String Algorithms

Ajay Kumar

Date 3 Feb 2023

Hashing

- Hashing algorithms are helpful in solving a lot of problems.
  We want to solve the problem of comparing strings efficiently. The brute
- force way of doing so is just to compare the letters of both strings, which has a time complexity of $O(min(n_1, n_2))$ if n1 and n2 are the sizes of the two strings.
- We want to do better. The idea behind strings is the following: we convert each string into an integer and compare those instead of the strings. Comparing two strings is then an O(1) operation.
- For the conversion, we need a so-called **hash function**. The goal of it is to convert a string into an integer, the so-called **hash** of the string.
- The following condition has to hold: if two strings s and t are equal (s = t), then also their hashes have to be equal (hash(s) = hash(t)). Otherwise, we will not be able to compare strings.

- Notice, the opposite direction doesn’t have to hold. If the hashes are equal (hash(s) == hash(t)), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply hash(s) = 0 for each s.
- Now, this is just a stupid example, because this function will be completely useless, but it is a valid hash function. The reason why the opposite direction doesn’t have to hold, if because there are exponential many strings.
- If we only want this hash function to distinguish between all strings consisting of lowercase characters of length smaller than 15, then already the hash wouldn’t fit into a 64-bit integer (e.g. unsigned long long) any more, because there are so many of them.
- And of course, we don’t want to compare arbitrary long integers, because this will also have the complexity O(n).
- So usually we want the hash function to map strings onto numbers of a fixed range [0, m), then comparing strings is just a comparison of two integers with a fixed length. And of course, we want hash(s) $\neq$ hash(t) to be very likely if s$\neq$ t.
- That’s the important part that you have to keep in mind. Using hashing will not be 100% deterministically correct, because two complete different strings might have the same hash (the hashes collide).
- However, in a wide majority of tasks, this can be safely ignored as the probability of the hashes of two different strings colliding is still very small. And we will discuss some techniques in this

Problem:

- You are given two strings s, t of lengths $N(\leq 10^5)$ that consist of lowercase letters. Determine if they are the same.

- Let MOD=$10^9 + 7$. Choose an integer r uniformly at random from the interval [0, MOD). Compute the sum of $r^i * s[i]$ over all i.
- This is called the rolling hash of s. We can consider it as a polynomial of r, let's denote it as S(r). Similarly, define T(r). The hash collision happens when s and t are different strings (thus S and T are different polynomials), but unluckily S(r) = T(r). Here we use Schwartz-Zippel lemma:

- Let $P(x_1, ..., x_k)$ be a non-zero (multi-variable) polynomial on a finite field F_{MOD}. If the variables are chosen independently and uniformly at random, the probability that $P(x_1, ..., x_k) = 0$ is at most D/MOD, where D is the total degree of the polynomial.

- In the case of rolling hash, we are only interested in single-variable polynomials. From the lemma above, we can prove that the collision probability is at most $N/MOD(<= 1/10^4)$. Thus I'd say this is a "good hash". My intuition tells that in practice the probability is as small as 1/MOD, but that looks at least very hard to prove. Let me give an interesting example that shows the intuition is not always correct. In the lemma F_{MOD} must be a field, and MOD must be a prime. What happens if the modulo is not prime, for example 2^64 (this would simplify the implementation). It turns out that the rolling hash doesn't work well for a certain case if the modulo is 2^64. Detailed description on Codeforces.