In the classic string matching problem, there's a string T of length t and a pattern P of length p. The problem is to find the places in T where the pattern occurs. Later in the course we'll have more to say about this problem, but for this lecture we will consider a beautiful use of hashing to solve it. This solution is due to Karp and Rabin. [1]

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

The idea here is to design a special kind of "rolling" hash function $h(S)$ on strings S that has the following properties:

1. After a small amount of precomputation, given the hash of the string $T[i \dots (i+p-1)]$, we can compute $h(T[(i+1) \dots (i+p)])$ with a constant number of arithmetic operations. (All our operations will be done modulo some prime number $q = O(t \cdot p^2)$.)
2. The probability of a collision will be "low". We'll be more precise later.

So ignoring the collisions, which have "low" probability, it's easy to see how this hash function can solve the classic text search problem. We first compute $h(P)$ the hash function of the pattern. Then try all substrings $T[i \dots (i+p-1)]$ for all $i \in \{0, 1, \dots, t-p\}$ and see which ones of them have hash value equal to $h(P)$, and output all locations where $h(T[i \dots (i+p-1)]) = h(P)$. [2] There are $t-p+1$ such strings of suitable length, so the algorithm will take time $O(t-p+1)$ for these tests, $O(p)$ to compute $h(P)$ and for the preprocessing, so overall $O(t+p)$ time.

For simplicity, assume that the alphabet $\Sigma = \{0, 1\}$. The Karp-Rabin hash family is the following. Choose a uniformly random prime $q \in \{2, \dots, K\}$ for some parameter K to be chosen later. (How do we choose a random prime? We will talk about this later.) Now, define

$$h_q(T[i \dots j]) = (T[i] \cdot 2^{j-i} + T[i+1] \cdot 2^{j-i-1} + \dots + T[j] \cdot 2^0) \bmod q$$

That is, interpret the $(j-i+1)$ bits in the substring as a number written in binary, and take the residue modulo q. This is often called the "fingerprint" of the substring.

Good. Now what is the hash of the "next" location, if we were to move the "window" over by 1? Just plug in the definition

$$h_q(T[i+1 \dots j+1]) = (T[i+1] \cdot 2^{j-i} + T[i+2] \cdot 2^{j-i-1} + \dots + T[j+1] \cdot 2^0) \bmod q$$

which is the same as

$$h_q(T[i+1 \dots j+1]) = \left(\left(h_q(T[i \dots j]) - T[i] \cdot 2^{j-i} \right) \cdot 2 + T[j+1] \cdot 2^0 \right) \bmod q$$

[1] 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 two of the max-flow algorithms you saw, 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.

[2] Note this is a Monte-Carlo algorithm, since the algorithm may make a mistake and output some location i such that $h(T[i \dots (i+p-1)]) = h(P)$ but $T[i \dots (i+p-1)] \neq P$. We will show the probability of this is small.

Now, suppose we are given $h_q(T[i \ldots (i+p-1)])$ and want to compute $h_q(T[(i+1) \ldots (i+p)])$. What do we do?

Easy. Take $h_q(T[i \ldots (i+p-1)])$, subtract $T[i] \cdot 2^{p-1} \bmod q$ from it (note that $T[i] \cdot 2^{p-1} \bmod q$ is either 0 or $2^{p-1} \bmod q$, and we can precompute and store $2^{p-1} \bmod q$), then multiply by 2, and add in $T[j+1]$. (All modulo q, of course.) A constant number of arithmetic operations modulo q, as advertised.

1.1 Probability of False Positives

What about the possibility of false matches? Suppose for any fixed location i, the probability of an incorrect match is δ. Then by a union bound over $t-p+1$ locations we perform the equality test, the probability of outputting some false positive is at most $(t-p+1) \cdot \delta \leq t\delta$. If we want the final probability of error to be at most $1/2$, we should ensure that $\delta \leq 1/(2t)$. How?

Notice the only randomness is in the choice of the random prime q. We make a mistake when the number represented by the p-bit substring $T[i \ldots (i+p-1)]$ (call this number a) is *not equal* to the number represented by the p-bit pattern P (call this second number b), but these two numbers are the same modulo the random q (i.e., $a \equiv_p b$). By the definition of being equivalent modulo q, this means that q divides $|a-b|$. Now $|a-b|$ is also a p-bit number, so it can have less than p distinct prime divisors. (Each prime divisor is at least 2, and $|a-b| < 2^p$.) And if $q|(a-b)$, then q must have been one of these "bad" values, these prime divisors.

We would like to claim that choosing a uniformly random prime number q in the range $\{2, \ldots, K\}$, the chance that we choose one of (at most) p bad values is smaller than $1/(2t)$. For this, it suffices to choose K large enough such that there are at least $2pt$ primes between 2 and K. For this we use the Prime Number theorem: if there are $\pi(x)$ primes between 0 and x, then the theorem says that $\lim_{n\to\infty} \frac{\pi(x)}{x/(\ln x)} = 1$. And while this is just an asymptotic statement, we also know that $\pi(n) \geq \frac{7}{8}\frac{n}{\ln n}$ (this was proved by Chebyshev back in 1848). Now setting K to equal, say, $10pt \ln pt$ ensures that $\pi(K) \geq 2pt$ for large enough pt, which proves the result.

If you want to reduce the error probability, you could either pick several independent primes q and perform the string matches in parallel (claiming that there is a match at location i only when all the fingerprints match), or you could increase the value of K, which increases the range from which you pick your random prime.[3]

1.2 Picking a Random Prime

One detail is how to pick a random prime in some range $\{0, 1, \ldots, M\}$. The answer is easy.

- Pick a random integer X in the range $\{0, 1, \ldots, M\}$.
- Check if X is a prime. If so, output it. Else go back to the first step.

How would you pick a random number in the prescribed range? Just 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

[3]Converting this to a Las Vegas test is a little more complicated. Clearly if the algorithm doesn't output location i, we know that $T[i \ldots (i+p-1)]$ does not match P. But checking if the locations that were output were bonafide matches might take a lot of time: imagine a text containing t copies of a, and a pattern with p copies of a. Naïvely checking for false matches might take $O(pt)$ time. There are ways to check in linear time if there exists a false match in the output locations which are not difficult; see the book for details.

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 (which may err, 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 primality test, which has a worse runtime, but is deterministic and hence guaranteed to be correct.

Finally, how many repetitions would you have to do in the above algorithm before you output a prime? Again, you can appeal to the Prime Number Theorem. The number of primes in the range $\{0, 1, \ldots, M\}$ is $\Omega(\frac{M}{\ln M})$, so a random number in this range is prime with probability $\Omega(\frac{1}{\ln M})$. Hence the expected number of repetitions is $O(\log M)$.