Lecture notes for Math182: Algorithms
Draft: Last revised June 23, 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

List of Figures v

**Introduction** ix

- Prerequisites xi
- Conventions and notation xi
- Acknowledgements xii

Chapter 1. Discrete mathematics and basic algorithms 1

1.1. Induction 1
1.2. Summations 2
1.3. Triangular number algorithms 5
1.4. Common functions 9
1.5. Fibonacci numbers 14
1.6. Exercises 14

Chapter 2. Asymptotics 17

2.1. Asymptotic notation 17
2.2. The Euclidean Algorithm 19
2.3. Exercises 19

Chapter 3. Sorting algorithms 21

3.1. Insertion sort 21
3.2. Merge sort 21
3.3. Lower bound on comparison-based sorting 21

Chapter 4. Divide-and-Conquer 23

4.1. The maximum-subarray problem 23

Chapter 5. Data structures 25

Chapter 6. Dynamic programming 27

Chapter 7. Greedy algorithms 29

Chapter 8. Elementary graph algorithms 31

Chapter 9. Path algorithms and network flow 33

Chapter 10. The Gale-Shapley algorithm 35

Chapter 11. P vs. NP 37

Appendix A. Pseudocode conventions and Python 39
A.1. Pseudocode conventions 39
A.2. Python 41
Bibliography 43
Index 45
Abstract

The objectives of this class 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 [5], [1], and [4]. Any and all questions, comments, typos, suggestions concerning these notes are enthusiastically welcome and greatly appreciated.
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. 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:

---

1Here 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 $10^{10^6}$ 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\footnote{This is the subject studied in Math151A/B.}. 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 \footnote{This is the subject studied in Math151A/B.} 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.
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 $\mathbb{N} = \{0, 1, 2, \ldots \}$.

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 “$\iff$” 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 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

\begin{enumerate}
  \item $a = bq + r$
  \item $0 \leq r < b$.
\end{enumerate}

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

\footnote{See [6] 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 \ref{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 \cite[1.4.2]{3}.

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:

\[ S := \{ n \in \mathbb{N} : P(n) \text{ is false} \} \subseteq \mathbb{N}. \]

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 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:

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

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:

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

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

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

---

2At 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 \( \pi : \mathbb{Z} \rightarrow \mathbb{Z} \) 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

\[
\sum_{0 \leq j \leq n} x^j = \frac{1 - x^{n+1}}{1 - x}.
\]

**Proof.** Note that

\[
\sum_{0 \leq j \leq n} x^j = 1 + \sum_{1 \leq j \leq n} x^j \quad \text{by 1.2.4}
\]

\[
= 1 + x \sum_{1 \leq j \leq n} x^{j-1} \quad \text{by 1.2.1}
\]

\[
= 1 + x \sum_{0 \leq j \leq n-1} x^j \quad \text{by 1.2.2}
\]

\[
= 1 + x \sum_{0 \leq j \leq n} x^j - x^n \quad \text{by 1.2.4}
\]

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

**Triangular Numbers 1.2.6.** Suppose \( n \geq 0 \). Then

\[
\sum_{0 \leq j \leq n} j = \frac{n(n + 1)}{2}.
\]
Proof. Note that
\[
\sum_{0 \leq j \leq n} j = \sum_{0 \leq n-j \leq n} (n - j) \quad \text{by 1.2.2}
\]
\[
= \sum_{0 \leq j \leq n} (n - j) \quad \text{by simplifying the domain}
\]
\[
= \sum_{0 \leq j \leq n} n - \sum_{0 \leq j \leq n} j \quad \text{by 1.2.3}
\]
\[
= n(n + 1) - \sum_{0 \leq j \leq n} j.
\]
By comparing the first term with the last and solving for \(\sum_{0 \leq j \leq n} j\), we get the desired formula. \(\square\)

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 \(n \geq 0\) 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
\[
\sum_{k=0}^{\infty} x^k = \lim_{n \to \infty} \frac{1 - x^{n+1}}{1 - x} = \frac{1}{1 - x}. \quad \square
\]

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 \(n\)th 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. This introduces the important idea of a loop invariant.

Why is Triangle correct? Intuitively it is because for each time we run the for loop on Lines 3-6 the value of $Sum$ is the partial sum $\sum_{i=1}^{j} i$ where $j$ is the index variable which runs from 0 to $n$. Thus, after the last iteration of the for loop, when
1.3. TRIANGULAR NUMBER ALGORITHMS

$j = n$, the value of $Sum$ is $\sum_{i=0}^{n} i$, which is the value we are computing. 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 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. For this we need to prove three things about the loop invariant:

(1) **Initialization:** 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 $j = 0$ the first time we encounter line 3 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 before an iteration of the loop, it remains true after the body of the for loop is run another time and prior to the next iteration. Usually this step is analogous to the inductive step of an induction proof.

(3) **Termination:** We need to show that after the loop terminates, the loop invariant gives us a useful property that helps show that the algorithm is correct.

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

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

**Proof.** The algorithm begins by assigning $Sum = 0$. Next, we will prove that the above loop invariant is correct.

(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 $j = k$ for some $0 \leq k < n$. Now we have $j = k + 1$ just prior to running line 4 and since the loop invariant is assumed correct, this means current value of $Sum$ is $\sum_{i=0}^{\max(j-1,0)} i = \sum_{i=0}^{k} i$. Now in line 4 we compute $Sum + j = \sum_{i=0}^{k} i + (k + 1) = \sum_{i=0}^{k+1} i$, and reassign $Sum$ to this value. Thus $Sum = \sum_{i=0}^{k+1} i$. Just prior to the next iteration, we have $j = k + 2$, and so $Sum = \sum_{i=0}^{k+1} + \sum_{i=0}^{\max(j-1,0)} i$, as desired.

(Termination) After the for loop terminates, 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. \(\text{Sum} = n\)  
   \(\text{cost: } c_1 \text{ times: } 1\)

2. // Initializes Sum to n  
   \(\text{cost: } 0 \text{ times: } 1\)

3. \(\text{Sum} = \text{Sum} \cdot (n + 1)\)  
   \(\text{cost: } c_2 \text{ times: } 1\)

4. // Multiplies Sum by n + 1  
   \(\text{cost: } 0 \text{ times: } 1\)

5. \(\text{Sum} = \text{Sum}/2\)  
   \(\text{cost: } c_3 \text{ times: } 1\)

6. // Divides by 2  
   \(\text{cost: } 0 \text{ times: } 1\)

7. return Sum  
   \(\text{cost: } c_4 \text{ times: } 1\)

Performing a similar running-time analysis 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. \(\text{return } n \cdot (n + 1)/2\)

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:

\[
\lfloor x \rfloor := \text{the greatest integer less than or equal to } x \quad \text{(floor of } x) \\
\lceil x \rceil := \text{the least integer greater than or equal to } x \quad \text{(ceiling of } x) 
\]

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 +, -, ·, /) 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 \( x \in \mathbb{R} \),

\[
x - 1 < \lfloor x \rfloor \leq x < \lceil x \rceil < x + 1
\]

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. \( \lfloor x + k \rfloor = \lfloor x \rfloor + k \)
5. \( \lceil x + k \rceil = \lceil x \rceil + k \)
6. \( x < k \iff \lfloor x \rfloor < k \)
7. \( k < x \iff k < \lceil x \rceil \)
8. \( x \leq k \iff \lfloor x \rfloor \leq k \)
9. \( \lfloor x \rfloor \leq k \iff k \leq \lfloor x \rfloor \)

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

\[
\lfloor \sqrt{\lfloor x \rfloor} \rfloor = \lfloor \sqrt{x} \rfloor
\]

**Proof.** Let \( m = \lfloor \sqrt{\lfloor x \rfloor} \rfloor \). 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

\[
\lfloor \sqrt{\lfloor x \rfloor} \rfloor = m = \lfloor \sqrt{x} \rfloor
\]
1.4. COMMON FUNCTIONS

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

\[
k \mapsto k \mod n : \mathbb{Z} \to \{0, \ldots, n-1\}
\]

defined for \( k \in \mathbb{Z} \) by

\[
k \mod n := \text{the unique } r \in \{0, \ldots, n-1\} \text{ such that there is } q \in \mathbb{Z}
\]

\[
such that k = nq + r \text{ and } 0 \leq r < n.
\]

In other words, \( k \mod n \) 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 \( k \mod 2 = 0 \) and \( k \) is odd iff \( k \mod 2 = 1 \).

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: \( n \mid k \)) if \( k \mod n = 0 \). Equivalently, \( n \) divides \( k \) if there exists \( q \in \mathbb{Z} \) such that \( k = nq \).

If \( n \mid k \) and \( q \in \mathbb{Z} \) is such that \( nq = k \), 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 mod 10 = 0, however 10 \( \nmid \) 51 because 51 mod 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 \( r := a \mod b \). Then

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

In particular, \( a \mod b = a - b \lfloor a/b \rfloor \).

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:

\[
p \text{ prime } :\iff p \neq 1 \& \forall d \in \mathbb{N} \,(d|p \to d = 1 \text{ or } d = p)
\]

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 \( a \mod{n} = b \mod{n} \), i.e., if \( a \) and \( b \) leave the same remainder upon division by \( n \). Equivalently:

\[
 a \equiv b \pmod{n} \iff n|a - b \\
\iff \text{there exists } q \in \mathbb{Z} \text{ such that } nq = a - b \\
\iff a \mod{n} = b \mod{n}.
\]

For example, since \( 12 \mod{5} = 2 = 7 \mod{5} \), 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 2.2. 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:

\[
(b^k) := \begin{cases} 
  b \times \cdots \times b & \text{if } k > 0 \\
  1 & \text{if } k = 0 \\
  b^{-1} \times \cdots \times b^{-1} & \text{if } k < 0 
\end{cases}
\]

This exponential function satisfies the following properties:

(P1) (Exponent rule) For every \( k, \ell \in \mathbb{Z} \), \( b^k b^\ell = b^{k+\ell} \).

(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 \((\dagger)\) 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 out there is a unique function \( \mathbb{R} \to \mathbb{R} \) which satisfies properties (P1) and (P2) above, with \( k, \ell \) 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:

\[
y = \log_b x :\iff b^y = x.
\]

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 \( b \neq 1 \))

---

4See, for instance, [2 §1.6] for an explanation how this can be done.
(4) \( \log_b a = \frac{\log_c a}{\log_c b} \) (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) \( \frac{d}{dx} \log_b x = \frac{1}{x \ln b} \)
(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) \( \ln(t) = \int_1^t \frac{dx}{x} \)

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

Fibonacci algorithms.

FIBONACCI(n)
1    if \( n = 0 \) or \( n = 1 \)
2        return \( n \)
3    else
4        return FIBONACCI(n - 1) + FIBONACCI(n - 2)

FIBONACCIFAST(n)
1    if \( n \leq 1 \)
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 \( n \)
10       // Use recursive formula for \( j \)th Fibonacci number to fill in \( j \)th entry
11    return \( F[n] \)

1.6. Exercises

Exercise 1.6.1. Write out the following two sums in full:
(1) \( \sum_{0 \leq k \leq 5} a_k \)
(2) \( \sum_{0 \leq k^2 \leq 5} a_{k^2} \)

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

\[
P(x) = \sum_{k=0}^{n} a_k x^k
= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n))),
\]

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

```plaintext
1  y = 0
2  for i = n downto 0
3    y = a_i + x \cdot y
```

(1) In terms of \( \Theta \)-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 1.6.5.** Suppose \( m, n \in \mathbb{Z} \) are such that \( m > 0 \). Prove that

\[
\left\lfloor \frac{n}{m} \right\rfloor = \left\lfloor \frac{n + m - 1}{m} \right\rfloor
\]

This gives us another *reflection principle* between floors and ceilings when the argument is a rational number.
Exercise 1.6.6. Find a necessary and sufficient condition on the real number $b > 1$ such that
\[
\left\lfloor \log_b x \right\rfloor = \left\lfloor \log_b \left\lfloor x \right\rfloor \right\rfloor
\]
holds for all real numbers $x \geq 1$.

Exercise 1.6.7. 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 1.6.8. 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 1.6.9. Suppose $x, y, z \in \mathbb{Z}$ are such that $y, z \geq 1$. Prove that $z (x \mod y) = (z x) \mod (z y)$.

Exercise 1.6.10. Suppose $a, b, r, s \in \mathbb{Z}$ are such that $r, s \geq 1$. Prove that if $a \mod rs = b \mod rs$, then $a \mod r = b \mod r$ and $a \mod s = b \mod s$.

Exercise 1.6.11. Suppose $b > 1$. Express $\log_b \log_b x$ in terms of $\ln \ln x$, $\ln \ln b$, and $\ln b$.

Exercise 1.6.12 (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 1.6.13 (Programming exercise). Let $F_0, F_1, F_2, \ldots$ be the sequence of Fibonacci numbers. Compute the following sum:
\[
\sum_{F_n \leq 10^7 \land F_n \mod 2 = 0} F_n
\]

Exercise 1.6.14 (Programming exercise). Determine the following number:
\[
\min \{ n \in \mathbb{N} : n \geq 1 \text{ and for each } k \in \{1, 2, \ldots, 30\}, k|n \}
\]
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 1.6.15 (Programming exercise). Let $P_n$ denote the $n$th prime number. So $P_1 = 2, P_2 = 3, P_3 = 5, P_4 = 7, \ldots$. Find $P_{100000}$.

Exercise 1.6.16 (Programming exercise). A unit fraction contains 1 in the numerator. The decimal representation of the unit fractions with denominators 2 to 10 are given:
\[
\begin{align*}
1/2 &= 0.5, & 1/3 &= 0.(3), & 1/4 &= 0.25, & 1/5 &= 0.2, & 1/6 &= 0.1(6), \\
1/7 &= 0.(142857), & 1/8 &= 0.125, & 1/9 &= 0.(1), & 1/10 &= 0.1
\end{align*}
\]
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.
Are not 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 \(\Theta\), \(O\), and \(\Omega\).

**\(\Theta\)-notation.** Recall that in Section 1.3 we saw that \(\text{TRIANGLE}(n)\) 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 \(\Theta(g(n))\) to be the following set of functions:

\[
\Theta(g(n)) := \{ f(n) : \text{there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}
\]

\[
0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0
\]

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 \(n \geq n_0\), \(0 \leq c_1 n \leq 2n + 1 \leq c_2 n\). We claim that \(c_1 := 1, c_2 := 3\) and \(n_0 := 1\) work. Indeed, suppose \(n \geq n_0 = 1\), then \(n \leq 2n \leq 2n + 1\) and \(2n + 1 \leq 2n + n = 3n\).
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 “∈” 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 \( N \to \mathbb{Z} \) (with domain \( N \)), 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-notations, 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.
2.3. Exercises

\( O \)-notation.
\( \Omega \)-notation.
\( o \) and \( \omega \)-notation.

2.2. The Euclidean Algorithm

\begin{verbatim}
EUCLID(a, b)
1  if b == 0
2     return a
3  else return EUCLID(b, a mod b)
\end{verbatim}

2.3. Exercises

Exercise 2.3.1. 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 2.3.2. Prove that for all \( m \in \mathbb{N} \), \( (\ln n)^m = o(n) \).
CHAPTER 3

Sorting algorithms

3.1. Insertion sort

\begin{verbatim}
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] i = i - 1
7         A[i+1] = [key]
\end{verbatim}

3.2. Merge sort

\begin{verbatim}
MERGE(A,p,q,r)
1   n1 = q - p + 1
2   n2 = r - q
3   let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i - 1]
6   for j = 1 to n1
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16      else A[k] = R[j]
17         j = j + 1
\end{verbatim}

\begin{verbatim}
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)
\end{verbatim}

3.3. Lower bound on comparison-based sorting
CHAPTER 4

Divide-and-Conquer

4.1. The maximum-subarray problem

\textsc{Find-Max-Crossing-Subarray}(A, low, mid, high)

1 \hspace{1em} \textit{left-sum} = -\infty
2 \hspace{1em} \textit{sum} = 0
3 \hspace{1em} \textbf{for} \hspace{0.5em} i = mid \hspace{0.5em} \textbf{downto} \hspace{0.5em} low
4 \hspace{1em} \hspace{1em} \textit{key} = \textit{key} + A[i]
5 \hspace{1em} \hspace{1em} \textbf{if} \hspace{0.5em} \textit{sum} > \textit{left-sum}
6 \hspace{1em} \hspace{1em} \hspace{1em} \textit{left-sum} = \textit{sum}
7 \hspace{1em} \hspace{1em} \hspace{1em} \textit{max-sum} = i
8 \hspace{1em} \hspace{1em} \textit{right-sum} = -\infty
9 \hspace{1em} \textit{sum} = 0
10 \hspace{1em} \textbf{for} \hspace{0.5em} j = mid + 1 \hspace{0.5em} \textbf{to} \hspace{0.5em} high
11 \hspace{1em} \hspace{1em} \textit{sum} = \textit{sum} + A[j]
12 \hspace{1em} \hspace{1em} \textbf{if} \hspace{0.5em} \textit{sum} > \textit{right-sum}
13 \hspace{1em} \hspace{1em} \hspace{1em} \textit{right-sum} = \textit{sum}
14 \hspace{1em} \hspace{1em} \hspace{1em} \textit{max-right} = j
15 \hspace{1em} \textbf{return} \hspace{0.5em} (\textit{max-left}, \textit{max-right}, \textit{left-sum} + \textit{right-sum})
CHAPTER 5

Data structures
CHAPTER 6

Dynamic programming
CHAPTER 7

Greedy algorithms
CHAPTER 8

Elementary graph algorithms
CHAPTER 9

Path algorithms and network flow
CHAPTER 10

The Gale-Shapley algorithm
CHAPTER 11

$P \text{ vs. } NP$
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:

\[ \text{Sum} = 0 \]

Here the = symbol is performing the action of assignment. In general, a line of pseudocode of the form

\[ \text{variable} = \text{expression} \]

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. \( \text{number} = 1 \)
2. \( \text{number} = \text{number} + 1 \)

What does this code do? The first line assigns the variable number the value of 1. Next the second line first computes the expression \( \text{number} + 1 \), 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.
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:

```python
sum = 0
print('Current value of variable sum is ' + str(sum))
sum = sum + 1
print('Current value of variable sum is ' + str(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:

```python
print('Hello world!')
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!
Bibliography

Index

for loop counter, 10
assignment, 39
asymptotically nonnegative, 18
asymptotically positive, 18
asymptotically tight bound, 18
base of logarithm, 14
binary logarithm, 14
ceiling, 10
chain of variable for summations, 4
congruence modulo n, 12
delimited summation notation, 2
distributive law, 6
divides, 11
Division Algorithm, 1
divisor, 11
dummy variable, 2
factor, 11
floor, 10
generalized summation notation, 9
geometric series, 5
geometric sum, 4
Horner’s rule, 15
index variable, 2
initialization, 7
interchanging order of summation, 4
logarithm, 13
loop invariant, 7
maintenance, 7
modulus operator, 11
natural logarithm, 14
prime number, 11
Principle of Induction, 2
pseudocode, 39
quotient, 11
remainder, 1
summation notation, 2
termination, 7
Well-Ordering Principle, 1