Code Jam 2013 - Round 2

Analysis: Ticket Swapping

The small dataset

Note that in this problem we treat the passengers as one "player" in a game, and assume they all cooperate to pay as little as possible in total to the city. This means it doesn't matter who actually pays for a given entry card. In particular, when the train leaves a station, the charge on each entry card in the train increases (by **N - i**, where **i** is the number of stations this card traveled so far). Since all passengers want to exit the subway eventually, all entry cards will have to be paid — so we might just as well immediately subtract this cash from the passenger "total" and move on.

As long as nobody exists the train, there's no need to exchange entry cards. Only once someone needs to exits, the passengers (along with the ones who just entered) need to gather and figure out which entry cards do the passengers who are just leaving take with them. They should choose the entry cards that have been on the train for the shortest amount of time so far, since on each subsequent stop the price for such an entry card will be larger than the price for any card that has been on the train longer (as the price is **N - i**, where **i** is the number of stations the card traveled so far). This means that at every station, the passengers should pool all the cards together, and then whoever wants to exit takes the entry cards with the smallest distance.

For the small dataset, a naive implementation of this algorithm will work. We can process station by station, holding a priority queue (or even just any collection) of the entry cards present. When anyone wants to exit, we iterate over the collection to find the card that has been on the train for the shortest amount, add its cost to the total passengers cost and remove it from the set. We also need to figure out how much the passengers *should* pay, but fortunately that's easy.

The large dataset

The large data set needs a bit more subtlety. With the insane amounts of passengers and stations we will need to avoid processing unnecessary events. First of all, we should process only the stations at which someone wants to enter or exit. This will make us process only O(**M**) stations, much less than the **N** we would process otherwise. Moreover, we should avoid processing passengers one by one.

To this end, notice that the order in which we want to give exit cards to passengers is actually a LIFO stack — whichever card came in last will go out first. So, we can keep the information about entry cards present in the train in a stack. Whenever a new group of passengers comes in, we take their entry cards and put them onto the stack (as one entry, storing the number of cards and the entry station). Whenever any group wants to leave, we go through the stack. If the topmost group of cards is big enough, we simply decrease its size, pay for what we took, and continue. If not, we take the whole group of cards, pay for it, decrease the amount of cards we need by the size of the group, and proceed through the stack.

This algorithm will take only O(**M**) time in total to process all the passengers — we will put on the stack at most **M** times, so we will take a whole group from the stack at most **M** times, and each group of passengers will decrease a group size (not take the whole group of cards) at most once, so in total — at most **M** such operations in the whole algorithm. Additionally, we need to sort all the events (a group of passengers entering or leaving) up front, so we are able to solve the whole problem in O(**M** log**M**).

Finally, when implementing, one needs to be careful. Due to the large amounts of stations and passengers involved, we have to use modular arithmetic carefully, because — as always with modular arithmetic — we risk overflow. In particular, whenever we multiply three numbers (which we do when calculating how much to pay for a group of tickets), we need to take care to apply the modulo after multiplying the first two.