CS 38/138: An Introduction to Algorithms

Spring 2016

Notes

Chinmay Nirkhe

Contents

1 Preface

I took this course in the spring of 2015 and was a TA for the course in 2016. I wrote these notes as an extension of most of the recitations I gave during the year and in particular tried to emphasize how to write proofs effectively and concisely. The first couple chapters will have plenty of examples of fully written proofs for algorithms and you should use these as templates for writing your solution sets. The later chapters will relax this slightly; I will be more succinct and might omit certain parts of the proof as exercises for you the reader.

About the structure of the course. When these notes were written, the course was 40% homework, 20% midterm, 40% final. That meant over the 7-8 sets, each problem on a set was worth about 1% of your grade. This is not something worth losing sleep over! Its far more important to get a deep conceptual understanding of the material. Most importantly, this was noticeable in the two exams. The exams will test you on slightly different things than the sets. While each set will generally introduce 1-2 new algorithm topics and test you on them, the exams will test you on all the topics till that date and in particular will test you on your ability to look at a problem and quickly figure out what type of algorithm it is looking for. More often that not, students spend too long on a problem on the exam trying to find an algorithm of the wrong type.

I hope you enjoy this course as much as I did and feel free to ask me any questions. If you spot errors in these notes please let me know right away as I guarantee you that there will be plenty. I had a great time writing these notes and as I wrote them, I realized there was so much I hadn't understood the first or second time around!

Additionally, a lot of this was copied from previous CS 38 notes written by Leonard Schulman, William Hoza, Nicholas Schiefer, and many others. Thank you!

–Chinmay Nirkhe, chinmay@caltech.edu

2 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).

2.1 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:

(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]

$$m \leftarrow \max_{x \in a}\{x\} \tag{2.1}$$

(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 (2.1), if written nicely, you won't even have to deal with indexing! Focus on making sure the algorithm is clear, not the implementation.

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

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

(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.

2.2 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 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 2.1 (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 2.2 (Optimization Problem)**.** An optimization problem is a function $f : \Sigma \to \mathbb{R}$[5] along with a subset $\Gamma \subseteq \Sigma$. The goal of the problem is to find the $x \in \Gamma$ such that for all $y \in \Gamma$, $f(x) \leq f(y)$.

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

$$x \leftarrow \arg\min_{y \in \Gamma} \{f(y)\} \tag{2.2}$$

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

[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 $g : \Sigma^* \to \mathbb{R}$.

[5] For the mathematicians out there reading this, you only need $f : \Sigma \to T$, where T is a set with a total ordering.

2.3 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.

2.4 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 2.3 (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 2.4 (Longest Increasing Subsequence)**.** This is a dynamic programming algorithm.[6] 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.[7]

1. For $j = 1$ to n:[8]
   (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.[9]

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.[10] 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

[6] 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.

[7] 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.

[8] 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.

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

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

stated above, recursively finds the solution to the subproblme.[11] 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: $O(n^2)$.[12]

[11] 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.

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

3 Runtime Complexity and Asymptotic Analysis

3.1 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, $f(n) \leq g(n)$.[13] Formally this means the following:

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

(b) (antisymmetry) If $f \leq g$ and $g \leq f$ then $f = g$.[14]

(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*.

Definition 3.1 (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 3.2 (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.

[13] 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 $f(n) > g(n)$.

[14] 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 3.3 (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 3.4. 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 linear 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 linear factors, its just that its not the concern of this course. A lot of work in other areas of computer science focus on the linear 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 linear 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 3.5 (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.

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 3.6 (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 $(x_1, \ldots, x_k)$.[15]

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.

3.2 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 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. [16] Is this a quantum machine? Also, interesting but in

[15] 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.

[16] If you consider a multi threaded machine with k threads, then any computation that takes time t to run on the multithreaded 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 multithreaded machine. If k is a constant, this doesn't affect asymptotic runtime.

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 now you could solve the problem of finding the mode of a set in $O(n)$ time with $O(1)$ additional space. Luckily, the models we're going to use have some logical intuition behind them and you wouldn't have any such silly pitfalls.

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.[17] We further assume that reading and writing a single bit takes constant time.

Recall that a n-bit integer can be stored using $O(\log n)$ bits. So addition, subtraction, and multiplication of n-bit integers naïvely takes $O(\log n)$ time.[18] 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 $O(n^3)$.[19] The complexity of the multiplication of the elements isn't directly relevant to the matrix multiplication algorithm itself and can be factored in later.

[17] 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.

[18] 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.

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

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. 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.

4 Introductory Topics

4.1 Recursion

Definition 4.1 (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.[20]

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. 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 4.2 (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 $\gcd(a, b) = \max\{\text{DIVS}(a, b)\}$.

Let's start by creating a naïve algorithm for the gcd problem.[21] 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 a lot of cases here, but under closer observation a lot of the checks were redundant. For example, if we showed that 5 didn't divide either 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 4.3. *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 4.4. *For integers a, b, not both 0,* $\text{DIVS}(a, b) = \text{DIVS}(a + kb, b)$ *for $k \in \mathbb{Z}$, and therefore* $\gcd(a, b) = \gcd(a + kb, b)$.

Proof. Apply induction. The gcd argument follows at is the max element of the same set. □

Let's make a stronger statement. Recall that one way to think about $a \pmod b$ is the unique number in $\{0, \ldots, b\}$ that is equal to $a + kb$ for some $k \in \mathbb{Z}$.[22] Therefore, the following corollary also holds.

Corollary 4.5. *For integers a, b, not both 0,* $\gcd(a, b) = \gcd(a \pmod b, b)$.

[20] 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.

[21] This is generally a good practice to follow especially in interview questions at companies. Start by stating a naïve algorithm, state its faults and how you could go about improving it.

[22] The more 'mathy' way of thinking about $a \pmod b$ 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 a group known as $\mathbb{Z}/b\mathbb{Z}$. Addition is defined on the conjugacy classes as a consequence of addition on any pair of elements in the conjugacy classes permuting the classes. Read any Abstract Algebra textbook for more information.

This simple fact is going to take us home. We've found a way to recursively reduce the larger of the two inputs (wlog [23] assume a) to strictly less than b. Because it's strictly less than b, we know that this repetitive recursion will actually terminate. By terminate, we mean that we will reach a base case that we know the solution of. In this case, let's assume our base case is naïvely that $\gcd(a, 0) = a \ \forall \ a$. Just for the sake of formality, I've stated this as an algorithm below:

Algorithm 4.6 (Euclid-Lamé)**.** Given integer inputs inputs a, b with $a \geq b$, if $b = 0$ then return a. Otherwise, return the $\gcd(b, a \pmod b)$ calculated recursively.[24]

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. A truly great proof would also say something about negative inputs and why this case isn't to be worried about (hint $\gcd(a, b) = \gcd(a, -b)$).

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:

$$F_k = \frac{1}{\sqrt{5}}\phi^k - \frac{1}{\sqrt{5}}\phi'^k \tag{4.1}$$

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 $\left|\phi'\right| < 1$ so F_k tends to $\phi^k/\sqrt{5}$. More importantly, it grows exponentially.

Most times, you're complexity argument will be the smallest argument. Let's make the following Theorem about the complexity:

Theorem 4.7. *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

$$b \pmod c \leq b - c \leq b - F_{k+1} < F_{k+2} - F_{k+1} = F_k \tag{4.2}$$

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 a nice exercise to finish this last bit.

4.2 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.[25] Formally for optimization

[23] without loss of generality.

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

[25] There of course also duals of minimization problems. Just consider the negation of the maximization problem as per usual.

problems,

Definition 4.8 (Duality)**.** A minimization problem $\mathcal{D}$ is considered the *dual* of a maximization problem $\mathcal{P}$ is the solution of $\mathcal{D}$ provides an upper bound for the solution of $\mathcal{D}$. 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 4.9. *For integers a, b, not both 0,* SUMS(a, b) = SUMS$(a + kb, b)$ *for any $k \in \mathbb{Z}$, and therefore* SUMS(a, b) = SUMS$(a \pmod b, b)$.

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

Theorem 4.10 (Strong Dual of GCD)**.** *For integers a, b, not both 0,*

$$\min\{\text{SUMS}(a, b)\} = \max\{\text{DIVS}(a, b)\} = \gcd(a, b) \tag{4.3}$$

Proof. Its easy to see as $\gcd(a, b)$ divides a and b then it divides any $ax + yb$ proving weak duality. For strong duality, assume for contradiction, that there exists (a, b) such that $a + b$ is the smallest.[26] But then the pair $(b, a - b)$ yields the same set of SUMS however, $b + (a - b) = b < a + b$, a contradiction. □

4.3 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 cost $O(1)$ 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)[27]. Then calculate x^{2^j} for $j = 1, \ldots, m$ as before but this time writing each of them into memory. This takes $O(\log n)$ space. If n has binary representation $(a_m a_{m-1} \ldots a_0)_2$ where $a_j \in \{0, 1\}$ then

$$x^n = \prod_{j=0}^{m} x^{a_m 2^j} \tag{4.4}$$

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.[28]

[26] 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 contraction. Here we define smallest by the magnitude of $a + b$.

[27] This makes $m \in O(\log n)$

[28] If we wanted to calculate all powers $x, \ldots, x^n$ 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.

5 Dynamic Programming

Before you get some alternate idea, let me state it that *Dynamic Programming is a form of recursion.* In Computer Science, you have probably heard the tradeoff between Time and Space. This has nothing to do with General relativity, but has to do with the trade off between the space complexity on the memory and the time complexity of the algorithm[29]. 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.

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.

5.1 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.

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

5.2 Tribonacci Numbers

I'll introduce computing 'tribonacci' numbers as a preliminary example[30] 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 as $T_7 = T_4 + T_5 + T_6$. 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.[31] To remedy this, we are going to write down a table of values. Here let's assume the word model again.

[29] I actual prefer to think about this as a 3-way tradeoff between time complexity, space complexity, and 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.

[30] The easiest example is Fibonacci numbers but as I've given you the explicit formula in the previous chapter, it seems moot. Although this problem is also rather easily solvable with recurrence relations, but bear with me.

[31] This isn't abusive notation. This is equivalent to saying the complexity is $O(b^k)$ for some base b. Check for yourself that this is true.

Algorithm 5.1 (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.[32] Therefore, $t[k] = T_k$.

Complexity. Calculation of each $T[j]$ is constant given the previously filled values. As $O(k)$ such values are calculated, the total complexity is $O(k)$.[33]

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

5.3 Generic Algorithm and Runtime Analysis

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

Algorithm 5.2 (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.

It's not necessary for "smallest" and "biggest" to refer to literal size. All that matters is that when the algorithm comes to a subproblem, it should be ready for it, i.e. it should already have solved the sub-subproblems that are useful for solving that subproblem.
The basic formula for the runtime of a DP algorithm is

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

(Warning: 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[34], so this $O(1)$ lookup time is correct. Therefore, the total time is

[32] 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.
[33] In one of the recitations, we will show how Fibonacci (also same complexity) can actually be run faster than $O(k)$ using repeated squaring.
[34] In practice, there is a more efficient dynamic programing algorithm called memoization, which we will cover soon.

bounded by the time complexity to solve a subproblem multiplied by the number of subproblems.

5.4 Edit Distance, an example

Now might be a good time and go back to Chapter 2 and read the example problem there if you haven't done so already.

Definition 5.3 (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*[35] 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, I'm going to 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. (Our final goal is to calculate $d(m, n)$ 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. 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, we write this as:

$$d(k, \ell) \leftarrow \min \begin{cases} d(k, \ell - 1) + 1 \\ d(k-1, \ell) + 1 \\ d(k-1, \ell-1) + 2 \cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{5.1}$$

Here $\mathbb{1}_{\{\cdot\}}$ 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 5.4 (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 by the

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

following rule:

$$t[k,\ell] \leftarrow \min \begin{cases} t[k,\ell-1]+1 \\ t[k-1,\ell]+1 \\ t[k-1,\ell-1]+2\cdot \mathbb{1}_{\{x_k \neq y_\ell\}} \end{cases} \tag{5.2}$$

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 a 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 these 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.

5.5 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 called 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. 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 5.5 (Memoized Dynamic Programming Algorithm)**.** Assume that $f(i) = g(f(j_1), \ldots, f(j_k))$ for some function g. Construct a table t. 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 linear-factor of subcases are being ignored then you cannot argue a different in complexity.

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 computers 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.[36] In the case of Tribonacci numbers as every prior Tribonacci number or Edit Distance as the entire table will be filled up, bottom-up method is better.

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.

5.6 1D k-Clustering

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

Definition 5.6 (1D Cluster radius)**.** Given a set X of n 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

$$r = \max_{1 \le i \le n} \left(\min_{1 \le j \le n} |x_i - c_j| \right) \tag{5.3}$$

Exercise 5.7 (1D k-Clustering)**.** Given a set X of n 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 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, pictorally, these sets partition the number line into distinct regions. Figure 5.1 demonstrates this partition.

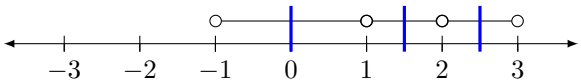

Figure 5.1: Assume that $C = \{-1, 1, 2, 3\}$. This divides the number line into partitions indicated by the blue lines. For any x_i the unique point of C within the partition is the closest point of C. In particular, for any point c_j, S_j is the set of x_i in its partition.

[36] 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.

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 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 5.8 (1D k-Clustering)**.** For each $j = 1$ to k and for each $t = j$ to n: If $j = 1$ then set

$$c[t, j] \leftarrow \left(\frac{x_t - x_1}{2}, \left\{ \frac{x_t + x_1}{2} \right\} \right) \tag{5.4}$$

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

$$c[t, j] \leftarrow \left(f(s), c_{s,j-1} \cup \left\{ \frac{x_t + x_{s+1}}{2} \right\} \right) \tag{5.5}$$

Algorithm Correctness. The table c stores the cost in the first component of the tuple and the optimal set of centers that generate it in the second component. In the case that $j = 1$, then we are considering a single center. The optimal center has minimum distance to 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 one of the clusters should end. We minimize over all the possibilities.

Algorithm Complexity. There are $O(nk)$ subproblems and each subproblems computation requires at most $O(n)$ steps. Therefore, $O(n^2 k)$.

Note: We can notice furthermore that $c_{s,j-1}$ will be a (weakly) monotone increasing quantity in s and that $(x_t - x_s)/2$ is monotone decreasing. This allows us to instead run a binary search to find the optimal s decreasing runtime to $O(nk \log n)$ as each subproblem now runs in $O(\log n)$.

5.7 How exactly hard is NP-Hard?

Definition 5.9 (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 can an analogous distinction.

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

5.8 Knapsack

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

Exercise 5.10 (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 $O(n2^n)$ and is incredibly inefficient.

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 5.11 (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 $V(i, W') = 0$.
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.[37] 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.[38]

5.9 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 5.12 (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).

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

[37] 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!

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

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 $i, j \notin R$, 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.

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

$$C(R, j) \leftarrow \min_{i \in R} \big(d_{ij} + C(R - \{i\}, i)\big) \tag{5.6}$$

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) and with a little work you can show that each subproblem is $O(n)$ time plus the subsubproblem time. 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.

5.10 Additional Example: Box Stacking

Exercise 5.13 (Box Stacking)**.** Given a collection of n 3D boxes of integer dimensions $\{b_1, \ldots, b_n\}$ where $b_i = (\ell_i, w_i, h_i)$, calculate the height of the tallest stack of boxes you can form under the following constraints: A box cannot be rotated[39] and a box b_i can only be stacked on b_j if the base of b_i fits completely within the base of b_j.

Algorithm 5.14 (Box Stacking)**.**
Algorithm Idea. Sort all the boxes by base area ($\ell_i \times w_i$). Clearly a box of smaller base area cannot fit below a larger. Let $m(i)$ be the height of the tallest stack of boxes with box b_i at the top. Here is a recursive definition for $m(i)$:

$$m(i) = \max \left\{ h_i, \max_{\substack{j>i: \\ w_i \le w_j, \ell_i \le l_j}} \{m(j) + h_i\} \right\} \tag{5.7}$$

Why is this the definition? We consider all possible blocks b_j that could be directly below b_i in the stack and then maximize recursively to get the maximum stack height. Because of the sorting, no block b_j with $j < i$ can fit below block b_i.

Algorithm Description. Sort the boxes by base area so that b_1 has the least base area. Create a table t of size n indexed by $1 \le i \le n$. Starting with $i = n$ to 1, define $t[i]$ by,

$$t[i] \leftarrow \max \left\{ h_i, \max_{\substack{j>i: \\ w_i \le w_j, \ell_i \le l_j}} \{t[j] + h_i\} \right\} \tag{5.8}$$

Return the following value:

$$\max \big\{ t[1], \ldots, t[n] \big\} \tag{5.9}$$

[39] Additional exercise: solve this problem where rotation is allowed. You can still come up an algorithm of the same complexity.

Algorithm Correctness. By the algorithm idea, $t[i] = m(i)$. The tallest stack is clearly the one maximizied over all choice of top box.

Algorithm Complexity. The space complexity is $O(n)$ as we only store table t. The algorithm complexity is $O(n \log n)$ for sorting followed by $O(n^2)$ computations as the calculation of each $t[i]$ is $O(n)$ given the previously solved subproblems and there are $O(n)$ subproblems. Total complexity: $O(n^2)$.

5.11 Additional Example: Longest Common Subsequence

Also available as CLRS Ex. 15-4.

Exercise 5.15 (Longest Common Subsequence)**.** Given input strings $x = [x_1 \dots x_m]$ and $y = [y_1 \dots y_n]$, find the length of the longest common subsequence (LCS). A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous[40]. Ex. If $x = [abcdgh]$ and $y = [aedfhr]$ then the LCS is $[adh]$ of length 3.

Algorithm 5.16.
Algorithm Idea. Let $\ell(x, y)$ be the length of the LCS of x and y. Here is a recursive definition of $\ell(x, y)$ for $m, n > 1$:

$$\ell(x_1 \dots x_m, y_1 \dots y_n) = \begin{cases} 1 + \ell(x_1 \dots x_{m-1}, y_1 \dots y_{n-2}) & \text{if } x_m = y_n \\ \max\left\{\ell(x_1 \dots x_{m-1}, y_1 \dots y_n), \ell(x_1 \dots x_m, y_1 \dots y_{n-1})\right\} & \text{o.w.} \end{cases} \tag{5.10}$$

Why is this the definition? If the last characters of x and y match, then the are necessarily in the LCS, so recursively check the substring excluding them. If the last characters don't match then, then at least one of them isn't in the LCS, so check both cases and choose the maximum.

Algorithm Description. Generate a table t of size mn indexed by tuples (i, j) for $1 \le i \le m$ and $1 \le j \le n$. Set $t[1, 1] \leftarrow 1$ if $x_1 = y_1$ and $t[1, 1] \leftarrow 0$ otherwise. Systematically, apply the following recursive definition to calculate all $t[i, j]$:

$$t[i][j] \leftarrow \begin{cases} 1 + t[i-1, j-1] & \text{if } x_i = y_j \\ \max\left\{t[i-1, j], t[i, j-1]\right\} & \text{o.w.} \end{cases} \tag{5.11}$$

Here we assume $t[i, j] = 0$ if $i < 1$ or $j < 1$. Return $t[m, n]$.

Algorithm Correctness. As an exercise, convince yourself that the algorithm idea above is correct. Then, as $\ell(x_1, y_1) = t[1, 1]$ and the similarity of (5.10) and (5.11), it follows that $t[i, j] = \ell(x_1 \dots x_i, y_1 \dots y_j)$ for $1 \le i \le m$ and $1 \le j \le n$. Then returning $t[m, n]$ yields the desired result.

Algorithm Complexity. The space complexity is $O(mn)$ as we only store t. The time complexity is $O(mn)$ as well because the calculation of each $t[i, j]$ is $O(1)$ given the subproblems have been solved and there are $O(mn)$ subproblems.

Exercise 5.17 (Longest Common Subsequence Extension)**.** Given an $O(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

[40] Additional exercise: consider the related problem of finding the longest common substring (here we require that the string be contiguous). You can build a related dynamic problem algorithm that is of the same complexity, but a smarter algorithm can be built using a generalized suffix tree to solve this in $O(n + m)$.

Algorithm 5.18. Sorting the sequence. Return the LCS of the sorted sequence and the original sequence.

Algorithm Correctness. The sorted sequence is by definition monotonically increasing so any subsequence has the same property. Therefore, the longest common subsequence with the original sequence is, by construction, (a) the longest and (b) monotonically increasing.

Algorithm Complexity. Sorting is an $O(n \log n)$ algorithm and LCS as shown above is $O(n^2)$ as the strings have the same length. Total complexity: $O(n^2)$.

5.12 Additional Example: Possible Scores

Exercise 5.19 (Possible Scores)**.** Consider a game where a player can score either $p_1, p_2, \ldots, p_m$ points in a move. Given a total score n, find number of ways to reach the total (order dependent).

Algorithm 5.20. Generate a table t of size $n+1$ indexed $0 \ldots n$. For base case, set $t[0] \leftarrow 1$. Then for $i = 1, ..., n$, set

$$t[i] \leftarrow \sum_{j=1}^{m} t[i - p_j] \tag{5.12}$$

where we use the notation $t[k] = 0$ for $k < 0$. Return $t[n]$.

Algorithm Correctness. We prove by induction. For $n = 0$ (base case), there is a unique way to reach the total: no scores were made. Assume for induction that correctness holds for all $n' < n$. Consider a score sequence $a_1, \ldots, a_k$ that sums to n. Then necessarily $a_1, \ldots, a_{k-1}$ sums to $n - a_k$. By the induction hypothesis, the number of ways to total $n - a_k$ is equal to $t[n - a_k]$. Therefore, by considering all possible a_k (i.e. the final score in the sequence), we count all ways to total n. The choices for a_k are precisely, $p_1, \ldots, p_m$, which the algorithm considers.

Algorithm Complexity. The space complexity is $O(n)$ as we store $(n+1)$ elements each requiring $O(1)$ space. The time complexity is $O(nm)$ as we calculate each $t[i]$ for $i = 0, \ldots, n$ and each calculating is the sum of m numbers. If m is small then the algorithm has complexity $O(n)$ which we will see next can be beat!

Algorithm 5.21 (A Better Algorithm)**.** In the previous algorithm, we calculated each $t[i]$ from $i = 0, \ldots, n$. But this is tedious as there is a lot of repetition. Let's adjust this by using the repetition of squaring trick. Informally, this states that we can calculate x^n in time $O(\log n)$ by squaring until we reach the exponent. For more details, read the week 1 recitation notes.

For this part, we are **not** going to assume $p_1, \ldots, p_m, m$ or n are small numbers. Rather they can be potentially large, so our complexity will rely on this. However, we will assume that $p_1 \leq \ldots \leq p_m$. Recall the recurrence relation:

$$a_n = a_{n-p_1} + a_{n-p_2} + \ldots + a_{n-p_m} \tag{5.13}$$

This can be expressed in the equivalent matrix form:

$$\begin{pmatrix} 0 & & \\ \vdots & \mathbf{I} & \\ 0 & & \\ 1 & \ldots & 1 \end{pmatrix} \cdot \begin{pmatrix} a_{n-p_m} \\ \vdots \\ a_{n-2} \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} a_{n-p_m+1} \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix} \tag{5.14}$$

where $\mathbf{I}$ is the identity matrix of size $p_m - 1$. Write

$$M = \begin{pmatrix} 0 & & \\ \vdots & \mathbf{I} & \\ 0 & & \\ 1 & \dots & 1 \end{pmatrix} \tag{5.15}$$

Then, by repeated application of (5.14),

$$M^{n-p_m} \cdot \begin{pmatrix} a_1 \\ \vdots \\ a_{p_m} \end{pmatrix} = \begin{pmatrix} a_{n-p_m+1} \\ \vdots \\ a_n \end{pmatrix} \tag{5.16}$$

By the repetition of squaring trick, we can calculate M^{n-p_m} in $O(\log n)$ matrix multiplications. We assume here that matrix multiplication can be performed in $O(k^\omega)$ time for a $k \times k$ matrix[41]. The result is $O({p_m}^\omega \log n)$ time algorithm for the matrix multiplication. We also need to calculate the initial vector; apply the initial approach detailed above to get a $O(mp_m)$ runtime algorithm for calculating the initial vector. Therefore the total complexity is

$$O\left(mp_m + {p_m}^\omega \log n\right) \tag{5.17}$$

If $p_1, \dots, p_m$ and m are small, then the complexity of this algorithm is $O(\log n)$ which is a considerable improvement over the $O(n)$ algorithm previously presented!

5.13 Additional Example: Number of paths with $\leq \ell$ turns

Exercise 5.22 (Number of paths with $\leq \ell$ turns)**.** Given m, n, ℓ integers, count the number of paths of length exactly $m + n$ to reach the bottom right square from the top left square of a $m \times n$ matrix with at most ℓ turns.[42] One can only move horizontally or vertically and a turn is considered any rotation of motion by 90 degrees.

Algorithm 5.23. If the length of the path is exactly $m+n$ then you can only move right ($\mathcal{R}$) or down ($\mathcal{D}$).

Define $c(i, j, k, d)$ as the number of paths of paths to square (i, j) with k more turns makeable with the next movement in direction d (here $d = \mathcal{R}$ or $\mathcal{D}$). Here is a recursive definition for $c(i, j, k, d)$:

$$c(i,j,k,d) = \begin{cases} c(i-1,j,k,\mathcal{D}) + c(i,j-1,k-1,\mathcal{R}) & \text{if } d = \mathcal{D} \\ c(i-1,j,k-1,\mathcal{D}) + c(i,j-1,k,\mathcal{R}) & \text{if } d = \mathcal{R} \end{cases} \tag{5.18}$$

Why is this the definition? In order to get to square (i, j) you must have come from $(i - 1, j)$ moving $\mathcal{D}$ or $(i, j-1)$ moving $\mathcal{R}$. Depending on the direction d that you end up after, it will cost you a turn if d disagrees with your direction of motion. Hence the $k - 1$, as you use a turn.

[41] The actual value of ω is a piece of great interest in theoretical computer science. There is a large class of problems that are shown to be as hard as matrix multiplication. A trivial bound for $\omega \leq 3$ using the standard matrix multiplication approach. Strassen's algorithm gives a bound for $\omega \leq \log_2 7 \approx 2.807$. The lower bound is known to be $\omega \geq 2$ because it is the minimal time required to access every element. It is an open problem if $\omega = 2$.

[42] Additional exercise: What happens when you remove this path length restriction? Can you still solve this problem using dynamic programming? If so come up with an algorithm of similar complexity.

Algorithm Description Generate a table t of size $2mn(\ell+1)$ indexed by tuples (i,j,k,d) where $1 \leq i \leq m, 1 \leq j \leq n, 0 \leq k \leq \ell$ and $d \in \{\mathcal{D}, \mathcal{R}\}$. For a base case, $t[1,1,\ell,d] = 1$ for either choice of d and $t[1,1,k,d] = 0$ for $k < \ell$. Systematically, apply the following definition to calculate all $t[i,j,k,d]$:

$$t[i,j,k,d] = \begin{cases} t[i-1,j,k,\mathcal{D}] + t[i,j-1,k-1,\mathcal{R}] & \text{if } d = \mathcal{D} \\ t[i-1,j,k-1,\mathcal{D}] + t[i,j-1,k,\mathcal{R}] & \text{if } d = \mathcal{R} \end{cases} \tag{5.19}$$

Here we assume $t[i,j,k,d] = 0$ if undefined. Then return the following sum

$$\sum_{\substack{0 \leq k \leq \ell \\ d \in \{\mathcal{D},\mathcal{R}\}}} t[m,n,k,d] \tag{5.20}$$

Algorithm Correctness. As an exercise, convince yourself that the algorithm idea above is correct. Then, by the base case of the algorithm and the similarity of (5.18) and (5.19), its easy to see that $t[i,j,k,d] = c[i,j,k,d]$. Then, it is only a matter of returning the sum of any t element of either direction having any of $0, \ldots, \ell$ turns left to make.

Algorithm Complexity. The space complexity is $O(mn\ell)$ ans we only store t. The time complexity is $O(mn\ell)$ as well because the calculation of each element of t is $O(1)$ given the subproblems have been solved and there are $O(mn\ell)$ subproblems.[43]

[43] Additional exercise: Now that you are well acquainted with dynamic programming algorithms as well as the repetition of squaring trick, construct a $O(nm \log \ell)$ algorithm to solve this problem. Hint: The approach is very similar to that applied to the possible scores problem.

6 Greedy Algorithms

The second algorithmic strategy we are gong 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.)

However, 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.

6.1 Fractional Knapsack

Let's consider a relaxation of the Knapsack problem we introduced in the Dynamic Programming chapter. 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 6.1 (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. 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 $0 \leq s_i \leq 1$.[44]

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 6.2. *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 $s_2 > 0$. Then for some small $0 < \delta$, we can define a new fractional subset by

$$s_1' = s_1 + \delta/w_1, \qquad s_2' = s_2 - \delta/w_2 \tag{6.1}$$

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

$$v_1\delta/w_1 - v_2\delta/w_2 = \delta(q_1 - q_2) > 0 \tag{6.2}$$

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 6.3 (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. □

[44] This is the *relaxation* of the indicator vector we formulated the Knapsack problem around.

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

$$q_i - q_j = \frac{v_i w_j - v_j w_i}{w_i w_j} > \frac{1}{w_i w_j} \geq 2^{-2k} \tag{6.3}$$

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 relaxation the constraint to allow fractional components of items, any optimal solution to fractional knapsack ≥ solution to classical knapsack. Also, our solution is almost integer; only the last item chosen is fractional.

6.2 Activity Selection

Exercise 6.4 (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.

As you will see in a second, the first option is the correct one. Take a moment to 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:

Lemma 6.5. *Suppose $S = ((a_{i_1}, b_{i_1}), \ldots, (A_{i_k}, b_{i_k}))$ is a feasible schedule not including (a', b'). Then we can exchange in (a_{i_ℓ}, b_{i_ℓ}) for (a', b') if $b' \leq b_{i_\ell}$ and if $\ell > 1$, then $b_{i_{\ell-1}} \leq a'$.*

Proof. We are forcing that the newt event ends before event ℓ and start after event $\ell - 1$. □

Theorem 6.6. *The schedule created by selecting the earliest-ending activity that doesn't conflict with those already selected is optimal and feasible.*

Proof. Feasibility is trivial given the construction. Let $E = ((a_{i_1}, b_{i_1}), \ldots, (A_{i_k}, b_{i_k}))$ by the output created by selecting the earliest-ending activity and $S = ((a_{j_1}, b_{j_1}), \ldots, (A_{j_\ell}, b_{j_\ell}))$ any other feasible schedule.

We claim that for all $m \leq \ell$, (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. Note $m \neq 1$ as the schedule E by construction takes the first-ending event. 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.

6.3 Minimum Spanning Trees

Let's transition to a graph theory problem. But first a plethora of definitions that are probably already familiar to you:

Definition 6.7 (Graph)**.** A graph $G = (V, E)$ is a set of vertices V and edges $E \subseteq \binom{V}{2}$ (a set of pairs of elements of V). Notationally, we write $n = |V|$ and $m = |E|$.

Definition 6.8 (Weighted Graph)**.** A weighted graph $G = (V, E, w)$ is a graph with a weight $w : E \to \mathbb{R}$ assigned to each edge.

Definition 6.9 (Path and Cycle)**.** A path 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. A cycle is a path with $v_0 = v_k$; a *simple cycle* has no repeated vertices.

Definition 6.10 (Connected)**.** A graph is connected if there is a path between every pair of vertices.

Definition 6.11 (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

$$w(G') = \sum_{(u,v) \in E'} w(u, v) \tag{6.4}$$

Definition 6.12 (Forest and Trees)**.** A forest in G is a subgraph of G without simple cycles and a tree is a connected forest.[45]

Definition 6.13 (Spanning Tree)**.** A spanning tree is a tree in G that connects all the vertices of G.

All these definitions, let to the following natural question.

Exercise 6.14 (Minimum Spanning Tree)**.** Given a connected graph $G = (V, E, w)$ with positive weights, find a spanning tree of minimum weight (an MST).

[45] To understand this naming convention, draw a couple examples and it will be clear instantaneously.

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

We start with the empty forest i.e. all vertices and no edges: $G_0 = (V, \emptyset)$. This is already a multicut S_0 of the graph G into n non-intersecting blocks, namely each block containing a unique vertex. Also, as there are no edges, this multicut S_0 coarsens G_0. Our strategy will be to add the edge of minimum weight of G that crosses S_0. This produces another forest G_1 (this time with $n-1$ non-intersecting blocks). Again this is a multicut S_1 and to build G_2, we add the edge of minimum weight of G that crosses S_1. And so on...

Algorithm 6.16 (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

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.

6.4 Matroids and Abstraction of Greedy Problems

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

Remark 6.17. *Given a finite-dimensional vector space X and two sets of linearly independent vectors V and W with (wlog) $|V| < |W|$, there is a vector $w \in W - V$ such that $V \cup \{w\}$ is linearly independent.*

We can think of this an exchange by thinking of it as 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 6.18 (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 6.19 (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. $I \subseteq J$ 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 6.20 (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 6.21 (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 6.22 (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 $x \in A \not| A| = |B|$, 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 $w(y) \leq w(x)$,

$$w(A) = w(B) - w(y) + w(x) \geq w(B) \tag{6.5}$$

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

Lemma 6.23. *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 6.24 (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:*

$$U' = \{y \in U \mid \{x, y\} \in \mathcal{F}\}, \qquad \mathcal{F}' = \{B \subseteq U - \{x\} \mid B \cup \{x\} \in \mathcal{F}\}, \qquad w' = w|_{U'} \tag{6.6}$$

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 6.25 (Correctness of the greedy matroid algorithm)**.** *The greedy algorithm presented in Algorithm 6.21 generates an optimal subset.*

Proof. By the contrapositive of Lemma 6.23, 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 6.22 guarantees that there is some optimal independent set containing x. Finally, Lemma 6.24 demonstrates that we can reduce to finding an optimal independent set on the contraction of M is sufficient. Its easy to see that Algorithm 6.21 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 6.26 (Runtime of the Greedy Matroid Algorithm)**.** *The runtime of Algorithm 6.21 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[46] 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. □

6.5 Kruskal's Algorithm

We introduced Kruskal's Algorithm already as Algorithm 6.16 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.

6.6 Metric Steiner Tree Problem

6.7 Huffman Codes

6.8 Prim's Algorithm

6.9 Clustering and Packing

6.10 Additional Example: Roadtrip

[47]

6.11 Additional Example: Coin Changing

[48]

6.12 Additional Caching: Offline Caching

[49]

[46] 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.
[47] Written by Celia Zhang.
[48] Written by Celia Zhang.
[49] Written by Celia Zhang.

7 Graph Algorithms

8 Branch and Bound

8.1 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.

8.2 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 8.1 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 8.2. 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.

8.3 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.

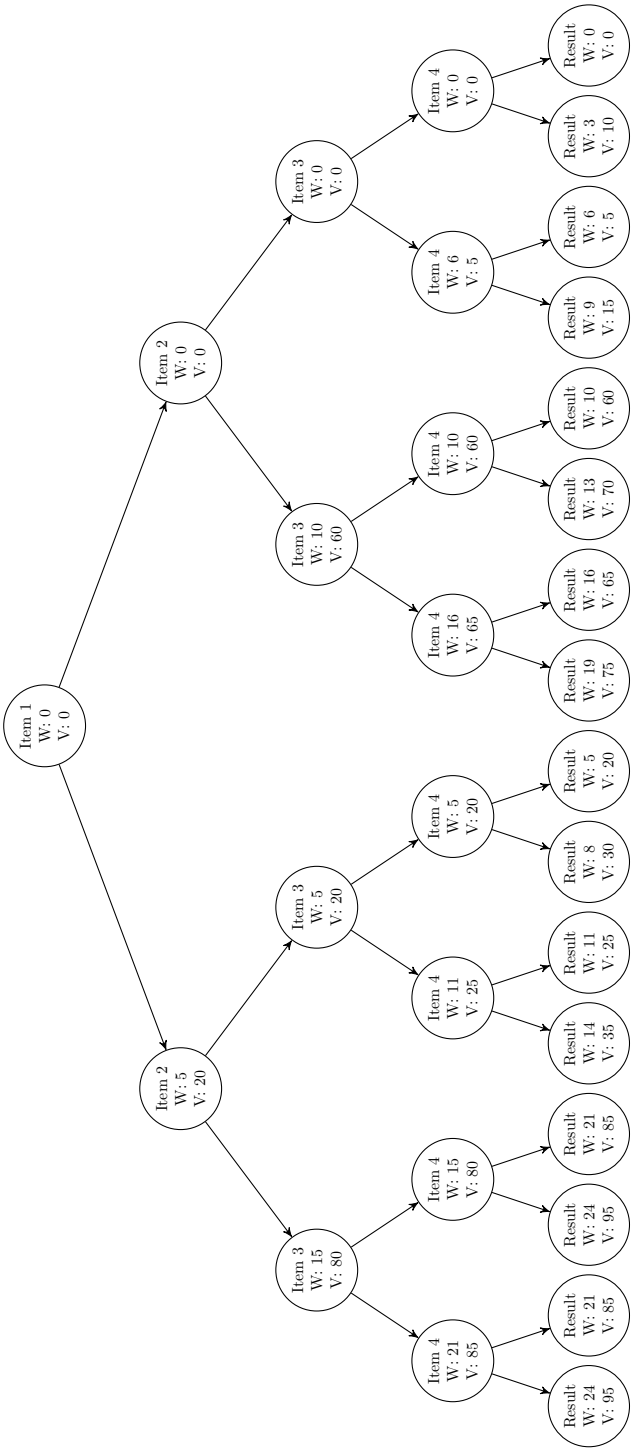

Figure 8.1: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 10. Here all paths are considered. Next figure shows the truncation we can do.

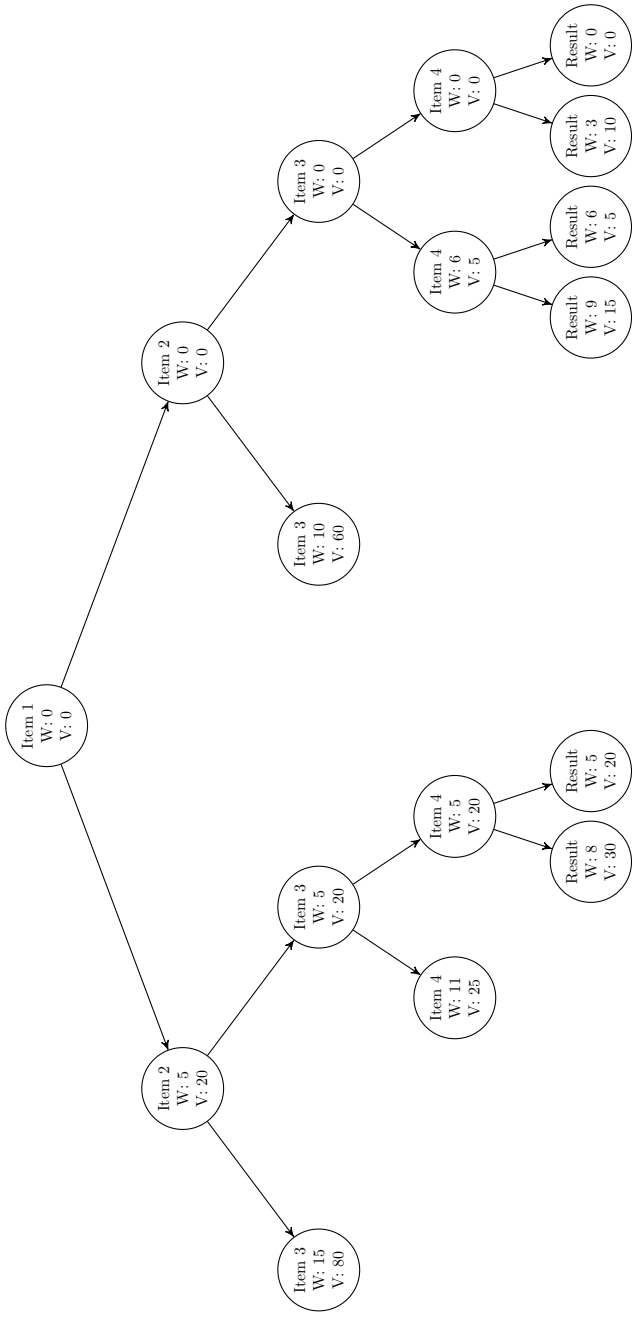

Figure 8.2: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 9. Here we truncate unfeasible paths early.

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.

8.3.1 Generic Algorithm

Algorithm 8.1 (Branch and Bound Algorithm)**.** For any problem,

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.

8.3.2 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

$$T = \left\{ \{0,1\}^k \;\middle|\; 0 \leq k \leq n \right\} \tag{8.1}$$

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:

$$f(\ell) = f(\ell_1 \dots \ell_n) = \mathbb{1}_{\{\ell_1 w_1 + \dots + \ell_n w_n \leq W\}} \cdot (\ell_1 v_1 + \dots \ell_n v_n) \tag{8.2}$$

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 $\ell_1 v_1 + \dots + \ell_n v_n$. Define the function $h : T \to \mathbb{R}$ as:

$$h(\mathbf{t}) = f(t_1 \dots t_k) = \begin{cases} \infty & \text{if } t_1 w_1 + \dots t_k w_k \leq W \\ 0 & \text{otherwise} \end{cases} \tag{8.3}$$

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 $t_1 w_1 + \dots + t_k w_k > W$. Any descendant ℓ of t will satisfy $\ell_1 = t_1, \dots, \ell_k = t_k$. Therefore,

$$(\ell_1 w_1 + \dots + \ell_k w_k) + (\ell_{k+1} w_{k+1} + \dots + \ell_n w_n) \geq t_1 w_1 + \dots t_k w_k > W \tag{8.4}$$

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.

8.4 Traveling Salesman Problem

9 Divide and Conquer

9.1 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.

9.2 Generic Algorithm Design

Algorithm 9.1 (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

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

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{9.1}$$

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

Theorem 9.2. *(Master Theorem) If $T(n)$ satisfies the above recurrence relation, then if*

- *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

$$T(n) = a^j T\left(\frac{n}{b^j}\right) + \sum_{k=0}^{j} a^k f\left(\frac{n}{b^k}\right) \tag{9.2}$$

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

$$T(n) = \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k f\left(\frac{n}{b^k}\right) \tag{9.3}$$

Let's consider the first case. We apply the relation to get[50]

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + a^{\log_b n} f(1) \sum_{k=0}^{\log_b n} c^{-k} \\ &= \Theta(a^{\log_b n}) + \Theta\left(a^{\log_b n} \frac{c}{c-1}\right) \\ &= \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \end{aligned} \tag{9.4}$$

For the second case,

$$\begin{aligned} T(n) &= \Theta(a^{\log_b n}) + \sum_{k=0}^{\log_b n} a^k \Theta\left(\left(\frac{n}{b^k}\right)^{\log_b a}\right) \\ &= \Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n} a^k \left(\frac{\Theta(n^{\log_b a})}{a^k}\right) \\ &= \Theta(n^{\log_b a} \log n) \end{aligned} \tag{9.5}$$

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

9.3 Quicksort

9.4 Lower bound on Sorting

9.5 Fast Integer Multiplication

9.6 Fast Division, Newton's Method

9.7 Convolution

9.8 Polynomial Multiplication

9.9 Fast Fourier Transform

9.10 Matrix Computations

9.11 Matrix Determinant and Inverse as hard as Multiplication

9.12 Strassen's Laser for Matrix Multiplication

9.13 Additional Examples: Closest Two Points

Problem Given a set $S = \{s_1, \ldots s_n\}$ where $s_i = (x_i, y_i)$ find the two closest points in euclidean distance.

Algorithm We consider a divide and conquer algorithm.[51] First sort the points by both x and y coordinate.This gives us two distinct permutations π, σ such that

$$x_{\pi(1)} \leq \ldots \leq x_{\pi(n)} \qquad y_{\sigma(1)} \leq \ldots \leq y_{\sigma(n)} \tag{9.6}$$

[50] 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.

[51] The way I've written an algorithm here is not the way you should write one in this course. I've interjected a lot of unnecessary details to guide the explanation.

Following the divide and conquer technique we calculate d the minimum distance of the following two recursive problems: $S_1 = \{s_{\pi(1)}, \ldots, s_{\pi(n/2)}\}$ and $S_2 = \{s_{\pi(n/2+1)}, \ldots, s_{\pi(n)}\}$. (Note these are presorted so we don't need to consider sorting them again.)

Now, we need to consider any point pairs that transcend this seperation. Recognize that we only care about points whose euclidean distance is less than d. Therefore, their x distance is also $< d$. So, we need to consider any point pairs s_i, s_j that have $x_i, x_j \in [x_{\pi(n/2)} - d, x_{\pi(n/2)} + d]$. Okay now let $T_1 \subseteq S_1$ be the subset $\{s_i : x_i \geq x_{\pi(n/2)} - d\}$ adn $T_2 \subseteq S_2$ be the subset $\{s_j : x_j \leq x_{\pi(n/2)} + d\}$. So we're interested in pairs from (T_1, T_2).

Naïvely, this will take $O(n^2)$ time because T_1 could equal S_1 and T_2 equal S_2. However, we know points are seperated by at least d on either side of $x_{\pi(n/2)}$. Therefore, there can only be one point per $(d/2) \times (d/2)$ square on either side. Therefore, for any point $s_i \in T_1$ we only need to consider the ten closest points in y distance to s_i from T_2.[52]

We've already sorted by y, so if we construct T_1 and T_2 in $O(n)$ time from the y sorted list, we can then find pairs that transcend the seperation in $10 \cdot n/2$ comparisons which is $O(n)$. By the Master Theorem, we get a runtime of $O(n \log n)$.

9.14 Additional Example: Convex Hull

[53]

9.15 Additional Example: Cartesian Sum of Sets

[54]

[52] If this is confusing, the Wikipedia page for this problem has a good picture.
[53] Written by Kevin Shu
[54] Written by Kevin Shu

10 Streaming Algorithms

10.1 Formalism

10.2 Additional Example: Uniform Sampling

11 Max-Flow Min-Cut

11.1 Flows and Capacitated Graphs

11.2 Theorem

11.3 Floyd-Fulkerson Algorithm

11.4 Edmonds-Karp Algorithm

12 Linear Programming