CSCI 6212 – Design and Analysis of Algorithms (Fall 2023)  
Project Assignment 3  
Abde Manaaf Ghadiali – G29583342, Gehna Ahuja – G35741419, Venkatesh Shanmugam – G27887303, Manikanta Allanki – G42549699

**Problem Statement - Maximum Value but Limited Neighbours**

You are given an array a[1…n] of positive numbers and an integer k. You must produce an array b[1…n], such that:

1. For each j, b[j] is 0 or 1,
2. Array b has adjacent 1s at most k times, and
3. sum\_{j=1 to n} a[j]\*b[j] is maximized.

For example, given an array [100, 300, 400, 50] and integer k = 1, the array b can be: [0 1 1 0], which maximizes the sum to be 700. Or, given an array [10, 100, 300, 400, 50, 4500, 200, 30, 90] and k = 2, the array b can be [1, 0, 1, 1, 0, 1, 1, 0, 1] which maximizes the sum to 5500. To be precise about the definition of adjacency: sequence [0, 1, 0, 1, 0, 1, 1, 1] has two adjacent 1s. Sequence [0, 1, 0, 0, 1, 1, 1, 1] has 3 adjacent 1s. Sequence [1, 0, 1, 1, 0, 1, 1, 1] also has 3 adjacent 1s.

**Theoretical Analysis**

**Notation** – Let MVLN[i][j] be the maximum sum of a[m]\*b[m] for 1 <= m <= j, subject to the constraints that there are at most i adjacent 1s in b[1…j].

**Proof of Optimality** – We can prove that MVLN[i][j] is the optimal solution by induction on i. The base case is when i = 1, in which case MVLN[1][j] = a[1]\*b[1] if j = 1 and 0 otherwise. This is clearly optimal for the subproblem with only one element.

For the inductive step, assume that MVLN[i-1][m] is optimal for any 0 <= m <= j. Then we can show that MVLN[i][j] is optimal by considering two cases:

1. If k = 0, then MVLN[i][j] = MVLN[i - 1][j] + MVLN[i - 1][j], which is the maximum sum we can get by appending a 0 to any optimal solution for the subproblem with i - 1 elements; if k = 1, then MVLN[i][j] = MAX(MVLN[i ][j-1], MVLN[i -1][j]) + a[j], which is the maximum sum we can get by appending a 1 to any optimal solution for the subproblem with i - 1 adajcent 1’s with j elements.

Therefore, by induction, MVLN[i][j] is optimal for the original problem.

**Recurrence Relation** – MAX(MVLN[i ][j-1], MVLN[i - 1][j - 1] + a[j], MVLN[i ][j -2] + a[j])

**Algorithm** –

def *maximum\_value\_limited\_neighbor(arr, k):*

*# Create a 2 D array : MVLN(i,k) where we store the max value till ithposition including k adjacent 1’s*

*MVLN = 2 D array (k+1) X len(arr)*

*#Create a back\_track array and copy of it to change when we go to new k*

*# initialize b\_track array with size (len(arr),len(arr)*

*# Base case for this problem would be when k=0 where we don’t have any adjacent 1’s*

*MVLN[0][0] = arr[0]; MVLN[0][1] = max(arr[1], MVLN[0][0]*

*# create a base case for back track*

*for i from 2 to len(arr) - 1:*

*MVLN[0][i] = max(MVLN[0][i-1], MVLN[0][i-2] + arr[i])*

*# Handling the base cases for for first an second indices*

*for j from 1 to k:*

*MVLN[k][0] = arr[0]*

*MVLN[k][1] = arr[0] +arr[1]*

*# populating MVLN arr for*

*for j from 1 to k:*

*for i from 2 to len(arr) - 1:*

*MVLN[j][i] = max(MVLN[j][i - 1], MVLN[j - 1][i - 1] + arr[i], MVLN[j][i - 2] + arr[i])*

*# back\_track array is populated by*

*# if max is MVLN[j][i - 1],b\_track[i] = 0 and b\_track[1…i-1] is same as b\_track[1….i] as b\_track[i]=0*

*#if max is MVLN[j - 1][i - 1] + arr[i] then b\_track[i]=1 b\_track[i-1]=1 and b[1…i-1] as same as b[1…i-1] of k-1*

*#if max is MVLN[j][i - 2] + arr[i] then b\_track[i]=1 b\_track[i-1]=0 and b[1…i-2] have same adjacent 1’s*

*b\_track\_copy = b\_track*

*Return MVLN[k][len(arr]-1], b\_track[-1]*

**Experimental Analysis**

* **GitHub Project Repository Link –** [**https://github.com/default741/CSCI\_6212\_Course\_Notes/tree/main/project-files/project-03**](https://github.com/default741/CSCI_6212_Course_Notes/tree/main/project-files/project-03)
* **Program Listing**

I have executed the code for values of n ranging from 2 to 1E5 with increments of 25% of the nearest multiple of 10, which can be seen in the following section of Output Numerical Data.

* **Data Normalization Notes**

To Normalize the theoretical time, we take the average of the experimental time and divide that by the average of the theoretical time, that give us the scaling constant which we multiply with the theoretical time. This gets the theoretical and experimental values in the same range. But the values of n are still too large to compare with the execution times. So, we log the values of n as well as we log the values of theoretical and experimental execution Times.

* **Output Numerical Data**



* **Graph**

* **Graph Observations**

We can visually observe from the graph that the Theoretical Execution Time and Experimental Execution Time follow a similar path for same values of n. Since we assume that the variable assignment and conditional checking and expression evaluation takes constant time for theoretical values, it does not hold true for experimental values. Hence, we see the Bumps in the graph for the experimental values.

**Conclusion:**

From the above plot we can see that the theoretical complexity matches approximately matching the curve of Experimental complexity. The theoretical time complexity is **0(n k)** and to get the array of which elements are chosen, we need the array with the size of (len(arr), len(arr)) so space complexity is o(n2)