Homework 6: Data Structures

Due: October 31st, 2024

Problem 1. You're an announcer at the hottest rock-paper-scissors competition in North America. The biggest match of the evening is coming up, Rocky Rick vs. Paper Pete, and you need make sure the audience stays engaged during the event. For past announcers, the biggest obstacle is keeping the audience engaged during the halftime show (the players need time to rest their hands).

In order to step things up, you're planning to do something that has never been done before and want to be able to say that this is the nth time that the score between Rocky Rick and and Paper Pete has been i to j at halftime. The main challenge that has prevented previous announcers from doing this is that you won't know what i and j are, as the game has not yet happened. You also cannot afford to scan through the entire list of previous halftime scores between Rick and Pete and count the number of i vs j appearances as the audience would surely revolt.

1. Devise an efficient method of processing the list of previous Rick-Pete halftime scores before their match begins, so that you can quickly say, right at the start of half-time, how many times the pair (i, j) has occurred at similar moments in the past. Your pre-match processing should take time proportional to the number of previous games and the querying task should take constant time.
2. Justify the runtime and correctness of your scheme.

Solution. 1. We can maintain the frequency of each half time score (i, j) into a hash map implementing chaining, using some hash function that depends on values i and j. Assuming we have a hash function that disperses the data in the table with sufficient evenness, operations **insert** and **get** would have an expected time of $O(1)$.

   i. When inserting a new score into the hash map, we can set its value to 1. When inserting a score that has already appeared into the hash map, we can increment its value.
   ii. When querying a score that is not in the hash map, we can return null or throw an error, indicating that this a score that has not been witnessed in previous games.

   Note that this scheme is *expected* time $O(1)$ because in a hash table, we have a worst case scenario where all numbers are mapped to a single slot in the map, in which case all elements would be in one chain. The run time of our algorithm would then no longer be constant.

2. i. By inserting the half time score of each previous game into the hash table, we are executing a constant operation with every previous game, meaning the processing phase takes time proportional to the number of previous games.
   ii. Because we assume we have an adequate hash function such that **get** is constant, querying should be a constant time operation.
   iii. The correctness of the algorithm follows from the hash map holding exactly the frequency of the key (i, j), since we increment each time we encounter it.

□

Problem 2. There are n marbles rolling along a straight one-lane track. They each are at some distance (in inches) away from the start of the track (inch 0) and are all traveling to the end of the track which is at inch *end*. You are given two arrays: *position* and *speed*. Both are of length n so *position*$[i]$ and *speed*$[i]$ denote the starting position and speed of the ith marble, respectively. The positions are given in inches and speeds are given in inches per second.

Since the track is narrow, marbles cannot roll pass other marbles: this means that when a faster marble catches up to a slower marble, they will travel together at the speed of the slower marble. We can then define a *group* of marbles to be a set of marbles (of size at least 1) that reach the end of the track at the same time. You are tasked with figuring out the number of groups of marbles that arrive at the end of the track.

1. Devise an efficient method to find the number of groups of marbles we expect to see at the end of track given their starting positions, speeds, and *end*.
2. Justify the runtime and correctness of your scheme.

Solution. 1. Consider the following algorithm:

Algorithm 1 Number of marble groups

```
Input: position, speed, end
Output: The number of marble groups at the end
function countMarbleGroups(position, speed, end)
    groupLeaders ← stack
    posSpeedPairs ← []                          ▷ Stores the pairs (position, speed)
    for i in range(0, n) do
        posSpeedPairs.append((position[i], speed[i]))
    sort posSpeedPairs by the first key, in decreasing order
    for i in range(0, n) do
        if len(groupLeaders) == 0 then
            groupLeaders.push(posSpeedPairs[i])
            continue
        currentLeader ← groupLeaders.top()
        leaderEndTime ← (end - currentLeader[0]) / currentLeader[1]
        marbleEndTime ← (end - posSpeedPairs[i][0]) / posSpeedPairs[i][1]
        if leaderEndTime < marbleEndTime then
            groupLeaders.push(posSpeedPairs[i]
    return len(groupLeaders)
```

2. The idea is to maintain a stack of the leading marbles for each group. We process the marbles from closest to end to closest to the start, and every time check if the current marble will reach the end sooner / later than the group leaders. If it reaches later than all the current groups, it is a new leader so we push to the stack.

   To prove this is the case, a marble catches up and joins a group ahead of it if and only if it is faster than the group. In this case, since we have a finite length track, this means that it

would reach the end sooner than the group ahead of it assuming no obstructions. If it does not catch up, then it itself must become a new group leader. Therefore, new group leaders arise by processing the marbles by largest position to smallest.

The top of the stack will always have the closest marble group to the current marble: indeed, we only ever add in a leader if it never reaches the marbles already processed in front of it, meaning it will always lag behind. As a result, the top of the stack is the closest group to the current marble. Comparing end times is a matter of using $d = r \cdot t$ (d is distance, r is rate, t is time), which is what happens in lines 12 to 14. Again, if the current marble reaches the end after the closest group, it itself must be a new leader since it will never reach the closest group. This shows our algorithm works as intended.

The runtime of this algorithm is $O(n \log n)$: this comes from the sorting at the start. The for loops all have constant time operations (e.g. append, push, top), so they all run in $O(n)$, giving a runtime of $O(n \log n) + O(n) = O(n \log n)$.

□