Algorithms Topics Review

Abhijit Chowdhary
Fundamental Algorithms
Prof. Siegel

December 26, 2018

> Beware of bugs in the below code: I have only proved it correct, not tried it.
>
> *Donald Knuth*

Contents

1 Induction

Induciton is an important tool in computer science, as it provides us one with an procedure to show correctness of our algorithms, and two can be directly translated to recursion.

1.1 Definition and Examples

Definition. Induction is a mathematical proof technique. Formally, suppose we have some program P, which has input n. Furthermore, suppose we can show that:

1) [Base Case] $P(0)$ is provably correct.

2) [Inductive Step] Suppose that $P(0), \ldots, P(k)$ is correct for some fixed $k \in \mathbb{N}$. Then we can show that $P(k+1)$ is provably correct.

Then it must be true that $\forall n, P(n)$ is correct. (Note that we don't need for *zero* that $P(0)$ is provably correct, but rather that $P(n_0)$ is provably correct, for some fixed n_0. We will typically take $n_0 = 0, 1$.)

Conceptually you can envision this as a staircase. Suppose that you can stand at the bottom of the staircase, and that if you're on step k, you can move to step $k+1$. Then we know that $\forall n$, we can stand on step n.

Take the following as applications of induction.

- Prove that $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$.

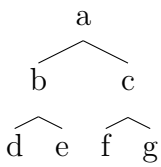
  Proof. Clearly for $n = 1$, we have that $1 = \frac{1(1+1)}{2} = 1$. Now suppose that $1+2+\cdots+n = \frac{n(n+1)}{2}$. We want to show that $1 + \cdots + n + 1 = \frac{(n+1)(n+2)}{2}$. Notice that:

  $$\sum_{i=1}^{n+1} i = n + 1 + \sum_{i=1}^{n} i = n + 1 + \frac{n(n+1)}{2}$$

  The last equality being true by our assumption. Now notice we can write $n(n+1)/2 = (n^2 + n)/2$, and we can write $n + 1 = \frac{2n+2}{2}$. Therefore adding the two expresions together results in $(n^2 + 3n + 2)/2 = (n + 1)(n + 2)/2$.

  □

- We call a **perfect binary tree** a tree structure such that each non-leaf vertex has exactly two children. So for example, a perfect binary tree of height 2 would be:

  Prove that a perfect binary tree of height n has exactly $2^n - 1$ vertices.

Proof. First suppose $n = 1$. Clearly a tree of height 1 has only one vertex, and indeed $2^1 - 1 = 1$.

Now suppose a tree of height n has $2^n - 1$ vertices. We want to show that a tree of height $n + 1$ must have $2^{n+1} - 1$ vertices. Consider the root of such a tree (in the example above it's vertex a). Notice that it's left subchild is a perfect binary tree of height n, and so is it's right subtree. Therefore, counting the verteces in each of those subtrees using our supposition that a tree of height n has $2^n - 1$ vertices gives us a count of $2(2^n - 1)$ for the vertices under the root. Therefore a tree of height $n+1$ has $2(2^n - 1) + 1$ vertices ($+1$ for the root). Notice that:

$$2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

and therefore we've shown our inductive step.

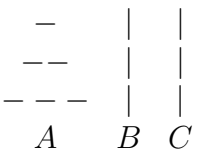
Thus, a perfect binary tree of height n has $2^n - 1$ vertices.

□

1.2 Towers of Hanoi and thinking recursively

The reason why we're reviewing induction, is that induction is very (read: ***VERY***) useful for proving correctness for recursive functions. You will use it in literally every single part of the rest of the course so it pays to get it down well. A classical example is Towers of Hanoi.

Definition. Towers of Hanoi is a mathematical puzzle. Suppose we have three rods, and a number of disks of different sizes $1 \to n$. which can slide onto any rod. The puzzle starts with the disks all arranged on the first rod in descending order such that n is at the bottom, and 1 is at the top. For example, for $n = 3$, the beginning configuration is:



The goal of this game is to move the entire stack of rings from pole A to pole B, while obeying the constraints:

- You may move one disk at a time with the command $Move(arg1, arg2)$ where this moves the disk on top of pole $arg1$ to the top of pole $arg2$.
- No larger disk may be placed on top of a smaller disk.

We want to write an algorithm to solve this puzzle, but to also do so with a minimal amount of $Move$ calls.

2 Asymptotics and Recurrences

2.1 Math Review

Before we talk about asymptotics and recurrences, we quickly review the logarithmic function log.

Definition. We define the **logarithm** to be the opposite of exponentiation, i.e. we say that $\log_b(x) = y$ exactly if $b^y = x$. For example $\log_2(64) = 6$, since $64 = 2^6$.

Here's a couple useful properties of the log that you will use in this section extensively.

(1) Product $\to$ Summation: $\log_b(xy) = \log_b(x) + \log_b(y)$.

Proof. By definition, we know that $b^{\log_b(xy)} = xy$. However, also notice that $x = b^{\log_b(x)}$ and $y = b^{\log_b(y)} \implies xy = b^{\log_b(x)} b^{\log_b(y)} = b^{\log_b(x)+\log_b(y)} \implies b^{\log_b(xy)} = b^{\log_b(x)+\log_b(y)}$.

□

(2) Quotient $\to$ Subtraction: $\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$.

Proof. This can be seen using the previous identity, and just taking $y \mapsto \frac{1}{y}$, and noticing that $\frac{1}{y} = \frac{1}{b^{\log_b(y)}} = b^{-\log_b(y)}$. □

(3) Power $\to$ Scalar Multiplication: $\log_b(x^p) = p \log_b(x)$.

Proof. By defintion:

$$x = b^{\log_b(x)} \implies x^p = \left(b^{\log_b(x)}\right)^p = b^{p \log_b(x)}$$

Therefore it must be true that $\log_b(x^p) = p \log_b(x)$.

□

(4) Change of Base: $\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$.

Proof. Starting from $x = b^{\log_b(x)}$, we can take $\log_k$ of both sides to recieve:

$$\log_k(x) = \log_k b^{\log_b(x)} = \log_b(x) \log_k(b)$$

Therefore, dividing through by $\log_k(b)$ gives us the desired identity.

□

2.2 Asymptotics Analysis

In the analysis of algorithms, one of our goals is to classify growth of the runtime and storage complexity of functions. To do so, it's in our interest to examine the limiting behavior of functions, i.e. when our input becomes large what does our function do. To this end we introduce Big O notation.

2.2.1 Big O notation

Big O notation exists to describe the limiting behavior of a function, and us Computer Scientists use it to classify algorithms in terms of their input size n. Due to this, from this point onwards all functions we define are non-negative.

We give both the informal and formal definition of a couple different useful characterizations. Let $f, g : \mathbb{R} \to \mathbb{R}$ be two functions.

- Upper bound: $\mathcal{O}$.

  Definition. Informally, we say that $f(x) = \mathcal{O}(g(x))$ if $f(x) \leq cg(x)$, c some constant, for sufficiently large x.

  Formally, $f(x) = \mathcal{O}(g(x))$ iff:

  $$\limsup_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| < \infty$$

  For example, $n^2 = \mathcal{O}(n^2)$ and $n^2 = \mathcal{O}(n^5)$ but $n^2 \neq \mathcal{O}(n)$. To highlight the sufficiently large x portion, notice that it is true that $n^2 = \mathcal{O}(n^3 - n^2)$ $(n^3 - n^2 > n^2, \forall n > 2.)$

- Lower Bound: Ω.

  Definition. Informally, we say that $f(x) = \Omega(g(x))$ if and only if $g(x) = \mathcal{O}(f(x))$, or that $f(x) \geq cg(x)$.

  Formally $f(x) = \Omega(g(x)) \iff \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0$.

- Tight bound: θ.

  Definition. We say that $f(x) = \theta(g(x))$ if $f(x) = \mathcal{O}(g(x))$ and $f(x) = \Omega(g(x))$. Another way to say this is that $\exists c \in \mathbb{R}$ such that $\frac{1}{c}g(x) \leq f(x) \leq cg(x)$, for sufficiently large enough x.

  Formally we must have that:

  $$\limsup_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| \in \mathbb{R}_{>0}$$

  To those that care, notice that having $f(x) = \mathcal{O}(g(x))$ and $\Omega(g(x)) \implies \lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right|$ exists. Instead lim inf and lim sup may converge to different values.

  So for example, $n^2 = \theta(n^2)$ but $n^2 \neq \theta(n)$ and $\neq \theta(n^3)$.

2.2.2 Properties of Big O

There's a couple of useful properties of Big O that are good to know. Let $f_1 = \theta(g_1)$ and $f_2 = \theta(g_2)$.

(1) Product: $f_1 f_2 = \theta(g_1 g_2)$. In addition $f\theta(g) = \theta(fg)$.

(2) Sum: $f_1 + f_2 = \theta(g_1 + g_2)$. In particular if $f_1 = \mathcal{O}(g)$ and $f_2 = \mathcal{O}(g)$ then $f_1 + f_2 = \mathcal{O}(g)$.

(3) Scalar multiplication: $\theta(kg) = \theta(g)$ supposing $k \neq 0$. In addition $kf_1 = \theta(g_1)$.

(4) Summation $\to$ maximization.: $\theta(g_1 + g_2) = \theta(\max(g_1, g_2))$.

2.2.3 Time Complexity Comparison

Typically for problems where we want to compare which function grows faster asymptotically, we really want to see at large x, which one dominates. Our main strategy to do this, is to reduce these functions down to functions which are more easily compared, which I call the time complexity classes. See the time complexity chart in the figures section for a good table of a commmon few which you should remember.

Take for examples:

(1) 2^n versus n^2.

Solution. It's clear that 2^n is exponential, which grows faster than n^2 which is polynomial.

□

(2) $n^{\log(n)}$ versus n^{100}.

Solution. The left hand function is somewhere above polynomial due to the power being an increasing function of n. Therefore, it grows faster than n^{100}.

□

(3) $n^{1/\log_2(n)}$ versus 2.

These are strange to compare, since we aren't sure what class the left hand function falls in. To try to fit it to something we understand, we remember that the logarithm takes powers to scalar multiplcation, so then we notice:

$$\log_2(n^{1/\log_2(n)}) = \frac{1}{\log_2(n)} \log_2(n) = 1$$

But also $\log_2(2) = 1$. Therefore we find that these functions are actually equal in terms of asymptotic growth.

2.3 Algorithm → Recurrence

Here we want to examine how to turn recursive code into reucurrences characterizing their properties. We do this by example.

2.3.1 Fibbonacci Sequence

```
1: procedure F(n)                                   ▷ a_1 = 1, a_2 = 1, a_n = a_{n-1} + a_{n-2}
2:     if n ≤ 2 then
3:         Return 1
4:     else
5:         Return F(n − 1) + F(n − 2)
6:     end if
7: end procedure
```

Suppose we want to characterize the runtime of the above algorithm using a recurrence equation. The idea is that we count the work per level, and add it. Letting $G(n)$ characterize the runtime of $F(n)$.

```
1: procedure F(n)
2:     if n ≤ 2 then                                ▷ θ(1)
3:         Return 1                                 ▷ θ(1)
4:     else                                         ▷ θ(1)
5:         Return F(n − 1) + F(n − 2)               ▷ G(n − 1) + G(n − 2) + θ(1)
6:     end if                                       ▷ θ(1)
7: end procedure
```

Notice that $\theta(1)$ summed a constant number of times is still $\theta(1)$, therefore we have:

$$G(n) = \begin{cases} \theta(1) & n \leq 2 \\ \theta(1) + G(n-1) + G(n-2) & \text{otherwise} \end{cases}$$

For characterizing things that aren't work, i.e. how many times the -1 operation happens in the above code, we can do a similar process. Let $H(n)$ characterize the number of -1 operations in $F(n)$. Then:

```
1: procedure F(n)
2:     if n ≤ 2 then                                ▷ 0
3:         Return 1                                 ▷ 0
4:     else                                         ▷ 0
5:         Return F(n − 1) + F(n − 2)               ▷ H(n − 1) + H(n − 2) + 1
6:     end if                                       ▷ 0
7: end procedure
```

Therefore:

$$H(n) = \begin{cases} 0 & n \leq 2 \\ 1 + H(n-1) + H(n-2) & \text{otherwise} \end{cases}$$

2.3.2 Merge Sort

Let $F(n)$ characterize the work of Merge Sort on an array of length n. Then:

```
1: procedure MERGESORT(A : Array)
2:     Let n = len(A).
3:     if n = 1 then                                              ▷ θ(1)
4:         Return A                                               ▷ θ(1)
5:     end if                                                     ▷ θ(1)
6:     B ← A[1...⌊n/2⌋]                                           ▷ θ(1)
7:     C ← A[⌈n/2⌉...n]                                           ▷ θ(1)
8:     Mergesort(B), Mergesort(C)                                 ▷ F(⌊n/2⌋) + F(⌈n/2⌉)
9:     Return Merge(B, C)                                         ▷ θ(n)
10: end procedure
```

Therefore we have:

$$F(n) = \begin{cases} \theta(1) & n = 1 \\ \theta(n) + F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

Similarily we can do this for $H(n)$ characterizing storage constraints, but I'll leave that as exercise.

2.3.3 Randomized Algorithms

A classic Siegel question is to characterize various facets of the randomized algorithm:

```
1: procedure RAND(n)
2:     if n ≤ 1 then
3:         Return 0
4:     end if
5:     Let x ← 1, 2 or 3 with probabilities 1/2, 1/3, 1/6 respectively.
6:     if x = 1 then
7:         Return 2Rand(n)
8:     else if x = 2 then
9:         Return 7Rand(n − 1) + 12Rand(n − 2)
10:    else
11:        Return nRand(n − 1)
12:    end if
13: end procedure
```

Define function F, G, H which characterize runtime, exact value returned, and number of times $x \leftarrow 3$ respectively. The trick to characterizing the above, is simply to take whatever happens inside of an if statement and multiply it by the probability of the event occuring. So:

$$F(n) = \begin{cases} \theta(1) & n \leq 1 \\ \theta(1) + \frac{1}{2}(\theta(1) + F(n)) + \frac{1}{3}(\theta(1) + F(n-1) + F(n-2)) + \frac{1}{6}(\theta(1) + F(n-1)) & \text{otherwise} \end{cases}$$

$$G(n) = \begin{cases} 0 & n \leq 1 \\ \frac{1}{2}(2H(n)) + \frac{1}{3}(7H(n-1) + 12H(n-2)) + \frac{1}{6}(nH(n-1)) & \text{otherwise} \end{cases}$$

$$H(n) = \begin{cases} 0 & n \leq 1 \\ \frac{1}{2}(0 + H(n)) + \frac{1}{3}(0 + H(n-1) + H(n-2)) + \frac{1}{6}(1 + H(n-1)) & \text{otherwise} \end{cases}$$

This always appears on the exam.

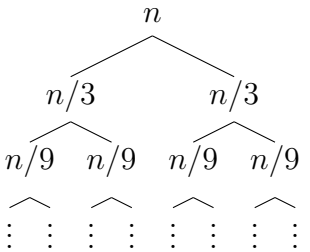
2.4 Solving Recurrences

Ok, so now that we know how to characterize certain aspects of recursive code using recurrences, one might ask, how do we solve recurrences? The idea stems behind something called the tree method. For example, consider the recurrence equation:

$$f(n) = \begin{cases} 1 & n = 1 \\ n + 2f(n/3) & \text{otherwise} \end{cases}$$

Here we call n the work function, 3 the decay rate, and 2 the branching factor.

What we do to solve this is *unroll* the recursive term over and over again. We represent each unrolling as a new level of the tree, so it may look like:



The top level looks like $n + 2f(n/3)$, but then we unroll $f(n/3)$ to $n/3 + 2(n/9)$, which is our second level. To then figure out what our summation totals to, we sum accross the level, and then downward. In the above example, the first level totals to n, the second to $(2/3)n$, the third to $(4/9)n$, etc. In fact, we will find that in general the jth level will sum to $\frac{2^j}{3^j}n$; this is guarenteed by the unrolling. But what about the leaf level?

In general, we have to examine the leaf level seperately, because the recurrence defines something different for it. Here, our recurrence says that each leaf has cost 1. Looking at our tree, we see that at each level the number of vertices doubles (powers of branching factor), therefore at the leaf level k we must have 2^k leaves. Each costs 1, therefore the contribution from the leaf level is 2^k.

Therefore our final total actually looks something like $n + \frac{2}{3}n + \frac{2^2}{3^n}n + \cdots + \frac{2^{k-1}}{3^{k-1}}n + 2^k$, for some stopping point k. Let's find k.

k is determined by the point we can *unroll* no longer, i.e. the decay rate has been applied sufficiently enough times such that we get down to our base case, 1. In other words, we want to solve for k where:

$$\frac{n}{3^k} = 1 \implies n = 3^k \implies k = \log_3(n)$$

Since we choose n a power of 3, we find that k is an integer like we wanted it to be. Then, we can write our final answer as:

$$f(n) = n\left(1 + \frac{2}{3} + \frac{2^2}{3} + \cdots + \left(\frac{2}{3}\right)^{\log_3(n)-1}\right) + 2^{\log_3(n)}$$

We make one note here about the leaf level. Notice that we can say:

$$2^{\log_3(n)} = \frac{n}{n}2^{\log_3(n)} = n\frac{2^{\log_3(n)}}{3^{\log_3(n)}} = n\left(\frac{2}{3}\right)^{\log_3(n)}$$

This looks exactly like our pattern! What's happening? What we've discovered is that our leaf level *obeys* our recurrence, and I claim that this is in general true if the work at the level matches the work function. For example, the work function $w(n)$ is n here. The base case is at $n = 1$, therefore looking at $w(1)$ we see it's 1. *This matches our base case*, i.e. that at $n = 1, f(n) = 1$. Whenever this happens, it's unecessary to compute the leaf level seperately, and we can pull it into our summation. Rough reasoning is that it's modeled perfectly by the recurrence. If we had it that when $n = 1, f(n) = 5$, we could not do this. Therefore we can write:

$$f(n) = n\left(1 + \frac{2}{3} + \frac{2^2}{3} + \cdots + \left(\frac{2}{3}\right)^{\log_3(n)}\right)$$

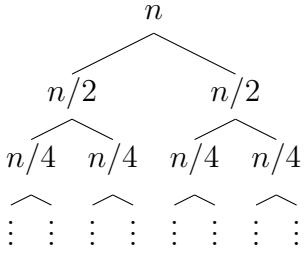
Let's solve a couple different types of common recurrences you might see.

2.4.1 Basic Recurrences

(1) Solve the recurrence:

$$A(n) = \begin{cases} 1 & n = 1 \\ n + 2A(n/2) & \text{otherwise} \end{cases}$$

Solution. For this we have work function $f(n) = n$, decay rate 2 and branching factor 2. The tree we draw looks like:



Summing across, we come across the surprising fact that each level sums to n. This is roughly because the decay rate and the branching factor cancel each other out. In general, we find this to be true when $f(\delta(n)) = \beta$ where β is the branching factor and δ is the decay rate as a function of n (can you see why?). Ok, so now considering the tree, we can ask how many terms are there? Recall this is solving for $\frac{n}{2^k} = 1 \implies k = \log_2(n)$. This tells us that there must be $\log_2(n)+1$ levels (k tells us kth level, but count starting from 0). Finally we notice that $f(1) = A(1)$, which implies that we don't have to do the leaf level seperately. Therefore our solution is:

$$A(n) = n + n + \cdots + n = n(\text{Number of levels}) = n(log_2(n) + 1)$$

□

(2) Solve the reucrrence:

$$B(n) = \begin{cases} 5, & n = 1 \\ n + 2B(n/2), & \text{otherwise} \end{cases}$$

Solution. The difference between this one at the last one is the leaf level. Here we have that $B(1) \neq f(1)$, which implies we need to compute the leaf level seperately. There are $2^{\log_2(n)} = n$ leaves, and each has work 5, which implies that the leaf level cost is $5n$. Therefore we can write our answer as:

$$B(n) = nlog_2(n) + 5n$$

□

2.4.2 Changing the Leaf Level

What if we have a recurrence like:

$$C(n) = \begin{cases} 8, & n = 8 \\ n + 3C(n/2), \text{otherwise} \end{cases}$$

In this case what's strange is that we've changed when we enter the leaf level. The tree method, however, handles this perfectly. Before we asked when $\frac{n}{b^k} = 1$ where b was the decay rate. Here the base case is when $n = 8$, so we instead ask $\frac{n}{2^k} = 8 = 2^3 \implies k = \log_2(n) - 3$. Then the problem is solved in the same exact manner as in the basic recurrences.

2.4.3 Multiple Decay Rates

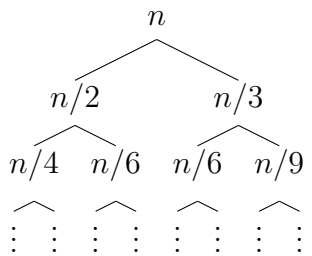
What happens if we have a recurrence like:

$$D(n) = \begin{cases} 1, & n = 1 \\ n + D(n/2) + D(n/3), & \text{otherwise} \end{cases}$$

What we lose is the guarentee that all leaves live on the same level. In fact, the end of the *unrolling* will come sooner for the harsher decay rate, and later for the kinder decay rate. In $D(n)$, we see that $\frac{n}{2^k} \to_k 1$ slower than $\frac{n}{2^k} \to_k 1$. To resolve this problem, what we do is approximate; we say that our leaf level k lives somewhere between the leaf level defined by 2 and 3.

If we follow the decay rate 2 all of the way down, we know that our leaf level is $k_2 = \log_2(n)$. If we follow 3, similarily we will recieve $k_3 = \log_3(n)$. We say that our "leaf level" lives between these two, or $\log_3(n) \leq k \leq \log_2(n)$.

Then to solve our problem we have the tree:



We find that our summation looks like:

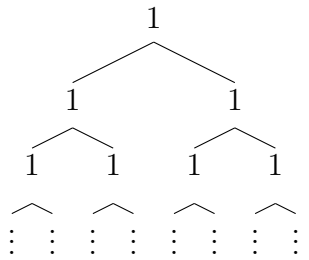
$$D(n) \approx n \left[1 + \frac{5}{6} + \frac{5^2}{6^2} + \cdots + \left(\frac{5}{6}\right)^k \right], \quad \log_3(n) \leq k \leq \log_2(n)$$

2.4.4 Strange Decay Rates

What happens if we have a recurrence like:

$$E(n) = \begin{cases} 1, & n = 1 \\ 1 + 2E(\delta(n)), & \text{otherwise} \end{cases}$$

Lots of people get stuck on this, but I claim that our tree method handles this perfectly. The tree itself looks like:



The real question is, where is the leaf level? Before we asked for decay rate $\delta(n) = \frac{n}{b}$, how many times do we have to apply it to get down to the base case. This formulated as $\frac{n}{b^k} = 1 \implies k = \log_b(n)$. Our question in general is for what k does $\delta^k(n) = 1$, where k isn't a power but repeated function compositions. For example if $\delta(n) = n - 1$, then we would be asking $n - k(-1) = 1 \implies k = n - 1$.

3 Dynamic Programming

Dynamic programming is a technique from optimization field, specally it's a method to solve problem if the problem exhibits **optimal substructure** and **overlapping subproblems**.

Definition. A problem is said to have **optimal substructure** if it can be constructed from optimal solutions of its subproblems.

For an example, suppose T a tree and consider the problem where $\forall v \in T$, you want to compute a field min such that $\min[v] = \min\{w.val : w \in ST(v)\}$ where $ST(v)$ denotes the subtree rooted by v. This problem exhibits optimal substructure, as for arbitrary vertex v, you have $\min[w]$ computed for all $w \in Children[v]$, then you can construct $\min[v]$ as the minimum of your childrens answers and your own.

Definition. A problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times.

For an example, consider the Fibonnacci number sequence $a_n = a_{n-1} + a_{n-2}, a_2 = a_1 = 1$. Consider the value a_5. Notice that we can express it as a graph (see figure 2) with many incoming edges (in particular a DAG). Whereas problems with optimal substructure but not overlapping subproblems look like the problem $\min[v]$ problem, and can be modeled by trees.

The idea behind the dynamic programming problems we deal with is that if we remember problems as we do it, we can elimate the "overlapping" problems and reduce our computational cost as much as possible.

We've discussed three problems that we call "representative" of a lot of dynamic programming techniques: GCD, Woodcut, and Matrix Chain. We find that many dynamic programming problems reduce down to similar techniques needed to solve these. In this section we discuss how to tackle these problems fully, but before that we introduce **memoization**.

3.1 Memoization through Fibbonacci

As discussed above, the Fibbonacci sequence $F_n = F_{n-1} + F_{n-2}, F_2 = F_1 = 1$, is a prime example of a dynamic programming problem, since it has both optimal substructure and

overlapping subproblems. Using the recurrence relationship defined above we can write recursive code to solve this problem:

```
1: procedure F(n)                                  ▷ a_1 = 1, a_2 = 1, a_n = a_{n-1} + a_{n-2}
2:     if n ≤ 2 then
3:         Return 1
4:     else
5:         Return F(n − 1) + F(n − 2)
6:     end if
7: end procedure
```

However, if we were to run just this we would find ourselves choked by the runtime cost almost immediately. This is a consequence of the overlapping subproblems portion of Fibbonacci, i.e. we're repeatedly computing the same pieces of data. How do we resolve this? We implement **memoization**.

Definition. Memoization is an optimization technique where we store and cache the results of expensive function calls. The idea is that if we want to compute said function again, we look in our cache first. (Memoization ← making a memo.)

Let's take a look at this implemented in the Fibbonacci sequence.

```
1: Let Memo[1..n] be a integer valued array initialized to −1.
2: procedure F(n)                                  ▷ Recursive or top down
3:     if n ≤ 2 then
4:         Return 1.
5:     else if Memo[n] ≠ −1 then                   ▷ If we have computed n, use that result
6:         Return Memo[n].
7:     else                                        ▷ Otherwise, compute it, store it, and return it.
8:         Memo[n] = F(n − 1) + F(n − 2).
9:         Return Memo[n].
10:    end if
11: end procedure
```

We can also implement this iteratively using a lookup table:

```
1: procedure F(n)                                  ▷ Iterative or bottom up
2:     Let Memo[1..n] be a integer valued array.
3:     Let Memo[1], Memo[2] = 1.
4:     for i = 3 → n do
5:         Memo[i] = Memo[i − 1] + Memo[i − 2]
6:     end for
7: end procedure
```

Without the memoization table, our runtime was exponential, but what about now? It's clear by the iterative version that our usage of memoization brings us to $\theta(n)$ work, which is basically a night and day difference. In general we can examing the new runtime of our dynamic programming problem by looking at the memoization table, and trying to figure out how long it would take to fill it.

3.2 Greatest Common Subsequence

The problem is as follows: Suppose you are given two sequence A and B. Your objective is to find a subsequence $S \subseteq A, B$ such that the length of S is maximized. We want to write code to solve for $len(S)$. This is a classic problem for applications in revision control, the diff utility on your computer, and many more NLP like reasons.

So how do we solve this problem using dynamic programming? Well first we try to fit it to the constraints of optimal substructure and overlapping subproblems. To that end, consider placing points i, j at the end of A, B:

$$A = a_1 a_2 a_3 \dots a_n \leftarrow i$$
$$B = b_1 b_2 b_3 \dots b_m \leftarrow j$$

Consider the local problem of whether a_i and b_j should be considered in our greatest subsequence:

(1) Suppose $a_i = b_j$. Then we know that the element $a_i \in A$ and $a_i \in B$, therefore being a candidate to go in S. Do we ever have reason to exclude something from S? In this problem no! Therefore we decide to add 1 to our length. But then how can we continue processing after selecting a_i and b_j? Well we still need to select the best subsequence from $A[1..i-1]$ and $B[1..j-1$. This sounds like a subproblem with the last character of each sequence excluded!

(2) Suppose $a_i \neq b_j$. Here's where things get a little bit more complicated. Before when they were equal, we could conclude that we can take both, but here we can't just say skip past this element. Take for example the strings:

$$A = aaaab, B = bbbba$$

Considering the last element, if we choose to ignore them because they're not equal, we reduce our problem to $aaaa, bbbb$, which has no GCS. However, clearly $\{a\}$ or $\{b\}$ solves our problem.

Instead what we decide to do is not blindly skip past a_i or b_j. We say instead that we skip past *either* a_i *or* b_j. How do we decide which one? The one which produces the longer subsequence.

Using the above analysis we then conclude that a solution to the GCS problem is:

$$GCS(i,j) = \begin{cases} 0, & ij = 0 \\ 1 + GCS(i-1, j-1), & A[i] = A[j] \\ \max(GCS(i-1,j), GCS(i,j-1)), & \text{otherwise} \end{cases}$$

where intially $i = n, j = m$.

```
1: Let A[1..n] and B[1..m] be two character arrays.
2: procedure GCS(i, j)
3:     if ij = 0 then
4:         Return 0.
5:     else if A[i] = B[j] then
6:         Return 1 + GCS(i − 1, j − 1).
7:     else
8:         Return max(GCS(i − 1, j), GCS(i, j − 1)).
9:     end if
10: end procedure
```

```
1: Let A[1..n] and B[1..m] be two character arrays.
2: Let Memo[1..n, 1..m] be a 2d array initialized to −1.
3: procedure GCS(i, j)
4:     if ij = 0 then
5:         Return 0.
6:     else if Memo[i, j] ≠ −1 then
7:         Return Memo[i, j].
8:     else if A[i] = B[j] then
9:         Memo[i, j] = 1 + GCS(i − 1, j − 1).
10:    else
11:        Memo[i, j] = max(GCS(i − 1, j), GCS(i, j − 1)).
12:    end if
13:    Return Memo[i, j].
14: end procedure
```

```
1: Let A[1..n] and B[1..m] be two character arrays.
2: Let Memo[1..n, 1..m] be a 2d array initialized to −1.
3: Let Decisions[1..n, 1..m] be a 2d array.
4: procedure GCS(i, j)
5:     if ij = 0 then
6:         Return 0.
7:     else if Memo[i, j] ≠ −1 then
8:         Return Memo[i, j].
9:     else if A[i] = B[j] then
10:        Memo[i, j] = 1 + GCS(i − 1, j − 1).
11:        Decisions[i, j] = (i − 1, j − 1).
12:    else
13:        v1, v2 = GCS(i − 1, j), GCS(i, j − 1).
14:        if v1 > v2 then
15:            Memo[i, j] = v1.
16:            Decisions[i, j] = (i − 1, j).
17:        else
18:            Memo[i, j] = v2.
19:            Decisions[i, j] = (i, j − 1).
20:        end if
```

21: **end if**
22: Return $Memo[i, j]$.
23: **end procedure**

4 Figures

4.1 Common Time Complexities

Notation	Name	Example
$O(1)$	constant	Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table
$O(\log \log n)$	double logarithmic	Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap
$O((\log n)^c)$ $c>1$	polylogarithmic	Matrix chain ordering can be solved in polylogarithmic time on a parallel random-access machine.
$O(n^c)$ $0<c<1$	fractional power	Searching in a k-d tree
$O(n)$	linear	Finding an item in an unsorted list or in an unsorted array; adding two *n*-bit integers by ripple carry
$O(n \log^* n)$	n log-star n	Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a fast Fourier transform; Fastest possible comparison sort; heapsort and merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; simple sorting algorithms, such as bubble sort, selection sort and insertion sort; (worst case) bound on some usually faster sorting algorithms such as quicksort, Shellsort, and tree sort
$O(n^c)$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs; finding the determinant with LU decomposition
$L_n[\alpha, c] = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$ $0<\alpha<1$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n)$ $c>1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the travelling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with Laplace expansion; enumerating all partitions of a set

Figure 1: Don't worry about polyarithmitic and L notation. (src. Wikipedia)

4.2 Fibbonacci Overlapping Subproblems

Figure 2: Fibbonacci has overlapping subproblems