CSCI570 Homework 3

Name: Saunak Sahoo **USC ID: 3896400217**

Ans 1

In order to minimize the cost of joining all the rods, the rods with the smallest lengths should be joined first to minimize their repeated costs in the final cost of joining all the rods. Hence, we need to arrange all the rods in ascending order which can be done with the help of a min heap

Algorithm:

```
1. Create a minHeap from all the rods based on length
2. Initialize the variable cost = 0
3. Perform the following while minHeap is not empty:
    a. Assign rod1 = minHeap.root
    b. Run deleteMin()
    c. Assign rod2 = minHeap.root
    d. Run deleteMin()
    e. Assign totalRodLength = rod1 + rod2
    f. Update cost = cost + totalRodLength
    g. minHeap.insert (totalRodLength)
4. Return cost
```

Ans 2

a)

Let us consider the following counterexample:

No of artists N = 5, No of performances n = 5 and no of days in the festival m = 3

Say the artists' deadlines are D = [1, 2, 3, 4, 5]

And their audience turnouts for those days are A = [1, 2, 3, 4, 5]

Using a greedy algorithm to allocate performances according to increasing order of deadlines over 3 days, we shall select artists A1, A2, A3. The audience turnout = 1 + 2 + 3 = 6

However, we can see that to maximise audience turnout we should select artists A3, A4, A5 to have audience turnout = 3 + 4 + 5 = 12 which is greater than the above audience turnout

b)

Let us consider the following counterexample:

No of artists N = 5, No of performances n = 5 and no of days in the festival m = 3

Say the artists' deadlines are D = [1, 2, 3, 4, 5]

And their audience turnouts for those days are A = [3, 1, 2, 5, 6]

Using a greedy algorithm to allocate performances according to increasing order of audience turnouts, we shall select artists A5 on day 1, A6 on day and A3 on day 3. The audience turnout = 6 + 5 + 2 = 13

However, we can see that to maximise audience turnout we should select artists A1, A4 and A5 to have audience turnout = 3 + 5 + 6 = 14 which is greater than the above audience turnout

c)

We can fill the performance days in reverse order, giving priority to eligible artists with the highest audience turnout

Algorithm:

1. Initialize an empty max heap called EligibleAudienceTurnout
2. Sort artists on decreasing order of their deadline in array D
3. Initialize an empty array of size m called ArtistsPerformances
4. Initialize variable i = 0
5. Iterate using a variable j where j goes from m to 1

```
    a. Perform the following while D[i] ≥ j (i.e. We are selecting all artists whose
       deadlines are upcoming ahead of the current artists slot i)
        i. Insert D.AudienceTurnout into EligibleAudienceTurnout
       ii. i = i +1
    b. Set bestAudienceTurnout = EligibleAudienceTurnout.getMax()
    c. EligibleAudienceTurnout.deleteRoot()
    d. ArtistsPerformances[j] = bestAudienceTurnout
6. Return array of ArtistsPerformances
```

Ans 3

For ALG $T(n) = 7t(n/2) + n^2$

Using Master's theorem:

a=7, b=2

c=$\log_b a$ = $log_2 7$

Thus, no of leaves = $O(n^c) = O(n^{\log_2 7})$

$f(n) = O(n^2)$ from recurrence relation

Here, we can see that $O(n^{\log_2 7}) = O(n^{2.81..}) > O(n^2)$

So using case 1 of Master's theorem the time complexity of ALG = $\theta(n^{\log_2 7})$

For ALG' $T'(n) = aT'(n/4) + n^2 \log n$

Using Master's theorem:

a=a, b=4

c=$log_4 a$

Thus, no of leaves = $O(n^{\log_4 a})$

$f(n) = O(n^2 \log(n))$

Now, for the time complexity of ALG' to include the number of leaves, we need to have

O(leaves) ≥ O(f(n)) i.e.

$O(n^{\log_4 a}) \geq O(n^2 \log(n))$

Both orders are polynomial, we know that $O(n^2 \log(n)) > O(n^2)$ but smaller than $O(n^{2+\epsilon})$ for any $\epsilon > 0$. So using this we can compare the exponents of the above inequation

$$\log_4 a \geq 2$$

$$a \geq 4^2$$

$$a \geq 16$$

Case 1: When a = 16

Time complexity of ALG' = $\theta(n^{\log_4 16} \log^2 n) = \theta(n^2 \log^2 n)$

In this scenario ALG is always faster with $O(n^{\log_2 7}) = O(n^{2.81..})$ > $\theta(n^2 \log^2 n)$ which is not the requirement of the question

Hence, we shall only be using:

Case 2: When a>16

Time complexity of ALG' using case 1 of Masters Theorem= $\theta(n^{\log_4 a})$

For ALG' to be asymptotically faster than ALG, it's time complexity needs to be lesser i.e.

$\theta(n^{\log_4 a})$ < $\theta(n^{\log_2 7})$

Comparing the exponent of n we have

$$log_4 a < log_2 7$$

$$a < 4^{\log_2 7} = 2^{2 \log_2 7} = 2^{\log_2 7^2}$$

$$a < 7^2 = 49$$

Combining this with Case 2, we have a>16 and a<49

So the highest value of a is 49

Ans 4

We can calculate the time complexity for each step of StrangeSort:

(a) If n ≤ 1, return A unchanged
In this scenario, there is nothing to be done hence this step is O(1)

(b) For each item x ∈ A, scan A and count how many other items in A are less than x.
For each item x, we will have to scan the entire array A to determine the count
Thus, time complexity = n elements X O(n) work to scan the array = $O(n^2)$

(c) Put the items with count less than n/2 in a list B
We would have to traverse the entire array of n elements to determine the count so this step is O(n)

(d) Put the other items in a list C
We would again have to traverse the entire array of n elements to determine the count so this step is O(n)

(e) Recursively sort lists B and C using StrangeSort
We would repeat the above steps recursively. This step would be included in breaking into subproblems

(f) Append the sorted list C to the sorted list B and return the result
Merging 2 sorted lists is O(n)

The algorithm breaks the array A into 2 subproblems of sizes n/2. Thus our recurrence relation is:

$$T(n) = 2T(n/2) + O(n^2) + O(n)$$

We know $O(n^2) > O(n)$ so we can ignore the last term

$$T(n) = 2T(n/2) + O(n^2)$$

Solving this recurrence relation using Master's theorem:

a=2, b=2

c=$\log_b a$ = $log_2 2 = 1$
Thus, no of leaves = $O(n^c) = O(n)$

$f(n) = O(n^2)$ from recurrence relation

We can see here that $O(n^2) > O(n)$

Using case 3 of Masters Theorem we have

Time complexity = $\theta(n^2)$

= $O(n^2)$ (if f(n) = θ(g(n)) then it follows that f(n) = O(g(n))

Ans 5

(a) T(n) = T(n/2) + 2^n

Here a=1, b=2

c=$\log_b a$ = $\log_2 1$ = 0

Thus, no of leaves = $O(n^c)$ = O(1)$O(n^c) = O(1) f(n) = O(2^n)$ from recurrence relation

We can see here that $O(2^n) > O(1)$

Using case 3 of Masters Theorem we have

Time complexity = $\theta(2^n)$

(b) T(n) = 5T(n/5) + n log n–1000n

Here a=5, b=5

c=$\log_b a$ = $\log_5 5$ = 1

Thus, no of leaves = $O(n^c) = O(n) O(n^c) = O(1) f(n) = O(n \log n)$ from recurrence relation

We can see here that $O(n \log n) > O(n)$

Using case 3 of Masters Theorem we have

Time complexity = $\theta(n \log n)$

(c) T(n) = $2T(n/2) + log^2 n$

Here a=2, b=2

c=$\log_b a$ = $\log_2 2$ = 1

Thus, no of leaves = $O(n^c) = O(n) O(n^c) = O(1) f(n) = O(\log^2 n)$ from recurrence relation

We can see here that $O(\log^2 n) < O(n)$

Using case 1 of Masters Theorem we have

Time complexity = $\theta(n)$

(d) T(n) = $49T(n/7) – n^2 logn^2$

Masters theorem cannot be applied to this problem because for this recurrence relation f(n) is a negative function

(e) T(n) = $3T(n/4) + nlogn$

Here a=3, b=4

c=$\log_b a$ = $\log_4 3$

Thus, no of leaves = $O(n^c) = O(n^{\log_4 3})$

$f(n) = O(n \log n)$ from recurrence relation

We can see here that $O(n \log n) > O(n^{\log_4 3})$

Using case 3 of Masters Theorem we have
Time complexity = $\theta(n \log n)$

Ans 6

Binary search on a sorted linked list is similar to a binary search on a sorted array - with the extra step of finding the middle element in a linked list

Algorithm:

```
1. Initialize 2 pointers that would point to the start and last node of the LinkedList:
   Start = head
   Last = null

2. Perform the following while Last = null OR Last ≠ Start

   a. Assign a pointer Middle to the middle node of the LinkedList. To find the middle
      of the LinkedList with Start and Last nodes:

      i. Initialize 2 pointers, one of which would iterate 2 nodes in the LinkedList for
         every node iterated by the other node
         Slow = Start
         Fast = Start.next

      ii. Perform the following while Fast ≠ Last

          1. Fast = Fast.next

          2. if (Fast ≠ Last)

             a. Slow = Slow.next

             b. Fast = Fast.next

      iii. Return the Slow pointer as the middle node

   b. If searchValue = Middle.data

      i. return true

      Else if (searchValue > Middle.data)

          Start = Middle.next

      Else if (searchValue < Middle.data)

          Last = Middle
```

c. If the searchValue was not found in step 2, this means the element was not found. Hence here we will return false

Time complexity:

The additional step of finding the middle element in a linked list traverses through the entire linked list and so is O(n).

The remaining steps are identical to Binary Search in an array, and so it is O(log n)

Thus, time complexity = O(log n) + O(n) = O(n)

Ans 7

Algorithm:

MergeSort function for a LinkedList with input head is as follows:

```
1. Assign a pointer Middle to the middle node of the LinkedList. To find the middle
   of the LinkedList with Start and Last nodes:
    a. Initialize 2 pointers, one of which would iterate 2 nodes in the LinkedList for
       every node iterated by the other node
       Slow = Start
       Fast = Start.next
    b. Perform the following while Fast ≠ Last
        i. Fast = Fast.next
        ii. if (Fast ≠ Last)
            1. Slow = Slow.next
            2. Fast = Fast.next
    c. Return the Slow pointer as the middle node
2. Assign a pointer SecondHead (for the head of the right sub-LinkedList)
3. Assign Middle.next = null (to bread the left sub-LinkedList)
4. Recursively call step 1 with head of left sub-LinkedList. Assign the head returned
   to the pointer HeadNew1
```

5. Recursively call step 1 with head of right sub-LinkedList i.e. SecondHead. Assign the head returned to the pointer HeadNew2
6. Finally merge the linked lists from the last 2 steps using the following merge function with inputs HeadNew1 and HeadNew 2
    a. Initialize a pointer Merged where we will store the merged list and store a dummy node in it
    Merged.data = -1
    b. Initialize a temporary pointer Temp and point this to the Merged list
    Temp = Merged
    c. Perform the following while HeadNew1 ≠ null AND HeadNew2 ≠ null
        i. if (HeadNew1.data < HeadNew2.data)
            1. Temp.next = HeadNew1
            2. HeadNew1 = HeadNew1.next

        Else
        a. Temp.next = HeadNew2
        b. HeadNew2 = HeadNew2.next
        ii. Temp = Temp.next
    d. Now we will add any remaining nodes of the first or second linked list to the merged linked list
        i. Perform the following while HeadNew1 ≠ null
            1. Temp.next = HeadNew1
            2. HeadNew1 = HeadNew1.next
            3. Temp = Temp.next
        ii. Perform the following while HeadNew2 ≠ null
            1. Temp.next = HeadNew2
            2. HeadNew2 = HeadNew2.next
            3. Temp = Temp.next
    e. Return Merged.next

Time Complexity:

The additional step of finding the middle element in a linked list traverses through the entire linked list and so is O(n).

The remaining steps are identical to Merge Sort in an array, and so it is O(nlogn)

Thus, time complexity = O(nlog n) + O(n) = $O(nlogn)$

Ans 8

Algorithm:

```
1. Initialize iterators for the 2 skylines i, j = 0
2. Initialize height1, height2, CombinedHeight = 0
3. Initialize empty array ResultSkyline[]
4. Fill arrays SkyA and SkyB with skyline information for the 2 parts of the festival with
   size 2m+1 and 2n + 1
5. Perform the following while i<SkyA.size/2 AND j<SkyB.size/2
   a. if SkyA[2i] < SkyB[2j]
      i. Height1 = SkyA[2i+1]
      ii. Append SkyA to ResultSkyline[]
      iii. CombinedHeight = max(Height1, Height2)
      iv. i = i +1
   b. Else
      i. Height2 = SkyB[2j+1]
      ii. Append SkyB to ResultSkyline[]
      iii. CombinedHeight = max(Height1, Height2)
      iv. j = j +1
   c. Append CombinedHeight into ResultSkyline[]
6. Perform the following while i < SkyA.size/2:
        a. if SkyA[2i] > SkyB[2j+1]
            i. Height2 = 0
        b. Height1 = SkyA[2i+1]
        c. CombinedHeight = max(h1, h2)
        d. Append SkyA[2i] to ResultSkyline[]
        e. Append CombinedHeight into ResultSkyline[]
        f. i = i + 1
    7. Perform the following while j < SkyB.size/2:
        a. if SkyB[2j] > SkyB[2j+1]
            i. Height1 = 0
        b. Height2 = SkyB[2j+1]
        c. CombinedHeight = max(h1, h2)
        d. Append SkyB[2i] to ResultSkyline[]
        e. Append CombinedHeight into ResultSkyline[]
        f. j = j + 1
    8. Initialize RightmostXCoordinate = max(skyA[2i+1], skyB[2j+1])
    9. Append RightMostXCoordinate into ResultSkyline[]
   10. Return ResultSkyline[]
```

Time Complexity:

The algorithm iterates through SkyA array m times and the SkyB array n times. So time complexity = O(m+n)

b) Algorithm:

```
    1. Initialize an array called Stages and populate it with tuples containing information of
       the stages
    2. If Stages.length ==1:
        a. return array containing this only tuple i.e.
           [Stages[0].Xleft,Stages[0].height,Stages[0].XRight]
    3. Else:
        a. Initialize variable MidPt = Stages.size/2
        b. Divide Stages into LeftPart and RightPart based on the MidPt
        c. Initialize empty arrays SkyA, SkyB and ResultSkyline
        d. SkyA = Goto step 2 using LeftPart as parameter
        e. SkyB = Goto step 2 using RightPart as parameter
        f. ResultSkyline = Use algorithm described in Q8a) with parameters SkyA and
           SkyB
    4. Return ResultSkyline
```

Time Complexity:

The algorithm divides the stages into LeftPart and RightPart so we have 2 subproblems

The Merge step has n operations in the worst case

Hence our recurrence relation is

$T(n) = 2T(n/2) + n$

We have a = 2, b = 2

c = $log_b a = log_2 2 = 1$

No of leaves = $n^c = O(n)$

$f(n) = O(n)$

Here we can see O(Number of leaves) = O(f(n))

So using Case 2 of Masters Theorem we have

Time Complexity = $\theta(n \log n) = O(n \log n)$

Ans 9

a) For a TargetAmount, we can sell tickets worth TargetAmount-2 and then sell either a 2 dollar ticket OR we can sell tickets worth TargetAmount-1 and then sell a 1 dollar ticket.

The subproblems for this problem is to count the number of ways we can sell tickets for all amounts starting from 0 to TargetAmount

b) The recurrence relation becomes T(n) = T(n-1) + T(n-2)

c) Assume that the target that is to be reached is stores in variable n

Algorithm:

```
1. Initialize Array of size n + 1
2. For i iterating from 0 to n:
    a. Array[i] = 0
3. Array[0] = 1
4. Array[1] = 1
5. For j iterating from 2 to n:
    a. Array[j] = Array[j-1] + Array[j-2]
6. Return Array[n]
```

d) The best cases are:

Case 1: n=0: In this scenario there is only 1 way to reach the target by selling no tickets

Case 2: n=1: In this scenario there is only 1 way to reach the target by selling a $1 ticket

e) Time Complexity:

For our algorithm, the array of size n+1 is iterated twice

Hence time complexity = O(n)

Ans 10

a) We are required to calculate the min no of packages from a subset of the packages to reach a smaller target. In other words, the subproblems are the min number of packages from the first i packages to reach a target weight j. i would go from 0 to n while j would go from 0 to W

b) We would have 2 cases:
Case 1: If we exclude the current package and find a combination of packages that add up to W from the previous packages

Case 2: If we include the current package and add this 1 to the solution of the number of packages that add up to W - Array[i] from the previous packages

The recurrence relation for the above is:
$T(n, W) = min(F(n-1, W), F(n-1, w-w_n)+1)$

c)

Algorithm:

```
1. Initialize W = target weight
2. Initialize PackageWeights array with package weight stores in it of size n
3. Initialize 2-dimensional Array of size [n+1][W+1]
4. For i iterating from 0 to n:
    a. For j iterating from 0 to W:
        i. If i == 0:
            1. Array[i][j] = $\infty$
        ii. Else:
            1. Array[i][j] = 0
5. For i iterating from 1 to n:
    a. For j iterating from 1 to W:
        i. If j - PackageWeights[i] < 0:
            1. Array[i][j] = Array[i-1][j]
        ii. Else:
            1. Array[i][j] = min(Array[i-1][j], Array[i-1][j-PackageWeights[i]] + 1)
6. Return Array[n][W]
```

d) Base cases:

Case 1: If W= 0: In this scenario we can reach the target with 0 packages i.e. the minimum

Case 2: If n=0 and W> 0: In this scenario we would not be able to reach the target in any manner, so the value of packages is ∞

e)

In steps 4&5, we iterate a 2 dimensional array of size n X W.

Hence time complexity = O(nW)