Q2.) Chapter 4, Exercise 6 (Competition Scheduling)

Ans.)

1.) Number of participants = p
For the algorithm, let there be two random participants, i and j
s_i -> swimming time for participant i
b_i -> Biking time for participant i
r_i -> riding time for participant i

We will keep track of the participant's observed completion time for all three events by assigning a pseudo-random number to denote their completion times. Also, to keep a track of the overall projected time for the event, we will initialize a "time" variable, which will be used to calculate the shortest completion time.

2.) The overall idea of the algorithm is to design an approach that will finish the event in the shortest possible time. For this problem, we will schedule the events in the decreasing order of biking + riding time and then claim that this minimizes the overall event time.

3.) **Pseudocode** –

```
Def smallest_completion_time:
        Time = 0  # Overall time tracker
        Participant = 0 # Number of participants

# Projected times for all the events
        completion_time = [0] * participant
         biking = [0] * participant
         swimming = [0] * participant
           riding = [0] * participant

        # Fill the respective event matrices by running a For loop

                for p in range(participant):
            swimming[p] = random.randint(random1, random2)
            biking[p] = random.randint(random1, random2)
            riding[p] = random.randint(random1, random2)

        If b_i + r_i < b_j + r_j :
          Swap the timings

        Check to see if our swapped schedule has a higher completion time
                If the length of completion time against the number of participants
                Update the completion time of all participants
                Else:
```

Let the projected time stand

Check the optimal time of all participants:

If biking + riding <= time – completion_time # This checks to see the minimal completion time of all participants post the swap

Else:

Time = time + (biking + riding) – (time -completion_time) # If the above condition is not met, then add the biking + riding time to a participant's total time and subtract the projected time

4.) Time complexity of the above algorithm will be $O(n^2)$

5.) The entire algorithm is based on the premise that we need to minimize the finishing time of the overall schedule. We are ordering the participants by their total biking + riding time to do this.

To prove that it works, we will use the Exchange Argument. Let there be an optimal solution that does not use this logic. The solution will again have two participants – 'i' and 'j.'

Participant 'j' is sent out directly after 'i'. However, we have assumed that $b_i + r_i < b_j + r_j$. Such a pairing is an "inversion" to our schedule.

Now considering the solution obtained by swapping the schedules of 'i' and 'j' -> 'j' completes the event before 'i'. This would mean that 'i' got out of the pool when previously 'j' was the one getting out of the pool. But, because $b_i + r_i < b_j + r_j$, 'i' must finish sooner in the swapped schedule than 'j' in the previous schedule. Thus, our swapped schedule does not have a greater completion time and is optimal.

***NOTE**: The last two For loops in the program application check for the conditions mentioned in the above paragraph.*

In the end, we will schedule basis our algorithm whose completion time is no greater than that of the original schedule. Hence, our algorithm is optimal.

Q1.) Chapter 4, Exercise 4; Data Mining

Ans.)

1.) Let S be the original unfiltered and unrefined set containing all the stocks that we are targeting

Let S* be the subsequence set that contains select elements from S. By default, we'll assume that S* is the shorter sequence and a sub-sequence of S i.e. n(S*) < n(S).

Let S = [s_1, s_2, ….., s_n] and S* = [s^*_1, s^*_2 , …., s^*_n]. We will design a greedy algorithm with a two-pointer system to match the elements that are common between the sets. Now, the question stem lays down a constraint wherein S* can be considered a sub-sequence only if elements from S when deleted are equal to the current ordering and elemental match with S* at that point.

Let m1, m2…, etc. be the matches found between the two elements. We will use 'i' to traverse and denote the current position in S. Similarly, 'j' will be used to traverse S* and indicate the current position in S*.

2.) We want to see if there is an underlying pattern in a sequence of events. However, we have to find the subsequence in a way that when other elements are deleted from the original sequence, they should match the randomly generated subsequence in both order and elements.

3.) **Pseudocode** –

```
Set i = j = 0

# n = length of S; k = length of  S*

For(i <= n  and j<= k; i++) {
        If s_i == s*_j then
            m_j = i
            j++
        else
            i = (i + 1) - 1  # Stop iteration at the unmatched index for further iterations

If j = k + 1

            Print("Subsequence Found")
            break
 Else
        Print("S* is not a subsequence of S")
        break

}
```

4.) **Walkthrough:**

Let S = [buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Apple]

S* = [buy Yahoo, buy eBay, buy Yahoo, buy Apple]

As per the pseudo-code ->

We check the first element of S and if it matches then we assign it to m_j and move to the next element in S*. If no match is found, then we reset the pointer of S to it's first element while keeping the pointer of S* constant at wherever the match has occurred.

In our example,

s_0 = buy Amazon
s^*_0 = buy Yahoo

Here, s_0 <> s^*_0 -> Else loop triggered -> i = (0+1)-1; j = 0 -> i gets incremented by 1 in the next call

s_1 = buy Yahoo
s^*_0 = buy Yahoo

Here, s_1 <> s^*_0 and the pointer for S* is also moved alongside the pointer for S.

This continues till all the matches are found.

5.) **Time Complexity**: The time complexity of the above algorithm will be O(n + k) = O(n), in this case as we traverse both the sets only once.

6.) **Proof**: We can prove its correctness by induction.

Step 1 -> Let j = 1 in the subsequence set

Let there be another subsequence, S** which has a set of matches 'x' with the original set
Now, as per the algorithm, 'm_1' will be the first match with s^{**}_1. So, $m_1 = x_1$ -(1)

Step 2 -> Assume that (1) holds true for all j = y –(2)

Step 3 -> Let j = y + 1

From Lemma 4.10, we know that a Greedy algorithm has the optimal max lateness, hence, we can safely say that a greedy algorithm finds the first matching index at the earliest instance.