CS 124 **Lecture 15**

15.1 Applications: Fingerprinting for pattern matching

Suppose we are trying to find a pattern string P in a long document D. How can we do it quickly and efficiently?

Hash the pattern P into say a 16 bit value. Now, run through the file, hashing each set of $|P|$ consecutive characters into a 16 bit value. If we ever get a match for a pattern, we can check to see if it corresponds to an actual pattern match. (In this case, we want to double-check and not report any false matches!) Otherwise we can just move on. We can use more than 16 bits, too; we would like to use enough bits so that we will obtain few false matches.

This scheme is efficient, as long as hashing is efficient. Of course hashing can be a very expensive operation, so in order for this approach to work, we need to be able to hash quickly on average. In fact, a simple hashing technique allows us to do so in constant time per operation!

The easiest way to picture the process is to think of the file as a sequence of digits, and the pattern as a number. Then we move a pointer in the file one character at a time, seeing if the next $|P|$ digits gives us a number equal to the number corresponding to the pattern. Each time we read a character in the file, the number we are looking at changes is a natural way: the leftmost digit a is removed, and a new rightmost digit b is inserted. Hence, we update an old number N and obtain a new number N' by computing

$$N' = 10 \cdot (N - 10^{|P|-1} \cdot a) + b.$$

When dealing with a string, we will be reading characters (bytes) instead of numbers. Also, we will not want to keep the whole pattern as a number. If the pattern is large, then the corresponding number may be too large to do effective comparisons! Instead, we hash all numbers down into say 16 bits, by reducing them modulo some appropriate prime p. We then do all the mathematics (multiplication, addition) modulo p, i.e.

$$N' = [10 \cdot (N - 10^{|P|-1} \cdot a) + b] \bmod p.$$

All operations mod p can be made quite efficient, so each new hash value takes only constant time to compute!

This pattern matching technique is often called *fingerprinting*. The idea is that the hash of the pattern creates an almost unique identifier for the pattern– like a fingerprint. If we ever find two fingerprints that match, we have a good reason to expect that they must come the same pattern. Of course, unlike real fingerprints, our hashing-based fingerprints do not actually uniquely identify a pattern, so we still need to check for false matches. But since false matches should be rare, the algorithm is very efficient!

See Figure 15.1 for an example of fingerprinting.

One question remains. How should we choose the prime p? We would like the prime we choose to work well, in that it should have few false matches. The problem is that for every prime, there are certainly some bad patterns and documents. If we choose a prime in advance, then someone can try to set up a document and pattern that will cause a lot of false matches, making our fingerprinting algorithm go very slowly.

A natural approach is to choose the prime p randomly. This way, nobody can set up a bad pattern and document in advance, since they are not sure what prime we will choose.

$$\begin{aligned} P &= 17935 \\ p &= 251 \\ P \bmod p &= 114 \\ D &= 6386179357342 \end{aligned}$$

$$\begin{aligned} 63861 \bmod p &= 107 \\ 38617 \bmod p &= 214 \\ 86179 \bmod p &= 86 \\ 61793 \bmod p &= 47 \\ 17935 \bmod p &= 114 \\ 79357 \bmod p &= 41 \\ 93573 \bmod p &= 201 \\ 35734 \bmod p &= 92 \\ 57342 \bmod p &= 114 \end{aligned}$$

Figure 15.1: A fingerprinting example. The pattern P is a 5 digit number. Note successive calculations take constant time: 38617 mod p = ((63861 mod p) - (60000 mod p)) $\cdot$ 10 + 7 mod p. Also note that false matches are possible (but unlikely); 57432 = 17935 mod p.

Let us make this a bit more rigorous. Let $\pi(x)$ represent the number of primes that are less than or equal to x. It will be helpful to use the following fact:

Fact: $\frac{x}{\log x} \leq \pi(x) \leq 2\frac{x}{\log x}$ for $x > 2$.

Consider any point in the algorithm, where the pattern and document do not match. If our pattern has length $|P|$, then at that point we are comparing two numbers that are each less than $10^{|P|}$. In particular, their difference (in absolute value) is less than $10^{|P|}$. What is the probability that a random prime divides this difference? That is, what is the probability that for the random prime we choose, the two numbers corresponding to the pattern and the current $|P|$ digits in the document are equal modulo p.

First, note that there are at most $\log_2 10^{|P|}$ distinct primes that divide the difference, since the difference is at most $10^{|P|}$ (in absoulte value), and each distinct prime divisor is at least 2. Hence, if we choose our prime randomly from all primes up to Z, the probability we have a false match is at most

$$\frac{\log_2 10^{|P|}}{\pi(Z).}$$

Now the probability that we have a false match anywhere is at most $|D|$ times the probability that we have a false match in any single location, by the union bound. Hence the probability that we have a false match anywhere is at most

$$\frac{|D| \log_2 10^{|P|}}{\pi(Z).}$$

Exercise: How big should we make Z in order to make the probability of a false match anywhere in the algorithm less than $1/100$?

How could we improve the probability of a false match? One way is to choose from a larger set of primes. Another way is to choose not just one random prime, but several random primes from Z. This is like choosing several hash functions in the Bloom filter problem. There is a false match only if there is a false match at every random prime we choose. If we choose k primes (with replacement) from the primes up to Z, the probability of a false match at a specific point is at most

$$\left(\frac{\log_2 10^{|P|}}{\pi(Z)}\right)^k.$$