MAXimal

Page Authors

String Hashing

Hashing algorithms are helpful in solving a lot of problems. But they have a big flaw that sometimes they are not 100% deterministically correct because when there are plenty of strings, hashes may 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.

Calculation of the hash of a string

The best and most widely used way to define the hash of a string S is the following function:

$hash(S) = S[0] + S[1] \cdot P + S[2] \cdot P^2 + S[3] \cdot P^3 + \ldots + S[N] \cdot P^N$ where P is some number.

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 $P = 31$.

Of course, it is desirable to store the hash value in the largest numeric type - int64 i.e. unsigned long long. It is obvious that if the length of the string gets to about 20 characters, the hash value will overflow. But the key point to notice here is that we do not care about these overflows, as these overflows are equivalent to keeping the hash modulo 2^{64} at all times.

Example of calculating the hash of a string s, which contains only lowercase letters:

```
const int p = 31;
unsigned long long hash = 0, p_pow = 1;
for (size_t i = 0; i <s.length (); ++ i)
{
    // Generally, it is recommended to take the "value" of 'a' to be 1
    // so that strings like "aaaaa" do not all hash to the value 0
    hash + = (s [i] - 'a' + 1) * p_pow;
    p_pow * = p;
}
```

In the majority of tasks, it makes sense to first calculate all the necessary powers of P and store them in an array.

Example tasks

Search for duplicate strings in an array of strings

Problem: Given a list of strings $S[1..N]$, each no longer than M characters, find all the duplicate strings and divide them into groups so that each group contains only the same string.

From the obvious algorithm involving sorting the strings, we would get a time complexity of $O(NMlogN)$ where the sorting requires $O(NlogN)$ 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 $O(NM + NlogN)$ time.

Algorithm: Calculate the hash of each string, and sort the strings on the basis of these hashes.

```
vector <string> s (n);
// ... Input the strings...

// Calculate all powers of p up to 10,000 - the maximum length of a string
const int p = 31;
vector <unsigned long long> p_pow (10000);
p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size (); ++ i)
    p_pow [i] = p_pow [i-1] * p;

// Calculate the hash of each string
```

```
// hashes[] stores the hash value and the index of the string in the array s
vector <pair <unsigned long long, int>> hashes (n);
for (int i = 0; i <n; ++ i)
{
    unsigned long long hash = 0;
    for (size_t j = 0; j <s [i] .length (); ++ j)
        hash + = (s [i] [j] - 'a' + 1) * p_pow [j];
    hashes [i] = make_pair (hash, i);
}

// Sort by hashes
sort (hashes.begin (), hashes.end ());

// Display the answer
for (int i = 0, group = 0; i <n; ++ i)
{
    if (i == 0 || hashes [i] .first! = hashes [i-1] .first)
        cout << "\n Group" << ++ group << ":";
    cout << " " << hashes [i] .second;
}
```

Fast calculation of hashes of substrings of given string S

Problem: Given a string S and indices I and J, find the hash of the substring $S[I..J]$.

By definition, we have:

$$H[I..J] = S[I] + S[I+1] * P + S[I+2] * P^2 + ... + S[J] * P^(J-I)$$

from:

$H[I..J] * P^I = S[I] * P^I + ... + S[J] * P^J$, that is

$$H[I..J] * P^I = H[0..J] - H[0..I-1]$$

The above property is very important, because from the above property, it follows that, knowing only the hashes of all prefixes of string S, we can calculate the hash of any substring in $O(1)$.

The only problem that we face in calculating these hashes is that we must be able to divide $H[0..J] - H[0..I-1]$ by P^I. This is not easy, because in general we compute hashes modulo 2^{64}. Therefore, for division by P^I, we need to find the modular multiplicative inverse of P^I (using the Extended Euclidean Algorithm) and then perform multiplication with this inverse.

However, there does exist an easier way. In most cases, rather than calculating the hashes of substring exactly, we calculate the hash multiplied by some particular power of P, which is sufficient for our purposes.

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.

For example, the following code calculates the hashes of all prefixes and then compares any two substrings in $O(1)$:

```
// input data
// s - the input string
// len - the length of the two substrings to be compared
// i1 - the start index of the first substring
// i2 - the start index of the second substring
string s; int i1, i2, len;

// calculate all powers of p
const int p = 31;
vector <unsigned long long> p_pow (s.length ());
p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size (); ++ i)
    p_pow [i] = p_pow [i-1] * p;

// Calculate the hashes of all prefixes
vector <unsigned long long> h (s.length ());
```

```
for (size_t i = 0; i <s.length (); ++ i)
{
    h [i] = (s [i] - 'a' + 1) * p_pow [i];
    if (i) h [i] + = h [i-1];
}

// Get the hashes of two substrings
unsigned long long h1 = h [i1 + len-1];
if (i1) h1 - = h [i1-1];
unsigned long long h2 = h [i2 + len-1];
if (i2) h2 - = h [i2-1];

// Get the two hashes multiplied by the same power of P and then
// compare them
if (i1 <i2 && h1 * p_pow [i2-i1] == h2 ||
    i1> i2 && h1 == h2 * p_pow [i1-i2])
    cout << "equal" << endl;
else
    cout << "different" << endl;
```

Applications of Hashing

Here are some typical applications of Hashing:

- Rabin-Karp Algorithm for string matching in a string in $O(N)$ time.
- Calculating the number of different substrings of a string in $O(N^2 logN)$ (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, first let's apply the brute force approach over substring length one by one: $L = 1..N$.

For every substring length L, we construct an array of hashes of all substrings of length L. Further, we get these hashes multiplied by the same power of P, and then sort the array. 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.

Implementation

```
string s; // Input string
int n = (int) s.length ();

// Calculate all powers of p
const int p = 31;
vector <unsigned long long> p_pow (s.length ());
p_pow [0] = 1;
for (size_t i = 1; i <p_pow.size (); ++ i)
    p_pow [i] = p_pow [i-1] * p;

// Calculate the hashes of all prefixes
vector <unsigned long long> h (s.length ());
for (size_t i = 0; i <s.length (); ++ i)
{
    h [i] = (s [i] - 'a' + 1) * p_pow [i];
    if (i) h [i] + = h [i-1];
}

int result = 0; // stores the final answer

// Iterate over length of substrings from 1 to n
for (int l = 1; l <= n; ++ l)
{
    // Need to find the number of distinct substrings of length l
```

```
        // Get the hashes for all substrings of length l
        vector <long long> hs (n-l + 1);
        for (int i = 0; i <n-l + 1; ++ i)
        {
            long long cur_h = h [i + l-1];
            if (i) cur_h - = h [i-1];
            // Get all the hashes to the same degree
            cur_h * = p_pow [n-i-1];
            hs [i] = cur_h;
        }

        // Count the number of different hashes
        sort (hs.begin (), hs.end ());
        hs.erase (unique (hs.begin (), hs.end ()), hs.end ());
        result + = (int) hs.size ();
    }

    cout << result;
```

Practice Problems

- A Needle in the Haystack - SPOJ
- Double Profiles - Codeforces
- Password - Codeforces
- SUB_PROB - SPOJ
- INSQ15_A

(c) 2014 translation by http://github.com/e-maxx-eng