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.

Solution: The idea here is to use a hash set to store all the values that we've seen in the array and ignore duplicates. Then, we'll start using the hash set to check if each number was present, starting from 1. 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 set can be unnecessary.

Solution to Bonus: The solution to the bonus question does not use hash sets, but it has a similar idea to cuckoo hashing. The idea is that we will consider every item once, and given its value, we will try to place it in its own index.

For example, 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. 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 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 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.

Solution: There are a few possible solutions to this problem, but it boils down to sorting an array with many duplicates.

- One possible way is to modify a balanced binary search tree to also store the counts of each item. 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. This takes $O(n \log t)$ time, 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 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.

  After processing the input, we'll sort the list of keys that we had obtained previously, which will take $O(t \log t)$ time.

  Then in the sorted order, we look up the counts for each key and add the key 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.

  In total, this solution takes $O(n + t \log t)$ time, which is equivalent to $O(n)$ since $t << n$.

- Finally, we can also consider quicksort with 3 way partitioning! Among the other comparison based sorting algorithms, this is optimal since it is best able to handle the situation where we have many duplicates and $t << n$. After each partitioning step, if the pivot has multiple duplicates, all the duplicates would already be in the correct index and we are left with two significantly smaller subarrays to recurse on since the duplicates would not be included in the recursive calls to quicksort. The sort will use a maximum of t pivots, hence the expected runtime is given by $O(n \log t)$.

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

Solution: The gist of the solution is to do two things:

1. Use both a BBST and a hash table. The hash table here will map each key to the node that it corresponds to.

2. Modify the BBST: for each node of the BBST, we also store a pointer to its predecessor and successor (let's call the pointers "previous" and "next").

- For insertions and deletions: perform the usual insertion and deletion procedure on the BBST, and add/ remove the corresponding entry in the hash table, which will take $O(\log n)$ time.

  However, there is also 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. 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 lookup queries: use the hash table to find the corresponding node in O(1) time.

- For successor and predecessor queries: first, perform a lookup query to obtain the node in the BBST, and then use either the "next" or the "previous" pointer to get the corresponding successor/ predecessor node that we need.

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 ≤ 1, then we want them to be stored in the same bucket. Conversely, if x and y have distance ≥ 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.

Solution:
- Choose buckets of size 3, and choose a random integer in the range $[0, 2]$ as the starting point for the first bucket. Let this point be a. Each bucket then starts immediately when the previous bucket ends.

- Define the hash function h as putting a point in the proper bucket. $h(x) = floor((a + x)/3)$.

- Now look at two points x and y, and assume that $y \geq x$. Notice that the bucket containing x will begin somewhere in the range $[x - 2, x]$, and that its start point was chosen uniformly at random.

- If $y - x = 0$, then y and x will always be in the same bucket.

- If $y - x = 1$, then y is in the same bucket as x if the bucket for x starts anywhere in the range $[x - 1, x]$. Since the integers in the range $[x - 1, x]$ is 2/3 the size of that in the range $[x - 2, x]$, the probability that x and y are in the same bucket is at least 2/3.

- If $y - x = 2$, then y is in the same bucket as x only if the bucket for x starts with x. Since x is only 1/3 of the integers in the range $[x - 2, x]$, 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.

- If $y - x \geq 3$, then y and x will always be in different buckets.

Problem 4.b.   Now let's extend this to two dimensions. You can imagine that you are imple-
menting 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 $h(``Bob'')$, 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 a. Overlay the map with 3-by-3 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.

Consider the case where $||y - x||_2 \leq 1$:

- In each dimension, the distance between x and y is at most 1. From part a, we can show that
  the probability that x and y are in the same bucket is at least $(2/3)(2/3) = 4/9$.

- However, this is not the tightest lower bound. We can do slightly better. Let the distances
  in the two dimensions be d_1 and d_2 respectively. Using Pythagoras' Theorem, we know that
  $d_1^2 + d_2^2 = ||y - x||^2$. By extension, $d_1^2 + d_2^2 \leq 1$.

- Let the event whereby x and y are in different "buckets" in dimension i be D_i. Notice that as
  long as the points are in different "buckets" in at least one dimension, they will be in different
  buckets in the 2-dimensional grid. Thus, the probability that x and y are in different buckets
  is (by a union bound):

$$P(D_1 \cup D_2) = P(D_1) + P(D_2) - P(D_1 \cap D_2) \leq P(D_1) + P(D_2) = d_1/3 + d_2/3$$

- Solving, we find that the maximum probabilty that the points are in different buckets is
  $\sqrt{2}/3 \leq 1/2$, when $d_1 = d_2 = \sqrt{2}/2$. This implies that with probability of at least $1/2$, the
  two points will be in the same bucket. If you do the calculation more carefully (without the
  upper bound), 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 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 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:

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

The hash function returns the index of the first movie encountered in permutation π that is also in the set of favourite movies U. For example, if π 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 π.

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:

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

The closer the are, they more likely they are in the same bucket! The further apart, the more likely they are in different 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 $1 - d(U_1, U_2)$.

6

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