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. There is a lot of additional information that I have included in the footnotes; I highly encourage you to read them as some of them are tangential musings while others actual carry rather pertinent information to the subject.

A word of warning, however. I go into a lot of detail about the mathematics (as it interests me). I've tried to make it as accessible as possible by adding definitions for mathematical concepts as well as the intuition behind some of these definitions. However, if you are lost, Wikipedia is a good source for these definitions.

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 and second times looking at this course!

–Chinmay Nirkhe, cnirkhe@gmail.com

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]

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

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

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

(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.

(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 $yes \to yes$

[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.

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.

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]

[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.

(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 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]

[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.

[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

[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.

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 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 computation power, 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 distinction of P and NP.

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

[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.

[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.

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.

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.

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

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.

[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.

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

This simple fact is going to take us home. We've found a way to recursively reduce the larger of the two inputs (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 $|\phi'| < 1$ 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.

[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.

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 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 $\text{SUMS}(a, b)$ 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, $\text{SUMS}(a, b) = \text{SUMS}(a + kb, b)$ for any $k \in \mathbb{Z}$, and therefore $\text{SUMS}(a, b) = \text{SUMS}(a \pmod b, b)$.*

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

Theorem 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. □

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

[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 $a + b$.

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[27]. 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, an example

I'll introduce computing 'tribonacci' numbers as a preliminary example[28] 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

[27] 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.

[28] 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.

be called $\exp(O(k))$ many times.[29] To remedy this, we are going to write down a table of values. Here let's assume the word model again.

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.[30] 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 $O(k)$.[31]

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."[32] 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.

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.)

[29] 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.

[30] Not all proofs of correctness will be this easy, but not will be too much more complicated. State what each element of the table should equal and argue its correctness.

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

[32] 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.

5.4 k-Clustering, an example

Now might be a good time and go back to Chapter 2 and read the full Dynamic Programming Example. A classic Dynamic Programming problem is that of k-Clustering.

6 Greedy Algorithms

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.

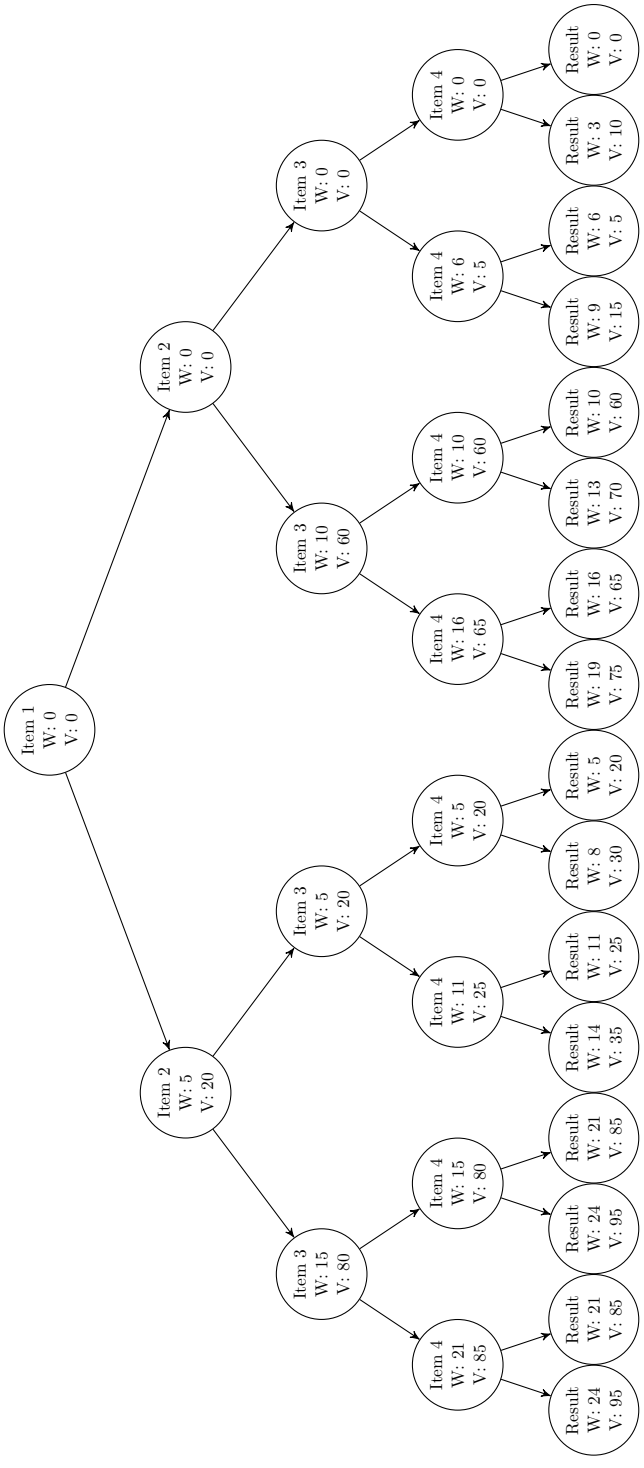

Figure 8.1: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 10. Here all paths are considered. Next figure shows the truncation we can do.

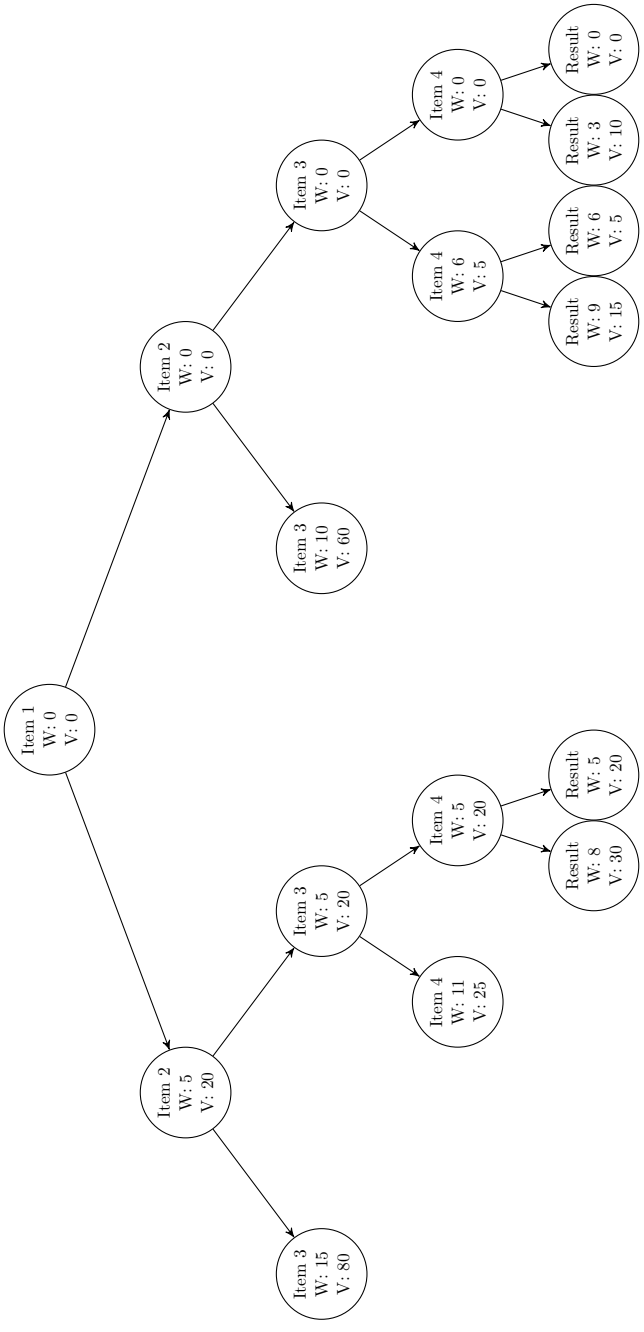

Figure 8.2: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 9. Here we truncate unfeasible paths early.

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.

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:

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

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

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

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

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

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.

9 Divide and Conquer

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.

9.1 Generic Algorithm Design

Algorithm

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 9.1. *(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[33]

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

[33] 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.

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. □

9.2 Closest Two Points, an example

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

Algorithm [34] We consider a divide and conquer algorithm. First sort the points by both x and y coordinate. This gives us two distinct permutations π, σ such that

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

Following the divide and conquer technique we calculate d the minimum distance of the following two recursive problems: $S_1 = \{s_{\pi(1)}, \dots, s_{\pi(n/2)}\}$ and $S_2 = \{s_{\pi(n/2+1)}, \dots, s_{\pi(n)}\}$. (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.[35]

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 $O(n \log n)$.

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

[35] If this is confusing, the Wikipedia page for this problem has a good picture.

10 Multiplicative Weights Algorithm

11 Max-Flow Min-Cut

12 Dynamic Programming