Hashing Techniques

Nicholas Ko {__jk__}

2026-07-01

Table of Contents

What is Hashing?

Dictionary Definition

Hashing is a one-way encryption technique that converts data into a “hashed value”.

Alright, but what does this mean?

One-sentence summary of the whole lecture

The ultimate purpose of hashing things in competitive programing, is to have the ability to check whether two things A and B are equal, without actually closely inspecting into them in each query.

Why is this nontrivial?

Suppose the two "things" A and B are strings. We can just use `A == B` in our code to test whether they are equal. Why bother having an entire lecture based on this?

The definition of "equality" may not be straightforward

Sometimes, we might define a string A to be equal to another string B, if

- they have the same frequency of letters, or
- A is some cyclic rotation of B.

Then, there is no longer an "obvious" way to check for equality. In these scenarios, hashing can quickly prove to be useful.

Why is this nontrivial?

The “equality” we are talking about can be up to some (self-defined) equivalence relation; it is not restricted to discussing whether the things are completely identical or not.

In the above example, we can use `A == B` to test whether the strings A and B are equal or not. Immediately, we can see a few limitations:

- If A and B are some other data types (e.g. graphs), there might not be an in-built == function.
- The time complexity is linear in the string length. We want to achieve a better time complexity.

Hash Functions

Suppose we are interested in testing whether two arrays of positive integers A and B are equal or not.

What are some **necessary conditions** for A and B to be equal? (Remember, the whole point of hashing is not to closely inspect into the arrays.)

- their size has to be the same,
- their smallest element has to be the same,
- their sum of elements has to be the same,
- ... and a zillion more.

The role of hash functions is to **describe such necessary conditions**.

Examples of Hash Functions

The examples that we have just discussed give rise to the hash functions

$$f_1(A) = |A|, \quad f_2(A) = \min A, \quad f_3(A) = \sum A.$$

Each of these hash functions uses one integer describe some characteristic of the array A (which is an array of integers).

We can of course combine several necessary conditions together. For example, the hash function

$$g(A) = (|A|, \min A, \sum A)$$

describes each array as a 3-tuple of integers.

Hash Functions

To check whether A is equal to B, we simply check if $f_1(A) = f_1(B)$ or not (or $f_2(A) = f_2(B)$, or $g(A) = g(B)$, or any hash function).

But obviously this is not reliable, it fails on many cases:
$f_1([1, 2, 3]) = 3 = f_1([5, 6, 7])$, yet $[1, 2, 3] \neq [5, 6, 7]$.

What if we use g as our hash function instead? That way, $g([1, 2, 3]) = (3, 1, 6)$ and $g([5, 6, 7]) = (3, 5, 18)$ which are not equal, so at least we managed to make the previous case work.

Still, with some effort, we can find a test case that fails:
$g([1, 3, 5]) = (3, 1, 9) = g([4, 1, 4])$, yet $[1, 3, 5] \neq [4, 1, 4]$.

Good Hash Functions

Since we are finding **necessary conditions** (which are in general *not* sufficient conditions), we cannot ever say with complete confidence that

$$A = B \iff f(A) = f(B)$$

unless we manage to find a necessary and sufficient condition.

In the above examples, we see that:

- If $f(A) \neq f(B)$, then we can say, *with certainty*, that $A \neq B$.
- If $f(A) = f(B)$, then A may or may not be equal to B.

The essence of a *good* hash function f is to make the situation of $f(A) = f(B)$ for some $A \neq B$ to be extremely unlikely.

Good Hash Functions

In competitive programming, all we care is whether problemsetters can easily construct an adverserial test case for our hash function to fail.

For example, we can set

$$f(A) = \sum_{i=1}^{|A|} A[i]^{i+100} \pmod{676767677}$$

and discover that problemsetters have no easy reliable way to generate two positive integer arrays $A \neq B$ such that $f(A) = f(B)$.

Therefore, we can say, somewhat confidently, that with such a hash function f,

$$A = B \iff f(A) = f(B)$$

What Does “Somewhat Confidently” Mean?

Our hash function

$$f(A) = \sum_{i=1}^{|A|} A[i]^{i+100} \pmod{676767677}$$

is rather unpredictable. So unpredictable, in fact, that I think it is fair to say that it spits a random integer between $[0, 676767677)$ out.

In other words, given two random arrays $A \neq B$, the probability that $f(A) = f(B)$ is $1/676767677$. This is negligibly small.

So, very loosely speaking, our “somewhat confidently” just now, actually means “with probability $1 - 1/676767677$”, but detailed analysis on hashing collisions are often a complicated matter.

Table of Contents

Our First Example

Problem

Given a string s, answer multiple queries, each of the following form:

- Given some $len < |s|$ and positions $0 \leq x, y < |s| - len$, determine whether the substrings $s[x\,..\,x + len)$ and $s[y\,..\,y + len)$ are equal.

Recall from our previous discussion: what are some **necessary conditions** for equality to hold, and that you can easily determine?

A sensible attempt would be that the sum of character values (taking 'a' as 1, 'b' as 2, and so on) in the two substrings has to be the same.

So, we let the hash value of a substring to be the sum of the character values within the substring. This can be easily computed with partial sums, leading to a $\mathcal{O}(|s|)/\mathcal{O}(1)$ solution.

Our First Example

The crucial issue of such a hash function is that **it has no way to differentiate the order of characters**.

In other words, "abac", "caba", "bcaa" (and all the other permutations) have the same hash value because characters have the same frequency in them.

This attempt is unsuccessful, but it gives us a clear message

A good function must be able to differentiate the order of characters.

Recall previously we had a good hash function $f(A) = \sum A[i]^{i+100}$ for arrays. It differentiates the order of characters by including the index i into the contribution of $A[i]$ as well.

This suggests we should probably do something similar here.

Our First Example

With some leap of faith, we can define the following hash function:

$$f(s) = \sum_{i=0}^{|s|-1} (s[i] - \text{'a'} + 1) \times B^i \pmod{p}$$

where we take $B = 133$ and $p = 998244353$ to be some arbitrary constants.

What does the expression mean? Our original attempt was the hash function $f_0(s) = \sum_{i=0}^{|s|-1} (s[i] - \text{'a'} + 1)$. By multiplying a factor of B^i to each term, it manages to take positions into account as well.

In other words, this is just the value modulo p when the string s is interpreted as a base-B integer, where the i-th digit is $(s[i] - \text{'a'} + 1)$.

Our Hash Function

For example,

$$f(\text{“abac”}) = 1 \times 133^0 + 2 \times 133^1 + 1 \times 133^2 + 3 \times 133^3 = 7075867$$

$$f(\text{“caba”}) = 3 \times 133^0 + 1 \times 133^1 + 2 \times 133^2 + 1 \times 133^3 = 2388151$$

so we are now successful.

One of the reasons why we insist on using $s[i] - \text{‘a’} + 1$ as the value of the digit as opposed to for example, $s[i] - \text{‘a’}$, is because we could then easily generate distinct strings with the same hash value:

$$f_1(\text{“a”}) = f_1(\text{“aa”}) = f_1(\text{“aaa”}) = \cdots = 0$$

Our Hash Function

To reiterate, as long as a function f

- is deterministic (i.e. if $s = t$, then $f(s) = f(t)$), and
- does not have easily-constructible adverserial test cases (i.e. if $f(s) = f(t)$, then we somewhat reliably have $s \neq t$),

then it is alright to use it as a hash function.

There are therefore countless ways to construct good hash functions, but the one introduced above is the most widely used to hash strings.

It is also known as the **polynomial hash**, since the formula above resembles a polynomial of degree $|s| - 1$: the coefficients are $s[i] - \text{'a'} + 1$, and the hash value is then the polynomial being evaluated at B.

Finishing off our First Example

Now that we have our hash function which can be conveniently (pre)computed, how do we turn it into a working solution?

Solution

First, build the partial sum $ps[i] = ps[i-1] + (s[i] - \texttt{`a'} + 1) \times B^i \pmod p$. This can be done in $\mathcal{O}(|s|)$ time. To compute the hash value of a substring $s[\ell..r]$, note that

$$f(s[\ell..r]) \equiv (ps[r] - ps[\ell-1]) \times B^{-\ell} \pmod p$$

By precomputing modular inverses beforehand as well, we can then compute the hash of any substring in $\mathcal{O}(1)$ time. For each query, we return "Yes" if $f(s[x..x+len-1]) = f(s[y..y+len-1])$, and "No" otherwise.

Some Strengths of Polynomial Hash

Why is our expression

$$f(s) = \sum_{i=0}^{|s|-1} (s[i] - \text{'a'} + 1) \times B^i \pmod{p}$$

so widely-used?

- It is intuitive and easy to write.
- If you append a character c to the back of s, you can easily calculate the new hash value; it is just the old hash + $(c - \text{'a'} + 1) \times B^{|s|}$.
- If you prepend a character c to the front of s, you can also easily calculate the new hash value; it is just $(c - \text{'a'} + 1) + B \times$ (the old hash).
- If you modify any character in s, you just add their difference multiplied by B to the power of the corresponding position.

An Easy Application

Here is a straightforward application on top of our previous example:

Problem

Given a string s, answer multiple queries, each of the following form:

- Given indices $0 \leq x, y < |s|$, determine the length of the common prefix of the substrings $s[x\,..\,x+len)$ and $s[y\,..\,y+len)$.

Solution

We already developed a way to check for substring equality in $\mathcal{O}(|s|)/\mathcal{O}(1)$. For each query, we binary search on answer and use hashing in our check function. Time complexity is $\mathcal{O}(|s|)/\mathcal{O}(\log|s|)$.

Note that this is much more easier to write than suffix array.

Another Easy Application

Problem

Given a string s, for each index $0 \leq i < |s|$, find the largest p such that $s[i-p\,..\,i+p]$ is a palindrome.

Our only tool so far is to do $\mathcal{O}(1)$ equality testing, how can we utilise this?

Solution

Let $t = s + rev(s)$. We then use hashing to test equality of substrings of t.

Note that $s[l..r]$ is a palindrome iff $t[\ell..r] = t\,[2|s| - r - 1..2|s| - \ell - 1]$, so again, we can simply do a binary search on answer just like above.

Time complexity is $\mathcal{O}(|s|)$/$\mathcal{O}(\log |s|)$.

Worked Example: Brperm

Problem: Brperm (RMI 2020 D1P2)

Define the *shift* of a k-bit integer $i = \overline{i_{k-1} \dots i_1 i_0}$, written in binary, to be $sh_k(i) := \overline{i_0 i_1 \dots i_{k-1}}$. For example, $sh_4(5) = sh_4(\overline{0101}) = \overline{1010}_2 = 10$.

Define a string a of length 2^m to be *nice*, if $a[i] = a[sh_m(i)]$ for all $0 \le i < 2^m$.

Given a string s, answer Q queries online, each of the following form:

- Given two integers i and k, determine if the substring $s[i..i + 2^k)$ is nice.

Constraints: $|s|, Q \le 5 \times 10^5$

Worked Example: Brperm

Same thing as before, we let our hash function f be the usual polynomial hash.

For each queried substring t of length 2^k, we want to check whether $f(t) = f(sh_k(t))$ or not.

We already know how to compute $f(t)$: just build the partial sum array on $s[i] \times B^i$.

What about $f(sh(t))$? After applying the sh function, the indices are no longer consecutive. From what we have seen so far, this cannot be hashed conveniently.

Nevertheless, with some leaps of faith and determination, we try to find some ways to convert it into a hash-friendlier form.

Worked Example: Brperm

Playing around with small examples are always helpful:

Index i	0	1	2	3
i in binary	00	01	10	11
$sh_2(i)$ in binary	00	10	01	11

Index i	0	1	2	3	4	5	6	7
i in binary	000	001	010	011	100	101	110	111
$sh_3(i)$ in binary	**000**	**100**	**010**	**110**	**001**	**101**	**011**	**111**

We observe that the sequence $sh_k(i)$ is simply two interleaved copies of $sh_{k-1}(i)$, prepended with 0 and 1 alternatingly.

Worked Example: Brperm

In fact, this simple observation allows us to precompute *all* hash values $f(sh(t))$, where $|t|$ is a power of 2. How might we do this?

Obviously, for $k = 0$ this is trivial.

Then, suppose we have computed the values $L := f\left(sh(s[i..i + 2^{k-1}))\right)$ and $R := f\left(sh(s[i + 2^{k-1}..i + 2^k))\right)$.

By definition, if $|t| = 2^m$, then $f(sh(t)) = \sum_{i=0}^{|t|-1}(t[i] - \text{'a'} + 1) \times B^{sh_m(i)}$.

Consider the expression $L + \sqrt{B}R$. In our $k = 2$ to $k = 3$ example, this is

$$(s[0]B^0 + s[2]B^1 + s[1]B^2 + s[3]B^3) + \sqrt{B}(s[4]B^0 + s[6]B^1 + s[5]B^2 + s[7]B^3)$$

Simplifying gives $\sum_{i=0}^{7} s[sh(i)](\sqrt{B})^i$, which is nothing but a polynomial hash with the base being $\sqrt{B}$.

Worked Example: Brperm

The only main issue left is that $\sqrt{B}$ might not necessarily exist. To resolve this, we simply start with using $B^{2^{\log|s|}}$ for $k = 0$. This way, we know that we will be using $B^{2^{\log|s|-k}}$ as our base for the hash function at the k-th iteration.

Such a pattern naturally extends to general values of k, giving a $\mathcal{O}(|s| \log |s|)/\mathcal{O}(1)$ solution.

Takeaways

- Polynomial hash works reliably for *any* base $B \neq 0, 1$. It is precisely this flexibility that allowed us to solve this problem.
- Hashing is powerful for maintaining information in consecutive intervals. The most natural thing to do when the indices are out-of-position, is to seek for patterns or invariants that move them back predictably.

Other Common Usages of String Hashing

Another common use of hashing is *wildcard matching*, which can be seen as an extension of testing for equality.

Suppose there are two strings s and t. On top of the usual alphabet, s can also contain some wildcard characters. We want to find all substrings of s that matches t.

For example, "`a*a*`" matches "`abac`" and "`aaab`", but not "`baaa`".

The main idea is that if s has no wildcards, then we can use the condition

$$\sum_i |s[i] - t[i]| = 0, \text{ which can be rewritten as } \sum_i (s[i] - t[i])^2 = 0$$

Other Common Usages of String Hashing

What if s now contains wildcards? How might we extend our idea above to make it work?

Observe that we can further rewrite our condition above to be

$$\sum_i C_i(s[i] - t[i])^2 = 0$$

for some coefficients C_i which are all nonzero.

If position i is a wildcard, then we simply set C_i to be 0 there; and if position i is a normal character, we set C_i to an arbitrary nonzero integer.

It is easy to see that this works somewhat confidently.

However, fully utilising this trick requires the use of Number Theoretic Transform (NTT) which is off-syllabus.

Table of Contents

Hashing is More Than Just Strings

Most hashing tutorials or blogs online only talks about hashing strings, which is what we have seen in the previous section.

However, hashing is *much much more* than strings! Floating-points, arrays, sets, multisets, trees, graphs... you name it.

The objective of this lecture is to expose you to a number of different ways of hashing different structures.

Starting from the Basics, Again

We revisit the guiding question at the very start of this lecture, but this time, we are interested in sets (unordered, without duplicates) rather than arrays.

Problem

Given two sets of positive integers A and B, determine whether they are equal.

Bear in mind that the essence of hashing is to find necessary conditions that are easily checkable. With this in mind, here are a few possible hash functions:

- Sum of elements: $f_1(A) = \sum_{a \in A} a$
- Sum of squares of elements: $f_2(A) = \sum_{a \in A} a^2$
- Xor-sum of elements: $f_3(A) = \bigoplus_{a \in A} a$
- ...and countless more.

Starting from the Basics, Again

Recall that xor-sums are simply sums in binary without carrying over. Consequently, in most cases, sums and xor-sums as hash functions behave alike.

Most literature prefer using xor-sum over the normal sum. It is because the linear independence between bits makes the collision frequency easier to analyse.

We will stick to the general convention of using xor-sums.

Starting from the Basics, Again

Xor-sums are convenient and very easy to compute, but it fails easily because, for example,

$$f(\{1,2,3\}) = f(\{4,5,6,7\}) = f(\{8,9,10,11\}) = 0$$

But what if we introduce some ***randomness***?

Why such a thought?

Suppose the elements in A and B are generated uniformly randomly in $[1, X]$.

Then, the collision probability when we use xor-sum as our hash function is around $1/X$. In other words, it works $1 - 1/X$ of the time. This aligns with our initial discussion towards the start of the lecture.

Starting from the Basics, Again

Why introduce randomness?

If we could find some way to "force" the elements in A and B to be uniformly generated, then we would be somewhat confident (with probability $1 - 1/X$) that we are safe.

The most intuitive way to "force" the elements to be uniformly random when the actual input is not, is to artificially add a layer of filter to make the elements random.

The idea goes like this: for each integer $i \in [0, X]$, we associate it with a random integer $r(i)$. Afterwards, simply replace every $i \in A, B$ by $r(i)$. Then, you can do the xor-sum hash somewhat confidently.

Starting from the Basics, Again

Note that for this method to work, $r(i)$ really needs to be unpredictable, otherwise we would still be able to construct failing cases systematically.

Also, the larger the range of values of $r(i)$ can take, the lower the collision frequency.

Exercise

Convince yourself (or prove) that the collision frequency decreases exponentially with respect to the number of bits that r takes.

We can therefore conclude, somewhat confidently, that

$$A = B \iff \bigoplus_{a \in A} f(r(a)) = \bigoplus_{b \in B} f(r(b))$$

Xor-hashing in Action

Problem: Three Occurrences (Codeforces 1418G)

Given a positive integer array $A[1], A[2], \ldots, A[N]$. Count the number of nonempty subarrays $A[L..R]$ where every number that appears in it appears exactly three times in the subarray.

Constraints: $N \leq 5 \cdot 10^5$, $1 \leq A[i] \leq N$

Xor-hashing in Action

Let's solve an easier version of the problem first:

Weak Version

Count the number of subarrays so that every number appears an even number of times.

It is obvious that every such subarray has xor-sum 0, so we might want to compute the partial xor-sums of each prefix, and the answer is then just $\sum_x \binom{occ(x)}{2}$.

However, as demonstrated above, this necessary condition is not sufficient. What we can do instead, is for every $i \in [1, N]$, assign some uniformly random value $r(i)$ chosen in $[0, 2^d)$. In other words, we pick a d-bit integer.

Xor-hashing in Action

Solution to Weak Version

We proceed as above: compute the partial sums by $ps[i] = ps[i-1] \oplus r(a[i])$.
Let $occ(x)$ denote the number of indices $i \in [0, n]$ where $ps[i] = x$.
Then, our answer is, somewhat confidently, $\sum_x \binom{occ(x)}{2}$.

Now something closer to the original problem: what if we wanted to count the number of subarrays so that the frequency of each number *is a multiple of* 3?

It is not difficult to observe that the solution we have described above still works, just that we change the $\oplus$ to calculating xors in ternary bits instead; the rest is the same (though with a slightly higher collision probability).

Xor-hashing in Action

Solution (to original problem)

We iterate in increasing order of R and find how many L-s satisfy the condition. We can use two-pointers to easily find the largest L (if any) such that the maximum occurrence of a number in $[L..R]$ is > 3.

So instead of directly summing with a closed formula as above (when we required frequencies are multiples of 3), we binary search to find how many valid indices L also have partial sum value $= ps[R]$, and add it to the answer.

This is $\mathcal{O}(nd)$, where normally $d = 32$ or 64 (why?).

The main takeaway is that k-ary xor-sums work really well to handle conditions related to frequencies being multiples of k, so we tweak our initial idea (by introducing randomness) to make it work for us.

Finding the Sets to be Hashed

Problem: Penacony (Codeforces 1996G)

There is a circular ring of size N: there are N nodes, and for every i, there is an undirected edge between node i and node $(i+1) \pmod N$.

You are also given M pairs of nodes (a_i, b_i).

Find the maximum number of roads that you can remove (among the N roads), so that for every pair of specified nodes, a_i and b_i are still connected afterwards.

Constraints: $N, M \leq 2 \cdot 10^5$

There is a $O(N \log N)$ solution by maintaining stuff with segment trees (exercise: find it!), but we aren't interested about it today.

Finding the Sets to be Hashed

Denote the edge between node i and node $i+1$ as edge i.

Consider maintaining for each edge i, the list of paths $a-b$ that uses the edge i. Denote this set as S_i. We can reformulate the statement in terms of these sets S_i as follows:

Reworded Statement

There are N sets $S_1, S_2, \ldots, S_N$, all initially empty. For each $1 \leq i \leq M$, WLOG assume $a_i < b_i$, and do exactly one of the following:

- insert i into each of the sets $S_{a_i}, S_{a_i+1}, \ldots, S_{b_i-1}$, or
- insert i into each of the sets $S_{b_i}, \ldots, S_N, S_1, \ldots, S_{a_i-1}$.

Find the maximum possible number of empty sets among $S_1, S_2, \ldots, S_N$ after all M operations.

How to Hash the Sets?

Why would such a reformulation in terms of sets be helpful?

The most naive solution is to directly exhaust which one of the two sides to pick for each path $a - b$. This is clearly unsatisfactory.

Observation

We WLOG assume to always make the first choice of inserting into $S_{a_i}, S_{a_i+1}, \ldots, S_{b_i-1}$.

If we wanted to change our mind and make the second choice instead, what we have to do is just to toggle all N sets by $\{i\}$ (i.e. if the set contains i, then remove it from the set; otherwise insert it to the set).

In light of this, what are we now trying to maximise? Do you see why this formulation is powerful yet?

How to Hash the Sets?

We note that the observation reduces the problem into the following:

Transformed Problem

There are N sets $S_1, S_2, \ldots, S_N$, all initially empty. For each $1 \leq i \leq M$, we insert i into each of the sets $S_{a_i}, S_{a_i+1}, \ldots, S_{b_i-1}$.

Output the size of the largest subset of $S_1, S_2, \ldots, S_N$ for which all the chosen sets have the same content.

Cool, now at least the most naive solution is in polynomial time. We just need to find a way to optimise it.

Hopefully by now the optimisation becomes obvious: we **maintain the hashes of the sets** to enable quick equality testing.

Finishing the Problem

However, we cannot naively update the hashes of the set one-by-one; that would be too slow.

Instead, we can maintain the **difference array** of the N hash values. That way, each update is done in $\mathcal{O}(1)$ time.

Solution

We maintain the hashes in an array $H[1..N]$, initially all 0s.

For each i, update by doing $H[a_i] := H[a_i] \oplus r(i)$ and $H[b_i] := H[b_i] \oplus r(i)$, where each $r(i)$ is generated uniformly randomly.

The answer is simply the largest number of equal elements in $H[1], H[1] \oplus H[2], \ldots, H[1] \oplus H[2] \oplus \cdots \oplus H[N]$ which can be calculated in $\mathcal{O}(n \log n)$.

Finishing the Problem

The constraints were initially written in terms of paths. By looking them at the constraints from the perspective of individual edges instead, and describing the problem in terms of sets, we then optimise it by xor-hashing.

Takeaways

- Xor-hashing is used to quickly compare whether the contents of sets are the same or not. Most of the time, you can "work backwards" to find out what are the appropriate things to store in the sets, if you can "feel" like the problem can be solved with xor-hashing.
- If directly computing each of the hashes is too time-inefficient, consider using the some basic optimisations techniques as you would on array problems: difference arrays, segment trees...

Hashing Colours

Problem: Colours (Jiangxi Province Selection 2017)

You are given an array $A[1], A[2], \ldots, A[N]$. Find the number of subsets S, such that every $s \in S$ occurs at least once in the array A, and the indices in A whose elements are in S form a consecutive (possibly empty) interval.

Constraints: $N \leq 3 \cdot 10^5$, $1 \leq A[i] \leq N$

Again, there is a $\mathcal{O}(N \log N)$ solution with data structures, but we are not interested in it for this lecture.

Let's say by some miracle (e.g. praying to the gods) you can sense that there is a hashing solution. What do you do now?

Hashing Colours

We “work (and think) backwards”:

- This is equivalent to counting the number of intervals $[L, R]$ such that the numbers appearing inside the interval are disjoint with the numbers appearing outside the interval.
- This is probably maintained by counting the pairs of (L, R) for which $H[L-1] = H[R]$, where $H[1..N]$ is some array representing the hash values of something that we should work out.

What are some properties that you want the array H to have? How can you make that happen?

Hashing Colours

We want to assign a hash value v to each position in the array, and that H is the partial xor-sum of the hash values, so that the condition of $H[L-1] = H[R]$ matches what we want to count.

For each value $1 \le i \le N$, if we denote its occurrences to be at positions $p_1 < p_2 < \cdots < p_k$, then it would be great if we had:

- $v[p_1] \oplus v[p_2] \oplus \cdots \oplus v[p_k] = 0$, and
- No proper subsets of $\{p_1, p_2, \ldots, p_k\}$ satisfy the condition above.

This is too unrealistic!

Supposer the hash values v are generated between $[0, 2^d)$.

It is a trivial fact from linear algebra that if $k > d$ then the two conditions above cannot be satisfied.

Hashing Colours

Do we really need that strong of a condition? **No!** It suffices to have:

- $v[p_1] \oplus v[p_2] \oplus \cdots \oplus v[p_k] = 0$, and
- No proper **subarrays** (i.e. $v[p_\ell] \oplus v[p_{\ell+1}] \oplus \cdots \oplus v[p_r]$) satisfy the condition above.

This is much more realistic!

Now this is much more realistic: the collision probability is now $\mathcal{O}(k^2/2^d)$ instead of $\mathcal{O}(2^k/2^d)$ as before: when $d = 64$, this is already negligible enough.

To do so, we simply set each of $v[p_1], v[p_2], \ldots, v[p_{k-1}]$ to be an integer generated uniformly randomly in $[0, 2^d)$, and then force $v[p_k]$ to be $v[p_1] \oplus v[p_2] \oplus \cdots \oplus v[p_{k-1}]$.

Hashing Colours

...and the solution falls apart naturally by our construction.

Solution

For every $1 \leq i \leq N$, extract its occurrences and generate the hash values v at their positions as described above.
The answer is just the number of pairs of (L, R) with $v[L] \oplus v[L+1] \oplus \cdots \oplus v[R] = 0$, which we already noted that it could be sped up by partial sums. Time complexity is $\mathcal{O}(N \log N)$.

Takeaways

Remember that "exponentials beat polynomials": as long as you control the number of collisions to be $\mathcal{O}(\text{poly}(n))/2^d$, it works well for n large enough.

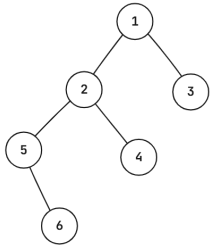
Table of Contents

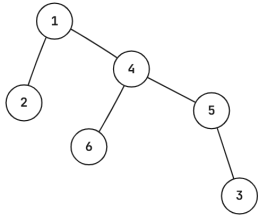
Hashing Trees

The problem that we are interested in this section is the following:

Problem

Given two **rooted** trees A and B, each of size N and has node 1 as the root, determine whether they are isomorphic (i.e. there exists a permutation p of $\{1, 2, \ldots, N\}$ such that there is an edge $u - v$ in $A \iff$ there is an edge $p[u] - p[v]$ in B).





Hashing Trees

Let f be the hash function, and u be any node, with children $v_1, v_2, \ldots, v_k$.

We expect a good hash function has $f(u)$ to depend on the hash values of all its children $f(v_1), f(v_2), \ldots, f(v_k)$. We also require the hash function to be symmetric over the k arguments $f(v_1), f(v_2), \ldots, f(v_k)$.

Then, one can show that

$$f(u) = 1 + \sum_{i=1}^{k} g(f(v_i)) \pmod{p}$$

where g is any sufficiently unpredictable function, has a collision probability of $\mathcal{O}(n^2/p)$. Actually p need not be a prime so it is common to take $p = 2^{64}$ and let it overflow naturally.

Hashing Trees

It is quite difficult to rigorously define what is “sufficiently unpredictable” and what is not. This is not of the interest of this lecture.

It is not recommended to take g as a polynomial as it is periodic (mod p) and something unpleasant might occur. Consult this blog for details.

Putting this out of our concern, we can just directly DFS to compute the hashes of each node. Then, we conclude that A and B are isomorphic $\iff$ $f(1)$ of $A = f(1)$ of B.

There is a very big advantage of taking $f(u)$ to be a linear combination of $g(f(v_i))$-s (i.e. no cross terms at all): if we decide that we want to **reroot** the tree, we can easily calculate the hash at new roots by using rerooting tree DP, without having to compute much new stuff every time.

Isomorphism for Slightly Expanded Graphs

Problem: Iso (NOI 2022 D2P1)

You are given a tree G of size N rooted at node 1, and another tree H of size $N + K$ rooted at node 1. Then for K separate times, you pick a non-root node in H and remove it.

Determine whether it is possible for G and H to be isomorphic after these K operations.

Constraints: $N \leq 5 \cdot 10^5$, $0 \leq K \leq 5$.

Note that when $K = 0$, this is just simply checking for tree isomorphism which we have already mentioned. How might we extend our previous solution?

Isomorphism for Slightly Expanded Graphs

We first compute the subtree hashes for each node in G and H, chances are that we will probably need them anyways.

Let the hashes of the root's children in G be $a_1, a_2, \ldots, a_g$, and that in H be $b_1, b_2, \ldots, b_h$.

Observation, informal statement

Intuitively, if the multisets $\{a_1, a_2, \ldots, a_g\}$ and $\{b_1, b_2, \ldots, b_h\}$ differ by more than K elements, then it means there are more than K distinct subtrees, so G and H cannot be isomorphic.

We formalise it as follows: whenever there are still common elements in the two multisets, this means they have some identical subtree (which we can safely delete that subtree as they are isomorphic).

Isomorphism for Slightly Expanded Graphs

Solution

After such deletions, if $g > h$ then clearly this is impossible.

Otherwise, if $g \leq h$, then we exhaust, for each subtree in G, which subtree in H it gets mapped to.
There are at most $K!$ possible ways to match, and for each of them we continue to exhaust down in the subtree and DFS to solve recursively.

This might look very slow, but if you do the math, this is actually at most $\mathcal{O}(NK!K^2)$

We can prune quite easily, such as making sure the subtrees that we want to match actually have reasonable subtree sizes that match, and is good enough.

A Short Reflection

Rather than summarising takeaways, it would be more beneficial to appreciate that trees (a recursively defined structure) can also be hashed.

Takeaway: How did we define the hash of a tree?

1. Identify when we want to regard two trees as equal (i.e. two trees are same $\iff$ multiset of subtree hashes of root nodes are the same).
2. Reflect this equivalence condition in our hash function (i.e. a sufficient condition of satisfying the above is when the hash function is "symmetric" in the hashes of the subtrees).
3. Now any such function is already a hash function.

A Short Reflection

Takeaway: How did we define the hash of a tree?

Among the countless hash functions which possess the above property, pick one that is

- "unpredictable and random enough", to minimise unwanted collisions, and
- its computation is manageable and convenient, so that it can be computed with the same complexity as you DFS along the tree without incurring extra time complexity.

Even if you encounter unfamiliar data types, it should not be difficult to come up with a good way to hash stuff by following such a paradigm.

Table of Contents

Hashing Big Integers

This is more intuitive than the previous sections: you just do all calculations modulo some prime $p \sim 10^9$ so that we can do the normal operations by `long long`-s.

Sometimes using one prime p alone is not enough to guarantee correctness; you can just take tuples of distinct primes, and calculate each component separately.

Chinese Remainder Theorem guarantees that such an operation is always well-defined.

Sounds intuitive and easy! Let's look at some examples to see why this simple idea is useful.

What If Primes Alone Are Not Enough?

Problem: Fibonacci Fusion (Unicup Season 3 Stage 7 Problem F)

Given N positive integers $A[1], A[2], \ldots, A[N]$, count the number of pairs $1 \leq i < j \leq N$ such that $A[i] + A[j]$ is a Fibonacci number.

Constraints: $N \leq 2 \cdot 10^5, \log_{10} A[i] \leq 2 \cdot 10^6, \sum \log_{10} A[i] \leq 5 \times 10^6$

Big integers are annoying...

We temporarily get rid of it first, and handle it later:

- Imagine having a datatype that can do additions, subtractions, multiplications, and equality testing of however large numbers, all in $\mathcal{O}(1)$ time. How would you do this problem?

The Version without Big Integers

We first sort the array A in increasing order.

Observation

Note that for every integer $x > 1$, there are at most two Fibonacci numbers within $[x, 2x]$.

Consequently, for every j there are at most two values of i that work.

Since we assumed we have the ability to do arithmetic with big integers, we can precompute all Fibonacci numbers up to $2 \max A$ (which there are around $\log_\varphi(2 \max A)$ such numbers).

For each j, we then binary search to find those (up to) two Fibonacci numbers in $(A_j, 2A_j]$, and binary search to check whether A_j subtracted from each of those Fibonacci numbers exist in $A[1..j)$ or not.

What are our limitations?

Now that we finish fantasising about the ability to cope with big integers, it's time to come back to reality and find out what doesn't work here:

1. We cannot compute all Fibonacci numbers up to $2 \max A$.
2. We cannot subtract A_j (a big integer) from a large Fibonacci number.

As mentioned in the overview, we try to compute everything in modulo p for some arbitrary prime(s) p. Now,

- We *can* compute all Fibonacci numbers modulo p
- We *can* subtract A_j from a large Fibonacci number modulo p

What are our limitations?

But we have a new problem: we **cannot** do binary search modulo p.

This is because when we are given two numbers modulo p, we cannot know which one is larger *before* taking mod.

In other words, we no longer have the ability to (efficiently) compare numbers.

Now our solution breaks apart; are there any ways to salvage it?

Current progress

1. Direct bigint computation: comparison is OK, but cannot compute large numbers
2. Hash modulo p: cannot compare, but compute large numbers is OK

What If Primes Alone Are Not Enough?

Easy, just combine the two methods together!

More precisely, we hash a big integer x into a pair of integer mod p and a floating-point by $f(x) = \{x \pmod p, \log x\}$. Then we can compare and compute large numbers at the same time.

We know that $\log x$ is going to be a bit imprecise, but it doesn't matter for us, as we are only using it to compare numbers approximately. If we want to compare precisely (i.e. when the two numbers are really close in absolute terms), we switch to compare by using the mod p component instead.

What If Primes Alone Are Not Enough?

All this is a bit sketchy, so we describe the solution concisely:

Solution

First, we compute all the Fibonacci numbers: the mod p component is just trivial DP. Since F_n can be approximated by $\varphi^n/\sqrt{5}$, the log component can be approximated by $n \log \varphi - \log \sqrt{5}$ for large n.

For each j, we compute $(A[j] \pmod p, \log A[j])$ directly. Then, we find the Fibonacci numbers with log values between $\log A[j]$ and $\log A[j] + \log 2$. We can then compute the value of $A[i] \pmod p$ that we expect and check whether it exists or not.

Time complexity is $\mathcal{O}(N \log N + \sum \log_{10} A[i])$ due to sorting.

What If Primes Alone Are Not Enough?

Takeaways

- Towards the start of the lecture, it is mentioned that a hash function is just a (combination of several) necessary conditions.
  Here, we needed a combination of numbers that represented the exact value (for the actual checking), and an approximate value (for us to slim down the range of checking to sufficiently small).
- In this problem, the hard part is adapting our original “imaginary” solution for bigints to a solution that is realistic. Hashing often reduces a lot of inconvenience and hassle in doing so.

Final Boss of the Lecture

Problem: Schoolgirls (Unicup Season 3 Stage 1 Problem B)

There is an initial set of N points forming a regular N-gon in anticlockwise order, denoted as $K_1, K_2, \ldots, K_N$.

An additional M points are generated in order as follows: each new point K_{N+i} is defined by three points $1 \leq a_i, b_i, c_i < N + i$ using the vector relation $K_{N+i} = K_{a_i} + K_{c_i} - K_{b_i}$ (coordinates calculated component-wise). Answer Q queries of the following form:

- Given R indices $P_1, P_2, \ldots, P_R$, determine if the points $K_{P_1}, K_{P_2}, \ldots, K_{P_R}$ form a regular R-gon (not necessarily in the given order, possibly degenerate).

Constraints: $N \leq 10^4$, $M \leq 3 \times 10^4$, $\sum R \leq 3 \times 10^4$

Assuming Infinite-Precision Arithmetic...

Let's strip away the annoying components first and focus on the geometry: imagine we could do floating-point arithmetic with 0 error.

How do we determine if R points form a regular R-gon?

Suppose the points form a regular R-gon.
Its centroid has coordinates $C := \frac{1}{R}(K_{P_1} + \ldots + K_{P_R})$.
If we translate everything so that the centroid is at the origin 0, then the set of points stays the same after applying a rotation by $\frac{2\pi}{R}$.

However, we are bound to fail if we work in $\mathbb{R}^2$ using floating-points. For any $N \neq 4$, we can construct a sequence of points that gets infinitesimally close to forming a regular polygon without ever actually being one.

Points as Complex Numbers

This means we *must* find an exact representation as all estimations won't work.

Just now, we translated everything to put the centroid at 0.

Helpful fact

$\mathbb{R}^2 \cong \mathbb{C}$.

Let's view the complex plane algebraically and place the original regular N-gon's center at the origin. Its vertices are just the powers of $\zeta_N = e^{2\pi i/N}$. In other words, the initial points are $\zeta_N^1, \zeta_N^2, \ldots, \zeta_N^N = 1$.

Points to Polynomials

The Algebraic Structure

New points are constructed from additions and subtractions only. So, every point P_i is an integer linear combination of the original points.

Therefore, **every constructible point is some polynomial $P(\zeta_N)$ evaluated at ζ_N** with integer coefficients.

We *could* just store each point as an array of its polynomial's coefficients.

The Issue

Doing this requires keeping track of up to $\mathcal{O}(N)$ integer coefficients per point. So, to construct a new point would take $\mathcal{O}(N)$ time, which is way too slow.

The Modular Hashing Trick

We want to map our complex geometry into normal modulo arithmetic, somewhere which we are comfortable to hash to.

Can we find some prime p and an integer mod p that has the same role of ζ_N in $\mathbb{C}$?

Pick some prime $p \equiv 1 \pmod{\text{lcm}(2, N)}$. (By Dirichlet's Theorem, there are infinitely many such primes.)

Let g be a primitive root mod p. Then, we observe that $g^{(p-1)/N}$ suits us well.

So, the initial points are hashed to $g^{(p-1)/N}, g^{2(p-1)/N}, \ldots, g^{N(p-1)/N} = 1$. Now, vector addition is just integer addition mod p.

Great, now that we know how to hash the points under some mod p, how shall we answer the queries?

Answering Queries

Solution

First, if all hashed points are equal, the polygon is degenerate and hence regular.

Fact: A regular R-gon requires an R-th root of unity to exist. By some algebra, this rotation only exists in our system if R divides $\text{lcm}(2, N)$. If it doesn't, then it cannot possibly be regular.

1. Compute the centroid $C = \frac{1}{R} \sum_{i=1}^{R} hash(P_i) \pmod p$.
2. Shift all points: $V_i = hash(P_i) - C \pmod p$.
3. Check for rotational equivalence by determining whether $\{V_1, \ldots, V_R\} = \{g^{(p-1)/R} V_1, \ldots, g^{(p-1)/R} V_R\}$ or not.

The Final Hashing Trick

How do we check multiset equality efficiently?

We could sort both shifted multisets in $\mathcal{O}(R \log R)$. But actually, there is a slicker method of **multiset hash**!

Pick a random integer $X \in \mathbb{Z}_p$. We hash the multiset by evaluating its polynomial roots:

$$H(V) = \prod_{i=1}^{R} (X - V_i) \pmod{p}$$

We just check if $\prod_{i=1}^{R}(X - V_i) \equiv \prod_{i=1}^{R}(X - \omega_R V_i) \pmod{p}$.

Final Boss of the Lecture

Takeaways

1. First think about how the solution might be solved without any practical concerns. Think about what concerns *can* and *cannot* be bypassed.
2. Look at *how* the points are generated. Analyse how the objects to be hashed can be written as combinations of base element(s). In this problem, the hash function is so nice (under addition and multiplication) that it is a **homomorphism**.
3. A way to hash multisets is to do the product polynomial hash $f(X) := \prod(X - a_i)$. It might be a surprising discovery at first, and the correctness and proof of it comes from the Schwartz-Zippel Lemma.

Conclusion

Hashing is a probabilistic method that lets us check whether two things are equal or not. The crux of hashing is determining a suitable hash function that there is difficult to create adversial collision tests.

- Problems related to hashing mostly revolve around the few configurations. The techniques in this lecture already covers a good proportion of the common techniques involved. Add these tricks into your toolbox kit.
- It is often a complicated and technical matter to rigorously analyse coliision frequencies. You may find a handful of thoughtful discussions found in Codeforces' catalog.