CS331: Algorithms and Complexity
Part I: Background

Kevin Tian

1 Introduction

This course is about the *design and analysis of algorithms*. An algorithm is a set of procedural instructions to perform some task. As stated, this definition seems to apply in almost any setting one can conceive of: we often would like "some task" to be completed, and of course to get it done, we would like to understand "ways of performing the task." Indeed, as we will see, the principles in this course apply to a surprisingly wide array of important computational tasks.[1]

In this course, we take the following view on what it means to understand a "way of performing a task" (i.e., *analyze an algorithm*). To analyze an algorithm, we must answer two questions.

1. **Correctness.** Does the algorithm actually complete the task we wanted?
2. **Complexity.** How many computational resources (e.g., time, space, energy, parallelism, randomness, ...) did the algorithm require to complete the task?

For the most part, we focus on runtime as our measure of complexity, but we will also discuss some of the other criteria mentioned above. The runtime of an algorithm gives us a concrete measuring stick by which we can evaluate it, and constrain its design. Almost every problem in this class will be of the form, for some computational task X, "Give an algorithm for solving X with runtime Y," or "Give an algorithm for solving X which runs as fast as possible." Sometimes, we replace runtime with a different measure of complexity. Later in the class, we instead consider the question of lower bounds rather than upper bounds on various complexity measures as well.

In this introductory set of notes, we provide preliminary tools which are frequently relevant in the rest of the course. These tools draw upon prerequisites for the course, and are as follows.

- **Proofs (Section 2).** In this course, the way we demonstrate the correctness, and establish the complexity, of an algorithm is through a mathematical proof. This may sound a lot scarier than it should — a proof is just a formal explanation of your argument. We describe the makings of a good proof, and recall several common proof strategies.
- **Asymptotics (Section 3).** We are interested in the scalability of algorithms: how do their complexities grow as a function of the input size? The idea is that for small inputs, anything will do the trick, so we really should be measuring algorithms by how they perform on large inputs. We discuss tools for bounding growing quantities, such as runtimes, asymptotically.
- **Graphs (Section 4).** Graphs are fundamental objects for describing relationships between objects (e.g., websites, locations, members of a social network, polygons in an animation, ...) and induce a wide array of useful computational tasks. We recall basic definitions.
- **Linear algebra (Section 5).** Matrices are basic ways of representing data. We recall basic definitions and structural properties of matrices, e.g., eigendecomposition.
- **Probability (Section 6).** Randomness can both be used in a problem assumption (e.g., a randomly sampled input) and as a tool in algorithm design. We recall basic definitions.
- **Data structures (Section 7).** Data structures store and manipulate data efficiently. We recall basic examples and their guarantees: lists, heaps, binary search trees, and hash tables.

[1] We informally use "computational task" to mean a task which we would like an automated solution for. For example, if the task is to count the number of times "the" appears on Wikipedia, one could in principle compute this number by hand. Of course, for such a large input text, we would prefer that a computer solve this problem.

Sections 2, 3, and 4 summarize material from CS 311, Section 5 summarizes material from Math 340L/Math 341/SDS 329C, Section 6 summarizes material from Math 362K/SDS 321, and Section 7 summarizes material from CS 314. Ideally, all of the tools we discuss are familiar to the reader. We include this introductory set of notes as a "crash course" reference, both to refresh the reader's memory, and to keep this course as self-contained as possible.

We aim to avoid tedium in this treatment, and so our descriptions are rather brief at times. The end of these notes provide references to more detailed expositions of everything covered here.

2 Proofs

This is a proof-based course; we do not consider an algorithm analyzed unless its correctness and complexity are proven (note that problems not about designing algorithms must have rigorous solutions as well). Mathematical proofs are simply formal ways of explaining your logic. They are essentially a form of essay, where you want to make a case for your argument, but your only move is to deduce a new statement from things you have already proven, along with standard math facts.

We will not be overly formal in this class; you do not need to justify any step of a proof that is straightforward, e.g., from basic calculations. However, we also want any logical leap that may not be obvious to the reader to be justified with a clear explanation. These are all very subjective terms, and practice helps significantly in striking the right balance between brevity and thoroughness. In this section, we first describe some basic principles of good proofwriting, and then give several examples of common proof techniques which one should always consider.

2.1 What is a good proof?

As discussed earlier, a proof is simultaneously an essay (in some sense, a piece of creative writing), and also must stick to a very rigid formula, of a sequence of logical deductions. Any proof that does not stick to this formula (e.g., makes an irrelevant statement or a false deduction) is either superfluous or wrong. Where the proofwriter has substantial freedom is in what sequence of deductions is chosen, and how these deductions are presented.

At the end of the day, proofs are a means to communicate ideas to a human reader, just like any other writing. If the reader cannot understand your point, the proof failed. As such, good proofs should adhere to general writing principles. However, there are a few characteristics specific to proofwriting. In [LLM10], overwhelmingly the conclusion is that a good proof should maintain a clear storyline. At any point, the reader should understand what the purpose of the currently developing argument is, and how it plays into the overall proof strategy. Towards this, please feel free to use any writing device you want; declaring the structure of the proof (your "game plan"), and explicitly portioning it off into smaller pieces or lemmas for clarity, often is useful.

As additionally mentioned by [LLM10], it is important to look back once the proof is done, for opportunities to revise and simplify. Often, we can introduce unnecessary complications as we are still working out the structure of the final proof. Removing these extra layers of indirection, and cutting to the meat of the argument, is an effective way to make your proof more readable.

2.2 Proof techniques

We now recall several common styles of arguments used in proofs, along with examples.

Implication. The simplest proof style is when the argument follows from a sequence of implications. That is, every statement is a direct consequence of one or more that came before it.

Lemma 1 (AM-GM inequality for 2 variables)**.** *Let $x, y \in \mathbb{R}_{\geq 0}$. Then, $\sqrt{xy} \leq \frac{x+y}{2}$.*

Proof. First, because squaring is a one-to-one, increasing function from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}$, the statement is equivalent to $xy \leq \frac{1}{4}(x^2 + 2xy + y^2)$. Rearranging, this in turn is equivalent to

$$\begin{aligned} xy \leq \frac{1}{4}(x^2 + 2xy + y^2) &\iff 4xy \leq x^2 + 2xy + y^2 \\ &\iff 0 \leq x^2 - 2xy + y^2 \iff 0 \leq (x-y)^2. \end{aligned}$$

The last statement is true for all $x, y \in \mathbb{R}_{\geq 0}$, concluding the proof. □

As we can see, every step of the proof of Lemma 1 directly follows from the previous one, along with standard manipulations (e.g., multiplying both sides by 4) which do not need to be justified. However, even in a direct proof, it can sometimes be helpful to rearrange the presentation of deductions to make it simpler for the reader to understand the proof structure.

Lemma 2 (Cauchy-Schwarz inequality)**.** *Let* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$*. Then,*[2]

$$\left(\sum_{i\in[d]} \mathbf{x}_i \mathbf{y}_i\right)^2 \leq \left(\sum_{i\in[d]} \mathbf{x}_i^2\right)\left(\sum_{i\in[d]} \mathbf{y}_i^2\right). \tag{1}$$

Proof. We first simplify the problem. Let

$$\mathbf{u} := \frac{\mathbf{x}}{\sqrt{\sum_{i\in[d]} \mathbf{x}_i^2}}, \ \mathbf{v} := \frac{\mathbf{y}}{\sqrt{\sum_{i\in[d]} \mathbf{y}_i^2}}. \tag{2}$$

We claim that it is enough to prove that $\sum_{i\in[d]} \mathbf{u}_i \mathbf{v}_i \leq 1$. Indeed,

$$\sum_{i\in[d]} \mathbf{u}_i \mathbf{v}_i \leq 1 \iff \sum_{i\in[d]} \mathbf{x}_i \mathbf{y}_i \leq \sqrt{\sum_{i\in[d]} \mathbf{x}_i^2}\sqrt{\sum_{i\in[d]} \mathbf{y}_i^2}$$

which yields (1) by squaring both sides. Now we conclude by showing $\sum_{i\in[d]} \mathbf{u}_i \mathbf{v}_i \leq 1$:

$$\sum_{i\in[d]} \mathbf{u}_i \mathbf{v}_i \leq \sum_{i\in[d]} \left(\frac{\mathbf{u}_i^2 + \mathbf{v}_i^2}{2}\right) = \frac{1}{2}\left(\sum_{i\in[d]} \mathbf{u}_i^2 + \sum_{i\in[d]} \mathbf{v}_i^2\right) = 1. \tag{3}$$

The inequality applied Lemma 1 for each $(x, y) \leftarrow (\mathbf{u}_i^2, \mathbf{v}_i^2)$; we then used the definitions (2). □

In proving Lemma 2, we noticed that when the problem is scaled so the right-hand side is $= 1$, it is much simpler to handle (using Lemma 1), so we first reduced to this case. There is also a more direct proof that plugs the definitions of $\mathbf{u}$ and $\mathbf{v}$ into (3) and expands. We instead portioned out the normalization (2), to make it more clear what this scaling is doing.

Incidentally, our proof of Lemma 2 is an example of a *reduction*: to solve the problem, it is enough to prove the simpler statement, $\sum_{i\in[d]} \mathbf{u}_i \mathbf{v}_i \leq 1$ for $\mathbf{u}, \mathbf{v}$ in (2). This method of problem solving, which leans upon existing building blocks to create more complex results ("how can I reduce the problem to something I already know how to solve?") is a basic pillar of algorithm design.

Contrapositive. Oftentimes, you will encounter statements of the form "if P, then Q" (or, for shorthand, "$P \implies Q$"). These are *implications*, which means that whenever P is true, Q is as well. It turns out that every implication has an equivalent reformulation, called a *contrapositive*. The contrapositive of $P \implies Q$ is $\neg Q \implies \neg P$, where $\neg$ means "not."[3]

Why are these statements equivalent? One can imagine that a priori, there are four possible states of the world, corresponding to the truth table of P and Q (e.g., one state is if P is false and Q is true). The implication $P \implies Q$ just means that we are ruling out one of these four states: if P is true, then Q cannot be false. This means that if Q is false, the only possible state of the world is if P is also false. This is exactly what the contrapositive claims.

When you encounter an implication that you do not know how to prove, it is worthwhile to consider whether the (equivalent) contrapositive is simpler. For example, it is often easier to prove that a number is composite (give a factorization) than to prove it is prime. To demonstrate this, we prove the following interesting property of *Mersenne primes* by considering its contrapositive.[4]

[2] We use $[d]$ to denote the natural numbers $1 \leq i \leq d$.

[3] For example, consider the (arguably true) implication "if you go to class, CS 331 notes will make sense." The contrapositive of this statement is "if CS 331 notes do not make sense, you did not go to class."

[4] As a fun fact, Mersenne primes have useful properties which make them important in computation. For example, the default pseudorandom number generation used by Python (the *Mersenne twister*) builds upon these properties.

Lemma 3. *If p is prime, and $p = 2^n - 1$ for some $n \in \mathbb{N}$, then n is also prime.*

Proof. The statement is an implication of the form "p is prime $\implies$ n is prime," so it suffices to prove the contrapositive, which is "n is not prime $\implies$ p is not prime." If $n \in \mathbb{N}$ is not prime, there are two cases: either $n = 1$, or n is composite.

Case 1: $n = 1$. In this case, $p = 2^n - 1 = 1$, so p is indeed also not prime.

Case 2: n is composite. In this case, $n = ab$ for some $a, b \in \mathbb{N}$ with $1 < a, b < n$. Then,

$$p = 2^{ab} - 1 = (2^a - 1)\left(\sum_{i=0}^{b-1} 2^{ia}\right). \tag{4}$$

Both terms on the right-hand side in (4) are natural numbers in $(1, p)$, so p is also composite. □

If the formula (4) is not familiar to the reader (it is a generalization of $x^2 - 1 = (x - 1)(x + 1)$ and $x^3 - 1 = (x - 1)(x^2 + x + 1)$), we formally prove it in Lemma 5 as an example of induction in proofs. The proof of Lemma 3 also was a good example of using casework, i.e., when there are multiple cases handled in different ways. In such situations, it is good practice to explicitly declare your different cases, and address them each in turn (clearly labeled).

Contradiction. Another useful proof technique is contradiction. Unfortunately, it shares a similar name with the contrapositive, another common proof technique, leading to some confusion. The goals of these two techniques are somewhat different. As we mentioned, the contrapositive is just an equivalent way of restating the exact (implication) statement one was trying to prove.

On the other hand, a proof by contradiction approaches proving a statement X by asking, what if X was not true? It then shows that if X were false, some other blatantly wrong thing would happen. The only "leap of faith" taken in arriving at this blatantly wrong conclusion was the assumption that X is false, so X must have in fact been true. Here is one of my favorite examples.

Lemma 4. *Let $n \in \mathbb{N}$. Consider a collection of $2n$ points in the plane, the first n of which, R, are colored red, and the last n of which, B, are colored blue. Suppose no three points in $R \cup B$ are collinear. Then, there is a way of pairing up points in R and B, such that if each pair is connected with a straight line segment, no two line segments intersect.*

Proof. There are only finitely many ways to pair up R and B. Therefore, amongst all of these possible pairings, there is one that minimizes the sum of the resulting n line segment lengths. We claim this minimum length pairing has no intersections.

We prove our claim by contradiction. Suppose it is false, which means the minimum length pairing has an intersection $x = rb \cap r'b'$, between the segments rb and $r'b'$, where r, r' are red points and b, b' are blue points. Then, consider an alternative pairing which leaves all other pairs untouched, but pairs r with b' and r' with b. This new pairing has a smaller total length than the old one:

$$|r'b| + |rb'| < (|r'x| + |bx|) + (|rx| + |b'x|) = (|rx| + |bx|) + (|r'x| + |b'x|) = |rb| + |r'b'|,$$

where we used the triangle inequality, and $|pq|$ denotes the length of the segment between p and q. Notice that our uses of the triangle inequality are strict ($<$ rather than $\leq$) because no three points are collinear. Thus, the old pairing was not minimum length, a contradiction. □

Interestingly, a proof of $P \implies Q$ which uses the contrapositive $\neg Q \implies \neg P$ is also a proof by contradiction. To prove the implication by contradiction, we assume $P \implies Q$ is false (which means there is a world where P and $\neg Q$ are both true) and try to arrive at an absurd scenario as a conclusion. If we prove the contrapositive $\neg Q \implies \neg P$, the absurd scenario is that if P and $\neg Q$ are both true, then both P and $\neg P$ are true. The proof of Lemma 4 is an example; it is a good exercise to think about what the contrapositive we implicitly proved was.

In general, contradiction is a much more versatile proof technique. In particular, there is substantial flexibility in what absurd scenario we arrive at to conclude the proof; it can be any false statement, and does not have to be of the form "P and $\neg P$ are both true."

Induction. The final proof technique we discuss is induction. Induction is useful when we wish to prove infinitely many statements, each parameterized by a natural number $n \in \mathbb{N}$ (for example, see Lemma 5). Let $S(n)$ denote the n^{th} such statement, so our goal is to prove $S(n)$ for all $n \in \mathbb{N}$. One can think of induction as creating mechanisms for proving all of these statements formulaically, by reusing work we have already done. The important question to ask at the end of the proof is: do the mechanisms we created allow us to establish $S(n)$, for any $n \in \mathbb{N}$?

The simplest example is usually referred to as just "induction." The idea of induction is to first prove $S(1)$ (the "base case"), and then establish the mechanism $S(n-1) \implies S(n)$ for all $n \geq 2$. Suppose we have both of these pieces (the base case and induction mechanism) proven; let us show that $S(100)$ then holds as an example. By using the mechanism, $S(100)$ holds if $S(99)$ holds. Applying the mechanism again, $S(99)$ holds if $S(98)$ holds, and so on. Finally, $S(2)$ is true if $S(1)$ is true, and we already proved $S(1)$. Therefore, after tracing back our sequence of implications, $S(100)$ is also true. The number $n = 100$ is clearly arbitrary, and the same logic holds for any $n \in \mathbb{N}$. We next give an example of this basic induction strategy.

Lemma 5. *Let $x \in \mathbb{R}$. For any $n \in \mathbb{N}$,*

$$x^n - 1 = (x-1)\left(\sum_{i=0}^{n-1} x^i\right). \tag{5}$$

Proof. We proceed by induction. When $n = 1$, the statement (5) reads $x - 1 = x - 1$, which is clearly true. Now supposing (5) is true for $n-1$, the claim for n follows from:

$$\begin{aligned} x^n - 1 &= \left(x^n - x^{n-1}\right) + \left(x^{n-1} - 1\right) \\ &= (x-1)x^{n-1} + (x-1)\left(\sum_{i=0}^{n-2} x^i\right) = (x-1)\left(\sum_{i=0}^{n-1} x^i\right). \end{aligned}$$

The second equality used the inductive hypothesis $x^{n-1} - 1 = (x-1)(\sum_{i=0}^{n-2} x^i)$. □

More generally, we are not limited to mechanisms of the form $S(n-1) \implies S(n)$; in a proof by induction, we can create any mechanisms we want, as long as they are true statements, and eventually establish all of the $S(n)$. An example is "strong induction," which relies on the mechanism $\bigwedge_{i=1}^{n-1} S(i) \implies S(n)$, where the notation $\bigwedge_{i=1}^{n-1} S(i)$ means all $S(i)$ hold for $1 \leq i \leq n-1$.

As a sanity check, assuming we have proven $S(1)$, the strong induction mechanism establishes $S(100)$ as follows: $S(1)$ implies $S(2)$, and then $S(3)$ is also implied (because $S(1)$ and $S(2)$ are now true). Continuing, we establish all of $S(1), S(2), \ldots, S(99)$, and then we can apply the mechanism to obtain $S(100)$. We conclude with a well-known example of strong induction.

Lemma 6. *Let $n \in \mathbb{N}$ with $n \geq 3$, and let T be a triangulation of a polygon P in the plane with n vertices. Then T consists of $n-2$ triangles.*

Proof. The base case is $n = 3$, in which case P is already a triangle, so T consists of $n-2$ triangles.

Now, suppose the statement is true for any polygon with k sides, for $3 \leq k \leq n-1$ (the strong induction hypothesis). Consider an arbitrary triangle A in a triangulation of polygon P with n sides. Depending on if the triangle shares zero, one, or two sides with P itself, removing the triangle splits P into either one, two, or three pieces, so we proceed by casework.

One piece. In this case, A shares two sides with P, and removing it deletes one vertex of P, so the remaining polygon has $n-1$ vertices. Thus, it is triangulated with $n-3$ triangles by the strong induction hypothesis. Adding A back gives $n-2$ triangles.

Two pieces. In this case, A shares one side with P, and two polygons remain with a and b sides respectively. We claim $a + b = n + 1$. To see this, $a + b$ is the number of sides remaining after removing A. Another way of counting this number is: A adds two diagonals to the original polygon (for a total of $n+2$ sides), and one of the sides is then removed, giving $n+1$ sides.

Finally, observe that any triangulation of an a-vertex polygon has $a-2$ triangles by the strong induction hypothesis, and similarly any triangulation of a b-vertex polygon has $b-2$ triangles.

Combined with the triangle A we removed, the total number of triangles in T is the desired

$$(a-2)+(b-2)+1 = a+b-3 = n-2.$$

Three pieces. In this case, A shares no sides with P (it is formed only using interior sides in T). Removing A splits P into three pieces, with a, b, and c sides respectively. A similar argument as before shows $a+b+c = n+3$. By strong induction, the total number of triangles is

$$(a-2)+(b-2)+(c-2)+1 = a+b+c-5 = n-2.$$

□

3 Asymptotics

Here we summarize the basics of asymptotics ("big O notation"). This notation is used frequently throughout the course to bound various quantities, e.g., runtimes. We provide some helpful tips which, ideally, should make understanding and applying asymptotics more painless.

The main motivation for asymptotic notation is to assess the scalability of algorithms. Often, we are in a situation where if the input size n is small, just about any algorithm (e.g., a naïve brute force choice) will complete the job in a reasonable amount of time. So how should we pick between algorithms? The principle behind asymptotics is that we should measure the performance of algorithms based on large inputs. Therefore, we study their behavior as $n \to \infty$.

Another motivation for viewing runtimes through the lens of asymptotics is the structure of algorithms. An algorithm often uses multiple subroutines, each with a different purpose (just think back to any complicated program you have written). If our goal is to speed up the overall program, we should understand which subroutine bottlenecks the runtime (e.g., if one subroutine takes up 90% of the time, we must improve it to have hope of substantial gains). Asymptotics quantify which runtimes we expect to dominate, so we do not overly "sweat the small stuff."

3.1 Definitions

We begin with the formal definitions of big O notation that we use in this course.

Definition 1. *Let $f(n)$, $g(n)$ be positive[5] functions on all $n \in \mathbb{N}$ with $n \geq n_0$, for a constant n_0.*

- *We write $f(n) = O(g(n))$ iff there is a constant $C > 0$ such that $f(n) \leq Cg(n)$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*
- *We write $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.*
- *We write $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*
- *We write $f(n) = o(g(n))$ iff $f(n) \neq \Omega(g(n))$.*
- *We write $f(n) = \omega(g(n))$ iff $f(n) \neq O(g(n))$.*

This is definitely a lot to digest. It is helpful to first understand the case of $g(n) = 1$.

Comparing to a constant. We have the following helpful characterizations.

Lemma 7. *Let $f(n)$ be a positive function on all $n \in \mathbb{N}$ with $n \geq n_0$ for a constant n_0.*

- *$f(n) = O(1)$ iff there is a constant $C > 0$ such that $f(n) \leq C$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*
- *$f(n) = \Omega(1)$ iff there is a constant $C > 0$ such that $f(n) \geq C$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*
- *$f(n) = \Theta(1)$ iff there are constants $C, C' > 0$ such that $C \leq f(n) \leq C'$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*
- *$f(n) = o(1)$ iff there is no constant $C > 0$ such that $f(n) \geq C$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*
- *$f(n) = \omega(1)$ iff there is no constant $C > 0$ such that $f(n) \leq C$, for all $n \in \mathbb{N}$ with $n \geq n_0$.*

[5] In this course, we always use asymptotic notation to describe positive quantities such as time or space.

Proof. All of these are clear from Definition 1 except potentially the statement about Ω. To see this, suppose $f(n) = \Omega(1)$, which implies $1 = O(f(n))$. Hence, there is a constant $C' > 0$ such that $1 \leq C' f(n)$ for $n \geq n_0$, so $f(n) \geq C := \frac{1}{C'}$ for $n \geq n_0$, as claimed. □

Lemma 7 has relatively simple interpretations. The first three statements are just about always being bounded by some constants, on all large enough inputs $n \geq n_0$. The last two statements are limits: they respectively say that $\lim_{n\to\infty} f(n) = 0$ and $\lim_{n\to\infty} f(n) = \infty$.

Beyond constants. The cool thing is that we can always reduce to the case of constants via division. So as long as we understand division and Lemma 7, we can compare any two functions.

Lemma 8. *Let $f(n), g(n)$ be positive functions on $n \in \mathbb{N}$ with $n \geq n_0$ for a constant n_0. Then $f(n) = O(g(n))$ iff $\frac{f(n)}{g(n)} = O(1)$; an analogous claim holds for each comparison in Definition 1.*

Proof. All the proofs are similar, so we only prove the statement for O. If $f(n) = O(g(n))$, then $f(n) \leq Cg(n)$ for all $n \geq n_0$ and some constant $C > 0$. This implies that $\frac{f(n)}{g(n)} \leq C \cdot 1$ for all $n \geq n_0$, so $\frac{f(n)}{g(n)} = O(1)$. The same proof works in reverse as well. □

On the n_0 parameter. Perhaps the most annoying thing about Definition 1 is the n_0 parameter. This part of the definiton can be helpful in proofs (e.g., if you are struggling to prove an inequality unless n is large enough), but it feels a bit unwieldy, so we wish to demystify it here.

To the best of my knowledge, the reason for including n_0 in the definition is almost entirely due to the inconvenient edge case $\log(1) = 0$. In particular, log is one of the most common functions encountered in algorithm design (discussed in Section 3.2), but if we do not include the $n \geq n_0$ restriction, no positive function $f(n)$ is $O(n \log(n))$, because $f(1) > 0 = 1 \cdot \log(1)$.

3.2 Basic examples

In principle, one can apply the theory of asymptotics to all sorts of ill-behaved functions. Fortunately, in this course, we will basically only consider three types of functions (beyond constants, of course): polynomials, exponentials, and logarithms. In my experience as an algorithms researcher, one rarely requires understanding the asymptotic behavior of any other function.

In this section, we give some basic rules for comparing these standard function types.

Lemma 9. *Let $f(n), g(n)$ be polynomials with positive leading coefficients. Then,*

$$f(n) = \begin{cases} o(g(n)) & \deg(f) < \deg(g) \\ \Theta(g(n)) & \deg(f) = \deg(g) \\ \omega(g(n)) & \deg(f) > \deg(g) \end{cases}.$$

Proof. We apply Lemma 7 and Lemma 8, along with the well-known fact that for polynomials f, g with positive leading coefficients A, B respectively,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \deg(f) < \deg(g) \\ \frac{A}{B} & \deg(f) = \deg(g) \\ \infty & \deg(f) > \deg(g) \end{cases}.$$

To see that the middle case above implies $\frac{f(n)}{g(n)} = \Theta(1)$, the limit definition implies that there is a constant $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $\frac{A}{2B} \leq \frac{f(n)}{g(n)} \leq \frac{2A}{B}$; note that these are both constants. □

For example, if $f(n) = 3n+18$ and $g(n) = 0.3n^2+99n-50$, we can apply Lemma 9 and immediately conclude that $f(n) = o(g(n))$, just by observing that $\deg(f) = 1$, $\deg(g) = 2$.

Lemma 10. *For any positive constants a, b, $\log^a(n) = o(n^b)$.*

Proof. By Lemma 7 and Lemma 8, we wish to show that

$$\lim_{n\to\infty} \frac{\log^a(n)}{n^b} = 0 \iff \lim_{n\to\infty} \frac{\log(n)}{n^{\frac{b}{a}}} = 0,$$

because a sequence of positive numbers vanishes in the limit iff all positive powers of the sequence also vanish; here we took the $\frac{1}{a}^{\text{th}}$ power. Finally, by L'Hôpital's rule, we have

$$\lim_{n\to\infty} \frac{\log(n)}{n^{\frac{b}{a}}} = \frac{a}{b} \cdot \lim_{n\to\infty} \frac{1}{n^{\frac{b}{a}}} = 0.$$

□

We hence have from Lemma 10 that polynomials of any degree grow much *faster* than logarithms. The simplifying trick we used of "taking a power of both sides" is very general; here is another use.

Lemma 11. *For any constants $a > 0, b > 1$, $n^a = o(b^n)$.*

Proof. Let $c := b^{\frac{1}{a}}$. By Lemma 7 and Lemma 8, we wish to show that

$$\lim_{n\to\infty} \frac{n^a}{b^n} = 0 \iff \lim_{n\to\infty} \frac{n}{c^n} = 0.$$

Here, we again took the $\frac{1}{a}^{\text{th}}$ power of each term. Now it suffices to apply L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{n}{c^n} = \lim_{n\to\infty} \frac{1}{\log(c)c^n} = 0.$$

□

Therefore, polynomials also grow much *slower* than arbitrary exponentials. It is particularly convenient that the three main function types we encounter in algorithm design form a hierarchy, such that each type completely dominates or is completely dominated by each other type.

For completeness, we mention that there are natural functions growing faster than any polynomial, but slower than any exponential. For example, $f(n) = \exp(\sqrt{\log(n)})$ is such a function.

3.3 More examples

We give a few more examples that capture important facts, and illustrate useful techniques for approaching asymptotics. The first points out that asymptotically, the maximum and the sum of two functions behave similarly. Therefore, if you ever encounter a maximum, consider replacing it with the sum, which often simplifies expressions.

Lemma 12. *For any positive functions $f(n), g(n)$ on $n \in \mathbb{N}$ with $n \geq n_0$, for a constant n_0,*

$$\max(f(n), g(n)) = \Theta\left(f(n) + g(n)\right).$$

Proof. For all $n \geq n_0$, $\frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$. □

The next showcases the power of guessing a well-behaved bounding function. For example, the Fibonacci sequence has a somewhat strange growth pattern, but by comparing it to simpler functions whose growth pattern we understand, we can still characterize its behavior.

Lemma 13. *Let $f(n)$ be the Fibonacci sequence (where we implicitly take $f(0) = 1$), defined by $f(1) = 2$, $f(2) = 3$, and $f(n) = f(n-1) + f(n-2)$ for all $n \geq 3$. Then, $\log(f(n)) = \Theta(n)$.*

Proof. We will prove that there are $C, C' > 0$ with

$$Cn \leq \log(f(n)) \leq C'n \text{ for all } n \in \mathbb{N},$$

(i.e., $n_0 = 1$). Equivalently, we wish to sandwich $f(n)$ between exponentials $\exp(Cn)$ and $\exp(C'n)$. We choose $C' = \log(2)$, $C = \log(1.5)$, so that the claim is $1.5^n \leq f(n) \leq 2^n$ for all $n \in \mathbb{N}$.

We first prove by strong induction that $f(n) \leq 2^n$ for all $n \in \mathbb{N}$. The base cases, $n = 1, 2$ are true by inspection. Now, supposing that $f(n-2) \leq 2^{n-2}$ and $f(n-1) \leq 2^{n-1}$ for some $n \geq 3$, we can inductively conclude that $f(n) \leq 2^{n-2} + 2^{n-1} \leq 2^n$ as desired.

Next, we again prove by strong induction that $f(n) \geq 1.5^n$ for all $n \in \mathbb{N}$. We directly verify the cases $n = 1, 2$. For $n \geq 3$, if $f(n-2) \geq 1.5^{n-2}$ and $f(n-1) \geq 1.5^{n-1}$, then we similarly have $f(n) \geq 1.5^{n-2} + 1.5^{n-1} = 1.5^n(\frac{1}{1.5^2} + \frac{1}{1.5}) > 1.5^n$ as desired, because $\frac{1}{1.5^2} + \frac{1}{1.5} = \frac{10}{9} > 1$. □

Finally, we mention the use of integration as a tool for understanding asymptotics.

Lemma 14. *Let $f(n) = n!$. Then, $\log(f(n)) = \Theta(n \log(n))$.*

Proof. Our goal is to provide asymptotic bounds for $\log(1) + \log(2) + \ldots + \log(n)$. To get a handle on this expression, we could first use the approximation (e.g., via an appropriate Riemann sum)

$$\begin{aligned}\log(1) + \log(2) + \ldots + \log(n) &\approx \int_1^2 \log(t)\mathrm{d}t + \int_2^3 \log(t)\mathrm{d}t + \ldots + \int_n^{n+1} \log(t)\mathrm{d}t \\ &= \int_1^{n+1} \log(t)\mathrm{d}t \approx n\log(n).\end{aligned}$$

Therefore, it is reasonable to guess that the growth behavior of $\log(f(n))$ should be similar to $n\log(n)$. To actually prove this asymptotic bound, observe that

$$\begin{aligned}\log(1) + \log(2) + \ldots + \log(n) &\leq n\log(n), \\ \log(1) + \log(2) + \ldots + \log(n) &\geq \frac{n}{2}\log\left(\frac{n}{2}\right) \geq \frac{n}{3}\log(n).\end{aligned}$$

The first inequality uses that each of the n terms on the left-hand side is at most $\log(n)$, and the second inequality uses that at least half the terms on the left-hand side are $\geq \log(\frac{n}{2})$. Finally, the last inequality holds for all $n \geq n_0 := 10$, so we are done. □

4 Graphs

Graphs are basic objects in computer science, with many fundamental applications. It is safe to say that graph algorithms is one of the deepest and most successful subareas in theoretical computer science, which has led to a plethora of powerful tools at our disposal for performing computations on graphs. Any time your application involves objects that can be compared or related in some way, it is worth considering representing the objects with a graph. The objects could be people in a community (with edges as connections), or websites on the internet (with edges as hyperlinks), or locations on a map (with weighted edges as distances), etc.

In Part V of the course notes, we go at depth into many of the exciting uses of graph algorithms. Our goal here is to recall some standard definitions and properties of graphs.

4.1 Definitions

In this course, we define graphs G by their *vertices* V, *edges* $E \in V \times V$, and *edge weights* $\mathbf{w} \in \mathbb{R}^E$, writing $G = (V, E, \mathbf{w})$ for short. Whenever $\mathbf{w}$ is not specified, i.e., $G = (V, E)$, we always assume that $\mathbf{w}$ is the all-ones vector, so every edge weight is 1. In this case, we call graph G *unweighted*.

When discussing a graph $G = (V, E, \mathbf{w})$ in an algorithmic context, we reserve the use of the letters n and m to mean the number of vertices and edges in G, i.e., $n := |V|$ and $m := |E|$. For $n \in \mathbb{N}$, we also use the notation $[n]$ to mean the set $\{i \in \mathbb{N} \mid i \leq n\} = \{1, 2, \ldots, n\}$. For a graph with n vertices, we often identify its vertices with the set $[n]$.

Graphs can either be *undirected* or *directed*. If left unspecified, we always assume graphs are undirected by default. We will always declare directed graphs explicitly (e.g., "let $G = (V, E, \mathbf{w})$ be a directed graph..."). Here are the key differences between directed and undirected graphs.

- In directed graphs $G = (V, E, \mathbf{w})$, each $(u, v) \in E$ has an orientation, pointing from u (the *tail*) to v (the *head*). So, edges are ordered tuples $(u, v) \in V \times V$.

- In undirected graphs $G = (V, E, \mathbf{w})$, each $(u, v) \in E$ is orientationless. So, edges are unordered tuples, and we treat $(u, v) \in V \times V$ and $(v, u) \in V \times V$ as identical objects.

If v is a vertex in graph $G = (V, E, \mathbf{w})$, we use $\deg(v) := \sum_{e=(v,u)\in E} \mathbf{w}_e$ to denote the degree of v. In particular, if G is undirected, $\deg(v)$ is the sum of weights of all edges with v as an endpoint, and if G is directed, $\deg(v)$ is the sum of weights of all edges with v as their tail. In unweighted undirected graphs, $\deg(v)$ is simply the number of neighbors of v (vertices adjacent to v).

Graphs can also either be *simple graphs* or *multigraphs*. In this course, we follow the convention that a simple graph is a graph with no self loops and no parallel edges, and that a multigraph is any graph that is not simple. If left unspecified, we always assume graphs are simple by default. Note that simple graphs with n vertices can have at most $n(n-1) \leq n^2$ edges.

For example, if $V = [6] := \{1, 2, \ldots, 6\}$, we could have $E = \{(1, 2), (3, 5), (1, 6), (4, 2), (2, 5), (6, 1)\}$. If $G = (V, E)$ is undirected, then G would be a multigraph (due to the presence of the parallel edges $(1, 6)$ and $(6, 1)$). If G were instead directed, then G would be simple.

We say that a graph $H = (V', E', \mathbf{w}')$ is a subgraph of another graph $G = (V, E, \mathbf{w})$ if one can obtain H by deleting some edges and vertices from G, i.e., $V' \subseteq V$, $E' \subseteq E$, and all edges in E' inherit their weights and directedness or undirectedness from E. A subgraph is called *induced* if any edge between two undeleted vertices is kept.

There are a few important types of graphs that we frequently encounter.

- **Paths:** A graph $G = (V, E, \mathbf{w})$ is a path if its vertices can be labeled by $[n]$ for $n := |V|$, so that E consists of the edges $(1, 2), (2, 3), \ldots, (n-1, n)$. Paths can be directed (in which case all the edges must point the same direction) or undirected.
- **Cycles:** A graph $G = (V, E, \mathbf{w})$ is a cycle if its vertices can be labeled by $[n]$ for $n := |V|$, so that E consists of the edges $(1, 2), (2, 3), \ldots, (n-1, n), (n, 1)$. Cycles can be directed (in which case all the edges must point the same direction) or undirected.
- **Trees and forests:** An undirected graph $G = (V, E, \mathbf{w})$ is a forest if it contains no cycles as subgraphs. A forest is a tree if it is *connected*, which means that for every pair of vertices $u, v \in V$, there is a path from u to v as a subgraph of G (i.e., v is reachable from u).

In this course, trees and forests will only be undirected, and we call a directed graph with no (directed) cycles as subgraphs a *directed acyclic graph* (DAG). Because trees are so fundamental to our development, we spend the next section specifically discussing them.

4.2 Trees

In this section, we provide some definitions that are specific to trees and forests.

In a *rooted tree* $T = (V, E, \mathbf{w})$, there is a special vertex $r \in V$ called the root. For each $v \in V$ with $v \neq r$, we define $\text{level}(v)$, the level of v, to be the number of edges on the unique path from r to v in G; the path is unique as otherwise, T would have a cycle. By default, we let $\text{level}(r) := 0$.

Rooted trees have a nice recursive structure, in terms of the level function. Let $v \in V$ be a non-root vertex in a rooted tree T with root r. For the unique path P from v to r, the neighbor of v along P is called its *parent*, and its other neighbors are called its *children*. We can thus characterize level as follows: for any non-root vertex v with parent u, $\text{level}(v) = \text{level}(u) + 1$.

We say a vertex v of a tree $T = (V, E, \mathbf{w})$ is a *leaf* if it has exactly one neighbor. The maximum value of $\text{level}(v)$ over vertices v of a rooted tree is called the *height* of the tree.

We say that a graph $T = (V, E, \mathbf{w})$ is a *binary tree* if the following properties hold.

1. T is a rooted tree, with root r, and r has two neighbors.
2. Every vertex $v \in V$ with $v \neq r$ has either one, two, or three neighbors.

Every non-root, non-leaf vertex v in a binary tree T thus has one parent and one or two children. We call such a v "internal." We label the children of an internal vertex "left" and "right" if there are two, and otherwise we assign the single child one of these labels arbitrarily.

If a binary tree has height h, it can have at most $2^h + 2^{h-1} + \ldots + 2 + 1 = 2^{h+1} - 1$ vertices. We call a binary tree with height h and $2^{h+1} - 1$ vertices a *complete binary tree* of height h.

In a rooted tree, if a vertex v can be reached from u by repeatedly moving to children, we say that v is a *descendant* of u and u is an *ancestor* of v. The subtree of a non-root vertex v is the induced subgraph on v and its descendants. We sometimes refer to the "left subtree" or "right subtree" of a non-leaf vertex in a binary tree: these are the subtrees of its left or right children.

We say an undirected graph is connected if every pair of vertices has a path between them. It is an exercise to check that if u, v are connected and t, u are connected, then t is also connected to v. Therefore, we can partition the vertices of any undirected graph G into subsets corresponding to *connected components*. Formally, each connected component is an induced subgraph of G, formed by taking a subset of vertices that are all connected and all edges between them, so that there are also no edges crossing between different connected components.

Here is a helpful characterization of forests, phrased in the language of connected components.

Lemma 15. *A tree with n vertices has $n - 1$ edges. More generally, a forest with n vertices and $k \in [n]$ connected components has $n - k$ edges.*

Proof. We first claim that every tree has a leaf. To see this, let P be any longest path in the tree, with endpoints u and v; there are a finite number of paths, so at least one of them is longest. We claim u is a leaf. Suppose for contradiction that u is not a leaf; it then has a neighbor t, other than its neighbor in the path. Clearly, $t \neq v$ (else the tree would contain a cycle), so we could add (t, u) to the path. Hence, P is not a longest path, a contradiction.

Next, we prove that a tree with n vertices has $n-1$ edges by induction. The claim is clear if $n = 1$. Suppose the claim is true for all trees with $n - 1$ vertices, for $n \geq 2$, and let T be a tree with n vertices. We wish to show T has $n - 1$ edges. Let v be an arbitrary leaf in T; we showed that at least one exists. If we delete v and its only adjacent edge from T, we obtain a tree with $n - 1$ vertices, which inductively has $n - 2$ edges. This is because T was acyclic, and removing edges from an acyclic graph cannot create a cycle. So, T has $(n - 2) + 1 = n - 1$ edges, as claimed.

Finally, the connected components of a forest are all acyclic. Thus, each connected component (which is both connected and acyclic) is a tree. If there are k trees in the forest with $n_1, n_2, \ldots, n_k$ vertices respectively, so that $n_1 + n_2 + \ldots + n_k = n$, the total number of edges in the forest is $(n_1 - 1) + (n_2 - 1) + \ldots + (n_k - 1) = (n_1 + n_2 + \ldots + n_k) - k = n - k$, as claimed. □

4.3 Representing a graph

In this course, unless otherwise specified, graph inputs to algorithms are given as *adjacency lists*. The adjacency list representation of unweighted graph $G = (V, E)$ consists of the following.

- Two lists L_V^{in} and L_V^{out} of length $n := |V|$ corresponding to vertices of G. The i^{th} element of L_V^{in} is a list of pointers to each edge of the form (u, v) in L_E, where v is the i^{th} vertex in V, i.e., all incoming edges to v. Similarly, L_V^{out} has pointers to all outgoing edges of each vertex.
- A list L_E of length $m := |E|$ corresponding to edges of G. The j^{th} element of L_E is a list of two pointers to u in L_V^{out} and v in L_V^{in}, where $e = (u, v)$ is the j^{th} edge in E. Note that in a directed graph, each edge explicitly remembers which vertices are the tail and head.

For example, consider our directed graph $G = (V, E)$ that we defined earlier, with $V = [6]$, and $E = \{(1, 2), (3, 5), (1, 6), (4, 2), (2, 5), (6, 1)\}$. The adjacency list representation of G would be:

$$\begin{aligned} L_V^{\text{in}} &= \{\{(6,1)\}, \{(1,2),(4,2)\}, \emptyset, \emptyset, \{(2,5),(3,5)\}, \{(1,6)\}\}, \\ L_V^{\text{out}} &= \{\{(1,2),(1,6)\}, \{(2,5)\}, \{(3,5)\}, \{(4,2)\}, \emptyset, \{(6,1)\}\}, \\ L_E &= \{\{1,2\}, \{3,5\}, \{1,6\}, \{4,2\}, \{2,5\}, \{6,1\}\}. \end{aligned}$$

Formally, the elements of L_V^{out}, L_V^{in}, and L_E are lists of pointers, with each pointer referencing an element of other lists. Assuming for simplicity that numbers and addresses take $O(1)$ space

to store, this shows that a graph can be represented in $O(m+n)$ space.[6] In particular, it is straightforward to check that L_V^{in} and L_V^{out} can be represented in $O(m+n)$ space, since each vertex takes $O(1)$ space to initialize a list, and each additional edge takes $O(1)$ space. Similarly, one can check that L_E can be represented in $O(m)$ space. The gold standard runtime for *graph algorithms*, or algorithms that take a graph as input, is $O(m+n)$, the size of the input.

In weighted graphs $G = (V, E, \mathbf{w})$, we similarly use adjacency lists, but slightly modify L_E to include an extra field per edge. The element of L_E corresponding to an edge $e = (u, v)$ with weight $\mathbf{w}_e$ contains pointers to u and v, as before, as well as a third field storing $\mathbf{w}_e$. Again, assuming weights take $O(1)$ space to store, the representation size is $O(m+n)$.

5 Linear algebra

In the modern era of computing, one burgeoning area for algorithms is data science. Typically in data science, the input is a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$, where the n rows each represent members of the dataset. The d entries of each row then correspond to various pieces of information about the associated member. For example, in linear regression, each entry is a feature which could be correlated with the response variable; if the response variable is likelihood to play professional basketball, features such as height, weight, age, etc. could be used.[7]

In Part VI of the course notes, we explore tools for manipulating matrices and solving other *continuous* algorithmic problems arising in data science. Here, we recall some relevant background from linear algebra. We omit most proofs for brevity, but highly recommend checking out the reference [Axl24], which contains full proofs and additional exposition on this material.

5.1 General notation

In this course, we represent matrices (with at least 2 rows and columns each) in capital boldface, to disambiguate them from vectors, which are denoted in lowercase boldface.

Let $\mathbf{A} \in \mathbb{R}^{n \times d}$ be a matrix, i.e., $\mathbf{A}$ has n rows, d columns, and real-valued entries. For $i \in [n]$, we refer to the i^{th} row of $\mathbf{A}$ by $\mathbf{A}_{i:} \in \mathbb{R}^d$, and for $j \in [m]$, we refer to the j^{th} column of $\mathbf{A}$ by $\mathbf{A}_{:j} \in \mathbb{R}^n$. By default, we treat these as column vectors; in particular, for any $k \in \mathbb{N}$, we use $\mathbb{R}^k$ synonymously with $\mathbb{R}^{k \times 1}$, and will specify that row vectors live in $\mathbb{R}^{1 \times k}$ explicitly.

When the dimension d is clear from context, we use $\mathbf{e}_i \in \mathbb{R}^d$ to denote the i^{th} *standard basis vector*, i.e., the vector which is zero in all coordinates, except it has a 1 in the i^{th} coordinate. We also let $\mathbf{0}_d$ denote the all-zeroes vector in $\mathbb{R}^d$, and $\mathbf{1}_d := \sum_{i \in [d]} \mathbf{e}_i$ denote the all-ones vector in $\mathbb{R}^d$. We similarly let $\mathbf{0}_{n \times d}$ denote the $n \times d$ all-zeroes matrix, and $\mathbf{I}_d$ denote the $d \times d$ identity matrix, which is the square matrix with ones along the diagonal and zeroes everywhere else.

The *transpose* of a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ is denoted $\mathbf{A}^\top$. In particular, $\mathbf{A}^\top \in \mathbb{R}^{d \times n}$, and for all $i \in [n]$, $j \in [d]$, the $(i,j)^{\text{th}}$ entry of $\mathbf{A}$ (denoted $\mathbf{A}_{ij}$) is the same as the $(j,i)^{\text{th}}$ entry of $\mathbf{A}^\top$ (denoted $\mathbf{A}^\top_{ji}$). We say matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ is *symmetric* if $\mathbf{A} = \mathbf{A}^\top$; note that symmetric matrices must be square.

For two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$, their *inner product*, or dot product, is $\langle \mathbf{u}, \mathbf{v} \rangle := \sum_{i \in [d]} \mathbf{u}_i \mathbf{v}_i$, where $\mathbf{v}_i$ means the i^{th} entry of a vector $\mathbf{v}$. We could also write this as $\mathbf{u}^\top \mathbf{v} = \langle \mathbf{u}, \mathbf{v} \rangle$.

More generally, we define the *matrix product* of $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $\mathbf{B} \in \mathbb{R}^{d \times k}$ by:

$$\mathbf{AB} \in \mathbb{R}^{n \times k}, \ [\mathbf{AB}]_{ij} = \mathbf{A}_{i:}^\top \mathbf{B}_{:j}, \text{ for all } i \in [n], \ j \in [k]. \tag{6}$$

This makes sense pictorially: $\mathbf{A}_{i:}^\top$ is the i^{th} row in $\mathbf{A}$, and $\mathbf{B}_{:j}$ is the j^{th} column in $\mathbf{B}$. As expected, $\mathbf{AB}$ has the same row count as $\mathbf{A}$, and the same column count as $\mathbf{B}$. Importantly, note that matrix multiplication only makes sense when $\mathbf{A}$'s rows have the same dimension as $\mathbf{B}$'s columns.

[6] Technically, vertex indices are numbers in $[n]$, and therefore require $O(\log(n))$ bits to describe. We will work in the word RAM model when discussing graph algorithms (discussed in more detail in Part II), and measure space usage in terms of the number of words, or memory registers corresponding to $O(\log(n))$-bit numbers.

[7] One could of course use categorical data for regression problems as well. However, increasingly, we see many different domains spanning vision, language, and audio, adopting a more continuous approach to representing data. For example, categorical data can first be pushed through a continuous transformation, e.g., Word2Vec.

The naïve algorithm for computing $\mathbf{AB}$ takes time $O(ndk)$, since each of the nk entries of $\mathbf{AB}$ takes $O(d)$ time to compute. As we will see, this is sometimes dramatically improvable.

The Euclidean norm of a vector $\mathbf{v} \in \mathbb{R}^d$ is denoted $\|\mathbf{v}\|_2$, and measures its straight-line distance to the origin $\mathbf{0}_d$, as seen through the Pythagorean theorem:

$$\|\mathbf{v}\|_2 := \sqrt{\sum_{i \in [d]} \mathbf{v}_i^2}.$$

Comparing with our earlier inner product definition, we can verify $\|\mathbf{v}\|_2^2 = \mathbf{v}^\top \mathbf{v}$ for any $\mathbf{v} \in \mathbb{R}^d$.

5.2 Linear subspaces

In this section, we go over the basics of *linear subspaces*. To define them, one must first understand the central notion of linear independence. We say that a set of vectors $\{\mathbf{a}_1, \ldots, \mathbf{a}_n\} \subset \mathbb{R}^d$ are *linearly dependent* if there exists a coefficient vector $\mathbf{c} \in \mathbb{R}^n$ which is not $\mathbf{0}_n$, and $\mathbf{c}_1\mathbf{a}_1 + \mathbf{c}_2\mathbf{a}_2 + \ldots + \mathbf{c}_n\mathbf{a}_n = \mathbf{0}_d$. If no such coefficients $\mathbf{c}$ exist, we say that the set of vectors is *linearly independent*.

One setting where people care a lot about linear dependences is solving systems of linear equations. Suppose you have a set of n linear equations in d unknowns, represented by $\mathbf{x} \in \mathbb{R}^d$:

$$\mathbf{a}_1^\top \mathbf{x} = \mathbf{b}_1, \ \mathbf{a}_2^\top \mathbf{x} = \mathbf{b}_2, \ \ldots, \ \mathbf{a}_n^\top \mathbf{x} = \mathbf{b}_n. \tag{7}$$

All the information in (7) can be more concisely summarized by the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n \times d}$ has $\mathbf{a}_i^\top$ as its i^{th} row, for all $i \in [n]$. We can view each of the n linear equations in (7) as a "hint" about the unknown vector $\mathbf{x}$. For example, if $\mathbf{a}_1 = \mathbf{e}_1$ is the first standard basis vector, then the linear equation $\mathbf{e}_1^\top \mathbf{x} = \mathbf{b}_1$ tells you the first coordinate of $\mathbf{x}$ is $\mathbf{b}_1$. More generally, each "hint" tells you some information restricting what the unknown $\mathbf{x}$ could be.

The point of linear dependence is to measure redundancy of these hints. If a set of vectors $\{\mathbf{a}_i\}_{i \in [n]} \subset \mathbb{R}^d$ is linearly dependent, i.e., $\mathbf{c}_1\mathbf{a}_1 + \mathbf{c}_2\mathbf{a}_2 + \ldots + \mathbf{c}_n\mathbf{a}_n = \mathbf{0}_d$ for coefficients $\{\mathbf{c}_i\}_{i \in [n]}$ that are not all zero, then assuming $\mathbf{c}_i \neq 0$, we can alternatively write

$$\mathbf{a}_i = -\frac{1}{\mathbf{c}_i} \sum_{\substack{j \in [n] \\ j \neq i}} \mathbf{a}_j \implies \mathbf{a}_i^\top \mathbf{x} = -\frac{1}{\mathbf{c}_i} \sum_{\substack{j \in [n] \\ j \neq i}} \mathbf{a}_j^\top \mathbf{x}.$$

So, if you knew the values of $\mathbf{a}_j^\top \mathbf{x}$ for all $j \neq i$, you could already deduce the value of $\mathbf{a}_i^\top \mathbf{x}$, and therefore it should not really be counted as a new "hint." Conversely, if $\{\mathbf{a}_i\}_{i \in [n]}$ are linearly independent, then each new $\mathbf{a}_i^\top \mathbf{x}$ that is measured gives new information about $\mathbf{x}$.

For a set of vectors $\{\mathbf{a}_i\}_{i \in [n]} \subset \mathbb{R}^d$, we define their span by

$$\text{Span}\left(\{\mathbf{a}_i\}_{i \in [n]}\right) := \left\{ \mathbf{v} \in \mathbb{R}^d \mid \mathbf{v} = \sum_{i \in [n]} \mathbf{c}_i \mathbf{a}_i \text{ for } \mathbf{c} \in \mathbb{R}^n \right\}. \tag{8}$$

Thus, $\text{Span}(\{\mathbf{a}_i\}_{i \in [n]})$ is all vectors which are linear combinations of $\{\mathbf{a}_i\}_{i \in [n]}$. Alternatively, it is the set of all "redundant hints" if already given linear measurements in $\{\mathbf{a}_i\}_{i \in [n]}$.

A linear subspace $S \subset \mathbb{R}^d$ is just the span of some set of vectors. It can alternatively be defined as any subset of $\mathbb{R}^d$ which has the property that for any two vectors $\mathbf{u}, \mathbf{v} \in S$ and scalars $\alpha, \beta \in \mathbb{R}$, $\alpha\mathbf{u} + \beta\mathbf{v}$ is also in S (i.e., linear subspaces are closed under linear combinations). The definition (8) indeed has this property, for any $\{\mathbf{a}_i\}_{i \in [n]}$. The less obvious direction is that any linear subspace can be written as a span. However, it turns out the following deep fact is true.

Fact 1. *Let $S \subset \mathbb{R}^d$ be a linear subspace. Then S is the span of any maximal set of linearly independent vectors $\{\mathbf{a}_i\}_{i \in [k]} \subset S \setminus \{\mathbf{0}_d\}$, i.e., a set such that including any other nonzero vector in S creates a linear dependence. Moreover, all such maximal sets $\{\mathbf{a}_i\}_{i \in [k]}$ have the same size.*

Fact 1 establishes the aforementioned connection between linear subspaces and spans, in that any subspace can be written as a span. The second part says if we want to understand the complexity of the linear subspace, i.e., the maximum number of independent "hints" it contains, there is no

ambiguity; no matter how we choose the hints, this number stays the same. We call this number the *dimension* of the linear subspace S, denoted by $\dim(S)$. We call any maximal set of linear independent vectors in S a *basis* for S. For example, the whole space $S = \mathbb{R}^d$ itself has dimension d, and $\{\mathbf{e}_i\}_{i\in[d]}$ is one basis for $\mathbb{R}^d$. This means that there does not exist a set of $d+1$ linearly independent vectors in $\mathbb{R}^d$, which should be familiar in the context of solving (7).

For a matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$, we call $\text{Span}(\{\mathbf{A}_{i:}\}_{i\in[n]})$ its *row span*, and $\text{Span}(\{\mathbf{A}_{:j}\}_{j\in[d]})$ its *column span*. It is a fundamental result in linear algebra that the row span and column span of any matrix have the same dimension. We call this dimension the *rank* of the matrix $\mathbf{A}$.

For any subspace, we can always find a basis satisfying a convenient property called *orthonormality*. Intuitively, orthonormality means that the basis can be treated as a coordinate system (e.g., $\{\mathbf{e}_i\}_{i\in[d]}$ is orthonormal). To motivate our definition, note that the Span operation has certain invariances. One is invariance to scaling: for example, $\text{Span}(\{\mathbf{a}_i\}_{i\in[k]}) = \text{Span}(\{2\mathbf{a}_i\}_{i\in[k]})$, since we can halve all coefficients in (8) when using the latter set, to get the same result.

Another is invariance to projection: for $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ and $\alpha \in \mathbb{R}$, $\text{Span}(\{\mathbf{u}, \mathbf{v}\}) = \text{Span}(\{\mathbf{u}, \mathbf{v} - \alpha\mathbf{u}\})$, so we can subtract a multiple of $\mathbf{u}$ from $\mathbf{v}$ to make it perpendicular to $\mathbf{u}$ and the span stays the same. For example, if $\mathbf{u} = \mathbf{e}_1$ and $\mathbf{v}$ is nonzero only in its first two coordinates (e.g., $\mathbf{v} = 0.1\mathbf{e}_1 + 1.2\mathbf{e}_2$), then for $\alpha = \mathbf{e}_1^\top \mathbf{v}$, $\mathbf{v}' = \mathbf{v} - \alpha\mathbf{e}_1$ is a multiple of $\mathbf{e}_2$ (e.g., $\alpha = 0.1$, $\mathbf{v}' = 1.2\mathbf{e}_2$). Clearly, $\text{Span}(\{\mathbf{u}, \mathbf{v}\}) = \text{Span}(\{\mathbf{u}, \mathbf{v}'\})$; both are all vectors in $\mathbb{R}^d$ with 0 as their last $d-2$ coordinates.

To state this phenomenon more generally, we require a definition. We say that a set $\{\mathbf{u}_i\}_{i\in[k]} \subset \mathbb{R}^d$ is *orthonormal* if the following conditions hold.

- For all $i \in [k]$, $\|\mathbf{u}_i\|_2 = 1$, where $\|\mathbf{v}\|_2$ is the Euclidean norm of $\mathbf{v} \in \mathbb{R}^d$.
- For all $i, j \in [k]$ with $i \neq j$, $\mathbf{u}_i$ and $\mathbf{u}_j$ are *orthogonal*, i.e., $\mathbf{u}_i^\top \mathbf{u}_j = 0$.

The first condition above captures invariance to scaling; we can normalize our basis vectors to unit length, without affecting their span (the "normal" part). The second condition above captures invariance to projection; this is a bit harder to see, but the idea is we can first subtract off any dependencies within our basis, to make it pairwise orthogonal (the "ortho" part).

We are now ready to state our second deep linear algebraic fact.

Fact 2. *For any set of linearly independent vectors $\{\mathbf{a}_i\}_{i\in[k]} \subset \mathbb{R}^d$, there is a set of orthonormal vectors $\{\mathbf{u}_i\}_{i\in[k]} \subset \mathbb{R}^d$ such that $\text{Span}(\{\mathbf{a}_i\}_{i\in[k]}) = \text{Span}(\{\mathbf{u}_i\}_{i\in[k]})$.*

Therefore, any linear subspace S has an orthonormal basis; we can first apply Fact 1 to find a set $\{\mathbf{a}_i\}_{i\in[k]}$ which spans S, and then apply Fact 2 to make the set orthonormal.

One concise way to represent orthonormality of a set $\{\mathbf{u}_i\}_{i\in[k]}$ is through the following equation:

$$\mathbf{U}^\top \mathbf{U} = \mathbf{I}_k, \text{ where } \mathbf{U} \in \mathbb{R}^{d\times k} \text{ has } \mathbf{U}_{:i} = \mathbf{u}_i \text{ for all } i \in [k]. \tag{9}$$

Let us walk through this notation together, as it is representative of the definitions we have built. The second half of the above expression just says $\mathbf{U}$ is a $d \times k$ matrix, whose i^{th} column is set to $\mathbf{u}_i$. Comparing to (6), the first half of the above expression says that for all $(i, j) \in [k] \times [k]$,

$$[\mathbf{U}^\top \mathbf{U}]_{ij} = \mathbf{U}_{:i}^\top \mathbf{U}_{:j} = \mathbf{u}_i^\top \mathbf{u}_j = [\mathbf{I}_k]_{ij}.$$

Because all the off-diagonal entries of $\mathbf{I}_k$ are 0, this implies $\mathbf{u}_i^\top \mathbf{u}_j = 0$ for $i \neq j$ (the "ortho" part); similarly, the on-diagonal equalities imply $\mathbf{u}_i^\top \mathbf{u}_i = \|\mathbf{u}_i\|_2^2 = 1$ (the "normal" part). Thus, $\{\mathbf{u}_i\}_{i\in[k]}$ is orthonormal iff $\mathbf{U}^\top \mathbf{U} = \mathbf{I}_k$. In the special case $k = d$, we have a set of d orthonormal vectors $\{\mathbf{u}_i\}_{i\in[d]} \subset \mathbb{R}^d$, satisfying $\mathbf{U}^\top \mathbf{U} = \mathbf{I}_d$. Pictorially, one should view such a set of vectors $\{\mathbf{u}_i\}_{i\in[d]}$ as a rotation of the standard orthonormal basis $\{\mathbf{e}_i\}_{i\in[d]}$; this rotation preserves lengths and pairwise orthogonality, but expresses $\mathbb{R}^d$ in a different coordinate system given by the new basis.

5.3 Spectral theory

We will require some basic facts about eigenvalues and eigenvectors in this course. Recall that $(\lambda, \mathbf{v})$ is an eigenvalue-eigenvector pair of a matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$ if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{10}$$

The first fact we need is that all *symmetric* matrices $\mathbf{A} \in \mathbb{R}^{d \times d}$ can be written in a special format.

Fact 3 (Spectral theorem)**.** *Let $\mathbf{A} \in \mathbb{R}^{d \times d}$ be symmetric. Then $\mathbf{A}$ has d eigenvalue-eigenvector pairs $\{\boldsymbol{\lambda}_i\}_{i \in [d]} \subset \mathbb{R}$, $\{\mathbf{v}_i\}_{i \in [d]} \subset \mathbb{R}^d$, where the $\{\mathbf{v}_i\}_{i \in [d]}$ are orthonormal, and*

$$\mathbf{A} = \sum_{i \in [d]} \boldsymbol{\lambda}_i \mathbf{v}_i \mathbf{v}_i^\top. \tag{11}$$

In other words, the eigenvectors of a symmetric matrix $\mathbf{A}$ are pairwise orthogonal, and induce the convenient form (11). The decomposition (11) is sometimes called an *eigendecomposition*.

As a sanity check, compare this formula with (10). For any $i \in [d]$, $\mathbf{v}_i$ is indeed an eigenvector:

$$\mathbf{A}\mathbf{v}_i = \left(\sum_{j \in [d]} \boldsymbol{\lambda}_j \mathbf{v}_j \mathbf{v}_j^\top \right) \mathbf{v}_i = \sum_{j \in [d]} \boldsymbol{\lambda}_j \mathbf{v}_j (\mathbf{v}_j^\top \mathbf{v}_i) = \boldsymbol{\lambda}_i \mathbf{v}_i,$$

since all terms except $j = i$ vanish, as $\mathbf{v}_j^\top \mathbf{v}_i = 0$ for $j \neq i$. Thus, (10) and (11) are consistent.

Finally, we give a generalization of Fact 3 to all (possibly asymmetric) matrices.

Fact 4 (Singular value decomposition)**.** *Let $\mathbf{A} \in \mathbb{R}^{n \times d}$ with $n \geq d$. Then, $\mathbf{A}$ has* singular values *$\{\boldsymbol{\sigma}_i\}_{i \in [d]} \subset \mathbb{R}_{\geq 0}$, and* left and right singular vectors *$\{\mathbf{u}_i\}_{i \in [d]} \in \mathbb{R}^n$, $\{\mathbf{v}_i\}_{i \in [d]} \in \mathbb{R}^d$, such that*

$$\mathbf{A} = \sum_{i \in [d]} \boldsymbol{\sigma}_i \mathbf{u}_i \mathbf{v}_i^\top. \tag{12}$$

Further, the sets $\{\mathbf{u}_i\}_{i \in [d]} \in \mathbb{R}^n$, $\{\mathbf{v}_i\}_{i \in [d]} \in \mathbb{R}^d$, are both orthonormal.

Fact 4 applies to tall matrices (i.e., $n \geq d$), but it also extends to wide matrices (i.e., $d \geq n$) by applying it to $\mathbf{A}^\top$. One useful fact is that $\mathbf{A}$'s rank is its number of nonzero singular values. To see why, note that whatever vector $\mathbf{x} \in \mathbb{R}^d$ is, the formula (12) shows that

$$\mathbf{A}\mathbf{x} = \sum_{i \in [d]} \left(\boldsymbol{\sigma}_i \mathbf{v}_i^\top \mathbf{x} \right) \mathbf{u}_i$$

must be some linear combination of the $\{\mathbf{u}_i\}_{i \in [d]}$ with nonzero corresponding $\boldsymbol{\sigma}_i$; suppose there are r such nonzero $\boldsymbol{\sigma}_i$. Then, the column span of $\mathbf{A}$ is spanned by an orthonormal basis of size r.

To build intuition for Fact 4, let us see how it relates to Fact 3. Any $\mathbf{A} \in \mathbb{R}^{n \times d}$ with $n \geq d$ has a singular value decomposition of the form $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$, where $\mathbf{U} \in \mathbb{R}^{n \times d}$ has columns $\{\mathbf{u}_i\}_{i \in [d]}$, $\mathbf{V} \in \mathbb{R}^{d \times d}$ has columns $\{\mathbf{v}_i\}_{i \in [d]}$, and $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}_{\geq 0}$ is a diagonal matrix, whose diagonal is $\boldsymbol{\sigma}$. Indeed, a slightly tedious expansion shows that under these definitions,

$$\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \sum_{i \in [d]} \boldsymbol{\sigma}_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{A},$$

matching the formula (12). Now, consider the matrix $\mathbf{A}^\top \mathbf{A} \in \mathbb{R}^{d \times d}$. This matrix is a natural matrix associated with $\mathbf{A}$; it is often called the *Gram matrix* of $\mathbf{A}$, and can be interpreted as a "second moment matrix," which is sort of a high-dimensional version of a variance. Anyways, the most important thing about the Gram matrix for our purposes is that it is *symmetric*. Therefore, we can use Fact 3. But actually, Fact 4 is even better! We can directly compute

$$\mathbf{A}^\top \mathbf{A} = \left(\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^\top \right) \left(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top \right) = \mathbf{V}\boldsymbol{\Sigma} \left(\mathbf{U}^\top \mathbf{U} \right) \boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^\top = \sum_{i \in [d]} \boldsymbol{\sigma}_i^2 \mathbf{v}_i \mathbf{v}_i^\top.$$

Here, we used $\mathbf{U}^\top \mathbf{U} = \mathbf{I}_d$ because $\mathbf{U}$ has orthonormal columns, as derived in (9).

So, Fact 4 tells us everything that Fact 3 stated about the symmetric matrix $\mathbf{A}^\top \mathbf{A}$. It actually further told us that all the eigenvalues $\boldsymbol{\lambda}_i = \boldsymbol{\sigma}_i^2$ are nonnegative. This means that the Gram matrix is an example of what are known as *positive semidefinite matrices*, the most important matrix family in applied mathematics other than symmetric matrices.

We will give more geometric interpretations of Facts 3 and 4 in Part VI of the course notes. These decompositions turn out to be crucial to visualizing and manipulating datasets stored as matrices.

6 Probability

Randomness can be an extremely useful resource in algorithm design. As a basic example you may have already seen, consider the QuickSort algorithm. A key step in QuickSort is solving the "approximate median" problem: given a list L of n integers, the goal is to return any element between the $(\frac{n}{4})^{\text{th}}$ largest and $(\frac{3n}{4})^{\text{th}}$ largest in L. Any deterministic algorithm for this problem requires querying the values of $\Omega(n)$ elements of L. However, returning $L[i]$ for a uniformly random index i solves the problem in $O(1)$ time, with a chance of about 50%!

In Part VII of the course notes, we give several examples of how randomness can significantly improve the performance of algorithms, as well as tools for reducing the chance that randomized methods fail. In this section, we recall basic definitions that we will use.

6.1 Definitions

The basic object in the study of probability is a *probability space*. Formally, a probability space is a triple consisting of $(\Omega, \mathcal{F}, \pi)$. It can be thought of intuitively as characterizing the behavior of a repeated experiment. The first component, Ω, is a "sample space" consisting of all possible outcomes of the experiment. The second component, $\mathcal{F}$, is a "event space" consisting of all possible interesting events we would like to understand; an event is just a subset of Ω. The last component, π, is a "probability measure," an assignment of probabilities to each outcome.

This all seems quite abstract, so it is helpful to explain an example. Suppose we wish to understand the behavior of an experiment where we toss 3 fair, two-sided coins. Specifically, we want to understand whether we obtain more heads or tails. This corresponds to two "interesting events" in $\mathcal{F}$: one consisting of all outcomes where the heads outnumber the tails, and one consisting of all other outcomes. Let us write a probability space for this experiment.

- $\Omega = \{\text{HHH}, \text{HHT}, \text{HTH}, \text{THH}, \text{HTT}, \text{THT}, \text{TTH}, \text{TTT}\}$ is the set of all possible outcomes.
- $\mathcal{F} = \{\{\text{HHH}, \text{HHT}, \text{HTH}, \text{THH}\}, \{\text{HTT}, \text{THT}, \text{TTH}, \text{TTT}\}\}$ is the event space. The first event is all outcomes with more heads than tails, and the second is the opposite.
- $\pi = \{\frac{1}{8}, \frac{1}{8}, \ldots, \frac{1}{8}\}$ is the uniform distribution over the 8 possible outcomes.

In this course, we only encounter probability spaces where Ω is finite (after all, we are studying finite-sized computers). Another example is $\Omega = [6]$, e.g., the possible sides that a dice could land on. We mention that all of the definitions here generalize to continuous (infinite) Ω as well.

For a finite set Ω, let $\mathcal{F}$ be the power set of outcomes, i.e., all subsets of Ω. Any subset of outcomes could be deemed an "interesting event." In this case, we can interpret the probability measure $\pi : \Omega \to \mathbb{R}_{\geq 0}$ as assigning masses to different elements of Ω, such that the total mass is fixed: $\sum_{\omega \in \Omega} \pi(\omega) = 1$. Then, the probability of an event $\mathcal{E} \in \mathcal{F}$ is just

$$\Pr_{\pi}[\mathcal{E}] := \Pr_{\omega \sim \pi}[\omega \in \mathcal{E}] = \sum_{\omega \in \mathcal{E}} \pi(\omega). \tag{13}$$

We also require defining *random variables*. Random variables are functions of the sample space Ω. For example, consider our earlier example, where Ω was the outcomes of three coin flips, and $\mathcal{F}$ was the events where heads outnumber tails or vice versa. We could define the *random variable* $X(\omega)$ which is the number of heads in ω. This random variable, which is based on the randomness of $\omega \in \Omega$ (the outcomes of the coin flips), is interesting in its own right. For example, it contains all the information needed to answer whether we had more heads or tails. In this course, we just abbreviate such a random variable as X instead of $X(\omega)$ for short, since the probability space will always be obvious from context.

Note that the "base outcomes" $\omega \in \Omega$ are themselves random variables (by applying the identity function on Ω). So, we sometimes abuse the term "random variable" everywhere for simplicity.

We finally consider induced distributions π on a random variable X, which applies (13) to the event $\mathcal{E}$ of all outcomes ω which map to some X. We then define $\pi(X)$ much the same way we do for probability spaces. For example, when Ω is the outcome of three coin flips as before, and X is the number of heads in an outcome, the induced distribution on X is $\pi(0) = \frac{1}{8}$, $\pi(1) = \frac{3}{8}$,

$\pi(2) = \frac{3}{8}$, and $\pi(3) = \frac{1}{8}$. We will also define $\Pr[X \in \mathcal{E}]$ the same way as in (13), replacing ω with the random variable X, for any event $\mathcal{E}$ that could occur on X's outcome. If Ω is the set of possible outcomes of X, we say that Ω is X's *support*, and that X is supported on Ω.

We will not be overly pedantic in this course. You do not need to explicitly declare probability spaces, or call π a measure; we are fine any semantically similar term, like "density" or "distribution." The purpose of this section is to give formal definitions, so if you really need to trace your calculation down the stack in order to debug what is wrong, you have a reference for doing so.

6.2 Conditioning and independence

Despite Section 6.1 seeming like a fairly complicated way of going about all this probability business, it is very helpful to formalize our intuition, since probability can often be very unintuitive. Perhaps the most counterintuitive operation in probability is conditioning.[8]

Let $X \sim \pi$ be a random variable taking on values in Ω. In conditioning, we want to understand its distribution, when we have some "partial information" about X. For example, let $\Omega = [6]$ be the set of possible fair dice roll sides, and π be uniform over outcomes $x \in \Omega$. Then, letting $S = \{1, 3, 5\}$ be the odd sides, we could ask about the conditional distribution if I promise you that $X \in S$. This distribution is denoted $\pi(X = x \mid X \in S)$ (which is $\frac{1}{3}$ for all three $x \in S$).

Intuitively, in the general case, we first shade in a pie chart with possible outcomes $X \in \Omega$ according to the distribution π, then erase all the outcomes not in S, and then "re-expand" the remaining outcomes proportionally to fill in the pie (so that the total probability mass returns to 1). This is our new conditional probability distribution.

Formally, let $S \subseteq \Omega$. We can define the conditional distribution of X, given the partial information that $X \in S$. This new *conditional* distribution is restricted to the new sample space S, and is:

$$\pi(X = x \mid X \in S) = \frac{\pi(x)}{\Pr_\pi[S]}, \text{ for all } x \in S. \tag{14}$$

Recall that we defined $\Pr_\pi[S]$ in (13). The denominator in the formula (14) is the "re-expanding of the pie chart," to make sure the new conditional distribution is a valid probability measure.

We next discuss what happens when you have multiple random variables in the picture, e.g., $X \sim \pi$ with sample space Ω, and $X' \sim \pi'$ with sample space Ω'. We want to understand the "joint" behavior of outcomes X and X' on these spaces: do X and X' interact at all? The set of possible outcomes on the joint outcome (X, X') behaves exactly the way you expect it would: it is pairs $(x, x') \in \Omega \times \Omega'$, i.e., a combination of one outcome for each of X and X'.

The tricky part is what happens to the joint distribution. In general, the joint distribution ν on the pair (X, X') could be one of many things, as long as it remains true that $X \sim \pi$ and $X' \sim \pi'$. For example, suppose $\Omega = \Omega' = [6]$, and π, π' are both the uniform distribution (so X, X' are dice rolls). If X and X' were *independent*, then we could expect the joint distribution ν to look like

$$\nu(x, x') = \frac{1}{36}, \text{ for all } x \in [6],\ x' \in [6], \tag{15}$$

i.e., there is a $\frac{1}{36} = \frac{1}{6} \times \frac{1}{6}$ chance of any outcome $(X, X') = (x, x')$. However, it is important to remember that independence is an assumption! An equally valid joint distribution, which is consistent with the information we are given (i.e., that the random variables X and X' are uniform on $[6]$), is the joint distribution ν which always sets $X = X'$:

$$\nu(x, x') = \begin{cases} \frac{1}{6} & x = x' \\ 0 & x \neq x' \end{cases}, \text{ for all } x \in [6],\ x' \in [6]. \tag{16}$$

More generally, in the discrete case, random variables X and X' are independent iff the conditional distribution on $X \mid X' = x'$ is the same, for any value of $x' \in \Omega'$. That is, knowledge of X' gives you no information about X. This is the case for the joint distribution (15): for any $x' \in [6]$, the

[8] If you want a challenge, consider the following (surprisingly tricky) puzzle, attributed to Elchanan Mossel: how many rolls of a fair 6-sided die do we need on average to see a 2, conditioned on only even numbers appearing?

conditional distribution $(X \mid X' = x')$ is still just the uniform distribution π. However, the story is very different for the joint distribution (16); given $X' = x'$, we know with certainty what X is, so the distribution of $(X \mid X' = x')$ is not π. Thus, X and X' are dependent in (16).

A useful property of independent random variables is that the probability of joint events factorize. More formally, suppose that X, X' are independent random variables, with distributions π, π' and supports Ω, Ω' respectively. Let ν be the joint distribution of X, X'. For any possible outcome $x \in \Omega$ and $x' \in \Omega'$, we can compute the joint probability

$$\begin{aligned}\Pr_{(X,X')\sim\nu}[(X,X') = (x,x')] &= \Pr_{(X,X')\sim\nu}[X = x] \cdot \Pr_{(X,X')\sim\nu}[(X,X') = (x,x') \mid X = x] \\ &= \Pr_{X\sim\pi}[X = x] \cdot \Pr_{(X,X')\sim\nu}[X' = x' \mid X = x] \\ &= \Pr_{X\sim\pi}[X = x] \cdot \Pr_{X'\sim\pi'}[X' = x'],\end{aligned} \tag{17}$$

where the first equality applied the definition (14), the second equality used that the conditioning event already guarantees $X = x$, and the last equality used the independence assumption.

More generally, for any pair of events $\mathcal{E} \subseteq \Omega$ and $\mathcal{E}' \subseteq \Omega'$, a similar factorization holds:

$$\begin{aligned}\Pr_{(X,X')\sim\nu}[X \in \mathcal{E} \text{ and } X' \in \mathcal{E}'] &= \sum_{x\in\mathcal{E}} \sum_{x'\in\mathcal{E}'} \Pr_{(X,X')\sim\nu}[(X,X') = (x,x')] \\ &= \sum_{x\in\mathcal{E}} \sum_{x'\in\mathcal{E}'} \Pr_{X\sim\pi}[X = x] \cdot \Pr_{X'\sim\pi'}[X' = x'] \\ &= \left(\sum_{x\in\mathcal{E}} \Pr_{X\sim\pi}[X = x]\right) \cdot \left(\sum_{x'\in\mathcal{E}'} \Pr_{X'\sim\pi'}[X' = x']\right) \\ &= \Pr_{\pi}[\mathcal{E}] \cdot \Pr_{\pi'}[\mathcal{E}'].\end{aligned}$$

Despite these definitions all seeming quite reasonable, conditioning on dependent random variables can have non-obvious consequences. We work through a few illustrative examples.

Puzzle 1. *There are two cards, both with a black side and a white side. They are placed on a table, both with one side face up uniformly at random. Conditional on at least one card having a white side facing up, what is the probability that both cards have white sides facing up?*

Puzzle 2 (Monty Hall problem)**.** *You are on a game show, where there are three doors. Uniformly randomly over all orders of the doors, the doors have behind them in some order: a car, a goat, and another goat. You select a door, and then the game show host behaves as follows.*

1. *If you selected the door with a car, the host reveals to you a uniformly random other door.*
2. *If you selected a door with a goat, the host shows you the other door with a goat.*

You cannot tell the difference between these two cases, since the host always shows you a goat behind a uniformly random door. You now have the option to keep your original door, or switch to the remaining unopened door. Assuming you like cars more than goats, should you switch?

In Puzzle 1, one might first suspect that one card being white has no bearing on the other card being white, since they are independently distributed (so the answer is $\frac{1}{2}$). However, we have some partial information: at least one of them was white. Out of the four possible outcome combinations $\omega \in \{\text{BB}, \text{BW}, \text{WB}, \text{WW}\}$, when the outcome is distributed according to the uniform distribution π, only three are possible conditional on what we now know. Thus,

$$\Pr_{\omega\sim\pi}[\omega = \text{WW} \mid \omega \text{ has at least one W}] = \frac{1}{3}, \Pr_{\omega\sim\pi}[\omega = \text{WB} \mid \omega \text{ has at least one W}] = \frac{1}{3},$$
$$\Pr_{\omega\sim\pi}[\omega = \text{BW} \mid \omega \text{ has at least one W}] = \frac{1}{3}, \Pr_{\omega\sim\pi}[\omega = \text{BB} \mid \omega \text{ has at least one W}] = 0.$$

So, the probability that both cards have white sides facing up, when the combination ω is drawn from the *conditional distribution* $\pi(\omega \mid \omega \text{ has at least one W})$, is only $\frac{1}{3} < \frac{1}{2}$.

How could this be, if the the random variables ω_1 and ω_2 (i.e., the two individual colors) are independent? The lesson learned from Puzzle 1 is that this is irrelevant: the conditional random variables $(\omega_1 \mid \omega \text{ has at least one W})$ and $(\omega_2 \mid \omega \text{ has at least one W})$ are dependent.

In Puzzle 2, we wish to compute the probability that the "third door," which is neither ours, nor the door the host opened, has a car behind it. There are two possible outcomes; you either originally chose a goat, or you originally chose the car. In the first, the "stay" case, you selected the door with the car (which occurs with probability $\frac{1}{3}$), and then the host reveals to you some other door. In the second, the "switch" case, you selected a door with a goat (with probability $\frac{2}{3}$), and then the host reveals to you the other goat. In the stay case, the third door does not hide a car; in the switch case, it does. So, if we switch, we get the car with probability $\frac{2}{3}$.

Again, the Monty Hall problem seems a bit strange; because the host always shows us a goat, it seems there should be a $\frac{1}{2}$ chance that our door holds the goat, since there is one goat and one car left, and all doors are symmetric. However, the point is that our door and the third door are not distributed the same way, *once we condition on the host's actions*. What the host does depends on what our door hides (and affects what the third door is), so conditioning changes the distributions. In a counterintuitive situation like this, it is better to formally work out the probability space, and formally declare descriptive outcomes (e.g., whether you picked a car or a goat).

6.3 Expectation and variance

In statistics, one often wants to understand a descriptive property of a distribution. For example, suppose random variable X is distributed $\sim \pi$: how much do most draws X vary, compared to your "typical sample from π?" What even is a "typical sample?" What does the shape of π look like, when plotted as a histogram; is it wide or narrow? We give quantitative ways to address these questions in Part VII of the notes. Here, we define some simple properties of distributions.

The *expectation* of random variable $X \sim \pi$ supported on $\Omega \subseteq \mathbb{R}$ is also known as its *mean* or *variance*. The expectation is the classic answer of how to best describe a "typical sample." It is denoted $\mathbb{E}_{X\sim\pi}[X]$, and is a weighted average of possible outcomes on X, weighted by their probabilities:

$$\mathbb{E}_{X\sim\pi}[X] := \sum_{x\in\Omega} \pi(x)x.$$

The *variance* of random variable $X \sim \pi$ supported on $\Omega \subseteq \mathbb{R}$ is a standard measure of how "spread" X is. We will see ways of quantifying this somewhat vague statement later in the course. For now, we just recall the definition: the variance is denoted $\mathrm{Var}_{X\sim\pi}[X]$, and the formula is:

$$\mathrm{Var}_{X\sim\pi}[X] := \mathbb{E}_{X\sim\pi}[X^2] - \mathbb{E}_{X\sim\pi}[X]^2.$$

The square root of the variance is the *standard deviation* of X. That this definition exists suggests that the variance is always nonnegative. This is indeed true, and it is because $f(X) = X^2$ is convex (roughly speaking, it is bowl-shaped; formally, $2 = f''(x) \geq 0$ for all $x \in \Omega$). It is a general fact called *Jensen's inequality* that for any convex function $f : \Omega \to \mathbb{R}$,

$$\mathbb{E}_{X\sim\pi}\left[f(X)\right] \geq f\left(\mathbb{E}_{X\sim\pi}\left[X\right]\right).$$

Applying this fact with $f(X) = X^2$ and rearranging shows the variance is indeed nonnegative.

If $X \sim \pi, X' \sim \pi'$ are independent, then letting ν be their joint distribution, it holds that

$$\mathbb{E}_{(X,X')\sim\nu}[XX'] = \mathbb{E}_{X\sim\pi}[X] \cdot \mathbb{E}_{X'\sim\pi'}[X'].$$

In other words, the expected product is the product of expectations (try proving this using a similar strategy to (17)!). Moreover, if X and X' are independent, then so are $f(X)$ and $g(X')$ for any functions f, g. Hence, we also have $\mathbb{E}_{(X,X')\sim\nu}[(XX')^2] = \mathbb{E}_{X\sim\pi}[X^2] \cdot \mathbb{E}_{X'\sim\pi'}[(X')^2]$, etc.

7 Data structures

One of the most appealing things about algorithms is that many problems share common structure. So, you can be lazy: every good trick you learn solves lots of problems.

A *data structure* is a formalization of this idea. It applies when we have abstracted operations which we would like to efficiently support over a set of objects, the data. Hopefully, this abstraction

is general enough to apply in many different concrete instances. It is useful because we can focus on the self-contained questions of how quickly operations can be implemented, and understanding various tradeoffs achievable between operation runtimes. These implementations, the inner workings of data structures, are some of the first algorithms we will encounter.

In this section, we recall the runtime guarantees and implementation details of several basic data structures. All of these data structures will be used throughout the course.

On memory. In this course, we focus on the *word RAM model* of computation. This model is useful because it is reasonably realistic, and makes accounting for runtimes fairly simple. In the word RAM model, we assume that all objects in the problem can be written to a chunk of memory with size w, e.g., $w = 64$. Each chunk is called a word. For example, the objects could be w-bit integers, w-bit floats, or elements in a set of size at most 2^w (represented by indices).

We assume throughout this course, unless specified otherwise, that all objects fit in a word. We essentially only assume otherwise when discussing algorithms taking integer inputs.

The main leap of faith that you have to take in the word RAM model is that any "reasonable" operations on words, e.g., bit shifts, bitwise and/or/xor/not, arithmetic, and so on, take $O(1)$ time. Technically, this assumption only makes sense if w itself is a constant, because otherwise runtimes should scale with w. However, this would prevent us from storing any object whose size grows with the input size n. For example, we could not even write down the number n in binary, because this takes $\lceil \log_2(n) \rceil = \omega(1)$ bits. This is a problem when, e.g., we want a pointer to the n^{th} vertex of a graph, or in general to store n objects with different names.

Nonetheless, algorithms researchers typically state runtimes in the word RAM model. In practice, hardware is often specially designed to support operations on a fixed-size chunk of bits. Hence, each word operation is a natural unit of time. Although we technically assume that $n \to \infty$, all inputs you would likely encounter in practice have $\log(n) = O(w)$, so that large integers and pointers to objects can fit in a few words, if not one. The word RAM model lets us do nice things like increment $x \leftarrow x + 1$ for $x \in [n]$, in $O(1)$ time rather than $O(\log n)$ time.

When we discuss space in the word RAM model, we similarly consider a word to use $O(1)$ space. We also assume that for any specified memory address pointing to a word in memory, we can look up the contents of that word in $O(1)$ time, and that all addresses take one word to write.

We use "address" and "pointer" interchangeably. We write $\&x$ to denote the address a where object x is stored. Conversely, for an address a, $*a$ denotes the object in the location that a points to.

7.1 Lists

We begin by discussing lists, which are some of the most primitive data structures.

Suppose our goal is to store a set of objects $x_1, x_2, \ldots$, from a universe Ω, in a data structure, called a list. We will assume that every object in Ω fits in one word of memory. There is also a special object called None, which signifies the absence of any object in Ω.

A list should support *insertion* and *deletion* access, i.e., adding or removing one object to or from our data structure, as well as *query* access, i.e., accessing the objects we are maintaining. As we will see, the types of operations we want to support changes their efficiency.

Arrays. The simplest way to implement a list is with an Array data structure. An Array allocates a chunk of words in memory. We describe a standard API for this data structure.

An Array has one public field: n, its size. It supports the following operations.

- Init(n), for $n \in \mathbb{N}$. Uses $O(1)$ time,[9] and initializes an empty Array of size n.
- Insert(x, i), for $x \in \Omega$, $i \in [n]$. Uses $O(1)$ time, and writes x into the i^{th} word.
- Delete(i), for $i \in [n]$. Uses $O(1)$ time, and overwrites None into the i^{th} word.
- Query(i), for $i \in [n]$. Uses $O(1)$ time, and returns the contents of the i^{th} word.

[9] It is a fun challenge to figure out how to allocate n words of memory in $O(1)$ time, if that memory has been touched previously. Recording the indices of the first and last blocks is not sufficient, since we cannot differentiate between a word's previous contents and new content. Conversely, explicitly clearing this memory takes $O(n)$ time.

An array is best suited for applications with static indexing, because it is somewhat rigid: its size is fixed, and it is expensive to move many objects around.

The API provided lets us implement several other basic operations. For example, we can move an object from location i to j in array A by using $x \leftarrow A.\mathsf{Query}(i)$, $A.\mathsf{Delete}(i)$, and $A.\mathsf{Insert}(x, j)$.

Linked lists. Another way of implementing a list is to use a LinkedList, which represents each object it maintains as a chunk in memory, with explicit pointers to the previous and next object in the list. That is, instead of objects occupying fixed words indexed by $i \in [n]$, they are maintained in an "index-less" fashion, and the data structure only stores the links between adjacent objects.

This design decision makes it straightforward to add an object between two adjacent objects at known addresses. If we maintained permanent indices, as in an Array, this could require shifting the indices of $\Theta(n)$ objects up by 1. Thus, inserting between adjacent objects can take $\Theta(n)$ time in an Array. Conversely, a LinkedList supports this operation in $O(1)$ time.

The tradeoff is that we can no longer efficiently query the i^{th} object, for a given $i \in [n]$, where n is the list size. Doing so would require repeatedly marching through pointers from the first object, taking i iterations. Instead, we can only quickly access objects in a linked list through their explicit memory addresses, when available. We now state a standard API for a linked list.

A LinkedList has one public field: n, its size. Each element e of the list has a public field $x \in \Omega$, and private fields prev and next pointing to adjacent elements. It supports the following operations.

- Init(). Uses $O(1)$ time, and initializes a LinkedList of size $n \leftarrow 0$.
- InsertAfter(x, a), for $x \in \Omega$, and where $a \in \mathbb{Z}$ with $0 \leq a \leq n$, or a is an address. If $a \in [n]$, it uses $O(n)$ time, and inserts a new element corresponding to x in between the a^{th} element in the list, and its former next element. If $a = 0$, the new element is inserted at the start of the list in $O(1)$ time. If a is an address, it uses $O(1)$ time, and inserts x in between the element located at address a, and its former next element.
- Delete(a), where $a \in [n]$ or a is an address. Uses $O(n)$ time if $a \in [n]$, and removes the a^{th} element of the list, or $O(1)$ time if a is an address, and removes the element at a.
- Query(a), where $a \in [n]$ or a is an address. Uses $O(n)$ time if $a \in [n]$, and returns the object stored at the a^{th} index, or $O(1)$ time if a is an address, and returns the object stored at a.

Linked lists are best used to manipulate objects we have stored explicit pointers to; otherwise, operations cost $O(n)$ time rather than $O(1)$, if we need to refer to objects by their indices.

We describe the implementation of InsertAfter; all other operations are similar. Our LinkedList maintains a private field corresponding to the address of its first element. If $a = 0$, the address previously pointing to the old first element f is overwritten by the address of a new element e with $e.x \leftarrow x$, and we set $e.\mathsf{prev} \leftarrow \mathsf{None}$, $e.\mathsf{next} \leftarrow \&f$, $f.\mathsf{prev} \leftarrow \&e$, and $n \leftarrow n + 1$ all in $O(1)$ time.

If $a \in [n]$ is an index, we first recursively call $f \leftarrow *f.\mathsf{next}$ starting from the first element f for $a - 1$ times, until we have found the a^{th} element f. We let $g \leftarrow *f.\mathsf{next}$. We then create a new object e with $e.x \leftarrow x$, and set $f.\mathsf{next} \leftarrow \&e$, $e.\mathsf{prev} \leftarrow \&f$, $e.\mathsf{next} \leftarrow \&g$, $g.\mathsf{prev} \leftarrow \&e$, and $n \leftarrow n + 1$. In other words, e is inserted between f and g. The bottleneck is finding f, which takes $O(n)$ time.

The case when a is an address is similar, except we can just directly set $f \leftarrow *a$ in $O(1)$ time.

Stacks and queues. The LinkedList API straightforwardly implies two other data structures, Stack and Queue, which maintain lists where we only manipulate starts or ends, respectively.

A Stack is the standard first-in-first-out (FIFO) data structure. Intuitively, it corresponds to a list arranged as a vertical stack, where we can only add or remove from the top of the stack (corresponding to the start of the list). It has no public fields, and supports the following operations.

- Init(). Uses $O(1)$ time, and initializes an empty Stack.
- Push(x), for $x \in \Omega$. Uses $O(1)$ time, and adds x to the top of the stack.
- Pop(). Uses $O(1)$ time, and returns the top object in the stack.
- Peek(). Uses $O(1)$ time, and returns the top object in the stack, without removing.

We can simply use a LinkedList to implement Stack. Recall that our LinkedList implementation stores a field a, its first element's address. Then, Push(x) is implemented by calling InsertAfter($x, 0$), Peek() calls Query(a) where a is the stored address field, and Pop() calls Query(a) and Delete(a).

A Queue is similar, but it is a last-in-first-out (LIFO) data structure. It supports adding objects to the *end* of a list, but instead removes objects from the *start*; intuitively, objects line up to be removed in their arrival order. It has no public fields, and supports the following operations.

- Init(). Uses time $O(1)$, and initializes an empty Queue.
- Enqueue(x), for $x \in \Omega$. Uses time $O(1)$, and adds x to the back of the queue.
- Dequeue(). Uses time $O(1)$, and returns and removes the object at the front of the queue.
- Peek(). Uses time $O(1)$, and returns the object at the front of the queue, without removing.

We implement Queue similarly to Stack, except we augment the LinkedList we use with one additional private field b, a pointer to the end of the list. Then, Dequeue() is implemented the same way as Pop(), and Peek() is identical. Finally, Enqueue(x) creates a new element e with $e.x \leftarrow x$, e.prev $\leftarrow *b$, and e.next $\leftarrow$ None. It then sets $*b$.next $\leftarrow \&e$, and overwrites $\&e$ into the field b.

7.2 Heaps

A heap maintains a set S of objects from a universe Ω. We assume Ω is *ordered*, i.e., for any two objects $x, x' \in \Omega$, either $x \leq x'$ or $x > x'$, and we can determine which case we are in using $O(1)$ time (for example, Ω could be $\mathbb{R}$, or your personal ordered ranking of ice cream flavors). The goal of a heap is to provide efficient access to the minimum object currently in S. Intuitively, the objects are non-mixing fluids with densities equal to object values. We throw all objects in our set S into a bucket, so the lightest rises to the top, which we would like to access quickly. Note that we have technically described a "min-heap," but we can implement a "max-heap" (which accesses the maximum object) by negating or otherwise renaming elements of Ω, to reverse their ordering.

Before discussing implementation details, we first provide a standard heap API.

A Heap has two public fields: n, the maximum size of the set S maintained by the Heap, and $k \in [n]$, the current size of S. It supports the following operations.

- Init(S, n), for S a set of objects in Ω with $|S| \in [n]$. Uses $O(n)$ time, and initializes a Heap with maximum size n, sets $k \leftarrow |S|$, and initializes the set maintained by the heap to S.
- Insert(x), for $x \in \Omega$. Can only be called if $k < n$. Uses $O(\log(n))$ time, and adds x to S.
- ExtractMin(). Uses $O(\log(n))$ time, and returns and removes the minimum object in S.
- PeekMin(). Uses $O(1)$ time, and returns the minimum object in S, without removing.
- Delete(a), where a is an address. Uses $O(\log(n))$ time, and removes the object in $*a$ from S.

As we can see, if all we want to do (beyond insertions and deletions into a set) is to interact with the minimum element, a heap provides very fast access. We now describe implementation details.

Representing a heap. It is helpful to think of our heap as a binary tree (see Section 4.2). Each vertex, or node, in the tree contains one object in S. The tree layout is fixed at initialization and never changes (though the actual objects stored in nodes may change). The layout is as follows: first, we create a complete binary tree of height $h = \lfloor \log_2(n+1) \rfloor - 1$ containing $2^{h+1} - 1$ nodes, and then add $n - (2^{h+1} - 1)$ additional leaves to the last level so there are n total nodes. The way we chose this value of $h = O(\log(n))$ is that we want $2^{h+1} - 1 \leq n < 2^{h+2} - 1$, and there is a unique such $h \in \mathbb{N}$. Among the n nodes, k contain objects from Ω, and $n - k$ are set to None.

In actuality, we will store the nodes of this tree in an Array of size n, crucially using that the size is fixed. The first array element is the tree root, the next two elements are the 1^{st} level, the next four elements are the 2^{nd} level, and so on. We make sure that the k nonempty nodes in our tree always correspond to the first k indices of the array. It is an exercise to check that if we have an index i corresponding to a non-root vertex v, v's parent is in index $\lfloor \frac{i}{2} \rfloor$; similarly, for a non-leaf v in index i, its left child is in index $2i$ and its right child is in index $2i + 1$. Thus, we can efficiently traverse the tree representation by indexing appropriately into our array.

When we described Array, we showed we can swap the objects at any two indices in $O(1)$ time. Thus, in the binary tree representation of the heap, we can swap the contents of a vertex at a given index with the contents of either its parent, or one of its children, in $O(1)$ time.

It is instructive exercise to translate the operations we describe on the tree representation to their implementations in the array. We provide such an explicit example for Init, in Algorithm 1.

Heap invariant. The key invariant that our implementation maintains is that: after each operation that modifies the heap (Init, Insert, ExtractMin, Delete), the object in every internal node is not larger than those in either of its children. We call this the "heap invariant."

Initializing a heap. We begin by writing the elements of S to the first k nodes in the tree by their array indices, which takes $O(k) = O(n)$ time as $k \leq n$. We could stop here, but this initialization could violate the heap invariant. Instead, we take some effort up front to fix violations.

A violation of the heap invariant occurs whenever a node's object is larger than one of its children's. We describe our algorithm for fixing such violations. Starting from the lowest level above the leaves, we traverse nodes left-to-right sequentially to fix potential violations. Suppose our traversal is currently fixing a node on level ℓ with object x. If x is larger than one of the objects in the node's children, we swap it with the smaller child object. We continue swapping x downwards, until reaching a leaf, or if x is no larger than both child objects. When we are done iterating through all the nodes on a level of the tree, we proceed to the level which is one smaller, until we reach the root. This completes the algorithm description, and pseudocode is provided in Algorithm 1.[10]

Algorithm 1: $\mathsf{Init}(S, n)$

```
1  Input: n ∈ ℕ, S a set of objects in Ω with |S| ∈ [n]
   // Writes S to an array, possibly violating the heap invariant.
2  A ← Array.Init(n)
3  k ← |S|
4  for i ∈ [k] do
5  |  A.Insert(S[i], i)
6  end
   // Fixes violations of the heap invariant, level by level in the tree representation.
7  ℓ ← ⌊log₂(k)⌋ − 1
8  while ℓ ≥ 0 do
9  |  for 2^ℓ ≤ i ≤ 2^(ℓ+1) − 1 do
10 |  |  while A.Query(i) > min(A.Query(2i), A.Query(2i + 1)) and i ≤ 2^(⌊log₂(k)⌋−1) − 1 do
11 |  |  |  j ← argmin_{j∈{2i,2i+1}}{A.Query(j)}
12 |  |  |  x ← A.Query(i)
13 |  |  |  A.Insert(A.Query(j), i)
14 |  |  |  A.Insert(x, j)
15 |  |  |  i ← j
16 |  |  end
17 |  end
18 |  ℓ ← ℓ − 1
19 end
```

Let us briefly relate our pseudocode to our earlier description. Each run of Lines 10 to 16 fixes one heap invariant violation, at a node with index i, on the ℓ^{th} level of the tree. It does so by repeatedly swapping the object in i with its smaller child object j if there is a heap invariant violation, until the index i is larger than $2^{\lfloor \log_2(k) \rfloor - 1} - 1$, the index of the last non-leaf node, so i is a leaf (in the subtree which consists of the nonempty nodes in the overall binary tree).

As stated in Section 1, we now must prove correctness, and analyze the complexity, of our algorithm.

The complexity bound is the easier of the two. Consider the runtime of Lines 10 to 16, for a node i on the ℓ^{th} level. Each time we run the while loop, all operations take $O(1)$ time from our

[10] We do not require pseudocode for every algorithm you ever come up with in this course; if it is one line of code, this is superfluous. However, if the algorithm is sufficiently complicated, e.g., it includes several different steps or loops, pseudocode is encouraged: it significantly reduces ambiguity and makes the algorithm easier to understand.

Array API. The number of loops is bounded by the number of times we can descend a level, before reaching the leaves: there are thus at most $h - \ell$ loops, where $h := \lfloor \log_2(k) \rfloor$ is the height of the tree, restricted to nonempty nodes, at initialization, since there are only k objects. There are also 2^ℓ nodes on level ℓ, so the total runtime is $O(2^\ell \cdot (h - \ell))$. Thus, summing over all levels ℓ,

$$\sum_{\ell=0}^{h-1} 2^\ell \cdot (h - \ell) = 1 + (1 + 2) + \ldots + \left(1 + 2 + \ldots + 2^{h-1}\right)$$
$$\leq 2 + 2^2 + \ldots + 2^h \leq 2^{h+1} = O(k) = O(n).$$

To prove Algorithm 1 is correct, i.e., that it maintains the heap invariant, we must show that there are no more violations after the algorithm is done. To see this, consider a node on level ℓ with index i. After the corresponding run of Lines 10 to 16 is complete, node i no longer violates the heap invariant, because either the while loop never ran (there was no violation at i to begin with), or i's contents are replaced with the smaller of its children's. Either way, i's new contents are now the smallest among it and its two children.

The key claim is that no future loops, i.e., swaps due to violations at higher levels than ℓ, ever violate the heap invariant at i again after completion. This proof of this claim is similar to our earlier argument; the only way a new violation could be introduced at i is if i was swapped with its parent, which was larger than one of its children. However, it will then immediately be swapped with its child again, fixing the violation.

Other operations. The rest of the heap operations are more straightforward to implement.

To implement Insert, since $k < n$, there are empty slots at the end of the Array, so we first call Insert$(x, k + 1)$; x is now stored in a leaf of the current tree. However, this may violate the heap invariant. We fix the heap invariant by starting at x's node, and repeatedly swapping it with its parent if the parent contains a larger value. The runtime of this implementation is $O(\log(n))$, because swaps take $O(1)$ time and the number of swaps is bounded by the tree height. These swaps all happen along P, the path from x's original node to the root. To see that swapping does not create any violations off of the path P, suppose we are about to swap x with y, the object in its parent. This only happens if $x < y$. Because of the heap invariant, y's other child object z satisfies $y \leq z$, so $x \leq z$ as well. Therefore, after the swap, we do indeed have $x \leq \min(y, z)$.

Implicitly, the above argument is using an induction hypothesis: that after completing all previous operations, we have preserved the heap invariant. Thus, invariants are a form of induction mechanism: assuming the invariant is preserved after any m number of operations, we prove that it is preserved after the $(m + 1)^{\text{th}}$ operation as well. The base case is always handled by Init.

To implement Delete, we first swap the object at address a we want to delete with the last element x of the array, i.e., the result of Query(k), and then remove it from the end of the array, decrementing k. However, this swap might create new heap invariant violations, if x is too large. We follow the pseudocode in Lines 10 to 16 to fix this violation, pushing x down to the leaf nodes again by repeatedly switching it with its smaller child. The proof that the invariant is preserved afterwards is identical to the proof in Init. The runtime is again bounded by the height of the tree, $O(\log(n))$.

The implementation of ExtractMin is similar, once we observe that the heap invariant forces the root to have the smallest element. This is because otherwise, there must be a violation on the path between the root and the node containing the smallest element. Thus, we can simply call Delete at the address &Query(1), containing the root node (written to the first index of the array).

Finally, our earlier observation implies that to implement PeekMin, we can just call Query(1).

7.3 Binary search trees

We develop binary search trees (BSTs) in this section. To motivate BSTs, suppose you instead want to design a data structure SortedList, which maintains a set S of at most n elements from an ordered universe Ω. The goal of SortedList is to support dynamic insertions and deletions of objects from Ω, and we want to make sure S is always *sorted*. That is, we should always be able to query the i^{th} largest object in S for any i. We now formally state this desired API.

A SortedList has two public fields: n, the maximum size of the set S maintained by the SortedList, and $k \in [n]$, the current size of S. It supports the following operations.

- Init(n), for $n \in \mathbb{N}$. Initializes a SortedList with maximum size n, with $S \leftarrow \emptyset$.
- Insert(x), for $x \in \Omega$. Can only be called if $k < n$. Adds x to S.
- Delete(a), where a is an address. Removes the object in $*a$ from S.
- Query(i), for $i \in [k]$. Returns the i^{th} largest object in S.

This is a very reasonable data structure to want to design, due to its relevance in many applications. For example, suppose you want to build an auction platform, where bids are dynamically placed and withdrawn, and you want to query bids by their rank. A SortedList is perfect for this task.

What sort of tradeoffs can we guarantee for the operations in a SortedList? Let us first see what our existing data structures yield. If we implemented SortedList using an instance of an Array, the key difficulty is that insertions could change the ranks of objects. For example, if $\Omega = \mathbb{R}$, and we first insert 1 and 2, where should we put them in the array? If we put them too close, we will pay a lot if future insertions place many objects in between 1 and 2, and we need to move everything around. If we put them too far, we risk running out of space for future insertions outside $[1, 2]$, and similarly may need to move things often. Overall, we can check that an Array implements Delete and Query in $O(1)$ time, but requires $O(n)$ time for Insert in the worst case, since it could require moving many objects. Since a Heap is based on an Array, it runs into similar issues.

If we instead use a LinkedList, we can make Insert take $O(1)$ time, but this only works if we know where to insert the object x. Even forgetting this issue, in order to implement Query, we do not have an explicit pointer to the i^{th} element of the list. Hence, Query(i) requires $O(n)$ time in the worst case, because we potentially need to march through pointers to the middle of the list.

Thus, both arrays and linked lists cannot implement the operations in a SortedList very efficiently; both require $\Theta(n)$ time for some operations, beyond Init. The goal of a binary search tree is to support all three of the operations Insert, Delete, and Query in a SortedList, in $O(\log(n))$ time. More formally, in this section we provide an implementation of the following API.

A BST has two public fields: n, the maximum size of the set S maintained by the BST, and $k \in [n]$, the current size of S. It supports Init (in $O(1)$ time), Insert, Delete, and Query (all in $O(\log(n))$ time), as described in our SortedList API. For convenience, we grant a BST two more operations.

- Search(x), for $x \in \Omega$. Uses $O(\log(n))$ time, and returns $\&v$ if x is contained in a node v of the tree, otherwise returning None.
- Index(a), where a is an address. Uses $O(\log(n))$ time, and returns $i \in [k]$, where $*a.x$ is the i^{th} largest object in S.

Notably, Index is a sort of inverse operation to Query: calling $\&$Query(i) for an index i gives the corresponding address, and calling Index(a) for an address a gives its node's rank.

Representing a BST. We represent a BST similarly to a LinkedList. Each object is stored in a vertex (node) of a graph occupying a chunk of memory, and has pointers to all of its neighbors. As the name BST suggests, we specifically use a binary tree; this is in contrast to linked lists, which represent sets as paths (see Section 4.1). We store a private field root containing the address of the root of the tree. We give each node v in the tree a public field $x \in \Omega$ (the object it contains), as well as private fields left, right, and parent, which are either addresses of other nodes, or None.

To efficiently implement our BST operations, we need to augment each node with some extra information. Every node v has additional private fields lsize and rsize, which store the number of nodes in its left and right subtrees, respectively. If v.left = None, then we let v.lsize = 0, and similarly v.right = None means v.rsize = 0. These fields are used to implement Query and Index.

BST invariant. As with heaps, our BST implementation maintains an extra key invariant, called the "BST invariant." This invariant requires that for every node v, the following conditions hold.

1. For every node u in the subtree rooted at v.left, $u.x \leq v.x$.
2. For every node u in the subtree rooted at v.right, $u.x > v.x$.

So, v's object is at least all objects in v's left subtree, and smaller than objects in its right subtree.

Implementing search. To see how the BST invariant is helpful, we show how it immediately implies an implementation of Search. For simplicity, we assume that the tree representation of a BST has height $O(\log(n))$ here. We will discuss how to enforce this assumption later.

We initialize v to the root node. If the argument x to Search is $v.x$, we return $\&v$. Otherwise, under the BST invariant, we know which subtree of v contains x, assuming x is actually in the tree. So, we can update $v \leftarrow *v.\text{left}$ or $v \leftarrow *v.\text{right}$ appropriately (depending on if $x < v.x$ or $x > v.x$), and recurse. If at some point, we cannot progress further down the tree and still have not found x, then we have contradicted our assumption that x is in the tree, and return None. We provide pseudocode in Algorithm 2, assuming T is a BST instance which satisfies the BST invariant.

Algorithm 2: Search(self, x)

```
1  Input: x ∈ Ω
2  v ← *T.root
3  while v.x ≠ x do
4      if x ≤ v.x then
5          if *v.left ≠ None then
6              v ← *v.left
7          end
8          else
9              Return: None
10         end
11     end
12     else
13         if *v.right ≠ None then
14             v ← *v.right
15         end
16         else
17             Return: None
18         end
19     end
20 end
21 Return: &v
```

Correctness of Algorithm 2 follows from the following inductive claim: at the start of the while loop in Lines 3 to 20, if x is in T at all, then it is in the subtree rooted at v. This is clearly true when $v = *T.\text{root}$, since v's subtree is the whole tree, showing the base case. Moreover, if x is contained in v's subtree, but it is not $v.x$, then it is in the left subtree iff $x < v.x$, and is in the right subtree otherwise. If the relevant subtree is empty, we can safely return None. These cases are handled by Lines 4 to 11, and Lines 12 to 19, respectively. This concludes our inductive proof.

For the runtime of Algorithm 2, each while loop takes $O(1)$ time, and advances v one level down the tree. There are only $O(\log(n))$ levels under our height bound, so the runtime is $O(\log(n))$.

Implementing query and index. We now show how to implement Query and Index for a BST instance T, assuming that it satisfies the BST invariant, and again, that it has height $O(\log(n))$.

To implement Query, we need to return the i^{th} largest object in the tree. To do so, we just need to be able to determine for a given node v, which of the following three cases we are in.

1. $v.x$ is the i^{th} largest object.
2. The i^{th} largest object is in v's left subtree.
3. The i^{th} largest object is in v's right subtree.

If we can determine which case applies at any node in $O(1)$ time, we can traverse down the tree, starting from the root, until we are at the i^{th} largest object (the first case). Either of the other two cases proceeds down the tree, so the overall time is proportional to the tree's height, $O(\log(n))$.

It is simple to make this determination at the root vertex $r \leftarrow *\mathsf{root}$. By the BST invariant, there are $r.\mathsf{lsize}$ objects which are at most $r.x$, and $r.\mathsf{rsize}$ objects which are greater. Therefore, $r.x$ is the $(r.\mathsf{lsize}+1)^{\text{th}}$ largest object, and we should proceed left if $i \le r.\mathsf{lsize}$ and right if $i \ge r.\mathsf{lsize}+1$.

We want to repeat this logic for a non-root vertex v. To do so, we reduce to the case where v is the root, by updating i so that our new goal is to find the i^{th} largest object in v's subtree, for an accurate value of i. We claim that Algorithm 3 performs this update correctly.

Algorithm 3: Query(self, i)

```
1  Input: i ∈ [T.k]
2  v ← *T.root
3  while i ≠ v.lsize + 1 do
4      if i ≤ v.lsize then
5          v ← *v.left
6      end
7      else
8          v ← *v.right
9          i ← i − v.lsize − 1
10     end
11 end
12 Return: v.x
```

The runtime of Algorithm 3 was analyzed already; each loop takes $O(1)$ time, and there are $O(\log(n))$ loops under our height bound. To prove correctness of Algorithm 3, we again use induction. Let x be the object we wish to find, i.e., the i^{th} largest object overall, for the initial value of i. We claim that whenever we begin a loop of Lines 3 to 11, x is still the i^{th} largest object in v's subtree, for the current i and v. The base case follows as in the first loop, v is the root, so we are indeed trying to find the i^{th} largest object in the tree rooted at v. In the inductive step, we suppose the claim holds when we start the loop, and prove it still holds if we start the next loop.

If $i = v.\mathsf{lsize}+1$, then we are done, since we terminate and there is no next loop. If $i \le v.\mathsf{lsize}$, then since all objects in the right subtree were larger than x by the BST invariant, x is still the i^{th} largest object in the subtree rooted at $*v.\mathsf{left}$, as claimed. This case is handled by Lines 4 to 6. Finally, if $i > v.\mathsf{lsize}+1$, then since there were $v.\mathsf{lsize}+1$ objects that were $\le x$ in v's subtree (contained in v and its left subtree), we update the index i to account for these objects which are no longer in $*v.\mathsf{right}$'s subtree. This case is handled by Lines 7 to 10, and completes our proof.

The implementation of Index is similar: here, we are given the address to a node v in the tree, and we need to determine the rank of $v.x$. Again, if v is the root, we should simply return $i \leftarrow v.\mathsf{lsize}+1$. If v is not the root, then its rank is certainly at least $v.\mathsf{lsize}+1$. However, there could be other objects $\le v.x$. For example, if v is in the right subtree of another node u, then $v.x > u.x \ge w.x$ for all w in u's left subtree, so we should also count all of these $u.\mathsf{lsize}+1$ many possible nodes w.

Fortunately, this is the only other case we have to worry about. For any node v in a binary tree, every other node w is either a descendant of v, an ancestor of v, or a descendant of one of v's ancestors. To see this, binary trees are connected, so there is always a path between v and w. If vertices along the v-w path only decrease in level, then w is a descendant; otherwise, the path goes up the tree before going down, so we reach an ancestor of v on the path to w.

Therefore, $w.x \le v.x$ iff w is in v's left subtree, or w is in the left subtree of a node u, which has v in its right subtree. The only such possible u are v's ancestors, so we check all nodes u on the path from v to the root, adding $u.\mathsf{lsize}+1$ to our final count if v is in u's right subtree. We check this by seeing if we went left to reach u from its child; the path back to v reverses this direction.

Pseudocode for Index is in Algorithm 4. The runtime is again dominated by the height, $O(\log(n))$.

Implementing insert and delete. The final two operations we must implement are Insert and Delete, which we sketch here. Importantly, these operations should preserve the BST invariant.

For Insert, we can build off Search. For intuition, if x is not in the tree currently, then by running Search, we actually learn a valid place to insert x. Specifically, Search(x) will return None, either

Algorithm 4: Index(self, a)

```
1  Input: a, the address of a node in T
2  v ← *a
3  count ← v.lsize + 1
4  while &v ≠ T.root do
5  |  (v, prev) ← (*v.parent, v)
6  |  if v.x < prev.x then
7  |  |  count ← count + v.lsize + 1
8  |  end
9  end
10 Return: count
```

on Line 9 or Line 17. If it terminates on Line 9, then x could have been v's left child, so creating a new node containing x and updating v.left to point to it suffices. Similarly, if Search exits on Line 17, we make the new node containing x the right child of the final v. Finally, in the case where a copy of x is already in the tree, we continue running the while loop instead of exiting when we find x. Analogously to Search, the runtime of this operation is $O(h)$, where h is the tree's height.

To handle a request Delete(a), let $v \leftarrow *a$ be the associated node. There are three cases.

If v has no children, we just remove the pointer to v from its parent (if any) to delete it.

If v has one child w, and v is not the root, we can "splice" w to become a child of v's parent u. This preserves the BST invariant, since the subtree rooted at w was already part of u's subtree. If v is the root and has one child w, we let w become the root and remove its pointer to v.

If v has two children, then suppose $v.x$ is the i^{th} largest object; we can compute i in $O(h)$ time using Index(a). Note that $i \neq 1$, since the smallest object in T cannot have a left child.

Let w be the node with the $(i-1)^{\text{th}}$ largest object, the *predecessor* of v, which we can also compute in $O(h)$ time using Query($i-1$). We claim that w is the rightmost vertex in v's left subtree. This is clearly the largest vertex in v's subtree that is no larger than v, but could w be outside v's subtree? We argued when describing Index that all such non-descendant vertices w are either ancestors of v, or descendants of v's ancestors. If w is a descendant of v's ancestor u, either $w.x \leq u.x < v.x$ or $v.x \leq u.x < w.x$, so $u.x$ is always between $w.x$ and $v.x$, and w is not the predecessor. Similarly, if w is v's ancestor, v must be in w's right subtree (else $w.x \geq v.x$), in which case $w.x$ is also at most the object in the rightmost vertex in v's left subtree. Hence, we can rule out this case too.

Thus, w is the rightmost vertex in v's left subtree, so it has no right child. We can then swap w and v, which does not affect the BST invariant, except between w and v themselves (as all other relative orders are preserved). We can then remove v using one of the previous cases, since it either has no children or one child. The overall runtime of these operations is $O(h)$ time.

We have shown how to implement Insert and Delete in $O(h)$ time, while preserving the BST invariant. However, we have not described how to maintain $h = O(\log(n))$ for these operations. This would be enough, since no other operations modify the tree layout. It turns out that there are ways of slightly modifying our implementations to ensure $h = O(\log(n))$; a BST implementation with this guarantee is called a *balanced binary search tree* (BBST). These modifications are not too complicated, but are outside the scope of this crash course. Descriptions of two BBST variants, *red-black trees* and *AVL trees*, are in [CLRS22], Section 13, and [Rou22], Section 11.4, respectively.

7.4 Hash tables

In Sections 7.2 and 7.3, we designed data structures which support a wide range of rank-based operations in $O(\log(n))$ time, over dynamic sets S of objects from an ordered universe Ω. However, for many applications, we are willing to compromise on this diverse functionality of our data structure, if we can give much faster implementations of a few important operations.

Hash tables are an example when we want to maintain a dynamic set S, and we only care about *set membership*: in the context of the BST API, as long as Insert, Search, and Delete are supported,

we are happy. The catch is that we want hash table operations to be blazingly fast: ideally, they will take $O(1)$ time, rather than $O(\log(n))$. Notice that all operations supported by a hash table have nothing to do with orderings, and apply fine to unordered universes Ω. A hash table is useful, for example, if you are tracking registered usernames on a website: we frequently need to know whether a username is taken, which is a set membership query. Hash tables also can keep track of visited nodes in graph search algorithms, among other applications all over algorithm design.

It is not known how to support all of the desired operations in hash tables in worst case $O(1)$ time. However, using randomization, we can achieve these runtimes on average, with the following API.

A HashTable has no public fields, and supports the following operations.

- Init(). Uses time $O(1)$, and initializes a HashTable with $S \leftarrow \emptyset$.
- Insert(x), for $x \in \Omega$. Uses expected $O(1)$ time, and adds x to S.
- Search(x), for $x \in \Omega$. Uses expected $O(1)$ time, and returns an address a with $*a = x$ if $x \in S$, otherwise returning None.
- Delete(a), where a is an address. Uses expected $O(1)$ time, and removes the object in $*a$ from S.

In general, guarantees for hash tables can be challenging to state precisely (i.e., what we mean by "expected time"). They often depend on problem assumptions (e.g., the tradeoffs are different if there are space constraints, or we have bounds on the load factor, the fraction of occupied slots out of all allocated). An example of a guarantee we can give is this: using $O(n)$ space, for any sequence of n calls to Insert, Search, and Delete, the expected runtime of the i^{th} operation for all $i \in [n]$, where expectations are taken over all of the randomness used by the HashTable, is $O(1)$.

The starting point is to consider how to store objects $x \in \Omega$ using $O(n)$ space. We could initialize an Array of size n in this much space, but we would need to figure out which index $f(x) \in [n]$ of the array we should store each $x \in \Omega$ in. Such a function, $f : \Omega \to [n]$, is often called a *hash map*.

We could get very lucky, and be in the case where all of the at most n objects inserted into S map to different elements of $[n]$. However, if $|\Omega| \gg n$, there is a chance of *collisions*, which is what happens when $f(x) = f(y)$ for two different objects x, y. If both x and y are inserted, it is unclear which gets to occupy the shared slot in the array. Moreover, this issue can happen $\Theta(n)$ times.

The big idea is to use randomization. Let $f : \Omega \to [n]$ have each $f(x)$ distributed uniformly in $[n]$, independently for all $x \in \Omega$. We begin by initializing a HashTable as a LinkedList L of size n, using n consecutive words. To handle a Insert(x) call, we place x in the $f(x)^{\text{th}}$ element of L, if empty. We can look this location up in $O(1)$ time, due to our initialization scheme. If it is full, we insert a new node in L right after its (original) $f(x)^{\text{th}}$ element.[11] The hope is that distinct $x, y \in \Omega$ will likely have $f(x) \neq f(y)$. Intuitively, each $i \in [n]$ defines a "bucket" of all $x \in \Omega$ with $f(x) = i$. Our goal is to use randomness to make our insertions hash to mostly different buckets.

The heart of the proof is the following key claim: for all $i \in [n]$, let X_i be the random variable which is the number of distinct $x \in \Omega$ inserted, with $f(x) = i$. Then for all objects x which are the target of an operation in the sequence (Insert, Search, or Delete), $\mathbb{E}[X_{f(x)}] \leq 2$. Intuitively, this upper bounds the maximum size of bucket $f(x)$ over the course of the algorithm, and the claim is it is at most two in expectation. Let us first see how this key claim implies all the other claims.

For all operations Insert(x), Search(x), or Delete($\&x$) where $x \in \Omega$ is in the hash table, we can bound its complexity by $O(X_{f(x)})$. Indeed, Insert(x) takes $O(1)$ time, Search(x) takes $O(X_{f(x)})$ time because we could potentially need to comb through the entire i^{th} bucket searching for x, and Delete is handled similarly to Search, before deleting appropriate pointers. Hence, the expected cost of the operation involving x is $\mathbb{E}[O(X_{f(x)})] = O(1)$, under our earlier key claim that $\mathbb{E}[X_{f(x)}] \leq 2$.

We establish $\mathbb{E}[X_{f(x)}] \leq 2$ via *linearity of expectation*, one of the most powerful probability facts.

Fact 5 (Linearity of expectation)**.** *Let X, Y be possibly dependent random variables supported in $\mathbb{R}$, and let $a, b \in \mathbb{R}$. Then $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.*

[11] This is just one way to resolve collisions. There are other strategies, e.g., *linear* or *quadratic probing*, and *open addressing*, which may have better practical performance, due to patterns in input sequences exploitable via caching.

Fact 5 is ridiculously useful: note it applies in any scenario, regardless of dependences. It can of course be recursed upon and generalizes to multiple (≥ 3) variables. It simplifies entire tedious calculations into a single line. Here we apply it to complete our HashTable analysis.

Let $\{x_j\}_{j\in[k]}$ be all the distinct objects in Ω inserted over the course of the input sequence, where $k \leq n$, and let x be some object under consideration. Then, we have the claim:

$$\begin{aligned}\mathbb{E}\left[X_{f(x)}\right] &= \mathbb{E}\left[\sum_{j\in[k]} \mathbb{I}_{f(x_j)=f(x)}\right] = \sum_{j\in[k]} \mathbb{E}\left[\mathbb{I}_{f(x_j)=f(x)}\right] \\ &\leq 1 + \sum_{\substack{j\in[k]\\ x_j\neq x}} \mathbb{E}\left[\mathbb{I}_{f(x_j)=f(x)}\right] \leq 1 + \frac{k}{n} \leq 2.\end{aligned}$$

In the above, we use $\mathbb{I}_{\mathcal{E}}$ to denote the 0-1 *indicator random variable* corresponding to an event $\mathcal{E}$, i.e., $\mathbb{I}_{\mathcal{E}} = 1$ if the event occurs, and it is 0 otherwise. The random variable $X_{f(x)}$ is just the sum of the indicator random variables $\mathbb{I}_{f(x_j)=f(x)}$ for all the inserted x_j, and the second equality used linearity of expectation. In the second line, we first pulled out the term $x_j = x$ (if x was inserted). We finally bounded all other indicator variables by $\frac{1}{n}$ in expectation, because the values $f(x_j)$ and $f(x)$ are independent and uniform in $[n]$, so there is a $\frac{1}{n}$ chance they collide.

We have not yet dealt with duplicates, but we can simply store a counter with each object. Instead of creating a new node for each duplicated object, we increment its counter instead. This way, costs are consistent with our definition of X_i, the number of *distinct* elements in bucket i.

We now discuss practical considerations. The first is regarding the bound on n up front. One may instead hope to extend to arbitrary sequence sizes, without needing to allocate a fixed amount of space, by using dynamic resizing. However, it takes time to allocate and deallocate space.

The standard measure of space efficiency is *load factor*, i.e., $\frac{k}{n}$ where you are using space for n objects, and the hash table holds k objects. There are simple tricks to ensure that the load factor is always at least a constant, or in other words, maintaining $k = \Omega(n)$. One such strategy is the following: anytime $k \leq \frac{n}{4}$, we deallocate half the space, and anytime $k \geq \frac{3n}{4}$, we double the allocated space. Clearly this maintains $k = \Omega(n)$, because k never falls to less than $\frac{n}{4}$.

The worry is that we need to spend a lot of time allocating and deallocating. However, we claim that each allocation or deallocation (henceforth, an event) later pays for itself, i.e., it adds only a $O(1)$ factor overhead to the cost of a long sequence. If we halve n, it takes at least $\frac{n}{8}$ more operations to trigger another event, since we need to reach either $k = \frac{3n}{8}$ or $k = \frac{n}{8}$ from our starting point, $k = \frac{n}{4}$. Similarly, if we double n, we need $\frac{n}{4}$ more operations for another event. Therefore, event costs average out to an "amortized" $O(1)$ per time step: a single operation may cost a lot, but costly operations are infrequent. These dynamic resizing tricks cause other complications, however, and require significantly modifying our arguments for bounding X_i, due to accumulations over time.

Finally, we mention another key issue: randomness. In practice, it is unrealistic to assume access to a uniformly random hash function f, as this would essentially require knowing the values of all $f(x)$, which is itself another sort of hashing problem. Instead, the hope is that we can simulate enough randomness using a few high-quality random bits, and a few more parameters which "twist" the randomness in an unpredictable way. Many of these twists involve number theory. Moreover, there is also the question of how to really get "high-quality random bits." This is sort of a philosophical question, but there is an entire field called *pseudorandomness* where the goal is to quantify guarantees when supposedly random bits are only approximately random.

Further reading

For more on Section 2, see Part I of [LLM10].

For more on Section 3, see Chapter 9.7 of [LLM10] or Chapter 3 of [CLRS22].

For more on Section 4, see Chapter 5 of [LLM10].

For more on Section 5, see Chapters 1 to 3 and 5 to 8 of [Axl24].

For more on Section 6, see Part IV of [LLM10].

For more on Section 7, see Chapter 6 and Part III of [CLRS22].

All of these resources are freely available online, or are accessible online through the UT library.

References

[Axl24] Sheldon Axler. *Linear Algebra Done Right*. 2024.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[LLM10] Eric Lehman, F.T̃homson Leighton, and Albert R. Meyer. *Mathematics for Computer Science*. 2010.

[Rou22] Tim Roughgarden. *Algorithms Illuminated*. Soundlikeyourself Publishing, 2022.