Foundation of Algorithm

Priyanka Dwivedi
Srikant Lakshminarayan

March 27, 2018

Problem - 1

Algorithm Used : Longest increasing Subsequence
Algorithm:

1. Use Longest increasing subsequence to calculate value that are in-place which is as follows.
   (a) Verbal Description of DP array: The Dp array is 1D array which stores the length of increasing subsequence for every given element 1,2...,k and a_k is part of last element of LIS.
   (b) Solution : Maximum in the dp array. Recurrence : dparray = 1 + dparray[j] or include that element where j is the index that maximizes as subject to j < k and $a_j < a_k$
2. Now we know which values are in place and we need to find which value are remaining to be put in correct position by subtracting total length of sequence by length of increasing subsequence.

Data Structure Used:
Python Inbuilt List : Complexity and Description of operation performed:

1. Access element at particular index : O(1)
2. Append Element in list:O(1)
3. Length of list len(n) : O(1)

Overall Complexity of Algorithm

1. Initialize dp array as 1 for sequence length 'n' O(n)
2. For every element fill dp array by checking whether any element less than that exist if yes what is the dparray value of that element. $O(n^2)$
3. Subtract the length of sequence - length of common subsequence to get element needs to be moved O(1)
4. Overall complexity $O(n^2)$

Correctness :
In this question, we use dynamic approach to fill longest increasing subsequence. We know LIS will give us value which are in increasing order i.e inplace and we need to find remaining value which needs to be move to sort cards. So we subtract total length of sequence by length of increasing subsequence and get number to moves required to sort the cards. This will always give correct value because we have track of cards which are in place by using longest increasing subsequence.

Problem - 2a

Algorithm Used :Greedy Approach

Algorithm:

1. The following greedy approach was used in fully parenthesizing the given algebraic expression to get the maximum value.
2. A given algebraic expression is taken in as list and and scanned one by one.
3. If there is a number, then it is put in stack.
4. If the element is a multiplication symbol den it is skipped and if the number is a addition symbol then one element from stack is popped and added with the number after the addition symbol and that new number is put in stack.
5. In this way we get a stack full of numbers which only have to be multiplied.
6. Multiplication will give the maximum number obtained by fully parenthesizing expression.

Data Structure Used:
Python Inbuilt List : Complexity and Description of operation performed:

1. Access element at particular index O(1)
2. Length of list len(n) O(1)

Overall Complexity of Algorithm

1. Iterate the expression O(n)
2. push in stack O(1)
3. Overall complexity O(n)

Correctness:
We have used Greedy approach to solve the problem. Since the two operators are + and * so if we first add up all value in sequence then multiply so the multiplication factor increases. If we first multiply then addition will not increase the factor by much as much as multiplication will.If we solve expression from left to right then we wont get maximum output. So parenthesizing around addition first and then multiplying increases the factor by m where m is the number to be multiplied after adding.
Ex : 3 * 4 + 5
If multiplied first then o/p is 17
If solved left to right then o/p is 17
If used greedy approach the o/p id 27.
Therefore we can say that using greedy approach and parenthesizing addition first and then around multiplication will give maximum output.

Problem - 2b

Algorithm Used : Dynamic Approach.
Heart of the Algorithm:

1. Verbal Description of DP array: The Dp array is 2D array which stores different maximum and minimum values when the parenthesis is applied at different positions.
2. Solution: Maximum in the dparray.
3. Recurrence :
   Base Case : S[L][R] = numbers in sequence when L == R
   Relation. : S[L][R] =min(S[L][R]+ S[k+1][R]+$a_{L-1}a_k a_R$)
   $S_{min}[L][R] = min_{k=L..R-1} S_{min}[L][k] sign[k]$
   $S_{max}[k+1][R] = S_{max}[L][K] sign[k] (S_{max}[k+1][R]$
   $S_{min}[L][R] = max_{k=L.\ldots..R-1} (S_{max}[L][K] sign[k]))$

$S_{max}[k+1][R] = S_{max}[L][k] sign[K] (S_{max}[k+1][R]$

Case maximum:- When the kth symbol is + then we take the maximum of k+1 R.
If it is - then we take minimum of K+1 R.
Both times we take maximum value of L and R
Case minimum:- When the kth symbol is + then we take the minimum of k+1 R.
If it is - then we take maximum of K+1 R.
Both times we take minimum value of L and R.

Data Structure Used:
Python Inbuilt List : Complexity and Description of operation performed:

1. Access element at particular index : O(1)
2. Length of list len(n) : O(1)

Overall Complexity of Algorithm

1. Take the sequence and split into two lists:- O(n)
2. Initialize DP array math.infinity O(nm)
3. Compute the cost for filling the matrix O(n^3)
4. Overall complexity O(n^3)

Problem - 3

Algorithm Used : Dynamic Programming Approach

Heart of the Algorithm:

1. Verbal Description of DP array: The Dp array is 2D array which stores different values of the headaches incurred while putting employees in line.
2. Final Solution : h[r][c]

3. Recurrence :

$$h[r][c] = \min \begin{cases} \text{r== 0 :} & (s + h[r][c-1], m + h[r-1][c-1]); \\ \text{c==0:} & s + h[r-1][c], m + h[r-1][c-1], \\ \text{r-1 and c-1>0} & s + h[r-1][c], s + h[r][c-1], m + h[r-1][c-1]; \\ \text{r-2 >0} & a, m + h[r][c-2]; \\ \text{c-2>0} & b, m + h[r-2][c]; \end{cases} \tag{1}$$

where s = sending single person from line
m = matching two person either from same line or one from each line
a,b = s+h[r-1][c],s+h[r][c-1],m+h[r-1][c-1]

Data Structure Used:
Python Inbuilt List : Complexity and Description of operation performed:

1. Access element at particular index : O(1)
2. Length of list len(n) : O(1)
3. To find maximum of n elements : O(n)

Overall Complexity of Algorithm

1. Initalize the array by 0 :- O(mn)
2. Compute and find minimum cost required to put employees in line O(mn)
3. Overall complexity **O(mn)**

Problem - 4

1. Algorithm Used : BFS.
   Algorithm:

   (a) Take maze from user
   (b) Get start location i.e in maze when 2 comes that is start location and start solving maze from that location.
   (c) To traverse maze we have used BFS.
      i. Store root element in queue.

ii. Mark it visited
iii. Check for 4 neighbors if there is a way and not in visited store in queue and loop runs until either queue is empty or reached destination.
iv. if destination is reached we have kept the track of parents so we backtrack to find the length of shortest path.

Data Structure Used:
Python Inbuilt List : Complexity and Description of operation performed:

(a) Access element at particular index : O(1)
(b) Length of list len(n) : O(1)
(c) To find maximum of n elements : O(n)

Overall Complexity of Algorithm

(a) Find start location	O(mn)
(b) Use BFS to travel till either maze is completely traversed or reached destination	O(mn)
(c) Backtrack to get path	O(mn)
(d) Overall complexity	O(mn)

Correctness:
In BFS we take root and check all the neighbor if they are destination or not. If destination not found we expand one child and check if we have reached destination and continues to go on until destination is found or reached end. this creates a ripple effect where we reach on every level and check for child if destination is reached or not. In other case i.e DFS we keep expanding single child and if no path found we travel back and expand other child. But in BFS we reach all levels at same time and get the shortest path to reach till that destination.In DFS it is not guaranteed that does not track of which node is visited before and don't know about distance of node as it just goes on expanding one of its child and traverse back if destination not found. On other hand BFS visits child in increasing order of levels and hence we know that we will get shortest path using BFS.