CP-Algorithms

Page Authors

Search

String Hashing

Table of Contents

Hashing algorithms are helpful in solving a lot of problems.

The problem we want to solve is the problem, we want to compare 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 n_1 and n_2 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 a 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 ($\text{hash}(s) = \text{hash}(t)$). Otherwise we will not be able to compare strings.

Notice, the opposite direction doesn't have to hold. It the hashes are equal ($\text{hash}(s) = \text{hash}(t)$), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply $\operatorname{hash}(s) = 0$ 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 comparison of two integers with fixed length. And of course we want $\operatorname{hash}(s) \neq \operatorname{hash}(t)$ 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 article how to keep the probability of collisions very low.

Calculation of the hash of a string

The good and widely used way to define the hash of a string s of length n is

$$\begin{aligned} \operatorname{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m, \end{aligned}$$

where p and m are some chosen, positive numbers. It is called a **polynomial rolling hash function**.

It is reasonable to make p a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of English alphabet, $p = 31$ is a good choice. If the input may contain both uppercase and lowercase letters, then $p = 53$ is a possible choice. The code in this article will use $p = 31$.

Obviously m should be a large number, since the probability of two random strings colliding is about $\approx \frac{1}{m}$. Sometimes $m = 2^{64}$ is chosen, since then the integer overflows of 64 bit integers work exactly like the modulo operation. However there exists a method, which generates colliding strings (which work independent from the choice of p). So in practice $m = 2^{64}$ is not recommended. A good choice for m is some large prime number. The code in this article will just use $m = 10^9 + 9$. This is a large number, but still small enough so that we can perform multiplication of two values using 64 bit integers.

Here is an example of calculating the hash of a string s, which contains only lowercase letters. We convert each character of s to an integer. Here we use the conversion $a \rightarrow 1$, $b \rightarrow 2$, $\dots$, $z \rightarrow 26$. Converting $a \rightarrow 0$ is not a good idea, because then the hashes of the strings a, aa, aaa, $\dots$ all evaluate to 0.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' +
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Precomputing the powers of p might give a performance boost.

Example tasks

Search for duplicate strings in an array of strings

Problem: Given a list of n strings s_i, each no longer than m characters, find all the duplicate strings and divide them into groups.

From the obvious algorithm involving sorting the strings, we would get a time complexity of $O(nm \log n)$ where the sorting requires $O(n \log n)$ comparisons and each comparison take $O(m)$ time. However by using hashes, we reduce the comparison time to $O(1)$, giving us an algorithm that runs in $O(nm + n \log n)$ time.

We calculate the hash for each string, sort the hashes together with the indices, and then group the indices by identical hashes.

```
vector<vector<int>> group_identical_strings(ve
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};

    sort(hashes.begin(), hashes.end());

    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashe
            groups.emplace_back();
        groups.back().push_back(hashes[i].seco
    }
    return groups;
}
```

Fast hash calculation of substrings of given string

Problem: Given a string s and indices i and j, find the hash of the substring $s[i \dots j]$.

By definition, we have:

$$\text{hash}(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \mod m$$

Multiplying by p^i gives:

$$\begin{aligned} \text{hash}(s[i \dots j]) \cdot p^i &= \sum_{k=i}^{j} s[k] \cdot p^k \mod m \\ &= \text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1] \end{aligned}$$

So by knowing the hash value of each prefix of the string s, we can compute the hash of any substring directly using this formula. The only problem that we face in calculating it is that we must be able to divide $\text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1])$ by p^i. Therefore we need to find the modular multiplicative inverse of p^i and then perform multiplication with this inverse. We can precompute the inverse of every p^i, which allows computing the hash of any substring of s in $O(1)$ time.

However, there does exist an easier way. In most cases, rather than calculating the hashes of substring exactly, it is enough to compute the hash multiplied by some power of p. Suppose we have two hashes of two substrings, one multiplied by p^i and the other by p^j. If $i < j$ then we multiply the first hash by p^{j-i}, otherwise we multiply the second hash by p^{i-j}. By doing this, we get both the hashes multiplied by the same power of p (which is the maximum of i and j) and now these hashes can be compared easily with no need for any division.

Applications of Hashing

Here are some typical applications of Hashing:

- Rabin-Karp algorithm for pattern matching in a string in $O(n)$ time
- Calculating the number of different substrings of a string in $O(n^2 \log n)$ (see below)
- Calculating the number of palindromic substrings in a string.

Determine the number of different substrings in a string

Problem: Given a string s of length n, consisting only of lowercase English letters, find the number of different substrings in this string.

To solve this problem, we iterate over all substring lengths $l = 1 \dots n$. For every substring length l we construct an array of hashes of all substrings of length l multiplied by the same power of p. The number of different elements in the array is equal to the number of distinct substrings of length l in the string. This number is added to the final answer.

For convenience we will use $h[i]$ as the hash of the prefix with i characters, and define $h[0] = 0$.

```
int count_unique_substrings(string const& s) {
    int n = s.size();

    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i] - 'a' + 1) * p_

    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l] + m -
            cur_h = (cur_h * p_pow[n-i-1]) % m
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
    return cnt;
}
```

Improve no-collision probability

Quite often the above mentioned polynomial hash is good enough, and no collisions will happen during tests. Remember, the probability that collision happens is only $\approx \frac{1}{m}$. For $m = 10^9 + 9$ the probability is $\approx 10^{-9}$ which is quite low. But notice, that we only did one comparison. What if we compared a string s with 10^6 different strings. The probability that the at least one collision happens is now $\approx 10^{-3}$. And if we want to compare 10^6 different strings with each other (e.g. by counting how many unique strings exists), then the probability of at least one collision happening is already ≈ 1. It is pretty much guaranteed that this task will end will a collision and returns the wrong result.

There is a really easy trick to get better probabilities. We can just compute two different hashes for each string (by using two different p, and/or different m, and compare these pairs instead. If m is about 10^9 for each of the two hash functions, than this is more or less equivalent as having one hash function with $m \approx 10^{18}$. When comparing 10^6 strings with each other, the probability that at least one collision happens is now reduced to $\approx 10^{-6}$.

Practice Problems

- A Needle in the Haystack - SPOJ
- Double Profiles - Codeforces
- Password - Codeforces
- SUB_PROB - SPOJ
- INSQ15_A
- SPOJ - Ada and Spring Cleaning
- GYM - Text Editor
- 12012 - Detection of Extraterrestrial
- Codeforces - Games on a CD
- UVA 11855 - Buzzwords
- Codeforces - Santa Claus and a Palindrome
- Codeforces - String Compression
- Codeforces - Palindromic Characteristics
- SPOJ - Test
- Codeforces - Palindrome Degree
- Codeforces - Deletion of Repeats