Probability of Collisions

Step 1

Up until this point, we have discussed **Hash Tables** and how they work: you have some array of length *M*, and to insert a key *k* into the array, you **hash** *k* to some index in the array and perform the insertion. However, we even though we addressed the fact that *collisions* can occur (i.e., multiple keys can **hash** to the same index in the array), we chose to effectively ignore them. We emphasized that *collisions* are the cause of slowness in a **Hash Table**, but aside from just repeatedly stating that collisions are bad for performance, we just glossed over them.

In this section, we will begin to chip away at *collisions* by focusing on *why they occur*, and as a result, *how* we can go about *avoiding* (or at least *minimizing*) them. We still won't be discussing *why they cause slowness* in a **Hash Table** nor how to go about *resolving* them, but these equally important topics will be extensively discussed soon. For now, just remember that collisions are bad for performance, and because of this, we want to design our **Hash Table** in such a way that we minimize them.

Thus far, we have witnessed two components of a **Hash Table** that we are free to choose:

- A **Hash Table** is backed by an array, so we have the power to choose the **array's size** (i.e., the **Hash Table's** *capacity*)
- A **Hash Table** computes indices from keys, so we have the power to choose the **hash function** that does this index mapping

To figure out how we can optimally design these two parameters of a **Hash Table**, we will take a journey through the realm of probability theory to see how we can probabilistically minimize collisions. Formally, we will try to find a way to compute the *expected number* of collisions, and we will then attempt to minimize this value.

Step 2

As we mentioned, our goal now is to figure out how we can *minimize* the expected number of collisions. To do so, we must turn to probability to begin to figure out how to even calculate the probability of encountering a collision in the first place.

To analyze the probability of collisions, we will use this basic identity to simplify our computation:

$$P(event) = 1 - P(no\ event)$$

The identity above can be read as "the probability of an event occurring is the same as 1 minus the probability of the event *not* occurring." The basic idea surrounding this identity is that, at the end of the day, there is a 100% chance that an event either will or will not occur (i.e., $P(event) + P(no\ event) = 1$). Consequently, we will say that, for a **Hash Table** with *M* slots and *N* keys, the probability of seeing at least one collision is $P_{N,M}(\geq 1\ collision) = 1 - P_{N,M}(no\ collision)$. For our purposes, we will assume that all slots of the **Hash Table** are equally likely to be chosen for an insertion of some arbitrary key.

Step 3

Let's apply the equation we learned in the last step, $P_{N,M}(\geq 1\ collision) = 1 - P_{N,M}(no\ collision)$, to calculate the probability of encountering a collision.

For example, suppose we have a **Hash Table** where *M* = 1 (i.e., our **Hash Table** has 1 slot) and *N* = 2 (i.e., we have 2 keys to insert). We know, this doesn't seem like the smartest decision, but suspend your disbelief for the sake of argument.

1. What is the probability that our ***first*** key does *not* face collision upon insertion in our not-so-smart **Hash Table**? The answer is clearly 100%: all the slots in the **Hash Table** are empty, meaning there is nothing for our first key to collide with.
2. Now, given that the first key did not face a collision, what is the probability that our ***second*** key does *not* face collision upon insertion in our not-so-smart **Hash Table**? The answer is 0%: all the slots in the **Hash Table** are now full, so it is impossible for

us to find an empty slot in which to insert our second key.

Now, if we want to calculate $P_{2,1}(\geq 1\ collision)$, we first need to figure out how to calculate $P_{2,1}(no\ collision)$. As you may have noticed, there are multiple events we need to take into account (the *first* **and** *second* key not facing a collision upon insertion) in order to figure out the *overall* probability of a collision not happening, and that these events are *conditional*: the outcome of earlier events affects the probabilities of later events. To refresh your memory about conditional probabilities, if we have two events *A* and *B* (dependent or independent: doesn't matter), we can compute the probability of *both* events occurring like so:

$$P(A\ and\ B) = P(A)P(B \mid A) = P(B)P(A \mid B)$$

Hopefully you can see that we are dealing with *conditional probabilities*: when we attempt to calculate the probability that the *i*-th key doesn't face a collision, we assume that each of the previous *i*−1 keys did not face collisions when they were inserted. Formally:

$$P_{N,M}(no\ collision) = P_{N,M}(1^{st}\ key\ no\ collision) * P_{N,M}(2^{nd}\ key\ no\ collision \mid 1^{st}\ key\ no\ collision)...$$

If you are uncomfortable with this notion of *conditional probability* or with basic probability in general, we recommend that you read this before you move on.

Using the equation above: $P_{N,M}(no\ collision) = 1 * \frac{M-1}{M} * \frac{M-2}{M} ... \frac{M-N+1}{M}$

We can therefore conclude that $P_{2,1}(\geq 1\ collision) = 1 - P_{N,M}(no\ collision) = 1 - (1 * 0) = 1$ or *100%*.

Let us now turn to a more complicated example in the next step.

Step 4

Let's see what happens when we increase the size of our **Hash Table** to *M* = 3 and we wish to insert *N* = 3 keys.

1. What is the probability that our ***first*** key does *not* face collision upon insertion in this **Hash Table**? 100%, because the entire backing array is empty, so there is nothing for our first key to collide with
2. Now, given that the first key did not face collision, what is the probability that our ***second*** key does *not* face collision upon insertion this **Hash Table**? 2/3 or 66.67%, because one of the three slots of the **Hash Table** is already full with the first key, so the second key can settle in any of the remaining 2 spots to avoid collision
3. Now, given that *neither* of the first two keys faced collisions, what is the probability that our ***third*** key does *not* face collision upon insertion in this **Hash Table**? 1/3 or 33.33%, because two of the three slots of the **Hash Table** are already full, so the third key can only settle in the remaining 1 spot to avoid collision

Therefore, $P_{3,3}(\geq 1\ collision) = 1 - P_{3,3}(no\ collision) = 1 - (1 * \frac{2}{3} * \frac{1}{3}) = 1 - \frac{2}{9} = \frac{7}{9} \approx 77.8\%$! This is definitely a bit concerning. We expected 3 keys to be inserted and thus created 3 slots, so why is probability of seeing a collision so high? Should we have created *more* than 3 slots?

Step 5

EXERCISE BREAK: Suppose you have a **Hash Table** that has 500 slots. It currently holds 99 keys, all in different locations in the **Hash Table** (i.e., there are no collisions thus far). What is the probability that the next key you insert will cause a collision? Input your answer in decimal form and round to the nearest thousandth.

Step 6

EXERCISE BREAK: Calculate $P_{3,5}(\geq 1\ collision)$. Input your answer in decimal form and round to the nearest thousandth.

Hint: $P_{N,M}(\geq 1\ collision) = 1 - P_{N,M}(no\ collision)$

Hint: Feel free to go back to an earlier page and use the steps we went through to calculate your answer.

To solve this problem please visit https://stepik.org/lesson/31371/step/6

Step 7

Referring back to the earlier example, we had 3 keys to insert, so we chose to create a **Hash Table** with a capacity of 3 in order to fit all the keys. However, we calculated an alarmingly high collision probability of 77.8% with that construction. What exactly is going on?

This is actually a pretty crazy phenomenon and it is illustrated best by the **Birthday Paradox.** The goal of the **Birthday Paradox** is to figure out how likely is it that at least 2 people share a birthday given a pool of *N* people.

For example, suppose there are 365 slots in a **Hash Table**: *M* = 365 (because there are 365 days in a non-leap year). Using the equations from the previous steps, we see the following for various values of *N* individuals:

- For *N*=10, $P_{N,M}(\geq 1\ collision) = 12\%$
- For *N*=20, $P_{N,M}(\geq 1\ collision) = 41\%$
- For *N*=30, $P_{N,M}(\geq 1\ collision) = 71\%$
- For *N*=40, $P_{N,M}(\geq 1\ collision) = 89\%$
- For *N*=50, $P_{N,M}(\geq 1\ collision) = 97\%$
- For *N*=60, $P_{N,M}(\geq 1\ collision) = 99+\%$

So, among 60 randomly-selected people, it is almost certain that at least one pair of them will have the same birthday. That is crazy! The fraction $\frac{60}{365}$ is a mere 16.44%, which is how full our **Hash Table** of birthdays needed to be in order to have almost guaranteed a collision. How can we use this intuition to help us build a **Hash Table** that minimizes the number of collisions?

Step 8

We prefaced this discussion about probability with the intention of selecting a capacity for our **Hash Table** that would help us *avoid* collisions in the average case. Mathematically, this translates into the following question: "What should be the capacity of our **Hash Table** in order to keep the *expected* (i.e., average-case) number of collisions relatively small?" Let's see how we can go about computing the **expected total number of collisions** for *any* arbitrary **Hash Table**.

Suppose we are throwing *N* cookies into *M* glasses of milk. The first cookie lands in some glass of milk. The remaining *N*−1 cookies have a probability $\frac{1}{M}$ of landing in the *same* glass of milk as the first cookie, so the average number of collisions with the *first* cookie will be $\frac{N-1}{M}$ (i.e., $\frac{1}{M}$ added *N*−1 times to account for the same probability for each remaining cookie left to throw).

Now, let's say that the second cookie lands in some glass of milk. The remaining *N*−2 cookies each have probability $\frac{1}{M}$ of landing in the same glass of milk as the second cookie, so the number of collisions with the second cookie will be $\frac{N-2}{M}$, etc.

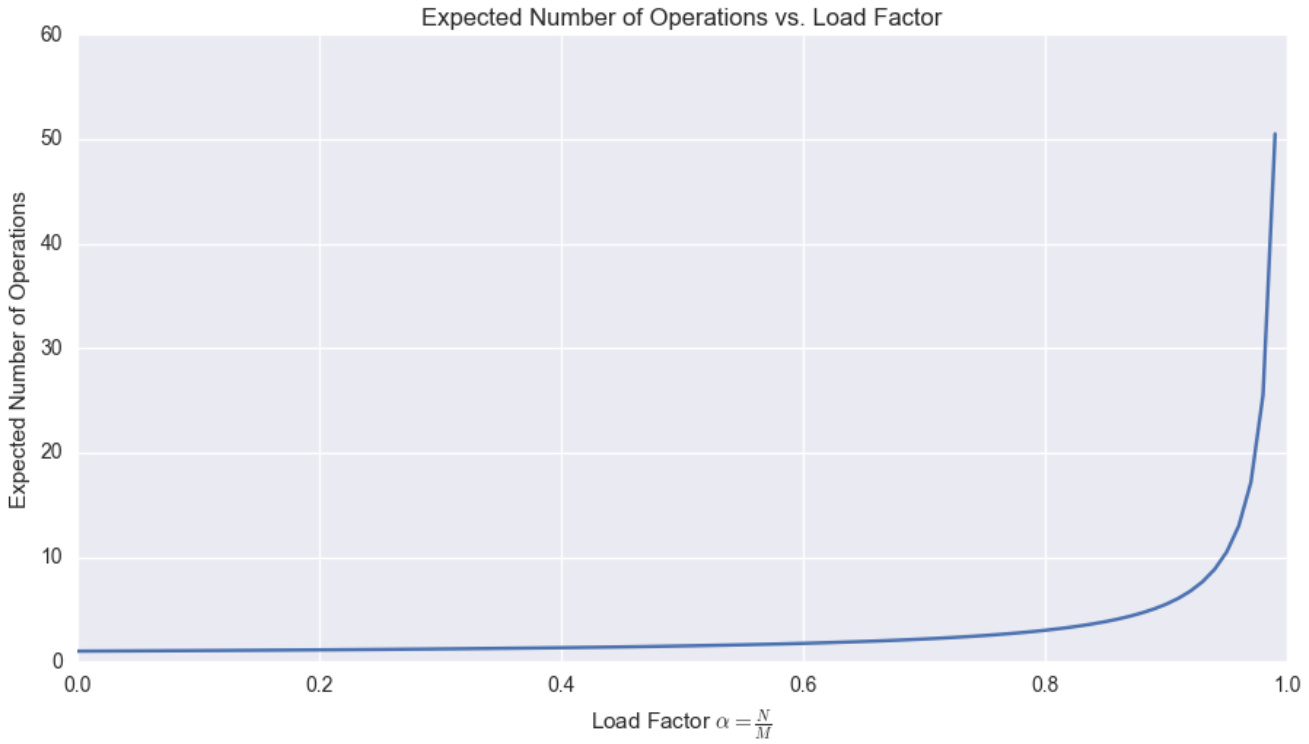
So, the **expected total number of collisions** is $\sum_{i=1}^{N-1} \frac{i}{M} = \frac{N(N-1)}{2M}$

If we want there to be 1 collision on average (which translates into the O(1) average-case time complexity we desire), we can see that the expected number of collisions will be 1 when $\frac{N(N-1)}{2M} = 1$, which, for a large *M*, implies $N = \sqrt{2M}$. In other words, if we will be inserting *N* elements into our Hash Table, in order to keep the make the expected number of collisions equal to 1, we need our **Hash Table** to have a capacity $M = \mathrm{O}(N^2)$.

We hope that the underlying logic is somewhat intuitive:the more extra space you have, the lower the expected number of collisions. As a result, we also expect a better average-case performance. However, as you might have inferred, this is extremely wasteful space-wise! In practice, making the expected number of collisions exactly 1 is a bit overkill. It turns out that, by some proof that is a bit out-of-scope for this text, we can formulate the expected number of operations a **Hash Table** will perform as a function of its **load factor** $\alpha = \frac{N}{M}$:

$$E(number\ of\ operations) = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

If we were to plot this function, we would get the following curve:



Notice that the expected number of operations stays pretty flat for quite some time, but when it reaches a load factor somewhere around 0.75, it begins to jump up drastically. By general "rule of thumb," it has been shown that we can experience O(1) performance on average when $\alpha = \frac{N}{M} \approx 0.75 \rightarrow M \approx 1.3N$. As a result, if we have a prior estimate of the magnitude of *N*, the number of elements we wish to insert, we would want to allocate a backing array of size approximately 1.3*N*. Also, throughout the lifetime of a specific instance of our **Hash Table**, if we ever see that the **load factor** is approaching 0.75, we should consider resizing our array (typically to twice the prior capacity) to keep it low.

With regard to resizing the backing array, however, you should note that, because one step of our indexing process was to mod by *M* in order to guarantee valid indices, we would need to re-hash (i.e., reinsert) all of the elements from the old array into the new one from scratch, which is a O(*n*) operation. In order to avoid performing this slow resizing operation too often, we want to make sure to allocate a reasonably large array right off the bat.

Step 9

We have now seen why, based on probabilistic analysis, it is important to use a **Hash Table** that has a capacity larger than the amount of keys we expect to insert in order to avoid collisions. However, it is important to note that we assumed that our keys were always equally likely to be **hashed** to each of our *M* indices; is this always a fair assumption?

Not *really*. Take for example the pair of **hash functions** $h(k) = 3k$ and $H(k) = h(k)\ \%\ M$, where *k* is the key, *M* is the capacity of the **Hash Table**, and $H(k)$ returns the index. As you can see, $h(k)$ only returns hash values that are multiples of 3. This shouldn't be a problem, *unless* the capacity of the **Hash Table**, M, happens to be a multiple of 3 as well (e.g. 6). If that happens, then it will never be possible to have $H(k)$ produce an index that *isn't* also a multiple of 3 because of the nature of the mod operation (e.g. 3(1) % 6 = index

3, 3(2) % 6 = index 0, 3(3) % 6 = index 3, etc.). Consequently, there will be some indices in the **Hash Table** that will always be empty (indices 1, 2, 4, and 5 in this example). As a result, by having the keys want to map to the same indices, we face a largely unequal distribution of probabilities across our **Hash Table**, and as a result, an increase in collisions.

What's the solution? We avoid getting trapped in the problem described above by always choosing the **capacity of our Hash Table to be a prime number.** By definition, a prime number is a natural number that that has no positive divisors other than 1 and itself. Consequently, modding by a prime number will guarantee that there are no factors, other than 1 and the prime number itself, that will cause the mod function to never return some index values. For example, using the pair of **hash functions** above with $M = 7$ we get the following equal distribution of hash values: 3(1) % 7 = index 3, 3(2) % 7 = index 6, 3(3) % 7 = index 2, 3(4) % 7 = index 5, 3(5) % 7 = index 1, 3(6) % 7 = index 4, 3(7) % 7 = index 0, etc.

Thus, when we choose a capacity for our **Hash Table**, even when we are resizing the **Hash Table**, we always want to round the capacity to the next nearest prime number in order to ensure that that $H(k)$ doesn't yield an unequal distribution. *However*, it is very important to note that while prime numbers smooth over a number of flaws, they don't necessarily solve the problem of unequal distributions. In cases where there is still not a good distribution, the *true* problem lies in a poorly developed hash function that isn't well distributed in the first place.

Step 10

What does all of this analysis tell us at the end of the day? All of the scary math we trudged through gave us formal proof for the following **Hash Table** design suggestions:

- To be able to maintain an average-case time complexity of O(1), we need our **Hash Table's** backing array to have free space
- Specifically, if we expect to be inserting *N* keys into our **Hash Table**, we should allocate an array roughly of size ***M* = 1.3*N***
- If we end up inserting more keys than we expected, or if we didn't have a good estimate of *N*, we should make sure that our **load factor** $\alpha = \frac{N}{M}$ never exceeds **0.75** or so
- Our analysis was based on the assumption that every slot of our array is equally likely to be chosen by our indexing **hash function**. As a result, we need a **hash function** that, on average, spreads keys across the array randomly
- In order to help spread keys across the array randomly, the size of our **Hash Table's** backing array should be **prime**

The analysis itself illustrated that, as the size of our data grows, collisions become inevitable. We have now discussed how to try to *avoid* collisions quite extensively, but we are still not equipped to *handle* collisions should they occur. In the next few sections, we fill finally discuss various **collision resolution strategies**: how our **Hash Table** should react when different keys map to the same index.