3 Introductory Topics

3.1 Recursion

Definition 3.1 (Recursive Algorithm)**.** A recursive algorithm is any algorithm whose answer is dependent on running the algorithm with 'simpler' values, except for the 'simplest' values for which the value is known trivially.[1]

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

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

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

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

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

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

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

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

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

Let's make a stronger statement. Recall that one way to think about $a \pmod b$ is the unique number in $\{0, \ldots, b-1\}$ that is equal to $a + kb$ for some $k \in \mathbb{Z}$.[2] Therefore, the following corollary also holds.

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

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

Algorithm 3.6 (Euclid-Lamé)**.** Given positive integer inputs a, b with $a \geq b$, if $b = 0$ then return a. Otherwise, return the $\gcd(b, a \pmod b)$ calculated recursively.[3]

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Duality

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

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

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

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

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

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

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

Proof. By Lemma **??**, we can restrict ourselves to positive (a, b). It's easy to see as $\gcd(a, b)$ divides a and b then it divides any $ax + yb$. Therefore $\gcd(a, b) \leq$ every element of $\mathrm{SUMS}(a, b)$ proving weak duality. To prove strong duality, let (a, b) be the pair such that $a + b$ is the smallest and $\min\{\mathrm{SUMS}(a, b)\} < \gcd(a, b)$.[4] But then $\mathrm{SUMS}(b, a - b) = \mathrm{SUMS}(a, b)$ by Lemma **??** and however, $b + (a - b) = b < a + b$, contradicting the assumed minimality of $a + b$. □

3.3 Repeated Squaring Trick

How many multiplications does it take to calculate x^n for some x and positive integer n? Well naïvely, we can start by calculating $x, x^2, x^3, \dots x^n$ 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, $x, x^2, x^{2^2}, \dots, x^{2^m}$ by calculating $x^{2^k} \leftarrow x^{2^{k-1}} \cdot x^{2^{k-1}}$. This is $O(m) = O(\log n)$ multiplications and costs $O(1)$ space as we only store the value of a single power of x at a time.

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

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

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

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

Therefore, using the powers we have written into memory, in an additional $O(\log n)$ multiplications we can calculate any power x^n. So any power x^n can be calculated using $O(\log n)$ multiplications and $O(\log n)$ space.[5]

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