CS 2800: Discrete Math Sept. 21, 2011

Lecture

Lecturer: John Hopcroft *Scribe: June Andrews*

Review

Last class we covered several applications:

Hashing

Let us say our application is to store the netids of students going to a concert. We could store the netids in an array. Now to check whether or not Emily went to the concert we have to check every cell of the array and see if that cell has Emily's netid. This could take a very long time. What's the worst case?

Instead of searching the array from left to right, let us consider hashing. In accessing elements of an array, it takes the same amount of time to access the first cell as it takes to access 10^{th}, 100^{th}, or even last cell. So if we have a function that tells us: if Emily's netid is in the array, it is stored in the y^{th} cell. Now by checking if Emily's netid is in the y^{th} cell, we can check in constant time, whether or not Emily attended the concert. This function is the hash function.

Given a netid, i, the hash function, H, will map a netid to a cell of the array, $H(i)$. Now to store or find i, we can access $H(i)$ in constant time.

The tricky part comes when for two different netids, $i \neq j$, map to the same cell, ie, $H(i) = H(j)$. This is known as a collision. Unfortunately, due to the *birthday problem* hash collisions are likely. So our implementation of hashing and arrays must include collision resolution. Ie, if two netids map to the same cell, we can't store them both in the same cell. How about the cell next to the one we mapped to? This is known as linear probing. When inserting a netid, i, we use $H(i)$ to find the first cell we try to store i in. If cell $H(i)$ is full, we move to the cell $H(i)+1$, if that cell is empty we store i there. If $H(i)+1$ is not empty we continue marching along the array until we find an empty cell to store i. Looking up i in the array is done in the same manner. We start looking at $H(i)$, if $H(i)$ is full, but does not contain i we keep checking cells until we either find i or find an empty cell, in which case we know i has not been stored in the array.

A small note: Unlike storing in a linked list, we can not tack new netids onto the end of the hash array. The values are inserted into the array, hence to insert new netids at least a majority of the array has to be empty. As the array fills up, we may have to resize. A resize is when we create a new array that is much larger, pick a new hash function and insert all values from our older smaller array into the new array using the new hash function.

When using hash functions, you want to know

Modular Arithmetic

Above we took the hashing function H for granted. We can construct a simple hashing function using modular arithmetic. Assume we are given 100 netids in the range of 0 to 1991249 and want to store them in an array of length 500. we can use the hash function $H(i) = i \mod 500$. Since i can be very large we want fast ways to compute modular arithmetic.

Handy modular arithmetic tricks exist! If we are only interested in the modular arithmetic of integers and if:

$$a_1 \equiv b_1 \pmod{n}$$
$$a_2 \equiv b_2 \pmod{n},$$

then:

$$\begin{aligned} a_1 + a_2 &\equiv b_1 + b_2 \pmod{n} \\ a_1 - a_2 &\equiv b_1 - b_2 \pmod{n} \\ a_1 a_2 &\equiv b_1 b_2 \pmod{n}. \end{aligned}$$

So let us compute $7787 \pmod 7$:

$$\begin{aligned} 7787 \pmod 7 &= 7 * 10^3 + 7 * 10^2 + 8 * 10 + 7 \pmod 7 \\ &= 0 + 0 + 3 + 0 \pmod 7 \\ &= 3 \pmod 7 \end{aligned}$$

Note: by using these arithmetic tricks we can prove that for mod 9 and mod 3, you can take the module of the sums of the digits of the numbers. Ie, $451 \pmod 9 = 10 \pmod 9 = 1 \pmod 9$. Pretty cool.

Another note, when using modular arithmetic we use equivalency, $\equiv$, rather than $=$. We know 7787 does not equal 3. But, if we're using modular arithmetic, in 7, then they behave the same when we add them to other numbers $7787 + x \pmod 7 = 3 + x \pmod 7$. So, under mod 7 arithmetic, 7787 and 3 are equivalent.

Random Sequences

If a sequence is random then no part of the sequence should be a simple variation of another part of the sequence. If I gave you '123456' as the first part of the sequence, but then gave you '23456' as the next part of the sequence, that is not very random - I've already seen it! Hence, if you have a random sequence and you run it through a compression algorithm (try to zip it for example), your zip file will be just as big as your random sequence. So we use the definition that a random sequence is any sequence you can not describe with any code that is shorter than the original sequence.

But computer codes are only so long. So with computer codes we can only describe short random sequences, and you'd have to have a different code for each random sequence. Inplace of random sequences, we use pseudorandom sequence generators.

Encryption

Suppose Alice wants to send banker Bob a message to pay her credit cards. Now, if eavesdropper, Eve, can intercept Alice's message and send Bob a message for Alice to pay Eve, Alice would be very unhappy. So Alice wants to send Bob a message that Eve can not read or mimic. This is the motivation for RSA. (which until late was completely unhackable - but it is still the best out there, and we can still use it with caution.)

Primes

The foundation to RSA is the use of prime numbers. So before we can understand how magical RSA is, let us understand some properties of prime numbers.

Theorem 1. *There are an infinite number of primes.*

Proof. We present a proof by contradiciton.

If the set of all primes is finite, then we can list all the primes in increasing order, $2, 3, 5, 7, 11, 13, 17, \ldots, p_n$, where p_n is the largest prime. Let $p' = 2 * 3 * 5 * \cdots * p_n + 1$. In fact, $p' > p_n$ is a new prime not in our list. For every prime p_i, $\frac{p'}{p_i} = 2 * 3 * \cdots * p_{i-1} * p_{i+1} * \cdots * p_n + \frac{1}{p_i}$, which is not an integer. Hence, no smaller prime, divides p'. If no primes divide p', then no composites divide p', and we can conclude that p' is not divisible by any other numbers than itself and 1. Hence, p' is a new prime and the set of primes can not be listed in a finite list. Hence, the set of primes is infinite. □

So there are infinite number of primes. But where are they?

Theorem 2. *Let $\Pi(n)$ be the number of primes $\leq n$. Then there exists($\exists$) constants c_1 and c_2 such that:*

$$c_1 \frac{n}{log(n)} \leq \Pi(n) \leq c_2 \frac{n}{log(n)}$$

Let's say we wanted to find a prime of a certain size. We could pick a random number, test it for primality, and continue sampling random numbers until we found a prime one. The theorem tells us what the density of primes is. We can use the density of primes to give us the expected number of times we have to pick randomly until we find a prime, *by the theorem this is about 100 times.*

The part we would like to improve is our ability to test whether or not a number is prime. We could do it by dividing by every number smaller than it, but for large numbers that would take a long time. We want an algorithm to primality in the length of the number, not the magnitude.

Lemma 3. *If a and b are relatively prime and a divides bc, then a divides c.*

Note, you may use the math guide for understanding *relatively prime* and *gcd*.

We start the proof with an example to gain some intuition. If $a = 2 * 3$, $b = 2 * 2$, and $c = 3 * 3$, then $a = 6$ does divide $bc = 36$. However, $a = 6$ does not divide $c = 9$, because a portion of what a was dividing in bc was in b. However, our lemma only holds if a and b are relatively prime. In our example $gcd(a, b) = 2$, hence a and b were not relatively prime. The idea is that if a divides bc and a and b are relatively prime, then all of what a is dividing in bc is in c.

Proof. Since a and b are relatively prime, there exist, ($\exists$), integers s and t such that $as + bt = 1$. Now:

$$\begin{aligned} asc + btc &= c \\ \frac{asc + btc}{a} &= \frac{c}{a} \\ sc + t\left(\frac{bc}{a}\right) &= \frac{c}{a}. \end{aligned}$$

The lemma states a divides bc. Hence, $\frac{c}{a}$ is the sum of multiples of integers. Hence, $\frac{c}{a}$ is an interger. Hence, a divides c. □

Lemma 4. *If a prime p divides a product of integers $a_1 * a_2 * \cdots a_n$, then p divides some a_i.*

Proof. We do a proof by induction on $n \geq 1$, the length of the product.

For the base case $n = 1$: We are given that prime p divides the product of integers a_1. Let, $i = 1$, we have found an i such that p divides $a_i = a_1$.

Inductive Step: We assume if a prime p divides a product of integers $a_1 * \cdots * a_n$, then there exists an a_i that p divides.

To prove by induction we must now show if a prime p divides a product of integers $a_1 * \cdots a_n * a_{n+1}$, then there exists an a_i that p divides.

We first note a fact about primes. If p is prime then the $\gcd(p, x)$ for any number x can only be 1 or p. If the $\gcd(p, x) = 1$, then p does not divide x. If $\gcd(p, x) = p$, then p does divide x. If you consider the definition of a prime and what the gcd is, then these facts become apparent.

Now there are two cases:

Case 1: $\gcd(p, a_{n+1}) = p$, if so let $i = n + 1$ and we have found an a_i that p divides.

Case 2: $\gcd(p, a_{n+1}) = 1$, if so p does not divide a_{n+1}. So, p and a_{n+1} are relatively prime and p divides $(a_1 * \cdots a_n)a_{n+1}$ and by the previous lemma p divides $a_1 * \cdots a_n$. We have reduced to our assumption for n, hence by our assumption there exists an i such that p divides a_i :)

In either case we produce an a_i that p divides, hence we have shown the inductive step to hold, and by induction we have proved our theorem.

□

Theorem 5 (Fundamental Theorem of Arithmetic)**.** *Any integer has a unique factorization.*

Proof. We take for granted that any number can be written as a product of primes (write the number as a product and recursively break up any composites.) The part we need to prove is the uniqueness of the factorization. We prove this by showing any two factorizations are the same. Let:

$$n = \Pi p_i = \Pi q_j,$$

where $p_1, p_2 \dots$ and $q_1, q_2, \dots$ are two prime factorizations. Now we get to use the previous lemma (note how we prove smaller lemmas in order to build up to proving a theorem!)

Because p_1 is a factor of n, we know p_1 divides n. Since p_1 divides n, p_1 divides $\Pi q_j = q_1 * q_2 * \dots = n$. Now by the lemma, there exists, $(\exists)$, $q_{j'}$ such that p_1 divides $q_{j'}$.

Because p_1 is prime and $q_{j'}$ is prime, it can only be that $p_1 = q_{j'}$.

If $p_1 = q_{j'}$, then p_1 does not contribute to the difference between the two factorizations. Let us now consider $\frac{n}{p_1}$. We can repeat the same process for p_2 and find the matching q_j. Let us consider $\frac{n}{p_1 * p_2}$, and iterate. Hence, for every p_i there is an equal $q_{j'}$. There was nothing unique about our assumptions of p_i and q_j, the entire process can be repeated to show that for every q_j there is an equal $p_{i'}$. Hence, if every p_i is matched by a q_j and every q_j is matched by a p_i, $\{p_1, p_2 \dots\} = \{q_1, q_2, \dots\}$. This allows us to conclude that any two factorizations of n are the same. □

We end with a teaser for next time.

We want to use these proofs about primes to come up with an encryption, E, /decryption, D, system that works as follows. For a message, m, $D(E(m)) = m$ and $E(D(m)) = m$. But in this system we want to be able to make E public and keep D private. So if Alice wants to send Bob a message, she sends the encryption, $E(m)$. Eve can only read Alice's message if she has D. RSA next time!