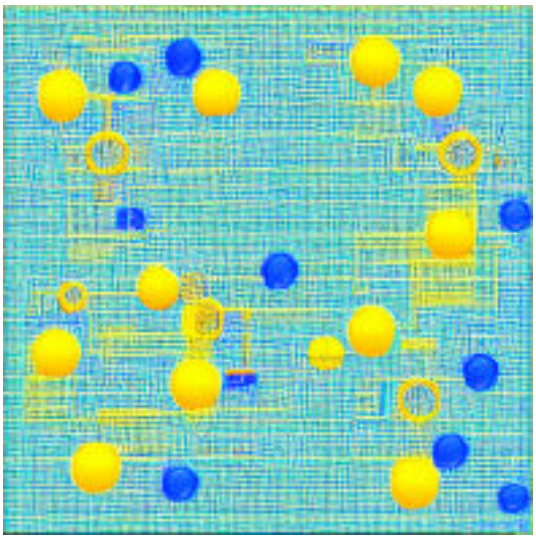


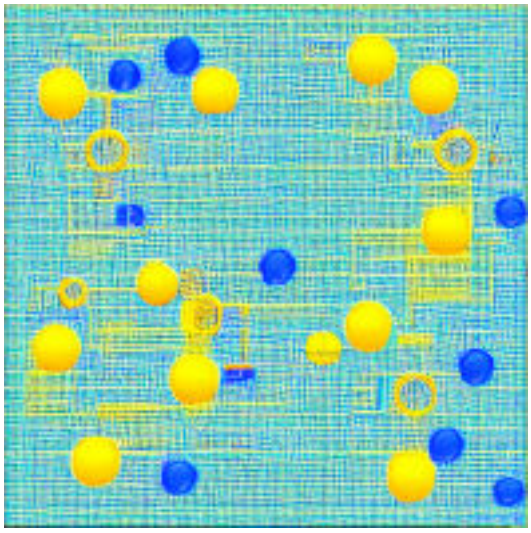
A First Course in Algorithms

Bo Waggoner

Aug 1, 2025

Table of contents

About this book



This book covers a one-semester undergraduate course in Algorithms. It grew out of a set of lecture notes and is best-suited as **technical reference material** rather than as a fully stand-alone algorithms text.

Some inline exercises are included.

Book homepage: https://bwag.prof/firstalg/

Prerequisites

Programming.

- Experience reading and writing short programs; familiarity with variables, loops, conditionals (if statements), and so on.
- Ability to step through the execution of a short program using pen and paper.
- Experience with recursion.
- Familiarity with python syntax is beneficial.

Data Structures.

- Arrays, lists, queues.
- Familiarity with binary search trees is beneficial.

Discrete Math.

- Proofs; proofs of "there exists" and "for all" statements.
- Proof by induction.
- Counting, summations, sums of series such as $1 + 2 + \cdots + n$.
- Modular arithmetic.
- Familiarity with graphs and trees is beneficial.
- Basic probability is slightly beneficial.

Calculus.

- Derivatives.
- Limits.
- Properties of exponentials and logarithms, such as $2^{x+y} = 2^x 2^y$ and $\log(x^k) = k \log(x)$.

License and Citation

Created with Quarto.

Citation:

```
@book{waggoner2025first,
  title = {A First Course in Algorithms},
  author = {Waggoner, Bo},
  year = {2025},
  publisher = {Self published},
  url = {https://bwag.prof/firstalg/}
}
```

Part I

Topic A: Big-O Notation and Complexity

1 Big-O Notation

This section recalls big-O notation. Throughout the course, we will use big-O to analyze algorithms in terms of time and space complexity.

Objectives. After learning this material, you should be able to:

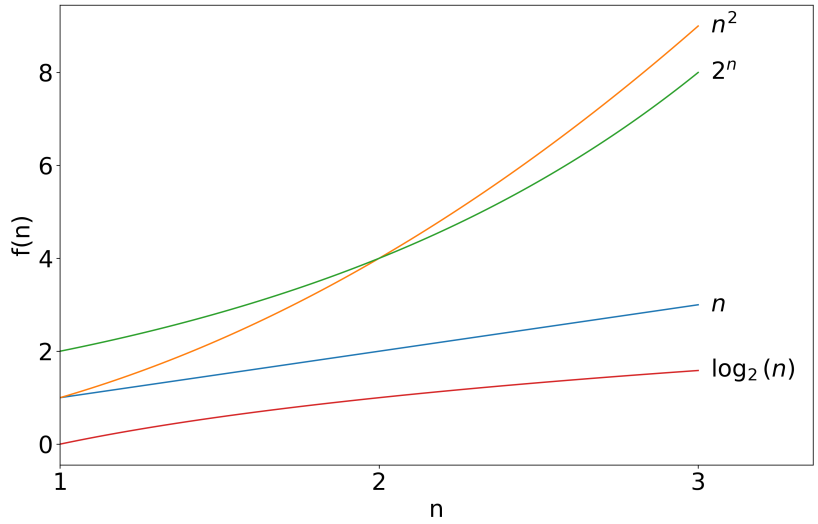
- Recall the formal definition of big-O notation, $f = O(g)$.
- Prove that $f = O(g)$ using a C, N proof.
- Prove that $f = O(g)$ using a limit proof.
- Recall the definitions of big-Omega $\Omega(g)$, big-Theta $\Theta(g)$, little-o $o(g)$, and little-omega $\omega(g)$.
- Sort functions by asymptotic growth rate, from least to highest.
- Given two functions f, g, state which of the following are true or false: $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$, $f = o(g)$, $f = \omega(g)$.

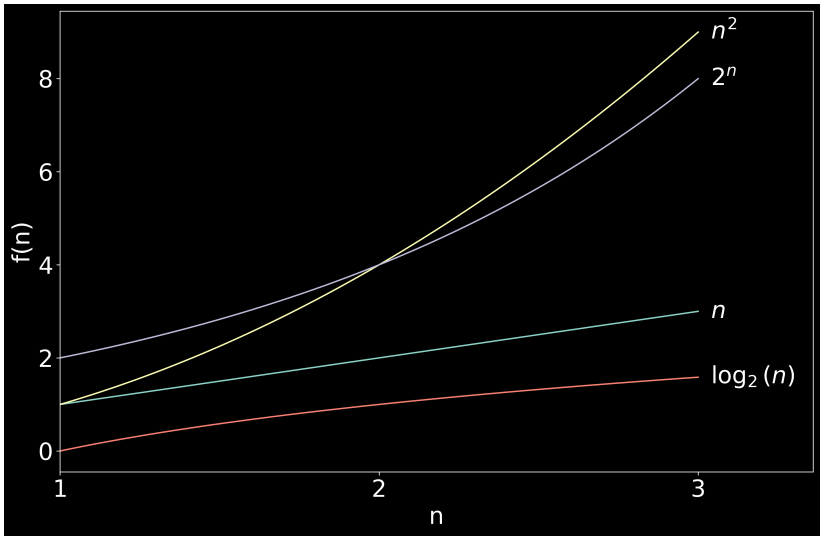
1.1 Motivation and Intuition

We often want to compare algorithms to see which is faster or uses less memory. But that depends on details: the hardware they're running on, the particular inputs, and more. So we'll step back and get a rough sense of an algorithm's performance by bounding its **asymptotic performance** as the inputs get larger and larger. We will discuss this in detail next chapter, but for now, just know that big-O will be our tool to get rough estimates that don't depend on details such as hardware performance.

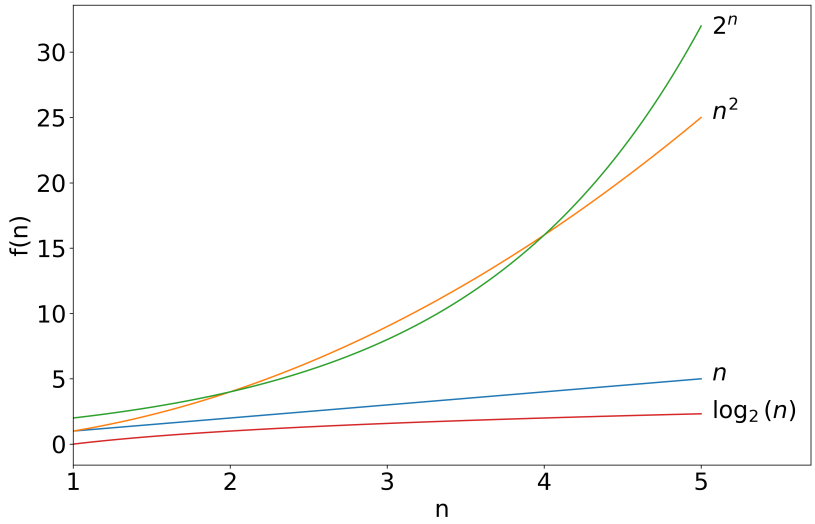
Important: big-O applies to any functions, not just runtime or space usage. We will consider the big-O properties of any function $f : \mathbb{R} \to \mathbb{R}$, although we will often have that $f(n)$ is the runtime of an algorithm on inputs of size n (discussed soon).

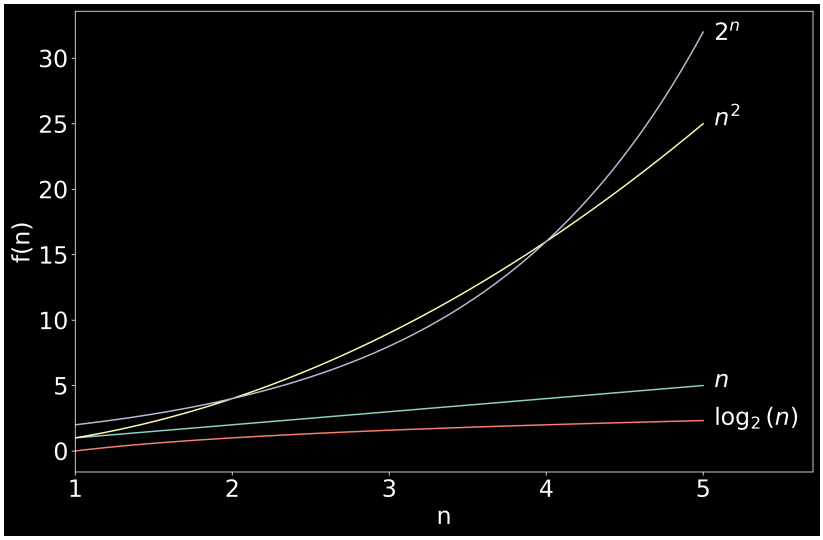
With big-O, we will focus on the "asymptotic growth rate" of functions as we zoom out and the functions grow toward infinity.



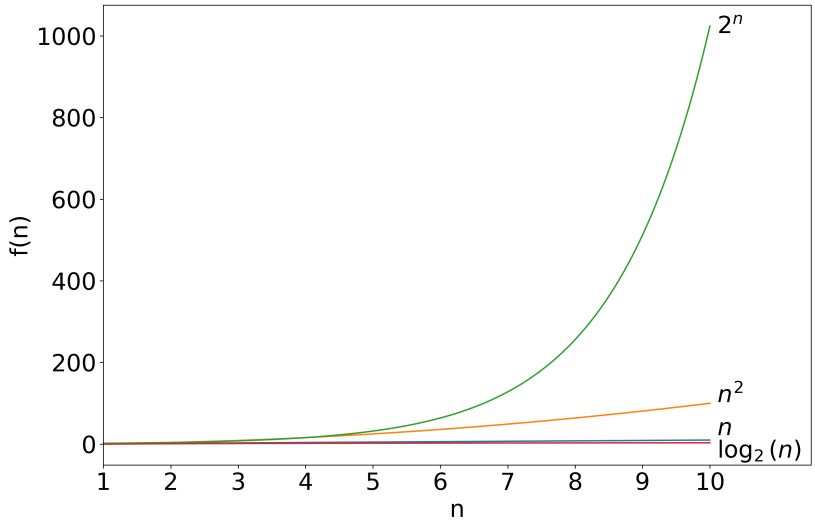


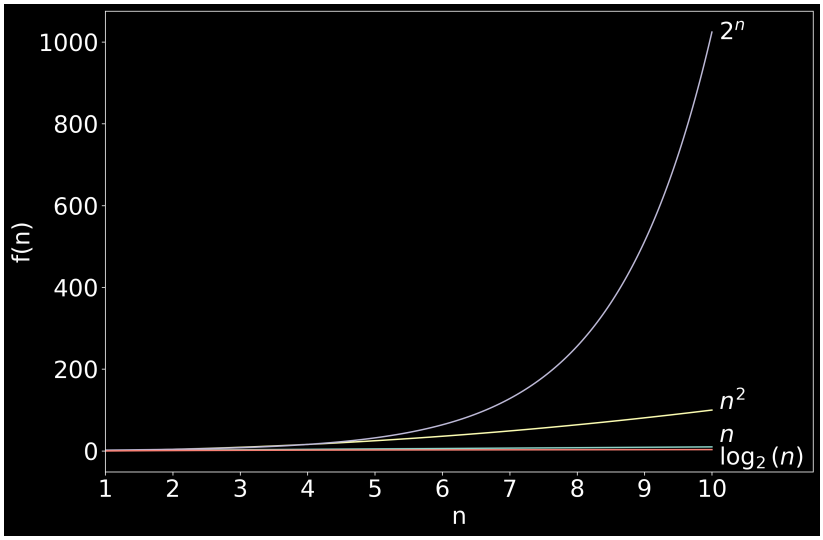
So far, n^2 is the largest function, but let's zoom out.





The picture changes as we continue to the right.





In fact, even if we multiply n^2 by a large constant, like 1000, by zooming out, it becomes clear that 2^n grows faster. This can also be illustrated with a chart.

n	$f(n) = n^2$	$h(n) = 1000n^2$	$g(n) = 2^n$
1	1	1,000	2
10	100	100,000	~1,000
20	400	400,000	~1,000,000
30	900	900,000	~1,000,000,000
40	1,600	1,600,000	~1,000,000,000,000
50	2,500	2,500,000	~1,000,000,000,000,000

Here, $h(n) = 1000n^2$ begins much larger than $g(n) = 2^n$, but as n increases, the latter quickly becomes much, much larger. We will see that this behavior still occurs if the 1000 in $h(n) = 1000n^2$ is replaced with any constant, however large. On the other hand, in a sense, $f(n) = n^2$ and $h(n) = 1000n^2$ grow at the same rate: the gap between them remains the same, multiplicatively speaking. We will formalize this idea with big-O.

1.2 Formalizing Big-O

Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. Recall this means that f and g are functions that take in real numbers and output real numbers. Also assume that $f(n) > 0$ and $g(n) > 0$ for all n, which will be the case for space and runtime and will make our lives simpler.

Definition 1.1 (Big-O)**.** Say that $f = O(g)$ if there exist positive numbers C, N such that, for all $n \geq N$, $f(n) \leq C \cdot g(n)$.

In this definition, N is a lower cutoff. We only consider the behavior of f and g for "large" inputs, i.e. $n \geq N$. To understand the constant factor C, consider the next example.

Example 1.1. Let $f(n) = 5n^2$ and $g(n) = n^2$. Here f is larger than g, but only by a constant factor. They grow at the same rate: $f = O(g)$. We can prove this by showing that the definition of big-O holds with $N = 1$ and $C = 5$.

Proof: for all $n \geq 1$, we have $f(n) \leq 5g(n)$, as required.

Example 1.2. Let $f(n) = n^2$ and $g(n) = n^3$. We will prove that $f = O(g)$ with constants $C = 1, N = 1$. To satisfy the definition, we have to prove that, for all $n \geq 1$, we have $n^2 \leq n^3$.

Proof: We have $n^2 = 1 \cdot n^2 \leq n \cdot n^2 = n^3$, completing the proof.

To understand these proofs, you can plot $f(n)$, $g(n)$, and $C \cdot g(n)$. Visualizing can also be helpful on the examples below.

Example 1.3. Let $f(n) = 10n^2$ and $g(n) = n^3$. We will prove that $f = O(g)$ with constants $C = 10, N = 1$. We have to prove that, for all $n \geq N$, we have $10n^2 \leq Cn^3$. In this case, we must prove for all $n \geq 1$ that $10n^2 \leq 10n^3$.

Proof: If $n \geq 1$, then $10n^2 \leq 10n^2 \cdot n$, so $10n^2 \leq 10n^3$.

Example 1.4. We can give an alternate proof of the previous example using different constants. Let us pick $C = 1, N = 10$. If $n \geq N$, then $n \geq 10$, so $10n^2 \leq n \cdot n^2 = n^3 = Cn^3$. We have proven that $10n^2 \leq 1 \cdot n^3$ for all $n \geq 10$.

As the examples show, there is generally not just one correct choice of N and C that works to prove $f = O(g)$. You can now try on the next example. Remember that you get to pick whichever N and C you want to make the proof work.

Exercise 1.1. Let $f(n) = 20n^2$ and $g(n) = 5n^3$. Prove that $f = O(g)$.

Note on writing functions. We will often refer to an expression such as $1000n^2$ as being a function, namely the function that maps n to $1000n^2$. If $f(n) = 4n$ and $g(n) = 1000n^2$, all of these are valid ways of writing the same thing:

- $f(n) = O(g(n))$
- $f = O(g)$
- $f = O(1000n^2)$
- $4n = O(g)$
- $4n = O(1000n^2)$

1.3 Big-O proofs

To come up with a big-O proof, you often need to do some scratch work and calculation to find N and C. You should first do all the scratch work to figure them out, then write a fresh clean proof using the numbers you have found.

Example 1.5 (Scratch work)**.** Suppose we are asked to prove that $5n^2 + 3 = O(n^2)$. We can first use a common approach for big-O, which is to upper-bound low-order terms by a higher order. In this case, $3 \leq 3n^2$, so $5n^2 + 3 \leq 5n^2 + 3n^2 = 8n^2$. However, here we need to be careful and notice that our inequality $3 \leq 3n^2$ only holds for $n \geq 1$. So to use it, we will need to choose N at least 1. Now, we note that we can choose $C = 8$ and we will be done, because $8n^2 = Cn^2$. We have found that $N = 1, C = 8$ will work for our proof.

Warning: the above scratch work is **not** a proof! Now that we've done the scratch work, we need to turn it into a good proof.

Example 1.6 (Finished proof)**.** We choose $C = 8, N = 1$. For all $n \geq 1$, we have $3 \leq 3n^2$. So in this case,

$$\begin{aligned} 5n^2 + 3 &\leq 5n^2 + 3n^2 && (1.1)\\ &= 8n^2 && (1.2)\\ &= Cn^2. && (1.3)\\ & && (1.4) \end{aligned}$$

This proves $5n^2 + 3 = O(n^2)$.

We call this a "C, N" proof that $f = O(g)$. Next we'll see a different type of proof, a "limit" proof.

Remark 1.1. In Equation 1.4, we had an array of equalities and inequalities. Such an array should be read as follows:

$$5n^2 + 3 \text{ is less than or equal to } 5n^2 + 3n^2 \tag{1.5}$$
$$\text{which is equal to } 8n^2 \tag{1.6}$$
$$\text{which is equal to } Cn^2 \ . \tag{1.7}$$

We showed that the first expression is less than or equal to something that is equal to something that is equal to the final expression. This allows us to conclude that the first expression, $5n^2+3$, is less than or equal to the last expression.

1.4 Calculus proofs

Often, the easiest way to prove a big-O statement is to use the following fact.

Proposition 1.1. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq D$ *for some real number* D*, then* $f = O(g)$*.*

Example 1.7. Let's show that, for $f(n) = 8n^4$ and $g(n) = n^4$, we have $f = O(g)$. We have $\frac{f(n)}{g(n)} = \frac{8n^4}{n^4} = 8$. So $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 8$, so $f = O(g)$.

Using Proposition 1.1 in this way is called a "limit proof" of big-O. Limit proofs often use the following:

Proposition 1.2 (L'Hopital's Rule)**.** *If* $f(n) \to \infty$ *and* $g(n) \to \infty$ *as* $n \to \infty$*, then* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$*, where* f' *and* g' *are the derivatives, assuming the limit exists.*

Example 1.8. Let's use a limit proof to show that, for $f(n) = 3n^2 + 2$ and $g(n) = n^3$, we have $f = O(g)$. We consider $\lim_{n\to\infty} \frac{f(n)}{g(n)}$. Both the numerator and denominator approach infinity as n grows. We have $f'(n) = 6n$ and $g'(n) = 3n^2$. So by L'Hopital's rule, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \frac{6n}{3n^2} = \frac{2}{n}$. Since $\lim_{n\to\infty} \frac{2}{n} = 0$, we conclude that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, which is a constant.

Example 1.9. Let's use a limit proof to prove that $n^2 = O(e^n)$. Both functions approach infinity. We remember that $\frac{d}{dn} n^2 = 2n$ and $\frac{d}{dn} e^n = e^n$. So by L'Hopital's rule, $\lim_{n\to\infty} \frac{n^2}{e^n} = \lim_{n\to\infty} \frac{2n}{e^n}$. Now we use L'Hopital's rule again to get $\lim_{n\to\infty} \frac{2}{e^n} = 0$. Since the limit is zero, we have $n^2 = O(e^n)$.

It's useful to remember that $2^n = e^{a \cdot n}$ for a positive constant a. Therefore, using the chain rule, $\frac{d}{dn}2^n = \frac{d}{dn}e^{a \cdot n} = ae^{a \cdot n} = a2^n$. In other words, the derivative of 2^n behaves almost like the derivative of e^n, except that a positive constant comes out front.

Similarly, we recall that $\frac{d}{dn}\ln(n) = \frac{1}{n}$, and $\log_2(n) = b\ln(n)$ for some positive constant b. This means that $\log_2(n) = O(\ln(n))$ and vice versa.

Let's use a limit proof to prove that $\log_2(n) = O(2^n)$. Both functions approach infinity as $n \to \infty$, so we can use L'Hopital's rule. $\lim_{n\to\infty} \frac{\log_2(n)}{2^n} = \lim_{n\to\infty} \frac{c(1/n)}{2^n}$ for some constant $c > 0$. Continuing, we get $\lim_{n\to\infty} \frac{c}{n2^n} = 0$, because the numerator is a constant and the denominator approaches infinity. Since the limit is zero, $\log_2(n) = O(2^n)$.

1.5 Comparing growth rates

Here are some of the most common types of functions you will encounter.

- **Constant functions:** $f(n) = a$ for some real number a. All constant functions can be described as $O(1)$.
- **Linear functions:** $f(n) = a \cdot n$ for some positive constant a, such as $f(n) = 4n$. We can abuse terminology and use "linear" to refer to functions of the form $a \cdot n + b$ for some positive constant a and some constant b, such as $f(n) = 4n + 2$.
- **Quadratic functions:** $f(n) = a \cdot n^2$ for some positive a.
- **Polynomial functions:** these are functions of the form $f(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ for some constants $a_0, \ldots, a_k$. Examples include $f(n) = 8n^3 - 2n^2 + 4n + 1$ and $f(n) = 5n^{100} + 2n^{33}$. Linear and quadratic functions are examples of polynomials, as are constant functions. In this context, we usually assume a_k is positive so that $f(n) \to \infty$ as $n \to \infty$. We say that the degree of the polynomial is k, where k is the highest power.
- More generally, if $f(n) = n^a$ for some positive constant a, even if a is not an integer, we abuse terminology and say that f has a polynomial growth rate.
- **Logarithmic functions:** these are functions of the form $f(n) = \log_2(n)$, or more generally $f(n) = a \cdot \log_2(n)$ for some positive constant a. Remember your rules of logarithms! $a\log_2(n) = \log_2(n^a)$.
- **Polylogarithmic functions:** these are less common, but they are functions of the form $\big(\log_2(n)\big)^k$ for some positive constant k, e.g. $(\log_2(n))^2$. Logarithmic functions are polylogarithmic, corresponding to the case $k = 1$.
- **Exponential functions:** these are functions of the form $2^{a \cdot n}$ for positive constant $a > 0$. Sometimes, other functions with n in the exponent, such as 2^{n^2}, are also referred to as being exponential in n.

We always have the following rules of comparison with regard to big-O:

$$\text{constant} \ll \text{polylogarithmic} \ll \text{polynomial} \ll \text{exponential}$$

where $\ll$ is shorthand for "is big-O of". Furthermore, we can say the following:

> If $f(n)$ is a polynomial of degree k, then $f = O(n^k)$.

And:

> If $k' \geq k$, then $n^k = O(n^{k'})$.

Another useful rule of thumb is that if $a, b > 1$, then $\log_a(n) = O(\log_b(n))$ and vice versa, where a and b are the bases of the logarithms. Because of this, we can be a bit informal and not always specify the bases of our logarithms. In this class, $\log(n)$ will generally mean log base 2 and $\ln(n)$ will always mean log base e. All of these facts can be proven using the C, N definition of big-O (or using calculus, for example).

Exercise 1.2. Suppose we are asked to compare $f(n) = 10\log(n)$ and $g(n) = 10^n$. Do we have $f = O(g)$, $g = O(f)$, both, or neither?

i Solution.

We have $f = O(g)$, but not the other way around. Logarithmic functions grow much slower than exponential functions.

Exercise 1.3. Compare $f(n) = 8n^3 + 2n^2$, $g(n) = 0.1n^4$, and $h(n) = (\log(n))^{300}$. Order them by asymptotic growth rate from slowest to fastest (smallest to largest).

i Solution.

h, f, g. Any polylogarithmic (that is $\log(n)$ raised to a power) grows slower than any polynomial (that is, n raised to a power). So h is the slowest growing. f grows asymptotically at the speed of n^3, which is slower than the n^4 of g.

Other functions. There are many functions that do not fall directly into the categories listed above. A common example in computer science is the function $f(n) = n\log(n)$. Comparing the growth rate of such functions may take some work.

Example 1.10. We prove that $n\log(n) = O(n^2)$ using a C, N proof.

In this class, you may take as given the fact that $\log(n) \leq n$ for all $n \geq 0$, when the log is base 2 or e. This can be proved directly with calculus.

Using this fact, $n\log(n) \leq n \cdot n = n^2$ for all $n \geq 0$. We conclude that, with $N = 0$ and $C = 1$, the definition of big-O is satisfied.

1.6 Other Asymptotic Notation

There are several other pieces of asymptotic notation to remember. We will often use these pieces of notation, although big-O is the most common.

Definition 1.2 (Big Omega)**.** We say $f = \Omega(g)$ if $g = O(f)$.

Here, Ω is the Greek letter capital Omega. Big Omega says that f grows asymptotically at least as fast as g. For example, $2^n = \Omega(n)$.

Definition 1.3 (Big Theta)**.** We say $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

Here, Θ is the Greek letter capital Theta. Big Theta says that f and g grow asymptotically at the same rate. For example, any two polynomials of the same degree are big-Theta of each other. E.g., take $f(n) = 3n^2 + 200$ and $g(n) = 45n^2 - 30$, then $f = \Theta(g)$. We can prove this by first proving $f = O(g)$, then proving $g = O(f)$.

Definition 1.4 (Little o)**.** We say $f = o(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

In other words, f does not grow as fast as g. For example, $n = o(n^2)$. In fact, if $k' > k$, the $n^k = o(n^{k'})$. This follows directly from the definition of little o.

Definition 1.5 (Little omega)**.** We say $f = \omega(g)$ if $g = o(f)$.

Here, ω is the Greek letter lowercase omega. Little omega says that f grows "strictly" faster than g. Little-omega mirrors little-o in the same way that big-Omega mirrors big-O.

Here is a quick guide, but remember that this comparison to inequality operators is only an analogy.

Symbol	**similar to**	**Meaning (up to a constant factor)**
$O(n)$	$\leq$	asymptotically at most
$o(n)$	$<$	asymptotically less; shrinking compared to
$\Omega(n)$	$\geq$	asymptotically at least
$\omega(n)$	$>$	asymptotically more; diverging compared to
$\Theta(n)$	$=$	asymptotically the same

Based on our previous discussions, we know the following general facts.

- If f is polylogarithmic and g is polynomial, then $f = o(g)$.
- If f is exponential and g is polynomial, then $f = \omega(g)$.
- If f is bounded by a constant and $g(n) \to \infty$, then $f = o(g)$.

You should now be able to answer the following questions.

- Let $f(n) = 2n^2$ and $g(n) = \log(n)$. Which of the following are true and which are false? $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$, $f = o(g)$, $f = \omega(g)$.
- Let $f(n) = 8^n$ and $g(n) = n\log(n)$. Which of the following are true and which are false? $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$, $f = o(g)$, $f = \omega(g)$.

One very useful fact is that if $f = O(g)$ and $h = O(g)$ also, then $f + h = O(g)$. We can interpret this as a fact about running times of algorithms: if we have two algorithms with running times f and h respectively, and both of them are $O(g)$, the running first one algorithm and then the other will give an asymptotic running time of $f + h = O(g)$ as well.

2 Analyzing Algorithms

This section introduces the analysis of time and space complexity of algorithms.

Objectives. After learning this material, you should be able to:

- Relate actual code running on hardware to our model of pseudocode used for time and space analysis.
- Use big-O notation to bound the time and space usage of an iterative algorithm.
- Analyze time and space usage of algorithms with nested for loops.

2.1 Background

Imagine we want to sort a list of numbers. We have two algorithms. Which is faster? The answer depends on the details, for example:

- What programming language is each written in?
- What hardware is each running on? What is the speed of the processor, the size of the cache, the memory available, and so on?
- What does the problem instance look like? For example, are the numbers already close to being sorted, or in a random order, or arranged in some pattern?

Our goal is a systematic study of algorithm performance that strikes a balance between two goals.

1. Our analysis should be relevant to practice, e.g. deciding which algorithm to use.
2. Our analysis should be generalizable: it shouldn't only hold for one particular programming language or hardware architecture.

We will introduce a method of analysis that strikes a balance. We will analyze pseudocode that can be translated into most programming languages. The pseudocode's commands will roughly correspond to steps that are taken on any modern CPU. Therefore, the time analysis of the psuedocode will be pretty closely related to the actual running time of an implementation of an algorithm.

2.2 Correctness of Algorithms

First, a quick note on correctness. There are two types of algorithms: correct, and incorrect. But what does it mean for an algorithm to be correct?

We will formalize the problem the algorithm is trying to solve as a mathematically well-defined function from inputs to outputs. The algorithm is **correct** if, for every possible input, it produces the correct output. We will often prove correctness of our algorithms, and to do so, we must always prove that they *always* produce the correct outputs.

2.3 Analysis of Pseudocode

Throughout the class, we will focus on algorithms written in **pseudocode**. Although the code will mostly look like python, it is not supposed to be a fully specified programming language, just a step-by-step description.

We will define how much time and space each operation of our pseudocode takes in theory. This will allow us to analyze the running time and space usage.

Algorithm 2.1 (Sum a list)**.**

```
sum(A):
    let n = len(A)
    let s = 0
    for i = 1 to n:
        s += A[i]
    return s
```

We won't worry about specific syntax, such as "let s = 0" versus some other way of initializing a variable, as long as the meaning is clear.

2.3.1 Pseudocode operations

In our pseudocode, we will generally have the following rules and operations. We will assume they take the given number of units of time, called *steps*, and the given amount of space.

- **Variables** can be integers, floating-point numbers, booleans, or characters in a string. They can also be objects, such as arrays.
- **Declaring a variable** or an array. This operation takes 1 step. Each variable takes 1 space. An array of n variables takes n space, but the array still only takes one time step to declare. We also assume that appending to an array and expanding it by one spot takes constant time.

- **Assigning a new value** to a variable. This takes one step and no additional space.
- **Reading the value** in a variable. This takes one step and no additional space.
- **Arithmetic operations** such as $+, -, *, /$. These take one step and use no space on their own. Other operations including modulo, exponentiation, and XOR are also generally allowed.
- **If, then, else statements.** These each take one step and use no additional space on their own.
- **For and while loops.** These take one step at each control flow location, i.e. the start and end of the loop. (We can think of them as combining an if/then/else statement with goto statements that take one step.) They use no additional space on their own.
- **Function calls.** Calling a function takes one step and uses one space in addition to the usage of the function itself. We will not generally worry about the space usage, but it is technically important for recursive algorithms.

Example 2.1. Let us analyze the time and space usage of alg. 2.1. First, let's look at time, or the number of steps taken to run the algorithm.

```
sum(A):
    let n = len(A)    # 2 steps: get length of A, store it
    let s = 0         # 1 step
    for i = 1 to n:   # 1 step each loop
       s += A[i]      # 5 steps each loop: read s,i,A[i]; add; store in s
    return s          # 1 step
```

Lines 2 and 3 take three steps total.

Lines 4 and 5 have a `for` loop. It executes n times. Each time, how many steps does it take?

- During each loop, we do one step in line 4. First, we initialize `i`, and then, each loop, we increment `i`. (This may take more than one step in reality, but we will summarize it as one step – discussed below.)
- During each loop, we do 5 steps in line 5.
- This is a total 6 steps per loop.

Since the loop executes n times, the total number of steps taken by lines 3 and 4 is $6n$ steps. Adding the time for lines 2, 3, and also line 6, we get a running time of $6n + 4$.

Now let's look at space usage.

```
sum(A):
    let n = len(A)   # 1 space
    let s = 0        # 1 space
    for i = 1 to n:  # 1 space for i (total)
```

```
        s += A[i]      # no new space
    return s           # no new space
```

Summing up the space used gives a total of 3 space used by the algorithm. We will generally not count the input as part of the space used, although usually, it won't matter because most algorithms use more space than the input.

Note on pseudocode conventions. Pseudocode isn't precisely defined to follow a highly specific format; that's part of the point. We may use slightly different notation and conventions, and you may as well. For example, in Algorithm 1, the array is 1-indexed, meaning that that first element appears at A[1], not A[0]. You may use zero-indexing if you like.

Note on realism. Our estimates here are very rough compared to reality. For example, computers can carry out additions extremely quickly compared to accessing memory storage, but here we treat both as taking the same amount of time. However, this rough approximation still turns out to usefully capture the differences in performance between algorithms much of the time.

Another simplification is that we are not considering parallelism (ability to execute multiple commands at the same time). This model can be extended to analyze parallel algorithms, but we won't do that in this class.

Note on exact runtime. Even in an algorithm as simple as Algorithm 1, it can be a bit unclear exactly how much time and space is used down to the exact number. For example, the for loop includes an implicit branch back to the start, which we did not include in our analysis. Luckily, this won't matter, because we are going to use big-O analysis, which will come out to the same answer regardless of these minor details. That is one of the main points of using big-O. We discuss this next.

2.4 Big-O analysis of pseudocode

Given that so many details are implementation-dependent, the running time and space usage we calculate is only approximate. For example, a slightly different analysis of Algorithm 1 could give a running time of $7n + 2$ instead of $6n + 2$.

Instead of worrying too much about this, we will "lean in" and use big-O to express the time and space usage of our algorithms. Both $7n + 2$ and $6n + 2$ are $O(n)$, so we say the time complexity of Algorithm 1 is $O(n)$. Similarly, the space usage is $3 = O(1)$, meaning a constant space independent of n (remember that we are not counting the input array in the space usage).

Therefore, instead of worrying about exact constants such as $6n$ versus $7n$, we will worry about the asymptotic growth rate, such as linear time ($O(n)$) versus quadratic time ($O(n^2)$). So our goal is to produce a big-O summary of the time and space usage of our algorithms.

2.4.1 Conducting a big-O analysis

In order to correctly analyze algorithms, this book suggests the following approach:

1. Analyze the algorithm exactly, as in Example 2.1, to produce an expression such as $6n+2$ for the time or space usage.
2. Use big-O to give a "simplified" big-O expression, such as $6n + 2 = O(n)$.

Simplified expressions should look like simple functions such as $O(n)$, $O(n \log(n))$, $O(n^2)$, etc. They should follow these guidelines:

- **Drop leading constants.** For example, instead of $O(5n^3)$, write $O(n^3)$, which has the same meaning but is simpler.
- **Drop low-order terms** from a summation. For example, instead of $O(n^3 + 2n^2)$, write the simpler and equivalent $O(n^3)$.
- **Simplify** inside expressions where possible. For example, instead of $O(\log(8n^3 + 5))$, write $O(\log(n))$. *(Can you prove that* $\log(8n^3 + 5) = O(\log(n))$*?)*
- However, avoid using an asymptotically loose upper bound. It is true that $\log(8n^3 + 5) = O(n^2)$, but $O(n^2)$ is too loose of a bound. The best answer here is $O(\log(n))$.

Simplified analysis of operations. With big-O analysis, the exact number of operations per line doesn't change the answer if it is a constant. Therefore, we can simplify our big-O analysis by pretending that every line of code that takes some constant number of steps, such as 2 or 4 or 5, only takes 1 step. This makes our arithmetic easier.

Example 2.2. We simplify the time complexity analysis of Example 2.1.

```
sum(A):
    let n = len(A)     # 1 step
    let s = 0          # 1 step
    for i = 1 to n:    # 1 step each loop
       s += A[i]       # 1 step each loop
    return s           # 1 step
```

The loop executes n times with 2 total steps per loop for a total of $2n$. So the total is $2n + 3 = O(n)$ runtime.

While it's not really true that `s += A[i]` takes one time step, it wasn't really true that it takes 5 time steps either. The true answer depends on the machine, the context it's running in, and other factors. This simplification still gives the right big-O time complexity.

However, **be careful**: not every line of code is 1 step. For example, an algorithm may include the line `let s = sum(A)`. The sum takes $O(n)$ time to calculate, not $O(1)$.

2.4.2 Worst-case analysis

Algorithm 1 used the same amount of time and space regardless of the input. This is not always the case. Consider the following algorithm.

Algorithm 2.2 (Find the first nonzero element)**.**

```
find_nonzero(A):
   n = len(A)
   for i = 1 to n:
      if A[i] != 0:
         return A[i]
   return "none"
```

Example 2.3. What is the time complexity of alg. 2.2?

```
find_nonzero(A):
   n = len(A)          # 1 step
   for i = 1 to n:     # 1 step per loop
      if A[i] != 0:    # 1 step per loop
         return A[i]   # 1 step if and when it executes
   return "none"       # 1 step if and when it executes
```

The total time usage is $2k + 2$, where k is the number of loops that execute. But how many is that? It depends on the input array. There is no single correct answer.

To address this, we will typically use a **worst-case analysis** of the time and space usage of our algorithms. Here, the question is to bound the running time in the worst-case (i.e. slowest) over all possible inputs. Because the worst case is that the loop executes n times, the worst-case running time of the algorithm is $2n + 2 = O(n)$.

Unless otherwise specified, in this course, we will always be using a worst-case analysis for both time and space.

Note on input size and parameters. If we say an algorithm has running time of e.g. $O(n)$, we must always ask carefully: what is n? In the above context, the input was a list of n

numbers. So the time of the algorithm grows linearly in the size of the input. Sometimes, our input will be described by multiple parameters. For example, we may have a matrix with n rows and m columns. Our running time bound may be a function of both n and m in this case.

In computational complexity theory, we often represent the input as a string of bits that encodes the problem, and we care about running time in terms of the number of bits. That will generally not be the case for this class, though.

2.5 Examples: Nested Loops

We'll now look at some more complex algorithms involving nested loops.

Example 2.4 (Counting equal pairs)**.** In this problem, we are given an array A. We must count how many pairs of indices (i,j) there are such that `A[i] == A[j]`.

Algorithm 2.3.

```
count_equal_pairs(A):
   let n = len(A)
   let s = 0
   for i = 1 to n:
      for j = 1 to n:
         if A[i] == A[j]:
            s += 1
   return s
```

Exercise 2.1. Bound the asymptotic time complexity of `count_equal_pairs`. Show your work.

> **i** Example solution.
>
> First, we'll write the number of steps each instruction takes, treating any constant number as one step.

```
count_equal_pairs(A):
   let n = len(A)          # 1 step
   let s = 0               # 1 step
   for i = 1 to n:         # 1 step each time executed
      for j = 1 to n:      # 1 step each time executed
         if A[i] == A[j]:  # 1 step each time executed
            s += 1         # 1 step each time executed
   return s                # 1 step
```

To count how many times each instruction is executed, we first look at the inner for loop, which covers lines 5-7. In the worst case, line 7 is executed every time, so the loop takes 3 steps every iteration. There are n iterations of the loop.
So every time the loop on lines 5-7 runs, it takes at most 3n steps.

```
count_equal_pairs(A):
   let n = len(A)           # 1 step
   let s = 0                # 1 step
   for i = 1 to n:          # 1 step each time executed
      #
      # at most 3n steps
      #
   return s                 # 1 step
```

The for loop starting in line 4 also runs n times, and we use $3n + 1$ operations each time. This gives a total of $n(3n + 1) = 3n^2 + n$ operations coming from that for loop:

```
count_equal_pairs(A):
   let n = len(A)            # 1 step
   let s = 0                 # 1 step
   #
   # at most 3n^2 + n steps
   #
   #
   return s                  # 1 step
```

The total number of steps is $3n^2 + n + 3 = O(n^2)$ steps, so the final answer is $O(n^2)$.

2.5.1 Counting duplicate pairs (without double-counting)

Consider the following exercise:

Exercise 2.2. If we run alg. 2.3 on input A = [3, 5, 8, 5], what will the output be?

> **i** Solution.
>
> It will be 6. The algorithm will find an equal pair every time i==j, i.e. at index 1, 2, 3, and 4. Then, it will find an equal pair when i==2 and j==4, i.e. A[2] == A[4] == 5. Then, it will also find an equal pair when i==4 and j==2. This adds up to 6.

Therefore, counting the number of "equal pairs" is not quite the same as counting the number of **duplicates** in the array without double-counting. For the example above, there is one pair of duplicates: indices `i==2` and `j==4`, which both contain the entry 5. To avoid double-counting, we can modify our algorithm as follows.

Algorithm 2.4.

```
count_duplicates(A):
   let n = len(A)
   let s = 0
   for i = 1 to n-1:
      for j = i+1 to n:
         if A[i] == A[j]:
            s += 1
   return s
```

The difference from Algorithm 3 is that now, the bounds of the for loops are different. Algorithm 3 looped over all pairs (i,j), e.g. (1,1), (1,2), (1,3), (1,4), (2,1), (2,2), (2,3), …. But here, Algorithm 4 only loops over pairs where i < j. This ensures that each pair of indices only gets considered once. For example, (1,2), (1,3), (1,4), (2,3), (2,4),….

Exercise 2.3. Bound the asymptotic runtime of alg. 2.4. Show your work.

> **i** Example solution.
>
> Again, we start by noting that each line, on its own, is a constant-time operation.

```
count_duplicates(A):
   let n = len(A)          # 1 step
   let s = 0               # 1 step
   for i = 1 to n-1:       # 1 step each time
      for j = i+1 to n:    # 1 step each time
         if A[i] == A[j]: # 1 step each time
            s += 1        # 1 step each time
   return s                # 1 step
```

Again, we see that the inner loop, lines 5-7, uses 3 steps each time it executes. But how many times does the inner loop execute? It depends on i.

- When i=1, it executes n-1 times (from j=2 up to j=n).
- When i=2, it executes n-2 times (from j=3 up to j=n).
- …
- When i=n-1, it executes 1 time (from j=n to j=n).

In each case, we can say that it executes exactly $n - i$ times, so the total number of steps for lines 5-7 is $3(n - i)$.

Remark 2.1 (Not part of solution.). It never hurts to do an example. Let's suppose n=4. Then when i=1, we take 3(4-1) = 9 steps. When i=2, we take 3(4-2) = 6 steps. When i=3, we take 3(4-3) = 3 steps. The total is 9 + 6 + 3 = 18.

With three steps each inner loop and $n - i$ inner loops, we get:

```
count_duplicates(A):
   let n = len(A)            # 1 step
   let s = 0                 # 1 step
   for i = 1 to n-1:         # 1 step each time
      # take 3(n-i) steps
   return s                  # 1 step
```

Now, we calculate the total time of the outer for loop, which starts on line 4. We need to add up the time taken from each iteration.

- i=1: $3(n - 1)$ steps
- i=2: $3(n - 2)$ steps
- …
- i=n-1: 3 steps

To add up the total number of steps, the expression is:

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

We can reverse the order of the sum to make it simpler:

$$3(1) + 3(2) + \cdots + 3(n-1) = 3\sum_{i=1}^{n-1} i.$$

We recall from Discrete Math that the sum of 1 up to $n-1$ equals $n(n-1)/2$. Therefore, the total number of steps for the loop is

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

We have reached this stage:

```
count_duplicates(A):
   let n = len(A)            # 1 step
   let s = 0                 # 1 step
   # take (3/2)n^2 - (3/2)n steps
   return s                  # 1 step
```

The total number of steps is therefore $\frac{3}{2}n^2 - \frac{3}{2}n + 3 = O(n^2)$ steps.

Comparison with Equal Pairs. Algorithm 4 is certainly faster than Algorithm 3, since its bounds on the for loops are smaller – sometimes, much smaller. However, we ended up with the same asymptotic bound on the runtime, namely quadratic: $O(n^2)$. In fact, Algorithm 4 takes around half the number of steps of Algorithm 3, but the "half" is a constant factor that does not ultimately affect the big-O runtime bound.

2.5.2 An example with doubling

Now we will consider an example with a "while" loop.

Example 2.5. What is the asymptotic runtime of the following algorithm? Show your work. For simplicity, you may suppose that n is a power of two, i.e. $n = 2^m$ for some integer m.

Algorithm 2.5.

```
celebrate(n):
   k = 1
   while k <= n:
```

```
    for i = 1 to k:
        print("hip hip")
    print("hooray")
    k *= 2
```

i Example solution.

We first consider the time taken in each line:

```
celebrate(n):
    k = 1                    # 1
    while k <= n:            # 1 each outer loop
        for i = 1 to k:      # 1 each inner loop
            print("hip hip") # 1 each inner loop
        print("hooray")      # 1 each outer loop
        k *= 2               # 1 each outer loop
```

The inner loop executes k times and takes 2 steps each time, for a total of $2k$.
The outer loop therefore takes $2k + 3$ time steps in each loop as a function of k:

```
celebrate(n):
    k = 1                # 1
    while k <= n:        # 1 each outer loop
        # 2k+3 steps
```

Now we need to add up the total time of the outer loops. k starts at 1 and doubles until it reaches n, so the time is bounded by

$$\begin{aligned}&(2(1)+3)+(2(2)+3)+(2(4)+3)+\cdots+(2(n)+3)\\&=2\left[1+2+4+\cdots+n\right]+\left[3+3+\cdots+3\right].\end{aligned}$$

We know that k can double $\log(n)$ times before it reaches n, where the logarithm is base 2. So we have $\log(n)+1$ copies of "3". We also know from Discrete Math that $1+2+4+\cdots+n \leq 2n$. For example, if $n = 16$, then $1+2+4+8+16 = 31 \leq 32$.
So our total runtime is at most $2(2n)+3(\log(n)+1) = 4n+3\log(n)+3$. Taking big-O, we get a time complexity of $O(n)$.

Part II

Topic B: Divide and Conquer

3 Proof by Induction

This section recalls numerical induction and introduces structural induction. These techniques will be crucial to design and analysis of algorithms throughout the course, particularly with Divide and Conquer, Graph Traversal, and Dynamic Programming.

Objectives. After learning this material, you should be able to:

- Prove numerical equalities and inequalities using induction and strong induction.
- Prove facts about structures such as strings and trees using structural induction.
- Relate induction to the concept of recursive algorithms.

3.1 Numerical Induction

In past courses, you have seen numerical induction. This is a method to prove a statement of the form "for all n, $P(n)$ is true." Here is an example.

Recall that the Fibonacci numbers are defined as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. The first few are 0,1,1,2,3,5,8,13,...

Exercise 3.1. Prove by induction that $F_n \leq 2^n$.

Example solution.

The first base case is $n = 0$. In this case, $F_0 = 0 \leq 1 = 2^0$. The second base case is $n = 1$. In this case, $F_1 = 1 \leq 2 = 2^1$.
For the inductive step, let $n \geq 2$. The inductive hypothesis (IH) is that for all $k < n$, we have $F_k \leq 2^k$. Using the IH:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} && \text{definition} \\ &\leq 2^{n-1} + F_{n-2} && \text{by IH} \\ &\leq 2^{n-1} + 2^{n-2} && \text{by IH} \\ &\leq 2^{n-1} + 2^{n-1} \\ &= 2\left(2^{n-1}\right) \\ &= 2^n. \end{aligned}$$

We have proven that $F_n \leq 2^n$. This completes the proof by induction.

Note: strong induction. The previous example is actually called "strong" induction. This is because, in order to prove the statement for n, we needed to assume it was true for all $k < n$. In contrast, weak induction only uses the inductive hypothesis that the statement is true for $k = n - 1$. However, the distinction between strong and weak induction will not be very important for this class; they are both proofs by induction.

Checking your familiarity. In order to tackle the next portion of the class, you should be comfortable with the following type of example. Can you solve it?

Exercise 3.2. Prove by induction that the sum of the first n odd numbers is equal to n^2.

3.2 "Non-numerical" induction

You probably also have used induction to prove facts that are not just numerical equalities or inequalities. Here is an example. Recall that a *prime factorization* of an integer is a list of prime numbers whose product equals the integer.

Exercise 3.3. Prove by induction that every integer $n \geq 2$ has a prime factorization.

Example solution.

The base case is $n = 2$. This has the prime factorization 2.
For the inductive step, assume that every $k < n$ has a prime factorization. We must prove this holds for n as well. There are two cases: n is prime, or n is composite. If n is prime, then it has the prime factorization n, so we are done. If n is compsite, then $n = ab$ for $a, b \in \{2, \ldots, n-1\}$. By inductive hypothesis, a and b each have a prime factorization. Multiplying them together gives a prime factorization of n.

3.3 Structural Induction

We will often deal with "structures" such as strings, graphs, and trees. Many of our algorithms' correctness and runtime guarantees will be proven by a type of induction. We will begin with an example.

Definition 3.1 (String)**.** Given a finite set X, called the *alphabet*, whose elements are called *characters*, a **string** is either:

- the empty string " "; or
- a character followed by a string.

For example, we can take the alphabet to the be 26 lowercase English letters, in which case a string would be all lists consisting of these letters, including the empty string.

The **length** $len(s)$ of a string s is defined inductively: if s is empty, its length is zero. If s consists of c followed by a string s', its length is $1 + len(s')$.

Proposition 3.1. *Let $m = |X|$, the size of the alphabet. Then there are m^t different strings of length t.*

Proof by induction. The base case is when $t = 0$, i.e. the string has length zero. In this case, it is the empty string, and there is only one empty string, so the number of strings is $1 = m^0$, so the formula is correct.

For the inductive case, fix $t \geq 1$ and suppose the claim is true for all $k < t$. A string of length t is of the form cs', where c is a character and s' is a string of length $t - 1$. By inductive hypothesis, there are m^{t-1} possible strings s'. For each s', we get a different string of length t by prepending each of the m possible characters. So there are $m \cdot m^{t-1} = m^t$ possible strings of length t. □

3.3.1 Binary trees

Here is another example of structural induction.

Definition 3.2 (Binary tree)**.** A **binary tree** is either:

- a single node with no children, called a *leaf*; or
- a node with at most two children, each of which is a binary tree.

The **root** of a binary tree is the node that is not a child of any other node. The **height** of a binary tree is the length of the longest path from the root to any leaf.

Proposition 3.2. *A binary tree of height h has at most $2^{h+1} - 1$ nodes.*

Proof. The base case is a leaf node, i.e. a tree of height 0. The number of nodes is $1 = 2^{0+1} - 1$, as needed.

Now let $h \geq 1$ and suppose the claim holds for all $k < h$. Consider a binary tree of height h. The root has at most two children. Each child is the root of a binary tree of height at most $h - 1$. So by inductive hypothesis, there are most $2^h - 1$ nodes in each subtree. Adding them together gives an upper bound of $2(2^h - 1) = 2^{h+1} - 2$ nodes. Adding the root node gives an upper bound of $2^{h+1} - 1$ nodes, proving the claim. □

4 Divide and Conquer Algorithms

This section introduces a powerful algorithmic paradigm, divide-and-conquer. This paradigm is closely related to recursion, and we introduce recursive algorithms as well.

Objectives. After learning this material, you should be able to:

- Solve problems by writing recursive divide-and-conquer algorithms.
- Prove correctness of recursive divide-and-conquer algorithms using structural induction.
- Analyze the time complexity of recursive divide-and-conquer algorithms using structural induction.

4.1 Recursion and Divide-and-Conquer

You have probably seen recursive algorithms before, so we will recall them briefly. An algorithm is *recursive* if, in order to compute its output, it calls itself on a modified input. An example of a recursive algorithm is this naive implementation of the Fibonacci numbers:

Algorithm 4.1 (Fibonacci numbers)**.**

```
fib(n):
   if n == 0:
      return 0
   elseif n == 1:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

We will be looking at recursive algorithms that fall into the following algorithmic paradigm.

Definition 4.1 (Divide-and-conquer)**.** A **divide-and-conquer** algorithm splits the input into parts, solves a useful subproblem on each part, then combines the solutions together to produce an answer to the original problem.

Often, divide-and-conquer algorithms are recursive in nature because the subproblems are of the same form as the original problem. This book will focus on recursive examples.

4.2 Binary Search

For our first example, we'll look at a very important algorithm: binary search. In this problem, we are given a sorted list of elements coming from some universe. Given an element, we need to find its location in the list, if present. The elements could be numbers, strings, or other objects. For simplicity, we will assume they are numbers, but the algorithm easily generalizes to other cases.

The Search Problem:

- **Input:** a sorted array A of numbers; a number x.
- **Output:** the index of A where x should be inserted to maintain the sorted order. If x is present, this should be an index where x is located.

The binary search algorithm is as follows. We will 0-index A for this algorithm. We use `A[i:j]` to denote the subarray of A from element i to j inclusive. If i > j, such as `A[0:-1]` or `A[5:4]`, this produces an empty array. We write `A[i:end]` to denote the subarray starting at index i and going to the end of the array. We assume that a subarray can be created, or referenced, in constant time.

Algorithm 4.2 (Binary search)**.**

```
binary_search(A, x):
   if len(A) == 0:
      return 0
   let i = int(len(A)/2)    # int() rounds down
   if x == A[i]:
      return i
   elseif x < A[i]:
      return binary_search(A[0:i-1], x)
   elseif x > A[i]:
      return i + 1 + binary_search(A[i+1:end], x)
```

Remember that with algorithms, we want to analyze correctness and efficiency (time and space). Let's start with correctness.

4.2.1 Correctness

Proposition 4.1. *Binary search correctly solves the binary search problem, i.e. given a valid input, it produces a correct output.*

Proof. We prove correctness by induction on the length of A. The base case is when A has length 0. In this case, the algorithm returns 0, which is the correct index.

For the inductive step, let the length of A be at least 1 and assume the algorithm is correct on all arrays shorter than A. Let $i = int(len(A)/2)$, where int() rounds down. We note that because $len(A) \geq 1$, we have $i < len(A)$, which means that the array $A[0 : i-1]$ and the array $A[i+1 : end]$ are both shorter than A.

There are three cases. First, if $x == A[i]$, then the algorithm correctly returns index i, and we are done.

Second, suppose $x < A[i]$. By inductive hypothesis, our recursive call on line 8 correctly returns the index to insert x in the sublist A[0:i-1]. We claim this index is also correct for inserting x into A. This follows immediately if the return value is in $0, \ldots, i-1$, which implies that x should be inserted somewhere before the end of the sublist. The other case is if the return value is i, meaning an insertion at the end of the sublist, implying $x > A[i-1]$. This is still correct, because we have supposed $x < A[i]$, so i is the correct position.

Third, suppose $x > A[i]$. By inductive hypothesis, our recursive call on line 10 correctly returns the index to insert x in the sublist A[i+1:end]. If this index is some $j > 0$, then x should be inserted somewhere after the beginning of this sublist, which means the same location, shifted by $i+1$, is correct to insert x into A. If the index is 0, this implies that $x \leq A[i+1]$. On the other hand, we have $x > A[i]$, so $i+1$ is correct. □

The exact code of binary search and its correctness proof are notoriously fiddly due to several edge cases. In such situations, it is helpful to keep in mind the famous quote of Donald Knuth: "Beware this code. I have not tested it, only proven it correct."

4.2.2 Runtime analysis

Now, we will analyze efficiency. We will focus on running time (can you bound the space use yourself?).

Proposition 4.2. *Binary search runs in time $O(\log(n))$, where $n = len(A)$.*

Proof. First, we prove by induction that, when we call binary_search(A,x), it makes at most $\log(n) + 1$ recursive calls total, where the logarithm is base 2. The first base case is when n == 0. In this case, we make 0 recursive calls. Since log(0) is not defined, we will define this case as being satisfied. The other base case is when n == 1. In this case, we either make no recursive calls, because $x == A[0]$, or we make one recursive call on a length-0 subarray. The maximum number of recursive calls is therefore $1 = \log(1) + 1$, as claimed.

For the inductive step, let $n \geq 2$. In this case, binary_search(A,x) either makes no recursive calls, or makes a call in line 8, or makes a call in line 10 (but not both). In either case, the

call is for a subarray of length at most n/2. By inductive hypothesis, after that call, at most $\log(n/2) + 1$ more calls are made. This makes the total number at most $1 + \log(n/2) + 1 = 1 + \log(n) - \log(2) + 1 = 1 + \log(n)$, as claimed.

Now we look at each call to binary_search(). By counting the operations, we can see that any call does at most a constant number of steps, about 10, plus any steps done by its own recursive call. Since we proved that we visit the function binary_search() at most $\log(n) + 1$ times total, this gives a total running time bound of $10(\log(n) + 1) = 10\log(n) + 10 = O(\log(n))$. □

4.3 Mergesort

As with binary search, Mergesort is a recursive divide-and-conquer problem. It also applies to lists of elements, and as with binary search above, we will focus on the case where the elements are numbers, but the ideas can easily extend to other types of list elements.

The Sorting Problem:

- **Input:** an array A of numbers.
- **Output:** an array containing the same numbers in sorted order.

The idea of Mergesort is to split the list in half and sort the left and right halves separately. Then, we need to run the merge subroutine to combine the two sorted sublists into our final sorted list.

Algorithm 4.3 (Mergesort)**.**

```
mergesort(A):
   if len(A) <= 1:
      return A
   let i = int(len(A)/2)
   let B = mergesort(A[0:i])
   let C = mergesort(A[i+1:end])
   return merge(B, C)

merge(B, C):
   let D = []
   let i = 0
   let j = 0
   while i < len(B) or j < len(C):
      if j >= len(C) or (i < len(B) and B[i] < C[j]):
         append B[i] to D
         i += 1
      else:
```

```
        append C[j] to D
        j += 1
    return D
```

4.3.1 Correctness analysis

First, we need a lemma that the "merge" subroutine is correct.

Lemma 4.1. *The subroutine merge(B,C), given two sorted lists, outputs a sorted list containing all of the elements in B and C.*

Proof. All elements of B and C eventually get added to D because we only increment i when we add B[i] to D and similarly for j and C[j]. Now we just need to show that D is sorted. To do so, we claim that in the while loop of lines 5-11, each iteration, the smallest remaining element of B[i:end] and C[j:end] is added to D. This follows because B is sorted, so its minimum remaining element is located at B[i], and similarly for C[j], and we add the smaller of these to D. (If we have already added all elements of B, then $i > len(B)$, so we add C[j]; and vice versa.) Because we always append the smallest remaining element to D, D is sorted. □

Proposition 4.3. *Mergesort correctly sorts the input.*

Proof. By induction. For base cases, if len(A) is 0 or 1, mergsort(A) returns A, which is correct.

For the inductive case, let $len(A) \geq 2$. The elements of A are partitioned into B and C in lines 7 and 8. Each of these is shorter than A, so by inductive hypothesis, B and C are correctly sorted by the recursive calls. By Lemma 4.1, merge(B,C) returns a sorted version of A, so mergesort is correct. □

4.3.2 Runtime analysis

We again assume that subarrays can be referenced in constant time.

Lemma 4.2. *merge(B,C) runs in time at most 10(len(B) + len(C)) + 4.*

Proof. We first analyze the while loop, lines 5-11. Inside the loop, we take a constant number of steps, upper-bounded by about 10. How many iterations does the while loop take? Each iteration, we either increase i or j, but not both. We stop incrementing i when it reaches len(B) and we stop incrementing j when it reaches len(C). When both happen, the while loop completes. Therefore, there are len(B) + len(C) iterations of the while loop, so lines 5-11 take a total of at most 10(len(B) + len(C)) time. Adding 4 steps for lines 2-4 and line 12 gives a running time bound of 10(len(B) + len(C)) + 4. □

Proposition 4.4. *mergesort(A) runs in time $O(n \log(n))$, where $n = len(A)$.*

Proof. We will use a similar strategy to binary search: we calculate how much work happens in each call to mergesort(), then we calculate the number of calls. However, we will see that this situation is a bit more complex.

It will be helpful to have an expression for the running time of mergesort on a given input size. **Define T(n)** to be the running time of mergesort on lists of length n.

First, how much work is done in mergesort(A) itself, ignoring recursive calls? The answer is about 6 steps plus the work done in merge(B,C), which gives a total (using the Lemma) of 10 + 10(len(B) + len(C)). We can observe that len(B) + len(C) = len(A), so this runtime is 10 + 10 len(A). To simplify analysis, let's consider the case $len(A) \geq 1$, so we can upper-bound this work by 10 + 10 len(A) $\leq$ 20 len(A).

Now, we need to consider the recursive calls. To simplify the analysis, we will pretend that $len(A) = 2^k$ for some integer k, i.e. the length is a power of 2. That implies that, because we divide the array in half each time, every subarray we create throughout the algorithm has even length until the length reaches 1.

We recursively call mergesort twice, each on a list of length n/2. This gives the following **recurrence relation:**

$$T(n) = 2T(n/2) + 20n$$

Here is how it relates to the code:

```
mergesort(A):                    # T(n) time total
   if len(A) <= 1:
      return A
   let i = int(len(A)/2)
   let B = mergesort(A[0:i])     # T(n/2)
   let C = mergesort(A[i+1:end]) # T(n/2)
   return merge(B, C)            # O(n)
```

Verbally, the running time of mergesort on a list of length n is equal to twice its running time on lists of length $n/2$, plus an extra $20n$ steps needed to merge the lists and complete the rest of the algorithm.

Solving the recurrence. Now the question is: what formula T(n) solves the recurrence? The method we will use is called the **recursion tree** method. We create a tree where each node represents a call to mergesort() and its two children represent the two recursive calls it makes. We can make the following key observations.

- Each child node represents a call on an array of half the length of its parent. Therefore, a node at distance t from the root represents a call to mergesort() with an array of length $len(A)/2^t = 2^{k-t}$.
- Therefore, the height of the tree is k, because after k halvings, we end up with an array of length 1 and make no more recursive calls.
- The recursion tree is a complete binary tree, which implies that it has 2^t nodes at distance exactly t from the root, for each $t = 0, \ldots, k$.

We are now ready to add up the total time complexity by summing over the "layers" of the recursion tree. At each layer $t = 0, \ldots, k$, there are 2^t nodes. Each one does a total amount of work $10 + 10 \text{ len(input)} = 10 + 10 2^{k-t}$. Summing, the total time complexity is:

$$\begin{aligned} T(n) &= \sum_{t=0}^{k} 2^t \left(20 \cdot 2^{k-t}\right) \\ &= 20 \sum_{t=0}^{k} 2^k \\ &= 20k2^k. \end{aligned}$$

Recalling that $n = len(A)$ and $k = \log(n)$, we can rewrite this running time bound as $T(n) = 20n \log(n) = O(n \log(n))$. □

As a note, we can check that $T(n) = 20n \log(n)$ satisfies our original recurrence:

$$\begin{aligned} 2T(n/2) + 20n &= 2\left(20(n/2)\log(n/2)\right) + 20n \\ &= 2\left(10n(\log(n) - 1)\right) + 20n \\ &= 2\left(10n\log(n) - 10n\right) + 20n \\ &= 20n\log(n) - 20n + 20n \\ &= 20n\log(n). \end{aligned}$$

4.4 Integer Multiplication

Let's consider one more problem before we discuss how to analyze divide-and-conquer algorithms in general.

The Integer Multiplication Problem:

- **Input:** two positive n-bit integers x, y.
- **Output:** the product xy.

The "grade-school" algorithm is to multiply each bit of x by each bit of y and take the results, suitably shifted, and add them all up. We won't go into to the analysis, but this takes $O(n^2)$ time, as we must consider all pairs of bits.

Can we multiply faster than that? Famous mathematicians thought no. But in fact, we can, using a divide-and-conquer algorithm. First, we'll look at an approach that is not ultimately faster, but paves the way.

Algorithm 4.4 (Divide-and-Conquer Multiplication)**.**

```
multiply(x, y):          # assume both have n bits
   let n = len(x)
   if n <= 1:
      return x*y
   let x0,x1 = split(x)
   let y0,y1 = split(y)
   a = multiply(x1,y1)
   b = multiply(x1,y0)
   c = multiply(x0,y1)
   d = multiply(x0,y0)
   return a*2^n + (b+c)*2^(n/2) + d

// Divide an integer into upper and lower halves
split(x):
   let x0 = x % 2^(n/2)      // % is the remainder operation
   let x1 = int(x/2^(n/2))   // int() rounds down
   return x0, x1
```

Let's take correctness as given and analyze this algorithm using the recurrence-tree method. Let $T(n)$ be the running time of multiply(). The split() subroutine is $O(n)$ time using bit operations. Line 11 is actually $O(n)$ as well, because adding is $O(n)$ and multiplying by a power of two is just a bit shift, so also $O(n)$. That leaves lines 7-10. Each calls multiply() on inputs of half the original size, so they take $T(n/2)$ time each.

$$T(n) = 4T(n/2) + O(n).$$

Let $n = 2^k$. The tree method gives that at each level $t = 0, 1, \ldots, \log(n)$, there are 4^t nodes and each node does $n/2^t = 2^{k-t}$ work.

$$
\begin{aligned}
T(n) &= \sum_{t=0}^{k} 4^t (2^{k-t}) \\
&= \sum_{t=0}^{k} 2^{2t} 2^{k-t} \\
&= \sum_{t=0}^{k} 2^{t+k} \\
&= 2^k \sum_{t=0}^{k} 2^t \\
&= 2^k (2^{k+1} - 1) \\
&\approx (2^k)^2 \\
&= n^2.
\end{aligned}
$$

A more rigorous proof wouldn't hurt, but we conclude the algorithm is $O(n^2)$. No faster! However, a clever observation allows us to speed it up asymptotically. We'll see this next.

4.4.1 Karatsuba's multiplication algorithm

We know that addition and subtraction of n-bit numbers are $O(n)$ time operations. Asymptotically, these are much, much cheaper than multiplication, which is $O(n^2)$ as far as we know. Therefore, we should be willing to do a few extra additions and subtractions if they can help us save even one multiplication. That is the idea behind the next algorithm.

Remember that $x = 2^n(x_1) + x_0$ and $y = 2^n(y_1) + y_0$, where x_1, x_0, y_1, y_0 are each $n/2$ bits. We want to compute $xy = (2^n x_1 + x_0)(2^n y_1 + y_0) = 2^{2n} x_1 y_1 + 2^n (x_1 y_0 + x_0 y_1) + x_0 y_0$. Let's take for granted that we'll need to compute $x_1 y_1$ and $x_0 y_0$. Can we get the middle term any easier?

The key observation is that we can compute the following product of $n/2$ bit numbers: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_0 y_1 + x_1 y_0 + x_0 y_0$. Subtracting off $x_1 y_1$ and $x_0 y_0$ leaves us with the middle term we need, $x_0 y_1 + y_0 x_1$.

Algorithm 4.5 (Karatsuba Multiplication)**.**

```
multiply(x, y):          # assume both have n bits
   let n = len(x)
   if n <= 1:
      return x*y
   let x0,x1 = split(x)
   let y0,y1 = split(y)
```

```
    a = multiply(x1,y1)
    d = multiply(x0,y0)
    x_sum = x1 + x0
    y_sum = y1 + y0
    product = multiply(x_sum, y_sum)
    b = a + d - product
    return a*2^n + b*2^(n/2) + d
```

We now show that Karatsuba's algorithm runs faster than the $O(n^2)$ algorithms above.

Proposition 4.5. *Karatsuba multiplication runs in time $O(n^c)$ where $c = \log_2(3) \approx 1.58$.*

Proof. We first need to write the recurrence. We have:

```
multiply(x, y):                      # T(n) steps
    let n = len(x)
    if n <= 1:                       # O(1)
        return x*y                   # O(1)
    let x0,x1 = split(x)             # O(n)
    let y0,y1 = split(y)             # O(n)
    a = multiply(x1,y1)              # T(n/2)
    d = multiply(x0,y0)              # T(n/2)
    x_sum = x1 + x0                  # O(n)
    y_sum = y1 + y0                  # O(n)
    product = multiply(x_sum, y_sum) # T(n/2)
    b = a + d - product              # O(n)
    return a*2^n + b*2^(n/2) + d     # O(n)
```

We can write this recurrence as:

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

It turns out that the constants on the $O(n)$ don't matter for the big-O solution to the recurrence. We can analyze it with the tree method. Just as with mergesort, there are $k = \log_2(n)$ levels of the tree, because we cut the input size in half at each level. But now, there are 3^t nodes at level $t = 0, 1, \ldots, \log_2(n)$. In each node at level t, we have an input size of $n/2^t$, so we do at

most $Cn/2^t$ work at that node for some C. The total is:

$$\begin{aligned}
T(n) &= \sum_{t=0}^{k} 3^t C \frac{n}{2^t} \\
&= Cn \sum_{t=0}^{k} \left(\frac{3}{2}\right)^t \\
&= C' n \left(\frac{3}{2}\right)^k && \text{for some } C' \text{ (geometric series)} \\
&= C' 3^k \\
&= C' 2^{k \log_2(3)} \\
&= C' n^{\log_2(3)}.
\end{aligned}$$

We used that the sum of an increasing geometric series, in this case $\frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^k$, is bounded by a constant times its final element. □

There are even faster algorithms than Karatsuba's for integer multiplication. They are also based on the idea of divide and conquer.

5 Failures of Divide and Conquer Algorithms

In this section, we look at cases where D&C fails due to a failure of approach or execution.

Objectives: After learning this material, you should be able to:

- Given an incorrect D&C algorithm, explain why it fails.
- Given an incorrect D&C algorithm, construct an explicit example input where it fails and show how.

5.1 An Incorrect Mergesort

Sometimes, a tempting D&C solution doesn't actually work. Other times, there are variants of correct solutions that have a mistake causing them to fail. By practicing to spot these errors, we can get a deeper understanding of Divide and Conquer as well as practice with an important Algorithms skill: constructing counterexamples.

Consider the following version of mergesort. What is the problem?

Algorithm 5.1 (Try to Mergesort)**.**

```
try_to_mergesort(A):
   if len(A) <= 1:
      return A
   let i = int(len(A)/2)
   let B = try_to_mergesort(A[0:i])
   let C = try_to_mergesort(A[i+1:end])
   return (B,C)     # list B followed by list C
```

The difference is in the last line: instead of using `merge(B,C)`, we simply concatenate B and C together.

If you understood mergesort completely, you might feel that it's obvious that this code will fail. But can you *prove* that it's wrong?

Proving incorrectness. The property of correctness is a "for all" statement: for every input, the algorithm produces a correct output. Therefore, the property of "not correct" is a "there

exists" statement: there exists an example where the algorithm produces an incorrect output. To prove incorrectness, all you need to do is give an single input and show why the algorithm produces an incorrect output.

Proposition 5.1. *Algorithm alg. 5.1 is incorrect.*

Proof. Consider the input A = [8,3]. We claim that the algorithm outputs [8,3], which is not sorted, so it is an incorrect output.

On input A = [8,3], the algorithm first splits [8] and [3] and recursively calls itself on each, resulting in B = [8] and C = [3]. It then concatenates them together, resulting in B followed by C, which is [8,3]. This output is not sorted. □

5.2 Another Incorrect Mergesort

Let's try again. What is the problem here?

Algorithm 5.2 (Try Again to Mergesort)**.**

```
try_again_mergesort(A):
    if len(A) <= 1:
       return A
    let i = int(len(A)/2)
    let B = try_again_mergesort(A[0:i])
    let C = A[i+1:end]
    return merge(B,C)
```

Exercise 5.1. Prove that alg. 5.2 is incorrect.

> **i** Example solution.
>
> *Aside.* The problem is that we only recursively sort B, but not C. That means that C may not be correctly sorted, leading to an incorrect result. But again, to prove it, we have to give a concrete example.
> *Proof.* Consider the input A = [1,2,4,3]. We claim the output will be [1,2,4,3], which is not correctly sorted.
> First, the algorithm divides A into [1,2] and [4,3]. Then, it recursively calls itself on B, which we can check will return B = [1,2]. Then it will merge B = [1,2] and C = [4,3]. Recalling how merge works, it will start from the beginning of each array and take the smaller element, so it will produce [1,2,4,3].

6 Recurrences and the Master Theorem

In this section, we learn a rule of thumb for analyzing the runtime of many Divide and Conquer algorithms, i.e. solving their recurrences.

Objectives. After learning this material, you should be able to:

- Write the recurrence of a divide-and-conquer algorithm in the form $T(n) = mT(n/k) + O(n^c)$.
- Use the master theorem, given, to conclude the overall running time of the algorithm.

6.1 General D&C framework

We will now see how to analyze the time complexity of divide-and-conquer algorithms in general. We can turn the tree method used above into a general rule-of-thumb that makes analysis much easier, as long as we can write the recurrence for a given D&C algorithm. Many divide-and-conquer algorithms are very similar to the algorithms above, so the general framework looks familiar:

Algorithm 6.1 (Divide-and-Conquer Framework)**.**

```
divide_and_conquer(L):
   if is_base_case(L):
      return solve_base_case(L)
   (L1,...,Lm) = divide(L)      # each Li has size L/k for some k
   A1 = divide_and_conquer(L1)
   A2 = divide_and_conquer(L2)
   # ...
   Am = divide_and_conquer(Lm)
   return combine(A1,...,Am)
```

We can see four steps:

- **Base case** stage (lines 2-3): we check if L is a "base case". For mergesort, the base case was a list of length 0 or 1.

- **Divide** L into k smaller objects (line 4). For mergesort, these were two smaller lists (the first and second halves).
- **Conquer** the objects with recursive calls (lines 6-8). For mergesort, this was sorting the two smaller lists.
- **Combine** the results (line 9). For mergesort, this was calling "merge" on the two smaller lists.

To analyze the time complexity, we want to first write a **recurrence**, which expresses the runtime $T(n)$ of the algorithm in terms of itself on smaller inputs.

```
divide_and_conquer(L):           // T(n) time complexity total
   if is_base_case(L):           // O(1)
      return solve_base_case(L)  // O(1)
   (L1,...,Lm) = divide(L)       // complexity depends on the problem
   A1 = divide_and_conquer(L1)   // T(n/k)
   A2 = divide_and_conquer(L2)   // T(n/k)
   // ...                        // ...
   Am = divide_and_conquer(Lm)   // T(n/k)
   return combine(A1,...,At)     // complexity depends on the problem
```

We have m recursive calls, each of which takes $T(n/k)$ time. Then we have some $O(1)$ operations for the base case, and finally the time taken by `divide()` and `combine()`. if these take time $O(n^c)$ for some constant $c \geq 0$, then we get the following general recurrence:

$$T(n) = mT(n/k) + O(n^c).$$

In this case, the following theorem gives the solution to the recurrence.

Theorem 6.1 ("Master theorem")**.** *If an algorithm satisfies the recurrence* $T(n) = mT(n/k) + O(n^c)$*, then its time complexity is:*

- $O(n^c)$ *if* $c > \log_k(m)$*; or*
- $O(n^{\log_k(m)})$ *if* $c < \log_k(m)$*; or*
- $O(n^c \log(n))$ *if* $c = \log_k(m)$*.*

We won't prove it here (see e.g. *Algorithms* by Dasgupta, Papadimitriou, and Vazirani), but the proof follows the tree method.

6.1.1 Putting it together

Here is the general approach to analyzing the time complexity of Divide-and-Conquer algorithms:

1. Write down the recurrence for $T(n)$:

a. First check how many recursive calls are made and what the size of the input is for each recursive call. For example, if there are 5 recursive calls and each is on one-third of the original input, then the total work being done is $5T(n/3)$.
b. Then analyze all non-recursive steps and subroutines, specifically divide() and combine(). Let's suppose the answer is $O(n^c)$ for some constant c. (Note that $c = 0$ is the case of $O(1)$, which is the case with binary search.)

2. Use the master theorem to solve the recurrence. For calculations, remember that $\log_k(m) = \log(m)/\log(k)$.

Example 6.1. For binary search, the recurrence is $T(n) = T(n/2) + O(1)$. Here $m = 1$, $k = 2$, and $c = 0$. We have $\log_k(m) = 0 = c$. So we are in the third case of the master theorem, so the complexity is $O(n^c \log(n)) = O(\log(n))$.

Example 6.2. For mergesort, the recurrence is $T(n) = 2T(n/2) + O(n)$. Here $m = 2$, $k = 2$, and $c = 1$. We have $\log_k(m) = 1 = c$. So we are in the third case again, so the complexity is $O(n^c \log(n)) = O(n \log(n))$.

Example 6.3. For Karatsuba multiplication, the recurrence is $T(n) = 3T(n/2) + O(n)$. Here $m = 3$, $k = 2$, and $c = 1$. We have $\log_k(m) \approx 1.58 > c$. So we are in the second case of the master theorem, so the complexity is $O(n^{\log_k(m)}) = O(n^{\log_2(3)})$.

Part III

Topic C: Graph Traversal

7 Graphs and Trees

This section introduces graphs and graph terminology. Many, many computational problems are modeled as problems to do with graphs, as we will soon see.

Objectives. After learning this material, you should be able to:

- Recall and apply definitions such as graph, neighbors, degree, paths, etc.
- Constrast the performance of the adjacency matrix and the adjacency list representations of a graph.
- Recall and apply definitions for trees, including root, children, height, etc.

7.1 Graphs, formally

As you know, graphs represent a set of objects (the vertices) and relationships between the objects (the edges). Grahps are used to model many settings, such as:

- Road networks
- Computer networks
- Social networks (friendship graphs)
- Web pages and hyperlinks
- Financial networks
- Biological systems
- States of a system and transitions between states

Formally:

Definition 7.1 (Graph)**.** A graph $G = (V, E)$ consists of a set of vertices (also called nodes), V, and a set of edges, E. The graph is *directed* if each edge $e = (u, v)$ represents an ordered pair, where the edge goes from u to v. The graph is *undirected* if each edge $e = \{u, v\}$ represents an unordered pair, where the edge equally connects u to v and v to u.

The number of vertices is $|V|$ and is usually called n. The number of edges is $|E|$ and is usually called m.

For example, in a social network, the vertices are people, and there is an edge (u, v) if person u is "following" person v. We also recall these useful graph concepts.

- A **neighbor** of u in an undirected graph is a vertex v with whom u shares an edge. We sometimes use $N(u)$ to denote the set of neighbors of u.
  - The **degree** of u in an undirected graph is the number of its neighbors.
- An **out-neighbor** of u in a directed graph is a v for which (u, v) is an edge. Similarly, an **in-neighbor** is a v for which (v, u) is an edge.
  - The **out-degree** is the number of out-neighbors; similarly for **in-degree**.
- A **path** in a graph is a list of vertices $v_1, \ldots, v_k$ where there is an edge from each vertex to the next. In a directed graph, the edges must "point" forward. The **length** of the path is the number of edges, i.e. $k - 1$.
- A **cycle** in a graph is a path $v_1, \ldots, v_k$ of length at least 2 where $v_k = v_1$, i.e. we end where we started.
  - Often, when we say cycle, we really mean **simple cycle**, which is a cycle that does not have any repeated edges or vertices. It is always good to double-check whether simple cycle is meant.

Test your understanding of the terminology:

Exercise 7.1. For the following directed graph (Figure 7.1), answer these questions:

1. List all the edges in the graph.
2. What is the out-degree of u?
3. Name all of the in-neighbors of v.
4. Is (u, v, w, x, y) a path in the graph?
5. Is (u, w, y, x, v) a path in the graph?
6. What is a cycle in the graph?

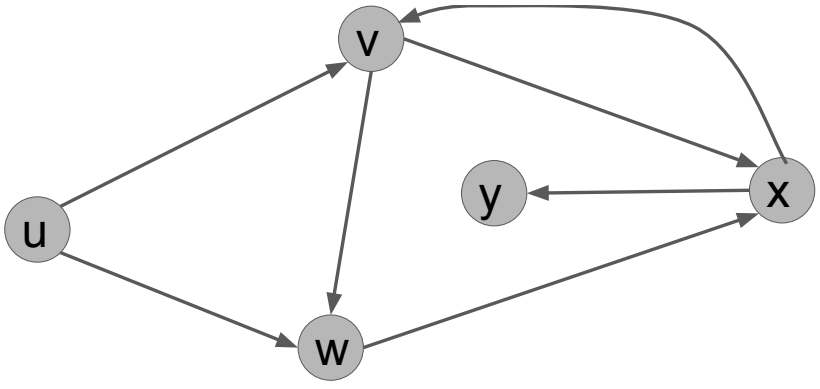


Figure 7.1

Solutions.

1. $(u, v), (u, w), (v, w), (v, x), (w, x), (x, v), (x, y)$.
2. 2.
3. u and x.
4. Yes.
5. No: there is no edge from y to x.
6. There are several answers. One cycle is v, w, x, v. This is a cycle because it is a path (a list of vertices where an edge connects each vertex to the next) and the last vertex in the list equals the first.

Here is a fun fact that will be useful later:

Proposition 7.1. *In an undirected graph, the sum of the degrees of the vertices is equal to twice the number of edges.*

Proof. Every edge has two endpoints, e.g. u and v. If we sum over the vertices the degree of each vertex, then for u, the edge is counted once, and for v, the edge is counted once. Therefore, each edge is counted twice in total. □

7.1.1 Adjacency matrix representation

How do we write a graph down for a computer to work with? There are two main ways. The first is called the *adjacency matrix* representation. For a graph with n vertices, it is an $n \times n$ matrix, where a 1 in the (u, v) entry represents that there is a directed edge from u to v. If the graph is undirected, then the matrix is symmetric, i.e. there is 1 at position (u, v) if and only if there is a 1 at position (v, u).

Example 7.1. For the example graph above (Figure 7.1), here is its adjacency matrix:

	u	v	w	x	y
u	0	1	1	0	0
v	0	0	1	1	0
w	0	0	0	1	0
x	0	1	0	0	1
y	0	0	0	0	0

The adjacency matrix data structure has the following important characteristics:

- **Space:** the size of the data structure is n^2 regardless of the number of edges.

- **Edge query:** checking whether an edge (u, v) exists or not is an $O(1)$ time operation. We can just jump to the appropriate location in the list and check.
- **Neighbors query:** getting the list of out-neighbors of a vertex u is an $O(n)$ time operation, because we need to scan the entire u row of the matrix to check for all the neighbors.

7.1.2 Adjacency list representation

The second main way to write down a graph is as an *adjacency list*. This consists of an array of lists. For each vertex, we give a list of its out-neighbors.

For the example graph above (Figure 7.1), here is its adjacency list:

vertex	out-neighbors
u	v, w
v	w, x
w	x
x	v, y
y	

The adjacency list data structure has the following characteristics:

- **Space:** the size of the data structure is $O(n + m)$, where n is the number of vertices and m is the number of edges.
- **Edge query:** checking whether an edge (u, v) exists is no longer constant time, but involves scanning through the list associated with u for the presence of v.
- **Neighbors query:** getting the list of out-neighbors of a vertex is now $O(1)$ time, because it is already written down for us.

We can see that the adjacency list might be much smaller to store than the adjacency matrix and can be faster if we need to iterate through the neighbors of vertices. However, the adjacency matrix is faster for looking up existence of given edges.

7.1.3 Weighted graphs

A **weighted graph** is a graph where, for each edge (u, v), we are given a weight $w(u, v) \in \mathbb{R}$. The graph can be either directed or undirected.

With adjacency matrices, we can represent the weight directly in the entry of the matrix, replacing 1 with the actual weight. Depending on the application, we may treat a weight of zero or a weight of infinity as "edge does not exist". With adjancency lists, we can simply store the weights in the lists next to the corresponding neighbor.

In a weighted graph, the **length** of a path is now the sum of the weights along the path. Here is an example weighted graph:

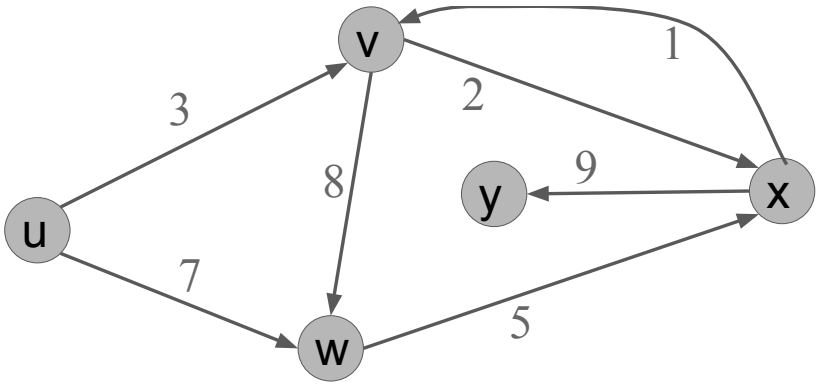


Figure 7.2

In this graph, the path u, v, w, x has length $3 + 8 + 5 = 16$.

7.2 Trees

A tree is a special type of graph. We will define trees inductively.

Definition 7.2 (Tree)**.** A tree is a graph organized as follows. There is a vertex r called the **root**.

- r by itself with no edges is a tree.
- Given a set of trees $T_1, \ldots, T_k$ with roots $v_1, \ldots, v_k$ respectively, the graph produced by connecting r to each of $v_1, \ldots, v_k$ is a tree. In this case, we say r is the *parent* of each v_i, which is a *child* of r.

Technically, this is called a *rooted* tree. If we form a rooted tree, then take away the designation of which node is the root node, we are left with a regular old **tree.** We will come back to those later in the course, but focus on rooted trees for now.

Let us practice induction by proving an important fact.

Theorem 7.1. *A tree with n vertices always has $n - 1$ edges.*

Proof. The base case is a tree with $n = 1$ vertex, which has $0 = n - 1$ edges as required.

For the inductive case, let $n \geq 2$ and suppose all trees with $t < n$ vertices have $t - 1$ edges. A tree on n vertices is formed by a root r connected to a set of trees $T_1, \ldots, T_k$. These have a

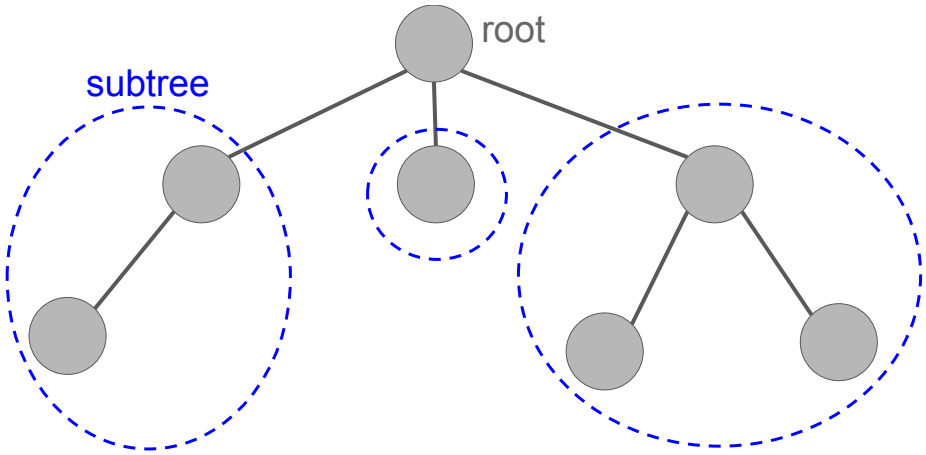


Figure 7.3

total of $n-1$ vertices, since adding the root makes n. Therefore, they have a total of $n-1-k$ edges, because each tree contributes one fewer edges than vertices by inductive hypothesis. Now we need to add the edges between r and the roots of each of the subtrees, which adds k edges. The total is $n-1$. □

We recall some more tree terminology:

- A vertex with no children is called a **leaf**.
- The **height** of a tree is the length of the longest path from the root to a leaf.
- An **ancestor** of a vertex is a parent, parent of a parent, etc. all the way back up to the root.
- A **descendent** of a vertex is a child, child of a child, etc.

8 Exploring Graphs

In this section, we'll learn about algorithms for exploring graphs by hopping from vertex to neighboring vertex.

Objectives. After learning this material, you should be able to:

- Describe search trees produced by exploring a graph.
- Execute depth-first search and breadth-first search on a graph and recall their runtime analyses.
- Contrast the search trees that can be produced by DFS versus by BFS.

8.1 Depth-first search

One of the most basic and important tasks to do on graphs is to explore them: starting from a known vertex, pick a neighbor, go to that neighbor, and repeat. In this way, we learn how vertices are related by distance. For example, we can learn if a graph is *connected* – can every vertex be reached from every other vertex?

We'll start with one of the most common exploration methods, **depth-first search (DFS)**. DFS is named "depth-first" because it explores as far as possible before "backtracking". We will compare it to breadth-first search later.

Algorithm 8.1 (Depth-First Search)**.**

```
DFS(G, s):
   # G = (V,E) is an undirected graph, s is a vertex
   for each v in V:
      let marked[v] = False
   explore(s)

# Subroutine: explore recursively
explore(v):
   set marked[v] = True
   print v                    # or do other useful work
   for each neighbor w of v:
```

```
    if not marked[w]:
        explore(w)
```

Exercise 8.1. Execute `DFS(G,u)` on the graph of Figure 7.1, reproduced here. In what order does it print the vertices?

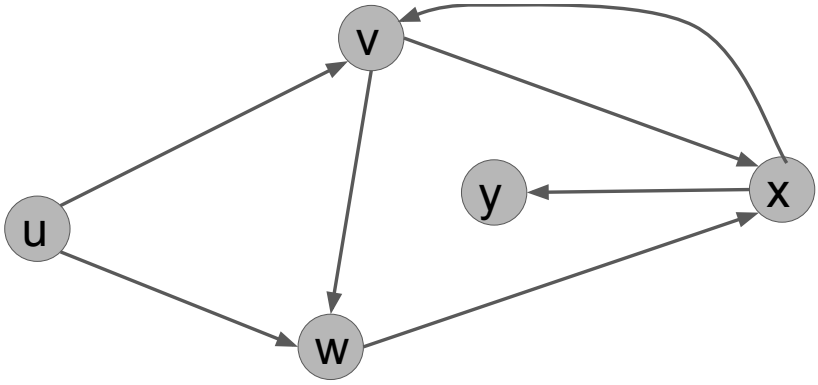
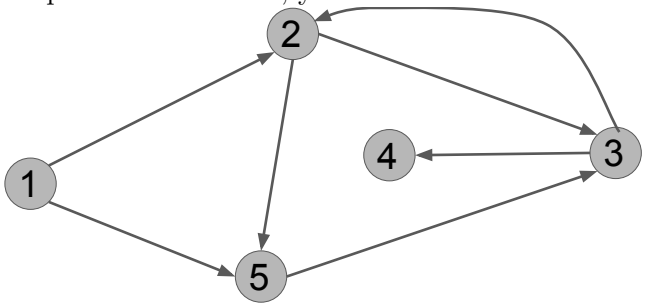


Figure 8.1

Hint: there are multiple correct answers, depending on in which order the neighbors of each vertex are listed.

Example solution.

Here is one answer: u v x y w.
To picture the answer, you can also draw on each vertex the order in which it is marked.



We can also represent the calls to `explore()` like this:

```
explore(u)
   explore(v)
      explore(x)
         explore(y)
      explore(w)
```

This represents the following order of calls to explore. We explore u, which steps to v, which steps to x, which steps to y. At that point, because y has no outgoing edges, the call `explore(y)` finishes and we backtrack to x. It has another out-edge to v, but v is already marked. So the call `explore(x)` finishes, and we backtrack to v. It has another out-edge to w, which is not yet marked, so we call `explore(w)`. This call does nothing because w's only out-neighbor is already marked. Then the call to explore(v) is finally done, and we go back to u. It has another out-neighbor w, but w is already marked by now, so we're done.

Notes. If the graph is fully connected, then we will eventually call `explore()` on every vertex. However, if the graph is not connected, then `DFS(G)` will only explore nodes reachable from s.. You can see this by trying the code on small examples of connected and disconnected graphs.

In many applications, we would replace line 3 of `explore(v)` with some useful operation involving vertex v. As long as that operation takes constant time, the complexity analysis below will still apply. If not, the analysis below could be modified accordingly.

8.1.1 Search trees

Given a graph, the execution of DFS produces a **search tree**, which we define as follows. The root is the first vertex r on which we call `explore(r)`. Then, for each node v, its children are all the nodes w where we call `explore(w)` from within the call to `explore(v)`.

For example, for the sample solution of Exercise 8.1, we get the following search tree (the blue edges), where the root is u.

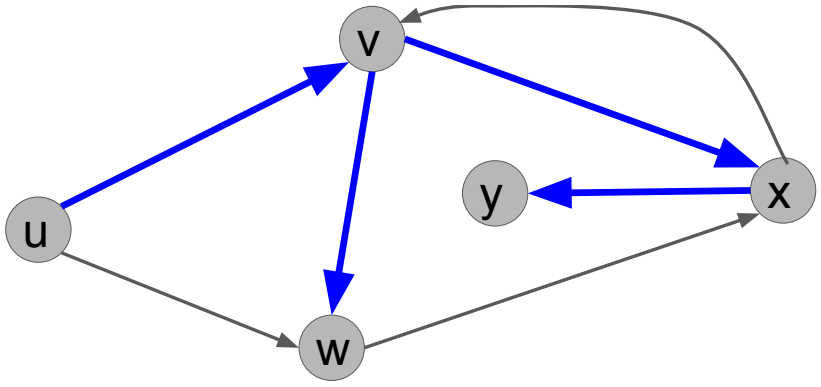


Figure 8.2

Exercise 8.2. Explain why the following search tree can **not** be obtained by running `DFS(u)`, no matter what order ties are broken in.

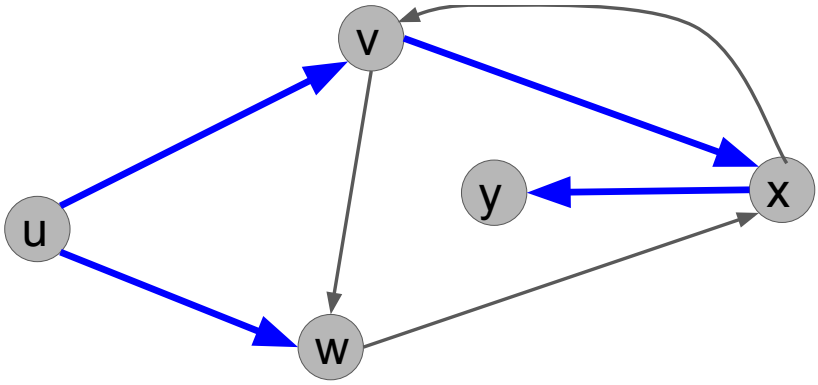


Figure 8.3

Solution.

From u, we either call `explore(v)` or `explore(w)` first. If we call `explore(v)` first, then v would always call `explore(w)` before u does, so the tree would have to have the edge (v,w) and not the edge (u,w). On the other hand, if we call `explore(w)` first, then that call would immediately call `explore(x)` because x would be unmarked. So the edge (w,x) would have to be present in the tree. Either way, the given search tree is not possible.

8.1.2 Runtime analysis

We now analyze the runtime of DFS. This is an interesting problem because it is recursive, but it is not a divide-and-conquer algorithm. We will need to carefully track the resource usage over the course of the algorithm. Recall that n is the number of vertices in G and m is the number of edges.

First, we need to settle on an input representation. We will choose an adjacency list. This implies that we can iterate over the neighbors $N(v)$ of v in time $O(|N(v)|)$, i.e. the number of neighbors. Recall that if we were using an adjacency matrix, then iterating through the neighbors of v would take $O(n)$ time regardless of the number of neighbors.

Now we prove some useful facts.

Lemma 8.1. *For each vertex u in the graph, `explore(u)` is called exactly once.*

Proof. Each vertex u starts unmarked (meaning that `marked[u]` is False) and is set to marked immediately when `explore(u)` is called. Since every call to `marked(u)` is protected by a statement of `if not marked[u]`, `explore(u)` can only be called once. But the for loop of line 4 of `DFS` ensures that explore is called for every vertex. □

Lemma 8.2. *Each call to 'explore(u)' does at most* $3|N(u)| + 2$ *steps, not counting recursive calls to 'explore()'.*

Proof. Lines 2 and 3 take 1 step each. Thanks to the adjacency list, the for loop over the $|N(u)|$ neighbors can be iterated through in constant time per neighbor. So the for loop takes 3 steps per loop, not counting work done within recursive calls. This gives a total of $3|N(u)| + 2$. □

Lemma 8.3. *The call to 'DFS(G)' does* $O(n)$ *work, not counting the call to 'explore()'.*

Proof. The for loop executes n times and does $O(1)$ work per loop. □

Now, we can analyze the runtime.

Proposition 8.1. *DFS runs in time* $O(n + m)$*, where* n *is the number of vertices and* m *is the number of edges.*

Proof. The total work can be counted as the work done within `DFS(G)`, which is called once, plus the sum of the work done in all the calls to `explore()`. We have shown that `DFS(G)` does $O(n)$ work internally. We have:

$$\begin{aligned}\sum_{u \in V} (3|N(u)| + 2) &= 3\left(\sum_{u \in V} |N(u)|\right) + 2n \\ &= 3(2m) + 2n \\ &= 6m + 2n.\end{aligned}$$

We used that the sum of the number of neighbors in the graph is equal to twice the number of edges. Adding this to the $O(n)$ used by DFS, we get a total running time bound of $O(n+m)$. □

8.1.3 Application: connectivity

DFS has many applications, but here is a simple one. Recall that an undirected graph is **connected** if there is a path from any vertex to any other vertex. Otherwise, it is disconnected.

Algorithm 8.2 (Connectivity)**.**

```
is_connected(G):
   pick any vertex s of G
   DFS(G, s)
   for each vertex v in G:
      if not marked[v]:
         return false
   return true
```

In alg. 8.2, we simply walk the graph from any starting vertex using DFS. When it returns, we check if every vertex has been marked. If so, we return true (the graph is connected), but if there is an unmarked vertex, we return false.

Exercise 8.3. What is the running time of `is_connected(G)`?

> **i** Solution.
>
> It is $O(n+m)$, because it calls DFS, which is $O(n+m)$, and then does $O(n)$ work itself due to the for loop.

Theorem 8.1. *The algorithm `is_connected(G)` is correct, i.e. returns true if and only if G is connected.*

Proof. First, we have to show it's correct on any connected graph. If G is connected, then for every v, there is a path from s to v. The path looks like $s, u_1, u_2, \dots, u_k, v$. Well, since there is an edge from s to u_1, we know that we call `explore(u1)` at some point. Since there is an edge from u_1 to u_2, we know we call `explore(u2)` at some point. Repeating, we eventually call `explore(v)`. So v is marked. This holds for every v, so `is_connected(G)` will be correct.

Now suppose G is not connected. Then there are two vertices v, w with no path between them. That means that s cannot have a path to both of them, since otherwise there would be a path between them that goes through s. So there is some vertex, call it v, with no path to s. But `DFS(G, s)` only explores s, vertices with an edge from s, vertices with an edge from those vertices, etc. So it only explores a vertex if there is a path to that vertex from s. So v is never marked, so `is_connected(G)` will be correct. □

Notice that in the proof, it didn't matter what vertex s we started from: the proof works for any s.

8.2 Breadth-first search

Now we'll look at a different way to explore graphs, **breadth-first search (BFS)**. Unlike DFS, in BFS we first look at all neighbors of the starting node, then all of its neighbors, and so on.

BFS will need a data structure called a **queue**, more specifically a **First-In-First-Out (FIFO)** queue. This data structure can be pictured as a list that supports the following operations:

FIFO Queue:

Operation	Time complexity	Meaning
append(v)	$O(1)$	add v to the end of the list
pop()	$O(1)$	remove and return the first item of the list
size()	$O(1)$	return current size of list

This data structure can be implemented e.g. with a linked list. Given a FIFO queue, BFS can be defined as follows:

Algorithm 8.3 (Breadth-First Search)**.**

```
BFS(G, s):
   # G = (V,E) is an undirected graph and s a vertex
   for each u in V:
      let marked[u] = False
   let Q = new FIFO_Queue
   Q.append(s)
   set marked[s] = True
   while Q.size() > 0:
      let u = Q.pop()
      print u                    # or do other useful work
      for each neighbor v of u:
         if not marked[v]:
            Q.append(v)
            set marked[v] = True
```

We can think of BFS as expanding outward in a wave, or in "layers". The zeroth layer is the start vertex, s. The next layer consists of all neighbors of s. These are all inserted into the queue before any other vertices, so they are popped from the queue before others as well. The next layer are all "neighbors of neighbors", and so on.

Exercise 8.4. Execute `BFS(G,u)` on this graph. In what order does it print the vertices?

> **i** Example solution.
>
> There are multiple solutions depending on the ordering of the vertices, but here is one: u, v, w, x, y. In other words, the vertices are popped from the queue in this order:

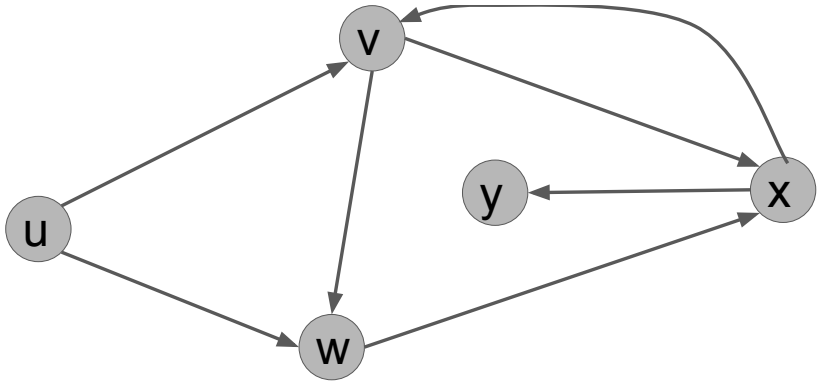


Figure 8.4

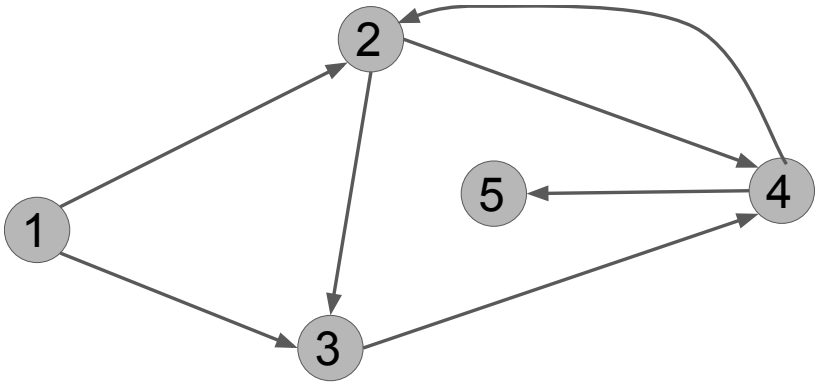


Figure 8.5

- First, the start vertex, u, is popped. Its two out-neighbors are v and w, and both are unmarked. It places them in the queue, let's suppose in the order v, w.
- Then, v, is popped. Its two out-neighbors are x and w. First, x is unmarked, so it is added to the queue. But w is already marked, so it is not added again.
- Then, w is popped. Its one out-neighbor is x. But x is already marked, so nothing happens.
- Then, x is popped. Its two out-neighbors are y and v. First, y is unmarked, so it is added to the queue. But v is already marked and is not added.
- Finally, y is popped. It has no out-neighbors.
- Now the queue is empty and the algorithm halts.

In the case of BFS, the search tree consists of edges (u, v) if, when we popped u from the queue and considered its neighbor v, we called `Q.append(v)`. This represents that we "found" v from u.

Exercise 8.5. What is a search tree corresponding to `BFS(G,u)`?

Example solution.

There are multiple correct answers. For the example BFS traversal in the sample solution to Exercise 8.4, the BFS search tree is:

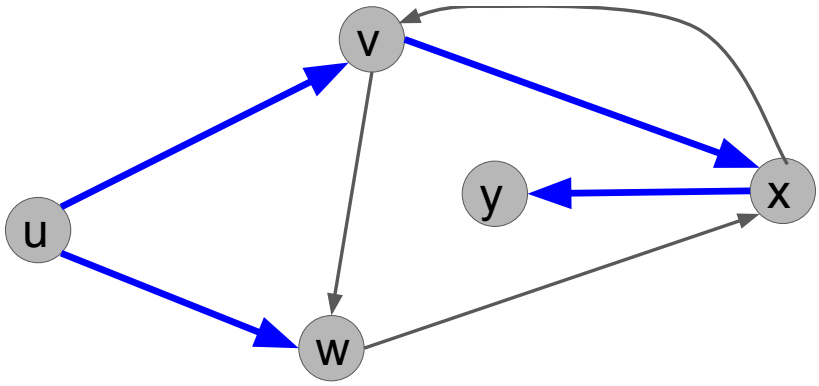


Figure 8.6

8.2.1 Comparing BFS and DFS

With practice, you should be able to run DFS and BFS in your head on small graphs and compare their search trees, as well as all possible search trees that each can produce. Can you solve this exercise?

Exercise 8.6. For the example graph above (Figure 8.4), is there any search tree that can arise from both `BFS(G,u)` and `DFS(G,u)`? If so, give it. If not, explain why not.

Solution.

The answer is no. To see why, note that a BFS search tree must contain both (u, v) and (u, w), because the first thing it does is pop u and add v and w (in some order). But we claim no DFS search tree can contain both those edges. To prove that, note the first recursive call is either `explore(v)` or `explore(w)`. If `explore(v)`, it must eventually called `explore(w)` from v before coming back to u, so (v, w) is in the DFS search tree, so (u, w) cannot be: we can't add v twice. On the other hand, if the first recursive call is `explore(w)`, then the next call will be `explore(x)` and from x we will call `explore(v)`, all before we backtrack to u. So we won't call `explore(v)` from u.

8.2.2 Runtime analysis

To analyze BFS, we need to make one observation.

Lemma 8.4. *In BFS, every vertex is added to Q at most once.*

Proof. Each place in the code that we add a vertex to Q, namely lines 6 and 13, we immediately set that vertex as marked. And we only ever add unmarked vertices to Q, so each vertex can only be added once. □

Theorem 8.2. *The runtime of BFS is $O(n+m)$, where n is the number of vertices and m is the number of neighbors.*

Proof. We can analyze BFS as follows.

```
BFS(G, s):
    # G = (V,E) is an undirected graph and s a vertex
    for each v in V:              # O(n) total
       let marked[v] = False      # O(1) each time
    let Q = new FIFO_Queue        # O(1)
    Q.append(s)                   # O(1)
    set marked[s] = True          # O(1)
    while Q.size() > 0:           # at most n iterations
       let v = Q.pop()            # O(1) each time
       print v                    # O(1) each time
       for each neighbor w of v:  # O(|N(v)|) iterations
          if not marked[w]:       # O(1) each time
             Q.append(w)          # O(1) each time
             set marked[w] = True # O(1) each time
```

As with DFS, the key for the for loop of line 11 is to not consider the worst-case every time. Instead, we add up the total amount of work done by those lines of the algorithm over the course of the algorithm. This total is $O(\sum_{v \in V} |N(v)|) = O(n+m)$. We also have that lines 8-10 contribute a total of $O(n)$, and the same for lines 2-7. The total is $O(n+m)$. □

9 Shortest Paths

We now turn to one of the most important graph problems, finding shortest paths through a graph.

Objectives. After learning this material, you should be able to:

- Execute BFS and Dijkstra's algorithm to find shortest paths in a graph.
- Show how BFS and Dijkstra's fail when their assumptions are not satisfied.
- Discuss the runtime analysis of Dijkstra's depending on the priority queue data structure.

9.1 The shortest paths problem

So far, we have used DFS and BFS to explore the structure of a graph without a particular destination in mind. Now, we'll consider the shortest paths problem. This problem needs to be solved every time you search for directions (driving, walking, etc.), as well as in many other cases.

Shortest paths problem.

- **Input:** a graph $G = (V, E)$ and a start vertex $s \in V$.
- **Output:** an array `dist` where, for each $t \in V$, `dist[v]` is the length of the shortest path from s to t.

This is often called the *single-source* shortest paths problem because, given a single source s, we find the shortest paths to all possible destinations t. Often, we only want to know the distance from s to one particular destination t, but the algorithms will turn out to be almost the same.

9.2 Unweighted graphs

First, we will consider unweighted graphs. Here the length of a path is the number of edges or "hops". We will modify BFS to solve this problem. In this variant, `dist[v]` represents the distance to v. We can also use it in the same way as the `marked` array from the previous BFS implementation. If `dist[v] == infinity`, then v is not yet marked. If `dist[v]` is a number, then `v` has been marked.

Algorithm 9.1 (BFS for Shortest Paths)**.**

```
BFS_dist(G, s):
   # G = (V,E) is an unweighted graph and s a vertex
   for each v in V:
      let dist[v] = infinity
   let Q = new FIFO_Queue
   set dist[s] = 0
   Q.append(s)
   while Q.size() > 0:
      let v = Q.pop()
      for each neighbor w of v:
         if dist[w] = infinity:   # not yet marked
            Q.append(w)
            set dist[w] = dist[v] + 1
   return dist
```

As an exercise, you can go back to the previous example graph, or draw a new one, and practice executing `BFS_dist`.

9.2.1 Correctness

Theorem 9.1. *`BFS_dist(G,s)` correctly solves the shortest-paths problem on unweighted graphs.*

Proof. One note is that, once a node's distance is set to a number, it is never updated. This follows because we only set a node's distance if it is currently infinity.

We proceed inductively. The claim we will prove is that we pop nodes from the queue in order of distance: all nodes at distance 0, all nodes at distance 1, and so on; and when we set a node's distance, it is correct.

The base case is $d = 0$. The only distance-zero node is the start node, s. The algorithm does set `dist[s] = 0` in line 6, then pop it from the queue first.

Inductively, suppose the claim is true up to distance d for some $d \geq 0$. We must prove it for distance $d + 1$. A node `w` is at distance $d + 1$ if there is a path $s, \ldots, w$ of distance $d + 1$, but there is no path of distance d or shorter. Because there's no path of distance d or shorter, by IH, we pop and set all the nodes up to distance d before we get to any node of distance $d + 1$. But a node is at distance $d + 1$ if it is a neighbor of a node at distance d. By IH, we have just popped all nodes of distance d and set the distances for all their unmarked neighbors to distance $d + 1$. That proves the inductive claim.

To finish, we should note that if there is no path at all from s to v, then at the end, we have `dist[v] = infinity`, which correctly indicates that there is no path. □

9.2.2 Time and space complexity

By checking the small changes we made from `BFS` to `BFS_dist`, we can confirm that the time complexity is still $O(n+m)$. It's also good to check the space usage. We have several variables representing nodes, which take $O(1)$ space. Then we have `dist`, and array of length n. Then we have `Q`. We have noted that every node is added to `Q` at most once, so it requires at most n space as well. We conclude that the algorithm uses $O(n)$ space. (Note that the input, an adjacency list, would take more space than this, $O(n+m)$.)

9.2.3 Finding the paths themselves

An important point is that `BFS_dist` found the *lengths* of the shortest paths, but it didn't actually find the paths themselves! It could tell us that a certain node had distance 103 from s, but not how to get there in 103 steps. However, this can be easily fixed. When we set `dist[w] = dist[v] + 1`, we simply add a note at w to say that the shortest way to get there is from v. And v will have its own note, and so forth. Here's the implementation; the two new lines are 5 and 16.

Algorithm 9.2 (BFS with Path Pointers)**.**

```
BFS_dist(G, s):
   # G = (V,E) is an unweighted graph and s a vertex
   for each v in V:
      let dist[v] = infinity
      let prev[v] = null      #<<
   let Q = new FIFO_Queue
   set dist[s] = 0
   Q.append(s)
   while Q.size() > 0:
      let v = Q.pop()
      for each neighbor w of v:
         if dist[w] = infinity:
            Q.append(w)
            set dist[w] = dist[v] + 1
            set prev[w] = v      #<<
   return dist, prev             #<<

get_path(s, t, prev):            #<<
```

```
path = [t]    # list with just t  #<<
while t != s:                     #<<
   set t = prev[t]                #<<
   if t == null:                  #<<
      return "no path exists"     #<<
   put t at front of path         #<<
return path                       #<<
```

For an example, we can revisit our example graph Figure 8.4 with source u (also shown below in Figure 9.1). Here, `prev` will point in the opposite direction of the blue arrows. For instance, the shortest path from u to y has length 3. When we call `get_path(u, y, prev)`, it follows these steps:

- We first put y in our list, called `path`.
- Then we get `x = prev[y]`. We put x at the front of our list, which is now `path = [x,y]`.
- Then we get `v = prev[x]`. We put v at the front of our list, which is now `[v,x,y]`.
- Then we get `s = prev[v]`. We put s at the front of our list, which is now `[s,v,x,y]`.
- Then we terminate the loop and return our list, `path = [s,v,x,y]`.

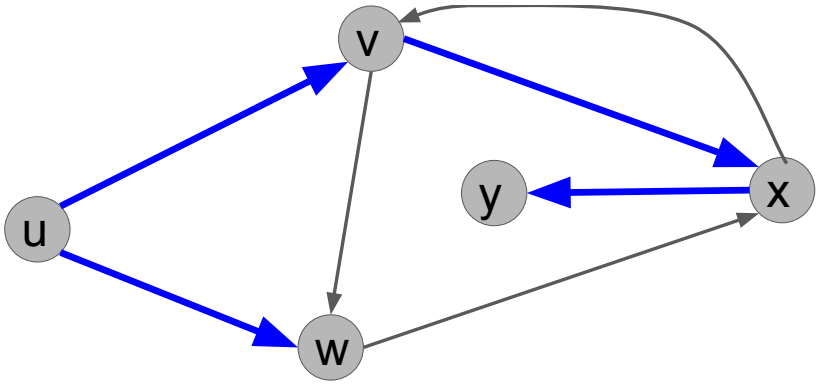


Figure 9.1

9.3 Weighted graphs

We'll now consider graphs with weights on the edges. As mentioned above, the length of a path is now the sum of the weights of the edges in the path.

9.3.1 Failure of BFS on weighted graphs

Does BFS find shortest paths on weighted graphs? We know it probably shouldn't, because it doesn't look at the edge weights at all. But how can it fail? You should try to solve the following exercise. A solution is not provided, because this is a problem every student needs to be able to do!

Exercise 9.1. Give a weighted graph G, source s, and destination t for which BFS(G,s) does not find the shortest path from s to t.

> **i** Hint.
>
> BFS will always find the smallest *number of hops* from s to t. What if there is a path with more hops, but shorter distance?

9.3.2 Dijkstra's algorithm

We'll now solve the problem for weighted graphs. Let $wt(u, v)$ be the weight of the edge from u to v. We assume the weights are positive, i.e. $wt(u, v) > 0$ for all u, v. We can let $wt(u, v) = \infty$ to denote that there is no edge between u and v.

9.3.2.1 The key fact

To create an algorithm, usually, we need a *fact*. The fact about how the problem is structured allows us to write an algorithm that leverages the structure. With BFS, a key fact we used was that if v is at distance $d + 1$, then there is a vertex u at distance d and an edge (u, v). Our key fact for weighted graphs is similar.

Fact 9.1. *On a weighted graph G with source vertex s, let $d(v)$ be a function denoting the length of the shortest path from s to v. Let $IN(v)$ be the set of in-neighbors of v, i.e. the vertices that have an edge to v. Then for any $v \neq s$:*

- *For all $u \in IN(v)$, we have $d(v) \leq d(u) + wt(u, v)$.*
- *Furthermore, $d(v) = \min_{u \in IN(v)} d(u) + wt(u, v)$.*

The fact is saying that the shortest path to v has to go from s to one of v's in-neighbors, then hop to v. Furthermore, the *shortest* path to v has to take the *shortest* path to one of its in-neighbors, then hop to v. This fact is illustrated in the next figure, where other edges of the graph are omitted and the dashed lines represent some shortest path through the graph.

For vertex v, we can check the two points of Fact 1 in the following table. We see that the distance to v is d(v) = 20, which is the minimum over these options.

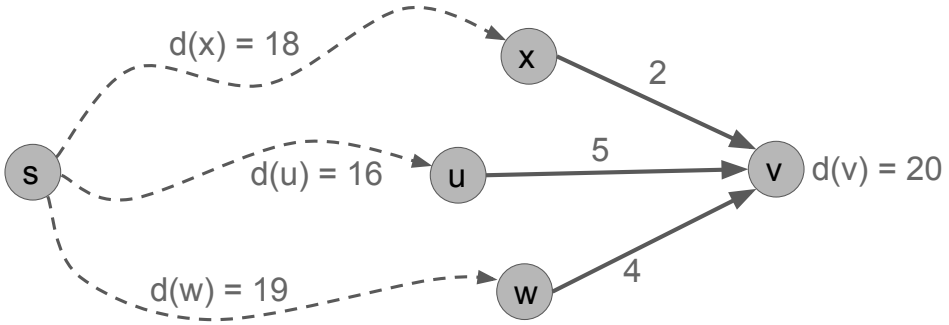


Figure 9.2

d(x) = 18	wt(x,v) = 2	d(x) + wt(x,v) = 20
d(u) = 16	wt(x,v) = 5	d(x) + wt(x,v) = 21
d(x) = 19	wt(x,v) = 4	d(x) + wt(x,v) = 23

9.3.2.2 Dijkstra's

The idea of Dijkstra's algorithm is similar to a modified breadth-first search. We will process vertices in order of their distance from the source. For each vertex, we will set the distances of its neighbors. However, instead of locking in the distance the first time, we will update the distance using the idea of the fact.

The other change is that we need a fancier data structure, which we call a *priority queue*. In a priority queue, every object in the queue has a value. We always pop the object with the smallest value. Here are the operations; we'll discuss the time complexity later.

Priority Queue:

Operation	Meaning
insert(u, d)	insert object u with value d
update(u, d)	update the value of u to be d
pop_smallest()	remove and return the object with smallest value
size()	return the number of objects in the queue

Now, we can give Dijkstra's algorithm.

Algorithm 9.3 (Dijkstra's)**.**

```
dijkstra(G, s):
   # G is a weighted graph with weights w(u,v)
   let Q = new Priority_Queue
   for all vertices u:
      let dist[u] = infinity
      Q.insert(u, infinity)
   set dist[s] = 0
   Q.update(s, 0)
   while Q.size() > 0:
      let u = Q.pop_smallest()
      for each neighbor v of u:
         set dist[v] = min(dist[v], dist[u] + w(u,v))
         Q.update(v, dist[v])
   return dist
```

As we mentioned, there are two main changes from BFS. The first is to use a priority queue so that we always pop the remaining vertex with minimum distance. The second is that when we process u, we update the distances of all its neighbors v. If `dist[v]` is currently larger than the distance available by going to u and then hopping to v, we update it.

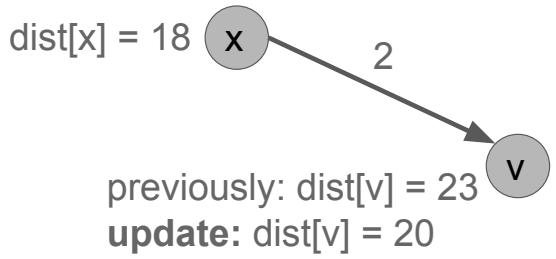


Figure 9.3: Line 11 of `dijkstra(G,s)`.

As usual with algorithms, the best way to understand it is to execute it by hand on some examples.

Exercise 9.2. Given the below graph, simulate the execution of Dijkstra's algorithm starting from u. Report, at the beginning of each iteration of the while loop, the state of the priority queue and the distance table, and which vertex is popped from the queue.

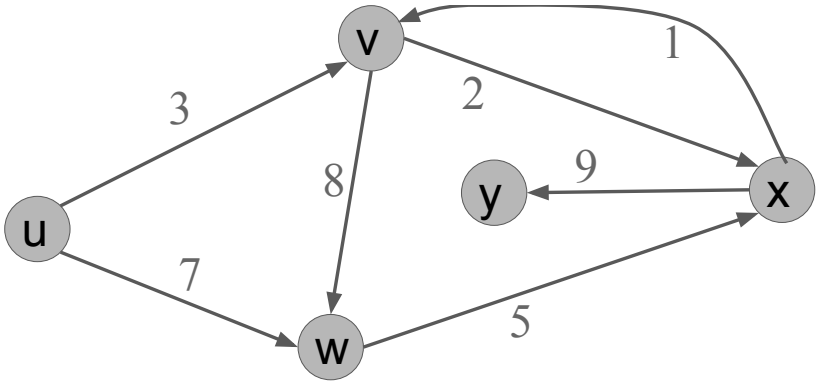


Figure 9.4

Example solution.

Round	dist[u]	dist[v]	dist[w]	dist[x]	dist[y]	queue	vertex popped
1	0	∞	∞	∞	∞	u, v, w, x, y	u
2	0	3	7	∞	∞	v, w, x, y	v
3	0	3	7	5	∞	x, w, y	x
4	0	3	7	5	14	w, y	w
5	0	3	7	5	14	y	y
6	0	3	7	5	14		

9.3.3 Reconstructing the actual paths

Just as with BFS, the initial version of Dijkstra presented only returns the *length* of the shortest path, not the path itself. But just as with BFS, it is pretty straightforward to modify the algorithm in the same way: we maintain a `prev` array, where `prev[v]` represents the previous vertex for v along the shortest path from s. Here is how we update the algorithm, in lines 7 and 15-16.

Algorithm 9.4.

```
dijkstra(G, s):
   # G is a weighted graph with weights w(u,v)
   let Q = new Priority_Queue
```

```
for all vertices u:
   let dist[u] = infinity
   Q.insert(u, infinity)
   let prev[u] = null      #<<
set dist[s] = 0
Q.update(s, 0)
while Q.size() > 0:
   let u = Q.pop_smallest()
   for each neighbor v of u:
      set dist[v] = min(dist[v], dist[u] + wt(u,v))
      Q.update(v, dist[v])
      if dist[v] == dist[u] + wt(u,v):  #<<
         set prev[v] = u                #<<
return dist, prev  #<<
```

In particular, in lines 15-16, if the current shortest path to v is indeed from u, then we set `prev[v] = u`. Given these changes, reconstructing a shortest path can be done with the same algorithm `get_path(s, t, prev)`, from BFS (alg. 9.2).

9.3.4 Correctness

Theorem 9.2. *For graphs with positive edge weights, Dijkstra's algorithm correctly solves the single-source shortest paths problem.*

Proof. Let us number the vertices in order that the algorithm pops them from the queue: $s = v_1, v_2, v_3, \dots, v_n$.

We prove by induction on $k = 1, \dots, n$ that, when we pop v_k, its distance is set correctly: `dist[v_k]` $= d(v_k)$. We note that distances start at ∞ and only decrease, and cannot fall below the true distances because every update corresponds to the distance of a path to v_k. So if `dist[v]` $= d(v)$ at any point, the equality holds true forever.

Base case: $k = 1$, i.e. the case of s. When we pop s, we correctly have `dist[s] = 0`.

Inductive case: Suppose that $v_1, \dots, v_k$ had their distances set correctly when popped. We first note that for all remaining v, we have `dist[v]` equal to the shortest path of the form $s, \dots, u, v$ where all of $s, \dots, u$ have already been popped. This follows because when each in-neighbor u of v was popped, its distance was set correctly by inductive hypothesis and `dist[v] = min(dist[v], dist[u] + wt(u,v))` was called.

Now consider the shortest path that is not of this form. It must be of the form $s, \dots, u, v', \dots, v$ where $s, \dots, u$ have been popped, but v' has not. Because v was the smallest object in the priority queue, we must have `dist[v] <= dist[v']`. But by the note above, since this path

starts $s, \ldots, u$, if it is a shortest path, it would have `dist[v'] = d(v')`. So the distance to v along this path, which is only longer, is longer than `dist[v]`. So this path is longer. So `dist[v]` is indeed equal to the shortest distance from s. □

9.3.5 Running time

First, let's analyze Dijkstra for a generic priority queue. Then, we'll plug in specific implementations of the queue. This time, we'll skip any $O(1)$ time operations and just look at how many times each loop runs.

```
dijkstra(G, s):
   # G is a weighted graph with weights w(u,v)
   let Q = new Priority_Queue
   for all vertices u:                          # n iterations
      let dist[u] = infinity for all u in V
      Q.insert(u, infinity)
   set dist[s] = 0
   Q.update(s, 0)
   while Q.size() > 0:                          # at most n iterations
      let u = Q.pop_smallest()
      for each neighbor v of u:                 # at most O(n + m) iterations total
         let dist[v] = min(dist[v], dist[u] + wt(u,v))
         Q.update(v, dist[v])
   return dist
```

The time complexity is therefore $O(n + m)$ plus the time for $O(n)$ calls to `Q.insert()` and `Q.pop_smallest()` + $O(n + m)$ calls to `Q.update()`. To complete the analysis, we need to know the total time taken by these priority queue operations.

9.3.5.1 A binary tree priority queue

One way to implement a priority queue is with a binary search tree, which keeps its objects sorted by value. In a binary search tree, all operations take $O(\log(n))$ time, where the maximum number of vertices in the tree is n.

Binary tree:

Operation	Time complexity	Meaning
insert(u, d)	$O(\log n)$	insert object u with value d

Operation	Time complexity	Meaning
update(u, d)	$O(\log n)$	update the value of u to be d
pop_smallest()	$O(\log n)$	remove and return the object with smallest value
size()	$O(1)$	return the number of objects in the queue

In this case, our running time is:

$$O(n + m + n\log(n) + (n + m)\log(n)) = O((n + m)\log(n)).$$

However, there are slightly faster data structures available. The best for Dijkstra's is the Fibonacci heap. It has the amazing property that, over the course of all n insertions and updates, the total time taken is $O(n)$. This is a slightly different statement than saying that the running time is $O(1)$ per update, because there could be a few updates that are slower than that. Hence, we call this kind of guarantee an $O(1)$ "amortized" per operation.

Fibonacci heap:

Operation	Time complexity	Meaning
insert(u, d)	$O(1)$ amortized	insert object u with value d
update(u, d)	$O(1)$ amortized	update the value of u to be d
pop_smallest()	$O(\log n)$	remove and return the object with smallest value
size()	$O(1)$	return the number of objects in the queue

With this data structure, our running time is:

$$O(n + m + n\log(n) + (n + m) \cdot 1) = O(n\log(n) + m).$$

Recall that, for unweighted graphs, BFS ran in time $O(n + m)$. Dijkstra's runs in time $O(n\log(n) + m)$, just slightly slower while handling any positive edge weights.

Part IV

Topic D: Greedy Algorithms

10 Greedy Algorithms and Knapsack

This section introduces greedy algorithms and the knapsack problem. Greedy is an important general algorithmic paradigm to know, including its pitfalls.

Objectives. After learning this material, you should be able to:

- Explain the concept of a "greedy" algorithm.
- Recognize exchange arguments for greedy proofs of correctness.
- Recognize variants of the knapsack problem and prove when greedy is optimal, or provide a counterexample when it is not.

10.1 Definition and knapsack

We call an algorithm **greedy** if it takes a sequence of decisions that each look "locally" optimal at the current moment, without appearing to worry about global solution quality. To get the idea, let's look at an example.

Knapsack problem with uniform weights.

- **Input:** a list of items $i = 1, \dots, n$, with a value $v_i \geq 0$ for each. Also, an integer $W \geq 0$.
- **Output:** the subset of W of the items that have the largest total value (i.e. sum of the values).

The problem's name comes from the image that our knapsack can fit at most k items, and we want to put the highest-value total set in it. Before continuing, can you solve this problem?

A natural "greedy" algorithm is the following, which we'll state informally rather than in pseudocode:

- repeat W times:
  - add the highest-value item remaining to the knapsack.

This algorithm is greedy: it makes a sequence of decisions that are optimal in the moment, i.e. taking the highest-value item available. But it turns out to be globally optimal too. This is probably not surprising in this case, but proving it will introduce some important ideas.

Proposition 10.1. *The greedy algorithm is correct, i.e. returns the subset of size W with highest total value.*

Proof. We use an *exchange argument*, which involves showing that any other solution can be improved by swapping or exchanging part of its solution out for the greedy algorithm's solution.

Let us re-name the items so that $v_1 \geq v_2 \geq \cdots \geq v_n$. The greedy algorithm's solution set is $\{1, \ldots, W\}$ with value $\sum_{j=1}^{W} v_j$.

To prove it's optimal, consider any other solution set S, a subset of $\{1, \ldots, n\}$ of size at most W. Suppose $i > W$ and $i \in S$. Then there must be some item $j \in \{1, \ldots, W\}$ such that $j \notin S$. Notice that $j < i$, so $v_j \geq v_i$.

Let's make a swap: remove i from S and add j instead. The value of S goes up by $v_j - v_i \geq 0$. So the value of S only improves.

Now let's just repeat this process over and over. Eventually, $S = \{1, \ldots, W\}$ because we swap in all of the first W items that were missing. The value of S only goes up each time. So the greedy solution set is optimal. □

While there's no universal definition of greedy algorithms, many of them share the above components:

- The algorithm's job is to construct a solution set.
- The goal is to maximize some total "value" of the solution across the entire set (a global objective).
- The algorithm proceeds by "greedily" choosing an item that looks the best "locally" in the current moment, and repeating this until the solution set is constructed.

We will see several other examples, but first, let's look at variants of the problem where greedy fails.

10.2 Failure of greedy algorithms

For algorithm designers, almost as important as designing correct algorithms is identifying failure points of incorrect ones. Let's look at a variant of knapsack where the greedy algorithm fails.

Knapsack problem.

- **Input:** a list of items $i = 1, \ldots, n$, with a value $v_i \geq 0$ for each and a weight $w_i \geq 0$ for each. Also, a weight limit $W \geq 0$.
- **Output:** the subset of the items that have the largest total value (i.e. sum of the values), whose total weight is at most W.

Let's do an example.

Item i	Value v_i	Weight w_i
1	10	1
2	15	3
3	7	2

Suppose the weight limit is $W = 4$. Our greedy algorithm is the same as before, but must respect the weight limit: add the highest value item, then the next, and so on. If an item doesn't fit in our weight limit, skip it, and once no more items fit, we're done.

Exercise 10.1. On the above instance with a weight limit $W = 4$, what output does this greedy algorithm produce? Is that optimal?

i Solution.

Greedy first takes item 2, which has a value of 15. It has weight 3. It then takes item 1, with a value of 10 and weight of 1. The total value is now 25 and total weight is 4. Now, we can't take any more items and stop.
Yes, 25 is the optimal possible solution for this instance. We can't fit all three items, and we have the two most valuable items.

Now comes the key question. Can you solve this one?

Exercise 10.2. Give a different weight limit W so that, on the above instance, the greedy algorithm fails to find the optimal solution. Explain how.

i Solution.

We can use $W = 3$. In this case, greedy will take item 2 and then stop, for a total value of 15. But we could take items 1 and 3 instead for a total value of 17, while still having a weight of 3.

It's worth reflecting on why greedy failed here. We needed to take into account the resources being used up (i.e. weight) by the items, not just their value.

Here is another exercise. It considers a different type of greedy rule that tries to account for the weights.

Exercise 10.3. Consider the following greedy algorithm for knapsack: always take the item with smallest weight, until the knapsack is full (i.e. no more items fit in the weight limit). Is this algorithm always optimal? If so, give a proof. If not, give a counterexample and explain.

Example solution.

It is not always optimal. There are many possible counterexamples, but the general idea is to make low-weight items have *very* low values.

Item i	Value v_i	Weight w_i
1	1	1
2	1	1
3	10	2

Suppose the weight limit is $W = 2$. This greedy algorithm will take items 1 and 2, for a total value of 2. But the optimal solution is to take just item 3, for a value of 10.

11 Interval Scheduling

This section discusses another set of problems where greedy algorithms can often be optimal.

Objectives. After learning this material, you should be able to:

- Define the interval scheduling problem and execute greedy algorithms for it.
- Prove that greedy is correct via an exchange argument.
- Decide for which variants greedy is not correct and provide counterexamples.

11.1 The interval scheduling problem

Interval scheduling is a problem where we have a resource available over time, and a set of requests to use the resource. An example would be a server and requests to use the server to run heavy computations; another would be a soccer field and requests to schedule games.

Each request will have a start and end time. Our goal is to schedule as many requests as possible, but we are not allowed to have overlapping requests.

Interval scheduling problem:

- **Input:** a list of requests $i = 1, \ldots, n$. Each has a start time $s(i)$ and an end time $t(i)$, which we can assume are integers.
- **Output:** a subset S of the requests that we want to schedule. The subset must not be overlapping, i.e. only one request may be executing at any given time. The goal is to maximize the size of the subset.

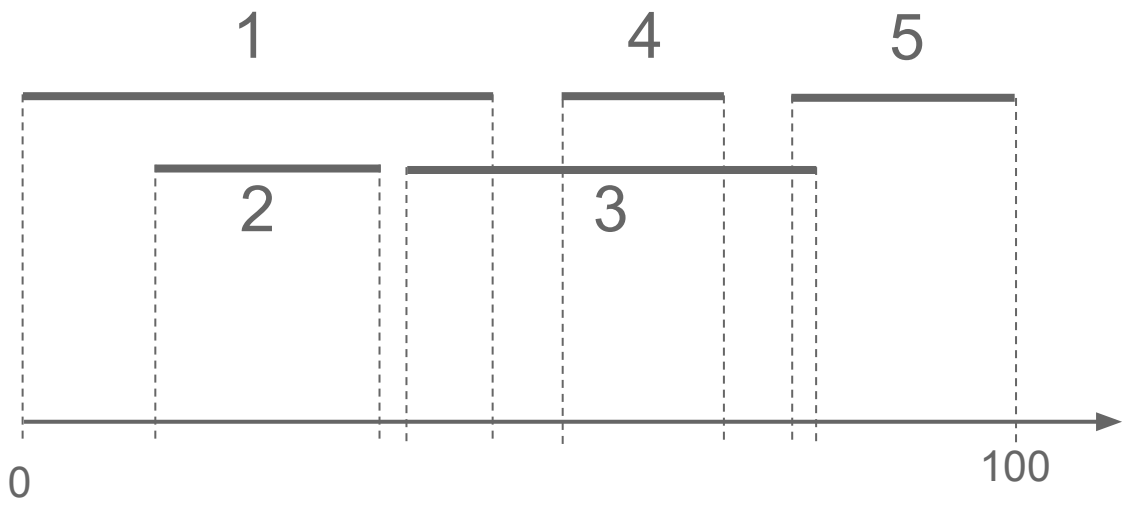
As always when you see a new problem, it's good to do some examples.

Exercise 11.1. Given the following instance of interval scheduling, what is the optimal solution? Are there more than one?

i	$s(i)$	$t(i)$
1	0	40
2	10	30
3	35	80
4	50	65

i	$s(i)$	$t(i)$
5	75	100

Hint: draw a diagram like this...



Solution.

The maximum number of requests we can schedule is three. There are two optimal solutions: $\{1, 4, 5\}$ and $\{2, 4, 5\}$. In either case, there are no overlaps and we schedule 3 different requests. There is no way to schedule four or five requests.

11.2 First greedy attempt

There are many possible greedy strategies here. Let's brainstorm a few greedy algorithms. Can you think of any?

A few to get started.

- Always pick the smallest interval (that doesn't overlap with any that you've picked so far).
- Always pick the an interval that overlaps with as few others as possible.

Can you think of other greedy strategies?

It turns out that a number of natural-looking greedy strategies don't actually work. Let's try disproving one.

Exercise 11.2. Using a counterexample, prove that the algorithm that always picks the smallest valid interval is not optimal.

Example solution.

There are many possible answers; here's one.

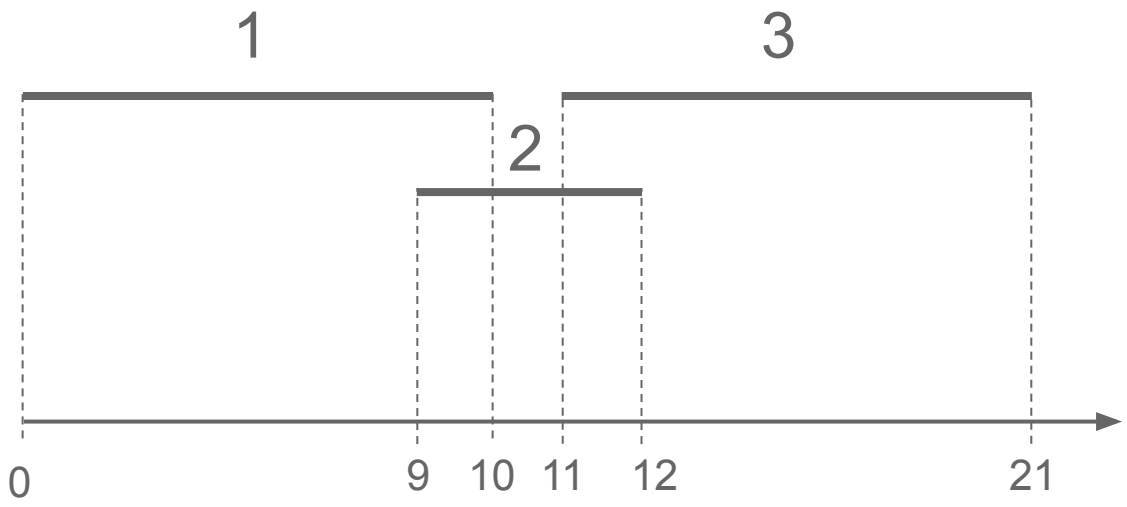


Figure 11.1: 3 intervals. We have $s(1) = 0, t(1) = 10$, meaning that the first interval starts at time zero and ends at time 10. We have $s(2) = 9, t(2) = 12$. And $s(3) = 11, t(3) = 21$.

The optimal solution is $\{1, 3\}$, giving us two intervals. But the greedy-by-smallest algorithm would first pick interval 2. Then it would not be able to pick any more intervals, so it would have a solution size of just one interval.

11.3 A correct greedy algorithm

It turns out this problem does have a greedy algorithm that optimally solves it, but we have to be a bit clever about what to be greedy with. Here's the algorithm:

- Select the interval that is scheduled to finish first, i.e. has minimum $t(i)$. Remove from consideration any intervals that overlap with it.

- Repeat until no intervals remain.

Theorem 11.1. *The above algorithm, greedy by first finish time, is correct for interval scheduling.*

Proof. We will use an exchange argument. Suppose greedy selects the requests $S = \{i_1, \dots, i_k\}$, in order of start time and finish time. Consider any other feasible solution (meaning no overlaps), $R = \{j_1, \dots, j_r\}$, also in order of start and finish time. We want to show that $k \geq r$, that is, greedy selects at least as many requests.

First, because of how greedy works, $t(i_1) \leq t(j_1)$. So we can remove j_1 from R and add i_1 and R is still feasible, and the number of requests has not changed. Next, consider j_2. We have $s(j_2) > t(j_1) \geq t(i_1)$. So greedy could have added j_2 next. But greedy added i_2. Therefore, because of how greedy works, $t(i_2) \leq t(j_2)$. So we can remove j_2 from R and add i_2 and R is still feasible, and the number of requests has not changed.

We repeat this argument until the very end. Suppose for contradiction that $k < r$, so after we exchange i_k into R for j_k, there still remains some request j_{k+1}. If this were possible, then greedy would have added j_{k+1} to its solution, since it is feasible (we must have $s(j_{k+1}) > t(j_k) \geq t(i_k)$). This is a contradiction. We conclude $k \geq r$. □

12 Minimum Spanning Trees

This section discusses a fundamental problem in graph theory and uses greedy algorithms to solve it.

Objectives. After learning this material, you should be able to:

- Define the minimum spanning tree (MST) problem.
- Use properties of spanning trees to solve related problems.
- Execute Prim's and Kruskal's algorithms.

12.1 Spanning trees

First, we need to recall the definition of a tree. Previously in this class, we defined *rooted trees*. We can think of a tree as a rooted tree, where we deleted all information about which node is the root and which nodes are children of which. Here is a formal definition:

Definition 12.1 (Tree)**.** An undirected graph $T = (V, E)$ is a tree if it is connected and has no simple cycles.

Remember that *connected* means there is a path from every vertex to every other vertex. A *simple cycle* is a path of length at least three that starts and ends at the same vertex and does not repeat edges. Here is a nice fact about trees.

Proposition 12.1. *Any tree on n vertices has exactly $n - 1$ edges.*

Proof. By induction on n. The base case is $n = 1$. A graph with one vertex will have no edges (we generally assume that self-loops are not allowed), so the formula is satisfied.

Now let $n \geq 2$ and suppose that any tree on $n' < n$ vertices has exactly $n' - 1$ edges. Consider a tree on n vertices. Let us delete any edge $\{u, v\}$. We first claim this disconnects the graph into two disjoint trees. First, deleting an edge cannot have created a cycle. Next, it is partitioned into two connected components: originally, every vertex was reachable from u, and the path from u either included the edge $\{u, v\}$ or not. If so, the vertex is still reachable from v after the disconnection, and if not, it is still reachable from u after the disconnection. Further every vertex is in either one case or the other, but not both, as otherwise there would be a cycle in

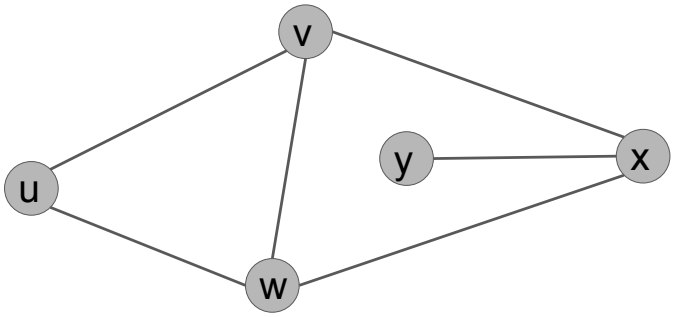
the original graph involving a path to the vertex from u and from v, along with the edge $\{u, v\}$. So the graph is partitioned into two disjoint connected acyclic graphs, i.e. trees.

The trees have $n_1, n_2 \geq 1$ vertices with $n_1 + n_2 = n$. By inductive hypothesis, they have $n_1 - 1$ and $n_2 - 1$ edges. Adding $\{u, v\}$, the total number of edges in our original tree is $n_1 - 1 + n_2 - 1 + 1 = n - 1$. □

Next, we can define a spanning tree. Intuitively, given a connected, undirected graph, we may want to delete as many edges as possible while keeping the graph connected. The minimum set of edges we need to keep form a tree, called a spanning tree.

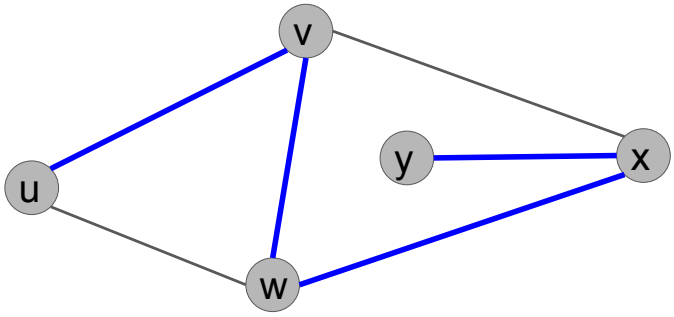
Definition 12.2 (Spanning tree)**.** In an undirected graph $G = (V, E)$, a spanning tree is a tree $T = (V, E')$ where $E' \subset E$. In other words, it is a subgraph of the original graph that includes all the vertices and is a tree.

Exercise 12.1. Give a spanning tree of this graph.



Example solution.

There are several solutions. Because there are 5 vertices, any subset of 4 edges that keep the graph connected form a spanning tree. Here is one (the thick blue edges):



12.2 Minimum spanning tree (MST) and reverse-deletion

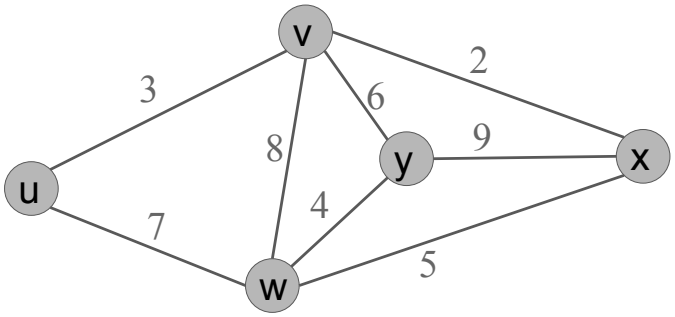
In the minimum spanning tree problem, our graph is undirected and weighted. We assume the weight $wt(u, v)$ on each edge is positive.

Minimum spanning tree (MST) problem:

- **Input:** a connected, weighted, undirected graph. We assume for simplicity that all edge weights are distinct.
- **Output:** a spanning tree where the sum of the edge weights is as small as possible.

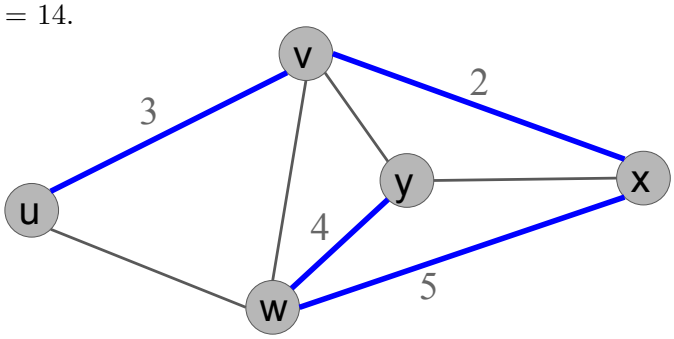
This problem can arise in many contexts. Picture a communication network, such as a network of computers. We want to ensure that the network is connected so that every node can send a message through the network to every other node. However, maintaining all of the links (edges) may be costly. We want to reduce the number of links to the minimum-cost set needed in order to maintain communication.

Exercise 12.2. Give a minimum spanning tree of this graph.



Solution.

There is only one minimum spanning tree, shown here in blue. Its total weight is 2+3+4+5 = 14.



12.2.1 The Reverse-Deletion Algorithm

It is worth noting the following point. Can you prove it?

Observation 12.1. *The minimum-cost set of edges that keep the graph connected will always form a tree.*

> **Proof.**
>
> Suppose we have a set of edges where the graph is connected, but not a tree. Then there is a cycle. If we delete one edge $\{u, v\}$ from the cycle, the graph is still connected, because we can still get from every node on the cycle to every other node. And deleting the edge has reduced the total cost of this set of edges. We can repeat this argument as long as the graph has cycles until we end with a connected, acyclic graph: a tree.

This observations gives an idea for our first greedy algorithm for MST.

The Reverse-Deletion Algorithm:

- Sort the edges from largest weight to smallest.
- Go through the list, deleting each edge unless doing so disconnects the graph.

For correctness, as with Dijkstra's algorithm, we need a key fact about the problem. Here is the fact:

Proposition 12.2. *For any simple cycle in the graph, the maximum-weight edge in the cycle is not part of* any *minimum spanning tree.*

Proof. Let $v_1, \ldots, v_k = v_1$ be a simple cycle and suppose without loss of generality that $\{v_1, v_2\}$ is the maximum-weight edge in the cycle. Suppose we have a spanning tree T containing $\{v_1, v_2\}$. We prove it is not a *minimum* spanning tree.

Delete $\{v_1, v_2\}$ from the tree, leaving us with two disjoint trees, say T_1 containing v_1 and T_2 containing v_2. Adding any edge between a vertex in T_1 and a vertex in T_2 will connect the trees into a spanning tree again. And in the original graph, there is a path $v_2, \ldots, v_{k-1}, v_1$. Since the start is in T_2 and the end is in T_1, at least one of these edges crosses over. And it must have smaller weight than $\{v_1, v_2\}$, so adding it results in a spanning tree with smaller weight than T, so T could not have been a MST. □

Theorem 12.1. *The Reverse-Deletion algorithm correctly produces a MST.*

Proof. Suppose that deleting an edge does not disconnect the graph. Then that edge must have been part of a simple cycle, because there is still a path between its endpoints. So by the Proposition, that edge cannot have been part of any minimum spanning tree. Therefore, if we delete it, a minimum spanning tree of the resulting graph is also a MST of the original graph. Now we just need to note that Reverse Deletion continues until the set of remaining edges is a spanning tree, because otherwise, by definition, there is a cycle and some edge could be deleted. Since it stops at a spanning tree, that spanning tree is a MST of the original graph. □

12.3 Kruskal's and Prim's algorithms

We will look at two other correct greedy algorithms for minimum spanning tree. They both rely on the following key fact.

Proposition 12.3. *Let R, S be any* cut *in the graph, meaning a partition of the graph into two disjoint sets of vertices. Let $\{u, v\}$ be the minimum-cost edge that crosses the cut. Then* every *MST of the graph contains $\{u, v\}$.*

Proof. Consider any spanning tree of the graph that does not contain e. We will show it is not a minimum spanning tree.

Suppose without loss of generality that $u \in R$ and $v \in S$. Because it's connected, there is a path in the spanning tree from u to v. This path "crosses" the cut at some point, i.e. includes some edge $\{u', v'\}$ with $u' \in R$ and $v' \in S$. Let us delete $\{u', v'\}$ and add $\{u, v\}$. By assumption, the total cost of the edges has decreased. Is it still a spanning tree? It is still connected, because there is now a path from u' back to you, across $\{u, v\}$ to v, and then to v'. So any prior path that used $\{u', v'\}$ can be transformed into a path that uses $\{u, v\}$. Similarly, there are no cycles, because if there is a cycle including $\{u, v\}$ now, we could transform it into a cycle that uses $\{u', v'\}$ in the original graph. (The cycle might not be *simple*, but we can turn it into a simple cycle from there.) □

This fact suggests an algorithm that is a sort of mirror to the Reverse Deletion algorithm.

Kruskal's algorithm:

- Sort the edges from smallest weight to largest.
- Go through the list, adding each edge to the graph unless doing so creates a cycle.

Theorem 12.2. *Kruskal's algorithm outputs a minimum spanning tree.*

Proof. When we add an edge $\{u, v\}$, let R be the set of vertices reachable from u along edges added so far, and let S be the remaining vertices. Note that adding any edge that crosses the cut would not create a cycle, since currently no edges cross the cut. Therefore, $\{u, v\}$ must be the minimum-weight edge that crosses this cut, since we are adding edges in order. So by the Proposition, every minimum spanning tree contains $\{u, v\}$, so we are safe to add it to our solution. Now we just need to show that Kruskal's produces a spanning tree. The output doesn't have cycles by definition. But it is a tree, since if it were disconnected at the end, there would be some edge that could have been added (as the original graph was connected), a contradiction. So Kruskal's produces a spanning tree, and it is minimum because it only contains edges that are in every MST. □

A small complaint about Reverse-Deletion and Kruskal's is that they are not very efficient, at least if implemented in a straightforward way. First, sorting the edges at the beginning takes $O(m \log(m))$ time, and m can be quite a bit larger than n. Second, checking in every loop for whether a cycle has been created takes a significant amount of time. A straightforward approach is to use BFS or DFS each time to check for a cycle. However, doing this every loop is a somewhat slow process. (For Kruskal's, the time complexity can be improved significantly with the union-find data structure, but we won't cover that here.)

Can we do better? Yes: here is Prim's algorithm.

Prim's algorithm:

- Pick any vertex u, and add it to our set R.
- Add the minimum-cost edge $\{u, v\}$ attached to u to our solution, and add its other endpoint v to R.
- Continuing, add the minimum-cost edge leaving R to our solution and add its other endpoint to R.
- Stop when $R = V$, the set of all vertices.

Theorem 12.3. *Prim's algorithm outputs a minimum spanning tree.*

Proof. At every step, we add the minimum-cost edge that crosses a cut in the graph, where the cut is R and $V \setminus R$. So by the Proposition, every edge we add must be part of every minimum spanning tree. And we certainly produce a spanning tree, because we never create a cycle and we eventually connect all vertices. □

12.3.1 Running time analysis of Prim's algorithm

To analyze running time, we should have a more precise definition of the algorithm. In fact, it will look extremely similar to Dijkstra's algorithm. As with Dijkstra's, we will use a Priority Queue.

Algorithm 12.1 (Prim's algorithm)**.**

```
prim(G):
   # G = (V,E) is a weighted, undirected, connected graph
   Q = new Priority Queue
   for all vertices u:
      best_weight[u] = infinity
      best_edge[u] = null
      marked[u] = false
      Q.insert(u, infinity)
   let s be any vertex of G
   Q.update(s, 0)
   let F = empty list       # edges of the spanning tree
   while Q.size() > 0:
      u = Q.pop_smallest()
      marked[u] = true
      F.add(best_edge[u])   # does nothing if best_edge[u] is null
      for v in N(u):
         if not marked[v] and wt(u,v) < best_weight[v]:
            best_weight[v] = wt(u,v)
            best_edge[v] == {u,v}
            Q.update(v, wt(u,v))
   return (V, F)
```

To understand the pseudocode, see if you can answer this question (you will want to take another look at Dijkstra's).

Exercise 12.3. What is the key difference between Prim's and Dijktra's algorithms? How do these differences result in one solving MST while the other solves shortest paths?

i Solution.

The key difference is that Dijkstra's tracks the smallest total distance from the source to a given vertex. Prim's tracks the smallest edge to the vertex from any previously-visited vertex. This means that Dijkstra keeps popping vertices with smallest distance from the source, but Prim keeps popping vertices that have the smallest edge to the current connected component. So Dijkstra is finding paths from s that are as short as possible, while Prim is connecting a tree as cheaply as possible.

The running time of this implementation is essentially identical to Dijksta's. If we use a Fibonacci heap, we obatin the following analysis, which is the same as in Dijkstra's.

Operations	Total time with F. heap
$O(n)$ calls to Q.insert()	$O(n)$
$O(n)$ calls to Q.pop_smallest()	$O(n \log(n))$
$O(m)$ calls to Q.update()	$O(m)$

The result is a running time of $O(m + n \log(n))$, just as with Dijkstra's algorithm. We also should note that we have $O(n + m)$ operations otherwise, but this will be dominated in big-O by the complexity of the heap operations.

Part V

Topic E: Max Flow

13 The Max Flow Problem

This section introduces the max flow problem.

Objectives. After learning this material, you should be able to:

- State the format of an input to the max flow problem.
- Define when a function is a valid flow in the graph.
- Define the amount of flow and the max flow.

13.1 Motivation and input

We have already looked at finding paths in a graph. What if we want to route a large amount of traffic across a graph, such as packets across a computer network? In this case, the traffic may need to take multiple routes at once. Each edge of the network can only handle so much traffic at once – the *capacity* of that edge. We'd like to figure out how to route the traffic so as to get the most possible across.

In the max flow problem, the **input** consists of:

- A directed graph $G = (V, E)$,
- a *source* $s \in V$ and *sink* $t \in V$, and
- a *capacity function* $c : E \to \mathbb{R}_{\geq 0}$.

We can also think of the capacity function as nonnegative edge weights. We extend c to be a function on all pairs of vertices, by defining $c(u, v) = 0$ if $(u, v) \notin E$. We will also make the following simplifying assumption, which isn't necessary, but makes our lives a bit easier:

> **Assumption: no anti-parallel edges.** We assume that if $(u, v) \in E$, then $(v, u) \notin E$. In other words, there is only one edge between any pair of vertices.

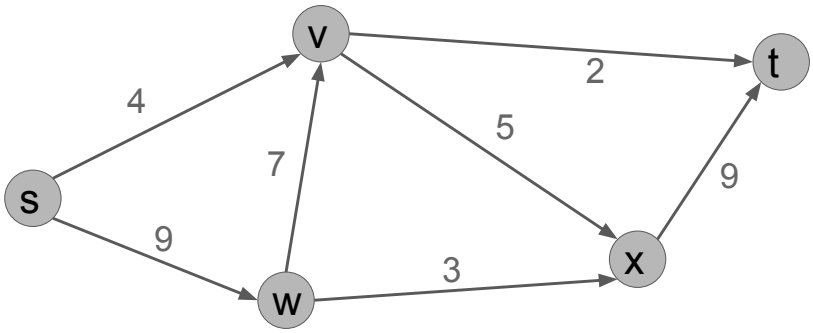


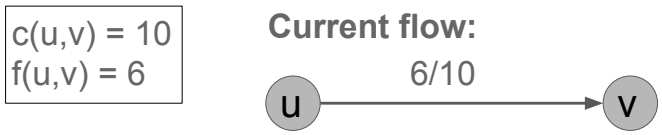
Figure 13.1

13.2 Valid flows

Given the input, a potential solution is any method of routing flow from s to t. But what does that mean, exactly? A function $f : V \times V \to \mathbb{R}_{\geq 0}$ is called a flow, and the flow is **valid** if:

- **(capacity constraint)** $f(u, v) \leq c(u, v)$ for all $u, v \in V$. This implies that if there is no edge (u, v), then $f(u, v) = 0$.
- **(flow constraint)** for all vertices $v \notin \{s, t\}$, $\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$.

We illustrate the current flow compared to the capacity of the edge using the notation "flow/capacity", e.g. 6/10.



Exercise 13.1. On the example graph of Figure 13.1 above, which of the following flows are valid? If invalid, indicate whether the flow or capacity constraint is violated. From now on, we will label each edge (u, v) with a label of the form 5/9, where here 5 represents $f(u, v)$ and 9 represents $c(u, v)$. We have also put the flows in blue here for added emphasis.

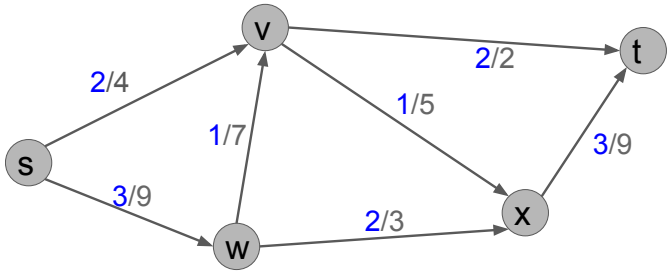


Figure 13.2: Part a.

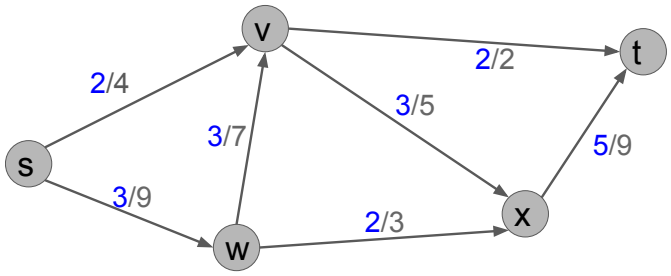


Figure 13.3: Part b.

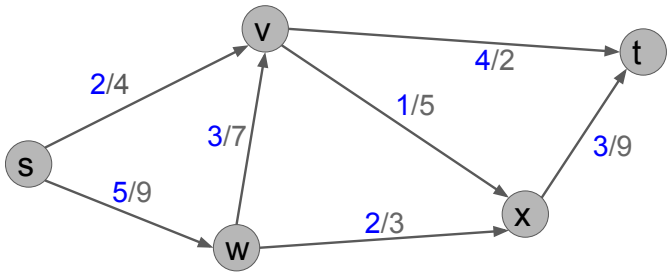


Figure 13.4: Part c.

Solution.

- Part a: valid.
- Part b: invalid. The flow constraint at w is violated: the total flow in is 3, but the total flow out is 5.
- Part c: invalid. The capacity constraint on (v, t) is violated: $c(v, t) = 2$, but $f(v, t) = 4$.

13.3 Max flow

Given a flow f, we define the **amount** of flow, written $|f|$ for short, to be the following quantity:

$$|f| = \sum_{v \in V} f(v, t) - \sum_{w \in V} f(t, w).$$

In other words, the total amount of flow $|f|$ is the net flow into the sink t.

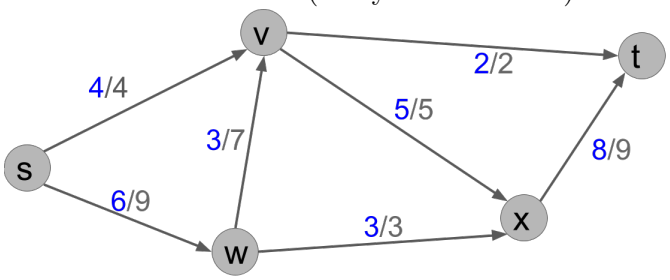
In the max flow problem, the **output** is:

- A valid flow f whose amount is maximized, over all possible valid flows.

Exercise 13.2. On the example graph of Figure 13.1, what is the max flow amount and what is the flow?

Solution

Here is a max flow f. The amount of $|f| = 10$. There are other slightly different flows that also have value 10 (can you find them?).



14 Ford-Fulkerson

This section describes the Ford Fulkerson framework for solving max flow.

Objectives. After learning this material, you should be able to:

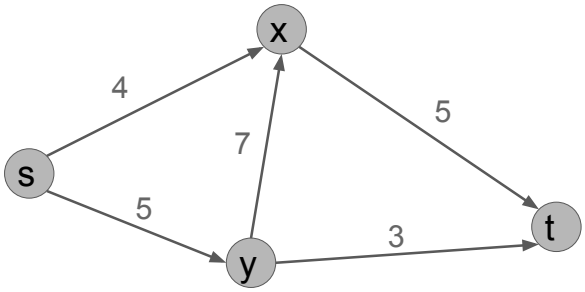
- Given a flow f, construct the residual graph G_f and residual capacity r_f.
- Find an augmenting path and use it to augment f.
- Use the above steps to run the Ford Fulkerson framework on examples.

14.1 Breaking and fixing the natural approach

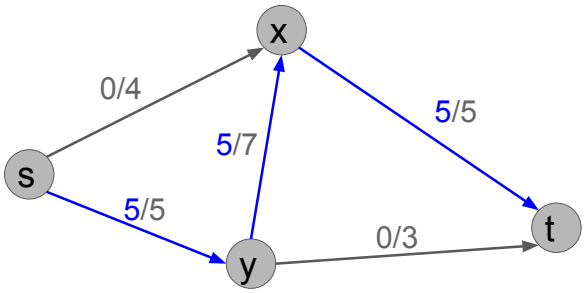
Here's a very natural algorithm that almost works:

- Find a path from s to t.
- Send as much flow along that path as possible.
- Repeat until impossible.

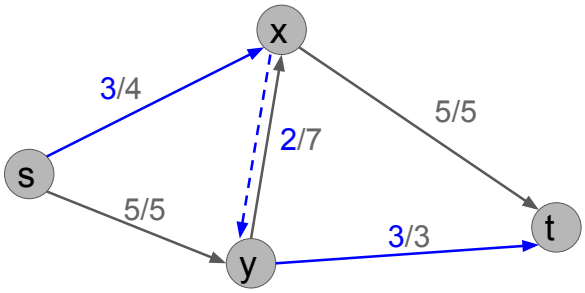
Unfortunately, this doesn't always work to find the max flow! We need the ability to "cancel" use of an edge and backtrack on some decisions. Luckily, it turns out that we can achieve this by pretending to send flow "backward" along that edge, which has the effect of cancelling out the forward flow. Here's an example:



We first send 5 units of flow along the path s, y, x, t. This results in the following flow.



Now, we are stuck, because there is no path from s to t along which we can send more flow. But we have not found the max flow yet. The way out is to send 3 units of flow along the path s, x, y, t. Even though (x, y) is not an edge in the graph, we can "send" flow along this edge by reducing the current amount of flow.



Another way to think of what happened is that we took 3 units of flow along (y, x) and re-routed it to (y, t). To replace that 3 units of flow at x, we brought over 3 units of flow along (s, x).

This idea leads to the Ford-Fulkerson framework for max-flow algorithms. First, we need to make some useful definitions in order to fully define the algorithm.

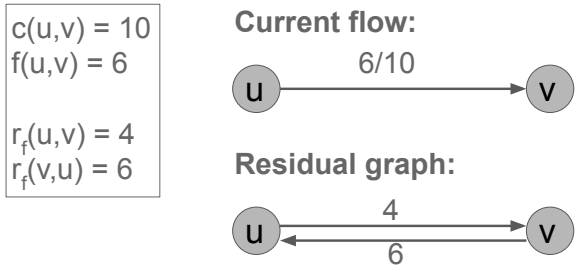
14.2 Residual capacity function and residual graph

The **residual capacity function** $r_f : V \times V \to \mathbb{R}_+$ is defined as follows:

- If $(u, v) \in E$, then $r_f(u, v) = c(u, v) - f(u, v)$. This is the amount by which we could increase the flow along this edge without breaking the capacity.
- Otherwise, if the reverse edge $(v, u) \in E$, then $r_f(u, v) = f(v, u)$. We can "increase" the flow from u to v by decreasing the flow from v to u. The current flow is $f(v, u)$, so that is how much we can decrease it.
- If neither (u, v) nor (v, u) is an edge, then $r_f(u, v) = 0$.

The **residual graph** $G_f = (V, E_f)$ is a graph where there is an edge $(u, v) \in E_f$ if $r_f(u, v) > 0$, otherwise $(u, v) \notin E_f$. It is important that if $r_f(u, v) = 0$, then the edge (u, v) is NOT in E_f.

We will draw r_f as edge weights on G_f, and we'll sometimes refer to both together as the "residual graph".

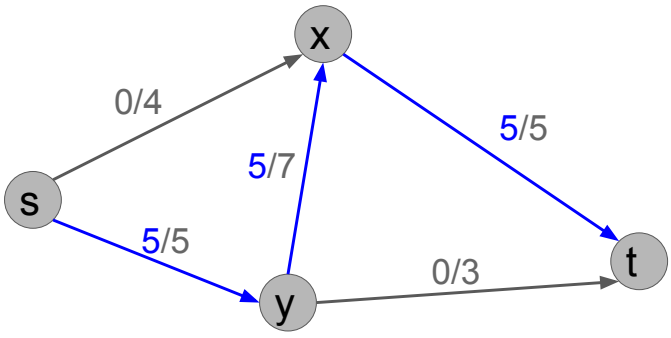


We can describe these operations more formally as follows.

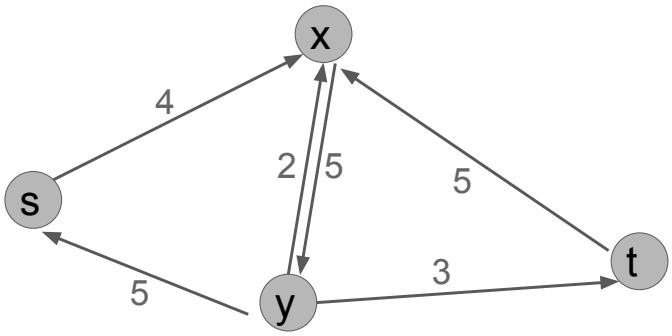
```
residuals(G, c, f):
   # G = (V,E) is a directed graph, c is a capacity function, f is a flow
   let Ef be an empty set
   for each edge (u,v) in G:
      set rf(u,v) = c(u,v) - f(u,v)
      if rf(u,v) > 0:
         add (u,v) to Ef
      set rf(v,u) = f(u,v)
      if rf(v,u) > 0:
         add (v,u) to Ef
   return rf, (V,Ef)
```

Exercise 14.1. Consider the following input graph and flow. Draw the residual graph. To do so, follow these instructions:

1. For the residual capacity function, calculate the residual capacity $r_f(u, v)$ of each edge (u, v) as well as its reverse (v, u).
2. For the residual graph, draw a copy of the graph's vertices. For every pair (u, v) where $r_f(u, v) > 0$, draw an edge from u to v. Label the edge with its residual capacity.
3. Important: if $r_f(u, v) = 0$, then edge (u, v) should NOT be included in the graph.



Solution.



Notice the differences between which edges are present in the residual graph, compared to the original graph:

- Even if there is an edge such as (s, y) in the original graph, it may not be present in the residual graph. This happens when flow equals capacity.
- Even if an edge such as (y, s) does not exist in the original graph, it may be present in the residual graph. This happens when the reverse direction (s, y) has flow.
- Even though the original graph has no antiparallel edges, the residual graph can have antiparallel edges such as (x, y) and (y, x). This happens when the flow is greater than zero but less than capacity.

14.3 Augmenting paths

The last concept we need is an **augmenting path**: a path $s = v_1, \ldots, v_k = t$ from s to t in the residual graph. Note that because this is a path in the residual graph, we have that $r_f(v_i, v_{i+1}) > 0$ for every edge along this path. In other words, we can send more flow along every edge in this path.

Algorithm 14.1 (Augmenting a flow along a path)**.**

```
augment(G, f, rf, path):
   # G is the input graph, f is a flow, rf is residual flow
   # path goes from s to t in the residual graph
   let d = min( rf(u,v) for (u,v) in path )
   for each edge (u,v) in path:
      if (u,v) in G:
         f(u,v) += d
      else:              // here, (v,u) must be in G
```

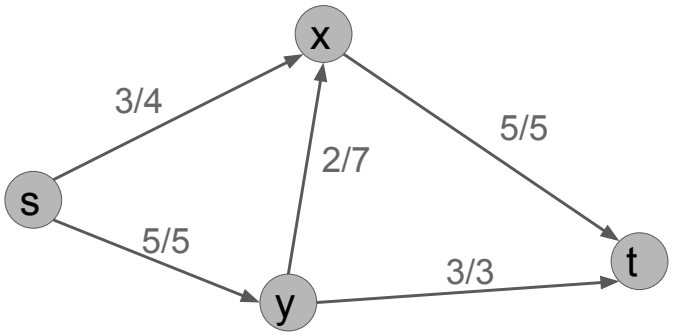
```
            f(v,u) -= d
    return f and d
```

Exercise 14.2. Find an augmenting path in the residual graph of the previous exercise. Then, augment the flow along the path. To do so, follow these instructions:

1. Give a path from s to t in the residual graph.
2. Give the amount we can augment, d.
3. Give the updated flow f after augmenting. To do so, draw the graph and label each edge with flow/capacity.

Solution.

1. s, x, y, t. *Note: this is the only path.*
2. The amount is 3. *Note:* $r_f(s,x) = 4, r_f(x,y) = 5, r_f(y,t) = 3$, *so the minimum is* 3.
3. Here is the updated flow:



14.4 Ford-Fulkerson Framework

The **Ford-Fulkerson Framework** is the following algorithmic framework:

Algorithm 14.2.

```
fordfulkerson(G, c):
    # G is a directed graph, c is a capacity function
    let f(u,v) = 0 for all vertex pairs u,v
    repeat:
        let Gf and rf = residuals(G, c, f)
        find a path from s to t in Gf
        let f and d = augment(G, f, rf, path)
```

```
    if d == 0:
        return f
```

Notice that line 6, where we find a path, is underspecified. How do we choose the path? There turn out to be several reasonable choices for this step. Because of this, Ford-Fulkerson is called a framework rather than an algorithm. To make it an algorithm, we need to specify how to implement line 6.

Exercise 14.3. Run the Ford-Fulkerson framework on the example graph of Figure 13.1, reproduced here. You can use any method to find a path in the residual graph.

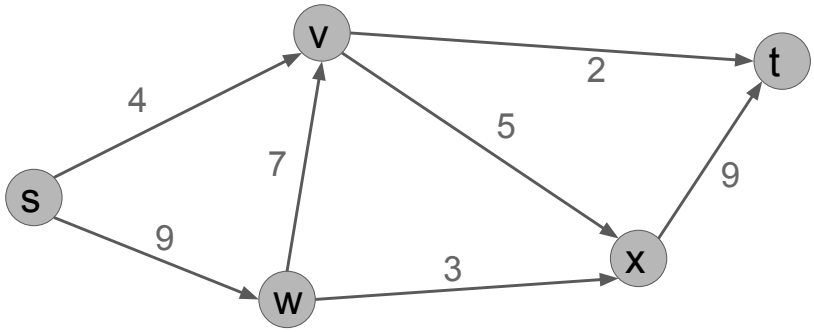


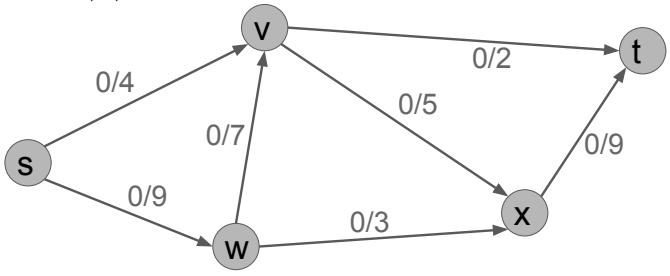
Figure 14.1

In each round, give the following things:

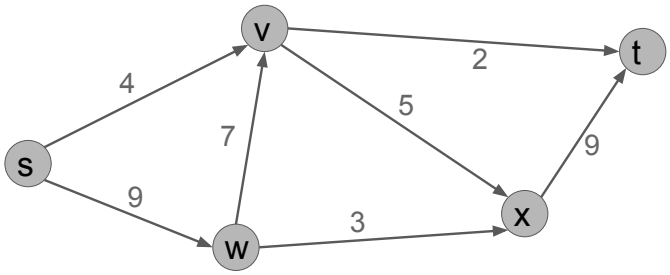
- The flow f (draw this on a graph, as above), and the amount $|f|$.
- The residual flow r_f, G_f, drawn as a graph (see above).
- The augmenting path you picked, the augmentation amount d.

i Example solution.

There are multiple possible answers, depending on the paths chosen. Here's one.
First round:
Flow: $|f| = 0$.

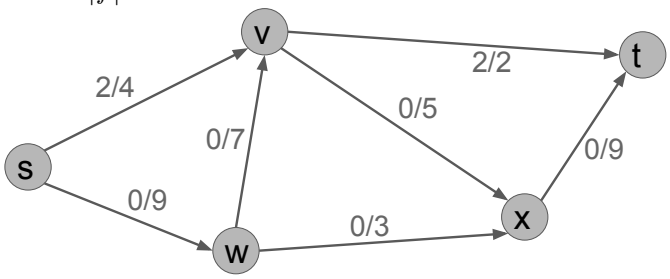


Residual flow:

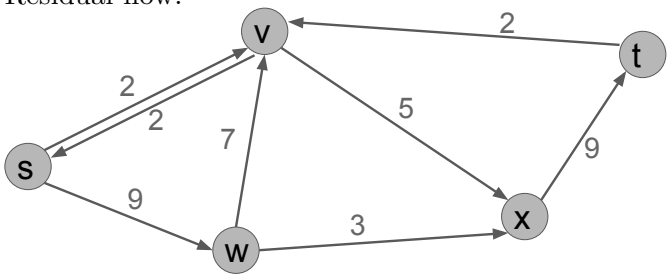


Path: s, v, t. Amount: 2.

Second round:
Flow: $|f| = 2$.

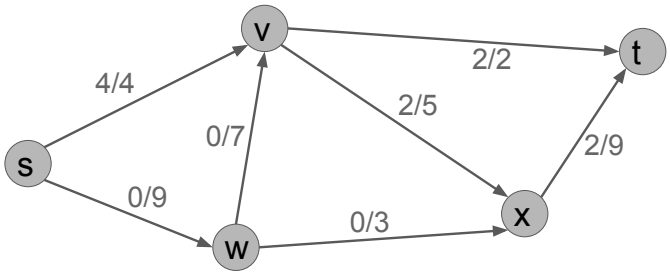


Residual flow:

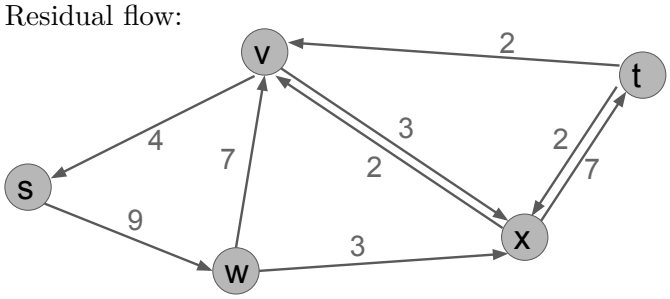


Path: s, v, x, t. Amount: 2.

Third round:
Flow: $|f| = 4$.



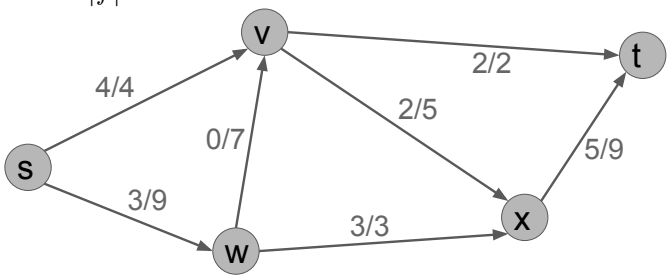
Residual flow:



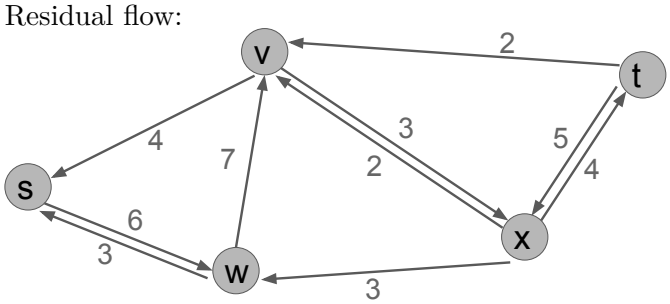
Path: s, w, x, t. Amount: 3.

Fourth round:

Flow: $|f| = 7$.



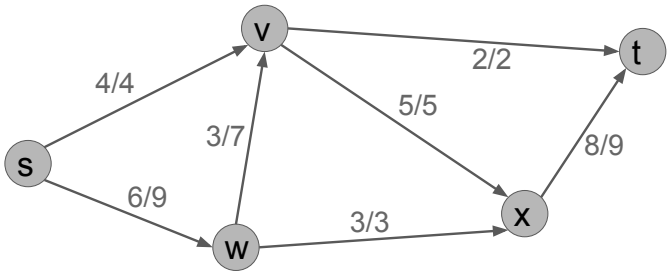
Residual flow:



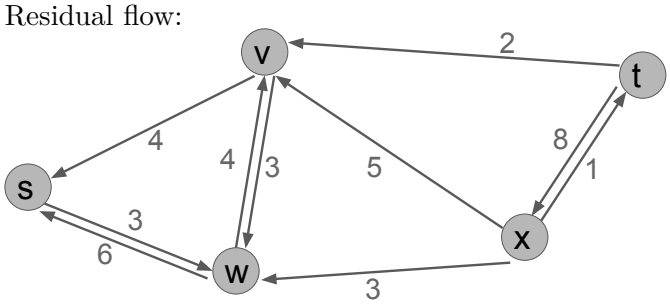
Path: s, w, v, x, t. Amount: 3.

Fifth round:

Flow: $|f| = 10$.



Residual flow:



Path: no path exists from s to t. Stop.

Final amount of flow: $|f| = 10$.

15 Correctness and Min Cut

This section proves Ford Fulkerson correct using the concept of the min $s-t$ cut.

Objectives. After learning this material, you should be able to:

- Define an $s-t$ cut (S,T) in the graph and find the value of the cut.
- Explain why Ford Fulkerson stops when it find both a max flow and a min cut.
- Use Ford Fulkerson to find a min $s-t$ cut in the graph.

15.1 Min cut

To prove that Ford Fulkerson is correct, we'll need a non-obvious, very mathematically nice strategy. The idea is that the maximum flow is limited by *bottlenecks* in the graph, where no more flow can squeeze through. If there is a bottleneck where the total capacity to squeeze through is 10, then the max the flow could possibly be is 10. The tricky part is that a bottleneck could contain multiple edges.

A $s-t$ **cut** in a directed graph $G=(V,E)$ is a partition of the vertices into two sets, S and T, such that $s \in S$ and $t \in T$.

Given a capacity function c, the **capacity** of an $s-t$ cut (S,T) is the total capacity of edges that cross from S to T, which is

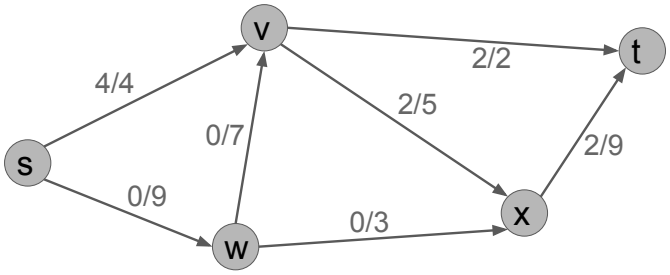
$$K(S,T) = \sum_{u \in S, v \in T} c(u,v).$$

Here, we only sum over pairs $(u,v) \in E$ such that $u \in S, v \in T$. Meanwhile, given a flow f, the **flow** across an $s-t$ cut (S,T) is the total *net* flow from S to T, which is

$$F(S,T) = \sum_{u \in S, v \in T} (f(u,v) - f(v,u)).$$

The **Min $s-t$ Cut problem** is, given an input graph to max flow, to find the $s-t$ cut that minimizes $K(S,T)$. This will turn out to be highly related to the max flow problem.

Exercise 15.1. Consider the following input graph and flow. Let $S = \{s, w, x\}$ and $T = \{v, t\}$. What is $K(S,T)$? What is $F(S,T)$? And what is the minimum $s-t$ cut in the graph and what is its capacity?



Solution.

$K(S,T) = 4 + 7 + 9 = 20$.
$F(S,T) = 4 + 0 - 2 + 2 = 4$.
The min cut is $S = \{s, w, v\}, T = \{x, t\}$ and its value is $2 + 5 + 3 = 10$.

It turns out that the flow across the cut is the same for any $s-t$ cut we choose! It's just the amount of flow.

Lemma 15.1. *Let f be a flow. Then for any $s-t$ cut (S,T), we have $F(S,T) = |f|$.*

Proof. Let us start with the cut $S = V\ \{t\}, T = \{t\}$. Here we have $F(S,T) = |f|$ by definition of the amount of flow: the net amount of flow into t. Now let us consider any $u \in V, u \neq s, u \neq t$. We will move u from the "S" part of the cut to the "T" part. But we claim $F(S,T)$ has not changed: the amount we've added is $\sum_{v \in S}(f(v,u) - f(u,v))$ and the amount we've subtracted is $\sum_{v \in T}(f(u,v) - f(v,u))$. By the net flow constraint, these add up to zero.

Now we repeat with any other vertex, and so on, and we can create any $s-t$ cut without changing the amount of flow across the cut. □

But we also know that flow across the cut is upper-bounded by the capacity of the cut.

Lemma 15.2. *Let f be a valid flow and (S,T) an $s-t$ cut. Then $F(S,T) \leq K(S,T)$.*

Proof. We have $F(S,T) = \sum_{u \in S, v \in T} f(u,v) - f(v,u) \leq \sum_{u,v} f(u,v) \leq \sum_{u,v} c(u,v) = K(S,T)$. □

Now we can prove that Ford-Fulkerson is correct.

Theorem 15.1. *Let f be a valid flow and G_f the residual graph. Then f is a max flow if and only if there is no path from s to t in G_f. Furthermore, in this case, the following cut is a min s-t cut: $\$S = \$$ the set of vertices reachable from s, and $T = V \setminus S$.*

Proof. If there is a path in G_f, then we can augment the flow along this path, so f could not be a max flow. On the other hand, suppose there is no path. Let S, T be defined as in the lemma statement. Then for every edge (u, v) with $u \in S, v \in T$, we have $f(u, v) = c(u, v)$, as otherwise (u, v) would be an edge in G_f. And we have $f(v, u) = 0$ for the same reason. So $F(S, T) = K(S, T)$.

But we know that, for all possible $s-t$ cuts (S', T'), we have $K(S, T) = F(S, T) = F(S', T') \leq K(S', T')$. This proves that (S, T) is a min cut. So we know that, for any flow f', we have $|f'| \leq K(S, T) = F(S, T) = |f|$. This proves that f is a max flow. □

Corollary 15.1. *For any algorithm following the Ford-Fulkerson framework, if it halts, then it is correct.*

This corollary follows because FF only halts if there is no path from s to t in the residual graph.

Corollary 15.2 (Max-Flow Min-Cut Theorem)**.** *In any graph, the max $s - t$ flow is equal to the min $s - t$ cut.*

The corollary follows because Ford-Fulkerson, assuming it halts, finds a max flow that equals a min cut, every time.

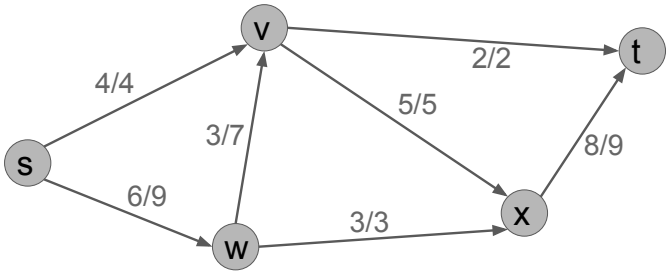
15.2 Using Ford-Fulkerson to find a min cut

Knowing what we now know, finding a min cut in a graph is straightforward:

1. Use a Ford-Fulkerson algorithm to find a maximum flow f.
2. Let G_f be the residual graph.
3. Let $S =$ the set of vertices reachable from s in G_f; we can use e.g. BFS to find this set. Let $T = V \setminus S$.
4. Return (S, T).

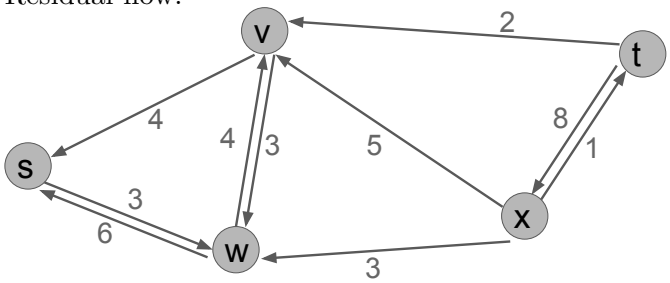
To recap, we know that (S, T) is a min cut because for every edge from S to T, the flow equals the capacity, and there are no reverse edges with flow (otherwise T would be reachable), so $F(S, T) = K(S, T)$.

Exercise 15.2. Here is a graph with a flow. Actually, this flow is already a max flow. Find the min cut by running an iteration of Ford Fulkerson starting from here, drawing the residual flow, and using it to find a min cut.



Solution.

Residual flow:



We have $S = \{s, v, w\}$ and $T = \{x, t\}$. The value of the min cut is 10.

16 Implementing Ford-Fulkerson and Running Time

This section discusses an implementation of Ford-Fulkerson called Edmonds-Karp and analyzes its running time.

Objectives. After learning this material, you should be able to:

- Implement the Edmonds-Karp algorithm on examples.
- Identify the time complexity of Edmonds-Karp.

16.1 Using Breadth-First Search

The part of Ford-Fulkerson that is not fully specified is how to choose a path from s to t in the residual graph, G_f. We will simply use BFS. The resulting algorithm is called Edmonds-Karp.

Algorithm 16.1.

```
edmondskarp(G, c):
   # G is a directed graph, c is a capacity function
   let f(u,v) = 0 for all vertex pairs u,v
   repeat:
      let Gf and rf = residuals(G, c, f)
      use BFS to find a path from s to t in Gf  #<<
      let f and d = augment(G, f, rf, path)
      if d == 0:
         return f
```

Because Edmonds-Karp is in the Ford-Fulkerson framework, we know that it is correct as long as it halts. We will show that it halts and give a bound on the number of time steps.

Proposition 16.1. *On a graph with n vertices and m edges, Edmonds-Karp runs in time $O(nm^2)$.*

Proof (optional).

In a round of Edmonds-Karp, an edge is called *critical* if we augment along a path containing that edge, and the residual capacity is equal to the augmentation amount. In other words, that edge has the minimum residual capacity of any edge along the path. Note that a critical edge is present in G_f in the current round, but disappears in the next round, because the edge's capacity is fully saturated.

First, we show that in each round, a vertex's distance from s in the residual graph G_f can only increase. (This is the unweighted distance, i.e. number of hops.) To see this, consider the change in the residual graph. When we augment along a path $s = v_1, \ldots, v_k = t$, we may add new reverse edges of the form (v_j, v_{j-1}) to G_f, and we remove forward edges if they are critical edges. Because $v_1, \ldots, v_k$ is a shortest path from s, adding reverse edges cannot make any paths shorter, since we would never go through v_{j+1} to get to v_j. And eliminating edges cannot make any paths shorter.

Next, we show that each edge (u, v) can only be critical in at most $n/2$ iterations. When an edge is critical, it is removed from the residual graph by augmenting. To become critical again, it must be added back into the flow graph by augmenting along the reverse direction, (v, u). But that implies the distance from s to u has increased by at least 2, since it was strictly less than the distance to v, and now it's strictly more, and the distance to v has not decreased. Since the distance begins at zero or more, and ends at n or less, and increases by at least 2 each round, the total number of times the edge is critical is at most $n/2$.

Finally, we put the pieces together. There are up to $2m$ potential edges in the residual graph: each edge in the original graph, and its reverse. Each can be critical at most $n/2$ times, and every round there is at least one critical edge, so there are at most $(2m)(n/2) = mn$ iterations. In each iteration, we do a constant amount of work and run BFS, which takes $O(m + n)$ time. We can assume that $m \geq n - 1$ here, as the graph can be assumed to be connected, so the running time of BFS is $O(m)$. Putting it all together, we have a running time of $O((mn)(m)) = O(nm^2)$.

17 Reductions and Applications

This section discusses how we can solve other problems by turning them into max-flow problems.

Objectives. After learning this material, you should be able to:

- Construct reductions from related problems or variants to max flow.
- Identify failures of incorrect reductions.

17.1 Variants of max flow

First, we will look at other versions of the max flow problem. It would be nice if we didn't have to come up with a new algorithm every time we changed the problem slightly! As we will see, often we can solve a slightly different problem by "reducing" it to max flow.

17.1.1 Antiparallel edges

A pair of antiparallel edges are (u, v) and (v, u), i.e. two directed edges between two vertices pointing opposite directions. Remember that we solved max flow assuming that the input graph contains no antiparallel edges.

Actually, one can modify our solution to account for antiparallel edges, but here is a method using a reduction.

- We take as input a flow instance G_1, c_1 that may contain antiparallel edges.
- We use it to construct a new flow instance G_2, c_2 that does not contain any antiparallel edges.
- We run our max flow algorithm, such as Edmonds Karp, on G_2, c_2. It produces some output flow f_2.
- We use f_2 to construct a solution f_1 for the original instance.

Here's how we construct G_2 and c_2.

- G_2 will contain all of the vertices of G_1, plus more vertices defined below.
- For every edge (u, v) in G_1, we can create a new vertex, call it $w_{u,v}$. Note that $w_{u,v}$ is different from $w_{v,u}$.

- Then, we create two new edges: $(u, w_{u,v})$ and $(w_{u,v}, v)$.
- We make the capacity of these new edges equal to the old capacity, i.e. $c_2(u, w_{u,v}) = c_2(w_{u,v}, v) = c_1(u, v)$.

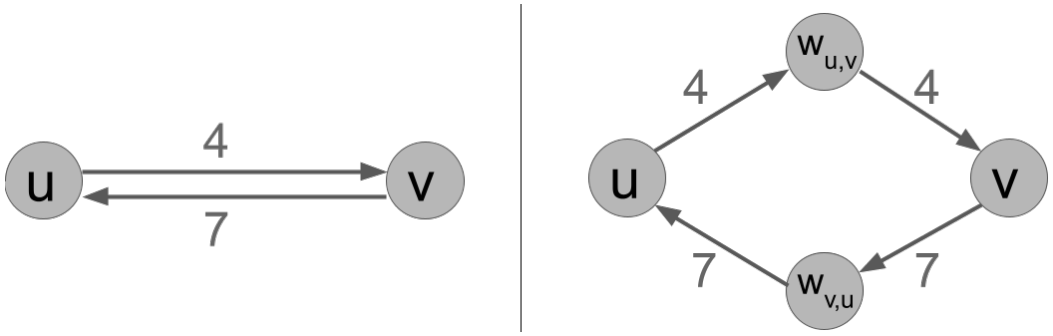


Figure 17.1: Left: an edge in the original graph, G_1. Right: the result in G_2 after the reduction.

Now, we need to explain how to convert f_2 into f_1.

- For each vertex of the form $w_{u,v}$, we let $f_1(u, v) = f_2(u, w_{u,v})$. Note by the net flow constraint, this equals $f_2(w_{u,v}, v)$.
- This defines the flow on every edge of the original graph G_1.

Proposition 17.1. *The above reduction is correct, i.e. f_1 is a maximum valid flow for G_1, c_1.*

Proof. First, we need to prove that G_2 has no antiparallel edges. That will tell us that f_2 is a correct output for G_2, c_2. Then, we need to prove that, since f_2 is correct, f_1 is correct.

To show G_2 has no antiparallel edges, notice that every edge in G_2 is either of the form $(u, w_{u,v})$ or $(w_{u,v}, v)$. In either case, there is no antiparallel edge. For example, if originally there was an antiparallel edge (v, u), then there will be an edge $(v, w_{v,u})$, but $w_{v,u}$ is a different vertex than $w_{u,v}$.

Now, we know f_2 is a max flow for G_2, c_2. Notice that the total amount of flow satisfies $|f_2| = |f_1|$. So we just need to show that the min $s-t$ cut the graphs are equal. This will imply that f_1 is a max flow for G_1, c_1. (Actually, we are using the fact that the max-flow min-cut theorem applies for graphs with antiparallel edges, but this turns out to be true.)

Let S_1, T_1 be an $s-t$ cut in G_1, c_1. The capacity of the cut is $K(S_1, T_1) = \sum_{u \in S_1, v \in T_1} c(u, v)$. Now consider the following cut in G_2, c_2: We first let $S_2 = S_1, T_2 = T_1$. Then, we add all vertices of the form $w_{u,v}$ to the same element as u. Now, we claim that $K(S_2, T_2) = K(S_1, T_1)$. This follows because the only edges crossing the cut are of the form $(w_{u,v}, v)$ and they cross the cut if and only if (u, v) crosses the cut S_1, T_1 in the original graph. And they have the same capacity, i.e. $c_2(w_{u,v}, v) = c_1(u, v)$. □

17.1.2 Multiple sources and sinks

Suppose we have a max flow problem, but there are multiple possible sources of flow and multiple possible sinks. The new goal is to maximize the total flow into all of the sinks. The constraints on a flow are the same, except that now the net flow constraint does not apply to *any* of the sources or sinks.

Exercise 17.1. Give a reduction from max flow with multiple sources and sinks to max flow. Sketch a proof of correctness.

> **i** Example solution.
>
> First we give the reduction. As before, we need two parts:
>
> 1. Given an instance G_1, c_1 of the problem of max flow with multiple sources and sinks, construct an instance G_2, c_2 of max flow.
> 2. Given the solution f_2 to max flow, construct the solution f_1 to max flow with multiple sources and sinks.
>
> For 1, Let G_1, c_1 be the original graph. We construct a new graph G_2 by taking a copy of G_1 and adding a "supersource" s^* and "supersink" t^*. For each source s in G_1, we add an edge (s^*, s) with capacity infinity. (Actually, we can just set the capacity to any large enough number, e.g. the sum of all the capacities of the original graph.) For each sink t in G_1, we add an edge (t, t^*) with capacity infinity as well.
> For 2, given f_2, we simply remove s^*, t^* and all of their edges, and this gives us f_1.
> Now that we have the reduction, we sketch a proof of correctness. G_2, c_2 is a valid instance of max flow, since it has only one source and one sink. Informally, the reduction works because we allowed it to send as much flow out of each source s and as much flow into each sink t as possible, while otherwise respecting all the constraints of a flow. More formally, we again need to know that the max-flow min-cut theorem applies with multiple sources and sinks, which it does (where the cut must have all sources in S and all sinks in T). In this case, we can see that a min cut of G_2, c_2 would never contain any of the "infinity" edges, so a min cut in G_2, c_2 is the same as in G_1, c_1. Since the flow amounts are the same as well, this proves that f_1 is optimal in G_1, c_1.

17.2 Bipartite Matching

Now we will use max flow to solve a problem that initially seems unrelated. A graph $G = (V, E)$ is **bipartite** if the vertices can be partitioned into two sets, V_1, V_2, such that every edge has one endpoint in V_1 and one in V_2.

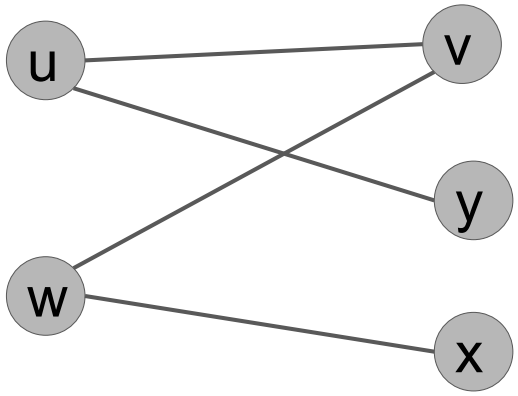


Figure 17.2: Bipartite graph with $V_1 = \{u, w\}$ and $V_2 = \{v, y, x\}$.

A **matching** in a graph is a set of edges where no two edges share an endpoint. The size of the matching is the number of edges.

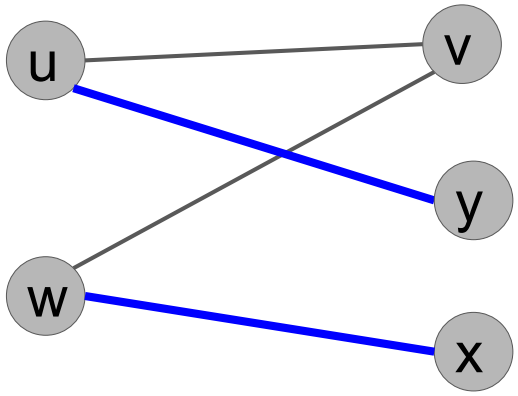


Figure 17.3: The matching $\{\{u, y\}, \{w, x\}\}$ of size two.

The **Maximum Bipartite Matching Problem:**

- **Input:** an undirected bipartite graph G.
- **Output:** a maximum-sized matching in G.

17.2.1 The reduction

To solve maximum bipartite matching, we will reduce it to max flow.

Specifically, let the input be $G = (V, E)$ with V partitioned into V_1, V_2. We create a new graph $G' = (V', E')$ and capacity c' as follows.

- Start with a copy V' of all the vertices V.
- For every edge of the form $\{u, v\} \in E$, where $u \in V_1$ and $v \in V_2$, create a directed edge $(u, v) \in E'$ and set $c(u, v) = 1$.
- Create a new vertex $s \in E'$, the source.
- Create an edge (s, u) for every $u \in V_1$, setting $c(s, u) = 1$.
- Create a new vertex $t \in E'$, the source.
- Create an edge (v, t) for every $v \in V_2$, setting $c(v, t) = 1$.

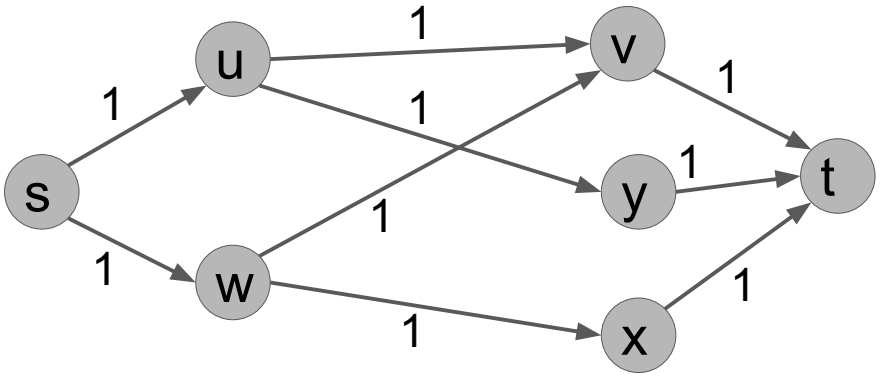


Figure 17.4: The reduction applied to Figure 17.2.

We then solve max flow on G', c' obtaining a flow f'. Given f', we convert it to a matching as follows:

- For each edge $\{u, v\}$ of the original graph, if $f'(u, v) > 0$, then we add $\{u, v\}$ to our matching.

17.2.2 Correctness and the Integrality Theorem

It turns out that this reduction is not always correct unless we use the right kind of max flow algorithm. Can you find the problem?

Exercise 17.2. For the example graph above (Figure 17.2, Figure 17.4), give an example flow f' that is a max flow in the reduction graph G', c', but does **not** yield a valid matching.

Hint: your flow amounts should not all be integers on all edges.

Example solution.

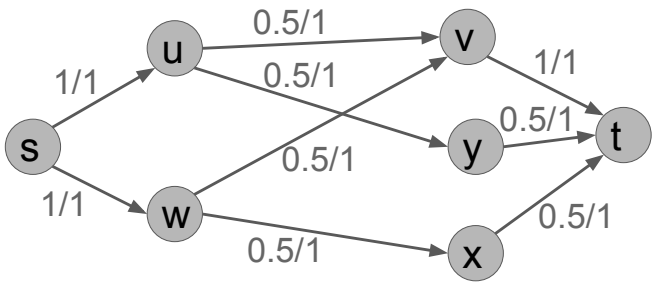


Figure 17.5: A flow of 2, which is the max possible.

According to the reduction, we would try to add all four middle edges, which would not yield a valid matching (the endpoints u, w, and v are all included twice).

Luckily, however, our reduction will be correct when using Ford-Fulkerson algorithms. Specifically, we will need to find a max flow where the flow amounts are 0 or 1. Luckily, when the input **capacities** are all 0 or 1, then so are the output **flows** – if we use a Ford-Fulkerson algorithm. This is a consequence of the following theorem.

Theorem 17.1 (The Integrality Theorem)**.** *A max-flow instance where all capacities are integers always has a solution where all flows are integers. Morever, any Ford-Fulkerson algorithm always finds an all-integer max flow.*

Proof. At the beginning of the algorithm, all residual capacities are integers, because the capacities are integers and the flow is zero everywhere. So regardless of what augmenting path is chosen, the augmentation amount is an integer.

Therefore, after the first step, the flow on every edge is an integer (either zero or the augmentation amount), and the capacities are always integers, so the residual capacities are still integers. So regardless of the augmenting path, the augmentation amount is an integer.

Continuing the argument, at every stage of the algorithm, the flow amount on every edge is always an integer, including when the algorithm halts. We also see that any Ford-Fulkerson algorithm must halt here, because every augmentation amount is at least 1, and we can only augment by 1 a finite number of times until we reach the max flow.)

We know that, when a FF algorithm halts, it must have found a max flow. Since we know the FF algorithm halts with an all-integer flow, we also conclude that an all-integer max flow must exist. □

Proposition 17.2. *When using a Ford-Fulkerson algorithm for max flow, our reduction is correct (i.e. we output a maximum matching).*

Proof. By the Integrality Theorem, all flows are integers. Since the capacities are all 1, this means every edge's flow is either zero or one.

Consider a vertex $u \in V_1$. It has exactly one incoming edge (from s). So the maximum amount of flow through u is 1. Because the flows are integers, this means that at most one edge out of u has flow 1. So at most one edge incident to u is in the matching.

Similarly, for any $v \in V_2$, it has exactly one outgoing edge (to t), so at most one edge incident to v is in the matching.

This proves that the matching we output is indeed a valid matching, i.e. it does not include more than one edge incident to any vertex. Furthermore, we note that the number of edges in the matching is exactly equal to the amount of flow. Now we need to prove that we output the *maximum* matching.

To do so, consider any valid matching in the graph. We can turn it into a valid flow in our flow graph: for every edge $\{u, v\}$ in the matching with $u \in V_1, v \in V_2$, we send one unit of flow along (s, u), one along (u, v) and one along (v, t). All of the capacity and flow constraints are satisfied because every u and v are only in the matching at most once. Furthermore, the amount of flow is exactly the number of edges in the matching.

Because every valid matching corresponds to a flow graph with the same amount of flow and vice versa, the maximum matching corresponds to the maximum flow. □

Part VI

Topic F: Dynamic Programming

18 DP Idea and Example

Dynamic programming (DP) is a general technique where we define the solution to a problem in terms of smaller subproblems. We'll start with an example, then describe the general approach.

Objectives: After learning this material, you should be able to:

- Name and explain the *components* of a dynamic programming (DP) algorithm.
- Solve the longest increasing subsequence problem using DP.

18.1 Longest increasing subsequence

This is the first problem we will solve with dynamic programming. Before defining it, some terminology: Given a list of numbers, also called a sequence, a **subsequence** is any result of deleting some elements of the list. For example, given the list $(1, 2, 3, 4, 5)$, the list $(1, 3, 5)$ is a subsequence but $(1, 5, 2)$ is not. A list of numbers is **(weakly) increasing** if each number is at least as large as the previous one.

The **longest increasing subsequence (LIS)** problem is:

- **Input:** a nonempty list of integers.
- **Output:** the length of its longest subsequence that is weakly increasing.

For the previous example, the longest increasing subsequence is the entire thing, $(1, 2, 3, 4, 5)$. If the input is $(1, 6, 5, 3, 4, 8)$, then the longest increasing subsequence is $(1, 3, 4, 8)$ with a length of 4 elements.

First, let's look at the brute-force algorithm. It considers each possible subsequence of the list. Among all the subsequences that are increasing, it takes the longest one.

Exercise 18.1. What is the time complexity of this algorithm? Suppose the input list has length n.

> **i** Solution.
>
> We can bound it by $O(n2^n)$. To get a subsequence, we pick a subset of the items $1, \ldots, n$. There are 2^n subsets. For each subset, we check whether it's increasing and how long it

is, which takes linear time in the size of the subset. In the worst case, the length of the subset is n, so we get $O(n2^n)$.

Brute force is much too slow, so now we'll solve this problem using dynamic programming. The idea behind the algorithm is to consider all the prefixes of the input list. First, we solve a variant of the LIS problem for the prefix of length one; this is easy. Then we use this to solve it for the prefix of length two. And so on up to the end.

Consider the above example, input $(1, 5, 6, 3, 4, 8)$. Suppose we're trying to solve the LIS problem for the prefix $(1, 5, 6, 3, 4)$, with the requirement that we must include the final element, 4. Well, an increasing subsequence that includes 4 can only be one of the following options:

- No prior element – the subsequence starts with 4, so its length is just one.
- 1 as the prior element. We would take the LIS ending with 1, and append 4, making it one longer.
- 3 as the prior element. We would take the LIS ending with 3, and append 4, again making it one longer.

The solution for the prefix ending at 4 is whichever of these options is the longest, which is of course the last one, giving a LIS of $(1, 3, 4)$. Now, we'll use this idea to go through the list and solve the problem up to each element so far, assuming we'll include that element.

Algorithm 18.1.

```
longest_increasing_subsequence(A):
   # A is a list of integers of length n
   let L[1] = 1
   for j = 2 to n:
      let L[j] = 1
      for i = 1 to j-1:
         if A[i] <= A[j]:
            set L[j] = max(L[j], L[i] + 1)
   return max(L)
```

The inner `for` loop on lines 6-8 implements the logic discussed above: it sets $L[j]$ to be 1 if there are no prior elements weakly smaller, and otherwise 1 plus the best of the previous eligible subsequences.

Proposition 18.1. *Algorithm 1 correctly solves the Longest Increasing Subsequence problem.*

Proof. We prove by induction the following statement: $L[j]$ is the length of the LIS of the input up to index j that includes the element at index j. This will prove correctness because we return $\max_j L[j]$, i.e. the longest increasing subsequence that ends at any location.

Base case: $L[1] = 1$ is correct, because we can take the first element to be a subsequence of itself, with length one.

Inductive case: Suppose $L[1], \ldots, L[j-1]$ are all correct. Now at index j, one possible subsequence is just the element $A[j]$ itself, which has length 1. The other kind of possibility is a subsequence that starts earlier and ends at j, with the prior index included being i. This can only occur if $A[i] \leq A[j]$, and if so, its maximum length is $L[i] + 1$ because we appended the element at index j. The algorithm takes the maximum over these possibilities, so $L[j]$ is correct.

Now if each $L[j]$ is correct, then the algorithm is correct because it returns the maximum of $L[j]$ for all j, one of which must be the LIS of the input. □

Proposition 18.2. *The running time of Algorithm 1 is $O(n^2)$ and space use is $O(n)$.*

Proof. Initialization runs in constant time and returning the answer runs in $O(n)$, finding the maximum element of L.

The outer `for` loop runs $n-1$ times, and each iteration, the inner `for` loop runs at most $n-1$ times, with constant-time operations. So the running time is $O(n^2)$.

The space usage is dominated by the array L, which is $O(n)$. □

Exercise 18.2. In Algorithm 1, why would it be wrong to return $L[n]$, the last entry of our answer array? Give an example input where returning $L[n]$ would be incorrect and explain how it fails.

> **i** Example solution.
>
> $L[n]$ is the length of the LIS that ends exactly at location n, but the LIS of the entire sequence may end earlier. An example is the input $(1, 9, 10, 5)$. Here the LIS has length 3 and is $(1, 9, 10)$, but $L[n] = 2$ because the LIS ending at the last element is $(1, 2)$.

Exercise 18.3. Simulate Algorithm 1 on input $(5, 1, 3, 2, 4, 0)$. What is L and what is the final solution?

Solution

Index	1	2	3	4	5	6
Input	5	1	3	2	4	0
L	1	1	2	2	3	1

The final solution is $\max_j L[j] = 3$.

18.1.1 Reconstructing the subsequence itself

Algorithm 1 returns the length of the LIS, but not the subsequence itself. Luckily, we can modify it quite easily to do this as well. The approach is similar to the modification of breadth-first-search and Dijkstra's algorithm to return the shortest path itself (not just its length). We have to keep track, for each result we got, "how we got there". The result is Algorithm 2.

Algorithm 18.2.

```
lis_2(A):
   # A is a list of integers of length n
   let L[1] = 1
   let prev[j] = null for all j   #<<
   for j = 2 to n:
      let L[j] = 1
      for i = 1 to j-1:
         if A[i] <= A[j]:
            set L[j] = max(L[j], L[i] + 1)
            if L[j] == L[i] + 1:  #<<
               set prev[j] = i    #<<
   let j = argmax(L)
   return L[j] and lis_reconstruct(prev, j)  #<<

lis_reconstruct(prev, j):  #<<
   let S = empty list      #<<
   while j is not null:    #<<
      add j to front of S  #<<
      j = prev[j]          #<<
   return S                #<<
```

Exercise 18.4. Revisiting our example, simulate Algorithm 2 on input $(5, 1, 3, 2, 4, 0)$. Give L, *prev*, and the final output.

Solution.

Index	1	2	3	4	5	6
Input	5	1	3	2	4	0
L	1	1	2	2	3	1
prev	null	null	2	2	3	null

The final solution is $\max_j L[j] = 3$ and the subsequence $(1, 3, 4)$. (Another answer is $(1, 2, 4)$.

18.2 Components of dynamic programming

Now that we've seen a dynamic programming algorithm, let's lay out the components that all DP algorithms have.

Dynamic programming algorithms always can be broken down into these components:

1. **Subproblem definition.** For example with LIS, subproblem j was "compute the length of the LIS of the prefix of the input up to j, requiring it to include the final element." We stored the solutions in an array L.
2. **Computing the final answer** from the subproblem answers. For LIS, we took the maximum solution to any subproblem, i.e. $\max_j L[j]$.
3. **Recurrence.** The *recurrence* states how to solve any given subproblem. It always has two parts:

- **Base case / initialization.** The base cases are the ones that don't rely on other subproblems. For LIS, the base case was a prefix of length 1, and we initialized $L[1] = 1$.
- **Inductive case:** how to solve a generic subproblem given the solution to "earlier" subproblems. For LIS, we said $L[j]$ was the maximum of 1 and $1 + L[i]$ over any $i < j$ where $A[i] \leq A[j]$.

4. **(Optional) reconstructing** the object that witnesses the solution. DP algorithms usually return the *size* or *value* of some object, for example, the length of the LIS. Then, they can usually be modified in a straightforward, formulaic way to construct that actual object itself, for example, the actual subsequence as we did above. This modification usually proceeds by remembering which choices we made when solving a subproblem, e.g. when setting $L[j] = L[i] + 1$, remembering which index i was used.

Every dynamic programming solution (at least in this class) is made up of the above components.

The key question you usually need to answer is: What are the subproblems, and what is the recurrence? Usually, the subproblems can be arranged in an array, since they must be solved in order. Often this array is multidimensional, as we will see. Sometimes the subproblem is essentially the same as the original problem, just on a prefix or subset of the input. But often the subproblem is slightly different, as in the LIS example where the subproblem required the subsequence to include the final element.

Once you define the above elements, the DP algorithm has essentially been defined:

- Create an array to store the subproblem solutions. (Later, this may be a 2d array or more.)
- Initialize the base case(s) of the recurrence.
- Iterate through the subproblems in dependency order and solve using the inductive case of the recurrence.
- Compute the final answer using the subproblem answers.

To reconstruct the witnessing object as well, it can generally be modified by creating a data structure that remembers the choices made, at each subproblem, when solving the recurrence; then backtracking through these choices.

Proofs of correctness. Every DP algorithm is proven correct with the following inductive proof:

1. (Base case) We prove the algorithm initializes the base cases or initial subproblems correctly.
2. (Inductive case) At each step, assuming the previous subproblems were solved correctly, we prove that the next subproblem is solved correctly.
3. By induction, steps (1) and (2) prove that all subproblems are solved correctly.
4. (Returning the final answer) We prove that, if all the subproblems were solved correctly, we return the final answer correctly.

For example, with the Longest Increasing Subsequence problem, the base case was that a prefix of length 1 has a LIS of length 1. Then, the inductive case was that for each subproblem $j = 2, \ldots, n$, assuming that $L[1], \ldots, L[j-1]$ were all correct, the algorithm computes $L[j]$ correctly (inductive step). Finallly, we argued that if $L[j]$ is correct for each j, then it is correct to return $\max_j L[j]$.

Because the proof always follows this template, we will always prove correctness by filling in the above parts: correctness of the recurrence – base case and inductive case – and of returning the final answer. We also need to make sure that we solve the subproblems in dependency order.

19 Knapsack

This section looks specifically at variants of the knapsack problem and their dynamic programming solutions.

Objectives. After learning this material, you should be able to:

- Solve each knapsack variant using dynamic programming.
- Identify the DP components of the knapsack algorithms.
- Solve new DP problems involving 2d arrays.

19.1 Duplicates allowed

In the knapsack problem, we are given a set of items $i = 1, \ldots, n$ each with a *value* $v_i \in \mathbb{R}_+$ (a positive number) and a *weight* or size $w_i \in \mathbb{N}$ (a nonnegative integer).

We are given a number $W \in \mathbb{N}$ which is the maximum weight our knapsack can hold, also called the capacity or size of the knapsack. We must find the max-value subset of items that can fit in the knapsack.

In the **duplicates allowed** version, there are unlimited copies of each item available.

Exercise 19.1. Given this input instance, what is the optimal solution? Suppose $W = 7$.

Item	Value	Weight
1	4	2
2	5	3
3	8	5

Solution.

The optimal solution is two copies of item 1 and one copy of item 2, for a value of 13. The total weight is $2 \cdot 2 + 3 = 7$, which is feasible as it matches the weight limit. We can check that every other feasible solution has lower value. For example, these are feasible solutions: three copies of item 1; or two copies of item 2; or one copy of item 3 and one copy of item 1.

Let's look for a DP solution. Recall the components of a DP solution: subproblem definition, computing the final value, the recurrence, and reconstructing the solution. Here a natural subproblem is to have a smaller-capacity knapsack. Let's try it: our **subproblem definition** is to let \$C[w] = \$ the maximum value we can fit in a knapsack of size w. With this subproblem, **computing the final value** is easy, as it is just $C[W]$.

For the **recurrence**, the **base case** is where $w = 0$, i.e. no items can fit, and the optimal value is zero. So we set $C[0] = 0$. For the **inductive case**: For $w \geq 1$, we set

$$C[w] = \max \begin{cases} C[w-1] \\ \max_{i:w_i \leq w} \; v_i + C[w - w_i] \end{cases} . \tag{19.1}$$

In other words, we can write the inductive case as an algorithm: * Set $C[w] = C[w-1]$, in other words, consider the optimal solution for a knapsack of size $w-1$. * That solution can fit in this knapsack as well, since this one is only larger. * For each item i: * Check if item i can fit in this knapsack, i.e. check if $w_i \leq w$. * If not, skip this item and keep going. * Put item i in the knapsack. We now have a space of $w - w_i$ remaining, and we have a value of v_i. * Fill the remaining space optimally. Luckily, we already solved that subproblem: it gives a value of $C[w - w_i]$. * Check if the resulting total value, $v_i + C[w - w_i]$, is better than our current solution. If so, keep it.

Claim 19.1. *The recurrence above is correct, i.e. \$C[w] = \$ the maximum value we can fit in a knapsack of size w.*

Proof. If $w = 0$, then $C[w] = 0$ because no items can fit.

Now, given $w \geq 1$, either no items fit, or at least one item fits. If no item fits, then $C[w] = C[w-1] = 0$, which is correct.

So now suppose that at least one item fits. The optimal solution has at least one item, say i. Now for the remaining space $w - w_i$ (which is at least zero since i fits in the knapsack), it must be used optimally. So the total value from the remaining space is $C[w - w_i]$, by inductive hypothesis. (If it were not used optimally, then we could get a better solution for the space and then add item i to it and obtain a better solution for $C[w]$, which contradicts the assumption that this is optimal.)

So $C[w] = v_i + C[w - w_i]$. So the recurrence is correct, since the optimal solution is the result of picking the best such item i. □

Combining these gives a dynamic programming algorithm:

Algorithm 19.1.

```
knapsack_dups(v, w, W):
   # v[i] = value, w[i] = weight, W = weight limit
   let C[0] = 0
   for x = 1 to W:
      C[x] = C[x-1]
      for i = 1 to n, if w[i] <= x:
         C[x] = max(C[x], v[i] + C[x - w[i]])
   return C[W]
```

Correctness: As usual in dynamic programming, correctness follows from correctness of the DP elements, which were argued above.

Efficiency: Space usage is dominated by C, which uses $O(W)$ space. For running time, we have nested loops, the outer one has W iterations and the inner one has n iterations, and the interior operations are constant time per iteration. So running time is $O(nW)$.

Exercise 19.2. Modify the algorithm to reconstruct the actual list of items in the optimal knapsack.

Hint: Recall that for reconstruction, we should keep track of the choices our algorithm needed to make at each subproblem. At subproblem $C[j]$, what were our choices? Then, how do we backtrack from the very end, i.e. $C[W]$, to the beginning, to reconstruct the set?

> **i** Solution.
>
> At each `x`, the choice was which item `i` to put in the knapsack. This took up `w[i]` space, and then we needed to add the rest of the solution, which was the optimal solution for `C[x - w[i]]`.

```
knapsack_dups_2(v, w, W):
   # v[i] = value, w[i] = weight, W = weight limit
   let C[0] = 0
   let Item[0] = null   #<<
   for x = 1 to W:
      C[x] = C[x-1]
      Item[x] = null    #<<
      for i = 1 to n, if w[i] <= x:
         C[x] = max(C[x], v[i] + C[x - w[i]])
         if C[x] == v[i] + C[x - w[i]]:  #<<
            Item[x] = i                  #<<
   return kd_reconstruct(w, W, C, Item)  #<<

kd_reconstruct(w, W, C, Item):  #<<
    let x = W                   #<<
    let solution = empty list   #<<
    while x > 0:                #<<
        if Item[x] == none:     #<<
            set x = x - 1       #<<
        else:                   #<<
            add Item[x] to solution  #<<
            set x = x - w[Item[x]]   #<<
    return solution             #<<
```

Instead of a list, we could use an array where the ith entry counts how many copies of item i are in the solution. This would be more space-efficient for large instances.

19.2 No Duplicates

In this version of the knapsack problem, there is just one copy of each item, but the rest of the problem (including the format of the input) is the same.

We might hope to modify the previous solution while keeping the subproblem $C[w]$ essentially the same. For example, by somehow remembering which items were used in $C[w]$. This turns out to fail, in part because there could be multiple optimal subsets for $C[w]$, and remembering all of them turns out to be prohibitive. Instead, we need a trick.

Subproblem definition. The trick is to *introduce an extra dimension* to our subproblems. Specifically, for our subproblem definition, let $C[k, w]$ be the maximum value one can obtain from a knapsack of size w using only items from the subset $\{1, \ldots, k\}$.

Computing the final solution. We will simply return $C[n, W]$ where n is the number of items and W is the knapsack capacity.

Recurrence. The **base case** is pretty straightforward: $C[k, 0] = 0$ for all item indexes k and $C[0, w] = 0$ for all capacities w.

For the **inductive case**, we set for $k \geq 1, w \geq 1$:

$$C[k, w] = \max \begin{cases} C[k-1, w] \\ v_k + C[k-1, w-w_k] \quad (\text{if } w_k \leq w) \end{cases}. \tag{19.2}$$

Claim 19.2. *The recurrence is correct, i.e. $C[k, w]$ = the maximum value obtainable from a knapsack of size w using only items $\{1, \ldots, k\}$.*

Proof. For the optimal solution with items $1, \ldots, k$ and capacity w, there are two possibilities: we either include item k, or we don't. If we don't, then the optimal solution uses only items $1, \ldots, k-1$, so its value is $C[k-1, w]$.

If we do, then the remaining space is $w - w_k$, and to fill it, we are only allowed to use items $1, \ldots, k-1$ because we just used item k. So the optimal way to fill the remaining space is $C[k-1, w-w_k]$, and our total value is $v_k + C[k-1, w-w_k]$. Note this is only possible if $w_k \leq w$, as otherwise item k cannot fit.

Since these are the only two possibilities (or only one possibility if $w_k > w$), and the recurrence chooses the best of both, it is optimal. □

Our algorithm is therefore:

Algorithm 19.2.

```
knapsack(v, w, W):
   let C[0,x] = 0 for all x = 1 to W
   let C[i,0] = 0 for all i = 1 to n
   for i = 1 to n:
      for x = 1 to W:
         let C[i,x] = C[i-1,x]
         if w[i] <= x:
            set C[i,x] = max(C[i,x], v[i] + C[i-1,w-w[i]])
```

Correctness. As usual for dynamic programming, correctness follows almost immediately from the above arguments that the three components (subproblem, final solution, recurrence) are correct.

Efficiency. Initialization takes $O(n + W)$ time, and returning the final result is constant time. There are nested loops of n and W iterations, with constant-time operations in the innermost loop, so runtime is $O(nW)$. Space is dominated by C, which uses $O(nW)$ space.

Exercise 19.3. How do we modify the no-duplicates knapsack algorithm to return the optimal subset of items (i.e. **reconstruct** the solution), not just its value?

Solution.

We can create an array `D[i,x] = False` if `C[i,x] == C[i-1,x]`, and otherwise `D[i,x] = True`, meaning that `C[i,x] == v[i] + C[i-1,w-w[i]]` and we used item i at this stage.
We then start with `D[n,W]`. If False, we go to `D[n-1, W]`. If True, we add item n to the knapsack and go to `D[n-1, W - w[n]]`. We continue in this way until we get to `D[0,x]` for any `x`.

20 Longest Common Subsequence

This section considers another DP example, longest common subsequence (LCS).

Objectives. After learning this material, you should be able to:

- Execute the LCS algorithm on example inputs.
- Identify the DP components of the LCS algorithm.
- Solve new DP problems similar to LCS.

20.1 The problem and algorithm

In the longest common subsequence problem, our goal is to compare two sequences to find the longest subsequence that they have in common. Recall that a subsequence does not have to be consecutive. This problem is a part of, for example, version control software like `git` that needs to track changes to a document.

- **Input:** two sequences A and B (we will suppose the elements are characters or inetegers).
- **Output:** the length of the longest sequence that is a subsequence of both A and B.

Exercise 20.1. Let A = "ALGORITHM" and B = "ANARCHISM". What is their longest common subsequence?

> **i** Solution.
>
> One answer is ARIM, for a length of 4. Another is ARHM, also with length 4.

As always, the first question is the **subproblem definition.** A natural first try is a smaller version of the original problem. In that case, let C[i,j] = the length of the longest common subsequence of A[1:i] and B[1:j], where A[1:i] denotes the prefix of A from characters 1 to i and similarly for B[1:j].

In this case, to **comptue the final answer**, we just return C[n,m], where n = len(A) and m = len(B).

For the **recurrence base case**, if either input has zero characters, then the answer is zero, so C[0,j] = 0 for all j and C[i,0] = 0 for all i.

For the **recurrence inductive case**, suppose $i, j \geq 1$. We need to consider cases. If A[i] == B[j], then one possibility for C[i,j] is a subsequence that ends with this character. The optimal length would be the length of a subsequence of A[1:i-1] and B[1:j-1], plus one more for this final character. This gives option `a := 1 + C[i-1,j-1]`. If A[i] != B[j], then we can set option `a := 0`.

Then, regardless of whether A[i] and B[j] are equal, we have two more options. If we do not include the last character of A in our subsequence, then our solution is the same as on A[1:i-1], which has value `b := C[i-1,j]`. And if we do not include the last character of B, then similarly the value is `c := C[i,j-1]`. (Note that if we do not include both, this will be covered by b and c.)

Putting these together, we can choose the best of these choices, `C[i,j] = max{a, b, c}`.

Combining all of the elements gives us this DP algorithm. Notice that we need to decide how to iterate through the subproblems. It's important to make sure that when we're solving subproblem (i,j), we've already solved all the subproblems it depends on: in this case, (i-1,j-1), (i-1,j), and (i,j-1).

Algorithm 20.1.

```
lcs(A, B):
   # A has length n, B has length m
   let C[i,0] = 0 for all i = 1 to n
   let C[0,j] = 0 for all j = 1 to m
   for i = 1 to n:
      for j = 1 to m:
         let a = 1 + C[i-1,j-1] if A[i] == B[j], else let a = 0
         let b = C[i-1,j]
         let c = C[i,j-1]
         let C[i,j] = max{a, b, c}
   return C[n,m]
```

Exercise 20.2. Execute the algorithm on input A = ALGORITHM, B = ANARCHISM. Fill in the two-dimensional table and give the final answer. Briefly explain how you filled in the squares where $i \leq 2$ and $j \leq 2$.

Solution.

	''	A	L	G	O	R	I	T	H	M
''	0	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1	1

N	0	1	1	1	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1	1	1
R	0	1	1	1	1	2	2	2	2	2
C	0	1	1	1	1	2	2	2	2	2
H	0	1	1	1	1	2	2	2	3	3
I	0	1	1	1	1	2	3	3	3	3
S	0	1	1	1	1	2	3	3	3	3
M	0	1	1	1	1	2	3	3	3	4

The final answer is C[n,m] = 4.
To fill in those squares, first we used the base case to set C[i,j] = 0 if i == 0 or j == 0. Then for C[1,1], because the first characters are both A, they match and we use C[1,1] = 1 + C[0,0] = 1. For C[1,2], they don't match, so we use the max of C[1,1] and C[0,2], which is 1. For C[2,1], they don't match, so we use the max of C[1,1] and C[2,0], which is 1. For C[2,2], they don't match, so we use the max of C[1,2] and C[2,1], which is 1.

Exercise 20.3. Explain how to modify the algorithm to reconstruct the longest subsequence itself. You do not need to write the entire code of a new algorithm, just describe how to do it. Briefly justify correctness.

Solution.

We make a second array D where D[i,j] tells us which choice we made when filling in C[i,j]. We can let D[i,j] = "a", "b", or "c" depending on which one achieved the max.
To reconstruct the subsequence afterward:

- Start at i = n, j = m. Let S be an empty sequence.
- If D[i,j] == "a": If A[i] == B[j], then we add this character to the beginning of S. Regardless, we then let i -= 1, j -= 1.
- If D[i,j] == "b", then we let i -= 1.
- If D[i,j] == "c", then we let j -= 1.
- Repeat until i or j is zero, then stop and return S.

This is correct because: if D[i,j] == "a", then the optimal longest common subsequence of A[1:i] and B[1:j] includes the current character A[i] == B[j], preceded by the optimal subsequence of A[1:i-1] and B[1:j-1]. So we add this character to S, and then recurse to the case (i-1,j-1). Similarly for the other options.

21 All-pairs shortest paths

This section uses dynamic programming to solve all-pairs shortest paths.

Objectives. After learning this material, you should be able to:

- Execute the Floyd-Warshall algorithm on example inputs.
- Identify the DP components of the Floyd-Warshall algorithm.
- Solve new DP problems involving adding an extra dimension.

21.1 The problem and algorithm

In the all-pairs shortest paths problem, our goal is to output a data structure giving the shortest path from *any* starting point to any end point.

- **Input:** Graph $G = (V, E)$ with n vertices, weighted, directed, with no negative cycles.
- **Output:** two-dimensional array \$D[u,v] = \$ length of shortest path from u to v.

The challenge here is that $D[u, v]$ does not give us any useful subproblem to work with. We need a new clever idea to introduce simpler subproblems that enable us to build up a solution. As with knapsack, we'll introduce an extra variable. The key idea is to consider paths that only use a subset of the vertices. We can grow the subset to build up more complex solutions.

Subproblem definition. Let \$d[u,v,k] = \$ length of the shortest path from u to v using as intermediate nodes only vertices $1, \ldots, k$.

Final solution. In particular, with n vertices, \$d[u,v,n] = \$ the length of the shortest path from u to v using all vertices. So if we set $D[u, v] = d[u, v, n]$ for all u, v, this will be correct.

Recurrence. For the **base cases**, we set $d[u, u, 0] = 0$ for all u. Then, for all edges (u, v) with length $w[u, v]$, we set $d[u, v, 0] = w[u, v]$. For all other pairs, we set $d[u, v, 0] = \infty$.

For the **inductive case** with $k \geq 1$, imagine we've solved $d[u, v, k-1]$ for all u, v and now we want to compute $d[u, v, k]$. We set:

$$d[u, v, k] = \min \begin{cases} d[u, v, k-1] \\ d[u, k, k-1] + d[k, v, k-1] \end{cases} . \tag{21.1}$$

Informally, this says we can either use the old route that didn't include k at all, or we can include k. If we do, then we must route from u to k somehow, using distance $d[u, k, k-1]$, and then route to v somehow, using distance $d[k, v, k-1]$.

Claim 21.1. *The recurrence is correct, i.e. $d[u,v,k] = $ the length of the shortest path from u to v that goes through only vertices $1, \ldots, k$.*

Proof. The shortest path from u to v, using only intermediate vertices $1, \ldots, k$, either uses vertex k or it doesn't. Suppose it doesn't. Then $d[u, v, k] = d[u, v, k-1]$, by definition.

Suppose it does. Then the shortest path using $1, \ldots, k$ has the form $u, \ldots, k, \ldots, v$. Then the portion $u, \ldots, k$ must be a shortest path from u to k using intermediate vertices $1, \ldots, k-1$. (Otherwise, we could take the shortest path and shorten the distance from u to v, a contradiction.) Similarly, the portion $k, \ldots, v$ must be a shortest path from k to v. So in this case, $d[u, v, k] = d[u, k, k-1] + d[k, v, k-1]$.

Since the shortest path must be one of these two cases, it is the smaller of the two. □

Putting the pieces together, we get this algorithm:

Algorithm 21.1.

```
floyd_warshall(G, w):
   let d[u,v,0] = 0         if u==v,
                = w[u,v]    if (u,v) is an edge,
                = infinity otherwise
   for k = 1 to n:
      for u = 1 to n:
         for v = 1 to n:
            set d[u,v,k] = min(d[u,v,k-1], d[u,k,k-1] + d[k,v,k-1])
   let D[u,v] = d[u,v,n] for all u,v
   return D
```

Correctness. As usual with dynamic programming, correctness follows from above arguments that the subproblem, final solution, and recurrence are correct.

Efficiency. Initialization requires up to $O(n^2)$ time, since we set $d[u, v, 0]$ for all pairs of nodes. Similarly, returning the solution requires constructing an $O(n^2)$ array, which has the same running time. There are three nested loops, each with n iterations, and constant-time operations within each. So the running time is dominated by $O(n^3)$.

The space includes D and local variables, but is dominated by d which uses $O(n^3)$ space.

21.1.1 Reconstructing the solution

In this case, we obtained the lengths of the shortest paths, but not the actual paths themselves. As usual, reconstructing the solution will involve remembering the choices made when solving the subproblems, but here the full procedure is a bit unusual.

A merge approach. The most direct approach, applying our usual DP approach, is as follows. Let us create a variable `inter[u,v]` standing for "intermediate" vertices between u and v. Initially, we set `inter[u,v] = none` for all u,v. Whenever we make a modification `d[u,v,k] = d[u,k,k-1] + d[k,v,k-1]`, we set `inter[u,v] = k`.

Now, we can reconstruct the path as follows:

- If `inter[u,v] == none`, then we must have followed an edge directly from u to v, so the path is just u, v.
- Otherwise, if `inter[u,v] == k`, then we make a recursive call to reconstruct the path from u to k, another to get the path from k to v, and we concatenate these.

A "next" approach. Notice that if a shortest path is of the form $u, x, \ldots, v$, then it is also true that $x, \ldots, v$ is a shortest path from x to v. This implies that we only need to know, for each pair u, v, what the "next" vertex is on a shortest path. If we find that it is x, then we continue by finding the next vertex on the path from x to v, etc.

So initialize `next[u,v] = v` if there is an edge (u, v) and otherwise `next[u,v] = none`. Whenever we make a modification `d[u,v,k] = d[u,k,k-1] + d[k,v,k-1]`, we can set `next[u,v] = next[u,k]`, since the shortest path to v proceeds by first taking the shortest path to k.

In this case, reconstruction is even easier:

- Begin the path with `u`.
- Let `x = next[u,v]`.
- Add `x` to the path.
- If `x == v`, stop.
- Otherwise, let `x = next[x,v]`, go to step 3, and continue.

Part VII

Topic G: Hash Tables and P vs NP

22 Hash Tables

Note: "Hash Tables" and "P vs. NP" are grouped together in Topic G because it's convenient for organization, as together they're about the same size as each other topic. However, they are separate concepts.

Hash tables are a very useful data structure that use randomness to achieve good performance.

Objectives: After learning this material, you should be able to:

- Use hash tables as sets, key-value stores, and for similar purposes.
- Analyze how hash tables with m bins and n items perform under perfect hashing, ideal (random) hashing, and worst-case hashing.
- Explain the birthday paradox.

22.1 A hash table as a data structure

Data structures store and retrieve information. A data structure can be broken down into:

- An interface: what operations does it support?
- An implementation: how do we achieve those operations, and what is the time and space complexity of each?

First, let's think of a hash table as a **Set**. It will support these operations:

Operation	Average-case time	Worst-case time	Description
Store(x)	$O(1)$	$O(n)$	Add x to the set
Get(x)	$O(1)$	$O(n)$	Check if x is in the set
Remove(x)	$O(1)$	$O(n)$	Delete x from the set

We will discuss the difference between average-case and worst-case time soon.

It is pretty straightforward to modify such a structure to store **key-value pairs**; such a data structure is often called a *table*, a *dictionary*, or a *map*. If we want to store a key k associated to a value v, then we simply treat k as we did x above, and when we go to store it in the data structure, we also store v along with it. Then, when we want to look up k, we can find it and also retrieve v.

Operation	Average-case time	Worst-case time	Description
Store(k,v)	$O(1)$	$O(n)$	Store key-value pair k, v
Get(k)	$O(1)$	$O(n)$	Retrieve value v for key k, if any
Remove(k)	$O(1)$	$O(n)$	Delete k and its value

One can also use a binary tree or other data structures to implement these operations, but not as quickly (in the average case).

An example use case would be counting the number of times each word appears in a document:

Algorithm 22.1.

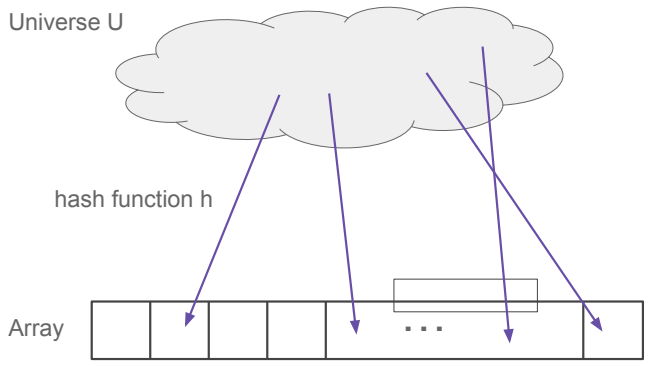
```
count_words(document):
   hash_table = new hash table
   for each word in document:
      previous_count = hash_table.Get(word)  # or zero if not yet stored
      new_count = previous_count + 1
      hash_table.Store(word, new_count)
```

22.2 Implementing hash tables

In general, we will implement a hash table by using an array to store the items. If we have n items, we would like to have an array of length $m = n$, so that there is one slot per item.

To store the items, we will choose in advance a **hash function** that assigns each item a location in the array.

Definition 22.1 (Hash function)**.** Given a universe U of items and an array length m, a hash function is a function $h : U \to \{1, \dots, m\}$ that assigns each item a location in the array.



However, the problem is that usually, there is a large "universe" of items that might arrive, much larger than n. For example, we may be storing 64-bit integers in a set. In this case, there are 2^{64} possible integers that might arrive. Even if only a small number of them do, we don't know in advance which ones.

Exercise 22.1. Suppose we are hashing 64-bit integers to $\{0, \ldots, 1023\}$. We will use a naive hash function, `h(x) = x % 1024`, where % is the "modulo" operator.

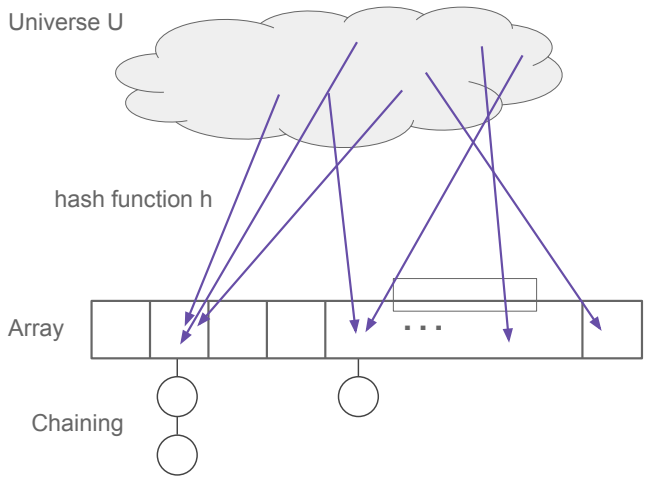
Part a. Give a set of 1024 integers that are "perfectly hashed": each one goes to a different location. Briefly explain.

Part b. Give a set of 1024 integers that are hashed in the worst way: they all go to the same location. Briefly explain.

> Sample solution.
>
> **Part a.** For example, we can take $\{0, 1, \ldots, 1023\}$. Each number is hashed to itself.
> **Part b.** For example, $\{1, 1025, 2049, \ldots\}$, that is, the numbers of the form $1024k + 1$ for $k = 0, \ldots, 1023$. All of these are hashed to slot 1.

22.2.1 Collisions and chaining

If two items are hashed to the same location, we call this a *collision*. Every hash table needs a strategy to handle collisions. We will focus on a straightforward strategy called *chaining*. In chaining, we simply make a list (such as a linked list) for each array location. If multiple items arrive, we add them to the list for that location.



Note 22.1. ***Hash table implementation.***

To summarize, the hash table is implemented as follows. Before starting, we pick a hash function h. We will discuss the choice of h later.

Store(x):

- *Let $j = h(x)$.*
- *If there is no item at location j, put x there.*
- *If there are already items there, add x to the linked list for location j.*

Get(x):

- *Let $j = h(x)$.*
- *If there is no item at location j, return None.*
- *If there is one or more items, look through the linked list for x and return whether you found it.*

Remove(x):

- *Follow the instructions for Get(x), but if you find x, delete it from the list.*

22.3 Analysis of hash tables

Suppose we will Store() n items total in a hash table with array length n. Then, we will make n calls to Get(). What will be the total time complexity?

Note 22.2. *We will assume that all of our hash functions run in constant time, $O(1)$, per call.*

In the **best case**, each item is hashed to a different location. Because all of the lists are of length 1, all of the store and retrieve operations run in constant time. Therefore, the total time is $O(n)$, and each Store() and Get() command runs in $O(1)$ time.

In the **worst case**, all of the items are hashed to the same location. The time for all of the Store and Get commands is $O(1 + 2 + \cdots + n) = O(n^2)$. Therefore, on average, each Store() and each Get() command runs in $O(n)$ time.

Now, let's consider the average case. To do so, we have to see what hash function we will actually use. We will use randomness to try to "spread out" the hash function's outputs. For analysis, we'll make this assumption.

#Ideal hash function An ideal hash function is one where each item's location is chosen uniformly at random and independently from $\{1, \ldots, m\}$.

Assuming an ideal hash function, each bin has *on average* one element. Therefore in the **average case**, each Store() command and each Get() command use O(1) operations. (To be more formal, one can show that the sum over all n Store() and Get() commands is $O(n)$, which gives $O(1)$ per command average-case.)

Remark 22.1. Using an ideal hash function is modeled by the "balls in bins" problem where we throw n balls (these are the items being hashed) into m bins (the array locations), where each ball lands in a independent, uniformly randomly chosen bin.

Exercise 22.2. Suppose we throw n balls into n bins, i.e. we hash n items into n array slots using an ideal hash function. What is the probability that bin number 1 is empty (has no items)?

> **i** Solution.
>
> Each ball misses bin 1 with probability $1 - \frac{1}{n}$. Since they're independent, the probability they all miss is $\prod_{j=1}^{n}(1 - \frac{1}{n}) = (1 - \frac{1}{n})^n \to \frac{1}{e}$ as $n \to \infty$. So the probability is about $\frac{1}{e} \approx 0.3678$.

Note 22.3.

23 Implementation

To recap, we hope to have a hash function that is as close to "ideal" as possible. In practice, there are many hash functions, and they often do have a randomly chosen component. They often work by using arithmetic operations on the bits of the input in order to "scramble" them and produce a randomly-distributed output.

Another implementation note is that, often, we don't know in advance how many items we'll store in the hash table. This can be addressed by starting with a small table and doubling the size of the table once the number of items grows large enough, reassigning all items to the new array slots. We repeat as needed.

23.1 Birthday paradox

When analyzing hash tables, one question is how many collisions will occur and *when* they will occur. Let's start by asking when the *first* collision will occur. How many balls do we need to throw into n bins before we expect that two balls have landed in the same bin?

Proposition 23.1. *If we throw k balls into n bins, the expected number of collisions is $\binom{k}{2}\frac{1}{n} = \frac{k(k-1)}{2n}$.*

Proof. Consider any pair of balls. What is the probability they land in the same bin? The answer is $\frac{1}{n}$: the first ball lands in some bin, and the second ball has a $\frac{1}{n}$ chance of landing in that bin.

But now the expected number of collisions is just the sum, over all pairs of balls, of the probability that pair collides. There are $\binom{k}{2} = \frac{k(k-1)}{2}$ pairs of balls, so the answer is $\binom{k}{2}\frac{1}{n}$. □

Corollary 23.1. *For $k \approx \sqrt{2n}$, after throwing k balls, there is in expectation one collision.*

This result is surprising because $\sqrt{2n}$ is much less than n; we probably get a collision very quickly! An example is the "birthday paradox". There are about 365 days in a year. If we pretend that each person's birthday is chosen independently and uniformly at random, then how many people do we need in a room before we expect that some pair share a birthday? The answer is only about $27 = \sqrt{2 \cdot 365}$.

This result implies that we need a strategy for handling collisions even when the number of items is much smaller than the size of the array.

24 P and NP

We now switch gears to discuss the complexity classes P and NP.

Objectives: After learning this material, you should be able to:

- Define P.
- Define NP using the verifier definition.
- List common problems in P and NP.
- Prove that a problem is in NP by giving a verifier.

24.1 Computational complexity and P

Throughout class, we have asked: how efficiently can you solve a problem? In particular, we often want to know the fastest runtime of any algorithm for a problem such as shortest paths or minimum spanning tree.

Given that, we can also start to classify problems themselves as "easy" (they have a fast algorithm) and "hard" (they have no fast algorithm). We call such classes *complexity classes*.

More precisely, we'll look at a very permissive definition of easy: running in polynomial time, i.e. running time $O(n^k)$ for some constant k. Almost all of the problems we've seen in this class fall into this definition, but we'll see that some care is needed. We'll also need a universal way to measure running time that works for all problems, whether they're on graphs, integers, lists, or any other type of input. We'll do so by measuring the input size in terms of the number of bits needed to write the input down.

Definition 24.1 (Polynomial time)**.** An algorithm is a polynomial-time algorithm if there exists a constant k such that, when the input is represented using at most n bits, the running time of the algorithm is at most $O(n^k)$.

Finally, we'll make things simpler by focusing on *decision problems.*

Definition 24.2 (Decision problem)**.** A decision problem is a problem where the answer is either yes or no.

Many problems can be converted into decision problems. For example, the shortest paths problem can be re-framed as a decision problem: is there any path of length at most d? If we can solve shortest paths in polynomial time, then we can solve the decision problem in polynomial time. And conversely, if we had some polynomial-time algorithm for the decision problem, we could use it to solve the original shortest-paths problem by binary search on d.

Definition 24.3 (P)**.** The complexity class P consists of all decision problems that have a polynomial-time algorithm.

We tend to think of P as problems that are "easy", or at least "tractable".

24.2 NP

Now we will look at a broader class of problems. Informally, NP will be the problems where we can "check" or "verify" a yes-answer in polynomial time. An example is the *Hamiltonian path* problem: given a directed graph, does there exist a path that visits every vertex exactly once? If the answer is yes, then we can ask for such a path and check that it does indeed vist every vertex once. This is easy to check, if we have the path. (In this case, the path itself is called the "witness" that the answer is yes for this given graph.) But it turns out that we don't know of an efficient (i.e. polynomial-time) algorithm to *find* such a path.

Definition 24.4 (Verifier)**.** A verifier for a decision problem is an algorithm that satisfies the following properties.

- The verifier takes as input a pair (I, W) where I is an instance of the decision problem and W is the "witness".
- The verifier outputs either "accept" or "reject".
- The size of the witness satisfies `length(W) <= p(length(I))` where p is some polynomial.
- If I is a "yes" instance of the decision problem, then there exists at least one witness W such that the algorithm accepts (I, W). (*"Completeness"*)
- If I is a "no" instance of the decision problem, there for any witness W, the algorithm rejects (I, W). (*"Soundness"*)

Example 24.1 (Hamiltonian Path)**.** As mentioned, the Hamiltonian Path decision problem is whether a directed graph has a path that visits every vertex exactly once. It has the following verifier:

- The witness string is a list of n of the vertices of the graph.
- Given (I, W), the verifier checks if W is a valid path in the graph and if it contains every vertex exactly once.
- If so, the verifier accepts, otherwise it rejects.

We can see that if there really does exist a Hamiltonian path $v_1, \ldots, v_n$, then for witness $W = (v_1, \ldots, v_n)$, the verifier will accept. On the other hand, suppose there doesn't exist any Hamiltonian path. In other words, the answer to the problem is "no". Then no matter what witness W is provided, the verifier will always reject, because no list of n vertices can be a valid path that contains every vertex once. So Hamiltonian path has a verifier.

Example 24.2 (Knapsack)**.** The knapsack decision problem is: given a list of objects with weights and values, and given a knapsack size W and value goal V, does there exist a subset of the items with total weight at most W and total value at least V? A verifier is the following:

- The witness string is a subset of the objects. (This is polynomially-sized, since it is a list of at most n numbers.)
- Given (I, W), the verifier adds up the values and weights of the objects.
- If the total weight is at most W and the total value is at least V, it accepts. Otherwise, it rejects.

If there really does exist such a subset of items, then that subset is a witness that will cause the verifier to reject. On the other hand, if no such subset exists, then the verifier will reject no matter what witness it is given. This follows because any subset will either have too much weight or not enough value. So knapsack has a verifier.

We can now define the complexity class NP.

Definition 24.5 (NP)**.** The complexity class NP consists of all decision problems that have a polynomial-time verifier.

We can check that our verifiers for Hamiltonian Path and Knapsack, above, both run in polynomial time. Therefore, both of these problems are in NP.

Exercise 24.1. The Hamiltonian Cycle problem is: given a directed graph G, does there exists a cycle that visits every vertex exactly once, except the start and end vertices? Prove that the Hamiltonian Cycle problem is in NP.

i Example solution.

The witness is the cycle that visits every vertex exactly once. The verifier, given the graph G and a witness string interpreted as a list of vertices, checks that each vertex is in the list once (except the start and end vertex) and checks that the list of vertices is actually a cycle in the graph. If all of these checks pass, it accepts.
This verifier runs in polynomial time, as this is only a linear number of checks (with an adjacency matrix; at most quadratic with an adjacency list). It accepts if and only if the witness is a Hamiltonian Cycle, by definition.

24.2.1 Comparing P and NP

First, we should note that every problem in P is also in NP.

Theorem 24.1. $\mathsf{P} \subseteq \mathsf{NP}$. *That is, any problem that has a polynomial-time algorithm also has a polynomial-time verifier.*

Proof. Suppose a problem has a polynomial-time algorithm M. Then the verifier is the following: Given (I, W) where I is an instance of the problem and W is a witness, we ignore the witness and just run the algorithm M on the input I. If M outputs "yes", we accept. If it outputs "no", we reject. This verifier runs in polynomial time. It satisfies completeness because for any yes instance I, regardless of what witness we use, the verifier accepts. It satisfies soundness because for any no instance I, regardless of what witness we use, the verifier rejects. □

The largest open problem in Computer Science is whether P and NP are actually the same, i.e. whether P = NP.

Open Problem 24.1. ***P = NP***

Does P *equal* NP*, or is* NP *a strictly larger set? That is, do there exist problems that have a polynomial-time verifier, but have no polynomial-time algorithm?*

It's worth mentioning that every problem in NP can be solved by a brute-force search of the following form:

- Given an instance I, iterate through all possible witness strings W.
- For each W, run the verifier to check if it accepts (I, W). If so, return "yes".
- If the verifier never accepted, then return "no".

However, this takes $\text{poly}(n)2^{\text{poly}(n)}$ time on inputs of length n. The question of P = NP is the question of whether problems that admit this type of brute-force search can always be sped up to polynomial time.

25 NP-Completeness

Finally, we will look at an amazing method to show that some problems are "at least as hard" as others: NP-Completeness

Objectives: After learning this material, you should be able to:

- Define NP-Complete.
- Name some common NP-complete problems
- Prove that a problem is NP-complete using a reduction.

25.1 NP-Completeness

Amazingly, there are problems in NP that are "at least as hard" as any other problem in NP. Such problems are called "NP-complete". To define NP-completeness, we will consider reductions, i.e. ways to transform some Problem A into another Problem B. If Problem B is easy to solve, this would imply that A is easy to solve too (by transforming it to B). We will find types of "Problem B" that *every* problem in NP can be reduced to.

Definition 25.1 (Mapping reduction)**.** A mapping reduction from decision problem A to decision problem B is an algorithm that, given an instance I of problem A, produces an instance I' of problem B such that I is a "yes" if and only if I' is a yes. It is a polynomial-time mapping reduction if the algorithm runs in polynomial time in the size of its input I.

We can now make an amazing definition: NP-Complete. A problem B is NP-Complete if every other NP problem reduces to it.

Definition 25.2 (NP-Complete)**.** A decision problem B is NP-Complete if it is in NP and, for every decision problem A in NP, there exists a polynomial-time mapping reduction from A to B.

Here is one reason NP-Completeness is important: if we have a single NP-Complete problem, and we can solve it efficiently, then we can solve *every* problem in NP efficiently.

Proposition 25.1. *If a decision problem B is NP-Complete and if we have a polynomial-time algorithm for B, then P=NP.*

Proof. Let A be any problem in NP. Because B is NP-complete, there is a polynomial-time mapping reduction M from A to B. Here is a polynomial-time algorithm for A:

1. Given an input I for A, use M to produce an input I' for B. Notice that, since M runs in polynomial time, the size of I' is at most $|I'| \leq p(|I|)$ for some polynomial p, where $|I|$ is the size of input I.
2. Run the polynomial-time algorithm for B on I'. This takes time $q(|I'|)$, where q is some polynomial. Return the answer.

This algorithm is correct by definition of mapping reduction. It also runs in polynomial time: The running time of M is polynomial in $|I|$, and the running time of the algorithm for B is at most $q(p(|I|))$, where q and p are polynomials. The composition of two polynomials is still a polynomial. □

For an example of the p and q used in the proof, imagine that our reduction takes an instance I of length n and produces an instance I' of length n^2. And suppose that the algorithm for B runs in time $O(m^3)$ on inputs of length m. Then the total running time is $O((n^2)^3) = O(n^6)$.

25.2 A first NP-Complete problem

It's not obvious whether there are any NP-Complete problems at all, but there are. Here is our first one: Satisfiability, or SAT for short.

Definition 25.3 (SAT)**.** The Satisfiability (SAT) problem is:

- **Input:** a boolean formula on n variables $x_1, \ldots, x_n$, i.e. a combination of logical AND, OR, NOT, and parenthetical groupings of variables.
- **Output:** "yes" if there exists an assignment of True or False to each variable such that the formula overall evaluates to True; "no" if no such assignment exists.

Proposition 25.2. *SAT is in NP.*

Proof. The witness string is the assignment of True/False to the variables $x_1, \ldots, x_n$. The verifier simply plugs the variables into the formula and evaluates it; if True, it accepts, otherwise it rejects. □

Theorem 25.1. *SAT is NP-Complete.*

Sketch. We already showed that SAT is in NP. We now must show that every problem A in NP has a polynomial-time mapping reduction to SAT.

Let A be a decision problem in NP with a verifier V. Here is our mapping reduction. First, we will represent the memory and variables used by V in binary. On input I of size n, V uses at most $p(n)$ total bits of memory over the whole course of its operation, for some polynomial p, including the input variables. Furthermore, it takes at most $q(n)$ total steps. We will have a new variable $x_{i,j}$ representing memory slot i at time step j, where True represents that the bit is set to 1 at that time.

For each possible step $i = 1, 2, ...$ of V, we can write down a Boolean formula f_i that evaluates to True if and only if that step was completed correctly. For example, if step i is represented as $z = z + 1$, then the formula would involve the bits of v at step $i - 1$ and step i and would evaluate to True if the addition was carried out correctly. (Checking this claim formally would take a lot more detail, but we will skip it here.) We also make a Boolean formula f_{accept} that evalutes to True if and only if the verifier accepts based on its memory state at the last step.

Now, our mapping reduction works as follows. Given an instance I of problem A, we construct the Boolean formula f_0 AND f_1 AND $\cdots$ AND f_{accept}. Here f_0 is a formula that says the memory of V is correctly initialized to the instance I and that the rest of the memory, except for W, is correctly initialized. We let the witness input W remain unconstrained.

The mapping reduction runs in polynomial time because each f_i can be constructed in polynomial time, and there are polynomially many of them. Further, it is correct because it returns a satisfiable formula if and only if there exists a setting of the witness W, and a setting of the state variables at each time step, such that the verifier runs as it is supposed to and accepts. If there is no such witness W, the formula is not satisfiable because any initial setting of W and setting of the state variables will either not represent a valid operation of V, or will result in V rejecting. □

25.3 More NP-Complete problems

Knowing about NP-Completeness is very important in practice, because many problems turn out to be NP-Complete. If you run into an NP-Complete problem, you know that it has no known polynomial-time algorithm. Therefore, you will need to think about how to approach it: maybe you can try an approximately-correct algorithm, a heuristic, or (if your problem is not too large) turn to algorithms out there tailored for solving NP-Complete problems as fast as possible.

What other problems are NP-Complete besides SAT? It turns out that now that we have one NP-Complete problem, we can have many more by a more simple approach.

Proposition 25.3. *If A is NP-Complete, and B is in NP, and A reduces to B with a polynomial-time mapping reduction, then B is NP-Complete too.*

Proof. For any problem X in NP, we can first take a polynomial-time mapping reduction from an instance I to an instance I' of A, then another reduction to an instance I'' of B. Since each reduction runs in polynomial time, this is a polynomial-time mapping reduction from X to B. □

To prove another problem is NP-Complete, we just have to show that SAT can be reduced to it. Here are some of the many known NP-Complete problems:

- Hamiltonian Path (see Section 24.2)
- Clique problem: given an undirected graph G and integer k, does there exist a subset of k vertices that are all pairwise neighbors with each other?
- Knapsack problem, decision problem version: given an instance and a threshold t, does there exist a set of items that fit in the knapsack with total value at least t?

Knapsack is an interesting case. Recall that we have a dynamic programming algorithm for knapsack that is better than the brute-force approach. However, the algorithm runs in time $O(nW)$ where n is the number of items and W is the weight limit of the knapsack. If we consider the number of bits in the input, it only takes $\log(W)$ bits to write down W, so the running time is actually exponential in the size of the input. However, this does show that NP-Complete problems can have better algorithms than pure brute-force search.

25.4 Proving a problem is NP-Complete

To prove a decision problem is NP-complete, we have to remember two steps:

1. Prove the problem is in NP.
2. Prove that an NP-Complete problem reduces to it.

Exercise 25.1. The *Hamiltonian Cycle* problem is, given a directed graph, to find a cycle that visits every vertex (except the start and end vertex) exactly once. Prove that Hamiltonian Cycle is NP-Complete.

Hint: reduce from Hamiltonian Path.

ℹ Example solution.

First, we prove the problem is in NP. The witness will be the cycle referenced in the problem. The verifier, given a graph and a list of vertices, checks that the list contains every vertex once except the start and end vertex, which equal each other, and that it is a valid cycle in the graph (i.e. follows existing edges).
Next, we reduce from Hamiltonian Path to Hamiltonian Cycle. Given an instance of Hamiltonian Path, we add a new vertex w to the graph and connect it both to and

from every existing vertex. If the previous graph had a Hamiltonian Path, then the new graph has a Hamiltonian Cycle by beginning at w, following the Path, and ending at w. Conversely, suppose the new graph has a Hamiltonian Cycle. It must contain w. By rearranging, we can put w first and last in the Hamiltonian Cycle, i.e. $w, v_1, \ldots, v_n, w$. Then the list $v_1, \ldots, v_n$ is a path that touches every vertex besides w, so it is a Hamiltonian Path in the original graph.