CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 8

For: March 9–March 13

Below is a proposed plan for tutorial for Week 8. In Week 8, there was just a midterm exam! So this week is dedicated to some fun problems! You can keep the session short, if everyone is tired. Or go through problems, if everyone is having fun. The problems today are not required, and can be skipped.

Plan:

1. Review questions from the midterm.
2. Choose any of the below problems and discuss.

Problem 1. Locality Sensitive Hashing.

So far, we have seen several different uses of hash functions. You can use a hash table 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 filer to maintain a set. You can 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 user's based on their preferences.

Problem 1.a. Assume the data you are storing is non-negative integers, and the property you want is that if two data points x and y are distance ≤ 1, then they are in the same bucket. Conversely, you want if x and y are distance ≥ 3, then they are not in the same bucket. 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 $\Pr[h(x) = h(y)] > 2/3$.
- If $|x - y| \geq 3$, then $\Pr[h(x) \neq h(y)] > 2/3$.

If their distance is between 1 and 3, then they might or might not be in the same bucket. 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 1.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 $h(\text{“}Bob'')$, 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 1.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 the their favorite movies. So, my favorite movies are $(M_{73}, M_{92}, M124)$. Bob's favorite movies are (M_2, M_{92}). Which are your favorite movies?

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 he 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 3. So the distance from me to Bob is $(1 - 1/3) = 2/3$. It turns out that this is a distance metric, and is quite a useful way to quantify how similar or far apart to sets are.

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

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

That is, if you go through the movies in order according to π, which is the first one that is in my set U? (Actually, we are taking the smallest index which leads to a movie in U.)

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 π is chosen as a uniformly random permutation, then:

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

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

Problem 2. 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: (which doesn't use hash functions): What if we wanted to do the same thing using $O(1)$ space? i.e. in-place.

Problem 3. Mr. Nodle is back with a new problem this week: He 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 up to t distinct types of coupons, and he can have any number of each type (including 0). In total he has n coupons. That is to say, all the coupons of type 1, plus all the coupons of type 2, and so on, plus all the coupons of type t, sum up to n.

He wishes to use one coupon a day, starting at day 1. Additionally, he wants to use up all the coupons of type 1 first, before using up all the coupons of type 2 and so on until all coupons are used up. The thing is that Nodle doesn't really keep his stuff in order and all the coupons he's amassed is now just sitting in a pile. He wishes to build a calendar that will state which coupon he will be using. i.g. Day 1: Use coupon type 2, Day 2: Use coupon type 2, Day 3: Use coupon type 100, and so on.

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 $n = 8$.

We're also given that 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 4. Let's try to improve a little upon the kind of data structures we've been using so far. 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 5. Mr.Govond wants you to sort a list of numbers from 1 to n. Seems simple enough right? However, there is catch! According to Mr.Govond, he wants the numbers to be ordered according to some permutation π. So for example, if there are $n = 6$ numbers, and the permutation $\pi = 3, 1, 2, 5, 4, 6$, when your sorting algorithm is done, it should output $3, 1, 2, 5, 4, 6$. How would you achieve this in as efficiently as possible?