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.

Solution: Choose buckets of size 3, and choose a random point in the range $[-3, 0]$ as the starting point for the first bucket. Each bucket then starts immediately when the previous bucket ends. Define the hash function h as putting a point in the proper bucket.
Now look at two points x and y, and assume (without loss of generality) that $y > x$. Notice that the bucket containing x will begin somewhere in the range $[x-3, x]$, and that it's start point was chosen uniformly at random. If $y - x < 1$, then y is in the same bucket as x if the bucket for x starts anywere in the range $[x-2, x]$. Since the range $[x-1, x]$ is $2/3$ the size of the range $[x-3, x]$, the probability that x and y are in the same bucket is at least $2/3$.
Alternatively, if $y-x > 2$, then y is in the same bucket as x only if the bucket for x starts somewhere in the range $[x-1, x]$. Since $[x-1, x]$ is only $1/3$ of the range $[x-3, x]$, we conclude that the probability of x and y being in the same bucket is at most $1/3$, i.e., with probability at least $2/3$, they are in different buckets.

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!)

Solution: The simplest possible idea is to divide the world up into grid squares, and require that two players be hashed to the same bucket only if they are in the same grid square. (Many old games used this type of strategy!) This is simple, but you might have two players that are very close together, but are in adjacent grid squares and so are in separate buckets.
You might try to extend the strategy from part 1. Overlay the map with grid squares, but choose a random start position for the $(0, 0)$ point of the grid. The challenge, now, is that it is a bit harder to compute the probabilities because there are two different dimensions. It is not hard to show that if $||y - x||_2 < 1$ (using the usual Euclidean distance), then the probability that x and y are in the same bucket is at least $(2/3)(2/3) = 4/9$, because in each dimension their distance is at most 1.
You can do a little better (but the math gets a little messier) by observing that if the distances are d_1 and d_2 in the two dimensions, we know that $d_1^2 + d_2^2 \leq 1$, so the probability of being in different buckets is (by a union bound) at most $d_1/3 + d_2/3$. Solving, we find the maximum at $\sqrt{2}/3 \leq 1/2$. This implies that with probability at least $1/2$, the two points will be in the same bucket. If you do the calculation more carefully, you will find that by choosing $d_1 = d_2 = \sqrt{2}/2$, you get a probability of them being in the same bucket of approximately 0.58.
A third solution is to just draw a one dimensional line at a random angle (going through $(0, 0)$). Just like in the previous part, you can divide the line into buckets. For each point, find the closest point on the line (i.e., a projection, using trigonometry) and map the point to the associated bucket. It turns out that this works pretty well too (but we won't do the math today).

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.

Solution: The users U_1 and U_2 are mapped to the same bucket if $\min_j(\pi(j) \in U_1) = \min_k(\pi(k) \in U_2)$. Imagine you are iterating through the movies in order of π. Eventually, you hit a movie that is in U_1 or U_2. If that first movie you hit is in *both* sets U_1 and U_2, then they map to the same bucket. Otherwise, if that movie is only in one of the two sets, then they map to different buckets. So we have to decide how likely that first movie is to be in both sets.
In total, there are $|U_1 \cup U_2|$ items in the two sets together. And those items are ordered in a uniformly random order by π (since we chose π at random). So we are equally likely to hit any of those $|U_1 \cup U_2|$ items first. How many of these items are in both sets? There are exactly $|U_1 \cap U_2|$ movies in both sets. So the probability that the first item found in the enumeration is one of these $|U_1 \cap U_2|$ shared movies is $|U_1 \cap U_2|/|U_1 \cup U_2|$. That is, the probability is exactly $1 - d(U_1, U_2)$.
So we have proved that the probability that U_1 and U_2 are hashed to the same bucket is $1-d(U_1, U_2)$.

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.

Solution: The idea here is to just use a hash table here to store all the values that we've seen in the array and ignore duplicates. Then after we've done that, we'll start using the hash table to check if 1 was present. If it was then was 2 present and so on. This would take $O(n)$ time in expectation given the right assumption for hash functions. Note that we could actually just use an array of size n instead to accomplish the same goal! Sometimes it's important to realise when a simple array would suffice and something as expensive as hash table can be unnecessary.
The solution to the bonus question does not use hash tables, but they do use a similar idea to cuckoo hashing. The full algorithm is quite tedious to explain, but the idea is pretty much that we will consider every item once, and given its value, we will try to place it in its own index. e.g. Based on the input given above, we would try to place item 8 at index 8, but there's an item of value 4 there, so then item 8 "kicks out" the 4 from that place. Now we try to place 4 in its index, but there's an item of value 3 there. So we repeat the same process with 3 before stopping because we realise that 3 is already sitting at index 3. Also take note that if there is a value larger than n, we can simply ignore that value since we know that it will not be part of consideration.
After we do this for all elements, we run through the array one more time and check if at index i the value i is stored. Then output the first index such that it fails.

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.

Solution: There are a few possible solutions to this problem, but it boils down to sorting an array with many duplicates. One possible way to do this is to modify a balanced binary search tree to also store the counts of each item. Then using this, insert all the items in the input, then do an in-order traversal, outputting the value at the current node for as many times as it was inserted into the balanced binary search tree. But this takes time $O(n \log t)$, since the size of the tree is at most t and we need to do n insertion operations.
Another solution that works better on average, given reasonable hashing assumptions is to hash the coupon types, and increment a counter to keep track of how much of each type we've seen. Additionally, for every new type of coupon that we hash into the table, we add it into a list. In expectation this should take at most $O(n)$ time.
Now after we have processed the input, we'll sort the list of keys that we had obtained previously, which will take $t \log t$ time.
Then we look up the counts for each key in the sorted order, and for each key we add it to the calendar as many times as it was counted based on the hash table. Since there were a total of n items, this should take at most $O(n)$ time.

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. Rank in $O(1)$ time
5. Select in $O(\log n)$
6. Find successor and predecessor in $O(1)$ time

Solution: The gist of the solution is to do two things: (1) Use both a BBST and a hash table; (2) for each node of the BBST, we also store a pointer to its predecessor and successor, let's call the pointers "previous" and "next". The hashtable here will map each key to the node that it corresponds to.
For insertions and deletions, we proceed as per usual on both data structures. But with the added work of having to maintain the two additional pointers. After we insert, we run the usual successor and predecessor algorithm, which should take $O(\log n)$ time, and then after those are found, we need to: (1) set the predecessor's "next" to the newly inserted node, (2) set the newly inserted node's "previous" to the predecessor, and the "next" to the successor" and (3) set the "previous" of the successor to the newly inserted node. Deletion works off the same idea.
For loopkups and rank, we can simply use the hashtable to find the corresponding node.
For rank, we simply run the usual algorithm on the BBST.
For successor and predecessor, we first use the hashtable to obtain the node in the BBST, and then use either the "next" or the "previous" pointer to get the corresponding node that we need.

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?

Solution: The idea here is to use a hash table again. This time the keys will be the digits that you are sorting and the value will be their rank. Then, through this we can create a comparator compares the ranks of the two keys. Using this comparator with any known sorting algorithm will give us the answer we are looking for. To generalise this any set of elements, which are to be sorted according to a certain permutation, this idea can be used again. This will run in worst case $O(n \log n)$ time since we can only use comparison-based sorting algorithms.