CS 515 - Midterm

Ga Young Lee (Student ID: 933-961-022)

February 9, 2021

Question 1

Recurrence: $S(n) = S(n-1) + S(n-2)$
Base cases:

- $S(0) = 1$
- $S(1) = 2$ the possibilities are $\{E, C\}$

Proof of Correctness: Proof of correctness is demonstrated by induction.

- **Base case** When n = 1, it is the base case, so S(1) = 1.
- **Inductive steps** Let A[1,...,n] be an array of n cells that can contain either E or C to comply with the distancing condition. When the number of seats is increased by one, from A[1,...,k] to A[1,...,k+1], we can determine the value of $k+1$th cell by considering the value of kth cell.
  - Specifically, if A[1,...,k] is ending with E, then either E or C can be placed in the last cell of A[1,...,k+1], which increases the possibility by 2.
  - If A[1,...,k] is ending with C, then only E can be the last value of A[1,...,k+1], which increases the possibility by 1.
- Following the induction, we can see that S(k+1) can be derived from S(k). Therefore, it shows the correctness of the recursive algorithm, $S(n) = S(n-1) + S(n-2)$.

Question 2

Algorithm

To find the maximum number of coats to wear following the constraints, we can refer to the Longest Increasing Path problem using Topological Sort.

To construct a graph, we can refer to the two constraints. The array, $S[1, ..., n]$, gives us the size of all coats, so that we can sort the coats in ascending order to indicate where to start. Also, the matrix, $incomp[1, ..., n][1, ..., n]$, provides a hint on the connectivity of the graph. If $incomp[i, j] = 1$, it means that vertex i and vertex j are not connected, which is the opposite of the standard adjacency matrix of a graph where connectivity is marked with 1. Given this setting, we can build a graph G = (V, E) where G contains V vertices and E edges and V[i] is the value of the longest path ending at i_{th} node. The general Longest Path algorithm can be described as follows:

$$LP(i) = \begin{cases} \max_{j, j \to i \in E}(LP(j) + 1), & j < i \exists \text{ an edge } j \to i \in E \\ 1 & \text{otherwise} \end{cases}$$

```
# LP(G[1..i], j): return the length of the longest increasing path from A[1..i]
each element is not greater than A[j].

def LP(G[1..i], k):
    # Base case:
    if i==1:
        if V[i] <= V[k]:
            l[i, k] = 1
        else:
            l[i,k] = None
    else:
        for each node j that have edge j to i
            l[i, k] = max(LP(G[1..j], j) if V[j] > V[i], LP(G[1..j], i) + 1 if V
[j] <= V[i])
    return l[i, k]

# Main function call:
# Given G[1..n]:
Initialize matrix l of size n by n with the diagonal filled with 1.
max(LP(G[1..i], max(V[1..n])), for each node i in G without degree = 0)

```

Runtime Complexity

The runtime complexity of this algorithm is $O(n^2)$ since we fill out a 2d table to keep track of the longest path of each vertex.

Proof of Correctness

- **Base Case**: If the graph has only one node (n = 1), the algorithm will return the one and only coat, because that's the longest path in this case.
- **Inductive Steps**: Assume that the algorithm finds the longest increasing path to a DAG with k nodes. Then, based on the assumption, we will build the case to show it holds for $k+1$ nodes well. When $n = k$, max(l[i, index of max V[1..k]]) will return the length of the longest increasing path. When there's an additional k_{th} node is included in the graph, the recursive will return max(`LP(G[1..j], j)` if `V[j]` $>$ `V[i]`, `LP(G[1..j], i)` + `1` if `V[j]` $<=$ `V[i]`), which consists of subproblems of k or less.
  - If `V[k+1]` ¡ `V[i]`, we will just return all the longest increasing path ending at node i as the lastly added node is not included in the longest path answer.
  - If `V[k+1]` $\leq$ `V[i]`, the current node is included in the answer and the recursive call will return the longest increasing path terminating at node i which is linked to `V[k+1]`. Given this, we will find the maximum of all the path ending at k+1 to find the longest path.

Therefore, the inductive proof shows that the algorithm is valid when $n = k+1$ given that it holds for $n = k$.

Question 3

To compute the heaviest possible path that starts from the root, we need to consider the given tree's shape. Let `max_weight` be the maximum weight of the path staring from r to its subtrees. Given this above setting, we can describe the recursive algorithm as follows:

Algorithm

```
# max_weight(r): return the maximum weight of the path starting from r
# c_l is left subtree of r, c_r is right subtree of r

def max_weight(r):
    # Base case:
    if r has no children:
        return 0
    else:
        # w_l is the weight of the edge between r and its left child
        # w_r is the weight of the edge between r and its right child
        return max(max_weight(c_l)+w_l, max_weight(c_r)+w_r)

# Main function call:
if max_weight(r) < 0:
   return 0
else:
   return max_weight(r)

```

Runtime Complexity

The recursive call is invoked at every node once to update the maximum weight. For each node, it involves a summation of its `max_weight` and the weight of the edge between the node and its parent. Therefore, the time complexity is $\mathcal{O}(V)$ where V is number of nodes.

Proof of Correctness

Proof of correctness is demonstrated by induction.

- **Base case** if G has only root node, `max_weight` returns 0, which is correct.
- **Inductive steps** Consider a tree with root r. Assume that the maximum weight of path to c_{ir}, the right subtree of the root, and the maximum weight of the path to c_{il}, the left subtree of the root, were computed as `max_weight(c_i_r)` and `max_weight(c_i_l)`. Also, let w_r and w_l be the weight of the edge connecting rightsubtree of root and leftsubtree of the root respectively. To compute `max_weight` for r, we need to compare the sum of `max_weight(c_i_l)` and w_l and `max_weight(c_i_r)` and w_r.
  - If `max_weight(c_i_r)` + w_r ¿ `max_weight(c_i_l)` + w_l, then we take the value of the heaviest path to the root's right subtree and add the weight of the edge connecting the right subtree to root.
  - Otherwise, take the value of the heaviest path to the root's left subtree and add the weight of the edge connecting the left subtree to root.

Based on the inductive steps, we can see that the heaviest path starting from root can be computed by referencing the heaviest path of its subtrees. Therefore, the recursive algorithm is valid.

Question 4

Algorithm

To compute the set of lines that do not intersect, assume that n points on $y = 0$ from left to right is the index of an array, $1, ..., n$ and the matching points on $y = 1$ be the value of the array at the corresponding index, $arr[1], ..., arr[n]$. In order to meet the condition that not a single pair intersects, the lines should be selected in a way such that if $m < l$ then $arr[m] < arr[l]$.

Given this setting, the problem is similar to longest increasing subsequence. There are many overlapping substructures, so we will use Memoization approach to solve the problem.

```
    # LIS returns length of the longest increasing subsequence in arr
    def LIS(arr):
        n = len(arr)

        # Declare the list (array) for LIS and
        # initialize LIS values for all indexes
        LIS = [1]*n

        # Compute optimized LIS values in bottom up manner
        for i in range (1 , n):
            for j in range(0 , i):
                if arr[i] > arr[j] and LIS[i] < LIS[j] + 1 :
                    LIS[i] = LIS[j]+1

        # Initialize maximum to 0 to get
        # the maximum of all LIS
        maximum = 0

        # Pick maximum of all LIS values
        for i in range(n):
            maximum = max(maximum , LIS[i])

        return maximum

```

Runtime Complexity

Te optimal solution of each subproblem is recorded in the table. There is a nested loop used to iterate each element in the array. Therefore, the time complexity is the product of the number of different problems and the amount of non-recursive work, so it is $O(n^2)$.

Proof of Correctness

Given an array, A, with n elements, we can show the proof of correctness of the algorithm using induction:

- **Base case**: When $n = 1$, $LIS(A) = 1$.
- **Inductive steps**: When $n = l$, let's assume that the algorithm holds and returns the length of the longest increasing subsequence, $LIS(A) = k$. When $n = l + 1$, we can consider the following two scenarios built on the previous case.
  - If the last element added, $arr[l + 1]$, is smaller than the smallest value of the temporary subsequence stored, it can be a new starting point of the longest increasing subsequence. Thus, record the value as the first element of Subsequence and iterate again to find the optimal answer.
  - If $arr[l + 1]$ is bigger than the largest element of the temporary subsequence stored, then append the value to the subsequence. As a result, the length of the longest increase subsequence is increased by one from the previous case and becomes $k + 1$.

Therefore, by induction, the above algorithm to find a set of lines where none of them intersect is valid.