CS 38: An Introduction to Algorithms

Spring 2017

TA Notes

Chinmay Nirkhe

0 Acknowledgments

I'd like to thank Ethan Pronovost, Leonard Schulman, Jalex Stark, William Hoza, and Nicholas Schiefer for assistance with content, past notes, diagrams, and editing.

Contents

1 Designing an Algorithm

Designing an algorithm is an art and something which this course will help you perfect. At the fundamental level, an algorithm is a set of instructions that manipulate an input to produce an output. For those of you with experience programming, you have often written a program to compute some function $f(x)$ only to find yourself riddled with (a) syntax errors and (b) algorithmic errors. In this class, we won't worry about the former, and instead focus on the latter. In this course, you will not be asked to construct any implementations of algorithms. Meaning we don't expect you to write any 'pseudocode' or code for the problems at hand. Instead, give an explanation of what the algorithm is intending to do and then provide an argument (i.e. proof) as to why the algorithm is correct.

A general problem you will find on your sets will ask you to *design* an algorithm X to solve a certain problem with a runtime Y[1]. Your solution should contain three parts:

1. An algorithm description.
2. A proof of correctness.
3. A statement of the complexity.

I strongly suggest that your solutions keep these three sections separate (see the examples). This will make it much easier for you to keep your thoughts organized (and the grader to understand what you are saying).

1.1 Algorithm Description

When specifying an algorithm, you have to provide the right amount of detail. I often express that this is similar to how you would right a lab report in a chemistry or physics lab today compared to what you would write in grade school. The level of precision is different because you are writing to a different audience. Identically, the audience to whom you are writing you should assume has a fair experience with algorithms and programming. If written correctly, your specification should provide the reader with an exercise in programming (i.e. actually implementing the algorithm in a programming language). You should be focusing on the exercise of designing the algorithm. In general, I suggest you follow these guidelines:

[1] If no runtime is given, find the best runtime possible.

(a) You are writing for a *human* audience. Don't write C code, Java code, Python code, or any code for that matter. Write plain, technical English. Its highly recommended that you use LaTeX to write your solutions. The examples provided should give you a good idea of how to weave in the technical statements and English. For example, if you want to set m as the max of an array a of values, **don't** write a for loop iterating over a to find the maximizing element. Instead the following technical statement is sufficient.[2]

$$m \leftarrow \max_{x \in a}\{x\} \tag{1.1}$$

(b) Don't spend an inordinate time trying to find 'off-by-one' errors in your code. This doesn't really weigh in much on the design of the algorithm or its correctness and is more an exercise in programming. Notice in the example in (1.1), if written nicely, you won't even have to deal with indexing! Focus on making sure the algorithm is clear, not the implementation.

(c) On the other hand, you can't generalize too much. There should still be a step-by-step feel to the algorithm description. However, there are some simplifications you can make. if we have in class already considered an algorithm X that you want to use as a subroutine to then by all means, make a statement like 'apply X here' or 'modify X by doing $(\dots)$ and then apply here'. Please don't spend time writing out an algorithm that is already well known.

(d) If you are using a new data structure, explain how it works. Remember that data structures don't magically whisk away complexity. For example a min heap is $O(1)$ time to find the minimum, but $O(\log n)$ time to add an element. Don't forget these when you create your own data structures. However, if you are using a common data structure like a stack, you can take these complexities as given without proof. Make a statement like 'Let S be a stack' and say nothing more.

1.2 Proof of Correctness

A proof of correctness should explain how the nontrivial elements of your algorithm works. Your proof will often rely on the correctness of other algorithms it uses as subroutines. Don't go around reproving them. Assume their correctness as a lemma and use it to

[2] It is my notational practice to set a variable using the $\leftarrow$ symbol. This avoids the confusing abusive notation of the $=$ symbol.

build a strong succinct proof. In general you will be provided with two different types of problems: Decision Problems and Optimization Problems. You will see examples of these types of problems throughout the class, although you should be familiar with Decision Problems from CS 21.

Definition 1.1 (Decision Problem)**.** A decision problem is a function $f : \Sigma \to \{\text{TRUE}, \text{FALSE}\}$.[3] Given an input x, an algorithm solving the decision problem efficiently finds if $f(x)$ is TRUE or FALSE.[4]

Definition 1.2 (Optimization Problem)**.** An optimization problem is a function $f : \Sigma \to \mathbb{R}$ along with a subset $\Gamma \subseteq \Sigma$. The goal of the problem is to find the $x \in \Gamma$ such that for all $y \in \Gamma$, $f(x) \leq f(y)$. We often call Γ the *feasible region* of the problem.

Recognize that as stated, this is a minimization problem. Any maximization problem can be written as a minimization problem by considering the function $-f$. We we call x the $\arg\min$ of f and could efficiently write this problem as finding

$$x \leftarrow \arg\min_{y \in \Gamma} \{f(y)\} \tag{1.2}$$

When proving the correctness of a decision problem there are two parts. Colloquially these are called $yes \to yes$ and $no \to no$, although because of contrapositives its acceptable to prove $yes \to yes$ and $yes \leftarrow yes$. This means that you have to show that if your algorithm return TRUE on input x then indeed $f(x) =$ TRUE and if your algorithm returns FALSE then $f(x) =$ FALSE.

When proving the correctness of an optimization problem there are also two parts. First you have to show that the algorithm returns a feasible solution. This means that you return an $x \in \Gamma$. Second you have to show optimality. This means that there is no $y \neq x \in \Gamma$ such that $f(y) < f(x)$. This is the tricky part and the majority of what this course focuses on.

Many of the problem in this class involve combinatorics. These proofs are easy if you understand them and tricky if you don't. To make things easier on yourself, I suggest

[3] Here Σ notes the domain on which the problem is set. This could be the integers, reals, set of tuples, set of connected graphs, etc.

[4] Often a decision problem f is phrased as follows: Given input (x, k) with $x \in \Sigma, k \in \mathbb{R}$ calculate if $g(x) \leq k$ for some function $g : \Sigma^* \to \mathbb{R}$.

that you break your proof down into lemmas that are easy to solve and finally put them together in a legible simple proof.

1.3 Algorithm Complexity

This is the only section of your proof where you should mention runtimes. This is generally the easiest and shortest part of the solution. Explain where your complexity comes from. This can be rather simple such as: 'The outer loop goes through n iterations, and the inner loop goes through $O(n^2)$ iterations, since a substring of the input is specified by the start and end points. Each iteration of the inner loop takes constant time, so overall, the runtime is $O(n^3)$.' Don't bother mentioning steps that *obviously* don't contribute to the asymptotic runtime. However, be sure to include runtimes for all subroutines you use. For more information on calculating runtimes, read the next section.

1.4 Example Solution

The following is an example solution. I've riddled it with footnotes explaining why each statement is important. Note the problem, I have solved here is a dynamic programming problem. It might be better to read that chapter first so that you understand how the algorithm works before reading this.

Exercise 1.3 (Longest Increasing Subsequence)**.** Given an array of integers $x_1, \ldots, x_n$, find the *longest increasing subsequence* i.e. the longest sequence of indices $i_1 < i_2 < \cdots < i_k$ such that $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_k}$. Design an algorithm that runs in $O(n^2)$.

Algorithm 1.4 (Longest Increasing Subsequence)**.** This is a dynamic programming algorithm.[5] We will construct tables ℓ and p where $\ell[j]$ will is the length of the longest increasing subsequence that ends with x_j and $p[j]$ is the index of the penultimate element in the longest subsequence.[6]

1. For $j = 1$ to n:[7]

[5] A sentence like this is a great way to start. It immediately tells the reader what type of algorithm to expect and can help you get some easy partial credit.

[6] We've told the reader all the initializations we want to make that aren't computationally trivial. Furthermore, we've explained what the ideal values of the tables we want to propagate are. This way when it comes to showing the correctness, we only have to assert that their tables are filled correctly.

[7] Its perfectly reasonable to use bullet points or numbers lists to organize your thinking. Just make sure you know that the result shouldn't be code.

(a) Initialize $\ell[j] \leftarrow 1$ and $p[j] \leftarrow$ NULL (soon to be changed).

(b) For every $k < j$ such that $x_k < x_j$: If $\ell[k] + 1 > \ell[j]$, then set $\ell[j] \leftarrow \ell[k] + 1$ and $p[j] \leftarrow k$.

2. Let j be the arg max of ℓ. Follow p backwards to construct the subsequence. That is, return the reverse of the sequence $j, p[j], p[p[j]], \ldots$ until some term has $p[j] =$ NULL.[8]

Proof of Correctness. First, we'll argue that the two arrays are filled correctly. Its trivial to see that the case of $j = 1$ is filled correctly. By induction on k, when $\ell[j]$ is updated, there is some increasing subsequence which ends at x_j and has length $\ell[j]$. This sequence is precisely the longest subsequence ending at x_k followed by x_j. The appropriate definition for $p[j]$ is immediate.[9] This update method is exhaustive as the longest increasing subsequence ending at x_j has a penultimate element at some x_k and this case is considered by the inductive step.

By finding the arg max of ℓ, we find the length of the longest subsequence as the subsequence must necessarily end at some x_j. By the update rules stated above, for $k = p[j]$, we see that $\ell[k] = \ell[j] - 1$ and $x_k < x_j$. Therefore, *a* longest subsequence is the solution to the subproblem k and x_j. The backtracking algorithm stated above, recursively finds the solution to the subproblem.[10] Reversing the subsequence produces it in the appropriate order.

Complexity. The outer loop runs n iterations and the inner loop runs at most n iterations, with each iteration taking constant time. Backtracking takes at most $O(n)$ time as the longest subsequence is at most length n. The total complexity is therefore: $O(n^2)$.[11]

[8] Resist the urge to write a while loop here. As stated is perfectly clear.

[9] We've so far argued that the updating is occurring only if a sequence of that length exists. We now only need to show that all longest sequences are considered.

[10] Backtracking is as complicated as you make it to be. All one needs to do is argue that the solution to the backtracked problem will help build recursively the solution to the problem at hand.

[11] Don't bother writing out tedious arithmetic that both of us know how to do.

2 Runtime Complexity and Asymptotic Analysis

2.1 Asymptotic Analysis

I'm sure all of you have read about Big O Notation in the past so the basic definition should be of no surprise to you. That definition you will find is sometimes a bit simplistic and in this class we are going to require more formalism to effectively describe the efficiency of our algorithms.

Bear with me for a bit, as I'm going to delve into a lot of mathematical intuition but I promise you that it will be helpful!

Let's form a *partial* ordering on the set of function $\mathbb{N} \to \mathbb{R}^+$ (functions from natural numbers to positive reals). Let's say for $f, g : \mathbb{N} \to \mathbb{R}^+$, that $f \leq g$ if for all but finitely many n, $f(n) \leq g(n)$.[12] Formally this means the following:

(a) (reflexivity) $f \leq f$ for all f.

(b) (antisymmetry) If $f \leq g$ and $g \leq f$ then $f = g$.[13]

(c) (transitivity) If $f \leq g$ and $g \leq h$ then $f \leq h$

What differentiates a partial ordering from a *total* ordering is that there is no idea that f and g are comparable. It might be that $f \leq g$, $f \geq g$ or perhaps neither. In a total ordering, we guarantee that $f \leq g$, or $f \geq g$, perhaps both.

Why is this important, you may rightfully ask. By defining this partial ordering, we've given ourselves the ability to define *complexity equivalence classes*.

[12] You might see this in the notation $\exists\ n_0 \in \mathbb{N}$ such that for all $n > n_0$, $f(n) \leq g(n)$. These are in fact equivalent. If $f(n) \leq g(n)$ for all but finitely many n (call them $n_1 \leq \cdots \leq n_m$) then for all $n > n_m$, $f(n) \leq g(n)$. Setting $n_0 = n_m$ completes this proof. For the other direction, let the set of finitely many n for which it doesn't satisfy be the subset of $\{1, \ldots, n_0\}$ where $f(n) > g(n)$.

[13] Careful here! When we say $f = g$ we don't mean that f and g are equal in the traditional sense. We mean they are equal in the asymptotic sense. Formally this means that for all but finitely many n, $f(n) = g(n)$. For example, the functions $f(x) = x$ and $g(x) = \lceil \frac{x^2}{x+10} \rceil$ are asymptotically equal.

Definition 2.1 (Big O Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f \in O(g)$ and (equivalently) $g \in \Omega(f)$ if $f \leq cg$ for some $c > 0$. Here we use '$\leq$' as described previously.

First recognize that $O(g)$ and $\Omega(f)$ are *sets* of functions. Let's discuss equivalence classes and relations for a bit.

Definition 2.2 (Equivalence Relation)**.** We say $\sim$ is an equivalence relation on a set X if for any $x, y, z \in X$, $x \sim x$ (reflexivity), $x \sim y$ iff $y \sim x$ (symmetry), and if $x \sim y$ and $y \sim z$ then $x \sim z$.

Our general definition for '=' fits very nicely into this definition for equivalence relations. But equivalence relations are more general than that. In fact they work with the definition of equality in a partial ordering above as well. Check this if you are unsure about it. Now, we can bring up the notation of an equivalence class.

Definition 2.3 (Equivalence Class)**.** We call the set $\{y \in X \text{ s.t. } x \sim y\}$, the equivalence class of x in X and notate it by $[x]$.

You might be asking yourself what does any of this have to do with runtime complexity? I'm getting to that. The point of all of these definitions about partial ordering and equivalence classes is that $f \in O(g)$ is a partial ordering as well! Go through the process of checking this as an exercise.

Definition 2.4. We say $f \in \Theta(g)$ and (equivalently) $g \in \Theta(f)$ if $f \in O(g)$ and $g \in O(f)$.

This means that $f \in \Theta(g)$ is an equivalence relation and in particular $\Theta(g)$ is an equivalence class. By now, perhaps you've gotten an intuition as to what this equivalence class means. It is the set of functions that have the same *asymptotic computational complexity*. This means that asymptotically, their values only deviate from each by a constant factor.

This is an incredibly powerful idea! We're now defined ourselves with the idea of equality that is suitable for this course. We are interested in asymptotic equivalence. If we're looking for a quadratic function, we're happy with finding any function in $\Theta(n^2)$. This doesn't mean per se that we don't care about constant factors, its just that its not the concern of this course. A lot of work in other areas of computer science focus on the constant factor.

What we're interested in this course is how to design algorithms for problems that look exponentially hard but in reality might have polynomial time algorithms. That jump is far more important than a constant factor.

We can also define o, ω notation. These are stronger relations. We used to require the existence of some $c > 0$. Now we require it to be true for all $c > 0$.

Definition 2.5 (Little O Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f \in o(g)$ and (equivalently) $g \in \omega(f)$ if $f \leq cg$ for all $c > 0$. Here we use '$\leq$' as described previously. Equivalently,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{2.1}$$

We can also discuss asymptotic analysis for functions of more than one variable. I'll provide the definition here for Big O Notation but its pretty easy to see how the other definitions translate.

Definition 2.6 (Multivariate Big O Notation)**.** Let $f, g : \mathbb{N}^k \to \mathbb{R}^+$. We say $f \in O(g)$ and (equivalently) $g \in \Omega(f)$ if $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$ for some $c > 0$ for all but finitely many tuples $(x_1, \ldots, x_k)$.[14]

A word of warning. Asymptotic notation can be used incredibly abusively. For example you might see something written like $3n^2 + 18n = O(n^2)$. In reality, $3n^2 + 18n \in O(n^2)$. But the abusive notation can be helpful if we want to 'add' or 'multiply' big O notation terms. You might find this difficult at first so stick to more correct notations until you feel comfortable using more abusive notation.

2.2 Random Access Machines and the Word Model

Okay, so we've gotten through defining Big O Notation so now we need to go about understanding how to calculate runtime complexity. A perfectly reasonable question to ask is 'what computer are we thinking about when calculating runtime complexity?'. A lot of you have taken courses on parallelization, for example. Are we allowed to use a parallel computing system here? These are all good questions and certainly things to be thinking

[14] It really helps me to think of this graphically. Essentially, this definition is saying that the region for which $f \not\leq cg$ is bounded.

about. However, for the intents of our class, we are not going to be looking at parallelization. We are going to assume a single threaded machine. [15] Is this a quantum machine? Also, interesting but in this case outside the scope of this course.

The most general model we could use would be a single headed one tape Turing machine. Although equivalent in terms of *decidability*, we know that this is not an efficient model particularly because the head will move around too much and this was incredibly inefficient. It was a perfectly reasonable model for us to use in CS 21 because the movement of the head can be argued to not cause more than a polynomial deviation in the complexity which was perfectly fine with us as we were really only concerned about the distinctions between P,NP,EXP, etc..

To define a model for computation, we need to define the costs of each of the operations. We can start from the ground up and define the time to flip a bit, the time to move the head to a new bit to flip, etc. and build up our basic operations of addition, multiplication from there and then move on to more complicated operations and so forth. This we will quickly find becomes incredibly complicated and tedious. However, this is the only actual method of calculating the time of an algorithm. What we will end up using is a simplification, but one that we are content with. When you think about an algorithm's complexity, you must always remember what model you are thinking in. For example, I could define a model where sorting is a $O(1)$ operation. This wouldn't be a very good model but I could define it anyways. Luckily, the models we're going to use have some logical intuition behind them and you wouldn't have any such silliness.

We are going to be using two different models in this class. The most common model we will be working in is the *Random-Access Machine (RAM) model*. In this model, instructions are operated on sequentially with no concurrency. Furthermore, we can write as much as we want to the memory and the access of any part of the memory is done in constant time.[16] We further assume that reading and writing a single bit takes constant time.

[15] If you consider a multi threaded machine with k threads, then any computation that takes time t to run on the multi-threaded machine takes at most kt time to run on the single threaded machine. And conversely, any computation that takes time t to run on the single threaded machine takes at most t time to run on the multi-threaded machine. If k is a constant, this doesn't affect asymptotic runtime.

[16] This is the motivation of the name Random-Access. A random bit of memory can be accessed in constant time. In a Turing machine only the adjacent bits of the tape can be accessed in constant time.

Recall that number between 0 and $n-1$ can be stored using $O(\log n)$ bits. So addition and subtraction take naïvely $O(\log n)$ time and multiplication takes $O((\log n)^2)$ time[17]. This model is the most accurate because it most closely reflects how a computer works.

However, as I said before this can get really messy. We will also make a simplification which we call the *word model*. In the word model, we assume that all the words can be stored in $O(1)$ space. There are numerous intuitions behind the word model but the most obvious is how most programming languages allocate memory. When you allocate memory for an integer, languages usually allocate 32 bits (this varies language to language). These 32 bits allow you to store integers between -2^{31} and $2^{31}-1$. This is done irrespective of the size the integer. So, operations on these integers are irrespective of the length.

This model is particularly useful if we want to consider the complexity of higher order operations. A good example is matrix multiplication. A naïve algorithm for matrix multiplication runs in $O(n^3)$ for the multiplication of two $n \times n$ matrices. By this we mean that the number of multiplication and addition operations applied on the elements of the matrices is $O(n^3)$.[18] The complexity of the multiplication of the elements isn't directly relevant to the matrix multiplication algorithm itself and can be factored in later.

For the most part, you will be able to pick up on whether the RAM model or the Word model should be used. In the example in the previous chapter, we used the Word model. Why? Because, the problem gave no specification as to the size of the elements $x_1, \ldots, x_n$. **If specified, then it implies the RAM model. Otherwise, use the word model.** In situations where this is confusing, we will do the best to clarify which model the problem should be solved in. If in doubt, ask a TA.

[17] You can also think about this as adding m bit integers takes $O(m)$ time. And you can store numbers as large as 2^m using m bits.

[18] Later we will show how to get this down to $O(n^{\log_2 7})$.

3 Introductory Topics

3.1 Recursion

Definition 3.1 (Recursive Algorithm)**.** A recursive algorithm is any algorithm whose answer is dependent on running the algorithm with 'simpler' values, except for the 'simplest' values for which the value is known trivially.[19]

The idea of a recursive algorithm probably isn't foreign to you. In this class, we will be looking at two different 'styles' of recursive algorithms: Dynamic Programming and Divide-and-Conquer algorithms. However let's take a look a look at a more basic recursive algorithm to start off. We will also introduce the notion of *duality* along the way.

Definition 3.2 (Greatest Common Divisor)**.** For integers a, b not both 0, let $\text{DIVS}(a, b)$ be the set of positive integers dividing both a and b. The greatest common divisor of a and b noted $\gcd(a, b) = \max\{\text{DIVS}(a, b)\}$.

What's a naïve algorithm for the gcd problem. We know that trivially $\gcd(a, b) \leq a$ and $\gcd(a, b) \leq b$ or equivalently $\gcd(a, b) \leq \min(a, b)$. A naïve algorithm could be to check all values $1, \ldots, \min(a, b)$ to see if they divide both a and b. This will have runtime $O(\min(a, b))$ assuming the word model.

We checked all the cases here, but under closer observation a lot of the checks were redundant. For example, if we showed that 5 wasn't a divisor of a or b, then we know that none of $10, 15, 20, \ldots$ divide them either. Let's explore how we can exploit this observation.

Lemma 3.3. *For integers a, b, not both 0, $\text{DIVS}(a, b) = \text{DIVS}(b, a)$ (reflexivity), and $\text{DIVS}(a, b) = \text{DIVS}(a + b, b)$.*

Proof. Reflexivity is trivial by definition. If $x \in \text{DIVS}(a, b)$ then $\exists\ y, z$ integers such that $xy = a, xz = b$. Therefore, $x(y + z) = a + b$, proving $x \in \text{DIVS}(a + b, b)$. Conversely, if $x' \in \text{DIVS}(a + b, b)$ then $\exists\ y', z'$ integers such that $x'y' = a + b, x'z' = b$. Therefore, $x'(y' - z') = a$ proving $x' \in \text{DIVS}(a, b)$. Therefore, $\text{DIVS}(a, b) = \text{DIVS}(a + b, b)$. □

Corollary 3.4. *For integers a, b, not both 0, $\text{DIVS}(a, b) = \text{DIVS}(a + kb, b)$ for $k \in \mathbb{Z}$, and therefore $\gcd(a, b) = \gcd(a + kb, b)$.*

[19] By simpler, I don't necessarily mean smaller. It could very well be that $f(t)$ is dependent on $f(t + 1)$ but $f(T)$ for some large T is a known base case. Or in a tree, the value could be based on that of its children, with the leafs of the tree as base cases.

Proof. Apply the lemma inductively. Then $\gcd(a,b) = \max\{\text{DIVS}(a,b)\} = \max\{\text{DIVS}(a+kb,b)\} = \gcd(a+kb,b)$. □

Let's make a stronger statement. Recall that one way to think about $a \pmod b$ is the unique number in $\{0,\ldots,b-1\}$ that is equal to $a+kb$ for some $k \in \mathbb{Z}$.[20] Therefore, the following corollary also holds.

Corollary 3.5. *For integers a,b, not both 0, $\gcd(a,b) = \gcd(a \pmod b, b)$.*

This simple fact is going to take us home. We've found a way to recursively reduce the larger of the two inputs (without loss of generality (wlog) assume a) to strictly less than b. Because it's strictly less than b, we know that this repetitive recursion will actually terminate. In this case, let's assume our base case is naïvely that $\gcd(a,0) = a$. Just for the sake of formality, I've stated this as an algorithm.

Algorithm 3.6 (Euclid-Lamé)**.** Given positive integer inputs a,b with $a \geq b$, if $b = 0$ then return a. Otherwise, return the $\gcd(b, a \pmod b)$ calculated recursively.[21]

To state correctness, its easiest to just cite the previous corollary and argue that as the input's strictly decrease we will eventually reach a base case. We don't need to consider negative inputs as $\gcd(a,b) = \gcd(|a|,|b|)$ by Corollary 3.4.

How do you go about arguing complexity? In most cases its pretty simple but this problem is a little bit trickier. Recall the Fibonacci numbers $F_1 = 1, F_2 = 1$ and $F_k = F_{k-1} + F_{k-2}$ for $k > 2$. I'm going to assume that you have remembered the proof from Ma/CS 6a (using generating functions) that:

$$F_k = \frac{1}{\sqrt{5}}(\phi^k - \phi'^k) \tag{3.1}$$

where ϕ, ϕ' are the two roots of $x^2 = x + 1$ (ϕ is the larger root, a.k.a the golden ratio). Note that $|\phi'| < 1$ so F_k tends to $\phi^k/\sqrt{5}$. More importantly, it grows exponentially.

Most times, your complexity argument will be the smallest argument. Let's make the following statement about the complexity:

[20] $a \pmod b$ is as the conjugacy class of a when we consider the equivalence relation $x \sim y$ if $x - y$ is a multiple of b. This forms the group $\mathbb{Z}/b\mathbb{Z}$. Addition is defined on the conjugacy classes as a consequence of addition on any pair of elements in the conjugacy classes permuting the classes.

[21] I write it as $\gcd(b, a \pmod b))$ instead of $\gcd(a \pmod b, b)$ here to ensure that the first argument is strictly larger than the second.

Theorem 3.7. *If $0 < b \leq a$, and $b < F_{k+2}$ then the Euclid-Lamé algorithm makes at most k recursive calls.*

Proof. This is a proof by induction. Check for $k < 2$ by hand. Now, if $k \geq 2$ then recall that the recursive call is for $\gcd(b, c)$ where we define $c := a \pmod b$. Now there are two cases to consider. The first is easy: If $c < F_{k+1}$ then by induction at most $k-1$ recursive calls from here so total at most k calls. ✓ In the second case: $c \geq F_{k+1}$. One more function call gives us $\gcd(c, b \pmod c)$. First, recall that there's a strict inequality among the terms in a recursive gcd call (proven previously). So $b > c$. Therefore, $b > b \pmod c$ as $c > b \pmod c$. In particular we have strict inequality, so $b \geq (b \pmod c) + c$ or equivalently $b \pmod c \leq b - c$. Then apply the bounds on b, c to get

$$b \pmod c \leq b - c \leq b - F_{k+1} < F_{k+2} - F_{k+1} = F_k \tag{3.2}$$

So in two calls, we get to a position from where inductively we make at most $k-2$ calls, so total at most k calls as well. ✓ □

The theorem tells us that Euclid-Lamé for $\gcd(a, b)$ makes $O(\log(\min(a, b)))$ recursive calls in the word model. I'll leave it as an exercise to finish this last bit.

3.2 Duality

Incidentally, this isn't the only problem that benefits from this recursive structure of looking at modular terms. We're going to look at a *dual* problem that shares the same structure. Formally for optimization problems,

Definition 3.8 (Duality)**.** A minimization problem $\mathcal{D}$ is considered the *dual* of a maximization problem $\mathcal{P}$ if the solution of $\mathcal{D}$ provides an upper bound for the solution of $\mathcal{P}$. This is referred to as *weak duality*. If the solutions of the two problems are equal, this is called *strong duality*.

Define SUMS(a, b) as the set of positive integers of the form $xa + yb$ for $x, y \in \mathbb{Z}$. With a little effort one can prove that like DIVS, the following properties hold for SUMS.

Lemma 3.9. *For integers a, b, not both 0,* $\text{SUMS}(a, b) = \text{SUMS}(a + kb, b)$ *for any* $k \in \mathbb{Z}$, *and therefore* $\text{SUMS}(a, b) = \text{SUMS}(a \pmod b, b)$.

It shouldn't be surprising then in fact there is a duality structure here. I formalize it below:

Theorem 3.10 (Strong Dual of GCD)**.** *For integers a, b, not both 0,*

$$\min\{\text{SUMS}(a, b)\} = \max\{\text{DIVS}(a, b)\} = \gcd(a, b) \tag{3.3}$$

Proof. By Lemma 3.9, we can restrict ourselves to positive (a, b). It's easy to see as $\gcd(a, b)$ divides a and b then it divides any $ax + yb$. Therefore $\gcd(a, b) \leq$ every element of $\text{SUMS}(a, b)$ proving weak duality. To prove strong duality, let (a, b) be the pair such that $a + b$ is the smallest and $\min\{\text{SUMS}(a, b)\} < \gcd(a, b)$.[22] But then $\text{SUMS}(b, a - b) = \text{SUMS}(a, b)$ by Lemma 3.9 and however, $b + (a - b) = b < a + b$, contradicting the assumed minimality of $a + b$. □

3.3 Repeated Squaring Trick

How many multiplications does it take to calculate x^n for some x and positive integer n? Well naïvely, we can start by calculating $x, x^2, x^3, \ldots x^n$ by calculating $x^j \leftarrow x \cdot x^{j-1}$. So this is $O(n)$ multiplications.

What if we wanted to calculate x^n where we know $n = 2^m$ for some positive integer m. This time we only calculate, $x, x^2, x^{2^2}, \ldots, x^{2^m}$ by calculating $x^{2^k} \leftarrow x^{2^{k-1}} \cdot x^{2^{k-1}}$. This is $O(m) = O(\log n)$ multiplications and costs $O(1)$ space as we only store the value of a single power of x at a time.

We can then extend this to calculate x^n for any n. Calculate the largest power m of 2 smaller than n (this is easy given a binary representation of n). Then $m \in O(\log n)$. Then calculate x^{2^j} for $j = 1, \ldots, m$ as before but this time writing each of them into memory. This takes $O(m) \subseteq O(\log n)$ space in the word model. If n has binary representation

[22] This is a very common proof style and one we will see again in greedy algorithms. We assume that we have a smallest instance of a contradiction and argue a smaller instance of contradiction. Here we define smallest by the magnitude of $a + b$.

$(a_m a_{m-1} \dots a_0)_2$ where $a_j \in \{0, 1\}$ then $n = \sum a_j 2^j$ and

$$x^n = \prod_{j=0}^{m} x^{a_j 2^j} \tag{3.4}$$

Therefore, using the powers we have written into memory, in an additional $O(\log n)$ multiplications we can calculate any power x^n. So any power x^n can be calculated using $O(\log n)$ multiplications and $O(\log n)$ space.[23]

[23] If we wanted to calculate all powers $x, \dots, x^n$ then the naïve method is optimal as it runs in $O(n)$. This method would take us $O(n \log n)$. This is a natural tradeoff and we will see it again in single-source vs. all-source shortest path graph algorithms.

4 Dynamic Programming

Dynamic Programming is a form of recursion. In Computer Science, you have probably heard the tradeoff between Time and Space. There is a trade off between the space complexity and the time complexity of the algorithm.[24] The way I like to think about Dynamic Programming is that we're going to exploit the tradeoff by utilizing the memory to give us a speed advantage when looking at recursion problems. We do this because, in general, "memory is cheap", and we care much more about being as time efficient as possible. We will see later on, however, cases where we care very much about the space efficiency (i.e. streaming algorithms).

Not all recursion problems have such a structure. For example the GCD problem from the previous chapter does not. We will see more examples that don't have a Dynamic Programming structure. Here are the properties you should be looking for when seeing if a problem can be solved with Dynamic Programming.

4.1 Principal Properties

Principal Properties of Dynamic Programming. Almost all Dynamic Programming problems have these two properties:

1. Optimal substructure: The optimal value of the problem can easily be obtained given the optimal values of subproblems. In other words, there is a recursive algorithm for the problem which would be fast if we could just skip the recursive steps.

2. Overlapping subproblems: The subproblems share sub-subproblems. In other words, if you actually ran that naïve recursive algorithm, it would waste a lot of time solving the same problems over and over again.

[24] I actual prefer to think about this as a many-way tradeoff between time complexity, space complexity, parallelism, communication complexity, and (in my opinion most importantly) correctness. This has lead to the introduction of the vast field of randomized and probabilistic algorithms, which are correct in expectation and have small variance. But that is for other classes particularly CS 139 and CS 150.

In other words, your algorithm trying to calculate $f(x)$ might recursively compute $f(y)$ many times. It will be therefore, more efficient to store the value of $f(y)$ once and recall it rather than calculating it again and again. I know that's confusing, so let's look at a couple examples to clear it up.

4.2 Tribonacci Numbers

I'll introduce computing 'tribonacci' numbers as a preliminary example. The tribonacci numbers are defined by $T_0 = 1, T_1 = 1, T_2 = 1$ and $T_k = T_{k-1} + T_{k-2} + T_{k-3}$ for $k \geq 3$.

Let's think about what happens when we calculate T_9. We first need to know T_6, T_7 and T_8. But recognize that calculating T_7 requires calculating T_6 as well since $T_7 = T_4 + T_5 + T_6$. So does T_8. This is the problem of overlapping subproblems. T_6 is going to be calculated 3 times in this problem if done naïvely and in particular if we want to calculate T_k the base cases of T_0, T_1, T_2 are going to be called $\exp(O(k))$ many times.[25] To remedy this, we are going to write down a table of values. Let's assume the word model again.

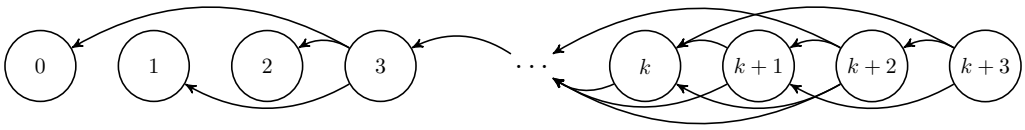

Figure 4.1: Recursion Diagram for Tribonacci Numbers Problem

Algorithm 4.1 (Tribonacci Numbers)**.** Initialize a table $t[j]$ of size k. Fill in $t[0] \leftarrow 1, t[1] \leftarrow 1, t[2] \leftarrow 1$. For $2 \leq j \leq k$, sequentially, fill in $T[j] \leftarrow t[j-1] + t[j-2] + t[j-3]$. Return the value in $t[k]$.

Proof of Correctness. We proceed by induction to argue $t[j] = T_j$. Trivially, the base cases are correct and by the equivalence of definition of $t[j]$ and T_j, each $t[j]$ is filled correctly.[26] Therefore, $t[k] = T_k$.

[25] $\exp(O(k)) = \{f : \exists c > 0, f(k) \leq e^{ck}\}$ where $\leq$ is the relation defined in Section 2. Defining $b = e^c$, then $\exp(O(k)) = \{f : \exists b > 0, f(k) \leq b^k\}$. Therefore, $\exp(O(k)) = \bigcup_{b \in \mathbb{R}^+} O(b^k)$.

[26] Not all proofs of correctness will be this easy, however, they won't be much more complicated either. Aim for 1-2 solid paragraphs. State what each element of the table should equal and argue its correctness.

Complexity. Calculation of each $T[j]$ is constant given the previously filled values. As $O(k)$ such values are calculated, the total complexity is $O(k)$.[27]

As T_j is a monotone increasing sequence in j, we know $T_j \geq 3T_{j-3}$. Therefore, $T_j = \Omega(b^j)$ where $b = \sqrt[3]{3}$. Its not difficult to see that storing $T_1, \ldots, T_k$ then takes $\Omega(b^k)$ space (it's a geometric series). We can make a constant factor improvement on the storage space by noticing that we don't need to store the entire table of $T_1, \ldots, T_k$, we only need to store the last three elements at any given time.

That example was far too easy but a useful starting point for understanding how Dynamic Programming works.

4.3 Generic Algorithm and Runtime Analysis

When you have such a problem on your hands, the generic DP algorithm proceeds as follows:

Algorithm 4.2 (Generic Dynamic Programming Algorithm)**.** For any problem,

1. Iterate through all subproblems, starting from the "smallest" and building up to the "biggest." For each one:
   (a) Find the optimal value, using the previously-computed optimal values to smaller subproblems.
   (b) Record the choices made to obtain this optimal value. (If many smaller subproblems were considered as candidates, record which one was chosen.)
2. At this point, we have the *value* of the optimal solution to this optimization problem (the length of the shortest edit sequence between two strings, the number of activities that can be scheduled, etc.) but we don't have the actual solution itself (the edit sequence, the set of chosen activities, etc.) Go back and use the recorded information to actually reconstruct the optimal solution.

[27] In one of the recitations, we will show how Fibonacci (also same complexity) can actually be run faster than $O(k)$ using repeated squaring.

It's not necessary for "smallest" and "biggest" to refer to literal size. All that matters is that the recursive calls are of "smaller" inputs and that eventually the recursive calls reach the base cases, the "smallest" inputs.

The basic formula for the runtime of a DP algorithm is

$$\text{Runtime} = (\text{Total number of subproblems}) \times \left(\begin{array}{c} \text{Time it takes to solve problems} \\ \text{given solutions to subproblems.} \end{array} \right)$$

Sometimes, a more nuanced analysis is needed. If the necessary subproblems are already calculated, then in the RAM model their lookup is $O(1)$. As our algorithm is to build up the solution, we guarantee that each subproblem is solved, so this $O(1)$ lookup time is correct. Therefore, the total time is bounded by the time complexity to solve a subproblem multiplied by the number of subproblems.

4.4 Edit Distance, an example

Definition 4.3 (Edit Distance)**.** For a given alphabet Σ, an *edit operation* of a string is an insertion or a deletion of a single character. The *edit distance*[28] is the minimum number of edit operations required to convert a string $X = (x_1 \dots x_m)$ to $Y = (y_1 \dots y_n)$.

For example, the edit distance between the words 'car' and 'fair' is 3: 'car' $\to$ 'cair' $\to$ 'air' $\to$ 'fair'. Note the *edit path* here is not unique. The problem is to calculate the edit distance in the shortest time.

The idea for dynamic programming is to somehow recursively break the problem into smaller cases. For convenience of notation, Write X_k to mean $(x_1 \dots x_k)$ (the substring) and similarly $Y_\ell = (y_1 \dots y_\ell)$. The intuition here is the problem of edit distance for $X = X_m$ and $Y = Y_n$ can be broken down into a problem about edit distance of substrings. Formally let $d(k, \ell)$ be the edit distance between X_k and Y_ℓ. The final goal is to calculate $d(m, n)$

[28] This is a actually a metric distance which we define when talking about clustering and packing.

then.

Let's consider what happens to the last character of X_k when calculating $d(k, \ell)$. One of 3 things happens:

(a) It is deleted. In which case the distance is equal to $1 + d(k-1, \ell)$.

(b) It remains in Y, but is no longer the right most character, and so a character was inserted. In which case the distance is $1 + d(k, \ell - 1)$.

(c) It equals the last character of Y, and so it remains the right most character of Y. In which case the distance is equal to $d(k-1, \ell-1)$.

Now, we don't know which of these 3 cases is going to occur. However, we do know that at least 1 of them must be true. Therefore, we can say that the distance is the minimum of these 3 values. Formally:

$$d(k, \ell) \leftarrow \min \begin{cases} d(k, \ell - 1) + 1 \\ d(k-1, \ell) + 1 \\ d(k-1, \ell - 1) + 2 \cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{4.1}$$

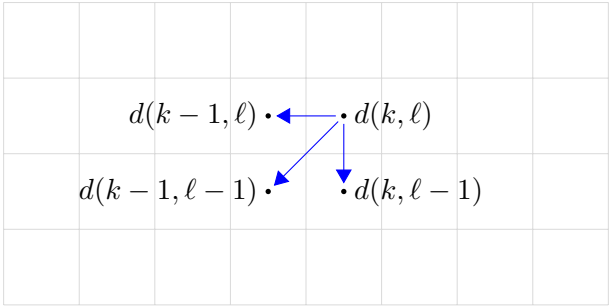

Figure 4.2: Depiction of recursive calls for Algorithm 4.2.

Here $\mathbb{1}_{\{.\}}$ is the indicator function; it equals 1 if the inside statement is true, and 0 otherwise. Therefore, its saying that the distance is 2 if they are not equal (because you'd have to add y_ℓ and remove x_k) and 0 if they are equal (no change required).

The base cases for our problem are $d(k,0) = k$ and $d(0,\ell) = \ell$ (insert or remove all the characters). This gives us all the intuition to formalize our dynamic programming algorithm:

Algorithm 4.4 (Edit Distance)**.** This is a dynamic programming algorithm. Generate a table of size $m \times n$ indexed as $t[k,\ell]$ representing the edit distance of strings X_k and Y_ℓ. Sequentially fill the table in order of increasing k and l by the following rule:

$$t[k,\ell] \leftarrow \min \begin{cases} t[k,\ell-1]+1 \\ t[k-1,\ell]+1 \\ t[k-1,\ell-1]+2\cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{4.2}$$

where any call to $t[k,0] = k$ and $t[0,\ell] = \ell$. When the table is filled, returned $t[m,n]$.

Algorithm Correctness. We verify that the table is filled correctly. For base cases if one string is empty then the minimum update distance is removing or adding all the characters. Inductively, we prove that the rest of the table is filled. If the last characters of the substrings agree, then an edit path is to consider the substrings without the last characters. Alternatively, the only two other edit paths are to add the last character of Y or remove the last character of X. We minimize over this exhaustive set of choices thus proving correctness.

Algorithm Complexity. Each entry of the table requires $O(1)$ calculations to fill given previous entries. As there are $O(mn)$ entries, the total complexity is $O(mn)$ in the word model.

4.5 Memoization vs Bottom-Up

In the previous problem, we generated a table of values t of size $m \times n$ and sequentially filled it up from the base cases to the most complex cases. At any given time, if we tried to calculate item $t[k,\ell]$, we were assured that the subproblems whose value $t[k,\ell]$ is based off of were already solved. In doing so, we had to fill up the entire table only to calculate the value at $t[m,n]$.

This method is the *Bottom-Up* approach. Start from the bottom and work up to calculating all the higher level values. An alternative to the Bottom-Up method is *memoization* (not memorization). If we need to calculate some value of the table $t[i]$ and it depends on values $t[j_1], t[j_2], \ldots, t[j_k]$ then we first check if the value of $t[j_1]$ is known. If it is, we use the value. Otherwise, we store in the computation stack our current position, and then add a new layer in which we calculate $t[j_1]$ recursively, and store its value. This may create more stack layers, itself. Once this stack computation is complete, the stack is removed and we go on to calculating $t[j_2], t[j_3], \ldots, t[j_k]$ one after another. Then we compute $t[i]$ based on the values of $t[j_1], \ldots, t[j_k]$.

Memoization should remind you of generic recursion algorithms with the additional step of first checking if a subproblems solution is already stored in the table and if so using it. More generally, the memoization algorithm looks like this:

Algorithm 4.5 (Memoized Dynamic Programming Algorithm)**.** Assume that $f(i) = g(f(j_1), \ldots, f(j_k))$ for some function g. Construct a table t and fill in all known base cases. Then,

(a) Check if $t[i]$ is already calculated. If so, return $t[i]$.

(b) Otherwise. Recursively calculate $f(j_1), \ldots, f_(j_k)$ and return $g(f(j_1), \ldots, f(j_k))$.

What are the advantages and disadvantages of memoization? First off, in all the problems we've looked at so far, the dependent subproblems have always had smaller indexes than the problem at hand. This is not always the case. A good example is a dynamic programming algorithm on a graph. Even if you number the vertices, you are still going to have all these weird loops because of the edges that you're going to have to deal with. Memoization helps here because you don't need to fill in the table sequentially.

Secondly, in practice memoization can be faster. Occasionally, not all the entries in the table will be filled meaning that we didn't perform unnecessary computations. However, the computation complexity of memoization and bottom-up is in general the same. Remember we're generally looking for worst case complexity. Unless you can argue that more than a constant-factor of subcases are being ignored then you cannot argue a difference in complexity.

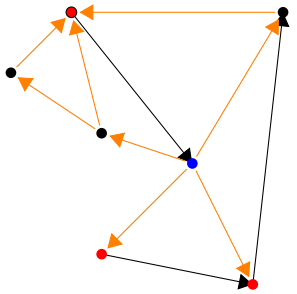

Figure 4.3: Memoization on a directed graph. Assume the red dots are base cases and the orange lines are the recursive calls. The recusion starts at the blue vertex.

Where memoization is worse is the use of the additional stack space. In the case of the Tribonacci numbers, you would generate a stack of size $O(n)$ due to the dependency of each T_j on T_{j-1}. In practice, the computation stack is held in the computer's memory while generally the data that you are manipulating is held on the hard drive. You might not be able to fit a computational stack of size $O(n)$ in the memory.[29] In the case of Tribonacci numbers as every prior Tribonacci number or Edit Distance we can guarantee the entire table will need to be filled. Therefore, bottom-up method is better as it does not use a large stack.

In this class, we won't penalize you for building a bottom-up or memoization dynamic programming algorithm when the other one was better; but it certainly something you should really think about in every algorithm design.

4.6 1D k-Clustering

A classic Dynamic Programming problem is that of 1-dimensional k-Clustering, our next example.

Definition 4.6 (1D Cluster radius)**.** Given a set X of n sorted points $x_1 < \ldots < x_n \in \mathbb{R}$ and set C of k points $c_1, \ldots, c_k \in \mathbb{R}$, the cluster radius is the minimum radius r s.t. every x_i is at most r from some c_j. Quantitatively, this is equivalent to

$$r = \max_{1 \le i \le n} \left(\min_{1 \le j \le k} |x_i - c_j| \right) \tag{4.3}$$

[29] There are interesting programmatic skirt arounds built for this in many functional languages such as Ocaml or Scala which rely considerably on large recursive algorithms.

Exercise 4.7 (1D k-Clustering)**.** Given a set X of n sorted points in $\mathbb{R}$, find a set C of k points in $\mathbb{R}$ that minimizes the cluster radius.

At first glance, this problem seems really hard and rightfully so. We're looking for k points $c_1, \ldots, c_k$ but they can sit anywhere on the number line. In that regard there are infinite choices that we are minimizing over. Let's simply start with some intuitions about the problem.

When we calculate the cluster radius, we can associate to each point x_i a closest point c_j. So let's define the set $S_j = \{x_i \ : \ c_j \text{ is the closest point of } C\}$. Since all the points lie on the number line, pictorially, these sets partition the number line into distinct regions. Figure 4.4 demonstrates this partition.

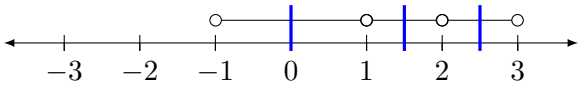

Figure 4.4: Assume that $C = \{-1, 1, 2, 3\}$. This divides the number line into partitions indicated by the blue lines. For any x_i the unique point of C within the partition is the closest point of C. In particular, for any point c_j, S_j is the set of x_i in its partition.

By figuring out this relation to partitions of the number line, we've reduced our problem to finding out which partition of n points minimizes the cluster radius. How many partitions are there? Combinatorics tells us there are $\binom{n-1}{k-1} \in O(n^{k-1})$ partitions. We can do even better.

Our intuition with any dynamic programming algorithm starts with how we state a subproblem. One idea is to define the subproblem as the minimum j-cluster radius for points $\{x_1, \ldots, x_t\}$ for $1 \leq j \leq k$ and $1 \leq t \leq n$. This is a good idea because (a) conditional on items $1, \ldots, t$ forming the union of j clusters, the rest of the solution doesn't matter on how the clustering was done internally and (b) there are $O(nk)$ subproblems.

Algorithm 4.8 (1D k-Clustering)**.** Construct a $n \times k$ table of indexed $r[t, j]$ and another indexed $c[t, j]$. In the second we will store a set of centers of magnitude j. For each $j = 1$

to k and for each $t = 1$ to n: If $j = 1$ then set

$$r[t, j] \leftarrow \frac{x_t - x_1}{2}, \qquad c[t, j] \leftarrow \left\{ \frac{x_t + x_1}{2} \right\} \tag{4.4}$$

Otherwise range s from 1 to $t - 1$ and set s to minimize $f(s) = \max \left\{ r[s, j-1], \frac{x_t - x_{s+1}}{2} \right\}$ and set

$$r[t, j] \leftarrow f(s), \qquad c[t, j] \leftarrow c[s, j-1] \cup \left\{ \frac{x_t + x_{s+1}}{2} \right\} \tag{4.5}$$

Algorithm Correctness. The table r stores the radius for the subproblem and c stores the the optimal set of centers that generate it. In the case that $j = 1$, then we are considering a single center. The optimal center is equidistant from the furthest points which are x_1 and x_t and hence is their midpoint.

Otherwise, we can consider over all possible splitting points s of where the right-most cluster should end. If the next-to-last cluster ends at point x_s, then the cluster radius of the first $j - 1$ clusters is recursively $r[s, j-1]$, and the the optimal j^{th} cluster center lies at $\frac{x_t + x_{s+1}}{2}$ with cluster radius $\frac{x_t - x_{s+1}}{2}$. The overall radius is the maximum of these two values. Therefore, the optimal clustering of the first t points into j clusters, with the second-to-last cluster ending at point x_s, has radius $f(s)$, with cluster set $r[s, j-1] \cup \left\{ \frac{x_t + x_{s+1}}{2} \right\}$. Minimizing over all possible values of s thus yields the optimal clustering of the first t points into j clusters.

Algorithm Complexity. There are $O(nk)$ subproblems and each subproblems computation requires at most $O(n + k)$ steps, iterating through the possible values of s and then writing down the centers. But as $k \leq n$ (otherwise the problem is trivial), we can lump these two together to say that the subproblem runtime is $O(n)$. Therefore, the overall runtime is $O(n^2 k)$.

Furthermore, notice that $r[s, j]$ will be a (weakly) monotone increasing quantity in s and that similarly $(x_t - x_s)/2$ is monotone decreasing in s. This allows us to instead run a binary search to find the optimal s instead of a linear scan, decreasing the runtime to $O(nk \log n)$ as each subproblem now runs in $O(\log n)$.[30]

[30] In which case, we cannot lump together the writing of the centers in the table and the iterating through possible values of s. The runtime of this is $O(\log n + k)$. However, there is a fix by instead storing a pointer

4.7 How exactly hard is NP-Hard?

Definition 4.9 (Pseudo-polynomial Time)**.** A numeric algorithm runs in *pseudo-polynomial time* if its running time is polynomial in the numeric value of the input but is exponential in the length of the input - the number of bits required to represent it.

This leads to a distinction of NP-complete problems into *weakly* and *strongly* NP-complete problems, where the former is anything that can be proved to have a pseudo-poly. time algorithm and those than can be proven to not have one. NP-hardness has an analogous distinction.

We're going to explore a weakly NP-complete problem next.

4.8 Knapsack

The following is one of the classic examples of an NP-complete problem. We're going to generate a pseudo-polynomial algorithm for it.

Exercise 4.10 (Knapsack)**.** There is a robber attempting to rob a museum with items of positive integer weight $w_1, \ldots, w_n > 0$ and positive integer values $v_1, \ldots, v_n$, respectively. However, the robber can only carry a maximum weight of W out. Find the optimal set of items for the robber to steal.

The brute force solution here is to look at all 2^n possibilities of items to choose and maximize over those whose net weight is $\leq W$. This runs in $O(n2^n \log W)$ and is incredibly inefficient. The $\log W$ term is due to the arithmetic complexity of checking if a set of weights exceeds W.

We apply our classical dynamic programming approach here. Define $S(i, W')$ to be the optimal subset of the items $1, \ldots, i$ that have weight bound W' and their net value and $V(i, W')$ to be their optimal value.

Algorithm 4.11 (Knapsack)**.** Using memoization, calculate $S(n, W)$ according to the following definitions for $S(i, W')$ and $V(i, W')$:

1. If $W' = 0$ then $S(i, W') = \emptyset$ and $V(i, W') = 0$.

to the subproblem from which the optimal solution is built and then backtracking to generate the k centers.

2. If $i = 0$ then $S(i, W') = \emptyset$ and $V(i, W') = 0$.

3. Otherwise, if $V(i-1, W') > V(i-1, W' - w_i) + v_i$

   - then $V(i, W') = V(i-1, W')$ and $S(i, W') = S(i-1, W')$.
   - If not, then $V(i, W') = V(i-1, W'-w_i)+v_i$ and $S(i, W') = S(i-1, W'-w_i) \cup \{i\}$.

Algorithm Correctness. For the base cases, if $W' = 0$ then the weight restriction doesn't allow the selection of any items as all items have positive integer value. If $i = 0$, then no items can be selected from. So both have $V = 0$ and $S = \emptyset$. For the generic problem (i, W') we consider the inclusion and exclusion of item i. If we include i then among the items $1, \ldots, i-1$ their weight cannot exceed $W' - w_i$. Hence their optimal solution is $S(i-1, W'-w_i)$ and the total value is $V(i-1, W'-w_i)+v_i$. If we exclude item i, then the value and solution is the same as the $(i-1, W')$ problem. By maximizing over this choice of inclusion and exclusion, we guarantee correctness as a consequence of the correctness of the subproblems.

Algorithm Complexity. There are $O(nW)$ subproblems and each subproblem takes $O(1)$ time to calculate given subproblems.[31] Therefore a total complexity of $O(nW)$.

We should note that the complexity $O(nW)$ makes this a pseudo-poly time algorithm. As the number W only needs $O(\log W)$ bits to be written, then the solution is exponential in the length of the input to the problem.[32]

4.9 Traveling Salesman Problem

Possibly the most famous of all NP-complete problems, the Traveling Salesman Problem is one of the classic examples of Dynamic Programming.

Exercise 4.12 (Traveling Salesman Problem)**.** Given a complete directed graph with $V = \{1, \ldots, n\}$ and non-negative weights d_{ij} for each edge (i, j), calculate the least-weight cycle that visits each node exactly once (i.e. calculate the least-weight Hamiltonian cycle).

[31] Actually, as I have stated the solution, copying $S(i-1, \cdot)$ into the solution of $S(i, W')$ is time-consuming. Instead, a backtracking solution is required to actually achieve this complexity. But totally doable!

[32] An interesting side note is that all known pseudo-polynomial time algorithms for NP-hard problems are based on dynamic programming.

Naïvely, we can consider all $(n-1)!$ cyclic permutations of the vertices. The runtime is $O(\exp(n \log n))$ by Stirling's approximation.

Our strategy, like always, is to find a good subproblem. For a subset $R \subseteq V$ of the vertices, we can consider the subproblem to be the least path that enters R, visits all the vertices and then leaves. Let's formalize this definition. For $R \subseteq V$ and $j \notin R$, define $C(R, j) =$ cost of cheapest path that leaves 1 for a vertex in R, visits all of R exactly once and no others, and then leaves R for j.

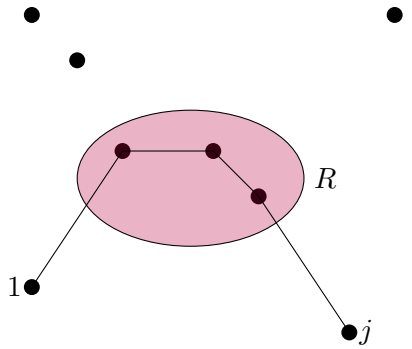

Figure 4.5: Subproblem of the Traveling Salesman Problem

Therefore an equivalent value for the traveling salesman problem is $C(R = \{2, \ldots, n\}, 1)$, as we can start the cycle from any vertex arbitrarily. The recursive solution for C is

$$C(R, j) \leftarrow \min_{i \in R} \left(d_{ij} + C(R - \{i\}, i) \right) \tag{4.6}$$

Let's briefly go over why this is correct: We are optimizing over all $i \in R$ for the node in the path prior to j. The cost would be the cost to go from 1 to all the vertices in $R - \{i\}$ then to i and then to j. That cost is precisely $C(R - \{i\}, i) + d_{ij}$.

Note that any set $R \subseteq V$ can be expressed by a vector of bits of length n (each bit is an indicator) you can show that each subproblem is $O(n)$ time plus the subsubproblem time as we're choosing $i \in R$ and $|R| \leq n$. As there are $O(n \cdot 2^n)$ subproblems, the total time complexity is $O(n^2 2^n)$, a significant improvement on the naïve solution.

5 Greedy Algorithms

The second algorithmic strategy we are going to consider is greedy algorithms. In layman's terms, the greedy method is a simple technique: build up the solution piece by piece, picking whatever piece looks best at the time. This is not meant to be precise, and sometimes, it can take some cleverness to figure out what the greedy algorithm really is. But more often, the tricky part of using the greedy strategy is understanding whether it works! (Typically, it doesn't.)

For example, when you are faced with an NP-hard problem, you shouldn't hope to find an efficient exact algorithm, but you can hope for an approximation algorithm. Often, a simple *greedy* strategy yields a decent approximation algorithm.

5.1 Fractional Knapsack

Let's consider a relaxation of the Knapsack problem we introduced earlier. A *relaxation* of a problem is when we simplify the constraints of a problem in order to make the problem easier. Often we consider a relaxation because it produces an *approximation* of the solution to the original problem.

Exercise 5.1 (Fractional Knapsack)**.** Like the Knapsack problem, there are n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ with a Knapsack capacity of W. However, we are allowed to select a fraction of an item. The output should be a fractional subset s of the items maximizing $v(s) = \sum_i s_i v_i$ subject to the capacity constraint $\sum s_i w_i \leq W$. By a fractional subset, we mean a vector $s = (s_1, \ldots, s_n)$ with all $0 \leq s_i \leq 1$.[33]

To solve this problem, let's introduce the notion of *quality* of an item: $q_i = v_i/w_i$. Intuitively, if the item was say a block of gold, it would be the dollar value of a single kg of the gold. The greedy strategy we are going to employ is going to picking items from highest to lowest quality. But first a lemma:

Lemma 5.2. *Let $q_1 > q_2$. Then in any optimal solution, either $s_1 = 1$ or $s_2 = 0$.*

Proof. A proof by contradiction. Assume an optimal solution exists with $s_1 < 1$ and $s_2 > 0$.

[33] This is the *relaxation* of the indicator vector we formulated the Knapsack problem around. In Knapsack, we restrict $s_i \in \{0, 1\}$.

Then for some small $\delta > 0$, we can define a new fractional subset by

$$s_1' = s_1 + \delta/w_1, \qquad s_2' = s_2 - \delta/w_2 \tag{5.1}$$

This will still be a fractional subset and still satisfy the capacity constraint and will increase the value by

$$\frac{v_1\delta}{w_1} - \frac{v_2\delta}{w_2} = \delta(q_1 - q_2) > 0 \tag{5.2}$$

This contradictions the assumed optimality. Therefore either $s_1 = 1$ or $s_2 = 0$. □

This tells us that if we sort the items by quality, we can greedily pick the items by best to worst quality.

Algorithm 5.3 (Fractional Knapsack)**.** Sort the items by quality so that $v_1/w_1 \geq \ldots v_n/w_n$. Initialize $C = 0$ (to represent the weight of items already in the knapsack). For each $i = 1$ to n, if $w_i < W - C$, pick up all of item i. If not, pick up the fraction $(W - C)/w_i$ and halt.

Proof of Correctness. By the lemma above, we should pick up entire items of highest quality until no longer possible. Then we should pick the maximal fraction of the item of next highest quality and the lemma directly forces all other items to not be picked at all. □

Complexity. We need to only sort the values by quality and then in linear time select the items. Suppose the item values and weights are k bits integers for $k = O(\log n)$. We need to compute each q_i to sufficient accuracy; specifically, if $q_i - q_j > 0$ then

$$q_i - q_j = \frac{v_i w_j - v_j w_i}{w_i w_j} > \frac{1}{w_i w_j} \geq 2^{-2k} \tag{5.3}$$

Therefore, we only need $O(k)$ bits of accuracy. This can be computed in $\tilde{O}(k)$ time per q_i and therefore total $n\tilde{O}(\log n)$. The total sorting time for per-comparison cost of k is $O(nk \log n)$. This brings the total complexity to $O(n \log^2 n)$. □

A final note about the fractional knapsack relaxation. By relaxating the constraint to allow fractional components of items, any optimal solution to fractional knapsack $\geq$ solution to classical knapsack. Also, our solution is almost integer; only the last item chosen is fractional.

5.2 Activity Selection

Exercise 5.4 (Activity Selection)**.** Assume there are n activities each with its own start time a_i and end time b_i such that $a_i < b_i$. All these activities share a common resource (think computers trying to use the same printer). A feasible schedule of the activities is one such that no two activities are using the common resource simultaneously. Mathematically, the time intervals are disjoint: $(a_i, b_i) \cap (a_j, b_j) = \emptyset$. The goal is to find a feasible schedule that maximizes the number of activities k.

The naïve algorithm here considers all subsets of the activities for feasibility and picks the maximal one. This requires looking at 2^n subsets of activities. Let's consider some greedy metrics by which we can select items and then point out why some won't work:

1. Select the earliest-ending activity that doesn't conflict with those already selected until no more can be selected.

2. Select items by earliest start time that doesn't conflict with those already selected until no more can be selected.

3. Select items by shortest duration that doesn't conflict with those already selected until no more can be selected.

The first option is the correct one. You can come up with simple counterexamples to problems for which the second and third options don't come up with optimal feasible schedules. Choosing the right greedy metric is often the hardest part of finding a greedy algorithm. My strategy is to try to find some quick counterexamples and if I can't really think of any start trying to prove the correctness of the greedy method.

Let's prove why the first option is correct. But first a lemma:

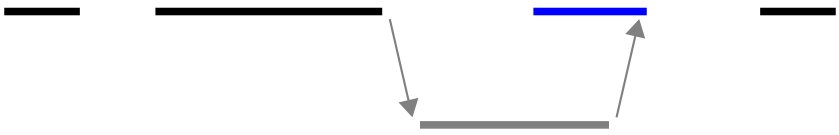

Figure 5.1: Acceptable substitution according to Lemma 5.5. The blue activity is replacable by the gray activity.

Lemma 5.5. *Suppose $S = ((a_{i_1}, b_{i_1}), \ldots, (A_{i_k}, b_{i_\ell}))$ is a feasible schedule not including (a', b'). Then we can exchange in (a_{i_k}, b_{i_k}) for (a', b') if $b' \leq b_{i_k}$ and if $k > 1$, then $b_{i_{k-1}} \leq a'$.*

Proof. We are forcing that the new event ends before event k and start after event $k-1$ See Figure 5.1. □

Theorem 5.6. *The schedule created by selecting the earliest-ending activity that doesn't conflict with those already selected is optimal and feasible.*

Proof. Feasibility follows as we select the earliest activity that doesn't conflict. Let $E = ((a_{i_1}, b_{i_1}), \ldots, (a_{i_\ell}, b_{i_\ell}))$ by the output created by selecting the earliest-ending activity and $S = ((a_{j_1}, b_{j_1}), \ldots, (a_{j_k}, b_{j_k}))$ any other feasible schedule.

We claim that for all $m \leq k$, (a_{i_m}, b_{i_m}) exists and $b_{i_m} \leq b_{j_m}$ (or in layman's terms the mth activity in schedule E always ends before then mth activity in schedule S). Assume the claim is false and m is the smallest counterexample. Necessarily $m = 1$ cannot be a counterexample as the schedule E by construction takes the first-ending event and must end before an event in any other schedule.

If m is a counterexample, then $b_{i_{m-1}} \leq b_{j_{m-1}} \leq a_{j_m}$ and $b_{i_m} > b_{j_m}$. This means that the event j_m ends prior to the event i_m and is feasible with the other events of E. But then event j_m would have been chosen over event i_m, a contradiction. So the claim is true.

Therefore, the number of events in E is at least that of any other feasible schedule S, proving the optimality of E. □

Let's take a moment to reflect on the general strategy employed in this proof: We considered the greedy solution E and any solution S and then proved that E is better than S by arguing that if it weren't, then it would contradict the construction of E.

5.3 Minimum Spanning Trees

Let's transition to a graph theory problem. The relevant graph definitions can be found at the beginning of Section 6 on Graph Algorithms.
All these definitions lead to the following natural question.

Exercise 5.7 (Minimum Spanning Tree)**.** Given a connected graph $G = (V, E, w)$ with positive weights, find a spanning tree of minimum weight (an MST).

Since we are interested in greedy solution, our intuition should be to build the minimum spanning tree up edge by edge until we are connected. My suggestion here is to think about how we can select the first edge. Since we are looking for minimum weight tree, let's pick the minimum weight edge in the graph (breaking ties arbitrarily). Then to pick the second edge, we could pick the next minimum weight edge. However, when we pick the third edge, we have no guarantee that the 3rd least weight edge doesn't form a triangle (i.e. 3-cycle) with the first two. So, we should consider picking the third edge as the least weight edge that doesn't form a cycle. And so on... Let's formalize this train of thought.

Definition 5.8 (Multi-cuts and Coarsenings)**.** A *multi-cut* $\mathcal{S}$ of a connected graph G is a partition of V into non-intersecting blocks $S_1, \ldots, S_k$. An edge *crosses* $\mathcal{S}$ if its endpoints are in different blocks. A multi-cut *coarsens* a subgraph G' if no edge of G' crosses it.

Figure 5.2: On the left, an example of a multi-cut $\mathcal{S}$ of a graph G. The partitions are the separately colored blocks. On the right, a subgraph G' of G that coarsens $\mathcal{S}$. No edge of G' crosses the partition.

We start with the empty forest i.e. all vertices and no edges: $G_0 = (V, \emptyset)$. Le the multicut $\mathcal{S}_0$ be the unique multicut where each vertex is in its unique block. Also, as there are no edges, this multicut $\mathcal{S}_0$ coarsens G_0. Our strategy will be to add the edge of minimum weight of G that crosses $\mathcal{S}_0$. This produces another forest G_1 (this time with $n-1$ trees in the forest). Let S_1 be the multicut formed by taking each tree as a block. To build G_2, we add the edge of minimum weight of G that crosses $\mathcal{S}_1$. G_2 has $n-2$ trees in the forest. We define S_2 to be the multicut formed by taking each tree as a block. And so on until we reach $\mathcal{S}_{n-1}$ which is the trivial partition. The edges in G_{n-1} form the MST.

Algorithm 5.9 (Kruskal's Algorithm for Minimum Spanning Tree)**.**

1. Sort the edges of the graph by minimum weight into a set S.
2. Create a forest F where each vertex in the graph is a separate tree.
3. While S is non-empty and F is not yet spanning
   (a) Remove the edge of minimum weight from S
   (b) If the removed edge connects two different trees then add it to F, thereby combining two trees into one.

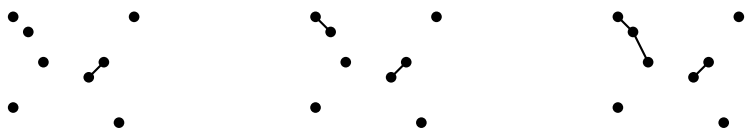

Figure 5.3: The first few iterations of Kruskal's Algorithm (Algorithm 5.9).

We have not argued the correctness of this algorithm; rather we have just explained our intuition. We will prove the correctness by generalizing this problem, and proving correctness for a whole class of greedy algorithms at once.

5.4 Matroids and Abstraction of Greedy Problems

Up till now, the examples we have seen of greedy algorithm rely on an exchange lemma. We've seen this similar property in linear algebra before.

Remark 5.10. *Given a finite-dimensional vector space X and two sets of linearly independent vectors V and W with $|V| < |W|$, there is a vector $w \in W - V$ such that $V \cup \{w\}$ is linearly independent.*

Equivalenlty, we can think of this as an exchange: If $|V| \leq |W|$ then you can remove any $v \in V$ and find a replacement from W to maintain linear independence. This notion of exchanging elements is incredibly powerful as it yields a very simple greedy algorithm to find the maximal linear independent set.

Let's consider an abstraction of greedy algorithms that will help formulate a generalized greedy algorithm.

Definition 5.11 (Matroid)**.** A matroid is a pair $(U, \mathcal{F})$, where U (the universe) is a finite set and $\mathcal{F}$ is a collection of subsets of U, satisfying

- (Non-emptyness) There is some set $I \in \mathcal{F}$ (equiv. $F \neq \emptyset$)
- (Hereditary axiom) If $I \subseteq J$ and $J \in \mathcal{F}$, then $I \in \mathcal{F}$.
- (Exchange axiom) If $I, J \in \mathcal{F}$ and $|I| > |J|$, then there is some $x \in I \setminus J$ so that $J \cup \{x\} \in \mathcal{F}$.

Definition 5.12 (Basis)**.** A basis I of a matroid $(U, \mathcal{F})$ is a maximal independent set such that $I \in \mathcal{F}$ and for any J s.t. $I \subsetneq J$ then $J \notin \mathcal{F}$.

In this context, we refer to sets in $\mathcal{F}$ as *independent sets.* A couple basic examples of matroids:

- The universe U is a finite set of vectors. We think of a set $S \subseteq U$ as independent if the vectors in S are linearly independent. A basis for this matroid is a basis (in the linear algebra sense) for the vector space spanned by U.
- The universe U is again a finite set of vectors. But this time, we think of a set $S \subseteq U$ as independent if $\text{Span}(U \setminus S) = \text{Span}(U)$. (This is the *dual matroid* to the previous matroid.) A basis for this matroid is a collection of vectors whose *complement* is a basis (in the linear algebra sense) for $\text{Span}(U)$.
- Let $G = (V, E)$ be a connected undirected graph. Then the universe U is the set of edges E and $\mathcal{F} =$ all acyclic subgraphs of G (forests).
- Let $G = (V, E)$ be a connected undirected graph again. Again $U = E$ but $\mathbf{F} =$ the subsets of E whose complements are connected in G. (This is the dual matroid of the previous example.)

Definition 5.13 (Weighted Matroid)**.** A *weighted matroid* is a matroid $(U, \mathcal{F})$ together with a weight function $w : U \to \mathbb{R}^+$. We may sometimes extend the function w to $w : \mathcal{F} \to \mathbb{R}^+$ by $w(A) = \sum_{u \in A} w(u)$ for any $A \in \mathcal{F}$.

Note: We assume that the weight of all elements is positive. If not, we can simply ignore all the items of negative weight initially as it is necessarily disadvantageous to have them in the following problem:

In the maximum-weight matroid basis problem, we are given a weighted matroid, and we are asked for a basis B which maximizes $w(B) = \sum_{u \in B} w(u)$. For the maximizes-weight matroid basis problem, the following greedy algorithm works:

Algorithm 5.14 (Matroid Greedy Algorithm)**.**

1. Initialize $A = \emptyset$.
2. Sort the elements of U by weight.
3. Repeatedly add to B the maximum-weight point $u \in U$ such that $A \cup \{u\}$ is still independent (i.e. $A \cup \{u\} \in \mathcal{F}$), until no such u exists.
4. Return A.

Let's prove the correctness of the remarkably simple matroid greedy algorithm.

Lemma 5.15 (Greedy choice property)**.** *Suppose that $M = (U, \mathcal{F})$ is a weighted matrioid with weight function $w : \mathcal{U} \to \mathbb{R}$ and that U is sorted into monotonically decreasing order by weight. Let x be the first element of U such that $\{x\} \in \mathcal{F}$ (i.e. independent), if any such x exists. If it does, then there exists an optimal subset A of U containing x.*

Proof. If no such x exists, then the only independent subset is the empty set (i.e. $\mathcal{F} = \{\emptyset\}$) and the lemma is trivial. Assume then that $\mathcal{F}$ contains some non-empty optimal subset B. There are two cases: $x \in B$ or $x \notin B$. In the first, taking $A = B$ proves the lemma. So assume $x \notin B$.

Assume there exists $y \in B$ such that $w(y) > w(x)$. As $y \in B$ and $B \in \mathcal{F}$ then $\{y\} \in \mathcal{F}$. If $w(y) > w(x)$, then y would be the first element of U, contradicting the construction.

Therefore, by construction, $\forall\ y \in B,\ w(x) \geq w(y)$.

We now construct a set $A \in \mathcal{F}$ such that $x \in A, |A| = |B|$, and $w(A) \geq w(B)$. Applying the exchange axiom, we find an element of B to add to A while still preserving independence. We can repeat this property until $|A| = |B|$. Then, by construction $A = B - \{y\} \cup \{x\}$ for some $y \in B$. Then as $w(y) \leq w(x)$,

$$w(A) = w(B) - w(y) + w(x) \geq w(B) \tag{5.4}$$

As B is optimal, then A containing x is also optimal. □

Lemma 5.16. *If $M = (U, \mathcal{F})$ is a matroid and x is an element of U such that there exists a set A with $A \cup \{x\} \in \mathcal{F}$, then $\{x\} \in \mathcal{F}$.*

Proof. This is trivial by the hereditary axiom as $\{x\} \subseteq A \cup \{x\}$. □

Lemma 5.17 (Optimal-substructure property)**.** *Let x be the first element of U chosen by the greedy algorithm above for weighted matroid $M = (U, \mathcal{F})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset on the following matroid $M' = (U', \mathcal{F}')$ with weight function w' defined as:*

$$U' = \{y \in U \mid \{x, y\} \in \mathcal{F}\}, \qquad \mathcal{F}' = \{B \subseteq U - \{x\} \mid B \cup \{x\} \in \mathcal{F}\}, \qquad w' = w|_{U'} \tag{5.5}$$

M' is called the contraction of M.

Proof. If A is an optimal independent subset of M containing x, then $A' = A - \{x\}$ is an independent set of M'. Conversely, if A' is an independent subset of M, then $A = A \cup \{x\}$ is an independent set of M. In both cases $w(A) = w(A') + w(x)$, so a maximal solution of one yields a maximal solution of the other. □

Theorem 5.18 (Correctness of the greedy matroid algorithm)**.** *The greedy algorithm presented in Algorithm 5.14 generates an optimal subset.*

Proof. By the contrapositive of Lemma 5.16, if we passover choosing some element x, we will not need to reconsider it. This proves that our linear search through the sorted elements of U is sufficient; we don't need to loop over elements a second time. When the algorithm selects an initial element x, Lemma 5.15 guarantees that there is some optimal

independent set containing x. Finally, Lemma 5.17 demonstrates that we can reduce to finding an optimal independent set on the contraction of M is sufficient. Its easy to see that Algorithm 5.14 does precisely this, completing the proof. □

Even though that was a long proof for such a simple statement, it has given us an incredible ability to demonstrate the correctness of a greedy algorithm. All we have to do is express a problem in the matroid formulation and presto! we have an optimal algorithm for it.

Theorem 5.19 (Runtime of the Greedy Matroid Algorithm)**.** *The runtime of Algorithm 5.14 is $O(n \log n + nf(n))$ where $f(m)$ is the time it takes to check if $A \cup \{x\} \in \mathcal{F}$ is independent given $A \in \mathcal{F}$ with $|A| = m$.*

Proof. The sorting takes $O(n \log n)$ time[34] followed by seeing if the addition of every element of U to the being built optimal independent set maintains independence. This takes an addition $O(nf(n))$ time, proving the runtime. □

5.5 Minimum Spanning Tree

5.5.1 Kruskal's Algorithm

We introduced Kruskal's Algorithm already as Algorithm 5.9 but we never proved its correctness or argued its runtime. We can prove the algorithms correctness by phrasing the algorithm as a matroid. In this case the universe $U = E$ and $\mathcal{F} =$ the set of all forests in G. Convince yourself that $(U, \mathcal{F})$ is indeed a matroid.

Corollary 5.20. *The Minimum Spanning Tree problem can be solved in $O(n^2 + m \log n)$ runtime where $n = |V|$ and $m = |E|$.*

Proof. Correctness then followed as an immediate consequence of formulation as a matroid and furthermore we got an initial bound on the runtime. Sorting the edges takes

[34] Note that this assumes that calculating the weight of any element x is $O(1)$. In reality, this time should also be taken into account.

$O(m \log m) = O(m \log n)$ time.[35] The time it takes to check if adding an edge to a pre-existing forest generates a new forest is naively $O(n)$ if we keep track of the vertices in each tree of the forest. Then if we are edge an edge (v_1, v_2) we only need to check if v_2 is in the tree that v_1 is. If it isn't, then we can add the edge and we adjust our records accordingly to indicate that the two trees were merged. Thus the total time required for this part of the algorithm is $O(n^2)$. Thus, the total runtime is $O(n^2 + m \log n)$. □

However, we can do better by using a different data structure to keep track of the forest so far. Notice, that if the graph is dense meaning that $m = \Omega(n^2)$, then this algorithm is optimal. However if $m = o(n^2)$, then we can do better if we can reduce the $O(n^2)$ term to $O(m \log n)$. Specifically, in order to bring the runtime down to $O(m \log n)$ we want to be able to perform $O(n)$ checks of adding an edge to a pre-existing forest generates a forest in $O(m \log n)$ time. The data structure we are going to use is a *disjoint-set data structure*. We actually will do the checks in total $O(n \log n)$ time.[36]

5.5.2 Disjoint-Set Data Structure

Also known as a union-find data structure, this data structure is optimized to keep track of elements partitioned into a number of disjoint subsets. The structure is generally designed to support two operations:

- *Find*: Determine which subset an element is in. Note, testing if two elements are in the same subset can be achieved by two find calls.
- *Union*: Join two subsets together.

There are a lot of ways that we could create a disjoint-set data structure. The simplest way is each set is stored as a list. Then find would take $O(n)$ as we would have to search every list. However, union would be $O(1)$ by simplying pointing the tail of one list to the head of the other. This is the structure, I alluded to in Corollary 5.20 and clearly would not solve the problem we have.

The better approach is to utilize the existing tree structure. We can define each tree in the forest as its own disjoint subset. Furthermore, as a subset is a tree, we can identify each

[35] As $m \leq n^2$, then $\log m \leq 2 \log n$ so $O(\log m) = O(\log n)$.
[36] Also $m \geq n - 1 = O(n)$ for connected graphs.

subset with the root of the tree.

Assume that we kept for each vertex x, we kept track of its parent $\pi(x)$ in the tree and set the parent of the root to itself. Then find(x) is simply recursively calling π until we get to r such that $\pi(r) = r$. The runtime of this is the depth of the tree.

To support the union operation is also simple. To run union(x, y) calculate roots r_x and r_y using find and then set $\pi(r_x) \leftarrow r_y$. However, this will run into issues that in worst case the tree is going to be severly unbalanced. To reconcile this, keep track of with each root, the depth of the tree. Point the root of the tree with smaller depth to the new tree. The depth of the new tree is at most 1 + the greater depth.

With a little more work, you can argue that the trees are going to be fairly balanced. In which case, the depth of any tree is at most $O(\log n)$. So find runs in $O(\log n)$. Since union requires running two find operations and then some $O(1)$ adjustments, its runtime is also $O(\log n)$.

Returning to Kruskal's algorithm, we can apply this data structue to bring the total runtime down to $O(n \log n + m \log n) = O(m \log n)$. We will soon beat this with Prim's algorithm.

Theorem 5.21. *Kruskal's algorithm for Minimum Spanning Tree has a faster runtime of* $O(m \log n)$.

5.5.3 Metric Steiner Tree Problem

Exercise 5.22 (Metric Steiner Tree Problem)**.** Given a finite metric space $G = (V, E, w)$, i.e. $w : E \to \mathbb{R}^+$ satisfies the triangle inequality), and a subset $R \subset V$ of the vertices, find the minimum-weight tree that contains R.

Although this problem looks similar to the MST problem, it is in fact significantly harder. It's actually NP-hard. But, it does exhibit a 2-factor poly. greedy algorithm.

Algorithm 5.23 (Metric Steiner Tree Greedy Algorithm)**.** We compute the all-pair shortest path function $w' : \binom{R}{2} \to \mathbb{R}^+$. We will discus how to do this later. $w'(r_1, r_2)$ is the length of the shortest path $r_1 \rightsquigarrow r_2$. Then, we compute the MST on R under metric w'.

Proof of Correctness. We need to show that this is a 2-factor algorithm. We argue that given a Steiner tree S, that $\exists$ a spanning path T in (R, w') that depth-first searches S, using each of its edges at most twice.

TODO: INSERT IMAGES FROM SCHULMAN SLIDES.

5.5.4 Prim's Algorithm

An alternative algorithm to Kruskal's that performs slightly faster is Prim's algorithm. In Prim's, instead of starting with the n trees consisting of single vertices and connecting trees to form a single tree, we start with an arbitrary vertex and grow the tree by adding the greedy edge. By the greedy edge, I mean the edge of least weight connecting from the tree to vertices not yet connected.

Algorithm 5.24 (Prim's)**.** Given a graph with a weight function $G = (V, E, w)$:

- Make a Priority Queue Q and add all vertices v to Q at distance ∞.
- Choose a root $r \in V$ arbitrarily. Decrease the key of r in Q to 0.
- While Q is non-empty
  - Set $u \leftarrow$ DeleteMin(Q).
  - For all edges $(u, v) \in E$, decrease the key of v in Q to $\min(\text{key}(Q, v), w(u, v))$.z

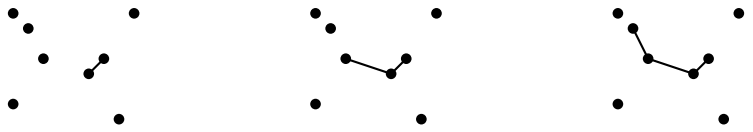

Figure 5.4: The first few iterations of Prim's Algorithm (Algorithm 5.24).

There are n inserts to the Q, n min-deletions, m decrease-keys called. If we implement it with a binary heap, then each of the operations is a $O(\log n)$ operation and the total complexity is $O(n \log n + m \log n) = O(m \log n)$.

However, if we build it witha Fibonacci, then both insertion and decrease-key are $O(1)$ and min-deletions are $O(\log n)$ giving the runtime of $O(m + n \log n)$.[37]

[37] We haven't discussed the implementation of the Fibonacci heap and we won't in this class. But this is

5.6 Clustering and Packing

An important problem, especially in Machine Learning, is the problem of clustering. Assume we have a metric space (V, w) where $w : V^2 \to \mathbb{R}^+$ is a metric i.e. $w(x, y) = 0$ iff $x = y$, $w(x, y) = w(y, x)$ and w obeys the triangle inequality. Our goal is to partition V into k blocks such that points in the same block are 'nearby' and points in different blocks are 'far' apart. This is not meant to be formal.

Most formalizations of 'near' and 'far' result in NP-hard optimization problems (in particular they are non-convex). Fortunately, most of these problems exhibit simple greedy strategies. However, the first problem we will see can actually be solved optimally through a greedy algorithm.

5.6.1 Maxi-min Spacing

Definition 5.25 (Multi-cut)**.** A multi-cut $\mathcal{S}$ is a set of subsets $\{S_1, \ldots, S_k\}$ of V such that $V = S_1 \cup \ldots \cup S_k$ and $S_i \cap S_j = \emptyset$ for all $1 \leq i < j \leq k$.

Definition 5.26 (Spacing)**.** The spacing of a cut $\mathcal{S}$, denoted spacing($\mathcal{S}$) is the least weight edge crossing $\mathcal{S}$.

$$\text{spacing}(\mathcal{S}) = \min_{u \in S_i, v \in S_j, i \neq j} w_{u,v} \tag{5.6}$$

Exercise 5.27 (Maxi-min Spacing Clustering Problem)**.** Given (V, w, k) find a k-way cut $\mathcal{S}$ with maximal spacing.

Lemma 5.28. *Running Kruskal's Algorithm until there are only k trees left produces a maxi-min spacing clustering.*

Proof. Let $\mathcal{S}$ be the Kruskal cut and $\mathcal{R}$ any other cut. Then as $\mathcal{S}$ and $\mathcal{R}$ both cover V, $\exists i$ s.t. S_i intersects two blocks of $\mathcal{R}$. Wlog, call these blocks R_1 and R_2 and as their intersections with S_i are non-empty, exist elements $a \in S_i \cap R_1$ and $d \in S_i \cap R_2$.

Consider the path $a \rightsquigarrow d$ in S_i. As $a \in R_1$ and $d \in R_2$ then there is an edge $b \sim c$ in $a \rightsquigarrow d$ crossing $\mathcal{R}$. But, recall how the Kruskal forest is built; namely edges are added in

a useful result to know. Fibonacci heaps have a better amortized runtime than most other priority queue data structures. Their name comes from their analysis which involves Fibonacci numbers. While they do have an amortized runtime that is faster, they are very complicated to code. Furthermore, in practice they have been shown to be slower because they involve more storage and manipulations of nodes.

non-decreasing order of weight. Therefore $w(b,c) \leq \text{spacing}(S)$. However, as $b \sim c$ crosses $\mathcal{R}$, $\text{spacing}(\mathcal{R}) \leq w(b,c)$. Therefore,

$$\text{spacing}(\mathcal{R}) \leq w(b,c) \leq \text{spacing}(\mathcal{S}) \tag{5.7}$$

Thus $\mathcal{S}$ is optimal to $\mathcal{R}$. As $\mathcal{R}$ was chosen arbitrarily, $\mathcal{S}$ is optimal. □

It should be noted that this algorithm gives us something more than just the maxi-min spacing clustering. The structure of each of the k trees gives us an MST for each cluster.

5.6.2 k-Center Covering and Packing

Definition 5.29 (Ball)**.** Let $G = (V, w)$ be a finite metric space. The (closed) ball with center x and radius r denoted $B(x,r)$ is

$$B(x,r) = \{v \in V : w(x,v) \leq r\} \tag{5.8}$$

The open ball $B^\circ(x,r)$ is the same definition except with a strict inequality $<$ instead of $\leq$.

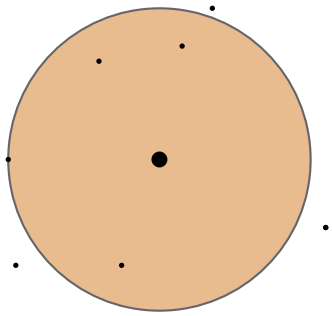

Figure 5.5: The points in orange are in the closed and open ball. The points on the boundary are only in the closed ball. The points outside are in neither.

Definition 5.30 (Covering)**.** Given points $x_1, \ldots, x_k \in V$, and a radius r, we say that $B(x_1,r), \ldots, B(x_k,r)$ are a (r,k)-covering of V if

$$V \subseteq \bigcup_{i=1}^{k} B(x_i,k) \tag{5.9}$$

The obvious two questions to ask for any finite metric space G are:

- For a fixed radius r, what is the *minimal* number k of balls needed to cover V?[38] We will notate this problem as $K_{\text{cover}}(V, r)$ or $K_{\text{cover}}(r)$ when the metric space is obvious.
- For a fixed number k of balls, what is the *minimal* radius r such that V has a (r, k) cover? We notate this problem as $R_{\text{cover}}(V, k)$ or $R_{\text{cover}}(k)$.

Theorem 5.31 (Hsu & Nemhauser '79)**.** *It is NP-hard to approximate $R_{\text{cover}}(k)$ to any multiplicative-factor better than 2.*

Theorem 5.32 (Feder & Greene '88)**.** *If the points belong to Euclidean-space, then it is NP-hard to approximate $R_{\text{cover}}(k)$ to any multiplicative-factor better than 1.82.*

Let's however see a 2-factor greedy approximation algorithm for $R_{\text{cover}}(k)$. First a definition.

Definition 5.33 (Distance to a Set)**.** Given a finite metric space $G = (V, w)$, a point $x \in V$, and a non-empty subset $S \subseteq V$, the distance $w(x, S)$ is defined by

$$w(x, S) = \min_{y \in S} w(x, y). \tag{5.10}$$

Algorithm 5.34 (Gonzalez '85, Hockbaum-Shmoys '86 for k-cover)**.** Pick any $x_1 \in V$. Then for $i = 2, \ldots, k$ (in order), choose x_i to be the furthest point from $x_1, \ldots, x_i$.

Proof of Correctness. For notational convenience define

$$R_C(x_1, \ldots, x_j) = \min\ r \text{ s.t. } \left(V \subseteq \bigcup_{i=1}^{j} B(x_i, r) \right) \tag{5.11}$$

By definition, for optimal $x_1^\star, \ldots, x_k^\star$, $R_{\text{cover}}(k) = R_C(x_1^\star, \ldots, x_k^\star)$. Notice equivalently that

$$R_C(x_1, \ldots, x_j) = \max_{v \in V} w(v, \{x_1, \ldots, x_j\}) \tag{5.12}$$

This is because every point $v \in V$ must be covered and in order to be covered the radius must be at least the distance from v to $\{x_1, \ldots, x_j\}$. The maximum distance overall v sufficiently covers all of V. Any smaller and some $v \in V$ isn't covered.

[38] Note, we provide no restriction of where the centers of the balls must be.

Let $x_1, \ldots, x_k$ be the points as picked by the algorithm. Notice that $w(x, S) \geq w(x, S \cup \{y\})$. Therefore by (5.12), adding a point x_j to the set of centers will only decrease the minimum cover radius. See Figure 5.6 for an illustration. Therefore,

$$R_C(x_1) \geq \ldots \geq R_C(x_1, x_2, \ldots, x_j) \geq \ldots \geq R_C(x_1, \ldots, x_n) = 0 \tag{5.13}$$

Figure 5.6: Progression of the greedy algorithm for $R_{\text{cover}}(k)$. x_1 is chosen arbitrarily and x_2, x_3, x_4 are chosen as the furthest point from the previously chosen points.

Consider how a point x_{k+1} would be selected. It would be selected as the furthest point from $x_1, \ldots, x_k$. Therefore,

$$w(x_i, x_{k+1}) \geq R_C(x_1, \ldots, x_k) \tag{5.14}$$

Adding to that (5.13), we get that the distance between any two points of $x_1, \ldots, x_k$ is at least $R_C(x_1, \ldots, x_k)$.

Notice that no ball of radius $r < R_C(x_1, \ldots, x_k)/2$ can cover two points of $x_1, \ldots, x_{k+1}$. This is a triangle inequality argument. Therefore, no choice of k centers for balls of radius $r < R_C(x_1, \ldots, x_k)/2$ will cover all $k+1$ points.

Therefore the minimal covering radius is at least $R_C(x_1, \ldots, x_k)/2 = R_{\text{cover}}(k)/2$, completing the proof. □

The other set of problems we are interested in are packing problems.

Definition 5.35 (Packing)**.** Given a finite metric space $G = (V, w)$, a (k, r)-packing is a set of pairwise disjoint balls $B(x_1, r), \ldots, B(x_k, r)$ such that $x_1, \ldots, x_k \in V$. By pairwise

disjoint we mean for any $1 \leq i < j \leq k$, $B(x_i, r) \cap B(x_j, r) = \emptyset$.[39]

The analagous two questions can be asked for packings. Notice that the maximization and minimizations have been switched however:

- For a fixed radius r, what is the *maximal* number k of balls capable of being packed in V?[40] We will notate this problem as $K_{\text{pack}}(V, k)$ or $K_{\text{pack}}(r)$ when the metric space is obvious.
- For a fixed number k of balls, what is the *maximal* radius r such that V has a (r, k) packing? We notate this problem as $R_{\text{pack}}(V, k)$ or $R_{\text{pack}}(k)$.

These two problems, covering and packing are in fact duals of one another. Meaning that the optimal solution of one gives a bound on the optimal solution of the other.

Theorem 5.36 (Covering and Packing Weak Duality)**.** $K_{\text{pack}}(r) \leq K_{\text{cover}}(r)$.

Proof. Assume for contradiction that duality does not hold. Then $K_{\text{pack}}(r) > K_{\text{cover}}(r)$. By the pidgeon-hole principle, at least two packing centers y_1, y_2 are contained in one of the covering balls $B(x, r)$. But then $x \in B(y_1, r)$ and $x \in B(y_2, r)$ so the balls are not pairwise disjoint. This contradicts being a packing. □

We leave the following other weak duality as an exercise:

Theorem 5.37 (Covering and Packing Weak Duality)**.** $R_{\text{pack}}(k+1) \leq R_{\text{cover}}(k)$.

We can sumarize all these results into this neat table:

Find	Cover	Pack	Weak Duality
K	$\min\{k : \exists(k, r) \text{ covering}\}$	$\max\{k : \exists(k, r) \text{ packing}\}$	$K_{\text{pack}}(r) \leq K_{\text{cover}}(r)$
R	$\inf\{r : \exists(k, r) \text{ covering}\}$	$\sup\{r : \exists(k, r) \text{ packing}\}$	$R_{\text{pack}}(k+1) \leq R_{\text{cover}}(k)$

[39] A word of warning: It is tempting to draw the balls for a packing in Euclidean geometry when trying to develop an intuition for this idea. Even if two balls overlap when drawn on paper, we are only interested in their overlap in regards to the finite metric space. Meaning, that the balls can overlap when drawn if there are no points of V in the overlapping region.

[40] Note, we provide no restriction of where the centers of the balls must be.

6 Graph Algorithms

6.1 Graph Definitions

Definition 6.1 (Graph)**.** A undirected graph $G = (V, E)$ is a set of vertices V and edges $E \subseteq \binom{V}{2}$ (the set of pairs of elements of V). Notationally, we write $n = |V|$ and $m = |E|$. A directed graph $G = (V, E)$ is a set of vertices V and edges $E \subseteq V \times V$.

Definition 6.2 (Weighted Graph)**.** A weighted graph $G = (V, E, w)$ is a graph with a weight $w : E \to \mathbb{R}$ assigned to each edge. Both undirected and directed graphs can be weighted.

Definition 6.3 (Path and Cycle)**.** A path p from $v_0 \rightsquigarrow v_k$ in G is a list of edges $\big((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\big)$. The length of a path is the number of edges, i.e. k. If th graph is weighted, then the length of the paths is the sum of the weights of the edges.

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \tag{6.1}$$

A cycle is a path with $v_0 = v_k$; a *simple cycle* has no repeated vertices.

Definition 6.4 (Connected)**.** A graph is connected if there is a path between every pair of vertices.

Definition 6.5 (Subgraph)**.** A subgraph of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E \cap \binom{V'}{2}$. The weight of the subgraph G' is

$$w(G') = \sum_{(u,v) \in E'} w(u, v) \tag{6.2}$$

Definition 6.6 (Forest and Trees)**.** A forest in G is a subgraph of G without simple cycles and a tree is a connected forest. We often refer to the distinct connected components of the forest as the trees of the forest. A *spanning tree* is a tree in G that connects all the vertices in G.

6.2 Single-Source Shortest Paths

Definition 6.7 (Single-Source Shortest Paths)**.** Given a directed weighted graph $G = (V, E, w)$ with non-negative weights $w : E \to \mathbb{R}^+$ and a vertex $s \in V$, the single-source

shortest paths is the family of shortest paths $s \rightsquigarrow v$ for every vertex $v \in V$.

The natural question of course is what is an efficient algorithm to calculate single-source shortest paths. We first notice that the family of single-source shortest paths has a compact representation as a directed tree rooted at s.

Lemma 6.8. *If $(s = v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k = v)$ is a shortest path $s \rightsquigarrow v$, then for any $i \leq k$, the shortest path $s \rightsquigarrow v_i$ is $(s = v_0, v_1), (v_1, v_2), \ldots, (v_{i-1}, v_i)$.*

Proof. Assume for some $i \leq k$, that there was a shorter path $p' : s \rightsquigarrow v$. Then the path p' with $(v_i, v_{i+1}), \ldots, (v_{k-1}, v_k)$ would be a shorter path, contradicting the assumed optimality. □

As a consequence of Lemma 6.8, we can see that the set of single-source shortest paths form a directed tree rooted at s. Furthermore, this offers us an efficient method of storing the single-source shortest paths. We could store them as the $n-1$ edges forming the tree, or for each vertex $v \in V$, we can store parent(v). The latter also allow efficient lookup of the shortest-path for any vertex: Recursively call parent$(\cdot)$ until s is reached.

We begin with a special case where $w = 1$ for all edges. The Breadth First Search Algorithm provides an efficient method for finding the single-source shortest paths in this scenario.

Algorithm 6.9 (Breadth First Search)**.** Let G be a directed graph and u a vertex in G. Initialize all the vertices as unmarked and let Q be an empty queue. Mark u and push u onto Q. While Q isn't empty,

- Pop a vertex from Q; call it a.
- For every vertex b s.t. $(a, b) \in E$, if b is unmarked,
  - Push b onto Q.
  - Set parent$(b) \leftarrow a$.
  - Set dist$(u, b) \leftarrow$ dist$(u, a) + 1$.

Complexity. If G is given by adjacency lists, the runtime is $O(n + m)$. If G is given by adjacency matrices, it costs us to look for the neighbors yielding a runtime of $O(n^2)$.

Definition 6.10. Let $V(u,k) = \{v \in V : \text{dist}(u,v) = k\}$, the set of vertices of distance k from u. We define distance in this case to be the length of the shortest path $u \rightsquigarrow v$.[41]

Lemma 6.11. *There exist times $t_0 < t_1 < \ldots < t_k < \ldots$ s.t. at time t_k,*

1. *The queue Q contains exactly $V(u,k)$.*
2. *The set of marked vertices is precisely $\bigcup_{i=0}^{k} V(u,i)$.*

Proof. Proceed by induction. The base case for $k = 0$ is trivial. For $k > 0$, at time t_{k-1}, Q contains exactly $V(u,k-1)$. Define z as the last element in the queue. Define t_k as the time after z has been popped and all its neighbors added to Q.

Then at t_k, we can guarantee that no vertex of $V(u,0) \cup \ldots \cup V(u,k-1)$ is in Q because of the marking of vertices. Furthermore, every vertex in the Q had a neighbor in $V(u,k)$ because of how vertices are added to the queue. These two statement imply that for every $v \in Q$ at t_k, $\text{dist}(u,v) > k-1$ and $\text{dist}(u,v) \leq k$, proving $\text{dist}(u,v) = k$ and $v \in V(u,k)$. Lastly, the vertices marked between t_{k-1} and t_k are $V(u,k)$ proving the second statement. □

To move to general weight functions, $w : E \to \mathbb{R}^+$ or even $w : E \to \mathbb{R}$ (in certain cases), we will need to do something slightly more tricky. But luckily these algorithms will be greedy just like this one.

6.3 Dijkstra's Algorithm

We can adapt the previous algorithm to solve $w : E \to \mathbb{R}^+$. We need to adjust the queue from Algorithm 6.9 to a *priority queue*.

Algorithm 6.12 (Dijkstra's Algorithm)**.** Let $G = (V, E, w)$ be a directed graph with $w : E \to \mathbb{R}^+$ and $u \in V$. Generate a priority queue P and push onto P evey element $v \in V$ with key ∞. Decrease key of u to 0. While P isn't empty,

- Pop the min weight element from P; call it a.
- For every vertex b s.t. $(a,b) \in E$, if $\text{key}(b) > \text{key}(a) + w(a,b)$,

[41] This is a well-defined metric.

  - Decrease key of b to $\text{key}(a) + w(a, b)$.
  - Set $\text{parent}(b) \leftarrow a$.
- Set $\text{dist}(u, a) \leftarrow \text{key}(a)$.

The resulting try by $\text{parent}(\cdot)$ is a single-source shortest paths function (not-necessarily unique) and $\text{dist}(u, \cdot)$ is the distance function.

Lemma 6.13. $\text{dist}(u, \cdot)$ *is the correct distance function.*

Proof. The proof is by induction and similar to that of Lemma 6.11. Let $\tilde{d}(u, \cdot)$ be the actual distance function. Clearly $\text{key}(u) = 0$ and is never changed so $\text{dist}(u, u) = 0 = \tilde{d}(u, u)$.

We claim that $\text{dist}(u, v) \geq \tilde{d}(u, v)$ for any $v \in V$. Let y be the *first* vertex popped with $\text{dist}(u, y) > \tilde{d}(u, y)$. Define $U \leftarrow$ set of vertices popped before y. Let $u \rightsquigarrow y$ be the shortest path (the one corresponding to $\tilde{d}(u, y)$). Define v as the last element of U in the path. Let x be the next element in the path.

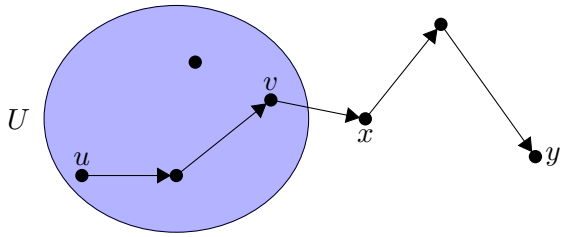

Figure 6.1: Diagram of Lemma 6.13.

By induction, we argue correctness of every element in U. Since x is on the shortest path $u \rightsquigarrow y$ then this is also the shortest path to x (Lemma 6.8). Therefore,

$$\tilde{d}(u, x) = \text{dist}(u, v) + w(v, x). \tag{6.3}$$

Then the decrease key call after popping v will guarantee that $\text{dist}(u, x) = \tilde{d}(u, x)$. If $x = y$, then y isn't a counterexample. So assume $x \neq y$. But as y was the *first* element popped after U, then y was popped before x. So,

$$\text{dist}(u, y) = \text{key}(y) < \text{key}(x) = \tilde{d}(u, x) \leq \tilde{d}(u, y) < \text{key}(y) \tag{6.4}$$

This is a contradiction, so no such first y exists proving correctness. □

Theorem 6.14 (Dijkstra's Runtime)**.** *If Dijkstra's is implemented with a binary heap, the runtime is* $O((n+m)\log n)$*. If Dijkstra's is implemented with a Fibonacci heap, the runtime is* $\Theta(m + n\log n)$*.*

Proof. From the algorithm, we can find that the runtime is $O(m \cdot T_{\text{decrease key}} + n \cdot T_{\text{pop min}})$. The time to decrease a key in a binary heap is $O(\log n)$ and to pop min is $O(\log n)$. Hence the statement. If we use a Fibonacci heap, the cost to decrease key drops to $O(1)$ explaining the runtime speedup. □

6.4 Bellman-Ford Algorithm

We now want to extend this to negative weights as well. However, this can run into some difficulties.

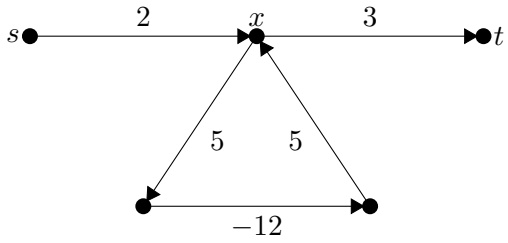

Figure 6.2: Example of an issue with negative weights.

In Figure 6.4, we notice that traversing the triangle yields a weight of -2. Its not difficult to see that the only paths $s \rightsquigarrow t$ are $s \to x$, traversing the triangle n times, and then $x \to t$. The weight of this path is $5 - 2n$ and so the minimum weight path is $-\infty$.

We call a cycle such as the triangle a *negative weight cycle* and we want to identify if these exist in the problem. Therefore, we can adjust the problem definition to:

Exercise 6.15. Let G be a directed weight graph with $w : E \to \mathbb{R}$ and $u \in V$. For every $v \in V$, report if there is a negative weight cycle on some path $u \rightsquigarrow v$ or if not the weight of the shortest path $u \rightsquigarrow v$.

The Bellman-Ford algorithm will fix this issue. Both Bellman-Ford and Dijkstra's work be relaxing the distance function. Meaning, gradually the estimate of the distance is reduced

and reduced until the optimal is achieved. However, Dijkstra's algorithm is a *greedy* algorithm and this strategy cannot work for our problem. (There are plenty of counterexamples you can draw.) We are going to go back and apply a dynamic programming approach to catch the negative weight cycles.

Algorithm 6.16 (Bellman-Ford)**.** Given a directed graph $G = (V, E, w)$ with $w : E \to \mathbb{R}$ and a vertex $u \in V$, set $\text{key}(u) = 0$ and $\text{key}(v) = \infty$ for $v \neq u$. Set the iteration number $I = 0$. Repeating until no key values change:

1. If $I = n$, halt and report that a negative cycle was found.
2. For each edge $(a, b) \in E$ (in any order)
   (a) Define $c \leftarrow \text{key}(a) + w(a, b)$.
   (b) If $c < \text{key}(b)$, then set $\text{key}(b) = c$ and $\text{parent}(b) = a$.
3. Increase $I \leftarrow I + 1$.

There are two principle differences between Dijkstra's and Bellman-Ford. One is that the edges we used to consider per iteration of the algorithm were the greedy choice of edges and now we need to consider all edges. The second is the halting condition if keys are decreasing for more than n rounds.

Proof of Correctness: By similar arguments to what we made for Dijkstra's, $\text{key}(v)$ is always an upper bound for $\text{dist}(u, v)$ (the true distance). Let k be the length of the shortest least weight path $u \rightsquigarrow v$. Due to the tree structure, by the kth iteration, $\text{key}(v) = \text{dist}(u, v)$.

If the graph lacks negative-weight cycles then as the length of any path without cycles it at most $n - 1$, by the $n - 1$th iteration, the key values should halt changing.

On the other hand, if there is a negative-weight cycle, then there is a path from u to every vertex on that negative weight cycle and that path has at most $n - 1$ edges. So by the $n - 1$th iteration, every key $< \infty$. The negative weight cycle must contain at least one edge of negative weight (call it (a, b)). Then on the nth iteration, $\text{key}(b)$ will decrease by at least $\big|w(a, b)\big|$. Then the alg. halts appropriately. □

Complexity. If the graph is given as adjacency lists, the runtime is $O(mn)$ as it runs n iterations of testing each edge. If the graph is given as an adjacency matrix, it takes $O(n^2)$ time per iteration for a total runtime of $O(n^3)$. A more nuanced analysis shows that for graphs lacking negative cycles with the length of least weight paths at most ℓ, the runtime is $O(m\ell)$.

6.5 Semi-Rings

We take a brief interlude from graph problems to introduce some more math to make the analysis nicer.

Definition 6.17 (Semi-ring)**.** A semi-ring is a set K along with two operations which we call $(+), (\times) : K \times K \to K$. These operations must satisfy:

- $(+)$ is commutative and associative. Meaning for $a, b, c \in K$, $a(+)b = b(+)a$, $(a(+)b) + c = a(+)(b(+)c)$.
- There is an identity element for $(+)$ called 0. For $a \in K$, $a(+)0 = a$.
- $(\times)$ is commutative and associative. Meaning for $a, b, c \in K$, $a(\times)b = b(\times)a$, $(a(\times)b)(\times)c = a(\times)(b(\times)c)$.
- There is an identity element for $(\times)$ called 1. For $a \in K$, $a(\times)1 = a$.
- There is a distributive law. For $a, b, c \in K$,

$$a(\times)(b(+)c) = (a(\times)b)(+)(a(\times)c)$$

A ring is a semi-ring with the additional constraint that $(+)$ has inverses. Given the notation of $(+), (\times)$ a natural realization of a semi-ring is with $K = \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{Z}/k\mathbb{Z}$ and $(+)$ and $(\times)$ being the operations that we traditionally define them as.[42] But, we can define $(+)$ and $(\times)$ as other functions as well and still produce semi-rings.

Take for example, $K = \mathbb{R} \cup \infty$, $(+) = \min$, and $(\times) = +$. We can verify that this is a semi-ring. The minimization operator is certainly commutative and associative as $\min(a, b) = \min(b, a)$ and $\min(a, \min(b, c)) = \min(a, b, c) = \min(\min(a, b), c)$. The identity

[42] All of these examples are rings.

element for $(+)$ is ∞ as $\min(a, \infty) = a$. As $(\times)$ is addition, then the criteria hold with identity element of 0. The distributive law is a fun exercise to check.

Definition 6.18 (Min-Sum Semi-Ring)**.** The min-sum semi-ring is defined by $K = \mathbb{R} \cup \infty$, $(+) = \min$, and $(\times) = +$.

Why do we care about semi-rings? We find that certain computational rpboelms can be expressed as known problems over a semi-ring. For example, we will see that the all-pairs shortest path problem can be solved rather easily via matrix multiplication over the min-sum semi-ring K.

6.6 All Pairs Shortest Paths

Definition 6.19 (All Pairs Shortest Paths)**.** Given a directed weighted graph $G = (V, E, w)$, for each $u, v \in V$, find the shortest path $u \rightsquigarrow v$.

Definition 6.20 (Adjacency Matrix)**.** Given a weighted directed graph $G = (V, E, w)$, the adjacency matrix A is the unique $n \times n$ matrix defined by

$$A_{ij} = \begin{cases} 0 & (i,j) \notin E \\ w_{ij} & (i,j) \in E \end{cases} \tag{6.5}$$

Lemma 6.21. *For a weighted directed graph $G = (V, E, w)$ with adjacency matrix A, the shortest path $i \rightsquigarrow k$ of length $\leq \ell$ has weight $(A^\ell)_{ik}$ where the matrix product is computed over the min-sum semi-ring.*

Proof. We prove by induction. For $\ell = 1$, this is trivially true: The path $i \rightsquigarrow k$ only exists if $(i, k) \in E$ and has weight $w_{ik} = A_{ik}$. Assume true now up to ℓ. Since we are computng over the min-sum semi-ring

$$(A^{\ell+1})_{ik} = (A \cdot A^\ell)_{ik} = \min_j \left(A_{ij} + (A^\ell)_{jk} \right) \tag{6.6}$$

If a path exists $i \rightsquigarrow k$ of length $\leq \ell+1$, it must have a second vertex j. The path weight will be the weight of $i \to j$ plus the weight of $j \rightsquigarrow k$ under the condition that the length $\leq \ell$. The equation above, exhaustively minimizes over all possible j proving correctness. □

This gives a natural algorithm for calculating the solution to the all-paths problem due to the following lemma you are probably familiar with.

Lemma 6.22. *For a graph G with n vertices, then the length of the shortest path $u \rightsquigarrow v$ is at most n (if it exists).*

Proof. If a path longer than n exists, it must have a repeated vertex by the pidgeon-hole principle. We can then cut out the loop from the path and decrease the length. We can iteratively apply this until we get a path of length $\leq n$. Therefore, the shortest path must have length $\leq n$. □

As the longest path has length $\leq n$, we only need to look at A^n to calculate the single-source shortest paths. This presents the following first-take algorithm:

Algorithm 6.23 (All-pairs Shortest Path)**.** For a graph G, calculate the adjacency matrix A. Using repeated-squaring calculate A^n under the min-sum semi-ring. Then A_{ij} is the minimum weiht of the path $i \rightsquigarrow j$. This algorithm runs in $O(n^\omega \log n)$ where ω is the exponent of matrix multiplication (naïvely $\omega = 3$).

Proof. The prior lemmas tell us that we onl need to look at A^n. Repeated matrix multiplication will take $O(\log n)$ matrix multiplications and their runtime is each going to be $O(n^\omega)$. □

There are a lot of interesting open questions about this approach. For one, how fast is min-sum matrix multiplication Can we improve past $O(n^\omega)$?

6.7 Floyd-Warshall Algorithm

In practice, we know $\omega < 2.373$ (thanks Virginia Williams!) but the constant associated with the runtime is gigantic. We will see how to prove that $\omega < \log_2 7$ later but for now, we can build a quick Dynamic Programming algorithm for all-pairs shortest path.

In the previous example, we generated 'subproblems' decided by if there was a path of $u \rightsquigarrow v$ that contained $\leq k$ vertices. For the next part, let's label the vertices with $1, \ldots, n$ for simplicity. We can define a subproblem as the least weight path from $i \rightsquigarrow j$ with only internal vertices from $\{1, \ldots, k\}$. See Figure 6.7 for an example.

We can define the solution to a subproblem in therms of $O(1)$ subsubproblems. Let $D_k(i,j)$ be the least weight path $i \rightsquigarrow j$ with only internal vertices from $\{1,\ldots,k\}$.

$$D_k(i,j) = \min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\} \tag{6.7}$$

The proof of this statement's correctness comes from its exhaustiveness. Either the minimal path includes vertex k or it doesn't. If it doesn't, then the answer is iven by the subproblem $D_{k-1}(i,j)$. If it does include k, then we can write it in terms of the two subproblems $D_{k-1}(i,k)$ and $D_{k-1}(k,j)$. The base cases are $D_0(i,j) = w_{ij}$.

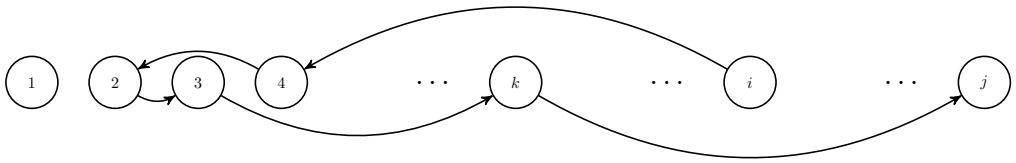

Figure 6.3: Example path $i \rightsquigarrow j$ that only includes vertices from $\{1,\ldots,k\}$

This yields the following Dynamic Programming Algorithm:

Algorithm 6.24 (Floyd-Warshall Algorithm for All-Pairs Shortest Path)**.** Initialize $D_0(i,j) = w_{ij}$. For $k = 1,\ldots,n$, we apply (6.7). The shortest-path $i \rightsquigarrow j$ has weight $D_n(i,j)$.

Proof of Correctness: We proved the inductive correctness of the definition earlier. Inductively, we consider every path when we consider $D_n(i,j)$. □

Complexity: The time and space is $O(n^3)$ due to the size of the table and each subproblem has $O(1)$ additional runtime over its subproblems. □

6.8 Johnson's Algorithm

Can we do better if we know some properties of the graph? For example, a very common problem is what can we say if the graph is sparse? We know m, the number of edges, is bounded above by $O(n^2)$. However, our runtime for Floyd-Warshall was not dependent on m.

Assume we had a potential function $\psi : V \to \mathbb{R}$ on the vertices. We can define a new weight function that accounts for the potential by:

$$\tilde{w}_{uv} = w_{uv} + \psi(u) - \psi(v) \tag{6.8}$$

We can see that shifting our weight function by this potential doesn't change our problems too much.

Lemma 6.25. *For any path $\gamma = (v_0, \ldots, v_k)$ be a path in the graph. And define the weight of a path as the sum of the weights of the edges in the path. Then*

$$\tilde{w}_\gamma = \tilde{w}_\gamma + \psi(v_0) - \psi(v_k) \tag{6.9}$$

Proof. The sum telescopes:

$$\begin{aligned} \tilde{w}_\gamma &= \sum_{j=0}^{k-1} \tilde{w}_{v_j v_{j+1}} \\ &= \sum_{j=0}^{k-1} w_{v_j v_{j+1}} + \psi(v_j) - \psi(v_{j+1}) \\ &= w_\gamma + \psi(v_0) - \psi(v_k) \end{aligned} \tag{6.10}$$

□

Corollary 6.26. *As the weight of any path $u \rightsquigarrow v$ is augmented by $\psi(u) - \psi(v)$, then solving the least weight problem over metric w or $\tilde{w}$ is equivalent.*

The question is then, what potential ψ can we write to solve this problem? A nice potential would be one with which we could argue that $\tilde{w}_{ij} \geq 0$. This way, we could just treat the problem as n single-source shortest path problems and apply Dijkstra's (one from every vertex). We show that such a potential function necessarily exists.

Lemma 6.27. *For any graph $G = (V, E, w)$ without negative-cycles, there exists a potential function ψ such that $\tilde{w}_{ij} \geq 0$ for all i, j. Furthermore, this potential can be calculated in $O(nm)$ time.*

Proof. Pick a vertex $s \in V$ as the start node. set $\psi(v)$ as the weight of the shortest path $s \rightsquigarrow v$. This can be calculated in $O(nm)$ with Belman-Ford algorithm. Therefore, for all

$u, v \in V$,

$$\psi(u) + w_{u,v} \geq \psi(v) \tag{6.11}$$

as this equivalent to the triangle inequality statement, $w_{s,u} + w_{u,v} \geq w_{s,v}$. From here, rearranging the previous equation shows that $\tilde{w}$ is non-negative. □

We can combine all these parts together to form the following algorithm:

Algorithm 6.28 (Johnson's)**.** Given a graph $G = (V, E, w)$ we calculate a potential function ψ as described in Lemma 6.27. If the subroutine Bellman-Ford declares that negative weight cycles exist, then abort. Otherwise, using the potential function, calculate $\tilde{w}$. Then, for each vertex $u \in V$, run Dijkstra's algorithm starting at v.

Proof of Correctness: The correctness follows as a direct consequence of Lemma 6.27 and the correctness of Bellman-Ford and Dijkstra. □

Complexity: The runtime is $O(mn + n^2 \log n)$ because Bellman-Ford runs in $O(mn)$ and we run n copies of Dijkstra's which run in $O(m + n \log n)$. □

6.9 Cut for Space: Savitch's Algorithm

We end this somewhat long excursion into shortest-path algorithms with a seemingly simple problem about how to calculate if a path exists but use very little space.

Exercise 6.29. What is the minimal space complexity of an algorithm to decide for a graph G with vertices s, t if there exists a path $s \rightsquigarrow t$?

We will show that there is a $O(\log^2 n)$-space algorithm for deciding this problem. This problem is actual an **NL**-complete problem, meaning that it is a complete problem that can be solved non-deterministically using $O(\log n)$ space. If someone can show that this problem can be solved deterministically in $O(\log n)$ space, this will show the complexity class collapse $\mathbf{L} = \mathbf{NL}$.

First a word on how small $O(\log n)$ space is. The space it takes to write down a specific vertex in memory requires $O(\log n)$ space. So, using $O(\log n)$ space means that the algorithm is only writing down effectively a constant number of vertices. To me that is an absurdly small! $O(\log^2 n)$ is a little nicer, this means we can write down logarithmically

many vertices. Furthermore, remember that space and time have a trade-off. We expect that the time complexity is going to be pretty bad...
Do you remember Zeno's paradox? In order for the tortoise to complete th race, he must first go halfway. But to go halfway, he must first go a quarter-way and so on...

What if we guessed (as in tried all n possibilities) for a halfway vertex of the path $s \rightsquigarrow t$. Then we can recursively guess a quarterway point, etc. As the path has length at most n, and this recursive problem has depth $O(\log n)$ and at each recursive step we only need to store $O(\log n)$ information. Let's formalize:

Algorithm 6.30 (Savitch's)**.** Define $\text{Savitch}(u, v, k)$ if there exists a path $u \rightsquigarrow v$ of length $\leq 2^k$. We return $\text{Savitch}(u, v, \lceil \log n \rceil)$. We can define $\text{Savitch}(u, v, 0) = \text{TRUE}$ if only if u and v are connected. Recursively,

$$\text{Savitch}(u, v, k) = \bigvee_{m \in V} \big(\text{Savitch}(u, m, k-1) \wedge \text{Savitch}(m, v, k-1)\big) \tag{6.12}$$

Proof of Correctness: Since we exhaustively consider all midpoints recursively, if a path exists it will be found. □

Complexity: As the algorithm is run with $k = O(\log n)$ at the top level, then the total depth is $O(\log n)$. Furthermore, each subproblem only requires storing the vertices defining the subproblem in the stack. This is a constant number of vertices, so $O(\log n)$ space. Applying a depth-first search (which we cover next), we can solve this problem in $O(\log^2 n)$ space.

6.10 Depth-First Search (still in draft...)

7 Branch and Bound

7.1 Preliminaries

So far we have covered Dynamic Programming, Greedy Algorithms, and (some) Graph Algorithms. Other than a few examples with Knapsack and Travelling Salesman, however, we have mostly covered **P** algorithms. Now we turn to look at some **NP** algorithms. Recognize, that, we are not going to come up with any **P** algorithms for these problems but we will be able to radically improve runtime over a naïve algorithm.

Specifically we are going to discuss a technique called Branch and Bound. Let's first understand the intuition behind the technique. Assume we are trying to maximize a function f over a exponentially large set X. However, not only is the set exponentially large but f might be computationally intensive on this set. Fortunately, we know of a function h such that $f \leq h$ everywhere.

Our naïve strategy is to keep a maximum value m (initially at $-\infty$) and for each $x \in X$, update it by $m \leftarrow \max\{m, f(x)\}$. However, an alternative strategy would be to calculate $h(x)$ first. If $h(x) \leq m$ then we know $f(x) \leq h(x) \leq m$ so the maximum will not change. So we can effectively avoid computing $f(x)$ saving us a lot on computation! However, if $h(x) > m$ then we cannot tell anything about $f(x)$ so we would have to compute $f(x)$ to check the maximum. So, in the worst case, our algorithm could take the same time as the naïve algorithm. But if we have selected a function h that is a tight bound (i.e. $h - f$ is very small) then we can save a lot of computational time.

Note: this is not Branch and Bound; this is only the intuition behind the technique. Branch and Bound requires more structure but has a very similar ring to it.

7.2 Knapsack, an example

Now that we've gone over some broader intuition, let's try an example and then come back for the formalism. Let's consider the Knapsack problem. We can rewrite Knapsack as a n-level decision problem, where at each level i, we choose whether to include item i in our bag or not. Figure 7.1 gives the intuition for this decision tree. As drawn, the left hand decision matches choosing the item and the right hand decision matches leaving the item.

In the end, we are left with 2^n nodes, each representing a unique choice of items and its respective weight and value. We are then left with maximizing over all choices (leaves) that satisfying our weight constraint W.

However, this wasn't efficient! We spent a lot of time traversing parts of the tree that we knew had surpassed the weight constraint. So, a smarter strategy would be to truncate the tree at any point where the weight up to that point has passed the weight constraint. We see this in Figure 7.2. Now, the problem is a lot faster as we've seriously reduced our runtime. Notice though, we cannot guarantee any truncations of the graph, so the asymptotic runtime of this problem is still $O(2^n)$ as we have 2^n possibilities for item sets.

7.3 Formalism

Let's formalize the intuition we built from Knapsack and generate a rigorous structure which we can use to solve other Branch and Bound problems. These are the qualities we are looking for in a Branch and Bound problem.

Properties of Branch and Bound Algorithm.

1. The problem should be expressible as a maximization of a function $f : L \to \mathbb{R}$ where L is the set of leaves of some tree T.
2. We can define a function $h : T \to \mathbb{R}$ defined on all nodes of the tree such that $f(\ell) \leq h(t)$ if ℓ is a descendant leaf of t. (Here t is any node in the graph. Note ℓ and t could be the same).

Note we can also write minimization problems in this manner by considering $(-f)$. So, for the rest of this lecture, I am only going to discuss maximization problems without loss of generality. This gives us a natural generic algorithm for solving a Branch and Bound problem.

Algorithm 7.1 (Branch and Bound Algorithm)**.** For any problem,

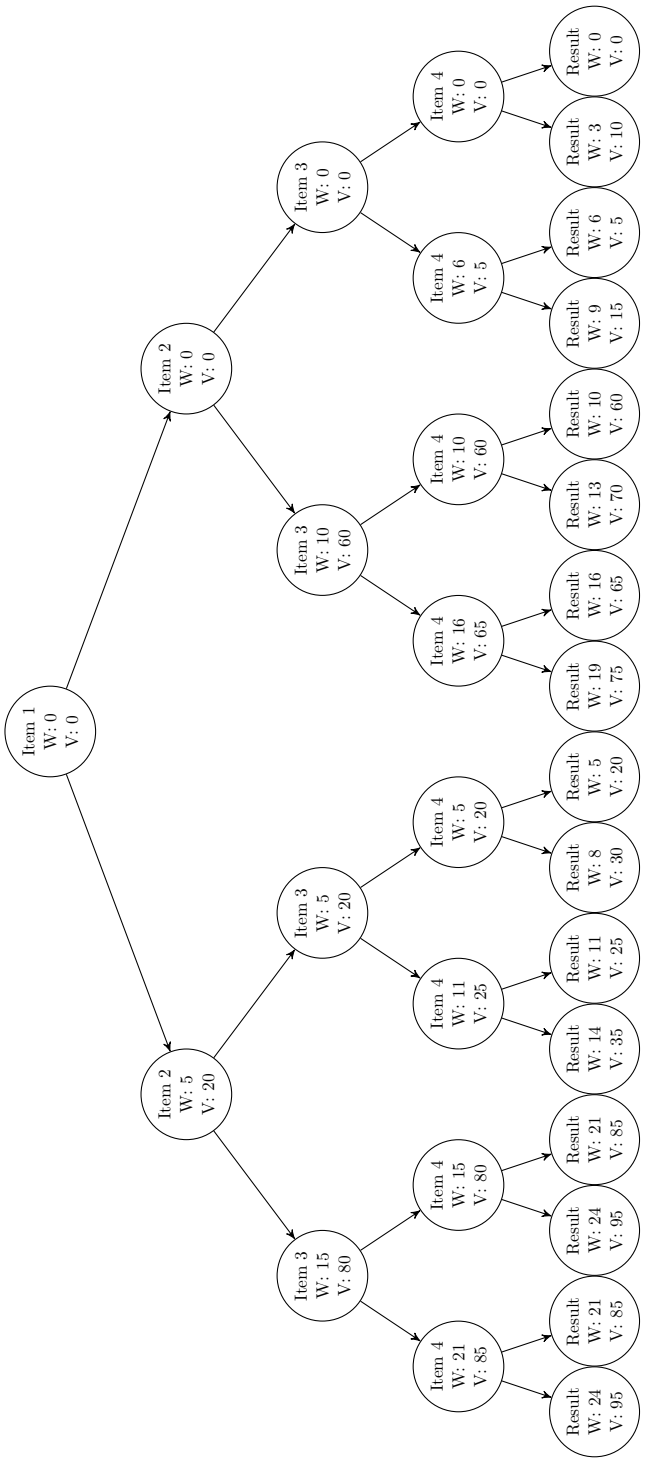

Figure 7.1: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 10. Here all paths are considered. Next figure shows the truncation we can do.

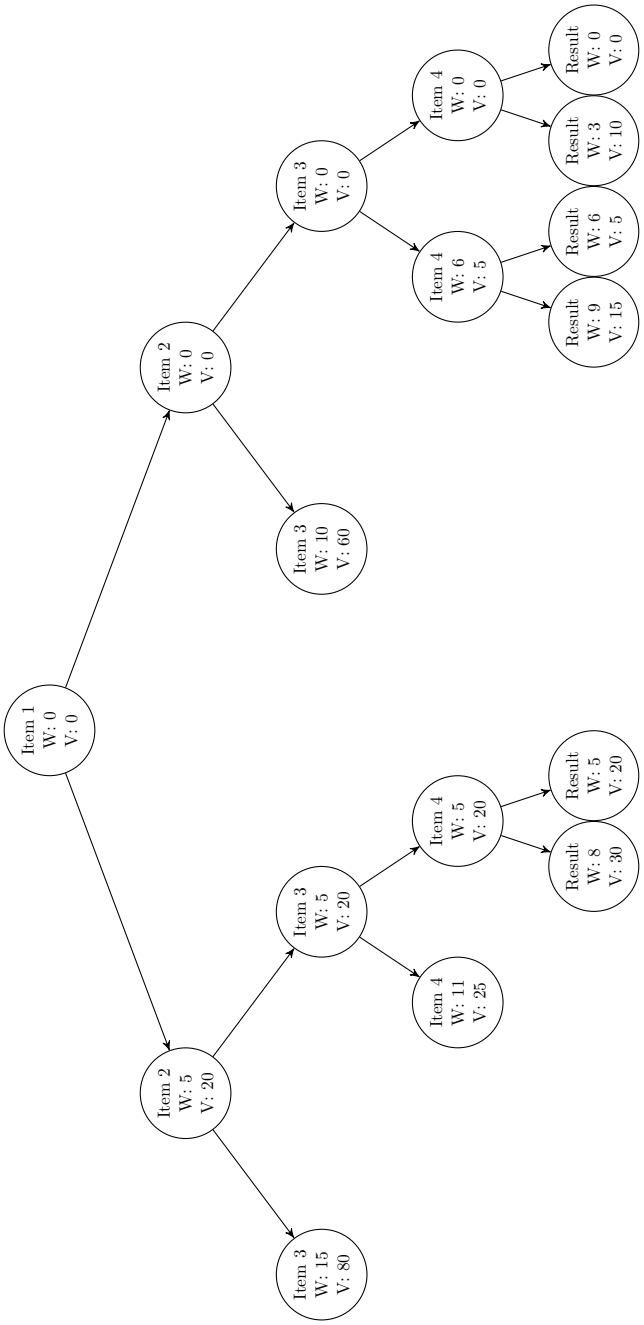

Figure 7.2: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 9. Here we truncate unfeasible paths early.

1. Write the problem as a maximization of $f : L \to \mathbb{R}$ where L is the set of leaves of a tree T.
2. Let $m \leftarrow -\infty$ and $x \leftarrow$ null. This is the current maximum value achieved and the leaf achieving it.
3. Beginning at the root r of T, traverse the tree in pre-order (i.e. run a calculation at node t, the traverse recursive each of its children). For every node t encountered do the following:
   (a) If t isn't a leaf, check if $h(t) < m$. If so, then truncate the traversal at t (i.e. don't consider any of t's descendants)
   (b) If t is a leaf, calculate $f(t)$. If $f(t) < m$, $m \leftarrow f(t)$ and $x \leftarrow t$ (i.e. update the maximum terms)
4. Return x.

Algorithm Correctness. The naïve algorithm would run by traversing the tree and updating the maximum at each leaf ℓ and then returning the maximum. The only difference we make is, we claim we can ignore all the descendants of a node t if $h(t) < m$. Why? For any descendant ℓ of t by the property above $f(\ell) \leq h(t)$. As $h(t) < m$ then $f(\ell) < m$ as well. Therefore, the maximum will never update by considering ℓ. As this is true for any descendant ℓ, we can ignore its entire set of descendants.

7.4 Formalizing Knapsack

Let's apply this formalism concretely to the Knapsack problem. We've already seen the natural tree structure. So let's define the functions f and h. Every leaf L can be expressed as a vector in $\{0,1\}^n$ where the ith index is 1 if we use item i and 0 if not. So $L = \{0,1\}^n$. Furthermore, we can define T as

$$T = \left\{ \{0,1\}^k \;\middle|\; 0 \leq k \leq n \right\} \tag{7.1}$$

Informally, every node in T at height k is similarly expressible as $\{0,1\}^k$ where the ith index again represents whether item i was chosen or not. Each node $v \in \{0,1\}^k$ for $0 \leq k \leq n$

has children $(v, 1)$ and $(v, 0)$ in the next level.

We can define the function $f : L = \{0, 1\}^n \to \mathbb{R}$ as follows:

$$f(\ell) = f(\ell_1 \dots \ell_n) = \mathbb{1}_{\{\ell_1 w_1 + \ldots + \ell_n w_n \leq W\}} \cdot (\ell_1 v_1 + \ldots \ell_n v_n) \tag{7.2}$$

Here $\mathbb{1}_{\{\cdot\}}$ is the indicator function. It is 1 if the statement inside is true, and 0 if false. Therefore, what f is saying in simple terms is that the value of the items is 0 if the weights pass the capacity and otherwise is the true value $\ell_1 v_1 + \ldots + \ell_n v_n$. Define the function $h : T \to \mathbb{R}$ as:

$$h(\mathbf{t}) = f(t_1 \dots t_k) = \begin{cases} \infty & \text{if } t_1 w_1 + \ldots t_k w_k \leq W \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

Let's verify that f and h have the desired relation. If $h(t) = \infty$ then obviously $f(\ell) \leq h(t)$, so we don't need to check this case. If $h(t) = 0$ then $t_1 w_1 + \ldots + t_k w_k > W$. Any descendant ℓ of t will satisfy $\ell_1 = t_1, \ldots, \ell_k = t_k$. Therefore,

$$(\ell_1 w_1 + \ldots + \ell_k w_k) + (\ell_{k+1} w_{k+1} + \ldots + \ell_n w_n) \geq t_1 w_1 + \ldots t_k w_k > W \tag{7.4}$$

Therefore, the indicator function is 0 so $f(\ell) = 0$ so $f(\ell) \leq h(t)$. So, we have completely written Knapsack in the Branch and Bound formalism. Effectively, we've converted our intuition of disregarding any item set that exceeds the weight capacity early into a formal function.

7.5 Traveling Salesman Problem

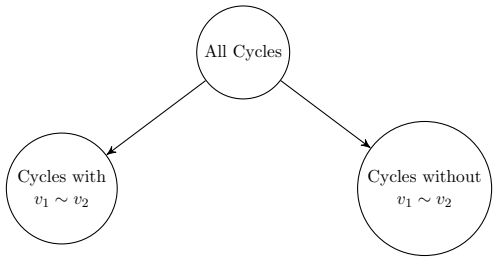

Figure 7.3: 1st level breakdown of the tree

Remember the traveling salesman problem? Given a set of verticies $V = \{v_1, \ldots, v_n\}$ and

a weight function $w : V^2 \to \mathbb{R}^+$, we want to find a Hamiltonian cycle of least weight.

So, if we were to define a tree structure to pose the Traveling Salesman Problem as a Branch and Bound problem, we would want the leaves of the tree to be Hamiltonian cycles. What about the rest of the tree? Well consider the tree as shown in Figure 7.3. At each level of the tree, we restrict ourselves to cycles that either contain or don't contain a specific edge. We do this for all $\binom{n}{2} = O(n^2)$ levels. Now, some of these branches will produce paths that aren't Hamiltonian cycles (more than two edges in the cycle per vertex) and we can truncate the tree there. Everything leaf remaining, is a Hamiltonian cycle.

Naïvely the tree contains an exponential number of vertices, so we are going to come up with a truncating function. The leaves of our tree are precisely Hamiltonian cycles and the objective function is $f(H) =$ sum of weight of edges in Hamiltonian cycle H.

As TSP is a minimizing problem, the truncating function h is going to be a lower bound for f. A simple naive lowerbound for the TSP is

$$\frac{1}{2}\sum_{v\in V}(\text{the 2 minimum weight edges incident on } v) \tag{7.5}$$

We can extend this intuition to the entire tree. If we are forced to include edge $v_i \sim v_j$ in our cycle, then for vertex v_j, one of the two edges weights we include in the truncating function sum must be $w(v_i, v_j)$. Similarly, if we are forced to exclude edge $v_i \sim v_j$ then $w(v_i, v_j)$ cannot be one of the two edges weights we include in the truncating function sum.

Its a little tedious, but you can work through the details to verify that this truncation function is well defined.[43]

[43] It is one of the few cases where it is easier to describe the function in words rather than symbol manipuations.

8 Divide and Conquer

Written by Ethan Pronovost. epronovost@caltech.edu

8.1 Mergesort

The divide and conquer technique is a recursive technique that splits a problem into 2 or more subproblems of equal size. General problems that follow this technique are sorting, multiplication, and discrete Fourier Transforms. For an example of this, consider the Mergesort algorithm.

Algorithm 8.1 (Mergesort)**.** Take a list $x[n]$ of integers. In the base case, if $n \leq 1$, return x.

Otherwise, split the list into the left and right halves, $l[\lceil \frac{n}{2} \rceil]$ and $r[\lfloor \frac{n}{2} \rfloor]$. Recursively sort both these lists. To combine these two lists into one list $a[n]$, set $i, j, k = 0$. For each k in $[0, n)$, if $l[i] \leq r[j]$, set $a[k] \leftarrow l[i]$ and increment i. Otherwise, set $a[k] \leftarrow r[j]$ and increment j. Finally, return the array a.

Proof of Correctness. Let us show that the returned array will always be sorted for all n. In the base case, if $n \leq 1$, then the list is already sorted. Inductively, take a list of length n. By induction, the sorted sublists l and r will be correctly sorted. For any indexes i and j, if $l[i] \leq r[j]$, then $l[i'] \leq r[j']$ for all $i' \leq i$ and $j' \geq j$. Therefore, merging the two lists in the described way will produce a total combined list in sorted order.

Complexity. Analyzing the recurrence relation, we make 2 calls to sort lists of half the size. The merge step requires iterating over both lists, which involves at most $n-1$ comparisons. Thus, $T(n) = 2 \cdot T(\frac{n}{2}) + (n-1)$. Using the *Master Theorem* defined below, it follows that $T(n) \in \Theta(n \log n)$.

8.2 Generic Algorithm Design

A *Divide and Conquer* algorithmic strategy applies when we can divide a problem into parts, solve each part, and then combine the results to yield a solution to the overall problem. The generic design is as follows:

Algorithm 8.2 (Divide and Conquer)**.**

1. For positive integer $b > 1$, divide the problem into b parts.
2. (Divide) Recursively solve the subproblems.
3. (Conquer) Consider any situations that transcend subproblems.

8.3 Complexity

By the construction, the time complexity of the algorithm $T(n)$ satisfies the recurrence relation:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \tag{8.1}$$

where $f(n)$ is the time it takes to compute step 3 (above). In class, we looked at the following theorem about runtime:

Theorem 8.3. *(Master Theorem) If $T(n)$ satisfies the above recurrence relation, then*

- *if $\exists\ c > 1, n_0$ such that for $n > n_0$, $af(n/b) \geq cf(n)$ then $T(n) \in \Theta(n^{\log_b a})$*
- *if $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} \log n)$*
- *if $\exists\ c < 1, n_0$ such that for $n > n_0$, $af(n/b) \leq cf(n)$ then $T(n) \in \Theta(f(n))$*

Proof. Convince yourself that $a^{\log_b n} = n^{\log_b a}$. By induction, its easy to see that

$$T(n) = a^j \cdot T\left(\frac{n}{b^j}\right) + \sum_{k=0}^{j} a^k f\left(\frac{n}{b^k}\right) \tag{8.2}$$

Apply to $j = \log_b n$ and recognize $T(1)$ is a constant so

$$T(n) = \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k f\left(\frac{n}{b^k}\right) \tag{8.3}$$

Let's consider the first case. We apply the relation to get[44]

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + a^{\log_b n} f(1) \sum_{k=0}^{\log_b n} c^{-k} \\ &= \Theta(a^{\log_b n}) + \Theta\left(a^{\log_b n} \frac{c}{c-1}\right) \\ &= \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \end{aligned} \tag{8.4}$$

For the second case,

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k \Theta\left(\left(\frac{n}{b^k}\right)^{\log_b a}\right) \\ &= \Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n} a^k \left(\frac{\Theta(n^{\log_b a})}{a^k}\right) \\ &= \Theta(n^{\log_b a} \log n) \end{aligned} \tag{8.5}$$

For the third case, recognize that it's nearly identical to the first except the summations are bounded in the other direction which leaves $f(n)$ as the dominating term. □

8.4 Quicksort

In most cases, the preferred sorting algorithm of choice is *Quicksort*. This algorithm appears very similar to Mergesort, but it partitions the inputs not by location, but by value. The space complexity is also very efficient.

In the quicksort algorithm, we choose a pivot value p. We then sort the array in a cursory way, such that all entries with value $\leq p$ are to the left of all entries with value $> p$. Recursively, then, we know that the final sorted list will be the sorted "less than" list, followed by the sorted "greater than" list, and so we recursively sort both parts.

An algorithm to sort an array X of length n in place around a pivot p is:

- Initialize $i \to 0$ and $j \to n-1$

[44] I've made the simplifaction here that $n_0 = 0$. As an excercise, convince yourself this doesn't effect anything, just makes the algebra a little more complicated.

- Increment $i++$ while $X[i] < p$ and decrement $j--$ while $X[j] > p$. If $i \geq j$, then return j, the index of the "boundary". Otherwise, swap $X[i]$ and $X[j]$, and repeat this loop.

In many naive implementations, the pivot value is taken to be an arbitrary value of the array (e.g. the first, last, or a random index). Clearly, this algorithm will perform best when the pivot divides the array roughly in half, and so we want the pivot to be as close to the median as possible.

8.5 Rank Selection

The *rank* of any element is the number of elements in the list with lesser or equal value. For example, the median will have rank $\frac{n}{2}$. With this definition, we give the following algorithm to find the element of a given rank in a list.

Algorithm 8.4. Rank Selection: Select(r, L) Write out the array L into 5 rows, and consider each column separately. We can trivially sort each column in constant time, so after an $\Theta(n)$ operation we can have a list of columns, sorted, such that the middle row has the medians of each column. Recursively sort these columns by their median value, yielding a final "median of medians" z.

Compare all of L to this value z, compute rank(z) and generate two lists: $L_1 := \{\ell \in L \mid \ell < z\}$ and $L_2 := \{\ell \in L \mid \ell > z\}$. If rank$(z) < r$, recursively call Select(r, L_1). Otherwise, call Select$(r - \text{rank}(z), L_2)$.

Clearly, applying this algorithm with an initial input $r = \frac{\text{len}(L)}{2}$ will yield the median. To explore the efficiency of this algorithm, we first give the following lemma.

Lemma 8.5 (Rank of z)**.** *The rank of the element z selected by the above algorithm is bounded by $\frac{3n}{10} \leq \text{rank}(z) \leq \frac{7n}{10}$, where n is the length of L.*

To show this, consider the diagram.

small	small	small	small	small	*	*	*	*
small	small	small	small	small	*	*	*	*
small	small	small	small	z	large	large	large	large
*	*	*	*	large	large	large	large	large
*	*	*	*	large	large	large	large	large

By the construction of the algorithm, each median value will be greater than the rows above it in that column, and to the median values of other columns to the left. Therefore, z must be greater than all the entries labeled "small" in the diagram. A similar argument goes for the larger case.

There are $\frac{3n}{10}$ such "smaller" values and "larger" values each, which gives the bound on the rank of z.

Runtime Let $T_{\text{SEL}}(n)$ be the runtime of this algorithm on a list of length n. In each iteration, we must sort each column (in $\Theta(n)$ time), and then recursively sort the median values (a subproblem of size $\frac{n}{5}$). Given the above lemma, we can bound the size of L_1 and L_2 by $\frac{7n}{10}$. Thus, an equation describing the runtime is

$$T_{\text{SEL}}(n) \leq T_{\text{SEL}}\left(\frac{n}{5}\right) + T_{\text{SEL}}\left(\frac{7n}{10}\right) + \Theta(n)$$

This is a slightly different form than the Master Theorem as given. However, it is a straightforward exercise to note that if $T(n) \leq \Theta(n) + \sum T(c_i n)$, where $\sum c_i < 1$, then indeed $T(n) \in \Theta(n)$. As this applies for the above case, we get that the complexity of this algorithm is linear in the length of the list.

Application to Quicksort Sorting the list about the pivot is already a $\Theta(n)$ computation, so adding this step beforehand to compute the median does not affect the asymptotic runtime. However, as you can probably imagine, the constants for this algorithm are rather large, so doing this at each step is by no means necessarily "quick".

8.6 Randomized Quicksort

Consider a different algorithmic approach, in which we select an item from the list to serve as the pivot uniformly at random.

The course slides give one way of showing that the expected number of comparisons performed by this algorithm is $T_Q(n) = 2(n+1)H_n - 4n$, where $H_n = \sum_{\ell=1}^{n} 1/\ell$. (We understand this function well: H_n converges to $\gamma + \log n$ where log is natural logarithm, and $\gamma \cong 0.577$ is the Euler-Mascheroni constant.) So, roughly $(2/\lg e)n \lg n$ (where lg is logarithm base 2).

Here is another way of doing the analysis. Think of the elements laid out in their sorted order $1, \ldots, n$. Think of the process of picking pivots as follows. We pick a random element i to be the first pivot—so let's give it the label $\ell(i) = 1$. Now all of $1, \ldots, i-1$ are going to be quicksorted separately, and all of $i+1, \ldots, n$ are going to be quicksorted separately. So in each of these regions we are going to uniformly pick a pivot; give each of these a label of 2. And so forth, in rounds, in each existing nonempty interval we pick a new element and label it with the number of the round.

Now, think of each comparison from the point of view of the non-pivot element of the pair. That is, from the point of view of the element which will receive the higher label. We can count all comparisons by "charging" them to the higher-label element. So think of some element j, $1 \leq j \leq n$. To how many lower-label pivots will it be compared?

An easy way to upper bound this is to use the case in which j itself has the highest possible label. And moreover, we'll bound the comparisons by the number of those against pivots $k < j$, plus the number against pivots $k > j$. Let's just write down the second case. What is the probability that a specific k occurs as a pivot of level ℓ that is compared to i? This is precisely the probability that when k is chosen to be a pivot, none of $i+1, \ldots, k-1$ have yet been chosen to be pivots. But since $i+1, \ldots, k$ were equally likely to be chosen at this juncture, this probability is at most $1/(k-i)$. So, the expected number of comparisons of j against k's in the range $k > j$, is at most $\sum_{k=i+1}^{n} \frac{1}{k-i} = H_{n-k}$. Now throwing in also $k < j$, and then summing over j, we have the upper bound $2\sum_{1 \leq j < k \leq n} \frac{1}{k-j} = 2\sum_{m=1}^{n-1} \frac{n-m}{m} \leq 2n\sum_{1}^{n-1} \frac{1}{m} \leq 2nH_n$.

8.7 Lower bound on Sorting

We'll describe here a lower bound on deterministic comparison-based sorting algorithms. (An extension of this argument works even for randomized comparison-based sorting algorithms.)

In a comparison-based sorting algorithm, the inputs are some $x_1, \ldots, x_n$, and all we learn about them we learn from comparisons "is $x_i < x_j$?".
When the inputs are restricted to be from a small domain, say $x_i \in \{0, 1\}$, sorting actually gets easier. So for the purpose of a lower bound, we rely on the fact that they come from a large enough domain that all might be distinct; since our algorithm must handle that case, we can just simplify our exposition by assuming that all are indeed distinct; and in fact we may as well assume they are just the numbers $1, \ldots, n$ in some order. That means we can think of the input x simply as a permutation of the numbers $1, \ldots, n$.

Now, our algorithm must put the inputs in correct order no matter what permutation they arrived in. That means that for any two distinct permutations x, x', at some point our algorithm must "tell them apart"—that is, there is some first comparison at which we ask "is $x_i < x_j$?" and get a different answer than we get for "is $x'_i < x'_j$?". (Since the algorithm can be adaptive, the queries from that point and on might be different in the two algorithms.)

But this means that if you look at the binary sequence of answers the algorithm gets to its comparison queries, that binary sequence is different for every permutation x. The number of permutations of n elements is $n! \cong e^{n \log n - n} = 2^{n \lg n - n/\log 2}$, so in order to distinguish them all, the number of comparisons on a worst-case input (and in fact on most inputs) has to be at least $n \lg n - n/\log 2$.

This $\Omega(n \log n)$ lower bound for sorting is within a constant factor of the upper bounds provided by the algorithms we have seen.

8.8 Fast Integer Multiplication

Consider two n digit integers X and Y. Let $X = X_L X_R$ as digits, where X_L is the left half of X, and X_R is the right half, and similarly $Y = Y_L Y_R$. Numerically, $X = 2^{\frac{n}{2}} \cdot X_L + X_R$,

and similarly for Y. The naive grade-school algorithm for multiplication would be to use

$$X \cdot Y = 2^n \cdot X_L \cdot Y_L + 2^{\frac{n}{2}} \cdot (X_L \cdot Y_R + X_R \cdot Y_L) + X_R \cdot Y_R. \tag{8.6}$$

Multiplying by an exponent of 2 can be easily performed by a bit shift. However, this approach uses 4 multiplications of $\frac{n}{2}$ digit numbers. Using the Master Theorem, the runtime would thus be $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$, and so $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$. This naive divide and conquer application does not yield a better runtime.

An improved algorithm is *Karatsuba Multiplication*. In this approach, we set $A = X_L \cdot Y_L$, $B = X_R \cdot Y_R$, and $C = (X_L + X_R) \cdot (Y_L + Y_R)$. Note that the middle term of (8.6) is given by $X_L \cdot Y_R + X_R \cdot Y_L = C - A - B$. Therefore, with only three multiplications, we rewrite this equation as

$$X \cdot Y = 2^n \cdot A + 2^{\frac{n}{2}} \cdot (C - A - B) + B. \tag{8.7}$$

This algorithm makes only 3 recursive calls, so $T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$, so that $T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$. This demonstrates the power of the divide and conquer approach.

8.8.1 Convolution

Given two arrays s and t of size n and m, respectively, the *convolution* $s \star t$ of the two arrays is an array of length $n + m$, with the terms given by

$$(s \star t)_k = \sum_{i+j=k} s_i \cdot t_j \tag{8.8}$$

If you think of the arrays as coefficients of polynomials (i.e. $s(x) = \sum_{i=0}^{n} s_i x^i$), then convolution is simply polynomial multiplication.

Observe that the binary representation of a number a can be thought of the coefficients $s_i \in \{0, 1\}$ that satisfy $s(2) = a$. The Karatsuba algorithm for multiplication can be generalized to convolution, yielding an $\Theta(n^{\log_2 3})$ algorithm.

8.9 Fast Division, Newton's Method

For a given $0 \neq t \in \mathbb{R}$, we must compute the inverse $\frac{1}{t}$. Noting that this inverse is the root of $f(x) = t - \frac{1}{x}$, we can use the Newton Raphson method of successive approximation to

find this root. Concretely,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} = x_i - x_i^2 t + x_i = 2x_i - tx_i^2.$$

Algorithm 8.6 (Fast Division)**.** Let p be the unique exponent for which $2^p \in [t, 2t)$. Initialize a seed $x_0 = 2^{-p}$. Iteratively, compute $x_{i+1} = 2x_i - tx_i^2$ until a sufficient accuracy is reached.

Proof of Correctness. By definition, $\frac{1}{2t} < x_0 \leq \frac{1}{t}$. We can define the associated error of this approximation as $\epsilon_0 = 1 - tx_0 < \frac{1}{2}$.

Iteratively, $\epsilon_{i+1} = 1 - tx_{i+1} = 1 - t(2x_i + tx_i^2) = (1 - tx_i)^2 = \epsilon_i^2$, and so $\epsilon_i \leq 2^{-2^i}$. In other words, each iteration doubles the number of significant digits in the approximation.

Thus, after sufficient iterations, any arbitrary accuracy can be achieved.

8.10 Fast Fourier Transform

We saw above in Section 8.8.1 an $\Theta(n^{\log_2 3})$ time algorithm for polynomial multiplication. Let us now consider a better approach for multiplying polynomials in $\mathbb{C}_n[x]$ (polynomials of degree at most $n-1$ with complex coefficients).

8.10.1 A Change of Basis

$\mathbb{C}_n[x]$ can be thought of as an n dimensional vector space over $\mathbb{C}$, with the vector components simply the coefficients of the polynomial. For example, for $n = 3$, we would map

$$(4 - 2i) + ix + 2x^2 \mapsto \begin{pmatrix} 4 - 2i \\ i \\ 2 \end{pmatrix}. \tag{8.9}$$

Simple checking will verify that this definition observes the definitions of a vector space (i.e. group addition and scalar multiplication by $\mathbb{C}$). This mapping defines the basis $\mathcal{B} = \{1, x, \ldots, x^{n-1}\}$. This basis makes polynomial addition trivially $\Theta(n)$, but requires

convolution to perform multiplication. However, just like any vector space, we can choose an alternative basis for $\mathbb{C}_n[x]$.

Theorem 8.7. *(Evaluation Map) Pick any n distinct points $\{\zeta_1, \ldots, \zeta_n\} \in \mathbb{C}$. Then the map*

$$\mathbb{C}_n[x] \to \mathbb{C}^n \qquad t(x) \mapsto (t(\zeta_1), \ldots, t(\zeta_n)) \tag{8.10}$$

is a linear isomorphism (i.e. injective and surjective).

Proof. To see that this map is linear, note that for any polynomial $t(x) \in \mathbb{C}_n[x]$ and scalar $\lambda \in \mathbb{C}$, $\lambda \cdot t(x) \mapsto (\lambda t(\zeta_1), \ldots, \lambda t(\zeta_n)) = \lambda \cdot (t(\zeta_1), \ldots, t(\zeta_n))$.

To see that this map is surjective, pick some set $\{y_i\}_{i=1}^n$ of desired values that a polynomial should take at $\{\zeta_i\}$ respectively.[45] Using *Lagrange interpolation*, define a list of n special polynomials

$$p_i(x) = \prod_{k \neq i} \frac{x - \zeta_k}{\zeta_i - \zeta_k}. \tag{8.11}$$

Note that for any $j \neq i$, then $p_i(\zeta_j) = 0$, since there will be a $\zeta_j - \zeta_j$ term in the numerator. On the other hand, $p_i(\zeta_i) = 1$, since each of the terms becomes 1. Therefore, restricted to the $\{\zeta_i\}$, each p_i is a "delta function". Since there are n points, and thus $n-1$ points where $k \neq i$, the degree of these polynomials is $n-1$, and so $p_i \in \mathbb{C}_n[x]$ (as we would hope).

Given these polynomials, and a set of desired values $\{y_i\}$, define

$$p(x) = \sum_{i=1}^{n} y_i \cdot p_i(x). \tag{8.12}$$

Then $p(x) \in \mathbb{C}_n[x]$ and $p(\zeta_i) = y_i$ for all i.

To show that this map is an isomorphism (i.e. 1-1), we consider the change of basis matrix

[45] What about the value at other points $x \notin \{\zeta_i\}$? The vector of the $\{y_i\}$ is the vector over the new basis that we are trying to create. With this basis, this is the most amount of detail we can contain.

that maps the vector of the coefficients to the new vector of the evaluations at $\{\zeta_i\}$.

$$\begin{pmatrix} t(\zeta_1) \\ t(\zeta_2) \\ \vdots \\ t(\zeta_n) \end{pmatrix} = \begin{pmatrix} 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{n-1} \\ 1 & \zeta_2 & \zeta_2^2 & \cdots & \zeta_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_n & \zeta_n^2 & \cdots & \zeta_n^{n-1} \end{pmatrix} \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{n-1} \end{pmatrix} \tag{8.13}$$

The square matrix used in this change of basis is the *Vandermonde matrix* generated by $\{\zeta_i\}$. This matrix is nonsingular (the determinant is $\prod_{0 \le i < j \le n-1}(\zeta_j - \zeta_i) \neq 0$ since all the points are distinct), and thus invertible. Therefore, this map is bijective.[46] □

Because this map is an isomorphism, we can take polynomials in the conventional representation, convert them to the *evaluation basis*, perform operations on them, and then convert them back.

8.10.2 Better Multiplication

In the original polynomial basis, addition was easy, but multiplication required convolution. However, in this new basis representation, $f(x) \cdot g(x)$ becomes $(f(\zeta_1) \cdot g(\zeta_1), \ldots, f(\zeta_n) \cdot g(\zeta_n))$, and so we can simply multiply each component. Since the degree of the product is at most $2n-2$, the product will be contained in $f(x) \cdot g(x) \in \mathbb{C}_{2n-1}[x]$, and so we should use $2n-1$ evaluation points.

This insight gives us an alternative approach to perform polynomial multiplication.

Algorithm 8.8 (Convolution via point-wise multiplication)**.** Let $s(x)$ and $t(x)$ be two polynomials in $\mathbb{C}_n[x]$.

1. Choose $2n-1$ points $\{\zeta_i\} \subset \mathbb{C}$.
2. Convert s and t to $(s(\zeta_1), \ldots, s(\zeta_{2n-1}))$ and $(t(\zeta_1), \ldots, t(\zeta_{2n-1}))$.
3. Multiply each component, yielding a vector $a = (s(\zeta_1)t(\zeta_1), \ldots, s(\zeta_{2n-1})t(\zeta_{2n-1}))$.
4. Invert the evaluation map to yield a polynomial $a(x) = s(x) \cdot t(x)$.

[46] This proof also shows surjectivity by simply comparing the dimensions of the vector spaces. However, it is good to see how to explicitly construct a preimage for any set of points $\{y_i\}$.

This algorithm looks great (no convolution!), but let us consider the complexity. The evaluation map and its inverse are matrix-vector multiplications. Simply writing down the full Vandermonde matrix is $\Theta(n^2)$, which is already worse than the $\Theta(n^{\log_2 3})$ runtime we got using the Karatsuba approach.

8.10.3 Fourier Transform

There are two key aspects of Algorithm 8.8 that allow us to improve the performance. The entire matrix is uniquely determined by the $2n-1$ points $\{\zeta_i\}$. Furthermore, we can choose these points in any way we like, so long as they are all distinct.

Let m be the smallest power of 2 greater than $2n-2$. Then clearly $\mathbb{C}_{2n-1}[x] \subset \mathbb{C}_m[x]$, so let us treat the problem of two polynomials of degree less than m. Let $\omega = e^{2\pi i/m}$ be a primitive m^{th} root of unity.[47] Define $\zeta_j = \omega^{j-1}$.

Given this choice, the Vandermonde matrix becomes

$$F_m(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{m-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(m-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{m-1} & \omega^{2(m-1)} & \omega^{2(m-1)} & \cdots & \omega^{(m-1)^2} \end{pmatrix}, \tag{8.14}$$

called the Fourier transform over $\mathbb{Z}/m$.[48]

Given the specific structure of this matrix, we will show a way to multiply a vector by this matrix, or its inverse, in $\Theta(m \log m)$ time.

8.10.4 FFT Divide and Conquer Approach

To divide the problem of multiplication by $F_m(\omega)$ into smaller subproblems, we'll split the columns of the matrix into the even and odd columns ($F_{m\ \text{EVEN}}(\omega)$ and $F_{m\ \text{ODD}}(\omega)$). To

[47] A primitive m^{th} root of unity ω is a number $\omega \in \mathbb{C}$ such that $\omega^m = 1$, but $\omega^i \neq 1$ for all $1 \leq i < m$.

[48] $\mathbb{Z}/m$ is the addative group of intergers modulo m. For example, the classic "clock arithmetic" where $12 \equiv 0$ and so $9+9 = 18 \equiv 6$ is $\mathbb{Z}/12$. This structure arrises from the multiplicative group of the primitive m^{th} root of unity because $\omega^m = 1 = \omega^0$, and so for any power i, $\omega^i = \omega^{i \mod m}$.

see how this approach works more clearly, we will look at the case of $m = 8$.

First consider the even columns. Since $\omega^8 = 1$, the even columns simplify down to

$$F_{8\ \mathrm{EVEN}}(\omega) = \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \\ \hline 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{array}\right) = \left(\begin{array}{c} F_4(\omega^2) \\ \hline F_4(\omega^2) \end{array}\right) = \begin{pmatrix} I_4 \\ I_4 \end{pmatrix} F_4(\omega^2) \tag{8.15}$$

where I_4 is the 4×4 identity matrix. Similarly, the odd columns are

$$F_{8\ \mathrm{ODD}}(\omega) = \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ \omega & \omega^3 & \omega^5 & \omega^7 \\ \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ \omega^5 & \omega^7 & \omega & \omega^3 \\ \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ \omega^7 & \omega^5 & \omega^3 & \omega \end{array}\right). \tag{8.16}$$

Observe that the i^{th} row of this matrix is ω^{i-1} times the corresponding i^{th} row of the even

column matrix. So we can express this matrix as

$$F_{8\,\mathrm{ODD}}(\omega) = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \\ \hline \omega^4 & 0 & 0 & 0 \\ 0 & \omega^5 & 0 & 0 \\ 0 & 0 & \omega^6 & 0 \\ 0 & 0 & 0 & \omega^7 \end{array}\right) \cdot F_4(\omega^2). \tag{8.17}$$

Using these facts, to multiply by $F_8(\omega)$, we can multiply by

$$\underbrace{\begin{pmatrix} 1 & & & & & & & \\ & & & 1 & & & & \\ & 1 & & & & & & \\ & & & & 1 & & & \\ & & 1 & & & & & \\ & & & & & & 1 & \\ & & & 1 & & & & \\ & & & & & & & 1 \end{pmatrix}}_{C^{-1}} \times \underbrace{\left(\begin{array}{cccc|cccc} 1 & & & & 1 & & & \\ & 1 & & & & \omega & & \\ & & 1 & & & & \omega^2 & \\ & & & 1 & & & & \omega^3 \\ \hline 1 & & & & \omega^4 & & & \\ & 1 & & & & \omega^5 & & \\ & & 1 & & & & \omega^6 & \\ & & & 1 & & & & \omega^7 \end{array}\right)}_{A} \times$$

$$\underbrace{\left(\begin{array}{cc|cc} F_4(\omega^2) & & & \\ & & & \\ \hline & & F_4(\omega^2) & \\ & & & \end{array}\right)}_{B} \times \underbrace{\begin{pmatrix} 1 & & & & & & & \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & & & & & & & 1 \\ & 1 & & & & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}}_{C} \tag{8.18}$$

The first matrix C shuffles around the rows of the column vector into the even and odd indices. We then recursively do two matrix-vector multiplications of size $\frac{m}{2}$, and use the result to get the even and odd rows. Applying C^{-1} restores the rows to the final correct order.[49]

Consider the time complexity of these operations. The first step, reordering the rows, is

[49] This last step isn't strictly necessary. Ommitting it simply uses a reordering of the evaluation basis.

$\Theta(m)$. We then apply two subproblems (B) of half the size, $2 \cdot T(\frac{m}{2})$. Multiplication by a diagonal matrix (A) can be done in $\Theta(m)$, as can the row reshuffling C^{-1}. The overall recursive relation is thus

$$T(m) = 2 \cdot T\left(\frac{m}{2}\right) + \Theta(m) \tag{8.19}$$

Applying the Master Theorem (8.3), we see that the time complexity of this algorithm is $\Theta(m \log m)$.

The last step of algorithm 8.8 requires multiplying by the inverse of $F_m(\omega)$.

Claim 8.9. $(F_m(\omega))^{-1} = \frac{1}{m} F_m(\omega^{-1})$.

Proof. To show that two matrices are inverses, it suffices to show that their product is the identity matrix, and so the terms of the product matrix are the Kronecker delta function $m_{i,j} = \delta(i-j)$.[50]

Solving for the terms of the product matrix,

$$\left(F_m(\omega) \cdot \frac{1}{m} F_m(\omega^{-1})\right)_{i,j} = \frac{1}{m} \sum_{\ell=0}^{m-1} \omega^{j\ell} \omega^{-i\ell} = \frac{1}{m} \sum_{\ell=0}^{m-1} (\omega^{j-i})^{\ell}.$$

As ω is an m^{th} root of unity, this sum will be 0 so long as $j \neq i$. If $j = i$, then this sum is simply one, and so

$$\frac{1}{m} \sum_{\ell=0}^{m-1} (\omega^{j-i})^{\ell} = \delta(j-i).$$

□

Given this, we can give an improved way to perform polynomial convolution.

Algorithm 8.10 (FFT)**.** Take two polynomials $t(x), s(x) \in \mathbb{C}_n[x]$. Let m and ω be defined as above.

1. Convert $t(x), s(x)$ to $(t(\omega^1), \ldots, t(\omega^{2m-1}))$ and $(s(\omega^1), \ldots, s(\omega^{2m-1}))$ with FFT. Time: $\Theta(n \log n)$

[50] The Kronecker delta function is defined by $\delta(x) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ else} \end{cases}$

2. Multiply the corresponding values, yielding $(t(\omega^1)s(\omega^1), \ldots, t(\omega^{2m-1})s(\omega^{2m-1}))$. Time: $\Theta(n)$.

3. Perform the inverse evaluation map with the FFT. Time: $\Theta(n \log n)$.

This algorithm yields a $\Theta(n \log n)$ algorithm for convolution (or equivalently polynomial multiplication).

9 Streaming Algorithms

We can imagine a situtation in which a stream of data is being recieved but there is too much data coming in to store all of it. However, we want to extract some information out of the stream of data without storing all of it. An example could be a company like Facebook or Google storing click data or login data. What they really care about is aggregate data and not individual datum.

The following are all examples of streaming problems that we might be interested in. Some you will find have very simple algorithms and some are rather complicated. As an aside, it should be noted that the majority of work in streaming algorithms requires using randomized algorithms (which you can learn all about in CS/CMS 139). Typically, this is done by selecting a small sample of the data and estimating the aggregate data from the small sample.

- Calculating the length of the stream.
- Calculating the mean or median of a set of values.
- Calculating the standard deviation of a set of values.
- Calculating the number of unique elements (assuming the stream is elements from $\{1, \ldots, n\}$).[51]
- Calculating the k most frequent elements.[52]
- Sampling elements according to the distribution of the stream.

9.1 Formalism

Definition 9.1 (Stream)**.** A stream σ is a sequence of elements $(s_1, \ldots, s_m)$ where each $s_i \in \{1, \ldots, n\}$. Typically, the length of the stream is unknown and n is large but $n << m$.

Definition 9.2 (Streaming Algorithm)**.** A streaming algorithm consists of 3 parts. An initialization, a processing step of each s_j, and an output step.

[51] Also known as the 0th moment
[52] This is the heavy-hitters problem.

An ideal streaming algorithm uses $o(m)$ memory and can process each element s_j in little time ($O(1), O(\log m), O(\log n)$, etc.). Why? If the stream uses $O(m)$ memory then it is no better than storing all the elements and computing the ideal value. We want to improve on this. The storage of any stream element s_j is $O(\log n)$ so manipulating it should take approximately that much time.

9.2 Uniform Sampling

Exercise 9.3 (Uniform Sampling)**.** Given a stream σ, for each $i \in \{1, \ldots, n\}$, define $f_i = \big|\{j : s_j = i\}\big|$, the number of occurances of value i in the stream. We can define a probability distribution P on $\{1, \ldots, n\}$ by $p_i = f_i/m$.[53] The goal is to output a single element sampled in accordance to the distribution P.

Example 9.4. For example, consider the stream $\sigma = (1, 3, 4, 5, 5, 2, 1, 1, 1, 7)$. In this case $p_1 = 4/10, p_2 = 1/10$, $p_3 = 1/10, p_4 = 1/10, p_5 = 2/10, p_7 = 1/10$. The goal would be to output a random variable in accordance to this distribution.

As we alluded to earlier, the ability to sample an element from the stream uniformly is rather helpful in many more complicated streaming algorithms. Turns out uniformly sampling an element from a stream is a harder problem than people anticipate. Let's consider a naïve algorithm.

Algorithm 9.5 (Naïve Sampling Algorithm)**.**

- Initialize: For each value $i = 1, \ldots, n$, initialize a counter $f_i \leftarrow 0$. Initialize $m \leftarrow 0$.
- Process s_j: If $s_j = i$, set $f_i \leftarrow f_i + 1$. Increment $m \leftarrow m + 1$.
- Output: Calculate $p_i = f_i/m$. Choose an i according to $P = (p_1, \ldots, p_n)$.

Its not too difficult to see what this algorithm is doing and hence its correctness. It is keeping a count of the number of each element. This is certainly an improvement over storing all the elements in the stream which takes $O(m \log n)$ space to store the entire stream. In this case we store n counters which range between 0 and m so require $O(n \log m)$ space in total. As $n << m$, this is better but not the best we can do.

[53] It is a small exercise to verify that $(p_1, \ldots, p_n)$ defines a probability distribution.

We are going to improve to using only $O(\log n)$ space. This is necessarily optimal as the output is of size $O(\log n)$.

Algorithm 9.6 (Uniform Sampling Algorithm)**.**

- Initialize: Set $x \leftarrow$ null.
- Process s_j: With probability $1/j$, set $x \leftarrow s_j$. Otherwise, do nothing.
- Output: x.

Definitely a simpler algorithm! Let's see why it works. We are going to consider a substream $\sigma_j = (s_1, \ldots, s_j)$, the first j elements of the stream σ.

Theorem 9.7. *Let X_j be the random variable output of Algorithm 9.6 on stream $\sigma_j = (s_1, \ldots, s_j)$. Then $\mathbf{Pr}(X_j = i) = f_i^{(j)}/j$ for each $i = 1, \ldots, n$, where $f_i^{(j)}$ is the number of instances of i in the first j elements.*

Proof. We proceed by induction on j. If $j = 1$, then the algorithm necessarily outputs s_1, and therefore appropriately samples from $\sigma_1 = (s_1)$. For the inductive step, assume correctness up to j. An equivalent way to formulate the algorithm given the inductive step is X_{j+1} remains X_j with probability $j/(j+1)$ and switches with probability $1/(j+1)$.

Notice that if $s_{j+1} = i$, then $f_i^{(j)} = f_i^{(j+1)}$ and for $i' \neq i$, $f_{i'}^{(j)} = f_{i'}^{(j+1)}$. Then, we can calculate

$$\begin{aligned}
\mathbf{Pr}(X_{j+1} = i) &= \mathbf{Pr}(X_{j+1} \text{ switches to } i) + \mathbf{Pr}(X_{j+1} \text{ remains } X_j)\mathbf{Pr}(X_j = i) \\
&= \frac{1}{j+1} + \frac{j}{j+1} \cdot \frac{f_i^{(j)}}{j} \\
&= \frac{f_i^{(j)} + 1}{j+1} \\
&= \frac{f_i^{(j+1)}}{j+1}
\end{aligned} \tag{9.1}$$

For $i' \neq i$, then

$$\begin{aligned}\mathbf{Pr}(X_{j+1} = i') &= \mathbf{Pr}(X_{j+1} \text{ remains } X_j)\mathbf{Pr}(X_j = i') \\ &= \frac{j}{j+1} \cdot \frac{f_{i'}^{(j)}}{j} \\ &= \frac{f_{i'}^{(j)}}{j+1} \\ &= \frac{f_{i'}^{(j+1)}}{j+1}\end{aligned} \tag{9.2}$$

thereby proving the inductive step. □

As a consequence of the theorem, it is easy to see the correctness of the algorithm.

The incredible advantage of this algorithm is that the output has no m dependence.[54]

[54] Aside from the ability to sample at probability $1/m$, but let's ignore that for now because this is a harder problem for a different day.

10 Max-Flow Min-Cut

10.1 Flows and Capacitated Graphs

Previously, we considered weighted graphs. In this setting, it was natural to thinking about minimizing the weight of a given path. In fact, we considered algorithms that calculate the minimum weight paths on graphs and we optimized for a variety of parameters (considering all-paths, negative weight edges, etc.). We now are interested in a related problem where our edges have a max capacity and we want to send as much 'flow' from one location to another.

Flow problems are a vary natural set of problems to consider for any graph. Consider the graph of the internet for example; each edge (a connection between two computers, servers, etc.) has a max bandwidth that it allows. We could be interested in calculating what the bottleneck step in sending packets from computer a to b is. For this, we would most likely frame the problem as a flow problem.

Definition 10.1 (Capacitated Network)**.** A capacitated network is $G = (V, C, s, t)$ where V is a finite set of vertices, $s, t \in V$ (colloquially referred to as the source and sink, respectively) and $C : V^2 \to \mathbb{R}^+$, a capacity function defined on all the edges.[55]

Definition 10.2 (s-t flow)**.** A s-t flow is a function $f : V^2 \to \mathbb{R}$ satisfying three conditions:

1. Skew symmetry: for all $u, v \in V$, $f(u, v) = -f(v, u)$
2. Flow conservation: for all interior vertices $u \in V - \{s, t\}$, $\sum_v f(u, v) = 0$
3. Capacity constraints, for all $u, v \in V$, $f(u, v) \leq C(u, v)$.

In layman's terms, the first condition gives the flow a direction per edge. The second condition forces that the only location that flow can be generated from and removed from are the defined source and sink vertices. The third condition forces that none of the edges are over capacity.

The natural question that we want to ask is how much flow is being sent across this network? To answer, this we need to define what a cut is and what the flow across a cut is.

[55] Here, we wrote C as a function on all vertex pairs. If you wish to think of this in the traditional graph sense then you could choose $E = \{e \in C : C(e) > 0\}$.

Definition 10.3 (s-t Cut)**.** A s-t cut is a set $S \subseteq V$ such that $s \in S$ and $t \in S^c = V - S$.

Definition 10.4 (s-t Cut Capacity and Flow)**.** The capacity of a cut S is

$$C(S, S^C) = \sum_{u \in S, v \in S^c} C(u, v). \tag{10.1}$$

And the flow across the cut is

$$f(S, S^C) = \sum_{u \in S, v \in S^c} f(u, v). \tag{10.2}$$

Definition 10.5 (Value of a flow f)**.** The value of a flow f is $\text{Val}(f)$ is

$$\text{Val}(f) = f(\{s\}, \{s\}^c) = \sum_{v} f(s, v) \tag{10.3}$$

By induction on the set S, one can prove that $\text{Val}(f) = f(S, S^c)$ for any s-t cut S.

10.2 Theorem

Corollary 10.6. *For any flow f and s-t cut S,* $\text{Val}(f) \leq C(S, S^c)$.

Proof. (Sketch.) Apply capacity constraints to the alternate definition of $\text{Val}(f)$ for any cut S. □

We now reach the most interesting statement about flows and cuts; (the previous corollary was the weak duality version). This is the strong duality.

Theorem 10.7 (Max-Flow Min-Cut)**.** *For any capacitated network $G = (V, C, s, t)$,*

$$\max_{\text{flow } f} \text{Val}(f) = \min_{\text{cut } S} C(S, S^c) \tag{10.4}$$

To prove Theorem 10.7, we will need a few additional definitions.

Definition 10.8. For a s-t flow f on a capacitated network, we can define the residual capacity function $C_f : V^2 \to \mathbb{R}^+$ as

$$C_f(u, v) = C(u, v) - f(u, v) \tag{10.5}$$

The residual capacity function is always positive due to capacity constraints on f. It can be thought of as the maximum additional flow that can be pushed across (u, v). We say that f *saturates* (u, v) if $C_f(u, v) = 0$. The residual network is defined as (V, C_f, s, t).

Lemma 10.9. *The following statements hold about flows:*

1. *f_2 is a flow in G_{f_1} iff $f_1 + f_2$ is a flow in G.*
2. *$\text{Val}(f_1 + f_2) = \text{Val}(f_1) + \text{Val}(f_2)$.*
3. *F_2 is a max flow in G_{f_1} iff $f_1 + f_2$ is a max flow in G.*
4. *If $f^\star$ is a max flow in G and f is any flow in G, then the value of a max flow in G_f is $\text{Val}(f^\star) - \text{Val}(f)$.*

Proof. For (1), by capacity constraints on $G_f(1)$, $f_2(e) \leq C(e) - f_1(e)$. Therefore, $f_1(e) + f_2(e) \leq C(e)$. The other direction follws identically. For (2), notice that $\text{Val}(\cdot)$ is a sum of linear functions. (3) follows from (1) and (2) and (4) follows from (2) and (3). □

The power of this lemma is that flow can be added incrementally until the maximum flow is achieved. This gives us the intuition for *augmenting paths*.

Definition 10.10 (Augmenting Path)**.** Given G, f, an augmenting path for f in G is a path $s \to t$ consisting of edges in the support of G_f (edges with positive capacity in G_f).

Definition 10.11 (Bottleneck Capacity)**.** Given G, f, we define the bottleneck capacity of an s-t path as the least residual capacity of its edges.

Therefore, a path is an augmenting path iff it has non-negative bottleneck capacity. We restate the max-flow min-cut theorem in the following alternate form:

Theorem 10.12 (Alternate Max-Flow Min-Cut Theorem)**.** *The following are equivalent:*

1. *There is an s-t cut S with $C(S, S^c) = \text{Val}(f)$.*
2. *f is a max flow in G.*

3. *There is no augmenting path for f in G.*

Proof. (1) implies (2) is proven by weak duality (Corollary 10.6). To show (2) implies (3), suppose there is an augmenting path p despite f being a max flow. Then we can add flow along the path equal to the bottleneck capacity of p, contradicting the maximality of f. Therefore, no augmenting path exists. To show (3) implies (1), assume there exists no augmenting paths. Define S as the set of vertices reachable by s in G_f. Here reachability is defined in the sense that $E = \{(u, v) : c_f(u, v) > 0\}$. Note that $t \notin S$ because if it were then there is an augmenting path $s \rightsquigarrow t$. Furthermore any edge between a vertex in S and S^c must be at full capacity. Then $C(S, S^c) = f(S, S^c)$, proving maximality. □

10.3 Floyd-Fulkerson Algorithm

This begs the natural question, how do we calculate the max-flow efficiently? Two algorithms are generally used: Ford-Fulkerson and Edmonds-Karp. Naturally, they have different complexities and are therefore useful in different situations. We present both here.

Algorithm 10.13 (Ford-Fulkerson)**.** Given a flow-network $G = (V, C, s, t)$ where we assume $C : V^2 \to \mathbb{Z}^+$ (integer capacities), we initialize a flow $f = 0$. We search for an augmenting path in G_f using a depth-first search. If we find a path, we augment f along the path by the bottleneck capacity and decrease the flow along the 'reverse-path' by the bottleneck capacity. We repeat searching for an augmenting path in G_f until one cannot be found.

Correctness: Correctness follows directly from Lemma 10.9.
Runtime: Each depth-first search costs $O(E)$. Since we assume integer capacities, the bottleneck capacity is at least 1. So at most $\text{Val}(f)$ repetitions of the depth-first search will run. Therefore, the runtime is $O(Ef)$.

It should be noted that Ford-Fulkerson is not guaranteed to terminate if the capacities are irrational.

10.4 Edmonds-Karp Algorithm

The alternative to Ford-Fulkerson is Edmonds-Karp. It is remarkably similar, except instead of performing a depth-first search we perform a breadth-first search and select the shortest path augmenting path. We restate it for consistency.

Algorithm 10.14 (Ford-Fulkerson)**.** Given a flow-network $G = (V, C, s, t)$ where we assume $C : V^2 \to \mathbb{Z}^+$ (integer capacities), we initialize a flow $f = 0$. We search for the shortest augmenting path in G_f using a breadth-first search. If we find a path, we augment f along the path by the bottleneck capacity and decrease the flow along the 'reverse-path' by the bottleneck capacity. We repeat searching for an augmenting path in G_f until one cannot be found.

Correctness: Correctness again follows directly from Lemma 10.9.
Runtime: Each breadth-first search costs $O(E)$. In each augmentation, we saturate at least one of the E edges. Therefore, this can occur at most E times befores the length of the path returned by the breadth-first search must increase. Furthermore, the maximum length of the path returned by the breadth-first search is V. It follows then that at most EV augmentations of the path occur. This yields the runtime $O(VE^2)$.

In general, use Edmonds-Karp over Ford-Fulkerson if $\text{Val}(f) = \Omega(EV)$ or if the capacities are irrational. (If they are rational, you can multiply all of them by a suitable parameter to make them all integers).

11 Linear Programming

11.1 Definition and Importance

The final topic in this course is Linear Programming. We say that a problem is an instance of linear programming when it can be effectively expressed in the linear programming framework. A linear programming problem is an *optimization* problem where the optimization function is a linear function. As we have seen before an optimization problem is the following:

Definition 11.1 (Optimization Problem)**.** An optimization problem is a function $f : \Sigma \to \mathbb{R}$ along with a subset $F \subseteq \Sigma$. The goal of the problem is to find the $x \in F$ such that for all $y \in F$, $f(x) \leq f(y)$. We often call Γ the *feasible region* of the problem.

In the case of linear programming, we take $\Sigma = \mathbb{R}^n$, require f to be linear i.e. $f(x) = c^T x$ for some $c \in \mathbb{R}^n$, and the feasible region F is a convex polytope sitting in n-dimensional space. Another way of thinking of a convex polytope is that it is the intersection of finitely many half-planes or as the set of points that satisfy a finite number of affine inequalites.

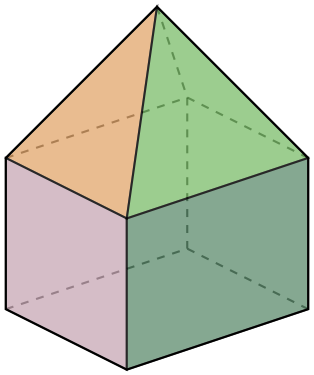

Figure 11.1: An example of a convex polytope. We can consider each face of the polytope as an affine inequality and then the polytope is all the points that satisfy each inequality. Notice that an affine inequality defines a half-plane and therefore is also the intersection of the half-planes.

Definition 11.2 (Convex Polytope)**.** The following are equivalent:

1. For $u_1, \ldots, u_m \in \mathbb{R}^n$ and $b_1, \ldots, b_m \in RR$, the set of $x \in \mathbb{R}^n$ s.t. $u_i^T x \leq b_i$ is a convex polytope.

2. Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$, the set of $x \in \mathbb{R}^n$ s.t. $Ax \leq b$ is a convex polytope.

3. Given a set of points $y_1, \ldots, y_k \in \mathbb{R}^n$, the convex hull $\mathbf{conv}(y_1, \ldots, y_k)$ is a convex polytope.A convex hull $\mathbf{conv}(y_1, \ldots, y_k)$ is the intersection of all convex sets containing $y_1, \ldots, y_k$.

I leave it as an exercise to show that the three definitions are equivalent.

Remark 11.3. *If F is a convex region and $x, y \in F$ then for every $\lambda \in [0,1], \lambda x + (1-\lambda)y \in F$. Equivalently, the line segment $\overline{xy} \subseteq F$.*

The standard form of a Linear Program is the following. We will see later how to convert all linear programs into the standard form with only a $O(1)$ blowup in complexity.

Standard form of a (Primal) Linear Program.

$$(\mathcal{P}) = \begin{cases} \max & c^T x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{cases} \tag{11.1}$$

The reason that we care about linear programs is that most of the problems we are interested in have an equivalent formulation as a linear program. Therefore, if we build and optimize techniques for solving such problems, they will directly produce better algorithms for all the problems. Let's first see how max flow can be expressed as a linear program.

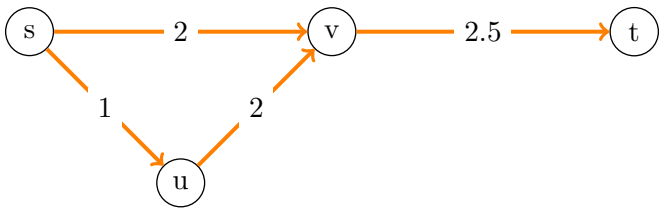

Figure 11.2: An example capacitated network

If we wanted to calculate the max flow in the capacitated network in Figure 11.2 then we could write that as the following linear program

$$\begin{aligned} \max \quad & f_{su} + f_{sv} \\ \text{s.t.} \quad & 0 \leq f_{sv} \leq 2 \\ & 0 \leq f_{su} \leq 1 \\ & 0 \leq f_{uv} \leq 2 \\ & 0 \leq f_{vt} \leq 2.5 \\ & f_{su} - f_{uv} = 0 \\ & f_{sv} + f_{uv} - f_{vt} = 0 \end{aligned} \tag{11.2}$$

We can make some simplifications by equating $f_{su} = f_{uv}$ and $f_{vt} = f_{uv} + f_{sv}$. Then we can rewrite the linear program in the standard form as

$$\begin{aligned} \max \quad & \begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} f_{su} \\ f_{sv} \end{pmatrix} \\ \text{s.t.} \quad & \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_{su} \\ f_{sv} \end{pmatrix} \leq \begin{pmatrix} 2 \\ 1 \\ 2 \\ 2.5 \end{pmatrix} \\ & f_{su}, f_{sv} \geq 0 \end{aligned} \tag{11.3}$$

This is just one instance of writing a problem we have seen as a linear programming problem. We will see more and we will also see an algorithm for how to solve linear programs.

11.2 Optimums in the Feasible Region

Definition 11.4 (Feasibility)**.** For a linear program in the standard form of (11.1), we say the *feasible region* is the set $F = \{x : Ax \leq b\}$. If $F \neq \emptyset$, we say F is *feasbile*. We say that the linear program is unbounded if F is unbounded.

A first question we need to ask ourselves when solving a linear program is whether a linear program is feasible. This itself is non-trivial. Assuming, however, that F is feasible, how would we go about finding an optimum? We exploit the two properties that the objective

function $c^T x$ is linear and that F is a convex polytope.

Lemma 11.5 (Local optimums are global optimums)**.** *Given $F \neq \emptyset$, if a point $x \in F$ is a local optimum then it is a global optimum.*

Proof. Assume for contradiction, x isn't a global optimum. Then $\exists y \in F$ s.t. $c^T x < c^T y$. But the line $\overline{xy} \in F$. Then for any $\lambda \in (0, 1]$,

$$c^T(\lambda x + (1-\lambda)y) = \lambda c^T x + (1-\lambda)c^T y > c^T x \tag{11.4}$$

Meaning, x is not a local optimum as moving towards y increases the objective function. □

Definition 11.6 (Vertex of a Polytope)**.** Recall that we say z is on the line segement $\overline{xy}$ if there $\exists \lambda \in [0, 1]$ s.t. $z = \lambda x + (1-\lambda)y$. A point $z \in F$ is a vertex of the polytope F if it is on no proper line segment contained in F.[56]

Remark 11.7. *If $v_1, \ldots, v_k$ are the vertices of a polytope F, then $F = \boldsymbol{conv}(v_1, \ldots, v_k)$. i.e. F is the convex hull of the vertices.*[57]

Theorem 11.8. *Let* **OPT** *be the optimal value of a standard form linear program and assume* **OPT** *is finite. Then* **OPT** *is achieved at a vertex.*

Proof. Let $v_1, \ldots, v_k$ be the vertices of F. Then every point $x \in F$ can be expressed as $\sum_{i=1}^{k} \lambda_i v_i$ with each $\lambda_i \geq 0$ and $\sum \lambda_i = 1$. By linearity of the objective function, if $\mathbf{OPT} = c^T x$ for $x \in \mathbb{R}^n$, then necessarily $c^T v_i \geq \mathbf{OPT}$ for some v_i, so the optimal value is achieved at the vertex. □

This statement is rather strong in that it means that we only need to check the vertices of a polytope in order to find the optimum. In particular, it also demonstrates that the proper way of thinking about a polytope is not by the number of linear equations that define it, but rather by the number of vertices. This is actually a very fruitful section of algorithm theory. There has been considerable work done to consider how to reduce the number of vertices. However, this is not always optimal. Consider the n-dimensional sube. It has 2^n vertices. In general if the polytope is defined by m half planes there are up $\binom{m+n}{n}$ vertices. We will soon see the simplex algorithm which will reduce the number of vertices we need to consider.

[56] A vertex is just the n-dimensional analog of what you think of as the vertex of a polytope in low dimension.

[57] This can be shown by induction on the dimension. The proof is a little tedious, so it has been ommitted.

11.3 Dual Linear Program

Consider the following simple linear program. Imagine there is a salesman who wants to sell either pens or markers. He sells pens for S_1 dollars and markers for S_2 dollars. He has restrictions due to labour, ink, and plastic. These are formalized in the following linear program.

$$\begin{aligned} \max \quad & S_1x_1 + S_2x_2 \\ \text{s.t.} \quad & L_1x_1 + L_2x_2 \leq L \\ & I_1x_1 + I_2x_2 \leq I \\ & P_1x_1 + P_2x_2 \leq P \\ & x_1, x_2 \geq 0 \end{aligned} \tag{11.5}$$

Let's consider the dual of the problem. Imagine now there are market prices for each of the three materials: labour, ink, and plastic. Call these y_L, y_I, y_P. Now the salesman is only going to sell pens if $y_LL_1 + y_II_1 + y_PP_1 \geq S_1$ and analagously for markers. Therefore, it is in the interest of the market prices to minimize the total available labour, ink, and plastic while still allowing the salesman to sell his goods. This is the dual problem and it can be expressed as follows.

$$\begin{aligned} \min \quad & y_LL + y_II + y_PP \\ \text{s.t.} \quad & y_LL_1 + y_II_1 + y_PP_1 \geq S_1 \\ & y_LL_2 + y_II_2 + y_PP_2 \geq S_2 \\ & y_L, y_I, y_P \geq 0 \end{aligned} \tag{11.6}$$

In general, the dual to the standard linear program expressed in (11.1) is

Standard form of a Dual Linear Program.

$$(\mathcal{D}) = \begin{cases} \min & b^T y \\ \text{s.t.} & A^T y \geq c \\ & y \geq 0 \end{cases} \tag{11.7}$$

The dual of a linear program is a powerful thing. An optimal solution to the dual gives us a bound on the solution to the primal.

Theorem 11.9 (Weak LP Duality)**.** *If $x \in \mathbb{R}^n$ is feasible for $(\mathcal{P})$ and $y \in \mathbb{R}^m$ is feasible for $(\mathcal{D})$, then*

$$c^T x \leq y^T A x \leq b^T y \tag{11.8}$$

Thus, if $(\mathcal{P})$ is unbounded, then $(\mathcal{D})$ is infeasible. Conversely if $(\mathcal{D})$ is unbounded, then $(\mathcal{P})$ is infeasible. Moreover, if $c^T x' = b^T y'$ with x' feasible for $(\mathcal{P})$ and y' feasible for $(\mathcal{D})$, then x' is an optimal solution for $(\mathcal{P})$ and y' is an optimal solution for $(\mathcal{D})$.

Proof. Assume $x \in \mathbb{R}^n$ is feasible for $(\mathcal{P})$ and $y \in \mathbb{R}^m$ is feasible for $(\mathcal{D})$. Therefore $Ax \leq b$ and $A^T y \geq c$. Multiplying the first equation by y^T on the left

$$y^T A x \leq y^T b = b^T y \tag{11.9}$$

Taking the transpose of the second equation ($y^T A \geq c^T$) and multiplying by x on the right

$$y^T A x \geq c^T x \tag{11.10}$$

Combining (11.9) and (11.10) generates (11.8). If $(\mathcal{P})$ is unbounded then for any M, $\exists$ a feasible x s.t. $c^T x > M$. By (11.8), any feasible y must satisfy $b^T y > M$ for any M. Clearly this is impossible so $(\mathcal{D})$ is infeasible. The converse follows similarly.

Assume x' wasn't optimal but $c^T x' = b^T y'$. Then $\exists x^\star$ feasible in $(\mathcal{P})$ with $c^T x' < c^T x^\star$. But then $b^T y' < c^T x^\star$, violating (11.8). So x' is optimal. A similar argument shows y' is optimal. □

11.4 Simplex Algorithm

11.5 Approximation Theory

11.6 Two Person Zero-Sum Games

11.7 Randomized Sorting Complexity Lower Bound

11.8 Circuit Evaluation

11.9 Khachiyan's Ellipsoid Algorithm

11.10 Set Cover, Integer Linear Programming