Hashing I : Universal Hash Functions[1]

- ***Hashing.*** Consider a universe U of objects which can be represented as integers (formally, a countable set). We use N to denote the size $|U|$ and this can be *huge*; think of U as the set of ASCII strings of length ≤ 50. Since each ASCII character has 128 possibilities, this set is of size 128^{50}. Now imagine there is a certain subset $D \subseteq U$ of $|D| = m$ of these strings we are interested in. In particular, we wish to store them in such a way that given any candidate string $x \in U$, we can answer the query "is $x \in D$" really fast. This operation is called the SEARCH operation.

- Let us consider two solutions. One could store the set D as a huge bit-array $A[0 : N-1]$ where $A[s] = 1$ iff $s \in D$ and 0 otherwise. In that case, given any $x \in U$, one can quickly answer the SEARCH query by accessing $A[x]$, which takes $O(1)$ time in the RAM model of computation. The issue with this is that the space required to store this is N bits, which is prohibitive. The other solution is store D as a binary search tree. In that case the SEARCH operation can be performed in $O(\log m)$ time, and takes $O(m)$ space[2] Can we get best of both worlds?

- An idea from balls-and-bins suggests itself. Imagine the m elements of D as m balls which are each assigned to one of n "bins", or rather, numbers in $\{0, 1, \ldots, n-1\}$ *randomly*. Formally, let h be a "random function" which maps every $x \in U$ to a random integer i in $\{0, 1, \ldots, n-1\}$. Then, to store the dictionary D, one initializes an n-sized data structure $T[0 : n-1]$, and simply stores $s \in D$ at location $T[h(s)]$. If two elements s_1 and s_2 are mapped to the same location, that is $h(s_1) = h(s_2) = j$, then $T[j]$ contains both s_1 and s_2 as a (linked) list.

  What's good about this idea? Well, the space is $O(n)$, the size of the structure T. If $n \approx m$, the space is good. How about the time taken for SEARCH? Given $x \in U$, the operation will investigate $T[h(x)]$. If it is empty, the algorithm says no. Otherwise, it performs a linear scan on the list at $T[h(x)]$. How much time does this take? Since h is "random", the ***expected*** number of collisions is $\approx \frac{m}{n}$, and thus the expected time (over the randomness in h) to search is going[3] to be $\approx \frac{m}{n}$, which is $\approx O(1)$ if $n \approx m$. And thus, we get the best of both worlds.

- There are *two* issues with the above idea. One, is that we would prefer a "worst case" result: for every x we take $O(1)$ time. The array solution had that guarantee. But the much bigger problem is with the "random function" h. How are we even going to *store* this function so as to compute $h(x)$ when SEARCH(x) is queried? Since h is random, that is, it is one of the N^n functions from U to $\{0, 1, \ldots, n-1\}$, the only way to store this is to have a table which tells $h(x)$ for all $x \in U$. This takes even more space than the array solution to the dictionary problem! Both the above problems can be solved, and this is going to be covered this week.

- ***Universal Hash Family.*** The main idea to solve the hash-function-storage problem is to *decrease* the randomness in h. Formally, instead of selecting h uniformly at random from ***all*** functions, one chooses h uniformly at random from a much smaller collection of functions. Since the collection is

[1]*Lecture notes by Deeparnab Chakrabarty. Last modified : 11th April, 2021*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2]we assume the elements in D fit in "word memory" and take $O(1)$ space.

[3]We are going to prove a formal version of this soon.

small, the space required to represent a single h will be small. And yet, we do not want to lose all the randomness, and still would like the intuition of "low probability of collision" to carry over. To this end, one makes the following definition.

Definition 1 (Universal Hash Family (UHF).)**.** A collection H of hash functions from $\{0, 1, \ldots, |U|-1\}$ to $\{0, 1, \ldots, n-1\}$ if for any two ***distinct*** $x, y \in U$ we have

$$\mathbf{Pr}_{h \in H}[h(x) = h(y)] \leq \frac{1}{n} \tag{1}$$

where the probability is over $h \in H$ *uniformly* drawn at random.

Universal Hash Families exist. We will soon see how to construct them. But before doing so, let us look at a few applications of Universal Hash Families.

- ***Implementing Dictionary with Expected*** $O(1)$ ***Search Time.*** We now show that the condition (1) suffices to implement the dictionary D such that SEARCH(x) takes $O(1)$ time in expectation over the choice of $h \in H$. The data structure initialization is as follows:

```
1: procedure PREPROCESSING:
2:     Choose a universal hash family $H$ of functions from $U$ to $\{0, 1, \ldots, n-1\}$ where
   $n = \alpha m$, where $\alpha$ is a constant.
3:     Randomly sample $h \in H$ uniformly at random.
4:     Initialize a hash table $T[0 : n-1]$ initialized to empty lists.
5:     For each $s \in D$, append $s$ to the list at $T[h(s)]$.
```

The space required by the hash table is $O(n) = O(\alpha m)$ (again, assuming the dictionary items fit in memory word). The space required to store the hash function is $\sigma(n, U)$. The time required is $O(m \cdot \tau)$ where τ is the time taken to compute the hash function. We will see that if $\log U$ also fits in word memory (or $O(1)$ words), then UHFs exist with both $\sigma(n, U)$ and τ being $O(1)$. Thus, the preprocessing time and space will be $O(m)$.

The SEARCH operation is what one expects : given $x \in U$, the algorithm searches for x in the list at $T[h(x)]$. The time taken is $\tau + O(|T[h(x)]|)$. Thus, if $\tau = O(1)$, the time is dominated by $|T[h(x)]|$. The next claim bounds this, and proves that the **expected time**, with the expectation taken over the randomness in the choice of h, and this holds for ***all*** x, is $O(1 + \frac{1}{\alpha})$.

Lemma 1. For any $x \in U$, $\mathbf{Exp}_{h \in H}[|T[h(x)]|] \leq 1 + \frac{1}{\alpha}$.

Proof. This is really the balls-and-bins proof, except instead of a completely random allocation rule, we are using what the UHF gives us.

For any $y \in U$, define $X_{x,y} := 1$ if $h(x) = h(y)$ and 0 otherwise, where h is uniformly drawn from H. The first observation is that

$$|T[h(x)]| = \sum_{s \in D} X_{x,s} \quad \Rightarrow \quad \mathbf{Exp}_h[|T[h(x)]|] = \sum_{s \in D} \mathbf{Exp}_h[X_{x,s}]$$

If $s \neq x$, then by (1), we have $\mathbf{Exp}_h[X_{x,s}] \leq \frac{1}{n}$. If $x = s$, $X_{x,s} = 1$. Therefore,

$$\mathbf{Exp}_h[|T[h(x)]|] \leq 1 + \frac{(|D| - 1)}{n} \leq 1 + \frac{m}{n} = 1 + \frac{1}{\alpha} \qquad \square$$

Remark: *Although we have only talked about searching a fixed* **static** *dictionary D, one can easily modify the structure so that* INSERT *and* DELETE *operations are also supported, which can insert or delete elements into the dictionary. Do you see how?*

- ***The Distinct Elements Problem.*** We give another simple application where hashing helps. In this problem we are given an array $A[1:n]$ where each element is from the universe U. The objective is to figure out if $A[1:n]$ consists of distinct elements from U, or are there duplicates. It is known that if only comparisons are allowed between elements of U, then one needs $\Omega(n \log n)$ comparisons. Which matches the trivial algorithm of "sort-and-check."

  Using hashing, one can get a randomized algorithm running in expected $O(n)$ time, once again assuming the number N fits in $O(1)$ words. Indeed, the algorithm is simple. Choose $m = n$ and hash $A[1:n]$ into a hash table $T[0:n-1]$ using a random h from any UHF H. To be precise, we sample $h \in H$ and then for each $1 \le i \le n$, we compute $j = h(A[i])$, check if $A[i]$ appears in $T[j]$. If it does appear, then we say NOT DISTINCT and abort. If we never abort, then we say YES, distinct.

  Is the algorithm is correct? When it says NO it produces a violation, and if $A[i] = A[j]$ for $i < j$, then we will say NO when processing $A[j]$. The expected running time is the expected number of collisions as before, and thus as before for every $1 \le i \le n$, we take $O(1)$ time. Thus, the algorithm's expected running time is $O(n)$.

- ***The Carter-Wegman Universal Hash Family.*** Recall, N is the size of the universe U. First, we pick a prime number $N < p \le 2N$. Such a prime always exists by Bertrand's postulate. For a fixed $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, \ldots, p-1\}$, define the hash function

$$h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod n \qquad \text{(Carter-Wegman)}$$

Remark: *Note that if the integers N, m fit in the word size in the word-RAM model, then computing $h_{a,b}(x)$ takes $O(1)$ time, and storing the hash function amounts to storing a, b, p which also takes $O(1)$ space.*

Exercise: *One may wonder why we are taking* $\bmod p$ *and then* $\bmod n$*? Why not just define $h_{a,b}(x) = (ax + b) \bmod n$ directly, where a, b are integers between 1 and $n-1$ and 0 and $n-1$, respectively. Argue that this family is* **not** *a UHF even when n is a prime.*

Theorem 1. The family

$$H := \{h_{a,b} : a \in \{1, 2, \ldots, p-1\} \text{ and } b \in \{0, 1, \ldots, p-1\}\}$$

is a universal hash family.

Proof. Throughout, we will use $\mathbb{Z}_p$ to denote the set $\{0, 1, \ldots, p-1\}$ which is the set of possible remainders. We need to show that if we pick $a \in \mathbb{Z}_p \setminus \{0\}$ and $b \in \mathbb{Z}_p$ uniformly at random, then

$$\mathop{\mathbf{Pr}}_{a,b} \left[((ax + b) \bmod p) \bmod n = ((ay + b) \bmod p) \bmod n\right] \le \frac{1}{n}$$

Note that the event in consideration occurs if and only if

$$(ax+b) \bmod p = k \text{ and } (ay+b) \bmod p = k+qn$$

for some $k \in \mathbb{Z}_p$ and some integer (possibly negative) q such that $(k+qn)$ lies in $\mathbb{Z}_p$. Also note: $x \neq y$ implies $q \neq 0$.

Let us call a pair (k,q) ***conflicting*** if $k \in \mathbb{Z}_p$, $k+qn \in \mathbb{Z}_p$, and $q \neq 0$. We first claim that the number of conflicting pairs are not too many.

Claim 1. The number of conflicting pairs (k,q) is at most $\frac{p(p-1)}{n}$.

Proof. For any fixed k, how many q's can make (k,q) conflicting? That is, how many q's are there such that $k+qn \in \{0,1,\ldots,p-1\}$. In other words, $qn \in \{-k,-k+1,\ldots,-k+p-1\}$. This is simply asking how many multiples of n can there be in p consecutive integers. Since p is a prime, and in particular, n doesn't divide p, this number is at most $\frac{p-1}{n}$. $\square$

Call a pair (a,b) ***bad*** if there exists a conflicting pair (k,q) such that (i) $(ax+b) \bmod p = k$ and (ii) $(ay+b) \bmod p = (k+qn)$. Here is the main observation: if we *fix* a conflicting pair (k,q), there is ***exactly one*** possible pair (a,b) such that (i) and (ii) occur. Why? Because considering a and b as the variables and x,y,k,q,m as fixed, we are asking how many solutions are there to the following system of linear equations in two variables

$$ax+b \equiv_p k \quad \text{and} \quad ay+b \equiv_p (k+qn)$$

This has exactly one solution since p is a prime. These are

$$a = \left(qn \cdot (y-x)^{-1}\right) \bmod p \quad \text{and} \quad b = (k-ax) \bmod p$$

where for any $z \in \mathbb{Z}_p \setminus \{0\}$, z^{-1} is the unique number in $\mathbb{Z}_p \setminus \{0\}$ such that $z \cdot z^{-1} \equiv_p 1$. This is called the multiplicative inverse of z.

In conclusion, the number of *bad* pairs (a,b) which lead to a collision $h_{a,b}(x) = h_{a,b}(y)$ is exactly the number of *conflicting* pairs. But the number of conflicting pairs is $\leq \frac{p(p-1)}{n}$. Therefore,

$$\mathop{\mathbf{Pr}}_{h\in\mathcal{H}}[h(x)=h(y)] = \mathop{\mathbf{Pr}}_{a,b}[(a,b) \text{ is bad}] \leq \frac{\frac{p(p-1)}{n}}{p(p-1)} = \frac{1}{n} \qquad \square$$