Problem Set 2

Name: Asem Ashraf

Collaborators: None

Problem 2-1.

(a)

(b)

(c)

(d)

Problem 2-2. Full mark.

(a) Since an in-place algorithm is required. Merge sort is not a viable option because it uses a linear number of extra temporary space. O(n) extra temporary space. Insertion sort is an inplace sorting algorithm that compared every two consecutive elements and swaps them if they are out of order. Which make insertion sort do an insertion every comparison. Insertion sort has complexity $O(n^2)$, which makes it does $O(n^2)$ insertions in the life span of its run-time. O(n^3 log n).
While selection sort is another in-place algorithm that does the same number of comparisons but only does an insertion after O(n) comparisons. And since get_ at(i) is O(1), while set_ at(i,x) is O(n log n). Selection sort will have O(n^2 log n) run-time.

(b) Since we are dealing with a static array. Access and editing time is constant. Which leaves the choice to the best sorting algorithms which is Merge sort. O(n log n).

(c) Since the swaps where made to adjacent elements, insertion sort will be able to sort this array back in O($loglogn$) time. Because insertion sort works on sorting an array by swapping each two adjacent element out of order. So, if the array was sorted, then only swapping of adjacent element were made to unsort it, insertion sort will sort it back in O($loglogn$) time by undoing every swap in reverse order till the original sorted array is reached.

Problem 2-3. Full mark.

We start by identifying which half Datum is in. Then we go to the end of that half. We know that Datum is between our position and the middle of the island. We start visiting the kilometers that are powers of 2 away from our end. i.e., 1, 2, 4, 8, 16, ..., q. We keep going in this path till either we have passed Datum, which will happen before we have passed $n/2$ by two steps. By then we would have a portion on the island that depend on k and definitely has datum in it. Then we run the normal binary search algorithm. this algorithm would have run time O(log$k$) because we the distance specified was defined by how far Datum is from an end.

Problem 2-4. Full mark.

We will need 3 data structure for this database.

Firstly, we will need One doubly linked list for all the messages sent to chat in order. Secondly, a linked list for each viewer that contains at its head the ban status of the viewer and each element after that contains a pointer to a message sent by the viewer in the doubly linked list. And finally, an array that contains all the viewers in sorted order along with a pointer to the head and tail of the linked list containing the ban status and messages of the user.

build(V): To initialize a new chat room we will fill in the viewers in the array in O(n) time, sort them in O($n$log$n$) time, and create a linked list for their message history in O(n) time.

send(v,m): To send a new message m to chat by viewer v, we will first need to check if that user is banned by searching for him in the array in O(log$n$) and checking their ban status. If the viewer is banned, the message wont be recorded. Otherwise, we will append to the end of the doubly linked list the message along with the pointer to the head of the messages history of that viewer in O(1), then we will append to the linked list of that viewer the pointer of that message in O(1) because the tail of the linked list is saved along the name of the viewer in the array and we have already searched for them before.

ban(v): If a user gets banned, we will search for him in the array O(log$n$), then delete all his messages from the doubly linked list one by one then clear his linked list and change their ban status to yes O(n_v). This will make banning a viewer take O(n_v + log$n$).

recent(k): If we want to access the latest k messages in the chat, we will go to the tail of the doubly linked list in O(1) and recover k messages in O(k) time.

Problem 2-5.

(a) Firstly, we store the booking schedules in memory in the form of an array having size that is big as the latest hour in the schedule. Each index marks the start of an hour. i.e., the index 0 marks the start of the first hour, index 1 marks the end of the first hour and the start of the next hour. Each element in that array contains the number of rooms to be booked during that hour.

If an array was created for each booking schedules B_1 and B_2 in the said format, each having size n_1 and n_2, where $n_1 n_2$, simply summing the contents of the same index into a new array (with size n_2) will yield the new booking schedule $B = |B_1| + |B_2|$.

To reformat B back. We group all the consecutive element with the same number of rooms together as a single booking. i.e., from index 2 to index 13, all the values are 2, then they are grouped together into single booking (2,2,13).

(b) First we find the range of hours that the request has, then we make a frequency array of the booking schedule hours like in the above question. Convert the frequency array to the booking schedule format.

(c)

```
def satisfying_booking(R):
    '''
    Input: R | Tuple of |R| talk request tuples (s, t)
    Output: B | Tuple of room booking triples (k, s, t)
            | that is the booking schedule that satisfies R
    '''
    B = []
    max=0
    for i in range(len(R)): # To find the total number of hours
        if R[i][-1]>max:
            max=R[i][-1]

    hours=[0]*max     # A frequency array like data structure to
        record number of different room in an hour initiated to zero

    for i in range(len(R)): # Filling the frequency array from the
        room requests
        for n in range(R[i][0],R[i][-1]):
            hours[n]+=1

    index,inner=0,0   # Two pointers used to make the known format of
         a booking schedule
                    # Defined in terms of indices
    while(1):
        if inner==len(hours): # The termination condition of the loop,
             terminates when the last index of hours is reached
            B.append((hours[index], index, inner)) # adds the booking
               schedule in the known format
            break
        if hours[index]==0:  # If the number of rooms in an hour is 0,
             skip this schedule
            index+=1
            inner+=1
            continue
        if hours[index]==hours[inner]:
            inner+=1
        else:
            B.append((hours[index],index,inner))
            index=inner
    return tuple(B)
```

Ran 5 tests in 0.004s. 5/5 OK. FROM THE FIRST TIME. HELL YEAH