm for triangular numbers:

TriangleFast(n)

1	$Sum = n$	*cost:* c_1 *times:* 1
2	// Initializes Sum to n	*cost:* 0 *times:* 1
3	$Sum = Sum \cdot (n+1)$	*cost:* c_2 *times:* 1
4	// Multiplies Sum by $n+1$	*cost:* 0 *times:* 1
5	$Sum = Sum/2$	*cost:* c_3 *times:* 1
6	// Divides by 2	*cost:* 0 *times:* 1
7	**return** Sum	*cost:* c_4 *times:* 1

Performing a similar running-time analysis[3] we find that the running time is a constant:

$$c_1 + c_2 + c_3 + c_4$$

Again, we don't care what constant this really is, just that it is a constant. We summarize this analysis by saying the running time of TriangleFast is $\Theta(1)$ (and also $O(1)$ and $\Omega(1)$). The takeaway here is that since any linear function will eventually dominate a fixed constant, we can conclude that TriangleFast will run faster (that is, finish in fewer steps) than Triangle for all sufficiently large values of n.

This also illustrates another common theme for this class: using mathematics (the formula 1.2.6 in this case), we can often come up with an algorithm which performs better than the "naive" algorithm we would first think of (TriangleFast vs. Triangle in this case).

Of course, we could just as easily perform the arithmetic operations done in TriangleFast all at once, resulting in a much shorter program:

TriangleFastV2(n)

1 **return** $n \cdot (n+1)/2$

This also runs in $\Theta(1)$ time and we might as well consider it equally fast as TriangleFast which also runs in $\Theta(1)$ time.

1.4. Common functions

In this section we review some common mathematical functions which show up in computer science and the analysis of algorithms.

Floors and ceilings. We will often be in a situation where we naturally want to work with *natural numbers* and *integers* (i.e., with $\mathbb{N}$ and $\mathbb{Z}$). For instance, our algorithms will deal with integers and the size of inputs to our algorithms will be expressed in natural numbers (number of elements in an array, number of bits, number of nodes and edges, etc.). However, in the analysis of algorithms we perform, we will often need to use techniques from calculus, which requires us to leave the realm of whole numbers and deal with real numbers (i.e., with $\mathbb{R}$). Thus, we need a systematic way to convert between real numbers and integers.

[3]Here we are pretending that integer addition, multiplication and division can all be done in constant time. As a rule of thumb, for integers with a small number of digits this is generally true and we will be happy to assume this in this class, although if your integers have a large number of digits (e.g., a million digits) then you need to consider efficient arithmetic algorithms. This is a story for another time.

The way we do this is with the **floor** (greatest integer) operation and **ceiling** (least integer) operation. For $x \in \mathbb{R}$ these are defined as follows:

$$\begin{aligned} \lfloor x \rfloor &:= \text{ the greatest integer less than or equal to } x \qquad (\text{floor of } x) \\ \lceil x \rceil &:= \text{ the least integer greater than or equal to } x \qquad (\text{ceiling of } x) \end{aligned}$$

For example, $\lfloor 2.5 \rfloor = 2$, $\lfloor -2.5 \rfloor = -3$, $\lceil 10 \rceil = 10$ and $\lceil -0.5 \rceil = 0$. The floor operation has the effect of always rounding *down*, and the ceiling operation has the effect of always rounding *up*; hence the names *floor* and *ceiling*.

From a logical point of view, the floor and ceiling operators can be a little tricky to master, but they are very useful functions and most programming languages include them as primitive operations (along with $+, \cdot, -, /$) so it is worthwhile to become familiar with them. One nice feature of these functions is that they serve as an indicator for when a real number is secretly an integer. For $x \in \mathbb{R}$ we have:

$$\lfloor x \rfloor = x \iff x \text{ is an integer} \iff \lceil x \rceil = x.$$

The following inequalities get used all the time: for $x \in \mathbb{R}$,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Finally, we have the following *reflection principles* which allow us to convert between ceilings and floors: for $x \in \mathbb{R}$,

$$\lfloor -x \rfloor = -\lceil x \rceil \quad \text{and} \quad \lceil -x \rceil = -\lfloor x \rfloor$$

All of these properties are useful when proving facts about floors and ceilings, as well as the following ten properties:

Fact 1.4.1. Suppose $x \in \mathbb{R}$ and $k \in \mathbb{Z}$. Then

(1) $\lfloor x \rfloor = k$ iff $k \leq x < k + 1$
(2) $\lfloor x \rfloor = k$ iff $x - 1 < k \leq x$
(3) $\lceil x \rceil = k$ iff $k - 1 < x \leq k$
(4) $\lceil x \rceil = k$ iff $x \leq k < x + 1$
(5) $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
(6) $\lceil x + k \rceil = \lceil x \rceil + k$
(7) $x < k$ iff $\lfloor x \rfloor < k$
(8) $k < x$ iff $k < \lceil x \rceil$
(9) $x \leq k$ iff $\lceil x \rceil \leq k$
(10) $k \leq x$ iff $k \leq \lfloor x \rfloor$

The following example shows how to go about proving something with floors:

Fact 1.4.2. For $x \in \mathbb{R}$, if $x \geq 0$, then

$$\left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor = \lfloor \sqrt{x} \rfloor$$

Proof. Let $m = \left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor$. Then by Fact 1.4.1(1) we have

$$0 \leq m \leq \sqrt{\lfloor x \rfloor} < m + 1.$$

Squaring both sides yields $m^2 \leq \lfloor x \rfloor < (m+1)^2$. Next, by Fact 1.4.1(10) and (7) we have $m^2 \leq x < (m+1)^2$. Next, we take a square root of this inequality which yields $m \leq \sqrt{x} < m + 1$. Finally, by Fact 1.4.1(1) again we get $\lfloor \sqrt{x} \rfloor = m$. Thus

$$\left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor = m = \lfloor \sqrt{x} \rfloor$$

and our assertion is proven. □

The modulus operator. Another important function in mathematics and computer science is the *modulus* operator. Recall that according to the Division Algorithm 1.1.2 for $a = 100$ and $b = 17$, we can divide a by b to get:

$$100 \;=\; 17 \cdot 8 + 15 \quad \text{and} \quad 0 \leq 15 < 17.$$

In this case we say that 15 is the *remainder* upon division of 100 by 17. In many situations, it will be desirable to get access to this remainder rather quickly. This is what the modulus operator does.

Definition 1.4.3. Fix $n \geq 1$. Define the **modulus operator** (with respect to n) to be the function

$$k \mapsto k \bmod n : \mathbb{Z} \to \{0, \ldots, n-1\}$$

defined for $k \in \mathbb{Z}$ by

$$k \bmod n \;:=\; \text{the unique } r \in \{0, \ldots, n-1\} \text{ such that there is } q \in \mathbb{Z}$$
$$\text{such that } k = nq + r \text{ and } 0 \leq r < n.$$

In other words, $k \bmod n$ is the remainder r you get when you divide k by n in the division algorithm. The modulus operator is also a primitive function which is included in many programming languages. For instance, in Python the modulus operator is denoted by $k\%n$.

The modulus operator has many uses. For instance, it allows us to test if a number is even or odd:

Fact 1.4.4. Suppose $k \in \mathbb{Z}$. Then k is even iff $k \bmod 2 = 0$ and k is odd iff $k \bmod 2 = 1$.

We can also use the modulus operator to define what it means for n to *divide* k (i.e., divide with no remainder):

Definition 1.4.5. Suppose $n \geq 1$ and $k \in \mathbb{Z}$. We say that n **divides** k (notation: $n|k$) if $k \bmod n = 0$. Equivalently, n divides k if there exists $q \in \mathbb{Z}$ such that $k = nq$. If $n|k$ and $q \in \mathbb{Z}$ is such that $na = k$, then we shall call n a **divisor** (or **factor**) of k, and we shall call q the **quotient** of k divided by n.

For example, $10|50$ because $50 \bmod 10 = 0$, however $10 \nmid 51$ because $51 \bmod 10 = 1$, i.e., dividing 51 by 10 leaves a remainder of 1.

The Division Algorithm, floors, and the modulus operator are all connected to each other by the following fact:

Fact 1.4.6. Suppose $a, b \in \mathbb{Z}$ and $b \geq 1$. Set $q := \lfloor a/b \rfloor$ and $r := a \bmod b$. Then

$$a \;=\; bq + r \quad \text{and} \quad 0 \leq r < b.$$

In particular, $a \bmod b = a - b \lfloor a/b \rfloor$.

We are also in a position now to define one of the most important notions in all of mathematics:

Definition 1.4.7. A natural number $p \in \mathbb{N}$ is called a **prime number** if $p \neq 1$ and its only nonnegative divisors are 1 and p; in symbols:

$$p \text{ prime } :\Longleftrightarrow\; p \neq 1 \& \forall d \in \mathbb{N}\ (d|p \to d = 1 \text{ or } d = p)$$

A natural number $n \geq 2$ which is not prime is called **composite**.

Here are some of the main properties of divisibility:

Divisibility Properties 1.4.8. *For every $a, b, c \in \mathbb{Z}$ the following hold:*

(D1) $a|0$, $1|a$, $a|a$
(D2) $a|1$ *if and only if* $a = \pm 1$
(D3) *if* $a|b$ *and* $c|d$, *then* $ac|bd$
(D4) *if* $a|b$ *and* $b|c$, *then* $a|c$
(D5) $a|b$ *and* $b|a$ *if and only if* $a = \pm b$
(D6) *if* $a|b$ *and* $b \neq 0$, *then* $|a| \leq |b|$
(D7) *if* $a|b$ *and* $a|c$, *then for every* $x, y \in \mathbb{Z}$, $a|(bx + cy)$

In particular, the divides relation | is reflexive (D1) and transitive (D4).

The "mod" notation has another important meaning: as an *equivalence relation*:

Definition 1.4.9. Fix $n \geq 1$. We say that two integers $a, b \in \mathbb{Z}$ are **congruent modulo** n (notation: $a \equiv b \pmod{n}$) if $a \bmod n = b \bmod n$, i.e., if a and b leave the same remainder upon division by n. Equivalently:

$$\begin{aligned} a \equiv b \pmod{n} &\iff n|a - b \\ &\iff \text{there exists } q \in \mathbb{Z} \text{ such that } nq = a - b \\ &\iff a \bmod n = b \bmod n. \end{aligned}$$

For example, since $12 \bmod 5 = 2 = 7 \bmod 5$, then it follows that $12 \equiv 7 \pmod{5}$. Here are some basic properties of the congruence relation which you are free to use:

Congruence Properties 1.4.10. *Fix $n \geq 1$. Then for every $a, b, c, d \in \mathbb{Z}$ the following hold:*

(C1) $a \equiv a \pmod{n}$
(C2) *if* $a \equiv b \pmod{n}$, *then* $b \equiv a \pmod{n}$
(C3) *if* $a \equiv b \pmod{n}$ *and* $b \equiv c \pmod{n}$, *then* $a \equiv c \pmod{n}$
(C4) *if* $a \equiv b \pmod{n}$ *and* $c \equiv d \pmod{n}$, *then* $a + c \equiv b + d \pmod{n}$ *and* $ac \equiv bd \pmod{n}$
(C5) *if* $a \equiv b \pmod{n}$, *then* $a^m \equiv b^m \pmod{n}$ *for every* $m \in \mathbb{N}$.

Remark 1.4.11. The primary role of elementary number theory in this class will be for the study the Euclidean Algorithm in Section 2.3. We will not have much use for the divisibility and congruence relation in the rest of the algorithms we'll study, although you may find them useful in the various programming exercises. However, it does not hurt to be aware of the notions in this subsection for the following reasons:

(1) The modulus operator `%` is used very frequently in programs so it is good to know what it does and how it works.
(2) Elementary number theory serves as the foundation for cryptography, a very important topic in computer science, especially in the field of computer security. See, for instance, Section 31.7 of [**1**].
(3) These notions are fundamental in mathematics, especially in algebra. Thus it is good to be introduced to these ideas, at least in passing.

Logarithms and exponential functions. *In this subsection we fix a real number $b > 0$.* Recall that if k is an integer, then the exponentiation b^k is defined

as:

$$(\dagger) \qquad b^k := \begin{cases} \underbrace{b \times \cdots \times b}_{k \text{ times}} & \text{if } k > 0 \\ 1 & \text{if } k = 0 \\ \underbrace{b^{-1} \times \cdots \times b^{-1}}_{-k \text{ times}} & \text{if } k < 0 \end{cases}$$

This exponential function satisfies the following properties:

- (P1) (Exponent rule) For every $k, \ell \in \mathbb{Z}$, $b^{k+\ell} = b^k b^\ell$.
- (P2) (Monotonicity) For every $k, \ell \in \mathbb{Z}$ such that $k < \ell$:
  - (a) if $b > 1$, then $b^k < b^\ell$
  - (b) if $b = 1$, then $b^k = b^\ell = 1$
  - (c) if $b < 1$, then $b^k > b^\ell$

What if we want to extend this definition to make sense for all $x \in \mathbb{R}$, not just $k \in \mathbb{Z}$? Obviously you cannot literally extend the definition (†) if k is not an integer, since it doesn't make sense to multiply a number times itself a fractional (or irrational) number of times. However, as it turns[4] out there is a unique functions "$x \mapsto b^x : \mathbb{R} \to \mathbb{R}$" which satisfies properties (P1) and (P2) above, with k, ℓ ranging over $\mathbb{R}$ instead of $\mathbb{Z}$. For this class, we will assume the existence of such an exponential function. For the sake of completeness, here are its primary properties:

Fact 1.4.12. Suppose $b > 0$, and $x, y \in \mathbb{R}$. Then

1. $b^0 = 1$
2. $b^1 = b$
3. $b^{-1} = 1/b$
4. $(b^x)^y = b^{xy}$
5. $(b^x)^y = (b^y)^x$
6. $b^x b^y = b^{x+y}$
7. if $b > 1$, then
   - (a) $x \mapsto b^x$ is strictly increasing: if $x < y$, then $b^x < b^y$
   - (b) $\lim_{x \to +\infty} b^x = +\infty$
   - (c) $\lim_{x \to -\infty} b^x = 0$
8. $b^x > 0$

$x \mapsto b^x$ is a continuous function

If $b \neq 1$, then b^x has a functional inverse (since it is strictly increasing/decreasing and continuous). This is called a *logarithm*.

Definition 1.4.13. Suppose $b > 0$ and $b \neq 1$. If $x > 0$, then we define the **logarithm of x to the base b** (notation: $\log_b x$) to be the unique $y \in \mathbb{R}$ such that $b^y = x$. In other words:

$$y = \log_b x :\Longleftrightarrow b^y = x.$$

Here are some of the important properties of logarithms:

Fact 1.4.14. Suppose $a, b, c, x \in \mathbb{R}$ are such that $a, b, c > 0$. Then

1. $a = b^{\log_b a}$ (if $b \neq 1$)
2. $\log_c(ab) = \log_c a + \log_c b$ (if $c \neq 1$)
3. $\log_b a^x = x \log_b a$ (if $b \neq 1$)

[4]See, for instance, [**2**, §1.6] for an explanation how this can be done.

(4) $\log_b a = \log_c a / \log_c b$ (if $b, c \neq 1$)
(5) $\log_b(1/a) = -\log_b a$ (if $b \neq 1$)
(6) $\log_b a = 1/\log_a b$ (if $a, b \neq 1$)
(7) $a^{\log_b c} = c^{\log_b a}$ (if $b \neq 1$)

The number b in the expression $\log_b x$ is called the **base** of the logarithm. Logarithms can have any base, but some bases are better suited than others for our purposes. Two important bases we will use are 2 and $e = \lim_{n\to\infty}(1 + 1/n)^n = \sum_{k=0}^{\infty} 1/k! = 2.71828....$ We have special notations for these logarithms as they arise so frequently:

(1) $\lg x := \log_2 x$ (the **binary logarithm**)
(2) $\ln x := \log_e x$ (the **natural logarithm**)

Finally, we conclude with some properties of logarithms from calculus:

Fact 1.4.15. Suppose $b > 0$ is such that $b \neq 1$. Then

(1) $\frac{d}{dx} \log_b x = 1/x \ln b$
(2) in particular, $\frac{d}{dx} \ln x = \frac{1}{x}$
(3) $\frac{d}{dx} \ln f(x) = f'(x)/f(x)$
(4) $\int \ln x dx = x \ln(x) - x + C$
(5) $\ln(t) = \int_1^t dx/x$

In general, to compute with logarithms other than the natural logarithm, you typically first convert to the natural logarithm using Fact 1.4.14, do the computation, then convert back.

1.5. Fibonacci numbers

In this section we give some more examples of algorithms, together with their analysis. Recall that the sequence $(F_n)_{n\geq 0}$ **Fibonacci numbers** is the sequence:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

defined by the recursive definition:

$$F_0 := 0,\ F_1 := 1, \quad \text{and} \quad F_n := F_{n-1} + F_{n-2} \quad \text{for } n \geq 2.$$

The Fibonacci numbers have many applications in such diverse fields as mathematics, computer science, biology, art, music, economics, and nature. At the moment our interest in them are that they serve as an example of something we can write algorithms to compute.

Fibonacci algorithms. Our goal is to have an algorithm which computes the nth Fibonacci number when the number n is input. As a first attempt, we can directly translate the recursive definition of $(F_n)_{n\geq 0}$ into a recursive algorithm, i.e., an algorithm which calls itself with a smaller input:

```
FIBONACCI(n)
1  if n == 0 or n == 1
2       return n
3  else
4       return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

For some intuition for how this algorithm works, let's trace the program for various examples of small input. First suppose we call FIBONACCI(0). Then the following happens:

(1) Before line 1 runs, we have only one variable n, and $n = 0$.
(2) In line 1, we check the condition "$n == 0$ or $n == 1$" for whether this is true or false. Since $n = 0$, then this statement is true, which means we next go to line 2.
(3) In line 2, the instruction is to return the value of n, so we return 0. Note that we have just shown that FIBONACCI(0) returns $0 = F_0$, as expected.

If we call FIBONACCI(1), then the algorithm will return $1 = F_1$, for the same reason. Let's see now what happens when we call FIBONACCI(2):

(1) In line 1 the condition "$n == 0$ or $n == 1$" is false since $n = 2$. This means that we do not go to line 2 and instead we go to line 3.
(2) Line 3 tells us that we should go to line 4.
(3) In line 4, we have to do several things. First we call FIBONACCI(1) (which will return 1), then we call FIBONACCI(0) (which will return 0), and then we add them together: $1 + 0 = 1$, and then we return this value 1. Thus calling FIBONACCI(2) will return $1 = F_2$, as expected.

We could keep going with these examples, (show FIBONACCI(3) returns $F_3 = 2$, show FIBONACCI(4) returns $F_4 = 5$, etc.) although it will be more useful to characterize what happens for arbitrary n. In other words, we will prove the *correctness* of the algorithm, i.e., we will prove that FIBONACCI does what it is purported to do. In order to do this, we must first ask and answer the question: what are we claiming the algorithm FIBONACCI does? Of course, the answer is: FIBONACCI(n) should return F_n.

The reason we ask and answer this question is because the answer tells us what we need to prove in order to prove the correctness of the algorithm. In this case, our answer tells us we should prove the following:

Theorem 1.5.1. *For every*[5] $m \geq 0$, FIBONNACI(m) *returns* F_m.

Now that we've isolated the statement we need to prove, we can discuss proof strategy. Here, the statement is of the form "For every $m \geq 0$,...". One tool that we have for proving such statements is *induction*. In order to know whether induction would be a worthwhile proof strategy or not, we should ask the question: will knowing the behavior of FIBONACCI(k) for $k < m$ help us determine the behavior for FIBONACCI(m)? In this case the answer is *yes* because FIBONACCI calls itself with smaller values as input. This is the indication to us that we should try proof by induction.

PROOF OF THEOREM 1.5.1. The statement we will prove by induction on $m \geq 0$ is:

$$P(m): \quad \text{"FIBONACCI}(m) \text{ returns } F_m\text{."}$$

We will prove two base cases.

($m = 0$) Suppose $m = 0$. Then FIBONACCI(0) returns $0 = F_0$ as we have discussed above.

[5]In the statement of Theorem 1.5.1 and its proof we will use the variable m instead of n, since n already shows up as the name of a variable in FIBONACCI.

($m = 1$) Suppose $m = 1$. Then FIBONACCI(1) returns $1 = F_1$ for similar reasons.

(Inductive Step) Suppose for some $m \geq 1$ we know that $P(0), \ldots, P(m)$ are all true. We will use this to show that $P(m+1)$ is true. We will show this by tracing the pseudocode with the value $n = m + 1$. Note that in this case $m + 1 \geq 2$. :

(1) In line 1 we check the condition "$n == 0$ or $n == 1$". Since $n = m+1 \geq 2$, this statement is false, so we proceed to line 3.
(2) Line 3 tells us to go to line 4.
(3) In line 4, we call FIBONACCI$(n-1)$, which is FIBONACCI(m) and we also call FIBONACCI$(n-2)$ which is FIBONACCI$(m-1)$. By the inductive assumption, both $P(m)$ and $P(m-1)$ are assumed to be true, so FIBONACCI(m) returns F_m and FIBONACCI$(m-1)$ returns F_{m-1}. Next, line 4 adds these two numbers together and returns them. Adding them together yields $F_m + F_{m-1} = F_{m+1}$. Thus we return F_{m+1}.

To summarize, we have just shown that calling FIBONACCI$(m+1)$ returns F_{m+1}, so the inductive step is done.

By the Principle of Induction, we can now conclude that $P(m)$ is true for all $m \geq 0$, and thus Theorem 1.5.1 is now proved. □

The next order of business is to analyze the running time of FIBONACCI. Since the flow of the algorithm behaves differently depending on whether $n \leq 1$ or $n \geq 2$, we will consider these two cases separately. First, suppose $n \leq 1$. Then our analysis is:

FIBONACCI(n)

1	**if** $n == 0$ or $n == 1$	*cost* : c_1 *times* : 1
2	**return** n	*cost* : c_2 *times* : 1
3	**else**	*cost* : ? *times* : 0
4	**return** FIBONACCI$(n-1)$ + FIBONACCI$(n-2)$	*cost* : ? *times* : 0

If we let $T(n)$ denote the running time of FIBONACCI(n), then we have just shown that if $n \leq 1$, then $T(n) = c_1 + c_2$ is a constant. If we combine these constants, we can instead say $T(n) = c$ if $n \leq 1$. Next we will do an analysis in the case where $n \geq 2$:

FIBONACCI(n)

1	**if** $n == 0$ or $n == 1$	*cost* : c_3 *times* : 1
2	**return** n	*cost* : ? *times* : 0
3	**else**	*cost* : c_4 *times* : 1
4	**return** FIBONACCI$(n-1)$ + FIBONACCI$(n-2)$	*cost* : $T(n-1) + T(n-2) + c_5$ *times* : 1

Here an explanation is in order for the cost of line 4. In line 4 we are really doing four things:

(1) Calling FIBONACCI$(n-1)$, which has cost $T(n-1)$
(2) Calling FIBONACCI$(n-2)$, which has cost $T(n-2)$
(3) Adding these together, which has constant cost
(4) Returning the sum, which also has constant cost, so we combine the constant costs of (3) and (4) into a single constant c_5

Summing all the associated costs of line 4 yields $T(n-1)+T(n-2)+c_5$. Thus for $n \geq 2$ we have

$$T(n) \;=\; c_3 + c_4 + T(n-1) + T(n-2) + c_5$$

If we collect all these constants we can say instead if $n \geq 2$, then $T(n) = T(n-1) + T(n-2) + d$ for some constant d. Putting everything together, we get a *recurrence relation* for the running time $T(n)$ of FIBONACCI:

$$T(n) \;=\; \begin{cases} c & \text{if } n = 0, 1 \\ T(n-1) + T(n-2) + d & \text{if } n \geq 2 \end{cases}$$

From this recurrence relation, and a little bit of work, we get the following:

Theorem 1.5.2. *The running time of* FIBONACCI(n) *is* $T(n) = \Theta(\phi^n)$*, where* $\phi = (1+\sqrt{5})/2$.

PROOF. This is Exercise 2.4.7 on HW2. □

The main takeaway from Theorem 1.5.2 is that FIBONACCI(n) runs in *exponential* time. In practice this means that it is SLOW. So slow, in fact, that if you were to race the computer by hand to compute F_{100}, then you would win, even though this might take you a few hours. So slow, that even a newborn baby would beat the computer; because the baby could just wait twenty years and then compute F_{100}. In fact, at a billion operations per second, it would take the computer several thousand years to finally compute F_{100} (if it didn't run out of memory space in the first few minutes, that is).

We don't say this to embarrass the computer. After all, we were the ones who wrote the program in the first place. Instead, our goal here is to bring awareness to a grave source of inefficiency so that in the future we can recognize it and avoid it. What makes FIBONACCI so inefficient? Let's investigate. For $n = 4$, the algorithm computes F_4 as follows (we'll abbreviate FIBONACCI as just FIB):

$$F_4 \;=\; \text{FIB}(3) + \text{FIB}(2).$$

So we have to compute FIB(3) and FIB(2). This is done also by calling FIB two more times each:

$$F_4 \;=\; \big(\text{FIB}(2) + \text{FIB}(1)\big) + \big(\text{FIB}(1) + \text{FIB}(0)\big)$$

Each of the FIB(1) and FIB(0) will simply return 1 and 0, respectively, but the FIB(2) still needs to call FIB twice:

$$F_4 \;=\; \big(((\text{FIB}(1)+\text{FIB}(0))+\text{FIB}(1))\big)+\big(\text{FIB}(1)+\text{FIB}(0)\big) \;=\; ((1+0)+1)+(1+0) \;=\; 3.$$

Notice that ultimately FIB(1) gets called 3 times and FIB(0) gets called 2 times in order to compute FIB(4). This is a little excessive and silly because once the computer figures out what FIB(0) and FIB(1) are, it shouldn't need to keep re-figuring it out. This is also the source of the inefficiency: we are asking the computer to do the same exact calculations again and again!

This also gives us an idea how to implement a faster algorithm. We can simply *store* the values of smaller Fibonacci numbers once we've calculated them, and then the next time we need those values, we can just *look up* the stored values instead of re-calculating them. This is also the way that human beings would naturally

compute larger Fibonacci numbers by hand. The following algorithm does just that:

```
FibonacciFast(n)
 1  if n ≤ 1
 2       return n
 3  let F[0..n] be a new array
 4  // initializes an array with n + 1 empty entries
 5  // F will store all Fibonacci numbers from F_0 to F_n
 6  F[0] = 0
 7  F[1] = 1 // Assign first two Fibonacci numbers to first two array entries
 8  for j = 2 to n
 9       F[j] = F[j − 1] + F[j − 2]
10       // Use recursive formula for jth Fibonacci number to fill in jth entry
11  return F[n]
```

Let's trace through this example to see how the code works. For $n = 0$ or $n = 1$, FibonacciFast works the same way as Fibonacci and immediately returns 0 or 1. Suppose $n = 4$. Then:

(1) In line 1, the condition "$n \leq 1$" is checked. Since $n = 4$, this condition is false so we move to line 3.
(2) In line 3, we initialize an array which can hold 5 entries. We'll denote this array as $F = [?, ?, ?, ?, ?]$.
(3) In line 6 we assign the value 0 to the 0th entry of the array. So now the array looks like $F = [0, ?, ?, ?, ?]$
(4) In line 7 we assign the value 1 to the 1st entry of the array. So now the array looks like $F = [0, 1, ?, ?, ?]$
(5) In line 8 we encounter a **for** loop. The counter j takes the value $j = 2$. The array still looks like $F = [0, 1, ?, ?, ?]$
(6) In line 9 we assign the $j = 2$nd array entry the value $F[1]+F[0] = 1+0 = 1$. The array now looks like $F = [0, 1, 1, ?, ?]$
(7) We go back to line 8, and the counter j takes the value $j = 3$. The array still looks like $F = [0, 1, 1, ?, ?]$
(8) We go to line 9 and assign the $j = 3$rd array entry the value $F[2] + F[1] = 1 + 1 = 2$. The array now looks like $F = [0, 1, 1, 2, ?]$
(9) We go to line 8, the counter j takes the value $j = 4$. The array still looks like $F = [0, 1, 1, 2, ?]$
(10) We go to line 9 and assign the $j = 4$th array entry the value $F[3] + F[2] = 2 + 1 = 3$. The array now looks like $F = [0, 1, 1, 2, 3]$
(11) We go back to line 8 one last time. The counter becomes $j = 5$. Since $5 > 4$, we do not go into the body of the loop again. The array still looks like $F = [0, 1, 1, 2, 3]$.
(12) We go to line 11 now. We return $F[n] = F[4] = 3$. Thus FibonacciFast(4) returns $3 = F_4$, as expected.

Now that we're convinced that FibonacciFast returns the Fibonacci numbers correctly, let's verify this. Again, the statement we wish to prove is:

Theorem 1.5.3. *For every $m \geq 0$,* FibonacciFast(m) *returns F_m.*

As we see, this is a statement of the form "For every $m \geq 0$,..." so the next question we should ask is: to induct or not to induct? To answer this question, we ask: would knowing the behavior of FibonacciFast(k) for $k < m$ help us in any way in determining the behavior of FibonacciFast(m)? Looking at the pseudocode for FibonacciFast, we see that nowhere does it call FibonacciFast for lower values, which is an indication that a proof by induction won't help us. Instead we will do a *direct proof*.

Proof of Theorem 1.5.3. Suppose $m \geq 0$. We have two cases to consider:

Case 1: ($m \leq 1$) In this case we see that because of lines 1 and 2, FibonacciFast(m) will return m. Since $m = 0 = F_0$ or $m = 1 = F_1$, this also means it returns F_m as desired.

Case 2: ($m \geq 2$) In this case, the condition in line 1 is false so we skip to line 3. Then $F = [?, ?, \ldots, ?]$ gets initialized as an array with $m+1$ entries. After lines 6 and 7 the array is $F = [0, 1, ?, \ldots, ?]$ with $F[0] = 0 = F_0$, $F[1] = 1 = F_1$, and the last $m-1$ entries are still empty. Next we enter line 8, our **for** loop condition. Since we will encounter line 8 multiple times, we need to know what's true about the array after every time line 8 is run, not just the first time. The claim about whats true after line 8 is our loop invariant. To come up with a loop invariant, we can look at our $n = 4$ example above and see what was true each time we finished line 8 (this is items (5), (7), (9), (11) from that list). Notice that we had the following situations occur:

- $j = 2$ and $F = [0, 1, ?, ?, ?]$
- $j = 3$ and $F = [0, 1, 1, ?, ?]$
- $j = 4$ and $F = [0, 1, 1, 2, ?]$
- $j = 5$ and $F = [0, 1, 1, 2, 3]$

The pattern we see is that the values of $F[0]$ through $F[j-1]$ are always F_0 through F_{j-1}. This will be our loop invariant:

> (Loop Invariant) Each time line 8 is run, for every $0 \leq k \leq j-1$ we have $F[k] = F_k$.

We will now establish that this loop invariant is correct.

(Initialization) The first time line 8 is run, we have $j = 2$ and $F[0, 1, ?, \ldots, ?]$ and so for every $0 \leq k \leq 2 - 1 = 1$ we have $F[k] = F_k$.

(Maintenance) Suppose we have just finished line 8, the current value of j is $j = j_0$ where $2 \leq j_0 \leq n$, and the loop invariant is assumed true for this value of j. Thus for every $0 \leq k \leq j_0 - 1$ we have $F[k] = F_k$. Next we enter line 9, so we assign $F[j_0] = F[j_0 - 1] + F[j_0 - 2] = F_{j_0-1} + F_{j_0-2} = F_{j_0}$. Now for every $0 \leq k \leq j_0$ we have $F[k] = F_k$. Next we go back to line 8 and increment the counter to $j = j_0 + 1$. Thus for every $0 \leq k \leq j_0 = j - 1$ we have $F[k] = F_k$. Thus the loop invariant is still true for the new value of j.

[Note: The Initialization and Maintenance step combined, now proved, tell us that the Loop Invariant is always true. The next step Termination is what helps us finish the overall proof since it tells the current state of the array after we are done with the **for** loop and move on to line 11.]

(Termination) The final time line 8 is run, we have $j = n + 1$. Since the loop invariant is true, we have $F[k] = F_k$ for every $0 \leq k \leq j - 1 = n$. In particular, $F[n] = F_n$. In line 11 we return $F[n]$, which means we return F_n, as desired. □

Remark 1.5.4. The proof of Theorem 1.5.3 is considered a *direct proof* and not a *proof by induction*. It is a direct proof because it proves FibonacciFast(m) returns F_m for each value of m separately. For instance, if you read the proof with the value $m = 100$ in mind, then it tells you directly why FibonacciFast(100) returns F_{100}, and at no point does the argument rely on or need the fact that FibonacciFast(99) returns F_{99}.

However, for this direct proof in the case when $m \geq 2$, we are responsible for proving what is the state of the array each time after line 8 is run. However in some sense, there is nothing special about line 8. Indeed, in order to prove something about the state of the system at the very last line, we are responsible for knowing what the state of the system is in every single line encountered before that. For lines 1,3,6,7 this is not an issue since these lines are only executed once and we can just say what happens. For line 8 this line is executed an arbitrary number of times, so our claim about the system after line 8 needs to be something which is true regardless of which time line 8 is run. This is what the loop invariant gives us. Then after establishing the loop invariant we proceed to line 11, and we can conclude from our knowledge about the state of the system after line 8 is run (which is the line executed immediately before line 11), what the state of the system is after line 11 is run, which is ultimately what we need to prove.

Since all of the nontrivial work involves proving the loop invariant, this is the reason why we sometimes only prove the loop invariant when proving the correctness of an algorithm (because knowing whats true after lines 1,3,6,7 is easy), but officially we should be making and proving claims about whats true after *every* line of code.

Finally, we analyze the running time of FibonacciFast(n). For this, we can assume that $n \geq 2$, since for $n = 0$ or $n = 1$ the running time will be just a constant (and we are interested really in the running time for large n). Note that

FibonacciFast(n)

1	**if** $n \leq 1$	*cost*:c_1 *times*:1
2	**return** n	*cost*:? *times*:0
3	let $F[0\,..\,n]$ be a new array	*cost*:c_2 *times*:1
4	// initializes...	*cost*:0 *times*:1
5	// F will store...	*cost*:0 *times*:1
6	$F[0] = 0$	*cost*:c_3 *times*:1
7	$F[1] = 1$ // Assign first...	*cost*:c_4 *times*:1
8	**for** $j = 2$ **to** n	*cost*:c_5 *times*:n
9	$F[j] = F[j-1] + F[j-2]$	*cost*:c_6 *times*:$n-1$
10	// Use...	*cost*:0 *times*:$n-1$
11	**return** $F[n]$	*cost*:c_1 *times*:1

Thus the running time is $T(n) = an + b$ for appropriate constants a, b. We summarize this as follows:

Theorem 1.5.5. *The running time of* FibonacciFast *is* $\Theta(n)$.

A linear running time is a huge improvement over an exponential running time. Using FibonacciFast, the computer will now easily outperform any human being (as it should).

1.6. Exercises

Exercise 1.6.1. Write out the following two sums in full:

(1) $\sum_{0\le k\le 5} a_k$
(2) $\sum_{0\le k^2\le 5} a_{k^2}$

Exercise 1.6.2. Evaluate the following summation:

$$\sum_{k=1}^{n} k2^k.$$

Hint: rewrite as a double sum.

Exercise 1.6.3. Suppose $x \neq 1$. Prove that

$$\sum_{j=0}^{n} jx^j \;=\; \frac{nx^{n+1} - (n+1)x^{n+1} + x}{(x-1)^2}.$$

Challenge: do this *without* using mathematical induction.

Exercise 1.6.4. Dr. I. J. Matrix has observed a remarkable sequence of formulas:

$$9\times 1+2 \;=\; 11, \quad 9\times 12+3 \;=\; 111, \quad 9\times 123+4 \;=\; 111, \quad 9\times 1234+5 \;=\; 11111.$$

(1) Write the good doctor's great discovery in terms of the Σ-notation.
(2) Your answer to part (1) undoubtedly involves the number 10 as base of the decimal system; generalize this formula so that you get a formula that will perhaps work in any base b.
(3) Prove your formula from part (2). The summation formulas from HW1 might be helpful.

Exercise 1.6.5 (Horner's rule)**.** The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) \;&=\; \sum_{k=0}^{n} a_k x^k \\ &=\; a_0 + x\big(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n))\big), \end{aligned}$$

given coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

(1) In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?
(2) Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?
(3) Consider the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-3

$$y \;=\; \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

(4) Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

Exercise 1.6.6. Suppose $m, n \in \mathbb{Z}$ are such that $m > 0$. Prove that

$$\left\lceil \frac{n}{m} \right\rceil \;=\; \left\lfloor \frac{n+m-1}{m} \right\rfloor$$

This gives us another *reflection principle* between floors and ceilings when the argument is a rational number.

Exercise 1.6.7. Find a necessary and sufficient condition on the real number $b > 1$ such that

$$\lfloor \log_b x \rfloor \;=\; \lfloor \log_b \lfloor x \rfloor \rfloor$$

holds for all real numbers $x \geq 1$.

Exercise 1.6.8. Suppose $0 < \alpha < \beta$ and $0 < x$ are real numbers. Find a closed formula for the sum of all integer multiples of x in the closed interval $[\alpha, \beta]$.

Exercise 1.6.9. How many of the numbers 2^m, for $0 \leq m \leq M$ (where $m, M \in \mathbb{N}$), have leading digit 1 when written in decimal notation? Your answer should be a closed formula.

Exercise 1.6.10. Suppose $x, y, z \in \mathbb{Z}$ are such that $y, z \geq 1$. Prove that $z(x \bmod y) = (zx) \bmod (zy)$.

Exercise 1.6.11. Suppose $a, b, r, s \in \mathbb{Z}$ are such that $r, s \geq 1$. Prove that if $a \bmod rs = b \bmod rs$, then $a \bmod r = b \bmod r$ and $a \bmod s = b \bmod s$.

Exercise 1.6.12. Suppose $b > 1$. Express $\log_b \log_b x$ in terms of $\ln \ln x$, $\ln \ln b$, and $\ln b$.

Exercise 1.6.13 (Programming exercise, also doable by hand)**.** If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000000.

Exercise 1.6.14 (Programming exercise)**.** Let $F_0, F_1, F_2, \ldots$ be the sequence of Fibonacci numbers. Compute the following sum:

$$\sum_{\substack{F_n < 10^7 \\ F_n \bmod 2 = 0}} F_n$$

Exercise 1.6.15 (Programming exercise)**.** Determine the following number:

$$\min\left\{n \in \mathbb{N} : n \geq 1 \text{ and for each } k \in \{1, 2, \ldots, 30\}, k | n\right\}$$

Note: the above number certainly exists since the above set contains the number 30!, so it is non-empty and thus has a minimum element by the Well-Ordering Principle.

Exercise 1.6.16 (Programming exercise)**.** Let P_n denote the nth prime number. So $P_1 = 2, P_2 = 3, P_3 = 5, P_4 = 7, \ldots$. Find P_{100000}.

Exercise 1.6.17 (Programming exercise)**.** A unit fraction contains 1 in the numerator. The decimal representation of the unit fractions with denominators 2 to 10 are given:

$$1/2 = 0.5, \quad 1/3 = 0.(3), \quad 1/4 = 0.25, \quad 1/5 = 0.2, \quad 1/6 = 0.1(6),$$
$$1/7 = 0.(142857), \quad 1/8 = 0.125, \quad 1/9 = 0.(1), \quad 1/10 = 0.1$$

where $0.1(6)$ means $0.166666...$, and has a 1-digit recurring cycle. It can be seen that $1/7$ has a 6-digit recurring cycle. Find the value of $d < 3000$ for which $1/d$ contains the longest recurring cycle in its decimal fraction part. Hint: first analyze by hand what you would have to do to notice that $1/7$ has a 6-digit recurring cycle, think about it in terms of the Division Algorithm.

Exercise 1.6.18 (Programming Exercise)**.** Recall the nth triangular number is given by $T_n = n(n+1)/2$, so the first few triangular numbers are

$$1,\ 3,\ 6,\ 10,\ 15,\ 21,\ 28,\ 36,\ 45,\ 55,\ \ldots$$

We can list the divisors of the first seven triangular numbers:

$$\begin{aligned} 1 &: 1 \\ 3 &: 1, 3 \\ 6 &: 1, 2, 3, 6 \\ 10 &: 1, 2, 5, 10 \\ 15 &: 1, 3, 5, 15 \\ 21 &: 1, 3, 7, 21 \\ 28 &: 1, 2, 4, 7, 14, 28 \end{aligned}$$

We see that 28 is the first triangular number to have more than five divisors. What is the value of the first triangular number to have over a thousand divisors?

Exercise 1.6.19 (Programming exercise)**.** The following iterative sequence is defined for the set of positive integers:

$$\begin{aligned} n &\to n/2 \quad (\text{if } n \text{ is even}) \\ n &\to 3n+1 \quad (\text{if } n \text{ is odd}) \end{aligned}$$

Using the rule above and starting with 13, we generate the following sequence:

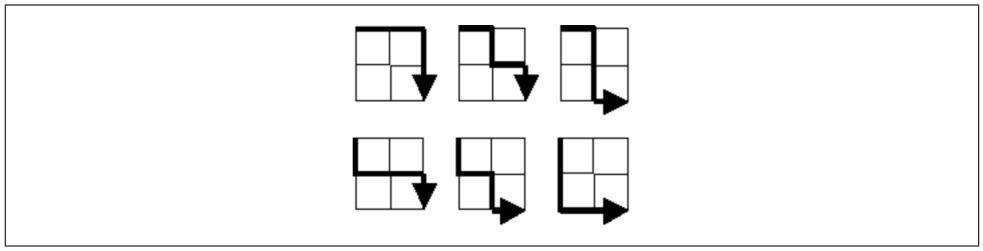
$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

It can be seen that this sequence contains 10 terms. It is conjectured that all starting numbers will finish at 1. Which starting number, under two million, produces the longest chain? [Note: once the chain starts the terms are allowed to go above two million.]

Exercise 1.6.20 (Programming exercise)**.** Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner. How many such routes are there through a 30×30 grid?

Exercise 1.6.21 (Programming exercise)**.** In the United Kingdom the currency is made up of pound (£) and pence (p). There are eight coins in general circulation:

$$1p,\ 2p,\ 5p,\ 10p,\ 20p,\ 50p,\ £1\ (100p),\ \text{and}\ £2\ (200p)$$



It is possible to make £2 in the following way:

$$1 \times £1 + 1 \times 50p + 2 \times 20p + 1 \times 5p + 1 \times 2p + 3 \times 1p$$

How many different ways can £2 be made using any number of coins? Here we don't care about the order of the coins, just how many of each of them there are.

Exercise 1.6.22 (Programming exercise)**.** An irrational decimal is created by concatenating the positive integers:

$$0.12345678910111213141516171819 20\ldots$$

It can be seen that the 12th digit of the decimal expansion is 1.

If d_n represents the nth digit of the fractional part, find the value of the following expression:

$$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$$

CHAPTER 2

Asymptotics

Asymptotics is the study of how functions grow as you take a limit (say, to infinity). As an example of an asymptotic argument, recall that in calculus you might immediately deduce upon inspection that the following limit is 0:

$$\lim_{n\to\infty} \frac{2n+1}{4n^3+5n+2} \;=\; 0.$$

The reasoning would be because both the numerator and denominator are polynomials, and the polynomial in the denominator has higher degree. You can compute this limit without doing any calculations at all if you just make these simple observations (in addition to knowing the rule for how the differing degrees of the polynomials in a rational function determines the limit). As far as this limit is concerned, the only thing you need to know about the numerator is that its leading term is "n" and the only thing you need to know about the denominator is that its leading term is "n^3". In essence, we "abstracted away" all the noise (the lower-order terms and the coefficients) and only focused on the aspects of the function that ultimately affect the limit.

In general this is the game you play in asymptotics. In this chapter, we will make these ideas precise.

2.1. Asymptotic notation

In this section we will finally define the notation we saw in the previous chapter which involved Θ, O, and Ω.

Θ-notation. Recall that in Section 1.3 we saw that TRIANGLE(n) had a running time of $(c_2+c_3)n+(c_1+2c_2+c_3+c_4)$ and from this we conclude that its running time was $\Theta(n)$. In this subsection we will formally define what this means.

Suppose $g:\mathbb{N}\to\mathbb{R}$ is a function. We define $\Theta(g(n))$ (pronounced "big theta of g of n" or "theta of g of n") to be the following set of functions:

$$\Theta(g(n)) \;:=\; \big\{f(n):\ \text{there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\big\}$$

In other words, the set $\Theta(g(n))$ is the set of all functions which are eventually "sandwiched" between two constant multiples of $g(n)$. For example:

Example 2.1.1. $2n+1 \in \Theta(n)$.

PROOF. We need to find $c_1, c_2, n_0 > 0$ such that for all $n \ge n_0$, $0 \le c_1 n \le 2n+1 \le c_2 n$. We claim that $c_1 := 1, c_2 := 3$ and $n_0 := 1$ work. Indeed, suppose $n \ge n_0 = 1$, then $n \le 2n \le 2n+1$ and $2n+1 \le 2n+n = 3n$. □

Since functions $f(n) \in \Theta(g(n))$ are eventually bounded above and below by constant multiples of $g(n)$, this means that as $n \to +\infty$ the function $f(n)$ "grows like $g(n)$", up to a constant factor. In this case we say that $g(n)$ is an **asymptotically tight bound** for $f(n)$. Before we go any further, some remarks are in order:

Remark 2.1.2. There are many customary abuses of notation and terminology involving the Θ-notation (and the other notations we'll learn). These abuses make the notation more useful to work with, but you need to be aware of them so that you use the notation correctly.

(1) Although technically $\Theta(g(n))$ denotes a *set* of functions, we will typically use "$=$" instead of "$\in$" to denote set membership. For example, instead of saying $2n + 1 \in \Theta(n)$ we will say $2n + 1 = \Theta(n)$. We interpret this notation as "$2n + 1$ *is* some function in $\Theta(n)$". Later if we use $\Theta(g(n))$ in an equation or inequality, then $\Theta(g(n))$ will serve as a placeholder for "some unknown or anonymous function in $\Theta(g(n))$".
(2) Although our definition was only for functions $\mathbb{N} \to \mathbb{Z}$ (with domain $\mathbb{N}$), we will often use the notation for functions which are *eventually* defined. For instance, we might talk about $\Theta(\lg \lg n)$, even though $\lg \lg n$ is not defined at $n = 0, 1$.
(3) We will also use the notation $\Theta(g(n))$ for functions $g : \mathbb{R} \to \mathbb{R}$, or even functions $g : D \to \mathbb{R}$, where $D \subseteq \mathbb{R}$ is such that $(a, +\infty) \subseteq D$ for some $a \in \mathbb{R}$. I.e., we will also use this notation for functions whose domain is some subset of the real numbers which is eventually defined for all sufficiently large real numbers. Sometimes, we might talk about $\Theta(\lg n)$ and not even specify if we mean $\lg n$ as a function with domain a subset of $\mathbb{N}$ or $\lg n$ as a function with a domain a subset of $\mathbb{R}$.

Asymptotic notation, like the Θ-notation only really works when the functions involved are asymptotically well-behaved. The functions we consider will be, since they are the running times of algorithms which are in general increasing functions (programs with bigger input typically take longer to run) and only take positive values. Sometimes we will wish to make precise the types of functions we are talking about, in which case the following definition is useful:

Definition 2.1.3. We say that a function $f(n)$ is **asymptotically nonnegative** if there exists an n_0 such that for all $n \geq n_0$, $f(n) \geq 0$. Likewise, we say that $f(n)$ is **asymptotically positive** if there exists an n_0 such that for all $n \geq n_0$, $f(n) > 0$.

It follows that if $g(n)$ is asymptotically nonnegative (respectively, asymptotically positive), then so is every function in $\Theta(g(n))$. Here is a useful fact for determining membership in $\Theta(g(n))$:

Fact 2.1.4. Suppose $g(n)$ is asymptotically positive. Given a function $f(n)$, if

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

exists and is a positive real number (not $+\infty$), then $f(n)$ is also asymptotically positive and $f(n) = \Theta(g(n))$.

Of course, Fact 2.1.4 does not always work. For instance, $2 + \sin n = \Theta(1)$, however $\lim_{n\to+\infty} 2 + \sin n$ does not exist.

O-notation. If Θ-notation is like a sandwich, then O-notation is the top part of the sandwich. By this we mean that O-notation provides an **asymptotic upper bound** for the growth of functions, not a lower bound. Given a function $g(n)$, we denote by $O(g(n))$ the set

$$O(g(n)) := \big\{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\big\}$$

Here, "$O(g(n))$" is pronounced "big-oh of g of n" or "oh of g of n". Just like with Θ-notation, it is customary to write "$f(n) = O(g(n))$" when we really mean "$f(n) \in O(g(n))$". It follows immediately that if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$. This is because Θ-notation provides us with more information (both an upper bound and lower bound) than the O-notation. In general the converse is *not* true, as the following example shows.

Example 2.1.5. $n = O(n^2)$, but $n \neq \Theta(n^2)$.

Proof. To show that $n = O(n^2)$, we need to find $c > 0$ and n_0 such that $0 \leq n \leq cn^2$ for every $n \geq n_0$. In this case $c := 1$ and $n_0 := 1$ work. To show $n \neq \Theta(n^2)$, it suffices to show that for every $c_1 > 0$ and every n_0, there is $n \geq n_0$ such that $n < c_1 n^2$. Let $c_1 > 0$ and n_0 be arbitrary. Consider $n := \max(n_0, \lfloor 1/c_1 + 1 \rfloor)$. Then $1/c_1 < n$, so $n < c_1 n^2$ in addition to $n \geq n_0$. □

Here is a useful fact for determining membership in $O(g(n))$:

Fact 2.1.6. Suppose $g(n)$ is asymptotically positive. Given an asymptotically nonnegative function $f(n)$, if

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

exists and is a nonnegative real number (not $+\infty$), then $f(n) = O(g(n))$.

Ω-notation. The Ω-notation expresses an **asymptotic lower bound** for a function. Given a function $g(n)$ we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or "omega of g of n") to be the set of functions

$$\Omega(g(n)) := \big\{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\big\}$$

We also write "$f(n) = \Omega(g(n))$" when we mean "$f(n) \in \Omega(g(n))$". It follows that if $f(n) = \Theta(g(n))$, then $f(n) = \Omega(g(n))$, since Θ-notation gives more information. However, the converse is not in general true:

Example 2.1.7. $2^n = \Omega(n)$, but $2^n \neq \Theta(n)$.

Proof. To show $2^n = \Omega(n)$, we need to find positive constants c, n_0 such that for all $n \geq n_0$ we have $0 \leq cn \leq f(n)$. We claim that $c := 1$ and $n_0 := 1$ work, i.e., we claim that for every $n \geq 1$, we have $n \leq 2^n$. This can be proved easily by induction (proof omitted).

Next, to show $2^n \neq \Theta(n)$, we need to show for every $c_2 > 0$ and every $n_0 > 0$, there exists $n \geq n_0$ such that $c_2 n < 2^n$. Let $c_2 > 0$ and $n_0 > 0$ be given. Note that $\lim_{n\to\infty} \frac{c_2 n}{2^n} = 0$. So there is some sufficiently large $n \geq n_0$ such that $c_2 n/2^n < 1$, i.e., $c_2 n < 2^n$. □

Here is a useful fact for determining membership in $\Omega(g(n))$:

Fact 2.1.8. Suppose $g(n)$ is asymptotically positive. Given a function $f(n)$, if

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

exists and is a positive real number or $+\infty$, then $f(n) = \Omega(g(n))$.

We also have the following relationship between the three notations:

Theorem 2.1.9. *For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

***o*-notation.** When we write $n = O(n^2)$, we are expressing that n^2 is an asymptotic upper bound for n. We are *not* expressing that n^2 is an asymptotically tight upper bound however. Indeed, $n \neq \Omega(n^2)$. To convey an upper bound that is not asymptotically tight, we employ "little-oh" notation. Given a function $g(n)$, we denote by $o(g(n))$ the set

$$o(g(n)) := \big\{f(n) : \text{for every positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\big\}$$

For example, $n = o(n^2)$, $n^2 = o(n^3)$, $1 = o(\lg n)$, etc. In general, if $f(n) = o(g(n))$, then $f(n) = O(g(n))$, but the converse is not true. This is because o-notation conveys more information than O-notation, namely, if $f(n) = o(g(n))$, then this means $g(n)$ is an asymptotic upper bound which is *not* asymptotically tight, whereas $f(n) = O(g(n))$ just means $g(n)$ is an asymptotic upper bound (which may or may not be asymptotically tight).

Here is a useful fact for determining membership in $o(g(n))$.

Fact 2.1.10. Suppose $g(n)$ is asymptotically positive. Given an asymptotically nonnegative function $f(n)$, if

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

then $f(n) = o(g(n))$.

***ω*-notation.** ω-notation is like o-notation, except for asymptotic lower bounds. That is, ω-notation is used to denote an asymptotic lower bound that is not asymptotically tight. Given a function $g(n)$, we define the set

$$\omega(g(n)) := \big\{f(n) : \text{for every positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\big\}$$

For instance, $n^2 = \omega(n)$, but $n^2 \neq \omega(n^2)$. o-notation and ω-notation are related as follows:

Fact 2.1.11. $f(n) = \omega(g(n))$ iff $g(n) = o(f(n))$.

The following is useful for determining membership in $\omega(g(n))$:

Fact 2.1.12. Suppose $g(n)$ is asymptotically positive. Given a function $f(n)$, if

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

then $f(n) = \omega(g(n))$.

2.2. Properties of asymptotic notation

In this section we discuss further facts about the asymptotic notation introduced in the last section.

Asymptotic notation in equations and inequalities. We have introduced the abuse of notation $n = O(n^2)$, where we treat a set membership like an equality. This abuse can be stretched a bit further as we can include asymptotic notation in more complicated equations and inequalities. For instance, we can write:

$$2n^2 + 3n + 1 \;=\; 2n^2 + \Theta(n)$$

How do we interpret this expression? Here we should think of "$\Theta(n)$" in the equation as standing for some anonymous function $f(n)$ such that $f(n) = \Theta(n)$. In this case, the anonymous function is $3n+1$, and indeed $3n+1 = \Theta(n)$. Comparing the lefthand side and righthand side of this equation, we can think of the righthand side as providing a more vague description of the function on the lefthand side. For instance, you can think of this as:

$$2n^2 + (\text{the specific linear function } 3n+1) \;=\; 2n^2 + (\text{some linear function})$$

From an asymptotic point of view, the advantage here is that the function $2n^2 + 3n+1$ is dominated by the term $2n^2$ as n gets very large, so collapsing the $3n+1$ into just $\Theta(n)$ allows us to suppress the lower-order terms which aren't very relevant for answering most asymptotic questions. Here, we only care that $2n^2 + 3n + 1$ is quadratic, so the $\Theta(n)$ hides the terms $3n + 1$ which would otherwise just be a distraction.

Remark 2.2.1. When using asymptotic notation in equations and inequalities, you need to be aware that the equality relation "=" is **not** symmetric. For instance, it would be incorrect to say that

$$2n^2 + \Theta(n) \;=\; 2n^2 + 3n + 1.$$

Here the righthand side contains more information than the lefthand side, which is not how asymptotic notation works. In general, going from left to right, you are only allowed to lose information, not gain information. We can summarize this in the following rule: *the righthand side of an asymptotic equation is never allowed to contain more information than the lefthand side of an equation.* This might seem strange, since everywhere else in mathematics, the equality relation "=" is symmetric (as well as reflexive and transitive). The reason there is no contradiction here is that "=" does not literally mean equality in this context. Really, when we write

$$2n^2 + 3n + 1 \;=\; 2n^2 + \Theta(n)$$

this is just logical shorthand for the more complicated expression:

$$\text{there exists a function } f(n) \in \Theta(n) \text{ such that } 2n^2 + 3n + 1 = 2n^2 + f(n)$$

which itself is logical shorthand for the even more complicated expression:

> there exists a function $f(n)$ and positive constants c_1, c_2, n_0 such that for every $n \geq n_0$, $0 \leq c_1 n \leq f(n) \leq c_2 n$, and $2n^2 + 3n + 1 = 2n^2 + f(n)$

As you can see, the Θ-notation allows us to express this complicated idea in a very compact form. This is the true power of asymptotic notation and you should keep in mind that expressions involving asymptotic notation are really just shorthand for more complicated expressions which you wouldn't want to be writing all of the time.

We can also use asymptotic notation on both sides of an equation. For instance

$$2n^2 + O(n) \;=\; \Theta(n^2)$$

This expression is interpreted by the following rule: *No matter how the anonymous function(s) is chosen on the lefthand side, there is a way to choose the anonymous function(s) on the righthand side to make the equation true.* For instance, if $O(n)$ on the left stands for $\lg n$, then we can choose $2n^2 + \lg n$ to be the function on the righthand side (which indeed is in $\Theta(n^2)$). The equation

$$2n^2 + O(n) \;=\; \Theta(n^2)$$

expresses something rather meaningful. Namely:

> Functions of the form $2n^2$ + (something at most linear) always grow asymptotically like n^2.

This one single fact encompasses many different situations you might encounter, and the asymptotic notation efficiently expresses this idea.

Finally, we can chain together multiple equations involving asymptotic notation, like so:

$$\begin{aligned} 2n^2 + 3n + 1 \;&=\; 2n^2 + \Theta(n) \\ &=\; \Theta(n^2) \end{aligned}$$

This is interpreted as:

$$2n^2 + 3n + 1 \;=\; 2n^2 + \Theta(n) \quad \text{and} \quad 2n^2 + \Theta(n) \;=\; \Theta(n^2)$$

where each equation separately gets interpreted by the above rules. Notice how going from left to right we are slowly losing information: we start out with something specific, $2n^2 + 3n + 1$, and at the end we are left with only the asymptotic essence $\Theta(n^2)$ of what we began with.

Comparing functions. Asymptotic notation is transitive:

Fact 2.2.2. Suppose $f(n)$ and $g(n)$ are asymptotically positive. Then

(1) If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$
(2) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
(3) If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$
(4) If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$
(5) If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$

The "big" notations are reflexive:

Fact 2.2.3. Suppose $f(n)$ is asymptotically nonnegative. Then

(1) $f(n) = \Theta(f(n))$
(2) $f(n) = O(f(n))$
(3) $f(n) = \Omega(f(n))$

The Θ-notation is also symmetric:

Fact 2.2.4. $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

The oh- and omega- notations are not symmetric, but they are related as follows:

Fact 2.2.5. Suppose $f(n)$ and $g(n)$ are asymptotically positive. Then

(1) $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
(2) $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Asymptotics of common functions. We now establish some more specific facts about the asymptotics of some of the types of functions we'll care about. First recall that a **polynomial in n of degree d** is a function $p(n)$ of the form

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

where $a_0, \ldots, a_d \in \mathbb{R}$ and $a_d \neq 0$. The polynomial $p(n)$ is asymptotically positive iff $a_d > 0$. In this case, we have $p(n) = \Theta(n^d)$, i.e., $p(n)$ grows asymptotically like the monomial n^d of its leading term. Given any function $f(n)$, we say that $f(n)$ is **polynomially bounded** if $f(n) = O(n^k)$ for some constant k. Of course, polynomials themselves are polynomially bounded, but other functions, like $n \lg n$ are as well.

Suppose $a, b \in \mathbb{R}$ are such that $a > 1$. Then we have the following limit:

$$\lim_{n\to\infty} \frac{n^b}{a^n} \;=\; 0$$

which shows that $n^b = o(a^n)$, i.e., *any* exponential will dominate *every* polynomial (provided the base of the exponential is > 1). In this limit, if we perform the substitution $\lg n$ for n and 2^a for a, we get

$$\lim_{n\to\infty} \frac{\lg^b n}{n^a} \;=\; 0$$

which shows that $\lg^b n = o(n^a)$. Thus *any* polynomial will grow faster than every power of a logarithm. We call a function $f(n)$ **polylogarithmically bounded** if $f(n) = O(\lg^k n)$ for some constant k.

Another function whose asymptotics we will be interested in is the **factorial**. We define the factorial of n, for $n \geq 0$ by the following recurrence:

$$n! \;=\; \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

In other words, $n! = 1 \cdot 2 \cdot 3 \cdots n$ for $n \geq 1$. The factorial is a very fast growing function. However, its definition is discrete by nature so it is not obvious how to compare it to, say, an exponential function e^n. In order to apply continuous analytical methods to analyzing the factorial, we use **Stirling's approximation**:

$$n! \;:=\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

This says that $n! = \Theta(\sqrt{n}(n/e)^n)$, which grows faster than any exponential. Here are some more bounds involving the factorial:

$$\begin{aligned} n! &= o(n^n) \\ n! &= \omega(2^n) \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

2.3. The Euclidean Algorithm

When simplifying fractions a/b, we want to know what is the best divisor which we can factor out of both a and b simultaneously, to then cancel out. For instance,

$$\frac{15}{21} = \frac{3 \cdot 5}{3 \cdot 7} = \frac{5}{7}.$$

The divisor we are looking for is called the *greatest common divisor* of a and b. In order to make this definition precise, first we need to observe the following:

Lemma 2.3.1. *Suppose $a, b \in \mathbb{Z}$ are such that either $a \neq 0$ or $b \neq 0$, i.e., suppose $(a, b) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$. Then the set of common divisors*

$$\mathrm{CD}(a, b) := \{d \in \mathbb{N} : d|a \text{ and } d|b\}$$

has the properties:

(1) $1 \in \mathrm{CD}(a, b)$ (so $\mathrm{CD}(a, b)$ is nonempty), and
(2) for every $d \in \mathrm{CD}(a, b)$, $d \leq \max(|a|, |b|)$ (so $\mathrm{CD}(a, b)$ is bounded above).

Thus the set $\mathrm{CD}(a, b)$ has a largest element.

Proof. (1) follows from 1.4.8(D1) and (2) follows from (D6). □

Definition 2.3.2. Define the **greatest common divisor** function of two integers to be the function

$$(a, b) \mapsto \gcd(a, b) : \mathbb{Z}^2 \setminus \{(0, 0)\} \to \mathbb{N}$$

defined by

$$\gcd(a, b) := \max \mathrm{CD}(a, b).$$

In other words, $\gcd(a, b) = d$ means:

(1) $d|a$ and $d|b$, in symbols:

$$d|a \wedge d|b$$

(2) for all $e \in \mathbb{N}$, if $e|a$ and $e|b$, then $e \leq d$, in symbols:

$$\forall e \ (e|a \wedge e|b \to e \leq d)$$

[Note: (1) asserts that d is a common divisor, and (2) asserts that d is greater than or equal to all other common divisors, taken together this means that d is the greatest common divisor.]

By convention, we extend the definition of gcd to all of $\mathbb{Z}^2$ by declaring $\gcd(0, 0) := 0$.

For small integers, the gcd is easy to compute by brute-force:

Example 2.3.3. Suppose $a = 24$ and $b = 45$. Then the set of positive divisors are respectively

$$\{1, 2, 3, 4, 6, 8, 12, 24\} \quad \text{and} \quad \{1, 3, 5, 9, 15, 45\}.$$

Thus the common divisors are

$$\mathrm{CD}(24, 45) \;=\; \{1, 3\}$$

and so $\gcd(24, 45) = 3$.

This suggests an algorithm for computing the gcd (for simplicity, we assume that the input $a, b \in \mathbb{N}$ is nonnegative):

NaiveGCD(a, b)

```
1  if a == 0
2        return b
3  if b == 0
4        return a
5  BestDivisor = 1
6  for d = 1 to min(a, b)
7        if a mod d == 0 and b mod d == 0
8              BestDivisor = d
9  return BestDivisor
```

To prove correctness of NaiveGCD, one would use the following loop invariant:

> (Loop Invariant) Each time line 6 has immediately run, the value of *BestDivisor* is
>
> $$\max\left\{e \in \{1, \ldots, \max(1, e-1)\} : e|a \text{ and } e|b\right\}$$

Furthermore, if $a, b \neq 0$, then the running time of NaiveGCD is $\Theta(\min(a, b))$, i.e., it is linear in the smaller argument. Is there any way to improve upon this running time? Fortunately, the answer is *yes*. For this we need the following facts about gcd:

Lemma 2.3.4. *Suppose $a, b \in \mathbb{Z}$ are such that $(a, b) \neq (0, 0)$. Then:*

(1) (Symmetry) $\gcd(a, b) = \gcd(b, a)$
(2) (Reduction) if $b \neq 0$, then $\gcd(a, b) = \gcd(b, a \bmod b)$
(3) (Stopping Conditions) $\gcd(a, 1) = 1$ *and* $\gcd(a, 0) = |a|$ *if $a \neq 0$.*

Proof. (1) Since $\mathrm{CD}(a, b) = \mathrm{CD}(b, a)$, it follows that

$$\gcd(a, b) \;=\; \max \mathrm{CD}(a, b) \;=\; \max \mathrm{CD}(b, a) \;=\; \gcd(b, a).$$

(2) Let $q \in \mathbb{Z}$ be such that $a = qb + a \bmod b$ and suppose $d \in \mathrm{CD}(a, b)$ is arbitrary. Then $d|a - qb = a \bmod b$, so $d \in \mathrm{CD}(b, a \bmod b)$. Conversely, if $d \in \mathrm{CD}(b, a \bmod b)$, then $d|qb + a \bmod b = a$, so $d \in \mathrm{CD}(a, b)$. Thus $\mathrm{CD}(a, b) = \mathrm{CD}(a, b \bmod a)$, and so $\gcd(a, b) = \gcd(a, a \bmod b)$.

(3) The only positive divisor of 1 is 1, so $\mathrm{CD}(a, 1) = \{1\}$ and $\gcd(a, 1) = 1$. Since $|a||a$ and $|a||0$, and $\gcd(a, 0) \leq \max\{|a|, |0|\} = |a|$, it follows that $\gcd(a, 0) = |a|$. □

The *Reduction* property tells us that the gcd of (a, b) is the same as the gcd of $(b, a \bmod b)$. Since $0 \leq a \bmod b < b$, at least one of the two arguments is smaller. This suggests the following recursive algorithm (we also assume the input $a, b \in \mathbb{N}$ is nonnegative):

EUCLID(a, b)
1 **if** $b == 0$
2 **return** a
3 **else return** EUCLID$(b, a \bmod b)$

Theorem 2.3.5. *For every $a, b \in \mathbb{N}$,* EUCLID(a, b) *returns* $\gcd(a, b)$.

PROOF. We will prove this by induction on the following statement:

$$P(n): \quad \text{"for every } b \in \mathbb{N}, \text{ if } b \leq n, \text{ then for every } a \in \mathbb{N},$$
$$\text{EUCLID}(a, b) \text{ returns } \gcd(a, b)\text{"}$$

(Base Case) Suppose $n = 0$. Then $b = 0$. Let $a \in \mathbb{N}$ be arbitrary. Then the condition "$b == 0$" in line 1 is true, so we go to line 2 and return a. By Lemma 2.3.4(3) (and the convention $\gcd(0, 0) = a$), it follows that $a = \gcd(a, b)$.

(Inductive Step) Suppose for some $n \geq 0$ we know that $P(n)$ is true, i.e., for every $a, b \in \mathbb{N}$ with $b \leq n$ we know EUCLID(a, b) returns $\gcd(a, b)$. Now we will show that $P(n + 1)$ is true. Suppose $b \leq n + 1$ and $a \in \mathbb{N}$ is arbitrary. If $b \leq n$, then we are done by $P(n)$, so suppose $b = n + 1 \geq 1$. Then the condition "$b == 0$" in line 1 is false, so we go to line 3. In line 3 we return EUCLID$(b, a \bmod b)$. Since $0 \leq a \bmod b < b = n + 1$, we know by $P(n)$ that EUCLID$(b, a \bmod b)$ returns $\gcd(b, a \bmod b)$. However, by *Reduction* we also know that $\gcd(a, b) = \gcd(b, a \bmod b)$. Thus EUCLID$(a, b)$ returns $\gcd(a, b)$. This finishes the inductive step.

By the Principle of Induction we conclude that $P(n)$ is true for all $n \in \mathbb{N}$. Thus the Theorem is proved. □

To analyze the running time of EUCLID, we need some information about how many recursive calls it makes. This analysis relies on the Fibonacci numbers in an essential way.

Lemma 2.3.6. *If $a > b \geq 1$ and the call* EUCLID(a, b) *performs $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.*

PROOF. We will prove the following statement by induction on $k \geq 1$:

$$P(k): \quad \text{"If } a > b \geq 1 \text{ and EUCLID}(a, b) \text{ performs } k \text{ recursive calls,}$$
$$\text{then } a \geq F_{k+2} \text{ and } b \geq F_{k+1}\text{"}$$

(Base Case) Suppose $k = 1$, i.e., we made 1 recursive call to EUCLID. This means that the test in line 1 failed, so $b \geq 1 = F_2$ and since $a > b$ we have $a \geq 2 = F_3$.

(Inductive Step) Suppose for some $k \geq 1$ we know that $P(k)$ is true, i.e., for every $a > b \geq 1$, if EUCLID made k recursive calls then $a \geq F_{k+2}$ and $b \geq F_{k+1}$. We will use this to prove $P(k + 1)$. Assume $a > b \geq 1$ is arbitrary such that EUCLID made $k + 1$ recursive calls. Since $k + 1 \geq 2$, when we run EUCLID(a, b), we make at least 2 recursive calls. In particular, the condition in line 1 fails so we go to line 3 and call EUCLID$(b, a \bmod b)$, our first recursive call. This means that calling EUCLID$(b, a \bmod b)$ results in $k \geq 1$ recursive calls. Since $1 \leq a \bmod b < b$, by $P(k)$, it follows that $b \geq F_{k+2}$ and $a \bmod b \geq F_{k+1}$. We now must show that $a \geq F_{k+3}$. First note that

$$\begin{aligned} b + (a \bmod b) &= b + (a - b \lfloor a/b \rfloor) \\ &\leq a \end{aligned}$$

since $a > b > 0$ implies $\lfloor a/b \rfloor \geq 1$. Thus

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+2} + F_{k+1} \\ &= F_{k+3}. \end{aligned}$$

This concludes the inductive step.

By the Principle of Induction we know that $P(k)$ is true for every $k \geq 1$. Thus the lemma is proved. □

The following is a consequence of Lemma 2.3.6:

Lamé's Theorem 2.3.7. *For any $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call* Euclid(a, b) *makes fewer than k recursive calls.*

This gives us an upper bound on the running time. To show that it is the best-possible upper bound, we need the following:

Lemma 2.3.8. *For $k \geq 2$, the call* Euclid(F_{k+1}, F_k) *makes exactly $k-1$ recursive calls.*

Proof. We will prove the following by induction on $k \geq 2$:

$$P(k): \quad \text{“Euclid}(F_{k+1}, F_k) \text{ makes exactly } k-1 \text{ recursive calls.”}$$

(Base Case) Suppose $k = 2$. The call Euclid(F_3, F_2) = Euclid$(2, 1)$ makes a recursive call to Euclid$(1, 0)$, and then stops. Thus we make $2 - 1 = 1$ recursive calls in this case.

(Inductive Step) Suppose for some $k \geq 2$ we know that Euclid(F_{k+1}, F_k) makes $k-1$ recursive calls (i.e., suppose $P(k)$ is true). Now we will consider what happens when we call Euclid(F_{k+2}, F_{k+1}). Since $F_{k+2} = F_{k+1} + F_k$, and $0 \leq F_k < F_{k+1}$, we have $F_{k+2} \bmod F_{k+1} = F_k$. When we run Euclid$(F_{k+2}, F_{k+1})$, we make our first recursive call to Euclid$(F_{k+1}, F_{k+2} \bmod F_{k+1})$, i.e., a call to Euclid$(F_{k+1}, F_k)$. By $P(k)$, then we make $k-1$ further recursive calls. Thus we make $k = (k+1) - 1$ recursive calls total. This proves $P(k+1)$.

By the Principle of Induction, we know that $P(k)$ is true for all $k \geq 2$, thus the lemma is proved. □

Theorem 2.3.9. *The running time of* Euclid(a, b) *is $O(\lg b)$.*

Proof. Let $a, b \in \mathbb{N}$. We may assume that $b \geq 1$, and to make the running time longer, we may add multiples of b to a to assume $a > b \geq 1$. We need to find the smallest value of k such that $b < F_{k+1}$. Since $F_{k+1} \approx \phi^{k+1}/\sqrt{5}$, we should find the value of k (as a real number) such that $b = \phi^{k+1}/\sqrt{5}$. Solving for k, we see that

$$k \;=\; \log_\phi(b/\sqrt{5}) - 1 \;=\; \log_\phi b - \frac{1}{2}\log_\phi 5 - 1$$

The true integer value of k will be close to this (for instance, take the ceiling). Since everything done in lines 1-2 takes a constant amount of work, and compute $a \bmod b$ also takes a constant amount of work, the running time is proportional to the number of recursive calls we make. If we denote the running time by $T(a, b)$, then this shows that

$$T(a, b) \;=\; O(\log_\phi b) \;=\; O(\lg b). \qquad \square$$

Thus the worst-case running time of Euclid is logarithmic, which is a major improvement over the linear running time of NaiveGCD. This analysis also gives us a new appreciation for the Fibonacci sequence: it suggests that the sequence of Fibonacci numbers are intrinsically related to the Euclidean Algorithm.

2.4. Exercises

Exercise 2.4.1. Suppose $f(n)$ and $g(n)$ are asymptotically positive functions. Prove that $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Exercise 2.4.2. Prove that for all $m \in \mathbb{N}$, $(\ln n)^m = o(n)$.

Exercise 2.4.3. Let

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

where $a_d > 0$, be a degree d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties:

(1) If $k \geq d$, then $p(n) = O(n^k)$.
(2) If $k \leq d$, then $p(n) = \Omega(n^k)$.
(3) If $k = d$, then $p(n) = \Theta(n^k)$.
(4) If $k > d$, then $p(n) = o(n^k)$.
(5) If $k < d$, then $p(n) = \omega(n^k)$.

Exercise 2.4.4. Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

(1) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
(2) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.
(3) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for sufficiently large n.
(4) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
(5) $f(n) = O((f(n))^2)$.
(6) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.
(7) $f(n) = \Theta(f(n/2))$.
(8) $f(n) + o(f(n)) = \Theta(f(n))$.

Exercise 2.4.5. Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?

(1) n^2
(2) n^3
(3) $100n^2$
(4) $n \lg n$
(5) 2^n
(6) 2^{2^n}.

Exercise 2.4.6. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$

in your list, then it should be the case that $f(n)$ is $O(g(n))$. Justify all consecutive comparisons.

(1) $g_1(n) = 2^{\sqrt{\lg n}}$
(2) $g_2(n) = 2^n$
(3) $g_3(n) = n^{4/3}$
(4) $g_4(n) = n(\lg n)^3$
(5) $g_5(n) = n^{\lg n}$
(6) $g_6(n) = 2^{2^n}$
(7) $g_7(n) = 2^{n^2}$

Exercise 2.4.7. Let $(F_n)_{n\geq 0}$ be the sequence of Fibonacci numbers, i.e., $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.

(1) Prove that for all $n \geq 0$, $F_n = (\phi^n - \hat{\phi}^n)/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Hint: Use that $\phi, \hat{\phi}$ are both roots of the quadratic polynomial $x^2 - x - 1$.
(2) Let $T(n)$ be the running time of FIBONACCI. Prove that $T(n) = \Theta(F_n)$.
(3) Prove that $F_n = \Theta(\phi^n)$. Conclude that $T(n) = \Theta(\phi^n)$ runs in exponential time.

Exercise 2.4.8. Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the **binary gcd algorithm**, which avoids the remainder computations used in Euclid's algorithm.

(1) Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.
(2) Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.
(3) Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
(4) Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ times. Assume that each subtraction, parity test, and halving takes unit time.

Exercise 2.4.9. Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional $n \times n$ array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$, that is, the sum $A[i] + A[i + 1] + \cdots + A[j]$, (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

Here is a simple algorithm to solve this problem.

```
1  for i = 1 to n
2      for j = i + 1 to n
3          Add up array entries A[i] through A[j]
4          Store the result in B[i, j]
```

(1) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).
(2) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)
(3) Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem – after all, it just iterates through the

relevant entries of the array B, filling in a value for each – it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

CHAPTER 3

Sorting algorithms

In this chapter we study an important and fundamental problem in computer science, the **sorting problem**. Sorting is an interesting problem because it is one that everyone has familiar with, even if they have no interest in computer programs. Indeed, at some point in your life you probably have done one of the following (or something similar):

- In a card game, arranging your hand of cards by suit and then from lowest to highest (or doing the same thing with the entire deck).
- Arranging your bank statements first by year, then by month.
- Alphabetizing books on a shelf by title, or by last name of author.
- Alphabetizing a stack of homework assignments by last name of student.
- etc. [put your own example here]

The sorting problem involves writing an algorithm that can perform these tasks, ideally as efficiently as possible. In its most essential form, the sorting problem can be summarized as:

Input: We are given a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: We want to output a permutation (reoreding) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

In this context, we will refer to the values a_i that we wish to sort as **keys**. In this chapter we will study two common sorting algorithms: Insertion-Sort in Section 3.1 and Merge-Sort in Section 3.2. Finally in Section 3.3 we will prove that the best achievable running time of a comparison-based sorting algorithm is $O(n \lg n)$.

3.1. Insertion sort

The algorithm Insertion-Sort mimics the way human beings might sort a deck of cards or alphabetize a stack of student homework. In terms of the deck-of-cards analogy, it works as follows:

(1) Start with an unsorted deck of cards.
(2) Select the top card to put into your new sorted deck. The new sorted deck now contains one card.
(3) Select the next card from the unsorted deck. Compare it to the card in the sorted deck, and insert it either before or after the first card.
(4) Select the next card (the third card) from the unsorted deck. Compare it to the two cards in the sorted deck and insert it in its correct place.
(5) Continue in this manner until all 52 cards are in the sorted deck.

The following is the pseudocode for Insertion-Sort. Be aware: in this algorithm our starting array index is 1, not 0, so $A[1]$ is the leftmost entry of the array.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

To help us get a feel for how INSERTION-SORT works, in Figure 3.1 we trace through the INSERTION-SORT pseudocode with input $A = \langle 3, 2, 1\rangle$. See also [**1**, Figure 2.2].

Correctness proof. We now proceed with proving correctness of INSERTION-SORT. In this context, the statement we wish to prove is the following:

Theorem 3.1.1. *At the end of* INSERTION-SORT(A)*, the array A consists of the original elements from A, but in sorted order.*

PROOF. In line 1 we immediately run into a **for** loop, so we will need a loop invariant. The way INSERTION-SORT works is that it first makes sure $A[1\,..\,2]$ is sorted, then $A[1\,..\,3]$ is sorted, and so on, until eventually $A[1\,..\,n]$ is sorted. This gives us our loop invariant:

> (Loop Invariant) After each time line 1 is run, the subarray $A[1\,..\,j-1]$ consists of the elements originally in $A[1\,..\,j-1]$, but in sorted order.

(Initialization) Immediately after line 1 is run the first time, the value of j is $j = 2$, and the subarray $A[1\,..\,j-1]$ is just a single element $A[1]$, which is (vacuously) in sorted order and it is also the original element from the subarray $A[1] = A[1\,..\,j-1]$.

(Maintenance) Suppose after line 1 has just run and the current value of j is $j = j_0$ where $2 \le j_0 \le A.length$. Furthermore, suppose we know that the loop invariant is currently true, i.e., the elements of the subarray $A[1\,..\,j_0-1]$ are the original elements from $A[1\,..\,j_0-1]$ except in sorted order. Next the goal is to insert the entry from the original jth spot, $key = A[j]$ into its correct place in the subarray $A[1\,..\,j_0]$. In line 4 we set $i = j_0 - 1$. Then we enter another loop (a **while** loop this time). Technically we should do a second loop-invariant argument here (the book [**1**] waves its hands at this point), but instead we'll just state the loop invariant and then see what termination gets us:

(**while** loop invariant) After each time line 5 runs: The elements of $A[1\,..\,i]$ are in sorted order, the elements of $A[i+1\,..\,j_0]$ are in sorted order. If $i \neq j_0 - 1$, then $A[i+1] = A[i+2]$, and the elements of $A[1\,..\,j_0]$ together with key are the original values of $A[1\,..\,j_0]$.

Suppose we prove this loop invariant is true. Then upon termination either two things happen:

Case 1: Suppose $i = 0$ and $A[i'] > key$ for every $i' \in \{1, \ldots, j_0 - 1\}$. In this case, the loop invariant says that $A[1\,..\,j_0]$ is in sorted order, $A[1] = A[2]$, and $key < A[2]$. Thus line 8 assigns $A[1]$ the value of key, so $A[1\,..\,j_0]$ is still in sorted order and it consists of the elements originally in the array $A[1\,..\,j_0]$.

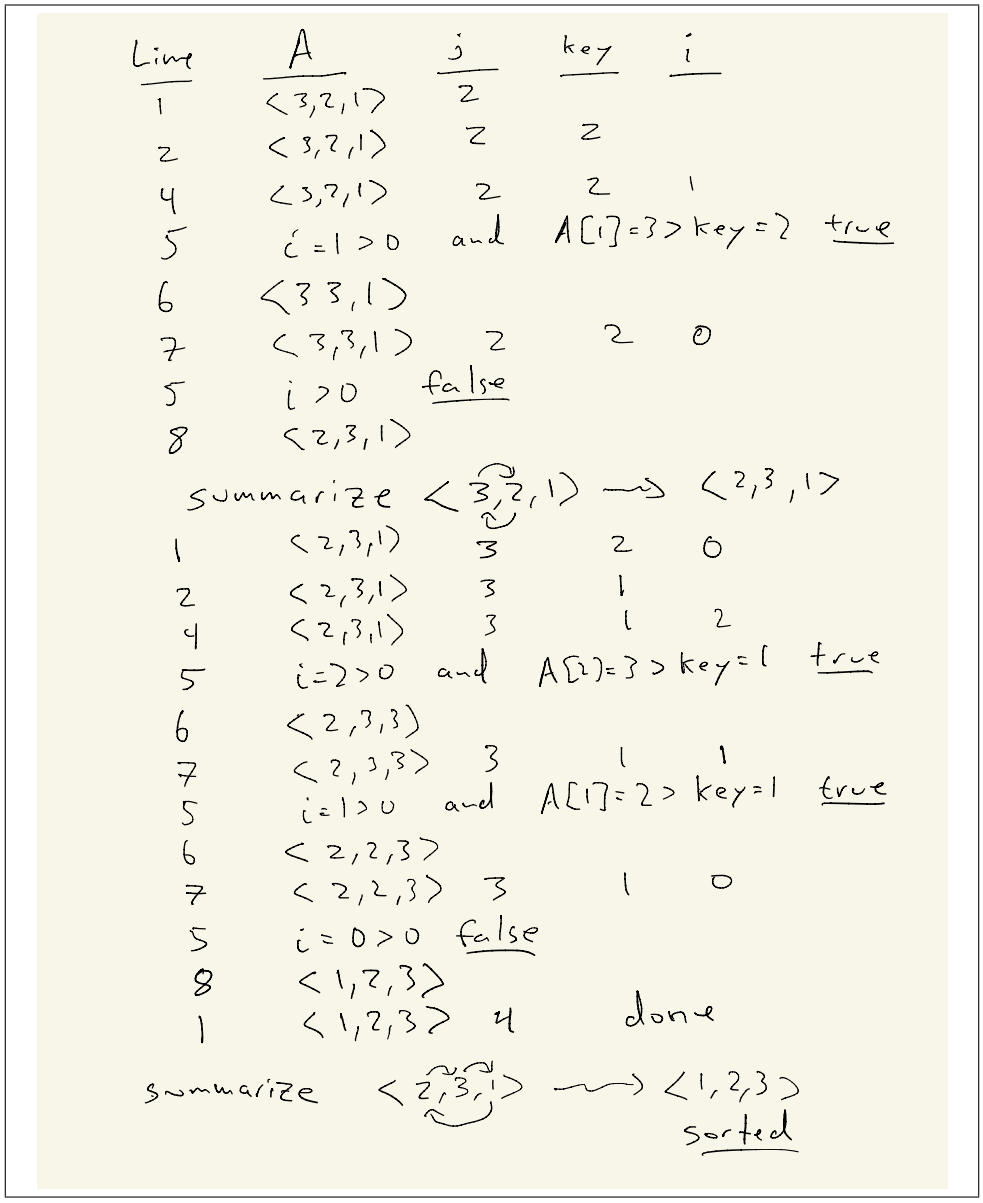


FIGURE 3.1. Here we trace through INSERTION-SORT with input $A = \langle 3, 2, 1 \rangle$. The first part of the code processes the 2 in the second spot, inserting it before the 3 in the first spot, moving 3 over one spot to the right. The next part of the code processes the 1 in the third spot, inserting it before the 2 and the 3, thus moving the 2 and the 3 over one spot to the right each.

Case 2: Suppose $i > 0$ and $A[i] \le key$, but $A[i'] > key$ for every $i' \in \{i + 1, \ldots, j_0 - 1\}$. Then in line 8 we assign $A[i + 1] = key$, so $A[1 \ldots, i + 1]$ is in sorted order, and since $A[i + 1] < A[i + 2]$, the array $A[i + 1 \,..\, j_0]$ is still in sorted

order, so $A[1\,..\,j_0]$ is in sorted order. Furthermore, since before line 8 we have $A[i+1] = A[i+2]$ (if $i \neq j_0 - 1$), and then afterwards $A[i+1] = key$, we have that the current elements of $A[1\,..\,j_0]$ are the original elements from the array $A[1\,..\,j_0]$.

Finally, we return to line 1 and increment j, so $j = j_0 + 1$. Thus the array $A[1\,..\,j-1]$ is in sorted order and consists of the elements which were originally in $A[1\,..\,j-1]$, so the loop invariant is still true.

(Termination) After the last time line 1 is run, the value of j is $j = A.\mathit{length}+1$, so the loop invariant says that $A[1\,..\,A.\mathit{length}] = A$ is in sorted order and consists of the original elements which were in the array. Thus INSERTION-SORT is correct. □

Running time analysis. We now look at the running time costs associated with INSERTION-SORT (suppose $n := A.\mathit{length}$):

INSERTION-SORT(A)

1	**for** $j = 2$ **to** $A.\mathit{length}$	$\mathit{cost} : c_1$ $\mathit{times} : n$
2	$\mathit{key} = A[j]$	$\mathit{cost} : c_2$ $\mathit{times} : n-1$
3	// Insert $A[j]$ into...	
4	$i = j - 1$	$\mathit{cost} : c_3$ $\mathit{times} : n-1$
5	**while** $i > 0$ and $A[i] > \mathit{key}$	$\mathit{cost} : c_4$ $\mathit{times} : \sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	$\mathit{cost} : c_5$ $\mathit{times} : \sum_{j=2}^{n}(t_j - 1)$
7	$i = i - 1$	$\mathit{cost} : c_6$ $\mathit{times} : \sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] = \mathit{key}$	$\mathit{cost} : c_7$ $\mathit{times} : n-1$

Here t_j is some number which satisfies $1 \leq t_j \leq j$, depending on how many times the **while** loop iterates for that value of j (which depends on where we need to place *key* into the sorted subarray $A[1\,..\,j-1]$). The running time $T(n)$ is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ &\quad + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1) \end{aligned}$$

In this situation, it is not clear what type of function this is since it depends on the values of t_j, i.e., it depends on how much the input A is already partially sorted. The best we can do is put bounds on this quantity.

The **best-case running time** occurs when the input is already sorted. In this case the **while** loop never has to run and so $t_j = 1$ for each j. This gives us a running time

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \\ &= an + b \end{aligned}$$

for appropriate constants a, b, with $a > 0$. Thus we can say that the best-case is $\Theta(n)$, and also that overall the running time is $\Omega(n)$. The **worst-case running time** occurs when the input is in the reverse of sorted order, because this will cause the **while** loop to always run a maximum number of times, so $t_j = j$ for each j. In

this case the running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j \\ &\quad + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1) \\ &= an^2 + bn + c \end{aligned}$$

for appropriate constants a, b, c with $a > 0$ (here we use the Triangular Number Formula to get the quadratic terms). Thus we can say that the worst-case is $\Theta(n^2)$ and overall the running time is $O(n^2)$. We summarize this analysis as follows:

Theorem 3.1.2. *The running time of* Insertion-Sort *can be characterized as follows:*

(1) The best-case running time is $\Theta(n)$.
(2) The worst-case running time is $\Theta(n^2)$.
(3) The overall running time is $O(n^2)$ and $\Omega(n)$.

Final remarks. Here are some additional features about Insertion-Sort you should be aware of:

(1) Despite its $O(n^2)$ running time, Insertion-Sort is generally considered a fast algorithm on *small input*, and in practice it does get used in this way.
(2) Insertion-Sort is an **in-place** sorting algorithm. This means that it only uses a constant amount of extra memory (for instance, to store the values of *key*, i and j) beyond what is needed to store A.
(3) Insertion-Sort is a **stable** sorting algorithm, i.e., if two keys have the same value, then in the sorted output they will occur in the same order as they occur in the input.

3.2. Merge sort

The next sorting algorithm we'll study is Merge-Sort. This algorithm has a much better worst-case running time of $\Theta(n \lg n)$. To motivate Merge-Sort, consider the following thought experiment:

> You are the teacher of a large course with 200 students and every students turns in a paper copy of their homework with their last name on it. For various reasons you need to alphabetize the entire stack of homework. You *could* do an insertion sort by hand, but since you are busy you decide you'll do the following:
> (1) Divide the stack of homework into two stacks of 100 homeworks each. You have two TAs in this hypothetical scenario, so you give one stack to each TA and tell them to sort it and bring back to you.
> (2) The TAs return two alphabetized stacks of 100 homework assignments to you. You still need to produce a single alphabetized stack of 200 homeworks, but now it is much easier. You can simply *merge* the two stacks together. I.e., you can have the two stacks in front of you facing up, and

you pick up homeworks one at a time from whichever stack has the alphabetically earlier name on it. This will take you a linear amount of time which is much better than the potentially quadratic amount of time you would have had to do using insertion sort.

We are not done yet. We have explained how *you* sorted the stack of 200, but we haven't explained yet how each TA sorted their stack of 100. They also are busy and decide to also delegate out the sorting. In this hypothetical scenario, each TA has two sub-TA underlings, who they give 50 homeworks each to, to go off and sort. The sub-TAs each bring back a stack of 50 sorted homeworks for the TA to merge together, which they will then return to you, for you to merge.

How do the sub-TAs sort their stack of 50 homeworks? They do the same thing with two sub-sub-TAs, and so on and so forth. Eventually at some level of this recursive delegation chain, someone will be handed a stack of 1 homework assignment and be asked to sort it. For this person, they will just immediately return it to the person that handed it to them and tell them that it is already sorted. This will kickstart the up-the-ladder sequence of merging which eventually stops at you.

This is how MERGE-SORT works.

To implement this method of sorting on a computer, the first thing we need is a really good *merging* subroutine. Here it is:

MERGE(A, p, q, r)

```
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_1 + 1] be new arrays
 4  for i = 1 to n_1
 5       L[i] = A[p + i - 1]
 6       // Copies A[p..q] into L[1..n_1]
 7  for j = 1 to n_1
 8       R[j] = A[q + j]
 9       // Copies A[q + 1..r] into R[1..n_1]
10  L[n_1 + 1] = ∞
11  R[n_2 + 1] = ∞
12  i = 1
13  j = 1
14  for k = p to r
15       if L[i] ≤ R[j]
16            A[k] = L[i]
17            i = i + 1
18       else A[k] = R[j]
19            j = j + 1
```

We will now explain what MERGE does:

(1) It takes as input an array A of some unknown length, together with index parameters $p \leq q < r$. We assume that the two subarrays$A[p..q]$ and

$A[q+1\,..\,r]$ are already sorted, and the goal is to merge these two subarrays into a sorted array which will be located at $A[p\,..\,r]$.

(2) In lines 3-9, we copy these two sorted subarrays into the arrays $L[1\,..\,n_1]$ and $R[1\,..\,n_2]$. Later we will merge L and R and store the overall sorted array back in $A[p\,..\,r]$.

(3) In lines 10-11, we attach a **sentinel** ∞ at the end of L and R. The value ∞ represents some value which is larger than every key being considered. Its purpose is to let us know when we reach it that we have already exhausted that particular subarray. In terms of our thought experiment, this would be like putting a piece of paper at the bottom of each stack to merge which says "there are no more homeworks in this stack".

(4) Lines 12-19 do the actual merging. The variables i and j serve as pointers to the next element in each array which should be merged.

In Figure 3.2 we trace through MERGE$(A, 5, 6, 8)$ where $A[5\,..\,8] = \langle 2, 3, 1, 4\rangle$.
We will delay our analysis of MERGE until after we have our full MERGE-SORT algorithm. Given MERGE as a subroutine, the overall MERGE-SORT is rather simple:

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

We will now explain what MERGE-SORT does:

(1) MERGE-SORT takes as input an array A together with two index parameters $p \leq r$. The goal of MERGE-SORT is to sort the subarray $A[p\,..\,r]$, i.e., after MERGE-SORT(A, p, r) ends, the elements of $A[p\,..\,r]$ are the original elements of $A[p\,..\,r]$, except now in sorted order.

(2) The first thing MERGE-SORT does is check whether there is more than one element. If $p = r$, then there is only one element to be sorted. Since a one-element array already is sorted, then we are done and do not proceed to lines 2-5.

(3) Otherwise, if $p < r$, then there is actual work to do. Line 2 roughly divides the subarray in half, by finding an appropriate middle index. Then we delegate sorting these two halves to a recursive call of MERGE-SORT. After lines 3-4, we now know that $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are both sorted, so in line 5 we merge them together. Then we are done.

(4) Finally, if we want to sort an entire array A, and not just a subarray, we call MERGE-SORT$(A, 1, A.\mathit{length})$.

Proof of correctness. In order to prove the correctness of MERGE-SORT, we first must prove the correctness of MERGE. The algorithm MERGE will sort for us two consecutive subarrays *provided* that the two subarrays are each separately sorted to begin with.

Proposition 3.2.1. *Suppose A is an array and $1 \leq p \leq q < r \leq A.\mathit{length}$. If $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted, then after running* MERGE(A, p, q, r) *the subarray $A[p\,..\,r]$ will be sorted and consist of the original elements in $A[p\,..\,r]$.*

PROOF. After lines 1-11 have run, the following is the state of all the variables:

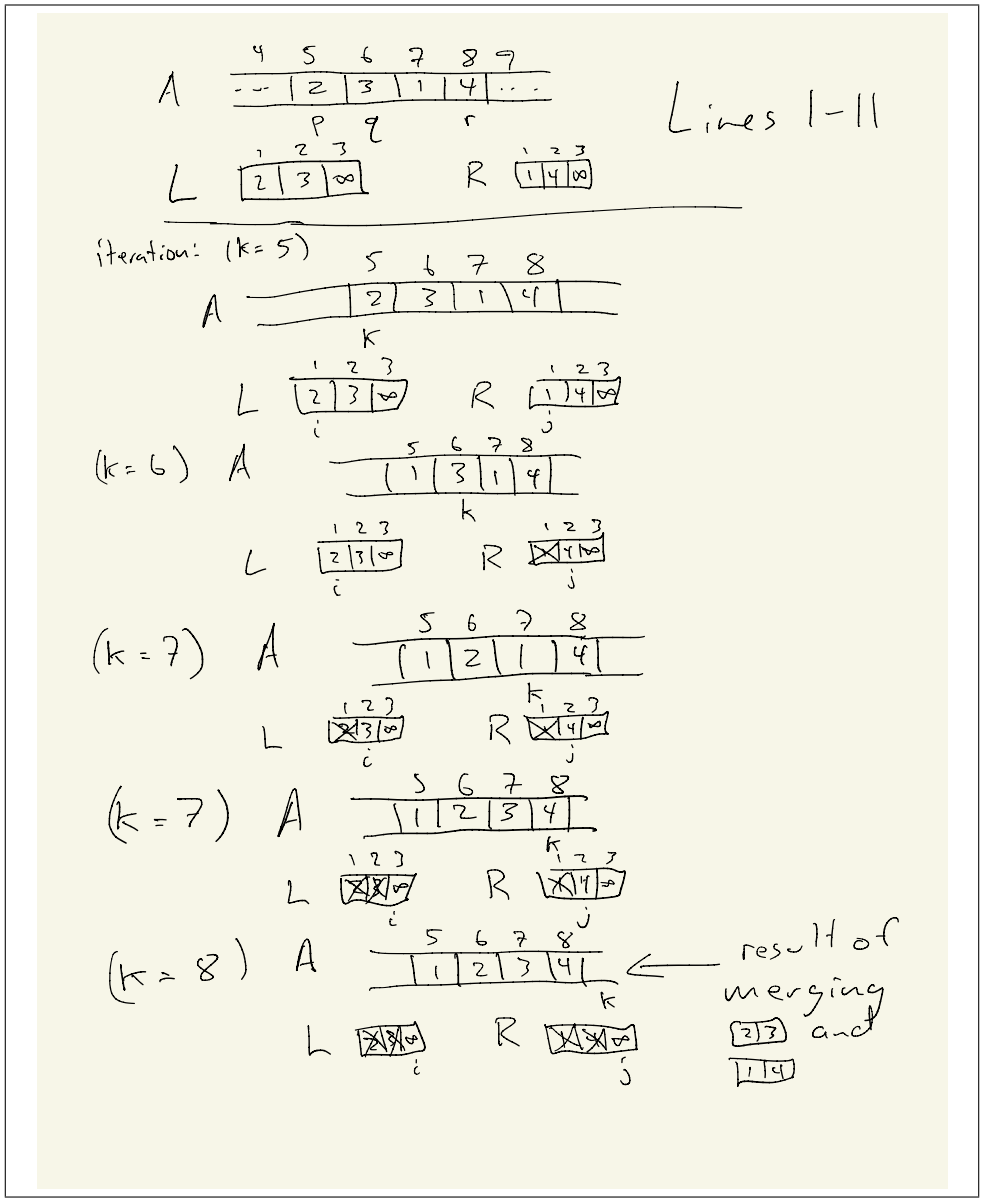


FIGURE 3.2. Here we trace through MERGE$(A, 5, 6, 8)$ where $A[5\,..\,8] = \langle 2, 3, 1, 4\rangle$. This means that we will merge $\langle 2, 3\rangle$ with $\langle 1, 4\rangle$, so we should end up with $A[5\,..\,8] = \langle 1, 2, 3, 4\rangle$, which we do. At the top we show the result of lines 1-11, after we have copied $A[5, 6]$ into $L[1, 2]$ and $A[6, 7]$ into $R[1, 2]$.

- the array A is unchanged
- for every $1 \le i \le n_1$, $L[i] = A[p + i - 1]$ (this could be rigorously proved with a loop invariant, but we won't bother). In particular, L is in sorted order.

- for every $1 \leq j \leq n_2$, $R[j] = A[q + j]$. In particular, R is in sorted order.
- $L[n_1 + 1] = R[n_2 + 1] = \infty$
- the points i and j have been initialized to $i = j = 1$.

We then enter line 12, so we need a loop invariant which says what should be true each time line 12 is run. Here it is:

> (Loop Invariant) After each time line 12 has run, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays which have not been copied back into A.

(Initialization) Immediately after line 12 has run the first time, $k = p$ so the subarray $A[p\,..\,k-1]$ is empty, so it contains the $k-p = 0$ smallest elements from L and R (vacuously). Furthermore, since L and R are in sorted order and $i = j = 1$, then $L[i] = L[1]$ and $R[j] = R[1]$ are the smallest elements of their arrays overall, and so they are also the smallest elements of their arrays which have not been copied back into A.

(Maintenance) Suppose we have just run line 12 and the current value of k is $k = k_0$ where $p \leq k_0 \leq r$. Furthermore, suppose we know the loop invariant is true at this point. First suppose $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied into A. In line 16 we copy it into spot $A[k_0]$, so now $A[p\,..\,k_0]$ contains the $k_0 + 1 - p$ smallest elements of L and R in sorted order. Next we increase i to $i = i+1$, so now $L[i]$ remains the smallest element of L not yet copied back into A. Then we go back to line 14 and increase k to $k = k_0 + 1$. We see the loop invariant still holds. In the case where $L[i] > R[j]$, we perform similar actions instead in lines 18-19 and the loop invariant is still true.

(Termination) Now that we know the loop invariant is always true, let's see what it says after the final time line 14 is run. In this case $k = r + 1$, so the subarray $A[p\,..\,r]$ contains the $r - p + 1$ smallest elements of $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$. Since we never copied in the sentinels ∞, this is precisely the $r - p + 1$ smallest elements of the original array $A[p\,..\,r]$, that is to say, it is all the elements of the array $A[p\,..\,r]$. Thus we are done. □

Now we will prove correctness of MERGE-SORT. Since MERGE-SORT calls itself, this will be a proof by induction.

Theorem 3.2.2. *Given an array A and indices $1 \leq p \leq r \leq A.length$, after* MERGE-SORT$(A, p, r)$ *is called, the elements of the subarray $A[p\,..\,r]$ will consist of the original elements of $A[p\,..\,r]$, except in sorted order.*

PROOF. We will prove the following statement by induction on $n \geq 0$:

$P(n)$: "For all such indices $p \leq r$, if $r - p \leq n$, then after MERGE(A, p, r) is called, the elements of the subarray $A[p\,..\,r]$ will consist of the original elements of $A[p\,..\,r]$, except in sorted order."

(Base Case) Suppose $n = 0$. Then $p = r$, so in line 1 the condition "$p < r$" is false, so we end the algorithm. In this case the subarray $A[p\,..\,r]$ is just a single element which is already in sorted order.

(Inductive step) Suppose for some $n \geq 0$ we know that $P(n)$ is true. Now suppose we have indices $p \leq r$ such that $r - p = n + 1$. Then the condition "$p < r$"

in line 1 is true so we proceed to line 2. We set $q := \lfloor (p+r)/2 \rfloor$. It follows that $p - q + 1 \leq r - n$ and $r - q \leq r - p = n$. Thus the inductive assumption $P(n)$ tells us that after lines 3 and 4 run, the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ consists of the original elements of those subarrays, except in sorted order. Next, when we call MERGE(A, p, q, r), Proposition 3.2.1 tells us that afterwards the elements of $A[p\,..\,r]$ will consist of the original elements of $A[p\,..\,r]$, except in sorted order. This finishes the proof of $P(n)$.

By the Principle of Induction we conclude that $P(n)$ is true for every $n \geq 0$. Thus the theorem is proved. □

Analysis of running time. Our next task is to analyze the running-time of of MERGE-SORT. For later benefit, it will be helpful to introduce a concept at this point which we will study more in the next chapter. The recursive MERGE-SORT algorithm is an example of a so-called **divide-and-conquer** algorithm. A divide-and-conquer algorithm divides the original problem into several similar subproblems of the same type, recursively solves the subproblems, and then combines the solutions of the subproblems into a solution of the original problem. For MERGE-SORT, this can be phrased as follows:

Divide: We first divide the n-element sequence into two subsequences of $\sim n/2$ elements each.

Conquer: We recursively sort the two subsequences using MERGE-SORT.

Combine: We merge the two sorted subsequences to produce the sorted answer.

For the rest of this section we let $T(n)$ denote the running time of MERGE-SORT on an array of length n. Based on the pseudocode of MERGE-SORT, there are two cases to consider, whether or not $n = 1$ or $n > 1$. When $n = 1$, MERGE-SORT(A, p, r) runs line 1, sees the condition "$p < r$" is false, since $p = r$, and returns without doing anything. Thus MERGE-SORT runs in constant time when $n = 1$:

$$T(n) \;=\; \Theta(1) \quad \text{when } n = 1.$$

Next we consider the case when $n \geq 2$. MERGE-SORT works correctly for any n, whether n is even or odd, but to simplify our analysis, we will assume that n is a power of 2 (we'll see in the next Chapter that this assumption doesn't affect our analysis). In this case the work performed in MERGE-SORT breaks down as follows:

Divide: We check line 1, the condition "$p < r$" is true (since $n = r - p > 1$), so we proceed to line 2. Then in line 2 we compute the value of q which essentially divides our array in half. Lines 1 and 2 each take constant time, so the *divide step* contributes $\Theta(1)$ amount of time.

Conquer: In lines 3-4, we recursively solve two subproblems, i.e., we call MERGE-SORT on two subarrays which each have length $n/2$. This contributes $2T(n/2)$ amount of time. (For arbitrary $n > 1$ this would contribute $T(\lceil n/2 \rceil) + T(\lfloor (\rfloor\, n/2))$ amount of time, which simplifies to $2T(n/2)$ when n is even).

Combine: In line 5, we combine the solutions to the two subproblems, by merging together the arrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ by calling MERGE. Based on the **for** loop structure in MERGE, we see that calling MERGE takes $\Theta(n)$ amount of time.

Combining all of these times together gives us

$$T(n) \;=\; 2T(n/2) + \Theta(n) + \Theta(1) \;=\; 2T(n/2) + \Theta(n) \quad \text{when } n > 1.$$

This gives us a recurrence for $T(n)$:

$$T(n) \;=\; \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

To make things more concrete, we can rewrite this recurrence as

$$(\dagger) \qquad T(n) \;=\; \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

Note: it is unlikely that the constant used in $\Theta(1)$ is actually the same as the constant used in $\Theta(n)$, and it is unlikely that the $\Theta(n)$ function is exactly of the form cn. However for asymptotic arguments these types of simplifying assumptions are ok. We'll learn more about this in the next chapter.

There are several ways to solve the recurrence (†). Figure 3.3 illustrates the **recursion tree** method. It works essentially by unrolling the recursive calls all the way to the base case ($n = 1$) when the recursion "bottoms out".

(1) As a first approximation, we see that $T(n)$ can be represented as a tree, with the root denoted by cn (the amount of non-recursive work done by MERGE-SORT in the *divide* and *combine* steps), and two leaves each denoted by $T(n/2)$ (the amount of recursive work done by MERGE-SORT in the two recursive calls, i.e., the *conquer steps*). Adding all the work in each of the levels together, we see that $T(n) = cn + 2T(n/2)$, which we already knew.

(2) In the next tree, we perform the same analysis on each of the two $T(n/2)$ nodes. Each $T(n/2)$ node from the first tree gets replaced with $cn/2$ (the amount of non-recursive work) and two leaves denoted by $T(n/4)$ (the amount of recursive work). Summing up all the work done at each level gets us

$$T(n) \;=\; cn + 2c(n/2) + 4T(n/4) \;=\; 2cn + 4T(n/4).$$

(3) We could keep going in this manner, for instance, the tree with 3 levels of recursion would tell us that

$$T(n) \;=\; 3cn + 8T(n/8).$$

(4) We see a pattern begin to emerge. This unrolling process cannot go on forever though, because at some point we will reach the base case. Using our assumption that n is a power of 2, we see that we will reach the base case after i levels, where $n/2^i = 1$. To determine the number of levels, we solve for i: $n = 2^i$, so $i = \lg n$. Including the top node, there are $\lg n + 1$ total levels of our tree. What is the total amount of work done at each level? At the top node we perform cn amount of work. Each subsequence level going down, the amount of work done at each node cuts in half, but there are twice as many nodes on that level, so all subsequent levels before the base case also performs a total amount of work. For the very bottom level, where we reach the base case, each leaf performs a constant c amount of work, and there are n total leaves (each element of the original array becomes a leaf exactly once), so the last row also takes

cn amount of work. To get the total amount of work we sum up all work done on every level:

$$T(n) \;=\; \sum_{i=0}^{\lg n} cn \;=\; cn(\lg n + 1) \;=\; cn \lg n + cn \;=\; \Theta(n \lg n).$$

We conclude the running time of MERGE-SORT is $\Theta(n \lg n)$. Of course, this recursion tree method is at best a semi-rigorous justification of the running time of MERGE-SORT. However it is very useful for showing us the big picture of the recursion and where the work actually gets done. And at the very least, it provides us with an educated guess as to what the running-time should be, which we could go back later and prove rigorously by induction if we were so inclined. In the next chapter we will see other methods of solving these types of recurrence relations.

3.3. Lower bound on comparison-based sorting

We have now see two sorting algorithms:

(1) INSERTION-SORT which has running time $\Omega(n)$ and $O(n^2)$.
(2) MERGE-SORT which has running time $\Theta(n \lg n)$.

At this point we ask the question:

> What is the best possible running time we could hope for in any sorting algorithm? Is it possible to do better than $\Theta(n \lg n)$?

The answer to this question is *no*, which we will prove in this section. First, we need to clarify the question. Here, when we say *sorting algorithm*, we really mean *comparison-based sorting algorithm.*

A **comparison-based sorting algorithm** is any algorithm which sorts an array A which is only permitted to compare keys in A. I.e., the only way our algorithm can interface with the keys of A is by performing tests between two keys like "$a_i \leq a_j$?" and "$a_i \geq a_j$?" (similarly with $<, >, =$). We are not allowed to, for instance, assume the a_i are numbers and perform arithmetic with them, or something like that (because our comparison-based sorting algorithms should also work for non-numeric sorting problems where we need to alphabetize names, and things like that).

The main result is the following:

Theorem 3.3.1. *Any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.*

It is not clear how to even prove something like Theorem 3.3.1 since it requires us to quantify over *all* algorithms. Indeed, it already took some effort to show two specific algorithms (INSERTION-SORT and MERGE-SORT) were $\Omega(n \lg n)$. As a warmup, we make the following observation:

Observation 3.3.2. *Every comparison-based sorting algorithm has running time $\Omega(n)$.*

PROOF SKETCH. Given an input array A of length n, our algorithm at the very least must read ever entry $A[i]$ of the array. Otherwise, if our algorithm left a particular entry unread, then if you fix all of the other entries, the algorithm should produce the same sorted output (i.e., produce the same permutation of the original entries) regardless of the value of the unread entry. However we could change the value

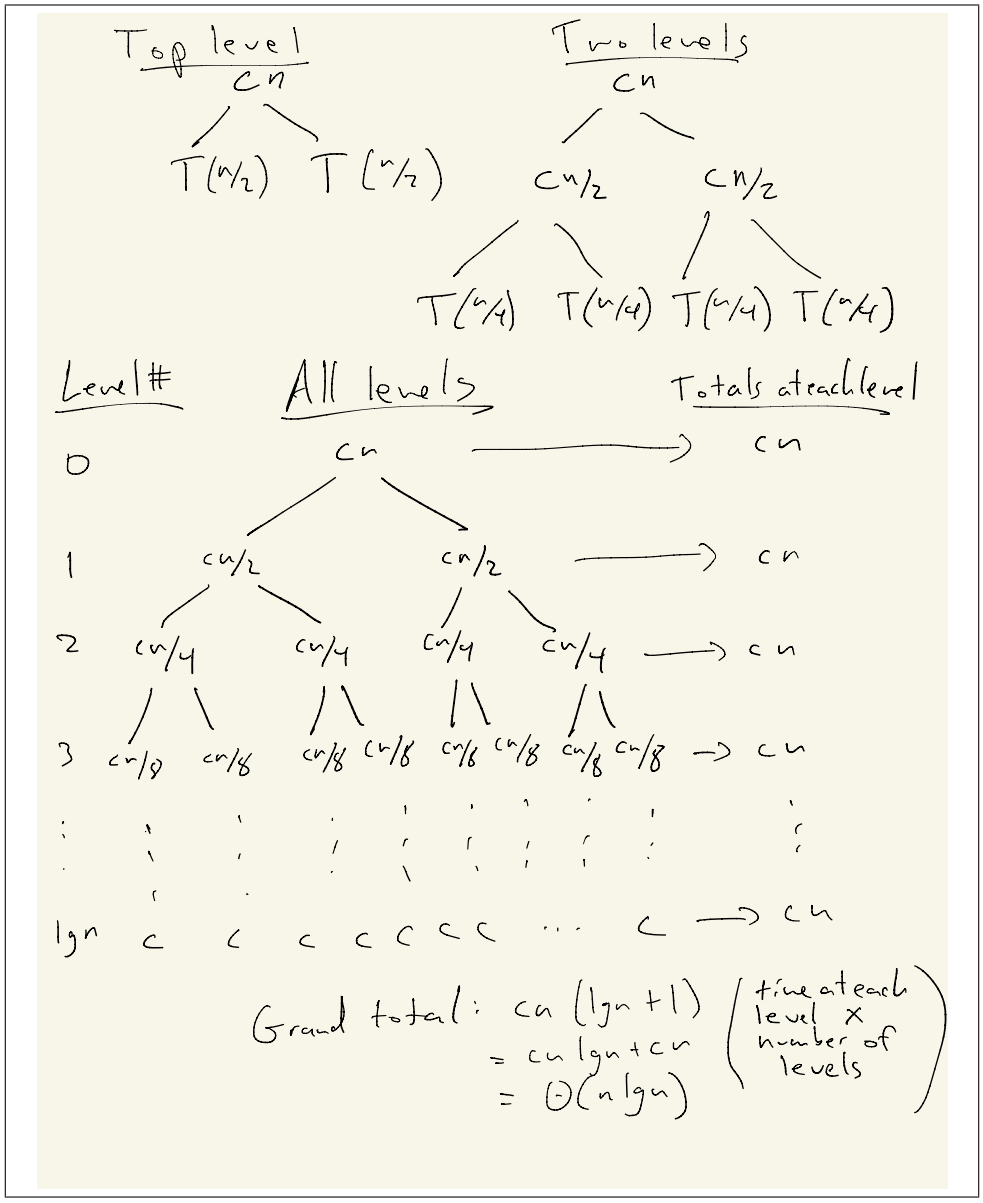


FIGURE 3.3. Here we consider the recursion tree of MERGE-SORT in three levels of detail: (1) With just the top level of recursion shown, (2) With the top two levels of recursion shown, and (3) all three levels of recursion shown.

of the unread entry to be, e.g., the minimum entry in the array or the maximum entry of the array. This should result in two different outputs of the algorithm (one where it puts the unread entry first, and one where it puts the unread entry last) provided the algorithm is correct, which is a contradiction. □

For the proof of Theorem 3.3.1, we will view an algorithm abstractly using the so-called **decision-tree model**. A **decision tree** is a fully binary tree that represents all possible paths that the course of the algorithm can take based on the results of the various comparisons. Since a comparison "$a_i \leq a_j$" can result in two outcomes, essentially *true* or *false*, the algorithm at that spot can only branch in two directions. The leaves of the tree at the bottom represent the output of the algorithm based on the outcome of all comparisons of the original keys which were performed. The output is given in terms of what permutation of the original keys is needed in order to sort the input. See Figure 3.4.
We now sketch the proof of Theorem 3.3.1.

PROOF SKETCH OF 3.3.1. Suppose we have an arbitrary comparison-based sorting algorithm. Suppose we input an array of size n. We will show that $\Omega(n \lg n)$ comparisons are required in the worst case. The proof is based on several key observations about this algorithm and its decision tree:

(1) The leaves of the tree must contain all possible permutations of n keys. For instance, suppose we have an arbitrary permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ of n elements, with inverse permutation π^{-1}. Suppose we have an already-sorted array $\langle a_1, \ldots, a_n \rangle$ of n distinct keys, with $a_1 < \cdots < a_n$. Consider the array $A := \langle a_{\pi^{-1}(1)}, a_{\pi^{-1}(2)}, \ldots, a_{\pi^{-1}(n)} \rangle$. Then in order for the algorithm to sort the array A, it must use the permutation π in order to output $\langle a_1, a_2, \ldots, a_n \rangle$, so the permutation π must show up as one of the leaves.
(2) There are $n!$ many permutations of n objects. Therefore our decision tree must contain at least $n!$ leaves.
(3) Suppose our decision tree has height $h \geq 0$, and the the number of leaves is ℓ. The maximum number of leaves on a binary tree of height h is 2^h, so $\ell \leq 2^h$. By (2), we have $n! \leq \ell$. Thus
$$n! \ \leq \ 2^h.$$
(4) By taking logarithms, we have
$$h \ \geq \ \lg(n!) \ = \ \Omega(n \lg n)$$
by Stirling's Approximation.
(5) In the worst-case, the number of comparisons needed to sort an array of size n is equal to the height h of the tree (select an unsorted input which takes that particular path in the algorithm), and so the worst-case number of comparisons is $\Omega(n \lg n)$. □

3.4. Exercises

Exercise 3.4.1. Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

BUBBLESORT(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

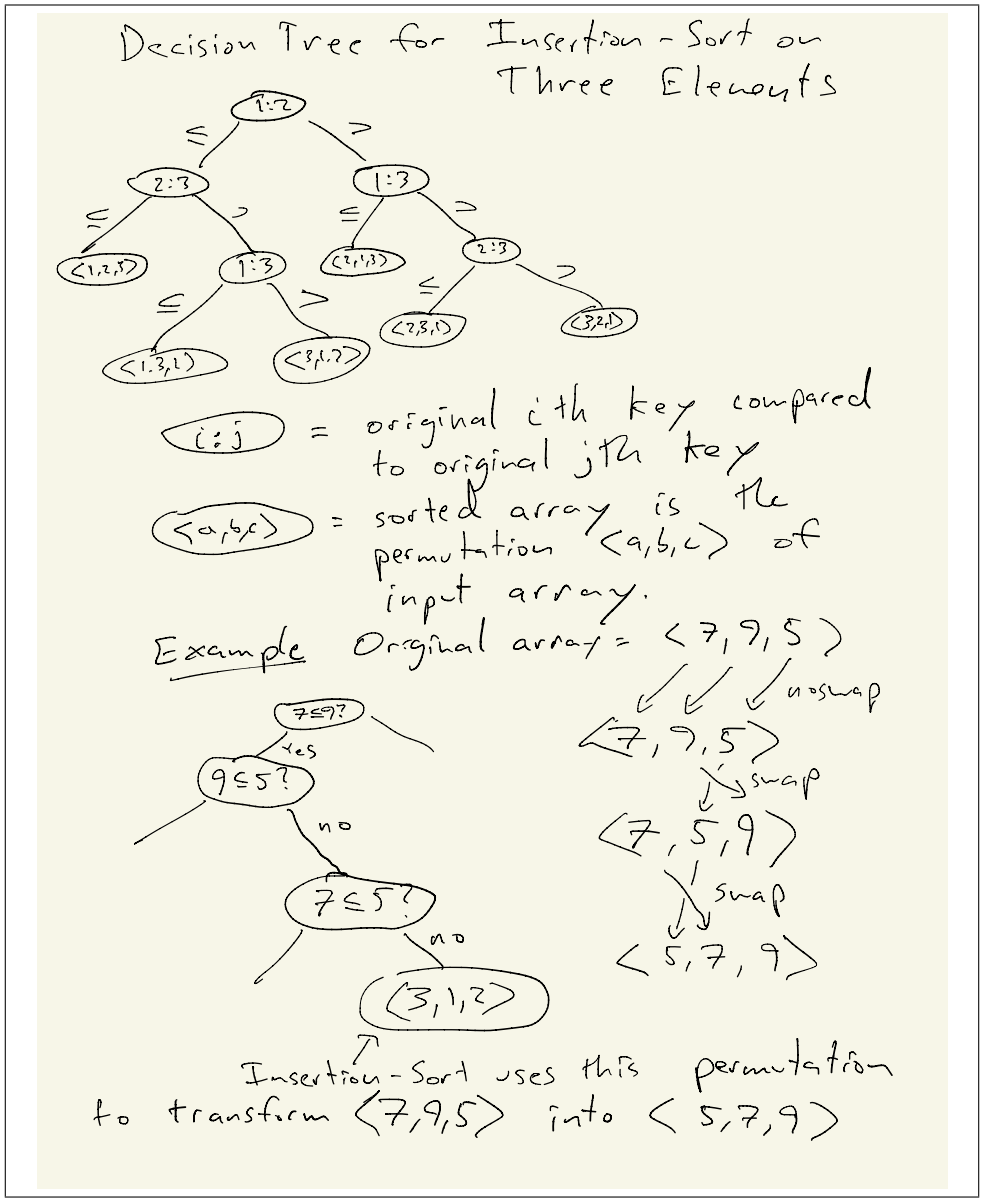


FIGURE 3.4. Here we show the decision tree for the INSERTION-SORT algorithm run on an input of size 3. The leaves of the tree represent the permutation of the original input needed in order to sort the original input. We also show an example of a particular path of the decision tree based on the input $\langle 7, 9, 5\rangle$ which requires the permutation $\langle 3, 1, 2\rangle$ in order to be put into sorted for $\langle 5, 7, 9\rangle$.

(1) Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n], \tag{$\dagger$}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (†).

(2) State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.
(3) Using the termination condition of the loop invariant proved in part (2), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove inequality (†). Your proof should use the structure of the loop invariant proof presented in this chapter.
(4) What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Exercise 3.4.2. Let $A[1\,..\,n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

(1) List the five inversion of the array $\langle 2, 3, 8, 6, 1\rangle$.
(2) What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?
(3) What is the relationship between the running time of insertion sort and the number of inversion in the input array? Justify your answer.
(4) Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint*: Modify merge sort.)

CHAPTER 4

Divide-and-Conquer

In this chapter we continue our discussion of *divide-and-conquer* algorithms which we began in Section 3.2 with MERGE-SORT. Recall that in general, divide-and-conquer algorithms are recursive programs where at each level of the recursion they do the following three steps:

Divide: Divide the problem into a certain number of subproblems that are smaller instances of the same problem.

Conquer: Solve each of the subproblems recursively (possibly using the same algorithm). If the subproblem is small enough, the recursion "bottoms out" and can be solved directly without recursion.

Combine: Given solutions to each of the subproblems, combine them into a solution to the original problem.

4.1. The maximum-subarray problem

The first divide-and-conquer algorithm we consider in this chapter is the **maximum-subarray problem**. This problem arises in many applications, sometimes masquerading in a different form (for instance, in [**1**, §4.1] the original problem involves stock prices and it is subsequently transformed into an instance of the maximum-subarray problem). The problem is as follows:

> Given an array A of numbers, find indices $i \leq j$ such that the sum of the numbers in the subarray $A[i\,..\,j]$ is maximum. Such a subarray is called a **maximum subarray**.

For example, the array

$$A[1\,..\,16] \;=\; \langle 13, -3, -25, 20, -3, -16, -23, \underline{18, 20, -7, 12}, -5, -22, 15, -4, 7\rangle$$

has a maximum subarray:

$$A[8\,..\,11] \;=\; \langle 18, 20, -7, 12\rangle$$

since $18 + 20 - 7 + 12 = 43$, which is the greatest sum of any of the subarrays of A. Of course, we can solve this problem by iterating over all possible subarrays:

```
NAIVE-MAX-CROSSING-SUBARRAY(A)
 1  max = −∞
 2  max-i = 0
 3  max-j = 0
 4  for i = 1 to A.length
 5       for j = i to A.length
 6            sum = sum of A[i] through A[j]
 7            if sum > max
 8                 max = sum
 9                 max-i = i
10                 max-j = j
11  return max, max-i, max-j
```

However, since $i \leq j$ must take all possible pairs of values, the running time of NAIVE-MAX-CROSSING-SUBARRAY is $\Omega(\binom{n}{2}) = \Omega(n^2)$, where $n = A.length$. Using a divide-and-conquer strategy, we will be able to achieve a running time which is $o(n^2)$.

A divide-and-conquer solution. Using the MERGE-SORT algorithm as inspiration, our first attempt might be as follows:

(1) Divide the array into a left half and a right half.
(2) Recursively determine the maximum subarray on each half separately.
(3) Combine the solutions by taking whatever is the greatest of the solutions on each half.

However, if we use this strategy then we are neglecting many of the subarrays of A which could achieve the maximum. For instance, this strategy won't check subarrays which begin in the left half, and end in the right half. It will only see subarrays contained entirely in the left half or entirely in the right half. This suggests instead we should do the following:

(1) Given the array $A[low\,..\,high]$, first find a suitable midpoint index mid.
(2) Recursively find the maximum subarray on the two halves separately: $A[low\,..\,mid]$ and $A[mid + 1\,..\,high]$
(3) Also find the maximum subarray which crosses the midpoint, i.e., a subarray of the form $A[i\,..\,j]$, where $i \leq mid$ and $j \geq mid + 1$.
(4) Combine the solutions by taking the maximum of all three subarrays.

The three options for where the maximum subarray can be found are shown in Figure 4.1.

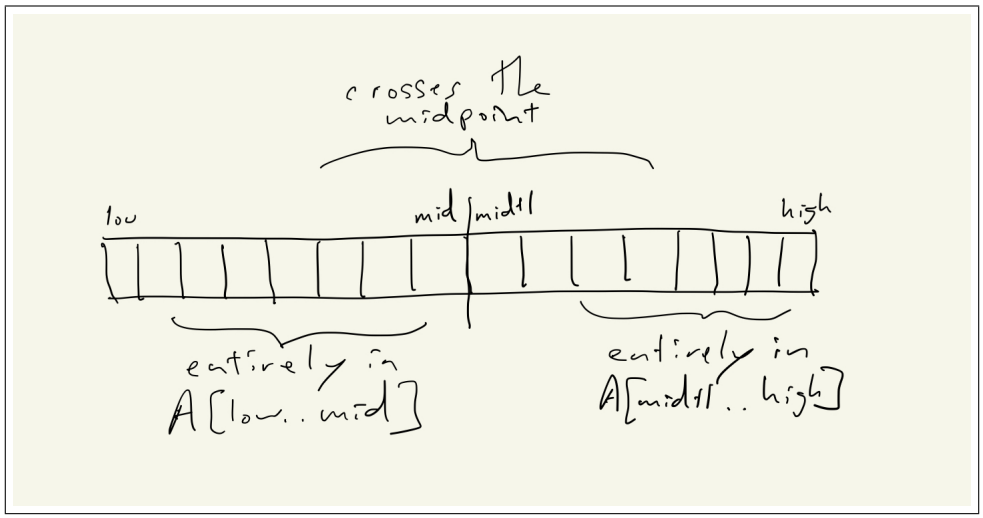


FIGURE 4.1. This shows the three locations where we may find our maximum subarray: contained in the left half $A[low\,..\,mid]$, contained in the right half $A[mid + 1\,..\,hight]$, or split between both halves and crossing the midpoint (of the form $A[i\,..\,j]$ where $i \le mid$ and $j \ge mid + 1$)

The first thing we need is a subroutine which finds the maximum subarray which crosses the midpoint:

FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$

```
1   left-sum = −∞
2   sum = 0
3   for i = mid downto low
4       sum = sum + A[i]
5       if sum > left-sum
6           left-sum = sum
7           max-sum = i
8   right-sum = −∞
9   sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Here are some comments on what FIND-MAX-CROSSING-SUBARRAY does (see Figure 4.2):

(1) In the **for** loop on lines 3-7, we find the maximum subarray of the form $A[i\,..\,mid]$, where $low \le i \le mid$.
(2) In the **for** loop on lines 10-14, we find the maximum subarray of the form $A[mid + 1\,..\,j]$, where $mid + 1 \le j \le high$.

(3) Then we know the maximum subarray which crosses the midpoint must be $A[\textit{max-left} \,..\, \textit{max-right}]$, so we return these two indices and the sum of this subarray.

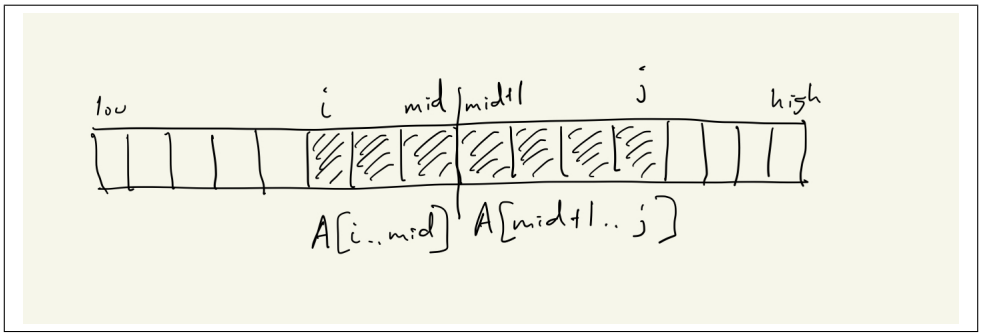


FIGURE 4.2. Finding the two subarrays which, when joined together, constitutes the maximum subarray of $A[\textit{low} \,..\, \textit{high}]$ which crosses the midpoint

Here is the main algorithm which implements the idea:

FIND-MAXIMUM-SUBARRAY($A, \textit{low}, \textit{high}$)

```
 1  if high == low
 2       return (low, high, A[low])
 3       // base case: only one element
 4  else mid = ⌊(low + high)/2⌋
 5       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 6       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 7       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 8       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 9            return (left-low, left-high, left-sum)
10       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
11            return (right-low, right-high, right-sum)
12  else return (cross-low, cross-high, cross-sum)
```

Here are some comments as to what this algorithm does:

(1) Lines 1-3 handle the base case: if the array only has one element, then automatically the maximum subarray is just the element itself.
(2) Line 5 determines the maximum subarray in the left half.
(3) Line 6 determines the maximum subarray in the right half.
(4) Line 7 determines the maximum subarray which crosses the midpoint, by calling the subroutine *Find-Max-Crossing-Subarray*.
(5) Lines 8-12 compares these three maximum subarrays and returns whichever one ultimately is the maximum.

We will omit a proof of correctness for these algorithms (although you should know how to do it). Instead we will dive right in to the main issue of determining the running time of FIND-MAXIMUM-SUBARRAY (recall, we want it to be $o(n^2)$).

Running time analysis. First, we should determine the running time of FIND-MAX-CROSSING-SUBARRAY. Suppose the input array $A[low\,..\,high]$ has size n, i.e., $n = high - low + 1$. We see that lines 1,2,8,9,15 all take a constant amount of time. Furthermore, the body of each **for** loop takes a constant amount of time each time its run. Thus the running time is determined by the number of iterations of each **for** loop. The first **for** loop on lines 3-7 performs $mid - low + 1$ iterations, and the second **for** loop on lines 10-14 performs $high - mid$ iterations. Thus the total number of iterations is $mid - low + 1 + high - mid = high - low + 1 = n$. We conclude that FIND-MAX-CROSSING-SUBARRAY runs in $\Theta(n)$ linear time.

Next we will analyze the running time of FIND-MAXIMUM-SUBARRAY. Let $n = high - low + 1$ be the size of the input array. The first case is when $high = low$, i.e., when A has one element. In this case we run lines 1-3 all in constant time and we're done. Thus

$$T(n) \;=\; \Theta(1) \quad \text{when } n = 1$$

Now suppose $n \geq 2$. We break down the running time into the three categories of a divide-and-conquer algorithm:

(1) (**Divide**) We run lines 1 and 4. This takes constant time $\Theta(1)$.
(2) (**Conquer**) We solve three subproblems. First we find the maximum subarray on the left half, which takes $T(n/2)$ amount of time. Then we find the maximum subarray on the right half which also takes $T(n/2)$ amount of time. Finally, we find the maximum subarray which crosses the midpoint, which takes $\Theta(n)$ amount of time.
(3) (**Combine**) In lines 8-12, we combine our three solutions by comparing them to each other to find out which one truly is best. This takes a constant $\Theta(1)$ amount of time.

To conclude, when $n \geq 2$ the running time is

$$\begin{aligned} T(n) \;&=\; \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &=\; 2T(n/2) + \Theta(n). \end{aligned}$$

We can summarize this recurrence as follows:

$$T(n) \;=\; \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

This is the same recurrence which MERGE-SORT satisfied in Section 3.2. There we argued that the running time was $\Theta(n \lg n)$, so the same should be true here (we'll see how one could prove this rigorously in the next section). To summarize:

Theorem 4.1.1. *The running time of* FIND-MAXIMUM-SUBARRAY *is* $\Theta(n \lg n)$.

4.2. The substitution method

In this section we discuss one of the strategies for rigorously proving Θ- or O-bounds on a function $T(n)$ which satisfies a recurrence relation. This strategy is called the **substitution method**. First a few words about recurrences in general.

Recurrences. A **recurrence** is a function or inequality which describes a function $T(n)$ in terms of its values on smaller inputs. For instance, in Section 3.2 we described the running time of MERGE-SORT with the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

Here, the second case $T(n) = 2T(n/2) + \Theta(n)$ tells us that we subdivided the original problem into two smaller subproblems, each of size $n/2$, and the *combine* step took linear time $\Theta(n)$. In general, recurrence relations (and divide-and-conquer algorithms) do not need to split the original problem into evenly-sized smaller subproblems. For instance:

- The recurrence $T(n) = T(n/3)+T(2n/3)+\Theta(n)$ might come for a problem where we split the original problem into two problems of sizes $n/3$ and $2n/3$ respectively, in addition to spending a linear amount of time $\Theta(n)$ on the combine step.
- The recurrence $T(n) = T(n-1) + \Theta(1)$ might come from a problem where we reduce the original problem into a problem of size one less, plus spending a constant amount of time $\Theta(1)$ on the combine step.

Technically speaking, the recurrence for MERGE-SORT really should be

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

Since we can't always assume that the n is always even (or a power of two, for that matter). However, for our purposes it is fine to ignore these sorts of details. This is because, if you can solve the recurrence by ignoring these details (say, get a solution $\Theta(n \lg n)$), then in theory you can go back and carefully prove that the recurrence with these details also satisfies the same recurrence. This is not as hard as it may seem, since if you already know the guess $\Theta(n \lg n)$, then this can guide you on proving the more technical version of the recurrence.

We also won't care too much about the base case either. Indeed, any fixed number of base cases will always run in constant time $\Theta(1)$. For this reason, we will often neglect to mention the base case at all, and so if we express a recurrence as

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

then it is understood that we are also assuming for some fixed n_0 (say, $n_0 = 1$) the condition:

$$T(n) = \Theta(1) \quad \text{if } n \leq n_0.$$

The substitution method. The substitution method provides a way to rigorously prove a Θ- or O- bound on a solution to a recurrence. The method consists of two steps:

(1) Guess the form of the solution.
(2) Use mathematical induction to find the constants and show that the solution works.

We will illustrate how this method works with an example.

Example 4.2.1. The recurrence

$$T(n) \;=\; 2T(\lfloor n/2 \rfloor) + n$$

satisfies $T(n) = O(n \lg n)$.

DISCUSSION. We need to show there is some $c > 0$ such that *eventually* $T(n) \leq cn \lg n$. We do not know what this value of c is, but we can assume it exists and hope that the inductive proof gives us some insight on how big this value of c must be. Assume for some n we know that for all $m < n$, $T(m) \leq cm \lg m$. Note that

$$\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \quad \text{by our assumption} \\
&\leq cn \lg(n/2) + n \\
&= cn \lg n - cn \lg 2 + n \\
&\leq cn \lg n - cn + n \\
&= cn \lg n - (c-1)n \\
&\leq cn \lg n
\end{aligned}$$

where the last inequality only works if $c - 1 \geq 0$, i.e., if $c \geq 1$. This tells us, after the fact, that the inductive step in our argument would work for any value of c, provided that $c \geq 1$.

Technically we are still not done yet. Since we are ultimately doing a proof by induction, we still need to prove the inequality for our base case(s). In particular, we need to know/determine what our base cases are and what value of c will also work for them. When doing this, we have two freedoms at our disposal:

(1) We can make the constant c has big as we want (when playing this game, there are no bonus points for obtaining the smallest possible c, use this to your advantage).
(2) The inequality $T(n) \leq cn \lg n$ only has to hold *eventually*. I.e., we only need to show there exists some n_0 such that for all $n \geq n_0$, $T(n) \leq cn \lg n$. In this case, n_0 is a value that *we get to choose* (also use this to your advantage).

Thus, in order to deal with base cases, you typically do two things:

(3) Choose n_0 large enough so that all smaller values referred to in the recurrence and in the recurrence inequality make sense and the recurrence inequality on these smaller values has a chance of being true for large enough c.
(4) Then choose c large enough so that the inequality you wish to prove inductively holds for the smaller values.

We illustrate this with an example. Suppose our only base case is $T(1) = 1$. In this situation, our inequality $T(n) \leq cn \lg n$ would be $T(1) = 1 \leq c1 \lg 1 = 0$, which can't possibly be true. Thus we can't start our inductive argument at $n = 2$ or $n = 3$, since this would require the inequality to be true at $n = 1$, which it isn't. Instead, let's start our inductive argument at $n_0 = 4$, and choose c large enough so that the inequality automatically holds for $n = 2$ and $n = 3$. Note that $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. We now want to choose $c \geq 1$

large enough such that $T(2) \leq c2\lg 2$ and $T(3) \leq c3\lg 3$. Thus, if we choose

$$c \;\geq\; \max\left\{1, \frac{4}{2\lg 2}, \frac{5}{3\lg 3}\right\} \;=\; \max\left\{1, 2, \frac{5}{3\lg 3}\right\}$$

then our base cases of $n = 2$ and $n = 3$ automatically hold, and our inductive proof starting at $n = 4$ is also guaranteed to work. If we wanted to, we could turn this discussion into a straightforward proof. □

PROOF. We want to show $T(n) = O(n\lg n)$. I.e., we need to find some $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $0 \leq T(n) \leq cn\lg n$. Choose $c := 2$ and $n_0 := 2$ (the choice of c guarantees $c \geq 1, 2, 5/3\lg 3$). We have two base cases to consider:

(Base Case $n = 2$) In this case, $T(2) = 4 \leq 2 \cdot 2\lg 2 = 4$.

(Case Case $n = 3$) In this case, $T(3) = 5 \leq 2 \cdot 3\lg 3 = 6\lg 3$.

(Inductive Step) Suppose for some $n \geq 4$ we know that the inequality holds for all $m < n$ such that $2 \leq m < n$. Note that

$$\begin{aligned} T(n) &= 2T(\lfloor n/2\rfloor) + n \\ &\leq 4\lfloor n/2\rfloor \lg \lfloor n/2\rfloor + n \quad \text{by inductive assumption} \\ &\leq 2n\lg(n/2) + n \\ &\leq 2n\lg n - 2n\lg 2 + n \\ &\leq 2n\lg n - 2n + n \\ &\leq 2n\lg n. \end{aligned}$$

This finishes our proof of the inductive step. We conclude $T(n) = O(n\lg n)$, as desired. □

Remark 4.2.2. In general, the trick of selectively increasing c and n_0 to handle annoying base cases always works. The important part of this proof is finding a c which works for the inductive part of the argument. After all, if you cannot do this, then this might mean that the original "guess" of $O(n\lg n)$ was wrong. Or it might mean you have to try something else. In general, we might not explicitly work out the details for handling the base case(s), but you should know how to do it.

A counterintuitive trick. Sometimes in your inequality you have to subtract a lower-order term to make the inductive step work. Here is an example:

Example 4.2.3. The recurrence

$$T(n) \;=\; T(\lfloor n/2\rfloor) + T(\lceil n/2\rceil) + 1$$

satisfies $T(n) = O(n)$.

DISCUSSION. We need to show there is some $c > 0$ such that eventually $T(n) \leq cn$. Suppose we have such a value of c and we know the inequality is true for all $m < n$. Note that

$$\begin{aligned} T(n) &= T(\lfloor n/2\rfloor) + T(\lceil n/2\rceil) + 1 \\ &\leq c\lfloor n/2\rfloor + c\lceil n/2\rceil + 1 \\ &= cn + 1. \end{aligned}$$

Unfortunately, since $cn < cn+1$, knowing $T(n) \leq cn+1$ does *not* imply $T(n) \leq cn$, which is what we want. It seems we are stuck because this shows for *no* value of $c > 0$ this inductive argument can work.

To overcome this difficulty, we will *subtract* a lower order term from our guess. Now, assume there are $c, d > 0$ such that eventually $T(n) \leq cn - d$. Assume that this inequality is true for all $m < n$. Note that

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\leq c\lfloor n/2 \rfloor - d + c\lceil n/2 \rceil - d + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \end{aligned}$$

where the last inequality holds provided $-2d + 1 \leq -d$, i.e., $1 \leq d$. Thus, as long as $d \geq 1$, then for any $c > 0$ our inductive argument will work. Of course, in reality we would also need to make sure c is large enough to also handle our base cases, but that is a separate issue. □

Changing variables. The last trick we will discuss in this section is *changing variables*. We will illustrate this also with an example:

Example 4.2.4. The recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

satisfies $O(\lg n \lg \lg n)$.

Discussion. First, we will disregard the floor and instead look at the recurrence

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

Next, we rename $m = \lg n$. Substituting this into the recurrence gives us

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

Next we define the function $S(m) := T(2^m)$. Now we get a recurrence on S:

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

We have already seen that $S(m) = O(m \lg m)$. Thus

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n).$$ □

4.3. The recursion-tree method

One of the drawbacks of the substitution method is that it does not provide us a good method of guessing the running time if we don't know it an advance. If we have a good guess, the substitution method provides a way of proving the guess is correct. But how do we make a guess in the first place? One method is to draw a **recursion tree**. In a recursion tree, we unravel the running time cost $T(n)$ in a tree diagram, where each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. Then we sum the total cost on each level and sum all of the levels in order to arrive at an estimate of the running time.

Usually for recursion-tree arguments, we permit ourselves a certain degree of "sloppiness" because ultimately we are just generating a guess at a good upper/lower bound for the function $T(n)$. Once we have this guess, we can go back and rigorously prove it is correct with the substitution method, thus justifying our sloppiness after-the-fact. We illustrate the recursion-tree method with worked out examples.

Example 4.3.1. The recurrence

$$T(n) \;=\; 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

satisfies $\Theta(n^2)$.

DISCUSSION. It is clear from the recurrence that $T(n) = \Omega(n^2)$. Suppose we did not have a guess at what an upper bound should be. To simplify things, we can remove the floor in the recurrence and replace $\Theta(n^2)$ with a more precise cn^2 for some constant $c > 0$. This gives us the simpler recurrence:

$$T(n) \;=\; 3T(n/4) + cn^2$$

to work with instead. These simplifications are reasonable for two reasons:

(1) If we are going to get a tight asymptotic bound for $T(n)$, it will probably be the same regardless of what the function $\Theta(n^2)$ actually is, so we might as well work with a simple $\Theta(n^2)$ function like cn^2.
(2) Likewise, if we get a tight asymptotic bound which works on powers of 4, this asymptotic bound probably will work for all n. If we work only with powers of 4 then taking the floor does nothing.

We illustrate the recursion tree in Figure 4.3. Here is an explanation:

(1) We assume that n is a sufficiently large power of 4. We want to decompose $T(n)$ into a tree which shows all the work done in each subproblem at each level of the recursion.
(2) At the first level, we use the recurrence itself $T(n) = 3T(n/4) + cn^2$ to show that we do cn^2 amount of work at the top level (think the *divide* and *combine* steps), and then solve three subproblems of size $n/4$ (think the *conquer* steps). Each of these subproblems are a leaf of the root.
(3) In the second level, we expand each of the leaves from the first level into their recursive calls. Since each leaf in the first level is performing $T(n/4)$ amount of work, the recurrence tells us that $T(n/4) = 3T(n/16)+c(n/4)^2$, so we replace each first-level leaf with a node which does $c(n/4)^2$ amount of work, with three new leaves, each performing $T(n/16)$ amount of work.
(4) In the general situation, we continue the pattern downward until our recursion "bottoms out" with leaves $T(1)$ at the bottom. There are several things we need to observe.
    (a) First, how many levels are there? Notice that at level ℓ we are solving a subproblem of size $n/4^\ell$. Thus the last level in the tree will be the unique ℓ such that $n/4^\ell = 1$, i.e., where $\ell = \log_4 n$. Thus there are $\log_4 n + 1$ total levels (including the 0th level corresponding to the root).
    (b) Second, how many nodes are at each level? Since the tree branches evenly with three branches going down, each level has 3 times as many nodes as the previous level. Thus level ℓ has 3^ℓ many nodes, for $0 \le \ell \le \log_4 n - 1$. Finally, level $\ell = \log_4 n$ has $3^{\log_4 n} = n^{\log_4 3}$ many nodes.
    (c) Third, what is the work done per node at each level? At level ℓ, we see the work done is $c(n/4^\ell)^2$ when $0 \le \ell \le \log_4 n - 1$, and the work per node is $T(1)$ at level $\ell = \log_4 n$.
    (d) Fourth, what is the total work done per level? Combining items (2) and (3) above, we see that for $0 \le \ell \le \log_4 n - 1$, the work at level

ℓ is $(3/16)^\ell cn^2$. At level $\ell = \log_4 n$, the work done is $T(1)n^{\log_4 3} = \Theta(n^{\log_4 3})$.

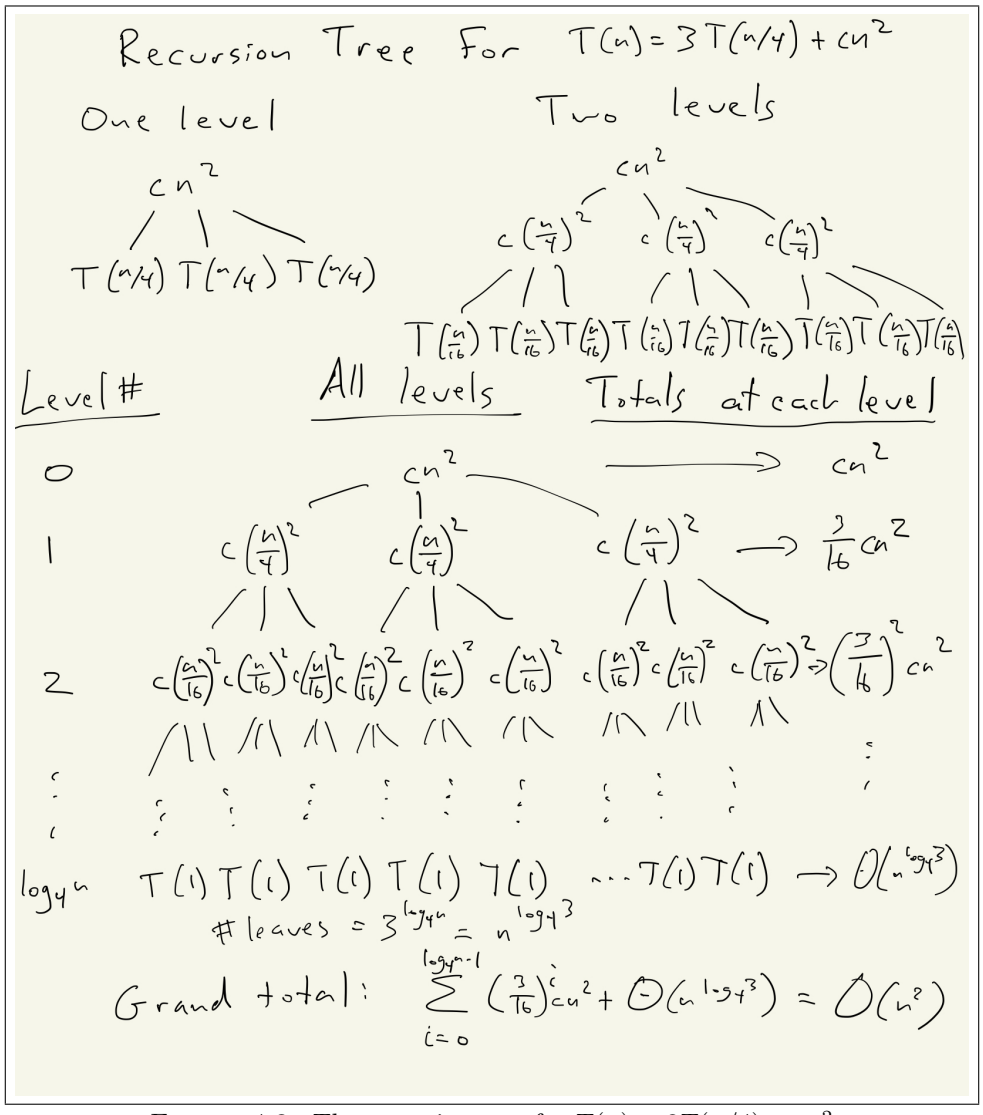


FIGURE 4.3. The recursion tree for $T(n) = 3T(n/4) + cn^2$

(5) Now that we know how much work is done at each level, by adding up all the levels we get

$$\begin{aligned} T(n) &= \sum_{\ell=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^\ell cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

by the geometric sum formula. Since we are going for an upper bound of $T(n)$, we can increase (and simplify) this expression a little bit by using the infinite geometric series instead:

$$\begin{aligned} T(n) &= \sum_{\ell=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^\ell cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{\ell=0}^{\infty} \left(\frac{3}{16}\right)^\ell cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2). \end{aligned}$$

Finally, at this point we would guess that $T(n) = O(n^2)$ (and thus $T(n) = \Theta(n^2)$).

Now that we are equipped with this guess, we could proceed to prove it is correct using the substitution method. We will go back to the original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Since $\Theta(n^2)$ is a function which is eventually bounded above by some cn^2, this gives us $T(n) \leq 3T(\lfloor n/4 \rfloor) + cn^2$ eventually. We will assume there is some $d > 0$ such that eventually $T(n) \leq dn^2$. Assuming this holds for smaller values $m < n$, note that

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &\leq ((3/16)d + c)n^2 \\ &\leq dn^2 \end{aligned}$$

where the last inequality holds provide $(3/16)d + c \leq d$, i.e., provided that $d \geq (16/13)c$. This tells us that no matter the value of c, we always have a way of choosing a value of d which makes the inductive proof go through. Combining this with the routine handling of base cases verifies that indeed $T(n) = O(n^2)$. □

Our next example of a recursion tree argument is less symmetric. This will show us that we don't always need a complete understanding of the recursion tree for the purposes of getting bounds.

Example 4.3.2. The recurrence

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

satisfies $O(n \lg n)$.

Discussion. As before, since we are only generating a guess for an upper asymptotic bound, we can replace $O(n)$ with cn for some constant $c > 0$. Thus we are working with the recurrence

$$T(n) = T(n/3) + T(2n/3) + cn.$$

In Figure 4.4 we present the recursion tree for this recurrence.

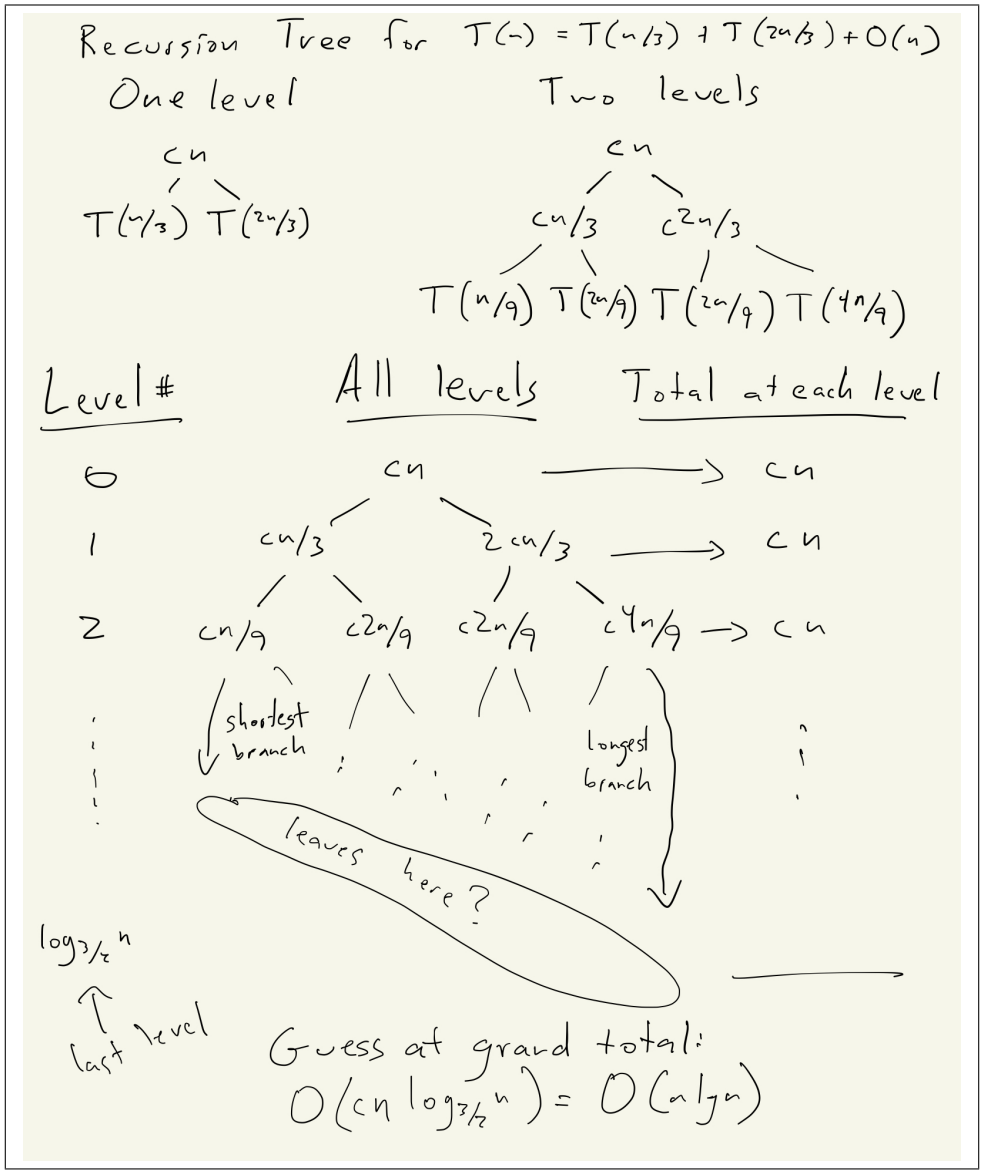


FIGURE 4.4. The recursion tree for $T(n) = T(n/3) + T(2n/3) + cn$

Here is an explanation for the recurrence tree:

(1) As before, we unravel the recurrence once, and then twice, to present the first two levels of the tree.
(2) In the full tree, we see that the total amount of work at each level is cn. This is because each problem of size m gets divided into two subproblems of sizes $m/3$ and $2m/3$, respectively.
(3) How many levels are there in this tree? This is not as simple as before, since this is not a complete tree and the split at each branch does not result in even-sized subproblems. If we follow the $n \to n/3 \to n/9 \to \cdots$ branch on the far left, we will reach a leaf as quickly as possible. If we

follow the $n \to 2n/3 \to 4n/9 \to \cdots$ path we will reach a leaf as slowly as possible. Thus the total number of levels is determined by this longest branch on the far left. Since at level ℓ the right-most branch considers subproblems of size $n/(3/2)^\ell$, the lowest level will correspond to when $n/(3/2)^\ell = 1$, i.e., for $\ell = \log_{3/2} n$.

(4) What about the leaves of the tree? For this tree this is not a simple question, since it is not a complete tree. All the leaves are spread out across different levels between where the shortest branch ends (level $\log_3 n$) and where the longest branch ends (level $\log_{3/2} n$). Since ultimately we are just trying to guess an asymptotic upper bound, we will ignore this question for now.

(5) Using just items (2) and (3), we will guess that $T(n) = O(cn \log_{3/2} n) = O(n \lg n)$. If we can prove this bound is correct using the substitution method, then there is no need to go back and consider the leaves.

For the substitution method, assume we have some $d > 0$ such that $T(m) \le dm \lg m$ for $m < n$. Note that

$$\begin{aligned} T(n) &\le T(n/3) + T(2n/3) + cn \\ &\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\ &= (d(n/3)\lg n - d(n/3)\lg 3) + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\ &= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\ &= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\ &= dn\lg n - dn(\lg 3 - 2/3) + cn \\ &\le dn\lg n \end{aligned}$$

where the last inequality only holds provided $d \ge c/(\lg 3 - 2/3)$. Since for any value of $c > 0$ we can always find such a suitable value of d that works, this verifies our $O(n \lg n)$ bound. □

4.4. The master method

We now arrive at our most powerful method for solving recurrences of a certain form: the *master method*. This is a theorem which automatically gives us a tight asymptotic bound for recurrences of the form

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

where $a \ge 1$ and $b > 1$; it applies to Example 4.3.1 and the recurrence in MERGE-SORT, but not to Example 4.3.2. Without further ado, here it is:

Master Theorem 4.4.1. *Let $a \ge 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

(1) (Leaf-heavy) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(2) (Middle case) If $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant k, then

(a) if $k > -1$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$,

(b) if $k = -1$, then $T(n) = \Theta(n^{\log_b a} \lg\lg n)$, and
(c) if $k < -1$, then $T(n) = \Theta(n^{\log_b a})$.

(3) (Root-heavy) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant ϵ, and the following holds:
- *(Regularity Condition) if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,*

then $T(n) = \Theta(f(n))$.

PROOF. See [**1**, §4.6] for a proof of a slightly weaker version. A proof of the version here can be found in [**4**]. □

Here are some examples[1] of how to use the Master Theorem:

Example 4.4.2. (1) Consider the recurrence

$$T(n) = 8T(n/2) + 1000n^2$$

In this case $a = 8, b = 2, f(n) = O(n^2)$ and $\log_b a = \log_2 8 = 3 > 2$, so we are in case (1). Thus $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$.

(2) Consider the recurrence

$$T(n) = 2T(n/2) + 10n$$

In this case, $a = 2, b = 2$, $\log_b a = \lg 2 = 1$, $f(n) = \Theta(n^2 \lg^0 n)$, and so we are in case 2 with $k = 0$. Thus $T(n) = \Theta(n \lg n)$.

Consider the recurrence

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

In this case $a = 2, b = 2, \log_b a = 1$, $f(n) = \Theta(n^2) = \Omega(n^1)$, so we are in Case 3 provided the regularity condition holds. In order to hold, we need to find a value of $c < 1$ such that

$$2(n/2)^2 = \frac{n^2}{2} \leq cn^2.$$

We see immediately that $c = 1/2$ works. Thus we are in Case 3 and so $T(n) = \Theta(n^2)$.

4.5. Strassen's algorithm for matrix multiplication

Omitted due to time. TLDR: Naive multiplication of two $n \times n$ matrices can be done using $\Theta(n^3)$ scalar multiplications. Strassen's algorithm is a very clever divide-and-conquer algorithm which ultimately does this with $\Theta(n^{\lg 7})$ scalar multiplications. See [**1**, §4.3].

[1] Taken from `https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)`

CHAPTER 5

Data structures

We have already encountered one type of data structure in previous chapters and algorithms: an *array*:

$$A[1\,..\,n] \;=\; \langle a_1, a_2, \ldots, a_n \rangle$$

Arrays are useful when we want to naturally order our data, especially if we intend to lookup specific values frequently.

Arrays are certainly not the only kind of data structure which gets used in algorithms. In fact, a major component of algorithm design is choosing (or inventing) the right data structure for the problem at hand. In this chapter we will learn about three new types of data structures: *heaps*, *priority queues*, *stacks*, and *queues*. In Section 5.2 we will see that heaps provide us with another efficient sorting algorithm: HEAPSORT.

5.1. Heaps

Recall that in Section 3.3 when we analyzed lower bounds on comparison-based sorting algorithms, we saw that nearly complete binary trees naturally arose in our analysis. The *heap* data structure builds this idea directly into the data structure itself.

A **(binary) heap** data structure is an array object that we can view as a nearly complete binary tree (see Figure 5.1). The phrase "nearly-complete" means that the tree is full on all levels, except possibly the last level. On the last level the heap gets filled from left to right. We will implement a heap as part of an array $A[1\,..\,n]$, where the root is $A[1]$, its left and right children, respectively, are $A[2]$ and $A[3]$, their left and right children are $A[4], A[5], A[6], A[7]$, etc. A heap comes with two attributes: $A.\mathit{length}$ which represents the total length of the array which stores the heap, and $A.\mathit{heap\text{-}size}$ which represents which part of the array contains valid elements of the heap (the other entries of the array are not part of the heap). Therefore in general we will have $0 \leq A.\mathit{heap\text{-}size} \leq A.\mathit{length}$.

Given an index i of a node of the heap, we can easily compute the indices of its parent, its left child, and its right child (provided they are in the heap):

```
PARENT(i)
1  return ⌊i/2⌋
LEFT(i)
1  return 2i
RIGHT(i)
1  return 2i + 1
```

All three of these algorithms run in $\Theta(1)$ constant time.

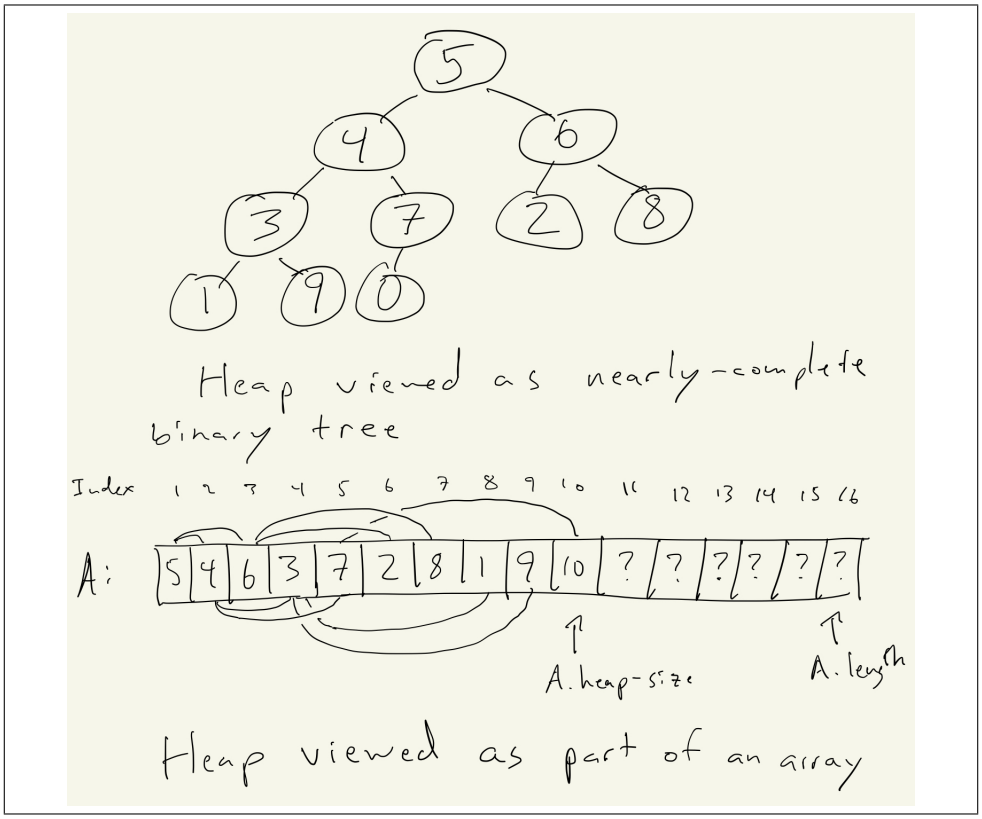


FIGURE 5.1. Here we illustrate a heap in two ways. First as an abstract nearly-complete binary tree. Second as an array. Note that this heap is neither a max-heap nor a min-heap. Furthermore, only the subarray $A[1\,..\,A.\textit{heap-size}]$ represents the heap, the rest of the subarray $A[A.\textit{heap-size}+1\,..\,A.\textit{length}]$ is not part of the heap.

In general there are two special types of heaps we will be interested in: *max-heaps* and *min-heaps*. Both of these satisfy a so-called **heap property**. In a **max-heap**, the **max-heap property** specifies that for every node i other than the parent

$$A[\textsc{Parent}(i)] \;\geq\; A[i]$$

i.e., the values of the nodes increase as you climb up the tree (see Figure 5.2). In a **min-heap**, the **min-heap property** specifies that for every node i other than the parent

$$A[\textsc{Parent}(i)] \;\leq\; A[i]$$

i.e., the values of the nodes decrease as you climb the tree. The theories of max-heaps and min-heaps are the same, except with inequalities reversed. Since the heapsort algorithm in the next section uses max-heaps, we will primarily focus on max-heaps. However, you should always be aware of when we are talking about

max-heaps and when we are talking about min-heaps. Sometimes we will say "heaps" when we are referring to either max-heaps or min-heaps.

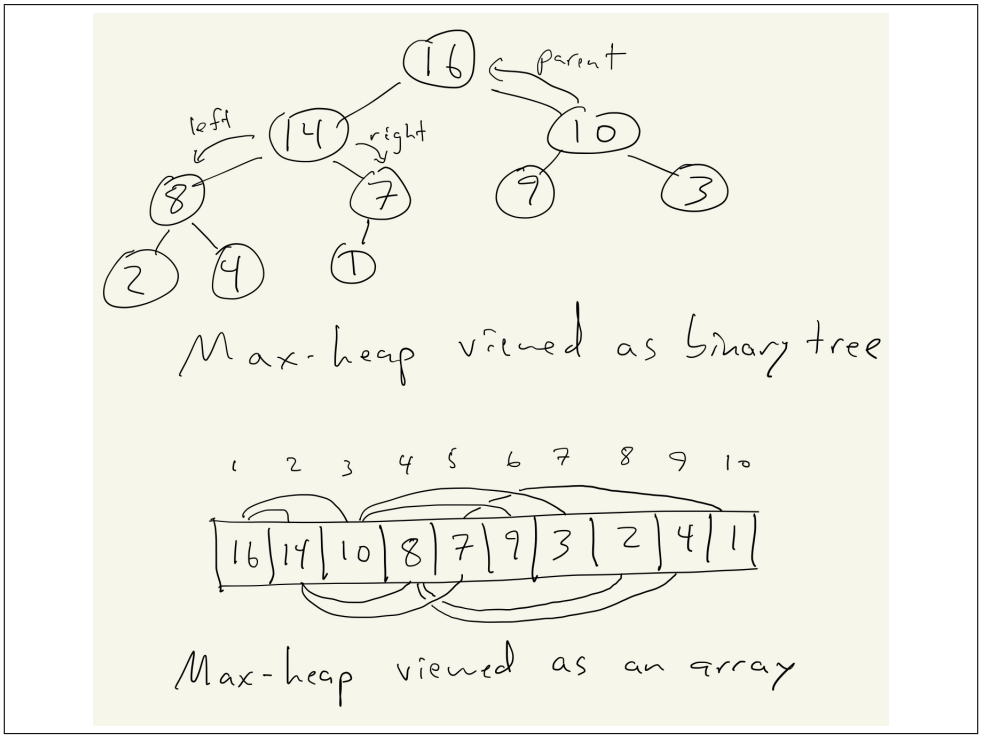


FIGURE 5.2. Here we illustrate a max-heap in two ways. First as a binary tree. Second as an array. The tree has height 3, the node 8 (corresponding to $A[4]$) has height 1.

Viewing a heap as a tree, we define the **height** of a node to be the number of edges on the longest simple downward path from the node to a leaf, and we define the **height** of the heap to be the height of the root. Since heaps are nearly-complete binary trees, the height of a heap with n elements is $\Theta(\lg n)$.

Maintaining the heap property. Suppose we have a heap which might not yet be a max-heap. How do we turn it into a max-heap? The MAX-HEAPIFY procedure below accomplishes this. It takes as input a heap A and a node i and arranges that the sub-heap with root i becomes a max-heap. A major assumption of MAX-HEAPIFY is that the sub-heaps with roots LEFT(i) and RIGHT(i) are already max-heaps, but perhaps the max-heap property is violated at node i. In other words, we will only use MAX-HEAPIFY at a node i if we are certain that the max-heap property is satisfied everywhere below node i

MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7        largest = r
8  if largest ≠ i
9        exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

The procedure MAX-HEAPIFY(A, i) works as follows (see also Figure 5.3):

(1) MAX-HEAPIFY(A, i) takes as input a heap A and a node i.
(2) Lines 1-7 checks if either of i's children nodes have a larger value than $A[i]$ (which would be a violation of the heap property).
(3) In lines 8-10, if there is a violation of the heap property with the left child, then we switch the values of node i with its left child and then recursively call MAX-HEAPIFY. If there is no violation with the left child and there is a violation with the right child (or if there is a violation with the left child, and the right child is bigger than the left child), then we switch the values of node i with the right child instead and recursively call MAX-HEAPIFY. Otherwise we do nothing.

The correctness of MAX-HEAPIFY can be argued by induction, using the assumption that the heaps with roots LEFT(i) and RIGHT(i) are already max-heaps. The running time analysis is as follows:

Proposition 5.1.1. *The running time of* MAX-HEAPIFY *on a heap of size n is $O(\lg n)$. Alternatively, the running time on a heap of height h is $O(h)$.*

PROOF. Let $T(n)$ be the running time of MAX-HEAPIFY on a heap of size n. Lines 1-8 take $\Theta(1)$ amount of time. If lines 9-10 are run, then this takes the amount of time to run MAX-HEAPIFY on the heap rooted at one of the children. The worst case happens when the left subtree is as big as possible compared to the right subtree, and this can happen when the bottom row of the original heap is half-full. I.e., the left subtree will have an entire extra full row than than the right subtree. In which case, the left subtree will be at most size $2n/3$. Thus the running time satisfies the inequality

$$T(n) \ \leq \ T(2n/3) + \Theta(1).$$

By the Master Theorem, we conclude that $T(n) = O(\lg n)$. □

Building a heap. We have already seen that MAX-HEAPIFY can help us fix a single violation of the max-heap property, but it can't by itself turn a non-max-heap into a max-heap. To do this, we have to iteratively call MAX-HEAPIFY from the bottom up on all non-leaf nodes as follows:

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3        MAX-HEAPIFY(A, i)
```

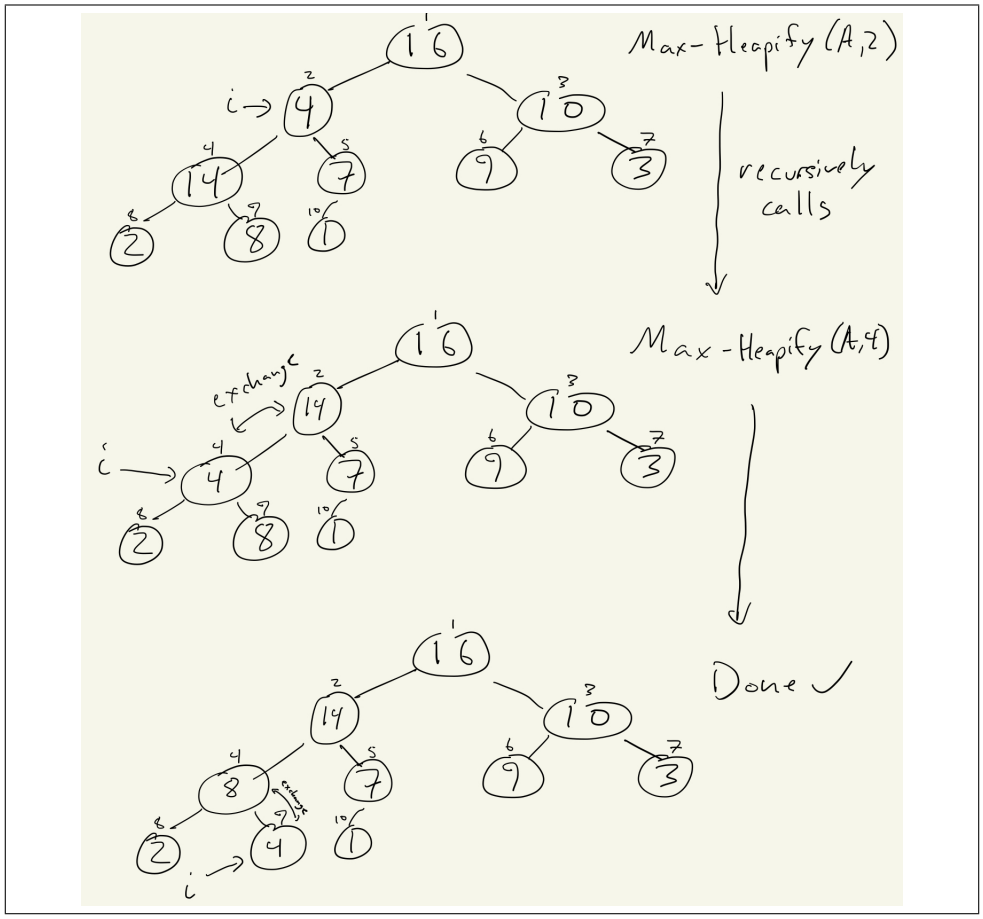


FIGURE 5.3. Here we call MAX-HEAPIFY$(A, 2)$. First it exchanges $A[2]$ with $A[4]$ (its left child). Then it recursively calls MAX-HEAPIFY$(A, 4)$, which exchanges $A[4]$ with $A[9]$ (its right child). Then it is finished and the heap with root 2 is now a max-heap.

BUILD-MAX-HEAP works as follows (see Figure 5.4):

(1) We assume that we want the entire array to be turned into a max-heap, thus in line 1 we set $A.\mathit{heap\text{-}size}$ to $A.\mathit{length}$.
(2) It is a fact that in a heap of size n, the nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are all leaves. Therefore these nodes are already the roots of max-heaps (of size 1), so we do not need to max-heapify these nodes.
(3) Thus, going from bottom to top, the first node we need to max-heapify is $\lfloor A.\mathit{length}/2 \rfloor$. In lines 2-3, we iteratively call MAX-HEAPIFY on the nodes $\lfloor A.\mathit{length}/2 \rfloor, \ldots, 1$, in descending order.

Here is a proof of correctness for BUILD-MAX-HEAP:

Proposition 5.1.2. *Given an array A, after* BUILD-MAX-HEAP(A) *runs, A is a max-heap.*

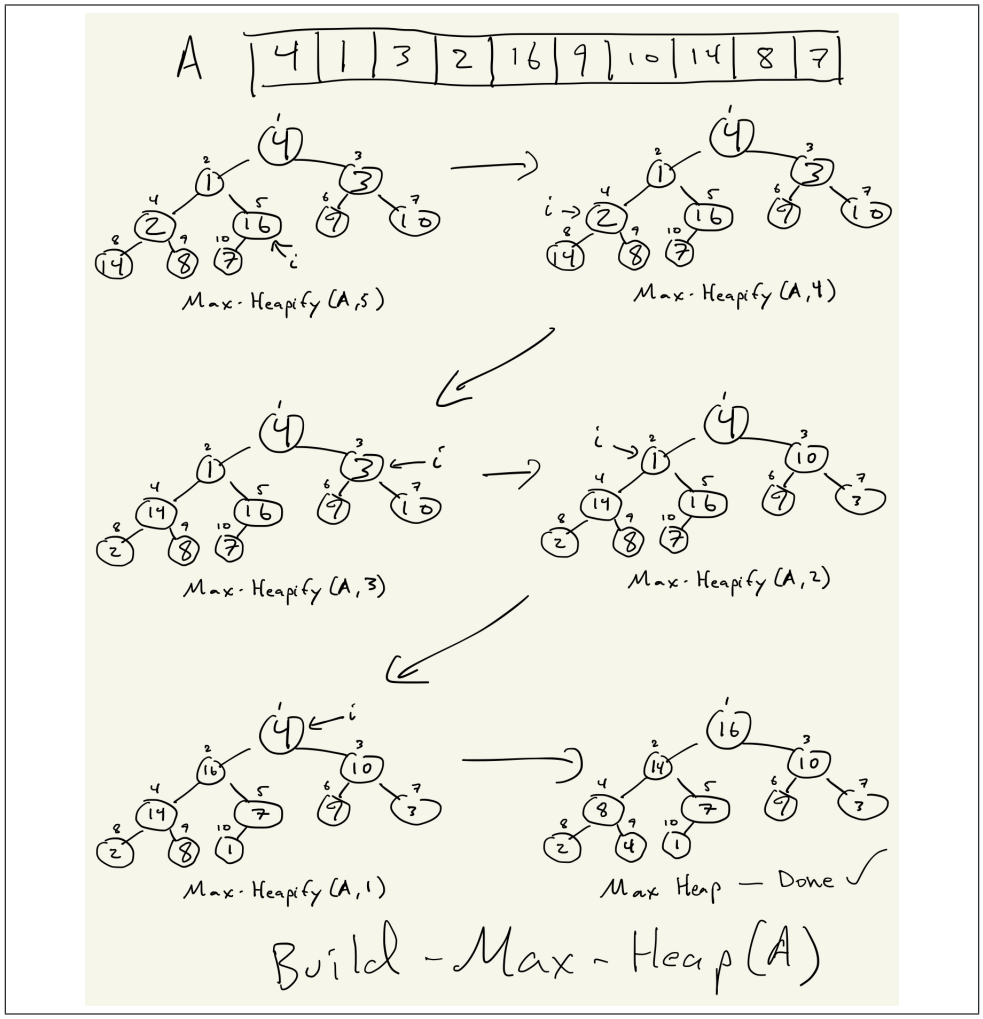


FIGURE 5.4. Here we call BUILD-MAX-HEAP on the array $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$. It iteratively calls MAX-HEAPIFY(5), MAX-HEAPIFY(4), $\ldots$, MAX-HEAPIFY(1).

PROOF. Let $n = A.\mathit{length} = A.\mathit{heap\text{-}size}$. We will prove the following loop invariant:

> (Loop Invariant) At the end of each time line 2 runs, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

(Initialization) The first time line 2 runs, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, so automatically is the root of a max-heap.

(Maintenance) Suppose line 2 has just run and the current value of i is $i = i_0$, where $\lfloor n/2 \rfloor \leq i_0 \leq 1$. Furthermore, assume we know that loop invariant is true at this point, i.e., we know the nodes $i_0 + 1, i_0 + 2, \ldots, n$ are all roots of a max-heap. Next in line 3 we call MAX-HEAPIFY(A, i_0), which we can do because LEFT(i_0) and RIGHT(i_0) (both $> i_0$) are roots of max-heaps by the loop invariant assumption. This makes i_0 the root of a max-heap. Furthermore, it preserves the property that

$i_0 + 1, \ldots, n$ are roots of max-heaps. Next we go back to line 2 and decrement i to $i = i_0 - 1$. Thus $i + 1 = i_0, \ldots, i + 2, \ldots, n$ are all roots of max-heaps so the loop invariant remains true.

(Termination) Now that we know the loop invariant is true, upon termination we have $i = 0$, which means the nodes $1, 2, \ldots, n$ are all roots of max-heaps. In particular, the node 1 is the root of a max-heap, which means that A is now a max-heap. □

As for the running time, we can get a loose upper-bound by noting that each call to MAX-HEAPIFY takes $O(\lg n)$ times, and we make $O(n)$ total calls. Thus BUILD-MAX-HEAP runs in $O(n \lg n)$ time. However, not every call to MAX-HEAPIFY operates on a heap of size n: most of the heaps are much smaller. Thus we can actually do much better than $O(n \lg n)$:

Theorem 5.1.3. *Given an array A of size n, the running time of* BUILD-MAX-HEAP(A) *is $\Theta(n)$.*

PROOF. It is clear that it is $\Omega(n)$, due to the **for** loop. For the upper bound, note that the height of an n-element heap is $\lfloor lgn \rfloor$. On HW4 we'll show that at a given height h there are at most $\lceil n/2^{h+1} \rceil$ nodes at that height. Thus we can bound the running time above by

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \;=\; O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Furthermore, we can bound the summation using the formula from Exercise 1.6.3:

$$\sum_{j=0}^{m} j x^j \;=\; \frac{mx^{m+2} - (m+1)x^{m+1} + x}{(x-1)^2}$$

Thus, with $x := 1/2$ we have

$$\sum_{h=0}^{\lfloor \lg n \rfloor} h(1/2)^h \;\leq\; \sum_{h=0}^{\infty} h(1/2)^h \;=\; \frac{1/2}{(1-1/2)^2} \;=\; 2.$$

Thus

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \;=\; O(n2) \;=\; O(n).$$ □

5.2. Heapsort

One of the nice things about a max-heap is that we know the largest element in the heap is always $A[1]$. We can leverage this observation into an efficient sorting algorithm called *heapsort*. The algorithm is as follows:

HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT works as follows (see Figure 5.5):

(1) First we take our array and turn it into a max-heap by calling BUILD-MAX-HEAP(A).
(2) Now we know that the largest element in the array is $A[1]$, which belongs at the end, so we exchange $A[1]$ with $A[n]$.
(3) We want to fix $A[n]$ in place and not operate on it anymore, so we decrease $A.\textit{heap-size}$ by one.
(4) Now every node in our heap, except node 1 is the root of a max-heap, so we are free to call MAX-HEAPIFY$(A, 1)$, which turns the heap into a max-heap.
(5) Now we know $A[1]$ is the largest element in our heap, second largest overall (after $A[n]$), so we replace $A[1]$ with $A[n-1]$. Then we fix the element $A[n-1]$ in place by decreasing $A.\textit{heap-size}$ once-again.
(6) We call MAX-HEAPIFY$(A, 1)$ to turn the remaining heap into a max-heap.
(7) The process repeats until all elements have been fixed in place in their correct order and our heap disappears.

To compute the running time of HEAPSORT on an array of size n, note that BUILD-MAX-HEAP(A) takes $O(n)$ amount of time. Then each of the $n-1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time. Thus an upper-bound on the running time is $O(n \lg n)$.

Remark 5.2.1. In general HEAPSORT combines the best features of MERGE-SORT and INSERTION-SORT:

(1) HEAPSORT has worst-case running time of $O(n \lg n)$ just like procMerge-Sort, which is better than the $O(n^2)$ worst-case running time of INSERTION-SORT.
(2) HEAPSORT sorts *in-place*, i.e., it only requires a constant amount of memory beyond what it takes to store the array A. Unlike MERGE-SORT, HEAPSORT does not need to make entire copies of the array A in order to sort A. This is a feature which HEAPSORT has in common with INSERTION-SORT, which also sorts *in-place*.

5.3. Priority queues

In this section we present another application of heaps: *priority queues*. Priority queues come in two forms: max-priority queues and min-priority queues. We will focus on max-priority queues which are based on max-heaps.

A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called a **key**. A **max-priority queue** must be able to do the following operations:

- INSERT(S, x), which inserts the element x into the set S, which is equivalent to the operations $S = S \cup \{x\}$.
- MAXIMUM(S), which returns the element of S with the largest key.
- EXTRACT-MAX(S), which removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k), which increases the value of x's key to the new value k, which is assumed to be at least as large as x's current key value.

A priority queue has many applications, for instance keeping track of jobs on a shared computer or printer. Each time a job is finished, the system needs to know which job to do next, and a priority queue can answer this question by producing the job with the largest key (i.e., the "highest priority"). New jobs getting added

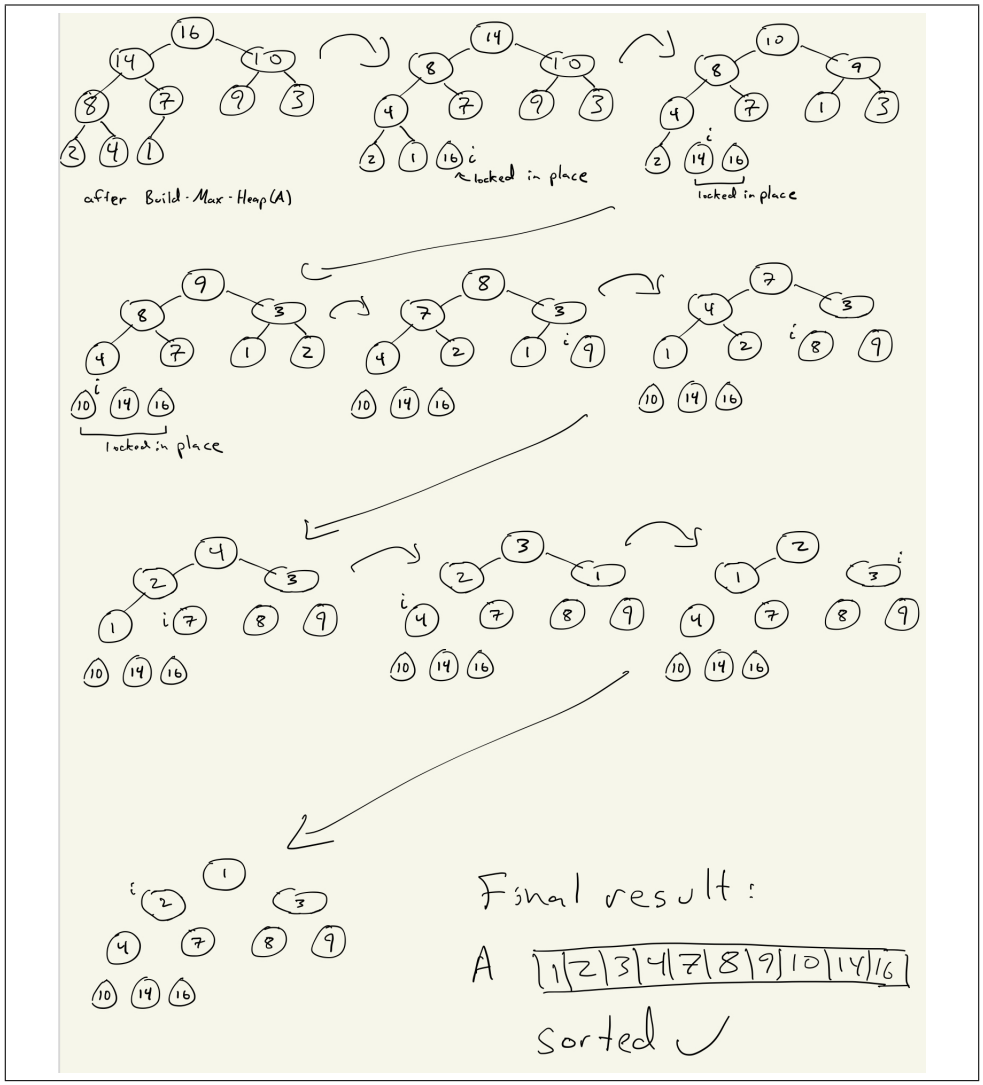


FIGURE 5.5. Here we call HEAPSORT on a certain array A. First we apply BUILD-MAX-HEAP(A) to get a max-heap. Then we iteratively take the max and put it at the end of the heap, make the heap one element smaller, then call MAX-HEAPIFY on the smaller heap to find the next max.

might have varying levels of priority, so we need to be able to insert new jobs and change the priorities of existing jobs. Priority queues are ideal for this type of thing.

Since heaps are good at keeping track of the largest element in a set, we will implement our priority queues as heaps. Assuming A is a max-heap, the following implementation of MAXIMUM runs in $\Theta(1)$ time:

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Often for priority queues, we like to simultaneously obtain the maximum element and remove it from the queue (for instance, a printer wants to process the highest-priority job while also removing that job from consideration for future jobs it should process). The following implements EXTRACT-MAX:

HEAP-EXTRACT-MAX(A)

```
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

HEAP-EXTRACT-MAX works as follows:

(1) First we check if the heap is empty. If it is, then we return an error.
(2) Otherwise, we store the max element $A[1]$ as max.
(3) Now the heap no longer needs to store $A[1]$, so we make $A[A.heap\text{-}size]$ the new $A[1]$, decrease $A.heap\text{-}size$ by one and run MAX-HEAPIFY$(A, 1)$ to maintain the max-heap property.
(4) Finally we return the original max element max.

Since lines 1-5 and 7 run in constant time and line 6 calls MAX-HEAPIFY$(A, 1)$ which runs in $O(\lg n)$ time, the algorithm HEAP-EXTRACT-MAX(A) runs in $O(\lg n)$ time.

The following implements INCREASE-KEY:

HEAP-INCREASE-KEY(A, i, key)

```
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

HEAP-INCREASE-KEY works as follows (see Figure 5.6):

(1) We call HEAP-INCREASE-KEY(A, i, key) where i is the element in the priority queue whose key we wish to increase, and key is the desired value.
(2) In lines 1-2 we return an error if key is smaller than i's existing key value.
(3) Otherwise in line 3 we replace i's key value with key.
(4) Since the max-heap property might not hold anymore, in lines 4-6 we bubble $A[i]$ upwards in the heap until it is in the correct spot and we have a max-heap again.

Since lines 1-3 take constant time, and the **while** loop in lines 4-6 runs a number of times bounded by the height of the tree, the running time of HEAP-INCREASE-KEY is $O(\lg n)$.

We implement INSERT as follows:

MAX-HEAP-INSERT(A, key)

```
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

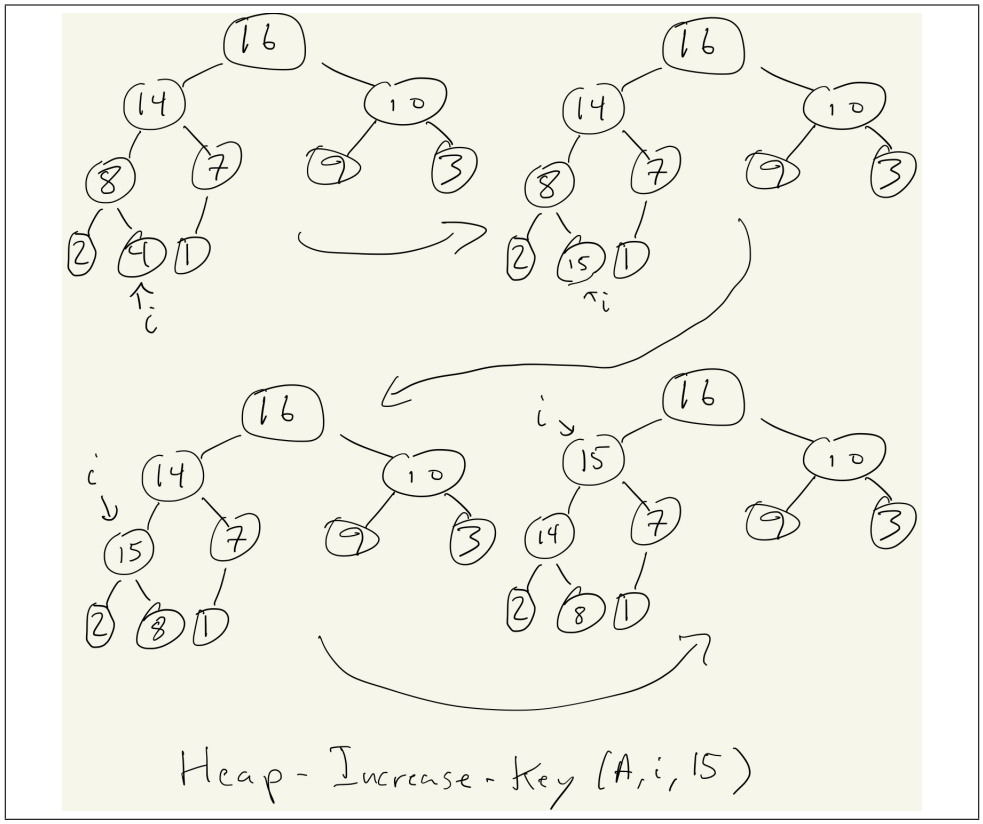


FIGURE 5.6. Here we call HEAP-INCREASE-KEY$(A, i, 15)$. First this changes the key $A[i] = 4$ to $A[i] = 15$. Next it bubbles up the key 15 until the max-heap property is satisfied.

MAX-HEAP-INSERT works as follows:

(1) We call MAX-HEAP-INSERT(A, key) where *key* is the new value we want to put into the priority queue.
(2) In line 1 we increase the heap size by 1, since we will be growing the heap by 1.
(3) In line 2 we set the last element in the heap to have value $-\infty$. The use of the sentinel $-\infty$ guarantees that $key > A[A.heap\text{-}size]$, so that line 3 won't return an error.
(4) In line 3 we increase the value of $A[A.heap\text{-}size]$ to be *key*. Since we are calling HEAP-INCREASE-KEY, we know that afterwards A retains the max-heap property.

Since lines 1-2 run in constant time and line 3 runs in $O(\lg n)$ time, the overall running time of MAX-HEAP-INSERT is $O(\lg n)$.

5.4. Stacks and queues

Stacks and queues are data structures which store a set S of keys and implement the following two operations:

- INSERT(S, x), which inserts the new key x into the set S.

- DELETE, which removes and returns a prespecified element from the set S and returns it.

Stacks. A **stack** data structure always deletes and returns whichever element was *most recently* inserted into the stack. In other words, stacks operate under a **last-in, first-out** (or **LIFO**) policy. A stack S can be implemented as an array $A[1..n] = \langle a_1, \ldots, a_n \rangle$, with an additional attribute $S.top$ so that the subarray $S[1..S.top]$ represents which elements are actually in the stack.

If S is a stack with $S.top = 0$, then we say that the stack is **empty** (even though the array S might not be empty). The following pseudocode checks if S is empty:

STACK-EMTPY(S)

```
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

The INSERT operation of a stack is often called PUSH. This is because we imagine a stack is like a physical stack, like a spring-loaded stack of cafeteria plates: to add a new plate we *push* it down onto the top of the stack. The following implements PUSH(S, x):

PUSH(S, x)

```
1  S.top = S.top + 1
2  S[S.top] = x
```

PUSH(S, x) first increases the stack size by 1, and then assigns the key at the top of the stack to be x. If $S.top$ exceed $S.length = n$, then we say that the stack **overflows**, which is an error (we don't check for stack overflow[1] in this implementation).

The DELETE operation of a stack is often called POP. This is because in our analogy of a physical stack (i.e., spring-loaded stack of cafeteria plates), when we grab the top plate from the stack it *pops out* a little due to the spring-loading. Here is our implementation of POP(S):

POP(S)

```
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top - 1
4       return S[S.top + 1]
```

POP(S) works as follows:

(1) First we check if the stack S is empty. If it is, then it doesn't have any elements to pop, so we get an **underflow** error. This is what lines 1-2 checks for.
(2) Otherwise, we decrease the size of the stack by 1, and return the old top of the stack.
(3) There is no need to actually delete $S[S.top + 1]$ from the array: it is no longer considered part of the stack and it will get overwritten in the future when we push more elements on to the stack.

[1] As it so happens, *stack overflow* is also the name of a popular website where people can ask and answer programming questions; see `https://en.wikipedia.org/wiki/Stack_Overflow`.

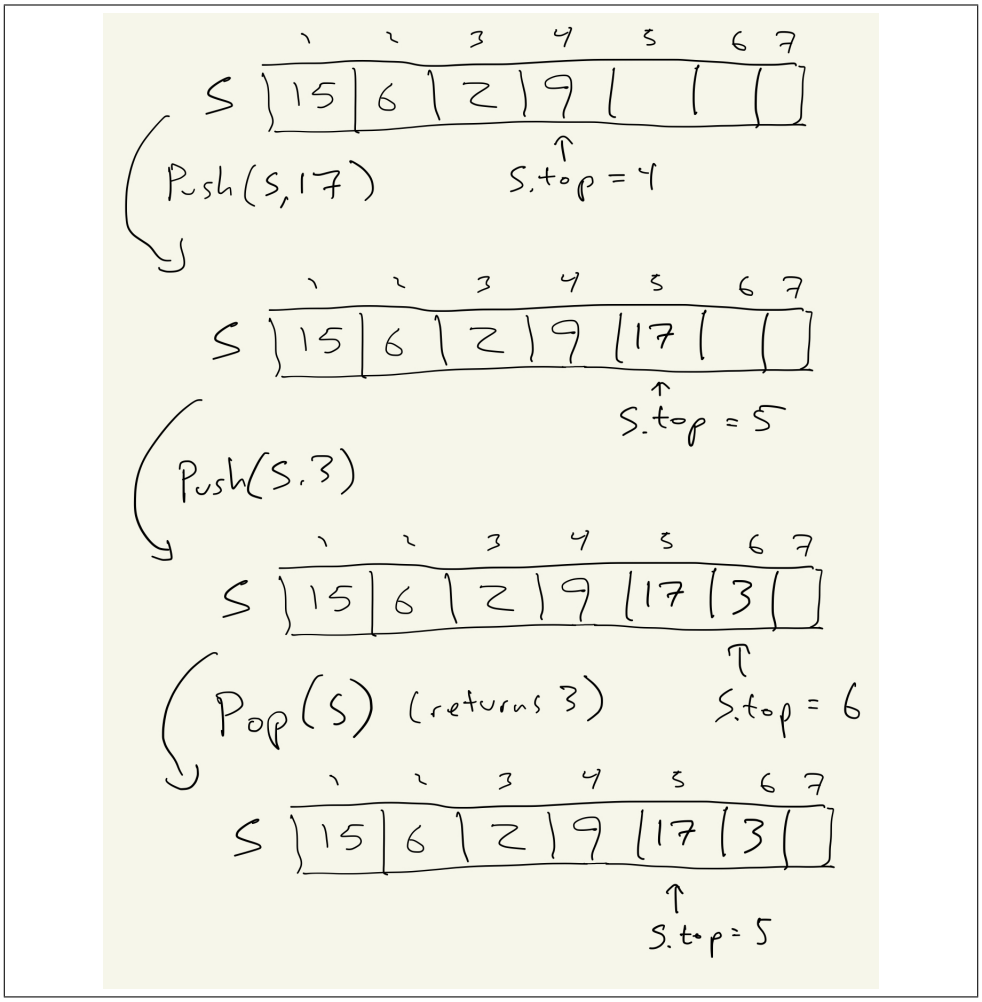


FIGURE 5.7. We start with a nonempty stack S. Then we call in succession PUSH$(S, 17)$, PUSH$(S, 3)$, and POP(S), which returns 3. At the end, 3 is still part of the *array* S, but not part of the *stack* S.

All three of the operations STACK-EMPTY, PUSH, and POP run in constant $\Theta(1)$ time. We illustrate the PUSH and POP operations in Figure 5.7.

Queues. A **queue** is like the opposite of a stack. In a stack, the element that has most recently inserted into the stack is always the first to leave. In a queue, the element which has been in the queue the longest is the first to get deleted. In other words, queues operate under a **first-in, first-out** (or **FIFO**) policy. A queue is like a line of customers waiting to check out at a store: you always join the back of the queue, and it is your turn to check out after everyone who was in the queue before you has already checked out.

One way to implement a queue is with an array Q. Since we wish to avoid constantly shifting all of our keys every time we delete someone, an array-based queue has a

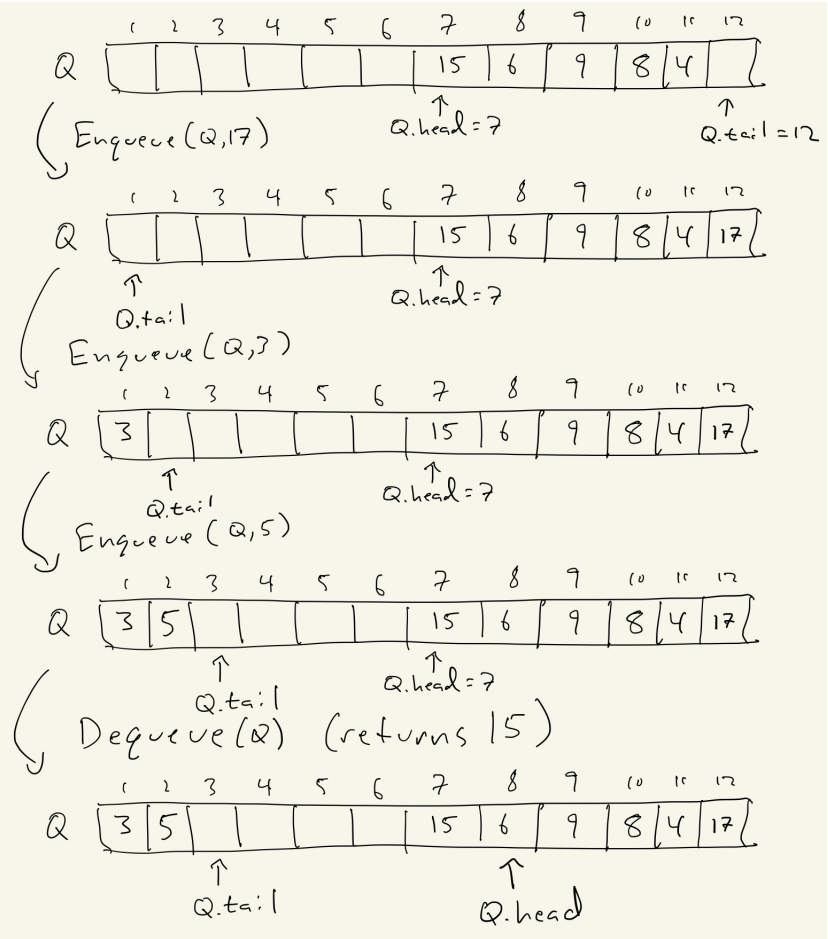


FIGURE 5.8. We start with a nonempty queue and call in succession ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, ENQUEUE(5) which adds 17, 3, and 5 to the back of the queue, in that order. Then we call DEQUEUE(Q) which returns 15 and advances the head of the queue to the next element after 15.

somewhat funny implementation where the queue *wraps around* the end of the array back to the front when it is long enough (see Figure 5.8). Queues have two relevant attributes:

- $Q.head$, which points to the first element in the queue (the head of the line, the next element to get deleted), and
- $Q.tail$, which points to the spot immediately after the last element. This is the spot where the next element which gets added to the queue will go. Thus the array entry $Q[Q.tail]$ is not part of the queue, unless the queue is full.

The operation INSERT(Q, x) for queues is called ENQUEUE(Q, x). It gets implemented as follows:

ENQUEUE(Q, x)

```
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

ENQUEUE works as follows:

(1) Since $Q.tail$ points to where the next element to be added should go, in line 1 we assign $Q[Q.tail] = x$.
(2) Next we need to advance $Q.tail$ to be the spot where the next new element should go. If $Q.tail$ is currently the rightmost spot in the array, then we need to wrap around and have $Q.tail$ point to the front of the array. This is what lines 2-3 do.
(3) Otherwise we advance $Q.tail$ one spot to the right in the array.

The operation DELETE for queues is called DEQUEUE. It gets implemented as follows:

DEQUEUE(Q)

```
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

DEQUEUE works as follows:

(1) Since we want to return the head of the queue, first we set $x = Q[Q.head]$, the item which is to be deleted.
(2) Next we have to advance $Q.head$ to be the next element in the queue. If $Q.head$ currently is the last spot in the array, then we need to wrap around to the front (this is what lines 2-3 do). Otherwise we advance $Q.head$ one entry to the right (line 4).
(3) Finally in line 5 we return x. There is no need to actually delete x from the array at this point: x is no longer part of the queue and it will get overwritten by future calls to ENQUEUE.

Note that both ENQUEUE and DEQUEUE run in constant time $\Theta(1)$. When $Q.head = Q.tail + 1$ or both $Q.head = 1$ and $Q.tail = Q.length$, then the queue is full. In this case, attempting to enqueue another element will result in an *overflow* error. Also, if $Q.head = Q.tail$, then the queue is empty and attempts to dequeue will result in an *underflow* error. In our pseudocode we did not check for these errors, although this is something you should do in any actual implementation.

CHAPTER 6

Dynamic programming

Dynamic programming algorithms is a lot like divide-and-conquer, where we solve problems by combining solutions to subproblems. However, divide-and-conquer typically deals with *disjoint* subproblems, whereas dynamic programming involves *overlapping* subproblems, i.e., when subproblems share subproblems. For instance, in our recursive FIBONACCI algorithm from Section 1.5, we saw that to compute FIBONACCI(4), we had to solve the subproblems FIBONACCI(3) and FIBONACCI(2), however both of these involved solving the same subproblem FIBONACCI(1) multiple times. A dynamic programming solution to computing Fibonacci numbers is to solve each subproblem once and then store the solution for future use. This was exactly what FIBONACCIFAST did. In fact, FIBONACCIFAST was our first example of *dynamic programming*. In fact, many of the programming exercises on the homework admit efficient dynamic programming solutions.

Dynamic programming is typically useful for **optimization problems**, i.e., problems where there are many different possible solutions, and we seek the solution which minimizes or maximizes some value. Dynamic programming typically involves the following three or four steps:

(1) Characterize the structure of an optimal solution.
(2) Recursively define the value of an optimal solution.
(3) Compute the value of an optimal solution, typically in a bottom-up fashion.
(4) Construct an optimal solution from computed information. This step is sometimes optional, depending on whether you want to know just the *value* of an optimal solution, or a specific optimal solution which gives the optimal value.

6.1. Rod cutting

In this section we consider our first optimization problem which has an efficient dynamic programming solution: the *rod-cutting problem*.

Consider the following scenario. We work for a company which sells rods of steel. We have a rod of steel of a certain length which we can cut into smaller pieces. Depending on the length of the piece it will sell for a certain amount. For example, suppose have a rod of length n inches and the following sample table determines the price we can sell a piece of each length:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	9	9	10	17	17	20	24	30

Then if $n = 5$, we could choose to not cut the rod and sell it for \$10, although it would be better to cut it into 3 pieces of lengths $2 + 2 + 1$ in order to sell it for

$\$5 + \$5 + \$1 = \$11$. The rod-cutting problem is to determine how to cut our rod in order to maximize our total revenue. More specifically:

> **Rod-cutting problem**: Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtained by cutting up the rod and selling the pieces.

Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

One of the rules of this problem is that we only allow cuts at integer lengths, so fractions of inches are not allowed. Still, given a rod of length n, there are $n-1$ places we could potentially cut it, leading to 2^{n-1} potential configurations. Thus a brute-force solution where we consider all possible ways of cutting the rod is not an option (as it will run in exponential time).

There is one helpful observation we can make though. Let r_k be the maximal possible revenue obtained by cutting a rod of length k, for $k = 0, \ldots, n$ (with $r_0 := 0$). If we were to commit to having a piece of length k (for $k \leq n$), then the maximum possible revenue in this case is $p_k + r_{n-k}$. This is because we get the revenue p_k for our piece of length k, plus we would cut the remaining rod of length $n-k$ into pieces to maximize the revenue from the remaining rod. Since our optimal solution must start with a piece of some definite length i (for $1 \leq i \leq n$), we can conclude

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \tag{\dagger}$$

This gives us a way of determining the optimal solution for rods of length n based on the optimal solutions of rods of length $1, \ldots, n-1$. This is an instance of what we call **optimal substructure**, i.e., the optimal solution must incorporate optimal solutions to related subproblems (this is a recurring theme in dynamic programming).

Recursive top-down implementation. Here is an algorithm which implements (†) in a top-down, recursive manner:

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Here is an explanation of CUT-ROD:

(1) CUT-ROD takes as input an array $p[1..n]$ of prices, and a length n.
(2) In lines 1-2 we check of $n = 0$, if it is, then we return the optimal value 0.
(3) In line 3 we initialize the variable $q = -\infty$. q represents the optimal revenue we are guaranteed so far. We initialize it with the sentinel value $-\infty$ so that it will immediately get reassigned the first time line 5 is run.
(4) In the **for** loop lines 4-5, we implement formula (†). Each time line 5 is run this performs a recursive call to CUT-ROD$(p, n-i)$, in order to compute r_{n-i}.
(5) After the **for** loop, the value of q is $q = r_n$, and we return it in line 6.

As it turn out, CUT-ROD is rather inefficient:

Proposition 6.1.1. CUT-ROD(p, n) *makes* 2^n *recursive calls to* CUT-ROD *(including the initial call). Thus the running time is* $\Omega(2^n)$.

PROOF. Suppose $n = 0$. Then the program returns in line 2 so there are no additional calls made to CUT-ROD beyond the initial call. Thus $T(0) = 1$. Next, suppose $n \geq 1$. Then we are guaranteed 1 call from the initial call, plus all the calls performed in line 5. Thus

$$T(n) \;=\; 1 + \sum_{i=1}^{n} T(n-i) \;=\; 1 + \sum_{i=0}^{n-1} T(i).$$

We will prove inductively that $T(n) = 2^n$. This is clear for $n = 0$. Now suppose $n \geq 1$ and we know $T(m) = 2^m$ for every $m < n$. Note that

$$T(n) \;=\; 1 + \sum_{i=0}^{n-1} T(i) \;=\; 1 + \sum_{i=0}^{n-1} 2^i \;=\; 1 + \frac{1-2^n}{1-2} \;=\; 1 + (2^n - 1) \;=\; 2^n. \quad \square$$

Using dynamic programming. Now that we've observed the straightforward recursive implementation to be inefficient, we will use dynamic programming to convert this algorithm into an efficient one. The reason CUT-ROD is so inefficient is that it solves the same subproblem multiple times. The dynamic programming solution to this is to solve each subproblem only once, and then saving its solution for looking up later, rather than recomputing it. Thus we will use additional memory to save computational time, a so-called **time-memory trade-off**.
In general there are two ways to implement a dynamic-programming approach to CUT-ROD (and other dynamic programming problems). The first approach is called **top-down with memoization**. This approach is a modification of the recursive approach of CUT-ROD, except after the first time we solve each subproblem we store its value. Then, each time we wish to solve a subproblem, we first check to see if we solved it before. If so, we return the value of the solution that we've stored, otherwise. If we haven't solved it before, then we proceed with solving it (and subsequently storing it). In this case, we say that the recursive algorithm has been **memoized**, i.e., it now remembers the solutions to subproblems it has solved previously.

MEMOIZED-CUT-ROD(p, n)
1 let $r[0\,..\,n]$ be a new array
2 **for** $i = 0$ **to** n
3 $\quad r[i] = -\infty$
4 **return** MEMOIZED-CUT-ROD-AUX(p, n, r)

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

The procedure works as follows:

(1) MEMOIZED-CUT-ROD creates a new array $r[0\,..\,n]$ of length $n+1$ (in line 1). This will serve as our storage device for when we solve subproblems. In lines 2-3 we initialize all the values of r to be $-\infty$, which is to represent that we don't know the answer yet to any of these subproblems. Then we pass all three objects p, n, and r to MEMOIZED-CUT-ROD-AUX to do the real work.
(2) In MEMOIZED-CUT-ROD-AUX, our job is to return the maximum revenue r_n.
(3) If $r[n] \geq 0$, then this means that subproblem has already been solved, so we return $r[n] = r_n$.
(4) Otherwise, in line 3 we check if $n = 0$. If it is, then we know the solution is 0, so we set $q = 0$ in line 4, set $r[n] = q$ in line 8 (so that we've stored the solution to this subproblem), and then we return the value of the solution in line 9.
(5) Otherwise if $n \geq 1$, then we go to line 5, set $q = -\infty$. Then in lines 6-7 we find the max revenue by iterating over all subproblems just like in CUT-ROD. Except now we recursively call MEMOIZED-CUT-ROD-AUX.
(6) After we find the max revenue in lines 6-7, we store it as $r[n]$ in line 8 and then return it.

The second dynamic programming approach is the **bottom-up method**. In this method, we do not rely on a recursive structure. Instead, we start with the smallest subproblem first and iteratively solve subproblems from smallest to largest, until we've solved the original problem for size n. Here is an implementation for the bottom-up method:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0 .. n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = −∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j − i])
7       r[j] = q
8  return r[n]
```

Here is how BOTTOM-UP-CUT-ROD works:

(1) First we create a new array $r[0\,..\,n]$ and set $r[0] = r_0 = 0$ in lines 1-2.

(2) The **for** loop in lines 3-7 iteratively solves the subproblems $r_1, r_2, \ldots, r_n$ in increasing order.
(3) For a given $j = 1, \ldots, n$, in line 4 we initialize $q = -\infty$, to represent we don't have the max revenue for subproblem j yet. Then in the **for** loop of lines 5-6, we solve the subproblem of size j. In line 7 we store the solution r_j as $r[j]$.
(4) Finally in line 8, our entire array r is full with the answers to all subproblems, and we return $r[n] = r_n$, which was the value originally desired.

Proposition 6.1.2. *The running times of* Memoized-Cut-Rod *and* Bottom-Up-Cut-Rod *are both* $\Theta(n^2)$.

Proof sketch. In Memoized-Cut-Rod, we need to actually solve each subproblem once, and each time we solve a subproblem of size i, we need to iterate a **for** loop i times. There are n subproblems, so we obtain a triangular sum $\sum_{i=1}^{n} i$ which yields $\Theta(n^2)$.

In Bottom-Up-Cut-Rod, we have a double **for** loop which also yields a triangular sum and hence $\Theta(n^2)$. □

Subproblem graphs. In order to design a dynamic programming algorithm, it is important to have a good understanding of what the space of subproblems looks like and how these subproblems depend on each other. The **subproblem graph** displays exactly this information. A subproblem graph is a graph where the vertices represent the subproblems, and the directed edges represent the dependencies between the subproblems.

In Figure 6.1 we show the full recursion tree for Cut-Rod$(p, 4)$, which illustrates all of the recursive calls to Cut-Rod for smaller subproblems. In particular, this shows how many time each subproblem gets solved, and which subproblems depend on other subproblems. Below that, we show the subproblem graph. Ultimately, there are only 5 subproblems, $n = 0, 1, 2, 3, 4$, and n depends on m iff $m < n$. It is good to have this subproblem graph in mind when designing the algorithm because it shows which subproblems must be solved before other subproblems are solved. Indeed, Figure 6.1 shows that in order to solve the $n = 4$ problem, we might as well start with $n = 0$, then solve $n = 1$, $n = 2$, $n = 3$, and then $n = 4$. In fact, this is exactly how Bottom-Up-Cut-Rod works.

In general, the subproblem graph might not be as simple as the one for the rod-cutting problem. For instance, instead of the subproblems being arranged in a one-dimensional line, they might be configured as some higher-dimensional lattice, depending on how many parameters are needed to specify a subproblem. The size and complexity of the subproblem graph helps determine the running-time complexity of the dynamic programming algorithm. Since we only need to solve each subproblem once, and each subproblem depends on the number of edges leaving that subproblem, the running time is typically linear in the number of vertices and edges present in the subproblem graph.

Reconstructing a solution. All of our algorithms so far only return the *value* of an optimal solution. In practice, however, we would also be interested in knowing a particular optimal solution as well. For instance, we would like to know precisely how to cut up our rod in order to achieve the maximum revenue, in addition to just knowing what that maximum revenue would be. Fortunately, this additional

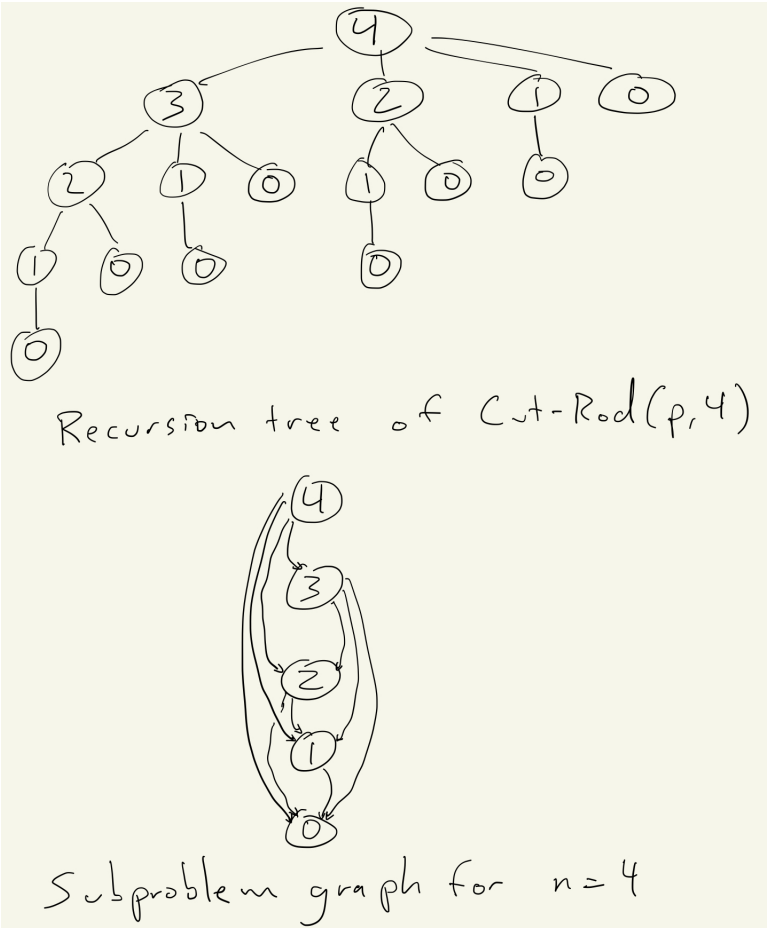


FIGURE 6.1. First we show the full recursion tree for CUT-ROD$(p, 4)$. Next we show the subproblem graph for $n = 4$, which is like a collapsed version of the recursion tree.

information is usually something we can keep track of in an extended version of our algorithms.

For instance, here is an extension of CUT-ROD which keeps track of the size of the cuts we need to do to achieve an optimal solution:

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[1..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4       q = -∞
 5       for i = 1 to j
 6            if q < p[i] + r[j - i]
 7                 q = p[i] + r[j - i]
 8                 s[j] = i
 9       r[j] = q
10  return r and s
```

EXTENDED-BOTTOM-UP-CUT-ROD works as follows:

(1) In general the algorithm runs the same as BOTTOM-UP-CUT-ROD except that it also maintains an array $s[1..n]$. The interpretation of this array is: the value of $s[j]$ is the size of the first cut you should make in order to achieve an optimal solution to subproblem $n = j$.
(2) In line 1 we create the array $s[1..n]$. Then in lines 7-8, each time we find an improved solution to subproblem j, we store both the value of the improved solution in line 7, and the size of the piece we should cut which gives us that solution.

You might think that s does not store enough information, since for each value of n it only tells us the size of one cut we should make to achieve an optimal solution, not the sizes of all cuts. However, this is all we really need to know, since the array tells us the size of one cut for *all* n. For instance, suppose we ran EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ and received the following output:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

If we had a piece of length $n = 9$, this information would tell us the following:

(1) $r[9] = 25$, so the max revenue we can get is 25. How do we achieve this?
(2) $s[9] = 3$. This tells us we should first make a cut of size 3. Then the rest of our rod has length $n = 6$.
(3) We need to cut the remaining $n = 6$ rod in an optimal way, since $s[6] = 6$, this tells us that the best thing we can do with a rod of length $n = 6$ is leave it as is.
(4) Therefore an optimal solution is $3 + 6$.

In general, to recover an optimal solution, we just jump backwards through the table making one cut at a time until we're told to leave our remaining piece as is. The following pseudocode implements this recovery:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3       print s[n]
4       n = n - s[n]
```

6.2. Matrix-chain multiplication

The next dynamic programming example we will consider is the matrix-chain multiplication problem. We are given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ and we want to compute the product

$$A_1 A_2 \cdots A_n.$$

We can evaluate this expression using a matrix multiplication algorithm, once we have *fully parenthesized* it to determine in which order do we want to do our multiplications. For instance, we can evaluate $A_1 A_2 A_3 A_4$ in five different ways:

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

Of course, by associativity of matrix multiplication, all five of these evaluations will result in the same matrix. What might be different, however, is how long each evaluation might take. We want to choose which parenthesization will result in the fewest number of scalar multiplications, and hence the fastest running time. We are assuming here that our matrices possibly are of different sizes, and so the order of evaluation does matter.

To understand how many scalar multiplications are necessary, the following algorithm is an implementation of the standard way to multiply two matrices $A \times B$:

MATRIX-MULTIPLY(A, B)

```
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                      for k = 1 to A.columns
8                           c_ij = c_ij + a_ik · b_kj
9       return C
```

According to MATRIX-MULTIPLY, provided that A and B are **compatible** (i.e., the number of columns of A is equal to the number of rows of B, so the matrix multiplication $A \times B$ is valid), the total number of scalar multiplications performed in line 8 is

$$A.rows \times B.column \times A.columns.$$

Thus, if A is a $p \times q$ matrix and B is a $q \times r$ matrix, then to compute $A \times B$ via MATRIX-MULTIPLY we will do pqr total scalar multiplications.

Before we proceed any further, we should convince ourselves that the order of evaluation really does make a difference. Suppose our chain of matrices is $\langle A_1, A_2, A_3 \rangle$, where A_1 is 10×100, A_2 is 100×5, and A_3 is 5×50. The number of scalar multiplications required for $((A_1A_2)A_3)$ is first $10 \cdot 100 \cdot 5 = 5000$ to compute A_1A_2, and then $10 \cdot 5 \cdot 50 = 2500$ to compute $(A_1A_2)A_3$, for a total of 7500 scalar multiplications (because A_1A_2 is 10×5). The number of scalar multiplications required

for $(A_1(A_2A_3))$ is $100 \cdot 5 \cdot 50 = 2500$ to compute A_2A_3, and $10 \cdot 100 \cdot 50 = 50000$ to compute $A_1(A_2A_3)$ for a grand total of 75000 scalar multiplications. Clearly the parenthesization $((A_1A_2)A_3)$ is preferable to $(A_1(A_2A_3))$, and this can be determined from the sizes of the matrices alone, and does not depend on the contents of the matrix.

We now state the problem we are interested in solving:

> The **matrix-chain multiplication problem** is as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in this problem, we are not actually computing the product of these matrices, we are merely determining the parenthesization. So really this problem is a problem about the sequence of dimensions $\langle p_0, p_1, \ldots, p_n \rangle$, and not a problem about the actual matrices $\langle A_1, A_2, \ldots, A_n \rangle$. In practice, if you had to actually compute the product $A_1A_2 \cdots A_n$, then you would first determine an optimal parenthesization, then proceed to evaluate the product in accordance with that parenthesization using MATRIX-MULTIPLY (or some other matrix multiplication algorithm).

Counting the number of parenthesizations. Before we proceed with designing the dynamic programming algorithm, we should first come to grips with how many total parenthesizations there are. Let $P(n)$ denote the total number of parenthesizations for a sequence of n matrices. Then $P(1) = 1$ as a base case. Suppose $n \geq 2$. Then there are $n - 1$ choices for the "top-level" parenthesization. For instance, for each $k = 1, \ldots, n-1$, we can group $A_1 \cdots A_n$ as $(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$, where we further parenthesize each factor $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$ separately. Thus for each k there are $P(k)P(n-k)$ ways to parenthesize the two subfactors. This yields a recurrence:

$$P(n) \;=\; \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

As it so happens, this is a recurrence for a very famous sequence of numbers in mathematics: the **Catalan numbers**[1]. One can easily show that $P(n) = \Omega(2^n)$, but in fact we even have $P(n) = \Omega(4^n/n^{3/2})$. Thus a brute-force algorithm for the matrix-chain multiplication problem would be very inefficient.

We will now walk through designing a dynamic programming algorithm for this problem following the steps outlined at the beginning of this chapter.

Step 1: The structure of an optimal parenthesization. Our first step is to examine what the structure of an optimal solution looks like. For brevity, given $i \leq j$, we shall denote

$$A_{i..j} \;:=\; A_iA_{i+1} \cdots A_j.$$

Suppose $i < j$ and we have an optimal parenthesization for $A_{i..j}$. Then there must be some k such that $i \leq k < j$ where the outer-most parenthesization splits between A_k and A_{k+1}, i.e.,

$$A_{i..j} \;=\; (A_{i..k})(A_{k+1..j})$$

[1] See `https://en.wikipedia.org/wiki/Catalan_number`

But for an optimal parenthesization of $A_{i..j}$, we must also have a parenthesization of the first product $A_{i..k}$ and the second product $A_{k+1..j}$. The two parenthesizations used here must also be optimal parenthesizations of $A_{i..k}$ and $A_{k+1..j}$. This is because the total number of scalar multiplications needed for $A_{i..j}$ is the total number to multiply $A_{i..k}$ by $A_{k+1..j}$, plus the number needed for $A_{i..k}$ plus the number needed for $A_{k+1..j}$. If we were using a non-optimal parenthesization for, say, $A_{i..k}$, we could get an improvement by switching to an optimal parenthesization for $A_{i..k}$, and this in turn would be an improvement for $A_{i..j}$.

In summary, we can find an optimal solution for $A_{i..j}$ by finding all the optimal solutions to $A_{i..k}$ and $A_{k+1..j}$, for every $k = i, \ldots, j-1$.

Step 2: A recursive solution. Now that we've seen that optimal solutions depend on optimal solutions to subproblems, we can come up with a recursive formula for the optimal solutions. Are subproblems are all of the form: what is an optimal parenthesization for $A_{i..j}$, where $1 \leq i \leq j \leq n$. Since the subproblems depend on two parameters i and j, we will store our solutions to subproblems in a two-dimensional array $m[i, j]$, where $m[i, j]$ is the minimum number of scalar multiplications required for some parenthesization of $A_{i..j}$.

If $i = j$, then our chain $A_{i..j} = A_i$ has length one, so no scalar multiplication are required. Thus $m[i, i] = 0$ for every i.

Otherwise, suppose $i < j$ and we know that the optimal parenthesization of $A_{i..j}$ involves splitting this product at k, for some $i \leq k < j$. Then we have the relation

$$m[i, j] \;=\; m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j$$

where $p_{i-1}p_k p_j$ is the total number of scalar multiplications needed for the $p_{i-1} \times p_k$ matrix $A_{i..k}$ times the $p_k \times p_j$ matrix $A_{k+1..j}$. Since in general we don't know in advance what the value of k is, we need to consider all possible values of k and take the minimum. Thus we arrive at the following recurrence:

$$m[i, j] \;=\; \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \left\{ m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j \right\} & \text{if } i < j. \end{cases}$$

Finally, in anticipation of needing to reconstruct an optimal parenthesization, we also have a two-dimensional array $s[i, j]$ where for $i < j$ we set $s[i, j] = k$ if k provides an optimal way of splitting $A_{i..j}$. Thus

$$s[i, j] \;=\; k \quad \text{such that} \quad m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j \;=\; m[i, j].$$

Step 3: Computing the optimal costs. We now compute the solutions to all the subproblems in a bottom-up manner. Since for every pair (i, j) with $1 \leq i \leq j \leq n$ we have a subproblem associated to $A_{i..j}$, we see that there are $\Theta(n^2)$ subproblems to solve. We will solve each of these subproblems in the correct order and store their solutions in a two-dimensional array. The implementation is as follows:

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4       m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6       for i = 1 to n − l + 1
 7            j = i + l − 1
 8            m[i, j] = ∞
 9            for k = i to j − 1
10                 q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11                 if q < m[i, j]
12                      m[i, j] = q
13                      s[i, j] = k
14  return m and s
```

MATRIX-CHAIN-ORDER works as follows (see Figure 6.2 for an example):

(1) We assume we that our matrix chain $\langle A_1, \ldots, A_n \rangle$ has compatible dimensions between adjacent matrices, i.e., the matrix A_i has size $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. In fact, we take as input an array of the relevant matrix sizes: $p = \langle p_0, p_1, \ldots, p_n \rangle$. Thus $p.\mathit{length} = n + 1$.
(2) In lines 1-2 we create the two dimensional array $m[i, j]$ which will store the optimal solutions to the $A_{i..j}$ subproblems, and also the two-dimensional array $s[i, j]$ which will store the index k which achieves the optimal cost for the $m[i, j]$ subproblem.
(3) In lines 3-4 we initialize the diagonal $m[i, i] = 0$, since computing the single matrix A_i (viewed as a 1-element matrix product) requires zero scalar multiplications.
(4) In the **for** loop of lines 5-13 we iterate over the lengths of nontrivial matrix chains in increasing order.
(5) For a fixed length $l = 2, \ldots, n$, in the **for** loop of lines 6-13, we iterate over all possible first indices i. Each index i also determines the index j being considered, $j = i + l - 1$.
(6) In line 8 we initialize $m[i, j] = \infty$ with a sentinel.
(7) In the **for** loop of lines 9-13, we define $m[i, j]$ according to the formula:

$$m[i, j] \;=\; \min_{i \le k \le j-1} \big\{ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \big\}$$

We also record the value $s[i, j] = k$ where k is the index which witnesses the optimal solution for $m[i, j]$.
(8) Finally in line 14 we return both tables m and s.

Because MATRIX-CHAIN-ORDER relies on a triple nested **for** loop, one can show that its running time is $O(n^3)$.

Step 4: Constructing an optimal solution. Finally, we have a routine which recovers an optimal solution from the table s:

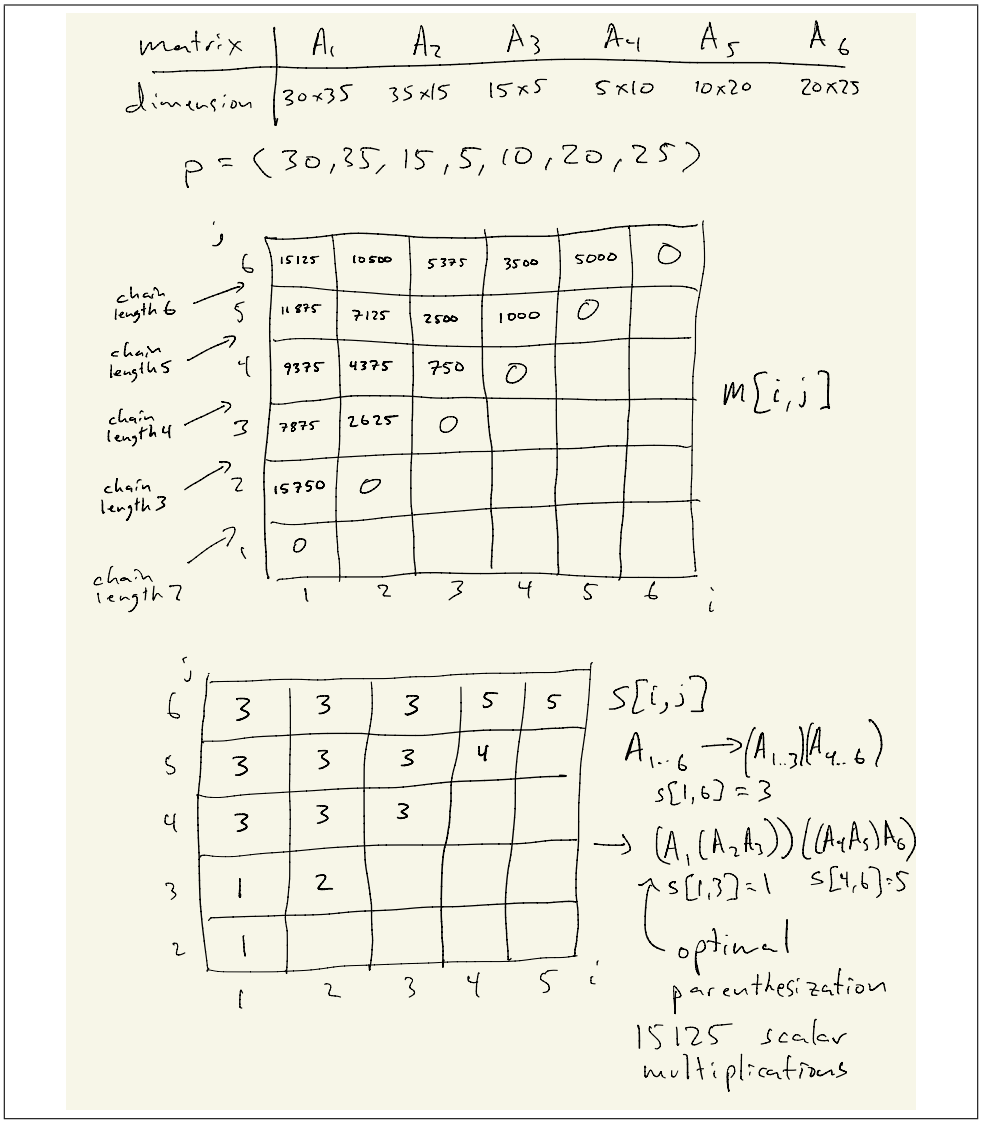


FIGURE 6.2. Here we find the optimal parenthesization for the chain $\langle A_1, A_2, \ldots, A_n \rangle$, where the array of dimensions is $p = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$. We compute iteratively the optimal solutions to the subproblems $A_{i..j}$ and store the answers in the two-dimensional array $m[i, j]$. We also store in $s[i, j]$ the index k which achieves the optimal solution for $m[i, j]$. This allows us to reconstruct the optimal parenthesization.

PRINT-OPTIMAL-PARENS(s, i, j)

```
1  if i == j
2       print "A_i"
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

PRINT-OPTIMAL-PARENS works as follows:

(1) The algorithm takes as input the table s as well as the indices i, j. The desired output is a printing of the optimal parenthesization of $A_{i..j}$.
(2) If $i = j$, then there is only one matrix, so the algorithm prints "A_i" (no parentheses necessary).
(3) Otherwise, the algorithm prints:

$$(A_{i..s[i,j]}A_{s[i,j]+1..j})$$

where $A_{i..s[i,j]}$ recursively is an optimal parenthesization for $A_{i..s[i,j]}$, and $A_{s[i,j]+1..j}$ is an optimal parenthesization for $A_{s[i,j]+1..j}$.

CHAPTER 7

Greedy algorithms

We saw in the previous chapter that dynamic programming algorithms are often useful for optimization problems. In this chapter we study another class of algorithms which can be used for optimization problems: *greedy algorithms*. A **greedy algorithm** is an algorithm which always makes a locally optimal choice (a greedy choice) in the hopes that it leads to a globally optimal solution.

7.1. An activity-selection problem

In this section we look at a typical example of an optimization problem which has a greedy algorithm solution: the *activity-selection problem*. Suppose we are in charge of scheduling a common resource (for instance, a classroom on a weekend for student organization meetings, or a multi-purpose room at the campus recreation center for sports clubs, etc.) and we receive a large number of requests for use of the resource, and our job is to determine a schedule which maximizes the total number of requests we can accommodate.

More specifically, suppose we have a set $S = \{a_1, \ldots, a_n\}$ of n proposed **activities** that wish to use a resource, and each activity a_i has a **start time** s_i and a **finish time** f_i, such that $0 \leq s_i < f_i < \infty$. The activity a_i takes place during the time interval $[s_i, f_i)$. We say that two activities a_i and a_j are **compatible** if the time intervals do not overlap, i.e., if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$, or equivalently, if $f_i \leq s_j$ or $f_j \leq s_i$.

> The **activity-selection problem**, we want to find a maximum-size subset of $S = \{a_1, \ldots, a_n\}$ of mutually-compatible activities. Note that we want to maximize the *number* of activities scheduled, not necessarily the total *duration* of the activities which are scheduled.

For convenience, we shall assume that the activities are sorted in increasing order of finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n.$$

For example, suppose the proposed set of activities is the following:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	8	10	11	12	14	16

We see that $\{a_3, a_9, a_{11}\}$ is a set of 3 mutually compatible activities. However, $\{a_1, a_4, a_8, a_{11}\}$ is a larger (and in fact, a largest) set of mutually compatible activities. The solution need not be unique, for instance, the set $\{a_2, a_4, a_9, a_{11}\}$ is also a largest set of mutually compatible activities.

Optimal substructure. We will first proceed the same way we did for dynamic programming problems: by observing that the activity-selection problem exhibits optimal substructures. As $S = \{a_1, \ldots, a_n\}$ denotes the set of all proposed activities, let S_{ij} denote the set of all proposed activities which start *after* activity a_i finishes and *before* activity a_j begins.

Suppose we are interested in finding a maximum set of mutually compatible activities inside S_{ij}, let's call the set A_{ij}. Furthermore, suppose A_{ij} contains the activity a_k. Then this gives rise to two subproblems, namely finding a maximum set of compatible activities in S_{ik} as well as S_{kj}. Define $A_{ik} := A_{ij} \cap S_{ik}$ and $A_{kj} := A_{ij} \cap S_{kj}$. Then $A_{ij} = A_{ij} \cup \{a_k\} \cup A_{kj}$, and so $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$. Furthermore, it is clear that A_{ij} must include optimal solutions to the subproblems S_{ik} and S_{kj} (if not, we could replace, for instance, A_{ik} with an optimal solution to ultimately make A_{ij} larger).

This suggests we might have a dynamic programming solution to the activity-selection problem. Suppose $c[i, j]$ is the size of an optimal solution for the set S_{ij}, and suppose this optimal solution contains the activity a_k. Then the discussion above tells us that

$$c[i, j] \;=\; c[i, k] + c[k, j] + 1.$$

Since in general we don't know a precise k for which $a_k \in A_{i,j}$, we obtain the following recurrence for $c[i, j]$:

$$c[i, j] \;=\; \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

If we wanted to, we could develop a dynamic programming algorithm which implements this recurrence: either a memoized version of a recursive algorithm, or a bottom-up iterative algorithm. However, as we will see in the next section, upon further analysis of this problem we can come up with an even more efficient algorithm.

Making the greedy choice. In both the rod-cutting problem and the matrix-chain problem of the previous chapter, we weren't able to determine any part of our optimal solution until we solved *all* subproblems and then reconstructed our entire solution in one shot from all of our solutions. For the activity-selection problem, however, we are actually able to begin to construct our optimal solution *without* solving all subproblems. In particular, we will not have to consider all possible choices for k involved in the above recurrence, just one choice: the *greedy choice*.

Suppose we wanted to commit to scheduling at least one activity which is guaranteed to be part of some optimal solution. Perhaps we want to commit to scheduling the first activity of the day. Intuitively, this should be the activity which leaves the resource available for as many other activities as possible. Thus it makes sense to commit to scheduling the activity with the *earliest finish time*, since all other choices of *first activity* would only leave less available time for scheduling other activities. We will show that always choosing the earliest finish time of the remaining compatible activities will yield an optimal solution.

Consider the set $S_k := \{a_i \in S : s_i \geq f_k\}$, i.e., the set of activities that start after a_k finishes. Because of the optimal substructure property, if an optimal solution contains a_k, then it must also contain an optimal solution to the subproblem S_k.

The following shows that our "earliest finish time" intuition is correct, it uses a standard "exchange" argument:

Theorem 7.1.1. *Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.*

PROOF. Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest start time. If $a_j = a_m$, then we are done. Otherwise, if $a_j \neq a_m$, consider the "exchanged" set $A'_k = A_k \setminus \{a_j\} \cup \{a_m\}$ (where we exchange a_j with a_m). The activities in A'_k are disjoint since they are disjoint in the original A_k, and $f_m \leq f_j$, so a_m is still disjoint from $A_k \setminus \{a_j\}$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k which includes a_m. □

This theorem shows us that, while an optimal solution doesn't *have to* include the earliest-finishing activity, we are always safe when choosing to include the earliest-finishing activity. Thus, while we might be able to implement a dynamic programming solution, we don't have to: we can just choose the activity with the earliest finishing time, and repeatedly make this choice for every subproblem which arises. In the dynamic programming problems we've seen, we don't know which choices are part of an optimal solution until the very end when we reconstruct the solution after solving enough subproblems; here we know in advance at least one valid choice we can make at each step which will be part of an optimal solution.

Furthermore, this solution to solving the activity-selection problem naturally leads to a top-down algorithm: we can make the greedy choice and then recursively solve the subproblem which consists of the remaining compatible proposed activities. We don't need to do a bottom-up algorithm where we systematically solve all possible subproblems and then afterwards reconstruct a solution.

A recursive greedy algorithm. The following is a recursive greedy algorithm which solves the activity-selection problem:

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

RECURSIVE-ACTIVITY-SELECTOR works as follows (see Figure 16.1 in [**1**]):

(1) We take as input arrays s and f which contains the start times and finish times, the index k which indicates that we are solving the subproblem S_k, and the number n of all proposed activities in the list. We assume that the activities are sorted so that the finish times are in increasing order. Furthermore, we assume that there is an artificial first activity a_0 with $s_0 = f_0 = 0$, so that we can solve the original problem by calling RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

(2) In line 1 we set $m = k + 1$. This is the index of the first activity in the list after a_k, although it might be incompatible with a_k. Thus the **while**

loop in lines 2-3 walks through the activities in increasing order until it finds the first activity which is compatible with a_k. Since the finish times are in increasing order, this will also be a compatible activity with the earliest finish time.

(3) If the **while** loop breaks because we found an activity a_m with $s_m \geq f_k$, then in lines 4-5 we add a_m to our optimal solution and proceed to solve the subproblem S_m by calling RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n).

(4) Otherwise, if the **while** loop breaks because $m > n$, then this means we've exhausted all activities on our list and there are no compatible activities in S_k. Thus we return $\emptyset$ in line 6.

The running time of RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$. This can be seen as follows: over all recursive calls, the value of m goes from 1 to $n + 1$ and only assumes each value once.

An iterative greedy algorithm. We can also convert our top-down recursive greedy algorithm into a top-down iterative greedy algorithm as follows:

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

GREEDY-ACTIVITY-SELECTOR works as follows:

(1) In line 1 we define n to be the total number of activities. In line 2 we make our first choice a_1, which is guaranteed to have the earliest finish time since the finish times are ordered. In line 3 we initialize the variable k to $k = 1$. The variable k keeps track of which subproblem S_k we are currently solving.

(2) In the **for** loop of lines 4-7 we step through the remaining activities in order. Each time we find an activity a_m compatible with a_k, we choose a_m to be part of our optimal solution (line 6) and then we update $k = m$.

CHAPTER 8

Elementary graph algorithms

8.1. Representations of graphs

There are two standard ways to represent a graph $G = (V, E)$, either as a collection of adjacency lists or as an adjacency matrix. Both of these ways work for either *undirected* or *directed* graphs. Typically we will represent our graphs using an adjacency list, although it is good to be aware of both methods. We illustrate both in Figure 8.1. These representation methods can be adapted for weighted graphs.

Adjacency-list representation. Given a graph $G = (V, E)$, the **adjacency-list representation** consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$, i.e., $Adj[u]$ consists of all the vertices adjacent to u in G. We will not care how each list $Adj[u]$ is implemented, since this can be done multiple ways (for instance, as an array, or a (singly/doubly) linked-list). We will consider the adjacency list to be an attribute of the graph G, so in pseudocode we will refer to $G.Adj$ as the adjacency list associated with G.

Adjacency lists are an especially ideal way to represent **sparse** graphs, i.e., graphs for which $|E|$ is much smaller than $|V|^2$. In general, the adjacency-list representation requires $\Theta(V + E)$ memory. One potential disadvantage of using an adjacency list representation is that to determine if $(u, v) \in E$ in general you need to search the list $Adj[u]$ for the vertex v, which might take a linear amount of time.

Adjacency-matrix representation. For an **adjacency-matrix representation** of a graph $G = (V, E)$, we assume that the vertices are indexed $1, 2, \ldots, |V|$ in some way. Then the adjacency-matrix representation of G is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency-matrix representation requires $\Theta(V^2)$ memory, regardless of the number of edges. This is not ideal if the graph is sparse, but it is if the graph is **dense**, i.e., if $|E|$ is known to be close to $|V|^2$. One potential advantage of using an adjacency-list is that determining whether $(u, v) \in E$ can be done in $\Theta(1)$ time.

8.2. Breadth-first search

In this section we investigate our first graph algorithm: **breadth-first search**. Given a graph $G = (V, E)$ and a distinguished **source** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex reachable from s. Breadth-first search accomplishes the following useful things:

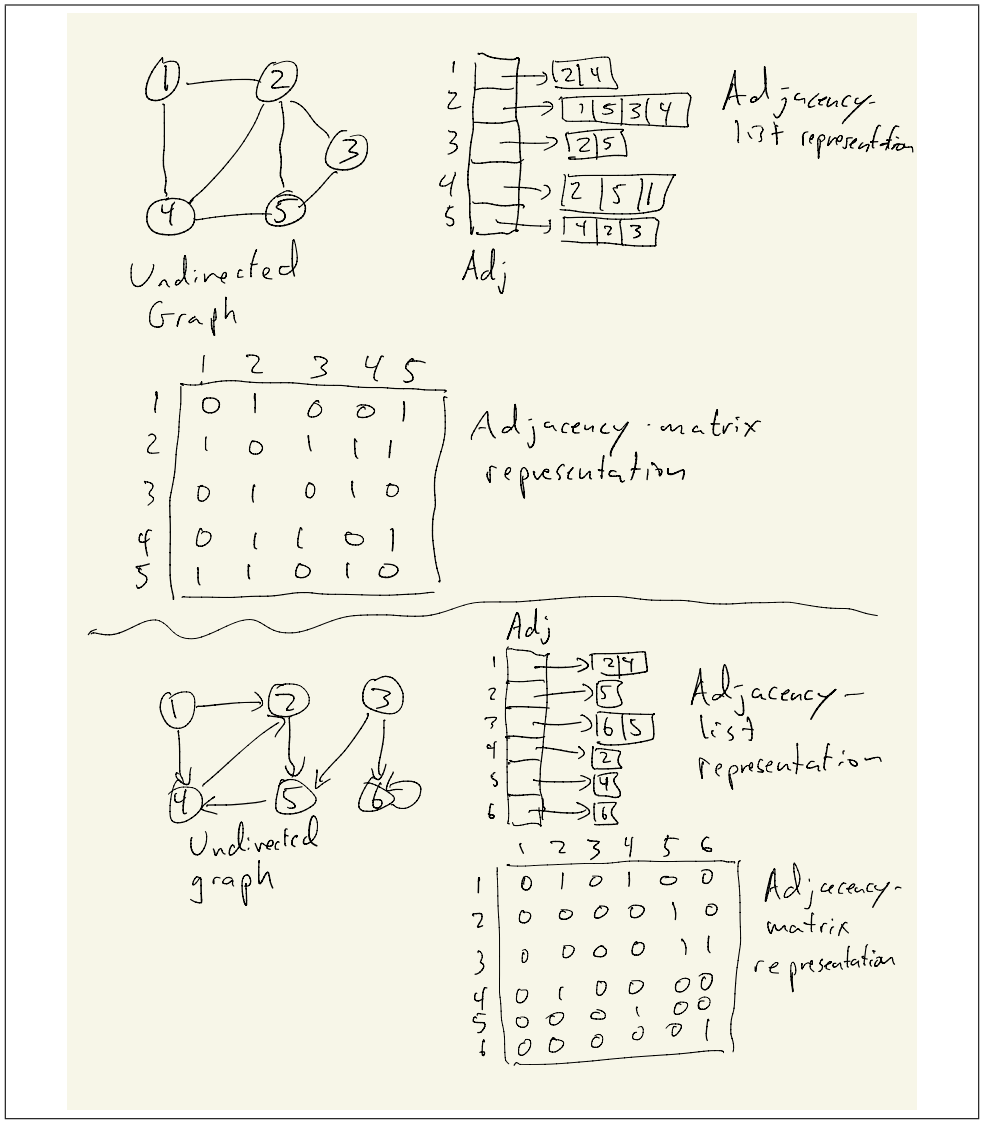


FIGURE 8.1. Here we show how both an undirected graph and a directed graph can be represented with either an adjacency-list or with an adjacency-matrix.

(1) It computes the distance from s to each reachable vertex (distance=path with fewest number of edges)
(2) It produces a "breadth-first tree" with root s which contains all reachable vertices.
(3) For any vertex v reachable from s, the unique simple path in this tree from s to v is a "shortest path" from s to v.

The *breadth* part of breadth-first search refers to the fact that this algorithm expands outward uniformly. At each point in the algorithm, we have a certain "frontier of discovery" and we expand this frontier between discovered and undiscovered

vertices uniformly across the breadth of the frontier. In other words, the algorithm first discovers all vertices distance 1 away from s, then distance 2 away from s, then distance 3 away from s, etc.

To help keep track of the discovery progress, during the course of BFS each vertex gets tagged with a color: WHITE, GRAY or BLACK. Here are the rules for these colors:

(1) All vertices start out WHITE. A WHITE vertex is an undiscovered vertex.
(2) A vertex is **discovered** the first time it is encountered in an algorithm. At this point the vertex becomes nonwhite (GRAY or BLACK).
(3) GRAY vertices represent the frontier of the search. They might have WHITE neighbors.
(4) BLACK vertices are discovered vertices not on the frontier of the search. The only have GRAY and BLACK neighbors.

In order to construct a breadth-first tree, we also give each vertex a **predecessor** or a **parent**. If a vertex v is first discovered while scanning the vertices which are adjacent to a vertex u, then the edge (u, v) is added to the tree and we set the predecessor of u. Initially all predecessors (except the predecessor of s) are initialized to be NIL. We denote the color and predecessor attributes of a vertex u by $u.color$ and $u.\pi$. Finally, each vertex gets tagged with a **distance** $u.d$ which represents the best known shortest distance from s to u. Initially all distances are initialized to ∞.

Here is the code for breadth-first search (relative to the source s):

BFS(G, s)
```
 1  for each vertex u ∈ V, V \ {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

BFS works as follows (see Figure 8.2 for an example):

(1) In lines 1-4 we initialize all vertices except for s. Every vertex gets color WHITE since it is not yet discovered, gets distance ∞ because we do not know any paths yet from s, and gets predecessor NIL since it is not yet part of the breadth-first tree.

(2) In lines 5-7 we set the data for the source s. The initial color of s is GRAY because it start out as the first (and only) vertex on the frontier of discovered vertices. The distance $s.d$ is set to 0 since it has distance 0 to itself, and its predecessor is NIL since it will be the root of our breadth-first tree.
(3) In line 8 we create an empty queue $Q = \emptyset$. The job of the queue Q is to keep track of which vertices are on the frontier and to tell us which vertex we need to process next. The reason we use a queue is because its first-in first-out (FIFO) policy ensures we process the entire breadth of the current frontier before processing new vertices outside the current frontier. Line 9 gets the process started by putting s in as the first element of the queue.
(4) The **while** loop of lines 10-18 does the main work of the algorithm. It first takes the vertex u which is at the front of the queue (line 11) and looks at each one of u's neighbors (lines 12-17). If a particular neighbor v of u is WHITE, this means that it is a new vertex so we need to include it as part of the frontier. Thus we color v GRAY, set its distance to be one more than u, and set its predecessor to be u. Finally we put v in the back of the queue (line 17) so that it will be eventually processed itself. The **while** loop maintains the following loop invariant:
    (Loop invariant for **while** loop) Each time after line 10 is run, the queue Q contains precisely the set of all GRAY elements.
    We won't need this loop invariant to prove correctness, although it helps with getting intuition for the algorithm.
(5) Once we are done completely processing all neighbors of u, we color u BLACK in line 18 to signify that we are completely done with this vertex.

Running-time analysis. Suppose we run BFS on a graph $G = (V, E)$. Lines 1-9 perform $O(V)$ amount of work (here we will abuse notation and write V for $|V|$ and E for $|E|$). In the **while** loop, each vertex $u \in V$ is processed at most once, because once a vertex is dequeued, it is never enqueued again. Thus in the **for** loop the adjacency list for u is scanned only once. Since the sum of all adjacency lists is $\Theta(E)$, the total time scanning adjacency lists is $O(E)$. Thus the total running time for the BFS procedure is $O(V + E)$.

Shortest paths. Next we want to show that BFS correctly computes the shortest-path distances. Given a graph $G = (V, E)$ and a source vertex $s \in V$, define the **shortest-path distance** $\delta(s, v)$ from s to v to be the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a **shortest path** from s to v.

Lemma 8.2.1. *Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,*

$$\delta(s, v) \;\leq\; \delta(s, u) + 1.$$

PROOF. If u is reachable from s (so the righthand side is finite), then v is reachable. Given a shortest path from s to u, we can attach the edge (u, v) to get a path from s to v of length $\delta(s, u) + 1$. A shortest path from s to v will either be this length or shorter. □

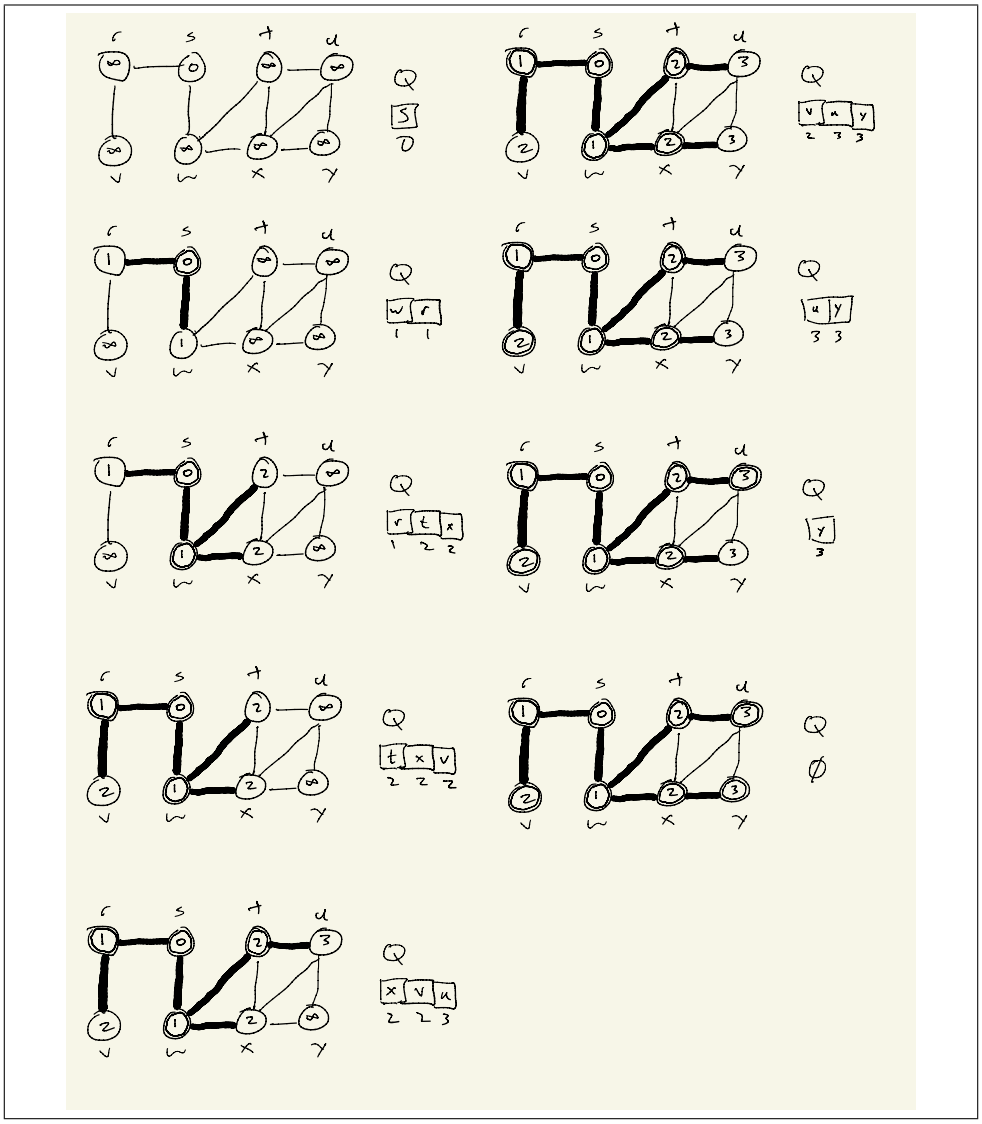


FIGURE 8.2. An example of BFS. The BLACK vertices are the ones with double circles and the WHITE vertices have distance ∞. All other vertices are GRAY.

Next we will show that $v.d$ always is an upper bound for $\delta(s, v)$.

Lemma 8.2.2. *Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.*

PROOF. We will prove the follow loop invariant:

> (Loop invariant for **while** loop) each time after line 10 is run, for every $v \in V$, $v.d \geq \delta(s, v)$.

(Initialization) After the first time line 10 is run, we have just enqueued s in line 9. At this point, $s.d = 0 = \delta(s, s)$. Also, for every $v \in V \setminus \{s\}$, we have $v.d = \infty$, and $\infty \geq \delta(s, v)$ is always true.

(Maintenance) Suppose line 10 has just run, $Q \neq \emptyset$, and we know the loop invariant is true. Then we proceed to line 11 and get u. In this iteration of the **while** loop, the only vertices with $v.d$ modified are among those in $G.Adj[u]$. Suppose v is a white vertex discovered adjacent to u. Then we set $v.d = u.d + 1$ and so

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \quad \text{by loop invariant assumption} \\ &\geq \delta(s, v) \quad \text{Lemma 8.2.2.} \end{aligned}$$

Thus the loop invariant is still true for all vertices that have their distance attribute updated during this iteration, so next time we go to line 10 the loop invariant remains true.

(Termination) Once $Q = \emptyset$, the algorithm stops and the loop invariant is true at this point, thus the lemma is true. □

The next lemma shows that at all times the queue Q contains at most two distinct d values:

Lemma 8.2.3. *Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r - 1$.*

Proof. We will prove this by induction on the number of queue operations. Thus we have to show that this property is preserved after each dequeue and enqueue which is performed.

Initially the queue contains only s, so the lemma is true.

(Dequeue) Suppose the statement of the lemma is currently true and we dequeue the vertex v_1. Then v_2 becomes the new head (or else the queue is empty and the lemma is vacuously true). Since we are inductively assuming that $v_1.d \leq v_2.d$, we have $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$, so the statement of the lemma still holds.

(Enqueue) Suppose we have just enqueued a vertex v in line 17. Now this vertex becomes v_{r+1}, the new tail. At this time we have just recently dequeued a vertex u whose adjacency list is currently being scanned, and by the inductive assumption, $v_1.d \geq u.d$. Thus $v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$. The inductive hypothesis also implies that $v_r.d \leq u.d + 1$ and so $v_r.d \leq u.d + 1 = v.d = v_{r+1}.d$. The other inequalities are unaffected. Thus the statement of the lemma still holds. □

Since each vertex receives a finite d value at most once during BFS, we have the following consequence of Lemma 8.2.3:

Corollary 8.2.4. *Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.*

We can finally show that BFS correctly computes the shortest-path distances:

Theorem 8.2.5 (Correctness of breadth-first search)**.** *Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given single source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that*

is reachable from the source s, and upon termination, $v.d = \delta(s,v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, on of the shortest paths from s to v is a shortest path from s to $s.\pi$ followed by the edge $(v.\pi, v)$.

PROOF. Assume towards a contradiction that some vertex receives a d value not equal to the shortest-path distance. Let v be such a vertex with minimal $\delta(s,v)$. Then $v \neq s$. By Lemma **??**, $v.d \geq \delta(s,v)$, and so $v.d > \delta(s,v)$. Vertex v must be reachable from s, for otherwise we would have $v.d = \infty = \delta(s,v)$ which would be the correct shortest-path distance. Suppose u is a vertex immediately preceding v on some shortest path from s, so $\delta(s,v) = \delta(s,u) + 1$. Thus $u.d = \delta(s,u) < \delta(s,v)$. Thus

$$v.d \;>\; \delta(s,v) \;=\; \delta(s,u) + 1 \;=\; u.d + 1.$$

Consider the point at which u was dequeued from Q during BFS in line 11. At this point v has one of three colors, which gives us three cases.

(v WHITE) Then line 15 sets $v.d = u.d + 1$, contradicting the above inequality.

(v BLACK) Then we have already removed v from the queue and so $v.d \leq u.d$, contradicting Corollary 8.2.4.

(v Gray) Then v was painted gray by some vertex w dequeued before u. Thus $v.d = w.d + 1$, and by Corollary 8.2.4, $w.d \leq u.d$. Thus $v.d = w.d + 1 \leq u.d + 1$, also contradicting the above inequality.

We conclude that $v.d = \delta(s,v)$ for all $v \in V$. All reachable vertices must be discovered, for otherwise we would have $v.d = \infty > \delta(s,v)$.

Finally, observe that if $v.\pi = u$, then $v.d = u.d + 1$. Thus we can obtain a shortest path from s to v by taking a shortest path fro s to $v.\pi$ and then traversing the edge $(v.\pi, v)$. □

Breadth-first trees. We now show that BFS builds a breadth-first tree. First, some definitions. Given a graph $G = (V,E)$ with source s, we define the **predecessor subgraph** of G has $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi \;=\; \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi \;=\; \big\{(v.\pi, v) : v \in V_\pi \setminus \{s\}\big\}$$

The predecessor subgraph G_π is a **breadth-first tree** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G. We call the edges of E_π **tree edges**. A breadth-first tree is always a tree since it is connected and $|E_\pi| = |V_\pi| - 1$.

Lemma 8.2.6. *When applied to a directed or undirected graph $G = (V,E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.*

PROOF. Line 16 of BFS sets $v.\pi = u$ iff $(u,v) \in E$ and $\delta(s,v) < \infty$. Thus V_π consists of the vertices in V reachable from s. Since G_π forms a tree (it is connected and $|E_\pi| = |V_\pi| - 1$), it contains a unique simple path from s to each vertex in V_π. By applying Theorem 8.2.5 inductively, we conclude that every such path is a shortest path in G. □

The following prints out the vertices on a shortest path from s to v assuming we have already run BFS.

```
PRINT-PATH(G, s, v)
1  if v == s
2       print s
3  Elseif v.π == NIL
4       print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6       print v
```

8.3. Depth-first search

In depth-first search (DFS), as opposed to BFS, we search "deeper" into the graph whenever possible. DFS will explore edges out of the most recently explored vertex. Once all of the edges going out of a vertex v has been explored, we backtrack to explore edges leaving the vertex from which v was discovered. Once we discover all vertices which are reachable from the source vertex, we pick a new undiscovered vertex and start the process over. We finish once all vertices in the graph have been discovered.

Like BFS, when we discover a vertex v during a scan of the adjacency list of the vertex u, we will set $v.\pi = u$. Unlike BFS, when DFS finishes the predecessor subgraph might not be a single tree but instead a collection of trees (i.e., a **forest**). We define the **predecessor subgraph** of a depth-first search slightly differently from that of a BFS. We let $G_\pi = (V, E_\pi)$, where

$$E_\pi \ := \ \big\{(v.\pi) : v \in V \text{ and } v.\pi \neq \text{NIL}\big\}.$$

The predecessor subgraph of a depth-first search forms a **depth-first forest** comprising several **depth-first trees**. The edges in E_π are **tree edges**.

Like BFS, in DFS we will also color the vertices with three colors (WHITE, GRAY and BLACK) to indicate the state of the vertex. A WHITE vertex is undiscovered, once it is discovered it is painted GRAY, and then after that vertex's adjacency list has been fully examined, it will be colored BLACK.

The DFS algorithm will also **timestamp** each vertex. Each vertex v will have two timestamps, the first timestamp $v.d$ will record the time when the vertex was discovered (painted GRAY) and the second timestamp $v.f$ will indicate when we are finished examining v's adjacency list (painted BLACK). The timestamps will provide information about the structure of the graph and will be useful in the analysis of DFS. In the algorithms DFS and DFS-VISIT, we will have a global variable *time* which we will use for the timestamping. Each vertex has two time attributes, and we will use integers for the time value, so the timestamps will be integers $1, 2, \ldots, 2|V|$. Furthermore, a vertex must be discovered before we can process its adjacency list, so for every vertex $u \in V$ we will have

$$u.d \ < \ u.f.$$

Furthermore, u will be colored WHITE prior to $u.d$, it will be colored GRAY between $u.d$ and $u.f$, and it will be colored BLACK after $u.f$.

The main procedure which drives the depth-first search is the following:

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

DFS works as follows:

(1) In lines 1-3 we initialize every vertex to have color WHITE and we initialize every predecessor to NIL.
(2) In line 4 we reset the global time to 0.
(3) In the **for** loop of lines 5-7, we visit each vertex u in the graph. If the vertex is nonwhite (GRAY or BLACK), this means it has already been discovered by some other vertex so we do nothing. Otherwise, if the vertex is WHITE, then we perform DFS-VISIT with source u.

The main technical work is performed by DFS-VISIT, it takes as input the graph G and a specified source vertex u:

```
DFS-VISIT(G, u)
 1  time = time + 1        // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]  // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK        // blackened u; it is finished
 9  time = time + 1
10  u.f = time
```

DFS-VISIT works as follows:

(1) When we call DFS-VISIT(G, u), this means we are discovering u for the first time.
(2) In line 1 we increase the time by one unit. Then we set $u.d = time$ to indicate that this is the time at which we are discovering u. We also set the color of u to GRAY to indicate that we are currently in the the process of looking at the vertices in the adjacency list of u.
(3) In the **for** loop of lines 3-6 we look at each vertex v in the adjacency list of u.
(4) For nonwhite vertices v we do nothing since these vertices are already discovered.
(5) If a vertex v is WHITE, then we set the predecessor of v to u, since u was the vertex which discovered v. Then we recursively perform DFS-VISIT(G, v).
(6) Once we are finished with all of u's neighbors, in line 8 we set the color of u to BLACK, we increase the current time by one, and set $u.f = time$ to indicate that this is the time at which we finished with u.

See Figure 22.4 in [**1**] for an example of DFS.

The running time of DFS is $\Theta(V+E)$. This is because each vertex is colored GRAY and then BLACK exactly once, and also for each vertex u, every edge leaving u is considered exactly once in line 4 of DFS-VISIT.

Properties of depth-first search. In this subsection we study properties of depth-first search. The first observation is that the predecessor subgraph G_π forms a forest of trees, because the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT, i.e., $u = v.\pi$ iff DFS-VISIT(G, v) was called during a search of u's adjacency list. More generally, v is a descendent of u iff v was discovered during the time v was gray.

Another observation is that the finishing times have a **parenthesis structure**, i.e., if we write "$(u$" when u was discovered and "$u)$" when we are finished with u, then the parentheses will be properly nested (see Figure 8.3). The next theorem makes this precise:

Theorem 8.3.1 (Parenthesis theorem)**.** *In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:*

(1) the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendent of the other in the depth-first forest,
(2) the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
(3) the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

PROOF. (Case 1) We first consider the case when $u.d < v.d$. This gives two subcases:

(Case 1a) Suppose $v.d < u.f$. In this case v was discovered while u was still gray, so v is a descendent of u. Furthermore, since v was discovered more recently than u, all of v's outgoing edges are explored, and v is finished, before the search returns to finish u. Thus $[v.d, v.f] \subseteq [u.d, u.f]$.

(Case 1b) Suppose $u.f < v.d$. Then $u.d < u.f < v.d < v.f$. In this case the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.

(Case 2) The case when $v.d < u.d$ is similar. □

The following is immediate from Theorem 8.3.1:

Corollary 8.3.2 (Nesting of descendants' intervals)**.** *Vertex v is a proper descendant of a vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.*

The next theorem gives another characterization of when one vertex is a descendant of another:

Theorem 8.3.3 (White-path theorem)**.** *In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.*

PROOF. ($\Rightarrow$) If $v = u$, then the path from u to v just contains u, which is still white when we set the value of $u.d$. Otherwise, suppose v is a proper descendant of u. By Corollary 8.3.2, $u.d < v.d$, so v is still white at the time $u.d$. Since this

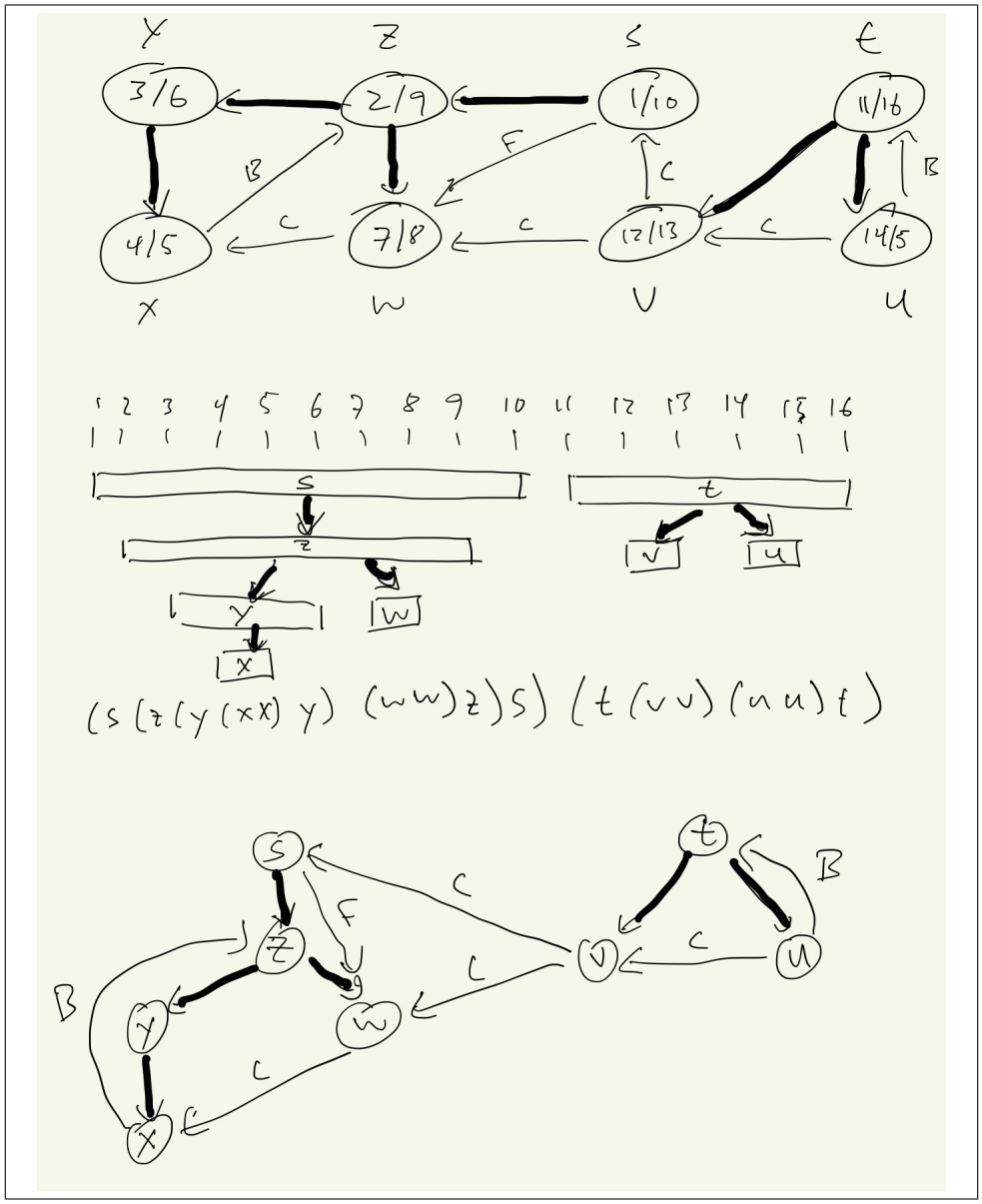


FIGURE 8.3. Here we show the result of DFS on a directed graph. Below that is an illustration of the intervals of the discovery times for each nodes. Finally, we show the directed graph with all tree and forward edges going down and all back edges going up.

is true for all descendants of u, every vertex on the unique simple path from u to v through the DFS forest is white.

($\Leftarrow$) Suppose towards a contradiction that there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the DFS forest. Without loss of generality, we may assume that v is the first vertex along the white

path from u to v that does not become a descendant. Let w be the predecessor of v in the path. Then w is a descendant of u. By Corollary 8.3.2, $w.f \leq u.f$. Since v must be discovered after u is discovered, but before w is finished (since there is an edge from w to v), we have $u.d < v.d < w.f \leq u.f$. By Theorem 8.3.1, the interval $[v.d, v.f]$ is contained within the interval $[u.d, u.f]$. Thus v must be a descendant of u by Corollary 8.3.2, a contradiction. □

Classification of edges. Depth-first search also provides information about the nature of the edges of $G = (V, E)$. First, we define the following four types of edges in terms of the depth-first forest G_π:

(1) **Tree edges** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was discovered by exploring edge (u, v).
(2) **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
(3) **Forward edges** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
(4) **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

In Figure 8.3 we illustrate the four types of edges in the bottom illustration. During DFS, we can classify the type of an edge (u, v) in terms of the color of v at the time (u, v) is explored:

(1) If v is WHITE, then (u, v) is a tree edge. This is because DFS will set $v.\pi = u$.
(2) If v is GRAY, then (u, v) is a back edge. This is because gray edges always form a linear chain of descendants which corresponds to the active recursive DFS-VISIT calls.
(3) If v is BLACK, then (u, v) is either a forward edge or a cross edge.

In a DFS of an undirected graph, only tree edges or back edges can occur:

Theorem 8.3.4. *In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.*

PROOF. Suppose (u, v) is an edge of G, and suppose $u.d < v.d$. Then the search must finish and discover v before it finishes u (while u is gray), since v is on u's adjacency list. If the first time the search explores (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction of v to u. Thus (u, v) is a tree edge in this case. Otherwise, if the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge since u is still gray at this time. □

Depth-first search has several important applications which we won't have time for. Specifically: topological sort (see [**1**, §22.4]) and strongly connected components (see [**1**, §22.5]).

8.4. Minimum spanning trees

Consider the following problem: we are designing an electronic circuit and we have to connect the pins of several components by wiring them together. To connect n

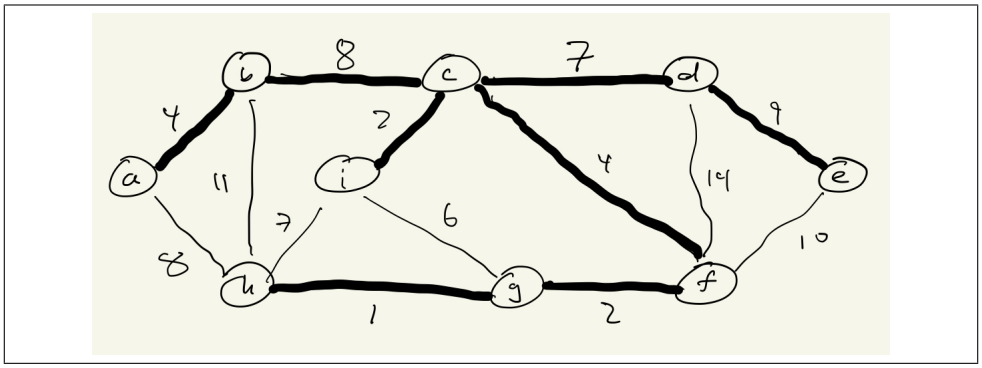


FIGURE 8.4. An example of a minimum-spanning tree

pins together, we need $n-1$ wires. There could be many ways to do this, but the way which uses the least amount of wire would be the most desireable.

We can model this problem as a graph problem. Suppose we have a connected undirected graph $G = (V, E)$ where V is the set of pins and E is the set of potential wires used to connect the pins together. For each edge $(u, v) \in E$, we have a weight $w(u, v) \in \mathbb{R}$ which represents the total cost (i.e., amount of wire) associated to using the edge (u, v). The goal is to find an acyclic $T \subseteq E$ which connects all of the vertices and minimizes the total cost:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

Since T is acyclic and connects all of V, it forms a tree which we call a **spanning tree** of G (since it "spans" all of G). The problem of finding a T which minimizes $w(T)$ is called the **minimum-spanning-tree problem**. For instance, in Figure 8.4 we give an example of a minimum-spanning tree.

Specifically, we will suppose $G = (V, E)$ is a connected, undirected graph equipped with a weight function $w : E \to \mathbb{R}$. The typical way to solve this problem is with a greedy algorithm where we grow a minimum spanning tree one edge at a time. We will maintain a set $A \subseteq E$ of edges which satisfies the following loop invariant:

> (Loop invariant) Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we will determine some edge (u, v) that we can add to A without violating this invariant, i.e., we will look for some edge (u, v) such that $A \cup \{(u, v)\}$ is also a subset of some minimum spanning tree. We call such an edge a **safe edge** for A, since we can safely add it to A while maintaining the variant. Here is the overall strategy written in pseudocode:

GENERIC-MST(G, w)

```
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

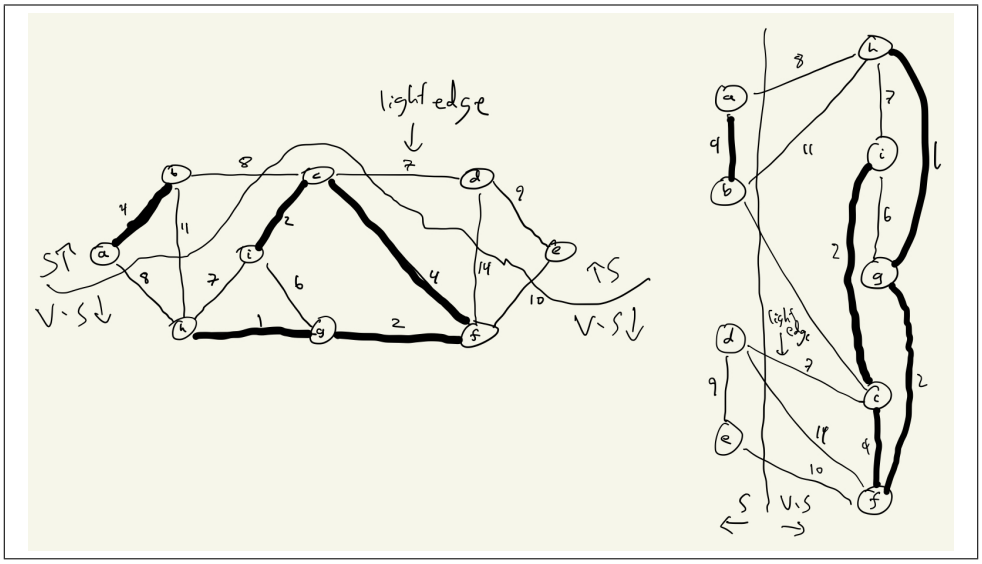


FIGURE 8.5. Here we show a cut $(S, V \setminus S)$ which respects A (the bold edges). We illustrate this cut in two different ways. We also indicate a light edge for the cut.

It is clear that this procedure will maintain the loop invariant (by definition of *safe edge*) and so it will result in a minimum spanning tree. What is not clear, however, is how to find a safe edge in line 3. We will now investigate how to go about finding a safe edge.

We define a **cut** $(S, V \setminus S)$ of an undirected graph $G = (V, E)$ to be a partition of the vertex set V. We say that an edge (u, v) **crosses** the cut $(S, V \setminus S)$ if one of the endpoints is in S and the other endpoint is in $V \setminus S$. We say a cut **respects** a set A if no edge in A crosses the cut. An edge is a **light edge** crossing a cut if its weight is the minimum of any edge which crosses the cut (there can be more than one light edges crossing a cut). More generally, we say that an edge is a **light edge** for a given property if it has minimum weight among all edges with a certain property.

The following gives one method of identifying edges which are safe for a set $A \subseteq V$. This forms the heart of the greedy strategy we will use and its proof is another "exchange" argument.

Theorem 8.4.1. *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.*

PROOF. Suppose T is a minimum spanning tree which contains A, and suppose T does not contain the light edge (u, v), for otherwise we would be done. We will use T to construct another minimum spanning tree $T' \supseteq A$ which does the light edge (u, v).

Since T is a tree, if we add (u, v) to T, this will form a certain cycle with the unique simple path p from u to v inside of T. Since u and v are on opposite sides

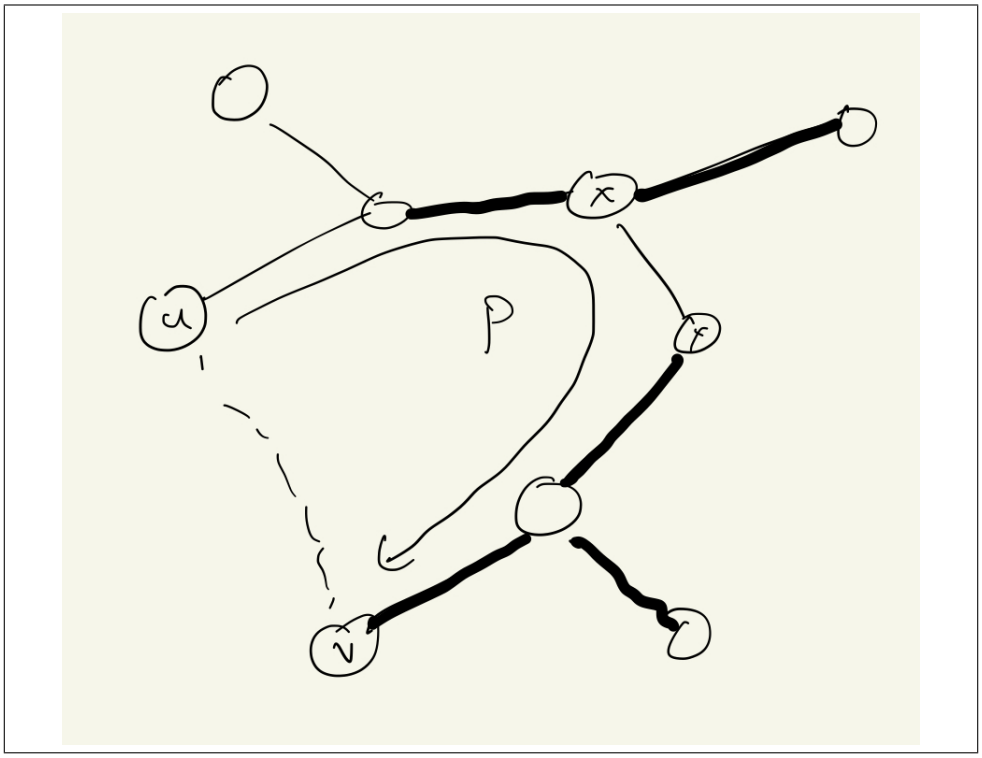


FIGURE 8.6. The proof of Theorem 8.4.1

of the cut, there is some edge (x, y) on the path p which also crosses the cut. The edge (x, y) is not in A, since the cut respects A. If we remove (x, y) from T this will break T into two components, which we can reconnect by adding back in the edge (u, v). Thus we get a new spanning tree $T' := T \setminus \{(x, y)\} \cup \{(u, v)\}$. (See Figure 8.6.)

Since (u, v) is a light edge for the cut and (x, y) also crosses the cut, it must be the case that $w(u, v) \leq w(x, y)$. Thus

$$w(T') \;=\; w(T) - w(x, y) + w(u, v) \;\leq\; w(T).$$

Thus T' is also a minimum spanning tree since T was. Furthermore, since $A \cup \{(u, v)\} \subseteq T'$ (because $(x, y) \notin A$), (u, v) is safe for A. □

Now that we know how to find a safe edge for A, we can make a few more observations about GENERIC-MST:

(1) At every step along the way, A is acyclic since it must be part of a minimum spanning tree.
(2) Also, $G_A = (V, A)$ is always a forest, and each connected component of G_A is itself a tree.
(3) Each safe edge (u, v) for A will connect two distinct components of G_A since $A \cup \{(u, v)\}$ must be acyclic.
(4) The **while** loop in lines 2-4 will execute $|V| - 1$, since this is the number of edges a spanning tree of G will contain.

(5) Initially when $A = \emptyset$, there are $|V|$ trees in G_A, each tree being a single vertex with no edges. Each time an edge is added, the number of trees is decreased by one, until the algorithm terminates with G_A being one single tree (the minimum spanning tree of G).

Of course, we still need to show how to implement Theorem 8.4.1 and how to choose a cut which respects A. In practice we will use the following:

Corollary 8.4.2. *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.*

PROOF. The cut $(V_C, V \setminus V_C)$ respects A, and (u, v) is a light edge for this cut. Thus (u, v) is safe for A. □

The textbook presents two algorithms which implement GENERIC-MST: Kruskal's algorithm and Prim's algorithm. For the sake of time we will only look at Prim's algorithm. Prim's algorithm is a special case of GENERIC-MST with the property that at every step the set A forms a single tree (as opposed to a disjoint union of several trees). At each step we adjoin a light edge which connects A to an isolated vertex. By Corollary 8.4.2 such an edge is always a safe edge and so when the algorithm terminates A will be a minimum spanning tree.

The main challenge with implementing Prim's algorithm is to use an efficient method of adjoining a light edge. The algorithm MST-PRIM takes as input the graph G, the weight function w and a root r of the minimum spanning tree to be grown (r can be any vertex). To assist in determining a light edge, the set of vertices which are not in A will be maintained by a min-priority queue Q based on a certain *key* attribute. For each vertex v, the value of $v.key$ is the minimum weight of any edge connecting v to a vertex in A; and $v.key = \infty$ if there is no such edge. Each vertex will also have an attribute $v.\pi$ which will ultimately be v's parent in the tree, although during the algorithm $v.\pi$ points to the vertex in A which gives the lightest edge connecting v to A. The set A from GENERIC-MST is not explicitly mentioned in the pseudocode for MST-PRIM, but it can be thought of as

$$A \;=\; \big\{(v, v.\pi) : v \in V \setminus \{r\} \setminus Q\big\}.$$

Upon termination, we will have $Q = \emptyset$ and thus the minimum spanning tree A for G will be

$$A \;=\; \big\{(v, v.\pi) : v \in V \setminus \{r\}\big\}.$$

MST-PRIM(G, w, r)

```
 1  for each u ∈ G.v
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

MST-PRIM works as follows:

(1) In lines 1-3 we initialize $u.key = \infty$ and $u.\pi = \text{NIL}$ for every vertex.
(2) In line 4 we initialize $r.key = 0$ to ensure that r is the first vertex extracted from the min-priority queue.
(3) In line 5 we put all of the vertices of G into the min-priority queue Q.
(4) In the **while** loop of lines 6-11 we iterate over the minimum elements of the queue, extracting the vertex with the minimum *key* value at each step until the queue is empty.
(5) In the **for** loop of lines 8-11, since we are adding u to the tree, we possible need to update the *key* value of every vertex which is connected to u. This is what is done in lines 9-11. If we update the *key* value of a non-A neighbor v of u, then we also update $v.\pi$ to point to u, since now the edge (u, v) is the lightest edge connecting v to the set A.

The correctness of MST-PRIM can be verified using the following loop invariant for the **while** loop of lines 6-11:

(**while** loop invariant) Immediately after each time line 6 is run:
(1) $A = \big\{(v, v.\pi) : v \in V \setminus \{r\} \setminus Q\big\}$.
(2) The vertices already placed into the minimum spanning tree are those in $V \setminus Q$.
(3) For all vertices $v \in Q$, if $v.\pi \neq \text{NIL}$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

The running time of MST-PRIM depends on how we implement our min-priority queue. If we implement our min-priority queue Q using a heap as in Section 5.3, then the initialization and BUILD-MIN-HEAP in lines 1-5 will run in O(V) time. The **while** loop will iterate $|V|$ times, and each EXTRACT-MIN operation will take $O(\lg V)$ time. Thus the total calls to EXTRACT-MIN will take $O(V \lg V)$ amount of time. Line 8 will run $O(E)$ times. The membership test in line 9 can be done in constant time if we maintain a boolean array of length $|V|$ which tells whether each vertex is a member of Q yet or not. Thus the total amount of time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$ (here we are using that $|E| \geq |V|$ since G is assumed to be connected).

CHAPTER 9

Single-source shortest paths

9.1. Single-source shortest paths

Suppose we are interested in taking a road trip from LA to some other to-be-determined city in the continental US. To help us determine our destination, we would like to know the shortest driving distance from LA to all other major cities. This is an instance of the **single-source shortest-paths problem**. Of course, one way to solve this problem is to consider all possible travel routes from LA to all other cities, and calculate the distance for each route, and take the minimum distance for each city. However, this strategy is not feasible because the total number of possible routes is becomes extremely large. Our goal in this chapter is to give an efficient solution to this problem.

This problem can be modeled with a weighted, directed graph. Specifically, suppose $G = (V, E)$ is a directed graph and $w : E \to \mathbb{R}$ is a weight function. The **weight** $w(p)$ of a path $p = \langle v_0, v_1, \ldots, v_k$ is the sum of the weights of its constituent edges:

$$w(p) \;=\; \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

We define the **shortest-path weight** $\delta(u, v)$ from u to v by

$$\delta(u, v) \;=\; \begin{cases} \min\{w(p) : u \rightsquigarrow^p v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

If $\delta(u, v) < \infty$, then a **shortest path** from u to v is defined to be any path p with weight $w(p) = \delta(u, v)$.

In the single-source shortest-paths problem, our goal is given a **source** $s \in V$, we wish to determine a shortest path $s \rightsquigarrow v$ for all vertices $v \in V$.

Optimal substructure of a shortest path. The first observation we can make is that shortest paths exhibit the optimal substructure property (thus opening up this problem to a dynamic or greedy solution):

Lemma 9.1.1 (Subpaths of shortest paths are shortest paths)**.** *Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.*

Proof. Suppose we decompose p into three paths $v_0 \rightsquigarrow^{p_{0i}} v_i \rightsquigarrow^{p_{ij}} v_j \rightsquigarrow^{p_{jk}} v_k$. Then $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Assume towards a contradiction there is a path $p'_{ij} : v_i \rightsquigarrow v_j$ with $w(p'_{ij}) < w(p_{ij})$. Then $v_0 \rightsquigarrow^{p_{0i}} v_i \rightsquigarrow^{p'_{ij}} v_j \rightsquigarrow^{p_{jk}} v_k$ is a path

$v_0 \rightsquigarrow v_k$ with weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$, contradicting the assumption that p is a shortest path $v_0 \rightsquigarrow v_k$. □

Negative-weight edges. As stated, our single-source shortest-paths problem allows for the possibility of edges with negative weights. Provided there are no negative-weight cycles on some path $s \rightsquigarrow v$, then this is not an issue and our BELLMAN-FORD algorithm will handle this case. However, if there is a negative weight cycle on some path $s \rightsquigarrow v$, then $\delta(s, v)$ is not well-defined. This is because we can traverse the negative-weight cycle arbitrarily many times to produce a path of arbitrarily large (but negative) weight. In this case, we define $\delta(s, v) := -\infty$. Our BELLMAN-FORD algorithm will also be able to detect the presence of negative-weight cycles which are reachable from s.

Cycles. In general our shortest paths will not contain any cycles. As we just mentioned, we will not allow negative-weight cycles to appear. If a path contains a positive-weight cycle, then by removing the cycle we will obtain a path of lower weight. Thus a shortest weight path cannot contain a positive-weight cycle. However, a shortest-path can contain a 0-weight cycle. However, these cycles are superfluous and removing them would result in another shortest path of the same weight. Thus without loss of generality we will assume that our shortest paths do not contain any cycles. In particular, we can restrict our attention to acyclic paths containing at most $|V| - 1$ edges.

Representing shortest paths. Just like in BFS and DFS, in our algorithm we will represent the shortest paths by giving each vertex v a **predecessor** attribute $v.\pi$. This attribute will be either another vertex or NIL. Ultimately, if v is reachable from s, then upon termination the chain of predecessors from v running back to s will be the reverse of a shortest-path from s to v.

However, during the running of our algorithm, the predecessors may not actually represent shortest paths; only upon termination will we expect this. At all times our predecessors will give rise to a **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ defined as follows:

$$V_\pi \;=\; \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

with edge set:

$$E_\pi \;=\; \big\{(v.\pi, v) \in E : v \in V_\pi \setminus \{s\}\big\}.$$

Our goal will be to show that upon termination, the predecessor subgraph is a *shortest-path tree*. A **shortest-path tree** rooted at s is a directed subgraph $G' = (V', E')$ of G such that

(1) V' is the set of vertices reachable from s in G,
(2) G' forms a rooted tree with root s, and
(3) for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

In general, neither shortest paths nor shortest-path trees are unique.

Relaxation. The BELLMAN-FORD algorithm (and Dijkstra's algorithm; see [**1**, §24.3]) will use the technique of **relaxation**. Each vertex v will have an attribute $v.d$, a **shortest-path estimate** which represents an upper bound on the weight of a shortest path from s to v. Over time the relaxation process will gradually lower

the values of $v.d$ until finally these attributes converge to the actual shortest-path weights.

The first step will be to initialize all vertices with appropriate values for the attributes $v.d$ and $v.\pi$:

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2        v.d = ∞
3        v.π = NIL
4  s.d = 0
```

Given an edge (u, v), **relaxing** the edge (u, v) involves checking to see if we can improve the shortest path estimate $v.d$ by first going through u and then along the edge (u, v). If so, then we update $v.d$ and $v.\pi$ accordingly:

RELAX(u, v, w)

```
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

The BELLMAN-FORD algorithm works by first calling INITIALIZE-SINGLE-SOURCE and then repeatedly calling RELAX on various edges. It will relax each edge $|V| - 1$ times. The challenge will be to show that the way in which we relax edges guarantees that G_π is a shortest-path tree upon termination.

Properties of shortest paths and relaxation. To assist with the analysis of BELLMAN-FORD, we will use the following six properties of shortest paths and relaxation. Each of these properties assumes that the only thing that has been done with the graph is calling INITIALIZE-SINGLE-SOURCE and then calling RELAX some number of times on various edges; these are the only two ways in which the attributes of vertices in V get modified by our algorithm:

Lemma 9.1.2 (Triangle inequality)**.** *For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.*

Lemma 9.1.3. *We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.*

Lemma 9.1.4 (No-path property)**.** *If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.*

Lemma 9.1.5 (Convergence property)**.** *If $s \rightsquigarrow u \to v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.*

Lemma 9.1.6 (Path-relaxation property)**.** *If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, thenn $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.*

Lemma 9.1.7 (Predecessor-subgraph property)**.** *Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.*

9.2. The Bellman-Ford algorithm

In this section we study the **Bellman-Ford algorithm** which efficiently solves the single-source shortest-path problem. The Bellman-Ford algorithm also returns a boolean value depending on whether there is a negative-weight cycle reachable from s: it returns FALSE if there is such a cycle, and TRUE otherwise. The algorithm proceeds by continually relaxing edges until the values $v.d$ converge to the actual shortest-path weights and the predecessors $v.\pi$ converge to a shortest-path tree.

BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

BELLMAN-FORD works as follows:

(1) In line 1 all the attributes are initialized with a call to INITIALIZE-SINGLE-SOURCE.
(2) In the **for** loop of lines 2-4, every edge of the graph is relaxed $|V| - 1$ times. Note that the way the iteration is done, between two successive relaxations of a given edge, all other edges are relaxed again.
(3) In lines 5-8 we check if there are any negative weight cycles in the graph.

The running time of BELLMAN-FORD is $\Theta(VE)$: the initalization in line 1 takes $\Theta(V)$ time, lines 2-4 take $\Theta(VE)$ time, and lines 5-8 take $\Theta(E)$ time.

Lemma 9.2.1. *Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2-4 of* BELLMAN-FORD*, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.*

PROOF. Consider a vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path, where $v_0 = s$ and $v_k = s$. Since shortest paths are simple, p has at most $|V|-1$ edges so $k \leq |V|-1$. Each of the $|V|-1$ iterations of the **for** loop in lines 2-4 relaxes all $|E|$ edges. In particular, in the ith iteration, (v_{i-1}, v_i) gets relaxed. By the path-relaxation property, it follows that $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. □

Corollary 9.2.2. *Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$. Then, for each vertex $v \in V$, there is a path from s to v if and only if* BELLMAN-FORD *terminates with $v.d < \infty$ when it is run on G.*

PROOF. Exercise. □

Theorem 9.2.3 (Corrected of the Bellman-Ford algorithm)**.** *Let* BELLMAN-FORD *be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns* TRUE*, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the*

predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

PROOF. We have two cases to consider:

(Case 1) Suppose G does not contain any negative-weight cycles. We first claim that at termination, $v.d = \delta(s,v)$ for all $v \in V$. If v is reachable from s, then this follows from Lemma 9.2.1. Otherwise, if v is not reachable from s, then this follows from the no-path property. By the predecessor-subgraph property, it follows that G_π is a shortest-paths tree. Finally, we must show that BELLMAN-FORD returns TRUE. Note that at termination for every edge $(u,v) \in E$,

$$\begin{aligned} v.d &= \delta(s,v) \\ &\le \delta(s,u) + w(u,v) \quad \text{by triangle inequality} \\ &= u.d + w(u,v), \end{aligned}$$

so none of the **if** tests in line 6 are passed. Thus BELLMANFORD returns TRUE.

(Case 2) Suppose G contains a negative-weight cycle reachable from s. Suppose this cycle is $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Assume towards a contradiction that BELLMAN-FORD returns TRUE. Then $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing these inequalities yields

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\le \sum_{i=1}^{k} \bigl(v_{i-1}.d + w(v_{i-1}, v_i)\bigr) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

Sine $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$, and so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d.$$

By Corollary 9.2.2, each $v_i.d$ is finite. Thus

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

a contradiction. Thus BELLMAN-FORD returns FALSE. □

APPENDIX A

Pseudocode conventions and Python

When discussing algorithms in a theoretical context (which is what we do in this class), it is often beneficial to describe the algorithms in as human-readable a form as possible. Thus, instead of specifying an algorithm in a language like C, C++, Java, or Python, we will instead write it in **pseudocode**. Pseudocode is in many ways similar to the syntax any number of modern computer languages, except that it emphasizes clarity and readability and it downplays technical issues of software engineering, memory management, or specific idiosyncrasies of any one particular language.

A.1. Pseudocode conventions

In these notes we will follow the same pseudocode conventions as in [**1**]. The textbook summarizes the conventions on pages 20-22, although we will elaborate a little more here. We also present the conventions in the order in which they get used for our algorithms.

Assignment. A *variable* in mathematics is typically some symbol which represents an unknown quantity of some type which we want to solve for. In an algorithm, a *variable* is a symbol or name which gets assigned some specific value. For example, in the algorithm TRIANGLE in Section 1.3 we introduce a variable *Sum* which initially is assigned the value 0, but during the course of the algorithm its value is constantly updated as our index increases. We initially assigned the variable *Sum* with the value 0 in Line 1 using the code:

$Sum = 0$

Here the $=$ symbol is performing the action of **assignment**. In general, a line of pseudocode of the form

variable $=$ *expression*

has the effect of first evaluating whatever *expression* refers to, then assigning (or reassigning) the *variable* to be that value.

As an example, consider the following two lines of pseudocode:

1 $number = 1$
2 $number = number + 1$

What does this code do? The first line assigns the variable *number* the value of 1. Next the second line first computes the expression $number + 1$, which is $1 + 1 = 2$, then it (re)assigns this value to the variable *number*. After these two lines of code are finished, the value of *number* is 2, not 1.

The moral of the story here is that the expression $=$ in pseudocode is *not* an assertion of equality (like it is in mathematics). Instead it is an instruction for a certain action to be performed (the action of *assignment*).

Comments. In line 2 of the algorithm TRIANGLE in Section 1.3 we had the pseudocode:

```
1  // Initializes Sum to 0
```

This is referred to as a *comment*. A **comment** in pseudocode (or regular code) is a non-executable statement which serves no purpose other than to give commentary to the human reader of the pseudocode what is going on. In the real world, it is very important to *document* your code with comments to help explain what your code does to the next person who needs read and edit your code (which might be yourself a few years later).

For loops. In the algorithm TRIANGLE in Section 1.3 we had the following pseudocode:

```
3  for j = 0 to n
4      Sum = Sum + j
5      // Replaces the current value of Sum with Sum + j
6      // This has the effect of adding j to Sum
```

This is an example of a **for** loop. A **for** loop is used if we want to run a set of instructions a certain number of times. In this example, the lines 4-6 will be executed $n+1$ times, whatever the value of n happens to be. The j in the **for** loop is often called the **counter** of the **for** loop. This is a variable that keeps track of which iteration of the **for** loop the program currently is in. For instance, if $n = 2$, then the above code will run as follows:

(1) In line 3, the counter variable gets set to $j = 0$.
(2) Next lines 4-6 get executed, with $j = 0$.
(3) Then we go back to line 3, the counter variable gets incremented to $j = 1$.
(4) Then lines 4-6 get executed, with $j = 1$.
(5) Then we go back to line 3, the counter variable gets incremented to $j = 2$.
(6) Then lines 4-6 get executed, with $j = 2$.
(7) Finally we go back to line 3 one last time. The counter variable gets incremented to $j = 3$. However the **to** n part (with $n = 2$) tells us that we are done with the **for** loop, since $j > n$, so we are now finished with this section of the program. If there is no more pseudocode after line 6, then the program terminates. Otherwise we proceed to line 7.

One of the advantages of using a **for** loop is that we can run a section of code a *variable* number of times. In TRIANGLE, we don't know in advance how many times we will need to run lines 4-6, since this depends on the specific value of n which is supplied as an argument.

Boolean expressions and equality.

If-then-else.

While loops.

Arrays.

A.2. Python

In this section we discuss the Python equivalents of the above pseudocode constructs.

Assignment. The operation of *assignment* works the same as in pseudocode. For example, if we run the following Python code:

```
sum = 0
print('Current value of variable sum is'+sum)
sum = sum+1
print('Current value of variable sum is'+sum)
```

then this will print:

```
Current value of variable sum is 0
Current value of variable sum is 1
```

Comments. In Python, comments are written using the symbol `#`. For example, the following Python code:

```
print('Hello world!')
#This is a comment
print(2+2) #This is also a comment, it can appear on the same line (
    but to the right of) actual code. If the comment is too long then
    it will automatically be continued to the next line when displayed
    , although this entire comment is still considered ``line 3''
```

will print:

```
Hello world!
4
```

Bibliography

1. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2009.
2. Allen Gehret, *Lecture notes for Math131A: Introduction to Real Analysis*, 2019, URL: `https://www.math.ucla.edu/~allen/131aSSC19_lecture_notes.pdf`. Last revised December 5, 2019.
3. ———, *Lecture notes for Math61: Introduction to Discrete Structures*, 2020, URL: `https://www.math.ucla.edu/~allen/61W20_lecture_notes.pdf`. Last revised March 10, 2020.
4. Michael T Goodrich and Roberto Tamassia, *Algorithm design: foundation, analysis and internet examples*, John Wiley & Sons, 2002.
5. Jon Kleinberg and Eva Tardos, *Algorithm design*, Pearson Education India, 2006.
6. Donald E. Knuth, *The art of computer programming*, vol. 1, Pearson Education, 1997.
7. Yiannis Moschovakis, *Notes on set theory*, second ed., Undergraduate Texts in Mathematics, Springer, New York, 2006. MR 2192215

Index