DAA Assignment 4

GROUP 17
PRAVALLIKA KODI (IIT2019234)
VISHWAM SHRIRAM MUNDADA (IIT2019235)
NOONSAVATH SRAVANA SAMYUKTA (IIT2019236)
B.tech Information Technology business Informatics
Indian Institute of Information Technology, Allahabad

4 April 2021

0.1 Question

Given a 2D array, find the maximum sum subarray in it.

0.2 Introduction

Here, different approaches are analysed and used to achieve results. These approaches are:
1) Brute force
2) Using Dynamic programming

In this report we will explain our solution approach. We explain our code in detail. We will discuss the time complexity analysis and the space complexity analysis. And last but not least, the conclusion.

0.3 Code Explanation

0.3.1 Brute force

In the brute force approach we try to check every possible rectangle in the given n X m 2D array(where n,m are number of rows and columns respectively). Set the position of the top-left and bottom-right corners of the sub-rectangle and adding the integers within it while iterating through all the rows sequentially. Parallelly we try to find the maximum subarray sum value.

Algorithm 1 Brute force

```
int A[101][101]

function MAIN()
maxSum <- INT_MIN
tempSum <- 0
x <- 0, y <- 0, z <- 0, w <- 0
n <- 0, m <- 0
input n, m

for i <- 0 to n-1
   for j <- 0 to m-1
      input A[i][j]
   end
end

for i <- 0 to n-1
   for j <- 0 to m-1
      for k <- 0 to n-1
         for l <- 0 to m-1
            tempSum = FINDSUM(i,
               j, k, l)

            if tempSum > maxSum
               x <- i
               y <- j
               z <- k
               w <- l
               maxSum = tempSum
            end if
         end
      end
   end
end

print x, y, z, w, maxSum
```

Algorithm 2 Maxsum Pseudo Code

```
function FINDSUM(x, y, z, w)
   sum <- 0

   for i <- x to z
      for j <- y to w
         sum <- sum + A[i][j]
      end
```

```
    end

    return sum
```

0.3.2 Dynamic programming

We use Kadane's algorithm to reduce the time complexity to O(n^2Xm).

The idea is to fix the top and bottom rows one by one and find the maximum sum contiguous columns for every top and bottom row pair.
We basically find left and right column numbers (which have maximum sum) for every fixed up and bottom row pair. To find the left and right column numbers, calculate the sum of elements in every column from top to bottom and store these sums in an array say temp[]. temp[i] indicates sum of elements from top to bottom in column i. If we apply Kadane's 1D algorithm on temp[], and get the maximum sum subarray of temp, this maximum sum would be the maximum possible sum with top and bottom as boundary rows. To get the overall maximum sum, we compare this sum with the maximum sum so far.

Algorithm 3 **Dynamic programming**

```
int A[101][101]

function MAIN()
    r <- 0, c <- 0, maxSum <-
        INT_MIN
    x <- 0, y <- 0, z <- 0, w <- 0
    input r, c

    for i <- 0 to r-1
        for j <- 0 to c-1
            input A[i][j]
        end
    end

    for i <- 0 to r-1
        vector<int> sum(c)
        for j <- 0 to r-1
            for col <- 0 to c-1
                sum <- sum + A[j][col]
            end

            vector<int> res <-
                KADANE(sum)
            if maxSum < res[2]
                x <- i
                y <- res[0]
                z <- j
                w <- res[1]
                maxSum = res[2]
            end if
        end
    end

    print x, y, z, w, maxSum


function KADANE(V)
    maxSum <- INT_MIN, tempSum <- 0
    st <- -1, end <- -1, localSt <-
        0

    for i <- 0 to length[V]
        tempSum <- tempSum + V[i]
        if maxSum < tempSum
            st <- localSt
            end <- i
            maxSum <- tempSum
```

```
    end if

    if tempSum < 0
      localSt <- i+1
      tempSum <- 0
    end if
  end

  vector<int> res <- {st, end,
     maxSum}
  return res
```

0.4 Algorithm Analysis

0.4.1 Brute force approach

In the brute force approach we try to check every possible rectangle in the given n X m 2D array(where n,m are number of rows and columns respectively).
This solution requires 6 nested loops :
2 for the summation of the sub-matrix O(n x m)

4 for start and end coordinate of the 2 axis O(n$^2Xm^2$)

The overall time complexity is O(n$^3Xm^3$)

As we did not allocate extra space in this approach, space complexity is O(1).

0.4.2 Dynamic programming

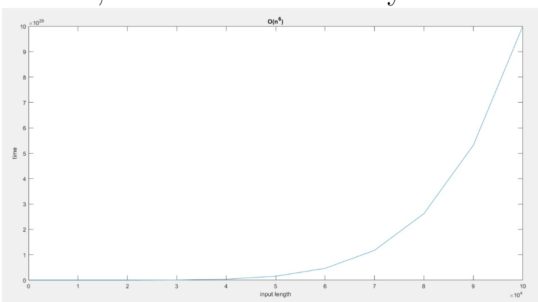
Basically, Dynamic algorithm (complexity: O(n)) is used inside a naive maximum sum subarray problem (complexity: O(n x m)).

This gives a total complexity of O(n^2Xm)

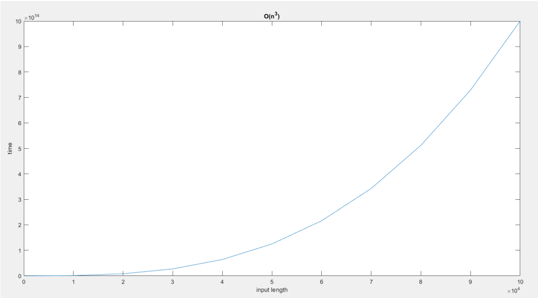
In this approach we have used temp array, where it's size is equal to number of columns.Therefore space complexity is O(m)

0.5 Graph analysis

Brute force
In the brute force approach as n increases, the time increases by n^6.



Dynamic programming
In the Dynamic programming approach as n increases, the time increases by n^3.



0.6 Conclusion

From the experimental study we concluded that the average running time of dynamic algorithm is best, which can be observed from the mutual graph of dynamic algorithm and Brute Force algorithm as shown.

0.7 References

https://www.geeksforgeeks.org/maximum-sum-rectangle-in-a-2d-matrix-dp-27/
https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

0.8 code

https://ideone.com/W0rVpg
https://ideone.com/Zw0Ape