CS3230 Design & Analysis of Algorithms **February 17, 2022**

Lecture 6: Fingerprinting & Streaming

Lecturer: Warut Suksompong

The goal of this lecture is to describe two important applications of hashing: string pattern matching and streaming algorithms.

6.1 String Pattern Matching

Pattern matching is a fundamental problem when working with strings. Let T and P be two strings of length n and m respectively, where $n \geq m$. We will refer to T as the *text string* and P as the *pattern string*. Each character in the strings is from[2] an alphabet of size ℓ. The goal of the pattern matching problem is to decide whether P appears as a substring in T or not.

Example 1. Suppose T is `abracadabra`. Then, $P_1 =$ `raca` does appear as a substring in T (from position 3) while $P_2 =$ `cara` does not.

We will think of both n and m being large. For example, if a search engine is searching all of the text on the internet, which is around 5 billion gigabytes, then $n \approx 2^{65}$ in the binary alphabet. Similarly, if a biologist is searching the human genome, then it can be viewed as a string of length approximately 6.4 billion in an alphabet of size 4. The pattern string length m can also be reasonably large; for example, the pattern could be a 1000-line program that is to be searched in a gigantic codebase.

Nevertheless, an important remark needs to be made here. Throughout, we'll assume that $\log n$ and $\log m$ are small enough that they can be stored in a *constant* number of machine words. Even for $n \approx 2^{65}$ as in the internet example above, $\lg n = 65$, which can be stored in less than three 64-bit words. Therefore, throughout this lecture, the word-RAM model will allow us to assume that: **operations on strings of length $O(\log n)$ can be executed in $O(1)$ time**.

Finally, we assume that ℓ is a small fixed number (e.g., 10 or 26).

6.1.1 First Attempt

The obvious algorithm just checks whether each substring of length m matches exactly with P. Here is the pseudocode (taken from CLRS, Section 32.1).

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

The running time of the algorithm is $\Theta((n-m+1)m)$ because there are $n-m+1$ iterations of the **for** loop with each iteration requiring $\Theta(m)$ time. Hence, if m is large, e.g., $n/3$, then the running time can be $\Theta(n^2)$. Can we do better?

[1] See also Section 32.1 and 32.2 of CLRS.

[2] In the lecture, we assumed the binary alphabet $\{0, 1\}$, meaning $\ell = 2$. Here, we will allow ℓ to be larger.

We describe an approach called *fingerprinting* that yields asymptotic savings and works quite well in practice. Fingerprinting involves two ideas:

(i) **[Faster equality checks]** The first idea is to speed up each equality check in line 4 of NAIVE-STRING-MATCHER by making it randomized. We will choose a hash function h randomly, and use it to hash P as well as each of $T[1\,..\,m], T[2\,..\,m+1], \ldots, T[n-m+1\,..\,n]$. Then, in line 4, instead of comparing P and $T[s+1\,..\,s+m]$ directly, we will compare the hashes $h(P)$ and $h(T[s+1\,..\,s+m])$. The output of h will be representable as a string of length $O(\log n)$, and hence by the important remark on the previous page, the equality check will be in $O(1)$ time.

(ii) **[Rolling hashes]** Just the first idea doesn't suffice though, because in order to hash $T[1\,..\,m], T[2\,..\,m+1], \ldots, T[n-m+1\,..\,n]$, the hash function needs to be applied $n-m+1$ times on inputs of length m, which requires $\Theta(m(n-m+1))$ time, comparable to that of NAIVE-STRING-MATCHER. The idea of rolling hashes allows computation of the $n-m+1$ hashes in $O(n)$ time. The main observation is that each pair of consecutive substrings $T[s+1\,..\,s+m]$ and $T[s+2\,..\,s+m+1]$ differs only by two characters. For the hashing scheme we consider here, the hash values will also change predictably based on these two characters, and hence, it will be possible to update the hash value in $O(1)$ time.

The algorithm we describe here was proposed in an influential paper by Richard M. Karp and Michael O. Rabin in 1987.

6.1.2 Division Hash

Given a prime number p, the *division method* defines the hash function $h_p : \mathbb{Z} \to \{0, \ldots, p-1\}$ by $h_p(x) = x \pmod p$. This hash function treats the input key to be an integer, and it hashes into one of p slots.

We will choose the value of p uniformly from the prime numbers in the range $\{1, \ldots K\}$, where K is a parameter we set. More precisely, the hash function is set to be h_p where p is selected as follows.

CHOOSE-PRIME(K)
1 Choose random number p uniformly from $\{1, \ldots, K\}$
2 **if** p is prime
3 **return** p
4 **else** CHOOSE-PRIME(K)

There are efficient primality checking algorithms that can be used in line 2; we don't describe them here. In order to analyze the performance of this algorithm, the following number-theoretic fact will be useful.

Fact 6.1.1. *There are $> N/\lg N$ prime numbers in the range $\{1, \ldots, N\}$.*

We show next that the number of times CHOOSE-PRIME calls itself is bounded in expectation. Assuming that line 2 can be implemented efficiently (as mentioned above), this implies that the expected time to choose the hash function is small.

Claim 6.1.2. CHOOSE-PRIME *terminates after an expected $O(\lg K)$ recursive calls.*

Proof. By Fact 6.1.1, the probability that the number p chosen in Line 1 is prime is $> 1/\lg K$. Thus, CHOOSE-PRIME can be viewed as repeated Bernoulli trials with probability of success $> 1/\lg K$ at every trial (recall the exercises in last week's tutorial). Then, the expected number of trials before the first success is $< \lg K$. □

Next, we bound the probability of collisions.

Claim 6.1.3. *If x, y are distinct integers such that $0 \le x < y < 2^b$, then:*

$$\Pr[h_p(x) = h_p(y)] < \frac{b \lg K}{K}$$

where the probability is over the choice of the prime p.

Proof. $h_p(x) = h_p(y)$ means that $y - x \equiv 0 \pmod{p}$. Let $z = y - x$. Since $z < 2^b$, we claim that z must have $< b$ distinct prime factors.[3] Suppose the distinct prime factors of z are $p_1, \ldots, p_t$ for $t < b$. Now, observe that in order for $z \equiv 0 \pmod{p}$ to hold, it must be that p equals one of these factors $p_1, \ldots, p_t$, since otherwise, p cannot divide z.

Recall that CHOOSE-PRIME chooses p uniformly from the primes in the range $\{1, \ldots, K\}$, and by Fact 6.1.1, there are $> K/\lg K$ primes in this range. Hence, the probability that p equals one of the primes $p_1, \ldots, p_t$ is $< \frac{t}{K/\lg K} < \frac{b \lg K}{K}$. □

Remark. Claim 6.1.3 doesn't exactly guarantee universality, which would have required that the probability bound be $1/K$. However, as we'll see, b will be $O(\lg K)$, and so the collision probability bound is $\lesssim (\lg^2 K)/K$. For sufficiently large K, the numerator is negligible compared to the denominator, and so, this bound is almost as good.

Applying division hash to strings. Although we defined the hash function h_p above to take integers as input, we will in fact be applying it to strings over the alphabet[4] $\{0, 1, \ldots, \ell - 1\}$. This is well-defined because given a string $X = \langle X[1], X[2], \ldots, X[m] \rangle$, we can think of it as the ℓ-ary representation of an integer $int(X)$, i.e.:

$$int(X) = \sum_{i=1}^{m} X[i] \cdot \ell^{m-i}.$$

So, if X is a string, then $h_p(X)$ really means $h_p(int(X))$, but we'll be writing $h_p(X)$ for simplicity.

6.1.3 Equality Checking

Given two strings X and Y in $\{0, 1, \ldots, \ell - 1\}^m$, our approach to check whether $X = Y$ is to check for equality between $h_p(X)$ and $h_p(Y)$, where p is drawn randomly using CHOOSE-PRIME(K). The goal of this section is to analyze what value of K we should choose so that the check returns an incorrect answer with low probability.

Claim 6.1.4. *If $X = Y$, then* $\Pr[h_p(X) = h_p(Y)] = 1$.

Proof. Obvious. □

Claim 6.1.5. *For any $\delta > 0$, if $X \neq Y$ and $K = \frac{2m}{\delta} \cdot \lg \ell \cdot \lg\left(\frac{2m}{\delta} \lg \ell\right)$, then:*

$$\Pr[h_p(X) = h_P(Y)] < \delta.$$

Proof. We apply Claim 6.1.3 with $b = m \lg \ell$ and the above value of K.

$$\begin{aligned}
\Pr[h_p(X) = h_p(Y)] &< \frac{b}{K} \lg K \\
&= \frac{m \lg \ell}{\frac{2m}{\delta} \cdot \lg \ell \cdot \lg\left(\frac{2m}{\delta} \lg \ell\right)} \lg K \\
&= \frac{\delta}{2} \cdot \frac{\lg\left(\frac{2m}{\delta} \cdot \lg \ell \cdot \lg\left(\frac{2m}{\delta} \lg \ell\right)\right)}{\lg\left(\frac{2m}{\delta} \lg \ell\right)} \\
&= \frac{\delta}{2}\left(\frac{\lg\left(\frac{2m}{\delta} \lg \ell\right) + \lg\lg\left(\frac{2m}{\delta} \lg \ell\right)}{\lg\left(\frac{2m}{\delta} \lg \ell\right)}\right) \leq \delta
\end{aligned}$$

where the last inequality follows since $\lg\lg\left(\frac{2m}{\delta} \lg \ell\right) \leq \lg\left(\frac{2m}{\delta} \lg \ell\right)$. □

[3] Proof: If z factorizes as $p_1^{a_1} p_2^{a_2} \cdots p_r^{a_r}$ for distinct primes $p_1, \ldots, p_r$ and integers $a_1, \ldots, a_r \geq 1$, and if $r \geq b$, then $z \geq 2^b$ as each prime factor $p_1, \ldots, p_r$ is at least 2. This is a contradiction.

[4] Any alphabet of size ℓ can be taken to be $\{0, 1, \ldots, \ell - 1\}$ without loss of generality for the purposes of the pattern matching problem.

Now, let's consider what happens when we plug in this new equality checker in line 4 of Naive-String-Matcher. Clearly because of Claim 6.1.4, there will be no false negatives: whenever P and $T[s+1\,..\,s+m]$ are equal, their hashes will also be equal. But can there be false positives?

Claim 6.1.6. *Set $\delta = \frac{1}{100n}$ in Claim 6.1.5. Suppose P does not occur in T as a substring. Then, the probability that $h_p(P)$ and $h_p(T[s+1\,..\,s+m])$ match for some choice of $s \in \{0, 1, \ldots, n-m\}$ is $< 1\%$.*

Proof. Let E_s be the event that $h_p(P) = h_p(T[s+1\,..\,s+m])$. By Claim 6.1.5, for any particular s, $\Pr[E_s] < \frac{1}{100n}$. Hence:

$$\Pr[E_0 \text{ or } E_1 \text{ or } \cdots \text{ or } E_{n-m}] \leq \sum_{s=0}^{n-m} \Pr[E_s] < \frac{n-m+1}{100n} < \frac{1}{100}.$$

The first inequality is due to the *union bound*: the probability of a union of events is at most the sum of the probabilities of the events. □

Therefore, for $K = O(mn \cdot \lg \ell \cdot \lg(mn \lg \ell))$, the probability of false positives for the resulting pattern matching problem is $< 1\%$. Now, note that K can be stored using $\lg K = O(\lg n + \lg \lg \ell)$ bits. By the discussion at the beginning of this section, this means that any hash value can be stored in a constant number of machine words. Hence, we can check for equality between hash values in $O(1)$ time, as we desired, as long as we allow an error probability of 1%.

6.1.4 Rolling Hash

Like we mentioned in Section 6.1.1, to improve over Naive-String-Matcher, we also need to ensure that we can update $h_p(T[s+1\,..\,s+m])$ to $h_p(T[s+2\,..\,s+m+1])$ efficiently. To do this, let's recall once again the integer values corresponding to the two substrings:

$$int(T[s+1\,..\,s+m]) = \sum_{i=1}^{m} T[s+i] \cdot \ell^{m-i}; \qquad int(T[s+2\,..\,s+m+1]) = \sum_{i=1}^{m} T[s+i+1] \cdot \ell^{m-i}$$

Claim 6.1.7.

$$int(T[s+2\,..\,s+m+1]) = \ell \cdot int(T[s+1\,..\,s+m]) - \ell^m \cdot T[s+1] + T[s+m+1].$$

Proof.

$$\begin{aligned}
&\ell \cdot int(T[s+1\,..\,s+m]) - \ell^m \cdot T[s+1] + T[s+m+1] \\
&= \ell \cdot \sum_{i=1}^{m} T[s+i] \cdot \ell^{m-i} - \ell^m \cdot T[s+1] + T[s+m+1] \\
&= \sum_{i=1}^{m} T[s+i] \cdot \ell^{m-i+1} - \ell^m \cdot T[s+1] + T[s+m+1] \\
&= \sum_{j=0}^{m-1} T[s+j+1] \cdot \ell^{m-j} - \ell^m \cdot T[s+1] + T[s+m+1] \qquad \text{(substituting } j = i-1\text{)} \\
&= \sum_{j=1}^{m} T[s+j+1] \cdot \ell^{m-j} = int(T[s+2\,..\,s+m+1])
\end{aligned}$$

□

Now, we can relate the hashes of the two substrings.

Claim 6.1.8. *For any p:*

$$h_p(T[s+2\,..\,s+m+1]) = \ell \cdot h(T[s+1\,..\,s+m]) - T[s+1] \cdot h_p(\ell^m) + T[s+m+1] \pmod p.$$

Proof. This readily follows from Claim 6.1.7 using the linearity of addition modulo p. More precisely, we use the following facts, where a, b, c are arbitrary integers:

(i) $(a+b) \mod p = ((a \mod p) + (b \mod p)) \mod p$

(ii) $(ca) \mod p = (c \cdot (a \mod p)) \mod p$

We don't give proofs here, but you should be able to convince yourselves of these. We now apply these identities to the conclusion of Claim 6.1.7. □

6.1.5 Wrapping Up

We are now ready to present the Karp-Rabin algorithm based on the ideas described above. We assume $\ell = 2$ in the code below.

```
KARP-RABIN(T, P)
 1  n = T.length
 2  m = P.length
 3  p = CHOOSE-PRIME(⌈200mn lg(200mn)⌉)
 4  hash-pattern = int(P) mod p
 5  offset = 2^m mod p
 6  hash-text = int(T[1..m]) mod p
 7  for s = 1 to n − m
 8      if hash-pattern == hash-text
 9          print "Pattern occurs with shift " s
10      if s < n − m
11          hash-text = (2 · hash-text − T[s + 1] · offset + T[s + m + 1]) mod p
```

It follows from our discussion above that KARP-RABIN runs in time $O(n)$, never outputs a false negative, and outputs false positives with probability $< 1\%$.

6.2 Streaming

The *streaming model* captures many settings where the processing device has much smaller space than the volume of data. Two classic examples are:

(i) Data may be stored on voluminous hard drives, while the processing takes place in a smaller RAM or cache.

(ii) Data may be arriving in a gigabit-per-second speed network link, while the processing takes place on-the-fly in a small router.

The question is whether we can maintain a small data structure that can return useful information about the data.

Formally, a stream is a sequence of insertions and deletions of items from a universe $\mathcal{U}$. We will consider $\mathcal{U}$ to be the set $\{1, \ldots, U\}$. So, for example, a stream can look like:

$$\textsc{Add}(3), \textsc{Add}(1), \textsc{Add}(7), \textsc{Add}(3), \textsc{Delete}(3), \textsc{Add}(1), \textsc{Delete}(3)$$

In an *insertion-only* stream, there are no DELETE's in the sequence. In this lecture, we consider streams that are not necessarily insertion-only but in the so-called *strict turnstile* model, which means that at no point in the sequence is an item deleted more times than it is added.

The problem we will consider is *frequency estimation*. For an item $i \in \mathcal{U}$, let its frequency, denoted f_i, be the number of times f_i occurs at the end of the stream, i.e., the number of insertions of i minus the number of deletions of i. So, in the above example, $f_3 = 0, f_1 = 2, f_7 = 1$. Note that the strict turnstile assumption mentioned above ensures that each $f_i \geq 0$. Our goal is to maintain a data structure occupying a small amount of space that can be used to (efficiently) return f_i for any query $i \in \mathcal{U}$.

There are two obvious ways to solve the problem, similar to the naive solutions discussed for the dictionary problem in the last lecture.

- We can construct a direct access table T of length U. The operation ADD$[i]$ causes $T[i]$ to be incremented, and DELETE(i) causes $T[i]$ to be decremented.
- Assuming that universe elements can be compared, we can store the given keys in the form of a binary search tree. This data structure uses $O(M)$ space where M is the number of distinct elements.

Both solutions are unsatisfactory because U and M can both be very large, making storage of this size impractical. For a concrete example, consider a network router which needs to return a count of the total number of packets containing a particular IP address i, among all the packets arriving on a stream. Here, U is of the order 2^{32}, assuming IP addresses are 32-bit integers, while M may be in the millions on a high-speed stream.

6.2.1 Hashing-based Solution

Just as in last lecture, we can alleviate the above space issues by using a hash table to store elements according to their hash values. More precisely:

INITIALIZE(U, k)
1 Choose hash function $h : [U] \to [k]$ from a universal family $\mathcal{H}$
2 Create table T of k entries, each initialized to be 0

ADD(i)
1 $T[h(i)] = T[h(i)] + 1$

DELETE(i)
1 $T[h(i)] = T[h(i)] - 1$

QUERY(i)
1 **return** $T[h(i)]$

We have the following guarantee:

Claim 6.2.1. *Let $\widehat{f_i}$ be the value returned by* QUERY(i)*. Then:*

$$f_i \leq \mathbb{E}[\widehat{f_i}] \leq f_i + M/k$$

where $M = \sum_{i=1}^{U} f_i$ is the total sum of the frequencies of all the elements. (The expectation is with respect to the choice of the hash function h.)

Proof. Let C_i be the set of all universe elements that collide with i. Clearly, $i \in C_i$. Note that:

$$\widehat{f_i} = \sum_{j \in C_i} f_j$$

as all elements in C_i contributed to the count at slot $h(i)$. By the strict turnstile assumption, each $f_j \geq 0$, and hence, $\widehat{f_i} \geq f_i$ (no matter the hash function).

It remains to prove the upper bound on $\widehat{f_i}$. Clearly, the worst case is that all other elements collide with i, so that i's count is distorted badly. But this is exactly what the universality of $\mathcal{H}$ is supposed to prevent. To analyze this issue, define a random variable X_{ij} which equals 1 exactly when $h(i) = h(j)$. So:

$$\widehat{f_i} = \sum_{j=1}^{U} f_j X_{ij}.$$

Note that if $i = j$, then $X_{ij} = 1$ whereas if $i \neq j$, $\Pr[X_{ij} = 1] \leq 1/k$. Therefore:

$$\begin{aligned}
\mathbb{E}[\widehat{f_i}] &= \mathbb{E}\left[\sum_{j=1}^{U} f_j X_{ij}\right] \\
&= \mathbb{E}\left[f_i + \sum_{j \neq i} f_j X_{ij}\right] \\
&= f_i + \sum_{j \neq i} f_j \cdot \mathbb{E}[X_{ij}] \\
&\leq f_i + \sum_{j \neq i} \frac{f_j}{k} \\
&\leq f_i + \frac{M}{k}
\end{aligned}$$

The third equality uses linearity of expectations and that the f_i's are not random. Only the X_{ij}'s are random variables. □

If we set $k = 1/\varepsilon$ for some $\varepsilon > 0$, we get that $f_i \leq \mathbb{E}[\widehat{f_i}] \leq f_i + \varepsilon M$. The space complexity of this scheme is calculated as follows. The number of rows in the table is $1/\varepsilon$, while each row holds a number of size $O(\lg M)$ bits. Additionally, using the universal hash family described in the previous lecture, the hash function can be stored using $O(\lg U \cdot \lg M)$ bits. Hence, in bits, the space complexity is $O((1/\varepsilon + \lg U) \lg M)$. Assuming that $\lg U$ and $\lg M$ are small enough to be contained in one machine word (as we justified earlier in this lecture), the space complexity in terms of words is $O(1/\varepsilon)$.

The above analysis forms the basis of what's known as the *Count-Min Sketch*. This algorithm gives a bound on the probability that $\widehat{f_i}$ deviates from f_i, instead of a bound on the expectation of the gap. At a high level, the idea is to use multiple copies of the data structure above, and queries are evaluated as the minimum of the over-estimates returned by each structure. Further details are beyond the scope of this module.