Hash function

Theory

16 minutes reading

Unskip this topic Start practicing

§1. Introduction

In theory, a **hash function** is any function that can take any large value and output a fixed-sized value. However, not all functions with this property are useful as different use cases need different properties. To understand it better, we will now look at some general properties of hash functions and a few examples of them.

§2. Defining a good hash function

What's the difference between just any hash function and a good one? To define a good hash function, we need to learn three of its properties: efficient, deterministic, and uniform.

An efficient hash function should compute the hash value in constant time: O(1) in the size of the input. Say you have an array of n integers. Then, a good hash function would take time O(n), as n is the size of the input, and would be able to compute hashes for $n > 100,000,000$ in under a second. Now say another hash function takes time O(n^2). Then it could barely compute hashes for $n \sim 10,000$ in one second.

If two inputs are equal, they must have the same hash value, that's why we need a hash function to be deterministic. There are two things to remember here:

- First, deterministic means that the function cannot be random. For example, the function that returns 0 or 1 randomly, independently of the input, is a hash function, but not a deterministic one.
- Secondly, imagine you have two distinct variables, both with the same value, say 7. For a computer, they take different spots in memory, so they are different, but these variables are equal if the values are compared. In this case, we want the hash function to return the same value. An example of a hash function that does not do that is the function that returns the address in memory of the value.

As you already know, the third property is uniform: the hash values are distributed uniformly. This means that the inputs should be mapped equally among the possible hash values. Another way to put it: if we group possible inputs by their hash value, we want the groups to have sizes close to each other.

Those are the general properties that any hash function should have. Let's look at some common ones to see how it works.

§3. Standard hash functions

To use hash functions, we need to learn the notation first. Hash functions are usually denoted by h. The hash value for a particular object x is denoted by $h(x)$. Hash functions used in day-to-day programming usually take one type of value and return integers. They are used in hash tables and have all three properties we discussed above. These are some of the easiest and most commonly used hashes:

1 required topic
Hashing: overview
5 dependent topics
Using external workflows
Hash table
Hashing strings and files
Insecure Cryptographic Storage
JWT Authentication

- Integers: we use the identity function, which always returns the value it is given: $h(x) = x$, or the modulo operation: $h(x) = x \ \% \ p$, for some number p (usually a prime). The modulo operation $x \ \% \ p$ returns the remainder when dividing x by p. Let's see how these functions work for some arbitrary number, say 10. The identity function will always return the number it is given, so we have $h(10) = 10$. For the modulo operation, if we pick $p = 7$, then we have $h(10) = 10 \ \% \ 7 = 3$. If we pick $p = 10$, then we have $h(10) = 10 \ \% \ 10 = 0$.
- Integer arrays: say the array has the form $[v_1, v_2, ..., v_n]$. The idea is to start from left to right and convert each value to a single integer. At each step, multiply the integer you have by some number and add the next value to it. We can write this mathematically as $h_0 = 0, \ h_{i+1} = h_i * p \ + \ v_{i+1}$, where p is some number (usually a prime). Then h_n will be the hash value. Let's look at what we get: $h_n = v_1 * p^{n-1} + v_2 * p^{n-2} + ... + v_n$. This considers every element but in different proportions. What if we take just the normal sum of all elements? Then all arrays with the same elements, possibly in a different order, will have the same hash value. But they are not equal! This trick helps distinguish between such cases.
- General arrays: what if your arrays have some other type of data, rather than integers? First, choose a hash function for that data type, let's call it h', which has the properties above and outputs integers. Then, you can do the same thing as before, but, instead of adding the next value, add its hash. Mathematically, the only change is: $h_{i+1} = h_i * p \ + \ h'(v_{i+1})$, that's it! For example, imagine we want to calculate the hash for an array of integer arrays. We can use the updated hash formula if we choose h' to be a hash function for integer arrays.
- Strings: strings are nothing more than arrays of characters. A character is, in general, an integer between 0 and 255, so we can compute the hash function in the same way as for an array of integers.

To make it clear, let's look at how we compute the hash value for the array $[1, 2, 3, 4]$ using $p = 5$:

- $h_1 = h_0 * p \ + \ v_1 = 0 * 5 \ + \ 1 = 1$
- $h_2 = h_1 * p \ + \ v_2 = 1 * 5 \ + \ 2 = 7$
- $h_3 = h_2 * p \ + \ v_3 = 7 * 5 \ + \ 3 = 38$
- $h_4 = h_3 * p \ + \ v_4 = 38 * 5 \ + \ 4 = 194$

So we have hash value 194. Note that the hash grows fast! Because of this, we usually choose another prime q different from p and much larger, and we modify the step to $h_{i+1} = (h_i * p \ + \ h'(v_{i+1})) \ \% \ q$. The modulo is the remainder of dividing by q, so, this way, the hash value will always be less than q. With a bit of math, we can show that taking the modulo at every step gives the same result as only taking the modulo of h_n, so we keep all the properties that we want.

Now, what happens if we change the order of the values in the array while keeping $p = 5$? For $[2, 1, 3, 4]$ the hash value is $h'_4 = 294$ and for $[1, 2, 4, 3]$ we have $h''_4 = 198$. The closer to the start, the bigger the difference is!

§4. Cryptographic hash functions

Cryptographic hashes are made to work with any input of any length and type by considering it as a sequence of bits: 1's and 0's. They also output a sequence of bits, but of fixed length. The computer can work quite well with it, but for us, it's quite hard to "see" if we don't format it clearly. Typically, there are a few hundred bits in the output, so what we do is consider it as a large number in base 2, convert it to base 16 and write it as a string.

Cryptographic hashes still respect the properties above, but they are more complex. They have fewer use cases but are extremely important in security. Let's see what properties they need:

- Imagine a company that stores a table of username-password pairs for a website you use. If that table gets leaked, then anyone can see your

password. So, what they do is save the hash of your password instead. Whenever you send them your password, they compute the hash and check if it is the same as the one saved. Now, if the table is leaked, an attacker must find a password that hashes to the same value as your password to be able to log in as you. So, the hash function must make it very difficult for an attacker to find such a password. Such a function is called **pre-image resistant**: given a hash value h, it is hard to find a message m with $hash(m) = h$.

- One of the ways we can make sure a message was not changed is by sending the hash of the message together with the message itself. Suppose that an attacker wants to change the message, but cannot change the hash. Then she or he has to find a different message that hashes to the same value. Even if the new message might not make sense, it can affect communication in some ways. To prevent such situations, we can use a **second pre-image resistant**, or **weak collision-resistant** hash function. When we use it, given a message m_1, it will be hard to find a different message m_2 with $hash(m_1) = hash(m_2)$.
- In some very specific use cases, finding any pair of messages that have the same hash might result in problems. A hash function that makes sure it is hard to find any messages m_1, m_2 such that $hash(m_1) = hash(m_2)$ is called **collision-resistant**, or **strong collision-resistant**.

When we say it's hard to find some value, that means that finding a value with the needed properties would take years, even with the most powerful supercomputers. If you think that it's hard to get those properties, you're right! Not everyone can create such a function. Fortunately, there are a few standard ones that are widely used today. Their implementations are complicated, so we will not go into detail.

First, let's look at MD5. It was created in 1992 as a better alternative to its predecessor, MD4. It takes any input and outputs a 128-bit hash value. Initially, it was believed to be collision-resistant, but in 2004 it was proved that it wasn't the case. It took 12 years and a lot of research to find this, so it's better to stick with existing hashes than try and create new ones!

Another cryptographic hash function, a more secure one, is SHA256. Its output is 256-bit long and is used in many places, one of them being the Bitcoin proof of work. Let's look at some examples and see what small changes in the input do to the hash value:

$$SHA256(\text{"}Hashes\ are\ fun!\text{"}) = c128f0e84f60397828b11eaa3cbb67262b99f4d11f3ad630139ffa36cc6002$$

$$SHA256(\text{"}hashes\ are\ fun!\text{"}) = 9659f1fcdf143e3e1f66a922d71500d86c3b7d8f3a5ef03e1d96765476323$$

$$SHA256(\text{"}Hashes\ are\ fun.\text{"}) = b2cde78b638667158fb91d0c665d1d20cc247925b45d9eccb7d25c08721fea1$$

$$SHA256(\text{"}Hashes\ are\ fnu!\text{"}) = c75bb5fed45a21695e0c607376cc1c925939a02c38fac8e7d5488b8820487bc$$

Even small changes produce huge differences! This is the consequence of collision resistance and pre-image resistance.

§5. Conclusion

Now you know what hash functions and cryptographic hash functions are. We learned the standard properties of hash functions: efficient, deterministic, and uniform, and how to build hash functions for some standard data types. We also saw the different use cases of cryptographic hash functions and their importance. Now you are ready to dig deeper and learn some other hashing techniques, for example, hash tables!

Report a typo

407 users liked this piece of theory. **42** didn't like it. **What about you?**

Start practicing

Unskip this topic

This content was created almost 3 years ago and updated 6 days ago. Share your feedback below in comments to help us improve it!

Comments (26) Useful links (3) Show discussion