NOI.PH Training: Hashing

Hashing

Kevin Charles Atienza

Contents

1 Introduction **2**

2 Hash tables **3**
2.1 Resizing scheme . . . 5
2.2 Dynamic hash sizes . . . 6
2.3 Performance . . . 6

3 List and string hashing **9**
3.1 A standard problem: comparing substrings . . . 9
3.2 The hashing solution . . . 10
3.3 Another standard problem: counting substrings of a fixed length . . . 12
3.4 The probability of failure . . . 13

4 Set hashing **15**
4.1 Maintaining sorted hashes . . . 15
4.2 Zobrist hashing . . . 15

5 Tree hashing **17**
5.1 Ordered tree hashing . . . 17
5.2 Unordered tree hashing . . . 17
5.3 Unrooted tree hashing . . . 18

6 More on Hashing **19**
6.1 Hashing in cryptography . . . 19
6.2 Further reading . . . 19

7 Problems **20**
7.1 Non-coding problems . . . 20
7.2 Coding problems . . . 21

1 Introduction

In this section, we will learn all about **hashing**.

Hashing can be easily summarized: It's just the technique of mapping objects from a large set to a small set. That's it!

Of course, this needs some elaboration. Why is that even useful? We'll go into that shortly.

A **hash function** is a function[1] from some large set to some other, usually smaller, set. Note that we didn't impose any other restrictions, so a hash function is more-or-less just a function. We just give them the special name "hash function" since we'll be using them for hashing, but any function can technically be considered a hash function. Although, as we shall see later on, some functions are better hash functions than others.

Some more jargon. Let $h : S \to T$ be a hash function. For $x \in S$, we say that $h(x)$ is the **hash of** x.[2] Any value in T is called a **hash**, or a **hash value**.

The key property of hash functions that we will use is this:

Property 1.1. The hashes of two equal objects are equal: for any hash function h, if $x = y$, then $h(x) = h(y)$.

This is of course a trivial assertion, but it emphasizes one of the important uses of hashing. Hashing is generally used as a substitute to *equality checking*. The idea is that if we want to check whether two things x and y are equal, we first check if their hashes, $h(x)$ and $h(y)$, are equal. If not, then x and y are definitely not equal, and we don't need to compare them. This is helpful if x and y are "big" but $h(x)$ and $h(y)$ are "small"; comparing small things is easier!

Obviously, the other direction is not true; two objects with the same hash are not necessarily equal. When two unequal objects have the same hash, we call it a **collision**. If we assume that $|S| > |T|$, then collisions are inevitable because of the pigeonhole principle.

We start with one of the most basic usages of hashing: hash tables.

[1]Here, we use "function" in the mathematical sense.

[2]Technically, we should say that $h(x)$ is the "hash of x with respect to h", but that's too wordy and pedantic, and pointless if it's clear from the context what hash function we're using.

2 Hash tables

Let's say we want to implement a familiar abstract data type called **map**. We define a map to be a data type consisting of a collection of key-value pairs such that each key appears at most once, and we define the operations to be:

- **insert**(k, v). Insert a key-value pair with key k and value v. If k doesn't already exist as a key, it deletes the old pair.
- **contains**(k). Check whether k exists as a key.
- **get**(k). Return the value v associated with key k if k exists as a key.
- **delete**(k). Remove the entry with key k if it exists.

There's a related abstract data type called **set** which is similar to a map but only has keys instead of values. A set can trivially be implemented with a map, so we may simply focus on maps.

You've probably already used a map in the form of C++'s `map`, and you've probably already learned that they can implemented with *binary search trees*.[3] Here, we will learn that they can also be implemented in a different way using hashing. The resulting structure will be called **hash tables**.

In general, we can build a hash table for any type of key as long as they're *hashable to integers*, i.e., there is a hash function that takes a key to some integer. But for now, let's assume that the keys will be integers for simplicity.

To start with, let's consider the case where the keys are integers and are bounded above by some number, say m. In this case, we can easily implement a map with an array of size m. The code would look very simple, like this:

```cpp
const int m = 100000;
class Map {  // assumes that keys are integers and values are strings.
    string table[m];
    bool present[m];
public:
    Map() {}
    void insert(int key, string value) {
        table[key] = value;
        present[key] = true;
    }
    bool contains(int key) {
        return present[key];
    }
    string get(int key) {
        return table[key];
    }
    void erase(int key) {
        table[key] = string();
        present[key] = false;
    }
};
```

This clearly works.[4] However, there are a few downsides:

[3]In fact, C++'s `map` is (usually) implemented internally using binary search trees.

[4]You can also use C++ templates to allow for value types other than `string`.

- It doesn't work for keys not in the range $[0, m-1]$.
- Even if the keys are in that range, this method wastes a lot of space if only a few keys are inserted. We probably want a structure whose size scales proportionally with the number of keys.
- Obviously, if the keys are not integers anymore, then this doesn't work.

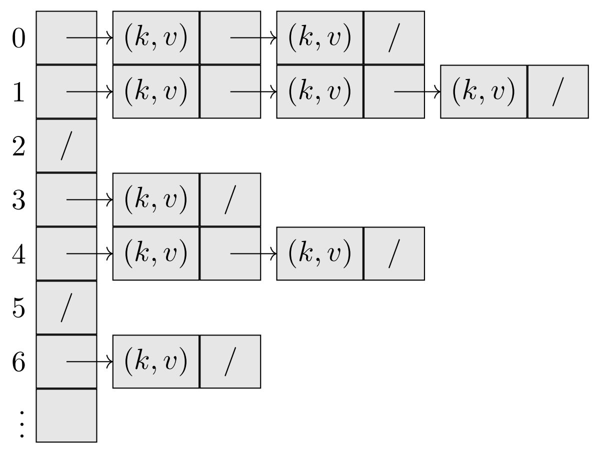
We can "fix" this by using hashing. The idea is to use a hash as a substitute for our key. Specifically, if h is a hash function that takes a key to the set $\{0, 1, \ldots, m-1\}$, then the above code can be modified to:

```
const int m = 100000;
int h(int x) {
    ... // implementation of h. It must return an integer in the range [0, m).
}
class Map {
    string table[m];
    bool present[m];
public:
    Map() {}
    void insert(int key, string value) {
        int hash = h(key);
        table[hash] = value;
        present[hash] = true;
    }
    bool contains(int key) {
        return present[h(key)];
    }
    string get(int key) {
        return table[h(key)];
    }
    void erase(int key) {
        int hash = h(key);
        table[hash] = string();
        present[hash] = false;
    }
};
```

We can then view the earlier implementation as a special case in which we use the identity hash function, $h(x) = x$.

This version looks a little better since we're not anymore restricted to keys that are bounded integers (or even integers at all), although now we will require a hash function to exist. However, one major downside is that there will be *collisions*. As mentioned before, this is unavoidable if the set of keys is larger than the set of hashes. Thus, this solution will sometimes be incorrect: it will consider objects with the same hash the same object.

There's an obvious fix. Instead of storing only a single value for every hash, we store a *list* of key-value pairs for every hash. And whenever two keys have the same hash, we revert to linear search.



Each list is called a *chain*, and the hash function is now used to map each key to the correct chain. Each chain can be implemented as a normal list structure (e.g., a `vector`).

With this change, we finally get a correct implementation a map. This technique is called **chaining**.[5]

There's one more issue we need to address. Our structure assumes that the number of chains, m, is fixed. Which m should we use? It should be clear that there's no single m that satisfies our requirements, because:

- If m is too small, then the pigeonhole principle guarantees that there will be a chain that's at least $\lceil n/m \rceil$ elements long, where n is the number of key-value pairs present. Thus, the operations will be slow, especially those on that chain.
- If m is too large, then we can't make too many hash tables since they will take up too much space.

What we really need is to make its size *dynamic*, i.e., it should depend on the number of keys present. So we will need a resizing scheme—a rule that determines when and how to resize the structure.

2.1 Resizing scheme

We would like the number of chains m to depend on the number of key-value pairs, say n. Ideally, we would like them to be proportional ($m = \Theta(n)$) so a hash table takes up $\Theta(n)$ space.

Actually, we can just do it the way `vector` does it: If the fraction $\frac{n}{m}$ goes above some threshold U or below some threshold L, then we resize by some constant factor.

The following code shows a way to do it. It tries to maintain that the *load factor* $\frac{n}{m}$ lies between L and U.[6]

[5]There are other techniques to handle collisions, e.g., *open addressing*, but those are more confusing to implement. You may choose to learn it from the Wikipedia page: https://en.wikipedia.org/wiki/Hash_table#Open_addressing

[6]The constants chosen are mostly arbitrary; there are definitely better choices. But I didn't bother optimizing them too much since you don't usually need to code hash tables; they are already implemented in the STL after all.

```
    int m;// current number of chains
    int n;// current number of key-value pairs
    double L = 1/3., M = 1/2., U = 2/3.;

    void resize(int mNew) {
        ... // do the resizing here, from m to mNew.
        m = mNew;
    }

    void try_rehash() {
        double load = double(n)/m;
        if (!(L <= load && load <= U)) {
            int mNew = max(1, int(n/M)); // we want n/m ~ M (and also m != 0).
            if (m != mNew) resize(mNew);
        }
    }
```

Similarly to vectors, one can:

Exercise 2.1. Show that the amortized running time of inserts and deletes becomes $\mathcal{O}(1)$, assuming that $0 < L < M < U$ and that `resize` takes $\mathcal{O}(m + m_{\text{new}})$ time.

However, a dynamic m introduces a new issue: our hash function assumes that m is fixed!

2.2 Dynamic hash sizes

We need a "hash function" that is flexible enough to work even if m changes.

The way this is usually done is simply by *hashing to integers than reducing modulo m*. In other words, instead of defining a hash function $h : S \to \{0, 1, \dots, m-1\}$, one just defines a hash function $h : S \to \mathbb{Z}$, and the hash table simply reduces it modulo m.

Actually, the "$\mathbb{Z}$" part is not exactly accurate. Typically, the hash values aren't necessarily (unbounded) integers. They are more commonly just 32-bit integers.

Of course, more generally, you can define a unique hash function $h_m : S \to \{0, 1, \dots, m-1\}$ for each distinct m, but you shouldn't bother, since the previous technique via modulo is much simpler, both conceptually and implementation-wise.

2.3 Performance

How good is our hash table? Is it fast? Does it run in $\mathcal{O}(1)$, given that we always maintain the load factor $\frac{n}{m}$ to be $\le U = \mathcal{O}(1)$?

Not necessarily! Remember that distinct keys can have the same hash, which means some chains may be longer than others. Operations on longer chains will naturally be slower.

In the extreme case, consider the case when we use a very simple hash function, say $h : x \mapsto 0$. Since $0 \bmod m$ is always 0, this hash function will force *everything* to end up in a single chain regardless of m, and our hash table degenerates into a list, with all operations running in $\Theta(n)$ time!

It should now be pretty clear that the performance depends heavily on the choice of hash function h. A poor choice of function h could lead to a bad performance.

So which hash function should we choose? We would like to find a *really good hash function* that doesn't make very long chains. We want the keys to somehow be uniformly distributed among the hashes $0, 1, \dots, m-1$. That way, inserting n distinct keys will give us a maximum chain length of $\lceil n/m \rceil = \mathcal{O}(1)$.

But there's a big problem: there's no such hash function! To see this, suppose that the number of distinct keys that exist in the world is s (it may be infinite). The pigeonhole principle guarantees that for any hash function to the set $\{0, 1, \dots, m-1\}$, there will be some hash value such that at least $\lceil s/m \rceil$ keys map to it. Since s is usually very large, even infinite, this means that any hash function is (theoretically) breakable: we can always find several distinct keys with the same hash.

For our hash table, this means that the worst case $\mathcal{O}(n)$ can *always* be triggered, regardless of the hash function we choose—any hash function can be broken. For any fixed hash function, there's always a set of n distinct keys with the same hash.

However, note that I only proved that these keys with the same hash *exist*; I didn't say that they're easy to find. Some simple functions like $h : x \mapsto x$ or $h : x \mapsto ax^2 + bx + c$ are easy to break, but more complicated functions are harder to break.

Exercise 2.2. How do you trigger collisions when $h : x \mapsto x \bmod 2^{32}$?

Therefore, what we want is for our hash values to be "unpredictable" or "random" in some way, that is, we want collisions to be hard to find. This makes the worst case harder to trigger. To make this more precise, here are two properties we want h to satisfy:

- *$h \bmod m$ uniformly distributes the keys to the hashes* $\{0, 1, \dots, m-1\}$. In other words, if we randomly choose a key x from the set of all possible keys, then its hash can be anything in $\{0, 1, \dots, m-1\}$ with (roughly) equal probability.

  In practice, this is not a strict requirement. We can still use hash families that don't distribute the keys uniformly. However, the further it is from uniform, the worse the performance will get.

- *It is difficult to find collisions.* Here, "difficult" depends on context. Usually, it means it is *computationally expensive*, and even this term is vague and depends on the context.

  For example, if you're working on a high-security application, then maybe you'd want it to take years before anyone finds even a single collision. But if you're just competing in a Codeforces round, then maybe it's enough to make sure that finding a collision is not possible within a few hours.

If h satisfies both of these, let's call it a **good hash function**. Note that simpler functions like $h : x \mapsto x$ are not good: they (usually) satisfy the first property but not the second.

What does it mean for us? It means, for example, that we can just assume that keys are essentially "random", since there's no easy way to relate any two distinct keys together. We will assume this when we do our performance analysis later. This is called the simple uniform hashing assumption. As a consequence, we may simply assume that the "probability" that $h(x) \equiv h(x') \bmod m$ is $\frac{1}{m}$ for any two distinct keys m and m'.[7]

So let's now analyze our hash table, assuming it uses a good hash function. We can ignore the dynamic resizing part since they can be *amortized away* just like in vectors. So the cost of an operation on key x is just proportional to the length of the chain of its hash $h(x) \bmod m$. Let's now compute the *expected* length of the chain.

[7]The word "probability" is in quotes because of some subtle technical issues we're sweeping under the rug!

Suppose that n distinct keys were inserted to our table. Then the expected length of $h(x) \bmod m$'s chain can be computed using the linearity of expectation:

$$1 + \underbrace{\frac{1}{m} + \ldots + \frac{1}{m}}_{n-1} < 1 + \frac{n}{m}.$$

The 1 at the beginning accounts for $h(x) \bmod m$ itself. But recall that the structure maintains that $\frac{n}{m} \le U$, so this is just $\mathcal{O}(1)$! This is why we usually say that hash maps run in $\mathcal{O}(1)$ expected time. In contrast, maps implemented with BSTs run in $\mathcal{O}(\log n)$ worst-case time.

Most programming languages have a built-in implementation of hash tables. For example, C++ has `unordered_set` and `unordered_map` and Java has `HashSet` and `HashMap`. Python's built-in `set` and `dict` are also implemented using hash tables.

Maps implemented with binary search trees require the keys to be totally ordered via `<`. The hash table doesn't require comparison. Instead, it requires the keys to be *hashable* to (32-bit or 64-bit) integers, i.e., you need to implement a hash function on your data type such that two equal objects have the same hash. The way to do that depends on the language:

- For C++, you need to implement `hash<T>` where `T` is your data type. You also need to implement equality checking by overloading `operator==`.[8]
- For Java, you need to implement the `hashCode` method and the equality check method `equals`.
- For Python, you need to implement the magic method `__hash__` and the equality check magic method `__eq__`.

For your custom types, it's usually not hard to come up with a hash function. However, it's usually better to delegate the hash computation to built-in ones, e.g., lists, since in most languages, these already have built-in hash functions which we can usually assume to be "good". (Although I suggest checking the details of hashing in your language anyway, e.g., are C++'s default hashes actually good?)

[8]See `http://en.cppreference.com/w/cpp/utility/hash` for more details.

3 List and string hashing

Hashing techniques are applicable well beyond hash tables. Hashing allows us to improve algorithms that require comparing many elements for equality. As mentioned in the first section, we usually do this by replacing comparison of objects by comparison of their hashes.

We start with a simple case: hashing *lists*. We will actually be discussing *strings* since a string is just a list of characters, hence a special case, but the techniques will also apply to lists of any (hashable) type. We also get to learn some standard applications of hashing.

3.1 A standard problem: comparing substrings

Consider the following (standard) problem.

Problem 3.1. Let s be a string of length n. Denote by $s_{i\dots j}$ the substring starting from index i and ending at index j, inclusive. Answer q queries. In each query, you are given four integers i_1, j_1, i_2, j_2, and you are asked to determine whether $s_{i_1\dots j_1}$ is equal to $s_{i_2\dots j_2}$.

The naive solution simply compares both substrings manually. Since substrings can be up to n in length, this takes $\mathcal{O}(n)$ per query, so $\mathcal{O}(qn)$ time overall, which is pretty slow.

We can slightly improve this by noticing that we don't need to compare two strings of different lengths. In other words, if $\mathit{length}(s_{i_1\dots j_1}) \neq \mathit{length}(s_{i_2\dots j_2})$, then we don't have to compare them at all since we already know they're not equal. Additionally, $\mathit{length}(s_{i\dots j})$ is simply $j - i + 1$, so this allows us to answer some queries in $\mathcal{O}(1)$!

We can restate this improvement in terms of hashes. Consider *length* to be a hash function. Then this hash function allows us to reduce the number of queries to answer by simply not comparing substrings with different hashes.

Unfortunately, it may happen that all pairs of query strings are of equal length, so the worst-case running time is not improved and is still $\mathcal{O}(qn)$. This is because *length* is not a particularly good hash function; collisions are easy to find! For example, "`ab`" $\neq$ "`cd`" but $\mathit{length}(\text{“}\mathtt{ab}\text{”}) = \mathit{length}(\text{“}\mathtt{cd}\text{”})$. So clearly, we want to use a hash function that's better than *length*.

Let's use a different one, then. Let's define *sum* as the function that returns the sum of the ASCII values of the letters of the string. For example,

$$\mathit{sum}(\text{“}\mathtt{banana}\text{”}) = 98 + 97 + 110 + 97 + 110 + 97 = 609.$$

Now, consider using *sum* as the hash function. Well, this seems better than *length*; unfortunately it's still not good enough since it's still not hard to break!

Exercise 3.1. Break the *sum* hash.

We can also use a hash called *sumlength* that's basically a combination of *sum* and *length*:

$$\mathit{sumlength}(s) = \mathit{sum}(s) \cdot 2^{32} + \mathit{length}(s).$$

This hash function has the property that $\mathit{sumlength}(s_1) = \mathit{sumlength}(s_2)$ if and only if $\mathit{sum}(s_1) = \mathit{sum}(s_2)$ and $\mathit{length}(s_1) = \mathit{length}(s_2)$ (well, more or less), so it is strictly stronger than either *sum* or *length*. However, it's still quite easy to break.

Exercise 3.2. Break the *sumlength* hash.

3.2 The hashing solution

Whatever function h we choose, our general strategy seems to look like the following:

- Fix a hash function h on strings.
- For every query (i_1, j_1, i_2, j_2):
  - If $h(s_{i_1 \ldots j_1}) \neq h(s_{i_2 \ldots j_2})$, then output **no**.
  - Otherwise, compare $s_{i_1 \ldots j_1}$ and $s_{i_2 \ldots j_2}$ manually. If they are equal, output **yes**, otherwise output **no**.

Ideally, h should be a good-enough hash function, i.e., it should detect unequal strings fairly well so that we minimize the number of times we have to manually compare the strings. I say "good enough" since we are still open to the possibility of *collision*.

The best case is when there are no collisions, but as I've said before, hash functions without collisions are impossible if there are fewer hash values than inputs. Of course, we can get rid of collisions totally if we allow our hash values to be arbitrarily large, but such functions are not useful since they are slow to compute. Indeed, if h has no collisions, then $h(t)$ must essentially encode all information needed to reconstruct t, and that usually means the hash value will be proportional in length to t! Thus, while we get rid of collisions, the running time isn't actually better than simply comparing the strings. As an extreme example, we could choose our hash function to be the identity $h(t) = t$, and we do indeed get rid of collisions, but notice that the algorithm reduces to brute force.

Thus, we impose the additional constraint that $h(s_{i \ldots j})$ *can be computed quickly*. In particular, the hash values will be "small". Ideally, we'd like $\mathcal{O}(1)$, but even something like $\mathcal{O}(\log n)$ should be fine.

However, there's a problem with this approach even if we have a hash function satisfying all these properties: the worst case running time is still $\mathcal{O}(qn)$ since it may happen that all pairs of query strings are equal! This is true no matter what h is.

Let's try to fix this by using the following alternative "solution":

- Fix a hash function h on strings.
- For every query (i_1, j_1, i_2, j_2):
  - If $h(s_{i_1 \ldots j_1}) \neq h(s_{i_2 \ldots j_2})$, then output **no**.
  - Otherwise, output **yes**.

In other words, we completely skip the manual comparison step and use the hash function as a complete substitute for comparison!

The good news is that this is now pretty fast; specifically, the worst case cost becomes $\mathcal{O}(q \cdot t(n))$ where $t(n)$ is the cost of computing the hash of a single substring. The bad news is that this is no longer completely correct because of collisions! As we have seen earlier, collisions are unavoidable.

But how bad is it, exactly? Let's see. Assume that h is a *good* hash function in the sense we described earlier. Thus, we can assume that two distinct strings hashing to the same value only happens by chance,[9] since being *good* implies that *collisions are hard to find*. And by the uniformity property of good hash functions, the chance of two unequal strings getting the same hash is only $\frac{1}{m}$. Obviously, our algorithm never incorrectly says that two equal strings are unequal. This means that the probability of answering a single query correctly is at least $1 - \frac{1}{m}$, and assuming independence of queries, the probability of correctly answering all queries is at least

$$\left(1 - \frac{1}{m}\right)^q.$$

This is strictly less than 1, so our solution may fail. But if we assume that $m \approx 10^9$ and $q \approx 10^5$, then the chance of succeeding is[10]

$$\approx \left(1 - \frac{1}{10^9}\right)^{10^5} \approx 1 - \frac{10^5}{10^9} \approx 99.99\%,$$

which is great! Furthermore, if the judge only prepared 30 test cases, then the probability of our submission getting all test cases correct is

$$\approx \left(\left(1 - \frac{1}{10^9}\right)^{10^5}\right)^{30} \approx 1 - \frac{3 \cdot 10^6}{10^9} \approx 99.7\%,$$

which is a pretty good bet, if you ask me! Thus, the algorithm above will get accepted with a very high probability. We just have to live with the fact that it has a tiny chance of failing.

The only thing that remains is actually choosing our hash function. As we have seen, *length*, *sum* and *sumlength* are not good. The problem with *length* is that it doesn't encode any information about the characters of the string, it only encodes length (obviously), and the problem with *sum* is that, while it does encode information about the characters, it doesn't encode anything about their positions in the string.

We need a hash function that's strong enough to be *good*, but also has the property that substrings can be hashed quickly. There's one such hash function that competitive programmers use, which we'll describe below.

Fix two numbers b and m, and a function v that maps characters to integers (a simple example is the map that takes a character to its ASCII value). We define the **polynomial hash function** $P_{b,m,v}$ as the hash function that takes the string s to

$$P_{b,m,v}(s) = (v(s_0) + v(s_1)b + v(s_2)b^2 + v(s_3)b^3 + \ldots) \bmod m = \left(\sum_{i=0}^{|s|-1} v(s_i)b^i\right) \bmod m.$$

In other words, we consider the characters of s as the coefficients of some polynomial (after converting them to integers via v), and evaluate that polynomial on b modulo m.

The nice thing about the polynomial hash is that it's strong enough to be *good*,[11] but that it's nice enough that there's a way to preprocess the string so that polynomial hashes can be computed in $\mathcal{O}(1)$! We will leave the details of the latter to the reader.

Thus, by using the polynomial hash, we end up with an algorithm that answers all queries in $\mathcal{O}(n + q)$ time.

[9] and not, say, because of the problem setter triggering it with suitably-chosen test data.

[10] using the approximation $(1 + x)^n \approx 1 + nx$ which is pretty good if $|nx| \ll 1$.

[11] Unfortunately, it is very difficult to formally prove that polynomial hashes are good, and in fact, it heavily depends on b and m. But you should be convinced of that intuitively by accepting that polynomials are "random enough" for many choices of b and m. This is all I can offer for now.

One final thing: the quality of the polynomial hash depends strongly on the choice of b and m (and to a lesser extent, v). For example, choosing $b = 1$, the polynomial hash reduces to *sum*, which is pretty bad. Ideally, we want a b such that the sequence $b^0, b^1, b^2, \ldots$ doesn't repeat early modulo m. We also want b to not be too small, e.g., $b = 2$ is a bad choice. (Why?) The choice of m is also important. First of all, it affects the probability of success, so we want to choose a sufficiently large m. There are other considerations, e.g., it turns out that a prime m is a better choice, since other natural choices like 2^{64} turn out to be somewhat weak, and ways to break them are quite well-known now; see `http://codeforces.com/blog/entry/4898`. There might be other considerations, but in general, it's very hard to say whether a choice of b and/or m is good or bad—I mean, it should be pretty clear that this leads into a number theory rabbit hole!

Exercise 3.3. Is $b = 2$ a bad choice? What about $b = m$? $b = m - 1$? Why?

3.3 Another standard problem: counting substrings of a fixed length

Here's another standard problem solvable with hashing, which will also show us a common pitfall with hashing techniques:

Problem 3.2. Given a string s and an integer k, how many distinct substrings of length k are there?

The most obvious solution is the following:

- Let s be a set, initially empty.
- For each substring t of length k, insert t into s.
- Return the size of s.

This is clearly correct, but it is slow. There are $n - k + 1$ substrings of length k, and if $k \approx \frac{n}{2}$, then $n-k+1$ and k are both $\Theta(n)$, so the total complexity is $\mathcal{O}(n^2 \log n)$ where the log factor comes from the usage of the set. Even if we use a hash table with a good hash function for our set, this is still $\mathcal{O}(n^2)$ expected time!

Now, if we have a hash function, then we have the following faster "solution":

- Let s be a set, initially empty.
- Let h be a hash function that maps strings to integers.
- For each substring t of length k, insert its hash $h(t)$ into s.
- Return the size of s.

The idea is that even though the strings are large, the hashes themselves are small. Unfortunately, this is not entirely correct because of collisions!

How can we fix this? Well, maybe we can choose to h to be a *perfect* hash function, i.e., one that has no collisions. However, as explained earlier, such a hash function might take too long to compute, so this is probably no better than brute force; we've just made it more complicated.

A better solution would be to use a *good* hash function. Thus, we get a solution that's correct with some probability. However, it can't be just any good hash function; it must also have the property that we can compute the hashes of all length-k substrings quickly.

Luckily, the *polynomial hash* function still works! This gives us an algorithm that works in $\mathcal{O}(n \log n)$ time. Furthermore, the requirements in this particular problem are a bit weaker since we only need to compute the hashes of length-k substrings. Specifically, it is enough for our hash function h to satisfy the following properties:

- **Right insertion.** If s is a string and c is a character, then $h(sc)$ can be computed quickly given $h(s)$.
- **Left deletion.** If s is a string and c is a character, then $h(s)$ can be computed quickly given $h(cs)$.

If h satisfies these properties, then we can easily compute the hashes of all length-k substrings by repeatedly inserting and deleting letters on both sides and computing the updated hash. We call a hash function satisfying the above a **rolling hash** function.

A simple example of a rolling hash function is *sumlength*, but as we've seen, this is not a good hash function. Luckily, the polynomial hash function is also a rolling hash:

- **Right insertion.** If we insert c to the right, we only need to add $v(c)b^{|s|}$ modulo m to the original hash, i.e., $h(sc) = \left(h(s) + v(c)b^{|s|}\right) \bmod m$. This can be done in $\mathcal{O}(\log |s|)$ with fast exponentiation, or even $\mathcal{O}(1)$ by precomputing the powers of b (since it is fixed).
- **Left deletion.** Similarly, we find out that $h(cs) \equiv v(c) + b \cdot h(s) \pmod m$, thus, we can compute $h(s)$ from $h(cs)$ as follows: $h(s) = b^{-1}\left(h(cs) - v(c)\right) \bmod m$, where b^{-1} is the modular inverse of $b \bmod m$. This only works if b is invertible mod m, but that's easy to guarantee; choosing a prime m is enough. Also, b^{-1} can be precomputed, so computing $h(s)$ from $h(cs)$ only takes $\mathcal{O}(1)$ as well!

Exercise 3.4. By reversing the polynomial hash, show that you can implement a rolling hash without needing the modular inverse of b.

Hence, by using polynomial hashes and a good choice of b and m, we can find the number of distinct substrings of length k in $\mathcal{O}(n \log n)$ time.

3.4 The probability of failure

Not so fast!

We've come up with an algorithm, and have chosen b and m so that the polynomial hash is *good*, but we still need to compute the probability of success! We can't just assume that the probability of success is high, even if our hash function is good.

To see why, let's try to compute the probability of success of our algorithm for the previous problem. Clearly, we will only succeed if, among all length-k substrings, there are no two strings with the same hash. Let's compute that probability in the worst case. For simplicity, let's assume $n = 10^5$ and $m = 10^9$. Note that there are n distinct substrings in the worst case.

This problem turns out to be well-known. This is commonly called the **birthday paradox** and is usually stated as follows: *What are the chances that, among random n people, at least one* *pair of people have the same birthday?* We say it's a paradox because of the surprising/counterintuitive result that the probability can be very high even with just a few people. Indeed, for just $n = 23$, this probability is greater than 50%, and for $n = 60$, the probability is greater than 99%. This is in spite of the fact that there are 365 days in a year!

Generalizing, we may assume that there are m days in a year, and similarly it turns out that even when $n \ll m$, the probability of failing can get pretty high. To approximate the probability, we assume that all comparisons among the $\binom{n}{2}$ pairs are independent.[12] The probability that all comparisons are all different becomes

$$\approx \left(1 - \frac{1}{m}\right)^{\binom{n}{2}}.$$

If we choose $n = 10^5$ and $m = 10^9$, we get the probability of success as[13]

$$\left(1 - \frac{1}{10^9}\right)^{\binom{10^5}{2}} \approx \left(1 - \frac{1}{10^9}\right)^{10^{10}/2} \approx e^{-5} \approx 0.67\%,$$

which is very bad! This will be even worse if you consider that there are multiple test cases; the probability of getting even just 3 test cases right is

$$\approx \left(e^{-5}\right)^3 \approx 0.0000306\%,$$

which is basically zero. Thus, our solution will almost certainly get rejected, even assuming that our hash function is *good*.

The lesson here is to always compute the chances of success when using hashing techniques, even just approximately.

To fix this, we simply have to realize that $m \approx 10^9$ is too low. We need to increase it so that the probability of succeeding becomes higher. The obvious way is to make it so that $m \approx 10^{18}$, that way, the probability of succeeding is

$$\approx \left(1 - \frac{1}{10^{18}}\right)^{10^{10}/2} \approx 1 - \frac{10^{10}/2}{10^{18}} \approx 99.9999995\%,$$

which is basically 100%.

The downside of using $m \approx 10^{18}$ is that computing polynomial hashes will give you some nasty overflow issues, since multiplying two 64-bit integers will probably overflow. Instead, a different solution is to simply use *two* polynomial hashes, h_1 and h_2, with different *coprime* moduli m_1 and m_2, both $\approx 10^9$, and combine both hashes into one (say by forming a pair, or combining them like $h_1(x) \cdot 2^{32} + h_2(x)$). Thus, the probability of failing is $\frac{1}{m_1 m_2} \approx \frac{1}{10^{18}}$ and so we again get the high probability of succeeding as above. But now, the moduli are small, so there are no overflows, yay!

Exercise 3.5. Why do we want m_1 and m_2 to be coprime?

To reiterate, always compute the chances of success when using hashing techniques. As a rule of thumb, if your solution requires c comparisons and the probability of a comparison failing (saying **yes** when it should be **no**) is p, then you should expect the probability of succeeding to be $(1-p)^c$. Typically, $p = \frac{1}{m}$ where m is the number of distinct hashes of of your hash function. But note that c can be really huge, as seen in the example above. As a rule of thumb, you can estimate c to be $\approx n$ if you have to perform n independent comparisons, but $\approx \frac{n^2}{2}$ if you have to compare n objects against one another, say when counting the number of distinct objects.

[12] this approximation is pretty good if $n \ll m$

[13] using the approximation $\frac{1}{e} \approx \left(1 - \frac{1}{n}\right)^n$ which is quite good when n is large

4 Set hashing

Aside from strings and lists, we also sometimes need to hash *sets*. There are obvious ways of hashing a set, e.g., we can just sort the values and compute the list hash, but sometimes, we would like our hash function to have other nice properties, similar to the *rolling* property we discussed earlier.

For example, consider the following problem.

Problem 4.1. You are given a set, initially empty. You are also given q operations. Each operation either inserts an element to or deletes an element from the set. Your task is to find the number of distinct *states* that the set has been in over the course of the q operations.

This is very similar to the distinct-substring problem, except we're dealing with sets. This time, we would like our hash function to satisfy the following properties (on top of being *good* as well):

- **Insert.** If we insert an element, then the new hash can be computed quickly.
- **Delete.** If we delete an element, then the new hash can be computed quickly.

Using such a hash function, we can easily solve the problem.

Exercise 4.1. Solve the distinct-set-hashes problem given such a hash function.

All that remains is to find a hashing scheme that allows us to do the above. Try it yourself first before proceeding!

4.1 Maintaining sorted hashes

One natural solution would be to start with the set hash I mentioned earlier, which is sorting the values and computing the list hash of the sorted list.

For the list hash, we can again use the polynomial hash. To handle updates, we need to make it dynamic, i.e., we need to be able to update the polynomial hash when we insert or delete elements. However, this time, the insertion/deletion can be anywhere in the list, not just at the ends, so the rolling property can't be used anymore.

It turns out that we can dynamically maintain the polynomial hash by building a *segment tree* on top of the sorted list! This will give us an $\mathcal{O}(\log n)$-time algorithm to maintain and update the hash of the set. I will leave the (somewhat complicated) details to the reader.

One advantage of this algorithm is that it can also be applied to *multisets*, i.e., "sets" that allow duplicate elements.

4.2 Zobrist hashing

It turns out that there's a much simpler hashing method than the previous one in the case of sets.

The idea is to first map all possible values to some random number, say a 64-bit integer. Let's say the value x maps to $v[x]$. Then the hash of a set is simply the XOR of all $v[x]$ for all elements x of the set.

Such a hash is very easy to update. For example, if we insert a new element x that doesn't exist in the set yet, then we simply XOR $v[x]$ with the current hash to get the updated hash. Similarly, if we delete an element x that exists in the set, then we also XOR $v[x]$ with the current hash to get the updated hash. Thus, deletion and insertion are the same operation, and they can be performed very quickly: $\mathcal{O}(1)$!

What's more, this hashing is very good against collisions! The probability of two distinct sets getting the same hash becomes $\approx \frac{1}{2^{64}}$.

This is called **Zobrist hashing** and it's actually used by some programs that play chess and Go to compute hashes of game states; the ability of the hash to be updated efficiently makes it suitable for it.[14]

Exercise 4.2. Explain why the probability of collision is $\approx \frac{1}{2^{64}}$.

Note that this algorithm can't be applied to multisets as it is, but there's a way to modify it so it can also support multisets. I leave it to the reader as exercise. (See problem **N8**)

[14] https://en.wikipedia.org/wiki/Zobrist_hashing

5 Tree hashing

On some occasions, you might find yourself needing to hash *trees* as well. Hashing trees (and graphs in general) is easy if we take the labels into account, since a tree is just a set of nodes V and a set of pairs of nodes E, and we already know how to hash sets.

However, in some problems, we would like to find a hash function that returns the same hash for *isomorphic trees*, regardless of how they're labelled/presented. An example is in the problem Tree Isomorphism.

There are different kinds of trees. For example, there are unrooted and rooted trees. Furthermore, there are ordered and unordered rooted trees. We would like to come up with a hashing algorithm for each of them.

5.1 Ordered tree hashing

An *ordered tree* is a rooted tree where the children of each node are ordered in some way. We would like to find a way to hash ordered trees.

Obviously, comparing two ordered trees is pretty simple; just recursively check if the corresponding children are equal. This runs in optimal $\mathcal{O}(n)$ time. However, just like in the case of strings, there is still some value in knowing how to hash ordered trees.

It turns out that there's a pretty simple hash for an ordered tree that runs in $\mathcal{O}(n)$! I will leave it to the reader to discover as an exercise.

Exercise 5.1. Find a good-enough hash function on ordered trees and that can be computed in $\mathcal{O}(n)$ time. Explain why it's "good enough".

Hint: Consider the *list* of children.

5.2 Unordered tree hashing

On the other hand, in *unordered* rooted trees, the children are not ordered. In other words, we don't distinguish trees even if we rearrange their children (or children's children, and so on).

First, how do you even compare two unordered trees without hashing? Clearly, you can't compare corresponding children anymore since they might not be in the same order in both trees.

To correctly compare two unordered trees, let's first ask: what do you call a collection whose ordering doesn't matter? And how do you compare whether two such collections are the same? The answer is a *multiset*, and to compare them, we simply sort them and compare the corresponding elements!

Now, this idea also gives us a method of hashing the tree. Since we already know how to hash multisets, this gives us a hash for unordered trees as well.

To summarize, to hash an unrooted tree, we first recursively hash the children, and then place the resulting hashes in a multiset, and then return the hash of that multiset (say by sorting and computing the list hash). This runs in $\mathcal{O}(n \log n)$ time.

5.3 Unrooted tree hashing

What about unrooted trees? Unfortunately, the previous recursive methods will not work anymore since there's no root! Given two unrooted trees, how do we even begin comparing them?

Well, the key here is to use a standard trick when dealing with unrooted trees: root them first! This is especially helpful since we already know how to hash rooted (unordered) trees. However, which nodes should we choose as roots? Unfortunately, we can't root them arbitrarily since we also need the roots to correspond to each other.

One insight is that we could look for some easily identifiable node in the tree and use that as the root. The easiest one would be the *tree center*: If two unrooted trees are the same and they have unique centers, then rooting them both at the centers will yield equal rooted trees!

This corresponds to a procedure of comparing two unrooted trees, and also a procedure to hash an unrooted tree: find its center, root the tree at the center, and hash the resulting rooted tree.

Unfortunately, this doesn't always work since there are cases when there are two centers! In that case, which center should we pick? Actually, the simplest way would be to simply try *both* centers and return the *smaller* hash!

Exercise 5.2. Show that we can hash an unrooted tree with two centers by rooting it at both centers, hashing both rooted trees, and returning the smaller hash. (In other words, show that two isomorphic trees get the same hash this way.)

The running time is basically the same as that for unordered rooted trees since the center can be found in $\mathcal{O}(n)$ time.

6 More on Hashing

6.1 Hashing in cryptography

Beyond hash tables and competitive programming, hashing also plays a role in cryptography. A **cryptographic hash** is simply a hash function that has some extra properties that make it desirable for use in cryptography. Some examples of useful properties are as follows:

- Collisions are *very* hard to find. In some security applications such as digital signatures, this is important. Usually, the requirements for the difficulty here are much stronger than in competitive programming, since we're dealing with security here. In fact, in some modern cryptographic hashes currently in use, no single collision has ever been found![15] Obviously, collisions exist. They're just really hard to find.

- It's easy to compute, but very hard to *invert*. Given some arbitrary hash value v, it's very hard to find an x whose hash is v, even with the knowledge that such an x exists. This is important in some security applications.

We will not go into more detail regarding the uses of hashing in cryptography since it is beyond the scope of the training material; instead, we refer the interested reader to Wikipedia.[16]

6.2 Further reading

Further reading on string hashing can be found here: `https://www.mii.lt/olympiads_in_informatics/pdf/INFOL119.pdf`

Also, see this post of Makoto Soejima (`rng_58`) for more about good and bad hashes: `http://rng-58.blogspot.jp/2017/02/hashing-and-probability-of-collision.html`

[15]For example, there are no known collisions in the SHA-256 algorithm; any instance of a collision would be very noteworthy and probably worthy of publication. Further reading: `https://crypto.stackexchange.com/questions/52578/are-there-any-well-known-examples-of-sha-256-collisions`

[16]`https://en.wikipedia.org/wiki/Cryptographic_hash_function`

7 Problems

Solve as many as you can! Ask me if anything is unclear.[17] In general, the harder problems will be worth more points.

7.1 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

Exercises mentioned in the main text are worth [5★].

Solve at least [95★].

N1 [5★] In our resizing scheme above for hash tables, show that a sequence of n inserts and deletes runs in $\mathcal{O}(n)$, starting with an empty table. In other words, the resizing cost is amortized $\mathcal{O}(1)$. *Hint:* Assume that $0 < L < M < U$.

N2 [5★] Show that $\mathit{sumlength}(s_1) = \mathit{sumlength}(s_2)$ if and only if $\mathit{sum}(s_1) = \mathit{sum}(s_2)$ and $\mathit{length}(s_1) = \mathit{length}(s_2)$. State your assumptions.

N3 [5★] Give an algorithm that takes a string of lowercase letters as input and outputs a different string of lowercase letters with the same *sumlength* hash if and only if such a different string exists.

N4 [20★] Show how to preprocess a string in $\mathcal{O}(n)$ time so that the polynomial hash of any substring can be computed in $\mathcal{O}(1)$ time. *Bonus:* [5★] Show that this preprocessing can be done without using the modular inverse of b.

N5 [10★] Show how to preprocess a string in $\mathcal{O}(n)$ time to answer the following query in $\mathcal{O}(1)$ time with high probability: given a substring, check if it is a palindrome.

N6 [10★] Show how to preprocess a string in $\mathcal{O}(n)$ time to answer the following query in $\mathcal{O}(\log n)$ time with high probability: given an index i, find the largest palindromic substring centered at index i.

N7 [10★] Show an $\mathcal{O}(n \log n)$-time algorithm to find the longest palindromic substring with high probability. Be careful: The longest palindromic substring can have even length! *Bonus:* [?★] Implement it.

N8 [20★] Modify Zobrist hashing so that it can also hash multisets. It should be computable in $\mathcal{O}(n)$ expected time. Explain why it's "good enough".

[17] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

7.2 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[18] Making the implementation self-contained in a class makes it very easy for you to reuse, which is handy for your future contests!

Solve at least [200★].

C1 [30★] Implement the solution to Problem **N3**.

C2 [20★] Implement the solution to Problem **N5**.

C3 [20★] Implement the solution to Problem **N6**.

C4 [15★] Provide 10^5 distinct strings of lowercase letters with the same *sumlength* hashes. The total length of the strings must not be ridiculously large. Then explain how you found those strings. Please send the program that generates the strings, and explain how it works.

C5 Provide 10^5 distinct strings of lowercase letters with the same polynomial hashes, for each of the (b, m) choices below, and where v takes letters to their ASCII values. The total length of the strings must not be ridiculously large. Then explain how you found those strings. Please send the program that generates the strings, and explain how it works.

(a) [5★] $b = 2$ and $m = 10^9 + 7$.

(b) [5★] $b = 25$ and $m = 10^9 + 7$.

(c) [5★] $b = 112345$ and $m = 10^9 + 9$.

(d) [5★] $b = 577245688$ and $m = 10^9 + 9$.

(e) [5★] $b = 88876050$ and $m = 10^9 + 7$.

(f) [10★] $b = 123456789$ and $m = 10^9 + 9$.

S1 [25★] **Spy Syndrome 2:** https://codeforces.com/problemset/problem/633/C

S2 [25★] **Restoring the Expression:** https://codeforces.com/problemset/problem/898/F

S3 [50★] **Tree Isomorphism:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/tree-isomorphism

S4 [25★] **Palindromic characteristics:** https://codeforces.com/problemset/problem/835/D

S5 [25★] **Sereja and Permutations:** https://www.codechef.com/problems/SEAPERM2

S6 [25★] **Chef and Isomorphic Array:** https://www.codechef.com/problems/ISOARRAY

S7 [25★] **Palindromic Queries:** https://www.codechef.com/problems/ANKPAL

S8 [50★] **Pseudo-Isomorphic Substrings:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/pseudo-isomorphic-substrings

S9 [50★] **Timofey and a flat tree:** https://codeforces.com/problemset/problem/763/D

[18] See also: https://en.wikipedia.org/wiki/Modular_programming.