Assignment 3

Vaibhav Agarwal(B15139)

Q1. Let the value of the optimal solution of the problem of size 'n' be denoted by OPT(n). Now, sort the tasks in increasing order of finish-time. Consider the jth task. We include this task into the schedule, if the sum of values of this task - v_j - and the optimal-value of the schedule preceding task 'j' and not conflicting with it – OPT(p(j)) – is at least as good as the optimal value till the (j-1)th task. Otherwise, we leave this task out of our schedule. Also, we update the optimal value of the schedule till task j as OPT (j) = max(vj + OPT (p(j)), OPT (j − 1)).
Sorting takes O(n.logn) time. Computation of conflicts, p(j) takes O(n2) time. M-Compute-Opt will compute the optimal values in at most O(n) operations and then the Find-Solution calls itself recursively only on strictly smaller values, making a total of O(n) recursive calls spending constant time per call. Therefore, the algorithm has an overall time-complexity of **$O(n^2)$**.

Psuedocode:

a)

ComputeOpt(j):

```
If j = 0 then
        Return 0
Else
        Return max(vj+Compute-Opt(p(j)), Compute-Opt(j − 1))
Endif
```

b)

MComputeOpt(j):

```
If j = 0 then
        Return 0
Else if M[j] is not empty then
        Return M[j]
Else
        Define M[j] = max(vj+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
        Return M[j]
Endif
```

c)

FindSolution:

```
If j = 0 then
        Output nothing
Else
        If vj + M[p(j)]≥ M[j − 1] then
                Output j together with the result of Find-Solution(p(j))
        Else Output the result of Find-Solution(j − 1)
```

Endif

Endif

Q2. The algorithm to count the number of inversions. At each divide, we count the number of inversions in each half. At each merge, we count the number of inversions while merging the two-halves by adding the number of values left in the 'smaller half' when an element from the 'bigger half ' is smaller.The space-complexity of the algorithm is O(n), as in Merge-Sort. The time-complexity of the algorithm is **O(n.logn)**. Merge-and-Count procedure takes O(n) time.There will (logn) merge operations.

Psuedocode:

MergeAndCount(A,B):

Maintain a Current pointer into each list,
Initialized it to point to the front elements
Maintain a variable Count for the number of inversions,
Initialize it to 0
While both lists are nonempty:
Let ai and bj be the elements pointed to by the Current pointer
Append the smaller of these two to the output list
If bj is the smaller element
then Increment Count by the number of elements remaining in A
Endif
Advance the Current pointer in the list from which the smaller element was selected.
EndWhile
Once one list is empty,
append the remainder of the other list to the output
Return Count and the merged list

SortAndCount(L):

If the list has one element then :
there are no inversions
Else :
Divide the list into two halves:
A : contains the first n/2 elements
B : contains the remaining n/2 elements
(r_A, A) = Sort-and-Count(A)
(r_B, B) = Sort-and-Count(B)
(r, L) = Merge-and-Count(A, B)
Endif
Return r = rA + rB + r, and the sorted list L

Q3. The algorithm used to solve the problem uses divide-and-conquer. First, sort the points on their x-coordinates, in ascending order. Next, find the closest-pair of points in the left half and right half of the plane separately. Let d be the minimum of the distances between points in these two pairs.
Now find the set of points which lie at a distance less than or equal to d from the halfline. Sort this set on y-coordinates. Next, search for the closest-pair of points in this sorted-list by considering every-pair of points. It can be shown that the points will lie within 15 positions of each other.
The time-complexity of the algorithm is O(n.logn).The sorting of the points takes O(n.logn) time. The remainder of the algorithm then divides the points in two equal halves and spends constant time to merge the solution from those two subproblems.
Therefore, the algorithm has a time-complexity of **O(n.logn)**.

Pseudo Code

ClosestPair(P):

Sort P on x coordinates (O(n log n) time)
(p^*_0, p^*_1) = ClosestPairRec(P)
Return (p^*_0 , p^*_1).

ClosestPairRec(P):

If $|P| \leq 3$
then find closest pair by measuring all pairwise distances
Endif
Construct Left, Right dividing the array in approximately half
(l^*_0,l^*_1) = ClosestPairRec(Left)
(r^*_0,r^*_1) = ClosestPairRec(Right)
δ = min($d(l^*_0 , l^*_1)$, $d(r^*_0 , r^*_1)$)
x^* = maximum x-coordinate of a point in Left set/midpoint
L = {(x,y) : x = x}*
S = points in P within distance δ of L.
S_y = Sort S on y coordinates Sy (O(nlogn) time)
For each point $s \in S_y$:
compute distance from s to each of next 15 points in Sy
Let s, s' be pair achieving minimum of these distances (O(n) time)
If d(s,s') < δ
then Return (s,s')
Else if $d(l^*_0, l^*_1) < d(r^*_0, r^*_1)$
then Return (l^*_0, l^*_1)
Else Return (r^*_0, r^*_1)
Endif

Q5. The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance d(u) from s; this is the “explored” part of the graph. Initially S = {s}, and d(s) = 0. Now, for each node v ∈ V −S, we determine the shortest path that can be constructed by traveling along a path through the explored part S to some u ∈ S, followed by the single edge (u, v). The time complexity remains **O(ELogV))** as there will be at most O(E) vertices in priority queue and O(Log E) is same as O(Log V).

.

Pseudo Code:

```
Dijkstra(Graph) :
        dist[source]  := 0
        for each vertex v in Graph:
                if v ≠ source
                        dist[v]  := infinity
                 add v to Q
        while Q is not empty:
                 v := vertex in Q with min dist[v]
                 remove v from Q

                for each neighbor u of v:
                        alt := dist[v] + length(v, u)
                        if alt < dist[u]:
                        dist[u]  := alt
        return dist[]
```