# A05G17Q17: Max sum subarray in 2-D array

IIT2018049-Harsh Kochar, IIT2018050-Avneesh Gautam, IIT2018051-Avishek

*IV Semester B.Tech, Information Technology,*
*Indian Institute of Information Technology, Allahabad, India*

***Abstract: In this paper, we have devised an algorithm to find the maximum sum subarray from the given 2-D array.***
***We have discussed the time complexity of the algorithm by both Apriori and Aposteriori analysis.***
***Key terms – subarray, kadane, contiguous***

## I. INTRODUCTION

In computer science, the maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum. Some properties of this problem are:
1. If the array contains all non-negative numbers, then the problem is trivial; a maximum subarray is the entire array.
2. If the array contains all non-positive numbers, then a solution is any subarray of size 1 containing the maximal value of the array.
3. Several different sub-arrays may have the same maximum sum.
The naive solution for this problem is to check every possible rectangle in the given 2D array. This solution requires 4 nested loops and time complexity of this solution would be $O(n^4)$.
Kadane's algorithm for 1D array can be used to reduce the time complexity. Simple idea of Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of the maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum, compare it with max_so_far and update max_so_far if it is greater than max_so_far.

## II. ALGORITHM DESCRIPTION

We will use Kadane's algorithm to solve the problem. The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We will basically find the top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate the sum of elements in every row from left to right and store these sums in an array, say temp[]. So temp[i] indicates the sum of elements from left to right in row 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 left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far. Hence, we will get the maximum sum subarray of the 2-D array using Kadane's algorithm.

## III. ILLUSTRATION

Here is the illustration for the designed algorithm. Let us use a small example to understand the algorithm.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | | 2 | left | 0 |
| 0 | 6 | 3 | 4 | 1 | | 0 | right | 0 |
| 2 | -2 | -1 | 4 | -5 | | 2 | sum | 5 |
| -3 | 3 | 1 | 0 | 3 | | -3 | maxsum | 5 |

finalleft = 0, finalright = 0, finaltop = 0,finaldown = 2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | | 3 | left | 0 |
| 0 | 6 | 3 | 4 | 1 | | 6 | right | 1 |
| 2 | -2 | -1 | 4 | -5 | | 0 | sum | 9 |
| -3 | 3 | 1 | 0 | 3 | | 0 | maxsum | 9 |

finalleft = 0, finalright = 1, finaltop = 0,finaldown = 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | 0 | left | 0 |
| 0 | 6 | 3 | 4 | 1 | 9 | right | 2 |
| 2 | -2 | -1 | 4 | -5 | -1 | sum | 9 |
| -3 | 3 | 1 | 0 | 3 | 1 | maxsum | 9 |

finalleft = 0, finalright = 1, finaltop = 0,finaldown = 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | -4 | left | 0 |
| 0 | 6 | 3 | 4 | 1 | 13 | right | 3 |
| 2 | -2 | -1 | 4 | -5 | 3 | sum | 17 |
| -3 | 3 | 1 | 0 | 3 | 1 | maxsum | 17 |

finalleft = 0, finalright = 3, finaltop = 1,finaldown = 3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | 0 | left | 0 |
| 0 | 6 | 3 | 4 | 1 | 14 | right | 4 |
| 2 | -2 | -1 | 4 | -5 | -2 | sum | 17 |
| -3 | 3 | 1 | 0 | 3 | 4 | maxsum | 17 |

finalleft = 0, finalright = 3, finaltop = 1, finaldown = 3

In every iteration,we apply kadane's algorithm and then compare it to the max sum.
We will follow the same procedure for the next 3 rows and will get the final positions of left, right, top, bottom and maxsum. Following the same, we will get the following couloured rectangle giving the maxsum in 2-D array.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 1 | -3 | -4 | 5 | 5 | left | 4 |
| 0 | 6 | 3 | 4 | 1 | 1 | right | 4 |
| 2 | -2 | -1 | 4 | -5 | -5 | sum | 6 |
| -3 | 3 | 1 | 0 | 3 | 3 | maxsum | 18 |

finalleft = 1, finalright = 3, finaltop = 1,finaldown = 3

## IV. ALGORITHM AND ANALYSIS

**Algorithm:**Max sum subarray in 2-D array
**Input:**2-D array elements
**Output:**Maximum sum

**Algorithm 1:** kadane(int* a,int* start,int* finish,int n)

```
1.  sum ← 0
2.  maxSum ← INT_MIN
3.  *finish ← -1
4.   local_start ← 0
5.  for i ← 0 to n
6.          sum ← sum + a[i]
7.          if sum < 0 then
8.                  sum ← 0
9.                  local_start ← i + 1
10.         else if sum > maxSum then
11.                 maxSum ← sum
12.                 *start ← local_start
13.                 *finish ← i
14.         i++
15. if *finish != -1 then
16.         return maxSum
17. maxSum ← a[0]
18. *start ← *finish ← 0
19. for i ← 1 to n
20.         if a[i] > maxSum then
21.                 maxSum ← a[i]
22.                 *start ← *finish ← i
23.         i++
24. return maxSum
```

**Algorithm2:** max_Sum(int arr[][], int row, int col)

```
1.  maxSum ← INT_MIN
2.  integer temp[row]
3.  for left ← 0 to col
4.      memset(temp, 0, sizeof(temp))
5.      for right ← left to col
6.          for i ← 0 to row
7.              temp[i] ← temp[i]+arr[i][right]
8.              i++
```

```
9.          sum ← kadane(temp,&start,&finish,
                         row)
10.         if sum > maxSum then
11.             maxSum ← sum
12.             finalLeft ← left
13.             finalRight ← right
14.             finalTop ← start
15.             finalBottom ← finish
16.         right++
17.     left++
18. print "maxSum"
```

APRIORI ANALYSIS

| Step | Time | Iteration |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 3 | 1 | 1 |
| 4 | 1 | 1 |
| 5 | 2 | n |
| 6 | 2 | n |
| 7 | 1 | n |
| 8 | 1 | n |
| 9 | 2 | n |
| 10 | 1 | n |
| 11 | 1 | n |
| 12 | 1 | n |
| 13 | 1 | n |
| 14 | 2 | n |
| 15 | 1 | 1 |
| 16 | 1 | 1 |
| 17 | 1 | 1 |
| 18 | 2 | 1 |
| 19 | 2 | n-1 |
| 20 | 1 | n-1 |
| 21 | 1 | n-1 |
| 22 | 1 | n-1 |
| 23 | 2 | n-1 |
| 24 | 1 | 1 |

Table1: Complexity analysis of kadane() function.

$$T = (1*1)+(1*1)+(1*1)+(1*1)+(2*n)+(2*n)+(1*n)+(1*n)+(2*n)+(1*n)+(1*n)+(1*n)+(2*n)+(1*1)+(1*1)+(1*1)+(2*1)+(2*(n-1))+(1*(n-1))+(1*(n-1))+(1*(n-1))+(2*(n-1))+(1*1)$$

$$T = 4+14n+5+7(n-1)+1$$

$$T = 14n+7n+10-7$$

$$T = 21n+3$$

$$T \ \alpha \ O(n)$$

Taking Column size `m` and Row size `n`.

| Step | Time | Iteration |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 3 | 2 | m |
| 4 | 1 | m*n |
| 5 | 2 | m*m |
| 6 | 2 | m*m*n |
| 7 | 2 | m*m*n |
| 8 | 2 | m*m*n |
| 9 | n | m*n |
| 10 | 1 | m*m |
| 11 | 1 | m*m |
| 12 | 1 | m*m |
| 13 | 1 | m*m |
| 14 | 1 | m*m |
| 15 | 1 | m*m |
| 16 | 1 | m*m |
| 17 | 1 | m |
| 18 | 1 | 1 |

Table1: Complexity analysis of maxSum() function.

$$T = 2*(1*1)+(2*m)+(1*m*n)+(2*m*m)+(2*m*m*n)+(2*m*m*n)+(2*m*m*n)+(n*m*n)+(1*m*m)+(1*m*m)+(1*m*m)+(1*m*m)+(1*m*m)+(1*m*m)+(1*m)+(1*1)$$

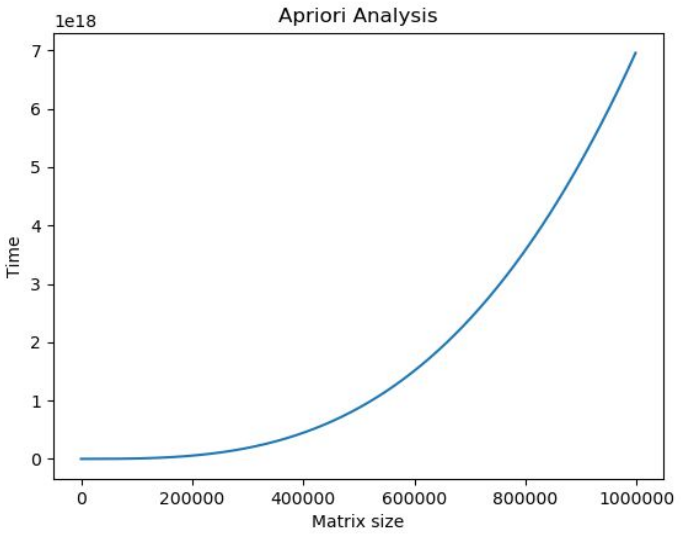
$$T = 2+2m+mn+2m^2+6m^2n+n^2m+7m^2+m+1$$

$$T = 6m^2n+n^2m+9m^2+mn+3m+3$$

$$T \ \alpha \ O(n^2m+m^2n)$$

If we take square matrix of order n,

$$T = 7n^3+10n^2+3n+3$$




$T \alpha O(n^3)$

Figure1:Comparison of Time complexity for Apriori analysis of Algorithm.

## V. EXPERIMENTAL ANALYSIS AND PROFILING

The program was run against various test-cases and the time required for each test-case was collected and then plotted against the size of the array.

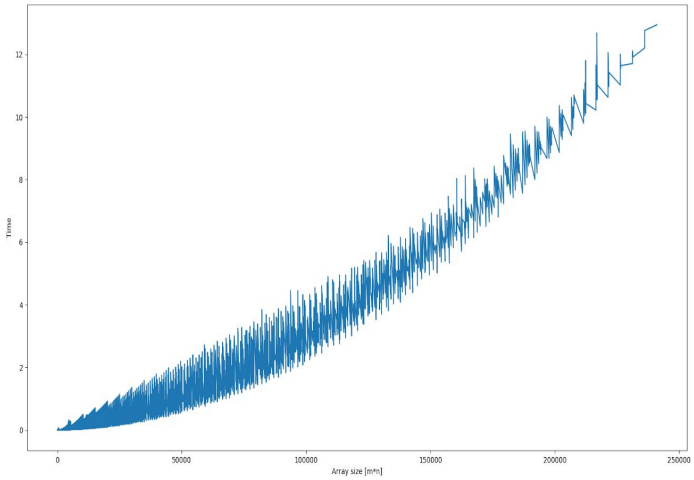


Figure2: Time complexity graph for Aposteriori analysis.

The above graph shows that the Apriori analysis is consistent with the Aposteriori analysis as the theoretical graph and the experimental graph has the same nature and hence we could say that the analysis made is true to a certain extent.

In this graph we have varied n,m from 1 to 500 keeping an interval of 10. The x axis represents n*m, wherein y axis represents time. The dataset was made from randomly generated numbers using python library random. Their values were in the range of 1 to 10,00,000. Some times same x can have more than 2 values in y because x represents n*m.

eg: for n=200,m=200 and n=1,m=400 and n=400, m=1. And time will differ significantly.

## VI. CONCLUSION

From this paper we can conclude that the time complexity of Kadane's algorithm is $O(n)$ resulting in the final algorithm to be $O(n^2m + m^2n)$ algorithm.

## VII. REFERENCES

[1.]Introduction to Algorithms by Tas H. Cormen, Charles E. Leiserson, Ronald L. Rivest.

[2.]https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

[3.]https://www.geeksforgeeks.org/merge-sort/https://www.geeksforgeeks.org/largest-sum-contiguous-subarry/

[4.]https://www.geeksforgeeks.org/maximum-sum-rectangle-in-a-2d-matrix-dp-27/