Algorithms II November 3, 2025

Lecture 13: Hashing

Notes by Ola Svensson[1]

In this lecture we introduce hashing and show how to construct a family of 2-universal hash functions. Finally, we discuss the power of two choices. The notes are largely based on

- The lecture notes of Lecture 1 in Sanjeev Arora's course "Advanced Algorithm Design" available here: *https://www.cs.princeton.edu/courses/archive/fall16/cos521/*

with some ingredients from

- The lecture notes of Lecture 3-4 in Shayan Oveis Gharan's course "CSE 521: Design and Analysis of Algorithms I" available here: *http://courses.cs.washington.edu/courses/cse521/17wi/*

1 Hashing: preliminaries

Hashing is a technique of mapping the input data (images, vectors etc.) of arbitrary size to a finite set of hash values using a suitable hash function. Usually a special data structure called hash table is created to store the hash values, which makes data search, insertion and deletion faster. Suppose we have a set of large images (say each of size 1 MB) and we want to store them.

Since each image has $1,000,000$ bits, we assume that we have a universe of numbers of all possible images $U = \{1, 2, \ldots, 2^{1000000}\}$. Say we want to store our images in a table of size N. Ideally we want $N \ll |U|$. So, we need a function $h : U \to \{1, 2, \ldots N\}$. Usually h is called a hash function. The question that we want to study is how to choose h.

More formally, we want to store a subset S of a large universe U, where $|U| \gg |S|$. For each $x \in U$, we want to support three operations

- *insert(x)*: insert x into S.
- *delete(x)*: delete x from S.
- *query(x)*: check whether $x \in S$.

A hash table can support all these three operations. We design a hash function $h : U \to \{1, 2, \ldots N\}$ such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size N.

The choice of the hash function may depend on the nature of the input data and their distribution. An ideal hash function should have the property that the probability that two or more input samples getting mapped to the same hash value is low, that is it should be almost injective. In other words, we wish to minimize the probability of having a collision. If two or more samples map to the same hash value, we store them in a linked list whose address is stored at the location of the hash value in the hash table. So, ideally we want the length of the largest list to be as small as possible to minimize the time to query a given image.

At first, one might suggest a function that maps each image $h(X_i) = i \bmod N$. But, if all of our images have the same reminder modulo N, then they all map to the same location of the hash-table and the hashing is useless. In general, for a fixed hash function, we cannot expect to prove any worst-case guarantee. So, instead we choose our hash function h from a family of functions $\mathcal{H}$ and we show that a random function chosen from $\mathcal{H}$ has a small number of collisions.

So, the question is how should we choose $\mathcal{H}$. Ideally, we want to choose $\mathcal{H}$ such that a random function maps each image to a uniformly and independently chosen location of the hash-table. To see

[1] **Disclaimer:** These notes were written as notes for the lecturer. They have not been peer-reviewed and may contain inconsistent notation, typos, and omit citations of relevant works.

that this would lead to few collisions, let L_x be the length of the linked list containing x; this is just the number of elements with the same hash value as x. Let random variable

$$I_y = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases}$$

So $L_x = 1 + \sum_{y \in S: y \neq x} I_y$. Furthermore, remember that expectation is a linear operator: the expectation of the sum of random variables is the sum of their expectations. (This simple fact saves the day in many probabilistic calculations.)

$$\mathbb{E}[L_x] = 1 \sum_{y \in S: y \neq x} \mathbb{E}[I_y] = 1 + \frac{|S| - 1}{N}. \tag{1}$$

Usually, we choose $N > |S|$, so this expected length is less than 2.

However, the problem is that to record the uniform hash function, we need $|\mathcal{U}| \log N$ bits, which is too big. This together with the observation that the above calculation (1) does not need full independence (pairwise independence actually suffice), motivates the next idea. Instead of choosing uniformly from all the possible mappings, we choose uniformly from a smaller set of functions with limited independence.

2 2-Universal Hash Families

Definition 1 (Carter Wegman 1979) *Family $\mathcal{H}$ of hash functions is 2-universal if for any $x \neq y \in U$,*

$$\mathbb{P}_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{N}.$$

Sometimes this definition is relaxed to allow $2/N$ (or $3/N$) instead of $1/N$, since that doesn't greatly affect the bucket sizes need to handle collisions.

We can design 2-universal hash families in the following way. Chose a prime $p \in \{|U|, \ldots, 2|U|\}$, and let

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0).$$

And let

$$h_{a,b}(x) = f_{a,b}(x) \mod N.$$

The reason this construction works is that the integers modulo p form a field when p is prime, meaning that it is possible to define addition, multiplication, and division (except division by 0, which is undefined) among them.

Lemma 2 *For any $x \neq y$, and $s \neq t$, the following system*

$$\begin{aligned} ax + b &= s \mod p \\ ay + b &= t \mod p \end{aligned}$$

has exactly one solution.

Proof Since $[p]$ constitutes a finite field, we have that $a = (x - y)^{-1}(s - t)$ and $b = s - ax$. ■

Notice that the above proof also says that there is no solution with $a \neq 0$ if $s = t$. Moreover, there are $p \cdot (p - 1)$ different choices of $a \in \{1, 2, \ldots, p - 1\}$ and $b \in \{0, 1, \ldots, p - 1\}$. Therefore, ($a$ and b are selected uniformly at random)

$$\mathbb{P}_{a,b}[f_{a,b}(x) = s \wedge f_{a,b}(y) = t] = \begin{cases} \frac{1}{p(p-1)} & \text{if } s \neq t, \\ 0 & \text{if } s = t. \end{cases} \tag{2}$$

Lemma 3 *$\mathcal{H} = \{h_{a,b} : a, b \in [p] \wedge a \neq 0\}$ is 2-universal.*

Proof For any $x \neq y$,

$$\begin{aligned}
\mathbb{P}\left[h_{a,b}(x) = h_{a,b}(y)\right] &= \sum_{s,t \in [p]} \mathbb{1}_{s=t \bmod N} \cdot \mathbb{P}\left[f_{a,b}(x) = s \wedge f_{a,b}(y) = t\right] \\
&= \frac{1}{p(p-1)} \sum_{s,t \in [p]: s \neq t} \mathbb{1}_{s=t \bmod N} \qquad \text{(by (2))} \\
&\leq \frac{1}{p(p-1)} \frac{p(p-1)}{N} \\
&= \frac{1}{N}\,.
\end{aligned}$$

The inequality follows because for each $s \in [p]$, we have at most $(p-1)/N$ different t such that $s \neq t$ and $s = t \bmod N$. ■

Notice to store a hash function from our 2-universal hash family, we only need to store $a \in [p]$ and $b \in [p]$. This requires $O(\log |U|)$ space which is in stark contrast to the completely random hash function which required $O(|U| \log N)$ bits. Let us now calculate the expected number of collisions:

$$\sum_{x \neq y \in S} \mathbb{P}_{h \in \mathcal{H}}\left[h(x) = h(y)\right] \leq \binom{|S|}{2} / N\,. \tag{3}$$

Hence, if we select N to be greater than $|S|^2$, then we can have no collisions with high probability using 2-universal hashing. But in reality, such a large table is often unrealistic. A more practical method to deal with collisions is to use the aforementioned linked list or to use two layer hash tables.

Two layer hash tables work as follows. Let s_i denote the number of collisions at location i. If we can construct a second layer table of size $\approx s_i^2$, we can easily find a collision-free hash table to store all the s_i elements.Thus the total size of the second-layer hash tables is $\sum_{i=1}^{N} s_i^2$. Note that $\sum s_i(s_i - 1)$ is just the number of collisions calculated in Equation (3) (times 2), so

$$\mathbb{E}\left[\sum_i s_i^2\right] = \mathbb{E}\left[\sum_i s_i(s_i - 1)\right] + \mathbb{E}\left[\sum_i s_i\right] \leq \frac{|S|(|S| - 1)}{N} + |S| \leq 2|S|\,,$$

where we assume that our hash table has capacity $N \geq |S|$.

Do we need to know the size of the set? The above calculation shows that if N, the size of the hash table, is roughly the same as N, the size of the set being hashed, then the expected bucket size at each hash location (and hence the expected lookup time) is at most $2m/n$, which is $O(1)$. This leads to the question: *Is it necessary to know $|S|$ before we pick the size of the hash table?* The answer is no. Instead it suffices to adaptively increase the size of the hash table. Suppose the current hash table has size 2^i. As elements of the set arrive, keep hashing them until the expected bucket size rises above 2. This means that the set must be now of size about 2^i. Now *rebuild* the hash table to one of size 2^{i+1} using a new hash function. The total time spent in rebuilding is only $O(2^{i+1})$, which is still a constant factor times the size of the set we already have, which is 2^i. Thus the entire hashing takes $O(1)$ time per element (this is called amortized analysis).

2.1 Pairwise independence

The proof of 2-universality can also be adapted (allow $a = 0$) to give pairwise independence: We say $\mathcal{H}$ is 2-wise independent if for any $x \neq y$ and any pair of numbers $s, t \in [N]$,

$$\mathbb{P}_{h \in \mathcal{H}}\left[h(x) = s, h(y) = t\right] = \frac{1}{N^2}\,.$$

We also note that 2-wise independence implies 1-wise independence. We say that a family $\mathcal{H}$ is 1-wise independent if for any $x \in U$ and $s \in [N]$,

$$\mathbb{P}_{h\in\mathcal{H}}\left[h(x) = s\right] = \frac{1}{N}\,.$$

More generally, we remark that we can extend the above techniques and obtain a family of hash functions that is k-wise independent. For some prime number p, consider the family of functions

$$f_{a_0,\ldots,a_{k-1}}(x) = a_{k-1}x^{k-1} + \ldots + a_1 x + a_0,$$

where $a_0, \ldots, a_{k-1}$ are uniformly chosen in $\{0, 1, \ldots, p-1\}$. Similar to the invertible argument we used above, the proof that this construction is k-wise independent follows from the fact that the Vandermonde matrix

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_k \\ x_1^2 & x_2^2 & \cdots & x_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{k-1} & x_2^{k-1} & \cdots & x_k^{k-1} \end{bmatrix}$$

is invertible for distinct $x_1, \ldots, x_k$. So, in general we can store a k-wise independent hash function with only $O(k \log |U|)$ amount of memory.

3 Load balancing

Now we think a bit about how large the linked lists (i.e., number of collisions) can get. Let us think for simplicity about hashing n keys in a hash table of size n. Also assume for simplicity, assume that each ball is assigned to a random bin. (In other words, the hash function is completely random instead of just 2-universal as above.) This is the famous balls-and-bins calculation.

Clearly, the expected number of balls in each bin is 1. But the maximum can be a fair bit higher. For a given i,

$$\mathbb{P}\left[\text{bin}_i \text{ gets more than } k \text{ elements}\right] \leq \binom{n}{k} \cdot \frac{1}{n^k} \leq \frac{1}{k!}\,.$$

(This uses the union bound, that the probability that any of the $\binom{n}{k}$ events happen is at most the sum of their individual probabilities.) By Stirling's formula,

$$k! \approx \sqrt{2nk}\left(\frac{k}{e}\right)^k\,.$$

So if we choose $k = O\left(\frac{\log n}{\log\log n}\right)$ then $\frac{1}{k!} \leq \frac{1}{n^2}$. Hence

$$\mathbb{P}\left[\exists \text{ a bin } \geq k \text{ balls}\right] \leq n \cdot \frac{1}{n^2} = \frac{1}{n}\,.$$

So with probability larger than $1 - 1/n$,

$$\text{max load} \leq O\left(\frac{\log n}{\log\log n}\right)\,.$$

By changing the parameters little, this success probability can be improved to $1 - 1/n^c$ for any constant c.

Exercise: Show that with high probability the max load is indeed $\Omega(\log n / \log\log n)$.

3.1 Improved load balancing: Power of two choices

The above load balancing is not bad; no more than $O\left(\frac{\log n}{\log \log n}\right)$ balls in a bin with high probability. Can we modify the method of throwing balls into bins to improve the load balancing? How about the method you use at the supermarket checkout: instead of going to a random checkout counter you try to go to the counter with the shortest queue? In the load balancing case (especially in distributed settings) this is computationally too expensive: one has to check all n queues. A much simpler version is the following: when the ball comes in, pick 2 random bins, and place the ball in the one that has fewer balls. Turns out this modified rule ensures that the maximal load drops to $O(\log \log n)$, which is a huge improvement. This is called the *power of two choices*. The intuition why this helps is that even though the max load is $O(\log n / \log \log n)$, most bins have very few balls. For instance at most 1/10th of the bins will have more than 10 balls. Thus when we pick two bins randomly, the chance is good that the ball goes to a bin with constant number of balls. For a proof of this cool fact, I recommend various lecture notes around the web.