CSCE 411 Course Notes

Calvin Beideman

Fall 2026

1 Foundational Mathematical Notation

In this chapter we establish some important definitions and notation that will be used throughout the course. Hopefully many of these definitions will be familiar to you from previous courses, but there may be concepts that are new to you or slight differences between the notation used in this course and what you are familiar with.

1.1 Sets

A **set** is an unordered collection of distinct things. Often these things will be numbers, but they could also be strings, or algorithms, or other sets, or basically anything else you can think of. We call the things in a set its **elements**. Sets can be finite or infinite. We can represent a finite set with a list of its elements, separated by commas and enclosed in curly braces. So $\{1, 2, 3, 4, 5\}$ is a set, as is $\{\text{“apple”}, \text{“banana”}, \text{“orange”}\}$, as is $\{\text{“apple”}, \sqrt{3}, -100, \{\{2\}\}\}$.

The **size** or **cardinality** of a set is the number of elements that it contains. We use $|A|$ to denote the size of the set A. For example, $|\{1, 2, 5\}| = 3$ and $|\{\{\}, \{1, 2, 3, 4\}, \{7\}, \{\{5\}, \{6\}, \{9\}\}\}| = 4$. A set cannot contain multiple copies of the same element, $\{2, 2\}$ is the same set as $\{2\}$. A set can, however, have zero elements. The set $\{\}$, which contains no elements, is known as the **empty set**, and we denote it with the special symbol $\emptyset$.

We use the symbol $\in$ to say that an object is part of a set. For example $2 \in \{1, 2, 5\}$. To say that an object is not an element of a set, we can use the symbol $\notin$. For example, $4 \notin \{1, 2, 5\}$. For two sets A and B, if *all* of the elements of A are also in B, we say that A is a **subset** of B, written $A \subseteq B$ (conversely we say that B is a superset of A).

Theoretical computer science (including algorithm design and analysis) is built on discrete mathematics. Unlike ”continuous mathematics” (such as calculus) which focuses on real numbers, discrete mathematics is built on the integers. We use $\mathbb{Z}$ to denote the set of integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$. In many contexts, we don't care about negative values or they don't make sense, so we restrict ourselves to the set of non-negative integers:

$$\mathbb{Z}_{\geq 0} := \{0, 1, 2, 3, \ldots\}.$$

Note that the notation $:=$ means “is defined as”. Sometimes we don't care about 0 either, and use the set of positive integers

$$\mathbb{Z}^+ := \{1, 2, 3, \ldots\}.$$

Many authors (including CLRS) use the term “natural numbers” for $\mathbb{Z}_{\geq 0}$, and denote it $\mathbb{N}$. However, this notation can be confusing, because other authors use ”natural numbers” and $\mathbb{N}$ to refer to $\mathbb{Z}^+$. To avoid confusion over whether 0 is included in the natural numbers, these notes and your homeworks/exams will avoid the term and instead use “non-negative integers” or “positive integers” and the symbols $\mathbb{Z}_{\geq 0}$ or $\mathbb{Z}^+$.

Listing all of the elements works great for describing small sets, but if a set has hundreds of elements, or worse, infinitely many, this notation is not very effective. In the previous paragraphs we defined an infinite set using ellipses i.e. we represented the infinite set of positive integers with $\{1, 2, 3, \ldots\}$. This approach can work when the pattern that the elements follow is very obvious, but

for more complicated sets it is likely to cause confusion. A better approach is to use "set builder notation" where we explicitly describe the pattern that the elements of the set follow. This notation typically takes the form $\{x \in S \mid P(x)\}$, where P is a function outputting true or false (a predicate) and S is some set which has already been defined. This expression is read as "The set of x in S such that $P(x)$". Sometimes you will also see this notation with a colon instead of a vertical bar (i.e. $\{x \in S : P(x)\}$). Set builder notation allows us to express any subset of a known set by describing some property that all of the elements of the set satisfy. For example, we can formally describe the set of even integers as $\{x \in \mathbb{Z} \mid x \bmod 2 = 0\}$.

Sometimes instead of describing a subset of an existing set, we want to create a new set by applying some function f to certain elements of an existing set. For this, we can use an alternative form of set builder notation: $\{f(x) \mid x \in S \text{ and } P(x)\}$. This defines a set, which contains $f(x)$ for each $x \in S$ which satisfies the property given by P. Note that some elements of our new set may be $f(x)$ for several different $x \in S$, but a set cannot contain the same element more than once. This notation gives us an alternative way of describing the set of even integers $\{2x \mid x \in \mathbb{Z}\}$, i.e. a number is an even integer if it can be written as $2x$ where x is an integer. Similarly, we can define the set of rational numbers by:

$$\mathbb{Q} := \{p/q \mid p, q \in \mathbb{Z}, q \neq 0\}.$$

That is, a number is a rational number if it can be written as p/q where p is an integer and q is a non-zero integer.

The real numbers "fill in the gaps" of the rational numbers by including infinitely many irrational values like $\sqrt{2}$ and π. We use $\mathbb{R}$ to denote the set of real numbers. We will not formally define the real numbers or make frequent use of them in this course. Just as we denote the positive integers with $\mathbb{Z}^+$, we denote the positive rationals with $\mathbb{Q}^+$, and the positive reals with $\mathbb{R}^+$. Similarly, we denote the non-negative rationals with $\mathbb{Q}_{\geq 0}$ and the non-negative reals with $\mathbb{R}_{\geq 0}$.

1.2 Combining Sets

We saw in the previous section how to define a new set in terms of an existing set using set builder notation. We can also use this notation to define a new set in terms of two or more existing sets. Certain ways of combining sets are common enough that they have their own special notation. For two sets S and T:

- The **union** of S and T is
$$S \cup T := \{x \mid x \in S \text{ or } x \in T\}.$$
- The **intersection** of S and T is
$$S \cap T := \{x \mid x \in S \textit{ and } x \in T\}.$$
- The **difference** of S and T is
$$S \setminus T := \{x \mid x \in S \text{ and } x \notin T\}.$$
Sometimes the difference is also written as $S - T$.
- The **(Cartesian) product** of S and T is
$$S \times T := \{(x, y) \mid x \in S \text{ and } y \in T\}.$$
That is, the product of S and T is the set of all pairs where the first element is from S and the second is from T.

One final useful set operation is the **powerset** of a set S:

$$2^S := \{T \mid T \subseteq S\}.$$

For example, if $S = \{1, 2, 3\}$,

$$2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Note that if S is a finite set, $|2^S| = 2^{|S|}$. The notation $\mathcal{P}(S)$ is also sometimes used for the powerset.

1.3 Exponents and Logarithms

Exponents and logarithms arise frequently when analyzing algorithms, so you should be comfortable working with them.

For a real number x and a positive integer a, x^a is defined as the result of multiplying x with itself, a times. For example $5^3 = 5 \cdot 5 \cdot 5 = 125$. In the expression x^a, we call x the "base" and a the "exponent". We generally read x^a as "x to the a" with a few exceptions. In particular, x^2 is read as "x squared" and x^3 is read as "x cubed". Exponents satisfy the following properties which you should be familiar with. Don't treat these equations as facts to memorize—instead you should make sure that you understand *why* they're true so that when they're relevant you can apply them automatically.

Theorem 1. *If x and y are positive real numbers and a and b are positive integers then*

(a) $x^a x^b = x^{a+b}$

(b) $(x^a)^b = x^{ab}$

(c) $x^a y^a = (xy)^a$

Proof. (a) By definition x^a is the product of a copies of x and x^b is the product of b copies of x. So when we multiply them together we obtain the product of $a + b$ copies of x.

(b) $(x^a)^b$ is the product of b copies of x^a. Each x^a is the product of a copies of x. Thus, in total we are multiplying ab copies of x. Note that this equation also implies that $(x^a)^b = (x^b)^a$. Swapping the exponents like this is sometimes helpful for simplifying expressions.

(c) $x^a y^a$ is the result of multiplying x with itself a times, multiplying y with itself a times, and then multiplying the two results. Since multiplication is commutative (xy is the same as yx), we can rearrange the order of the multiplications without changing the result. In particular, we can pair up the x's and y's to rewrite our product of a copies of x and a copies of y as the product of a xy pairs, i.e. $(xy)^a$.

□

The definition of exponentiation as "repeated multiplication" makes sense for positive integer exponents, but it doesn't really make sense for other values. What would something like $5^{-2/3}$ mean? How can we multiply a number with itself negative two thirds of a time? To extend the definition of exponentiation beyond repeated multiplication, mathematicians looked at what definitions would preserve the useful properties given in Theorem 1. In particular, these properties give us values for all rational exponents as follows:

- For every real number x, we define $x^0 = 1$, because to preserve property (a) we need $x^1 \cdot x^0 = x^1$. (Note that in particular we define $0^0 = 1$, unlike in some branches of mathematics where it is considered indeterminate).
- For every non-negative real number x and every positive integer b, we define $x^{1/b}$ as the (non-negative real) solution to $y^b = x$ (i.e. the number which when multiplied with itself b times gives x). This is because to preserve property (b) we need $(x^{1/b})^b = x^1 = x$. $x^{1/b}$ is also known as "the b-th root of x" and can be written with the alternative notation $\sqrt[b]{x}$. Just as x^2 and x^3 are called "x squared" and "x cubed", we refer to $x^{1/2}$ as "the square root of x" (typically written as $\sqrt{x}$, note that we omit the 2) and we refer to $x^{1/3}$ as "the cube root of x".
- For every non-negative real number x and all positive integers a and b, we define $x^{a/b}$ as $(x^{1/b})^a$. Once again, this preserves property (b).
- For every positive real number x and every positive rational number a, we define $x^{-a} = 1/x^a$. This is because to preserve property (a) we need $x^a x^{-a} = x^0 = 1$.

We've now defined x^a whenever x is a positive real number and a is a rational number. We could extend our definition to cover every real number a using limits, but the details are outside the focus of this course (since we will not need tools from calculus elsewhere). While we will omit the formal definition, you may assume in this course that the properties in Theorem 1 hold for all real numbers a and b.

Logarithms are the inverse of exponents. $\log_b x$ is defined as the power that you have to raise b to in order to get x, i.e. $\log_b x$ is the value of y for which $b^y = x$.

The following theorem gives some properties of logarithms which you should understand and be comfortable using. Part (c) in particular allows us to convert a logarithm from one base to another, which can be useful for analyzing divide and conquer algorithms that we will discuss later in the course.

Theorem 2. *If a is a real number and b, c, x, and y are positive real numbers with $b, c \neq 1$ then*

(a) $\log_b(xy) = \log_b(x) + \log_b(y)$

(b) $\log_b(x^a) = a \log_b(x)$

(c) $\log_b x = (\log_c x) \log_b c$

Proof. (a)

$$\begin{aligned} b^{\log_b(x)+\log_b(y)} &= b^{\log_b(x)} b^{\log_b(y)} && \text{by part (a) of Theorem 1} \\ &= xy && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{\log_b(x)+\log_b(y)} = xy$, by the definition of log, $\log_b(xy) = \log_b(x) + \log_b(y)$.

(b)

$$\begin{aligned} b^{a \log_b(x)} &= (b^{\log_b(x)})^a && \text{by part (b) of Theorem 1} \\ &= x^a && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{a \log_b(x)} = x^a$, by the definition of log, $\log_b(x^a) = a \log_b(x)$.

(c)

$$\begin{aligned} b^{(\log_c x) \log_b c} &= (b^{\log_b(c)})^{\log_c x} && \text{by part (b) of Theorem 1} \\ &= c^{\log_c x} && \text{by the definition of log} \\ &= x && \text{by the definition of log} \end{aligned}$$

Thus, since $b^{(\log_c x) \log_b c} = x$, by the definition of log, $\log_b x = (\log_c x) \log_b c$.

□

When we have an exponent where the base is a logarithm, we will often use the following shorthand notation to avoid extra parentheses

$$\log_b^k n := (\log_b n)^k.$$

For example, $(\log_2 5)^3$ can be written as $\log_2^3 5$.

1.4 Summations and Products

We use the symbol $\sum$ (the capital Greek letter sigma) to represent a summation. You can think of this like a mathematician's version of a for loop. If f is some function which has been defined on integers and a and b are integers, then the expression

$$\sum_{i=a}^{b} f(i)$$

is equivalent to running the pseudocode

```
sum ← 0
For i from a to b:
   sum ← sum +f(i)
Return sum
```

For example, $\sum_{i=3}^{5} i^2 = 3^2 + 4^2 + 5^2 = 50$ (note that the upper and lower bounds for our iteration variable i can be written either above and below the $\sum$, as in the first example, or just to the right, as in the example in this paragraph).

Sometimes we want to iterate over something other than a range of integers. $\sum$ notation can also be used for looping over the elements of an arbitrary set:

$$\sum_{x \in S} f(x)$$

This expression represents applying the function f to every element of the set S and adding up the results.

Just as we use $\sum$ to denote sums, we use $\prod$ (the capital Greek letter pi) to denote products. The use is exactly the same, except with $\prod$ notation we are *multiplying* the results of each iteration of the loop instead of adding them. For example, $\prod_{i=3}^{5} i = 3 \cdot 4 \cdot 5 = 60$.

The summation in the following theorem shows up frequently enough in analyzing algorithms that you should be familiar with it and know how to simplify it.

Theorem 3.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof. We note that

$$2\sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n + 1 + 2 + \cdots + (n-1) + n.$$

Since addition is commutative (we can rearrange the order in which we add numbers without changing the sum), we can rewrite this sum as

$$(1+n) + (2 + (n-1)) + (3 + (n-2)) + \cdots + ((n-2)+3) + ((n-1)+2) + (n+1).$$

This new version has n pairs of values and each pair adds up to $n+1$. Thus,

$$2\sum_{i=1}^{n} i = n(n+1),$$

and

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

□

We'll learn how to simplify some other summations in Chapter 3.

1.5 Other Important Mathematical Notation

For a real number x, we use $\lfloor x \rfloor$ to denote the *floor* of x, which is the largest integer less than or equal to x. Similarly we use $\lceil x \rceil$ to denote the *ceiling* of x, which is the smallest integer greater than or equal to x. For example:

- $\lfloor 4.5 \rfloor = 4$
- $\lceil 4.5 \rceil = 5$
- $\lfloor -2.7 \rfloor = -3$
- $\lceil -2.7 \rceil = -2$
- $\lfloor 7 \rfloor = 7$
- $\lceil 7 \rceil = 7$

For a non-negative integer n, we use $n!$ to denote n *factorial*, which is $n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$. For example, $5! = 120$. We define $0! := 1$.

1.6 Arrays and Strings

In this class, we will use many of the abstract data types and data structures you learned about in CSCE 221. The most important abstract data type is the **list** ADT, which represents an *ordered* collection of objects. Unlike a set, a list may contain multiple copies of the same value.

Lists are commonly implemented using the **array** data structure. The core property of arrays in contrast to other implementations of the list ADT (like linked lists) is that arrays allow for random access, meaning that we can get the element at any arbitrary position in the array in constant time, without having to first look through earlier elements of the array. As in many programming languages, we will use square brackets to denote indexing into an array. For example, $A[5]$ refers to the element at index 5 in the array A. Unlike some programming languages you may be familiar with (such as C++), in this course we will typically view arrays as beginning at index 1. Therefore we will use $A[1, \ldots, n]$ to denote an arbitrary array of size n.

We will sometimes use $A[i : j]$ to refer to the subarray starting at index i and ending at index j, i.e. $A[i, i+1, \ldots, j-1, j]$. For example, if A is the array $[3, 1, 4, 1, 5, 9]$ then $A[1] = 3$, and $A[2 : 4] = [1, 4, 1]$. Note that this convention is different from some programming languages like Python where $A[i : j]$ does not include $A[j]$.

An **alphabet** is a finite, nonempty set. We call the elements of the alphabet **symbols** or **characters**. Common choices in this course will be the English alphabet (the set containing the letters a through z) and the alphabet $\{0, 1\}$. We often use the character Σ as a variable for an alphabet (note that this is the same Greek letter used for describing summations, do not confuse the two uses). A **string** x over the alphabet Σ is a finite sequence of symbols $x_1 x_2 x_3 \ldots x_n$, where $x_i \in \Sigma$ for each $1 \leq i \leq n$. We typically write the symbols in the string right next to each other, but we can add commas or spaces if this would be ambiguous. We use $|x|$ (or sometimes $\text{len}(x)$) to denote the length or size of x, which is the number of symbols that it contains. We use Σ^n to denote the set of all length-n strings over the alphabet Σ. We use Σ^* to denote the infinite set of all strings over the alphabet Σ (of any length). Thus $0011010 \in \{0, 1\}^7$, and therefore also $0011010 \in \{0, 1\}^*$ (we call strings over the alphabet $\{0, 1\}$ **binary strings**), and if Σ is the English alphabet then

$$\text{hello} \in \Sigma^*.$$

The empty string is a valid string over any alphabet, and we denote it with the special character ε. Given two strings x and y over alphabets Σ and Γ, their **concatenation** is the string over the alphabet $\Sigma \cup \Gamma$ which consists of all the symbols of x followed by all the symbols of y. We denote x concatenated with y as $x \cdot y$ or simply xy. For example, if x is the binary string 010 and y is the binary string 1011, xy is the binary string 0101011. We use x^n to denote the string obtained by

concatenating x with itself n times. For example, $(01)^4$ is 01010101. Be careful not to confuse this with exponentiation. To determine whether x^n means "x to the power n" or "the concatenation of n copies of x" you need to know whether x is being treated as a number or a string.

1.7 Comprehension Questions

1. What is the cardinality of the set $\{\{3, 7, 11\}, \{A, B\}\}$?
2. What is $|\{x \in \mathbb{Z} \mid x^2 < 10\}|$? (Note: don't confuse this with the set $\{x \in \mathbb{Z}_{\geq 0} \mid x^2 < 10\}$!)
3. What is $\{1, 3, 5\} \setminus \{1, 2, 3\}$?
4. What is $\{1, 2\} \times \{3, 4\}$?
5. What is $2^{\{1,2\}}$?
6. Let x and n be positive integers greater than 1 such that $(x^2x^3)^5 = x^n$. What is the value of n?
7. What is the value of $\log_2(8^{32})$?
8. What is the value of $\sum_{i=1}^{100} i$?
9. What is the value of $\prod_{i=1}^{5} i$?
10. What is the value of $\lfloor -2.1 \rfloor \cdot \lceil 3 \rceil$?
11. Which of these strings are in $\{0, 1\}^3$?
    - ε
    - 01
    - 101
    - 012
    - 1000
12. Let $\Sigma = \{a, b, c, d\}$, and consider the two strings ab and cd. What is $ab \cdot cd$?
13. Let $\Sigma = \{0, 1\}$, and consider the two strings 10 and 100. What is $10 \cdot 100$?
14. Compute $|\{s \in \{0, 1, 4\}^3 : s_1 = s_2\}|$.

1.8 Exercises

1. Let n be a positive integer greater than 1 and let x be a positive real number such that $\left(\frac{3}{2}\right)^{\log_2 n} n = n^x$. What is the value of x?
2. Let a, b, c, x, y be integers greater than 2 and let α and β be real numbers such that

   $$\log_b(x^a y^c) = \alpha \log_c(x) + \beta \log_b(y).$$

   What are the values of α and β? (Express your answer in terms of numeric constants and/or a, b, and c).
3. Let n be an integer greater than 99. Simplify

   $$\sum_{i=99}^{2n} (i - 3)$$

   (i.e. give an expression which is equivalent to this one but doesn't use a summation).

2 Logic and Proofs

This chapter will review mathematical logic and basic techniques for formally proving mathematical claims. These proof techniques will be essential for proving the correctness and runtime of the algorithms we will develop and analyze in this course. Hopefully much of this content is familiar to you from CSCE 222, but it's important to establish a common foundation to ensure that everyone is set up for success when we begin applying these proof techniques to new topics.

2.1 Predicate Logic

A **predicate** is some statement which is true or false. For example $10 < 25$, $19^2 = 400$, and "The string 'strawberry' contains 2 r's" are all predicates (the first one is true, the other two are false). On the other hand an expression like $2 + 2$ is not a predicate, since it does not express a claim and therefore does not have a truth value. Similarly, $x > -2$ is not a predicate, because x is not defined, meaning that the statement does not have a well-defined truth value.

If we want to include variables in our predicate, we need to **quantify** them, meaning we need to describe what values the variable can take on. There are two main logical quantifiers.

- The universal quantifier "for all" is used to say that some expression is true for every element of a given set. We could convert our expression $x > -2$ into the (false) predicate "For all integers x, $x > -2$" by adding a universal quantifier. Note that the choice of which set to quantify over is very important. We could convert the same expression into the true predicate "For all positive integers x, $x > -2$" by choosing to quantify over $\mathbb{Z}^+$ instead of $\mathbb{Z}$.

  We represent the universal quantifier symbolically using the symbol $\forall$. So the predicate we just described could be written as $\forall\, x \in \mathbb{Z}^+, x > -2$.

  Note that any "for all" statement is true if the set that we are quantifying over is the empty set. For example

  $$\forall\, x \in \emptyset,\ 2 + 2 = 5$$

  is a true predicate! We say that a predicate universally quantifying over the empty set is **vacuously true**.

- The existential quantifier "there exists" is used to say that some expression is true for at least one element of a given set. We could convert our expression $x > -2$ into the (true) predicate "There exists an integer x such that $x > -2$" by adding an existential quantifier. Once again, the choice of set to quantify over is important. We could make a false predicate based on the same expression if we chose to quantify over the set $\{-3, -4, -5\}$.

  We represent the existential quantifier symbolically using the symbol $\exists$. For example: $\exists\, x \in \mathbb{Z}, x > -2$.

  Just as every "for all" statement about the empty set is true, every "there exists" statement about the empty set is false. For example

  $$\exists\, x \in \emptyset,\ 2 + 2 = 4$$

  is a false predicate. There is no x in the empty set for which $2 + 2 = 4$, because there is no x in the empty set at all!

We can modify and combine predicates with the logical operators "and" (denoted symbolically as $\wedge$), "or" (denoted symbolically as $\vee$), and "not" (denoted symbolically as $\neg$). So if A and B are two predicates then:

- $A \wedge B$ is true if A and B are both true and false otherwise.
- $A \vee B$ is true if either A or B is true, and false if A and B are both false. Note that in mathematics/computer science "or" almost always means "inclusive or", so $A \vee B$ is true if A and B are both true.

- $\neg A$ is true if A is false, and false if A is true.

The logical operators in the previous paragraph all correspond to common operators in many programming languages. There is another important logical operator which is not common in programming: implication. "A implies B" is a predicate which is true if A and B are both true, or if A is false. This predicate can also be written as "If A, then B" or denoted symbolically as $A \Rightarrow B$. Many of the predicates we are interested in proving are implications.

Implications can be confusing, because the way they are used in mathematics and computer science is different from the way that "if" is used in regular English. It's important to remember that $A \Rightarrow B$ is always true when A is false. For example "If I am on Mars, then $2 + 2 = 5$" is a true predicate. Since I am not on Mars, the truth value of $2 + 2 = 5$ does not affect the truth value of the whole expression (If you are reading this from a Martian colony in the distant future, replace Mars with a planet you are not on).

Closely related to implication is **bi-implication**, also known as "if and only if". "A bi-implies B" or "A if and only if B" or "A iff B" all mean that A and B are either both true or both false. Bi-implication is written symbolically as $A \Leftrightarrow B$. $A \Leftrightarrow B$ is logically equivalent to $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

2.2 What is a Proof?

A **proof** is a rigorous argument designed to convince other people that a given claim is true. A correct proof could be written fully in predicate logic, relying only on basic axioms of set theory. Such proofs are very tedious to write and to read, however, so typical proofs are written mostly in English. This means that what constitutes a "correct" proof is to some degree a social convention. A correct proof will never claim a false statement, but a proof which makes only true statements can still be considered incorrect if it skips too many steps and it's not clear how the lines of the proof connect to each other.

For example, let's say that I want to prove the following uninteresting theorem:

Theorem 4. $((19 \cdot 15) + 7)^2 - 9 > 10000$

Proof. We first calculate that

$$19 \cdot 15 = 285.$$

Therefore

$$(19 \cdot 15) + 7 = 285 + 7 = 292.$$

Thus,

$$((19 \cdot 15) + 7)^2 = 292^2 = 292 \cdot 292 = 85264.$$

Finally,

$$((19 \cdot 15) + 7)^2 - 9 = 85264 - 9 = 85255 > 10000.$$

□

Alternatively, we could give the following shorter proof of the same theorem:

Proof. We note that

$$((19 \cdot 15) + 7)^2 - 9 = 85255 > 10000.$$

□

For the purposes of this course, the second proof would be entirely sufficient. You may assume that the reader of your proof is capable of performing basic arithmetic and do not need to show every step of the calculation. On the other hand, you should not assume that your reader is comfortable manipulating complicated summations, and if you need to do this, you should carefully show the steps.

How much detail your proof should go into depends on the audience that you are writing for. A proof written for CS PhD's may skip many steps as "obvious" that should be fully justified if the proof is targeted at CS undergraduates. You should view the audience of the proofs you write in

this class as a fellow student in this course who is not quite as clever as you. You may assume basic results from prerequisite courses like Calculus and CSCE 222 without proving them, and you may use any theorem proven in these notes. The original parts of your proof, however, should be carefully justified. You want to make an airtight case which would convince a reader who is skeptical of the claim you are proving.

One good test for deciding whether you need to justify something in your proof is to think about what you would say if someone asked you why a particular line of the proof follows from the previous one. If you can think of a short obvious answer, the connection is probably clear enough. If you realize that *you* couldn't explain the connection, then that's a sign that your proof is missing something there.

When in doubt, err on the side of providing more detail. We will not penalize you for providing an overly rigorous proof, but we will penalize you for leaving out important details. You are also welcome to ask in office hours if you have justified a particular part of your proof sufficiently.

2.3 Proving Predicates

In the previous section we saw an example of how to prove the simplest kind of predicate, a formula with no variables or quantifiers. Most predicates we are interested in proving, however, will involve a universal or existential quantifier (or both).

To prove a universal statement of the form $\forall\, x \in S,\, P(x)$, we need to choose an *arbitrary* $x \in S$ and show that $P(x)$ holds. Here "arbitrary" means that our proof should work for any choice of x from S, we can't assume that the x we chose has any special property. Sometimes it's helpful to imagine here that x is being chosen for you by an enemy who wants your proof to fail.

For example, suppose that we want to prove the following theorem:

Theorem 5. *Let* $S := \{n^2 \mid n \in \mathbb{Z}^+\}$. *Then* $\forall\, x \in S,\, x > 1 \Rightarrow x^2 > x + 10$

Proof. Let $x \in S$ be arbitrary. If $x \leq 1$ the implication is automatically true, so assume $x > 1$. Since $x \in S$, x is the square of a positive integer. Since the next smallest square number after 1 is 4, $x > 1 \Rightarrow x \geq 4$. Therefore:

$$x^2 \geq 4x = x + 3x \geq x + 12 > x + 10.$$

□

We can view this kind of proof as providing a template for creating infinitely many proofs, one for each element of S. We can pick any element of S to be x, plug it into this proof, and the proof will show us why $x > 1 \Rightarrow x^2 > x + 10$ for that value of x. Note that in this proof we explicitly considered the case where our arbitrary x was less than or equal to 1. While it is important to understand that we are proving a claim about all x, including values of x which are less than or equal to 1, this step is typically omitted in proofs of "for all" statements including implications, because it would be the same every time (every implication is true whenever its left side is false). Therefore we can shorten the proof slightly by starting "Let $x \in S$ be an arbitrary integer with $x > 1$".

Proving a "there exists" statement is usually easier. To prove $\exists\, x \in S,\, P(x)$, we simply need to find a single specific $x \in S$ for which $P(x)$ is true. For example:

Theorem 6. *Let* $S := \{n^2 \mid n \in \mathbb{Z}^+\}$. *Then* $\exists\, x \in S,\, x^2 < \log_2(x) + 10$

Proof. Choose $x = 1$. $x \in S$, since $1 \in \mathbb{Z}^+$ and $1^2 = 1$. For this choice of x, we have:

$$x^2 = 1^2 = 1 < 10 = 10 + 0 = 10 + \log_2(1).$$

□

Sometimes we will want to prove statements involving *nested quantifiers*, i.e. statements that have both "for all" and "there exists" statements. When parsing these statements it is important to pay attention to the order of the quantifiers. To prove a statement of the form $\forall\, x \in S,\, \exists\, y \in T,\, P(x, y)$, we need to show that for any arbitrary x in S, we can pick some y in T such that $P(x, y)$ is true (the choice of y will probably depend on the value of x).

For example,

Theorem 7. $\forall x \in \mathbb{Z}, \exists y \in \mathbb{Z}^+, y > x$.

Proof. Let $x \in \mathbb{Z}$ be arbitrary. If $x < 0$, choose $y = 1$, and then $y \in \mathbb{Z}^+$ and $y > x$. Otherwise, $x \geq 0$, so choose $y = x + 1$. Since $x \geq 0$, $y \in \mathbb{Z}^+$, and $y = x + 1 > x$. □

Notice that we had to pay attention to the sets that x and y were quantified over here. It would be easy to simply say "choose $y = x + 1$" without considering a special case for negative x, but since the theorem says that y must be positive, that proof would be incorrect.

If we switch the order of the quantifiers in Theorem 7, we get a different predicate

$$\exists y \in \mathbb{Z}^+, \forall x \in \mathbb{Z}, y > x.$$

This predicate claims that there exists a single y, such that for every integer x, y is larger than that x. Since the existential quantifier comes first now, we have to choose y before x, meaning that our choice of y cannot depend on x like it did in the proof of Theorem 7. There is no integer which is larger than all other integers, so this predicate is false.

2.4 Disproving Predicates

Sometimes we want to disprove a predicate (i.e. to show that it is false). This is equivalent to proving the negation of the predicate. Therefore, it's important to understand how to negate universal and existential quantifiers.

The negation of the statement $\forall x \in S, P(x)$ is $\exists x \in S, \neg P(x)$. This means that to disprove a "for all" statement, it's enough to find a single counterexample. Finding one $x \in S$ for which $P(x)$ is false, proves that $P(x)$ can't be true for every single $x \in S$.

The negation of the statement $\exists x \in S, P(x)$ is $\forall x \in S, \neg P(x)$. That is, to show that a "there exists" statement is false, we need to prove that $P(x)$ is false for every single x in S. We can prove this using any of our usual techniques for proving a "for all" statement, like the "choose an arbitrary element and show the claim holds for it" method that we saw in the previous section or induction which we will talk about in Section 3.1.

2.5 Using Cases

Often in a proof we need to divide a claim up into several cases and prove each of them separately. For example, to prove a statement of the form $A \vee B \Rightarrow C$, we may have one case where we assume A is true, and then prove C, and a second case where we assume B is true, and then prove C. Note that these cases may overlap (it's possible that A and B are both true), which is not a problem. The important thing is that our cases are comprehensive. Any situation where $A \vee B$ is true will fall into at least one of our cases.

Cases can also be helpful when proving a "for all" statement. As we've seen, the standard approach to proving a statement of the form $\forall x \in S, P(x)$ is to take an arbitrary $x \in S$ and show that $P(x)$ holds. Sometimes we need a different proof structure depending on some property of the arbitrary x we chose (for example, we might use a different proof for odd x than for even x). This is another natural place to use case work. Again it's important to justify that every $x \in S$ will fall into at least one of our cases. Here is an example of using casework to prove a "for all" statement.

Theorem 8. *Let $S := \{n^2 \mid n \in \mathbb{Z}_{\geq 0}\}$. Then $\forall x \in S$, the last digit of x is either 0, 1, 4, 5, 6, or 9.*

Proof. Let $x \in S$ be arbitrary. By the definition of S, x is a perfect square, so $\sqrt{x} \in \mathbb{Z}_{\geq 0}$. Let $n := \sqrt{x}$. We note that the last digit of n^2 depends only on the last digit of n (this follows from the grade school algorithm for multiplication). We case on the last digit of n.

- If the last digit of n is 0, then the last digit of n^2 is 0
- If the last digit of n is 1, then the last digit of n^2 is 1
- If the last digit of n is 2, then the last digit of n^2 is 4

- If the last digit of n is 3, then the last digit of n^2 is 9
- If the last digit of n is 4, then the last digit of n^2 is 6
- If the last digit of n is 5, then the last digit of n^2 is 5
- If the last digit of n is 6, then the last digit of n^2 is 6
- If the last digit of n is 7, then the last digit of n^2 is 9
- If the last digit of n is 8, then the last digit of n^2 is 4
- If the last digit of n is 9, then the last digit of n^2 is 1

These are all the possible values for the last digit of n. Thus, in every case, the last digit of n^2 (which by definition equals x) is either 0, 1, 4, 5, 6, or 9. □

2.6 Using the Contrapositive

Remember that an implication is a statement of the form "If A then B", written $A \Rightarrow B$. The standard way to prove an implication is to start by assuming A is true and then use this to show that B is true (perhaps by applying definitions or performing algebraic manipulation).

Sometimes it's easier to prove an implication if we first convert it into a different equivalent form. One of the most common ways to do this is by taking the **contrapositive**. The contrapositive of $A \Rightarrow B$ is $\neg B \Rightarrow \neg A$. Just like the original implication the contrapositive is false only when A is true and B is false, so the two forms are logically equivalent, and we can choose to prove whichever is more convenient. Here is an example of a proof by contrapositive:

Theorem 9. $\forall\, x, y \in \mathbb{Z}_{\geq 0},\ x + y > 21 \Rightarrow x > 10 \vee y > 11.$

Proof. Let x and y be arbitrary non-negative integers. We want to prove that $x + y > 21 \Rightarrow x > 10 \vee y > 11$. We will prove the contrapositive:

$$\neg(x > 10 \vee y > 11) \Rightarrow \neg(x + y > 21).$$

By De Morgan's Laws, this is equivalent to

$$x \leq 10 \wedge y \leq 11 \Rightarrow x + y \leq 21.$$

If $x \leq 10$ and $y \leq 11$, then $x + y \leq 10 + 11 = 21$, which proves the theorem. □

Here it was not quite clear how to directly prove the implication starting from the fact $x + y > 21$ and trying to prove an "or" statement, but the proof of the contrapositive is very short and simple. Note that taking the contrapositive only affected the implication, not the quantifiers, we were still proving a "for all" statement about non-negative integers.

2.7 Proving Subset Relationships and Set Equality

To prove that one set S is a subset of another set T we need to prove the predicate

$$\forall\, x \in S,\ x \in T.$$

Just like proving any other "for all" statement, this will typically involve taking an arbitrary element of S, and then showing that it is in T (usually by expanding the definitions of S and T and showing how they relate to each other).

Proving that two sets are equal requires a little more effort. The most common approach is a proof by **double containment**, which means showing $S \subseteq T$ and $T \subseteq S$ (if both of these are true, then we must have $S = T$). Here is an example of a set equality proof (which includes two subset proofs):

Theorem 10. *Let $S := \{25x + 15y \mid x, y \in \mathbb{Z}\}$, and let $T := \{5n \mid n \in \mathbb{Z}\}$. Prove that $S = T$.*

Proof. We prove this by double containment:

- First we show $S \subseteq T$. Let $a \in S$ be arbitrary. By the definition of S, a can be expressed as $25x + 15y$ for some integers x and y. We need to show that there exists an integer n such that $a = 5n$. Let $n := 5x + 3y$. Then we have that

  $$5n = 5(5x + 3y) = 25x + 15y = a.$$

  So by the definition of T, $a \in T$.

- Now we show $T \subseteq S$. Let $a \in T$ be arbitrary. By the definition of T, $a = 5n$ for some integer n. We need to show that there exist integers x and y such that $25x + 15y = a$. We note that $25(-1) + 15(2) = 5$. Let $x := -n$ and let $y := 2n$ (clearly these are both integers). Then we have that

  $$25x + 15y = 25(-n) + 15(2n) = -25n + 30n = 5n = a.$$

  Thus, by the definition of S, $a \in S$.

□

If you are asked to prove set equality it is important to remember to prove both directions.

2.8 Acknowledgments

Some of the topics and problems in this chapter are inspired by ones in Building Blocks for Theoretical Computer Science by Margaret Fleck, the textbook that I used to teach Discrete Structures at UIUC.

2.9 Comprehension Questions

1. Determine which of the following predicates are true and which are false:

   (a) $3 > 4 \vee 3 > 2$

   (b) $3 > 4 \wedge 3 > 2$

   (c) $3 > 4 \Rightarrow 3 > 2$

   (d) $3 > 2 \Rightarrow 3 > 4$

   (e) $\forall x \in \mathbb{Z}, x \neq 4.11$

   (f) $\exists x \in \mathbb{Z}, x \neq 4$

   (g) $\forall x \in \{-3, 3\}, x^2 = 9$

   (h) $\exists x \in \mathbb{Z}^+, x \leq 0$

   (i) $\forall x \in \emptyset, x^2 = -1$

   (j) $\exists x \in \emptyset, x = x$

   (k) $\forall x \in \mathbb{Z}, x > 4 \Rightarrow x^2 > x$

   (l) $\forall x \in \mathbb{Z}, x > 0 \Rightarrow 2 + 2 = 4$

   (m) $\forall x \in \mathbb{Z}^+, 2 + 2 = 4 \Rightarrow x > 0$

   (n) $\exists x \in \mathbb{Z}^+, 2 + 2 = 4 \Rightarrow x = 7$

   (o) $\exists x \in \mathbb{Z}^+, x > 7 \Rightarrow x = -3$

   (p) $\exists x \in \mathbb{Z}^+, x > 0 \Rightarrow x = 5$

   (q) $\exists x \in \mathbb{Z}^+, x > 0 \Rightarrow x = -3$

   (r) $\forall x \in \mathbb{Z}^+, x > 7 \Rightarrow x \neq 1$

2. Find the negation of each of the following predicates

   (a) $\forall\, x \in \mathbb{Z},\, \exists\, y \in \mathbb{Z}^+,\, x + y = 7$

   (b) $\exists\, x \in \mathbb{Z},\, \forall\, y \in \mathbb{Z}^+,\, x + y = 7$

   (c) $\exists\, x, y \in \mathbb{Z}^+, x + y = 7$

   (d) $\forall\, x \in \mathbb{Z}_{\geq 0}, x > 5 \Rightarrow x \neq 3$

3. Find the contrapositive of each of the following implications:

   (a) $x > 5 \Rightarrow x^2 > 10$

   (b) x^2 is odd $\Rightarrow x$ is odd

   (c) $x \notin A \cap B \Rightarrow x \notin A$

4. True or False? For all sets A and B, $A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A)$.

5. Fill in the blank: When we convert the implication $A \Rightarrow B$ to the equivalent implication $\neg B \Rightarrow \neg A$, we have taken the _____.

6. Suppose that we want to prove "$\forall\, x \in \mathbb{R}^+, x^2 > 0$." What is the main flaw with the following "proof"?

   Proof. Let $x \in \mathbb{R}^+$ be arbitrary. Then we have that

   $$x^2 = x \cdot x \geq x \geq 0.$$

   □

7. Suppose that we want to prove that $\{2n \mid n \in \mathbb{Z}_{\geq 0}\} = \{n \in \mathbb{Z}_{\geq 0} \mid$ the last digit of n is either $0, 2, 4, 6$ or $8\}$." What is the main flaw with the following "proof"?

   Proof. Let $S := \{2n \mid n \in \mathbb{Z}_{\geq 0}\}$ and let $T := \{n \in \mathbb{Z}_{\geq 0} \mid$ the last digit of n is either $0, 2, 4, 6$ or $8\}$. Let $x \in S$ be arbitrary. Then $x = 2n$ for some $n \in \mathbb{Z}_{\geq 0}$. We case on the last digit of n:

   - If the last digit of n is 0 or 5, the last digit of $x = 2n$ is 0, so $x \in T$.
   - If the last digit of n is 1 or 6, the last digit of $x = 2n$ is 2, so $x \in T$.
   - If the last digit of n is 2 or 7, the last digit of $x = 2n$ is 4, so $x \in T$.
   - If the last digit of n is 3 or 8, the last digit of $x = 2n$ is 6, so $x \in T$.
   - If the last digit of n is 4 or 9, the last digit of $x = 2n$ is 8, so $x \in T$.

   Since n must end with one of these digits, we conclude that in every case $x \in T$ and thus $S = T$. □

2.10 Exercises

1. Prove that $\forall\, a \in \mathbb{Z}, \exists\, b, c \in \mathbb{Z}, a \neq b \wedge a \neq c \wedge b + c = a$.

2. Prove that $\forall\, a, b \in \mathbb{Z},\ a < b \Rightarrow \exists\, q \in \mathbb{Q},\ a < q < b$.

3. Prove that $\forall\, x \in \mathbb{Q}^+, \forall\, y \in \mathbb{R}, \frac{y+1}{2} \in \mathbb{Q} \Rightarrow \frac{1}{x} + y \in \mathbb{Q}$

4. Prove that if $r \in \mathbb{R}^+ \setminus \mathbb{Q}^+$ and $q \in \mathbb{Q}^+$, then $rq \in \mathbb{R} \setminus \mathbb{Q}$ (i.e. prove that multiplying a positive irrational number with a positive rational number gives an irrational number).

5. Define a new operation $\ominus$ by $a \ominus b := 2(a + b)$. **Disprove** the following claim:
   Claim: $\forall\, x, y, z \in \mathbb{R}, x \ominus (y \ominus z) = (x \ominus y) \ominus z$

6. Let A, B, and C be sets. Prove that

$$(A \setminus B) \cup (B \setminus C) \subseteq (A \cup B) \setminus (A \cap B \cap C)$$

7. Prove that for every positive integer n, if n^2 is even, then n is even.

8. Let Σ be an alphabet. Prove that

$$\{x \cdot y \mid x, y \in \Sigma^* \text{ with } |x| = |y|\} = \{s \in \Sigma^* \mid |s| \text{ is even }\}$$

3 Advanced Proof Techniques

This chapter will discuss some more advanced proof techniques, building on what we discussed in the previous section. We'll start in Section 3.1 by reviewing induction, a powerful technique you've probably seen before. In Section 3.2, we introduce correctness proofs for algorithms, the main kind of proof we'll be using in this course, and in Section 3.3, we'll introduce loop invariants, a way to prove the correctness of certain programs using induction. Finally in Section 3.4, we'll discuss proof by contradiction, a powerful technique for proving that particular objects (or algorithms) can't exist, which we'll rely on later in the course, particularly in Chapter 7.

3.1 Induction

One of the most powerful proof techniques is induction. Induction is a technique for proving a "for all" statement quantified over non-negative integers. It is based on the important axiom that every non-empty subset of $\mathbb{Z}_{\geq 0}$ has a minimum element.

An induction proof proves a statement of the form $\forall\, n \in \mathbb{Z}_{\geq 0}$, $P(n)$ by proving

$$\forall\, n \in \mathbb{Z}_{\geq 0}\, (\forall\, k \in \mathbb{Z}_{\geq 0} \text{ with } k < n,\, P(k)) \Rightarrow P(n)$$

Here we have replaced the original predicate $P(n)$ with the new predicate "If $P(k)$ for all $0 \leq k < n$, then $P(n)$". Since we have given ourselves an additional assumption to work with, this new predicate is often easier to prove. This form of induction is often known as "strong" induction, in contrast to "weak" induction where we only assume $P(n-1)$ instead of assuming $P(k)$ for all $0 \leq k < n$. There is no particular benefit to using weak induction, however, so I encourage you to always use the form presented above.

It may not be clear however why proving the inductive version of the statement implies the original statement. To see why induction is valid, it is helpful to think about it as giving us a template to create a proof of $P(n)$ for whatever value of n we want. For $n = 0$, the assumption "$P(k)$ holds for all $0 \leq k < n$" is vacuously true (there are no possible choices of k which are ≥ 0 but also < 0), so our assumption has given us nothing and our induction proof must directly prove $P(0)$ (we often call this a base case). For $n = 1$, our induction proof shows that $P(0) \Rightarrow P(1)$. Since we already have a direct proof that $P(0)$ is true, we can combine these to get a proof that $P(1)$ is true (we know that $P(0)$ is true, and we know "if $P(0)$ is true then $P(1)$ is true", so we conclude $P(1)$ is true). Similarly, our induction proof shows that $P(0) \wedge P(1) \Rightarrow P(2)$. Since we have now shown $P(0)$ and $P(1)$, we can combine these proofs together with the proof that $P(0) \wedge P(1) \Rightarrow P(2)$ to get a proof that $P(2)$ is true. Continuing the pattern, we can use our induction template to construct a direct proof of $P(n)$ for any given n by first constructing direct proofs for $P(0), \ldots, P(n-1)$.

Don't despair even if the correctness of induction feels suspect, however. It's possible to construct correct induction proofs even if the fact that induction works still feels like magic.

Let's see an example of the kind of claim that we can prove with induction.

Theorem 11. *$\forall\, n \in \mathbb{Z}_{\geq 0}$, $n^3 - n$ is divisible by 3*

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then $n^3 - n = 0^3 - 0 = 0$, which is divisible by 3.
- Induction Hypothesis: Suppose that $\forall\, k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $k^3 - k$ is divisible by 3.
- Induction Step: We will show that $n^3 - n$ is divisible by 3. Since $n > 0$, $n - 1 \in \mathbb{Z}_{\geq 0}$, so by our induction hypothesis we have that $(n-1)^3 - (n-1)$ is divisible by 3. We note that

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

  Therefore $n^3 - n - 3(n^2 - n)$ is divisible by 3. Since n is an integer, $3(n^2 - n)$ is divisible by 3 as well, so we conclude that $n^3 - n$ is divisible by 3.

□

This proof illustrates the main pieces of an induction proof.

- First we explicitly state what variable we are inducting on. Sometimes this will be obvious, but other times the choice of induction variable will be trickier (we may need to induct on the sum or difference of two values for example).
- Next we have the base case, where we show the claim for 0 (or whatever the smallest value the claim applies to is) directly
- Then we state our induction hypothesis, which is just assuming that the claim holds for all values of k less than n (We don't literally have to call our variables n and k, but it's important that we call the variable we're assuming the claim holds for something different from the variable we are inducting on. If we're inducting on n and assume that the claim holds for n, our induction hypothesis is already assuming what we're trying to prove in the induction step!)
  - Sometimes it may be convenient to instead assume the claim holds for values up to and including n, then prove it for $n+1$. This is also fine, as long as you use the same form in the induction hypothesis and induction step).
- Finally, we have the induction step, which is usually the meat of the proof. This is where we prove that our claim holds for n using the induction hypothesis. (If your induction step does not use your induction hypothesis either you did something wrong or you didn't need to be using induction in the first place)

At its core, induction is a technique for proving universal statements about non-negative integers. We can apply induction to many other kinds of problems however, by converting them into universal statements about non-negative integers. The key is usually choosing an appropriate non-negative integer variable to induct on. Here is an example of using induction to prove a theorem about strings:

Theorem 12. *Let Σ be an alphabet and let $x, y \in \Sigma^*$ be non-empty strings such that $xy = yx$. Then there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $x = z^a$ and $y = z^b$.*

Proof. We prove this by induction on $|xy|$.

- Base case: $|xy| = 2$ (this is the smallest possible length for xy since x and y are nonempty). In this case, x and y must both have length 1, so since $xy = yx$, we have that $x = y$. Therefore we can choose $z = x$ and $a = b = 1$, and we have that $z^a = x$ and $z^b = y$.
- Induction Hypothesis: Assume that for all $u, v \in \Sigma^*$ with $|uv| < |xy|$, $uv = vu \Rightarrow \exists\, z \in \Sigma^*, \exists\, a, b \in \mathbb{Z}^+,\ u = z^a \wedge v = z^b$.
- Induction Step: We consider two cases:

  Case 1: Suppose $|x| = |y|$. Since $xy = yx$, we know that x is the same as the first $|x|$ many characters of y, and since $|y| = |x|$, this is all of y, so $x = y$. Therefore we can choose $z = x$ and $a, b = 1$.

  Case 2: Otherwise, $|x| \neq |y|$. Assume without loss of generality that $|x| > |y|$. Since $xy = yx$, the first $|y|$ characters of x are a copy of y, so we can write x as yw, for some non-empty string $w \in \Sigma^*$. Substituting this into our equality $xy = yx$, we get that $ywy = yyw$. Peeling off y from the front of both strings, this implies that $wy = yw$. Furthermore, since $|y| > 0$, $|wy| < |xy|$. Therefore, by our induction hypothesis, there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $w = z^a$ and $y = z^b$. For the same z, we have that $x = yw = z^b z^a = z^{a+b}$, so x and y can both be created by concatenating copies of z, which is what we wanted to show.

□

Note that this induction proof would actually be complete without the base case. The case where $|xy| = 2$ is already covered in the inductive step by our special case for when $|x| = |y|$. There's no real harm in covering this case twice, though, so it's fine to include a base case even when it's technically not needed.

This proof also used the useful phrase "without loss of generality". This phrase (sometimes abbreviated WLOG) is essentially a shortcut for doing a proof by cases where the proofs for the different cases are exactly the same except with the names of the variables swapped. In this case, rather than giving separate proofs for the case where $|x| > |y|$ and the case where $|y| > |x|$, we arbitrarily choose to assume $|x| > |y|$, and use the phrase "without loss of generality" to let the reader know that they could obtain the proof for the other case by simply swapping x and y in the rest of the proof.

Back in Section 1.4 we saw a simple proof that $\sum_{i=1}^{n} i = n(n+1)/2$. This is probably the most useful summation formula to know for analyzing algorithms, but the second most useful is the one we'll prove now, using induction.

Theorem 13. *For all $n \in \mathbb{Z}_{\geq 0}$ and all $x \in \mathbb{R} \setminus \{1\}$,*

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

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then since $x \neq 1$ (and thus $x - 1 \neq 0$) we have

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

- Induction Hypothesis: Assume that for all $k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $\sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}$
- Induction Step: We have that

$$\begin{aligned}
\sum_{i=0}^{n} x^i &= x^n + \sum_{i=0}^{n-1} x^i && \text{(Since } n > 0\text{)} \\
&= x^n + \frac{x^n - 1}{x - 1} && \text{(by our IH)} \\
&= \frac{x^{n+1} - x^n}{x - 1} + \frac{x^n - 1}{x - 1} \\
&= \frac{x^{n+1} - 1}{x - 1}
\end{aligned}$$

□

Note that we split the summation from 0 to n into the last term plus a summation from 0 to $n - 1$ so that we could apply the induction hypothesis. This is a standard technique that we can use to apply induction to many summations. Finding the right formula for a summation can be tricky, but if we have a guess at the right answer (e.g. based on finding a pattern in the value of the summation for small values of n) this technique will usually let us prove that the guess is correct.

3.2 Proving the Correctness of Algorithms

The most common proofs you will need to write in this course are proofs of correctness for algorithms. A proof of correctness for an algorithm can be complicated and may need to be broken down into several pieces, but at the highest level it's just a proof of a "for all" statement, similar to the ones we've seen in sections 2.3, 2.5 and 3.1. In particular, proving an algorithm means proving that for all

inputs the algorithm produces the correct output. For example, if SORT is an algorithm designed to sort an array of integers (in order of increasing value), then to prove the correctness of SORT we need to prove that:

"For every array A of integers, after calling SORT(A), A is sorted."

Sometimes correctness proofs are simple enough that we don't need any special techniques. Consider the following algorithm which finds the square root of any number whose square root is an integer:

```
IntegerSquareRoot(x):
  i ← 0
  While (i + 1)^2 ≤ x:
      i ← i + 1
  Return i
```

Theorem 14. *For every non-negative integer x such that $\sqrt{x}$ is an integer,* INTEGERSQUAREROOT(x) *returns $\sqrt{x}$.*

Proof. Let $x \in \mathbb{Z}_{\geq 0}$ such that $x = k^2$ for some $k \in \mathbb{Z}_{\geq 0}$. For every $i < k$, $(i+1)^2 \leq k^2 = x$, while if $i = k$ then $(i+1)^2 > k^2$. Thus, the while loop will continue until $i = k$. Thus, when the algorithm returns i it is returning $k = \sqrt{x}$. □

Note that if we want to prove an algorithm correct it is important to be clear about what inputs the algorithm is expected to handle. INTEGERSQUAREROOT would not be an algorithm that correctly returns $\sqrt{x}$ if the input could be any integer (e.g. the algorithm fails to compute $\sqrt{2}$, because $\sqrt{2}$ is not an integer). While data validation is important in real-world programming, the correct way to handle invalid inputs tends to be application specific. In this course, if you are asked to design an algorithm that takes a specific kind of input (e.g. a square number), your algorithm may do whatever it likes when the input is not of the type given in the problem, and you do not need to consider these kinds of inputs when proving the correctness of your algorithm.

For recursive algorithms, induction is often a natural approach to proving correctness (we'll look at this more when we talk about divide and conquer algorithms in Chapter 8). For example, consider the following algorithm:

```
Fact(n):
  If n ≤ 1:
      Return 1
  Return n * Fact(n − 1)
```

Theorem 15. *For every positive integer n,* FACT(n) *returns $n!$ (recall from Section 1.5 that $n!$ is n factorial)*

Proof. We prove this by induction on n:

- Base Case: If $n = 1$ then FACT immediately returns 1, which is 1!
- Induction Hypothesis: Suppose that for every $1 \leq k < n$, FACT(k) returns $k!$.
- Induction Step: Since $n > 1$, FACT(n) returns $n \cdot$ FACT($n-1$). By our induction hypothesis, FACT($n-1$) returns $(n-1)!$. Thus, FACT(n) returns $n(n-1)! = n!$.

□

3.3 Loop Invariants

The algorithms we saw in the previous section were so simple that they arguably don't even need a full proof of correctness (While it's always nice to rigorously prove things, in an exam where functions like INTEGERSQUAREROOT or FACT might be a subroutine in a larger program it would be sufficient to say that their correctness follows from the definition of square root or factorial). Now let's look at an algorithm whose correctness proof is a bit less trivial.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 16. *For every array A of integers, after calling* InsertionSort*(A), A is sorted in increasing order.*

Before proving Theorem 16, let's think a bit about what the right approach is. We need to prove that InsertionSort(A) sorts A. Induction is the right idea here, but we have to be careful about what we induct on and what claim we prove inductively. Unlike in our correctness proof for Fact we can't induct on the input directly because the input to InsertionSort is an array, not an integer. We could try inducting on the length of A, but since InsertionSort doesn't recursively call itself, it's not clear how to use an induction hypothesis about the behavior of InsertionSort on arrays of length $< n$ to say something about the behavior of InsertionSort on arrays of length n.

Instead, the correct approach is to induct on the number of iterations of the outer for loop. Our induction hypothesis can't be that the array is sorted (clearly it need not be sorted after just one iteration of the loop), but that each iteration makes increasing progress towards sorting the array, in such a way that by the end of the n'th iteration, the array is sorted. A claim of the form: "After i iterations of this loop, $P(i)$ is true" (where $P(i)$ is some predicate depending on i) is called a *loop invariant*. Defining a suitable loop invariant and then proving it (usually via induction) is a common technique for proving the correctness of algorithms.

Let's prove the correctness of insertion sort using a loop invariant:

Proof. (of Theorem 16) Let A be an array of integers.

We will prove the following loop invariant for every $k \in \{1, \ldots, n\}$: After k iterations of the outer for loop, $A[1:k]$ is sorted. Note that this invariant implies that after n iterations of the for loop (i.e. at the end of the algorithm), $A[1:n]$ is sorted, meaning the entire array has been sorted. Thus, proving this invariant is sufficient to prove the theorem.

We will prove the invariant by induction on k.

- Base Case: Suppose $k = 1$. The first iteration of the outer loop does nothing (because the inner loop runs 0 times). However, $A[1, \ldots, k] = A[1, \ldots, 1]$ consists of just a single element, so it is automatically sorted.

- Induction Hypothesis: Suppose that for every $1 \leq \ell < k$, we have that $A[1, \ldots, \ell]$ is sorted after ℓ iterations of the outer for loop.

- Induction Step: By our induction hypothesis, we know that at the start of the k'th iteration (i.e. after the $(k-1)$'th iteration) $A[1:k-1]$ is sorted. Let x be the value in $A[k]$ at the beginning of the k'th loop iteration. In the k'th iteration of the outer loop, the inner loop repeatedly swaps x with elements of the sorted subarray $A[1, \ldots, k-1]$, starting from the right, until either it reaches the beginning of the array, or it finds an element which is less than or equal to x.

  If we swap x all the way to the beginning of the array, then we know that x is smaller than every other element of $A[1:k]$ and thus belongs in $A[1]$, where it has ended up. Meanwhile all of the elements $A[1:k-1]$ have shifted right one position, but they remain in the same relative order. By our induction hypothesis they were already sorted, so therefore $A[1:k]$ is now sorted.

  On the other hand if we stop swapping x and break out of the loop at some $j > 1$, then we found some j such that $A[j] \leq x$, and our new array begins $A[1:j], x, A[j+1, k-1]$. Since we broke out of the inner for loop for this value of j, but not the previous one, we have that $A[j] \leq x < A[j+1]$. Furthermore, by our IH, $A[1:j]$ and $A[j+1:k-1]$ were already sorted,

so in the rearranged array we have $A[m] \leq A[m+1]$ for all m from 1 to $k-1$, and $A[1:k]$ is sorted.

□

3.4 Proof by Contradiction

The last type of proof it's important to be familiar with is proof by contradiction. In a proof by contradiction we start by assuming the opposite of what we're trying to prove and then show that this implies some statement which we know is false. Just like with induction, it's important to let the reader know when you're using this proof technique, so a proof by contradiction usually begins with "Assume for the sake of contradiction that [opposite of claim you're trying to prove]". "Assume for sake of contradiction" can be abbreviated as AFSOC.

A proof by contradiction is complete when you have shown that the thing you have assumed implies a false statement. Often this will be a contradiction to what you assumed at the start of the proof (e.g. you assumed a number was odd then proved it was even), but it can also be a regular logical impossibility, like deriving that $1 = 2$.

Proof by contradiction is commonly used when we want to show that a particular object (for example, an algorithm) cannot exist. One of the earliest examples of this kind of proof comes from the Greek mathematician Euclid, over 2000 years ago, who proved that there are infinitely many primes (Euclid's proof was not explicitly a proof by contradiction, but the core idea of it is the same as the proof I present here).

Theorem 17. *There is no largest prime number*

Proof. AFSOC that there is a largest prime number, call it p. Let n be the number of prime numbers less than or equal to p, and call these primes $p_1, \ldots, p_n$, with $p_n = p$. Let $m := \prod_{i=1}^{n} p_i$. By definition, m is divisible by every prime less than or equal to p. Therefore, $m+1$ is not divisible by any of these primes, meaning that either $m+1$ is prime, or it is divisible by some other prime bigger than p. In either case, this contradicts our assumption that p is the largest prime (because $m+1 > m \geq p$). Therefore, there is no largest prime number. □

Here's another classic example of proof by contradiction. Note that the claim "$\sqrt{2}$ is irrational" is actually a claim of the form "an object does not exist" in disguise. If we unpack the definition of a rational number, saying that "$\sqrt{2}$ is not a rational number" means saying that there do not exist two integers whose ratio is $\sqrt{2}$. Therefore we can prove this claim by contradiction if we assume that there are two such integers, and then show that this implies nonsense.

Theorem 18. *$\sqrt{2}$ is irrational.*

Proof. AFSOC that $\sqrt{2} \in \mathbb{Q}$. Then $\exists\, p, q \in \mathbb{Z}$ such that $\sqrt{2} = p/q$ and p and q are relatively prime (i.e. the fraction p/q is in simplest terms). We have that $2 = (p/q)^2 = p^2/q^2$, so $2q^2 = p^2$. Therefore p^2 is even, which means that p must also be even (since squaring an odd number gives an odd number). Thus, $p = 2r$ for some integer r. Since we already concluded that $2q^2 = p^2$, this means we now have $2q^2 = 4r^2$, and thus $q^2 = 2r^2$, so q^2 is also even, and therefore q is even as well. But this contradicts our assumption that p and q were relatively prime! (they share a factor of 2, so p/q can be simplified). We have reached a contradiction, so our initial assumption that $\sqrt{2} \in \mathbb{Q}$ must be false, and $\sqrt{2}$ is irrational. □

Now that we've discussed a bunch of valid proof techniques, I want to leave you with a word of warning about an invalid "proof" technique that students sometimes use. While showing that $\neg P \Rightarrow$ False is a valid way to prove that P is true, showing that $P \Rightarrow$ True does *not* mean that P is true (remember that any implication with a false premise is true). For example, if I start from $1 = 2$, I can multiply both sides by 0 to obtain the true statement $0 = 0$, but clearly this does not imply that $1 = 2$.

3.5 Acknowledgments

Some of the topics and problems in this chapter are inspired by ones in Building Blocks for Theoretical Computer Science by Margaret Fleck, the textbook that I used to teach Discrete Structures at UIUC.

3.6 Comprehension Questions

1. Suppose that we want to prove the claim "For every positive integer n, $n^2 + n$ is even." by induction on n. Which of the following would be an appropriate induction hypothesis to use?

   (a) Assume that for every positive integer k, $k^2 + k$ is even.

   (b) Assume that for every integer $k < n$, $k^2 + k$ is even.

   (c) Assume that for every positive integer $a < n$, $a^2 + a$ is even.

   (d) Assume that $n^2 + n$ is even.

   (e) Assume that for every integer n with $0 \leq n < k$, $n^2 + n$ is even.

2. Given that $2^{20} = 1048576$, what is the value of $\sum_{i=1}^{19} 2^i$?

3. Suppose that we design an algorithm ALG to solve the following problem:

   Problem: Given a positive integer n, determine whether or not n is a power of 2 (i.e. determine whether or not $\log_2 n$ is an integer).

   Which of the following statements would we need to prove in order to show that ALG correctly solves the problem? (select all that apply)

   (a) For every positive integer n which is a power of 2, ALG(n) returns true.

   (b) For every positive integer n which is not a power of 2, ALG(n) returns false.

   (c) For every non-positive integer n, ALG(n) returns false.

   (d) For every non-positive integer n, ALG(n) reports that the input is invalid.

4. Which of the following loop invariants would be most appropriate to use in proving the correctness of the sorting algorithm below?

```
ShiftySort(A[1, ..., n]):
  For i from 1 to n − 1:
      For j from n − i to n − 1:
          If A[j] > A[j + 1]:
              Swap A[j] with A[j + 1]
          Else:
              Break
```

   (a) After k iterations of the outer for loop, $A[1, \ldots, k]$ is sorted.

   (b) After k iterations of the outer for loop, $A[1, \ldots, k + 1]$ is sorted.

   (c) After k iterations of the outer for loop, $A[n - k + 1, \ldots, n]$ is sorted.

   (d) After k iterations of the outer for loop, $A[n - k, \ldots, n]$ is sorted.

   (e) After k iterations of the outer for loop, $A[n - k - 1, \ldots, n]$ is sorted.

5. Find the main flaw in the following proof (it's not enough to give a counterexample to the claim, you should explain why the induction proof fails).

   Claim: All cats are the same color.

   Proof: We will show that for any positive integer n, for every set of n cats, all those cats are the same color. Since the size of the set of all cats on earth is some positive integer n,

this suffices to prove the claim. The proof is by induction on n.

Base Case: $n = 1$. Certainly, in a group of one cat, all cats have the same color (the color of that single cat). In order to have two cats of different colors we need at least two cats.

Induction Hypothesis: For every $k < n$, for every group of k cats, all cats in the group are the same color.

Induction Step: Let S be an arbitrary set of n cats. Let $c \in S$ be an arbitrary cat in the set. We want to show that every other cat in S has the same color as c. Let d be another cat in S. By the induction hypothesis, all of the cats in the set $S - \{c\}$ have the same color. Also by the induction hypothesis, all of the cats in the set $S - \{d\}$ have the same color. Therefore, both c and d are the same color as all of the cats in $S - \{c, d\}$, and therefore c and d are the same color as each other. Since c and d were chosen arbitrarily, this proves that all the cats in S are the same color.

3.7 Exercises

1. Use induction to show that

$$\forall n \in \mathbb{Z}^+ \quad \sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

2. Use induction to show that

$$\forall n \in \mathbb{Z}^+, \quad \sum_{i=1}^{n} \lfloor \log_7 i \rfloor = \lfloor \log_7 n \rfloor (n+1) - \frac{7^{\lfloor \log_7 n \rfloor + 1} - 7}{6}$$

3. Use induction to show that

$$\forall n \in \mathbb{Z}_{\geq 0}, \quad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

4. Prove that $\sqrt{3}$ is irrational.

5. Prove that $\log_2 3$ is irrational.

6. Consider the following algorithm:

```
RPM(x, y)
    r ← 0
    a ← x
    b ← y
    While b > 0:
        If b mod 2 = 1 :
            r ← r + a
        a ← 2a
        b ← ⌊b/2⌋
    Return r
```

Prove that for every $x, y \in \mathbb{Z}_{\geq 0}$, $\text{RPM}(x, y) = xy$, i.e. prove that RPM multiplies its inputs.

7. We define a function $f(n) : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ as follows:

$$f(0) = 2$$
$$f(1) = 7$$
$$f(k) = f(k-1) + 2f(k-2) \qquad \forall\, k \geq 2$$

   Prove that $\forall\, n \in \mathbb{Z}_{\geq 0},\ f(n) = 3 \cdot 2^n + (-1)^{n+1}$

8. Prove that the sorting algorithm below is correct (i.e. that given an array of integers, after running the algorithm the array will be sorted in ascending order). Hint: Start by showing that the algorithm always terminates.

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```

4 Asymptotic Analysis

You've almost certainly seen Big-O notation in a previous CS course. But you may not be familiar with the formal definitions of Big-O or its relatives Big-Ω, Big-Θ, little-o, and little-ω, and you probably don't have a lot of experience using these definitions in proofs.

In Section 4.1 we formally define Big-O notation and give some examples of using it to prove one function is Big-O of another. In Section 4.2 we will introduce related notation and show how to apply it. Finally in Section 4.3 we'll discuss some useful properties of Big-O and see some examples of more abstract proofs with these definitions.

4.1 Big-O

While you're probably familiar with using Big-O to describe the runtime of algorithms, it's actually a way to compare the growth rates of arbitrary functions. Informally, for functions f and g we say that one function $f(n) \in O(g(n))$ if $f(n)$ is upper bounded by a constant multiple of $g(n)$ for all large enough n.

Formally, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$O(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \leq cg(n)\}.$$

(Note that in a slight abuse of notation, we use "$\forall\, n \geq n_0$" to mean "$\forall\, n \in \mathbb{Z}$ such that $n \geq n_0$". The fact that n is an integer is implied by the fact that f is defined as a function on integers)

That is, $O(g(n))$ is the set of all functions f for which we can choose positive constants c and n_0 such that for every integer $n \geq n_0$ (i.e. every sufficiently large input), $f(n) \leq cg(n)$ ($f(n)$ is upper bounded by a constant multiple of $g(n)$). You can think of $O(g(n))$ as the set of all functions whose "dominant term" is less than or equal to $g(n)$'s (up to constant factors). For example, the set $O(2n^2 - 1)$ includes the following functions (among infinitely many others):

- $2n^2 - 1$ (we can choose $c = 1$ and $n_0 = 1$)
- $3n^2$ (we can choose $c = 3$ and $n_0 = 3$)
- $1000n$ (we can choose $c = 1000$ and $n_0 = 1$)

To prove that $f(n) \in O(g(n))$ for some specific functions f and g, we need to apply the definition. We need to show that f satisfies the condition for membership in $O(g(n))$, i.e. that $\exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \leq cg(n)$. As we saw in Section 2.3, to prove a "there exists" statement, we need to find specific values for the variables which make the statement true. In this case that means that we need to give a specific c and n_0 such that $f(n) \leq cg(n)$ for every $n \geq n_0$ (and then justify that the c and n_0 we chose actually do satisfy this predicate).

Let's see a couple examples.

Theorem 19.

$$3n^3 + 5n^2 - n + 14 \in O\left(\frac{n^3}{2}\right)$$

Proof. We need to choose positive constants c and n_0 such that $\forall\, n \geq n_0$, $3n^3 + 5n^2 - n + 14 \leq cn^3/2$. Let $c = 44$ and $n_0 = 1$. For every $n \geq 1$, we have $n^3 \geq n^2 \geq n \geq 1$, so

$$3n^3 + 5n^2 - n + 14 \leq 3n^3 + 5n^3 + 14n^3 = 22n^3 = 44 \cdot \frac{n^3}{2}.$$

□

Note that we don't need to choose the smallest possible value of c. Rather, pick whichever value of c makes the proof easiest. In the previous proof we chose $c = 44$, because it was twice the sum of the non-negative coefficients in $3n^3 + 5n^2 - n + 14$, allowing us to upper bound each term in the polynomial with a multiple of n^3. We also didn't need to explain why we chose this c in the

proof. Remember that the purpose of a proof is to convince your reader that a claim is true, not to explain how you discovered that the claim is true. Therefore a Big-O proof only needs to show that particular values of c and n_0 work, not explain how you came up with them.

Sometimes there isn't a natural inequality that we can use to directly prove the claim we want. In these cases, we may need to use induction.

Theorem 20.

$$2^n \in O(n!)$$

Proof. Let $n_0 = 1$ and $c = 2$. We will prove by induction on n that $2^n \leq 2 \cdot n!$ for every $n \geq 1$.

Base Case: As a base case, suppose $n = 1$. Then

$$2^n = 2 = 2 \cdot 1 = 2 \cdot 1! = 2 \cdot n!$$

Induction Hypothesis: Suppose that for all positive integers $k < n$, $2^k \leq 2 \cdot k!$.

Induction Step: Suppose $n \geq 2$. Applying the induction hypothesis to $n - 1$, we have that

$$\begin{aligned}
2^n &= 2 \cdot 2^{n-1} \\
&\leq 2 \cdot 2 \cdot (n-1)! && \text{(by our IH)} \\
&\leq 2 \cdot n \cdot (n-1)! && \text{(since } n \geq 2\text{)} \\
&= 2 \cdot n! && \text{(by the definition of factorial)}
\end{aligned}$$

□

In both of the previous examples, we were able to choose $n_0 = 1$ as long as we picked a large enough c. This almost always works, however if $g(n)$ sometimes outputs 0, then we usually need to pick n_0 large enough to guarantee that $g(n) > 0$ for all $n \geq n_0$. Note that $g(n)$ cannot output negative values (for positive integer inputs) because we've defined big-O only for functions with co-domain $\mathbb{R}_{\geq 0}$ (it's possible to extend the definition to functions with negative outputs, but those functions aren't very relevant in this course, since it's generally impossible for an algorithm to use a negative amount of resources).

Theorem 21.

$$\log_2(2n + 2) \in O(\log_3(n))$$

Proof. Let $n_0 = 2$ and $c = 6$. Then we have that for every $n \geq n_0$,

$$\begin{aligned}
\log_2(2n+2) &\leq \log_2(4n) && \text{(because } n \geq 1\text{)} \\
&= \log_2(4) + \log_2(n) && \text{(by Theorem 2)} \\
&= \log_2(n) + 2 \\
&\leq 3\log_2(n) && \text{(because } n \geq 2, \text{ so } \log_2(n) \geq 1\text{)} \\
&= 3\log_2(3)\log_3(n) && \text{(by Theorem 2)} \\
&\leq 3\log_2(4)\log_3(n) \\
&= 6\log_3(n)
\end{aligned}$$

□

Note that we couldn't choose $n_0 = 1$ here, because $\log_2(2 \cdot 1 + 2) = 2$, while $\log_3(1) = 0$, and 2 is not less than $0 \cdot c$ for any value of c.

Let's look at a trickier big-O proof.

Theorem 22.

$$\log_2^2 n \in O(n)$$

Proof. Let $n_0 = 1$ and $c = 49$. We will prove that $\log_2^2 n \leq 49n$ for all $n \in \mathbb{Z}^+$ by induction on n.

Base Cases: If $1 \leq n < 128$ then

$$\log_2^2 n < \log_2^2 128 = 7^2 = 49 = 49 \cdot 1 \leq 49 \cdot n.$$

Induction Hypothesis: Suppose that for all positive integers $k < n$, $\log_2^2 k \leq 49k$.

Induction Step: Let $n \geq 128$. Then

$$\begin{aligned}
\log_2^2(n) &= (\log_2(n/2) + 1)^2 && \text{(by Theorem 2)} \\
&= \log_2^2(n/2) + 2\log_2(n/2) + 1 \\
&\leq \log_2^2(n/2) + 3\log_2(n/2) && \text{(since } n \geq 2 \text{ so } \log_2 n \geq 1\text{)} \\
&= \left(1 + \frac{3}{\log_2(n/2)}\right) \log_2^2(n/2) \\
&\leq (3/2)\log_2^2(n/2) && \text{(because } n > 128 \Rightarrow \log_2(n/2) \geq 6 \Rightarrow 3/\log_2(n/2) \leq 1/2\text{)} \\
&\leq (3/2)\log_2^2(\lceil n/2 \rceil) \\
&\leq (3/2)(49)\lceil n/2 \rceil && \text{(by our IH)} \\
&\leq 49 \cdot \frac{3}{2} \cdot \frac{n+1}{2} \\
&= 49 \cdot \frac{3n+3}{4} \\
&\leq 49n && \text{(because } n \geq 3\text{)}
\end{aligned}$$

Thus, $\log_2^2 n \leq 49n$, as desired. □

There are several things to note about this proof. The choice of $c = 49$ and the base case of $n < 128$ may seem like arbitrary magic numbers, but they actually come from thinking about the structure of our induction proof.

The starting point for developing this proof was the logic of the induction step. We need to relate $\log_2^2(n)$ to $\log_2^2(k)$ for some $k < n$. $\log_2(n)$ doesn't have a clear relationship with $\log_2(n-1)$, but we do know a nice relationship between $\log_2(n)$ and $\log_2(n/2)$ (namely $\log_2(n) = \log_2(n/2) + 1$). Unfortunately, we can't necessarily apply our induction hypothesis to $n/2$, because $n/2$ may not be an integer. That's why before applying the IH we need to upper bound $n/2$ with $\lceil n/2 \rceil$, which is an integer.

If we can say that $\log_2^2(n) \leq (1+\varepsilon)\log_2^2(\lceil n/2 \rceil)$, for some $\varepsilon > 0$ then we can apply the IH to get that

$$\log_2^2(n) \leq (1+\varepsilon)c\lceil n/2 \rceil \leq (1+\varepsilon)c \cdot \frac{n+1}{2}.$$

In order to upper bound this with cn, we need $\varepsilon < 1$ and n sufficiently large. We could pick any value $0 < \varepsilon < 1$ and calculate a corresponding value of c that would make the proof work, but $\varepsilon = 1/2$ is an easy choice. In order to upper bound $3/\log_2(n/2)$ with $1/2$, we need $n \geq 128$, which is why we made $1 \leq n < 128$ our base case.

Assuming $n \geq 128$, the calculations in our induction step actually go through for any choice of c. The reason we chose $c = 49$ was because $49 = \log_2^2 128$, and this makes the proof of the base case simple.

Note that we could not have chosen $n_0 = 128$ using this same proof setup. In order to assume the induction hypothesis at $n = 128$, we need the base case to have proved our claim for all $1 \leq k < 128$. In particular, our proof uses the fact that the inequality holds for $n = 64$ to prove that it holds for $n = 128$, which would be a problem if we didn't have base case covering $n = 64$ (we could validly prove $\log_2^2(n) \in O(n)$ using $n_0 = 64$ and $c = 1$ by tweaking this proof slightly, I encourage you to think about how).

One final note on notation. Often you will see $f(n) \in O(g(n))$ written as $f(n) = O(g(n))$. This notation is slightly misleading, since, unlike equality, the big-O relationship is not commutative (e.g.

$n = O(n^2)$, but $n^2 \neq O(n)$), but it's very widespread, so you should understand it (As computer scientists, you should already be used to the idea of a one-way equals, since many programming languages use = for assignment. For example, in C++, "x = 5" is valid code whereas "5 = x" is not).

4.2 Other Asymptotic Comparisons

Big-O functions sort of like a $\leq$ relation for functions. $O(g(n))$ is the set of all functions which are eventually less than or equal to $g(n)$, up to constant factors. It's natural to also define a corresponding notion of $\geq$ for functions. This is called Big-Ω (read "Big Omega"). Similarly we have a version of "=", for functions. We use Big-Θ (read "Big Theta") to denote the set of functions which grow at the same rate as $g(n)$ up to constant factors.

Formally, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$\Omega(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \geq cg(n)\}$$

(this is just the big-O definition with $f(n) \leq cg(n)$ changed to $f(n) \geq cg(n)$), and

$$\Theta(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \exists\, c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

(here we need f to be bounded on both sides by constant multiples of g).

Note that the definition of big-Ω immediately implies that if $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$, and vice-versa (we can use the same values for c and n_0 in both definitions).

Theorem 23.

$$\forall\, a, b \in \mathbb{Z}^+ \text{ such that } a, b \geq 2, \log_a(n) \in \Theta(\log_b n)$$

Proof. Let a and b be positive integers greater than or equal to 2. Let $c_1 = c_2 = \log_a b$, and let $n_0 = 2$. Then, by Theorem 2, $\log_a n = c_1 \log_b n = c_2 \log_b n$, so for all $n \geq n_0$, we have

$$c_1 \log_b n \leq \log_a n \leq c_2 \log_b(n),$$

so $\log_a(n) \in \Theta(\log_b n)$ □

By Theorem 23, the base of a logarithm doesn't really matter from an asymptotic perspective (it only changes the value of the function by a constant factor), so when dealing with logarithms in Big-O and related notations, we often omit the base and simply write $O(\log n)$. Do be careful with this, however. If a logarithm shows up inside of an exponent, the base is relevant again, $O(\log_2 n)$ is the same as $O(\log_4 n)$, but $O(4^{\log_4 n})$ is the same as $O(n)$, while $O(4^{\log_2 n})$ is the same set as $O(n^2)$.

Finally, we have notions of $<$ and $>$ for functions, which we call little-o and little-ω ("little omega"). For a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$o(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \forall\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \leq cg(n)\}$$

to be the functions that are "less than" $g(n)$, asymptotically. Note that we have changed the "$\exists\, c$" from the Big-O definition into a "$\forall\, c$". This definition is saying that for any possible choice of c, no matter how tiny, $f(n)$ will eventually become smaller than $cg(n)$ (and stay smaller for all large enough n).

Similarly, for a function $g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ we define

$$\omega(g(n)) := \{f : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0} \mid \forall\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{Z}^+, \forall\, n \geq n_0, f(n) \geq cg(n)\}$$

to be the functions which grow faster than $g(n)$ by more than a constant factor.

Let's see an example of using these new definitions:

Theorem 24.

$$\frac{n^3}{10} - 3n^2 + 1000 \in \omega(n^2)$$

Proof. We need to show that for every possible choice of c, there exists some n_0 such that for every $n \geq n_0$, $\frac{n^3}{10} - 3n^2 + 1000 \geq cn^2$.

Let c be an arbitrary positive real number. Choose $n_0 = \lceil 10(c+3) \rceil$, so that we know n_0 is an integer with $n_0 \geq 10(c+3)$. Then we have

$$n^3/10 - 3n^2 + 1000 > n^3/10 - 3n^2 = (n/10 - 3)n^2 \geq (n_0/10 - 3)n^2 \geq ((10(c+3))/10 - 3)n^2 = cn^2.$$

□

4.3 Properties of Big-O and Friends

In this section we'll show a variety of useful properties that will allow us to express the asymptotic behavior of new functions using the behavior of functions we've already seen.

Theorem 25. *Let $f, g : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$. Then $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.*

Proof. To prove an "if and only if" statement, we need to show both directions.

First we prove the forward direction ($f(n) \in \Theta(g(n)) \Rightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$). Suppose $f(n) \in \Theta(g(n))$. Let $c_1, c_2 \in \mathbb{R}^+$, $n_0 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$ (these exist by the definition of big-Θ). Then, by the definition of big-O, using the constants c_2 and n_0, $f(n) \in O(g(n))$, and by the definition of big-Ω, using the constants c_1 and n_0, $f(n) \in \Omega(g(n))$.

Now we prove the back direction ($f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))$). Suppose $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Let $c_1 \in \mathbb{R}^+$, $n_1 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1, f(n) \geq c_1 g(n)$ (these constants exist by the definition of big-Ω), and let $c_2 \in \mathbb{R}^+$, $n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_2, f(n) \leq c_2 g(n)$ (these constants exist by the definition of big-O). Then, if we let $n_0 = \max(n_1, n_2)$ (so that $n \geq n_0 \Rightarrow n \geq n_1$ and $n \geq n_2$) we have that $\forall\, n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$, so $f(n) \in \Theta(g(n))$. □

Theorem 26. *If $f_1, f_2, g_1, g_2 : \mathbb{Z}^+ \to \mathbb{R}_{\geq 0}$ are functions such that $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then:*

(a) $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

(b) $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Proof. (a) Let $c_1, c_2 \in \mathbb{R}^+$, $n_1, n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ and $\forall\, n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$ (such constants must exist by the definition of big-O). To show that $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$ we choose the constants $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$. Then, by our choice of c_1, c_2, n_1, and n_2, we have that for every $n \geq n_0$

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq (c_1 + c_2) g_1(n) + (c_1 + c_2) g_2(n) = (c_1 + c_2)(g_1(n) + g_2(n)).$$

So by the definition of big-O, $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$.

(b) Let $c_1, c_2 \in \mathbb{R}^+$, $n_1, n_2 \in \mathbb{Z}^+$ be constants such that $\forall\, n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ and $\forall\, n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$ (such constants must exist by the definition of big-O). To show that $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$ we choose the constants $c = c_1 \cdot c_2$ and $n_0 = \max(n_1, n_2)$. Then, by our choice of c_1, c_2, n_1, and n_2, we have that for every $n \geq n_0$

$$f_1(n) \cdot f_2(n) \leq c_1 g_1(n) \cdot c_2 g_2(n) = c(g_1(n) \cdot g_2(n)).$$

So by the definition of big-O, $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.

□

If a problem specifically asks you to show that one function is big-O (or big-Ω, etc.) of another using the formal definition (like in the exercises for this chapter), then you should make sure to give an explicit choice of c and n_0 and justify why they work. When using Big-O to describe the runtime of your algorithms, however, you do not need to go into the full details of a big-O proof. If you've concluded that the runtime is $O(n^2) + O(n)$, you can and should say that this is $O(n^2)$ without further proof.

4.4 Comprehension Questions

1. Suppose that we want to prove that $3n^2 + 10n + 2 \in O(n^2)$ using the formal definition of big-O. Which of the following values for c and n_0 could we choose? (select only one)

   (a) $c = 10, n_0 = 1$
   (b) $c = 7, n_0 = 3$
   (c) $c = 5, n_0 = 4$
   (d) $c = 3, n_0 = 100$

2. Suppose that we want to prove that $n! \in O(n^n - 1)$ using the formal definition of big-O. Which of the following values for c and n_0 could we choose? (select only one)

   (a) $c = 0.5, n_0 = 3$
   (b) $c = 0.6, n_0 = 2$
   (c) $c = 1, n_0 = 1$
   (d) $c = 500, n_0 = 1$

3. Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ defined by $f(n) = n^2(n \bmod 2)$ and $g(n) = n^2(n \bmod 3)$. Which of the following is true? (select all that apply)

   (a) $f(n) \in o(g(n))$
   (b) $f(n) \in \Theta(g(n))$
   (c) $f(n) \in \omega(g(n))$
   (d) None of the other answers are true

4. Let f and g be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ defined by $f(n) = 3n^3 + 10n$ and $g(n) = 10n^3 + 5$. Which of the following is true? (select all that apply)

   (a) $f(n) \in o(g(n))$
   (b) $f(n) \in \Theta(g(n))$
   (c) $f(n) \in \omega(g(n))$
   (d) None of the other answers are true

5. If $f(n) \in \omega(g(n))$, which of the following must also be true? (select all that apply)

   (a) $f(n) \in O(g(n))$
   (b) $f(n) \in \Omega(g(n))$
   (c) $f(n) \in \Theta(g(n))$
   (d) $f(n) \notin O(g(n))$
   (e) $f(n) \notin \Omega(g(n))$
   (f) $f(n) \notin \Theta(g(n))$

4.5 Exercises

1. Prove that $3n^5 + 4n^3 + 12 \in O(n^5)$ using the formal definition of big-O.

2. Prove that $\max(0, n^3 - 10n - 99) \in \Omega(n^2)$ using the formal definition of big-Ω

3. Let $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$ be functions which output only positive values, and suppose that $f(n) = O(g(n))$. Prove that there exists a constant $c \in \mathbb{R}^+$ such that for all $n \geq 1$, $0 \leq f(n) \leq cg(n)$ (i.e. show that for functions from $\mathbb{Z}^+$ to $\mathbb{R}^+$, we can always choose $n_0 = 1$ in the definition of big-O).

4. (a) Let f_1, f_2, f be functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ such that $f_1(n) \in \Theta(f(n))$ and $f_2(n) \in \Theta(f(n))$. Prove that $f_1(n) + f_2(n) \in \Theta(f(n))$

   (b) Let f be a function from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ and let $f_1, f_2, f_3 \ldots$ be an infinite sequence of functions from $\mathbb{Z}^+$ to $\mathbb{R}_{\geq 0}$ such that $\forall\, i \in \mathbb{Z}^+, \quad f_i(n) = \Theta(f(n))$. Define a new function $g(n) := \sum_{i=1}^{n} f_i(n)$. Is it necessarily true that $g(n) = \Theta(nf(n))$? Determine whether or not this is true, and give either a proof or a counterexample.

5 Modeling Computation and Runtime

What does it mean for an algorithm to be efficient? This depends on the context. In general an efficient algorithm is one which doesn't use "too much" of any resource. There are lots of resources that we might care about, but the two that we will care about in this course are *time* (how long the algorithm takes to run) and *space* (how much computer memory the program needs access to).

Comparing the runtime of two algorithms is more difficult than it may sound. In Sections 5.1 and 5.2 we will discuss several choices we need to make in order to meaningfully define what it even means for one algorithm to be faster than another. In Section 5.3, we will discuss a simplified model of computation called the (word) RAM model that allows us to describe algorithms more rigorously than we have done so far and to formally define their runtime. In Section 5.4, we'll show that despite its simplicity, we can actually implement the kinds of features we expect from a modern programming language in the RAM model. In Section 5.5 we'll see an example of a full program in the RAM model and rigorously analyze its runtime. Finally in Section 5.6, we'll show how to use our RAM model-based definition of runtime to analyze algorithms written in pseudocode, like those we've seen in Chapter 3.

5.1 Measuring the Runtime of Algorithms

Given an algorithm, how do we determine how fast it runs? For example, suppose we want to analyze the runtime of the algorithm below (which we previously saw in Section 3.3). How should we do it?

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

One approach would be to implement the algorithm in a real programming language, then run it and time how long the program ran for. This can be a good solution if we know exactly how the algorithm is going to be used. As a general way of comparing algorithms, however, it has serious problems:

- How do we choose what language/implementation to use? (This may affect the runtime significantly)
- What hardware do we run the program on? (The same code can run much faster on some computers than others)
- What input do we run the program on? (Insertion sort will be much faster on an array that is already mostly sorted than on an array that is in reverse sorted order. We need to know what kind of arrays we expect to get as inputs)
- How do we control for other factors that can impact the runtime? (Even on the same input and the same computer, the same program will not always take the same amount of time because of factors like what other processes are using RAM in the background).

Because of these reasons and others, the standard way of comparing the runtime of algorithms is to use a theoretical model. Instead of measuring the real time that a program takes, we measure the number of "steps" that the algorithm performs, where a step is a single basic instruction. On a real computer, these steps will not all take the exact same amount of time, but as long as we make reasonable choices about what to count as a step, the time each one takes on a real computer should be similar. We'll discuss how we determine what exactly counts as a single step of computation in Sections 5.2 and 5.3.

Counting computation steps in an abstract model solves most of the issues mentioned above, but it still leaves us with the problem of choosing what input to run the program on. The truth is that we cannot accurately capture the runtime of an algorithm by running it on any single input. Instead it makes more sense to view the runtime as a function of the input. Assuming that the input to our algorithm is encoded as a single string, we define, for each $s \in \Sigma^*$, Runtime(s) as the number of steps that our algorithm takes on input s.

Typically what we are most interested in is how the *size* (number of characters) of the input affects the runtime. To compute this we need some way to summarize the runtime of the algorithm across all inputs of a given size. Here are three natural ways we could define R(n), the runtime of our algorithm on inputs of size n.

- The *best-case* runtime is defined as $R_{best}(n) := \min_{s \in \Sigma^n}$ Runtime(s)
- The *worst-case* runtime is defined as $R_{worst}(n) = \max_{s \in \Sigma^n}$ Runtime(s)
- The *average-case* runtime is defined as $R_{average}(n) = (1/|\Sigma|^n) \sum_{s \in \Sigma^n}$ Runtime(s)

The best-case is mostly useful for impressing naive investors, but it can occasionally give insights on how to speed up an algorithm (perhaps we can find a way to achieve the best-case runtime on a larger set of strings). The average-case runtime sounds practical, but it is usually complicated to calculate, and assuming that the input is a uniformly random string is often not realistic. The worst-case runtime is what we will focus on in this course. When we talk about the "runtime" of an algorithm, we mean "the worst-case runtime function R_{worst}". An advantage of the worst-case runtime is that it gives us a guaranteed upper bound on how long our algorithm will take (on inputs of size n), even when we know nothing about the specific inputs our algorithm will be given.

We'll calculate the worst-case runtime of InsertionSort later in this chapter, after clarifying a few more details about how we analyze runtimes.

5.2 Defining Steps of Computation

As we discussed in the previous section, we would like to define the runtime of an algorithm on an input as the number of steps/instructions/lines of code that the algorithm executes when run on that input. We have to be careful how we define a "step", however, because different choices can give very different answers for the runtime of the same algorithm.

For example, consider the following high level pseudocode for an algorithm which determines whether or not a given string is a palindrome (reads the same backwards and forwards).

```
IsPalindrome(s):
    Return s = reverse(s)
```

What is the runtime of IsPalindrome? To answer this, let's try to count the number of "steps" that this algorithm takes when we implement it in an actual programming language.

Here is a simple Python implementation of IsPalindrome.

```
def is_palindrome(s):
    return s == s[::-1]
```

It's fine if you aren't familiar with this syntax, all the program is doing is comparing the string s with its reverse and returning whether or not they are equal. In Python, we can do this in a single line of code, regardless of how long s is. So if we define a "step" as "a single line of Python", then this algorithm runs in $\Theta(1)$ time.

If we wanted to write the same algorithm in C (or C++), then we don't have built in syntax for reversing an array, so we would need to write something like

```
bool is_palindrome(char s[], int size) {
    char* reverse = malloc(size);
    for(int i = 0; i < size; i++) {
```

```
        reverse[i] = s[size-i-1];
    }
    for(int i = 0; i < size; i++) {
        if (s[i] != reverse[i]) {
            return false;
        }
    }
    free(reverse);
    return true;
}
```

This version of the function has two for loops, each of which runs for up to `size` iterations, so if we define a "step" as a line of C code, then this algorithm runs in $\Theta(n)$ time.

The first formal models of computation were lambda calculus and Turing machines, developed in the 1930s by Alonzo Church and his student, Alan Turing. Defining these models is outside the scope of this course, but it can be proven that they have the same power as modern programming languages in the sense that any function which can be computed by a Python program (for example) can also be computed by some Turing machine. This does not mean, however, that a Turing machine can compute any function with the same asymptotic efficiency as a Python program.

A key distinction between Turing machine and actual programming languages like C is that Turing machines do not have random access, i.e. a Turing machine has no equivalent to the expression `s[i]` in our C program. Instead, a Turing machine has a single "head", which moves through memory one symbol at a time. Thus, if the Turing machine takes an array s as input and wants to access $s[i]$, it has to "move right" i times (i.e. the Turing machine treats an array more like a doubly linked list). If we were to implement the `is_palindrome` on a Turing machine it would look something like this (once again, it is not important to understand the details of this program):

```
is_palindrome(s):
    while head is not blank:
        char c = head
        write blank
        move right
        if head is blank:
            accept
        while head is not blank:
            move right
        move left
        if head != c:
            reject
        write blank
        move left
        while head is not blank:
            move left
        move right
    accept
```

This program features nested while loops. The outer loop iterates through $\lfloor n/2 \rfloor$ pairs of characters, and for each pair the inner while loops have to travel all the way across the input array and then all the way back to check if the first character of the input matches the last one. Thus, if we define a "step" as a single Turing machine instruction, then IsPalindrome runs in $\Theta(n^2)$ time.

So which of these is the right answer? Does IsPalindrome run in $\Theta(1)$, $\Theta(n)$, or $\Theta(n^2)$ time? A good way to build intuition here is to try running the Python and C programs that we wrote on an actual computer. While we said before that "the time the algorithm takes on an actual computer" is too imprecise to be a definition of runtime, we still want the asymptotic runtimes of programs in our theoretical model to match their asymptotic runtimes on actual hardware (i.e. if the number of

seconds that a program takes to run grows quadratically with the input size, ideally it should be an $O(n^2)$ time algorithm in our model).

If we run both the Python and C programs on strings of various sizes we find that the runtime for both programs seems to be linear in the length of the input string, suggesting that $O(n)$ is the right answer for this algorithm. In fact, if we look at how the Python interpreter translates the line `return s == s[::-1]` into machine code, we'll find that it uses a loop structure similar to our C program.

This doesn't mean that "a line of C code" is the right definition of a step of computation, however. We can still write single lines of C code that take a long time to run (e.g. if `a` and `b` are structs, then `a=b` copies all the attributes of `b` into `a`, which may take a long time if the struct is large). Furthermore, even if we did find a programming language where each instruction takes about the same amount of time (maybe some version of assembly), real programming languages are far too complicated, with too many instructions and edge cases to make them a reasonable definition of computation (the C++ standard, for example, is over 1000 pages long!). We want a simple minimalist model (like a Turing machine, which can be described in 1 page), but one which is powerful enough to allow manipulating arrays/strings with the same asymptotic efficiency as C or Python. It turns out that the right answer is essentially a Turing machine augmented with the power of random access, i.e. a "Random Access Machine".

5.3 The RAM Model

The property of being able to jump to any desired location in an array (or more generally, in memory) just using the index is called "random access". Essentially all modern computers rely heavily on RAM (random-access memory) so to model the time that operations take on real computers, it makes sense to use a model which includes random access. You can think of the RAM model as a very simple programming language.

Every RAM model program takes a string as input and returns a string as output (i.e. computes a function from $\Sigma^* \to \Sigma^*$ for some alphabet Σ). This is not actually a major limitation, because any type of input or output that a regular program uses is ultimately represented using a string of bytes. We'll talk more about encoding objects as strings in Section 6.1.

Memory in the RAM model consists of one long array of cells, which we call M. Each cell of M stores a single character from Σ. Initially the size of M is equal to n, the input length, and M stores the input to the algorithm in cells $M[1]$ through $M[n]$. While running, the machine can increase the size of M by using the "Malloc" instruction, which adds one cell to the end of M containing a specified character.

In addition to the memory array (which can grow arbitrarily large) a RAM model program also has a finite number of variables. RAM model programs have only two types of variables, character variables, which store a single symbol from Σ and integer variables. Integer variables in the RAM model function like a cross between int and pointer types in a language like C, because we can perform arithmetic with them, but we can also use them to index into memory. Like integer and pointer variables in C or many other languages, integer variables in the RAM model cannot store arbitrarily large integers. Instead, integer variables can store values in the range $[-2^w, 2^w]$, where w is a value called the "word size".

On a real computer, the word size is a fixed constant. These days most computers have a word size of 64 (i.e. a 64-bit operating system), but in the 90s, a word size of 32 was most common, and in the 70s many computers used a word size of 16. Because the word size bounds the values of variables which are used as indices into memory (i.e. addresses), the word size for a particular computer needs to be large enough to store all of the possible indices of memory locations. So if our computer has word size w, the maximum amount of RAM that it can use is 2^w. This means that with a word size of 16, we can use up to 64 kilobytes of RAM, with a word size of 32, we can use up to 4 gigabytes, and with a word size of 64, we can use up to 16 exabytes (about 17 billion gigabytes). This is why the word size used in actual computers has increased over time, because the amount of memory that we can cram into a computer chip has increased exponentially over the past several decades.

In our model, using a fixed word size doesn't work. We want to be able to describe algorithms that work for arbitrarily large inputs (because we are interested in the asymptotic behavior of their runtime), and so our machine's memory has unbounded size. Therefore, we define the word size w as $\max(64, \lceil \log_2(\text{memory_size}+1) \rceil)$, so that we always have $\text{memory_size} < 2^w$, meaning that we can store the index of any memory cell in an integer variable, but we also have $w \geq 64$ so that we store the kind of values that fit in an integer variable on a modern computer even if our program's memory is very small (e.g. if our program takes an empty string as input, then memory is initially empty). Note that this means that the word size can change over the course of the program as we allocate more memory.

Formally, a program in the (word)-RAM model consists of:

- An alphabet Σ
- A finite collection of variables (sometimes called "registers"), each of which is either a character variable (storing a single character) or an integer variable (storing a single integer between -2^w and 2^w, inclusive) including the following special variables which cannot be assigned to directly, only modified by the Malloc instruction):
  - memory_size: an integer variable initialized with the number of characters in the input
  - word_size: an integer variable initialized with $\max(64, \lceil \log_2(\text{memory_size}+1) \rceil)$
- A finite list of instructions (or "lines of code") where each instruction is one of the following:
  - $a \leftarrow b$
    - a must be a variable and b must be a variable or constant of the same type (integer or character).
    - This instruction copies the value of b into the variable a.
  - $M[i] \leftarrow c$
    - i must be an integer variable or constant and c must be a character variable or constant.
    - This instruction copies the value of c into position i in memory. If i is out of bounds, no memory is changed.
  - $c \leftarrow M[i]$
    - i must be an integer variable or constant and c must be a character variable.
    - This instruction copies the character at position i of memory into c. If i is out of bounds, c is left unchanged.
  - $i \leftarrow j$ op k
    - i must be an integer variable, j and k must be integer variables or constants, and op must be one of the following: $+, -, *, /$.
    - This instruction stores the result of j op k in i. / performs integer division (i.e. the fractional part is discarded). If j op k would not fit in an integer variable (because it is $< -2^w$ or $> 2^w$) or is undefined (because of division by 0), i is left unchanged.
  - If a op b: goto ℓ
    - a must be a variable, b must be a variable or constant of the same type (integer or character) as a and ℓ must be the number of an instruction (i.e. an integer between 1 and the total number of instructions), and op must be one of the following: $<, \leq, =, \geq, >$.
    - If the comparison a op b is true, then this instruction causes the machine to jump to instruction number ℓ and continue executing instructions from that position. If the comparison is false, the machine proceeds to the instruction immediately after this one, as usual.

- $i \leftarrow \text{decode}(start, size)$
  - i must be an integer variable, while $start$ and $size$ must be integer variables or constants.
  - This instruction interprets the string of length $size$ symbols beginning at position $start$ in memory (i.e. $M[start : start + size - 1]$) as the representation of a single integer, and stores that integer in i. If these characters do not represent a valid integer (either they don't encode an integer or the encoded integer is $< -2^w$ or $> 2^w$) then i is left unchanged.
- $M[start, size] \leftarrow \text{encode}(i)$
  - $i, start$, and $size$ must be integer variables or constants.
  - This instruction encodes i as a string over Σ and stores the resulting string beginning at index $start$. If the encoding of i has fewer than $size$ characters, it is padded with leading 0s until it has length $size$ (this is analogous to how an integer in C++ takes up 4 bytes, whether the number being stored is 1 or 100000). If the encoding of i has more than $size$ characters, then memory is left unchanged.
- $\text{Malloc}(c)$
  - c must be a character variable or constant.
  - This instruction adds a single new cell storing c to the end of M. It also increases memory_size by 1. If this causes the length of M to become 2^w, then this instruction also increases w by 1 (meaning that integer variables can now store larger values, so that it is possible to index into this new memory).
- $\text{Output}(start, size)$
  - $start$ and $size$ must be integer variables or constants.
  - Ends the program, outputting the string of length $size$ beginning at $M[start]$ (i.e. $M[start], \ldots, M[start + size - 1]$).

Since the RAM model is not a real programming language, different authors have their own slightly different versions of it. We could've allowed more instructions, or slightly fewer, without changing the asymptotic runtime of programs in the model (we'll see in Section 5.4 how some features we didn't define as part of the model, like loops and functions, are actually pretty easy to implement).

The one major difference you may encounter relates to the word size. The earliest descriptions of the RAM model did not feature a word size: variables could store arbitrarily large integers. This leads to an unrealistic model, however, because real computers cannot perform arithmetic on arbitrarily large integers in constant time. In fact, the time to perform arithmetic operations is a major bottleneck in areas like cryptography, where working with numbers with thousands of digits is common. Therefore the version of the RAM model where the values for integer variables are limited by the word size (often called the word-RAM model to distinguish it from the version without a word size) is the one most commonly used in modern algorithms research, and is the only one we will care about in this course. We'll discuss the runtime of arithmetic with large integers in our model in Chapter 9.

5.4 Implementing Quality of Life Features in the RAM Model

The instruction set we gave in Section 5.3 may seem pretty restrictive. It lacks a lot of nearly universal programming language features like loops, functions, and arrays. It turns out, however, that even with the limited instruction set of the RAM model, we can simulate all of these features and more. In this section we'll give some examples of how to do this.

First let's look at implementing if statements. An if-else statement like the one below (where "do X" is a stand-in for additional blocks of code) can be implemented in the RAM model using our conditional goto.

```
if x = y:
    do A
else:
    do B
do C
```

An equivalent RAM model program is:

1: If $x = y$: goto 4
2: do B
3: If $0 = 0$: goto 5
4: do A
5: do C

Note that we used the always true statement $0 = 0$ to convert the conditional goto into a statement that always jumps to the given line, so that we can avoid executing the "if" and "else" cases together. Also note that because our program moves on to the next line after the if when the condition is not met, we put the code from the else block before the code from the if block.

Our translation of an if-else statement relied on the condition in the if statement having a simple form like "x = y", because our conditional goto instruction requires the condition to be a basic comparison between two variables. We can simulate more complicated boolean expressions in the RAM model, however. For example, we can translate

```
if w = x and y = z:
    do A
do B
```

into

1: w_equals_x $\leftarrow 1$
2: If $w = x$: goto 4
3: w_equals_x $\leftarrow 0$
4: y_equals_z $\leftarrow 1$
5: If $y = z$: goto 7
6: y_equals_z $\leftarrow 0$
7: true_count $\leftarrow$ w_equals_x + y_equals_z
8: If true_count < 2: goto 10
9: do A
10: do B

Here we have created new "boolean" variables (integer variables that we only set to 0 or 1) to represent the truth values of the two sides of the "and". To check if both are true, we add up the truth values and check if the total is 2. We can implement other logical operators in a similar way (e.g. an "or" statement is true if the sum of the truth values of both sides is at least 1).

Now that we know how to implement a variety of conditionals, we can also implement loops. A while loop like

```
while x ≤ y:
    do A
do B
```

becomes

1: If $0 = 0$: goto 3
2: do A
3: If $x \leq y$: goto 2
4: do B

and a for loop like

```
for i from 1 to n:
    do A
do B
```

becomes

1: $i \leftarrow 1$
2: If $i > n$: goto 6
3: do A
4: $i \leftarrow i + 1$
5: If $0 = 0$: goto 2
6: do B

Conditional goto allows us to repeatedly return to the beginning of the loop as long as the loop condition is met.

Our RAM model has only two primitive data types, characters and integers, but we can create more complicated data types in memory out of these basic building blocks. For example, if we want to store an array of characters (i.e. a string), we can use two variables, one to store the index where the array starts, and one to store the length of the array. For example, we could represent a string s with the variables s_{start} and s_{length}. Then, to access $s[i]$, we simply need to access $M[s_{start} + i - 1]$ (since our indices for arrays and strings begin at 1).

We can represent an array of integers in a similar way, however, if we want to allow the integers in the array to be arbitrarily large, we should add a third variable keeping track of the number of characters we're using to represent each integer. Given variables A_{start}, A_{length}, and $A_{element_size}$, we can access $A[i]$ via $\text{decode}(A_{start} + (i-1) * A_{element_size}, A_{element_size})$.

We can also implement function calls in our RAM model, although the process is a bit more complicated (akin to how function calls are actually translated into low-level machine code, which also relies on goto instructions).

- We can jump to the code of a function with an unconditional goto (if $0 = 0$ goto)
- In order to return from the function, however, we need a way to remember which line of code called the function.
- Since we can have arbitrarily long chains of functions calling each other, we need to store this line number to return to in memory.
- Similarly, the parameters/local variables of the function also need to be stored in our memory array M
  - We could try making the parameters global variables (the only kind we have in the RAM model) instead, but this approach cannot handle recursive functions, which can have multiple different copies of their local variables existing at the same time.
- In essence what we want is an array of "stack frames" where each frame stores the local variables for a function as well as the instruction to return to when the function finishes.
- The details of this process are too technical for this course, but hopefully this gives you a sense of how we could translate large complicated programs into the RAM model

There are many more features we could implement, but this should be enough to give you some idea of how we can reduce more complicated code down to the instruction set from the RAM model. It turns out that in fact any function mapping strings to strings which can be computed by a program in C or Python or another normal programming language can also be computed by a program in the RAM model. Proving this is rather tedious, however, and beyond the scope of this course.

5.5 Analyzing the Runtime of Algorithms

Now that we've fully described a computational model, we can finally define the runtime of an algorithm. The (worst-case) runtime of an algorithm ALG is a function $f : \mathbb{Z}_{\geq 0} \to \mathbb{Z}^+$ where $f(n) :=$ the maximum over all $x \in \Sigma^n$ of the number of RAM model instructions that ALG(x) executes.

Let's look at a couple of examples of full programs in the RAM model and analyze their runtimes. We'll start with the IsPalindrome algorithm that we discussed in Section 5.2.

```
IsPalindrome(s):
  Return s = reverse(s)
```

Here is an implementation of IsPalindrome in our model. Lines 1 through 9 compute the reverse of the input string, storing it in positions $n+1$ through $2n$ in memory, and then lines 10 through 32 compare the string with its reverse to check for equality. As you can see, even an algorithm that is extremely conceptually simple can have a rather long description in the RAM model!

- Alphabet: Any set containing all possible input characters as well as English letters a-z.
- Integer variables: memory_size, word_size, input_size, i, j
- Character variables: temp, forward_char, reverse_char

1: input_size ← memory_size
2: i ← 0
3: If i = input_size: goto 9
4: j ← input_size - i
5: temp ← $M[j]$
6: Malloc(temp)
7: i ← i + 1
8: If 0 = 0: goto 3
9: i ← 1
10: If i > input_size: goto 25
11: forward_char ← $M[i]$
12: j ← i + input_size
13: reverse_char ← $M[j]$
14: i ← i + 1
15: If forward_char = reverse_char: goto 10
16: If memory_size ≥ 5: goto 19
17: Malloc('a')
18: If 0 = 0: goto 16
19: $M[1]$ ← 'f'
20: $M[2]$ ← 'a'
21: $M[3]$ ← 'l'
22: $M[4]$ ← 's'
23: $M[5]$ ← 'e'
24: Output(1,5)
25: If memory_size ≥ 4: goto 28
26: Malloc('a')
27: If 0 = 0: goto 25
28: $M[1]$ ← 't'
29: $M[2]$ ← 'r'
30: $M[3]$ ← 'u'
31: $M[4]$ ← 'e'
32: Output(1,4)

Alright, so what is the runtime of this algorithm?

Theorem 27. *The runtime of* IsPalindrome *is* $12n + 11 + 3\max(0, 4 - 2n)$.

Proof. We break the program into pieces.

- Lines 1-2 run once each, so the total time (number of instructions run) for these lines is just 2.
- Lines 4-8 each run input_size many times (since i starts at 0 and they run until i is equal to input_size. Line 3 runs input_size + 1 times (since it is also executed in the final iteration when i is equal to input_size). Thus the total time for lines 3-8 is $6n + 1$.

- Line 9 runs once, for a total time of 1.
- The runtime of the rest of the program depends on the actual characters in the input—in particular it depends on whether or not the input string is a palindrome. Let's separately consider both cases
- First suppose that the input string is a palindrome
  - Since the input is a palindrome, forward_char will always be equal to reverse_char on line 15. Thus, lines 11-15 run input_size many times (once for each character of the input), and line 10 runs input_size + 1 times (since it also runs in the final iteration when i is equal to input_size + 1). Thus, the total time for lines 10-15 is $6n + 1$
  - After matching all of the characters in the forward and reverse strings, the program jumps to line 25, so lines 16-24 do not run at all.
  - When we reach line 25, memory_size is equal to double input_size (since we doubled the size of memory to create the reversed copy). Therefore lines 26-27 run $\max(0, 4 - 2n)$ times and line 25 runs $1 + \max(0, 4 - 2n)$ times (since it runs each time our memory size is less than 4, plus one extra time when our memory size is at least 4). Thus, the total time for lines 25-27 is $1 + 3\max(0, 4 - 2n)$.
  - Lines 28-32 run once each, for a total time of 5
  - Thus, if the input is a palindrome the total runtime of IsPalindrome is

$$2 + (6n + 1) + 1 + (6n + 1) + 1 + 3\max(0, 4 - 2n) + 5 = 12n + 11 + 3\max(0, 4 - 2n)$$

- Now suppose that the input x is not a palindrome. In this case, there must be some index i where $x[i] \neq \text{reverse}(x)[i]$. Furthermore, the first index where this happens must be less than or equal to $n/2$ (i.e. before the middle of the string), because when we are comparing x to its reverse, every character in the first half gets paired with one from the second half and vice-versa, so every unmatched pair has a character from the first half of the string.
- Therefore, lines 10-15 each run at most $\lfloor n/2 \rfloor$ times before we reach the non-matching pair of characters and continue on to line 16 (and they run exactly this many times if the input is an "almost palindrome" where only the middle pair of characters don't match), so the worst-case runtime lines 10-15 is $6\lfloor n/2 \rfloor$
- As above, we note that the current memory_size is $2n$, so lines 16-18 run $1 + 3\max(0, 5 - 2n)$ times in total.
- Lines 19-24 run once each, for a total time of 6
- line 24 ends the program, so lines 25-32 never run
- Thus, the worst-case total runtime when the input is not a palindrome is

$$2 + (6n + 1) + 1 + 6\lfloor n/2 \rfloor + 1 + 3\max(0, 5 - 2n) + 6 = 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)$$

The overall worst-case runtime is the maximum of the worst-case runtime for palindromes and the worst-case runtime for non-palindromes:

$$\max(12n + 11 + 3\max(0, 4 - 2n), 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)).$$

We note that if $n \geq 3$, then $\max(0, 4 - 2n) = \max(0, 5 - 2n) = 0$, and so the runtime on palindromes is strictly larger than the runtime on non-palindromes. We can also check that for $n = 2$, the runtime for palindromes is still larger (the extra iterations of the loop comparing characters takes longer than the extra time to output false). Finally, if $n \leq 1$, the input must be a palindrome (an empty string

or a single character string is always the same as its reverse). Thus, for all input lengths the worst case input is a palindrome, and we can simplify the runtime to:

$$12n + 11 + 3\max(0, 4 - 2n).$$

□

That was an insane amount of effort that we would like to never go through again. Fortunately, calculating the *exact* runtime of algorithms in our model is not important (it depends too much on the specific choice of instruction set for the RAM model, which, as we previously noted is somewhat arbitrary). Instead what we really care about is the asymptotic runtime of the algorithm. Using the tools we learned in Chapter 4, we can confirm that the runtime expression we obtained for IsPalindrome is $\Theta(n)$. As we'll see in the next section, however, as long as all we care about is this final asymptotic answer, we can avoid needing to write out the fully messy RAM model program and calculating the exact runtime function.

5.6 Pseudocode

The simplicity of the RAM model is useful as a definition for runtime. We can calculate the exact number of steps needed for any operation that our program uses by breaking it down into individual RAM model instructions. Writing anything but the simplest algorithms directly in the RAM model is rather tedious, however, and in this course we are not going to concern ourselves with the exact number of steps that an algorithm takes.

Therefore, for the rest of this course, instead of describing algorithms using only RAM model instructions, we will write them in higher level pseudocode, like what we used in Chapter 3 when we discussed proving the correctness of algorithms.

Pseudocode does not have a specific list of valid instructions. In a similar way to how proofs can be written in regular English, using mathematical notation only when it is the clearest way to convey an idea, you can use everyday language in pseudocode instead of trying to adhere to a strict set of syntax rules. I will attempt to use a consistent style for the pseudocode algorithms in these notes, but you do not have to follow it when describing your own algorithms. If you prefer to make your pseudocode look more like your favorite programming language, that's fine, as long as the intent would be clear to someone who doesn't know that language.

A heuristic for whether something is a valid "single instruction" in pseudocode is whether you (or a slightly less clever classmate) would be able to break it down into the basic RAM model instructions. For example, in the previous section, we saw how to write a loop in the RAM model that reverses a string. So it's fine to write pseudocode like the following (which we've seen a bunch of times at this point), where we treat "reverse" as a built in operation, as long as we remember that this instruction actually takes $O(|s|)$ time.

```
IsPalindrome(s):
    Return s = reverse(s)
```

To analyze the runtime of an algorithm written in pseudocode, we don't need to convert each line into RAM model instructions. Instead all we need to figure out is the asymptotic behavior of the RAM model equivalent (e.g. could we implement this in a constant number of RAM model instructions). Generally it will be clear which lines run in constant time, and you can state this without justification (i.e. you don't need to describe the RAM model program at all). Be careful not to make the assumption that every non-loop line of pseudocode corresponds to a constant number of RAM model instructions, however. As we've seen with "reverse" this isn't always the case. We'll see more examples in chapter 9 of pseudocode lines that may look constant time at first glance but actually require many RAM model instructions.

Let's see an example of analyzing the asymptotic runtime of a pseudocode algorithm.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 28. *The runtime of* InsertionSort *is* $\Theta(n^2)$.

Proof. First we show that the runtime is $O(n^2)$. We note that the outer loop runs n times, and the inner loop runs at most n times, so the body of the inner loop runs at most n^2 times. The body of the loop takes $O(1)$ time per iteration, so the overall runtime is $O(n^2)$.

To show that the runtime is $\Theta(n^2)$, however, we also need to show that the worst-case runtime is $\Omega(n^2)$, i.e. we need to show that for every sufficiently large n we can find an input on which InsertionSort actually takes $\Omega(n^2)$ time.

Suppose that A is in reverse sorted order (i.e. sorted from largest to smallest). Then at the start of every iteration of the outer for loop, $A[i]$ will be smaller than every element of $A[1 : i-1]$, so the inner for loop will run the full $i-1$ times. Each iteration of the inner for loop includes at least one instruction, so we can lower bound the total runtime with the total number of iterations of the inner for loop.

Summing across all n iterations of the outer for loop, we find that the total number of iterations of the inner for loop is

$$\sum_{i=1}^{n}(i-1) = \sum_{i=1}^{n-1} i = (n-1)n/2 = n^2/2 - n/2 \in \Omega(n^2).$$

Note that we simplified the summation using Theorem 3. □

If you are paying very close attention, you may notice that this proof cheated. Look back and see if you can find the false claim.

With what we've stated so far, we can't assume that the body of the loop runs in $O(1)$ time! While comparing two integer variables is a constant time operation in the RAM model, remember that integer variables in the RAM model can store values at most 2^w, where $w = \max(64, \lceil \log_2(\text{memory_size}+1) \rceil)$. Thus, if our array A contains very large numbers, we may not be able to compare them in constant time. Consider, for example, the case where A contains just two numbers, each with millions of digits.

If A actually contains incredibly huge numbers, however, then n is not actually a good measure of the input size. e.g. if A has two numbers with millions of digits, then input_size is "millions", but n is only 2. It is much more convenient to work with the number of elements in an array rather than the total number of characters/bits. Therefore, unless otherwise stated, whenever we have an algorithm which takes an array of integers as input, we will implicitly assume that each of the elements of the array has at most $100w$ digits (there's nothing special about 100 here, it's just a nice large constant). That way we guarantee that we can store each one in a constant number of RAM model variables and perform comparisons (and arithmetic) on them in constant time. Given this new assumption, the proof of Theorem 28 is correct.

5.7 Comprehension Questions

1. When we talk about the "runtime" of an algorithm in this course, which runtime function do we mean?
   (a) The best-case runtime
   (b) The average-case runtime
   (c) The worst-case runtime

2. True or false: in the RAM model, the word size is a fixed constant which cannot change during the execution of a program.

3. Suppose that we run a RAM model program with the empty string as input. At the start of that program, what is the largest value that it can store in an integer variable?

4. Which of the following are valid single instructions in a RAM model program? (assume that a, b, and c are integer variables).

   (a) $M[a] \leftarrow b + c$

   (b) If $a \bmod 2 = 1$: goto 1

   (c) $a \leftarrow \text{decode}(b, c)$

   (d) $a \leftarrow b/c$

   (e) Malloc

   (f) Output(a, c)

5. Give a big-Θ expression for the runtime of IsPalindrome.

6. Give a big-Θ expression for the runtime of InsertionSort.

7. Give an exact expression (not big-O/big-Θ) for the runtime of each of the following RAM model programs in terms of the input size n:

   1: If memory_size ≥ 10: goto 4
   2: Malloc('0')
   3: If $0 = 0$: goto 1
   4: Output(1,1)

8. Below is an algorithm which takes as input a non-empty array of n distinct integers and outputs the median. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
FindMedian(A[1, ..., n]):
  While |A| > 2:
      x ← max(A)
      y ← min(A)
      remove x from A
      remove y from A
  If |A| = 1:
      Return A[1]
  Else:
      Return (A[1] + A[2])/2
```

9. Below is an algorithm which takes as input a string of length n and reverses it. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
Reverse(s):
  n ← |s|
  For i from 1 to n:
      temp ← s[n]
      For j from n down to i + 1:
          s[j] ← s[j − 1]
      s[i] ← temp
  Return s
```

5.8 Exercises

1. The RAM model does not have a "mod" operation. Show that nevertheless we can implement $a \leftarrow b \mod c$ in a constant number of RAM model instructions. That is, give a sequence of RAM model instructions which will set a to $b \mod c$, assuming that a, b, c are existing integer variables. You may assume that b and c both have **positive** values. You may introduce any additional variables you wish.

2. Translate this pseudocode into a RAM model program.

```
CountAs(s):
    c ← 0
    For i from 1 to |s|:
        If s[i] = 'a':
            c ← c + 1
    Return c
```

3. In Section 5.6 we claim that we can represent numbers with at most $100w$ digits in a constant number of RAM model variables and perform arithmetic on them in constant time. Justify this by explaining how to actually represent numbers with $100w$ binary digits and perform addition on them in the RAM model (Hint: start by showing how to represent and manipulate numbers with $2w$ binary digits).

4. Determine with proof the asymptotic runtime of this algorithm

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```

6 Decidability

In Chapter 5 we saw how to formally define an algorithm as a program in the RAM model, and gave some intuition for why this definition allows us to describe any algorithm that we can write on an actual computer.

In this chapter we will explore the topic of computability. In Section 6.1 we discuss how we can encode objects as strings so that we can create algorithms with inputs or outputs of whatever type we want, including algorithms which take other algorithms as input. In Section 6.2 we will define decidability and look at how to prove that a language is decidable. Finally, in Section 6.3 we will see our first example of a problem that algorithms cannot solve.

6.1 String Encodings and Universal Machines

A RAM model program takes as input a single string. This may seem limiting, since many functions we care about involve inputs other than strings (such as integers, arrays, or graphs). It turns out, however, that we don't lose much by restricting ourselves to string inputs, because all kinds of other objects can be *encoded* as strings. For some object A, we use $\langle A \rangle$ to denote the string representation of A. Similarly for a sequence of objects $A_1, \ldots, A_k$ we use $\langle A_1, \ldots, A_k \rangle$ to denote a single string encoding all of the objects.

Note that there are a few objects we care about which cannot be encoded as strings. In particular, we cannot encode arbitrary real numbers as strings. Intuitively, this is because real numbers can have infinitely long decimal representations with no pattern to the digits, and our algorithms can only take *finite* input strings. Thus, our algorithms cannot take arbitrary real numbers as inputs or outputs, only ones which can be expressed as finite strings (e.g. rational numbers, which can be encoded using their numerator and denominator, even if their decimal representations would be infinitely long).

There are many different ways to encode the same object as a string (for example, we could represent integers in binary or in base-10 or in some more exotic system), and in future chapters it will sometimes be important to specify which one we're using. For this chapter, however, we are only focused on the correctness of our algorithms/programs, not their runtime or space usage, so the particular encoding scheme that we use will generally not matter. Therefore we will not add unnecessary clutter by formally specifying what encoding schemes we are using.

Whenever we write that an algorithm takes a particular kind of object as input, we implicitly mean that the algorithm interprets its input as an encoding of that object (i.e. if we say that an algorithm takes an integer as input, the actual input is the string $\langle n \rangle$, but the first thing that our algorithm does is convert this string into an integer).

When we say that an algorithm returns a particular object, we mean that the algorithm outputs the encoding of that object. For example, when working with the alphabet $\Sigma = \{0, 1\}$, the instruction "Return True" may actually be implemented by outputting the string "1", but we will not get into these encoding details in our pseudocode.

One important class of objects which we can encode with strings is algorithms themselves. After all, the source code of a program in the RAM model (or any programming language) is just a long string. This means that we can write algorithms that take the encodings of other algorithms as input (you probably use such a program frequently—one example is a compiler). Just as for other objects, we will use $\langle P \rangle$ to represent the encoding of the algorithm P as a string (we will adopt the convention of naming generic algorithms with P for "program" to avoid confusion with arrays which we often call A).

A **universal machine** is an algorithm that takes as input the encoding of an algorithm and a string, and simulates running the algorithm on the string (You can think of a universal machine as the RAM model's equivalent of the Python interpreter). More formally, a universal machine takes as input a string in the form $\langle P, x \rangle$, where P is an algorithm and x is a string. If $P(x)$ outputs a string, then the universal machine outputs the same string. If $P(x)$ loops forever (never reaching an "Output" instruction) then the universal machine also loops forever when run on $\langle P, x \rangle$.

It's not obvious that a universal machine should exist. Describing how to construct a universal machine was one of the major contributions of Alan Turing's paper "On Computable Numbers, with an Application to the Entscheidungsproblem", which is also where he first defined Turing machines. At a very high level, our universal machine can use an integer variable to keep track of which line of P it is currently simulating, and designate one section of its memory for storing the code of P, another section to store the memory of P, and a final section for scratch space used to help with tasks like converting the memory addresses used by P into addresses in the universal machine's memory. Formally describing a universal machine is tedious and not the focus of this course, however, so we omit the details. For the purposes of algorithms you write in this course, you may assume that a universal machine exists and therefore that algorithms which you write can simulate the execution of an algorithm given its encoding.

It's important to note that while simulating P with a universal machine preserves the output, it does not necessarily preserve the runtime. Running a universal machine on $\langle P, x\rangle$ will generally be asymptotically slower than directly running $P(x)$. Nevertheless, since we are not concerned with runtime in this chapter, we will allow ourselves to treat an algorithm which we have read from its source code like a regular function we have written and include in our pseudocode lines like "result $\leftarrow P(x)$" as an abbreviation for "Simulate running $P(x)$ and store the output in result".

6.2 Decidability

For the rest of this chapter, we will focus on algorithms that return only True or False.

We define a **language** as a set of strings. That is, for a fixed alphabet Σ, a language is a subset of Σ^*. For example, if our alphabet is $\{0, 1\}$, then $\{010, 1111, \varepsilon\}$ is a language of size 3, and $\{1^n 0 \mid n \in \mathbb{Z}^+\}$ is a language of infinite size. For an algorithm P, we define the **language of** P, denoted $L(P)$ by

$$L(P) := \{x \in \Sigma^* \mid P(x) \text{ returns True}\}.$$

If L is a language and P is an algorithm with $L(P) = L$, we say that P **accepts** L. Note that P may return False or loop on strings not in L. We say that a language L is **acceptable** or **semi-decidable** if there exists an algorithm that accepts L.

If L is a language, P is an algorithm that accepts L, and P returns False on all strings in $\Sigma^* \setminus L$, we say that P **decides** L. We say that a language L is **decidable** if there exists some algorithm that decides L. Note that every decidable language is decided by infinitely many different algorithms (we can always add extra pointless lines of code, e.g. increment a variable then decrement it). An algorithm which always returns True or False (never gets stuck in an infinite loop) is called a **decider**. Note that if P is a decider then P decides $L(P)$, while if P is not a decider then P merely accepts $L(P)$.

The simplest way to prove that a language is decidable is to give an algorithm and prove that the algorithm decides the language.

Theorem 29. *Let*

$$L := \{\langle n, p\rangle \mid n, p \in \mathbb{Z}^+, \text{ and there are at least } p \text{ prime numbers between 1 and } n\}$$

Show that L is decidable.

Proof. We show that L is decidable by giving an algorithm to decide it.

```
L-DECIDER(n, p)
    count ← 0
    For i from 2 to n:
        prime ← True
        For j from 2 to i − 1:
            If i mod j = 0:
                prime ← False
        If prime:
            count ← count+1
    If count ≥ p:
        Return True
    Return False
```

We note that, by definition, a positive integer $i \geq 2$ is prime if and only if it is not divisible by any positive integer between 2 and $i-1$, or equivalently if $i \mod j \neq 0$ for every j between 2 and $i-1$. Therefore, after the outer for loop of L-DECIDER the variable count stores the number of primes between 2 and n. This is the same as the number of primes between 1 and n, since 1 is not prime. Since L-DECIDER returns True if and only if count $\geq p$, L-DECIDER returns True if $\langle n, p \rangle \in L$ and returns False otherwise. Thus, L-DECIDER decides L. □

There are several things to note about this proof:

- The definition of a decidable language is one with an algorithm which decides it. Therefore to prove that a language is decidable, we can give an algorithm and then prove that this algorithm correctly decides the language.
- It's not sufficient to simply give the algorithm. We need to be able to prove that our algorithm actually does what we claim it does.
- Although we formally define an algorithm as a RAM model program, to prove that an algorithm exists it is sufficient to give high level pseudocode, since as we discussed in Section 5.6, any feature we can implement in a regular programming language we can also implement in the RAM model.
- Recall that an algorithm always takes a string as input. When we describe an algorithm as taking some other kind of input, we implicitly assume that the algorithm returns False when given a string that does not encode an object of this form. For example, in our proof we assumed that L-DECIDER takes as input a pair of positive integers, n and p (because this is what elements of L encode). What we mean by this is that L-DECIDER will begin by checking if its input encodes a pair of positive integers and return False if it does not.
- Any algorithm which decides a language is sufficient to prove that the language is decidable. It doesn't have to be a fast or clever algorithm (we also don't have to actually calculate the runtime of the algorithm). Often, the easiest way to prove a problem is decidable is some kind of "brute force" algorithm. There are much faster algorithms for this primality problem, but using one of them would just be making our correctness proof harder.

Theorem 30. *The language*

$$\{\langle P, x, k \rangle \mid P \text{ is an algorithm}, x \in \Sigma^*, k \in \mathbb{Z}^+, \text{ and } P(x) \text{ halts after at most } k \text{ steps}\}$$

is decidable.

Proof. Call this language L. We show that L is decidable by giving an algorithm to decide it.

```
L-DECIDER(P, x, k)
    Simulate P(x) for k steps
    If P(x) halted within the simulation:
        Return True
    Return False
```

We note that simulating $P(x)$ for only k steps can be done in finite time, even if $P(x)$ loops forever. To prove that L-DECIDER correctly decides L we need to show that $\forall \langle P, x, k\rangle \in L$, L-DECIDER$(P, x, k)$ returns True and $\forall \langle P, x, k\rangle \in \Sigma^* \setminus L$, L-DECIDER$(P, x, k)$ returns False. In this case both directions are straightforward. Let P be an algorithm, x be a string, and k be a positive integer.

- Suppose $\langle P, x, k\rangle \in L$. Then, by the definition of L, $P(x)$ halts after at most k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will halt within the simulation, so L-DECIDER(P, x, k) will return True.
- Now suppose $\langle P, x, k\rangle \in \Sigma^* \setminus L$. Then by the definition of L, $P(x)$ is still running after k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will not halt, and so L-DECIDER$(\langle P, x, k\rangle)$ will return False.

□

In this proof our algorithm included the line "Simulate $P(x)$ for k steps". From our discussion of universal machines, we know that simulating another algorithm (including simulating another algorithm for a fixed number of steps) is something that algorithms can do. Therefore, we will allow this "simulation" operation as a single line of pseudocode, even if actually implementing it in the RAM model would be complicated. Similarly since we are not discussing the details of encodings that we are using in this chapter, we will allow algorithms to convert between objects and their encodings without specifying how this is done. For example, if we have an algorithm that takes as input $\langle P\rangle$, the encoding of an algorithm P, we could write a line like

$$n \leftarrow \text{ the number of instructions in } P$$

without discussing how we obtained the number of instructions from the string $\langle P\rangle$.

It's important to understand that whether or not a language is decidable does not depend on whether we *know* an algorithm which decides the language, and in fact it is possible to prove that a language is decidable without constructing a specific algorithm which decides it, as the following problem demonstrates (we call such a proof "non-constructive").

Theorem 31. *Let*

$$L = \{\langle n\rangle \mid n \in \mathbb{Z}_{\geq 0} \textit{ and the base-10 expansion of } \pi \textit{ contains } n \textit{ consecutive } 7s\}$$

Then L is decidable.

Proof. We consider two cases, and show that in either case, an algorithm that decides this language exists.

- First, suppose that for every positive integer k, there is a sequence of k consecutive 7s somewhere in the decimal expansion of π. In this case, L is decided by the following algorithm:

```
L-DECIDER(n)
  Return True
```

- Now suppose that there is some integer k such that the decimal expansion of π does not contain a sequence of k consecutive 7s. Choose the minimum such k. By our choice of k, π contains a sequence of $k-1$ consecutive 7s, and this sequence contains within it a sequence of i consecutive 7s for each $i < k$ (simply take the first i elements of the sequence). On the other hand, π cannot contain a sequence of i consecutive 7s for any $i \geq k$, because this sequence would contain a sequence of k consecutive 7s, and by our choice of k, no such sequence exists. Therefore, L is decided by the following algorithm:

```
L-DECIDER(n)
  If n < k :
      Return True
  Return False
```

Note that we do not know which of these cases holds (and in the second case we don't know the value of k), and thus we don't know which of these algorithms decides L. Nevertheless, we know that one of the algorithms we described above must decide L, so L is decidable even if we aren't sure which of the algorithms we presented is the one that actually decides it.

Sidenote: Even though π is transcendental, it's actually an open problem whether or not its decimal expansion contains strings of 7s (or any other digit) of arbitrary length. □

6.3 The Halting Problem

Before Turing's famous 1936 paper it was widely believed that every problem could be solved by a sufficiently clever algorithm—i.e. that every language was decidable. One of Turing's largest contributions was proving that there exist undecidable languages—problems that algorithms cannot solve. Formally we say that a language is **undecidable** if it is not decidable. In this section we will see the first and most famous undecidable language and see how to prove that it is undecidable.

We define

$$\text{HALTS} = \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}.$$

That is, HALTS is a language of algorithm-string pairs where the algorithm halts (outputs a value) when run on the string. This would be a useful language to have a decider for. For instance, when writing a homework autograder, it would be useful to have a way to distinguish between programs that loop forever versus ones that are very slow.

Intuitively, we might believe that HALTS should be decidable by some kind of brute force approach. Perhaps we can just run the algorithm for a long time and see if it ever ends up on the same instruction with the same data in its variables and in memory? Repeating all of these things would indeed imply that the algorithm is looping. Unfortunately since a RAM model program can expand its memory an arbitrary number of times, it's possible for a RAM model program to loop without ever repeating the same exact memory contents (A trivial example is a program that just repeatedly calls Malloc forever). There is no way we can brute force over the infinitely many potential memory configurations that may occur while running a given algorithm, so this approach to deciding HALTS does not work.

Stronger evidence that the Halting problem must be hard comes from the fact that an algorithm which decides HALTS could solve essentially every open question in mathematics. Proofs in mathematics can be written in an extremely formal language which is verifiable by a computer. For example, a number of important mathematics proofs have been written and verified in Lean. In principle, any mathematical proof could be written in Lean (although the proof might be very long), and we could construct an algorithm that takes a string and determines whether or not it is a Lean proof of a particular theorem. For example, we could create the following algorithm:

```
RIEMANN(x)
  For i from 0 to ∞
      For each string s ∈ Σ^i
          If s encodes a proof of the Riemann Hypothesis in Lean:
              Return True
```

The Riemann Hypothesis is one of the most famous unsolved problems in mathematics, with a one million dollar prize for whoever solves it. If the Riemann hypothesis is provable (e.g. because it is true), then it must have a Lean proof, which has some finite length n. Since there are only finitely many strings of length at most n, our algorithm RIEMANN will check all of them in finite time, find the proof of the Riemann hypothesis and return True. On the other hand if the Riemann Hypothesis is false, then there cannot be a proof of it, so RIEMANN will loop forever. Thus, if we could decide whether or not RIEMANN halts (the choice of input does not matter here, since RIEMANN ignores its input string x), then we could determine whether or not the Riemann hypothesis is true. There's nothing special about the Riemann hypothesis for this purpose either, we could construct a similar algorithm for any mathematical conjecture. The idea of a single program which can solve every mathematical conjecture may seem a bit too good to be true.

Indeed, Turing proved the following theorem:

Theorem 32. HALTS *is undecidable.*

Proof. We prove this by contradiction.

Suppose for contradiction that HALTS is decidable. Then, by the definition of a decidable language, there exists an algorithm HALTS-DECIDER which decides HALTS. The key concept we will use to obtain a contradiction is running an algorithm on its own encoding (source code). We construct a new algorithm CONTRARY which takes as input an algorithm P, then asks HALTS-DECIDER whether the algorithm halts when run on its own source code. If HALTS-DECIDER says that $P(\langle P \rangle)$ halts, then CONTRARY enters an infinite loop. If HALTS-DECIDER says that $P(\langle P \rangle)$ loops forever, then CONTRARY immediately halts.

```
CONTRARY(P)
  If HALTS-DECIDER(P, ⟨P⟩) returns True:
      Loop forever
  Return True
```

Now consider what happens if we run CONTRARY($\langle$CONTRARY$\rangle$). Then CONTRARY will call HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$). We consider two cases:

- Suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns True. Thus, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Loop forever" line and loop forever. Therefore $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur.
- Now suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns False. Therefore, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Return True" line and halt.
  Thus, $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur either.

Thus, whether $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle$ is in HALTS or not, we get a contradiction, meaning that our initial assumption that HALTS-DECIDER exists must have been false, and HALTS is undecidable. □

Note that HALTS is semi-decidable, as shown by the following simple algorithm that accepts HALTS:

```
HALTS-ACCEPTOR(P, x)
  Run P(x)
  Return True
```

The "Run $P(x)$" line will eventually finish running if and only if $P(x)$ halts. Thus, if $P(x)$ halts, HALTS-ACCEPTOR returns True, and if $P(x)$ loops forever then so does HALTS-ACCEPTOR.

6.4 Comprehension Questions

1. True or false? Every acceptable language is decidable.
2. Suppose that U is a universal machine. We define two languages:

$$L_1 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$
$$L_2 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}$$

   Which of the following is true?

   (a) U decides the language L_1.

   (b) U decides the language L_2.

   (c) U accepts the language L_1, but does not decide it.

(d) U accepts the language L_2, but does not decide it.

(e) U does not accept or decide any of these languages.

3. True or false? The language $L := \{\langle n \rangle : n \in \mathbb{Z}_{\geq 0}$ and the base-10 expansion of π contains at most n consecutive 0s$\}$ is decidable.

4. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ (i.e. the digits 0-9 as well as a decimal point). Which of the following sets are languages? (Select all that apply)

   (a) Σ^*

   (b) $\mathbb{R}^+$

   (c) $\mathbb{Z}^+$

   (d) $\{\langle x \rangle \mid x \in \mathbb{Z}^+\}$

5. True or false? The language HALTS is acceptable.

6. Two of the following four languages are decidable. Determine which two.

   (a)
   $$\text{HAS-MULT} := \{\langle P \rangle \mid P \text{ is an algorithm which includes an instruction of the form } i \leftarrow j * k\}$$

   (b)
   $$\text{USES-MULT} := \{\langle P, x \rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ executes an instruction of the form } i \leftarrow j * k\}$$

   (c) NOT-HALTS $:= \Sigma^* \setminus$ HALTS

   (d) HALTS-FAST $:= \{\langle P, x \rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts after at most 100 steps$\}$

6.5 Exercises

1. Prove that if L_1 and L_2 are two decidable languages then $L_1 \cup L_2$ is decidable.

2. Prove that there exist two undecidable languages L_1 and L_2 such that $L_1 \cup L_2$ is decidable.

3. Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. Prove that if L and $\Sigma^* - L$ are both semi-decidable, then L is decidable.

4. Prove that the following language is decidable:

   NO-MAL-HALTS $:= \{\langle P, x \rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts without calling Malloc.$\}$

5. A graviton is a hypothetical elementary particle. Physicists are currently not sure whether or not gravitons exist. Prove that the following language is decidable (no knowledge of physics required):
   $$\text{GRAVITONS} := \begin{cases} \{\text{“yes”}\} & \text{if gravitons exist} \\ \{\text{“no”}\} & \text{otherwise} \end{cases}$$

6. We say that a RAM model program P is “pretty small” if it satisfies the following conditions:

   - P's alphabet is $\Sigma = \{0, 1\}$
   - P has at most 411 instructions.
   - All integer constants used in P are between 0 and 411.

   Prove that the following language is decidable:
   $$\text{SMALL-HALTS} := \{\langle P \rangle \mid P \text{ is a “pretty small” program and } P(\varepsilon) \text{ halts}\}$$
   (Hint: How many meaningfully distinct “pretty small” programs are there)

7 Undecidability Reductions

In the previous chapter we saw our first example of an undecidable language: HALTS. In this chapter we will discuss *reductions*, a powerful technique for relating a new problem to one that we've already solved which we will use throughout the rest of the course. In Section 7.1 we will define reductions and use them to show that a variety of languages about the encodings of algorithms are undecidable. In Section 7.2, we will introduce an undecidable problem which is *not* about the encodings of algorithms called the Post Correspondence Problem, and show how PCP can also be used as a basis for undecidability reductions. Finally in Section 7.3 we will extend our discussion of the limits of computation beyond algorithms that return a boolean by defining uncomputable functions and showing that a particular function on the integers is uncomputable.

7.1 Turing Reductions

One of the most powerful concepts in computer science is *reduction*. Reducing problem A to problem B means giving an algorithm to solve A, assuming that you already have an algorithm for B. Reductions are used in everyday life. For example, if I am lost in Texas I can reduce the problem of getting home to the problem of finding Highway 6 (since I already know how to navigate to my apartment from Highway 6). They are also commonly used in programming. For example, say that you write a function to find the minimum element of an array by calling a "sort" function from the standard library and then taking the first element of the array. You have reduced the problem of finding the minimum element to the problem of sorting an array. Reductions are more than just a technique for solving new problems, however. Reductions give us a way to compare the "difficulty" of problems, and as we'll see in this section, they are a useful tool for proving that a language is undecidable.

For two languages A and B we say that $A \leq_T B$ (read "A **Turing reduces to** B") if it is possible to construct a decider for A using a decider for B as a subroutine. We use a less than or equal symbol because Turing reductions allow us to compare the difficulty with respect to decidability. In particular, if $A \leq_T B$ then we can say the following:

1. If B is decidable, then so is A.
2. If A is undecidable, then so is B.

Conclusion 1 follows from the definition of a Turing reduction. If $A \leq_T B$, then we can construct a decider for A given a decider for B, and if B is decidable then we have a decider for B. Conclusion 2 is simply the contrapositive of conclusion 1.

Theorem 33. *The language*

$$\text{RET-TRUE} := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$

is undecidable.

Proof. We could use the same technique as in Theorem 32 to prove this language undecidable, but in the interest of demonstrating reductions, we will instead prove this by showing that

$$\text{HALTS} \leq_T \text{RET-TRUE}.$$

We need to construct a decider for HALTS, assuming that we have access to a decider for RET-TRUE. Let RTDecider be a decider for RET-TRUE. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate P(y)
            Return True
    Return RTDecider(Helper, x)
```

To show that HaltsDecider correctly decides HALTS we need to show that for every $\langle P, x\rangle \in$ HALTS, HaltsDecider(P, x) returns True and for every $\langle P, x\rangle \notin$ HALTS, HaltsDecider(P, x) returns False.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $P(x)$ halts, so Helper(x)'s simulation of $P(x)$ will finish and Helper(x) will return True. Therefore $\langle$Helper$, x\rangle \in$ RET-TRUE and thus RTDecider(Helper$, x)$ returns True, so HaltsDecider(P, x) also returns True.
- Now suppose that $\langle P, x\rangle \notin$ HALTS. Then $P(x)$ loops, so Helper(x)'s simulation of $P(x)$ will loop forever and Helper(x) will never return True. Therefore $\langle$Helper$, x\rangle \notin$ RET-TRUE and thus RTDecider(Helper$, x)$ returns False, so HaltsDecider(P, x) also returns False.

Thus, we have shown that HALTS $\leq_T$ RET-TRUE, so since HALTS is undecidable, RET-TRUE is also undecidable. □

This proof may seem a little counterintuitive. After all, we constructed a decider for HALTS which we just showed in the previous chapter is impossible! This reduction is essentially just another form of proof by contradiction. We showed that if RET-TRUE were decidable, then HALTS would be as well. Since we know HALTS is not decidable, neither is RET-TRUE.

It's also important to recognize that our decider for HaltsDecider constructed a helper algorithm Helper, *but never actually ran* Helper. If HaltsDecider simulated running Helper (or P) we couldn't say that HaltsDecider is a decider, because the simulation of P might cause HaltsDecider to loop forever. But since Helper is never run, just written as source code that gets passed into RTDecider (which always halts because we assumed it is a decider), HaltsDecider is guaranteed to always return an answer without infinite looping.

Theorem 34. *The language*

$$\text{EMPTY} := \{\langle P\rangle \mid P \text{ is an algorithm and } L(P) = \emptyset\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{EMPTY}.$$

Let EmptyDecider be a decider for EMPTY. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            Return True
    Return ¬EmptyDecider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$ and therefore returns True. Thus $L($Helper$) = \Sigma^* \neq \emptyset$, so $\langle$Helper$\rangle \notin$ EMPTY, so EmptyDecider(Helper) returns False, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L($Helper$) = \emptyset$, so $\langle$Helper$\rangle \in$ EMPTY, so EmptyDecider(Helper) returns True, so HaltsDecider(P, x) returns False.

□

Notice that once again, our machine HaltsDecider created a helper machine Helper without actually running it. This time Helper ignores its own input string and runs $P(x)$ (rather than $P(y)$) so that the behavior of Helper on *all* strings depends on the behavior of M on the specific string x. The other major difference compared to the previous reduction is that HaltsDecider is doing the *opposite* of what EmptyDecider does, because $P(x)$ halting corresponds to $\langle \text{Helper} \rangle \notin \text{EMPTY}$.

Theorem 35. *The language*

$$\text{ACCEPTS-2} := \{\langle P \rangle \mid P \text{ is an algorithm and } |L(P)| = 2\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{ACCEPTS-2}.$$

Let A2Decider be a decider for ACCEPTS-2. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            If y = ε or y = "411":
                Return True
            Return False
    Return A2Decider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x \rangle \in \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$. Therefore Helper(y) returns True ε and "411" and returns False on all other inputs. Thus $L(\text{Helper}) = \{\varepsilon, \text{"411"}\}$, so $|L(\text{Helper})| = 2$, so $\langle \text{Helper} \rangle \in \text{ACCEPTS-2}$, so A2Decider(Helper) returns True, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x \rangle \notin \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L(\text{Helper}) = \emptyset$, so $\langle \text{Helper} \rangle \notin \text{ACCEPTS-2}$, so A2Decider(Helper) returns False, so HaltsDecider(P, x) returns False.

□

At this point you may be seeing the general pattern. To prove that some language L containing the encodings of algorithms is undecidable, we show that $\text{HALTS} \leq_T L$. To prove this we construct a decider for HALTS. This decider for HALTS constructs a helper machine which runs $P(x)$ and then returns True if its input y is in some set $S \subseteq \Sigma^*$ (where the choice of S depends on L). If $\langle P, x \rangle \in \text{HALTS}$, then $L(\text{Helper}) = S$, otherwise $L(\text{Helper}) = \emptyset$. If machines with language S are in L and machines with language $\emptyset$ are not, or vice-versa, then we can decide HALTS by running our decider for L on Helper and returning either the same thing (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = S$) or the opposite (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = \emptyset$).

This general pattern can work even for some languages that look more complicated, like the following one which involves pairs of algorithms:

Theorem 36. *The language*

$$\text{UNIQUE-INTERSECT} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } |L(P_1) \cap L(P_2)| = 1\}$$

is undecidable.

Proof. To prove that UNIQUE-INTERSECT is undecidable, we show that

$$\text{HALTS} \leq_T \text{UNIQUE-INTERSECT}.$$

Let UIDecider be a decider for UNIQUE-INTERSECT. We create the following decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Run P(x)
            If y = ε:
                Return True
            Else:
                Return False
    Return UIDecider(Helper, Helper)
```

Correctness:

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $L(\text{Helper}) = \{\varepsilon\}$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\{\varepsilon\}| = 1$. Thus $\langle \text{Helper}, \text{Helper}\rangle \in$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return True, so HaltsDecider(P, x) will return True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then $L(\text{Helper}) = \emptyset$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\emptyset| = 0$. Thus $\langle \text{Helper}, \text{Helper}\rangle \notin$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return False, so HaltsDecider(P, x) will return False.

□

So far all of the reductions we have seen have been from HALTS. This isn't the only language we can reduce from, however. To prove that a language is undecidable, we can reduce from any language that already know is undecidable (so far we've added RET-TRUE, EMPTY, ACCEPTS-2, and UNIQUE-INTERSECT to that list).

Theorem 37. *The language*

$$\text{FINITE} := \{\langle P\rangle \mid P \textit{ is an algorithm and } L(P) \textit{ is finite}\}$$

is undecidable.

Proof. To prove that FINITE is undecidable, we will show that

$$\text{EMPTY} \leq_T \text{FINITE}.$$

Let FiniteDecider be a decider for FINITE. We create the following decider EmptyDecider for EMPTY:

```
EmptyDecider(P)
    Σ ← the alphabet of P
    Write out the source code of the following algorithm Helper:
        Helper(x)
            For num_steps from 1 to ∞:
                For n from 0 to num_steps:
                    For each s ∈ Σ^n:
                        Simulate P(s) for num_steps steps
                        If the simulation returned True:
                            Return True
    Return FiniteDecider(Helper)
```

Correctness:

- Suppose that $\langle P \rangle \in$ EMPTY. Then P never returns True for any input, so no matter what string s HELPER simulates P on, $P(s)$ will never return True, and therefore HELPER(x) will never return True for any x. Thus $L(\text{HELPER}) = \emptyset$. Therefore $|L(\text{HELPER})| = |\emptyset| = 0$, which is clearly finite, so $\langle \text{HELPER} \rangle \in$ FINITE. Therefore FINITEDECIDER(HELPER) will return True, so EMPTYDECIDER(P) will return True.
- Suppose that $\langle P \rangle \notin$ EMPTY. Then there exists some $y \in \Sigma^*$ such that $P(y)$ returns True. Let t be the number of steps that it takes for $P(y)$ to return True. We note that each iteration of the outer for loop of HELPER eventually finishes (i.e. runs in finite time) because there are only a finite number of strings of length at most num_steps, and we run P for only a finite number of steps on each of them. Thus, if it has not already returned True, HELPER will eventually reach an iteration of the outer for loop where num_steps is $\max(|y|, t)$. In this iteration, it will simulate $P(y)$ for at least t steps, and therefore the simulation will return True and so HELPER will return True.

  Therefore, HELPER will return True for every input string (since HELPER does the same thing for every input string x), and thus $L(\text{HELPER}) = \Sigma^*$, which is infinite. Therefore $\langle \text{HELPER} \rangle \notin$ FINITE, so FINITEDECIDER(HELPER) will return False, and therefore EMPTYDECIDER(P) will return False.

□

In this case, reducing from EMPTY did not make it easier to show that FINITE was undecidable (see if you can find an alternative proof using a reduction from HALTS). In the next section, we'll see an example of a problem where HALTS is actually not the best problem to reduce from to prove undecidability.

However, this proof did allow us to show off a technique for running a program on infinitely many strings in parallel (despite the fact that our model of computation is inherently sequential!). The approach that we used in this proof's helper machine of running P on increasingly large sets of strings for increasingly large numbers of steps (which guarantees that if there is any string for which P returns True, we will find it in some finite amount of time) is called **dovetailing**, and it occasionally comes up in other problems in complexity theory.

Note that our reduction would not have worked if we had tried to use a simpler helper machine like the one below which runs P on strings one at a time:

```
HELPER(x)
  For n from 0 to ∞:
      For each s ∈ Σ^n:
          Simulate P(s)
          If the simulation returned True:
              Return True
```

If, for example, $P(\varepsilon)$ loops forever, then this version of the helper algorithm will get stuck simulating $P(\varepsilon)$ forever, and never move on to longer strings. Thus, even if P returns True on some other string, this version of the helper machine would never find it. That's why we needed the dovetailing technique.

7.2 The Post Correspondence Problem

Not all undecidable languages are about encodings of algorithms. In 1946, Emil Post described the following problem, which became known as the Post Correspondence problem, and showed that it cannot be solved by any algorithm.

In the **Post correspondence problem**, we are given some finite set of dominoes where each domino has a string written on the top and the bottom. For example, we might be given the set:

$$\left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}.$$

The goal of the problem is to find a sequence of one or more copies of dominoes from this set such that the concatenation of the strings on the top of the dominoes is the same as the concatenation of the strings on the bottom of the dominoes. We call such a sequence a **match**. For example, the following arrangement would be a match for the set of dominoes above:

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right].$$

Both the top and the bottom strings are *abcaaabc*. On the other hand the set of dominoes below has no match:

$$\left\{\left[\frac{ab}{a}\right], \left[\frac{cab}{bc}\right], \left[\frac{a}{\varepsilon}\right], \left[\frac{aaa}{b}\right], \left[\frac{dcb}{bca}\right]\right\}.$$

To see this we note that we cannot use the fifth domino, because it has a d on top and none of the dominoes have a d on the bottom, and we cannot form a solution out of the first four dominoes, because on each of them the top string is longer than the bottom string, so we cannot have a sequence of dominoes where the top and bottom strings are even the same length, let alone equal to each other.

Theorem 38. *The language*

$$\text{PCP} := \{\langle P\rangle \mid P \text{ is an instance of the Post correspondence problem with a match}\}$$

is undecidable.

We will not prove this theorem, because a full proof would take several pages and get deep into the nitty-gritty instruction sets of RAM model programs. The very high level idea is to prove that HALTS $\leq_T$ PCP by constructing a set of dominoes for which any solution to the problem would have the matching top and bottom string represent the sequence of instructions and configurations that P goes through when run on x. This sequence of configurations only terminates (and thus corresponds to a finite sequence of dominoes) if $P(x)$ eventually halts.

Even though we did not prove it ourselves, we can use Theorem 38 to prove the undecidability of other languages.

Theorem 39. *The language*

$$\begin{aligned}\text{MPCP} := \{\langle P, d\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & d \text{ is a domino in } P \text{ and } P \text{ has a solution beginning with } d\}\end{aligned}$$

is undecidable.

Proof. To prove that MPCP is undecidable, we will show that PCP $\leq_T$ MPCP. Let MPCPDecider be a decider for MPCP. We create the following decider PCPDecider for PCP:

```
PCPDecider(P)
    For each domino d in P:
        If MPCPDecider(P, d) returns True:
            Return True
    Return False
```

□

Correctness:

- Suppose $\langle P\rangle \in$ PCP. Then P has a match, and this match must begin with some domino x from P. Since there are only finitely many dominoes in P and MPCPDecider(P, d) runs in finite time for each of them (since MPCPDecider is a decider), the for loop in PCPDecider will eventually reach an iteration where $d = x$ and it will run MPCPDecider(P, x). By our choice of x, MPCPDecider(P, x) returns True, so PCPDecider(P) will return True.

- Suppose $\langle P \rangle \notin$ PCP. Then P has no solution, so it does not have a solution starting with d for any domino d in P. Therefore all of the calls that PCPDecider makes to MPCPDecider will return False, so PCPDecider(P) will return False.

PCP has been used as a starting point to prove several other more interesting languages undecidable, although the proofs are beyond the scope of this course.

7.3 Uncomputable Functions

So far in this chapter and the previous one, we have focused on programs that only return True or False (or loop forever) in order to prove that languages are decidable or undecidable. The idea of decidability can naturally be extended to arbitrary functions, however. We say that a function $f : X \to Y$ is **computable** if there is some algorithm that computes it, i.e. if there exists an algorithm P such that for every $x \in X$, $P(\langle x \rangle)$ outputs $\langle f(x) \rangle$ (We do not care what P does when its input string does not encode an element of f's domain X. We can generally just assume that the algorithm returns ε in this case). We say that a function is **uncomputable** if it is not computable.

While Turing reductions are only defined between languages, we can prove that a function is uncomputable using a similar technique: assume for sake of contradiction that the function is computable, then use the algorithm which computes the function to create a decider for HALTS (or some other known undecidable language). Since HALTS is undecidable, this is a contradiction, and so our assumption that the function was computable must have been false.

Let's see an example of this technique. We define a **binary RAM model program** as a RAM model program with alphabet $\Sigma = \{0, 1\}$ which does not use any integer constants other than 0 and 1. We define the **busy beaver function**, $BB : \mathbb{Z}^+ \to \mathbb{Z}^+$ as follows: $BB(n)$ is the maximum number of steps that an n-instruction binary RAM model program can take on input ε before halting (here the max is across all n-instruction binary RAM model programs which halt on input ε).

Theorem 40. *BB is uncomputable.*

Proof. AFSOC that BB is computable. Let Beaver be an algorithm that computes BB.

We construct the following decider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of a binary RAM model program Helper:
        Helper(y)
            Return P(x)
    n ← the number of instructions in Helper
    t ← Beaver(n)
    Simulate Helper(ε) for t steps
    If the simulation of Helper halted:
        Return True
    Return False
```

We note that we have assumed here that we can simulate P using a helper algorithm which is a *binary* RAM model program. i.e. we have assumed that there exists a universal machine which uses $\Sigma = \{0, 1\}$ and uses no integer constants besides $\{0, 1\}$. This is true, and you may assume this fact in your proofs, but we will not prove it, since we did not even formally prove that universal machines exist in the first place.

We will now prove that HaltsDecider correctly decides HALTS:

- Suppose that $\langle P, x \rangle \in$ HALTS. Then Helper(ε) will halt. Since Helper is a binary RAM model program with n instructions, we know that it halts in at most $BB(n)$ steps. Since Beaver(n) computes $BB(n)$, this means that the simulation of Helper(ε) will halt within the t steps that we run it for, so HaltsDecider(P, x) will return True.

- Suppose that $\langle P, x \rangle \notin$ HALTS. Then Helper(ε) loops forever, so when we simulate it for t steps it will not halt. Thus HaltsDecider(P, x) will return False.

Thus, we have constructed a decider for HALTS, but by Theorem 32, HALTS is undecidable. This is a contradiction, so our initial assumption that BB was computable must be false. □

The busy beaver function is traditionally defined in terms of Turing machines. Since we have avoided describing the technical details of Turing machines, in this course we will use our version based on binary RAM model programs. Note that the restriction to programs that only use the constants 0 and 1 is necessary for the function to be well-defined, because if we allow arbitrarily large constants then we can make small programs that run arbitrarily long. For example, in the following program we can set c to any positive constant to obtain a 5-line program that runs for more than $3c$ steps.

1: i ← 0
2: Malloc('0')
3: i ← i + 1
4: If i < c: goto 2
5: Output(1,1)

We can create a binary RAM model program with the same functionality as the one above, but only by letting the number of instructions grow in proportion to c (i.e. by replacing the loop with c copies of lines 2 and 3).

7.4 Comprehension Questions

1. Suppose that A and B are languages and $A \leq_T B$. Which of the following can we conclude? (select all that apply)

   (a) If A is decidable, then B is decidable.

   (b) If B is decidable, then A is decidable.

   (c) If A is undecidable, then B is undecidable.

   (d) If B is undecidable, then A is undecidable.

2. Let IS-EVEN $:= \{\langle n \rangle \mid n \text{ is an even integer}\}$. Let SUBSET-SUM $:= \{\langle S, k \rangle \mid S \subseteq \mathbb{Z}, k \in \mathbb{Z}, \exists T \subseteq S, \sum_{x \in T} x = k\}$. Which of the following are true?

   (a) IS-EVEN $\leq_T$ HALTS

   (b) IS-EVEN $\leq_T$ SUBSET-SUM

   (c) SUBSET-SUM $\leq_T$ HALTS

   (d) SUBSET-SUM $\leq_T$ IS-EVEN

   (e) HALTS $\leq_T$ IS-EVEN

   (f) HALTS $\leq_T$ SUBSET-SUM

3. Define a language RET-FALSE $:= \{\langle P, x \rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns False}\}$. Suppose we wish to show that HALTS $\leq_T$ RET-FALSE. We assume that we have a decider RFDecider for RET-FALSE and give the following decider for HALTS:

   ```
   HaltsDecider(P, x)
     Write down the source code of an algorithm Helper.
     Return RFDecider(Helper, x)
   ```

   To finish this proof we need to describe how to design Helper. Which of the following choices for Helper would allow HaltsDecider to correctly decide HALTS?

   (a)
   ```
   Helper(y)
     Run P(x)
     Return True
   ```

(b)
```
HELPER(y)
  Run P(x)
  Return False
```

(c)
```
HELPER(y)
  If y = x:
      Return False
  Run P(x)
  Return True
```

(d)
```
HELPER(y)
  If y = x:
      Return True
  Run P(x)
  Return False
```

(e)
```
HELPER(y)
  Run P(x)
  Return y = x
```

4. Suppose that we have an instance of the Post correspondence problem with the following 5 dominoes:

   (a) $\left[\frac{a}{bc}\right]$
   (b) $\left[\frac{b}{ab}\right]$
   (c) $\left[\frac{ca}{a}\right]$
   (d) $\left[\frac{cba}{cb}\right]$
   (e) $\left[\frac{acb}{ab}\right]$

   Which of these dominoes must come first in any match for this instance of the problem?

5. Let P be an instance of PCP and let d be a domino from P. Which of the following statements are guaranteed to be true? (select all that apply)

   (a) If $\langle P \rangle \in$ PCP then $\langle P, d \rangle \in$ MPCP.
   (b) If $\langle P, d \rangle \in$ MPCP then $\langle P \rangle \in$ PCP.

6. True or false? $\forall\, n \in \mathbb{Z}^+, \exists\, t \in \mathbb{Z}^+$, every n instruction binary RAM model program which halts on input ε halts on ε after at most t steps.

7.5 Exercises

1. Prove that the following language is undecidable:

$$L := \{\langle P \rangle \mid P \text{ is an algorithm and there exist at least two distinct input strings } x, y \in \Sigma^* \text{ such that } P(x) \text{ and } P(y) \text{ loop forever.}\}$$

2. Define

   REJECTS-3 := $\{\langle P \rangle \mid P$ is an algorithm which returns False for exactly 3 different input strings$\}$

   Prove that REJECTS-3 is undecidable.

3. Define

$$\text{EQ} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) = L(P_2)\}.$$

   Prove that EQ is undecidable.

4. Define

$$\text{DISJOINT} := \{\langle P_1, P_2\rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) \cap L(P_2) = \emptyset\}.$$

Prove that DISJOINT is undecidable.

5. Prove that RET-TRUE $\leq_T$ HALTS
6. Prove that EMPTY $\leq_T$ HALTS
7. Prove that the following language is undecidable

$$\begin{aligned}\text{ODD-PCP} :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & \text{which has a solution using an odd number of total dominoes}\}\end{aligned}$$

8. Prove that the following language is undecidable:

$$\begin{aligned}L :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem where}\\ & \text{every non-empty top/bottom string on a domino begins}\\ & \text{with the symbol 'a' and } P \text{ has a match}\}\end{aligned}$$

9. Show that the function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$f(n) :=$ the number of n-instruction binary RAM model programs which halt on ε
is uncomputable.
10. Show that the function $g : \{P \mid P \text{ is an algorithm}\} \times \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$g(P, n) :=$ the number of strings $x \in \Sigma^n$ for which $P(x)$ returns True
is uncomputable.
11. Show that the function $h : \{P \mid P \text{ is an binary RAM model program}\} \to \mathbb{Z}^+$ defined by
$h(P) :=$ The number of instructions in the shortest binary RAM model program with the same language as P (i.e. the P' with the fewest instructions for which $L(P') = L(P)$)
is uncomputable.
12. We define the following languages:

$$\begin{aligned}\text{MM3} :=\{\langle S\rangle : & S \text{ is a set of } 3\times 3 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\\ \text{MM4} :=\{\langle S\rangle : & S \text{ is a set of } 4\times 4 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\end{aligned}$$

For example, suppose S is a set containing the following matrices:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}, \begin{bmatrix}-1&-2&-3\\4&5&6\\-7&-8&-9\end{bmatrix}, \begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}, \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}$$

Then $\langle S\rangle \in$ MM3 because

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix} = \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$$

MM3 is known to be undecidable (you may assume this). Prove that MM4 is undecidable.

8 Divide and Conquer

In the last chapter we saw the power of reductions for proving problems undecidable. In this chapter and the next we will see the power of reduction for creating efficient algorithms for computable functions. Divide and conquer is a technique closely related to recursion where we solve a problem by reducing it to a smaller version of the exact same problem.

In Sections 8.1 through Section 8.3 we'll look at some common algorithms which use divide and conquer. In Section 8.4 we will discuss *recurrences*, which allow us to express the runtime of divide and conquer algorithms, and in Section 8.5, we'll see how to convert them into the normal asymptotic bounds we're used to. In the next chapter we will explore another important application of the divide and conquer technique.

8.1 Binary Search

Binary search is one of the simplest and most versatile algorithms. It's an algorithm you may have used yourself before even learning to program. For example, suppose that your professor tells you to turn to page 394 in your textbook. In order to find the appropriate page, you start by opening to the middle of the textbook, and see that you're on page 250. This comes before 394, so you keep a finger in page 250 and flip to somewhere that's about in the middle of the second half of the book, which gets you to page 375. This still isn't far enough, so you hold page 375 and flip to halfway through the remainder of the book, getting page 437. Now you've gone too far, so you find somewhere halfway between 375 and 437, page 406. After also visiting pages 390 and 398, you finally arrive at page 394, having had to look at 7 total page numbers on the way. This is much more efficient than if you had simply flipped through the pages one by one, looking at all of the first 394 pages to find the one you wanted.

The idea of binary search is to rapidly find some desired object in a large search space by repeatedly performing checks that allow you to divide the search space in half (i.e. halve the number of possible solutions you need to consider). The canonical application of binary search is finding an entry in a sorted array. This standard binary search algorithm is given below:

```
BinarySearch(A[1, ..., n], x)
    ℓ ← 1
    u ← n
    While ℓ ≤ u:
        m ← ⌊(ℓ + u)/2⌋
        If A[m] = x :
            Return True
        If A[m] > x:
            u ← m − 1
        Else:
            ℓ ← m + 1
    Return False
```

Theorem 41. *For every sorted array of integers A and every integer x,* BinarySearch *returns True if and only if x is in the array A. Furthermore the runtime of* BinarySearch *is $O(\log n)$.*

Proof. We begin by proving the correctness. We will prove the following loop invariant for the while loop: At the start of each iteration of the loop, if x is in A then x is in $A[\ell : u]$ (i.e. we will induct on the number of iterations of the while loop).

As a base case, note that after 0 iterations of the while loop (i.e. before the first iteration), $\ell = 1$ and $u = n$, so if x is in A then it is in $A[\ell : u]$.

Now suppose as an induction hypothesis that at the start of some iteration of the loop the invariant holds. We consider three cases:

- Suppose $A[m] = x$: Then $x \in A$ and the algorithm correctly returns True and there is no next iteration of the loop.

- Suppose $A[m] > x$: Then, since the array is sorted, every element in $A[m : u]$ is greater than x. Thus, if x was in $A[\ell : u]$ then x must be in $A[\ell : m-1]$, so if the loop invariant held at the start of this loop iteration it also holds at the start of the next iteration.
- Suppose $A[m] < x$: Then, since the array is sorted, every element in $A[\ell : m]$ is less than x. Thus, if x was in $A[\ell : u]$ then x must be in $A[m+1 : u]$, so if the loop invariant held at the start of this loop iteration it also holds at the start of the next iteration.

Thus, our invariant holds at the start of every iteration. We note that $u - \ell$ begins at $n-1$ and decreases with every iteration of the loop where we don't return True (because either ℓ increases or u decreases). Therefore, after a finite number of iterations either we have returned True already or we have $u - \ell < 0$, meaning that $u < \ell$. At this point, our loop invariant implies that x is not in A and the algorithm correctly returns False. Therefore the algorithm is correct because it always terminates in finite time (never enters an infinite loop) and whenever it returns a value that value is correct.

Now we analyze the runtime. Since u and ℓ are always between 1 and n, all of the arithmetic in the loop is with positive integers less than or equal to n (i.e. numbers whose value is polynomial in the input length) so we can perform the arithmetic in the loop in constant time, and thus each loop iteration runs in $O(1)$ time.

We note that the size of our search interval, $A[\ell : u]$ is initially n. In each iteration of the loop where we do not return, the size of this interval decreases by a factor of at least 2. By definition, $\log_2(n)$ is the number of times that we need to divide n by 2 in order to get down to 1, so after $O(\log n)$ iterations of the loop, if we have not already returned, the size of $A[\ell : u]$ will be 0. At this point, we exit the loop and return False. Since each iteration is constant time, the total runtime for the algorithm is $O(\log n)$.

We note that the theorem only asks us to prove a big-O bound so we do not need to give an example where the algorithm takes $\Omega(\log n)$ time. □

It's important to note that the correctness argument for binary search crucially hinges on the fact that the input array is *sorted*. Without this assumption, we can't do anything better than a $\Theta(n)$ time linear search (i.e. just iterate through every element of the array and check if it's the one we're looking for).

8.2 Applications of Binary Search

Binary search has applications far beyond just searching in arrays. In this section we will look at a few of them.

In Section 3.2 we saw the following algorithm for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer n:

```
IntegerSquareRoot(x):
    i ← 0
    While (i + 1)^2 ≤ x:
        i ← i + 1
    Return i
```

This algorithm works, but it's not particularly efficient. The while loop runs for $\Theta(\sqrt{x})$ iterations. Since the length of the input is $n \in \Theta(\log x)$, this means the number of iterations is $\sqrt{2^n} = 2^{n/2}$, so the algorithm is exponential time before even considering the time for computing $(i+1)^2$.

We can use the idea behind binary search to create a much more efficient algorithm for this problem, which avoids squaring every positive integer less than $\sqrt{x}$.

```
FastSquareRoot(x, ℓ ← 0, u ← x):

  If ℓ = u:
      Return ℓ
  m ← ⌊(ℓ + u)/2⌋
  If (m + 1)^2 ≤ x:
      Return FastSquareRoot(x, m + 1, u)
  Else:
      Return FastSquareRoot(x, ℓ, m)
```

Theorem 42. *For every positive integer x,* FastSquareRoot(x) *returns* $\lfloor\sqrt{x}\rfloor$. *Furthermore,* FastSquareRoot *runs in time $O(n^3)$, where n is the number of binary digits in x.*

Proof. We first prove by induction on $u - \ell$ that if $\ell \leq \lfloor\sqrt{x}\rfloor \leq u$ then FastSquareRoot(x, ℓ, u) returns $\lfloor\sqrt{x}\rfloor$.

- Base Case: If $\ell = u$ and $\ell \leq \lfloor\sqrt{x}\rfloor \leq u$ then $\ell = \lfloor\sqrt{x}\rfloor$, so when the algorithm returns ℓ it returns $\lfloor\sqrt{x}\rfloor$.
- Induction Hypothesis: Suppose that for all $\ell', u' \in \mathbb{Z}_{\geq 0}$ with $u' - \ell' < u - \ell$ and $\ell' \leq \lfloor\sqrt{x}\rfloor \leq u'$ we have that FastSquareRoot(x, ℓ', u') returns $\lfloor\sqrt{x}\rfloor$.
- Induction Step: Let $\ell \leq \lfloor\sqrt{x}\rfloor \leq u$. We note that $\ell \leq m < u$. We consider two cases:
  - If $(m+1)^2 \leq x$, then $m + 1 \leq \sqrt{x}$, and $u - (m + 1) \leq u - (\ell + 1) < u - \ell$, so by our IH, FastSquareRoot$(x, m + 1, u)$ returns $\lfloor\sqrt{x}\rfloor$.
  - Otherwise, $(m+1)^2 > x$, so $\sqrt{x} < m+1$ and thus $\lfloor\sqrt{x}\rfloor \leq m$. Furthermore, $u - m < u - \ell$, so by our IH FastSquareRoot(x, ℓ, m) returns $\lfloor\sqrt{x}\rfloor$.

  In either case, our recursive call returns $\lfloor\sqrt{x}\rfloor$, and therefore FastSquareRoot(x, ℓ, u) does as well.

To finish proving the correctness of FastSquareRoot we note that when we first call FastSquareRoot(x) the default parameters of $\ell = 0$ and $u = x$ satisfy $\ell \leq \lfloor\sqrt{x}\rfloor \leq u$, so by our inductive proof, FastSquareRoot(x) returns $\lfloor\sqrt{x}\rfloor$.

Now we analyze the runtime. As in analysis of binary search, we note that each recursive call $u - \ell$ decreases by a factor of 2, so since $u - \ell$ is initially x after $\Theta(\log x) = \Theta(n)$ recursive calls we will reach a call where $\ell = u$ and the function immediately returns. Before recursing each call to FastSquareRoot has to compute $\lfloor(\ell + u)/2\rfloor$ and $(m + 1)^2$. Since $\ell, u, m \leq x$, these numbers have length $O(n)$. Adding two n-digit numbers takes time $\Theta(n)$ and multiplying two n-digit numbers with the grade school algorithm (multiply each pair of digits) takes time $\Theta(n^2)$, so each call to FastSquareRoot does $O(n^2)$ work outside of the recursive calls, and the total runtime of the algorithm is $O(n) \cdot O(n^2) = O(n^3)$. (We will discuss algorithms for arithmetic on large numbers in more detail in Chapter 9) □

Sidenote: We could improve the runtime of this algorithm using one of the faster multiplication algorithms that we will discuss next chapter, but improving the runtime from $\Omega(2^{n/2})$ to $O(n^3)$ is already an exponential improvement over our previous square root algorithm.

The core property that allowed us to speed up the algorithm with the binary search technique is that squaring is a *monotone* function (over the non-negative integers), meaning that if $a > b$, then $a^2 > b^2$. Because of this, whenever we have the possible values for the square root narrowed down to a range we can pick a value m in the middle of the range and check how m^2 compares with x. If m^2 is bigger than x, then we can immediately rule out every number bigger than m, because their squares will also be bigger than x. Similarly, if m^2 is smaller than x, then we can rule out every number smaller than m, because their squares will be smaller than m^2. Either way, we cut our search space in half. This same "cut the search space in half" technique can be applied to a wide variety of problems.

Sometimes, we don't have an upper bound on the size of the space that we're searching. In this case, we can use a variant of binary search known as exponential search. For example, suppose that we are trying to determine the maximum weight that a bridge can support, rounded to the nearest kilogram. To help us, we have a function SimulateWeight which takes as input a number of kilograms, simulates putting that much weight on the bridge, and returns whether or not the bridge survived (this is obviously a very simplified model of civil engineering). We would like to figure out the maximum weight the bridge can support using as few calls as possible to SimulateWeight. Assuming that we have no clue how much weight the bridge can support, what strategy should we use?

Unlike in the square root problem, here we aren't given an upper bound, so we can't start halfway in-between the upper and lower bound. Instead, we can use the following approach (assuming that our bridge can support at least 1kg):

```
FindMaxWeight()
  i ← 1
  While SimulateWeight(i)
      i ← 2i
  Binary search between i/2 and i for the largest integer for which SimulateWeight returns True
```

Here we repeatedly double i until we find a value which is larger than the maximum weight that the bridge can support. When we break out of the while loop we know that i kilograms break the bridge, but $i/2$ do not, so we can binary search in the range $i/2$ to i to find the maximum weight the bridge supports. If the maximum weight that the bridge can support is y, then we use $\Theta(\log y)$ calls to SimulateWeight to find an $i > y$ and $\Theta(\log y)$ calls to SimulateWeight to binary search for y in the range from $i/2$ to i, so in total our algorithm makes $\Theta(\log y)$ calls to SimulateWeight. Note that for this problem we have to express our runtime in terms of the size of the answer rather than the size of the input (FindMaxWeight doesn't even use an input!). Also note that once again the reason we can use binary search is because our function SimulateWeight is monotone. If i kilograms of weight break the bridge, then we know that j kilograms will as well for any $j > i$.

We simply wrote "binary search" here rather than writing out the full details of a binary search algorithm. Because binary search is such a broadly applicable algorithm, it is common to use it as a subroutine in other algorithms. The details of maintaining the upper and lower bounds, and the inductive proof of correctness are a bit tedious to go through every time we want to use this algorithm. In your algorithms you may feel free to call binary search as a subroutine, and assume its correctness. Be careful, however, to make sure that the input you're searching through actually meets the preconditions for binary search (i.e. you're searching a sorted array, or trying to find a threshold for a monotone function).

8.3 Mergesort

Binary search involves reducing a problem to a single smaller version of the same problem. Having only one subproblem to solve is efficient, but not always possible. Sometimes the best we can do is to reduce the original problem to two or more smaller versions. In this case, once we've recursively solved the smaller problems, we have multiple answers and we need to combine them in some way to obtain the answer to the original problem.

A simple divide and conquer algorithm based on this approach is mergesort, developed by John von Neumann in 1945. We start by introducing the helper subroutine Merge and proving its correctness and runtime.

```
Merge(A[1,...,m], B[1,...,n])
  Result ← an empty array of length m + n
  i, j, k ← 1
  While k ≤ m + n:
      If i = m + 1:
          Result[k] ← B[j]
          j ← j + 1
      Else If j = n + 1:
          Result[k] ← A[i]
          i ← i + 1
      Else If A[i] < B[j]:
          Result[k] ← A[i]
          i ← i + 1
      Else:
          Result[k] ← B[j]
          j ← j + 1
      k ← k + 1
  Return Result
```

Theorem 43. *If A and B are sorted arrays of integers, then* Merge(A, B) *returns a sorted array containing all the elements of A and B combined. Furthermore,* Merge(A, B) *runs in time $O(m+n)$.*

Proof. We note that k keeps track of the next position that we have to fill in the merged array Result, while i keeps track of the index of the first element of A that we have yet to include in the merged list and j keeps track of the first element of B that we have yet to include in the merged list. Since A and B are sorted, the smallest element of A that we have yet to include in our list will always be $A[i]$, and similarly the smallest element of B we have yet to include will be $B[j]$.

Thus, each iteration of the while loop takes the smallest element we have yet to add to Result and copies it to Result. Once one of A or B runs out of elements to copy (i.e. i reaches $m + 1$ or j reaches $n + 1$), we take all the remaining elements from the other list. Thus, since at every step we are adding the smallest element we have yet to add, at the end of the while loop Result is a sorted list containing all the elements of A and B combined.

Each iteration of the while loop runs in constant time, since the values we are copying and incrementing can be expressed in a constant number of words, so since the while loop runs $m + n$ times, the total runtime is $\Theta(m + n)$. □

Now we can define MergeSort, which uses Merge.

```
MergeSort(A[1,...,n])
  If n < 10:
      InsertionSort(A)
      Return
  m ← ⌊n/2⌋
  MergeSort(A[1 : m])
  MergeSort(A[m + 1 : n])
  A ← Merge((A[1 : m]), A[m + 1 : n])
```

Theorem 44. *For every array A of integers, after calling* MergeSort*(A), A is sorted in increasing order.*

Proof. We prove this by induction on n.

- Base case: If $n < 10$, then MergeSort simply calls InsertionSort. By Theorem 16, InsertionSort correctly sorts A.
- Induction Hypothesis: Suppose that for every array B of length $k < n$, MergeSort(B) correctly sorts B.

- Induction Step: Suppose $n \geq 10$. Then $A[1:m]$ and $A[m+1:n]$ are both arrays of length less than n, so by our IH, MergeSort correctly sorts them. By Theorem 43 Merge correctly combines the sorted subarrays $A[1:m]$ and $A[m+1:n]$ into a sorted version of A.

□

Analyzing the runtime of MergeSort is more complicated. The call to InsertionSort takes constant time (since we only call it on an array with less than 10 elements), as does computing m (since n is a value which can fit in a single integer variable) and by Theorem 43 the call to Merge takes $\Theta(n)$, so the total time for everything outside of the recursive calls is $\Theta(n)$. On top of this, however, the algorithm makes 2 recursive calls. We can express the runtime of this algorithm using a *recurrence*:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n).$$

Implicitly we also have a base case $T(n) = \Theta(1)$ if $n < 10$, but since the base case of almost any recurrence describing an algorithm's runtime will be "constant time when the input size is constant" we generally omit this. In the next section we'll see how to use this recurrence to derive a non-recursive expression for the runtime of MergeSort.

8.4 Analyzing the Runtime of Divide and Conquer Algorithms

Using induction to prove a recurrence

Theorem 45. *The runtime of* MergeSort *is* $O(n \log n)$

Proof. We noted in the previous part that MergeSort satisfies the recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$, so we need to justify that this recurrence satisfies $T(n) = O(n \log n)$. Let $f(n)$ be the exact function hidden by the $\Theta(n)$ term in the recurrence for T. By the definition of big-Θ there exist constants c and n_0 such that $\forall\, n \geq n_0, f(n) \leq cn$. Let $d := \max_{1 \leq n < n_0} f(n)$. Then $\forall\, n \in \mathbb{Z}^+, f(n) \leq cn + d$, and thus $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$. Let $b := \max_{0 \leq n < 10} T(n)$ be the maximum number of steps taken by MergeSort in the base case of the recurrence.

We will prove by induction on i that for every positive integer i. $T(2^i) \leq (b + c + d)2^i \log 2^i$. Note that when for any $i > 0$, $\lfloor 2^i/2 \rfloor = \lceil 2^i/2 \rceil = 2^{i-1}$.

- Base Case: Suppose $1 \leq i \leq 3$. Then $2^i < 10$, so $T(2^i) \leq b \leq (b + c + d)2^i(\log_2 2^i)$.
- Induction Hypothesis: Suppose that for every $0 \leq j < i$, $T(2^j) \leq (b + c + d)2^j \log 2^j$.
- Induction Step: Suppose $i > 3$. Then we have that

$$\begin{aligned}
T(2^i) &\leq T(\lfloor 2^i/2 \rfloor) + T(\lceil 2^i/2 \rceil) + c2^i + d \\
&= 2T(2^{i-1}) + c2^i + d \\
&\leq 2(b + c + d)2^{i-1} \log_2(2^{i-1}) + c2^i + d && \text{(by our IH)} \\
&= (b + c + d)2^i(\log_2(2^i) - 1) + c2^i + d \\
&\leq (b + c + d)2^i(\log_2(2^i) - 1) + (b + c + d)2^i \\
&= (b + c + d)2^i \log_2 2^i
\end{aligned}$$

Thus, we have upper bounded $T(n)$ with a constant multiple of $n \log n$ when n is a power of 2. To extend this for n not a power of 2, we note that $T(n)$ is a non-decreasing function and we can upper bound any positive integer n with a power of 2 (specifically $2^{\lceil \log_2 n \rceil}$ which is less than $2n$, so

$$T(n) \leq T(2^{\lceil \log_2 n \rceil}) \leq (b+c+d)2^{\lceil \log_2 n \rceil} \log_2 2^{\lceil \log_2 n \rceil} \leq (b+c+d)(2n) \log_2(2n) \leq 4(b+c+d)n \log_2 n.$$

Thus, we can upper bound $T(n)$ with $4(b + c + d)n \log_2 n$ for every integer $n \geq 2$, so $T(n) \in O(n \log n)$. □

Some notes on this proof:

- We could prove that $T(n) = \Omega(n \log n)$ (and thus $T(n) = \Theta(n \log n)$ using a very similar technique. Instead of upper bounding $\Theta(n)$ with $cn + d$, we lower bound it with $cn - d$, for different constants c and d. Instead of upper bounding $T(n)$ with $T(2^{\lceil \log_2 n \rceil})$ we lower bound it with $T(2^{\lfloor \log_2 n \rfloor})$.
- The technique of upper bounding $T(n)$ with a sufficiently "nice" number (a power of 2 in this case) to avoid dealing with floors and ceilings without affecting the asymptotic behavior of the recurrence applies broadly when solving similar recurrences. Therefore, in the future if we have a recurrence like $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$ we will simply write it as $T(n) = 2T(n/2) + \Theta(n)$ since avoiding floors and ceilings is cleaner and does not affect the asymptotic runtime.
- This type of induction proof worked because we knew that the correct runtime was $\Theta(n \log n)$, but it doesn't help us figure out that the answer to the recurrence is $n \log n$. There are other techniques for doing that, like recursion trees, which you may have seen in previous courses. For the sake of time, we will not cover those techniques in this course. Instead, we will see a theorem in the next section which allows us to quickly obtain the correct answer for most recurrences which arise from divide and conquer algorithms.

8.5 The Master Theorem

Theorem 46. *Let $a, b, e \in \mathbb{Z}^+, c, d \in \mathbb{R}_{\geq 0}$ be constants, and let T be a recurrence of the form*

$$T(n) = aT(n/b) + \Theta(n^c \log^d n) \qquad \text{for } n \geq e$$
$$T(n) = \Theta(1) \qquad \text{for } 0 \leq n < e$$

Then,

1. *If $c < \log_b a$, $T(n) = \Theta(n^{\log_b a})$*
2. *If $c = \log_b a$, $T(n) = \Theta(n^c \log^{d+1} n)$*
3. *If $c > \log_b a$, $T(n) = \Theta(n^c \log^d n)$*

The proof of the master theorem is fairly involved and relies on simpler techniques for solving recurrences that we have intentionally avoided discussing in this course. If you are interested in details on how the master theorem is derived, Section 4.6 of CLRS is a good starting point.

The high level intuition of the theorem is that to solve a recurrence we need to compare the work done in the initial call (ignoring the recursive calls) with the work done across all of the base cases (a recurrence like mergesort's which repeatedly splits in two will have many different recursive calls reach the base case).

- Higher values of a correspond to more work done in the base cases, since a is the number of times that our function calls itself recursively, and more recursive calls corresponds to more calls to the base case.
- Higher values of b correspond to more work done in the initial call, since b determines how much smaller the subproblems are compared to our original problem, i.e. how fast we reach the base case.
- Higher values of c correspond to more work done in the initial call, since the n^c term is most impactful in the initial call and has no influence over the work at the base case.

There are three possible cases that the recurrence can fall into based on the relative values of a, b, and c.

- The first case is where the influence of a wins, and the time for the recurrence is dominated by the base cases. In this case, c and d don't show up in the runtime, the asymptotic time is just the number of base case calls
- The second case is where the competing influences of a vs. b and c are exactly equal, so the time spent on the base cases ($n^{\log_b a}$) is about the same as the time spent in the initial call ($n^c \log^d n$), and in fact this is the same amount of time spent at each of the $\log n$ levels of recursion. Thus, the total runtime is equal to the work of the initial call times $\log n$, $\Theta(n^c \log^{d+1} n)$. Note that in this case this is the same as $\Theta(n^{\log_b a} \log^{d+1} n)$.
- In the third case, the influence of b and c wins. The size of the subproblems shrinks fast enough, that the work of the initial call dominates the runtime, and so the asymptotic runtime is just the non-recursive work of the initial call.

In the recurrence for mergesort, $a = 2$, $b = 2$, $c = 1$, and $d = 0$ (and $e = 10$, but this doesn't actually impact the asymptotic runtime). Thus, plugging it into the theorem, we see that $c = \log_b a$, so mergesort falls into case 2 (the balanced case), and the asymptotic runtime is $\Theta(n \log n)$, as we proved in Theorem 45. The recurrence for binary search would be $T(n) = T(n/2) + \Theta(1)$ ($a = 1, b = 2, c = 0, d = 0$, which also falls into case 2, yielding the runtime of $\Theta(\log n)$ which we derived in Theorem 41.

Not all recurrences can be solved using the master theorem, but it works for most recurrences arising from natural divide and conquer algorithms, including all of the recurrences you will be expected to derive an asymptotic bound for in this course.

8.6 Comprehension Questions

1. What is the recurrence for Binary Search?
2. What is the runtime of FastSquareRoot? (Give a tight Big-O bound)
3. What is the recurrence for Mergesort?
4. Let $T(n) = 1$ for $n < 10$ and $T(n) = 3T(n/3) + \log_3 n$ for $n \geq 10$. Give a closed-form big-Θ bound for $T(n)$.
5. Let $T(n) = 1$ for $n < 3$ and $T(n) = 4T(n/2) + n \log_2 n$ for $n \geq 3$. Give a closed-form big-Θ bound for $T(n)$.

8.7 Exercises

1. Consider the following algorithm for finding the minimum element of an array:

```
FindMin(A[1, ..., n])
  If n = 1:
      Return A[1]
  B, C ← empty integer arrays
  For i from 1 to n:
      If i mod 3 ≠ 1:
          Add A[i] to the end of B
      If i mod 3 ≠ 2:
          Add A[i] to the end of C
  Return min(FindMin(B), FindMin(C))
```

   (a) Give a recurrence for the runtime of this algorithm and justify why the recurrence represents the algorithm's runtime (assume that adding an element to the end of an array is implemented efficiently and does not require resizing the array for every single addition).

   (b) Solve your recurrence to give a big-Θ expression for the runtime of FindMin.

2. For $2 \leq i \leq n-1$, we say that an element $A[i]$ of an array $A[1, \ldots n]$ is a *local minimum* if $A[i-1] \geq A[i] \leq A[i+1]$. For example, in the array

$$A = [10, 6, 1, 5, 6, 8, 12, 12, 15, 14, 8, 7, 9, 11, 12],$$

indices $3, 8$, and 12 correspond to local minimums

Give an algorithm which takes as input an array $A[1, \ldots, n]$, for some $n \geq 3$, where $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$, and finds the index of a local minimum in this array. Your algorithm should be as efficient as possible (asymptotically). [Hint: Why do the given conditions guarantee that a local minimum exists?]

3. Give a recurrence for the runtime of the following algorithm, then solve it to obtain a big-Θ expression:

```
TriMergeSort(A[1, ..., n])
  If n = 1:
      Return
  B_1 ← A[1, ..., ⌊n/3⌋]
  B_2 ← A[⌊n/3⌋ + 1, ..., ⌊2n/3⌋]
  B_3 ← A[⌊2n/3⌋ + 1, ..., n]
  TriMergeSort(B_1)
  TriMergeSort(B_2)
  TriMergeSort(B_3)
  i ← n
  While i > 0:
      j ← the index of whichever of B_1, B_2, B_3 has the largest last element
      A[i] ← the last element of B_j
      remove the last element of B_j
      i ← i − 1
```

4. Let $T(n) = 3T(n-1) + \Theta(n)$. Prove that $T(n) = O(3^n)$.

5. Let $T(n) = 2T(\sqrt{n}) + \Theta(1)$. Prove that $T(n) = O(\log n)$

6. Suppose you are given two sorted arrays of integers $A[1 \,..\, n]$ and $B[1 \,..\, n]$ such that all $2n$ elements of these arrays are distinct. Describe a fast algorithm (better than linear time) to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1 \,..\, 8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1 \,..\, 8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

your algorithm should return the integer 9. (Hint: What can you learn by comparing one element of A with one element of B?)

7. For an array $A = [a_1, \ldots, a_n]$, we define the k-step rotation of A as the array $[a_{k+1}, a_{k+2}, \ldots, a_{n-1}, a_n, a_1, a_2, \ldots, a_k]$ formed by taking the first k elements of A and moving them to the end of the array. Give an algorithm which takes as input an array of n distinct integers which is a k-step rotation of a *sorted* array (for an unknown value of k) and outputs the number of steps that the sorted array was rotated.

   For example, given the array

$$A = [18, 20, 37, 41, 82, 95, 105, 3, 7, 10, 11],$$

your algorithm should output 4, because this array is formed by rotating a sorted array 4 steps. Your algorithm should run in time $O(\log n)$. Briefly justify the correctness and runtime of your algorithm.

8. We say that an array $A[1, \ldots, n]$ of integers is a **mountain** if there exists some index i such that for every $1 \leq j < i$ we have $A[j] < A[j+1]$, and for every $i \leq j < n$ we have $A[j] > A[j+1]$. That is, a mountain consists of a strictly increasing sequence followed by a strictly decreasing sequence. For example,

$$[1, 3, 10, 7, 5, 4, -2, -7, -15]$$

is a mountain, but

$$[5, 4, -3, 8, 10]$$

is not. Give and analyze an algorithm which takes as input an array A which is a mountain, as well as an integer x and returns either

- An index j such that $A[j] = x$, or
- -1 if x is not in A.

You may assume that all of the elements of A are distinct. Your algorithm must run in $O(\log n)$ time.

9. Let $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ be a strictly increasing function (meaning that $\forall\, a, b \in \mathbb{Z}^+$ with $a < b$, $f(a) < f(b)$. Suppose that you can compute $f(x)$ in time $O(x)$ for any positive integer x (note that this implies that $f(x)$ has at most $O(x)$ digits). You are given a number $y \in \mathbb{Z}^+$, and you are told that $\exists\, z \in \mathbb{Z}^+$ such that $f(z) = y$. Give an $O(z \log z)$ time algorithm to find this z.

9 Running Times for Arithmetic

In Chapter 5 we stated that we can only assume that basic arithmetic operations run in constant time when the numbers we are working with are relatively small, i.e. when the number of digits is $O(\log n)$.

However, very large numbers occur with some frequency in computer science. Modern cryptography in particular is built on the inefficiency of performing certain operations on large numbers, and many cryptographic algorithms involve multiplying numbers with thousands of digits.

In this chapter we will examine in detail the runtime of performing arithmetic on large numbers. In Section 9.1 we will clarify the encoding that we are using for large numbers and look at the runtime for comparing them. In Section 9.2 we will analyze the runtime for addition (and subtraction). In the remaining three sections we will analyze three different algorithms for multiplication. First we will discuss the ordinary grade school algorithm in Section 9.3. Then we will consider a divide and conquer based approach in Section 9.4. Finally, we will look at another divide and conquer multiplication algorithm in Section 9.5 and discuss the fastest known algorithms for multiplication.

9.1 Comparison

Let's start by considering the runtime to compare two large integers. Previously we've been informal about how we encode numbers as strings, but for this chapter it will be important to be precise. We will view a non-negative integer as a string of decimal digits. Thus, our alphabet is $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and for each $n \in \mathbb{Z}^+$, $\langle n \rangle$ is the base-10 representation of n. Thus, a string $x = x_1 x_2 \dots x_n$ represents the number $\sum_{i=0}^{n-1} 10^i x_{n-i}$. For example, the string "123" represents $3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 = 3 + 20 + 100$. Note that the strings "0123" and "00123" (along with infinitely many strings with more leading zeros) also represent this number. When we write $\langle n \rangle$, however, we mean the representation with no leading 0s. While we will not bother ensuring that every algorithm in this chapter returns an encoding with no leading 0s, we can guarantee this in linear time with an extra step to remove leading 0s before returning if it is important. We represent the number zero with a single 0 (which we do not consider a leading 0).

All the algorithms we discuss in this chapter would work equally well in another base (e.g. binary instead of base-10) with the same asymptotic runtime, but we choose to work in base-10 since it is the most familiar to most of us.

To get a feel for algorithms working with large numbers, we begin by looking at the time to compare two non-negative integers. We can start by comparing the lengths of the numbers. If one of the numbers has fewer digits than the other, then that number is certainly smaller. If the numbers have the same number of digits, we compare their digits from first to last (most significant to least significant), looking for the first place where they differ. In this first position of difference, whichever number has the larger digit is the larger number. If the two numbers have the same digit at every position, then they are equal.

This approach to comparing large numbers is described in detail by the following algorithm LessThan.

```
LessThan(x, y)
    Remove any leading 0s from x and y
    If len(x) < len(y):
        Return True
    If len(y) < len(x):
        Return False
    For i from 1 to len(x):
        If x_i < y_i:
            Return True
        If y_i < x_i:
            Return False
    Return False
```

Theorem 47. *For all non-negative integers x and y, LessThan(x, y) runs in time $O(len(x)+len(y))$.*

Proof. Removing leading zeros from x and y takes time $O(\text{len}(x) + \text{len}(y))$. We note that $\text{len}(x)$ and $\text{len}(y)$ are values less than the length of the input, so they fit in integer RAM model variables and we can compare them in constant time. The for loop runs at most $\text{len}(x) \in O(\text{len}(x) + \text{len}(y))$ times. Inside the loop we are comparing single digits, which have value at most 9, so each iteration of the loop takes constant time. Thus, the overall runtime of the algorithm is $O(\text{len}(x) + \text{len}(y))$. □

We note that our upper bound of $O(\text{len}(x) + \text{len}(y))$ is tight (i.e. the runtime of LessThan(x, y) is $\Theta(\text{len}(x) + \text{len}(y))$) when x and y have the same length. In the worst case, x and y are equal, so the algorithm has to look at all of the digits of each number which takes $\Omega(\text{len}(x) + \text{len}(y))$ time. Furthermore, this runtime is optimal, *any* (deterministic) algorithm for this problem must look at all $\text{len}(x) + \text{len}(y)$ digits of x and y in the worst case, because if there is some digit that our algorithm doesn't look at, then the algorithm cannot distinguish the case where x and y are the same from the case where they differ at exactly that digit.

We note that if we have an algorithm for LessThan, we don't have to separately describe and analyze algorithms for related comparisons like greater than and equal to. Instead, we can *reduce* these operators to less than, as shown below:

```
GreaterThan(x, y)
    Return LessThan(y, x)
```

```
EqualTo(x, y)
    Return ¬LessThan(x, y) ∧ ¬LessThan(y,x)
```

9.2 Addition

Now let's look at the runtime to add two non-negative integers. Here again the best algorithm is one that you are already familiar with from grade school. We add pairs of digits, going from the right side of the inputs to the left. When the sum of two digits will not fit in a single digit, we use a "carry" to incorporate the tens digits of this sum into our addition of the pair of the digits to the left (which represent values ten times as large). This algorithm is specified by the pseudocode below.

```
Add(x, y)
    m ← len(x)
    n ← len(y)
    ℓ ← max(m, n)
    Pad x and y with 0s from the front until they both have length ℓ
    r ← a string of ℓ zeros
    carry ← 0
    For i from ℓ down to 1:
        r_i ← (x_i + y_i + carry) mod 10
        carry ← ⌊(x_i + y_i + carry)/10⌋
    If carry > 0:
        Prepend carry to the front of r
    Return r
```

Theorem 48. *For all non-negative integers x and y, Add(x, y) runs in time $O(m + n)$ where m is the length of x and n is the length of y.*

Proof. Once again, the length of x and y fit in RAM model integers, so we can compute their max in constant time. Padding x and y to have length ℓ and prepending carry at the end can be done in $O(m + n)$ time since $\ell \leq m + n$. Our for loop runs $O(m + n)$ times for the same reason, and each iteration of the for loop takes constant time since x_i, y_i and carry are all at most 9, so we can do arithmetic on them in constant time. Thus, the algorithm runs in time $O(m + n)$. □

As with LessThan, this runtime is the best that we can hope for. The optimality of Add follows from the fact that any algorithm for adding two integers, no matter how clever, must at least *write down the answer*. For non-negative integers x and y, $x + y$ has at least $\max(\text{len}(x), \text{len}(y)) \in \Omega(m + n)$

digits, so even if we compute each character of the output in constant time, just writing it down already takes $\Omega(m+n)$ time, and thus no algorithm for addition can be faster than this.

We note that we could create an $O(m+n)$ time algorithm SUBTRACT in very similar fashion. The details are left as an exercise to the student.

9.3 Multiplication

For the rest of the chapter, we will consider the problem of multiplying two large positive integers. Once again we will start by considering a typical grade school algorithm (I will present the algorithm that I learned as a child in the US. There are several other well-known multiplication algorithms with equivalent runtimes. Depending on where you went to school you may have learned a slightly different algorithm).

The idea of long multiplication is to break down the problem of multiplying an m digit number with an n digit one into mn single digit multiplications plus n additions. The algorithm multiplies each digit of y with all of x by going from right to left, multiplying digits and keeping track of a carry, similarly to the ADD algorithm. Then the algorithm uses the fact that

$$x \cdot y = \sum_{i=1}^{n} x \cdot y_i \cdot 10^{\text{len}(y)-i}$$

```
LONGMULTIPLY(x, y)
  If len(x) < len(y):
      Swap x and y
  m ← len(x)
  n ← len(y)
  r ← a string of m zeros
  For i from n down to 1:
      z ← a string of m zeros
      carry ← 0
      For j from m down to 1:
          z_j ← (x_j · y_i + carry)  mod 10
          carry ← ⌊(x_j · y_i + carry)/10⌋
      If carry > 0:
          Prepend carry to the front of z
      Append n − i zeros to the end of z
      r ← ADD(r, z)
  Return r
```

Theorem 49. *For all non-negative integers x and y,* LONGMULTIPLY(x, y) *runs in time $O(mn)$ where m is the length of x and n is the length of y.*

Proof. The work outside of the for loops can be done in $O(m)$ time. We note that the first two lines guarantee that $m \geq n$. The outer for loop runs n times.

The inner for loop runs m times and does $O(1)$ work per iteration (since the numbers that we are performing arithmetic with inside this loop are constant size). Thus the total work of the inner for loop is $O(m)$ each iteration. Outside of the for loop, creating z and carry, prepending carry to z and appending $n-i$ zeros can all be done in $O(m)$ time. By Theorem 48 we know that the call to ADD runs in time $O(m + \text{len}(r))$. The length of r is initially m and increases by at most 1 with each iteration of the outer for loop, so $\text{len}(r) \in O(m+n) = O(m)$ (since $m \geq n$). Thus, the runtime of the call to ADD is $O(m)$ and therefore the total runtime of each iteration of the outer for loop is $O(m)$.

Since the outer for loop runs $O(n)$ times and does $O(m)$ work each time, the total runtime for the loop is $O(mn)$ and thus this is the total runtime of the algorithm. □

Note that if y has constant length (e.g. if we're multiplying x by a constant, like 411), then the runtime of this algorithm is $O(m)$, which is the best that we can hope for since, as in the case of ADD, just writing down the answer to this multiplication problem takes $\Omega(m)$ time.

If the two numbers have about the same length, however (i.e. if $m = n$), then this algorithm runs in $O(n^2)$, which, perhaps surprisingly, is far from optimal. In the next section we'll look at a new approach for multiplication using a divide and conquer strategy.

9.4 Multiplication with Divide and Conquer

Our divide and conquer multiplication algorithm is based on splitting each of the numbers in half and multiplying the halves. In particular, we note that if x and y are n-digit numbers for some even integer n, we can express x and y as

$$x = 10^{n/2}a + b, \text{ and} \qquad y = 10^{n/2}c + d,$$

for some non-negative integers a, b, c, d.

Then,

$$x \cdot y = (10^{n/2}a + b)(10^{n/2}c + d) = 10^n ac + 10^{n/2}ad + 10^{n/2}bc + bd.$$

Multiplying by powers of 10 is very easy, since we can just add 0s to the end of the number. Thus, if we can compute the four smaller products ac, ad, bc, bd, we can compute xy using only additions. The full pseudocode of the algorithm is given below:

```
DC-MULTIPLY(x, y)
  n ← max(len(x),len(y))
  Pad x or y with leading zeros so that they both have length n
  If n = 1:
      Return x_1 · y_1
  a ← x_1 x_2 ... x_⌊n/2⌋
  b ← x_⌊n/2⌋+1 ... x_{n-1} x_n
  c ← y_1 y_2 ... y_⌊n/2⌋
  d ← y_⌊n/2⌋+1 ... y_{n-1} y_n
  ac ← DC-MULTIPLY(a, c)
  ad ← DC-MULTIPLY(a, d)
  bc ← DC-MULTIPLY(b, c)
  bd ← DC-MULTIPLY(b, d)
  Append 2(n − ⌊n/2⌋) zeros to the end of ac
  Append n − ⌊n/2⌋ zeros to the end of ad
  Append n − ⌊n/2⌋ zeros to the end of bc
  Return ADD(ADD(ac, ad), ADD(bc, bd))
```

Theorem 50. *If x and y are n-digit non-negative integers then* DC-MULTIPLY(x, y) *runs in time $O(n^2)$.*

Proof. All of the steps outside of the recursion can be performed in $O(n)$ time. The algorithm performs 4 recursive calls to DC-MULTIPLY, and each recursive call is on inputs of length $n/2$ (ignoring floors and ceilings). Thus, the runtime $T(n)$ of DC-MULTIPLY satisfies the recurrence:

$$T(n) = 4T(n/2) + O(n).$$

By the master theorem, this recurrence solves to $T(n) \in \Theta(n^2)$. □

If you are paying attention, you'll note that the runtime bound that we proved for DC-MULTIPLY in Theorem 50 is the same as the bound that we proved for LONGMULTIPLY in Theorem 49 (specialized to the case where $m = n$). Divide and conquer didn't help at all! Was this whole algorithmic technique a scam?

It turns out the divide and conquer technique on its own is not enough to speed up multiplication. We need one more clever idea, which we'll see in the next section.

9.5 Karatsuba's Algorithm

In the mid 20th century it was commonly believed that no multiplication algorithm could do significantly better than LongMultiply. In 1960, prominent mathematician Andrey Kolmogorov held a seminar where he formally conjectured that multiplying two n-digit integers required $\Omega(n^2)$ time. One of the students attending the seminar was Anatoly Karatsuba who, less than a week later, disproved the conjecture by presenting the algorithm we will discuss in this section.

The basic structure of KaratsubaMultiply is the same as DC-Multiply. We divide x into two halves, a and b, and divide y into two halves c and d. Karatsuba's core insight is that in order to compute the product

$$x \cdot y = (10^{n/2}a + b)(10^{n/2}c + d) = 10^n ac + 10^{n/2}ad + 10^{n/2}bc + bd$$

we do not need to perform four multiplications. We can rewrite

$$10^n ac + 10^{n/2}ad + 10^{n/2}bc + bd = 10^n ac + 10^{n/2}(ad + bc) + bd = 10^n ac + 10^{n/2}((a+b)(c+d) - ac - bd) + bd.$$

Thus, instead of computing the two products ad and bc, we can compute the single product $(a+b)(c+d)$ and then subtract out the ac and bd that we already needed to compute in order to get $ad + bc$. This trick allows us to reduce the 4 recursive calls of DC-Multiply to 3, at the cost of a few extra addition and subtractions. This turns out to make a big difference in the runtime of the algorithm.

```
KaratsubaMultiply(x, y)
  n ← max(len(x),len(y))
  Pad x or y with leading zeros so that they both have length n
  If n = 1:
      Return x_1 · y_1
  a ← x_1 x_2 ... x_⌊n/2⌋
  b ← x_⌊n/2⌋+1 ... x_{n-1} x_n
  c ← y_1 y_2 ... y_⌊n/2⌋
  d ← y_⌊n/2⌋+1 ... y_{n-1} y_n
  ac ← KaratsubaMultiply(a, c)
  bd ← KaratsubaMultiply(b, d)
  // ad_bc ← (a + b)(c + d) − ac − bd
  ad_bc ← Subtract(Subtract(KaratsubaMultiply(Add(a, b), Add(c, d)), ac), bd)
  Append 2(n − ⌊n/2⌋) zeros to the end of ac
  Append n − ⌊n/2⌋ zeros to the end of ad_bc
  Return Add(Add(ac, ad_bc), bd)
```

Theorem 51. *If x and y are n-digit non-negative integers then* KaratsubaMultiply(x, y) *runs in time $O(n^{\log_2 3}) \approx O(n^{1.58})$.*

Proof. Once again, all of the steps outside of the recursion can be performed in $O(n)$ time (all of the numbers we are passing to Add or Subtract have $O(n)$ digits). There are three recursive calls, and each recursive call is on an input of size $n/2$ (we are safely ignoring floors and ceilings as well as the potential 1 extra digit produced by the calls to Add(a, b) and Add(c, d)). Thus, the runtime $T(n)$ of KaratsubaMultiply satisfies the recurrence:

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

By the master theorem, this recurrence solves to $T(n) = \Theta(n^{\log_2 3})$, substantially improving on the runtime of LongMultiply. □

Karatsuba's algorithm is not in fact the fastest known algorithm for multiplication.

- Only a few years after Karatsuba's discovery, Andrei Toom and Stephen Cook developed a generalization of Karatsuba's algorithm which splits a number into k parts to achieve a runtime of $O(n^{\log(2k-1)/\log k})$ for any integer $k \geq 2$ (the hidden constant in the big-O expression becomes worse rapidly as k grows, however).

- In 1971, Arnold Schönhage and Volker Strassen developed an algorithm based on the *Fast Fourier Transform* (which is beyond the scope of this course) that multiplies two n digit non-negative integers in $O(n \log n \log \log n)$ time. This was the fastest multiplication algorithm for decades.
- In 2019, David Harvey and Joris van der Hoeven developed an $O(n \log n)$ time algorithm for multiplying two n-digit integers. This is conjectured to be the optimal runtime for multiplication, but we still don't have a better lower bound than $\Omega(n)$ (which comes from the time needed to write down the output).

In future chapters, we will generally avoid worrying about the encodings of integers, and instead rely on the algorithms that we discussed in this chapter. Thus, when we write $x+y$, you should assume that if x and y are too large to fit in integer variables, we are calling ADD and this operation takes $\Theta(\text{len}(x)+\text{len}(y))$ time. When we write $x \cdot y$ you may assume that we can perform the multiplication in time $O(\text{len}(x)\text{len}(y))$ using LONGMULTIPLY or in time $O(n \log n)$, where $n = \max(\text{len}(x),\text{len}(y))$, using the Harvey-van-der-Hoeven algorithm.

9.6 Comprehension Questions

1. Give a recurrence for the following algorithm:

```
DC-ADD(x, y)
  n ← max(len(x),len(y))
  Pad x or y with leading zeros so that they both have length n
  If n = 1:
      Return x_1 + y_1
  a ← x_1 x_2 ... x_⌊n/2⌋
  b ← x_⌊n/2⌋+1 ... x_{n-1} x_n
  c ← y_1 y_2 ... y_⌊n/2⌋
  d ← y_⌊n/2⌋+1 ... y_{n-1} y_n
  ac ← DC-ADD(a, c)
  bd ← DC-ADD(b, d)
  Append n − ⌊n/2⌋ zeros to the end of ac
  Return ADD(ac, bd)
```

2. Suppose that we want to multiply an n digit number by 411. Using the best known algorithm, what is the fastest asymptotic runtime that we can achieve for this multiplication, as a function of n?
3. True or false? No algorithm for adding two arbitrary n-digit positive integers can run in $o(n)$ time in the worst case.
4. True or false? It has been proven that every algorithm for multiplying two arbitrary n-digit positive integers requires $\Omega(n \log n)$ time in the worst case.

9.7 Exercises

1. Give an $O(m+n)$ time algorithm SUBTRACT(x, y) which computes $x - y$ for large integers x and y. Assume that we have added "$-$" to our alphabet and can represent negative numbers.
2. A *complex number* is a number of the form $a+bi$, where $a, b \in \mathbb{R}$ and $i := \sqrt{-1}$ is the imaginary unit. The product of two complex numbers $a+bi$ and $c+di$ is

$$(a+bi)(c+di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i.$$

Suppose that you have access to subroutines to add and multiply real numbers. Clearly we can calculate this product using 4 real number multiplications: ac, ad, bc, and bd. Give an algorithm to calculate $(a+bi)(c+di)$ using only *three* real number multiplications.

3. Consider the following algorithm whose correctness we proved back in Theorem 15:

```
FACT(x):
  If x ≤ 1:
      Return 1
  Return x * FACT(x − 1)
```

   (a) How many multiplications does FACT perform in the worst case, in terms of $n := \text{len}(x)$?

   (b) Prove that any algorithm for computing $x!$ must have a runtime which is $\Omega(n10^n)$.

   (c) What is the asymptotic runtime of FACT, in terms of n, assuming that the multiplications are done with LONGMULTIPLY?

4. Hexadecimal (also known as base-16) is a system for representing numbers where in addition to the ten digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, there are also digits A, B, C, D, E, and F (corresponding to the decimal numbers $10, 11, 12, 13, 14$, and 15, respectively), and place values are powers of 16, instead of powers of 10. For example, the hexadecimal value $3AF$ when converted to decimal would become $3 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 = 943$. Design an algorithm which takes as input an n-digit number in hexadecimal (represented as a string) and outputs the same number in base-10. Your algorithm's runtime should be better than $\Theta(n^2)$.

5. Recall that a degree d polynomial in x is an expression of the form $c_0 + c_1x + c_2x^2 + \cdots + c_dx^d$, where $c_0, \ldots, c_d$ are called the coefficients of the polynomial. Give an algorithm which takes as input two degree d polynomials in x with integer coefficients and outputs their product. For example, given the polynomials $2x^2 + 3x - 4$ and $-3x^2 + 5x - 2$ your algorithm should output $8 - 26x + 23x^2 + x^3 - 6x^4$. Assume that you can perform addition and multiplication of integers in constant time. Your algorithm should run in time $O(d^{1.99})$ or better.

10 Recursive Backtracking

In the previous two chapters we saw efficient divide and conquer algorithms for several problems, including sorting an array, searching a sorted array, and multiplying large numbers. In this chapter, we will explore divide and conquer algorithms based on *recursive backtracking*, a general technique that we can apply whenever the solution to a problem can be generated by making a bunch of small choices. Unlike the divide and conquer algorithms we've seen previously, the ones in this chapter will not be very efficient. However, they will prepare us to learn about efficient dynamic programming algorithms in Chapter 11.

10.1 Subset Sum

In the subset sum problem, we are given a set $S \subseteq \mathbb{Z}^+$ of positive integers and a non-negative integer $k \in \mathbb{Z}_{\geq 0}$, and want to determine whether or not there exists a set $T \subseteq S$ with $\sum_{x \in T} x = k$ (i.e. whether there is a subset of S whose elements sum to exactly k). For example, if $S = \{2, 3, 7, 11, 19\}$ and $k = 24$, we should return True, because $\{2, 3, 19\} \subseteq S$ and $2 + 3 + 19 = 24$.

How can we solve this problem with a divide and conquer approach? (i.e. how can we convert the problem to a smaller version of itself?). One natural idea is to choose a first element x to include in our subset, and then recursively choose the rest of the elements so that they sum to $k - x$. We give a recursive algorithm based on this approach below:

```
SubsetSumSolver(S, k)
    Remove from S all elements which are greater than k
    If S = ∅:
        If k = 0:
            Return True
        Return False
    For each x ∈ S:
        If SubsetSumSolver(S \ {x}, k − x):
            Return True
    Return False
```

Theorem 52. *For every finite $S \subset \mathbb{Z}^+$ and $k \in \mathbb{Z}_{\geq 0}$* SubsetSumSolver$(S, k)$ *returns whether or not there exists $T \subseteq S$ with $\sum_{x \in T} x = k$. Furthermore* SubsetSumSolver *runs in time $\Omega(n!)$, where $n = |S|$.*

Proof. We prove the correctness of SubsetSumSolver by induction on k.

- Base case: If $k = 0$, then the first line will remove all elements from S, so S will be the empty set and the algorithm will immediately return True, which is correct, since every set has a subset summing to 0, namely the empty set.
- Induction hypothesis: For every $0 \leq k' < k$ and every finite $S' \subset \mathbb{Z}^+$, SubsetSumSolver(S', k') returns whether or not there exists $T \subseteq S'$ with $\sum_{x \in T} x = k'$.
- Suppose $k > 0$. We first note that removing the elements of S which are greater than k cannot change whether or not S has a subset summing to k. Since all elements of S are positive, any subset which includes an element $x > k$ must have a sum of at least x, and thus cannot have a sum of k.

  If S is empty after these removals, then it cannot have a subset summing to k, since $k \neq 0$, so the algorithm correctly returns False. Otherwise, we consider each element of S for inclusion in our subset. We note that if $T \subseteq S$ sums to k and y is an element of T, then $T \setminus \{y\}$ is a subset of $S \setminus \{y\}$ which sums to $k - y$. Since $y > 0$, $k - y < k$. Thus, by our IH, when our loop reaches y, SubsetSumSolver$(S \setminus \{y\}, k - y)$ will return True, and so will SubsetSumSolver(S, k).

  Conversely, if SubsetSumSolver$(S \setminus \{y\}, k - y)$ returns true for some y, then by our IH we know that there exists some $T' \subseteq S \setminus \{y\}$ with $\sum_{x \in T'} = k - y$. Then $T' \cup \{y\}$ is a subset of S summing to k, so when the algorithm returns True, it is correct.

We have shown that if it has not returned beforehand the algorithm returns True in the loop if and only if S has a subset summing to k. Therefore, if we reach the end of the loop and none of the recursive calls have returned True, then S does not have a subset summing to k and the algorithm correctly returns False.

Now we analyze the runtime. We only need to prove a lower bound, so we can choose a particular class of inputs to focus on (one that we think represents a worst-case for the algorithm). Suppose that $k = 1 + \sum_{x \in S} x$. Then there is no subset of S summing to k and we will never discard elements for being greater than k, so every subproblem will make a recursive call for every element in the set.

Each recursive call decreases the size of our set by 1, so in total we make:

- 1 call with a set of size n
- n calls with a set of size $n-1$
- $n(n-1)$ calls with a set of size $n-2$
- ...
- $\prod_{j=0}^{i-1}(n-j)$ calls with a set of size $n-i$

In particular, this means that we make $n!$ calls to our base case where the size of our set is 0. Thus, we can lower bound the total runtime with $n!$.

□

A runtime of $\Omega(n!)$ is extremely inefficient. Let's see if we can do a little bit better. SubsetSumSolver is slow in part because it considers the same subset many times. For example, the algorithm considers adding 1 first then 2 separately from adding 2 first, then 1. We can avoid this redundancy by applying an arbitrary ordering to our set, and only considering adding elements in this fixed order. One natural way to do this is to represent our set S as an array A. Since we are now considering the elements in a fixed order, the choice that we make is not "which element should we include next" but "should we include $A[i]$ in the subset?" Including $A[i]$ should only be an option if $A[i] \leq k$, otherwise we're forced to skip it.

Since we are trying to be efficient here, we want to avoid overhead from modifying and copying an array, so instead, we will keep the same array A (which we implicitly pass by reference) and pass our recursive calls an extra parameter i representing the index of the array to start at. Based on our discussion in the previous paragraph, we can write a recursive function $SS(A, k, i)$ which determines whether or not $A[i:n]$ has a subset summing to k:

$$SS(A,k,i) = \begin{cases} True & \text{if } k = 0 \\ False & \text{if } i > n \text{ and } k > 0 \\ SS(A,k,i+1) & \text{if } i \leq n, k > 0 \text{ and } A[i] > k \\ SS(A,k-A[i],i+1) \vee SS(A,k,i+1) & \text{otherwise} \end{cases}$$

This recursive function immediately translates into a recursive algorithm

```
SecondSubsetSumSolver(A[1,...,n], k, i = 1)
  If k = 0:
      Return True
  If i > n:
      Return False
  If A[i] > k:
      Return SecondSubsetSumSolver(A, k, i + 1)
  Return SecondSubsetSumSolver(A, k − A[i], i + 1) ∨ SecondSubsetSumSolver(A, k, i + 1)
```

Theorem 53. *For every $n, k \in \mathbb{Z}_{\geq 0}$ and every array A of n positive integers,* SecondSubsetSumSolver(A, k) *returns whether or not there exists a subset T of the elements of A with $\sum_{x \in T} x = k$. Furthermore* SecondSubsetSumSolver *runs in time $\Theta(2^n)$.*

Proof. We will sketch the correctness proof rather than using a full induction proof, since the argument is similar to what we already showed in Theorem 52. We note that if the algorithm returns True, then it found a subset of the elements of A which sum to k (we can trace up the call stack from the call to SecondSubsetSumSolver which returned True and see where the algorithm recursed on SecondSubsetSumSolver$(A, k - A[i], i + 1)$).

On the other hand, if S has a subset T summing to k, then our algorithm will find it through the chain of function calls where it recurses on SecondSubsetSumSolver$(A, k-A[i], i+1)$ (corresponding to adding $A[i]$ to our subset) whenever $A[i] \in T$ and SecondSubsetSumSolver$(A, k, i + 1)$ whenever $A[i] \notin T$ (corresponding to omitting $A[i]$ from our subset). The algorithm will not know what T is, but we are trying all options through recursive brute force, so one chain of recursive calls will happen to choose exactly the elements in T.

Now we justify the runtime. SecondSubsetSumSolver does $\Theta(1)$ work outside of its recursive calls, and makes 2 recursive calls to inputs of size $n - 1$. In the worst case (e.g. if $k > \sum_{i=1}^{n} A[i]$, the algorithm will not return until the base case where $i = n + 1$, so the runtime satisfies the recurrence $T(n) = 2T(n - 1) + \Theta(1)$.

This is not a recurrence that we can directly apply the master theorem to, since we are not dividing the input size by a constant. However, we can still use the theorem if we are a bit clever. Define a new function T' whose domain is powers of 2 by $T'(x) := T(\log_2 x)$. Then, by our recurrence for T, T' satisfies the recurrence

$$T'(x) = 2T((\log_2 x) - 1) + \Theta(1) = 2T'(x/2) + \Theta(1),$$

since $\log_2(x/2) = \log_2(x) - 1$. The master theorem tells us that this recurrence solves to $T'(x) \in \Theta(x)$, and thus $T(\log_2 x) \in \Theta(x)$. Plugging in n for $\log_2 x$ we get that $T(n) \in \Theta(2^n)$. □

SecondSubsetSumSolver is much faster than SubsetSumSolver, but the runtime is still exponential in n. Next chapter when we talk about dynamic programming, we will see a way to further improve the runtime of this algorithm.

10.2 Longest Increasing Subsequence

Let's see another example of a problem that we can solve through recursive backtracking. First we need to define the notion of a subsequence. For a sequence/string/array $x = x_1, x_2, \ldots, x_n$, a *subsequence* is a sequence $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where $1 \leq i_1 < i_2 < i_3 < \cdots < i_k \leq n$. That is, a subsequence of x is a list of elements from x in the same order that they appear in x. Unlike a *substring*, however, the elements do not have to be consecutive in x, the subsequence can skip over elements. For example, 1,3,7 is a subsequence of 1,2,3,5,6,7,9.

Now we can define the problem we are interested in. In the longest increasing subsequence problem we are given as input an array A and we want to find the length of the longest subsequence of A where all the elements are in increasing order. For example if $A = [10, 5, 3, 4, 1, 6, 12, 9, 20, 2, 3, 4]$ then a longest increasing subsequence is $3, 4, 6, 12, 20$, which has length 5.

As we did in SecondSubsetSumSolver, we can view a solution to longest increasing subsequence as constructed from a sequence of binary choices of "include $A[i]$ or skip it". This suggests a recursive structure where we relate the length of the longest increasing subsequence of $A[i, \ldots, n]$ to the length of the longest increasing subsequence of $A[i + 1, \ldots, n]$.

We have to be a bit more clever here, however, because it's not immediately clear how knowing the length of the longest increasing subsequence of $A[i + 1, \ldots, n]$ helps us find the length of the longest increasing subsequence of $A[i, \ldots, n]$.

Using the example array $A = [10, 5, 3, 4, 1, 6, 12, 9, 20, 2, 3, 4]$ from above, suppose that we are at index $i = 3$ with $A[i] = 3$ and a recursive call tells us that the longest increasing subsequence of $A[4 : n]$ has length 4. This tells us that the length of longest increasing subsequence of $A[3 : n]$ is either 4 (if the best subsequence for $A[3 : n]$ is the same as the best one for $A[4 : n]$) or 5 (if we can add 3 to the beginning of a longest increasing subsequence for $A[4 : n]$), but it's not enough information to tell whether 4 or 5 is the right answer.

Even if our recursive call returned an actual maximal subsequence, we still wouldn't have enough information to solve the problem. For example, $1, 2, 3, 4$ is a longest increasing subsequence of $A[4 : n]$, and we clearly cannot add 3 to the beginning of this sequence. Nevertheless, the longest increasing subsequence of $A[3 : n]$ does have length 5, because we can add 3 to the beginning of an alternative longest increasing subsequence for $A[4 : n]$, such as $4, 6, 12, 20$.

What we need is some way to tell our recursive call when we have already chosen $A[i]$ for our subsequence, and all subsequent elements must be larger than this. In the subset sum problem, we were able to accomplish the goal of letting the recursive call know about our previous choices by modifying the k parameter. In this case, our problem doesn't have a second integer parameter, but that doesn't mean we can't add one! By introducing a second parameter representing "the previous element that we picked", we can give a recurrence for longest increasing subsequence.

Let $LIS(i, j)$ denote the length of the longest increasing subsequence of $A[i, \ldots, n]$ where every element is larger than $A[j]$. We give the following recurrence:

$$LIS(i,j) = \begin{cases} 0 & \text{if } i > n \\ LIS(i+1,j) & \text{if } i \leq n \text{ and } A[i] \leq A[j] \\ \max\{LIS(i+1,j), 1 + LIS(i+1,i)\} & \text{otherwise} \end{cases}$$

Here our recurrence represents the choice of whether or not to include $A[i]$ in our subsequence. If $A[i] \leq A[j]$, we have no choice, we have to skip $A[i]$. If $A[i] > A[j]$, we can include it, in which case we increase the length of our sequence by 1, but change our lower bound on future elements to $A[i]$, or we can skip it. We choose whichever option gives a longer subsequence. We give the following pseudocode for LIS:

```
LIS-Solver(A[1, ..., n], i, j):
  If i > n:
      Return 0
  If A[i] ≤ A[j]:
      Return LIS-Solver(A, i + 1, j)
  Return max(LIS-Solver(A, i + 1, j), 1 + LIS-Solver(A, i + 1, i))
```

Note that the pseudocode directly translates the recurrence. This time there is not really a natural default value for the extra parameter j that we have added, so we need to specify how to use our recursive algorithm for solving our modified version of the function (where we require all elements of our sequence to be larger than $A[j]$) in order to solve the original problem.

In this case, to solve the original problem, we need to compute $\max_{1 \leq i \leq n}(1 + \text{LIS-Solver}(A, i + 1, i))$. That is, we take the max, over all choices of first element, of the longest increasing subsequence we can get starting with that element.

Once again, this algorithm runs in exponential time. Just like SecondSubsetSumSolver, LIS satisfies the recurrence $T(n) = 2T(n-1) + \Theta(1)$ in the worst case, so the runtime is $\Theta(2^n)$ by a similar analysis.

Note that if we wanted to we could easily modify our algorithm to return the actual longest subsequence (instead of just the length). We would simply change our base case to return an empty sequence and instead of returning $\max(\text{LIS-Solver}(A, i+1, j), 1+\text{LIS-Solver}(A, i+1, i))$ we would return the longer of the sequence returned by $\text{LIS-Solver}(A, i + 1, j)$ and the sequence returned by $\text{LIS-Solver}(A, i + 1, i))$ with $A[i]$ inserted at the front.

10.3 Designing Recursive Backtracking Algorithms

The general approach to designing recursive backtracking algorithms is to solve a problem through *recursive brute force*. We need to:

1. Determine how to construct a solution to the problem by making a sequence of small choices (i.e. figure out the recursive structure of solutions)

2. Determine what information our algorithm needs to have about the previous choices in order to make the next choice in the sequence validly. Often this will require modifying the original problem to add an extra parameter.
    - For subset sum the information about our previous choices was communicated entirely by changing k
    - For longest increasing subsequence we had to pass along the extra parameter j keeping track of the last element we added to the sequence
    - Keeping this summary as small as possible will help us convert our recursive backtracking algorithm to an efficient DP algorithm later. This is why we chose to have our longest increasing subsequence algorithm pass along only the previous element, not all of the elements chosen so far.
3. Clearly state the version of the problem that you are solving. For example, for longest increasing subsequence we defined our problems by saying "Let $LIS(i, j)$ denote the length of the longest increasing subsequence of $A[i, \ldots, n]$ where every element is larger than $A[j]$."
4. Construct a recursive function describing the solution to your chosen modification of the problem in terms of the solutions to smaller subproblems.
    - Typically this recursive function will have the structure:
        - Try all possible options for the first choice in our sequence of choices
        - For each possible option, recursively make the rest of the choices in a way that optimizes our objective
        - Combine the result of the recursive calls in order to determine the best first choice.
5. (Optional) Write recursive pseudocode based on your recurrence. Often the pseudocode is almost word-for-word the same as the recurrence, so this step is not strictly necessary. Sometimes there may be important implementation details to discuss, however, and if you aren't pressed for time then it never hurts to explicitly give pseudocode.
6. Describe how to use your recursive function to solve the original problem
    - That is, describe what value(s) to use for any additional parameters you have added.
    - e.g. for longest increasing subsequence we took $\max_{1 \le i \le n}(1 + \text{LIS-SOLVER}(A, i + 1, i))$.

10.4 Longest Common Subsequence

One simple problem about subsequences with a variety of practical applications is *longest common subsequence*. In this problem we are given two arrays $A[1, \ldots, m]$ and $B[1, \ldots, n]$ and we want to find the longest sequence which is a subsequence of both A and B. For example, if $A = [1, 7, 3, 12, 15, 2, 2, 1, 7]$ and $B = [3, 1, 12, 14, 9, 2, 5, 1, 7, 16]$ then a longest common subsequence is $[3, 12, 2, 1, 7]$, which has length 5 ($[1, 12, 2, 1, 7]$ is also a longest common subsequence).

Solving longest common subsequence is useful for comparing two versions of a file and determining what has changed between them (e.g. in a version control system like git). The longest common subsequence problem also arises in biology, when trying to compare two DNA sequences for common ancestry.

Let's apply the recursive backtracking techniques that we have discussed to solving longest common subsequence. For simplicity, we will focus on finding the *length* of the longest common subsequence. We need to figure out how to view constructing a longest common subsequence as making a series of small choices.

A natural approach is to say that each choice is one element of the common subsequence. So our recursive function should consider all possible choices for the first element of the subsequence and then recursively compute which choice maximizes the length of the rest of the subsequence.

Every pair of a character from A and a matching character from B (not necessarily at the same index) forms a potential starting point for our common subsequence. In order to make sure that the remaining elements that we choose all occur after the first elements we pick, our recursive call needs to pass along the indices of the elements that we picked in both A and B.

Thus, the version of the problem that we are solving involves two extra index parameters, which we can call i and j. Let $LCS(i, j)$ denote the length of the longest common subsequence of $A[i : m]$ and $B[j : n]$. To simplify the description of our recurrence, we define an auxiliary function I. Let $I(i, j) \subseteq \{i, i+1, \ldots, m\} \times \{j, j+1, \ldots, n\}$ be defined by

$$I(i, j) := \{(i', j') \mid i \leq i' \leq m, j \leq j' \leq n \text{ and } A[i'] = B[j']\}.$$

That is, $I(i, j)$ is the set of all pairs of indices (i', j') in our range of valid indices where $A[i'] = B[j']$. Then LCS satisfies the following recurrence:

$$LCS(i, j) = \begin{cases} 0 & \text{if } I(i, j) = \emptyset \\ \max_{(i',j') \in I(i,j)} 1 + LCS(i' + 1, j' + 1) & \text{otherwise} \end{cases}$$

This recurrence yields the following recursive algorithm:

```
LCS-Solver(A[1,...,m], B[1,...,n], i = 1, j = 1):
  I ← ∅
  For i' from i to m:
      For j' from j to n:
          If A[i'] = B[j']
              Add (i', j') to I
  If I = ∅:
      Return 0
  Return 1 + max_{(i',j')∈I} LCS-Solver(A, B, i' + 1, j' + 1)
```

The solution to our original problem is simply LCS-Solver(A, B) (which sets i and j to 1).

In the worst case where all the elements of A and B are identical, this algorithm again takes exponential time (the analysis is a little more involved than the previous two problems, and not particularly important). Next chapter we will see how to convert all of the exponential time recursive algorithms we have seen in this chapter into polynomial time dynamic programming algorithms.

10.5 Comprehension Questions

1. What does SubsetSumSolver$(\{1, 2, 3, 4, 5\}, 13)$ return?
2. True or false? SecondSubsetSumSolver runs in time $O(n^2)$.
3. What is a longest increasing subsequence of $A = [3, 4, 1, 7, 12, 15, 8]$?
4. True or false? If $LIS(i, j)$ is the length of the longest increasing subsequence of $A[i, \ldots, n]$ where every element is larger than $A[j]$ then LIS satisfies the recurrence

$$LIS(i, j) = \begin{cases} 0 & \text{if } i > n \\ \max\{LIS(i + 1, j), 1 + LIS(i + 1, i)\} & \text{otherwise} \end{cases}$$

5. What is a longest common subsequence of $A = [2, 3, 8, 7, 1]$ and $B = [1, 7, 8, 7, 3, 2, 1]$?

10.6 Exercises

1. Give a recursive backtracking algorithm that takes as input an array $A[1, \ldots, n]$ of integers and computes the length of the longest *decreasing* subsequence of A. You should justify the correctness of your algorithm, but you do not need to analyze the runtime.

2. We define a *valley* as a sequence of integers $v_1, \ldots, v_k$ with the following three properties:

   - k is odd
   - For every $1 \leq i \leq (k-1)/2$, we have that $v_i \geq v_{i+1}$
   - For every $(k+1)/2 \leq i \leq k-1$, we have that $v_i \leq v_{i+1}$.

   For example, the sequence $5, 4, 4, 7, 10$ would be a valley, but the sequence $7, 6, 5, 8, 9, 11, 13$ would not be. Give a recursive backtracking algorithm which takes as input an array $A[1, \ldots, n]$ of positive integers and returns the longest subsequence of A which is a valley. You should justify the correctness of your algorithm, but you do not need to analyze the runtime.

3. You are planning a road trip between cities A and B. Fortunately, they are connected by a long straight highway. Unfortunately, you are going to have to stop for fuel several times, and all of the gas stations on this highway use a very unusual pricing system. Gas is sold by the tank, meaning that you pay the same fixed amount to refill your tank, whether that takes one gallon or twenty.

   Suppose that you are given d, the distance in miles that your car can travel on a full tank of gas, as well as two arrays $G[1, \ldots, n]$ and $P[1, \ldots, n]$, where $G[i]$ is the location of the ith gas station, given as a distance in miles from your starting point, and $P[i]$ is the price (in dollars) to fill your tank at the ith gas station. Assume that your trip begins at the first gas station (so $G[1] = 0$) with an empty tank of gas, and ends at the nth gas station (where you do not need to refuel). Assume that it is possible to complete the trip (i.e. there is no stretch of d miles with no gas stations). Give a recursive backtracking algorithm to calculate the minimum amount that you can pay for gas while making this trip. You should justify the correctness of your algorithm, but you do not need to analyze the runtime.

4. Susan is going door to door in her neighborhood selling boxes of cookies. Susan's cookies come in two different kinds. She has x boxes of chocolate chip cookies and y boxes of oatmeal raisin cookies. Susan's neighborhood consists of n houses arranged in a straight line. Each household is willing to buy at most one box of cookies, and they each value the two kinds of cookies differently. Fortunately, Susan has done extensive market research, and has compiled two arrays, $A[1, \ldots, n]$ and $B[1, \ldots n]$, where $A[i]$ is the price (in cents) that the ith house will pay for a box of chocolate chip cookies, $B[i]$ is the price that they will pay for a box of oatmeal raisin cookies (You may assume that all of these prices are non-negative integers).

   Susan needs to decide which of her two kinds of cookies to offer at each house in order to maximize the revenue she earns from her limited cookie stash. Give a recursive backtracking algorithm to help her with this problem. Your algorithm should take as input the values x and y and the arrays A and B. Your algorithm should output a length n array where each entry is either "chocolate chip", "oatmeal raisin", or "nothing", and for each i the ith entry describes which type of cookie Susan should sell at the ith house in order to maximize the total amount she earns from selling these cookies. You should justify the correctness of your algorithm, but you do not need to analyze the runtime.

5. Suppose you are given a list of n positive integers separated by plus and minus signs, for example

   $$5 + 2 - 3 + 7 - 4 - 12$$

   You can change the value of this expression by adding parentheses in different positions. For example:

   $$\begin{aligned} 5 + 2 - 3 + 7 - 4 - 12 &= -5 \\ 5 + 2 - (3 + 7) - (4 - 12) &= 5 \\ 5 + (2 - 3 + 7 - (4 - 12)) &= 19 \end{aligned}$$

Give a recursive backtracking algorithm to find a placement of parentheses which maximizes the value of the expression. You may only use the parentheses to group additions and subtractions, you are not allowed to create implicit multiplication (e.g. parenthesizing $5 - 2$ as $5(-2)$ is not allowed). You should justify the correctness of your algorithm, but you do not need to analyze the runtime.

11 Dynamic Programming

In this chapter we will see efficient dynamic programming algorithms for all of the problems we discussed in Chapter 10. We start by motivating the concept of dynamic programming using Fibonacci numbers in Section 11.1. Then we convert our recursive algorithm for subset sum into a dynamic programming algorithm in Section 11.2. Having seen these two examples, we describe the general pattern for dynamic programming algorithms in Section 11.3, then give two more examples, longest increasing subsequence and longest common subsequence, in Sections 11.4 and 11.5.

11.1 Fibonacci Numbers

The Fibonacci numbers are defined by the recurrence:

$$\begin{aligned} F(1) &= 1 \\ F(2) &= 1 \\ F(x) &= F(x-1) + F(x-2) \qquad \text{for } x \geq 3 \end{aligned}$$

We can give a very simple recursive algorithm for computing $F(x)$ using this recurrence:

```
FIB(x)
  If x ≤ 2:
      Return 1
  Return FIB(x − 1) + FIB(x − 2)
```

We note that the runtime of $\text{FIB}(x)$ is $\Omega(F(x))$, since calling $\text{FIB}(x)$ results in $F(x)$ total recursive calls. The Fibonacci recurrence is not one that we can solve with the techniques that we have seen, but we can get a lower bound by noting that $F(x) = F(x-1)+F(x-2) \geq 2F(x-2)$. The recurrence $F(x) = 2F(x-2)$ solves to $F(x) \in \Theta(2^{x/2})$ (we can apply the same trick that we used in the proof of Theorem 53 by defining a new function by $F'(y) = F(\log_2 y)$ and using the master theorem on F'). Thus, $F(x) \in \Omega(2^{x/2})$, so the runtime of this algorithm is exponential in x (and doubly exponential in the length of the input).

Why is FIB so slow? If we look at the actual recursive calls that the algorithm makes, we see that

- $\text{FIB}(x)$ calls $\text{FIB}(x-1)$ and $\text{FIB}(x-2)$
- Our 1 call to $\text{FIB}(x-1)$ calls $\text{FIB}(x-2)$ and $\text{FIB}(x-3)$
- Our 2 calls to $\text{FIB}(x-2)$ each call $\text{FIB}(x-3)$ and $\text{FIB}(x-4)$
- Our 3 calls to $\text{FIB}(x-3)$ each call $\text{FIB}(x-4)$ and $\text{FIB}(x-5)$
- And so on...

The problem is that our algorithm keeps making the *same* recursive calls over and over. We're making $\Omega(2^{x/2})$ recursive calls, even though the only inputs we're passing into the function are integers between 1 and x. We could make the algorithm much faster if we just computed $\text{FIB}(i)$ once for each i from 1 to x.

The following algorithm exploits this idea:

```
FASTFIB(x)
  If x ≤ 2 :
      Return 1
  A ← an array of length x
  A[1] ← 1
  A[2] ← 1
  For i from 3 to x
      A[i] ← A[i − 1] + A[i − 2]
  Return A[x]
```

Note that while FIB was a recursive algorithm, FASTFIB is an iterative one. Instead of breaking the problem of computing $F(x)$ down into smaller problems, we start from the base cases of $F(1)$ and $F(2)$ and build up to $F(x)$, storing all of the intermediate values that we compute in an array. We call this process *memoization* and it is the tool that we use to convert any recursive algorithm into a dynamic programming one.

Note, however, that just because we are only computing x Fibonacci numbers, this does not mean that FASTFIB runs in $\Theta(x)$ time! As we saw in Chapter 9, we need to consider the size of these numbers. By our analysis above, $F(x) = \Omega(2^{x/2})$ so it is a number with $\Omega(x)$ digits, meaning that the additions inside the for loop take $\Omega(x)$ time in the worst case. In fact, we can show that the runtime of FASTFIB is $\Theta(x^2)$. It's also important to note that, while this runtime is exponentially faster than the runtime for FIB, it's still exponential in the length of the input, since the length of x is $\Theta(\log x)$ (We can't hope for a subexponential time algorithm since the length of $F(x)$ is exponential in the length of x).

11.2 Some Supplemental SecondSubsetSumSolver Speedups

In Section 10.1 we saw the following recurrence and corresponding exponential recursive algorithm for subset sum.

$$SS(A,k,i) = \begin{cases} True & \text{if } k = 0 \\ False & \text{if } i > n \text{ and } k > 0 \\ SS(A,k,i+1) & \text{if } i \leq n, k > 0 \text{ and } A[i] > k \\ SS(A,k-A[i],i+1) \vee SS(A,k,i+1) & \text{otherwise} \end{cases}$$

```
SecondSubsetSumSolver(A[1,...,n], k, i = 1)
  If k = 0:
      Return True
  If i > n:
      Return False
  If A[i] > k:
      Return SecondSubsetSumSolver(A, k, i + 1)
  Return SecondSubsetSumSolver(A, k − A[i], i + 1) ∨ SecondSubsetSumSolver(A, k, i + 1)
```

Let's try to speed up this algorithm by using the bottom-up iterative technique we saw in Section 11.1. This time our recurrence has 3 parameters instead of 1, so we can't easily store the answers to all relevant subproblems in a 1-dimensional array. We can note however that the array A is the same for all of our subproblems, so we can identify our subproblems just by the non-negative integers k and i. Since we have two non-negative integer parameters, a natural data structure to represent them is a 2-D array.

We can rewrite the recurrence in a way better suited for array indices as follows:

$$SS(i,j) = \begin{cases} True & \text{if } j = 0 \\ False & \text{if } i > n \text{ and } j > 0 \\ SS(i+1,j) & \text{if } i \leq n, j > 0 \text{ and } A[i] > j \\ SS(i+1,j-A[i]) \vee SS(i+1,j) & \text{otherwise.} \end{cases}$$

Note that we have renamed k to j here to avoid confusion between the sums we are trying to get in our subproblems and the k which is part of the original input to the subset sum problem. We have also removed A as a parameter, since it is common to all subproblems, and we have rearranged the arguments to SS to be in alphabetical order.

While we typically assume that array indices start at 1 in this course, we don't need to rigidly hold to this convention when it comes at the expense of algorithmic clarity. The techniques we discussed for simulating arrays in the RAM model back in Section 5.4 can easily be extended to

arrays starting from 0, or even from a negative index. The important thing is to remember that an array needs space for all indices between its minimum index and max index, so if we have an array with indices ranging from a to b, it takes up space $\Theta(b-a)$ (assuming each element fits in a constant number of RAM model variables).

For this problem it is natural to let the first dimension of our DP array (corresponding to i) range from 1 to $n+1$ (we need to allow indices larger than n since our base case is when $i > n$) and let the second dimension of the array (corresponding to j) range from 0 to k. We give the following pseudocode. Note that the values we assign to the entries of D are taken directly from our recurrence, simply replacing SS with D.

```
SpeedierSubsetSumSolver(A[1,...,n], k):
    D[1,...,n+1][0,...,k] ← an array of booleans
    For i from 1 to n+1:
        D[i,0] ← True
    For j from 1 to k:
        D[n+1,j] ← False
    For i from n down to 1:
        For j from 1 to k:
            If A[i] > j:
                D[i,j] ← D[i+1,j]
            Else:
                D[i,j] ← D[i+1,j-A[i]] ∨ D[i+1,j]
    Return D[1,k]
```

One part of this algorithm which may still seem mysterious or arbitrary is the order in which we chose to fill in the entries of D. Why did we start by filling in all the entries where $i = n+1$ and then proceed to fill in the other entries in order of *decreasing* i value, rather than using a more normal loop structure where i goes from 1 to n?

The answer comes from looking at our recurrence. In order to be able to fill in our DP array while iterating through the entries only once each, we need to ensure that whenever our algorithm is filling in $D[i,j]$ we have already filled in all the entries that $D[i,j]$ depends on, i.e. if $SS(i,j)$ calls $SS(i',j')$ we need to make sure that our algorithm fills in $D[i',j']$ before $D[i,j]$. Thus, we start by filling in the base cases, where $i = n+1$, since these don't depend on any other entries. Then we fill in the other entries in order of decreasing value of i, because in every recursive call that SS makes, i increases. By going through the entries of D from largest i to smallest i we ensure that whenever we are computing $D[i,j]$, the entries that it depends on, namely $D[i+1,j]$ and possibly $D[i+1,j-A[i]]$, are already filled in.

Theorem 54. SpeedierSubsetSumSolver *runs in time $O(nk)$ and uses $O(nk)$ space.*

Proof. Creating an array of size $O(nk)$ on the first line takes $O(nk)$ time. Filling in the base cases takes time $O(n+k)$. The nested for loops iterate through $O(nk)$ entries of D, and fill in in each one in $O(1)$ time since we are only performing a constant number of operations for each (the comparison $A[i] > j$ takes time $O(1)$ since we know $0 \leq j \leq k$). Thus, the overall runtime is $O(nk)$. Storing the array D takes $O(nk)$ space, and all other variables in the algorithm take constant space, so the space usage is $O(nk)$. □

It's important to realize that dynamic programming algorithms do not always run in polynomial time. Dynamic programming allows us to design algorithms whose runtime depends on the number of different subproblems rather than on how many recursive calls those subproblems make, but the degree to which this speeds up the algorithm depends on how many recursive calls were using the exact same inputs. If there are exponentially many different inputs to our recursive function, then our DP table will have exponential size and filling it in will take exponential time.

In particular, note that the runtime bound of $O(nk)$ that we proved in Theorem 54 is actually still exponential in the input length. If k is a number with n digits then the value of k is about 10^n and our runtime bound becomes $O(n10^n)$. If k is small, however this algorithm is efficient, and even if k is large this algorithm is still much faster than the recursive backtracking version we developed last chapter.

11.3 Designing Dynamic Programming Algorithms

Let's describe and generalize the process that we saw in the previous two sections. How do we develop a dynamic programming algorithm?

The starting point of any dynamic programming algorithm is a recurrence. We can view dynamic programming as simply smart recursion. The challenging part of coming up with a dynamic programming algorithm is coming up with the right recurrence. This is why we spent Chapter 10 talking about designing recursive algorithms. Once we figure out the right recurrence, the steps to create the DP algorithm are pretty mechanical.

1. Formulate the problem recursively
   (a) Describe in English what your subproblems are (i.e. what problem your recurrence is solving)
      - Often this will require introducing additional parameters. Clearly explain what each one means.
   (b) Give a recurrence or a recursive algorithm to solve your subproblems.
      - Your recurrence should solve the problem using smaller versions of the exact same problem.
      - Make sure to include the base case.
   (c) Describe how to use your recurrence to solve the original problem (e.g. what should the default values be for your extra parameters?).
2. Give an algorithm which builds solutions to your recurrence from the bottom up
   (a) Identify what subproblems you need to solve (what are the possible inputs to your recurrence?)
   (b) Choose an appropriate memoization data structure (e.g. a multidimensional array)
   (c) Figure out the evaluation order (make sure that when you are filling in an entry of your DP array, you have already filled in the entries it depends on)
   (d) Analyze the runtime (usually just (# of subproblems) $\cdot$ (work per subproblem))

When you are asked to give an algorithm for a problem and want to use dynamic programming, you should make sure to define your subproblems in English (step 1(a)) so that it's clear what you're trying to do, whether or not you intend to give full pseudocode for your algorithm.

Then you should either explain the remaining steps (1(b), 1(c), and 2(a)-(d)) in English, at which point you have fully defined an algorithm, or you should give iterative pseudocode for your algorithm, which implicitly answers all the questions from these steps. In this chapter we will both explain the choices we are making for the algorithm in English and give the iterative pseudocode in order to maximize clarity.

11.4 Longest Increasing Subsequence Revisited

For another example of dynamic programming, let's revisit the longest increasing subsequence problem which we looked at in Section 10.2. Recall that we are given an array A of integers, and the goal is the find the length of the longest subsequence which is strictly increasing (meaning each element is greater than the previous one).

Recall that we defined $LIS(i, j)$ as the length of the longest increasing subsequence of $A[i, \ldots, n]$ where every element is larger than $A[j]$ and gave the following recurrence:

$$LIS(i,j) = \begin{cases} 0 & \text{if } i > n \\ LIS(i+1,j) & \text{if } i \leq n \text{ and } A[i] \leq A[j] \\ \max\{LIS(i+1,j), 1 + LIS(i+1,i)\} & \text{otherwise} \end{cases}$$

We can memoize the function LIS into a 2D array $D[1, \ldots, n+1][1, \ldots, n]$. Each entry $D[i, j]$ depends only on entries in the next row $D[i+1, \cdot]$, so we can fill the array from bottom to top (we arbitrarily choose to fill each row from left to right).

```
LIS-Solver(A[1, ..., n]):
  D ← a 2D array of size (n + 1) × n
  For j from 1 to n                    ▷ Base cases
      D[n + 1, j] ← 0
  For i from n down to 1
      For j from 1 to n
          If A[i] ≤ A[j]
              D[i, j] ← D[i + 1, j]
          Else
              D[i, j] ← max{D[i + 1, j], 1 + D[i + 1, i]}
  Return max_{1≤i≤n}(1 + D[i + 1, i])
```

Theorem 55. LIS-Solver(A) *returns the length of the longest increasing subsequence of* A*, runs in time* $\Theta(n^2)$ *and uses* $\Theta(n^2)$ *space.*

Proof. The correctness of this algorithm follows from the correctness of the recurrence: Here our recurrence represents the choice of whether or not to include $A[i]$ in our subsequence. If $A[i] \leq A[j]$, we have no choice, we have to skip $A[i]$. If $A[i] > A[j]$, we can include it, in which case we increase the length of our sequence by 1, but change our lower bound on future elements to $A[i]$, or we can skip it. We choose whichever option gives a longer subsequence. To solve the original problem, we compute $\max_{1 \leq i \leq n}(1 + D[i+1, i])$ because this is the max, over all choices of first element, of the longest increasing subsequence we can get starting with that element.

Now we analyze the runtime. Our array has $\Theta(n^2)$ entries, and we can compute each one in $\Theta(1)$ time (computing a new entry requires comparing $A[i]$ and $A[j]$ and then either copying a value we already computed or taking the max of two values we already computed and copying one of them, all of which are constant time operations). Therefore the total runtime to fill in our table is $\Theta(n^2) \cdot \Theta(1) = \Theta(n^2)$. Computing the final answer from our table requires taking the max over n entries which takes $\Theta(n)$ time. Thus, our algorithm runs in $\Theta(n^2)$ time.

Finally, we note that since our DP table has $\Theta(n^2)$ entries, the space complexity is $\Theta(n^2)$.

□

If we want to be clever, we can reduce the space complexity of our DP algorithm by defining our subproblems and recurrence slightly differently.

Let $LIS(i)$ denote the length of the longest increasing subsequence of $A[i, \ldots, n]$ that begins with $A[i]$. We give the following recurrence:

$$LIS(i) = 1 + \max_{i < j \leq n \text{ with } A[j] > A[i]} LIS(j).$$

Here our recurrence represents choosing the next element in the sequence after $A[i]$. We define $\max \emptyset = 0$ so that this recurrence covers the base cases. To solve the original problem, we need to compute $\max_{1 \leq i \leq n} LIS(i)$.

We can memoize the function LIS into a 1D array $D[1, \ldots, n]$. Each entry $D[i]$ depends only on subsequent entries, so we can fill the array in order of decreasing values of i.

```
LIS-Solver2(A[1, ..., n]):
  D ← a 1D array of size n
  For i from n down to 1
      max_len ← 0
      For j from i + 1 to n
          If A[j] > A[i] and D[j] > max_len
              max_len ← D[j]
      D[i] ← 1 + max_len
  Return max_{1≤i≤n} D[i]
```

Theorem 56. LIS-SOLVER2(A) *returns the length of the longest increasing subsequence of A, runs in time $\Theta(n^2)$ and uses $\Theta(n)$ space.*

Proof. Once again, our algorithm is simply implementing our recurrence, so the correctness follows from the correctness of the recurrence. If we have an increasing subsequence starting with $A[i]$ then the valid choices for the next element are all elements after index i which are greater than $A[i]$, and we pick whichever one gives us the longest overall subsequence. The total length of our subsequence is 1 (for $A[i]$) plus the length of the subsequence starting with the next element.

Our array has $\Theta(n)$ entries now, but each one takes $\Theta(n)$ time to compute, since computing $D[i]$ requires looping over all of $A[i+1, \ldots, n]$. Therefore the total runtime to fill in our table is $\Theta(n) \cdot \Theta(n) = \Theta(n^2)$. Computing the final answer from our table requires taking the max over n entries which takes $\Theta(n)$ time. Thus, our algorithm runs in $\Theta(n^2)$ time. Since we are only using a 1D array this time, however, the space complexity is only $\Theta(n)$. □

This example illustrates that there can be multiple valid choices for how we define subproblems, and they are not always equivalent. In this example, our second version of the subproblems gave the same runtime but better space complexity. In general it's possible that some subproblem definitions may yield faster runtimes than others. When developing your recurrence it's worth taking time to think whether there may be multiple options for how to define the subproblems and which one will yield the smallest DP array.

11.5 Longest Common Subsequence Revisited

Now let's revisit the last problem we saw a recursive backtracking algorithm for last chapter: longest common subsequence. Recall that in this problem we are given two arrays $A[1, \ldots, m]$ and $B[1, \ldots, n]$ and we want to find the longest sequence which is a subsequence of both A and B.

Recall that in Section 10.4 we defined $LCS(i, j)$ as the length of the longest common subsequence of $A[i : m]$ and $B[j : n]$. We defined an auxiliary function I by

$$I(i, j) := \{(i', j') \mid i \leq i' \leq m, j \leq j' \leq n \text{ and } A[i'] = B[j']\}.$$

That is, $I(i, j)$ is the set of all pairs of indices (i', j') in our range of valid indices where $A[i'] = B[j']$. Then we gave the following recurrence for LCS:

$$LCS(i, j) = \begin{cases} 0 & \text{if } I(i, j) = \emptyset \\ \max_{(i', j') \in I(i,j)} 1 + LCS(i' + 1, j' + 1) & \text{otherwise} \end{cases}.$$

We noted that the solution to the original problem is $LCS(1, 1)$.

Since this recurrence has two parameters, we can memoize it with a 2-D array $D[1, \ldots, m + 1][1, \ldots, n + 1]$. Since each entry in the array depends only on entries with larger values of i and j, we fill in the array in order of decreasing values of i.

We summarize these choices with the following pseudocode:

```
LCS-SOLVER(A[1, ..., m], B[1, ..., n]):
  D[1, ..., m + 1][1, ..., n + 1] ← a 2-D array of integers
  For i from m + 1 down to 1
      For j from 1 to n + 1
          max_len ← 0
          For i' from i to m:
              For j' from j to n:
                  If A[i'] = B[j']
                      max_len ← max(max_len, 1 + D[i' + 1, j' + 1])
          D[i, j] ← max_len
  Return D[1, 1]
```

Theorem 57. LCS-SOLVER *returns the length of the longest common subsequence of A and B, runs in time $\Theta(m^2 n^2)$ and uses $\Theta(mn)$ space.*

Proof. **Correctness:** Every pair of a character from $A[i, \ldots, m]$ and a matching character from $B[j, \ldots, n]$ (not necessarily at the same index) forms a potential starting point for a common subsequence of these two arrays. The innermost two loops in our algorithm implement the "max" in our recurrence by considering all possible pairs of starting characters. Then, for each pair of matching characters, we determine the longest common subsequence we could obtain starting with those characters, which is 1 (for the first character we've chosen) plus the longest common subsequence of the remainder of the arrays.

Runtime: We have $\Theta(mn)$ entries in our DP array, and we spend $\Theta((m-i)(n-j))$ time filling in the entry $D[i, j]$ (since we need to iterate over all pairs (i', j') with $i \leq i' \leq m$ and $j \leq j' \leq n$). This is $O(mn)$ for all entries and $\Omega(mn)$ for at least a constant fraction of the entries (e.g. the ones where $i \leq m/2$ and $j \leq n/2$). Thus the overall runtime is $\Theta(mn) \cdot \Theta(mn) = \Theta(m^2n^2)$.

Space: We use an $m \times n$ array (plus a constant number of integer variables), so the algorithm takes $\Theta(mn)$ space. □

We've shown that LCS-Solver is a polynomial time algorithm, which is a big improvement over the exponential time algorithm we gave in Section 10.4. Nevertheless, we can still try to do better.

In Section 11.4 we were able to improve the space complexity of our algorithm by giving a recurrence with fewer parameters, allowing us to use a smaller DP array (at the cost of taking longer to compute each entry).

For the longest common subsequence problem, having our recurrence take in an index for each array seems fairly necessary, but we can design a recurrence that will let us compute the entries of the DP array more efficiently. Since looking ahead at all the possible choices for the next element of the common subsequence isn't allowing us to use a smaller DP table here, it makes sense to try a perspective like the one we took for our first algorithm for longest increasing subsequence and view the choice we are making at each step of the recurrence as "Should we include or skip the element at the current index" rather than "Which element should we add to our common subsequence next?".

In particular, if the first elements of A and B are the same, we might as well use that element as the first element of a common subsequence (Any common subsequence of $A[i, \ldots, m]$ and $B[j, \ldots, n]$ that didn't start with $A[i]$ could be made longer by prepending $A[i]$). Otherwise, we know that our common subsequence can't use both the first element of A and the first element of B, so we can pick one of these two to discard and recurse on the subproblem where one of the arrays is one element shorter. Since we don't know which array we should discard the first element from, we use our standard recursive backtracking approach of trying both and taking the max.

Once again we'll define $LCS(i, j)$ as the length of the longest common subsequence of $A[i : m]$ and $B[j : n]$. This time we give the following alternative recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i > m \text{ or } j > n \\ 1 + LCS(i+1, j+1) & \text{if A[i] = B[j]} \\ \max(LCS(i+1, j), LCS(i, j+1)) & \text{otherwise} \end{cases}.$$

We can memoize this recurrence into an $m + 1 \times n + 1$ array. This time, our recurrence has recursive calls where i stays the same, so just evaluating our subproblems in order of decreasing value of i is not enough to ensure that when we are computing $D[i, j]$ we have already computed the entries it depends on. We also need to make sure that, for a given value of i, we iterate through the values of j from largest to smallest (because our base case is when $j > n$ and j increases when we recurse keeping i the same).

This gives us the following DP pseudocode:

```
LCS-Solver2(A[1,...,m], B[1,...,n]):
  D[1,...,m+1][1,...,n+1] ← a 2-D array of integers
  For i from m+1 down to 1
      For j from n+1 down to 1
          If i > m or j > n:
              D[i,j] ← 0
          Else If A[i] = B[j]:
              D[i,j] ← 1 + D[i+1, j+1]
          Else:
              D[i,j] ← max(D[i+1, j], D[i, j+1])
  Return D[1,1]
```

Once again we have $\Theta(mn)$ subproblems, but this time it takes only $\Theta(1)$ work to solve each of them, so our overall runtime is $\Theta(mn)$

11.6 Comprehension Questions

1. Suppose that we are creating a dynamic programming algorithm for solving a problem about an array A and want to memoize the following recurrence into an array $D[1,\ldots,n][1,\ldots,n]$

$$f(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ f(i+1,j) & \text{if } i < j \text{ and } A[i] \geq A[j] \\ \max\{f(i+1,j), 1+f(i,j-1)\} & \text{otherwise} \end{cases}.$$

   Which of the following evaluation orders would be appropriate for filling in our DP table? (select all that apply)

   (a) Fill in the entries from smallest value of i to largest. For entries with the same value of i, fill them in from smallest value of j to largest.

   (b) Fill in the entries from smallest value of i to largest. For entries with the same value of i, fill them in from largest value of j to smallest.

   (c) Fill in the entries from largest value of i to smallest. For entries with the same value of i, fill them in from smallest value of j to largest.

   (d) Fill in the entries from largest value of i to smallest. For entries with the same value of i, fill them in from largest value of j to smallest.

2. What is the runtime of FastFib(x) in terms of x (not the input length n)?

3. In this chapter we designed a DP algorithm SpeedierSubsetSumSolver for subset sum. What is the runtime of SpeedierSubsetSumSolver($A[1,\ldots,n], n^3$), i.e. what is the running time of the algorithm if we require k to be n^3? Express your answer as a tight big-O bound in terms of n.

4. In this chapter we designed a DP algorithm LIS-Solver2 based on the recurrence

$$LIS(i) = 1 + \max_{i<j\leq n \text{ with } A[j]>A[i]} LIS(j).$$

   What is the space complexity of LIS-Solver2?

5. Suppose that we are trying to solve a problem where the input is an array $A[1,\ldots,n]$. We have come up with a recurrence for the problem and memoized it into an $n \times n$ array which our DP algorithm will fill in. What can we say about the runtime of this DP algorithm?

   (a) The runtime of the algorithm is $O(n^2)$

   (b) The runtime of the algorithm is $\Theta(n^2)$

   (c) The runtime of the algorithm is $\Omega(n^2)$

   (d) This is not enough information to say anything about the algorithm's runtime.

11.7 Exercises

1. We define a sequence of integers $x_1, x_2, \ldots, x_n$ as *converging* if for every $i \geq 3$, x_i lies in-between x_{i-1} and x_{i-2} (i.e. either $x_{i-2} < x_i < x_{i-1}$ or $x_{i-1} < x_i < x_{i-2}$). For example, $1, 100, 40, 60, 47, 59, 50, 52$ is a converging sequence. Give a polynomial time algorithm which takes as input a sequence $a_1, \ldots a_n$ of n distinct positive integers, represented by an array $A[1, \ldots, n]$ and returns the length of a longest converging subsequence of $a_1, \ldots, a_n$. For example, if

$$A = [101, 1, 40, 100, 90, 80, 30, 50, 43, 87, 48, 46, 44, 47, 103, 102]$$

then your algorithm should return 8 because a longest converging subsequence of A is $1, 100, 30, 50, 43, 48, 44, 47$, which has length 8.

2. You are planning a road trip between cities A and B. Fortunately, they are connected by a long straight highway. Unfortunately, you are going to have to stop for fuel several times, and all of the gas stations on this highway use a very unusual pricing system. Gas is sold by the tank, meaning that you pay the same fixed amount to refill your tank, whether that takes one gallon or twenty.

   Suppose that you are given d, the distance in miles that your car can travel on a full tank of gas, as well as two arrays $G[1, \ldots, n]$ and $P[1, \ldots, n]$, where $G[i]$ is the location of the ith gas station, given as a distance in miles from your starting point, and $P[i]$ is the price (in dollars) to fill your tank at the ith gas station. Assume that your trip begins at the first gas station (so $G[1] = 0$) with an empty tank of gas, and ends at the nth gas station (where you do not need to refuel). Assume that it is possible to complete the trip (i.e. there is no stretch of d miles with no gas stations). Give an efficient algorithm to calculate the minimum amount that you can pay for gas while making this trip.

3. Susan is going door to door in her neighborhood selling boxes of cookies. Susan's cookies come in two different kinds. She has x boxes of chocolate chip cookies and y boxes of oatmeal raisin cookies. Susan's neighborhood consists of n houses arranged in a straight line. Each household is willing to buy at most one box of cookies, and they each value the two kinds of cookies differently. Fortunately, Susan has done extensive market research, and has compiled two arrays, $A[1, \ldots, n]$ and $B[1, \ldots n]$, where $A[i]$ is the price (in cents) that the ith house will pay for a box of chocolate chip cookies, $B[i]$ is the price that they will pay for a box of oatmeal raisin cookies (You may assume that all of these prices are non-negative integers).

   Susan needs to decide which of her two kinds of cookies to offer at each house in order to maximize the revenue she earns from her limited cookie stash. Give an efficient algorithm to help her with this problem. Your algorithm should take as input the values x and y and the arrays A and B. Your algorithm should output a length n array where each entry is either "chocolate chip", "oatmeal raisin", or "nothing", and for each i the ith entry describes which type of cookie Susan should sell at the ith house in order to maximize the total amount she earns from selling these cookies.

4. Suppose you are given a list of n positive integers separated by plus and minus signs, for example

$$5 + 2 - 3 + 7 - 4 - 12$$

   You can change the value of this expression by adding parentheses in different positions. For example:

$$\begin{aligned} 5 + 2 - 3 + 7 - 4 - 12 &= -5 \\ 5 + 2 - (3 + 7) - (4 - 12) &= 5 \\ 5 + (2 - 3 + 7 - (4 - 12)) &= 19 \end{aligned}$$

   Give an efficient algorithm to find a placement of parentheses which maximizes the value of the expression. You may only use the parentheses to group additions and subtractions, you are not allowed to create implicit multiplication (e.g. parenthesizing $5 - 2$ as $5(-2)$ is not allowed).

5. We define a *ones decomposition* as a valid arithmetic expression that can be created using only the 5 symbols $1, +, \times, ($, and $)$. Each ones decomposition represents some positive integer, and most positive integers have multiple ones decompositions. For example, the number 14 can be represented by any of the following ones decompositions

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1+1)\times(1+1+1))+((1+1)\times(1+1))+1$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1)\times(1+1+1))+1)$$

Give an algorithm which takes as input a positive integer x and outputs the minimum number of 1s in a ones decomposition for x. For example, given the input 14, your algorithm should return 8 (the last ones decomposition given above uses a minimum number of 1s). The runtime of your algorithm should be polynomial in x.

6. We define a balanced string of braces recursively as follows:

- ε (the empty string) is a balanced string of braces.
- If s is a balanced string of braces then so is $\{s\}$.
- If s is a balanced string of braces then so is $\langle s\rangle$.
- If s and t are balanced strings of braces, then so is st (the concatenation of s and t).

For example, the string $\{\langle\rangle\{\}\}\langle\rangle$ is a balanced string of braces, but $\{\langle\}\rangle$ is not.

Suppose that you are given an array $A[1,\ldots,n]$ where each element of A is one of $\langle$, $\rangle$, $\{$, or $\}$.

(a) Give an $O(n)$-time algorithm to determine whether or not A represents a balanced string of braces.

(b) Give an $O(n^3)$-time algorithm to find the length of the longest subsequence of A which is a balanced string of braces.

7. Linear Town is a town consisting of a single very long straight road running east to west. In the middle of this street is a pizza place where you have been hired as a pizza delivery driver. When you show up for your first day of work, you're informed that a driver quit in the middle of the previous shift, leaving you with a stack of n pizzas that urgently need to be delivered.

The customers who ordered these pizzas are furious at how long they have had to wait and have started using fake accounts to leave 1 star reviews of your restaurant. For every minute that a customer goes without receiving their pizza, they will leave another 1 star review. Your goal is to deliver all the pizzas while incurring as few negative reviews as possible.

Fortunately, there's no traffic right now, so you will be able to drive at a consistent speed while delivering the pizzas. Since the entire city is one long street, instead of needing to keep track of a list of addresses, the location of a house can be expressed just by how many minutes it takes you to drive there from the restaurant and whether it is west or east of the restaurant.

You are given a sorted array $A[1,\ldots,n]$ of integers, where $A[i]$ is the number of minutes you need to drive east from the restaurant to reach the house (negative numbers correspond to houses which are west of the restaurant). Give an $O(n^2)$-time algorithm to determine the minimum number of 1-star reviews that you can incur in the process of delivering the pizzas.

For example, if $A=[-10,-9,-8,1,7]$, then the optimal route is $[1,-8,-9,-10,7]$ which incurs 63 1-star reviews (5 while driving to 1, 36 more while driving to -8, 3 more while driving to -9, 2 more while driving to -10, and 17 more while driving to 7).

12 Greedy Algorithms

In this chapter we'll look at an approach for solving problems with a recursive structure which is faster than dynamic programming when it works, but works on a more limited set of problems. In Section 12.1 we'll consider the interval partitioning problem, where we assign classes to classrooms, and in Section 12.2 we'll look at scheduling jobs on a computer in a way that minimizes the maximum lateness. Finally, in Section 12.3 we'll discuss the general pattern for greedy algorithms, and contrast it with the dynamic programming pattern that we learned about last chapter.

12.1 Interval Partitioning

Let's start by examining a problem faced by universities like Texas A&M. We are given a list of classes, each of which has a start time and an end time. We want to assign the classes to the classrooms. Classes cannot be assigned to the same classroom if their meeting times overlap. Since space at the university is limited, we would like to minimize the total number of classrooms that we use.

In the real world there are a variety of additional constraints. For example, classes have different numbers of students and classrooms have different capacities. Some classes meet only one day a week, while others meet on two or three. Some classes' instructors may have preferences for particular rooms. For the sake of keeping the problem simple to analyze, we will ignore all of these issues in this section and focus only on the constraint that we cannot have two classes meeting at the same time in the same room.

More formally, in the interval partitioning problem our input is an array $A[1, \ldots, n]$ of pairs of positive integers, representing start times and finish times for classes. We say that two classes $c_i = (s_i, f_i)$ and $c_j = (s_j, f_j)$ *conflict* if $s_i \leq s_j < f_i$ or $s_j \leq s_i < f_j$ (i.e. if one of them starts while the other is ongoing). Our goal is to find the minimum number of classrooms that we need to be able to assign these classes to rooms such that no classes which conflict are assigned to the same classroom.

This is a problem that we could definitely solve using the recursive backtracking technique from Chapter 10. We simply iterate through the list of classes, and for each one we try assigning it to each possible classroom (including creating a new classroom), recursively assigning the rest of the classes in a way that doesn't conflict.

But while it's easy to design a correct recursive backtracking algorithm, it's not clear how to design one that will convert into an efficient dynamic programming algorithm. In particular, it's not clear what short summary of our choices so far we could use to keep the number of subproblems small.

But what if we could speed up the algorithm in a different way? What if instead of trying out each class in every available classroom, we just arbitrarily choose one, without recursively checking how many rooms we'll need for the rest of the classes. Maybe things will work out anyway?

We call this approach "greedy", because we are making a sequence of small choices (like in recursive backtracking or DP algorithms), but instead of looking at the long-term consequences of every possible choice, we just make a choice that seems to make sense based on what we've seen so far, without even looking at the rest of the array.

The greedy algorithm we just discussed is given below. We iterate through the classes and assign each one to the first room where it has no conflicts. If it has a conflict with every classroom, we create a new room for the class.

```
GreedyAlg(A[1,...,n])
  rooms ← an empty 2-D array of classes
  For i from 1 to n:
      For j from 1 to size(rooms)+1:
          If j > size(rooms):
              Append an array containing only A[i] to rooms
          Else if A[i] does not conflict with any class in rooms[j]:
              Add A[i] to rooms[j]
              Break
  Return size(rooms)
```

Theorem 58. GreedyAlg *does not correctly solve the interval partitioning problem.*

Proof. Suppose that $A = \{(1,4),(8,10),(3,7),(6,9)\}$. Since GreedyAlg assigns each class to the first room that it fits in, it will start by assigning $(1,4)$ to room 1. $(8,10)$ does not conflict with $(1,4)$, so it will also be assigned to room 1. Then, since $(3,7)$ conflicts with $(1,4)$, the algorithm will create a second room to assign it to. Finally since $(6,9)$ conflicts with both $(8,10)$ and $(3,7)$, the algorithm will create a third room to assign it to and will return 3.

In fact, however, it is possible to fit these four classes into two rooms. If we assign $(1,4)$ and $(6,9)$ to room 1 and $(3,7)$ and $(8,10)$ to room 2, then neither room has a conflict. □

As the previous example illustrates, intuitive-sounding greedy algorithms can frequently fail, returning incorrect answers for certain inputs. Even if we cannot come up with a specific example where a greedy algorithm fails, we should not trust it unless we can give a proof that it will always return the correct answer. This is true for all algorithms, but it's particularly important for greedy algorithms, because they are especially prone to sounding "obviously" correct even when they don't work.

In this case, our failed greedy algorithm is actually salvageable. As the bad example in our incorrectness proof shows, a failure mode of GreedyAlg is scheduling two classes in the same room with an awkward gap in-between them, too small to fit another class. The way to avoid this is to sort the classes and consider them in order of increasing start time.

Sorting the classes by start time also allows us to make the algorithm more efficient. If we know that class i starts later than any class we have scheduled so far, then to determine if class i can be scheduled in room j, it's sufficient to check if class i starts after the end of the last class we scheduled in room j. Since all the other classes in room j must start earlier than the last class we scheduled there, if class i does not conflict with the last class scheduled in the room, it cannot conflict with any of the others either.

Based on these observations, we give the following improved algorithm:

```
GreedyIntervalPartition(A[1,...,n])
  Sort A by start time
  lastClass ← an empty array of integers
  For i from 1 to n:
      For j from 1 to size(lastClass)+1:
          If j > size(lastClass):
              Append A[i].finish to lastClass
              Break
          Else if A[i].start ≥ lastClass[j]:
              lastClass[j] ← A[i].finish
              Break
  Return size(lastClass)
```

Theorem 59. *For every array of classes A,* GreedyIntervalPartition *returns the minimum number of classrooms needed to schedule these classes without conflicts. Furthermore,* GreedyIntervalPartition *runs in $O(n^2)$ time.*

Proof. We note that lastClass[j] stores the finish time of the last class that we scheduled for room j. Thus, our inner loop finds the first room where we can schedule class $A[i]$ without a conflict

and assigns $A[i]$ there. Therefore, when the outer loop finishes, we have found a sequence of room assignments using size(lastClass) rooms with no conflicts.

It remains to prove that it is not possible to assign the classrooms of A using fewer rooms than our algorithm does. Let k be the number of rooms that GreedyIntervalPartition uses. Let c_k be the first class assigned to room k. In order for this to happen, it must have been the case that when we were searching for a room for c_k, every room $1 \leq i < k$ had a class c_i with $c_i.finish > c_k.start$. Furthermore, since we sorted the classes by start time, every class considered before c_k must start at the same time as c_k or earlier, so we have $c_i.start \leq c_k.start$ for every $i \in \{1, \ldots, k-1\}$.

Therefore, at the time $c_k.start$, all k of the classes $c_1, \ldots, c_k$ are taking place. This means that no two of these classes can be scheduled in the same room, so every valid assignment of these classes to rooms must use at least k rooms.

Finally, we analyze the runtime of our algorithm. Sorting A takes $\Theta(n \log n)$ time (e.g. using mergesort). Our outer for loop runs $O(n)$ times and our inner for loop runs $O(k)$ times. The work inside of the inner for loop is $O(1)$, so the total time for the outer for loop is $O(nk)$. Thus, the overall runtime of our algorithm is $O(\max\{n \log n, nk\})$. Since $\log n \in O(n)$ and $k \in O(n)$ (we cannot do worse than scheduling every class in its own room), this runtime is $O(n^2)$. □

12.2 Job Scheduling

Let's see another variety of scheduling problem which can be solved using a greedy algorithm. In the single machine scheduling problem we are given a set of jobs J which our machine needs to run. Each job j has two attributes:

- $j.time$ is the time that j takes our machine to complete (measured e.g. in minutes)
- $j.deadline$ is the time by which we are supposed to complete j (measured in minutes since the time when we start running jobs).

There are a variety of questions we could ask in this setting. e.g. we can ask whether it is possible to sequence the jobs in a way that satisfies all of the deadlines. We could seek to finish as many jobs on time as possible. We could seek to minimize the average lateness. The objective we will focus on in this section, however, is minimizing the *maximum* lateness.

We will assume that we have our machine running jobs non-stop until all of them are finished. So the choice that we need to make is what order to process the jobs in. i.e. we need to choose an ordering $j_1, j_2, \ldots, j_n$ for the jobs. Given such a sequence, the time at which we finish job j_i is

$$t_i := \sum_{k=1}^{i} j_k.time,$$

the total runtime of j_i plus all of the jobs that we run before it. The lateness of job j_i is

$$\ell_i := \max(0, t_i - j_i.deadline)$$

i.e. it is the difference between when we complete the job and the deadline, assuming that we complete the job late. Our goal is to find a sequence for the jobs which minimizes $\max_{i=1}^{n} \ell_i$.

For example, if our jobs (represented by (time, deadline) pairs) are $A = (1, 7), B = (5, 2), C = (3, 3), D = (20, 5)$ and we choose to process the jobs in the order A, B, C, D (from shortest time to longest), then they finish at times $t_1 = 1, t_2 = 6, t_3 = 9, t_4 = 29$, and their lateness is $\ell_1 = 0, \ell_2 = 4, \ell_3 = 6, \ell_4 = 24$, for a maximum lateness of 24. A better ordering would have been B, C, D, A, which gives us finishing times of $t_1 = 5, t_2 = 8, t_3 = 28, t_4 = 29$ with lateness $\ell_1 = 3, \ell_2 = 5, \ell_3 = 23, \ell_4 = 22$, for a maximum lateness of 23.

This time, the appropriate algorithm is one familiar to procrastinating students everywhere: "focus on whichever job has the earliest deadline". That is, our algorithm is just to sort the jobs from earliest deadline to latest, so simple that even giving pseudocode would be superfluous. The interesting step is proving that this algorithm actually does keep the maximum lateness as small as possible.

Theorem 60. *For any set of jobs J, the maximum lateness is minimized by processing the jobs in order from earliest deadline to latest.*

Proof. We prove this by induction on n.

- Base case: If $n = 1$, then there is only one possible ordering, so ordering by deadline performs the same as any other ordering and is therefore optimal.
- Induction Hypothesis: Suppose that for every set of fewer than n jobs, ordering them from smallest to largest deadline minimizes the maximum lateness.
- Induction Step: Let J be a set of $n > 1$ jobs, let $j_1, \ldots, j_n$ be these jobs ordered from earliest to latest deadline. Let $L := \max_{i=1}^{n} \ell_i$ be the maximum lateness of these jobs. We consider two cases:
  - First suppose that $L = \ell_n$, i.e. that j_n is a job with maximum lateness in our schedule. In this case, $L = t_n - j_n.deadline = (\sum_{i=1}^{n} j_i.time) - j_n.deadline$.

    We note that no matter how we order the jobs, the last job will always finish at the same time $(\sum_{i=1}^{n} j_i.time)$. Thus, the lateness of the last job will always be the sum of the runtimes of all the jobs minus the deadline of the last job. Since every other job has a deadline $\leq j_n.deadline$, any other job will have a lateness of at least L when processed last. Since some job must be processed last, this means that every ordering has maximum lateness at least L, and thus our ordering minimizes the maximum lateness.
  - Now suppose that $L \neq \ell_n$, i.e. that the last job is not the latest one. Then $L = \max_{i=1}^{n-1} \ell_i$. By our induction hypothesis, any ordering of $j_1, \ldots, j_{n-1}$ has maximum lateness at least L.

    Thus, for any ordering of $j_1, \ldots, j_n$, if we remove j_n, the maximum lateness is at least L. Adding j_n back cannot decrease the maximum lateness (it can only delay jobs not cause them to finish earlier), so therefore the original schedule of n jobs also had maximum lateness at least L. Thus, every ordering of the n jobs has maximum lateness at least L, and thus our ordering minimizes the maximum lateness.

This proof relied on an *exchange argument* to show that our choice to make the job with the latest deadline last was optimal. We noted that exchanging the job that we put last with any other job could not improve the maximum lateness. Thus, our choice of the last job was optimal, and by induction, our choice of the remaining jobs was also optimal. This kind of argument occurs commonly in analyzing greedy algorithms.

□

12.3 Greedy Algorithms vs Dynamic Programming

The exact definition of a greedy algorithm is a bit fuzzy, and it's not important to be able to classify whether or not a particular algorithm counts as "greedy". As a general rule of thumb, however, greedy algorithms are ones that:

- Make small incremental decisions without backtracking
- Make locally-optimal (greedy) decisions, without considering the global picture
- Often follow simple fixed priority rules

For example, our algorithm for interval partitioning assigned the classes to rooms one at a time (making small incremental decisions), without looking ahead at what classes were coming next (making locally-optimal decisions) following the simple rule to evaluate classes in order from earliest to latest start time.

To recognize when a greedy algorithm applies, you should look for two key characteristics: optimal substructure and a greedy choice property.

Optimal substructure is actually a requirement not only for greedy algorithms, but also for dynamic programming and recursive backtracking ones. This is the property that an optimal solution can be built by making a single choice and then optimally solving a smaller version of the same problem (or a generalized version of the problem, since for both DP algorithms and greedy algorithms it's often necessary to keep some summary of our choice so far). For example, an optimal room assignment for n classes can be built by choosing a room for the first class, and then choosing an optimal assignment for the remaining $n-1$ classes (given an array of times when each room becomes available). A maximum-lateness-minimizing ordering for n jobs can be found by assigning the job with the last deadline to the final position, and then finding a maximum-lateness-minimizing ordering for the remaining $n-1$ jobs.

The advantage of greedy algorithms over dynamic programming is that while dynamic programming relies on overlapping subproblems, so that the total number of possible subproblems is small (polynomial), greedy algorithms can work even when the number of possible subproblems is large, because we do not have to explore all of them.

The greedy choice property says that a globally optimal solution can be reached by making locally optimal (greedy) choices. This is the part that can require a clever proof, often built on induction and some kind of exchange argument. The problems that we gave dynamic programming algorithms for in the previous chapter have optimal substructure (e.g. a longest increasing subsequence for A can be built by adding an element onto a longest increasing subsequence for some subarray of A), but do not have this greedy choice property.

Note that to be able to prove the greedy choice property, we need to be using the right greedy choice rule. As we saw in Section 12.1, it's easy to give a natural looking greedy choice rule that does not correctly solve the problem.

12.4 Comprehension Questions

1. Consider the sequence of classes $(10, 15), (1, 7), (2, 9), (14, 18), (8, 13), (3, 11)$. How many classrooms do we need to schedule these classes?

2. Suppose that we have jobs with the following (time, deadline) pairs. How should we sequence them to minimize the maximum lateness? $A = (12, 9), B = (8, 10), C = (14, 7), D = (5, 11)$

3. True or false? If GreedyIntervalPartition(A) returns k, then there is some point in time at which k classes from A are all meeting simultaneously.

4. Consider the following algorithm for job scheduling: "Order the jobs from shortest time to longest time". Give a counterexample to prove that this greedy algorithm does not necessarily minimize the maximum lateness.

5. True or false? Every problem that can be solved in polynomial time with dynamic programming can also be solved in polynomial time with a greedy algorithm?

12.5 Exercises

1. For the third chapter in a row, you are planning a road trip between cities A and B. They are still connected by a long straight highway. This time, however, you are rich, and don't care at all about how much you spend on gas. Instead, your most valuable resource is time, and so your goal is to minimize the number of times that you need to stop for fuel.

   Suppose that you are given d, the distance in miles that your car can travel on a full tank of gas, as well as a *sorted* array $G[1, \ldots, n]$, where $G[i]$ is the location of the ith gas station, given as a distance in miles from your starting point. Assume that your trip begins at the first gas station (so $G[1] = 0$) with an empty tank of gas, and ends at the nth gas station (where you do not need to refuel). Assume that it is possible to complete the trip (i.e. there is no stretch of $> d$ miles with no gas stations). Give an $O(n)$ time algorithm to calculate the minimum number of times that you will need to refuel in order to complete this trip.

Make sure to prove that your algorithm is correct, i.e. that the number of stations you return really is the minimum possible.

2. You are the sole employee at a call center. n different customers have just called in at the same time. Fortunately, you have supernatural powers to help you do your job. Before picking up the phone, you can immediately tell how long it will take you to handle each of these calls. You have created an array $A[1, \ldots, n]$, where $A[i]$ is the time that it will take to handle the i'th call. You would like to choose an ordering $i_1, \ldots, i_n$ in which to take the calls, which will minimize the total waiting time of your customers. Note that the total wait time is

$$\sum_{j=1}^{n} \sum_{k=1}^{j-1} A[i_k].$$

That is, the total wait time is the sum of the wait times of all the customers, and the wait time of a customer is the total time you spend helping everyone whose call you take before theirs.

Give an $O(n \log n)$ time algorithm to find an ordering for the calls which minimizes the total wait time.

3. We define a balanced string of parentheses recursively as follows:

- ε (the empty string) is a balanced string of parentheses
- If s is a balanced string of parentheses then so is (s)
- If s and t are balanced strings of parentheses, then so is st (the concatenation of s and t).

For example, the string (())() is a balanced string of parentheses because it is created by applying the third rule to (()) and (). () is balanced because it is created by applying the second rule to ε (which is balanced by the first rule), and (()) is balanced because it is created by applying the second rule to (). On the other hand)(is not a balanced string of parentheses since it cannot be constructed according to these rules.

Suppose that you are given an array $A[1, \ldots, n]$ where each element of A is either (or).

(a) Give an $O(n)$-time algorithm to determine whether or not A represents a balanced string of parentheses.

(b) Give an $O(n)$-time algorithm to find the length of the longest subsequence of A which is a balanced string of parentheses.

4. Suppose that you start with the number $x = 1$, and you may repeatedly choose between incrementing x (increasing its value by 1) and tripling x (multiplying its value by 3). Your goal is to reach a target number y using as few of these steps as possible. For example, if y is 411, then the optimal number of steps is 9, and the nine step sequence to get x from 1 to 411 is:

$$1 \to 3 \to 4 \to 5 \to 15 \to 45 \to 135 \to 136 \to 137 \to 411.$$

Give an algorithm which takes as input a positive integer y and outputs the minimum number of steps to obtain y (starting from $x = 1$) using this process. The running time of your algorithm should be polynomial in the length of the input (i.e. your running time should be $O((\log y)^c)$ for some constant c). Make sure to prove the correctness of your algorithm.

5. In the class selection problem, we are given as input an array A of n classes, each with a start time and an end time, and the goal is to choose a maximum size subset of the classes such that no two are running at the same time (i.e. for any distinct classes x and y which are chosen, either $x.end \leq y.start$, or $y.end \leq x.start$).

In this problem, we consider 5 different greedy algorithms for this problem. For each of the following algorithms, either *prove* that it always returns an optimal schedule, or give a concrete

example (i.e. a list of start and end times) on which the algorithm does not generate an optimal schedule. Assume that each algorithm has the base cases of "If there are no classes, return the empty schedule", and "If there is only one class, choose that class." Assume that ties are broken arbitrarily (i.e. in a manner outside of your control). You do not need to analyze the runtime of these algorithms.

(a) Choose the class x that starts last, discard the classes that conflict with x, and recurse.

(b) Choose the class x that ends last, discard the classes that conflict with x, and recurse.

(c) If no classes conflict, choose them all. Otherwise, discard the class x with the longest duration and recurse.

(d) If no classes conflict, choose them all. Otherwise, discard the class x which conflicts with the most other classes and recurse.

(e) If any class x is completely contained in another class y (i.e. y starts before x and ends after x) then discard y and recurse. Otherwise, choose the class z that ends last, discard the classes that conflict with z and recurse.

6. Suppose that you are given a base-10 positive integer x with n-digits as well as a number $k < n$, and you want to return the smallest number that can be obtained by removing exactly k digits from x. e.g. if $x = 1432219$ and $k = 3$, you should return 1219. Give an $O(n)$ time algorithm for this problem.

7. An *Egyptian fraction* is a sum of a finite sequence of fractions which each have numerator 1, and which have *distinct* positive integer denominators e.g. $\frac{1}{3} + \frac{1}{10} + \frac{1}{17}$. Give an algorithm which takes as input a rational number $r = p/q$, where $p, q \in \mathbb{Z}^+$ with $p < q$, encoded as a numerator-denominator pair. Your algorithm should return an Egyptian fraction which is equal to r. Your Egyptian fraction should use at most p total unit fractions. For example, if $r = 5/6$, you could return $1/2 + 1/3$. You could also return $1/2 + 1/4 + 1/12$. You could not return $1/6 + 1/6 + 1/6 + 1/6 + 1/6$, because the denominators of these fractions are not distinct. Make sure to prove the correctness of your algorithm.

   Fun fact 1: The earliest known algorithm for representing an arbitrary rational number as an Egyptian fraction was developed in 1202 by Fibonacci (the same mathematician after whom the Fibonacci numbers are named).

   Fun fact 2: While this problem only asks you to represent p/q with an Egyptian fraction using at most p unit fractions, it is possible to give an algorithm that uses $O(\log \log q)$ unit fractions. This was proven in a September 2026 OpenAI preprint.

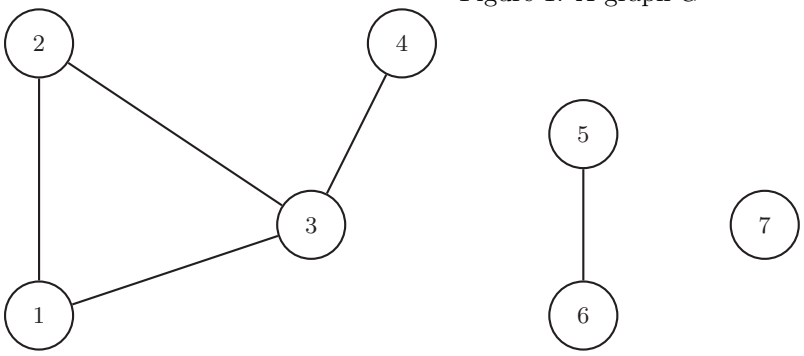
13 Graphs

In this chapter, we introduce *graphs*, a structure that we will continue to use heavily for the rest of the course.

13.1 A Whole Lot of Definitions

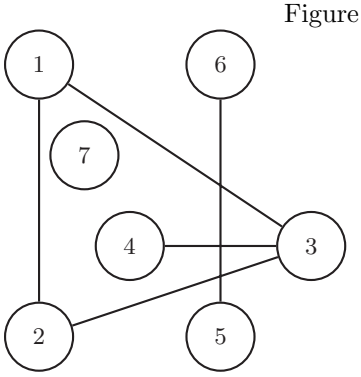
A **graph** is an ordered pair (V, E), where V (called the "vertices") is a non-empty finite set and E (called the "edges") is a set of pairs of vertices from V, i.e. every element of E is a set $\{u, v\}$ of size 2, where $u, v \in V$. When discussing a single graph, we will use n to denote $|V|$, the number of vertices and m to denote $|E|$, the number of edges. We use $V(G)$ to denote the vertex set of a graph G and $E(G)$ to denote the edge set of G. We call u and v the **endpoints** of the edge $e = \{u, v\}$. We say that a vertex v is **incident** to an edge e if $v \in e$.

Figure 1: A graph G



We represent a graph pictorially by using circles for vertices and lines connecting them for edges. Figure 1 depicts a graph $G = (V, E)$ where the vertex set is $V := \{1, 2, 3, 4, 5, 6, 7\}$ and the edge set is $E := \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{5, 6\}\}$. The placement of the vertices in space is arbitrary, all that matters is which vertices are connected by edges. Generally we try to draw a graph in a way which minimizes crossing edges to make this as easy to see as possible, but this is not a requirement. Figure 2 depicts the exact same graph G as Figure 1.

Figure 2: The same graph G as the previous figure

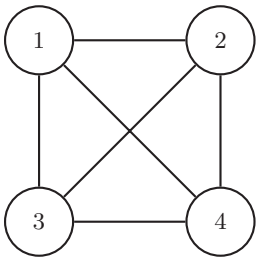


For a vertex (singular of vertices) v, we define the **degree** of v, written $\deg(v)$ as the number of edges incident to v. For example, in our graph G from the figures, $\deg(1) = \deg(2) = 2$, while $\deg(3) = 3$, $\deg(4) = \deg(5) = \deg(6) = 1$, and $\deg(7) = 0$. We call a vertex of degree 0 an **isolated vertex** (thus, 7 is an isolated vertex in G).

We define the **neighbors** of v as the vertices which share an edge with v (not including v itself). We use $N(v)$ to denote the set of neighbors of v. In our example G, $N(1) = \{2, 3\}, N(3) =$

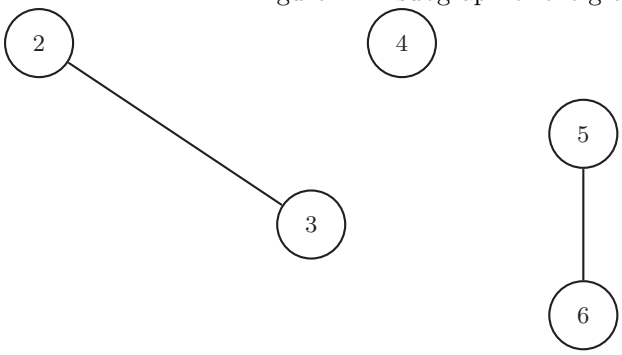
$\{1, 2, 4\}, N(4) = \{3\}$, and $N(7) = \emptyset$. Note that by definition $\deg(v) = |N(v)|$. We say that a graph is **k-regular** if all of its vertices have degree k. If G is an $(n-1)$-regular graph on n vertices, we call G a **complete graph**. We denote the complete graph on n vertices by K_n.

Figure 3: The four vertex complete graph K_4



We say that a graph G' is a subgraph of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. For example, the graph in Figure 4 is a subgraph of the graph in Figure 1. Note that every graph is a subgraph of itself.

Figure 4: A subgraph of the graph from Figures 1 and 2



A **walk** is a sequence of vertices $v_1, \ldots, v_k$ such that for every $1 \leq i < k$, $\{v_i, v_{i+1}\} \in E$ (i.e. there is an edge between each pair of consecutive vertices in the walk). A **path** is a walk which does not repeat any vertices. If a walk begins at u and ends at v, we call it a "walk from u to v" or a $u \to v$ walk. Similarly a path that begins at u and ends at v is called a $u \to v$ path. The length of a walk or path is the number of edges it traverses, i.e. one less than the number of vertices in the sequence.

For example, in the graph from Figure 1, 1,3,4,3,1,2,3 is a walk of length 6, which is not a path. 4,3,1,2 is a path of length 3. 5,6 is a path of length 1. Any single vertex, e.g. 2, forms a path of length 0. All of these paths are also walks.

We say that two vertices u and v are **connected** if there exists a $u \to v$ path. A **connected component** is a maximal set of vertices, all of which are connected to each other. The graph in Figure 1 has three connected components: 1,2,3,4, 5,6, and 7. If all of the vertices of G are in the same connected component we say that G is **connected**.

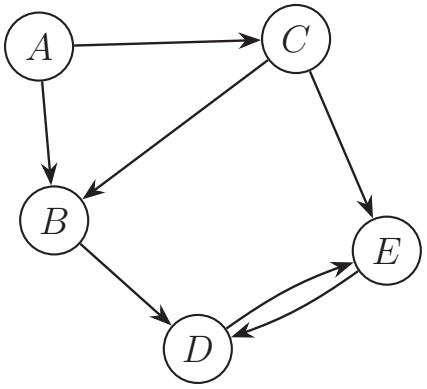
A **cycle** is a sequence of vertices $v_1, \ldots, v_k, v_1$, such that $v_1, \ldots, v_k$ is a path, and $\{v_k, v_1\}$ is an edge not connecting consecutive vertices in the path (note that this implies that a cycle in an undirected graph has at least 3 distinct vertices). The only cycle in the graph from Figure 1 is 1,2,3,1, which has length 3. Note that we could also write this cycle in other ways, such as 2,1,3,2. Since all these sequences correspond to the same set of vertices and edges, we consider all of them the same cycle. If G contains a cycle, we call G **cyclic**. Otherwise, we call G **acyclic**. The graph in Figure 1 is cyclic, while the graph in Figure 4 is acyclic.

13.2 Graph Variations

There are several important variants of the graph definition which occur commonly in computer science. In particular, we can add directions to the edges, in which case we have a **directed** graph, add weights to the edges, in which case we have a **weighted** graph, or allow multiple edges between the same pair of vertices, as well as self-loops, in which case we have a **multigraph**. These three variations can be used in any combination. We could, for example, have a directed multigraph. The type of graph we discussed in the previous section is sometimes called an undirected, unweighted simple graph to emphasize that it does not include any of these variations. In this course, however, we will generally omit these qualifiers. Whenever we talk about a graph we mean an undirected graph unless we explicitly specify a directed graph.

A directed graph is like an ordinary graph but instead of each edge being a set of two vertices, an edge is an ordered pair of vertices (u, v). We represent a directed edge with an arrow from u to v. Note that while in an undirected graph we can only have one pair of edges between each pair of vertices, in a directed graph we can have two edges between the same vertices, because (u, v) and (v, u) are different edges. We depict an edge (u, v) in a directed graph using an arrow from u to v, as shown in Figure 5.

Figure 5: A directed graph



Many of the definitions from the previous section extend naturally to directed graphs, but there are a few differences. While a vertex in an undirected graph has a single set of neighbors, vertices in directed graphs have **in-neighbors** and **out-neighbors**, where $N_{in}(v) := \{u \in V \mid (u, v) \in E\}$ and $N_{out}(v) := \{u \in V \mid (v, u) \in E\}$. Similarly, we define the in-degree of v as $\deg_{in}(v) := |N_{in}(v)|$ and the out-degree of v as $|N_{out}(v)|$. In the graph from Figure 5, $N_{in}(C) = \{A\}$ and $N_{out}(C) = \{B, E\}$ so $\deg_{in}(C) = 1$ and $\deg_{out}(C) = 2$.

While a cycle in an undirected graph must have at least three vertices, in a directed graph it is possible to have a 2-vertex cycle in a directed graph. For example, D, E, D is a cycle in the graph in Figure 5. Unlike in an undirected graph, the existence of a $u \to v$ path in a directed graph does not imply the existence of a $v \to u$ path. There are several ways to extend the concept of connectivity to directed graphs. The one which will be relevant for this course is called strong connectivity. We say that two vertices u and v are **strongly connected** if G contains both a $u \to v$ path and a $v \to u$ path. A strongly connected component is a maximal set of vertices, every pair of which is strongly connected. The graph in Figure 5 contains four strongly connected components: $\{A\}, \{B\}, \{C\}$, and $\{D, E\}$.

A multigraph is like a graph but instead of being a set of edges, E is a multiset of edges, meaning that it can contain multiple copies of the same edge. We also allow a multigraph to contain loops: edges where both endpoints are the same vertex. Note an ordinary graph can be viewed as a special case of a multigraph. Some graph algorithms extend naturally to multigraphs.

A weighted graph is an ordered pair (G, w) where G is a graph and $w : E(G) \to \mathbb{R}$ is a function assigning a "weight" to each edge of G. What these weights represent will depend on the specific problem we are interested in. Unless otherwise specified, we will assume, as we've done for array

elements, that the weights of edges in our graphs are polynomial in the number of vertices of vertices in the graph (i.e. that the weights for an n-vertex graph can be represented with $O(\log n)$ bits) so that we do not need to worry about the runtime for arithmetic when adding weights.

13.3 Proofs with Graphs

In this section, we will see some examples of proofs involving graphs, and prove some useful graph properties.

There is a close relationship between the sum of the degrees of the vertices in a graph and the number of edges in that graph. The following theorem (often known as the "handshake lemma") expresses this relationship. It was originally proven by Leonhard Euler in a paper which initiated the modern study of graph theory.

Theorem 61. *1. In every graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$.*

2. In every directed graph $G = (V, E)$, $\sum_{v \in V} \deg_{out}(v) = \sum_{v \in V} \deg_{in}(v) = |E|$.

Proof. We can prove each of these properties with a double counting argument, where we show that each edge is counted the same number of times on each side of the equation.

1. Let $G = (V, E)$ be a graph. We note that $\sum_{v \in V} \deg(v)$ counts each edge $\{u, v\}$ exactly twice—once in $\deg(u)$ and once in $\deg(v)$. Thus $\sum_{v \in V} \deg(v) = 2|E|$.

2. Let $G = (V, E)$ be a directed graph. We note that each edge is an out-edge of exactly one vertex, so $\sum_{v \in V} \deg_{out}(v)$ counts each edge exactly once. Similarly, each edge is an in-edge of exactly one vertex, so $\sum_{v \in V} \deg_{in}(v)$ also counts each edge exactly once. Thus, $\sum_{v \in V} \deg_{out}(v) = \sum_{v \in V} \deg_{in}(v) = |E|$.

□

We can answer other questions about the degrees in a graph easily using Theorem 61.

Theorem 62. *There does not exist a 3-regular graph with 5 vertices.*

Proof. In a 3-regular graph $G = (V, E)$ with n vertices, we have that $\sum_{v \in V} \deg(v) = \sum_{v \in V} 3 = 3n$, so by Theorem 61, $3n = 2|E|$, and thus $|E| = \frac{3}{2}n$. If $n = 5$, then $\frac{3}{2}n = \frac{15}{2} = 7.5$. Since the number of edges on a graph must be an integer, we conclude that a 3-regular graph with 5 vertices cannot exist. □

Now let's prove some results about walks and paths.

Theorem 63. *Let $G = (V, E)$ be a graph and $u, v \in V$ be vertices. If G contains a $u \to v$ walk, then G contains a $u \to v$ path.*

Proof. Let $W = v_1, v_2, \ldots, v_k$ be a *shortest* $u \to v$ walk in G (note that $v_1 = u$ and $v_k = v$). Assume for sake of contradiction that W is not a path. Then W must have some repeated vertex, i.e. there exist distinct integers $1 \leq i < j \leq k$ such that $v_i = v_j$. Then $W' := v_1, v_2, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k$ is also a $u \to v$ walk (each pair of consecutive vertices in W' also occur consecutively in W, so since W is a walk there must be an edge between them). We note that W' is shorter than W, since we have omitted the vertices $v_{i+1}, \ldots, v_j$ (and this is at least one vertex since $i < j$). Thus, W' contradicts our choice of W as the shortest $u \to v$ walk, and so our initial assumption that the shortest walk was not a path, must have been false. Therefore, there exists a $u \to v$ path in G. □

We note that Theorem 63 also holds for directed graphs with basically the same proof.

Theorem 64. *Let $G = (V, E)$ be a graph and $u, v, w \in V$ be distinct vertices. If G contains a $u \to v$ walk, and a $v \to w$ walk, then G contains a $u \to w$ path.*

Proof. By Theorem 63, since G has a $u \to v$ walk it has a $u \to v$ path $P_1 = u_1, u_2, \ldots, u_k$, and since it has a $v \to w$ walk it has a $v \to w$ path $P_2 = v_1, v_2, \ldots, v_\ell$. Let x be the first vertex of P_1 which also appears in P_2 (we know that P_1 and P_2 must have at least one common vertex since $u_k = v_1 = v$). Let i, j be integers such that $u_i = x = v_j$ (these are uniquely defined since P_1 and P_2 are paths and thus do not repeat vertices).

Then $P := u_1, u_2, \ldots, u_{i-1}, x, v_{j+1}, v_{j+2}, \ldots, v_\ell$ is a $u \to w$ path. In particular, we know that none of the vertices $u_1, \ldots, u_{i-1}$ are the same as any of the vertices $v_{j+1}, \ldots, v_\ell$ because we chose x as the *first* vertex which is in both P_1 and P_2. □

13.4 Trees

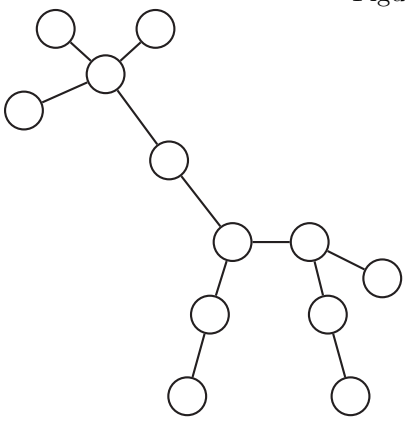
A tree is a graph $T = (V, E)$ which satisfies the following three properties:

- T is connected
- T is acyclic
- $m = n - 1$ (the number of edges in T is exactly one less than the number of vertices).

In fact to show that a graph is a tree it suffices to show only two of these properties, because having any two implies having the third, we will prove this later.

You are likely familiar with *binary* trees from your data structures course, but trees in graph theory are not the same. By default, a tree is not rooted, meaning that there is no special root vertex and no notion of one vertex being a "child" of another. Furthermore, we put no restriction on the number of neighbors that a vertex in a tree can have. Figure 6 depicts my favorite tree, called the G-raph, because it a graph in the shape of a giraffe (invented by one of my friends in undergrad).

Figure 6: The G-raph, my favorite tree



We call a degree-1 vertex in a tree a **leaf**. The G-raph has 6 leaves.

Trees are important because they connect a set of vertices using as few edges as possible. We will discuss applications of this more when we talk about minimum spanning trees in a few chapters, for now we will just gain some practice working with trees by proving the following theorem.

Theorem 65. *Any n-vertex graph $T = (V, E)$ which satisfies two of the following three properties satisfies the third one as well:*

- *T is connected*
- *T is acyclic*
- *$m = n - 1$ (the number of edges in T is exactly one less than the number of vertices).*

Proof. We have three claims to prove:

1. First we show that if T is acyclic with $m = n - 1$, then T is connected.

   Consider building T by adding its edges one at a time, starting from an empty graph (one with all of T's vertices, but no edges).

   Initially this graph has n connected components. Every time we add an edge which connects two vertices from different connected components, they merge and the number of connected components decreases by one.

   If we were to add an edge connecting two vertices u and v from the same connected component, this would form a cycle: since u and v were connected, T contained a $u \to v$ path, and adding the edge $\{v, u\}$ to this path gives a cycle. Since T is acyclic, we know that this will never happen. Every edge we add must merge two connected components.

   Since T begins with n connected components and every edge we add merges two of them, after adding all $n - 1$ edges, T will have $n - (n - 1) = 1$ connected component, meaning that all of T is connected.

2. Next we prove that if T is connected with $m = n - 1$, then T is acyclic.

   We use a similar argument to the previous part. Suppose we create T by adding edges 1 at a time starting from an empty graph. Each edge we add can merge at most two connected components. In order to get from n components down to 1 with only $n - 1$ edges, every single edge we add must merge two different connected components.

   An edge $\{u, v\}$ that connects two different connected components cannot form a cycle (if it did that would imply that u and v were already connected via the rest of the cycle and therefore were in the same connected component). Thus, none of the edges we add can create a cycle and so T is acyclic.

3. Finally we show that if T is connected and acyclic then T has $n - 1$ edges.

   To illustrate how to apply induction to graphs, we prove this by induction on n.

   - Base Case: If $n = 1$, then the only possible graph consists of a single vertex and no edges. This graph is connected and acyclic and it has $n - 1$ edges.
   - Induction Hypothesis: Suppose that for every $k < n$, every k-vertex graph which is connected and acyclic has exactly $k - 1$ edges.
   - Induction Step: Let G be an n-vertex tree. Since G is acyclic and has at least one edgge (since it is connected and $n > 1$) it must contain a vertex of degree 1 (otherwise we could obtain a cycle by walking around the graph arbitrarily exiting each vertex with a different edge than the one we used to enter until we repeat a vertex). Removing this degree 1 vertex yields a graph on $n - 1$ vertices which is still connected and acyclic, so by our IH, this graph has $n - 2$ edges. Adding back the removed vertex and edge we find that G has $n - 1$ edges as desired.

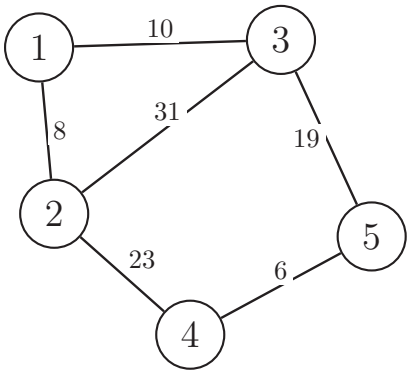
□

13.5 Representing Graphs

In order to analyze the runtime of algorithms on graphs, we need to define how we are encoding graphs as strings (since under our RAM model the input and output of every algorithm are just strings). There are two main ways of encoding graphs: adjacency matrices and adjacency lists. We will discuss both of them here (there are many other ways of encoding graphs which are useful in more specific applications, but we will not get into those in this course). In this section we will assume that the vertex set is $V := \{1, \ldots, n\}$.

In the adjacency matrix representation, we encode $G = (V, E)$ with an $n \times n$ array, where $A[i, j] = 1$ if $\{i, j\} \in E$ and 0 otherwise. Note that the adjacency matrix of an undirected graph is

Figure 7: A weighted graph



symmetric. In contrast for a directed graph we define $A[i, j]$ to be 1 if $(i, j) \in E$, so $A[i, j] = A[j, i]$ only if the directed graph has directed edges between i and j in both directions.

For a weighted graph, we can compactly encode the weights by letting $A[i, j] = w(\{i, j\})$ if $\{i, j\} \in E$ (if our graph can include 0-weight edges, however, we will have to select some value other than 0 to represent "this edge does not exist"). For example, the weighted graph in Figure 7 can be represented by the adjacency matrix:

$$\begin{bmatrix} 0 & 8 & 10 & 0 & 0 \\ 8 & 0 & 31 & 23 & 0 \\ 10 & 31 & 0 & 0 & 19 \\ 0 & 23 & 0 & 0 & 6 \\ 0 & 0 & 19 & 6 & 0 \end{bmatrix}$$

An adjacency matrix takes $O(n^2)$ space. It allows us to check whether a particular edge $\{u, v\}$ is present in the graph in time $O(1)$, we just check $A[u, v]$. Iterating through all of the neighbors of a vertex v takes $\Theta(n)$ time, however, since we need to loop over all of the entries in row v of the adjacency matrix.

Unfortunately for the humble adjacency matrix, iterating through the neighbors of a vertex is a much more common component of graph algorithms than checking for the presence of a random edge. Nevertheless, adjacency matrices are valuable to understand, because there are many contexts when an input which we want to treat as a graph is represented in an adjacency-matrix like form (e.g. if we are just given a weight function which takes two vertices and returns the weight of the connection between them, then we can treat this function as an adjacency matrix).

The more common way to represent graphs is using an adjacency list. Here we use an array A of size n, where $A[i]$ is a linked list containing all of the neighbors of vertex i. If G is a directed graph, then $A[i]$ is a linked list of the out-neighbors of i. If G is a weighted graph, then instead of the nodes in our linked lists storing just vertices, they store (vertex, weight) pairs, where the weight of the nodes storing j in $A[i]$ is the weight of the edge $\{i, j\}$. The graph in Figure 7 would be represented with the following adjacency list:

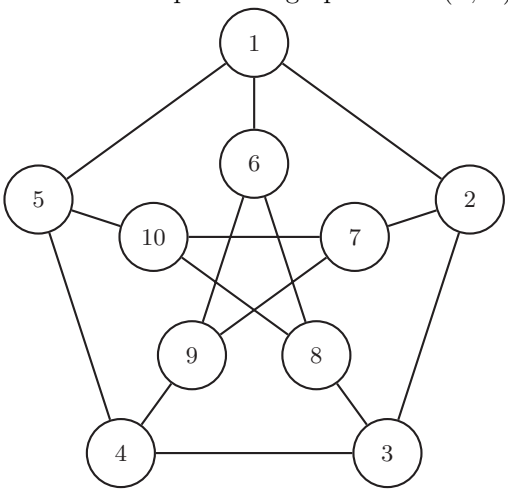
$$\begin{aligned} A[1] &: (2, 8) \to (3, 10) \\ A[2] &: (1, 8) \to (3, 31) \to (4, 23) \\ A[3] &: (1, 10) \to (2, 31) \to (5, 19) \\ A[4] &: (2, 23) \to (5, 6) \\ A[5] &: (3, 19) \to (4, 6) \end{aligned}$$

With an adjacency list, checking whether a particular edge $\{u, v\}$ exists takes $O(\deg(u))$ time, since we need to iterate through u's entire linked list, which has $\deg(u)$ nodes. Iterating through

the neighbors of v, however, takes only $O(\deg(v))$ time, a substantial improvement over adjacency matrices unless our graph is very dense (meaning the average degree is $\Omega(n)$). In this course when an algorithm outputs a graph or takes one as input, we will assume that it is represented as an adjacency list unless otherwise specified.

13.6 Comprehension Questions

1. Consider the pictured graph $G = (V, E)$ (this graph is known as the Petersen graph):



   (a) How many vertices does this graph have?
   (b) How many edges does this graph have?
   (c) What is $N(6)$ in this graph?
   (d) What is $\deg(5)$ in this graph?
   (e) Give an example of a cycle in this graph. What is its length?
   (f) Give an example of a path of length 6 in this graph.
   (g) Give an example of a walk of length 12 in this graph.
   (h) Does this graph contain K_3 as a subgraph?
   (i) Is this graph 2-regular?
   (j) Represent G as an adjacency matrix
   (k) Represent G as an adjacency list

2. Let G be a graph with 400 edges. What is the sum of the degrees of all the vertices in G?

3. Let $G = (V, E)$ be a directed graph and let $u, v \in V$. Which of the following are necessarily true? (Select all that apply)

   (a) If G contains a $u \to v$ walk then G contains a $u \to v$ path
   (b) If G contains a $u \to v$ path then G contains a $u \to v$ walk
   (c) If G contains a $u \to v$ walk then G contains a $v \to u$ walk
   (d) If G contains a $u \to v$ path then G contains a $v \to u$ walk

4. Which of the following is NOT a property of every tree T?

   (a) T is connected
   (b) T is acyclic
   (c) T has a root

(d) $m = n - 1$ (the number of edges in T is exactly one less than the number of vertices).

5. What is the runtime of checking whether an edge $\{u, v\}$ exists in a graph G...

   (a) ...assuming G is represented by an adjacency list?

   (b) ...assuming G is represented by an adjacency matrix?

13.7 Exercises

1. Prove that K_n has $\frac{n(n-1)}{2}$ edges.

2. Let G be a graph with minimum degree d (meaning that a vertex of G with the smallest degree has degree d). Prove that G has a path of length d.

3. Let T be a tree with at least two vertices. Prove that T has at least two leaves.

4. Let $G = (V, E)$ be a graph. Prove that G is a tree if and only if for every pair of distinct vertices $u, v \in V$ there is a unique path from u to v in G.

5. Let G be a graph with $n \geq 2$ vertices. Prove that G contains two vertices with the same degree.

6. Let G be a graph where every edge has one endpoint with even degree and one endpoint with odd degree. Prove that G has an even number of edges.

7. Prove that every 2-regular graph is composed of one or more disjoint cycles.

8. Prove that every tree with $n \geq 2$ vertices has $2 + \sum_{v \in V : \deg(v) \geq 3} (\deg(v) - 2)$ leaves.

9. Let G be a graph, let C be a cycle in G, let k be a positive integer and let P be a path of length k between two vertices of C. Prove that G contains a cycle of length at least $\sqrt{k}$.

14 Depth First Search

In this chapter we will see our first major graph algorithm and explore the surprisingly large number of problems that it can be applied to.

14.1 Connectivity

Recall from Chapter 13 that we say two vertices u and v in a graph G are connected if G contains a path from u to v. We say that G itself is connected if every pair of vertices in G is connected (i.e. if G has only one connected component).

One of the most natural questions about a vertex is whether two vertices are connected. For example, in a graph representing a transit network, we may want to determine whether it's possible to get from point A to point B. In a graph representing genealogies, we may want to determine whether two people are related, i.e. whether they are connected through family links that we can trace.

So how can we determine if two vertices a and b are connected? We can try to solve this problem recursively, using the techniques from Chapter 10. If a and b are the same vertex, then they are connected. Otherwise, a is connected to b if and only if, one of a's neighbors is connected to b. Based on this we could give the following algorithm:

```
WanderAround(a, b)
  If a = b:
      Return True
  For each vertex c ∈ N(a):
      If WanderAround(c, b):
          Return True
  Return False
```

If WanderAround returns True, then we know that a and b are connected, and if it returns False, we know that a and b are not connected. This algorithm has a very serious correctness issue, however. Do you see what it is? (Think back to Chapter 7)

The problem is that returning True and returning False are not the only things that this algorithm can do. It can loop forever! Nothing in the algorithm prevents us from wandering back to vertices that we've already visited, so we can wander from a vertex u to a vertex v, and then wander from v right back to u. In fact, if a and b are not connected, but a has at least one neighbor, then WanderAround is guaranteed to get caught in an infinite loop, wandering through the same vertices over and over without ever finding b.

To fix this, we need some way to keep track of which vertices we've already been to so that we don't visit them again. If we continue to follow our standard recursive backtracking model, we could handle this by modifying our recursive function to take a list of all the vertices we've already used on the current path as an additional parameter (i.e. pass along a summary of our previous choices), as shown below:

```
WanderBetter(a, b, S = ∅)
  If a = b:
      Return True
  Add a to S
  For each vertex c ∈ N(a) \ S:
      If WanderBetter(c, b, S):
          Return True
  Return False
```

Here S is the set of all the previous vertices on our path from a. WanderBetter always terminates, which is an improvement over WanderAround, but it's runtime is quite bad. The number of subsets of V (i.e. $|2^V|$) is exponential in n. If G is a complete graph, then we can visit the vertices in any order, so we do end up with exponentially many subproblems. Dynamic programming

can't help us here either, since each subproblem has different parameters (in particular a different set S).

A better approach is to use a global array to keep track of all of the vertices that have been visited by any of our recursive calls. We'll assume here and for the rest of the chapter that $V = \{1, \ldots, n\}$. If this is not the case, we can assign each vertex a number from 1 to n arbitrarily and use these numbers as the indices into our global arrays.

```
visited ← an array of n copies of False
DepthFirstSearch(v)
  If not visited[v]:
      visited[v] ← True
      For each vertex w ∈ N(v):
          DepthFirstSearch(w)
```

Theorem 66. DepthFirstSearch(v) *visits exactly the vertices which are connected to v. (i.e. when the algorithm terminates, visited[x] is True if x is in v's connected component, and False otherwise). Furthermore,* DepthFirstSearch(v) *runs in time $\Theta(m' + n')$, where n' is the number of vertices in v's connected component and m' is the number of edges between these vertices.*

Proof. To show that DepthFirstSearch(v) visits exactly the vertices connected to v we need to show that the algorithm visits every vertex connected to v and that every vertex the algorithm visits is connected to v. We can prove both directions by induction:

- First we prove that every vertex connected to v is visited by the algorithm. Let y be a vertex which is connected to v. Then there exists a $v \to y$ path. Let k be the length of this path. We will prove that DepthFirstSearch(v) visits y by induction on k.
  - Base Case; If $k = 0$, then $y = v$, so DepthFirstSearch(v) immediately visits y.
  - Induction Hypothesis: Suppose that for every $0 \leq \ell < k$, DepthFirstSearch(v) visits all vertices which can be reached from v by a path of length ℓ.
  - Induction Step: Let $v_1, \ldots, v_{k+1}$ be the length k path from v to y. Then v_k is connected to v by a path of length $k - 1$. By our IH, this means that DepthFirstSearch(v) visits v_k. When this happens, the algorithm will recursively call DepthFirstSearch on each neighbor of v_k, including $v_{k+1} = y$, so y is visited.
- Now we prove that every vertex the algorithm visits is connected to v. We will prove by induction on i that the i'th vertex the algorithm visits is connected to v.
  - Base Case: The first vertex the algorithm visits is v. v is connected to itself.
  - Induction Hypothesis: Suppose that the first $i - 1$ vertices the algorithm visits are all connected to v.
  - Induction Step: Every recursive call the algorithm makes is to a neighbor of a vertex it has already visited. Therefore, if y is the i'th vertex visited, y is a neighbor of some vertex x which was visited earlier. By our IH, x is connected to v. Thus, since y is connected to x by the edge $\{x, y\}$, y is also connected to v.

Now we analyze the runtime of DepthFirstSearch. Let C be the subgraph of G consisting of the vertices in v's connected component and the edges between them. As we showed above, DepthFirstSearch(v) visits exactly the vertices in C. The first time that DepthFirstSearch is called on a vertex u, it performs $\Theta(\deg(u))$ work, outside of the recursive calls, since the algorithm iterates through the neighbors of u, which takes $\Theta(\deg(u))$ time with an adjacency list. Each subsequent call to DepthFirstSearch on the same vertex takes constant time, since DepthFirstSearch will note that the vertex has already been visited and immediately halt.

Thus, each vertex u is responsible for $\deg(u)$ recursive calls and the calls on that vertex take $\Theta(1 + \deg(u))$ total time. Therefore the total time across all recursive calls is $\Theta(\sum_{u \in C} 1 + \deg(u)) = \Theta(n' + \sum_{u \in C} \deg(u)) = \Theta(m' + n')$, by the handshake lemma. □

From Theorem 66, it follows that we can use DepthFirstSearch to check whether two vertices a and b are connected by calling DepthFirstSearch(a) and then checking afterwards if visited[b] is true. Moreover, we can use DepthFirstSearch to find the entire connected component of a given vertex (it's just all the vertices visited by the DFS).

What if we want to find all of the connected components in a graph? In this case we can use multiple calls to depth first search. Every time the depth first search finishes, we restart from a new vertex that we haven't visited yet. The algorithm below implements this idea. We replace the "visited" array with a global array "component" to store an integer representing the component of each vertex (vertices whose number in this array are the same are part of the same component).

```
FindComponents(G = (V, E))
  component ← an array of n 0s
  numComponents ← 1
  For each v ∈ V:
      If component[v] = 0:
          FindComponentsHelper(v)
          numComponents ← numComponents + 1
  Return component

FindComponentsHelper(v)
  If component[v] = 0:
      component[v] ← numComponents
      For each vertex w ∈ N(v):
          FindComponentsHelper(w)
```

Theorem 67. FindComponents(G) *returns an array "component" such that for every pair of vertices $u, v \in V(G)$, component[u] = component[v] if and only if u and v are in the same connected component of G. Furthermore,* FindComponents *runs in $O(m+n)$ time.*

Proof. We note that FindComponentsHelper runs DepthFirstSearch except that instead of marking vertices as visited, it labels them with a component number. By the same arguments we used in proving Theorem 66, the first call that FindComponents makes to FindComponentsHelper will find a component and label all of its vertices with component 1. Then the algorithm will find a vertex not part of this first component (i.e. one that still has a component label of 0) and run DFS from it, this time labeling each visited vertex with component 2. This will continue until all vertices are labeled with their component.

By Theorem 66, each call that FindComponents makes to FindComponentsHelper runs in time $\Theta(m' + n')$, where n' is the number of vertices in the component found and m' is the number of edges. Since each vertex and each edge is in exactly one component, the total time for these calls across all components is $\Theta(m+n)$. The work of FindComponents outside of the recursive calls can all be done in $O(n)$ time, so the total runtime is $\Theta(m+n)$. □

So far we have discussed DFS for undirected graphs, but DFS is also useful for directed graphs. For a vertex v in a directed graph $G = (V, E)$, we define reach(v) := $\{w \in V \mid G$ contains a $v \to w$ path$\}$ to be the set of all vertices reachable from v (Note that these vertices may not have paths back to v, so this is not a connected component). We can compute reach(v) in linear time using the directed version of DFS given below.

```
visited ← an array of n copies of False
DirectedDFS(v)
  If not visited[v]:
      visited[v] ← True
      For each vertex w ∈ N_out(v):
          DirectedDFS(w)
```

Theorem 68. DirectedDFS(v) *visits exactly the vertices in reach(v) in $O(m+n)$ time.*

Proof. The proof is basically the same as the one we used for Theorem 66. We can prove that every vertex $w \in \text{reach}(v)$ is visited by induction on the length of the $v \to w$ path. We can prove that only vertices in $\text{reach}(v)$ are visited by induction on the number of recursive calls (visiting an out-neighbor of a vertex in $\text{reach}(v)$ always gives us another vertex in $\text{reach}(v)$).

Since our DFS visits at most all n vertices and explores at most all m edges, the runtime is $O(m+n)$. □

Finding strongly connected components in a directed graph is a bit more complicated, we will discuss this problem in Section 14.4.

14.2 Finding Cycles

Recall from Chapter 13 that a cyclic graph is one which contains at least one cycle, while an acyclic graph is one which does not contain any cycles. Determining whether or not a graph is acyclic is a common problem which has several applications. For example, we know that a tree is connected and acyclic. We may want to check if a graph has these properties to determine if it is a tree. Directed acyclic graphs are useful for representing dependencies (e.g. between tasks). If we have a set of tasks to complete, some of which depend on other tasks being completed first, we would like to make sure that there are no circular dependencies, i.e. that there is a valid order to complete the tasks in.

We can detect cycles with DFS. We present the algorithm here for directed graphs, but a similar approach also works for undirected graphs. We use a second array "onPath" to keep track of the vertices on the path that we followed to reach the vertex we are currently visiting. Then, if we ever see an edge back to an earlier vertex on the path, we know that G has a cycle.

```
CycleDetector(G = (V, E))
    visited ← an array of n copies of False
    onPath ← an array of n copies of False
    For each v ∈ V:
        If CycleDetectorHelper(v):
            Return True
    Return False

CycleDetectorHelper(v)
    If not visited[v]:
        visited[v] ← True
        onPath[v] ← True
        For each vertex w ∈ N_out(v):
            If onPath[w]:
                Return True
            Else if CycleDetectorHelper(w):
                Return True
        onPath[v] ← False
    Return False
```

Theorem 69. CycleDetector(G) *returns True if and only if G has a cycle. Furthermore,* CycleDetector(G) *runs in $O(m+n)$ time.*

Proof. We need to prove two directions.

- First we show that if G has a cycle then CycleDetector(G) returns True. Let C be a cycle in G. Let v be the first vertex of C that we call CycleDetectorHelper on. When we visit v, we add it to our current path, setting onPath[v] to True. We will only set onPath[v] back to False after we have fully explored all the out-neighbors of v. By Theorem 68, since CycleDetectorHelper is performing a depth-first search, we know that it will visit all of the vertices in $\text{reach}(v)$ before the call to CycleDetectorHelper(v) returns (unless it returns True early by finding a different cycle). In particular, if it has not already returned True, CycleDetectorHelper will visit the vertex u that precedes v in the cycle.

Thus, we will visit u while onPath[v] is True. At this point, CycleDetectorHelper will see the edge (u, v) and return True.

- Now we will show that if CycleDetector(G) returns True, then G has a cycle. The key observation is that the vertices for which onPath is true always form a path ending at the vertex we are currently visiting (every time we visit a new vertex we add it to our path, every time we backtrack, we remove the vertex we left from our path). Thus, if we ever encounter an edge from the vertex v that we are visiting to a vertex w with onPath[w] True, we can form a cycle in G by following the (v, w) edge, then following the path that our recursive calls used to reach v from w.

The runtime analysis for CycleDetector is similar to the analysis we did when proving Theorem 67. CycleDetector is simply running DFS from all vertices with a constant amount of additional work per vertex to update the onPath array. Since the visited array is shared across all runs of DFS, the total amount of work done per vertex v is $\Theta(1 + \deg(v))$, so the total time for all the runs of DFS is only $O(m + n)$. □

14.3 Topological Sort

In the previous section we gave the example of task dependencies as an application for checking if a graph is acyclic. When we have a dependency graph, however, often we don't just want to know that it is acyclic, we also want to find an order in which we can perform the tasks that satisfies all of the dependencies.

For example, given a list of packages, we might want to find an order to install them such that when we try to install a package we've already installed the packages it requires. Or given a list of courses we might want to find an order to take them in such that before we take a course we've taken its prerequisites. Both of these are examples of the topological sorting problem.

Formally, we define a topological sort of a directed acyclic graph $G = (V, E)$ as an ordering (permutation) of the vertices of V such that for every edge $(u, v) \in E$, u occurs before v in the ordering. Note that only an acyclic graph can have a topological sort, since no matter how we order a cycle, the vertex we put last will have an edge to one of the earlier vertices.

We can find a topological sort using depth first search by simply modifying our search algorithm to add a vertex to the beginning of our ordering after the algorithm has completely finished exploring the vertex.

```
TopSort(G = (V, E))
  R ← an empty list
  visited ← an array of n copies of False
  For each v ∈ V:
      TopSortHelper(v)
  Return R

TopSortHelper(v)
  If not visited[v]:
      visited[v] ← True
      For each vertex w ∈ N_out(v):
          TopSortHelper(w)
      Prepend v to the beginning of R
```

Theorem 70. *For every directed acyclic graph $G = (V, E)$, TopSort(G) returns a topological sort of G.*

Proof. Note that TopSortHelper(v) adds v to R the first time it is called. Subsequent calls to TopSortHelper(v) do not change R, because v is already marked as visited. Thus, every unique vertex we visit with TopSortHelper gets added to R exactly once. Since TopSort calls TopSortHelper on every vertex, this means that when TopSort(G) returns, R is an ordered list of the vertices of G where each vertex appears exactly once.

It remains to show that this ordering is actually a topological sort, i.e. that for every edge (u, v), u occurs before v in R. Let $(u, v) \in E$. Since G is acyclic, we know that G does not contain a $v \to u$ path (because such a path plus the (u, v) edge would form a cycle). Thus, $u \notin \text{reach}(v)$. We consider two cases:

- First suppose that TopSortHelper(v) is called before TopSortHelper(u). Since TopSortHelper is just performing DFS with the additional step of adding v to R at the end, by Theorem 68, the recursive calls made by TopSortHelper(v) will visit only vertices in reach(v). But as we noted above, $u \notin \text{reach}(v)$. Thus, when the for loop in TopSortHelper(v) ends, u will still be unvisited, and therefore will not be in R. Thus, v will be prepended to R first, meaning that in the final vertex ordering, u occurs before v.
- Now suppose that TopSortHelper(u) is called before TopSortHelper(v). In this case, TopSortHelper(u) will recursively call TopSortHelper(v). Thus, the for loop in TopSortHelper(u) will not end until after the first call to TopSortHelper(v) has returned and v has been added to R. Thus, u will be prepended to R later than v, and thus u will occur before v in the final ordering.

In either case, u occurs before v in R. Thus, every edge of G has its endpoints appear in the correct order in R, so R is a topological sort.

□

14.4 Strongly Connected Components

Recall that two vertices u and v in a directed graph G are strongly connected if G has both a $u \to v$ path and a $v \to u$ path. A strongly connected component is a maximal set of vertices that are all strongly connected to each other.

We said in section 14.1 that while DFS can compute the connected component of a vertex in an undirected graph, it does not compute the strongly connected component of a vertex in a directed graph. For example if G consists of just a long path, then running DirectedDFS from the first vertex s in the path will visit all of the vertices in the path, but since none of the other vertices has a path back to s, the strongly connected component of s is just s itself.

In order to actually find the strongly connected component of v, we need to know not just which vertices v can reach, but also which vertices can reach v. The way to compute this is to use the **reverse** of G, which is defined as the graph obtained by flipping the direction of all of the edges in G (i.e. for every edge $(u, v) \in E(G)$, reverse(G) instead has the edge (v, u)). We note that the reach of v in G' is exactly the set of vertices which have paths to v in G. This gives us the following algorithm for computing the strongly connected component of a vertex:

```
StronglyConnectedComponent(v)
    visited ← an array of n copies of False
    DirectedDFS(v)
    reachV ← visited
    Reverse all of the edges of G
    visited ← an array of n copies of False
    DirectedDFS(v)
    Return {u ∈ V | reachV[u] = True} ∩ {u ∈ V | visited[u] = True}
```

We note that this algorithm runs in time $O(m + n)$ since we are making two calls to DFS, each of which takes $O(m + n)$ time, and the other operations of the algorithm can each be done in $O(n)$ time.

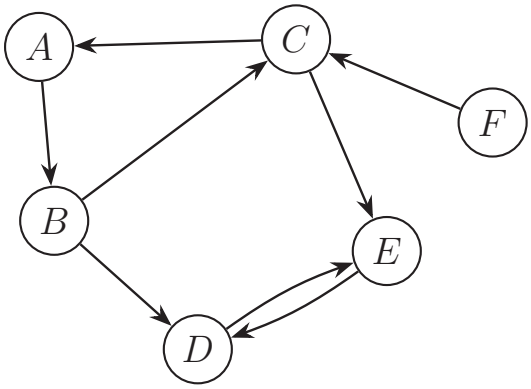
The meta-graph of a directed graph G is a graph G' where the vertices are strongly connected components of G. There is an edge (C_1, C_2) between two components C_1 and C_2 in the meta-graph if and only if there is an edge from some vertex in C_1 to some vertex in C_2 in G. The meta-graph of G is always a DAG (since if it had a cycle, the strongly connected components in the cycle could

all be merged into a single strongly connected component). We can compute the meta-graph in $O(n(m+n))$ time by repeating StronglyConnectedComponent once from each vertex.

With a bit more effort it is possible to compute all strongly connected components with a single depth first search that does some extra bookkeeping, giving a runtime of $O(m+n)$ for computing the meta-graph. Tarjan's algorithm, developed by Robert Tarjan in 1972, was the first algorithm to do this. You may use this algorithm as a subroutine in algorithms that you write, but we will not describe or analyze it here. See Section 6.6 of Erickson if you are interested in the details.

14.5 Comprehension Questions

1. Consider the following graph:



   (a) What is reach(C) in this graph?

   (b) List the strongly connected components of this graph.

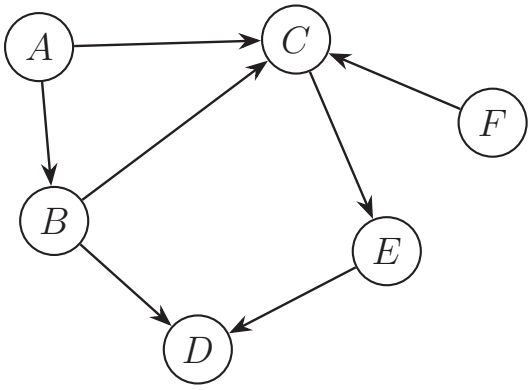
   (c) Is this graph a DAG?

2. True or false? Running DirectedDFS(v) on a vertex v in a directed graph will visit all of the vertices in v's strongly connected component.

3. True or false? If G is a directed acyclic graph, then all of G's vertices form a single strongly connected component.

4. What is the runtime of counting the number of connected components in an undirected graph, using the results from this chapter?

5. Give a topological sort of the following graph:



14.6 Exercises

1. A *Hamiltonian path* in a (directed or undirected) graph G is a path that visits every vertex of G. Prove that if a directed graph G has a *unique* topological sort, then G has a Hamiltonian path.

2. In Section 14.2 we saw an algorithm to determine whether or not a graph G contains a cycle in $O(m+n)$ time. Now suppose that we are given an undirected graph $G = (V, E)$ and a vertex $v \in V$ and want to determine whether or not G has a cycle containing the vertex v. Give an $O(m+n)$ time algorithm for this problem.

3. Prove that if G is a DAG then the following algorithm correctly finds a topological sort of G. Make sure to justify why the algorithm will always be able to find a vertex of in-degree 0.

```
SourceRemover(G)
  If |V(G)| = 1
      Return the sequence containing the lone vertex of G
  s ← a vertex in V(G) such that deg_in(s) = 0
  G' ← G with vertex s removed
  R ← SourceRemover(G')
  Prepend s to the beginning of R
  Return R
```

4. Consider the following variant of the job scheduling problem from Section 12.2: We are given a set of n jobs J which our machine needs to run. Each job j has three attributes:

   - *j.time* is the time that j takes our machine to complete (measured e.g. in minutes)
   - *j.deadline* is the time by which we are supposed to complete j (measured in minutes since the time when we start running jobs).
   - *j.dependencies* is a (possibly empty) list of other jobs that must be completed before running j

   (a) Give an $O(n^2)$ time algorithm to determine whether it is possible to complete the jobs in an order which satisfies all of the dependencies.

   (b) Give an $O(n^2)$ time algorithm to find an ordering of the jobs which satisfies all of the dependencies and has the minimum possible maximum lateness out of all orderings that satisfy the dependencies (Hint: start by choosing which job to schedule last).

5. Your friend's Go club (`https://en.wikipedia.org/wiki/Go_(game)`) recently held a tournament in which players could freely play games against whoever was available, without any assigned bracket or rounds. Unfortunately, the club forgot to come up with a system for determining who won the event. Knowing that you are an algorithms expert, the club president has come to you for help. Suppose that you are given as input a list of the n go club members and a list of k tournament games, where each game is a triple consisting of which player played the black pieces, which player played the white pieces, and whether white or black won. Do not make any assumptions about which members played each other. Some players may have more games than others, and a given pair of players may have played against each other once, more than once, or not at all.

   (a) The club president has asked you to rank all the players based on their performance in the tournament. To avoid any complaints about the methodology, the president wants a ranking (i.e. an ordered list of the club members) such that nobody is ranked higher than someone who beat them in the tournament. That is, if player A won against player B in a tournament game, then player A must be ranked higher than player B. Give an efficient algorithm to determine whether there exists such a ranking, given the results from the tournament.

(b) Give an efficient algorithm to generate a ranking satisfying the condition in part (a), assuming such a ranking exists.

(c) It turns out that there is no way to rank the members which satisfies the club president's condition, so the president has decided to allow ties in the ranking (e.g. several players may share 2nd place). Nevertheless, the president would like to minimize the number of players who tie for their position. Give an efficient algorithm to compute a rank for each player such that no player has a strictly better ranking than someone who beat them in a tournament game, and the total number of players who share their rank with somebody else is minimized.

15 Shortest Paths: Part 1

In the shortest $s \to t$ path problem, we are given a directed graph $G = (V, E)$ and two vertices $s, t \in V$. Our goal is to find a minimum length path from s to t. We will focus on the directed graph version of the problem because any algorithm for shortest paths in directed graphs can easily be applied to undirected graphs if we simply replace each undirected edge with two directed edges, one in each direction.

The shortest $s \to t$ path problem has natural practical applications, for example in a maps app that needs to give the user efficient directions between their location and a desired destination. Interestingly the fastest algorithms for the shortest $s \to t$ path problem can also solve a more general problem: Single source shortest paths. In the single source shortest path problem, we are given a graph $G = (V, E)$ and a single vertex s. The goal is to find the shortest path from s to every other vertex.

There are many natural variants on the shortest paths problem. The most common one is to allow the input graph to have edge weights. In a weighted graph we define the length of a path as the sum of the weights of the edges in the path, rather than just the number of edges in the path. In this chapter we will see efficient algorithms for single source shortest paths in unweighted graphs as well as graphs with non-negative edge weights. In the next chapter we will look at how to handle negative edge weights and explore an even broader generalization of the shortest $s \to t$ path problem.

15.1 Shortest Paths in Unweighted Graphs: BFS

Intuitively, breadth first search is simple. First we label s itself with a distance of 0. Then we label all the neighbors of s with a distance of 1. Then we label all of the unlabeled neighbors of those neighbors with a distance of 2. Then we label all the unlabeled neighbors of those vertices with a distance of 3. We continue this process until we've labeled every vertex in the graph with its distance from s.

While this algorithm can give us the distance of each vertex from s, it's not immediately clear how to recover the actual shortest paths from s to each vertex. Returning an array of n paths could require $\Omega(n^2)$ space and thus $\Omega(n^2)$ time (suppose that G is just a long path and s is the first vertex. Then the shortest paths have length $1, \ldots, n-1$, so by Theorem 3 their total length is $n(n-1)/2$). There is a more efficient approach, however, which takes advantage of the fact that shortest paths overlap a lot.

Note that shortest paths exhibit optimal substructure: To find the shortest path from s to t, we just need to choose which neighbor u of t comes immediately before t in the path, then find a shortest path from s to u. The shortest path from s to t will then be the shortest $s \to u$ path plus the $u \to t$ edge (using anything but the shortest path to get from s to u would just make our $s \to t$ path longer). Therefore, to keep track of all the shortest paths, we just need to remember the *predecessor* of each vertex: the vertex immediately before it in a shortest path. Storing a predecessor takes only $O(n)$ space. Given the predecessor list, we can read out the shortest path to any given t by following the predecessors until we get to s. If we view the predecessor of each vertex as its parent in a rooted tree, we get a tree rooted at s which we call a **shortest path tree**.

We give the following algorithm based on these ideas. BreadthFirstSearch uses an array D to store the distance of each vertex from s and an array *pred* to store the predecessor of each vertex.

```
BreadthFirstSearch(G = (V, E), s)
    D, pred ← length n arrays
    For each v ∈ V:
        D[v] ← ∞
        pred[v] ← Null
    D[s] ← 0
    Frontier ← {s}
    NextFrontier ← ∅
    While Frontier ≠ ∅:
        For each vertex u ∈ Frontier:
            For each vertex v ∈ N_out(u):
                If D[v] = ∞ :
                    D[v] ← D[u] + 1
                    pred[v] ← u
                    Add v to NextFrontier
        Frontier ← NextFrontier
        NextFrontier ← ∅
    Return D, pred
```

Theorem 71. *For every directed graph $G = (V, E)$ and vertex $s \in V$,* BreadthFirstSearch(G, s) *returns arrays D and pred such that for every $v \in V$, $D[v] = dist(v)$ and (if $dist(v) \neq \infty$ and $v \neq s$) a shortest $s \to v$ path ends with the edge $(pred[v], v)$. Furthermore,* BreadthFirstSearch(G, s) *runs in time $O(m + n)$.*

Proof. We claim that the while loop in BreadthFirstSearch satisfies the following invariant. After i iterations of the loop:

1. $D[v] = dist(v)$ for every vertex v with $dist(v) \leq i$.
2. $D[v] = \infty$ for every vertex with $dist(v) > i$.
3. Frontier contains exactly the vertices v with $dist(v) = i$.

We prove this by induction on i:

- Base Case: This is true at the start of the first iteration (i.e. after 0 iterations):
    1. We set $D[s]$ to 0 before the loop and s is the only vertex as distance 0.
    2. We set $D[v]$ to ∞ for all other vertices before the loop.
    3. At the start of the first iteration Frontier contains only s.
- Induction Hypothesis: After $i - 1$ iterations of the loop, every vertex v with $dist(v) \leq i - 1$ has $D[v] = dist(v)$, every vertex v with $dist(v) \geq i$ has $D[v] = \infty$, and Frontier contains exactly the vertices with $dist(v) = i - 1$.
- Induction Step: Let $i > 0$. Let v be a vertex with $dist(v) = i$. Let u be the vertex preceding v on a shortest $s \to v$ path. We note that $dist(u) = dist(v) - 1 = i - 1$. Thus, by our IH, at the start of iteration i of the loop, $D[u] = i - 1$, $D[v] = \infty$, and u is in Frontier. Also by our IH, we know that every vertex x in Frontier has $D[x] = i - 1$. Thus, if v is assigned a new value in D in this iteration, it will be assigned the correct value of $(i - 1) + 1 = i$. Since u is in the Frontier, we know that v will be assigned a new value in D this iteration, since it will be explored as a neighbor of u if it wasn't already explored as a neighbor of some other vertex in Frontier. Thus, v will be added to NextFrontier, which becomes Frontier at the end of the loop iteration. Therefore, at the end of iteration i, v will be in Frontier and have $D[v] = i$. Since v was an arbitrary vertex at distance i, this implies that at the end of iteration i, every vertex at distance i will have the correct value in D and will be in the Frontier.

  It remains to show that no vertex v with $dist(v) > i$ will have its value in D updated in iteration i or get added to the Frontier. To show this, let v be a vertex assigned a value in D

during iteration i. Since we only assign a new D value to vertices whose current value in D is ∞, v must not have been assigned a finite value in D previously. By our IH, this implies $dist(v) \geq i$. Additionally, to be visited in this iteration, v must be a neighbor of a vertex u which was in Frontier at the start of iteration i. By our IH, u must satisfy $dist(u) = i - 1$. Thus, we can obtain a path of length i to v by following the length $i - 1$ $s \to u$ path and then the (u, v) edge, so $dist(v) \leq i$. We conclude that $dist(v) = i$. Thus, the vertices we assign values in D and add to the frontier this iteration of the loop are exactly the ones at distance i, and all three parts of our loop invariant hold.

This loop invariant immediately implies that when BreadthFirstSearch returns, $D[v] = dist(v)$ for every vertex. We note that when the algorithm assigns u to $pred[v]$, u is a neighbor of v with $D[u]$ (and thus $dist(u)$, by our invariant) equal to $D[v] - 1 = dist(v) - 1$. Therefore, there is indeed a shortest path to v which ends by following the $(pred[v], v)$, and so the pred array defines a valid shortest path tree.

Finally, we analyze the runtime. The work before the while loop takes $O(n)$ time (to initialize v and $pred$). By our loop invariant, each vertex v gets added to Frontier at most once (in particular, v is added to Frontier in iteration $dist(v)$ of the loop). When the algorithm examines a vertex v from Frontier it performs $O(1 + \deg_{out}(v))$ work iterating through the neighbors of v. The extra work of the while loop per iteration is constant, and the while loop runs for at most n iterations (A shortest path can include at most all of the vertices, so no vertex reachable from s can be at distance $\geq n$). Thus the total work of the while loop is $O(n + \sum_{v \in V}(1 + \deg_{out}(v))) = O(m + n)$, and so the whole algorithm runs in $O(m + n)$ time. □

15.2 Shortest Paths with Non-Negative Edge Weights: Dijkstra's

Now we consider weighted graphs. In particular, graphs with *non-negative* edge weights (we'll see how to handle negative edge weights next chapter). The algorithm we will use for this problem is called Dijkstra's algorithm after Edsger Dijkstra who discovered it in the 1950s (possibly first, although several other researchers independently developed the same algorithm around the same time).

Like BFS, Dijkstra's creates an array D of distance values and an array $pred$ representing the parent relationships in a shortest path tree. Unlike in BFS, we may need to update the entries in D and $pred$ multiple times. Throughout the algorithm we will maintain the invariant that $D[v] \geq dist(v)$ (i.e. $D[v]$ is an upper bound on the distance to v), and there exists an $s \to v$ path of length $D[v]$ which ends with the edge $(pred[v], v)$ (assuming $D[v] < \infty$). Initially these invariants are true because $D[s] = 0$ and $D[v] = \infty$ for every other vertex (infinity is a very safe upper bound on $dist(v)$).

For a given array of distance upper bounds D, we say that an edge (u, v) is *tense* if $D[v] > D[u] + w(u, v)$. If an edge is tense, we can improve our distance upper bounds by running the following simple procedure, which we refer to as **relaxing** the edge:

```
Relax(e = (u, v))
    D[v] ← D[u] + w(u, v)
    pred[v] ← u
```

We note that relaxing an edge preserves our invariants for D and $pred$: If there is an $s \to u$ path of length $D[u]$, then we can obtain an $s \to v$ path of length $D[u] + w(u, v)$ by simply following the $s \to u$ path then following the edge (u, v).

Note that we can view the step in BreadthFirstSearch where we update $D[v]$ and $pred[v]$ as relaxing the edge (u, v) if we view our unweighted graph as a graph where all edge weights are 1. For arbitrary edge weights, however, we cannot simply use the expanding frontier idea from BFS. We need to be open to revisiting a vertex when we discover a path that uses more edges but has shorter total length.

The core idea of Dijkstra's algorithm is to repeatedly relax all of the out edges of the vertex with minimum $D[v]$ among those whose out edges we haven't relaxed yet. In order to efficiently find the vertex with minimum $D[v]$ we will use a data structure called a priority queue. You should have

seen priority queues before in CSCE 221, but it's okay if you don't remember them. The important thing to know is that a priority queue contains (element, priority) pairs and efficiently supports the operations INSERT for adding an element to the queue, DECREASEKEY for reducing the priority value of an existing element, and EXTRACTMIN, which removes and returns the element with the smallest priority value. We will use $D[v]$ as the priority value for v, so that EXTRACTMIN will get the vertex from our queue with minimum $D[v]$. We give the algorithm below:

```
DIJKSTRA'S(G = (V, E), s)
  D, pred ← length n arrays
  For each v ∈ V:
      D[v] ← ∞
      pred[v] ← Null
  D[s] ← 0
  Q ← a priority queue containing (s, 0)
  While Q is not empty:
      u ← Q.EXTRACTMIN()
      For each vertex v ∈ N_out(u):
          If (u, v) is tense:
              RELAX(u, v)
              If v ∉ Q
                  Q.INSERT(v, D[v])
              Else:
                  Q.DECREASEKEY(v, D[v])
  Return D, pred
```

Theorem 72. *For every weighted directed graph $G = (V, E)$ with non-negative edge weights and every vertex $s \in V$,* DIJKSTRA'S(G, s) *returns arrays D and pred such that for every $v \in V$, $D[v] = dist(v)$ and (if $dist(v) \neq \infty$ and $v \neq s$) a shortest $s \to v$ path ends with the edge $(pred[v], v)$.*

We will sketch the proof of Theorem 72, see CLRS or Erickson for a full proof (you will not be expected to know the details of this proof).

Let v_i be the vertex that DIJKSTRA'S extracts from Q in iteration i of the while loop, and let $V_i := \{v_1, v_2, \ldots, v_i\}$. The correctness of Dijkstra's flows from a few core observations:

- Each vertex which is reachable from s eventually gets inserted into the priority queue (This follows by induction on the length of the shortest path to the vertex, using the observation that an edge (u, v) where $D[u]$ is finite and $D[v]$ is ∞ will always be tense).
- After each iteration of the while loop, for each vertex v, $D[v]$ is the length of the shortest path from s to v using only vertices in $V_i \cup \{v\}$. (We can prove this inductively together with the following points)
- When we extract v from Q, $D[v] = dist(v)$. (Since all edge weights are non-negative, the shortest path to v can only use vertices whose distance from s is $\leq dist(v)$)
- Once a vertex is extracted from the priority queue it never gets reinserted (This follows from the previous point. If $D[v] = dist(v)$, then no edge to v can become tense again, because that would imply a shorter path to v)
- $dist(v_i) \leq dist(v_j)$ if $i \leq j$ (This follows from the previous two points and the fact that we always remove the vertex with minimum $D[v]$)

Theorem 73. *If G has non-negative edge weights, then Dijkstra's can be implemented to run in time $O(m + n \log n)$.*

Proof. By the observations above, each vertex gets extracted from the priority queue at most once, so DIJKSTRA'S makes $O(n)$ calls to EXTRACTMIN and $O(n)$ calls to INSERT. Since an edge can only be relaxed when we remove its first vertex from the queue, this also means that we relax each

edge at most once, and thus we make $O(m)$ calls to DecreaseKey. The additional work done by the algorithm can be accomplished in $O(m+n)$ time, so the overall runtime is $O(n)$ times the runtime of ExtractMin plus $O(n)$ times the runtime of Insert plus $O(m)$ times the runtime of DecreaseKey.

With the binary heap implementation of priority queues that you wrote in CSCE 221, all three of these operations run in $O(\log n)$ time, so we get an overall runtime of $O(n \log n + m \log n)$. We can do better, however. If we implement our priority queue using a Fibonacci heap, then Insert and DecreaseKey both run in $O(1)$ time (amortized), while ExtractMin remains $O(\log n)$. Thus, using this implementation, the runtime for Dijkstra's becomes $O(m + n \log n)$. This is not a data structures course, so we will not discuss how to implement a binary heap, let alone a Fibonacci heap, but you may freely assume a Fibonacci heap implementation of priority queues and therefore assume that Dijkstra's runs in $O(m + n \log n)$ time. □

15.3 Shortest Paths with Constraints: Graph Transformations

Many times in the real world, we face variations of the shortest path problem with additional constraints. That is, there is some property P that we need our $s \to t$ path to satisfy, and we want to find the shortest $s \to t$ path from among those with property P. For example, in addition to their length, edges may have a cost, and we want to find the shortest path that isn't too expensive. Or perhaps we need to stop for gas at some point along a trip, so we want to find the shortest route from s to t which passes a gas station.

We could solve problems like this by developing a new algorithm which is some variant of BFS or Dijkstra's and proving that it correctly computes the shortest path with our desired property. Typically an easier approach, however, is to modify our input graph into a format where we can directly apply a graph algorithm we know. This way, we only need to analyze the correctness and runtime of making our modification to the graph, we don't need to worry about a complicated induction proof like the ones we used to prove the correctness of our algorithms in the previous two sections.

One particularly useful technique for modifying graphs is called *graph layering* and it consists of creating multiple copies (layers) of the vertices from the original graph, and strategically choosing which edges should go between vertices in the same layer and which should go between vertices in different layers.

For example, consider the gas station constraint mentioned above. Suppose we are given a list L of n locations and an $n \times n$ array where $A[i, j]$ stores the time to drive from $L[i]$ to $L[j]$ (these times are all non-negative). We are also given a list $S \subseteq L$ of $k \leq n$ locations that are gas stations, as well as a starting location s and an ending location t. We want to find the shortest path from s to t that stops at a gas station.

One way to solve this problem would be to use Dijkstra's algorithm to calculate the shortest path from s to each gas station and then to calculate the shortest path from each gas station to t. If there are many gas stations, however, then this approach may be rather slow. We might need to run Dijkstra's $\Omega(n)$ times (remember that running Dijkstra's once can get us the shortest $s \to g$ path for every gas station $g \in S$, but it does not get us the shortest path from each gas station to t)

Instead, we can solve this problem with only one call to Dijkstra's if we modify the input graph. We create a weighted directed graph G as follows:

- The vertex set is $L \times \{0, 1\}$. That is, each vertex is a pair of a location and an integer. Vertices of the form $(\ell, 0)$ will represent being at location ℓ before we've made our stop for gas. Vertices of the form $(\ell, 1)$ will represent being at location ℓ after we've stopped for gas

- The edge set contains all edges of the form $((L[i], x), (L[j], x))$ where $1 \leq i, j \leq n$, $x \in \{0, 1\}$, and $i \neq j$. The weight of the edge $((L[i], x), (L[j], x))$ is $A[i, j]$. That is, all the original edges of our graph appear in both layers. Additionally, our edge set contains all edges of the form $((S[i], 0), (S[i], 1))$ where $1 \leq i \leq k$. These edges have weight 0. These additional edges allow us to freely reach layer 1 when we're at a gas station.

To find the shortest $s \to t$ path that stops for gas, we simply need to find the shortest $(s, 0) \to (t, 1)$ path in this graph G. Since the only places we can transition from layer 0 to layer 1 in G are at gas stations, every $(s, 0) \to (t, 1)$ path must stop for gas, so we only need the one call to Dijkstra's.

Whenever we use a graph transformation like this we have to be careful when analyzing the runtime. We cannot simply say that this algorithm runs in $O(m \log n)$ time because that's the runtime of Dijkstra's. After all, the runtime of Dijkstra's depends on the number of vertices and edges in the graph that we run it on, not on the size of the inputs to our original problem. We need to figure out how the size of our constructed graph relates to the size of our input(s). We also need to take into account the time to construct the new graph.

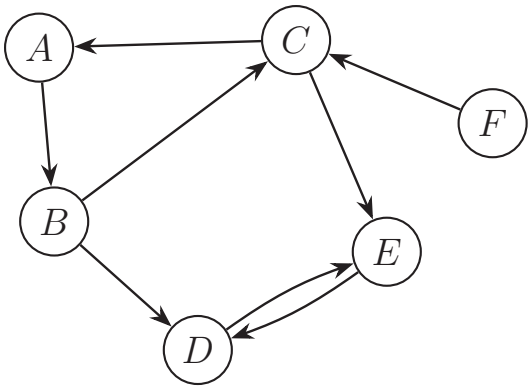
In this case, our new graph has $2n$ vertices and $2n(n-1)+k$ edges (each layer has $n(n-1)$ edges within the layer, and there are k edges between the layers). Since $k \leq n$ we can say that the number of vertices is $O(n)$ and the number of edges is $O(n^2)$. Thus, running Dijkstra's on this graph takes time $O(n^2 + n \log n) = O(n^2)$. Constructing the graph can also be done in $O(n^2)$ time, since we only need to perform constant work per vertex and edge, so the overall runtime of this approach is $O(n^2)$.

Sidenote: We could also solve this particular problem without graph layering while still using only a constant number of calls to Dijkstra's. The key idea is to create the reverse graph G^R (see Section 14.4) and run Dijkstra's starting from t in G^R to get the shortest path from each vertex to t.

15.4 Comprehension Questions

1. After running BreadthFirstSearch on the graph below starting from vertex A, what will $pred[D]$ be?

2. After 1 iteration of the while loop of BreadthFirstSearch on this graph, Frontier is $\{B\}$. What will Frontier be after the next iteration of the loop?



3. True or false? While running BreadthFirstSearch (on an unweighted graph G and a vertex s in G), if $pred[v] \neq Null$, then $D[v] = dist(v)$.

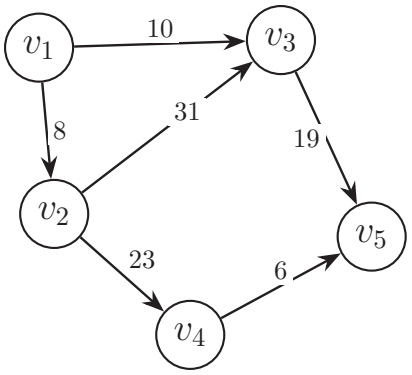
4. Suppose that we are computing shortest paths with Dijkstra's algorithm and we are about to relax the edge (u, v). Currently $D[u] = 10$, $D[v] = 15$, and $w(u, v) = 3$. Which of these will change after we relax (u, v), and what will the new value be?

5. What priority queue data structure can we use to get Dijkstra's algorithm to run in $O(m+n \log n)$ time? What runtime does it give us for ExtractMin?

6. Suppose that we are trying to compute shortest paths from A in the graph below (not necessarily with Dijkstra's) and we have the current distance upper bounds:

$$D[v_1] = 0, D[v_2] = 8, D[v_3] = 39, D[v_4] = 31, D[v_5] = 37.$$

Which edges in the graph are currently tense?



15.5 Exercises

1. In this problem, we consider a variation of the shortest path problem. You are given a graph where some of the edges are blue and others are red, and your goal is to find the shortest path which doesn't use too many red edges. More formally, suppose that you are given as input a (undirected, unweighted) graph $G = (V, E)$, a set $R \subseteq E$ of red edges, a positive integer k, and two vertices $s, t \in V$. Give an algorithm which returns a minimum-length path from s to t among those which use at most k red edges, or correctly reports that there is no s to t path in G which uses at most k red edges. The runtime of your algorithm should be $O(k(m + n))$.

2. You are given an directed weighted graph (G, w) with positive edge weights where each edge e has an associated color $c(e) \in \{\text{red, green, blue}\}$. You are also given two vertices $s, t \in V(G)$. We say that a walk $v_1, \ldots, v_k$ in G is *tricolor* if it alternates between red, green, and blue edges–i.e. for every $1 \leq i < k$,

   - If $i \mod 3 = 1$, then (v_i, v_{i+1}) is red
   - If $i \mod 3 = 2$, then (v_i, v_{i+1}) is green
   - If $i \mod 3 = 0$, then (v_i, v_{i+1}) is blue

   Give an efficient algorithm to find the shortest tricolor walk from s to t in G. If there is no tricolor walk from s to t, your algorithm should correctly report this.

3. Suppose you are interested in finding the shortest paths from a single vertex s in a strongly connected weighted, directed graph (G, w), where the edge weights are arbitrary integers (could be positive, negative, or zero). A suspicious-looking stranger comes up to you and says that they've already solved this problem. To prove it, they show you a spanning tree T for G rooted at s, which they claim is a shortest path tree. You don't trust this person, so you'd like to check their work, in a way that's faster than simply computing the shortest path tree yourself. Give an algorithm which takes as input G, w, s and T, and determines in $O(m)$ time (where m is the number of edges in G) whether or not T is in fact a shortest paths tree for G rooted at s.

4. A *vertex-weighted graph* is a pair (G, w) where G is a (directed or undirected) graph, and w is a function assigning a weight to each *vertex* of G (w can be represented by an array or a dictionary). The weight (also called the length) of a path in a vertex-weighted graph is the sum of the weights of the vertices in the path.

   Show how to solve the shortest paths problem in vertex-weighted graphs. In particular, for each of the following restrictions on (G, w), give an algorithm which takes as input a vertex weighted graph (G, w) and a vertex s and returns a shortest paths tree rooted at s. Each of your algorithms should be efficient as possible, asymptotically (you should analyze the big-Θ runtime of your algorithms, but you do not need to prove that your algorithm's runtime is the best possible)

(a) First, suppose that (G, w) is a *directed* vertex-weighted graph where every vertex weight is *positive*.

(b) Now suppose that (G, w) is an *undirected* vertex-weighted graph where every vertex weight is *positive*.

5. In the land of Algorithmica, all roads are toll roads. At the end of any trip, you must make a cash payment to the government equal to the total tolls of the roads that you traveled on. John has a map of Algorithmica showing n cities connected by m toll roads. The map also lists the toll $t(r)$ for each road r, which is a positive integer number of cents. John would like to find the cheapest route between city s and city t. Unfortunately, John does not have any pennies, and the toll collectors are very picky and only accept exact change. Give an efficient algorithm to find the cheapest route from s to t whose cost is a multiple of 5.

16 Shortest Paths: Part 2

In this chapter we will look at how to compute shortest paths in graphs where edge weights may be negative. In particular, we will see an algorithm to solve the single source shortest path problem which is efficient even when edges can have arbitrary weights. Then we will look at an even more general shortest-path problem, where we want to find the shortest path between *every* pair of vertices and give a clever dynamic programming algorithm to solve it.

16.1 Negative Weights

Negative weights introduce some new complexities to the concept of a shortest path. Recall that a path in a graph cannot repeat vertices. In many practical applications of shortest path algorithms this is an unnatural constraint. If going in a circle somehow allowed you to get from point A to point B faster, people would be willing to do it. That is, often what we are actually interested in computing is often a shortest *walk*. In a graph with non-negative edge weights, this difference doesn't matter. Repeating a vertex can never make our walk shorter, so for any s and t, there will be a shortest $s \to t$ walk which is a path (there could theoretically also be shortest $s \to t$ walks which are not paths, if the graph has a zero-weight cycle).

With negative weights, however, the notion of a shortest walk sometimes stops making sense. In particular, if our graph contains a *cycle* of negative weight which is reachable from s, then we can obtain arbitrarily short walks from s by walking to the cycle, walking around it until we feel like our walk length is sufficiently negative, and then walking to our destination. Thus, even when dealing with negative weights, we prefer to focus on graphs without negative-weight cycles (in these graphs it remains the case that some shortest $s \to t$ walk is always a path). It's not obvious how to determine whether a graph has a negative-weight cycle, but we'll see an algorithm that can do this (in addition to efficiently solving single source shortest paths with negative weight edges) in the next section.

Although Theorem 72 only stated the correctness of Dijkstra's algorithm for non-negative edge weights, Dijkstra's algorithm does correctly compute a shortest path tree even for graphs with negative edge weights, as long as the graph does not have a negative-weight cycle. The runtime, however, can be extremely bad, $\Theta(2^n)$ in the worst case (See Section 8.6 of Erickson for an example where Dijkstra's runs very slowly). With negative edges, we no longer have a guarantee that each vertex is extracted from our priority queue at most once, so we can end up relaxing the same edges over and over and over again. Note that if G has a negative-weight cycle, then Dijkstra's algorithm can run forever, continually relaxing edges round and round the cycle. We need a better approach if we want to quickly find shortest paths in graphs with negative edge weights.

16.2 Negative Edge Weights: Bellman-Ford

Perhaps surprisingly, the algorithm for handling negative edge weights is simpler than Dijkstra's algorithm. Indeed the algorithm we will describe in this section was actually discovered before Dijkstra's algorithm. It was first described by Alfonso Shimbel in 1955, but was popularized by Richard Bellman and Lester Ford Jr. who described it in later papers and is today known as the Bellman-Ford algorithm.

Dijkstra's algorithm can be viewed as a greedy algorithm, focusing on relaxing the edges of whichever vertex currently has the lowest distance upper bound. The Bellman-Ford algorithm, on the other hand, is a brute force algorithm. Rather than strategically selecting edges to relax, the algorithm simply relaxes all edges in the graph, and then repeats this process until there are no more tense edges. As we will prove below, this takes at most $n - 1$ iterations, so if there are still tense edges after $n - 1$ rounds of relaxation our algorithm can correctly report that there is a negative cycle.

```
BELLMAN-FORD(G = (V, E), s)
    D, pred ← length n arrays
    For each v ∈ V:
        D[v] ← ∞
        pred[v] ← Null
    D[s] ← 0
    Repeat n − 1 times:
        For each edge (u, v) ∈ E:
            If (u, v) is tense:
                RELAX(u, v)
    If G contains a tense edge:
        Return "Negative Cycle!"
    Return D, pred
```

Theorem 74. *For every weighted directed graph $G = (V, E)$ and every vertex $s \in V$:*

- *If G contains a negative-weight cycle reachable from s,* BELLMAN-FORD(G, s) *correctly reports this.*
- *Otherwise,* BELLMAN-FORD(G, s) *returns arrays D and pred such that for every $v \in V$, $D[v] = dist(v)$ and (if $dist(v) \neq \infty$ and $v \neq s$) a shortest $s \to v$ path ends with the edge $(pred[v], v)$.*

Furthermore, BELLMAN-FORD(G, s) *runs in time $\Theta(mn)$*

Proof. We first note that if G has a negative-weight cycle reachable from s, then no matter how much we relax edges, G will always have a tense edge. (every time we relax an edge in the cycle, the next one becomes tense). Thus, if G has a negative-weight cycle reachable from s, BELLMAN-FORD will correctly report this.

Now we will assume that G has no negative-weight cycles reachable from s. We will prove by induction on i that after i iterations of the "Repeat $n-1$ times" loop, $D[v] = dist(v)$ for every vertex v such that some shortest $s \to v$ path (i.e. the one of minimum total weight) uses at most i edges.

- Base Case: The only vertex whose shortest path from s uses 0 edges is s itself. Since we set $D[s]$ to 0 before the loop, after 0 iterations of the loop we have that $D[v] = dist(v)$ for every vertex v for which the shortest $s \to v$ path has at most 0 edges.
- Induction Hypothesis: Suppose that after $i-1$ iterations of the loop, $D[v] = dist(v)$ for every vertex v such that some shortest $s \to v$ path uses at most $i-1$ edges.
- Suppose $i > 0$. By our IH, at the start of iteration i of the loop, every vertex v for which some shortest $s \to v$ has fewer than i edges already has $D[v] = dist(v)$. We note that such a vertex cannot have any tense incoming edges, since this would imply that there is a walk to v shorter than $dist(v)$ (which would correspond to a path of length less than $dist(v)$ since G has no negative cycles). Thus, all these vertices will still have $D[v] = dist(v)$ after this loop iteration.

  It remains to consider the vertices v for which the shortest $s \to v$ path with a minimum number of edges has exactly i edges. Let v be such a vertex and let $v_1, v_2, \ldots, v_{i+1}$ be a shortest $s \to v$ path (so $v_1 = s$ and $v_{i+1} = v$).

  We note that by the optimal substructure of shortest paths, $v_1, v_2, \ldots, v_i$ is a shortest $s \to v_i$ path, and uses only $i-1$ edges. Thus, by our IH, $D[v_i] = dist(v_i)$. Also, since $v_1, \ldots, v_i, v_{i+1}$ is a shortest $s \to v$ path, $dist(v) = dist(v_i) + w(v_i, v) = D[v_i] + w(v_i, v)$. Thus, if $D[v] > dist(v)$, this iteration of the loop will relax the edge (v_i, v) and $D[v]$ will become $dist(v)$.

Since every vertex reachable from s must be reachable on a path of at most $n-1$ edges, we have proven that after all $n-1$ iterations of the for loop, all of our distance upper bounds in D will be tight, and therefore our shortest path tree will be correct.

Finally, we analyze the runtime. The "repeat $n-1$ times" loop runs $\Theta(n)$ times and the "for each" loop runs $\Theta(m)$ times. Checking if an edge is tense and relaxing it takes constant time, so the total work of the "repeat" loop is $\Theta(mn)$. The work of the algorithm outside of the loop can also be done in $O(mn)$ time, so the overall runtime is $\Theta(mn)$. □

16.3 All Pairs Shortest Paths: Floyd-Warshall

In this section we consider a generalization of the shortest paths problem, all pairs shortest paths (APSP), where the goal is to compute the shortest paths between every pair of vertices in a graph. We can represent the solutions to all pairs shortest paths using n shortest path trees (one rooted at each vertex) represented as predecessor arrays as we've done for single source shortest paths. To simplify the algorithm and its analysis, however, in this section we will only focus on simply computing the *length* of all shortest paths (although the algorithm we give could be extended to create a shortest path tree for each vertex). That is, our goal is to compute a 2-D array D where $D[i, j]$ is the length of the shortest path between vertex i and vertex j.

The obvious way to compute all pairs shortest paths is to simply run a single source shortest path algorithm n times, starting once from each vertex in G. If our graph is unweighted, we can use BFS and this takes $O(n(m+n)) = O(mn+n^2)$ time. If it has non-negative edge weights we can use Dijkstra's and this takes $O(n(m+n\log n)) = O(mn+n^2\log n)$ time. If our graph has arbitrary edge weights, then we can use Bellman-Ford and this takes $O(n(mn)) = O(mn^2) \leq O(n^4)$ time. However, for graphs with arbitrary edge weights, there is a faster approach.

We can solve APSP faster using dynamic programming. As usual, the key is coming up with the right recurrence. While there are several ways that define a recurrence that we can memoize into a polynomial time algorithm, the clever choice that allows us to obtain a DP algorithm faster than repeatedly running Bellman-Ford is probably not the first one that you would think of. Assume again that our vertex set is $\{1, \ldots, n\}$. We define $APSP(i, j, k)$ as the length of the shortest $i \to j$ path which uses only vertices from $\{1, \ldots k\} \cup \{i, j\}$.

We give the following recurrence

$$APSP(i,j,k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } i = j \\ w(i,j) & \text{if } k = 0 \text{ and } (i,j) \in E \\ \infty & \text{if } k = 0,\ i \neq j, \text{ and } (i,j) \notin E \\ \min \begin{cases} APSP(i,j,k-1) \\ APSP(i,k,k-1) + APSP(k,j,k-1) \end{cases} & \text{otherwise} \end{cases}$$

Here the recursive case considers the choice of whether or not to use vertex k in our $i \to j$ path.

We can memoize the function $APSP$ into a 3-D array $D[1, \ldots, n][1, \ldots, n][0, \ldots, n]$. Each entry $D[i, j]$ depends only on entries with a smaller value of k $D[\cdot, \cdot, k-1]$, so we fill the array from smallest k to largest (we arbitrarily choose to, for a given k, fill in order of increasing values of i and j).

```
FLOYD-WARSHALL(G = (V, E))
    D[1,...,n][1,...,n][0,...,n] ← an array of integers
    For i from 1 to n:
        For j from 1 to n:
            D[i, j, 0] ← ∞
            If i = j:
                D[i, j, 0] ← 0
            If (i, j) ∈ E :
                D[i, j, 0] ← w(i, j)
    For k from 1 to n:
        For i from 1 to n:
            For j from 1 to n:
                D[i, j, k] ← min{D[i, j, k − 1], D[i, k, k − 1] + D[k, j, k − 1]}
    Return D[·, ·, n]
```

This algorithm is known as the Floyd-Warshall algorithm, after Robert Floyd and Stephen Warshall who described it in 1962.

Theorem 75. *For every weighted directed graph $G = (V, E)$ which does not contain a negative-weight cycle,* FLOYD-WARSHALL(G) *returns an array D such that for every pair of vertices $s, t \in V$, $D[s, t] = dist(s, t)$. Furthermore,* FLOYD-WARSHALL *runs in $\Theta(n^3)$ time.*

Proof. The correctness of this algorithm follows from the correctness of the recurrence: As we noted above, the recursive case of our recurrence considers the choice of whether or not to use vertex k in our $i \to j$ path. If the shortest $i \to j$ path which uses only vertices from $\{1, \dots k\} \cup \{i, j\}$ does not use vertex k, then it's the same as the shortest $i \to j$ path which uses only vertices from $\{1, \dots k-1\} \cup \{i, j\}$, and its length is $APSP(i, j, k-1)$. Otherwise, the shortest $i \to j$ path which uses only vertices from $\{1, \dots k\} \cup \{i, j\}$ uses vertex k, so it consists of taking a shortest $i \to k$ path followed by a shortest $k \to j$ path, and its length is $APSP(i, k, k-1) + APSP(k, j, k-1)$. A standard induction argument shows that $D[i][j][k]$ is the length of the shortest $i \to j$ path using only vertices from $\{1, \dots, k\} \cup \{i, j\}$ for all valid i, j, k.

We note that if $k = n$, $\{1, \dots, k\} \cup \{i, j\} = \{1, \dots, n\} = V$ for any i, j, so $D[i, j, n]$ is just the length of the shortest $i \to j$ path in G. Thus, the final 2-D array $D[\cdot, \cdot, n]$ that we return contains the length of the shortest $i \to j$ path for every pair of vertices i and j.

Now we analyze the runtime. Our array has $\Theta(n^3)$ entries, and we can compute each one in $\Theta(1)$ time (each entry only requires look at/summing a constant number of other entries). Therefore the total runtime is $\Theta(n^3) \cdot \Theta(1) = \Theta(n^3)$.

Finally, we note that since our DP table has $\Theta(n^3)$ entries, the space complexity is $\Theta(n^3)$. □

We note that although Theorem 75 assumed that the input graph has no negative cycles, it's not a big deal to handle negative cycles when computing APSP. Before running Floyd-Warshall we can check for negative cycles using Bellman-Ford in $O(mn) = O(n^2 n) \leq O(n^3)$ time, which will not increase the overall runtime of the algorithm (Bellman-Ford only finds negative-weight cycles reachable from the starting vertex, but we can guarantee that all negative cycles will be reachable from our starting point by creating a new vertex s which has an out-edge to every vertex of G and running Bellman-Ford starting from this vertex. The runtime remains $O(n^3)$).

Until recently it was commonly believed that APSP was unlikely to admit an $o(n^3)$ algorithm (i.e. that Floyd-Warshall was asymptotically the best algorithm possible). However, in October 2026, Alman and Williams published an $O(n^{2.9995})$ algorithm for APSP (based on an algorithm initially developed by an internal model at Anthropic), narrowly improving over Floyd-Warshall. Determining the actual best possible runtime for APSP remains an open area of research.

16.4 Comprehension Questions

1. True or false? If $G = (V, E)$ is a graph with negative edge weights, but no negative-weight cycle, and $s \in V$ then Dijkstra's(G, s) will correctly compute $dist(s, v)$ for each vertex $v \in V$.

2. True or false? If $G = (V, E)$ is a directed graph where every cycle has positive-weight, and $s, t \in V$, then every shortest $s \to t$ walk is a path.

3. Let $G = (V, E)$ be a graph, $s \in V$, and $e \in E$. When running Bellman-Ford(G, s), what is the maximum number of times that e could be relaxed?

4. In our correctness proof for Bellman-Ford we showed that, assuming G has no negative cycles, after i iterations of the "repeat $n-1$ times" loop, Bellman-Ford will have $D[v] = dist(v)$ for which vertices v?

5. Asymptotically, how many distinct subproblems does the Floyd-Warshall algorithm use?

6. In English, what does $APSP(i, j, k)$ represent (where $APSP$ is the recurrence used by Floyd-Warshall)?

16.5 Exercises

1. Suppose that you are given a directed graph $G = (V, E)$ with n vertices and m edges where each edge e has an arbitrary integer edge weight $w(e) \in \mathbb{Z}$, as well as two vertices $s, t \in V$. Your goal is to find the shortest $s \to t$ path which uses an even number of edges (or correctly report that there is no $s \to t$ path with an even number of edges). You may assume that G

does not contain a negative-weight cycle. Give an efficient algorithm for this problem and analyze its runtime in terms of m and/or n.

2. Give an $O(m+n)$ time algorithm which takes as input a weighted directed acyclic graph $G = (V, E)$ and a vertex $s \in V$ and outputs an array giving the distance from s to t for each $t \in V$. (Note: the key property that will allow you to design a faster algorithm than Bellman-Ford is that the graph is a DAG).

3. Barterville is a town which has not invented money, so all purchases are made by exchanging one kind of fruit for another. Each merchant in Barterville sells exactly one kind of fruit, and has a list of other fruit they accept in payment, as well as how much of the other fruit they consider equivalent to one of their fruits. For example, Alvin the apple merchant will exchange 1 apple for either 3 strawberries, 20 blueberries, or 0.1 watermelons.

   You are an enterprising but lazy young merchant in Barterville. You have done your market research by collecting a list of the n fruits currently being sold in the Barterville market. You've also collected a list of the m fruit pairs that can be exchanged, and for each pair (a, b) you've recorded a value $q(a, b)$, the quantity of fruit b which can be traded for 1 of fruit a.

   Your goal is to find a way to make a profit without producing any fruit of your own. In particular, you want to find a way to start with some fruit and perform a sequence of trades using the prices set by the Barterville merchants which will leave you with more of your original fruit than you started with. For example, if you can trade 0.1 watermelons for 1 apple, 0.05 apples for 1 blueberry, and 100 blueberries for 1 watermelon, then you can make a profit by starting with a watermelon, trading it for 10 apples, trading those for 200 blueberries, and trading the blueberries for 2 watermelons.

   Give an algorithm to determine whether or not there is a sequence of trades that will allow you to make a profit. Your algorithm should run in $O(mn)$ time.

4. Suppose that you are given a graph G with integer edge weights that can be positive, negative, or zero, but with no negative length cycle. Give an $O(mn)$ time algorithm to determine whether or not G contains a zero-length cycle (Hint: perturb the edge weights slightly then use an algorithm from this chapter). Make sure you justify why your algorithm always finds a zero-length cycle when one exists, but never reports a zero-length cycle when none exists.

5. Suppose that you are given a strongly connected directed weighted graph $G = (V, E)$, as well as a subset $A \subseteq E$ of the edges which are "awesome". You may assume that G has no negative-weight cycle. You want to find, for every pair of vertices $u, v \in V$, the shortest $u \to v$ walk which uses at least one awesome edge. Give an $O(n^3)$ time algorithm for this problem.

17 Minimum Spanning Trees

In this chapter we will define spanning trees and look at the two most well known algorithms for computing the minimum-weight spanning tree in a weighted graph.

17.1 Spanning Trees

Recall that we define a tree as a graph which is connected and acyclic with $m = n - 1$. We proved in Theorem 65 that having any two of these three properties implies the third, so to show that a graph is a tree we can show that it has whichever two of these properties are most convenient to prove.

Recall also that a graph G' is a subgraph of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$, i.e. G' can be constructed from G by selectively removing (possibly 0) vertices and/or edges. We say that G' is a **spanning** subgraph of G if G' is a subgraph with $V(G') = V(G)$, i.e. G' is created with only edge deletions. A spanning tree is a spanning subgraph which is a tree.

Theorem 76. *A graph G has a spanning tree if and only if G is connected.*

Proof. First we prove that if G is connected then it has a spanning tree. We can find a spanning tree of a connected graph in $O(m + n)$ time by running BFS (or DFS) from any arbitrary vertex and adding to our tree the edges that we use to visit new vertices. Since G is connected, BFS will visit all vertices, and thus the shortest paths tree will be a spanning tree.

Now let G be a graph with a spanning tree. We need to show that G is connected. Let T be a spanning tree for G. To show that G is connected we need to show that for every pair of vertices in G there is a path in G between those vertices. Let $u, v \in V(G)$ be arbitrary. Because T is a spanning subgraph of G, $V(T) = V(G)$, so u and v are also vertices of T. Because T is a tree, T is connected, so there exists a $u \to v$ path P in T. Because T is a subgraph of G, all of the vertices and edges of P are also present in G, so P is a $u \to v$ path in G, and thus u and v are connected. Since u and v were arbitrary vertices, this means that G is connected. □

Note that unlike the shortest path trees we've discussed over the previous two chapters, a spanning tree is not a rooted tree (we could arbitrarily pick a vertex to treat as the root, but there is no distinguished choice for this vertex). Unlike our SSSP algorithms, an MST algorithm does not take in a source vertex s. Also, while shortest path algorithms are most commonly defined for directed graphs (since shortest paths in undirected graphs can be seen as a special case of shortest paths in directed graphs), spanning trees are defined only for undirected graphs.

A graph can have many spanning trees, exponentially many even if it is very dense. For example, a complete graph (one with an edge between every pair of vertices) on n vertices has n^{n-2} distinct spanning trees (this is known as Cayley's formula, and was first proven in 1860).

Out of the many possible spanning trees, we would often like to find the best one. To compare different spanning trees, we define the weight of a tree as the sum of the weights of its edges:

$$w(T) := \sum_{e \in E(T)} w(e).$$

We define a **minimum spanning tree** for G as a spanning tree with minimum weight, among all spanning trees for G.

Throughout this chapter we will focus on weighted graphs with *distinct* edge weights. The assumption that all edge weights are distinct makes several of our proofs and algorithms shorter and simpler. While this assumption may seem unrealistic, in actuality it is generally easy to convert an MST algorithm that works only for graphs with distinct edge weights into one that works with arbitrary integer weights. We can simply choose tiny perturbation values $0 < \epsilon_1 < \epsilon_2 < \ldots, < \epsilon_{m-1} < \epsilon_m$ such that $\sum_{i=1}^{m} \epsilon_i < 1$ and then add ϵ_i to the weight of the i'th edge. For example, if our graph had edge weights of $5, 3, 5, 7, 7, 7$ and 12, we might change these to $5.01, 3.02, 5.03, 7.04, 7.05, 7.06$, and 12.07.

These ϵ_i values give us a "tiebreaker" between identical edges, ensuring that all edge-weights in the graph are distinct, but since the total weight of all the ϵ_i's is less than 1 the weight of the MST increases by less than 1, and thus the extra weight cannot cause any tree that wasn't tied for the lightest in the original graph to become an MST in the new graph.

17.2 Properties of MSTs

We begin our study of minimum spanning trees by proving some properties that will be helpful in designing and analyzing MST algorithms. The proofs in this section also illustrate some general techniques that are useful for proofs involving spanning trees.

Often in these proofs we want to discuss a graph created by adding or removing a single edge from another graph. To do this more easily we define the following notation:

$$G + e := (V(G), E(G) \cup \{e\}), \qquad G - e := (V(G), E(G) \setminus \{e\}).$$

That is, $G+e$ is the graph created by adding edge e to G and $G-e$ is the graph created by removing edge e from G.

We begin by proving that our assumption of distinct edge weights in the graph guarantees that we can only have one minimum spanning tree. This means that for the rest of the chapter we can talk about "the" minimum spanning tree in a graph, rather than needing to consider the possibility that multiple trees tie for the minimum weight.

Theorem 77. *If G is a connected weighted graph with distinct edge weights, then G has a unique minimum spanning tree.*

Proof. Assume for sake of contradiction that there exist two distinct minimum spanning trees T_1 and T_2 for G. Since T_1 and T_2 are both spanning trees, $V(T_1) = V(T_2) = V(G)$ and $|E(T_1)| = |E(T_2)| = n-1$. Thus, each tree must have at least one edge the other does not. Let e be the lightest (minimum weight) edge which is in one tree but not the other. Without loss of generality, assume $e \in E(T_1)$.

Consider the graph G' obtained by adding e to T_2. Since T_2 is connected, T_2 already contains a path between e's endpoints, so adding e to T_2 creates a cycle. Call this cycle C. Since T_1 is acyclic, but contains e, there must be some other edge of C which is in T_2 but not T_1. Let f be such an edge. Since all edge weights in G are distinct and we chose e as the lightest edge which is in one tree but not the other, $w(e) < w(f)$.

Now consider the graph $T' := T_2 - f + e$. We note that T' has $n-1$ edges (since we added and removed one edge from a tree). Let $v_1, v_2, \ldots, v_k$ be the vertices of cycle C in order with $e = (v_1, v_k)$ and $f = (v_i, v_{i+1})$ (for some $1 < i < k$). We note that removing f from T_2 splits T_2 into two connected components, one containing $v_1, \ldots, v_i$ (among other vertices) and the other containing $v_{i+1}, \ldots, v_k$ (among other vertices). Adding e to $T_2 - f$ reconnects these two components, so T' is a connected graph with $n-1$ edges on vertex set V and thus is a spanning tree for G. Furthermore $w(T') = w(T_2) - w(f) + w(e) < w(T_2)$ (since $w(e) < w(f)$). Thus, there is a spanning tree with smaller weight than T_2, contradicting our assumption that T_2 was a minimum spanning tree. Therefore, G has only one MST. □

A **cut** in a graph $G = (V, E)$ is a partition of the vertex set V into two non-empty sets $S, V \setminus S$. We say that an edge e **crosses** the cut, if e has one endpoint in S and one endpoint in $V \setminus S$.

The following theorem is known as the "light edge property", and is the core result behind the MST algorithm we will see in the next section. The proof uses the same "swapping edges in a cycle" technique that we introduced in the proof of Theorem 77.

Theorem 78. *Let G be a connected weighted graph with distinct edge weights and $S, V \setminus S$ be a cut in G. Let e be the minimum weight edge crossing this cut. Then the minimum spanning tree for G contains e.*

Proof. As in the previous proof, we will use contradiction. Let T be a minimum spanning tree for G and suppose for contradiction that $e \notin T$. Then adding e to T creates a cycle C. Call the vertices of C $v_1, v_2, \ldots, v_k$, where v_1 is the endpoint of e in S and v_k is the endpoint of e in $V \setminus S$. Then, there must be at least one place where the path $v_1, v_2, \ldots, v_k$ transitions from S to $V \setminus S$ (since it starts in S and ends in $V \setminus S$). Let $1 \leq i < k$ be an integer such that $v_i \in S$ and $v_{i+1} \notin S$. Call the edge $\{v_i, v_{i+1}\}$ f. Then f crosses the cut $S, V \setminus S$. Since e is the lightest edge crossing this cut, $w(f) > w(e)$.

Consider the graph $T' := T - f + e$. Note that T' still has $n - 1$ edges. Furthermore, T' is still connected, by the same argument we used in the proof of Theorem 77 (removing f splits T into two components, one of which contains $v_1, \ldots, v_i$ and the other of which contains $v_{i+1}, \ldots, v_k$. Adding e reconnects these components). Thus, T' is a spanning tree for G with $w(T') = w(T) - w(f) + w(e) < w(T)$, contradicting our assumption that T was a minimum spanning tree. □

17.3 The Prim-Jarník Algorithm

The light edge property implies that we can construct an MST by repeatedly taking cuts in G and choosing the lightest edge crossing them to be part of our tree. This is what the first algorithm we will consider does.

The algorithm was first described by Vojtěch Jarník in 1929, although it is most commonly known today as "Prim's algorithm" after Robert Prim, who independently rediscovered it in 1957. As a compromise, we will refer to it as the Prim-Jarník algorithm.

```
PRIM-JARNÍK(G)
  v ← an arbitrary vertex in V
  T ← ({v}, ∅)
  While V(T) ≠ V(G):
      e ← the lightest edge crossing the cut V(T), V(G) \ V(T)
      Add e to E(T)
      Add e's endpoint from V(G) \ V(T) to V(T)
  Return T
```

Theorem 79. *Let G be a connected weighted graph with distinct rational edge weights. Then* PRIM-JARNÍK(G) *computes the minimum spanning tree of G.*

Proof. We note that every edge we add to T is the lightest edge crossing some cut in G. Thus, by Theorem 78, every edge we add to T is part of the MST. Since every edge we add includes exactly one vertex which isn't part of our current tree, we know that each iteration of the while loop finds a new edge from the MST to add. Furthermore since we start with one vertex in T and each iteration adds only one vertex, the loop runs for $n - 1$ iterations, so by the time we return T it contains all $n - 1$ of the edges from the MST and thus T is the MST for G. □

Theorem 80. PRIM-JARNÍK *can be implemented to run in time $O(m + n \log n)$.*

Proof. As we showed in the proof of Theorem 79, the while loop in PRIM-JARNÍK runs $n - 1$ times. The work outside of the loop can be done in constant time. The tricky part is analyzing the work inside of the loop, in particular, how long does it take to find the lightest edge crossing $V(T), V(G) \setminus V(T)$? This depends on how we represent T.

One simple approach would be to use a length n boolean array to keep track of whether or not each vertex is in T, and a list of the edges currently in T. We can find the lightest edge across $V(T), V(G) \setminus V(T)$ by iterating through all the edges of G, and keep track of the lightest one among those which have one endpoint true in our boolean array and the other false. This takes $O(m)$ time. Thus, each iteration of the loop takes $O(m)$ time and so the total runtime of the algorithm with this approach is $O(mn)$.

We can do better, however. Instead of iterating through all of the edges every iteration of the while loop, we can use a priority queue to store the edge between $V(T)$ and $V(G) \setminus V(T)$. To get the lightest edge from $V(T)$ to $V(G) \setminus V(T)$, we call EXTRACTMIN on our priority queue until we get an edge crossing $V(T), V(G) \setminus V(T)$. When we add a new vertex to T, we add its edges to vertices in $V(G) \setminus V(T)$ to the priority queue. Each edge is inserted into the priority queue at most once and extracted at most once, so we make $O(m)$ calls to EXTRACTMIN, each of which takes $O(\log n)$ time (whether we use a binary or Fibonacci heap). Thus, the total runtime of our algorithm becomes $O(m \log n)$.

We can do even better with a slightly different perspective. To avoid needing $O(m)$ extractions from our priority queue, instead of storing edges in our queue we can store *vertices*. We let the

priority queue store each vertex in $V(G) \setminus V(T)$ and the key for a vertex v is the weight of the lightest edge from $V(T)$ to v. We can use ExtractMin on this priority queue to get the next vertex v we should add to T. Then we can find the lightest edge e from v to the current vertices of T, add v and e to T, and then we call DecreaseKey to update the keys of v's neighbors based on their edges from v. With this approach we make only $O(n)$ total calls to ExtractMin along with $O(m)$ calls to DecreaseKey. Thus, with a Fibonacci heap, the total runtime of the algorithm becomes $O(m + n \log n)$.

□

17.4 Kruskal's Algorithm

The other algorithm that we will present for computing an MST is also a very natural greedy algorithm. We start with a subgraph consisting of the empty graph on $V(G)$ and then repeatedly add to our subgraph the cheapest edge that connects two disconnected components of our subgraph until the whole subgraph is connected. This algorithm was developed by Joseph Kruskal in 1956.

```
Kruskal's(G)
  A ← an array of G's edges, sorted in order of increasing weight
  T ← (V(G), ∅)
  For i from 1 to m:
      If adding A[i] to T would not create a cycle:
          Add A[i] to T
  Return T
```

Theorem 81. *Let G be a connected weighted graph with distinct rational edge weights. Then* Kruskal's(G) *computes the minimum spanning tree of G.*

Proof. Let $e = (u, v)$ be an edge that Kruskal's adds to T. Let U be the component of T containing u at the start of the iteration of the for loop where e is added to T. We note adding any edge which crosses the cut $U, V \setminus U$ to T would connect two different connected components and therefore would not create a cycle. But since U is a connected component (meaning that it is a maximal set of connected vertices), none of these edges have been added to T yet. Since we are considering the edges in increasing order of weight (because A is sorted) and e is the first edge crossing this cut that we've considered, e must be the lightest edge crossing $U, V \setminus U$. Therefore by Theorem 78, every edge that Kruskal's adds to T is part of the MST.

Furthermore, we note that Kruskal's will add all edges of the MST to T, because an MST is acyclic, so each of these edges can be added without creating a cycle. Therefore, when the algorithm returns T, T is the MST for G. □

Theorem 82. Kruskal's *can be implemented to run in time $O(m \log n)$.*

Proof. Sorting the edges by weight takes $O(m \log m)$ time (note that since $m \leq n^2$ this is $O(m \log n)$). The for loop runs m times. This time it's not as clear how to track whether adding an edge to T would create a cycle. T can have many components and we need some way to keep track of which is which.

We can keep an array labeling each vertex with a component number (initially each vertex has its own component). This allows us to check whether an edge would create a cycle in constant time (we just check whether its two endpoints have different component labels). When we add an edge, connecting two components, we relabel all of the vertices in one of the two merged components with the label of the other component. Unfortunately, since a component can have $\Omega(n)$ vertices, this can take $\Theta(n)$ time. Thus, with this approach our runtime is $\Theta(m \log n + n^2)$.

A cleverer approach is to have a "component" object consisting of a pair of integers where the first one is an identifier for the component and the second one is the size of the component. Each vertex maintains a pointer to its current component (initially we have n components, with identifiers ranging from 1 to n, each with a size of 1).

To check if two vertices are in the same component, we simply check if the identifiers for their components are the same. Since our components now keep track of their size, we can make merging more efficient. In particular, when we merge two components, we can change the pointers for all of the vertices from the *smaller* of the two merged components to point to the component object for the *larger* of the two components, and update this component's size to include the added vertices.

With this approach, whenever we need to update the component for a given vertex v, v's new component must be at least twice as big as its old component (since updating v means that v's component was the smaller one in the merge). We can double the size of v's component only $O(\log n)$ times before it includes all n vertices. Thus, with this approach, the total runtime for the for loop becomes $O(m + n \log n)$.

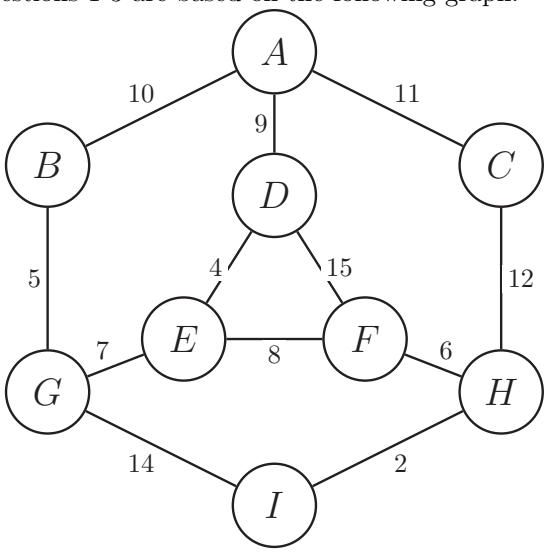
Since $m \in \Omega(n)$ (because G is connected), $m \log n \in \Omega(m + n \log n)$, so our runtime is dominated by the time to sort the edges and our overall runtime is $O(m \log n)$.

□

If our edge list is already sorted or if it can be sorted in linear time (e.g. because the edge weights are all relatively small integers and can be sorted with radix sort) then Kruskal's with the implementation we just discussed runs in $O(m + n \log n)$ time the same as Prim-Jarník. Kruskal's algorithm can actually be made to run even faster, by using a specialized "union-find" data structure, but the details are beyond the scope of this course.

17.5 Comprehension Questions

Questions 1-3 are based on the following graph:



1. Suppose that we are using Prim-Jarník to find an MST in the graph and so far the only edge we have added to our subgraph is H, I. Which edge will we add next?

2. Suppose that we are using Kruskal's to find an MST in the graph and so far the only edge we have added to our subgraph is H, I. Which edge will we add next?

3. Highlight all the edges in the MST for this graph.

4. True or false? Every connected graph with integer edge weights has a unique minimum spanning tree.

5. What will the runtime of Kruskal's be if instead of using an array of component numbers/pointers, we check whether adding an edge (u, v) would create a cycle by running DepthFirstSearch(T, u) and seeing if we visit v?

6. What would the runtime of Prim-Jarník be if instead of using a Fibonacci heap, we used a binary heap (where ExtractMin and DecreaseKey both take $\Theta(\log n)$ time)?

17.6 Exercises

1. In this chapter we proved that if all of the edges in a graph have distinct rational weights then Prim-Jarník(G) and Kruskal's(G) find minimum spanning trees. In this problem we show that this assumption was not necessary.

   - Prove that if G is a graph with rational edge weights then Prim-Jarník(G) finds one of the minimum spanning trees of G.
   - Prove that if G is a graph with rational edge weights then Kruskal's(G) finds one of the minimum spanning trees of G.

2. Suppose that you are given a weighted graph (G, w) and a minimum spanning tree T for G, along with an edge $e \in E(G)$ and a new weight x for e. Give an $O(m)$ algorithm for finding a new MST for G after $w(e)$ is updated to x. [Hint: You may want to case on whether or not $e \in T$ and/or whether $w(e)$ was increased or decreased]

3. Let (G, w) be a (undirected) connected graph where every edge has a distinct edge weight.

   (a) Prove that for every cycle in G, the *maximum* weight edge in the cycle is *not* part of the minimum spanning tree.

   (b) Show that the following algorithm correctly finds an MST in G. Analyze the algorithm's runtime.

```
CycleRemovalMST(G, w)
  C ← a cycle in G (or Null if G is acyclic)
  While C ≠ Null:
      e ← the maximum weight edge in C
      G ← G \ e
      C ← a cycle in G (or Null if G is acyclic)
  Return G
```

4. You are visiting a magnificent garden, consisting of a collection of flower beds connected by winding trails. You have acquired a map of the garden, listing all of the flower beds and all of the trails between them (each trail connects exactly two distinct flower beds). Your map also lists, for each trail, the number of minutes it takes to walk along it while fully appreciating the natural surroundings (you may assume that this time does not depend on which direction you are walking on the trail, nor on whether or not you have already walked this same trail).

   In order to maximize your enjoyment, you want to visit all of the flower beds in the garden, while fully appreciating the natural surroundings on every trail you take (you may start and end at whichever flower beds you wish). We define a *stroll* as a walk through the garden which does this. Since there may be many different strolls, you're interested in finding the fastest one. This turns out to be a difficult problem, however (we'll see why later in the course). You aren't in too much of a hurry, so you're willing to settle for a stroll whose length is at most twice that of the best possible stroll. Give a polynomial-time algorithm which finds such a stroll. [Hint: Use a minimum spanning tree]

5. Suppose that you are given a weighted graph (G, w) where all of the edges have integer weights between 1 and 411 (inclusive). Give an algorithm to find a minimum spanning tree for G in $O(m + n)$ time.

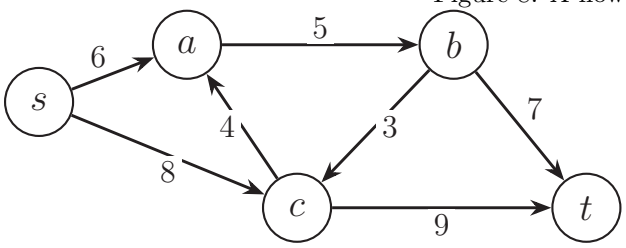
18 Max-Flow

In this chapter, we introduce a new problem to solve in directed graphs: Maximum flow. Like the shortest $s-t$ path problem, maximum flow asks about getting between two vertices s and t. In this problem, however, our goal is not to get between s and t as fast as possible, but to get as much "flow" as possible between s and t, across edges of limited capacity. This problem was originally studied to model the flow of goods from factories to consumers across rail networks, and remains relevant for many problems in logistics today.

18.1 Flows and Cuts

A **flow network** is a directed graph G with two special vertices s and t called the source and sink, respectively (s and t must be different vertices), where each edge e has an associated non-negative value $c(e)$ called the **capacity**.

Figure 8: A flow network



An **s-t-flow** (or simply a **flow**) is a function $f : E(G) \to \mathbb{R}_{\geq 0}$ which satisfies the following two constraints:

- $\forall\, e \in E(G), 0 \leq f(e) \leq c(e)$ (we call this the capacity constraint: the flow through an edge is at most its capacity)
- $\forall\, v \in V(G) \setminus \{s,t\}, \sum_{u \in N_{in}(v)} f(u,v) = \sum_{w \in N_{out}(v)} f(v,w)$ (we call this the flow conservation constraint: for every vertex besides the source and sink, the total flow into the vertex must equal the total flow out of the vertex).

For convenience we will often interpret f as a function from $V \times V$ to $\mathbb{R}_{\geq 0}$ with $f(u,v) = 0$ for every $(u,v) \notin E$.

The **value** of a flow f, denoted $|f|$, is the net flow out of s:

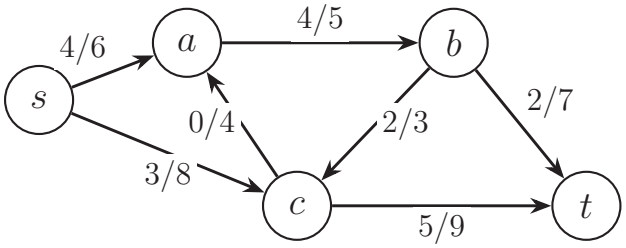
$$|f| := \left(\sum_{w \in N_{out}(s)} f(s,w) \right) - \left(\sum_{u \in N_{in}(s)} f(u,s) \right).$$

Note that by the conservation constraint, this is also equal to the net flow into t. The figure below illustrates a flow of value 7 in the network from Figure 8. Note that in this figure X/Y is not a fraction, it denotes sending X units of flow through an edge of capacity Y.

A **maximum flow** is a flow with maximum value across all flows in a network. The flow in Figure 9 is not a maximum flow (can you find a flow which achieves the actual maximum flow value of 13 for the network from Figure 8?). The central problem we will consider in this chapter is how to compute a max-flow. Before we can introduce an algorithm for this problem, however, it will be helpful to discuss cuts, which turn out to be closely related to flows.

Recall from Section 17.2 that a **cut** in an undirected graph is a partition of the vertices into two non-empty sets $S, V \setminus S$. Previously we discussed cuts for undirected graphs, where we said that an edge $\{u,v\}$ crosses the cut if one of u, v is in S and the other is in $V \setminus S$. We can define a related notion of cuts for flow networks.

Figure 9: A flow of value 7

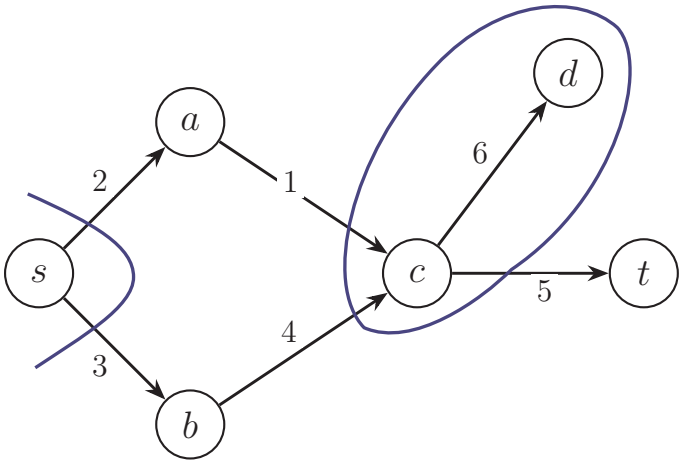


An s-t-**cut** (or simply a **cut**) in a flow network G is a partition of $V(G)$ into two sets S and T such that $s \in S$ and $t \in T$ (note that S, T being a partition means that we must have $S \cup T = V(G)$ and $S \cap T = \emptyset$). We say that an edge (u, v) **crosses** the cut if $u \in S$ and $v \in T$. Notably, we do *not* say that (u, v) crosses the cut if $u \in T$ and $v \in S$. We denote the set of edges crossing the cut S with $\delta_{out}(S)$.

The **value** (or **capacity**) of a cut S, T, denoted $c(S, T)$, is the total capacity of the edges crossing the cut. A cut with minimum value among all s-t-cuts in G is called a minimum s-t-cut (or simply min-cut).

The figure below depicts an s-t-cut with $S = \{s, c, d\}$ and $T = \{a, b, t\}$. Note that the definition of a cut does not require S or T to be connected. Since $s \in S, t \in T, S \cap T = \emptyset$, and $S \cup T = V$, this is a valid cut. Here $\delta_{out}(S) = \{(s, a), (s, b), (c, t)\}$. Note that (a, c) and (b, c) do not cross the cut. The value of this cut is $2 + 3 + 5 = 10$. This is not a min-cut in the network. The actual min-cut with a value of 4, is $S = \{s, a\}, T = \{b, c, d, t\}$.

Figure 10: An s-t-cut



18.2 Max-Flow Min-Cut

At first glance max-flow and min-cut may not seem related. They actually have a very close connection, however. For a flow f and a cut S, T we define the flow across the cut by

$$f(S, T) := \left(\sum_{u \in S, v \in T} f(u, v) \right) - \left(\sum_{u \in T, v \in S} f(u, v) \right).$$

That is, $f(S, T)$ is the total flow from S to T minus the total flow from T back to S. The following theorem says that the flow through any s-t-cut is always equal to the value of the flow (recall that this is defined as the flow out of s).

Theorem 83. *Let G be a flow network, f be an s-t-flow in G, and S, T be an s-t-cut in G. Then $|f| = f(S,T)$.*

Proof.

$$\begin{aligned}
f(S,T) &= \sum_{u\in S, v\in T} f(u,v) - \sum_{u\in T, v\in S} f(u,v) \\
&= \sum_{u\in S, v\in T} f(u,v) - \sum_{u\in S, v\in T} f(v,u) \\
&= \sum_{u\in S, v\in T} f(u,v) + \sum_{u\in S, v\in S} f(u,v) - \sum_{u\in S, v\in T} f(v,u) - \sum_{u\in S, v\in S} f(v,u) \\
&= \sum_{u\in S}\left(\sum_{v\in T} f(u,v) + \sum_{v\in S} f(u,v) - \sum_{v\in T} f(v,u) - \sum_{v\in S} f(v,u)\right) \\
&= \sum_{u\in S}\left(\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u)\right)
\end{aligned}$$

Note that $\sum_{v\in V} f(u,v)$ is the total flow out of vertex u and $\sum_{v\in V} f(v,u)$ is the total flow into vertex u. Thus, by our flow conservation constraint, for all $u \in V \setminus \{s,t\}$, we have $\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u) = 0$. We know that $s \in S$ and $t \notin S$ by the definition of an s-t-cut, so therefore

$$\sum_{u\in S}\left(\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u)\right) = \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) = |f|.$$

□

Theorem 83 implies the following corollary:

Theorem 84. *In any flow network G, the value of a maximum flow is less than or equal to the value of a minimum cut.*

Proof. Let f be a flow in G. By Theorem 83 for any cut $c(S,T)$,

$$|f| = f(S,T) = \sum_{u\in S, v\in T} f(u,v) - \sum_{u\in T, v\in S} f(u,v) \leq \sum_{u\in S, v\in T} f(u,v) \leq \sum_{u\in S, v\in T} c(u,v) = c(S,T).$$

In particular if f is a max-flow and S, T is a min-cut then $|f| \leq c(S,T)$. □

Theorem 84 tells us that we can upper bound the max-flow value in a network with the min-cut value. It turns out that these two values are actually *equal* as the following theorem states. This theorem is known as the max-flow min-cut theorem and was first proved by Lester Ford (same Ford who developed the Bellman-Ford algorithm) and Delbert Fulkerson in 1954.

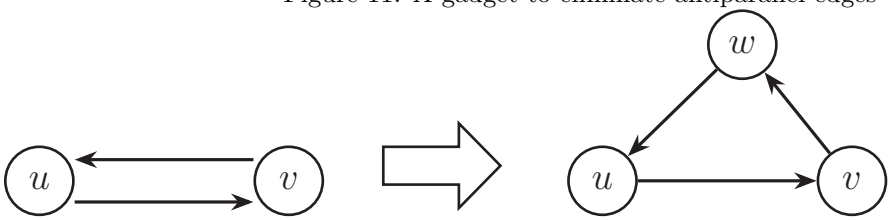
Theorem 85. *In every flow network with integer capacities, the value of a maximum s-t-flow is equal to the value of a minimum s-t-cut.*

Theorem 84 gives us half of this theorem. To finish the proof, we also need to show that the max flow value is *at least* the min-cut value. Ford and Fulkerson's approach to this was to give an algorithm which computes a flow with a value equal to the minimum cut value. We will present and analyze this algorithm in the next section. Theorem 85 in fact holds for arbitrary real capacities, but the proof is more involved and not important for this course.

18.3 Finding a Max-Flow: Ford-Fulkerson

In order to state Ford and Fulkerson's algorithm, we need a simplifying assumption and a few more definitions. The simplifying assumption we will use is that the input flow network G has no antiparallel edges, meaning that for every pair of vertices $u, v \in V$, at most one of the edges (u, v) and (v, u) is present in G. This is a safe assumption to make, because we can convert a flow network with antiparallel edges into an equivalent network which doesn't have them in $O(m + n)$. In particular, if we have a pair of vertices u, v such that (u, v) and (v, u) are both edges in G, we can introduce a new vertex w and replace the (v, u) edge with the length 2 path v, w, u. We set the capacity of both edges on this path equal to the capacity of (v, u) in the original graph. This transformation does not affect the maximum flow value since any flow which used the (v, u) edge can now use the v, w, u path and vice-versa. The transformation is illustrated in Figure 11.

Figure 11: A gadget to eliminate antiparallel edges



Henceforth we will assume that our input flow network has no antiparallel edges.

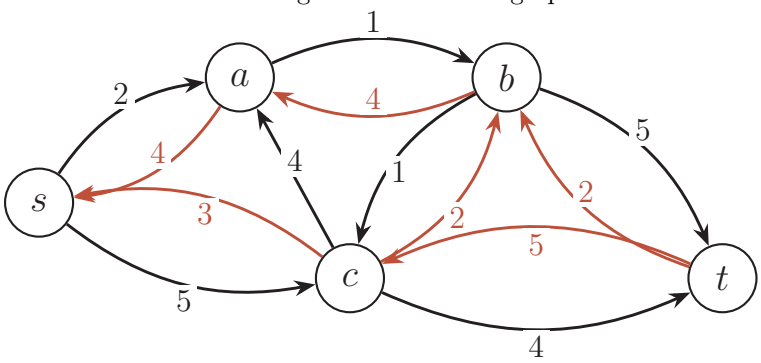
For a flow network G, and a flow f in G, we define the **residual capacity function** c_f by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E(G) \\ f(v, u) & \text{if } (v, u) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

Note that our assumption of no antiparallel edges guarantees that the first two cases are mutually exclusive (this is why we needed the assumption). We define the **residual graph** G_f for the flow f in the network G as the flow network on the same vertex set as G with the same source and sink, where the edge set is $E_f := \{(u, v) \mid u, v \in V(G) \wedge c_f(u, v) > 0\}$ and the capacity function is c_f.

Observe that the residual graph can have antiparallel edges even though the original network does not. Figure 12 illustrates the residual graph G_f for the flow from Figure 9. Edges that were part of the original graph G are depicted in black (note, however that their capacities have been reduced based on the amount of flow through them), while the new reverse edges are depicted in red.

Figure 12: Residual graph for the flow from Figure 9



We define an **augmenting path** for a flow f as an $s - t$ path in the residual graph G_f. The **residual capacity** of an augmenting path P is the amount of flow that we can push through that

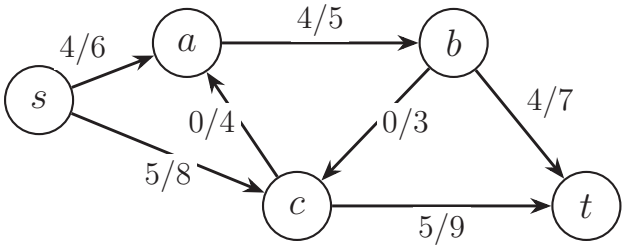
path in the residual graph: $c_f(P) := \min_{e \in P} c_f(e)$ (here we abuse notation slightly by viewing P as a set of edges rather than a sequence of vertices). For example, for the flow f from figure 9, s, c, b, t is an augmenting path. It has residual capacity 2, since that is the capacity in the residual graph of (c, b), the edge on this path with lowest residual capacity.

To **augment** f along a path P in the residual graph means doing the following for each edge (u, v) in P:

- If $(u, v) \in E(G)$, increase $f(u, v)$ by $c_f(P)$
- If $(v, u) \in E(G)$, decrease $f(v, u)$ by $c_f(P)$

For example, if we augment the flow from Figure 9 along the path s, c, b, t, we obtain the flow shown below:

Figure 13: A flow of value 9



It may not be clear that f remains a valid flow after augmenting. We will prove this now.

Theorem 86. *Let f be a flow and let f' be the result of augmenting f along an augmenting path P in the residual graph. Then f' is a valid flow of value $|f| + c_f(P)$.*

Proof. To show that f' is a valid flow we need to show that f' still satisfies the capacity and flow conservation constraints.

- First we show that f' satisfies the capacity constraints. Let $e = (u, v)$ be an arbitrary edge of G. We need to show that $0 \leq f'(e) \leq c(e)$. We consider three cases:
  - Suppose that $(u, v) \in P$. Then

    $$f'(u, v) = f(u, v) + c_f(P) > f(u, v) \geq 0$$

    and

    $$f'(u, v) = f(u, v) + c_f(P) \leq f(u, v) + c_f(u, v) = f(u, v) + c(u, v) - f(u, v) = c(u, v).$$

  - Suppose that $(v, u) \in P$. Then

    $$f'(u, v) = f(u, v) - c_f(P) \geq f(u, v) - c_f(v, u) = f(u, v) - f(u, v) = 0$$

    and

    $$f'(u, v) = f(u, v) - c_f(P) \leq f(u, v) \leq c(u, v).$$

  - Suppose that neither (u, v) nor (v, u) is in P. Then augmenting along P does not change the flow through (u, v), so $f'(u, v) = f(u, v)$ and so, since f is a valid flow, $0 \leq f'(e) \leq c(e)$.
- We need to show that for each vertex $v \in V(G) \backslash \{s, t\}$, $\sum_{u \in N_{in}(v)} f'(u, v) = \sum_{w \in N_{out}(v)} f'(v, w)$.If P does not visit v, then all of the edges incident to v have the same values under f' as under f so flow conservation is maintained. Otherwise, P must enter and exit v. We note that:

- If P enters v on an edge $(u, v) \in E(G)$ then f' increases the incoming flow to v along this edge by $c_f(P)$, while if P enters on an edge (w, v) with $(v, w) \in E(G)$, then f' decreases the outgoing flow along (v, w) by $c_f(P)$. Either way, the contribution from P's incoming edge causes v's in-flow to increase by $c_f(P)$ relative to its out-flow.
- If P exits v on an edge $(v, w) \in E(G)$, then f' increases the outgoing flow along (v, w) by $c_f(P)$, while if P exits on an edge (v, u) with $(u, v) \in E(G)$, then f' decreases the incoming flow along (u, v) by $c_f(P)$. Either way, the contribution from P's outgoing edge causes v's out-flow to increase by $c_f(P)$ relative to its in-flow.

These two effects cancel each other out, so the out-flow from v remains equal to the in-flow.

Finally, we note that f' increases the net flow out of s (and thus the value of the flow) by $c_f(P)$ by increasing the amount of flow through an edge (s, v) or by decreasing the amount of flow through an edge (v, s). □

Finally, we are ready to state the Ford-Fulkerson algorithm.

```
FORD-FULKERSON(G)
  f ← an empty flow (i.e. f(e) = 0   ∀ e ∈ E(G))
  While there exists an augmenting path P in the residual graph G_f:
      Augment f along P
  Return f
```

Theorem 87. *If G is a flow network with integer capacities,* FORD-FULKERSON(G) *returns a maximum flow in G.*

Proof. As shown in Theorem 86, every time we augment along an augmenting path, we increase the value of our flow by the residual capacity of the augmenting path. Since all of the capacities in G are integers, all of the capacities in G_f will be integers for any integer flow f. Since, by definition, G_f contains no 0 capacity edges, this means that if P has an augmenting path, that path has residual capacity at least 1. Thus, each iteration of the while loop, the value of our flow increase by at least 1. Since the maximum flow value in any network with integer capacities is some finite value, this means we must eventually reach a point where there are no augmenting paths and return f. We need to show that at this point f is a maximum flow. We will do this by showing that f has the same value as some cut. By Theorem 84 this implies that f must be a maximum flow (since no flow can have value greater than any cut).

Let f be the flow that FORD-FULKERSON returns. Since the residual graph contains no augmenting paths, we know that there is no s-t-path in G_f. Let $S := reach_{G_f}(s)$ and $T := V \setminus S$. None of the edges from S to T are present in G_f (otherwise we could use one of those edges to reach more vertices from s). By the definition of G_f, this means that for every edge $(u, v) \in E(G)$ with $u \in S, v \in T$ we have $f(u, v) = c(u, v)$. and for every edge $(u, v) \in E(G)$ with $u \in T, v \in S$, we have $f(u, v) = 0$. Therefore, $f(S, T) = c(S, T)$. By Theorem 83, $|f| = f(S, T)$, so by Theorem 84, f is a maximum flow. □

Note that Theorem 87 implies that every flow network with integer capacities has an integral maximum flow (i.e. we do not need to consider sending fractional amounts of flow through edges). The assumption that the edge capacities are integers is necessary in order to guarantee that the while loop terminates in finite time. With irrational capacities it is actually possible to create a graph where FORD-FULKERSON runs forever.

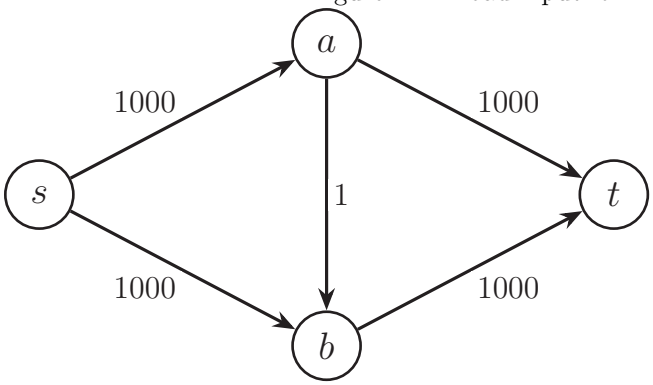
We note that Theorem 87 completes the proof of Theorem 85. The proof of Theorem 87 also gives us a way to efficiently find a min-cut, given a max flow. If f is a max-flow (computed by FORD-FULKERSON or any other algorithm), then for $S := reach_{G_f}(s)$, we have that $S, V \setminus S$ is a min s-t-cut.

18.4 Speeding Up Max-Flow: Edmonds-Karp

While Ford-Fulkerson will always find a maximum flow in a flow network with integer capacities, it may do so very slowly if we choose the wrong paths to augment along. For example, for the graph shown in Figure 14, if we alternate augmenting along s, a, b, t and augmenting along s, b, a, t, it will take 2000 rounds of augmentation to obtain the maximum flow in the graph, even though there are only 4 vertices in the network!

We can speed up the algorithm exponentially by being strategic about which paths we choose to augment along. This idea was developed by Jack Edmonds and Richard Karp in 1972.

Figure 14: A bad input for Ford-Fulkerson



Edmonds-Karp(G)
  $f \leftarrow$ an empty flow (i.e. $f(e) = 0 \quad \forall\, e \in E(G)$)
  While there exists an augmenting path in the residual graph G_f:
    Use BFS to find an augmenting path P with the fewest possible edges
    Augment f along P
  Return f

Since Edmonds-Karp is just a particular implementation of Ford-Fulkerson (rather than augmenting along an arbitrary augmenting path, we are augmenting along the shortest one) the correctness of this algorithm follows immediately from Theorem 87. The runtime analysis is a bit more complicated, but it is possible to prove the following theorem (see CLRS or Erickson for details):

Theorem 88. *If G is a flow network with integer capacities,* Edmonds-Karp(G) *outputs a maximum s-t-flow in G in $O(m^2 n)$ time.*

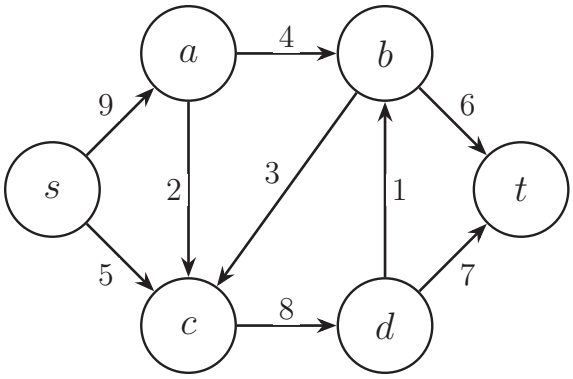
Over the 50 years since Edmonds and Karp described this algorithm in 1972, faster algorithms for max-flow were steadily developed, using increasingly complicated techniques, culminating in a 2022 algorithm by Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva which computes a maximum flow in a flow network with integer capacities in $O(m^{1+o(1)})$ time (almost linear!). As usual, you may assume a subroutine to solve max-flow in $O(m^{1+o(1)})$ time in your algorithms without needing to read and understand CKLPGS's 100 page paper.

18.5 Comprehension Questions

1. Suppose that a flow network has an $s \to t$ flow of value 100. What can we say about the min $s \to t$ cut in this network?

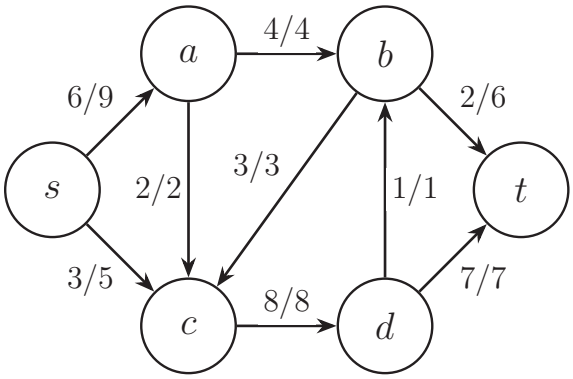
2. Describe a minimum s-t cut in the flow network below by giving the vertices which are part of S and those which are part of T.

   $S :=$ _____



$T :=$ _____

3. Below is a flow in the network from the previous problem. Give an example of an augmenting path in the residual graph for this flow.



Augmenting path: _____

4. What is the residual capacity of your augmenting path from the previous question?

Residual Capacity: _____

18.6 Exercises

1. Let G be a flow network, and suppose that (S_1, T_1) and (S_2, T_2) are two distinct min s-t-cuts in the network. Prove that $(S_1 \cup S_2, T_1 \cap T_2)$ is also a min s-t-cut in the flow network.

2. Let f and f' be s-t-flows with $|f| < |f'|$. Prove that there exists an s-t flow with value $|f'| - |f|$ in the residual graph G_f.

3. Let G be a flow network with integer edge capacities, let e be an edge of positive capacity in G, and let f be an integral max-flow for G (meaning f sends an integer amount of flow through every edge).

   - Give an $O(m+n)$-time algorithm which takes G, f, and e and computes the new maximum flow f' for the graph G' where e's capacity is increased by 1.

- Give an $O(m+n)$-time algorithm which takes G, f, and e and computes the new maximum flow f' for the graph G' where e's capacity is decreased by 1.

4. You are the owner of a large restaurant, and a group of customers has just come in asking to be served. Rather than ordering like normal customers, each of these customers has given you a list of dishes from your menu that they would be happy to eat. You want to serve each customer one of the dishes that customer has selected. In order to maintain the pristine reputation of your restaurant, you refuse to serve any two of the customers the exact same dish. Furthermore, each dish must be prepared by a chef who has truly mastered that dish, and each chef can prepare at most 4 dishes (to avoid making the customers wait too long). Suppose that you are given the following:

   - A number of customers: x,
   - A number of dishes on the menu: y,
   - A number of chefs: z,
   - A 2-D boolean array of customer preferences $Customer[1,\ldots,x][1,\ldots,y]$, where $Customer[i][j]$ is true if and only if the ith customer is happy to eat the jth dish on the menu,
   - A 2-D boolean array of chef specialties $Chef[1,\ldots,z][1,\ldots,y]$, where $Chef[i][j]$ is true if and only if the ith chef is able to prepare the jth dish on the menu.

   Give a polynomial time algorithm to determine whether or not it is possible to serve every customer a unique dish that they would be happy to eat while having each chef prepare at most 4 dishes and only prepare dishes they are able to make.

19 Bipartite Matching

In this chapter we will look at the problem of computing maximum matchings which is an important application of max-flow algorithms. We say that a graph G is bipartite if it can be partitioned into two disjoint sets L and R such that every edge in the graph crosses the cut L, R (i.e. every edge has one endpoint in L and one endpoint in R). Bipartite graphs arise naturally in problems where the goal is to match two different kinds of things (e.g. students and courses). In this chapter we will explore algorithms for solving matching problems in bipartite graphs.

19.1 Matchings: Maximum and Perfect

For a (not necessarily bipartite) graph G, we say that a **matching** is a subset M of the edges of G such that no two edges in M share an endpoint. We say that a vertex v is **matched** by M if one of the edges in M has v as an endpoint. We say that M is a:

- **maximal matching** if there is no matching M' in G with $M \subsetneq M'$ (i.e. every edge not in M shares an endpoint with one of the edges in M)
- **maximum matching** if there is no matching M' in G with $|M'| > |M|$ (i.e. M is a matching with the maximum possible number of edges)
- **perfect matching** if every vertex in G is matched by M (i.e. if $|M| = n/2$)

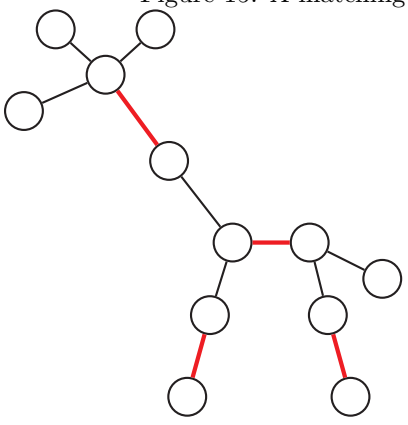
Note that every perfect matching is maximum, but not every maximum matching is perfect. Every graph has a maximum matching (even a graph with no edges has the empty matching $M = \emptyset$), but not every graph has a perfect matching.

At first the distinction between a maximal and maximum matching may not be clear. If we cannot add any more edges to M doesn't that mean that M uses the maximum possible number of edges? Not necessarily, as the following figure illustrates!

The red edges in Figure 15 form a matching of size 4 in the G-raph. This matching is maximal, because every edge not in the matching shares an endpoint with an edge from the matching (and therefore couldn't be added to the matching). This is not a maximum matching however, because the G-raph has a matching of size 5 (can you find it?).

The maximum matching in the G-raph is still not a perfect matching. We can see that this graph has no perfect matching from the fact that no matching could match both of the ossicone vertices (they both share a single neighbor, the head vertex, and only one of the two can be matched with that neighbor, meaning that any matching must leave at least one of the two unmatched).

Figure 15: A matching in the G-raph, which is maximal, but not maximum



The fact that we can have maximal matchings which are not maximum implies that the obvious greedy algorithm for finding a matching "while there exists an edge which shares no endpoints with

any edge in the matching, add such an edge to the matching", does not find a maximum matching in every graph (it can get stuck in a locally optimal matching like the one in Figure 15). Therefore if we want to compute a maximum matching we need to be a little bit more clever.

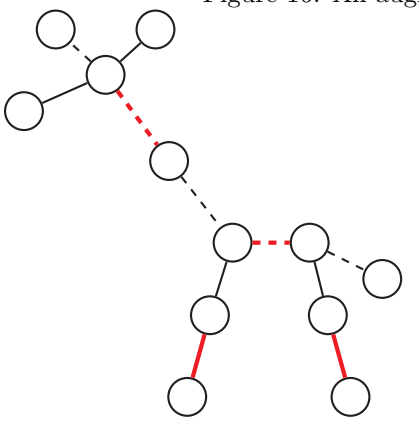
19.2 Characterizing Maximum Matchings

In this section we will prove a pair of theorems that will allow us to characterize when a matching is maximum and when a graph will have a large maximum matching.

Let $G = (V, E)$ be a graph and M be a matching in G. We say that a path P in G is **M-alternating**, if P alternates between using edges in M and edges in $E \setminus M$. We say that P is **M-augmenting** if it is an M-alternating path which begins and ends with vertices which are not matched by M.

Just as an augmenting path in the residual graph of a flow network implies that we can obtain a larger flow, an M-augmenting path implies that we can obtain a larger matching than M. In particular, we can increase the size of our matching by 1 by toggling which edges from the augmenting path P are part of the matching (swapping the edges from $E(P) \cap M$ with the ones from $E(P) \setminus M$). The dashed lines in the figure below form an augmenting path in the matching from Figure 15. Note that we could increase the size of our matching from 4 to 5 by swapping the two matching edges (the dashed red lines) on the augmenting path with the three non-matching edges on the path (the dashed black edges).

Figure 16: An augmenting path for the matching from Figure 15



In fact, any time that our matching is not maximum, we can increase its size by finding an augmenting path, as the following theorem states. The first proof of this theorem was published by Claude Berge in 1957, so it is known as Berge's theorem.

Theorem 89. *M is a maximum matching for a graph G if and only if G has no M-augmenting paths.*

Proof. To prove an "if and only if" statement we need to show two directions:

- First we show that if M is a maximum matching then M has no augmenting paths. We prove the contrapositive: If M has an augmenting path, then M is not a maximum matching.

  Let M be a matching which has an augmenting path, and let P be an M-augmenting path. Then, as noted above, we can swap all of the edges from M which are part of P for the edges of P which are not part of M to obtain a larger matching. More formally, let $M' := (M \setminus (E(P) \cap M)) \cup (E(P) \setminus M)$. We note that since P alternates between edges in $E \setminus M$ and edges in M, but begins and ends with unmatched vertices, P contains one more edge from $E \setminus M$ than from M, so $|M'| = |M| + 1$. Furthermore, M' is still a matching: It matches all of the vertices that M matched, as well as two additional vertices that were unmatched by M

(the start and end of P), so each edge of M' is matching two distinct vertices. Thus, since there is a matching larger than M, M is not a maximum matching.

- Now we show that if G has no M-augmenting paths, then M is a maximum matching for G. Again we prove the contrapositive: If M is not a maximum matching then G has an M-augmenting path.

  Suppose that M is a matching for G which is not a maximum matching. Then there exists some matching M' with $|M'| > |M|$. Consider the subgraph $G' := (V(G), M \cup M')$. Since each vertex in G has at most one edge in M and at most one edge in M' (since they are matchings), G' consists of disjoint paths and cycles. Furthermore, since edges in the same matching do not share an endpoint, each of these paths and cycles must alternate between edges of M and edges of M'. In particular, this means that all of the cycles must have an equal number of edges from M and M'. Thus, since $|M'| > |M|$, there must be some path P in G' which has more edges from M' than from M. This P must begin and end with vertices unmatched by M, so it is an M-augmenting path in G, and thus G has an M-augmenting path.

□

Berge's theorem gives us a characterization of maximum matchings in arbitrary graphs. It also suggests an algorithm for finding a maximum matching "start with an empty matching, then repeatedly find an augmenting path and toggle its edges until there are no augmenting paths." One detail of this algorithm is rather underspecified, however. How do we find an augmenting path? This turns out to be a fairly non-trivial problem. While it is possible to design an algorithm for computing a maximum matching in an arbitrary graph by repeatedly finding augmenting paths (look up Edmonds' Blossom Algorithm if you're curious), we will not discuss algorithms for maximum matching in arbitrary graphs in this course. Instead we will restrict our attention to a special class of graphs where finding a maximum matching is significantly easier.

Recall that a graph G is bipartite if we can divide the vertex set into two (not-necessarily equally sized) parts L and R such that all of the edges have one endpoint in L and the other in R (we often draw a bipartite graph with all the vertices of L on the left and all the vertices of R on the right). Note that for any matching M in a bipartite graph we must have $|M| \leq |L|$ and $|M| \leq |R|$, so G cannot have a perfect matching if $|L| \neq |R|$.

The following theorem, which was proven in 1935 by Philip Hall and is known as Hall's theorem, gives us a useful characterization of when a bipartite graph has a matching which matches all of the vertices on one side of the bipartition.

Theorem 90. *Let $G = (L \cup R, E)$ be a bipartite graph. Then G has a matching of size $|L|$ if and only if $\forall\ S \subseteq L$, $|S| \leq |N(S)|$ (Here $N(S) := \{y \in R : \exists\ x \in S, \{x, y\} \in E\}$).*

Proof. To prove an "if and only if" statement we need to show two directions:

- First we show that $(G$ has a matching of size $|L|) \Rightarrow (\forall\, S \subseteq L, |S| \leq |N(S)|)$. Let M be a matching in G of size $|L|$. Since every edge of G has one of its endpoints in L and no two edges in M can share an endpoint, we know that each vertex in L must be matched under M.

  Let $S \subseteq L$ be arbitrary. Let $T = \{y \in R : \exists\ x \in S, \{x, y\} \in M\}$. Note that since $M \subseteq E$, $T \subseteq N(S)$. We have shown above that M must match all vertices of L. In particular M matches all vertices in S. Since M is a matching, no two of its edges share an endpoint, so each vertex of S is matched to a distinct vertex in R. Therefore $|S| = |T| \leq |N(S)|$.

- Now we show that $(\forall\, S \subseteq L, |S| \leq |N(S)|) \Rightarrow (G$ has a matching of size $|L|)$. We will prove the contrapositive: If the size of a maximum matching in G is less than $|L|$, then there exists a set $S \subseteq L$ for which $|S| > |N(S)|$.

  Let M be a maximum matching in G and assume $|M| < |L|$. Then M does not match all of the vertices in L. Let $x \in L$ be a vertex which is not matched by M. Let

  $$T := \{v \in V : v \text{ can be reached through an } M\text{-alternating path starting from } x\},$$

By Theorem 89, since M is a maximum matching, there are no M-augmenting paths in G. Therefore x cannot reach an unmatched vertex along an alternating path, so every vertex in $T - \{x\}$ is matched. Let $T_L := T \cap L$ and $T_R := T \cap R$.

We note that every M-alternating path starting from x must begin with an edge of $E \setminus M$ (since x is unmatched) so it will travel from L to R on edges in $E \setminus M$ and from R to L on edges in M. Therefore, for every vertex $y \in T_R$, there is an M-alternating path from x to y which ends with an edge in $E \setminus M$. Let $z \in L$ be the vertex that y is matched with. Adding the edge y, z to the M-alternating path from x to y gives an M-alternating path from x to z, so $z \in T_L$. This means that every vertex of T_R is matched to a vertex of T_L, but T_L also contains the extra unmatched vertex x, so $|T_L| \geq |T_R| + 1$

Now we will show that $N(T_L) \subseteq T_R$. Let $v \in T_L$ be arbitrary. We need to show that $N(v) \subseteq T_R$. If $v = x$, this is true, since every neighbor w of v can be reached by the length one M-alternating path (v, w). Otherwise, v is matched by M, let $u \in R$ be the vertex that v is matched with. Since $v \in T$, there exists an M-alternating path P from x to v. As noted above, this path reaches vertices in L by following edges in M, so the last edge of P must be $\{u, v\}$. Removing this edge from P gives an M-alternating path from x to u, so $u \in T_R$. Similarly, any other vertex in R which is visited by P must be in T_R (since we can simply remove all the edges of P after reaching that vertex to get an alternating path from x to that vertex). Let w be a neighbor of v which is not visited by P. Since M can match v with only one other vertex, $\{v, w\} \notin M$. Therefore adding $\{v, w\}$ to P gives an M-alternating path from x to w, so $w \in T_R$ as well. Thus, $N(v) \subseteq T_R$, so $N(T_L) \subseteq T_R$.

Putting together these facts we have that

$$|T_L| > |T_L| - 1 \geq |T_R| \geq |N(T_L)|.$$

So G has a set $S \subseteq L$ for which $|S| > |N(S)|$.

□

Hall's theorem is useful for proving that a graph has a perfect matching without needing to explicitly construct one. For example, Hall's theorem allows us to easily prove the following theorem:

Theorem 91. *If k is a positive integer and $G = (L \cup R, E)$ is a k-regular bipartite graph, then G has a perfect matching.*

Proof. We note that since every vertex in L has degree k and every edge has one endpoint in L, $m = |L|k$. Similarly since every vertex in R has degree k and every edge has one endpoint in R, $m = |R|k$. Therefore, $|L| = |R|$, so a matching which matches all of the vertices of L would be a perfect matching.

To show that G has a matching which matches all vertices of L we will show that it satisfies Hall's condition: $\forall\, S \subseteq L, |S| \leq |N(S)|$. Let $S \subseteq L$ be arbitrary. Since every vertex in L has degree k, there are exactly $|S|k$ edges incident to one of the vertices in S. Since each vertex in R has degree k, no vertex in R may be incident to more than k of these edges. So the edges of vertices in S are incident to at least $|S|k/k = |S|$ vertices in R. Thus, $|N(S)| \geq |S|$, so by Theorem 90, G has a perfect matching. □

19.3 Finding Maximum Matching in Bipartite Graphs

So how do we find a maximum matching in a bipartite graph? The fastest and easiest way is via a reduction to max-flow!

Given a bipartite graph $G = (V, E)$ with bipartition $V = L \cup R$, we construct a flow network $G' = (V', E')$ as follows:

1. $V' = V \cup \{s, t\}$, i.e. we add two new vertices to G to serve as our source and sink.
2. E' contains the following edges:

- For each $u \in L$, E' contains an edge (s, u)
- For each $\{u, v\} \in E$, with $u \in L$, $v \in R$ E' also contains (u, v) (i.e. we direct the edges from L to R).
- For each $v \in R$, E' contains an edge (v, t).
- All edges in E' have capacity 1.

We note that every integral flow in G' corresponds to a matching in G and vice-versa:

- Since the edges (s, u) each have capacity 1, an integer flow in G' can send flow out of $u \in L$ through at most 1 edge, and since the edges (v, t) also have capacity 1, an integer flow in G' can send flow into a vertex $v \in R$ through at most 1 edge, so for any integer flow f, the edges of G that f sends positive flow through form a matching in G.
- Conversely, given a matching M in G, we can create a corresponding flow in G' by sending 1 unit of flow from s to each vertex in L which is matched by M, sending 1 unit of flow through each edge of M, and sending 1 unit of flow from each vertex of R which is matched by M to t.

Since all the edge capacities in G' are integers, we know that G has an integral maximum flow. G' has $O(n)$ vertices and $O(m + n)$ edges, so we can compute a max flow in G' (and thus find a maximum matching in G) using the algorithm of CKLPGS in $O((m + n)^{1+o(1)})$ time.

Note that if we computed this maximum flow using the Ford-Fulkerson (or Edmonds-Karp) algorithm, then the augmenting paths in the residual graph that we push flow through while running the algorithm would correspond to M-augmenting paths in G, demonstrating that our two definitions of "augmenting paths" in fact have a close relationship.

19.4 Stable Matchings

In the United States, doctors who have graduated from medical school must go through a residency program, where they obtain hands-on experience working in a hospital, before they can obtain a medical license.

This process can be fiercely competitive, from the perspective of both the young doctors and the hospital residency programs. By 1950, hospitals desperate to secure the top students were issuing "exploding" residency offers, where they would call a final-year medical student to offer them admission to a residency program and require them to decide whether or not to accept the offer on the spot before hanging up the phone.

Frustrated by these conditions, students appealed to their schools, and in 1952 a non-profit known as the National Resident Matching Program was created to fairly match doctors and hospitals for residency.

Every spring, hospitals submit a ranked list of students they would like to admit, and students submit ranked lists of hospitals where they would like to go for residency after graduation. On the third Friday in March (match day), the NRMP releases an assignment of students to hospitals, avoiding months of stressful negotiations between individual students and programs.

But how does the NRMP compute an assignment that keeps everyone happy? How do we even define what a "fair" assignment means? That is the problem that we'll discuss in this section.

The algorithm that we discuss in this section is essentially the algorithm that the NRMP actually uses to solve a massive doctor-hospital pairing problem in the real world. However, in order to simplify the algorithm and analysis a bit, we will focus on an unrealistic version of the problem with the following assumptions:

- We have n doctors and n hospitals (for some positive integer n).
- Each hospital is looking to hire only a single doctor.
- Each doctor has a preference list ranking all the hospitals from the one that they would most like to work at to the one that they would least like to work at. We will assume that these are represented as a 2-D array D where $D[i, j]$ is Doctor #i's j'th favorite hospital.

- Each hospital has a preference list ranking all of the doctors from the one that they would most like to hire to the one that they would least like to hire. We will assume that these are represented as a 2-D array H where $H[i, j]$ is Hospital $\#i$'s j'th favorite doctor.

Under these assumptions, we can view the problem as finding a perfect matching in a bipartite graph, where L is the set of doctors and R is the set of hospitals. We don't want just any perfect matching, though (otherwise we could just pair the i'th doctor with the i'th hospital for each $1 \leq i \leq n$), we want to find a *stable* perfect matching.

For a matching M between doctors and hospitals, we say that a doctor-hospital pair (d, h) is an **unstable pair** (you may also see the term "blocking pair" used) if d and h are not matched by M, but they each prefer each other over their matches under M. We say that a matching is **stable** if it has no unstable pair. Stability is a desirable property for a matching since it ensures that no doctor and hospital have an incentive to ignore the NRMP's assignment and cut their own backroom deal. Note that a stable matching does not mean that everyone is happy with their assignment. Some doctor d may get their last choice hospital, but as long as every other hospital other than the one d is matched with has been assigned a doctor they like better than d, none of them have an incentive to take d, and there is no unstable pair.

It is not obvious from the definition that a stable perfect matching always exists. We will prove that one does, however, by giving an efficient algorithm to find one. This algorithm has been in use by the NRMP since 1952, but it is named after David Gale and Lloyd Shapley, who proved in 1962 that the algorithm always produces a stable matching. This work contributed to Shapley being awarded the Nobel prize in economics in 2012.

```
GaleShapley(D, H)
  M ← ∅
  While |M| ≠ n:
      h ← a hospital not matched by M
      d ← the top doctor on h's preference list
      If d is not matched by M:
          Add (d, h) to M
      Else if d prefers h to the hospital h' they are currently matched with:
          Remove (d, h') from M
          Add (d, h) to M
      Remove d from h's preference list
  Return M
```

We can view this algorithm as hospitals making a sequence of offers to doctors. When a doctor receives her first offer, they tentatively accept it. Then, when they receive a new offer, if it is better than their current offer, they reject the previous hospital and tentatively accept the new one. Once every doctor has an offer, the doctors all accept their current offers.

Theorem 92. GaleShapley(D, H) *returns a stable perfect matching. Furthermore,* GaleShapley *can be implemented to run in* $O(n^2)$ *time.*

Proof. We first show that GaleShapley returns a perfect matching. We note that at every point during the algorithm M is always a valid matching, since we only add an edge (d, h) to M when d and h are both currently unmatched (possibly because we just discarded an edge that was previously used to match d). We also note that once a doctor is matched by M, they stay matched in every future iteration of the while loop (a doctor only discards their current hospital to accept a better offer). An unmatched doctor accepts any offer they received, becoming matched. Thus, if every doctor eventually receives an offer, then every doctor is eventually matched.

As long as some doctor is unmatched, some hospital will also be unmatched and will make offer to the top doctor they have yet to scratch off their preference list. Since every hospital begins with all n doctors on its preference list, this process can continue for only a finite number of iterations until either all doctors are matched and the algorithm terminates or some hospital reaches the end of its preference list. If a hospital h reaches the last doctor on its preference list it means that every

other doctor must already be matched to one of the other $n-1$ hospitals (otherwise they would not have rejected h's earlier offer), so the final doctor must be unmatched and will accept h's offer, terminating the algorithm.

Next we show that the perfect matching that GaleShapley returns is stable. Let (d, h) be any doctor-hospital pair that are *not* matched by M. We need to show that (d, h) is not an unstable pair. Let d' be the doctor that h is matched with by M. We consider two cases:

- If h prefers d' over d, then (d, h) is not unstable, because h would not prefer to switch to d.
- If h prefers d over d', then h must have made an offer to d before making its offer to d'. Since d and h did not end up matched in M, d must at some point have rejected h for a hospital h' that d prefers over h. After rejecting h for h', d would only reject h' for another hospital that they prefer even more. Thus, whichever hospital d ends up matched with under M must be a hospital that they prefer to h. Therefore, d would not prefer to switch to h and (d, h) is not an unstable pair.

Thus in both cases, (d, h) is not unstable. Since (d, h) was an arbitrary unmatched pair, we conclude that M has no unstable pairs and thus is a stable matching.

Finally, we analyze the runtime. We note that we can implement the hospitals moving down their preference lists by storing each hospital's preference list in a linked list. We can create these n length n linked lists in $O(n^2)$ time, and using them we can get a hospital's most preferred doctor or remove that doctor from the preference list in constant time.

Since each iteration of the loop removes a doctor from a hospital's preference list, the algorithm's while loop runs for $O(n^2)$ iterations. We first show that we can find an unmatched hospital in constant time. To do this, we maintain a list of unmatched hospitals. Each iteration we pop the first hospital h from this list. If we fail to match h, we insert h back at the front of the list, otherwise we insert the hospital that h displaced (if any) into the list.

As noted above, we can get the top doctor on a hospital's preference list in constant time. We can check whether a doctor is matched in constant time by maintaining an array storing each doctor's current match (or Null if they are unmatched).

Finally, we need to show that we can determine in constant time whether d prefers h over the hospital h' that d is currently matched with. With our stated input representation, where each doctor has an array listing their hospitals from most preferred to least preferred, determining whether d prefers h over h' takes $O(n)$ time, we need to iterate through the whole list looking for the two hospitals. Thus, before the loop we preprocess the doctors' preference lists into a more convenient format. Call the hospitals $h_1, \ldots, h_n$. For each doctor d we create an array A_d where $A_d[i]$ is the rank of h_i in d's preference list (so if d's number one choice is h_5, then $A_d[5] = 1$). We can create this array A_d from the preference list in $O(n)$ time (we simply iterate through the list once, filling in the appropriate entries of A_d as we go). Thus, we can do this for all doctors in $O(n^2)$ time before the loop, and then in the loop we can determine whether d prefers h_i to h_j by simply comparing $A_d[i]$ and $A_d[j]$.

Thus, with $O(n^2)$ preprocessing, we can ensure that each iteration of the loop runs in $O(1)$ time. Since the loop runs $O(n^2)$ times, this implies that the total runtime for the algorithm is $O(n^2)$.

□

19.5 Comprehension Questions

1. True or false? If M is a maximal matching in a graph G, then G does not contain any M-augmenting path.
2. If $n \geq 4$ is an integer and G is a graph consisting of a cycle on n vertices, then G has a perfect matching.
3. Let G be a 3-regular bipartite graph with n vertices. What can we say about the size of a maximum matching in G? (select the most precise true statement)

(a) The size of a maximum matching in G is n.

(b) The size of a maximum matching in G is at most n.

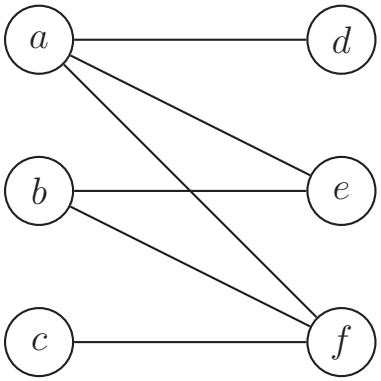
(c) The size of a maximum matching in G is $n/2$.

(d) The size of a maximum matching in G is at most $n/2$.

(e) The size of a maximum matching in G is $n/3$.

(f) The size of a maximum matching in G is at most $n/3$.

4. $M = \{\{a, e\}, \{b, f\}\}$ is a matching in the bipartite graph below. Give an example of an M-augmenting path in the graph.



5. Consider an instance of the stable matching problem where we wish to pair up 4 doctors, A, B, C, and D with 4 hospitals W, X, Y, and Z. The preference lists of the doctors and hospitals are given below (from most preferred to least preferred):

$$\begin{array}{ll} A : W, X, Y, Z & W : B, D, A, C \\ B : W, X, Y, Z & X : A, B, C, D \\ C : W, X, Y, Z & Y : B, A, D, C \\ D : W, X, Y, Z & Z : D, A, C, B \end{array}$$

(a) A-W,B-X,C-Y,D-Z is not a stable matching for these preference lists. Show why by giving an unstable pair for this matching:

(b) Give an example of a stable matching for this set of preference lists:

19.6 Exercises

1. Suppose that you are given an $n \times n$ grid where some of the cells are white and others are black. Your goal is to mark n of the *white* cells in such a way that every row contains exactly one marked cell, and every column contains exactly one marked cell. Give an algorithm which takes as input an $n \times n$ array $White[1 \ldots n][1 \ldots n]$ of booleans, where $White[i][j]$ is true if and only if the square at row i and column j is white. Your algorithm should either output a list of n (i, j) pairs indicating white cells to mark, or should correctly report that there is no valid set of cells in this grid. Your algorithm should run in polynomial time.

   For example, if $White$ is the array:

$$\begin{bmatrix} false & true & true \\ true & false & true \\ false & true & false \end{bmatrix}$$

then your algorithm should output $[(1,3),(2,1),(3,2)]$ (or some permutation of this list), corresponding to marking the top right, left middle and bottom middle cells.

2. Let $G = (V, E)$ be a (not necessarily bipartite) unweighted undirected graph. Initially none of the vertices of G are marked. Two friends, Alice and Bob, play the following game on this graph:

   - Alice begins the game by choosing any vertex of G and marking it.
   - Bob chooses an unmarked vertex adjacent to the one that Alice just marked and marks it.
   - Alice chooses an unmarked vertex adjacent to the one that Bob just marked and marks it.
   - The players continue to alternate turns, and on each turn a player marks an unmarked vertex adjacent to the vertex their opponent marked on the previous turn.
   - When a player has no legal move (because all of the vertices neighboring the one their opponent just marked have already been marked earlier in the game), they lose.

   In this problem you will help Alice and Bob develop winning strategies for this game.

   (a) Prove that if G has a perfect matching then Bob has a strategy which guarantees him a win no matter which vertices Alice chooses.

   (b) Prove that if G does not have a perfect matching then Alice has a strategy which guarantees her a win no matter which vertices Bob chooses.

3. A *cycle cover* of a directed graph G is a collection of cycles in G such that each vertex of G is in exactly one of the cycles. Give an efficient algorithm to determine whether or not a directed graph has a cycle cover (Hint: Use bipartite matching).

4. We saw that the Gale-Shapley algorithm can be used to find a stable matching between newly graduated doctors and hospitals looking to hire them, in the special case where each hospital has only one open position, each doctor has ranked all the hospitals, and each hospital has ranked all of the doctors. In this problem, you will generalize that result to a more realistic scenario.

   Suppose now that there are x doctors and y hospitals. Each hospital may have a different number of available positions. Each doctor has a preference list which is a partial ranking of the hospitals, i.e. they have ranked some *subset* of the hospitals from most wanted to least wanted (each doctor separately chooses their own set of hospitals to rank). Different doctors may rank different numbers of hospitals. Similarly, each hospital has a preference list which is a ranking for some subset of the doctors. In this context, we say that a doctor prefers a hospital that they have ranked over every hospital they have not ranked, and a doctor has no preference between hospitals they have not ranked (and vice-versa for hospitals). We say that doctor d and hospital h form an unstable pair if doctor d prefers hospital h over the hospital she is currently assigned to, and either: (i) hospital h has an open position (they are not hiring their maximum number of doctors) or (ii) hospital h prefers doctor d over at least one of the doctors they are hiring.

   Give an efficient algorithm which takes as input:

   - The number of doctors, x,
   - The number of hospitals, y,
   - A 1-D array $Positions[1, \ldots, y]$, where $Positions[i]$ is the maximum number of doctors that the ith hospital is willing to hire,
   - A 2-D array $Doctors[1, \ldots x][]$ (with rows of uneven length), where $Doctors[i]$ is the preference list of the ith doctor, and
   - A 2-D array $Hospitals[1, \ldots, y][]$ (with rows of uneven length), where $Hospitals[i]$ is the preference list of the ith hospital.

Your algorithm should output an assignment of doctors to hospitals where:

- Every doctor is assigned to a hospital,
- At most $Positions[i]$ doctors are assigned to hospital i, for every i, and
- There are no unstable pairs.

Note that you may need to pair up a doctor and a hospital which have not put each other on their preference lists. You may assume that the total number of positions is greater than or equal to the total number of doctors.

5. For a graph $G = (V, E)$ we define a **vertex cover** as a set of vertices $S \subseteq V$ such that for every edge $e \in E$, $e \cap S \neq \emptyset$ (i.e. at least one of e's two endpoints is in S). A minimum vertex cover is one which uses the minimum number of vertices (among all vertex covers in G).

   (a) Prove that if M is a matching and S is a vertex cover then $|S| \geq |M|$.

   (b) Give an example of a graph where the size of a minimum vertex cover is larger than the size of a maximum matching (Hint: your graph will not be bipartite)

   (c) It turns out that in bipartite graphs, the size of a maximum matching is equal to the size of a minimum vertex cover (this is known as Kőnig's theorem). In the remaining parts of this problem we will prove this theorem.

   Let G be a bipartite graph with bipartition L, R. From part (a) we already know that the size of the minimum vertex cover is at least the size of the maximum matching, so we just need to prove that if M is a maximum matching in a bipartite graph, then there exists a vertex cover S with $|S| = |M|$.

   Begin by proving that if M matches all of the vertices in L, then G has a vertex cover of size $|M|$.

   (d) Now suppose that M does not match all of the vertices of L. Let L_M be the set of vertices from L that M matches. We define a new *directed* graph G' on vertex set $\{s\} \cup L \cup R$ with the following edges:

   - For each edge $\{u, v\} \in M$ with $u \in L, v \in R$, G' has an edge (v, u).
   - For each edge $\{u, v\} \in E \setminus M$, G' has an edge (u, v).
   - For each vertex $u \in L \setminus L_M$, G' has an edge (s, u)

   Let $X := \text{reach}_{G'}(s)$. Prove that $S := (L \setminus X) \cup (R \cap X)$ is a vertex cover for G.

   (e) Finish the proof by showing that every vertex of S is matched by a distinct edge from M and therefore $|M| \geq |S|$.

20 Linear Programming

In this chapter we will introduce a powerful algorithmic technique called linear programming which provides a general framework that can solve a wide variety of problems, including ones we have previously encountered like shortest paths and maximum flow. In this chapter we will focus on applications of linear programming, treating algorithms for linear programming as black boxes. In the next chapter, we will look in more detail at techniques for solving linear programs.

20.1 What is a Linear Program?

Contrary to our modern concept of "programming", a linear program does not involve any code. However, when the term was coined by Tjalling Koopmans in the 1940s, computers were in their infancy, and "programming" meant something more like "optimization."

A linear program P consists of:

- A list of n variables $x_1, \ldots, x_n$,
- A list of linear constraints in those variables, where each constraint is either a linear $\leq$ inequality, a linear $\geq$ inequality, or a linear equation (strict inequalities (e.g. $<$) are not allowed), and
- An objective which is either minimizing or maximizing a linear function.

A linear function is one of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n$, where $a_1, \ldots, a_n \in \mathbb{R}$ are constants (which could be positive, negative, or zero). That is, a linear function is the sum of terms each of which is a variable multiplied with a constant. We can view a linear function as a function f operating on the vector $x = x_1, \ldots, x_n$, so that our constraints have the form $f_i(x) \leq b_i$, $f_i(x) \geq b_i$, or $f_i(x) = b_i$, where $f_1, \ldots, f_m$ are linear functions and $b_1, \ldots, b_m \in \mathbb{R}$ are constants. Note that a linear function can include terms like $x_1 - x_2$ (this is the same as $x_1 + (-1)x_2$), but cannot include expressions like x_1^2, $x_1 \cdot x_2$, or x_1/x_2.

For example, this is a linear program:

Figure 17: A linear program

$$\begin{array}{lrcl} \text{minimize} & x_1 - 2x_2 + x_3 & & \\ \text{subject to} & 2x_1 + 5x_3 + x_4 & \leq & 8 \\ & 3x_1 - 4x_2 + 2x_4 & \geq & -3 \end{array}$$

Here the variables are x_1, x_2, x_3, x_4, the constraints are $2x_1+5x_3+x_4 \leq 8$ and $3x_1-4x_2+2x_4 \geq -3$, and the objective is to minimize $f(x) = x_1 - 2x_2 + x_3$ (we call the function f we are trying to maximize/minimize the objective function).

We call an assignment of values from $\mathbb{R}$ to the variables of an LP a **feasible solution** to the LP if it satisfies all of the constraints, regardless of what objective value it achieves. For example, $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0$ is a feasible solution to the LP in Figure 17.

Figure 18: An infeasible linear program

$$\begin{array}{lrcl} \text{minimize} & x_1 + x_2 & & \\ \text{subject to} & x_1 & \leq & 5 \\ & 2x_1 + x_2 & \geq & 10 \\ & x_1 + x_2 & = & -3 \end{array}$$

Some LPs have no feasible solutions. For example, consider the LP in Figure 18. Combining the third constraint with the second we have that

$$2x_1 + x_2 = x_1 + (x_1 + x_2) = x_1 - 3 \geq 10.$$

$x_1 - 3 \geq 10$ implies that $x_1 \geq 13$. But by the first constraint, we have that $x_1 \leq 5$. There is no value we can pick for x_1 which is simultaneously greater than or equal to 13 and less than or equal to 5, so this LP has no feasible solutions. We call a linear program with no feasible solutions **infeasible**, while a linear program with at least one feasible solution is called **feasible**. Note that whether or not an LP is feasible depends only on the constraints, not on the objective function.

We call a feasible solution an **optimal solution** if no other feasible solution has a better value for the objective function ("better" meaning "larger" in a maximization LP and "smaller" in a minimization LP). We call the objective value of an optimal solution the **optimal value** of the LP. For example, $x_1 = 3, x_2 = 4$ is an optimal solution for the LP below. Note that an LP may have more than one optimal solution. This LP has infinitely many, including $x_1 = 3.5, x_2 = 3.5$, $x_1 = 12, x_2 = -5$, and $x_1 = \pi$, $x_2 = 7 - \pi$. All of these optimal solutions attain the same objective value. The optimal value for this LP is 7.

$$\begin{array}{ll} \text{maximize} & x_1 + x_2 \\ \text{subject to} & x_1 + x_2 \leq 7 \end{array}$$

Note that all of the coefficients and constants being integers does not guarantee that a linear program has an integer optimal solution. For example, the unique optimal solution for the LP below is $x_1 = 16/5, x_2 = 3/5$, with an objective value of $19/5$.

$$\begin{array}{lr} \text{maximize} & x_1 + x_2 \\ \text{subject to} & 2x_1 + x_2 \leq 7 \\ & x_1 + 3x_2 \leq 5 \end{array}$$

It is possible for a feasible linear program to have no optimal solution if it is possible to achieve arbitrarily large value for the objective value. For example, in the LP below, we can achieve an objective value of k, for any $k \in \mathbb{R}^+$ by setting $x_1 = -k$ and $x_2 = 2k$, so that our objective value is $x_1 + x_2 = k$, but $2x_1 + x_2 = 0 \leq 7$, so the constraint is satisfied. We call a linear program **unbounded** if there are feasible solutions with arbitrarily good objective values (where "good" means large for a maximization LP and "small" for a minimization LP), and **bounded** otherwise. Note that, unlike feasibility, whether or not a linear program is bounded depends on the objective function.

$$\begin{array}{ll} \text{maximize} & x_1 + x_2 \\ \text{subject to} & 2x_1 + x_2 \leq 7 \end{array}$$

We say that a linear program is in **standard form** (also sometimes called "canonical form") if:

- It has n constraints of the form $x_i \geq 0$
- The remaining m constraints are each of the form $a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n \leq b_j$, where $a_{j1}, \ldots, a_{jn}$, and b_j are constants (i.e. the remaining constraints are all $\leq$ inequalities with a constant on the right hand side).
- The objective is maximization (rather than minimization).

We can convert any LP into standard form via a few transformations:

1. If the objective is minimization, we can convert it into maximization by simply multiplying the objective function by -1. Minimizing $f(x)$ is the same as maximizing $-f(x)$.
2. To get all of our constraints in the form $f_i(x) \leq b_i$, we first convert each equation $f(x) = b$ into two inequalities: $f(x) \leq b$ and $f(x) \geq b$. Then we convert each $\geq$ inequality into a $\leq$ inequality by multiplying both sides by -1 (since $f(x) \geq b \Rightarrow -f(x) \leq -b$).
3. To add the restriction that all of our variables are non-negative, we introduce two new variables y_i and z_i for each variable x_i in the original LP. We add the constraints $y_i \geq 0$ and $z_i \geq 0$, and we replace every occurrence of x_i in the constraints and objective function with $y_i - z_i$.

   To see why this works, we note that every real number can be expressed as the difference of two non-negative real numbers: to express a positive value k we can set $y_i = k$ and $z_i = 0$, while to express $-k$ we set $y_i = 0$ and $z_i = k$.

For example, suppose that we want to convert the LP below (which is the one from Figure 18) into standard form:

$$\begin{array}{ll} \text{minimize} & x_1 + x_2 \\ \text{subject to} & x_1 \leq 5 \\ & 2x_1 + x_2 \geq 10 \\ & x_1 + x_2 = -3 \end{array}$$

We begin by converting the objective into maximization:

$$\begin{array}{ll} \text{maximize} & -x_1 - x_2 \\ \text{subject to} & x_1 \leq 5 \\ & 2x_1 + x_2 \geq 10 \\ & x_1 + x_2 = -3 \end{array}$$

Next we convert the equation into two inequalities:

$$\begin{array}{ll} \text{maximize} & -x_1 - x_2 \\ \text{subject to} & x_1 \leq 5 \\ & 2x_1 + x_2 \geq 10 \\ & x_1 + x_2 \geq -3 \\ & x_1 + x_2 \leq -3 \end{array}$$

Then we convert the $\geq$ inequalities into $\leq$ inequalities:

$$\begin{array}{ll} \text{maximize} & -x_1 - x_2 \\ \text{subject to} & x_1 \leq 5 \\ & -2x_1 - x_2 \leq -10 \\ & -x_1 - x_2 \leq 3 \\ & x_1 + x_2 \leq -3 \end{array}$$

Finally, we replace x_1 and x_2 with the new variables y_1, y_2, z_1, z_2 to introduce non-negativity constraints to get a standard form version of the LP:

$$\begin{array}{ll} \text{maximize} & -y_1 + z_1 - y_2 + z_2 \\ \text{subject to} & y_1 - z_1 \leq 5 \\ & -2y_1 + 2z_1 - y_2 + z_2 \leq -10 \\ & -y_1 + z_1 - y_2 + z_2 \leq 3 \\ & y_1 - z_1 + y_2 - z_2 \leq -3 \\ & y_1 \ , \ z_1 \ , \ y_2 \ , \ z_2 \geq 0 \end{array}$$

An LP in standard form can be represented by:

- An $m \times n$ matrix A, where $A[i, j] = a_{ij}$ is the coefficient of x_j in the i'th constraint,
- A length m vector b, where b_i is the constant on the right hand side of the i'th inequality
- A length n vector c, where c_j is the coefficient of x_j in the objective function.

Under this representation, the LP can be compactly expressed as:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0. \end{array}$$

This formulation also allows us to analyze the LP using the tools of linear algebra. However, since linear algebra is not a prerequisite for this course, we will focus on solving and applying LPs over proving their properties. Nevertheless, we'll see some reasons to care about having LPs in standard form next chapter.

20.2 Formulating Problems as Linear Programs

So what are linear programs good for? Quite a lot, actually! Many of the problems we've seen in previous chapters can be seen as special cases of linear programming.

For example, recall the maximum flow problem from Chapter 18. A flow in a flow network is defined by the capacity and flow conservation constraints, both of which are based on linear functions of the flow through each edge. Thus, we can model max flow with the following LP, which has a variable x_e for each edge in G representing the amount of flow we sent through edge e (recall that $\delta_{in}(v)$ is the set of incoming edges to v, $\delta_{out}(v)$ is the set of outgoing edges of v, and c_e is the capacity of the edge e): Here our objective is to maximize the value of the flow, i.e. the net flow out of s.

Figure 19: A linear program for maximum flow

$$\begin{array}{lrll} \text{maximize} & \sum_{e\in\delta_{out}(s)} x_e - \sum_{e\in\delta_{in}(s)} x_e & & \\ \text{subject to} & x_e & \leq c_e & \forall\, e \in E \\ & x_e & \geq 0 & \forall\, e \in E \\ & \sum_{e\in\delta_in(v)} x_e - \sum_{e\in\delta_{out}(v)} x_e & = 0 & \forall\, v \in V \setminus \{s,t\} \end{array}$$

The single source shortest paths problem from Chapters 15 and 16 can also be modeled as a linear program. Again the key observation is that we already have a linear constraint on the shortest path distances. Remember that Dijkstra's algorithm and Bellman-Ford were both based on edge relaxations taking advantage of the fact that if $(u, v) \in E$ then $dist(s, v) \leq dist(s, u) + w(u, v)$, because we can always get to v by following the shortest $s \to u$ path and then following the edge (u, v). Thus, if we define a variable x_v to represent $dist(s, v)$ for each vertex v, then our variables should satisfy the constraint $x_v \leq x_u + w(u, v)$ (or equivalently $x_v - x_u \leq w(u, v)$) for each edge (u, v). Furthermore, we know that x_s should be 0.

It's not clear what our objective should be for the shortest paths LP. At first it seems like we should use a minimization LP, since we are looking for the shortest paths. However, our constraints impose no lower bounds on the distance values, so if our goal was minimization, then we could simply set x_v to 0 for every vertex, or even to negative values.

The constraints we have established guarantee that our distance values will not produce any tense edges, i.e. they cannot be too large. In particular, summing the constraints for the edges along a shortest $s \to v$ path $s = v_1, v_2, \ldots, v_{k-1}, v_k = v$ we have that

$$\sum_{i=1}^{k-1}(x_{v_{i+1}} - x_{v_i}) \leq \sum_{i=1}^{k-1} w(v_i, v_{i+1}) = dist(s, v).$$

Canceling out terms, the summation on the left side becomes $x_{v_k} - x_{v_1} = x_v - x_s = x_v - 0 = x_v$, so we have that $x_v \leq dist(s, v)$. Thus, our constraints guarantee that each variable x_v is less than or equal to the length of the shortest $s \to v$ path. To guarantee that we also have $x_v \geq dist(s, v)$, the right idea is to make our objective maximizing the sum of the variables x_v. This will ensure that, in an optimal solution for the LP, x_v will be exactly $dist(s, v)$ (assuming every vertex is reachable from s).

Putting these ideas together, we can write the following LP for computing the shortest paths from s.

Figure 20: A linear program for single-source shortest paths

$$\begin{array}{lrll} \text{maximize} & \sum_{v\in V} x_v & & \\ \text{subject to} & x_s & = 0 & \\ & x_v - x_u & \leq w(u,v) & \forall (u,v) \in E \end{array}$$

Note that this LP can handle negative weight edges. If there is a negative weight cycle, the LP will be infeasible (no assignment of distance values to the vertices in the cycle can satisfy the "no tense edges" constraints).

If there are vertices unreachable from s, then this shortest paths LP will be unbounded. To avoid this, before creating the LP we can run DFS (or BFS) from s to determine reach(s) and remove from G all of the vertices that s cannot reach.

Linear programming can also solve problems that we haven't seen before. One important example is a generalization of maximum flow known as *min-cost max-flow*. As in a regular max-flow problem, we are given a flow network consisting of a directed graph $G = (V, E)$ with special vertices s and t where each edge e has a capacity $c(e)$. Now, however, each edge e also has an associated cost or weight $w(e)$ (since we are already using $c(e)$ for capacity, we will stick to weights). Our goal is still to find a maximum flow from s to t (i.e. a flow with the largest capacity), however if there are multiple maximum flows, then we must output the one with the minimum total weight, where the weight of a flow f is defined as $\sum_{e \in E} w(e)f(e)$.

In order to model this problem as a linear program we need to figure out how we can encode our dual objectives of maximizing the flow capacity and minimizing the weight of the flow, when our LP can only have one objective function. The idea here is that since we are requiring the flow value to be the maximum possible, we can encode our requirement for the flow out of s as a *constraint* rather than the objective. In particular, we can first solve the LP from Figure 19 to find the max flow value OPT for G, then create the following LP for min-cost max-flow:

Figure 21: A linear program for min-cost maximum flow

$$
\begin{array}{lrll}
\text{minimize} & \sum_{e \in E} w(e) x_e & & \\
\text{subject to} & x_e \leq & c_e & \forall\, e \in E \\
& x_e \geq & 0 & \forall\, e \in E \\
& \sum_{e \in \delta_{in}(v)} x_e - \sum_{e \in \delta_{out}(v)} x_e = & 0 & \forall\, v \in V \setminus \{s, t\} \\
& \sum_{e \in \delta_{out}(s)} x_e - \sum_{e \in \delta_{in}(s)} x_e = & OPT &
\end{array}
$$

Like the linear program for regular max-flow, this LP will have an optimal solution which assigns each variable an integer value, assuming that all of the edge capacities are integers. We will not formally prove this fact, but you may assume it for the exercises and quizzes (you should *not* assume this for every LP you write, because it is not true for general LPs).

20.3 Algorithms for Linear Programming

In order to discuss the runtime for solving linear programs we first need to consider how to encode linear programs as strings (i.e. how do we define the input size for an algorithm that takes a linear program). Throughout this section we will assume that n is the number of variables in our LP and m is the number of constraints.

Similar to the assumptions we have previously made for array elements and edge weights in graphs, we will assume, unless otherwise specified, that when an algorithm takes a linear program as input, the values of the constants in the LP (e.g. the coefficients of the variables) will be integers whose values are polynomial in the number of variables (i.e. $O(n^c)$ for some constant c), so that they fit in a constant number of RAM model integer variables and we can perform arithmetic on them in constant time. It is not clear that this restriction guarantees that the optimal solution to our LP (if it exists) has a small representation, but it turns out that if all of the constants in the LP are polynomially-bounded integers, then if the LP is feasible and bounded, it has an optimal solution where all of the variables are assigned rational values which can be expressed as strings of length $O(n \log n)$.

The first algorithm for solving arbitrary linear programs was developed by George Dantzig in 1947 and is known as the simplex algorithm. We will discuss this algorithm in detail in the next chapter. While the theoretical guarantees on its runtime are not great (it can take exponential time

on carefully constructed bad inputs), the simplex algorithm is simple to implement and efficient in practice for many real-world LPs, so it is still commonly used.

The first polynomial time algorithm for solving a linear program is the ellipsoid method, developed in 1979 by Leonid Khachiyan. For a linear program with n variables , the ellipsoid method runs in time $O(n^7 m \log n)$. For almost any practical LP, this is far slower than the simplex method.

The fastest known algorithms for linear programming are based on "interior point methods" pioneered by Narendra Karmarkar in the 1980s. In particular, the current best-known algorithm was published by Cohen, Lee and Song in 2019. This algorithm solves a linear program using a series of matrix multiplications, and the runtime of the algorithm depends on the runtime for matrix multiplication. With the known results at the time that Cohen, Lee and Song published their algorithm, it ran in time $O((n+m)^{2.37})$. Very recent improvements in the runtime for matrix multiplication (in particular, an October 2026 paper by Amanov, van den Berg and Zuiddam) improves the runtime of Cohen, Lee, and Song's algorithm to $O((n+m)^{2.25+o(1)})$.

20.4 Comprehension Questions

1. Suppose that P is a linear program with a unique optimal solution. Which of the following must be true about P? (select all that apply)

   (a) P is feasible

   (b) P is bounded

   (c) P has the same number of variables as constraints

   (d) P is in standard form

2. Bureaucracy Inc. makes 3 products, A, B, and C, out of widgets, gizmos, and doodads. Each product requires different components and sells for a different price. The table below shows the quantities of widgets, gizmos, and doodads required to make each product, as well as the price each product sells for.

Product	Widgets	Gizmos	Doodads	Sell Price
Product A	2	3	0	10
Product B	4	0	2	15
Product C	3	5	1	25

   Bureaucracy Inc. currently has an inventory of 200 widgets, 100 gizmos, and 50 doodads, and would like to combine these to make products which sell for the maximum total amount. They want to formulate the optimal combination of products as a linear program with variables a, b, and c, representing the quantities of products A, B, and C. Which of the following makes the most sense as the objective function for this linear program?

   (a) Maximize $a + b + c$

   (b) Maximize $10a + 15b + 25c$

   (c) Maximize $200a + 100b + 50c$

   (d) Minimize $a + b + c$

   (e) Minimize $10a + 15b + 25c$

   (f) Minimize $200a + 100b + 50c$

3. Which of the following makes the most sense as the constraints for the linear program described in the previous problem?

(a)

$$\begin{aligned} 2a + 3b &\leq 10 \\ 4a + 2c &\leq 15 \\ 3a + 5b + c &\leq 25 \\ a, b, c &\geq 0 \end{aligned}$$

(b)

$$\begin{aligned} 2a + 3b &\leq 200 \\ 4a + 2c &\leq 100 \\ 3a + 5b + c &\leq 50 \\ a, b, c &\geq 0 \end{aligned}$$

(c)

$$\begin{aligned} 2a + 4b + 3c &\leq 10 \\ 3a + 5c &\leq 15 \\ 2b + c &\leq 25 \\ a, b, c &\geq 0 \end{aligned}$$

(d)

$$\begin{aligned} 2a + 4b + 3c &\leq 200 \\ 3a + 5c &\leq 100 \\ 2b + c &\leq 50 \\ a, b, c &\geq 0 \end{aligned}$$

4. Convert the following LP to standard form:

$$\begin{array}{ll} \text{minimize} & x_1 - 3x_2 \\ \text{subject to} & -2x_1 + 5x_2 = 7 \end{array}$$

5. Recall that in the linear program for single source shortest paths we defined a variable x_v for each vertex in the graph, designed to represent $dist(s, v)$. What was our objective function for this LP?

20.5 Exercises

1. The government of Algorithmica is attempting to pass a budget for the country. A budget consists of allocating a non-negative amount of money to each of x different programs, $p_1, \ldots, p_x$. Approving a budget requires unanimous support from the country's y senators, $s_1, \ldots, s_y$. Each senator s_i has a set of programs S_i they care about, and will only support the budget if those programs collectively receive at least \$10,000 in funding. The government has b dollars of tax revenue to allocate among all the programs. Give a polynomial time algorithm that takes as input the total available money, b, the list of programs $p_1, \ldots, p_x$ and the sets of senator preferences $S_1, \ldots, S_y$ and determines whether or not there exists a budget that will be approved by all of the senators.

2. In the min-weight perfect matching problem we are given a weighted bipartite graph $G = (L \cup R, E)$ and the goal is to find a perfect matching in G with minimum total weight (among all perfect matchings), or correctly report that no perfect matching exists. Note that the weight of a matching M is $\sum_{e \in M} w(e)$. Give a polynomial time algorithm to solve the min-weight perfect matching problem.

3. Suppose that you are given n points (x_i, y_i) in the Cartesian plane with integer coordinates (i.e. a scatter plot). Your goal is to fit a line through these points. A line is defined by an equation of the form $ax + b$. For given values of a and b we define the error of the line $ax + b$ at representing the given points as $\sum_{i=1}^{n} |ax_i + b - y_i|$. Give a polynomial time algorithm to find values for a and b which describe a line with minimum error for representing the given points.

4. You have been put in charge of shipping logistics for a shockingly successful new chain of stores selling identical tungsten cubes. You are in charge of making a plan to ship cubes from k different factories to ℓ different branches of your store. You are given the following:

   - An array of integers $F[1, \ldots, k]$ where $F[i]$ is the number of cubes produced at factory i each week
   - An array of integers $S[1, \ldots, \ell]$ where $S[i]$ is the maximum number of cubes that store i can sell in a week.
   - An array of integers $P[1, \ldots, \ell]$ where $P[i]$ is the price (in dollars) that store i sells each of its cubes for (some areas are willing to pay more than others)
   - A 2-D array of integers $C[1, \ldots, k][1, \ldots, \ell]$, where $C[i, j]$ is the cost (in dollars) to ship a single cube from factory i to store j (your business is extremely bad at taking advantage of economies of scale, so you ship each cube individually).

   Give a polynomial time algorithm to determine the maximum profit that you can earn in a week by shipping cubes to stores and selling them. Be careful to ensure that your solution ships an integer number of cubes to each store (nobody wants to buy half a cube).

5. Suppose that you are given a system of n linear inequalities of the form $\alpha_i x + \beta_i y \leq \gamma_i$. These inequalities define a convex polygon P in the 2-D plane.

   (a) Give an LP whose solution computes the maximum *perimeter* axis-aligned rectangle contained within P (if P is empty, your LP should be infeasible). Axis-aligned means that the sides of the rectangle must be horizontal and vertical.

   (b) Give an LP whose solution computes the maximum *area* axis-aligned *square* contained within P (if P is empty, your LP should be infeasible). Axis-aligned means that the sides of the square must be horizontal and vertical.

21 Solving Linear Programs

In this chapter we will discuss the simplex method, the simplest and most commonly used algorithm for solving arbitrary linear programs.

21.1 Equational Form

In Section 20.1 we defined standard form for LPs where the objective is maximization, all variables are constrained to be non-negative, and all other constraints are $\leq$-constraints. We also saw how to convert an arbitrary LP to standard form. The simplex method, however, operates on LPs in a slightly different form known as **equational form**. We say that an LP is in equational form if:

- It has n constraints of the form $x_i \geq 0$
- The remaining m constraints are each of the form $a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n = b_j$, where $a_{j1}, \ldots, a_{jn}$, and b_j are constants (i.e. the remaining constraints are all equations with a constant on the right hand side).
- The objective is maximization (rather than minimization).

Note that the only difference between equational form and standard form is that our constraints (other than the non-negativity constraints) are equations rather than $\leq$-inequalities.

We have previously seen how to convert equations into inequalities to get an LP into standard form. We can also convert inequalities into equations to convert a standard form LP into equational form. The key idea is to introduce additional variables which we call *slack variables* which will show up in only one constraint each, and store the difference between the left and right hand sides of the original constraint. We introduce a new variable s_j for each of the $m \leq$ constraints in the standard form LP. We replace each constraint $f_j(x) \leq b_j$ with the new constraints $f_j(x) + s_j = b_j$ and $s_j \geq 0$.

Since we've converted all the inequalities into equations and added a non-negativity constraint for each new variable, our new LP is in equational form. Every feasible solution x in our original standard form LP corresponds to a solution to the equational form LP where we set each x_i to the same value as in x and set $s_j = b_j - \sum_{i=1}^{n} a_{ji}x_i$ for each $1 \leq j \leq m$. Since x satisfied the $\leq$ constraints, each s_j will be non-negative. On the other hand, if we have a solution x, s to the equational form LP, then we can ignore s and x is a solution to the original standard form LP. Since each slack variable is non-negative, x must satisfy the $\leq$ constraints.

For example, if we take the standard form LP from the end of Section 20.1, reproduced below,

$$\begin{array}{lrcl}
\text{maximize} & -y_1 + z_1 - y_2 + z_2 & & \\
\text{subject to} & y_1 - z_1 & \leq & 5 \\
& -2y_1 + 2z_1 - y_2 + z_2 & \leq & -10 \\
& -y_1 + z_1 - y_2 + z_2 & \leq & 3 \\
& y_1 - z_1 + y_2 - z_2 & \leq & -3 \\
& y_1 \ , \ z_1 \ , \ y_2 \ , \ z_2 & \geq & 0
\end{array}$$

we can convert it to the following equivalent LP in equational form:

$$\begin{array}{lrcl}
\text{maximize} & -y_1 + z_1 - y_2 + z_2 & & \\
\text{subject to} & y_1 - z_1 + s_1 & = & 5 \\
& -2y_1 + 2z_1 - y_2 + z_2 + s_2 & = & -10 \\
& -y_1 + z_1 - y_2 + z_2 + s_3 & = & 3 \\
& y_1 - z_1 + y_2 - z_2 + s_4 & = & -3 \\
& y_1 \ , \ z_1 \ , \ y_2 \ , \ z_2 \ , \ s_1 \ , \ s_2 \ , \ s_3 \ , \ s_4 & \geq & 0
\end{array}$$

A **linear combination** of a set of equations $L_1 = R_1, L_2 = R_2, \ldots, L_k = R_k$ is an equation in the form $\sum_{i=1}^{k} c_i L_i = \sum_{i=1}^{k} c_i R_i$ where $c_1, \ldots, c_k$ are constants. That is, a linear combination of equations is formed by multiplying each equation by a constant and then adding them together.

For example, the equation $8y = 16/3$ is a linear combination of the equations $x + 2y = 3$ and $6x - 12y = 2$, because we can obtain it by adding 2 times $x + 2y = 3$ to $-1/3$ of $6x - 12y = 2$ ($2(x + 2y) + (-1/3)(6x - 12y) = 8y$ and $2(3) + (-1/3)(2) = 16/3$).

We say that a constraint is **redundant** if it is a linear combination of other constraints in the LP. We will also assume for the rest of this chapter that our LPs in equational form do not contain any redundant constraints. This is a natural restriction, because if we have redundant constraints, we can delete one of them without changing the set of feasible solutions for the LP (any assignment to the variables which satisfies the other constraints will also satisfy a linear combination of them).

21.2 The Geometry of Linear Programs

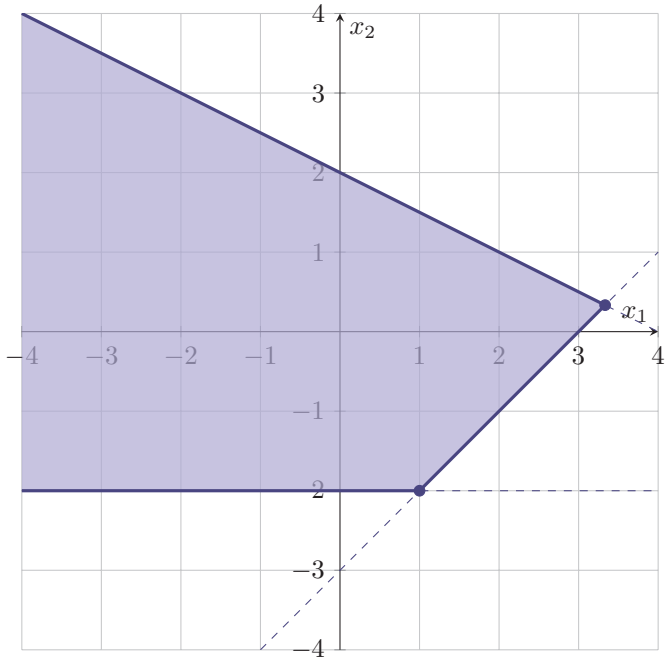
In this section we will discuss a geometric perspective on linear programming in order to develop an intuition for what the simplex is doing and why it works. If P is a linear program with variables $x_1, \ldots, x_n$, we can view a solution $x_1 = y_1, x_2 = y_2, \ldots, x_n = y_n$ to P as representing a point $y = (y_1, y_2, \ldots, y_n)$ in n-dimensional space. We define the **feasible region** of a linear program P as the set of points in n-dimensional space which are feasible solutions to P (i.e. the set of points whose coordinates satisfy all of the inequalities).

For example, Figure 23 depicts the feasible region of the LP below.

Figure 22: A linear program

$$\begin{array}{lrcl} \text{maximize} & x_1 - 2x_2 & & \\ \text{subject to} & x_1 + 2x_2 & \leq & 4 \\ & x_1 - x_2 & \leq & 3 \\ & x_2 & \geq & -2 \end{array}$$

Figure 23: The feasible region for the LP in Figure 22



For an LP with two variables, each constraint divides the Cartesian plane in half along a line. The shaded region is the part of the plane which is on the correct side of all three lines. Note that the

feasible region extends infinitely off to the left of the picture. The feasible region of an LP depends only on the constraints, not the objective function. We can view the objective function as giving us a direction that we want to travel, and an optimal solution is a point that is as far as possible in the objective direction while remaining in the feasible region.

The feasible region of a linear program has some nice properties. In particular, it is *convex*, which means that for any two points in the feasible region, any point on the line between them is also in the feasible region. We state this formally in the following theorem (the proof is not complicated, but we omit it to avoid getting bogged down in linear algebra).

Theorem 93. *If $y = (y_1, \ldots, y_n)$ and $z = (z_1, \ldots, z_n)$ are feasible solutions to a linear program P, then for every $0 \leq \alpha \leq 1$, $\alpha y + (1 - \alpha)z$ (i.e. the solution $x_i = \alpha y_i + (1 - \alpha)z_i$) is also a feasible solution to P.*

We say that a solution x is a **vertex** of P (not to be confused with a vertex of a graph) if x lies on a corner of the feasible region, i.e. if x does not lie on a line between any two distinct points in the feasible region for P. For example, the vertices for the LP in Figure 22 are $(10/3, 1/3)$ and $(1, -2)$. Vertices of the LP in Figure 22 correspond to points where two of the lines in Figure 23 intersect, i.e. points where the left hand side is exactly equal to the right hand side for two of the three constraints in the LP.

The unique optimal solution to the LP in Figure 22 is the vertex $(1, -2)$. This is not a coincidence, it turns out that when an LP has an optimal solution, it *always* has an optimal solution at a vertex.

Theorem 94. *If P is an LP with an optimal solution (i.e. if P is feasible and bounded) and P has a vertex, then P has an optimal solution which is a vertex.*

We will not prove this theorem either, but an intuition is that if we start anywhere in the feasible region and move in the direction of the objective function until we get stuck, then we will either be at a corner of the feasible region or we will hit a boundary of the feasible region which is directly perpendicular to our objective direction, in which case any point on that boundary is an optimal solution, including one of its corners (e.g. if we change the objective of the LP in Figure 22 to "maximize $x_1 + 2x_2$", then there are infinitely many optimal solutions along the line $x_1 + 2x_2 = 4$, but one of them is the vertex $(10/3, 1/3)$).

Theorem 94 is the inspiration for the simplex method. We give high level pseudocode for the method below (this version assumes that the input LP is feasible and bounded, although as we'll see later the simplex method can be used to verify both of those conditions).

```
SIMPLEX(P)
  Find a vertex y of the feasible region for P
  While there exists a vertex of the feasible region adjacent to y with a larger objective value:
      y ← an adjacent vertex with a higher objective value
  Return y
```

That is, the idea of the simplex method is to simply walk around the border of the feasible region, moving from corner to corner in a way that always makes progress in the direction of the objective function. This is essentially a greedy algorithm, but Theorem 93 guarantees that it can never get stuck in a local optimum. The line between our current point y and an optimal solution for the LP always lies within the feasible region, so we can always make progress in the direction of the objective function. Since every iteration of the while loop improves the objective value, we will never repeat vertices, so the simplex method is guaranteed to terminate (although it may take exponential time), because an LP can have only finitely many vertices (each one corresponds to a subset of the constraints).

In Section 21.3 we will explain how we can find an adjacent vertex with higher objective value (we call the process of moving from one vertex to an adjacent vertex "pivoting"), and in Section 21.4 we will explain how to implement the first line and find a vertex in the first place (or determine that the LP is infeasible).

21.3 Pivoting

Vertices of LPs in equational form have a particularly nice structure. Let P be an LP in equational form with n variables and m equations. Recall that vertices occur where boundaries of the feasible region intersect. Since P is in equational form, it is bounded by the non-negativity constraints, and each of its vertices will be a point where at most m of the variables are non-zero.

To find a vertex for P, we can choose a set of m variables which we will not force to be 0 (we call these variables a **basis**), and solve our system of equations to obtain values for these variables assuming that the other variables are 0. If the values that we obtain are all non-negative, then the solution that we've found is feasible, and we call it a **basic feasible solution**. By restricting our input to LPs in equational form, the simplex method consists of pivoting between basic feasible solutions. Pivoting to an adjacent vertex involves bringing a new variable into the basis and kicking an old variable out (so that our basis always has size m, where m is the number of equations in the LP).

So how do we find the solution associated with a given basis? The idea is to take linear combinations of our equations in order to obtain equations which involve only 1 variable from our basis (you may have learned this as the "elimination method" in middle or high school algebra). For example, suppose that we have the following LP and want to find a solution with basis x_1, x_2, x_3.

$$\begin{array}{lrcl}
\text{maximize} & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 & & \\
\text{subject to} & x_1 + 2x_3 + 3x_4 + x_5 & = & 7 \\
& 2x_1 + x_2 + x_4 + x_5 & = & 4 \\
& x_1 + 4x_2 + 3x_3 - 3x_5 & = & 18 \\
& x_1, x_2, x_3, x_4, x_5 & \geq & 0
\end{array}$$

We start by adding or subtracting multiples of the equations from each other to eliminate x_1 from all but 1 of the equations. If we subtract the first equation from the third, we eliminate x_1 from the third equation, obtaining the equivalent LP:

$$\begin{array}{lrcl}
\text{maximize} & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 & & \\
\text{subject to} & x_1 + 2x_3 + 3x_4 + x_5 & = & 7 \\
& 2x_1 + x_2 + x_4 + x_5 & = & 4 \\
& 4x_2 + x_3 - 3x_4 - 4x_5 & = & 11 \\
& x_1, x_2, x_3, x_4, x_5 & \geq & 0
\end{array}$$

Then if we subtract twice the first equation from the second equation, we get an equivalent LP where only the first equation involves x_1:

$$\begin{array}{lrcl}
\text{maximize} & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 & & \\
\text{subject to} & x_1 + 2x_3 + 3x_4 + x_5 & = & 7 \\
& x_2 - 4x_3 - 5x_4 - x_5 & = & -10 \\
& 4x_2 + x_3 - 3x_4 - 4x_5 & = & 11 \\
& x_1, x_2, x_3, x_4, x_5 & \geq & 0
\end{array}$$

Next we eliminate x_2 from all equations but the second by subtracting four times the second equation from the third equation to obtain:

$$\begin{array}{lrcl}
\text{maximize} & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 & & \\
\text{subject to} & x_1 + 2x_3 + 3x_4 + x_5 & = & 7 \\
& x_2 - 4x_3 - 5x_4 - x_5 & = & -10 \\
& 17x_3 + 17x_4 & = & 51 \\
& x_1, x_2, x_3, x_4, x_5 & \geq & 0
\end{array}$$

We divide the third equation by 17 so that it has only a single copy of x_3:

$$\begin{array}{lrcrcrcrcl}
\text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 \\
\text{subject to} & x_1 & & & + & 2x_3 & + & 3x_4 & + & x_5 = 7 \\
& & & x_2 & - & 4x_3 & - & 5x_4 & - & x_5 = -10 \\
& & & & & x_3 & + & x_4 & & = 3 \\
& x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 \geq 0
\end{array}$$

Finally, we eliminate x_3 from the first two equations, by subtracting twice the third equation from the first equation and adding four times the third equation to the second equation. This at last gives us an LP equivalent to our original LP where each variable of our basis appears in an equation without the other two:

$$\begin{array}{lrcrcrcrcl}
\text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 \\
\text{subject to} & x_1 & & & & & + & x_4 & + & x_5 = 1 \\
& & & x_2 & & & - & x_4 & - & x_5 = 2 \\
& & & & & x_3 & + & x_4 & & = 3 \\
& x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 \geq 0
\end{array}$$

Setting x_4 and x_5 to 0, we can read values for x_1, x_2, x_3 off the right hand side and we find that $x = (1, 2, 3, 0, 0)$ is a basic feasible solution for the original linear program, with an objective value of $1(1) + 2(2) + 3(3) = 14$.

To move to an adjacent vertex, we need to replace one of x_1, x_2, x_3 in our basis with either x_4 or x_5. We could try repeating the above process from scratch for all 3×2 combinations of a variable leaving the basis and variable entering the basis, but it's possible to do things a bit faster.

Let's suppose that we decide that we want to bring x_4 into the basis. Then, we are going to increase x_4 (which is 0 in our current solution) while keeping $x_5 = 0$. In order to figure out which variable leaves, we can express x_1, x_2, x_3 in terms of x_4 using the last version of the LP that we wrote above. We get that

$$\begin{aligned}
x_1 &= 1 - x_4 \\
x_2 &= 2 + x_4 \\
x_3 &= 3 - x_4
\end{aligned}$$

Thus, if we increase x_4 while keeping the equations true, x_1 and x_3 will decrease, while x_2 will increase. We reach a new basic feasible solution when one of our current basis variables hits 0 and drops out of the basis. Which one hits 0 first? Solving the equations above, we find that x_1 would hit 0 at $x_4 = 1$, while x_3 would hit 0 at $x_4 = 3$ (x_2 will never hit 0 as we increase x_4). Thus, x_1 hits 0 first, and so in our new basic feasible solution, x_4 replaces x_1. We have $x_4 = 1$, $x_2 = 2 + 1 = 3$, and $x_3 = 3 - 1 = 2$. Thus, if we bring x_4 into the basis, our new basic feasible solution would be $(0, 3, 2, 1, 0)$ with an objective value of $2(3) + 3(2) + 1(4) = 16$. This is larger than the objective value of our previous basic feasible solution $(1, 2, 3, 0, 0)$, so we pivot to $(0, 3, 2, 1, 0)$ (if bringing in x_4 had decreased the objective value we would have to try bringing in x_5 instead).

From our previous version of the LP where x_1, x_2, x_3 are by themselves, we can easily obtain a new version where x_2, x_3, and x_4 are by themselves instead. We add the first equation to the second and subtract the first equation from the third, so that our first equation now isolates x_4 instead of x_1:

$$\begin{array}{lrcrcrcrcl}
\text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 \\
\text{subject to} & x_1 & & & & & + & x_4 & + & x_5 = 1 \\
& x_1 & + & x_2 & & & & & & = 3 \\
& -x_1 & & & + & x_3 & & & - & x_5 = 2 \\
& x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 \geq 0
\end{array}$$

To continue the simplex method we need to find another adjacent vertex (basic feasible solution to pivot to). Since we just removed x_1 from our basis, adding it back would return us to the same

vertex that we already tried with the basis x_1, x_2, x_3. Therefore, the only neighboring vertex we need to consider is the one we would reach by introducing x_5 to the basis.

Like we did before, we represent our current variables in terms of x_5 to see how they change when x_5 enters the basis:

$$\begin{aligned} x_2 &= 3 \\ x_3 &= 2 + x_5 \\ x_4 &= 1 - x_5 \end{aligned}$$

This time, x_4 is the only variable that decreases as x_5 increases, so x_4 will be the variable to leave the basis, hitting 0 at $x_5 = 1$. Our new basic feasible solution is $(0, 3, 3, 0, 1)$ with an objective value of $2(3) + 3(3) + 5(1) = 20$. Once again, our objective value has increased, so we pivot to this new vertex, and rewrite our LP to isolate x_5 instead of x_4 (by adding the first equation to the third):

$$\begin{array}{lrcrcrcrcl} \text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 \\ \text{subject to} & x_1 & & & & & + & x_4 & + & x_5 = 1 \\ & x_1 & + & x_2 & & & & & & = 3 \\ & & & & & x_3 & + & x_4 & & = 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 \geq 0 \end{array}$$

Again adding x_4 to the basis would only send us back to the vertex we were just at, so we continue by looking at the vertex we get by adding x_1 into the basis. Expressing the basis variables in terms of x_1 we have:

$$\begin{aligned} x_2 &= 3 - x_1 \\ x_3 &= 3 \\ x_5 &= 1 - x_1 \end{aligned}$$

So as we increase x_1, x_5 hits 0 first at $x_1 = 1$. The new basic feasible solution would be $(1, 2, 3, 0, 0)$, with an objective value of 14 (this is in fact the same vertex that we started at). The objective value of this vertex is smaller than our current objective value of 20. Since introducing x_1 or x_4 into the basis sends us to a solution with a smaller objective value, we conclude that we have reached an optimal solution for the LP, $(0, 3, 3, 0, 1)$ with objective value 20.

Note that our initial choice to bring x_4 into the basis instead of x_5 was arbitrary and ended up making things harder for ourselves. If we had started by bringing in x_5, we would have immediately reached the optimal solution, using only 1 pivot step instead of 2. There is no known method to choose pivots that get us to the optimal solution as fast as possible (or even in a guaranteed polynomial number of pivots).

Nevertheless, there are a variety of "pivot rules" that we can use to try to make smarter pivoting decisions with the simplex method. One simple pivot rule is to look at all adjacent vertices and choose the one which increases the objective value the most: e.g. instead of just bringing x_4 into the basis, we look at the new basic feasible solution we get by introducing x_4 and the one we get by introducing x_5 instead and pivot to whichever one has the larger objective value. This works pretty well in practice, although it is still possible to construct linear programs for which this rule takes exponentially many pivots to reach the optimal solution.

If the LP was unbounded then at some point while running the simplex method we would have reached a point where there is a variable that we could bring into the basis and increase as much as we want without any other variables going to 0.

21.4 Finding a Basic Feasible Solution

One step that we omitted in the previous section was how we found our starting basis in the first place. In the previous section we guessed that there would be a basic feasible solution with basis x_1, x_2, x_3 and we turned out to be right. This is not guaranteed to work, however.

For example, if we had started by trying to obtain a basic feasible solution for the LP in the previous section with basis x_1, x_2, x_4, then using the elimination method to isolate each variable would have given us the LP:

$$\begin{array}{lrcrcrcrcl} \text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 & \\ \text{subject to} & x_1 & & & - & x_3 & & & + & x_5 & = -2 \\ & & & x_2 & + & x_3 & & & - & x_5 & = 5 \\ & & & & & x_3 & + & x_4 & & & = 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & \geq 0 \end{array}$$

Reading out the values for x_1, x_2, x_4 here we get the vector $x = (-2, 5, 0, 3, 0)$. But while these value satisfy the equations, this is *not* a feasible solution, because it does not satisfy the non-negativity constraints.

In order to find a basic feasible solution without having to guess the correct basis, we can use "two-phase" simplex method, where we create a new LP which has an obvious basic feasible solution, and whose solution gives us a basic feasible solution for the original LP. In particular, we use a similar technique to the one that we used for converting a standard form LP to equational form.

First, if there are any equations where the right hand side is negative, we multiply both sides of those equations by -1 (this will yield an equivalent equation). Then, we convert each equation $a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n = b_j$ in the LP to $a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n + z_j = b_j$ where z_j is a new variable which is only used in that equation. Since we have already guaranteed that the constants b_j on the right side of each equation are non-negative, this guarantees that our new LP has a basic feasible solution with basis $z_1, \ldots, z_m$ where we set all of the variables of the original LP to 0 and set each $z_j = b_j$. In order to try to find a basis for the original LP we make the objective of our new LP "minimize $z_1 + z_2 + \cdots + z_m$". For example, if our original LP is

$$\begin{array}{lrcrcrcrcl} \text{maximize} & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & + & 5x_5 & \\ \text{subject to} & x_1 & & & - & x_3 & & & + & x_5 & = -2 \\ & & & x_2 & + & x_3 & & & - & x_5 & = 5 \\ & & & & & x_3 & + & x_4 & & & = 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & \geq 0 \end{array}$$

then our new LP would be

$$\begin{array}{lrcrcrcrcrcrcrcl} \text{minimize} & & & & & & & & & z_1 & + & z_2 & + & z_3 & \\ \text{subject to} & -x_1 & & & + & x_3 & & & - & x_5 & + & z_1 & & & = 2 \\ & & & x_2 & + & x_3 & & & - & x_5 & & & + & z_2 & = 5 \\ & & & & & x_3 & + & x_4 & & & & & & + & z_3 = 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & , & z_1 & , & z_2 & , & z_3 \geq 0 \end{array}$$

If the new LP has an optimal solution with objective value 0 (i.e. a solution where all of our new variables are 0) we can use it as a starting basic feasible solution to run the simplex method on the original LP with the original objective function. On the other hand, if our new LP has an optimal solution with objective value greater than 0, then the original LP must be infeasible.

21.5 LP Duality

We will conclude our discussion of linear programming by talking about an important property of LPs known as *duality*, which can allow us to relate LPs for seemingly unrelated problems.

Suppose that we want to upper bound the optimum value for an LP in standard form. To take a concrete example, let's say that we want to give an upper bound for the best possible objective value in the LP below:

$$\begin{array}{lrcrcrcl} \text{maximize} & x_1 & + & x_2 & & & & \\ \text{subject to} & 2x_1 & + & x_2 & + & 4x_3 & \leq & 2 \\ & x_1 & + & 2x_2 & + & 3x_3 & \leq & 3 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0 \end{array}$$

One observation that we can immediately make is that the optimal objective value is at most 2. This is because our non-negativity constraint guarantees that our objective function is less than the left hand side of our first constraint. That is,

$$x_1 + x_2 \leq 2x_1 + x_2 + 4x_3 \leq 2.$$

We can do better than this, though, by taking a linear combination of the constraints. If we take 1/3 of the first constraint plus 1/3 of the second constraint we get

$$x_1 + x_2 + (7/3)x_3 \leq 5/3.$$

Since all of the coefficients on the left hand side of this inequality are greater than or equal to the corresponding coefficients in our objective function (in this case the coefficients in the objective function are $1, 1$, and 0), the left hand side of this inequality gives us another upper bound on the objective function, and we can say that the optimal value is at most $5/3$.

What is the best upper bound we could obtain by taking a linear combination of the constraints like this? To determine this we can construct a new LP, where the variables represent the coefficients in a linear combination of the constraints of the original LP. We call the original LP the **primal** and this new LP the **dual**. The dual has a variable y_j for each constraint $\sum_{i=1}^{n} a_{ji}x_i \leq b_j$ in the primal. For example, the dual to our LP above will have two variables y_1 and y_2

In order to guarantee that our linear combination of the constraints actually forms an upper bound on the objective function, we need to guarantee that the coefficient of each variable in the left hand side of our linear combination of constraints is at least the coefficient of that variable in the objective function, i.e. $\sum_{j=1}^{m} a_{ji}y_j \geq c_i$. For our LP above this means that we need:

- $2y_1 + y_2 \geq 1$ (so that y_1 times the first constraint plus y_2 times the second will have a coefficient of at least 1 for x_1),
- $y_1 + 2y_2 \geq 1$ (so that y_1 times the first constraint plus y_2 times the second will have a coefficient of at least 1 for x_2),
- $4y_1 + 3y_2 \geq 0$ (so that y_1 times the first constraint plus y_2 times the second will have a coefficient of at least 0 for x_3),

We also need all of our new variables to be non-negative (since multiplying a constraint by a negative value flips the direction of the inequality).

Finally, in order for our linear combination to form the best possible upper bound on the objective value, we want the right hand side to be as small as possible. i.e. we want to minimize $\sum_{j=1}^{m} b_j y_j$. Putting it all together, we find that the dual of the LP from the start of this section is the following LP:

$$\begin{array}{ll} \text{minimize} & 2y_1 + 3y_2 \\ \text{subject to} & 2y_1 + \ y_2 \geq 1 \\ & y_1 + 2y_2 \geq 1 \\ & 4y_1 + 3y_2 \geq 0 \\ & y_1 \ , \ \ y_2 \geq 0 \end{array}$$

Recall that an LP in standard form can be expressed as:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0. \end{array}$$

The corresponding dual LP is:

$$\begin{array}{ll} \text{minimize} & b^T y \\ \text{s.t.} & A^T y \geq c \\ & y \geq 0. \end{array}$$

The reason we care about duality is that there is a close relationship between the optimal solutions to the primal and the dual. In particular, they always have equal value, as the following theorem states (this theorem is often known as "strong LP duality").

Theorem 95. *If an LP P in standard form is feasible and bounded then so is its dual D (and vice-versa). Moreover, their optimal values are equal.*

Since we want to avoid getting too into the algebraic weeds of linear programming, we will not prove this theorem. It has a variety of applications. In particular, we can view the max-flow min-cut theorem (Theorem 85) as a special case of LP duality.

21.6 Comprehension Questions

1. Convert the following LP to equational form:

$$\begin{array}{ll} \text{maximize} & 2x_1 + 3x_2 \\ \text{subject to} & x_1 - 2x_2 \leq 4 \\ & x_1 \ , \quad x_2 \geq 0 \end{array}$$

2. Suppose that $x = (1, 2, 3)$ and $x = (1, 2, 5)$ are both feasible solutions to an LP P with three variables. Give another solution that must be feasible for this LP.

3. True or false? If P is an LP with n variables and m constraints then the number of vertices in P's feasible region is $O(mn)$.

4. Suppose that P is a feasible and bounded LP in standard form with a feasible solution that achieves an objective value of 14. What can we say about the optimal objective value for the dual of P?

   (a) The optimal objective value for the dual is at most 14.

   (b) The optimal objective value for the dual is exactly 14.

   (c) The optimal objective value for the dual is at least 14.

   (d) There is not enough information provided to say anything about the optimal objective value for the dual.

5. How many variables does the dual of the LP below have?

$$\begin{array}{lrrrrrr} \text{maximize} & x_1 & - 2x_2 & + & x_3 & & \\ \text{subject to} & 2x_1 & & + 7x_3 & + 12x_4 & \leq 17 \\ & 3x_1 & + 8x_2 & & + 19x_4 & \leq 28 \\ & x_1 \ , & x_2 \ , & x_3 \ , & x_4 \geq & 0 \end{array}$$

21.7 Exercises

1. Convert the LP below to equational form

$$\begin{array}{lrrrrrr} \text{minimize} & x_1 & - 2x_2 & + & x_3 & & \\ \text{subject to} & 2x_1 & & + 7x_3 & + 12x_4 & \leq 50 \\ & 3x_1 & + 8x_2 & & + 19x_4 & \geq 79 \end{array}$$

2. Suppose we are running the simplex method on the LP in equational form below, using the pivot rule "pivot to whichever adjacent vertex increases the objective value the most". If we are currently at the vertex $x = (5, 8, 0, 0)$, which vertex will we pivot to next? What is the optimal solution to the LP?

$$\begin{array}{lrrrrrr} \text{maximize} & x_1 & - 2x_2 & + & x_3 & & \\ \text{subject to} & 2x_1 & + 5x_2 & + & 7x_3 & + 12x_4 = 50 \\ & 3x_1 & + 8x_2 & + & 11x_3 & + 19x_4 = 79 \\ & x_1 \ , & x_2 \ , & x_3 \ , & x_4 \geq & 0 \end{array}$$

3. Find the dual of each of the following LPs (you do not need to solve the LP).

   (a)

$$\begin{array}{ll} \text{maximize} & x_1 - 2x_2 + x_3 \\ \text{subject to} & 2x_1 + 5x_2 + 7x_3 + 12x_4 \leq 50 \\ & 3x_1 + 8x_2 + 11x_3 + 19x_4 \leq 79 \\ & x_1 \ , \ x_2 \ , \ x_3 \ , \ x_4 \geq 0 \end{array}$$

   (b)

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 + 3x_4 - 2x_5 \\ \text{subject to} & 3x_2 - x_3 + 7x_5 \leq 14 \\ & -x_1 + 4x_3 + 3x_5 \leq 8 \\ & x_1 + x_2 + x_3 + x_4 + x_5 \leq 12 \\ & x_1 \ , \ x_2 \ , \ x_3 \ , \ x_4 \ , \ x_5 \geq 0 \end{array}$$

4. Suppose we are running the simplex method on the LP in equational form below, using the pivot rule "pivot to whichever adjacent vertex increases the objective value the most". If we are currently at the vertex $x = (3, 1, 0, 0)$, which vertex will we pivot to next? (You do not need to solve the LP)

$$\begin{array}{ll} \text{maximize} & x_3 + x_4 \\ \text{subject to} & x_1 + 3x_2 + x_3 = 6 \\ & 2x_1 + 4x_2 + x_4 = 10 \\ & x_1 \ , \ x_2 \ , \ x_3 \ , \ x_4 \geq 0 \end{array}$$

5. Find an optimal solution to the LP below. Show your work.

$$\begin{array}{ll} \text{maximize} & 2x_1 + 3x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 4 \\ & 3x_1 - x_2 \leq 3 \\ & x_1 \ , \ x_2 \geq 0 \end{array}$$

6. An integer program is a linear program with the additional constraint that all variables must take integer values. This additional constraint allows integer programs to model some problems that linear programs cannot. Unfortunately solving integer programs turns out to be much harder than solving linear programs (at least with the best known algorithms). Thus, when the natural way to model a problem is with an integer program, computer scientists often consider the *LP relaxation* of that IP, which is the LP obtained by removing the constraint that the variables must have integer values. Sometimes it is possible to argue that this LP relaxation will have the same optimum value as the integer program, meaning that the LP relaxation still has an integral optimum solution.

   (a) Give an integer program for the bipartite matching problem (where we are given a bipartite graph G and need to find a matching in G with as many edges as possible). [Hint: create one variable for each edge of G]

   (b) Prove that the LP relaxation of your integer program from part (a) has an integral optimum solution [Hint: It might be helpful to apply what you know about flows]

   (c) Compute the dual of your LP relaxation from part (b)

   (d) Can you figure out what problem the dual you found in part (c) represents?