CRYPTOGRAPHY: HASHING FUNCTIONS

SIDHARTH SHARMA

ABSTRACT. In this article, we discuss primarily Universal Hashing, Hash Tables, and Perfect Hashing. Hashing is used in Bitcoin, cryptographic voting, as well as in computer programming. We will also discuss the requirements of hashing and examples of hashing functions that are broken.

1. Introduction and Requirements

Hashing functions, as the name implies, are simply just functions that map one input to an output. They take an input with length x, and they transform it into a fixed-size string using a mathematical algorithm. So, even if you change the amount of digits in a number and then you hash it, you will still get the same length result. Hash functions are supposed to have these characteristics:

- **Resistance to preimages:** Given x, it is infeasible to find an m such that $H(m) = x$.
- **Resistance to second-preimages:** Given m, it is infeasible to find an m' which is distinct from m and such that $H(m) = H(m')$.
- **Resistance to collisions:** It is infeasible to find m and m', which are distinct from each other, and such that $H(m) = H(m')$.
- **Efficiency:** Hash functions are supposed to be quick at producing the output given the input.

Keep in mind that every hashing algorithm has its kryptonite, which is called a *pathological data set*. This causes set causes a large amount of collisions, which can severely slow the process down.

2. Hash Tables

Hash functions are extensively used in something called a Hash table. The Hash function takes in an input, and then according to the output, it can be put in a table. The input is called a *key*, and the the place where it is "kept" is called a *bucket*.

For instance, let us let our hashing algorithm be a sum-of-digits function (Given a number n, it computes the sum of the digits of n). This is a really bad algorithm[1], and you would probably not want to use it in real life, but it will do for this example.

Date: November 2019.

[1]It is really bad because it has a large amount of collisions, and it is not really even a hash function because it does not convert each input into a same sized output. Try hashing the numbers 73 and 50.

Based on the result of the sum of digits, we will put it in a certain row. We will want 18 rows in our hash table, and ideally, we will want only one bucket per row. However, let's say that we want to hash 64. We put it into a hash table and we put it in row 10, because the sum of its digits is 10. Now let us hash 73. Where do we put it? We can put it in the same bucket, which is called chaining, using a linked list.

This illustrates why collisions are not good at all. The algorithm is slowed down dramatically as it has to iterate over the entire list, and finally get to an open slot, which takes $O(n)$ steps.

Definition 1. The Load Factor α of a hash table is:

$$\alpha = \frac{\text{Number of objects in a hash table}}{\text{Number of buckets in a hash table}}.$$

Ideally, the load factor should be well below 1. You may not be able to control the numerator, but if the load factor increases, you would want to increase the number of buckets.

Theorem 2.1 (Pigeonhole Principle)**.** *Suppose there are n pigeons in m holes. If $m < n$, then there must be a hole containing at least two pigeons. More generally, if $km < n$, then there must be a hole containing at least $k + 1$ pigeons.*

Corollary 2.2. *Given a hash function $h : U \longrightarrow \{1, 2, 3, 4, \ldots, n-1, n\}$, in which U is the universe which contains all the possible keys, and n is the number of buckets, and assuming the $|U|$ is far greater than n, then there must be a collision.*

Proof. This is a simple application of the Pigeonhole Principle. We will let $|U|$ be the number of pigeons, and n is the number of holes, or buckets. And since $|U| > kn$, there must be a hole, or bucket that has that contains more than $k + 1$ pigeons by the Pigeonhole Principle. In other words, there must be a collision. ■

There are 2 main ways to not have a pathological data set. The first way is to use a cryptographic hash function, such as SHA-2. For these, it is infeasible to reverse engineer and find the pathological data set.

The other way is randomization, in which you randomly pick a hash function from a family of hash functions. We will focus more on the second solution.

We will now move on to more types of hashing that try to resolve to resolve collisions.

3. Universal Hashing

Rather than using just using one hashing algorithm, which has its own kryptonite, Universal hashing relies on a family of hashing functions. When a hash function is needed, we can randomly pick a hash function to use. Sure that attacker can find a pathological data

set for one of the hash functions, but it is going to be random every time, so the attacker will not always get the hash function that he has broken.

Definition 4. Let H be a set of hash functions from U to $\{1, 2, 3, 4, \dots, n\}$. H is called *universal* if and only if the probability that $x, y \in U$, with $x \neq y$, collide is less than or equal to $\frac{1}{n}$.

Definition 3.1. A *lookup* for an item in a hash table is a process which consists of the following:

- Locating the bucket the key maps to
- Iterating over the linked list in the bucket to locate the item

Theorem 3.2 (Carter-Wegman)**.** *All operations in universal hash functions run in O(1) time.*[2]

Proof. For proving this, we will analyze the operation of an unsuccessful lookup in hash table.

Let S be the data set and we will look for an x that is not in the table. So, recalling the definition if lookup, the lookup time for the item is:

- First locate the bucket that the key maps to. This operation is takes $O(1)$ time.
- Once we locate the bucket, we iterate over the linked list to locate the item. This operation clearly takes $O(n)$ as we iterate over each and every item (x is not in the table).

Thus, the total time taken is $O(1) + L$, where L is the length of the linked list. Our aim is to find an upper bound on L, which is the expected value of L.

For any $x, y \in U$, with $x \neq y$, $Z_y = 1$ if and only if $h(x) = h(y)$; otherwise it is 0. Therefore, we can get that

$$L = \sum Z_y$$

where y takes all element of S. Now we have gotten this big and complicated variable L and we have broken it down into smaller *indicator* variables.

If we can prove the $E[L]$ is constant, then we have completed the proof. So the expected value of L is:

$$E[L] = \sum E[Zy]$$

Then, we can just use properties of expected values and substitute:

$$E[Z_y] = \sum [\text{Probability}(h(x) = h(y))]$$

[2]In the theorem, $O(1)$ is the expected runtime.

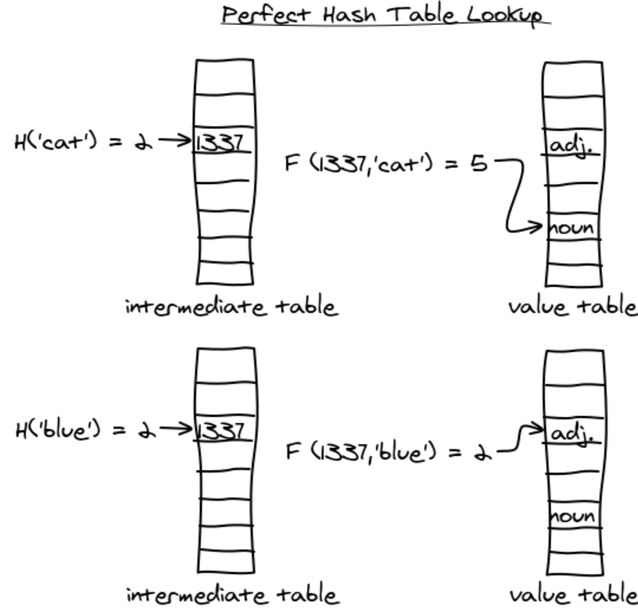

Figure 1. A Lookup in Minimal Perfect Hashing

We know that if we have n buckets, the probability that $h(x) = h(y)$ is at most $\frac{1}{n}$. Therefore,

$$\sum E[Z_y] \leq \sum \frac{1}{n}$$

So this can be expressed as $\frac{|U|}{n}$. Since n is the amount of buckets, this is just α, or the load factor! Hence the expected length of the linked list is constant. ■

4. Minimal Perfect Hashing

Perfect hashing is used for building hash tables with no collisions. However, it is only possible to build one when we know all of the keys prior.

To do this, we can use 2 levels of hash functions. The first one which we will denote $h(\text{key})$. This is the function that gets you a position in an intermediate table or *array.*The second hash function, which we will denote as $f(d, \text{key})$, uses some information, which is d, from the array G to find a position for the key. This is Minimal Perfect Hashing.

But how will we ensure that there are no collisions? We have to carefully make the intermediate array G. A new paper [BBD09] says that we can construct it in linear time.

5. Perfect Hashing

Definition 5.1. A hash function is *perfect* for U if all look ups involve $O(1)$ work.

The first method for this is for constructing perfect hash functions for a given universe U. You would need $O(N^2)$ space for this, where N is the size of our universe.

Theorem 5.2. *If H is universal and $M = N^2$, then $Pr_{h \in H}(\text{no collisions in } U) \geq \frac{1}{2}$.*

Proof. The amount of pairs (x, y) is $\binom{n}{2}$. Then for each pair (x, y), the probability that hey collide is less than or equal to $\frac{1}{m}$. We get this by definition of universal. Thus the probability that there is a collision $\leq \frac{\binom{n}{2}}{m} < \frac{1}{2}$. ∎

6. The MD5 Hashing Algorithm

One of the more well-known cryptographic hash functions is MD5, which was created by Ronald Rivest.

It first divides the input into blocks of 512 bits. Then, 64 bits are appended at the last block. The 64 bits are there to "remember" the length of the original input. Also, if the last block is less than 512 bits, we would have to "pad" the end of the block.

Next, each block is further separated into 16 words of 32 bits each. We will denote these blocks as $M_0, M_1, \ldots, M_{15}$.

Then there are these "buffers", which are the words A,B,C, and D:

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

Then, MD5 also has 4 auxiliary operators:

```
F(X,Y,Z) = (X and Y) or (not(X) and Z)
G(X,Y,Z) = (X and Z) or (Y and not(Z))
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X or not(Z))
```

The table that MD5 uses has 64 elements, which are indicated as K_i, is computed prior to speed up the computation. To compute the elements, MD5 uses a function, which is: $K_i = |\sin(i + 1)| \cdot 232$.

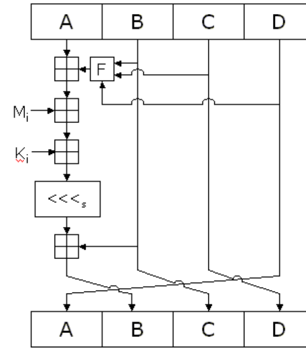

Figure 2. An example of an operation

Now to process each of the blocks, MD5 mixes the contents of the buffers with the input using the four functions. There are 4 rounds, which is 16 operations. Here is an example of an operation. To see an example of an operation, refer to Figure 2.

MD5 has been broken, with respect to collisions. They have found a pathological data set, which causes a ton of collisions. It is recommended that MD5 not be used. For more information, read [WY05].

References

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 682–693, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.