In today's lecture, we will talk about randomization and hashing in a slightly different way. In particular, we use arithmetic modulo prime numbers to (approximately) check if two strings are equal to each other. Building on that, we will get an randomized algorithm (called the *Karp-Rabin fingerprinting scheme*) for checking if a long text T contains a certain pattern string P as a substring.

Objectives of this lecture

In this lecture, we will:

- Cover some facts about prime numbers that are useful for randomized hashing schemes
- See a new application of hashing to string equality checking
- Look at the Karp-Rabin pattern matching algorithm

Recommended study resources

- CLRS, *Introduction to Algorithms*, Section 32.2, The Rabin-Karp algorithm
- DPV, *Algorithms*, Section 1.3.1, Generating random primes

1 How to Pick a Random Prime

In this lecture, we will often be picking random primes, so let's talk about that. (In fact, you do this when generating RSA public/private key pairs.)

How to pick a random prime in some range $\{1, \ldots, M\}$? Here's the most straightforward approach:

Algorithm: Random prime generation

- Pick a random integer x in the range $\{1, \ldots, M\}$.
- Check if x is a prime. If so, output it. Else go back to the first step.

Okay this is not quite complete, we have to fill in some details. How would you pick a random number in the prescribed range? Pick a uniformly random bit string of length $\lfloor \log_2 M \rfloor + 1$. (We assume we have access to a source of random bits.) If it represents a number $\leq M$, output it, else repeat. The chance that you will get a number $\leq M$ is at least half, so in expectation you have to repeat this process at most twice.

How do you check if x is prime? You can use the Miller-Rabin randomized primality test[1] (which may produce false positives, but it will only output "prime" when the number is composite with very low probability). There are other randomized primality tests as well, see the Wikipedia page. Or you can use the Agrawal-Kayal-Saxena[2] primality test, which has a worse runtime, but is deterministic and hence guaranteed to be correct. We won't cover those algorithms in this course, so for now, just know that they exist, we can use them, and know that they run in $O(\text{polylog}\, M)$ time.

[1] `http://en.wikipedia.org/wiki/Miller-Rabin_primality_test`
[2] `http://en.wikipedia.org/wiki/AKS_primality_test`

2 How Many Primes?

You have probably seen a proof that there are infinitely many primes. Here's a different question that we'll need for this lecture.

For positive integer n, how many primes are there in the set $\{1, 2, \ldots, n\}$?

Let there be $\pi(n)$ primes between 0 and n. One of the great theorems of the 20^{th} century was the Prime Number theorem:

Theorem 1: The prime number theorem

The prime counting function $\pi(n)$ satisfies

$$\lim_{n\to\infty} \frac{\pi(n)}{n/(\ln n)} = 1.$$

And while this is just a limiting statement, an older result of Chebyshev (from 1848) says that

Theorem 2: Chebyshev

For $n \geq 2$, the prime counting function $\pi(n)$ satisfies

$$\pi(n) \geq \frac{7}{8}\frac{n}{\ln n} = (1.262\ldots)\frac{n}{\log_2 n} > \frac{n}{\log_2 n}$$

Here are two consequences of this theorem. The first is that a random integer between 1 and n is a prime number with probability at least $\frac{1}{\log_2 n}$. Put another way, we also get the following useful fact:

Corollary 1: Density of primes

If we want at least $k \geq 4$ primes between 1 and n, it suffices to have $n \geq 2k \log_2 k$.

Proof. Just plugging in to Theorem 2, we get $\pi(2k\log_2 k) \geq \frac{2k\log_2 k}{\log_2(2k\log_2 k)} \geq \frac{2k\log_2 k}{\log_2 2+\log_2 k+\log_2\log_2 k} \geq k$. $\square$

2.1 Tighter Bounds

The following even tighter set of bounds were proved by Pierre Dusart in 2010.

Theorem 3: Dusart

For all $n \geq 60184$ we have:

$$\frac{n}{\ln n - 1.1} > \pi(n) > \frac{n}{\ln n - 1}$$

Because this is a two-sided bound, it allows us to deduce a lower bound on the number of primes in a range. For example, the number of 9-digit prime numbers (i.e. primes in the range $[10^8, 10^9 - 1]$) is

$$\pi(10^9 - 1) - \pi(10^8 - 1) > \frac{10^9 - 1}{\ln(10^9 - 1) - 1} - \frac{10^8 - 1}{\ln(10^8 - 1) - 1.1} = 44928097.3\ldots$$

From this we can infer that a randomly generated 9 digit number is prime with probability at least $0.049920\ldots$. Thus, the random sampling method would take at most 21 iterations in expectation to find a 9-digit prime.

3 The String Equality Problem

Here's a simple problem: we're sending a Mars lander. Alice, the captain of the Mars lander, receives an N-bit string x. Bob, back at mission control, receives an N-bit string y. Alice knows nothing about y, and Bob knows nothing about x. They want to check if the two strings are the same, i.e., if $x = y$. [3]

One way is for Alice to send the entire N-bit string to Bob. But N is very large. And communication is super-expensive between the two of them. So sending N bits will cost a lot. *Can Alice and Bob share less communication and check equality?*

If they want to be 100% sure that $x = y$, then one can show that fewer than N bits of communication between them will not suffice. But suppose we are OK with being correct with probability 0.9999. Formally, we want a way for Alice and Bob to send a message to Bob so that, at the end of the communication:

- If $x = y$, then $\Pr[\text{ Bob says } \texttt{equal} \text{ }] = 1$.
- If $x \neq y$, then $\Pr[\text{ Bob says } \texttt{unequal} \text{ }] = 1 - \delta$.

Here's a protocol that does almost that. We will *hash* the strings using the hash function $h_p(x) = (x \bmod p)$ for a random prime p, then check whether the hashes are equal.

Algorithm: Randomized string equality test

1. Alice picks a random prime p from the set $\{1, 2, \ldots, M\}$ for $M = \lceil 2 \cdot (5N) \cdot \log_2(5N) \rceil$.
2. She sends Bob the prime p, and also the value $h_p(x) := (x \bmod p)$.
3. Bob checks if $h_p(x) \equiv y \mod p$. If so, he says `equal` else he says `unequal`.

For now, let's not worry about where the particular value of M came from: it will arise naturally. Let's see how this protocol performs.

Lemma 1

If $x = y$, then Bob always says `equal`.

Proof. Indeed, if $x = y$, then $x \mod p = y \mod p$. So Bob's test will always succeed. □

Lemma 2

If $x \neq y$, then $\Pr[\text{ Bob says } \texttt{equal} \text{ }] \leq \frac{1}{5}$.

Proof. Consider x and y and N-bit binary numbers. So $x, y < 2^N$. Let $D = |x - y|$ be their difference. Bob says `equal` only when $x \mod p = y \mod p$, or equivalently $(x - y) = 0 \mod p$. This means p divides $D = |x - y|$. In words, the random prime p we picked happened to be a divider of D. What are the changes of that? Let's do the math.

The difference D is a N-bit integer, so $D \leq 2^N$. So D can be written (uniquely) as $D = p_1 p_2 \cdots p_k$, each p_i being a prime, where some of the primes might repeat[4]. Each prime $p_i \geq 2$, so $D = p_1 p_2 \cdots p_k \geq 2^k$. Hence

[3] E.g., this could be the latest update to the lander firmware, and we want to make pretty sure the file did not get corrupted

[4] This unique prime-factorization theorem is known as the fundamental theorem of arithmetic.

$k \leq N$: the difference D has at most N prime divisors. The probability that the randomly chosen prime p is one of them is

$$\frac{N}{\text{number of primes in } \{1, 2, \ldots, M\}}.$$

We want this to be at most 1/5, i.e., we would like that the number of primes in $\{1, 2, \ldots, M\}$ is at least $5N$. But Corollary 1 says that choosing $M \geq 10N \log_2 5N$ will give us at least $5N$ primes. Hence

$$\Pr[\text{ Bob says } \texttt{equal} \text{ and hence errs }] \leq \frac{N}{\text{number of primes in } \{1, 2, \ldots, M\}} \leq \frac{N}{5N} \leq \frac{1}{5}.$$

□

3.1 Communication cost

Naïvely, Alice could have sent x over to Bob. That would take N bits. Now she sends the prime p, and x mod p. That's two numbers at most $M = 10N \log_2 5N$. The number of bits required for that:

$$2 \log_2 M = 2 \log_2(10N \log_2 5N)) = O(\log N).$$

To put this in perspective, suppose x and y were two copies of all of Wikipedia. Say that's about 25 billion characters (25 GB of data!). Say 8 bits per character, so $N \approx 2 \cdot 10^{11}$ bits. With the new approach, Alice sends over $2 \log_2(10N \log_2 5N)) \approx 86$ bits, or 11 bytes of data. That's a lot less communication!

3.2 Reducing the Error Probability

If you don't like the probability of error being 20%, here are two ways to reduce it.

Approach #1 Have Alice repeat this process multiple times independently with different random primes, with Bob saying `equal` if and only if in all repetitions, the test passes. For example, for 10 repetitions, the chance that he will make an error (i.e., say `equal` when $x \neq y$) is only

$$(1/5)^{10} = \frac{1024}{10^{10}} \leq 0.000001.$$

That's a 99.999% chance of success! In general, if we repeat R times, we get the probability of error is at most $(1/5)^R$, so if we desire an error probability of δ, we should do $R = \log_5(1/\delta)$ repetitions. Since each round requires communicating $O(\log N)$ bits, the total number of bits that Alice must communicate is

$$O(\log(1/\delta) \log N).$$

Can we do better than this?

Approach #2 Have Alice choose a random prime from a larger set. For some integer $s \geq 1$, if we choose $M = 2 \cdot sN \log_2(sN)$, then the arguments above show that the number of primes in $\{1, \ldots, M\}$ is at least sN. And hence the probability of error is $1/s$. If we desire an error probability of δ, then we must choose $s = 1/\delta$. For example, to obtain the same 99.999% chance of success, we would pick $s = 1/10^{-6} = 10^6$. Now Alice is communicating two integers at most $2 \cdot sN \log_2(sN)$, so the number of bits is

$$\begin{aligned} 2 \log_2(2 \cdot sN \log_2(sN)) &= 2 \log_2 s + 2 \log_2 N + 2 \log_2(\log_2(sN)) + 2, \\ &= O(\log s + \log N), \\ &= O(\log(1/\delta) + \log N). \end{aligned}$$

This is much better than Approach #1!

4 The Karp-Rabin Algorithm (a.k.a. the "Fingerprint" Method)

Let's use this idea to solve a different problem.

Problem: Pattern matching

In the *pattern matching* problem, we are given, over some alphabet Σ,

- A *text* T, of length m.
- A *pattern* P, of length n.

The goal is to output the locations of all the occurrences of the pattern P inside the text T. E.g., if $T = \texttt{abracadabra}$ and $P = \texttt{ab}$ then the output should be $\{0, 7\}$.

There are many ways to solve this problem, but today we will use randomization to solve this problem. This solution is due to Karp and Rabin.[5] The idea is smart but simple, elegant and effective—like in many great algorithms. To simplify the presentation, we will start by assuming that $\Sigma = \{0, 1\}$, i.e., all of our strings are written in binary, but all of the ideas generalize to larger alphabets.

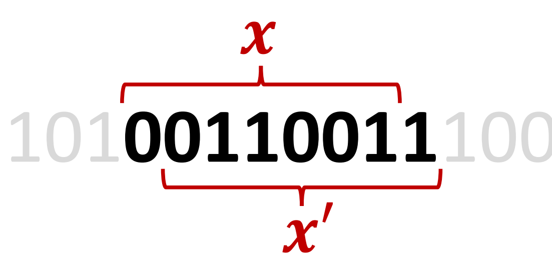
4.1 The Karp-Rabin Idea: "Rolling the hash"

As in the last section, if we interpret the string written in binary *as an integer*, lets use the hash function

$$h_p(x) = (x \bmod p)$$

for some randomly chosen prime p.

Now look at the string x' obtained by dropping the leftmost bit of x, and adding a bit to the right end. E.g., if $x = \texttt{0011001}$ then x' might be $\texttt{0110010}$ or $\texttt{0110011}$. If I told you $h_p(x) = z$, can you compute $h_p(x')$ fast?



Let x'_{lb} be the lowest-order (rightmost) bit of x', and x_{hb} be the highest order (leftmost) bit of x. Now observe that

- removing the high-order bit (x_{hb}) is just equivalent to subtracting $x_{hb} \cdot 2^{n-1}$,
- shifting all of the remaining bits to one higher position is equivalent to multiplying by 2,
- appending the low-order bit x'_{lb} is equivalent to just adding x'_{lb}.

Therefore, we can write

$$x' = 2(x - x_{hb} \cdot 2^{n-1}) + x'_{lb}$$

[5] Again, familiar names. Dick Karp is a professor of computer science at Berkeley, and won the Turing award in 1985. Among other things, he developed several max-flow algorithms we will cover later in the course, and his 1972 paper showed that many natural algorithmic problems were NP complete. Michael Rabin is professor at Harvard; he won the Turing award in 1976 (jointly with CMU's Dana Scott). You may know him from the popular Miller-Rabin randomized primality test (the Miller there is our own Gary Miller); he's responsible for many algorithms in cryptography.

Since $h_p(a+b) = (h_p(a) + h_p(b)) \mod p$, and $h_p(2a) = 2h_p(a) \mod p$, we then have

$$h_p(x') = (2h_p(x) - x_{hb} \cdot h_p(2^n) + x'_{lb}) \mod p.$$

Take a moment to understand the significance of this fact. Given the hash $h_p(x)$ for the substring x and the value $h_p(2^n)$, we can compute the hash of the next adjacent substring $h_p(x')$ in just a constant number of arithmetic operations modulo p. This is an enormous speedup compared to computing $h_p(x')$ from scratch which would take $O(n)$ arithmetic operations.

4.2 The pattern matching algorithm

To keep things short, let $T_{i...j}$ denote the string from the i^{th} to the j^{th} positions of T, inclusive. So the string matching problem is: output all the locations $i \in \{0, 1, \ldots, m-n\}$ such that

$$T_{i...i+(n-1)} = P.$$

Here's the algorithm.

Algorithm: Karp-Rabin pattern matching

1. Pick a random prime p in the range $\{1, \ldots, M\}$ for $M = \lceil 2sn \log_2(sn) \rceil$ (we'll choose s later.)
2. Compute $h_p(P)$ and $h_p(2^n)$, and store these results.
3. Compute $h_p(T_{0...n-1})$, and check if it equals $h_p(P)$. If so, output `match at location 0`.
4. For each $i \in \{0, \ldots, m-n\}$
   (i) compute $h_p(T_{i+1...i+n})$ using $h_p(T_{i...i+n-1})$
   (ii) If $h_p(T_{i+1...i+n}) = h_p(P)$, output `match at location` $i+1$.

Notice that we'll never have a false negative (i.e., miss a match) but we may erroneously output location that are not matches (have a false positive) if we get a hash collision! Let's analyze the error probability, and the runtime.

Probability of Error We do m different comparisons, each has a probability $1/s$ of failure. So, by a union bound, the probability of having at least one false positive is m/s. Hence, setting $s = 100m$ will make sure we have at most a $\frac{1}{100}$ chance of even a single mistake.

This means we set $M = (200 \cdot mn) \log_2(200 \cdot mn)$, which requires at most $\log_2 M + 1 = O(\log m + \log n)$ bits to store. Hence our prime p is also at most $O(\log m + \log n)$ bits.[6]

Running Time Let's say we can do arithmetic and comparisons on $O(\log M)$-bit numbers in constant time. And let's not worry about the time to pick a random prime for now.

- Computing $h_p(x)$ for n-bit x can be done in $O(n)$ time. So each of the hash function computations in Steps 2 and 3 take $O(n)$ time.
- Now, using the idea in Section 4.1, we can compute each subsequent hash value in $O(1)$ time! So iterating over all the values of i takes $O(m)$ time.

That's a total of $O(m+n)$ time! And you can't do much faster, since the input itself is $m+n$ bits long.

[6] If we you do the math, and say $m, n \geq 10$, then $\log_2 M \leq 4(\log_2 m + \log_2 n)$. Now, just for perspective, if we were looking for a $n = 1024$-bit phrase in Wikipedia, this means the prime p is only $4(\log_2 2^{38} + \log_2 2^{10}) \leq 192$ bits long.

4.3 Extensions and Connections

General alphabets For simplicity, we looked at the case where $\Sigma = \{0, 1\}$. The Karp-Rabin algorithm generalizes naturally to larger alphabets. Instead of treating the input as a number in binary, treat it as base-$|\Sigma|$. For example, if the text contains only lower-case English words, we would use base-26. The formula for rolling the hash still works, except we replace 2 with $|\Sigma|$, and the range $\{1, \ldots, M\}$ from which we should select our prime becomes slightly larger.

Exercise: Larger alphabets

Suppose we use an alphabet Σ which has size $|\Sigma|$ for Karp-Rabin. What should we use as our new value of M, and how does this affect the number of bits required to store the prime p?

Other pattern matching algorithms and problems There are other (deterministic) fast ways of solving the pattern matching problem we mentioned above. See, e.g., the Knuth-Morris-Pratt algorithm[7], and the suffix tree and suffix array data structures[8]. You don't need to know those for this course, but they are very interesting and useful. The advantage of the Karp-Rabin approach is not only the simplicity, but also the extensibility. You can, for example, solve the following 2-dimensional problem using the same idea.

Exercise: 2D pattern matching

Given a $(m_1 \times m_2)$-bit rectangular text T, and a $(n_1 \times n_2)$-bit pattern P (where $n_i \leq m_i$), find all occurrences of P inside T. Show that you can do this in $O(m_1 m_2)$ time, where we assume that you can do modular arithmetic with integers of value at most $\text{poly}(m_1 m_2)$ in constant time.

[7] `https://en.wikipedia.org/wiki/Knuth_Morris_Pratt_algorithm`
[8] `http://www.cs.cmu.edu/~15451-f19/LectureNotes/lec24-sufftree.pdf`