Q 2.1)

Begin by sorting an array in descending order in time. Then initializing a customers’ minimum prices and push all elements from into it in order (so that the top of the stack is the least minimum price) in time. (Storing original index positions during sorting and mapping it back at the end).

For each element in an array , checking if current element is less than the value on the top of . If it is true then iterating to the next element without popping from (which means the customer couldn’t buy the jewelry because its price is not at least his minimum price). Otherwise, allocating that jewelry (using its index) to the customer from the top of (using his original index that is mapped) and pop him out before moving on to the next element (similar to allocating the lowest price item the customer could buy to him). Continue until is empty (no more customers) or have iterated through all elements (making a jewelry could be allocated only once and a customer could only buy 1 jewelry).

We allocate each customer the jewelry with the lowest price possible that hasn’t been already assigned. Then the number of allocated jewelries would be the maximum number of jewelries that could be successfully allocated.

Let an allocation of a jewelry to a customer be given by a pair , where is the customer, and is the jewelry. let be the minimum price of the customer, and let be the price of the jewelry.

**Claim.** The greedy strategy allocates customer the jewelry , but the alternative strategy allocates the same item to the customer .

Since the greedy strategy allocated customer the jewelry , then . Also, since the greedy strategy allocates each customer the jewelry with the least price but higher than his minimum price (that hasn’t already been allocated), we necessarily have , otherwise, the greedy strategy would not have allocated to . If the alternative strategy allocated to a different customer, say , this customer must also have a minimum price lower than the price of (since ). Hence, we can modify the allocation made by the alternative strategy to adhere to the greedy policy by swapping the allocations of the two customers.

**Claim.** The greedy strategy allocates customer the jewelry , but the alternative strategy doesn’t allocate any jewelry.

In this case, the alternative strategy must have allocated to some customer, otherwise the allocation would not be optimal, as we would be able to add the allocation . Hence, suppose that the alternative strategy allocated to a customer . Since the greedy strategy considers customers in increasing order of minimum price, we necessarily have . (If then the greedy strategy would have allocated a jewelry first) But since the greedy strategy allocated to , . Hence, we could modify the allocation made by the alternative strategy to adhere to the greedy policy by allocating to rather than allocating it to .

Hence, we have shown that for each possible violation, a modification could be made to the allocation to make it adhere to the greedy policy. Hence, if an allocation contains multiple violations, we could apply these modifications one by one, transforming the allocation into one that would be produced by the greedy strategy. Thus, any optimal allocation could be transformed into an allocation that adheres to the greedy policy, and therefore the greedy strategy is optimal.

Overall time complexity is; for sorting an array then initializing and pushing items into the stack, for iterating through length array which checking condition, peeking and popping all take , totaling .

Q 2.2)

By initializing a counter for each jewelry in an array in time, counting how many customers are capable to buy each of them, all starting with 0. A counter counting for the index of the first element we would like to start with when iterating through an array (to know which customer we are trying to allocate an item at a time) starting with 0. And a counter for each customer in an array in time, counting how many jewelries each of them could buy, starting with 0.

First of all, for each customer (array and ), iterating through all elements in . Checking each jewelry if current customer is able to buy it (), incrementing of the customer by 1 if he could afford it. This takes time.

After that, sorting customers in both array and the same way in a descending order by their in time (giving customers with less opportunities a chance to pick a jewelry first since it is more likely that they might end up having nothing left to pick if they were to pick last). (Storing original index positions during sorting and mapping it back at the end).

Then, for each customer (array and ), starting with customer, iterating through and check for every jewelry he could buy () then increasing those jewelries’ by 1. If the current customer is not customer then we wouldn’t increase the that is 0 even though the customer is able to afford it (because we are trying to allocate a jewelry to customer at the time and trying to see which jewelries customer is competing with the other in order to buy them). This takes time.

Next, we then allocate the first jewelry (using its index) with least to customer using his original index. Reset each jewelry’s back to 0 in time, incrementing by 1 then repeat everything again starting with customer until . This would allocate the least competitive item that a customer could afford to him.

Delete allocated jewelry?

By initializing and in time. Iterating through length array but through elements each of the time when is increased by 1 every loop from 0 to and for each element, we iterate through length array twice (checking item’s condition and changing counter take constant time) which costs . That is,

Which . So, overall time complexity is in total.

Q 2.3)

Begin by having a counter for each day, counting how many customers are available in each day in time (iterating through all elements in an array and sum the values up for days). And having a counter for each day, counting how many customers have ordered on that day to make sure that any day with order excess 5 wouldn’t be assigned (initializing in time for days). And other counters just like Q2.2’s algorithm.

Then using the same algorithm as in Q2.2. However, each time after knowing which jewelry to allocate to a customer, assigning the last least day which that the customer is free to the customer (by iterating through elements in of that customer and keep tracking the day that meets the condition in time) then incrementing of the assigned day by 1. If there are multiple same least and days then assign the first one with least . If no days could be assigned to the customer, then don’t allocate the jewelry. Before moving on to next customer, updating of each day by decreasing the days current customer is available by 1 (since the customer would already get an item if possible and wouldn’t compete with the other to find a day with some orders left anymore) in another time (iterating days in of customer).

Update busy when move on

Doing it this way minimize the number of customers who walk away with nothing because of the same reason as Q2.2 plus customers would be assigned by the day that has least competition (helping to have least days with excess orders) so no one is preventing other from having no available days unless they are sharing only same 1 opportunity but 1 of them would walk away with it affecting nothing to the number of people walking away.

By initializing and in time. Iterating through array and times but through elements each of the time when is increased by 1 every loop from 0 to and for each element, we iterate through length array twice (checking item’s condition and changing counter take constant time) then iterating through elements in twice which costs since . That is,

Which . So, overall time complexity is since in total.