CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 8

For: March 6–March 10

Problem 1. (Priority queue)

There are situations where, given a data set containing n unique elements, we want to know the top k highest value elements. A possible solution is to store all n elements first, sort the data set in $O(n \log n)$, then report the right-most k elements. This works, but we can do better.

(a) Design a data structure that supports the following operation better than $O(n \log n)$:

- `getKLargest()`: returns the top k highest value elements in the data set.

(b) Instead of having a static data set, you could have the data streaming in. However, your data structure must still be ready to answer queries for the top k elements efficiently. Expand on your data structure to support the following two operations better than $O(n \log n)$:

- `insertNext(x)`: adds a new item x into the data set.
- `getKLargest()`: returns the current top k highest value elements in the data set.

For example, if the data set contains {1, 13, 7, 9, 8, 4} initially and we want to know the top 3 highest value elements, calling `getKLargest()` should return the values {13, 9, 8}.

Suppose we then add the number 11 into the data set by calling `insertNext(11)`. The data set now contains {1, 13, 7, 9, 8, 4, 11} and calling `getKLargest()` should return {13, 11, 9}.

Solution: For part (a), we can quick-select the k^{th} largest element in expected $O(n)$ time. The elements that are required will be found between this element and the end of the array.

For part (b), a key thing to realise is that we only need to store the largest k elements at any time. We can maintain a min priority queue of no more than k elements. For every element that is given to us, add it into the priority queue. If the priority queue contains more than k elements, keep removing the smallest element until the priority queue size is k. Both insert and remove-min operations run in $O(\log k)$ time since the priority queue contains at most k elements.

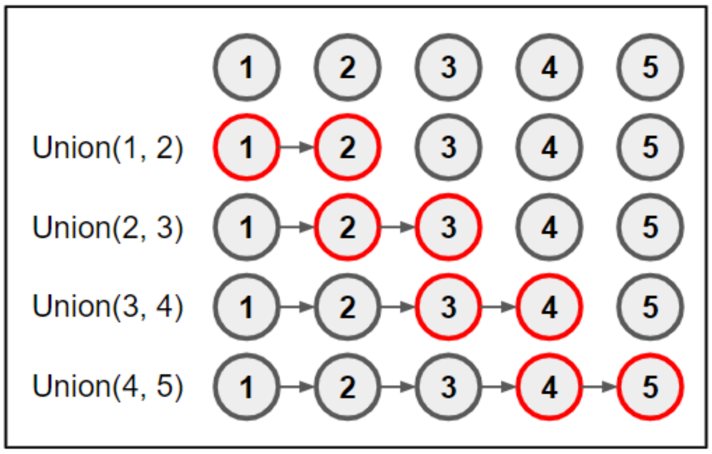
For `insertNext(x)`, we can add it into the priority queue. If the priority queue contains more than k elements, keep removing the smallest element until the priority queue size is k. Both insert and remove-min operations run in $O(\log k)$ time since the priority queue contains at most k elements.

For `getKLargest()`, we can simply return all the elements in the priority queue.

The overall complexity over n calls of `insertNext` is $O(n \log k)$. Certain languages (such as C++) use this idea to implement `partial_sort`. Priority queue must be used here (instead of waiting for all elements before we do a quick select) so that we can always have access to k largest elements throughout all stages of the stream instead of only at the end.

Problem 2. (Union-Find Review)

Problem 2.a. What is the worst-case running time of the find operation in Union-Find with path compression (but no weighted union i.e Quick-Union with past compression)?



Solution: The worst-case is $\Theta(n)$. You can construct a linear length tree quite easily by just doing Union(1,2), Union(2,3), Union(3,4), ..., Union(n-1, n) as shown above.

Problem 2.b. Here's another algorithm for Union-Find based on a linked list. Each set is represented by a linked list of objects, and each object is labelled (e.g., in a hash table) with a set identifier that identifies which set it is in. Also, keep track of the size of each set (e.g., using a hash table). Whenever two sets are merged, relabel the objects in the smaller set and merge the linked lists. What is the running time for performing m Union and Find operations, if there are initially n objects each in their own set?

More precisely, there is: (i) an array id where $id[j]$ is the set identifier for object j; (ii) an array $size$ where $size[k]$ is the size of the set with identifier k; (iii) an array $list$ where $list[k]$ is a linked list containing all the objects in set k.

```
Find(i, j):
    return (id[i] == id[j])

Union(i, j):
    if id[i] == id[j]: return;
    if size[i] < size[j] then Union(j,i)
    else: // size[i] >= size[j]
        k1 = id[i]
        k2 = id[j]
        for every item m in list[k2]:
            set id[m] = k1
```

```
append list[k2] on the end of list[k1] and set list[k2] to null
size[k1] = size[k1] + size[k2]
size[k2] = 0
```

Assume for the purpose of this problem that you can append one linked list on to another in $O(1)$ time. (How would you do that?)

Solution: Find operations obviously cost $O(1)$. For m union operations, the cost is $m \log m$. Note that since there are m union operations, the biggest set after those operations is of size $O(m)$. The only expensive part is relabelling the objects in `list[k2]`. And notice that, just like in Weighted Union, each time we union two sets, the size of the smaller set at least doubles. So each object can be relabelled at most *logm* times (as we can double the size of a set at most *logm* times). Since each object in that set was updated at most *logm* times, the total cost is $m \log m$. Of course, notice that any one operation can be expensive (each union operation have different costs). For example, the last union operation might be combining two sets of size $m/2$ and hence have cost m, while the first union operation would have a cost of $O(1)$.

The appending of one linked list to the end of another is pretty easily done in $O(1)$ through manipulation of the head and tail pointers.

Problem 2.c. Imagine we have a set of n corporations, each of which has a (string) name. In order to make a good profit, each corporation has a set of jobs it needs to do, e.g., corporation j has tasks $T^j[1 \dots m]$. (Each corporation has at most m tasks.) Each task has a priority, i.e., an integer, and tasks must be done in priority order: corporation j must complete higher priority tasks before lower priority tasks.

Since we live in a capitalist society, every so often corporations decide to merge. Whenever that happens, two corporations merge into a new (larger) corporation. Whenever that happens, their tasks merge as well.

Design a data structure that supports three operations:

- `getNextTask(name)` that returns the next task for the corporation with the specified name.
- `executeNextTask(name)` that returns the next task for the corporation with the specified name and removes it from the set of tasks that corporation does.
- `merge(name1, name2, name3)` that merges corporation with names `name1` and `name2` into a new corporation with `name3`.

Give an efficient algorithm for solving this problem.

Solution: This can be solved using the same technique as the previous one, but now instead of using linked lists, you can use a heap to implement a priority queue for each corporation. To get the next task, just use the usual heap operation of extract-max. To merge two corporations, take the smaller remaining set of tasks and add them all to the heap for the corporation with the larger set of tasks. (Use, e.g., a hash table to store a pointer to the proper heap for each corporation.) Each time corporations merge, each item in the smaller heap pays $\log nm$ to be inserted into the new heap. (There are at most nm tasks in total.) Using the same argument as before, each item only has to be copied into a new heap at most $\log n$ times, and so the total cost of operations is $O(nm(\log n)(\log nm)) = O(nm(\log^2(n) + \log(n)\log(m))$.

You might also observe that there is a better solution. For example, you can implement each of the corporations as a $(2, 3)$ tree, and there is an efficient algorithm for merging two $(2, 3)$ trees of size n in time $O(\log n)$. This, however, requires first designing mergeable $(2, 3)$-trees. Alternatively, you may use heap structures which allow for cheap merging (such as a Binomial Heap or Fibonacci Heap).

Problem 3. **Optional. No need to prepare!** TAs will pick some puzzles and solve as a class!

Problem 3.a. [*Warm-up*] **Heaven or Hell**

You have two doors in front of you. One door leads to heaven, and the other to hell. There are two male guards, one by each door. One guard always tells the truth, and the other always lies, but you don't know who is who.

You can only ask **one** question to **one** guard to find the door to heaven. What question would you ask?

Solution: *"If I ask the other guard about which door leads to heaven, what would his answer be?"*
Irrespective of whom do you ask this question, you will always get an answer which leads to hell. So pick the alternative.

Problem 3.b. *What..?* **The Housewife and the Bartender**
Source: AlphaLab

A housewife finally has time for a break. She goes to a bar and challenges a bartender to guess her kids' ages:

"I have three children and the product of their ages is 72."

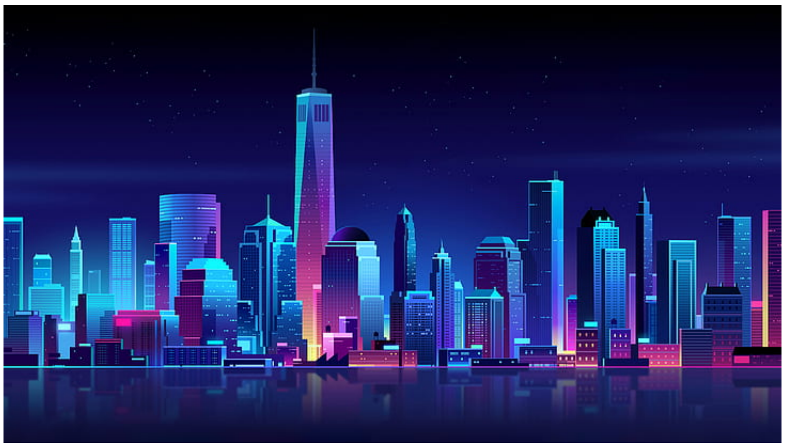
The bartender claims he doesn't know.

"The sum of their ages is the number of this building"

The bartender does some calculation but claims he still doesn't know.

"Well, guess i'm getting a free drink. It's nice to relax. I've been playing chess with my youngest every night, she's improving too quickly."

Now he knows! So what are the ages?



Solution: Of all the possible factorisations of 72 comprising of 3 factors, only {2, 6, 6} and {3, 3, 8} both have the same product and sum. Hence, the bartender couldn't have known the individual ages until he knows a youngest child exist.

Note that each response by the bartender conveys some form of information. The first response is slightly 'trivial' (if 72 has a unique factorisation, the bartender would've known the breakdown) but the same concept is applied in the subsequent steps which helps frame the solution.

Problem 3.c. *'Shared' Information* - **100 green-eyed Captives**
Source: TED-Ed

100 green-eyed logicians are held captive by a (un)reasonable dictator, Tdolf Ailer, on a secluded island. There is no escape but there is in place, one peculiar rule: *Any prisoner can approach the guard at night and ask to leave. If the prisoners have green eyes, they'll be released. Otherwise, they are tossed into the volcano.*

As it happens, all 100 prisoners have green eyes! But they've lived there since birth. And Tdolf has ensured they can't learn their own eye colour. There are no reflective surfaces (even water is in opaque containers), and most importantly, they cannot communicate among themselves. However, they do see each other during each morning headcount. Nevertheless, they all know no one would ever risk trying to leave without absolute certainty of success.

After much pressure from human rights groups. Tdolf reluctantly agrees to let you visit the island and speak to the prisoners under the following conditions:

- You may only make **1 statement**
- You cannot tell them any new information

You thought long and hard on what to say to avoid incurring the wrath of the dictator. Finally, you tell the crowd: *"At least one of you has green eyes!"*

The dictator is suspicious but reassures himself that your statement has no consequential impact. You leave, and life seemingly goes on as before. But on the 100th morning after your visit, all 100 prisoners are gone, each having asked to leave the previous night.

So how did you outsmart the dictator?
Hint: *Start with a much smaller group! What's the minimum group size you can consider?*

Solution: https://youtu.be/98TQv5IAtY8?t=120
This is a bit difficult to think about on such a large scale, so let's start by looking at some smaller cases to gain some intuition. Consider just 2 captives. Each sees one person with green eyes. If both stay put for the first night, each of them acquire new information - the other person is seeing someone with green eyes (recall that they now know at least one of them has green eyes).

It's slightly trickier with 3 captives, Andre, Jonas, and Seth. Andre knows Jonas and Seth have green eyes. Andre waits out for the 2nd morning and sees everyone present, which makes sense since Jonas and Seth each sees at least one other person with green eyes. Andre waits out for the 3rd morning and still, he sees everyone present. If instead he had blue (or any other coloured) eyes, Jonas and Seth would have left. The fact they stayed behind suggests they are seeing 2 person with green eyes. So Andre happily leaves on the 3rd night. So did the other 2.

We can then inductively apply this reasoning for 100 people which takes 100 nights.

Problem 3.d. *An optimization problem* - **Cross The Bridge**
Source: TED-Ed

Quick! Your team needs to get away from a bunch of zombies ***fast***. With you, there are the young biologist, the middle-aged physicist, and the old mathematician.

There is only one way to safety - across an old rope bridge, spanning a massive gorge. Here's how fast each of your team can dash across:

- You: 1 minute
- Biologist: 2 minutes
- Physicist: 5 minutes
- Mathematician: 10 minutes

The mathematician calculated that you have just **17 minutes** before the zombies catch up. So, you only have that much time to get everyone across and cut the ropes. Sadly, the bridge can only hold **two people** at a time and they need to **walk at the pace of the slower one**. Worse, the place is pitch black! And you only have **1 lantern** that illuminates a tiny area. You have no other tools, and must make sure everyone is safely across before the first zombie steps onto the bridge.

Can you figure out a way to have everyone successfully escape?

Solution: https://youtu.be/7yDmGnA8Hw0?t=126
The key idea is to minimise the time wasted by the two slowest people by having them cross together. And because you have to make a couple return trips back and forth, you will want to have the fastest people available to do so. Here's the breakdown:

1. You and the biologist run across in **2 minutes**
2. You run back in **1 minute**
3. The mathematician and the physicist walk across in **10 minutes**
4. The biologist runs back in **2 minutes**
5. You and the biologist run accross in **2 minutes**