Hashing

Jubayer Nirjhor

May 2019

1 Introduction

One fine morning, you're at the school office trying to pay some fees. The accountant is going through a huge notebook full of student names, trying to match each one with yours. It's already been 20 minutes and the morning doesn't seem fine anymore.

Now imagine the same scenario, but the huge notebook is sorted by a unique identity number for each student, starting from 1. You tell yours to the accountant, and she's got the match in less than a minute.

This simple trick is an example of hashing. Hashing is the process of mapping data of arbitrary size to data of a fixed, preferably small, size. The map used in this process is called a *hash function*. Using a roll number for each student is a typical example of a hash function. Hash functions are used to compress large data into small ones while trying to preserve useful properties as much as possible. This allows for fast insertion, deletion, and look up without the need of any sophisticated data structure or algorithm.

Although the mapped data can be anything, we typically choose the integers because of the computational simplicity they offer. Different choices of hash functions in different contexts can be surprisingly powerful. In this article, we'll look at some of these hash functions and their applications.

2 Hash Functions

Let's consider a map $f : A \to B$. This map transforms some data in set A to some data in set B. How do we decide if it's a good (useful) hash function?

- Well first of all, it must be a function: we must have $f(x) = f(y)$ whenever $x = y$.
- We want the hash values to be sufficiently small. So the set B should have a small enough size in a practical sense.
- The definition of the function f should be simple enough to quickly compute the values at different points.

- Hash functions are less likely to be bijective because the size of set A is typically way larger than the size of set B. We utilize the fact that in practice, we'll encounter a sufficiently smaller subset of set A. So if our hash function maps the elements of A uniformly over the set B, we're less likely to encounter two different elements $x, y \in A$ with $f(x) = f(y)$. In other words, we want our hash function to face less *collisions*.

Example time! Let us denote the set of all length 3 words made up of lowercase English characters by $\mathbb{S}$. This set contains exactly 26^3 elements (why?). Let's consider some hash functions $f : \mathbb{S} \to \mathbb{N}$.

- $f(s) = 0$ for any $s \in \mathbb{S}$. This is a perfectly valid hash function because, well, it's a function. But as you can guess, it's a useless one. It cannot differentiate between the words `abc` and `efg` since both these words map to the same value 0.

- $f(s) =$ sum of positions of each letter in the word s. We assume the positions of `a`, `b`, ..., `z` as $1,\ 2, ...,\ 26$. This is better than the previous one. We have $f(\texttt{abc}) = 1+2+3 = 6$ and $f(\texttt{efg}) = 5 + 6 + 7 = 18$. Great! Different words seem to obtain different values. But we can quickly spot that this function doesn't depend on the order of characters in the word. For example, we have $f(\texttt{abc}) = f(\texttt{bac}) = 6$. So this is still not good enough. But this one definitely has less *collisions* than the previous one (duh).

  We'll look at some good solutions in the next section.

3 String Hashing

So our target is to map a text T of arbitrary finite size assuming a finite alphabet Σ, to an integer. As we've seen earlier, sum of character positions is far from suitable since they don't at all encode the ordering of characters involved. A simple solution is, instead of naively adding the characters, to assign a *weight* to each character based on its position. A safe choice is choosing a fixed integer b and assigning the weight b^i at position i. So the string $T = T_0T_1T_2 \cdots T_{n-1}$ (each T_i is the position of the i-th character in T) would get mapped to

$$f(T) = T_0 + T_1b + T_2b^2 + \cdots + T_{n-1}b^{n-1} = \sum_{k=0}^{n-1} T_k b^k$$

Yes, we're simply evaluating the polynomial with the text characters as coefficients at the point $x = b$. If we take $b > |\Sigma|$, then this value essentially gives us the unique decimal value of base b representation of the text.

All quite good so far. But the catch is that the value of f can get quite large as the text size increases. This property is against that of a good hash function: we want our mapped values to stay sufficiently small. Easy fix: take the resulting value modulo some small number m, so the mapped values always stay in range $[0, m)$.

But how to ensure less collisions? We want the mapped values to be uniformly distributed over $[0, m)$. But notice that for a fixed integer t

$$\{kt \bmod m : k \in \mathbb{Z}\} = \left\{ k \gcd(t, m) : 0 \leq k < \frac{m}{\gcd(t, m)} \right\}$$

This just means that the multiples of a fixed number t modulo m only covers the multiples of $\gcd(t, m)$ that are less than m (why?). So to ensure less collisions, we should ensure $\gcd(t, m) = 1$ for maximum number of integers t. Which numbers m have this property? The primes! So we should take a large enough prime number as m.

With this base and modulus pair (b, m) we have a sufficiently good hash function for any text T, known as a *polynomial rolling hash function*. For example, if the texts are generated from lowercase English letters only (so $|\Sigma| = 26$), we might take $b = 29$ and $m = 10^9 + 7$ (a prime), and have a good enough hash function that can be computed in linear time.

Data: A text T, a base b, a modulus m
Result: An integer, the hash value
$hash \leftarrow 0$
$power \leftarrow 1$
for $i \leftarrow 0$ **to** $|T| - 1$ **do**
  $hash \leftarrow (hash + power \times T_i) \bmod m$
  $power \leftarrow (power \times b) \bmod m$
end
return *hash*

Algorithm 1: Rolling Hash

Notice that the structure of this hash function allows us to quickly compute the hash of any substring of the given text T. Indeed, let us compute the prefix hash values hash_0, hash_1, ..., $\text{hash}_{|T|}$ where $\text{hash}_0 = 0$ and for any $i > 0$

$$\text{hash}_i = T_0 + T_1 b + \cdots + T_{i-1} b^{i-1} = \text{hash}_{i-1} + T_{i-1} b^{i-1}$$

which can be computed in a linear pass of the text T (because of the recursive definition). Now consider integers $0 \leq l \leq r < |T|$ and we want to compute the value $\text{hash}(l, r)$ of the substring consisting of the range $[l, r]$ in text T. Then

$$\begin{aligned}
\text{hash}(l, r) &= T_l + T_{l+1} b + \cdots + T_r b^{r-l} \\
&= b^{-l} \left(T_l b^l + T_{l+1} b^{l+1} + \cdots + T_r b^r \right) \\
&= b^{-l} \left[(T_0 + T_1 b + \cdots + T_r b^r) - \left(T_0 + T_1 b + \cdots + T_{l-1} b^{l-1} \right) \right] \\
&= b^{-l} \left[\text{hash}_{r+1} - \text{hash}_l \right]
\end{aligned}$$

If we precompute the powers of b^{-1} (the modular inverse of b) and the values hash_i, then we can compute any substring hash in $\mathcal{O}(1)$ time.

Enough chitchat. Let's get down to some applications!

- *Given a text T and a pattern P, find all positions where this pattern appears in the text.* For example: the pattern $P =$ `aba` occurs 3 times in the text $T =$ `ababacabad`. Naively matching character by character starting from each position in T is very slow since the time complexity is $\mathcal{O}(|T||P|)$. However, using the hash_i values as discussed above, we can simply obtain the hash of the substring starting at each position in T and match it with the hash of P which results in an $\mathcal{O}(|T| + |P|)$ algorithm. This simple algorithm is known as the *Rabin-Karp algorithm.*

- *Given a text T and q queries asking whether the substring $T[l...r]$ is a palindrome, answer them.* A string is a palindrome if and only if the reversed string matches the original one. So we keep the hash_i values for both the text T and the reverse of the text T. This way we can quickly compute both the hashes of substring $T[l...r]$ and the reverse of $T[l...r]$ and compare them. This results in an $\mathcal{O}(|T| + q)$ algorithm which is far better than the naive $\mathcal{O}(|T|q)$ algorithm.

Here's a sample C++ code that solves the pattern matching problem.

```
#include <bits/stdc++.h>

using namespace std;

const int B = 29;
const int N = 100010;
const int MOD = 1e9 + 7;

int bigMod (int a, int e) {
  if (e == -1) e = MOD - 2;
  int ret = 1;
  while (e) {
    if (e & 1) ret = ret * 1LL * a % MOD;
    a = a * 1LL * a % MOD, e >>= 1;
  }
  return ret;
}

char t[N], p[N];
int _hash[N], inv[N];

inline int range (int l, int r) {
  int ret = (_hash[r + 1] - _hash[l]) * 1LL * inv[l] % MOD;
  if (ret < 0) ret += MOD;
  return ret;
}

int main() {
  inv[0] = 1, inv[1] = bigMod(B, -1);
  for (int i = 2; i < N; ++i) {
```

```
    inv[i] = inv[i - 1] * 1LL * inv[1] % MOD;
  }

  scanf("%s", t);
  int n = strlen(t);
  int power = 1;
  for (int i = 0; i < n; ++i) {
    _hash[i + 1] = (_hash[i] + power * 1LL * (t[i] - 'a' + 1)) % MOD;
    power = power * 1LL * B % MOD;
  }

  scanf("%s", p);
  int m = strlen(p);
  int pattern_hash = 0;
  power = 1;
  for (int i = 0; i < m; ++i) {
    pattern_hash = (pattern_hash + power * 1LL * (p[i] - 'a' + 1)) % MOD;
    power = power * 1LL * B % MOD;
  }

  for (int i = 0; i + m - 1 < n; ++i) {
    if (range(i, i + m - 1) == pattern_hash) {
      printf("Match found starting at position %d.\n", i);
    }
  }
  return 0;
}
```

We conclude this section by noting that since the polynomial rolling hash function takes in values in $[0, m)$, for k strings the probability of no collision is around

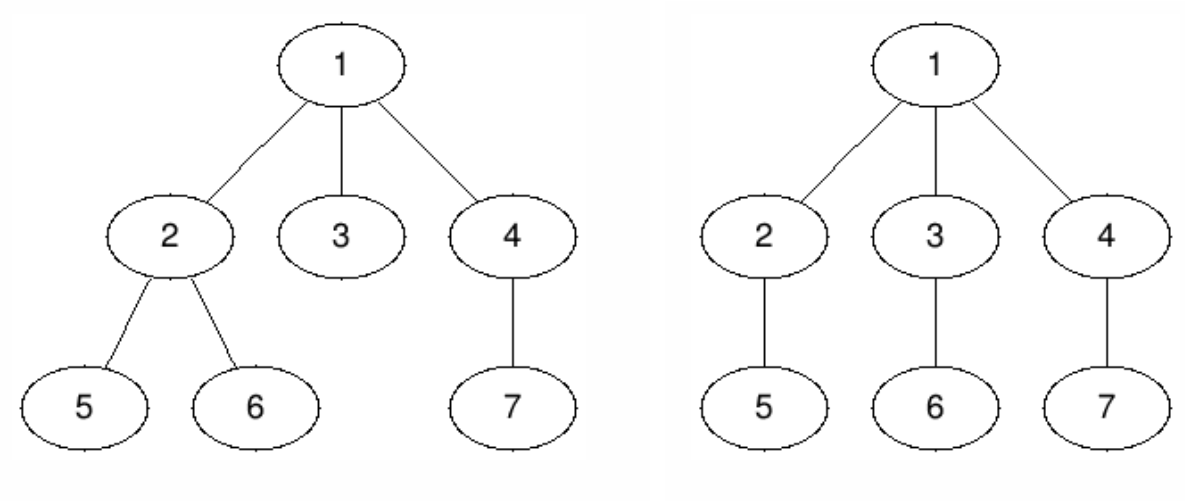
$$\frac{m}{m} \times \frac{m-1}{m} \times \frac{m-2}{m} \times \cdots \times \frac{m-k+1}{m}$$

which is basically 0 for $k \approx 10^6$ and $m \approx 10^9$. We can easily fix this by using two different (b, m) pairs to compute two different hash values and use the pair of hash values for comparison. Then the probability of at least one collision becomes very low. Of course, we can compute more hash values for better safety as well, in exchange of more computation time.

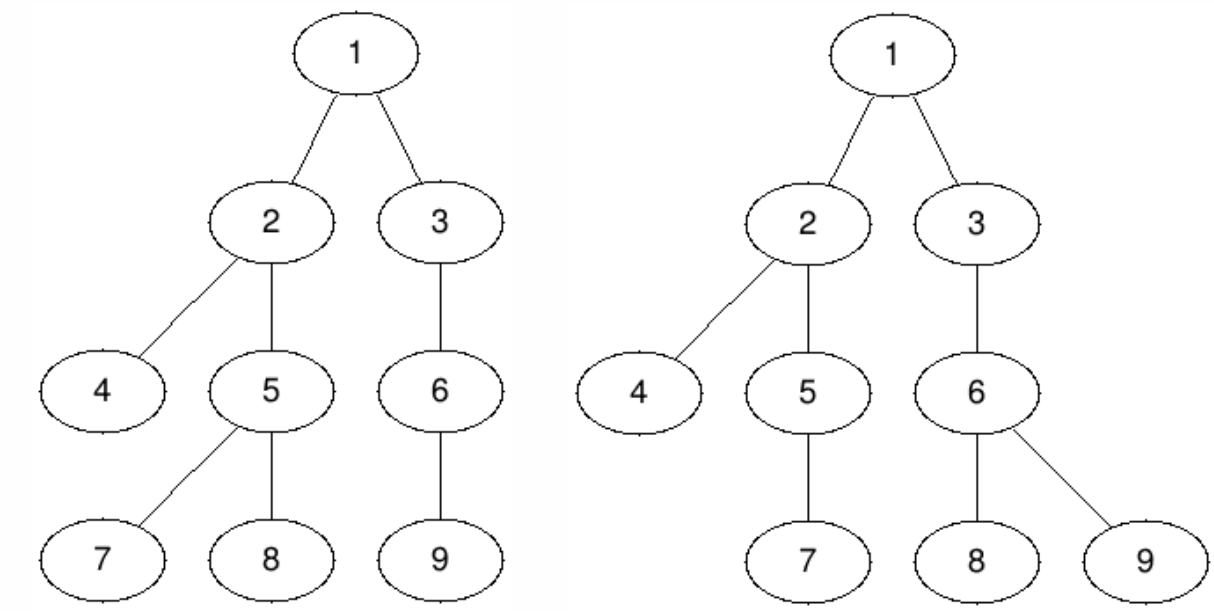
4 Hashing Rooted Trees

Given two trees, we call them isomorphic to each other if they are structurally the same trees. Formally, they have the same number of vertices n and there exists a permutation p of $\{1, 2, ..., n\}$ such that there's an edge (i, j) in the first tree if and only if there's an edge (p_i, p_j) in the second tree. Now our target is to test if two rooted trees are isomorphic.

Let's see. What property remains same in two isomorphic rooted trees? Is it the number of vertices on each level? Here's a counterexample.



Same number of vertices on each level, yet these trees are not isomorphic to each other. Maybe we need the degree sequence to be same? Turns out false as well.



These two trees have the same degree sequence of vertices, yet not isomorphic to each other. The property we need is actually a recursive definition. Let's denote the root by r and its children by $c_1, c_2, ..., c_k$. Let's denote the hash value of the subtree rooted at u by $f(u)$. Then $f(r)$ should merge the values of $f(c_1), f(c_2), ..., f(c_k)$ in an unordered fashion.

Before diving into the solution, we state a fact that we'll use in the analysis.

Schwartz–Zippel Lemma. *Consider a nonzero polynomial P on n variables of degree d over a field $\mathbb{F}$. For any finite $S \subseteq \mathbb{F}$, the probability of a random point in S^n being a root of P is at most $\frac{d}{|S|}$.*

The proof is left to the reader as an exercise (hint: induct on the number of variables). As a consequence, the probability of two different polynomials P, Q colliding on the same point x modulo m is simply the probability of x being a root of $P - Q$ modulo m, which is $\frac{\max\{\deg P, \deg Q\}}{m}$.

Now let's consider a simpler problem: given two lists of same size $\{a_1, a_2, ..., a_n\}$ and $\{b_1, b_2, ..., b_n\}$, are these two the same list? Of course, there's an $\mathcal{O}(n \lg n)$ deterministic solution that sorts both lists and checks if the values at each position match. But there's a linear time solution that uses hashing. We take a prime modulus m and a random integer $t \in [0, m)$. Then we simply compute the values $(t + a_1)(t + a_2) \cdots (t + a_n)$ and $(t + b_1)(t + b_2) \cdots (t + b_n)$, both modulo m and compare them. Since these are two different polynomials of degree n evaluated at t, the probability of collision is $\frac{n}{m}$, which is somewhat small for $n \approx 10^5$ and $m \approx 10^9$ (can be improved by more hashes).

We can use the same idea of assigning polynomials in case of rooted trees as well. Suppose the tree has depth h, then the hash function will be a polynomial on h variables $x_1, ..., x_h$. Let's assign the tree on a single vertex the hash value 1. So the leaves get mapped to 1. Now we recursively define the hash value of the subtree rooted at u with children $c_1, c_2, ..., c_k$ whose depth is h_u by

$$f(u) = (x_{h_u} + f(c_1))(x_{h_u} + f(c_2)) \cdots (x_{h_u} + f(c_k))$$

similar to the list problem. This definition combines the children hash values in an unordered fashion, and we can guarantee the uniqueness of the polynomial for each tree up to isomorphism because every polynomial has a unique factorization. Furthermore, notice that this polynomial has degree d on a tree with d leaves. Now we simply evaluate this polynomial at a random point in $\{0, 1, ..., m-1\}^h$, so the probability of collision is $\frac{d}{m}$ by the above lemma: pretty small (again, can be improved by more hashes).

Finally, here's some C++ code testing if two rooted trees are isomorphic (assuming both are rooted at 1).

```
#include <bits/stdc++.h>

using namespace std;

const int N = 100010;
const int MOD = 1e9 + 7;

int n, x[N], h[N];
vector <int> g[2][N];
```

```
int get (int id, int u = 1, int from = -1) {
  vector <int> childs;
  for (int v : g[id][u]) if (v - from) {
    childs.emplace_back(get(id, v, u));
    h[u] = max(h[u], 1 + h[v]);
  }
  if (childs.empty()) return 1;
  int ret = 1;
  for (int value : childs) ret = ret * 1LL * (x[h[u]] + value) % MOD;
  return ret;
}

int main() {
  for (int i = 0; i < N; ++i) {
    x[i] = rand() * 1LL * rand() % MOD;
  }
  cin >> n;
  for (int i = 1; i < n; ++i) {
    int u, v;
    scanf("%d %d", &u, &v);
    g[0][u].emplace_back(v);
    g[0][v].emplace_back(u);
  }
  for (int i = 1; i < n; ++i) {
    int u, v;
    scanf("%d %d", &u, &v);
    g[1][u].emplace_back(v);
    g[1][v].emplace_back(u);
  }
  int firstTree = get(0);
  int secondTree = get(1);
  puts(firstTree == secondTree ? "Isomorphic" : "Not Isomorphic");
  return 0;
}
```

5 Big Hashes Can Be Useful!

It's not always true that hashes should only be smaller than the original data. Sometimes mapping to bigger data helps solve problems as well. Let's go through an example.

- *Given a list of m edges on a graph of n vertices. You're given q queries each asking if each connected component of the graph formed only by the edges with indices in range $[l, r]$ contains an Eulerian cycle. An Eulerian cycle is a path that starts and finishes at the same vertex and visits each edge exactly once.*

A necessary and sufficient condition for a connected graph having an Eulerian cycle is each vertex having an even degree (try to prove this). Having this, our problem reduces

to checking if each vertex has an even degree in the graph formed by edges in range $[l, r]$. But the degree of a vertex is the number of times it appears on the edge list! So we have a simpler problem to solve: given an array and queries giving a range $[l, r]$, check if each number in index range $[l, r]$ has an even frequency.

This problem admits an interesting, although somewhat wrong, solution. Let's compute the prefix xor of the array values: we compute $p_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$ for each $1 \leq i \leq n$ with $p_0 = 0$. Then the xor of values in range $[l, r]$ can be computed simply as $p_r \oplus p_{l-1}$ (recall that xor is associative and $x \oplus x = 0$ for any x). Now if every element appears an even number of times on range $[l, r]$, what can we say about their xor? It's zero! Because the values cancel in pairs. So we only need to check if $p_r \oplus p_{l-1}$ is zero and we're done!

Well, not quite! While the xor of the elements being zero is a necessary condition, it's not a sufficient one. For example, the xor of the values $\{1, 2, 3\}$ is 0, although each one appears an odd number of times. The problem is, there are certain sets of values that just happen to have an even number of bits on each position. This should be rare, but such a sequence can definitely be there.

But notice that we only care about the frequency of each value, not the value itself. So if we replace the same values by some other value, it doesn't affect our problem. Let's hash each different number in the array to a very large unique random value. This way, although the problem remains same, the probability of such 'bad' sequences occurring becomes really really low (we need an even number of bits on 64 bit positions, an almost impossible requirement when the values are chosen at random). So our previous solution will work well.

This allows us to process the queries in the very efficient $\mathcal{O}(m+q)$ time and $\mathcal{O}(m)$ space.

6 Further Reads

- *Birthday problem.* `https://en.wikipedia.org/wiki/Birthday_problem`
- *String hashing.* `https://cp-algorithms.com/string/string-hashing.html`
- *Why 2^t is a bad modulus.* `https://codeforces.com/blog/entry/4898`
- *Designing strong hash functions.* `https://codeforces.com/blog/entry/60442`