**CS2040S: Data Structures and Algorithms**

# Discussion Group Problems for Week 8

*For: March 8–March 12*

## Problem 1. The Missing Element

Let's revisit the same old problem that we've started with since the beginning of the semester, finding missing items in the array. Given $n$ items in no particular order, but this time possibly with duplicates, find the first missing number if we were to start counting from 1, or output "all present" if all values 1 to $n$ were present in the input.

For example, given $[8, 5, 3, 3, 2, 1, 5, 4, 2, 3, 3, 2, 1, 9]$, the first missing number here is 6.

**Bonus:** (no need for hash functions): Can we do the same thing using $O(1)$ space? i.e. in-place.

## Problem 2. Coupon Chaos!

Mr. Nodle has some coupons that he wishes to spend at his favourite cafe on campus, but there are different types of coupons. In particular, there are $t$ distinct coupons types, and he can have any number of each type (including 0). He has $n$ coupons in total.

He wishes to use one coupon a day, starting from day 1. He wishes to use his coupons in ascending order and will use up all his coupons that are of a lower type first before moving on to the next type. Nodle wishes to build a calendar that will state which coupon he will be using.

- The list of coupons will be given in an array. An example of a possible input is: $[5, 20, 5, 20, 3, 20, 3, 20]$. Here, $t = 3$, and $n = 8$. The output here would be $[3, 3, 5, 5, 20, 20, 20, 20]$.
- Since the menu at the cafe that he frequents is not very diverse, there aren't many different types of coupons. So we'll say that $t$ is much smaller than $n$.

Give as efficient an algorithm as you can, to build his calendar for him.

## Problem 3. Data Structure 2.0

Let's try to improve upon the kind of data structures we've been using so far a little. Implement a data structure with the following operations:

1. Insert in $O(\log n)$ time
2. Delete in $O(\log n)$ time
3. Lookup in $O(1)$ time
4. Find successor and predecessor in $O(1)$ time

**Problem 4. Locality Sensitive Hashing (Optional)**

So far, we have seen several different uses of hash functions. You can use a hash function to implement a *symbol table abstract data type*, i.e., a data structure for inserting, deleting, and searching for key/value pairs. You can use a hash function to build a fingerprint table or a Bloom filter to maintain a set. You can also use a hash function as a "signature" to identify a large entity as in a Merkle Tree.

Today we will see yet another use: clustering similar items together. In some ways, this is completely the opposite of a hash table, which tries to put every item in a unique bucket. Here, we want to put similar things together. This is known as *Locality Sensitive Hashing.* This turns out to be very useful for a wide number of applications, since often you want to be able to easily find items that are similar to each other.

We will start by looking at 1-dimensional and 2-dimensional data points, and then (optionally, if there is time and interest) look at a neat application for a more general problem where you are trying to cluster users based on their preferences.

**Problem 4.a.** For simplicity, assume the type of data you are storing are *non-negative integers.* If two data points $x$ and $y$ have distance $\leq 1$, then we want them to be stored in the same bucket. Conversely, if $x$ and $y$ have distance $\geq 2$, then we want them to be stored in different buckets. More precisely, we want a random way to choose a hash function $h$ such that the following two properties hold for every pair of elements $x$ and $y$ in our data set:

- If $|x - y| \leq 1$, then $\Pr[h(x) = h(y)] \geq 2/3$.
- If $|x - y| \geq 2$, then $\Pr[h(x) \neq h(y)] \geq 2/3$.

How should you do this? How big should the buckets be? How should you (randomly) choose the hash function? See if you can show that the strategy has the desired property.

**Problem 4.b.** Now let's extend this to two dimensions. You can imagine that you are implementing a game that takes place on a 2-dimensional world map, and you want to be able to quickly lookup all the players that are near to each other. For example, in order to render the view of player "Bob", you might lookup "Bob" in the hash table. Once you know $h(\text{"}Bob''\text{)}$, you can find all the other players in the same "bucket" and draw them on the screen.

Extend the technique from the previous part to this 2-dimensional setting. What sort of guarantees do you want? How do you do this? (There are several possible answers here!)

**Problem 4.c.** What if we don't have points in Euclidean space, but some more complicated things to compare. Imagine that the world consists of a large number of movies $(M_1, M_2, \ldots, M_k)$. A user is defined by their favorite movies. For example, my favorite movies are $(M_{73}, M_{92}, M_{124})$. Bob's favorite movies are $(M_2, M_{92})$.

One interesting question is how do you define distance? How similar or far apart are two users? One common notion looks at what fraction of their favorite movies are the same. This is known as Jacard distance. Assume $U_1$ is the set of user 1's favorite movies, and $U_2$ is the set of user 2's favorite movies. Then we define the distance as follows:

$$d(U_1, U_2) = 1 - \frac{|U_1 \cap U_2|}{|U_1 \cup U_2|}$$

So taking the example of myself and Bob (above), the intersection of our sets is size 1, i.e., we both like movie $M_{92}$. The union of our sets is size 4. So the distance from me to Bob is $(1 - 1/4) = 3/4$. It turns out that this is a distance metric, and is quite a useful way to quantify how similar or far apart two sets are.

Now we can define a hash function on a set of preferences. The hash function is defined by a permutation $\pi$ on the set of all the movies. In fact, choose $\pi$ to be a random permutation. That is, we are ordering all the movies in a random fashion. Now we can define the hash function:

$$h_\pi(U) = \min_j(\pi[j] \in U)$$

The hash function returns the index of the first movie encountered in permutation $\pi$ that is also in the set of favourite movies $U$. For example, if $\pi$ is $\{M_{43}, M_{92}, ..., M_{124}, ..., M_{73}, ...\}$ and $U = \{M_{73}, M_{92}, M_{124}\}$, $h_\pi(U)$ will give 1 as it maps to the index of $M_{92}$ in $\pi$.

This turns out to be a pretty useful hash function: it is known as a MinHash. One useful property of a MinHash is that it maps two similar users to the same bucket. Prove the following: for any two users $U_1$ and $U_2$, if $\pi$ is chosen as a uniformly random permutation, then:

$$\Pr\left[h_\pi(U_1) = h_\pi(U_2)\right] = 1 - d(U_1, U_2)$$

The closer the are, they more likely they are in the same bucket! The further apart, the more likely they are in different buckets.

**Problem 5. Optional Reading: Learned Bloom Filters**
There is always a possibility of a false positive when non-member objects coincidentally map to set bit positions in the bloom filter array, but can we improve the bloom filter in terms of space while achieving the same false positive probability through the use of machine learning? Read more about it here: http://mybiasedcoin.blogspot.com/2018/01/some-notes-on-learned-bloom-filters.html?m=1