Viagogo Coding Challenge - Grid Developer Test

Zihan Ni

Build & Run

In the submission there will be a bin/ and a src/ directory which contains the binary .class files and the .java sources files, respectively. The files are derived from an Eclipse project so you can directly open the folder using Eclipse and build & run it there. To make running simpler, I have compiled the program and all dependencies into a .jar file which could be run directly. Just type in command: **java -jar solution.jar** in terminal and the program should run.

Assumptions

1. For ticket prices, we assume ticket prices would range from **$1.00 to $1000.00** with 2 decimal places accuracy.
2. For Events, we assume that each Event will contain **0-20** tickets, inclusively.
3. For the Grid which represents the 2D world, we assume there will be **10-100** events after initiation.
4. For closest events, if an Event has **zero ticket** available we will just **skip it**.
5. Our Coordinates will be in **[-10, 10]** range and only operate in integer scale. If user input is **out of this range** we will ask for input again. We do not have to do this since even if the Coordinate is out of this range we can still find the closest points in Manhattan Distance. However, it makes more sense for input Coordinate to be **within our world**.

Questions

How might you change your program if you needed to support multiple events at the same location?

Answer: If multiple Events could appear at the same location, we will change our grid to be a HashMap where the key is still Coordinate but the value will be a Set of Events. In this way, one Coordinate will be matched to a set of Events, where each Event contains zero or more tickets. Since we do not need to sort and rank the Events, a Set will do the job for us.

How would you change your program if you were working with a much larger world size?

Answer:
Even though the question asks for a solution in a small world. I designed my program with scalability in mind and implement it with time complexity consideration. Here are some things that contributes to scalability:

1. The program is object oriented and the structure is well defined: Ticket, Coordinate, Event, and Grid.
2. We are using ArrayList to keep tickets in Events. ArrayList is dynamic array and is able to scale freely.
3. In Grid, I use a HashMap to keep Coordinate and Event instead of using a 2D array which will waste a large amount of storage for empty Coordinates. In our Grid, given a certain Coordinate, insertion and retrieval will be constant time!
4. When we get our closest events, we keep a min-Heap to store our element to reap the benefit of the sorting property of Heap. We only need to add Coordinates to our Heap once, and pop the top Coordinates to get our 5 closest events, which takes constant time. This is scalable when we do not want to include Events that do not have any tickets available, since we can pop however many closest Events we want and take the top 5 with tickets available.

Future Improvements

Currently, we are adding all Coordinates to our Heap and then take the top elements. This is redundant given a very large size of Coordinates. In the future, we can improve our getClosestEvents method to be faster by only keeping a size k Heap and keep throwing the furthest element out when we add new Coordinates to our Heap to get the closest k Coordinates.

Moreover, we can use multithreading to process a very large world of events. However, we have to be very careful of using locks to control the ticket prices and events in order to maintain consistency across threads.