Lecture 8: String Hashing

Objectives of this lecture

- Introduce the concept of hashing.
- Introduce hashing for strings.
- Combine hashing for strings with segment trees to design data structure for dynamic string processing.

1 Philosophy

Hashing is a general term that means a randomized procedure for compressing a large space of possible inputs down to a single integer. In this lecture we describe a specific way of hashing strings, and certain applications.

2 Example Problems

2.1 Hashing a string

Let $s = s_1 \ldots s_n$ be a string (assume that n is large, like a million or even large). Let's say we want to compress the string down to a single 32 or 64 bit integer. How do we do this?

Here is a *polynomial hash* introduced by Rabin and Karp. Let p be a prime number that fits in 32 bits (you can pick your favorite, turns out $10^9 + 7$ and $10^9 + 9$ are both primes that fit neatly in 32 bits). Let x be a random integer in the range $\{0, 1, \ldots, p-1\}$. Define the hash of s as:

$$H_x(s) = \sum_{i=1}^{n} s_i x^i \pmod{p}.$$

Here I am being a bit lazy and using s_i to denote an integer: for example let the letter a corresponds to $'1'$, b is $'2'$, etc.

Polynomial hashing is a powerful technique for string matching. We will informally argue that if strings s and t are different, then their hashes are not the same with high probability.

Collision analysis. Let $s = s_1 \dots s_n$ and $t = t_1 \dots t_n$. Let's understand if it's possible to get "unlucky" and for $H_x(s) = H_x(t)$ even if $s \neq t$. Expand out $H_x(s) = H_x(t)$:

$$\sum_{i=1}^{n} s_i x^i = \sum_{i=1}^{n} t_i x^i \pmod p$$

$$\sum_{i=1}^{n} (s_i - t_i) x^i = 0 \pmod p.$$

Consider the left hand side of the equation. This is a polynomial in x and it is a nonzero polynomial because $s_i \neq t_i$ for some i. The degree is at most n. Now we require that x is random and the following lemma.

Lemma 1: One-Dimensional Schwarz-Zippel Lemma

et $P(x)$ be a nonzero polynomial of degree at most n. Then the probability over $x \in \{0, 1, \dots, p-1\}$ that $P(x) = 0 \pmod p$ is at most n/p.

This follows from the fact that a polynomial of degree n over a finite field has at most n roots.

Thus as long as $p > 10n$ there's a 90% chance that the hashes don't collide. You can repeat the same algorithm over and over to increase the success probability.

In practice, you probably shouldn't pick x to be too small or its easy to manually construct hash collisions.

Note. If you didn't understand the last few minutes, that's fine. For now, it's OK to just take on faith that if two strings s and t are not the same, then their hashes are equal with very low probability (just assume 0). We won't discuss this for the rest of the lecture.

2.2 String matching

Problem: String matching

Let s and t be input strings. Find all locations where t appears as a substring of s, in time $O(|s| + |t|)$.

For notation let the length of s be n, length of t be m, and $n \leq m$. The idea is just to compute the hash of each length n substring of t and check if it matches the hash of s. In other words, for each $i \in \{1, 2, \dots, m-n+1\}$ we have to find

$$V[i] := \sum_{j=0}^{n-1} t[j+i] x^i.$$

To start just compute $V[m-n+1]$ directly. Now we can use the following formula to get the value of $V[i]$ from $V[i+1]$:

$$V[i] = x\left(V[i+1] - t[n+i] x^{n-1}\right) + t[i].$$

Thus we can find all $V[i]$'s in total time $O(m)$ as desired.

2.3 Dynamic Longest Common Prefix

Problem: Dynamic LCP

Let s_1 and s_2 be strings of length n. Design an algorithm that supports the following operations.

1. UPDATE(b, i, c), where $b \in \{1,2\}$ and $i \in \{1,2,\ldots,n\}$ and c is a character. This sets $s_b[i]$ to the character c.
2. LCP(i, j). Find the largest integer ℓ such that the string $s_1[i, i+1, \ldots, i+\ell]$ is equal to $s_2[j, j+1, \ldots, j+\ell]$.

We give an algorithm that supports UPDATE in time $O(\log n)$ and supports LCP in time $O(\log^2 n)$.

Hashing. Let p be a large prime and let $x \in \{1, \ldots, p-1\}$ be the base for the hash.

Binary search over ℓ. By binary searching over ℓ, we can reduce to checking equality: decide if $s_1[i, i+1, \ldots, i+\ell]$ is equal to $s_2[j, j+1, \ldots, j+\ell]$ as strings. If we can do this in time $O(\log n)$, then we can answer LCP in time $O(\log^2 n)$, since binary search uses $O(\log n)$ iterations.

Checking equality. The hashes of the two strings are respectively

$$s_1[i, i+1, \ldots, i+\ell] \to \sum_{t=0}^{\ell} s_1[i+t]x^t, \quad \text{and}$$
$$s_2[j, j+1, \ldots, j+\ell] \to \sum_{t=0}^{\ell} s_2[j+t]x^t.$$

At a high level, the idea is to use a segment tree (point update and range query) to able to query both of these values in $O(\log n)$ time per query.

Let's explain how to do this for s_1 (and s_2 is the same). Maintain an array $w[i]$ filled with $w[i] = s_1[i]x^i$. Upon UPDATE, just update the relevant entry of $w[i]$. To find the hash of $s_1[i, i+1, \ldots, i+\ell]$, call a range sum query on the range $[i, i+\ell]$ – this will give you the value

$$\sum_{t=0}^{\ell} s_1[i+t]x^{i+t} = x^i \sum_{t=0}^{\ell} s_1[i+t]x^t,$$

i.e., x^i times the desired hash values. There's two approaches going forwards, depending on how comfortable you are with modular inverses.

Modular inverse approach. The hash of $s_1[i, i+1, \ldots, i+\ell]$ is x^{-i}RANGESUM$(s_1, i, i+\ell) \pmod p$. You can compute x^{-i} by finding the modular inverse of x modulo p and taking it to the i-th power.

Without modular inverses. As described above, the hashes of $s_1[i, i+1, \ldots, i+\ell]$ and $s_2[j, j+1, \ldots, j+\ell]$ are given by $x^{-i}\textsc{RangeSum}(s_1, i, i+\ell) \pmod p$ and $x^{-j}\textsc{RangeSum}(s_2, j, j+\ell) \pmod p$ respectively. So we want to check if

$$x^{-i}\textsc{RangeSum}(s_1, i, i+\ell) \equiv x^{-j}\textsc{RangeSum}(s_2, j, j+\ell) \pmod p.$$

We can instead multiply out the negative exponents to get an equivalent equation to check:

$$x^{j}\textsc{RangeSum}(s_1, i, i+\ell) \equiv x^{i}\textsc{RangeSum}(s_2, j, j+\ell) \pmod p.$$

This way we can avoid negative exponents.