COT 5405 ANALYSIS OF ALGORITHMS

PROGRAMMING PROJECT

Spring 2022

Team Members	
Mandar Palkar	2140-6740
Siddhi Wadgaonkar	9544-2212

CONTENTS

INTRODUCTION

After understanding the requirements provided in the Problem Statement, we analyzed and deduced that the solution to the given problem can be found using the Maximum-Subarray for Problem 1. Then this solution can be extended to find maximum sub-matrix in 2-D.

We have discussed each approach and implementation in the below sections, followed by the comparative analysis of each of the implementations.

Steps to run

- Navigate to source folder
- Execute "`make`" command
- Example: `java –jar MarsBase.jar 3b < input1.txt`

DESIGN AND ANALYSIS OF ALGORITHMS

PROBLEM 1

 # ALG1 – Brute Force

Pseudo-code

```
MAXIMIZE-AIR-QUALITY-INDEX(A)
1    n = A.Length
2    maxSumSoFar = -INF
3    for left = 0 to n-1
4         for right = left to n – 1
5              currSum = 0
6              for temp = left to right
7                   currSum = currSum + A[temp]
8              if currSum > maxSumSoFar
9                   maxSumSoFar = currSum
10                  L = left
11                  R = right
12   return maxSumSoFar, L, R
```

Proof of Correctness

Loop Invariant:
At the start of each iteration an array A[0..n-1] of the for loop which spans from line 3 to 11, the `maxSumSoFar` contains the maximum sum of a subarray till A[0...left - 1].

Initialization:
At the start of the iteration of the loop both the left and right variables are initialized to 0. Hence the subarray with maximum sum will be the first element itself in the very first iteration.

Maintenance:
At every step we are calculating sum of every possible subarray of air-indices using the nested loops on line 3 to 6. We are comparing the sum with previous maximum we have found and replacing the `maxSumSoFar` with sum calculated in the current iteration. Hence, at every step `maxSumSoFar` has the maximum sum upto that index.

Termination:
The program will terminate when left = n, i.e. when the last element is evaluated. So as per the loop invariant will have sum of maximum subarray for A[0..n-1] in `maxSumSoFar.`

Time Complexity

As we can see there are three nested loops in the algorithm. First two loops are to calculate all possible subsequences and the third loop is to calculate the sum of each subsequence. This gives us a time complexity of $O(n^3)$.

Space Complexity

Since we do not maintain any previous state here, the auxiliary space complexity is O(1).

 # ALG2 – Dynamic Programming ($O(n^2)$)

Pseudo-code

```
MAXIMIZE-AIR-QUALITY-INDEX(A)
1    maxSumSoFar = -INF
2    n = A.Length
3    temp = 0
4    Array DP[0…n+1]
5    DP[0] = 0
6    L = -1
7    R = -1
8    for i = 1 to n
9         DP[i] = DP[i-1] + A[i-1]
10   for left= 1 to n
11       for right= left to n
12           temp = DP[right] - DP[left-1]
13   if temp> maxSumSoFar
14         maxSumSoFar = currSum
15         L = left - 1
16         R = right - 1
17   return maxSumSoFar, L, R
```

Proof of Correctness

Loop Invariant:

At every iteration of the loop 11-14 the `maxSumSoFar` contains the maximum sum for subarray that exists in `A[0..left-1]`, whereas temp keep tracks of sum of the elements in the subarray of `A[left…right]`.

Initialization:

For Array `A[0..n-1]` , every element at index `i+1` in array `DP[0...n]` contains sum of all the elements in the subarray `A[0..i]`. Hence for two distinct indexes `i & j`, `DP[j]-DP[i-1]` will give sum of all the elements for subarray `A[i..j].`

Initially the subarray is empty hence this sum is 0 hence `DP[0]` is explicitly set to zero.

Maintenance:

At every loop iteration the we calculate sum for every subarray combination that exists in A[left...right] as follows

`F[left,i] = DP[i] - DP[left-1] .....` $i \in$ [left, right]

we compare the sum calculated with maximum sum calculated so far and replace the maximum sum with the current sum if current sum is greater.

Termination:
The program terminates when `left = n`, and as per the loop invariant we have sum of maximum subarray till `A[0..n-1]`

Time Complexity
This is similar to ALG1, except that there is no third loop to calculate the sum. Hence the time complexity is **$O(n^2)$.**

Space Complexity
The DP array of size n makes the space complexity of this algorithm **O(n).**

 # ALG3 TASK 3A – Dynamic Programming (O(n))

Pseudo-code

```
HELPER(DP,index)
1     if index >= A.Length
2          return 0
3     if DP[index] > 0
4          return A[index]
5     curr = DP[index]
6     sum = DP[index]+ helper(A, index+1)
7     DP[index] = max(curr, sum)
8     return DP[index]

MAXIMIZE-AIR-QUALITY-INDEX(A)
1     n = A.Length
2     maxSumSoFar = -INF
3     x = -INF
4     s=0
5     e=0
6     DP [0…n+1]
7     for i = 0 to n
8         x = HELPER(DP,i)
9         if x > maxSumSoFar
10            maxSumSoFar = x
11            s = i
12    x = maxSumSoFar
13    e = s
14    While x > 0
15        x = x - A[e]
16        if x == 0
17             Break
18        e = e + 1
19
20    return maxSumSoFar, L, R
```

Time Complexity

As we are iterating through the array only once, and no other computation is larger, the running time complexity of this algorithm is **O(n).**

Space Complexity

The DP array definition gives us a space complexity of **O(n)**.

 # ALG3 TASK3B – Dynamic Programming (O(n))

Pseudo-code

```
MAXIMIZE-AIR-QUALITY-INDEX(A)
1    maxSumSoFar = -INF
2    n = A.Length
3    L = -1
4    R = -1
5    currSum = 0
6    currStart = 0
7    for i = 0 to n
8        currSum = currSum + A[i]
9        if maxSumSoFar < currSum
10           maxSumSoFar = currSum
11           l = currStart
12           r = i
13       if currSum < 0
14           currSum = 0
15           currStart = i+1
16   return maxSumSoFar, L, R
```

Let us assume that the maximum sum of the subarray up to (i-1) position is S(i-1).

To find the maximum sum up to (i)th position, we try to add the S(i-1) to the (i)th element only if the sum until (i-1) was positive i.e. greater than 0. Adding a negative pre-sum will always reduce the overall sum, and since we have to maximum our sum, we only consider the previous sum if it is adding positively to our upcoming sum.

Based on the above theory, the recurrence relation is:

S(i) = A(i) + S(i-1), if S(i-1) > 0 else A(i)

Proof of Correctness

Initialization:
During the Initialization, i.e. the first iteration of the loop, we consider an array that ends at position 1. Since this will be the first element, the value of the first element will become the maximum value until this iteration.

Maintenance:

In each step, we check if the maximum value so far is positive, and if yes we add it to the current element. Else, we keep the sum as only the current value. That ensures the current maximum value is maximum so far. Hence, at every step `maxSumSoFar` has the maximum sum up to that index.

Termination:
The program will terminate when i = n, i.e. when the last element is evaluated. So as per the loop invariant will have sum of maximum subarray for A[0..n-1] in `maxSumSoFar.`

Time Complexity
As we are iterating through the array only once, and no other computation is larger, the running time complexity of this algorithm is **O(n).**

Space Complexity
There is no additional space required as only one value is maintained, making the space complexity of the algorithm to be **O(1)**.

 # ALG4 – Brute Force ($O(n^6)$)

Pseudo-code

```
MAXIMIZE-AIR-QUALITY-INDEX(MAT)
1     maxSumSoFar = -INF
2     rows = MAT.Length
3     cols = MAT[0].Length
4     xLeft = -1
5     yLeft = -1
6     xRight = -1
7     yRight = -1
8
9     for r1 = 0 to rows
10         for c1 = 0 to cols
11             for r2 = r1 to rows
12                 for c2 = c1 to cols
13                     currSum = 0
14                     for m = r1 to r2
15                         for n = c1 to c2
16                             currSum += MAT[m,n]
17                     if maxSumSoFar < currSum
18                         maxSumSoFar = currSum
19                         xLeft = r1
20                         yLeft = c1
21                         xRight = r2
22                         yRight = c2
23    return maxSumSoFar, xLeft, yLeft, xRight, yRight
```

Proof of Correctness

Loop Invariant:
At the start of each iteration an array `MAT[0..n-1][0..m-1]` of the for loop which starts from line 9, the `maxSumSoFar` contains the maximum sum of a subarray computed so far.

Initialization:
At the start of the iteration of the loop all the coordinate variables r1, c1, r2, c2 are initialized to 0. Hence the submatrix with maximum sum will be the first element i.e. `MAT[0][0].`

Maintenance:
At every step we are calculating sum of every possible submatrix of air-indices using the nested loops on line 9 to 12. We are comparing the sum with previous maximum we have found and replacing the `maxSumSoFar` with sum calculated in the current iteration. Hence, at every step `maxSumSoFar` has the maximum sum from MAT[r1][c1] (top left) MAT[r2][c2] (bottom right).

Termination:
The program will terminate when r1 = n and c1 = m, i.e. when the last element is evaluated. So as per the loop invariant will have sum of maximum subarray for MAT[0..n-1][0..m-1] in `maxSumSoFar.`

Time Complexity

As we can see there are six nested loops in the algorithm. First four loops are to calculate all possible submatrices and the fifth and sixth loop are to calculate the sum of each submatrix defined by the first four loops. This gives us a time complexity of **$O(n^6)$.**

Space Complexity

Since we do not maintain any previous state here, the auxiliary space complexity is **O(1).**

ALG5 – DYNAMIC PROGRAMMING ($O(n^4)$)

Pseudo-code

```
MAXIMIZE-AIR-QUALITY-INDEX(MAT)
1     rows = MAT.length
2     cols = MAT[0].length
3     SUB [0…rows, 0..cols]
4     xLeft = -1
5     yLeft = -1
6     xRight = -1
7     yRight = -1
8     maxSumSoFar = -INF
9
10    for r1 = 0 to rows
11       For r2 = 0 to cols
12          if r1 == 0 || r2 == 0
13             SUB[r1,r2] = 0
14           else
15             SUB[r1,r2] = SUB[r1-1,r2] + SUB[r1,r2-1]
                                  -SUB[r1-1,r2-1]+SUB[r1-1,r2-1]
16
17    for r1 = 0 to rows
18      for r2 = r1 to rows
19         for c1 = 0 to cols
20             for c2 = c1 to cols
21              submatrix_sum = SUB[r2+1,c2+1] - SUB[r2+1,c1]-
      SUB[r1,c2+1]+ sub[r1,c1]
22                If submatrix_sum > maxSumSoFar
23                   maxSumSoFar = submatrix_sum
24                   xLeft = r1
25                   yLeft = c1
26                   xRight = r2
27                   yRight = c2
24    return maxSumSoFar, xLeft, yLeft, xRight, yRight
```

Proof of Correctness

Loop Invariant:
At every iteration of the loop 11-14 the `maxSumSoFar` contains the maximum sum for submatrix from r1, c2, r2, c2, whereas `temp` keeps track of sum of the elements in the subarray of `MAT[r1][c1] to MAT[r2][c2]`.

Initialization:
At the start of the iteration of the loop all the coordinate variables r1, c1, r2, c2 are initialized to 0. Hence the submatrix with maximum sum will be the first element i.e. `MAT[0][0].`

Maintenance:
At every loop iteration the we calculate sum for every submatrix combination that exists in `MAT[r1,c1]` to `MAT[r2,c2],` we compare the sum calculated with maximum sum calculated so far and replace the maximum sum with the current sum if current sum is greater.

Termination:
The program terminates when `MAT[n-1][m-1] is evaluated`, and as per the loop invariant we have sum of maximum subarray till `MAT[n-1][m-1].`

Time Complexity
There are 4 nested loops to iterate through every row and column to find all possible submatrices. The last two loops in ALG4 are optimized here. Hence the time complexity is **$O(n^4)$.**

Space Complexity
The SUB array of size m x n makes the space complexity of this algorithm **O(m x n).**

ALG6 – DYNAMIC PROGRAMMING ($O(n^3)$)

Pseudo-code

HELPER-ALG3B(A)

```
1     maxSumSoFar = -INF
2     n = A.Length
3     currSum = 0
4     currStart = 0
5     for i = 0 to n
6         currSum = currSum + A[i]
7         if maxSumSoFar < currSum
8             maxSumSoFar = currSum
9             l = currStart
10            r = i
11        if currSum < 0
12            currSum = 0
13            currStart = i+1
14    return maxSumSoFar
```

MAXIMIZE-AIR-QUALITY-INDEX(MAT)

```
1     maxSumSoFar = -INF
2     rows = MAT.Length
3     cols = MAT[0].Length
4     prefix [0…rows+1,0…cols+1]
5     xLeft = -1
6     yLeft = -1
7     xRight = -1
8     yRight = -1
9     tt = -1
10    rr = -1
11    tt = -1
12    bb = -1
13
14    for i = 0 to rows
15         for j = 0 to cols
16            if j == 0
17               prefix[i,j] = MAT[i,j]
18            else
19               prefix[i,j] = prefix[i,j] + prefix[i,j-1]
20
21    maxSum = -INF
22
23    for i = 0 to n
24        for j = i to n
25            V[0…rows]
```

```
26            for k = 0 to m
27                ele = 0
28              if i == 0
29                 ele = prefix[k,j]
30              else
31                 ele = prefix[k,j] – prefix[k,i-1]
32              V[k] = ele
33
34            maxTempSum = HELPER-ALG3B(V)
35
36            if maxSum < maxTempSum
37                maxSum = temp
38                tt = i
39                bb = j
40                xLeft = ll
41                xRight = rr
42
43    yLeft = tt
44    yRight = bb
45    return maxSum, xLeft, yLeft, xRight, yRight
```

Proof of Correctness

Prefix Calculation

`prefix[i,j] = prefix[i,j] + prefix[i,j-1] if j != 0 else prefix[i,j] = MAT[i,j]`

Initialization

Before the initialization task we compute the prefix matrix that computes and stores the prefix sum of all rows in `prefix[0…n-1][0..m-1].` In the internal loop, initially an auxiliary array stored the maximum sum for each row of current submatrix.

Maintenance

At every step, for all combinations of starting and ending indices of columns of submatrix, if the value at current position is greater than 0, then the current sum is updated else it is kept as the value at position itself, line 28-31. This breaks down the problem from 2-D to 1-D. After following the same for all submatrices, the algorithm implemented in Task 3b is used to compute the sum for all sub tasks. `maxSumSoFar` will be updated in case a new maximum is found in any of the combinations.

Termination

The algorithm terminates after all combinations of columns are computed and `maxSumSoFar` contains the answer.

Time Complexity

As we iterate through two loops for prefix sum calculation, it gives us $O(n^2)$. Then the three nested loops that divide the tasks give us $O(n^3)$. Since the Kadane's algorithm to solve the subproblem is O(n), by adding all the complexities, we get the worst-case time complexity of this ALG6 to be **$O(n^3)$.**

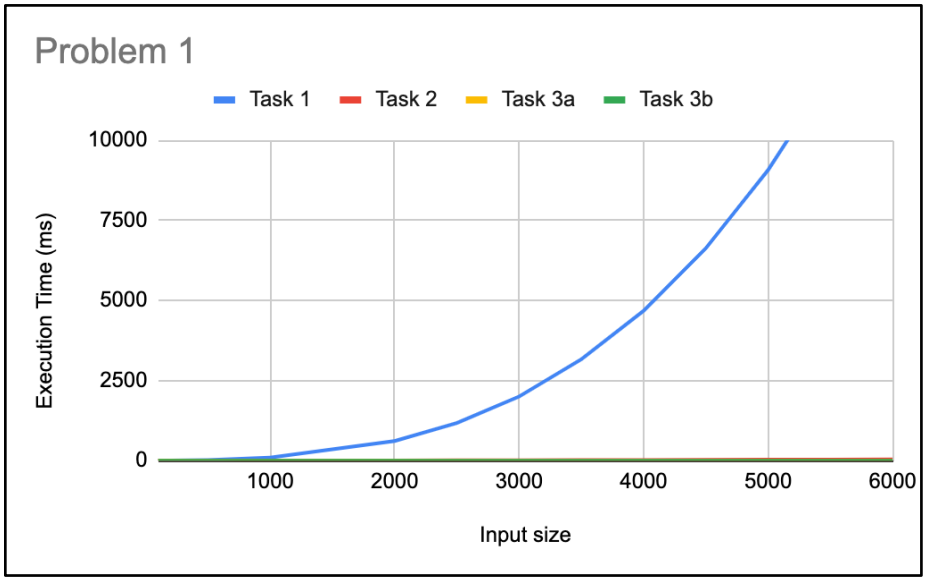
Space Complexity

The prefix matrix gives us the space complexity of **O(m x n).**

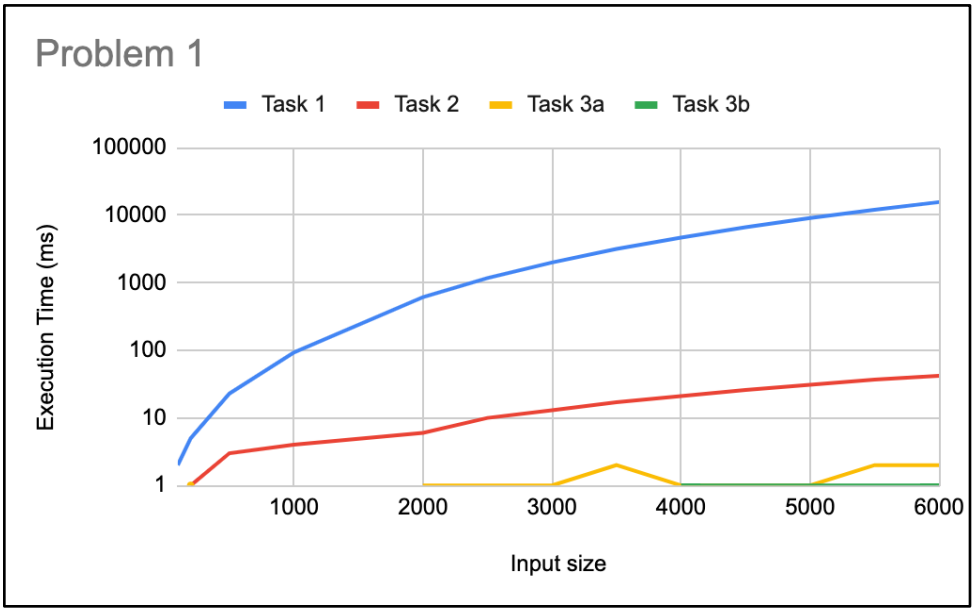
EXPERIMENTAL COMPARATIVE STUDY

Problem 1

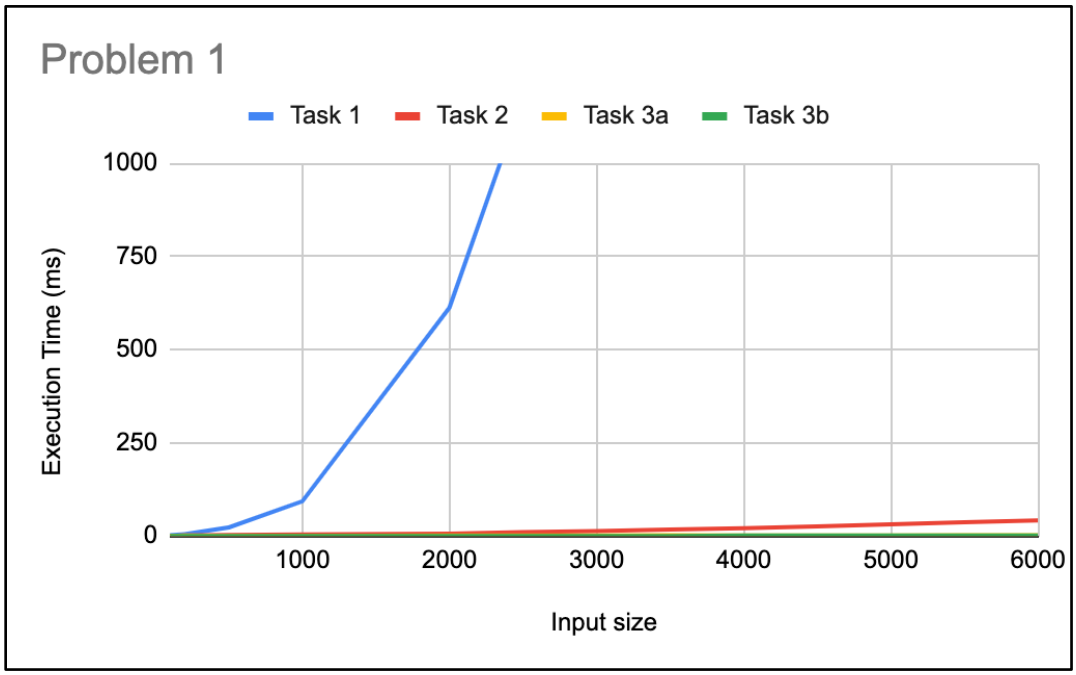
For Problem 1, we executed our implementations for the 4 tasks with the following inputs and noted the execution time values in milliseconds. Below is the graph visualization of the execution time values for different input sizes.

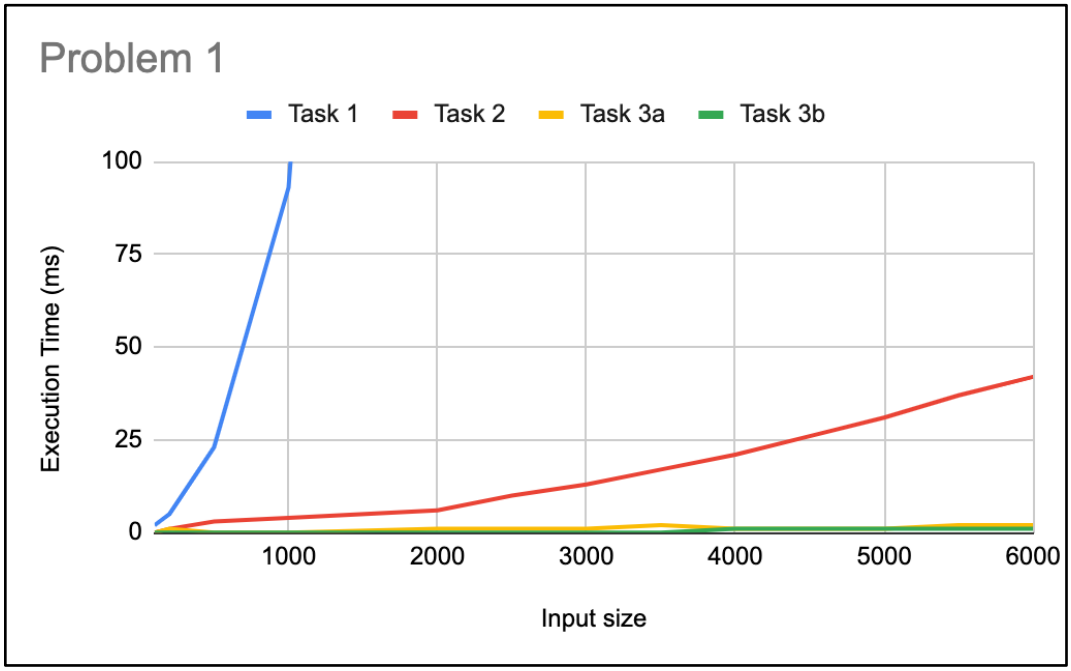


For Task 1, as the input size increases, we can clearly see the graph trend like the $O(n^3)$ graph. Below is the logarithmic representation of the graph to get a better understanding.

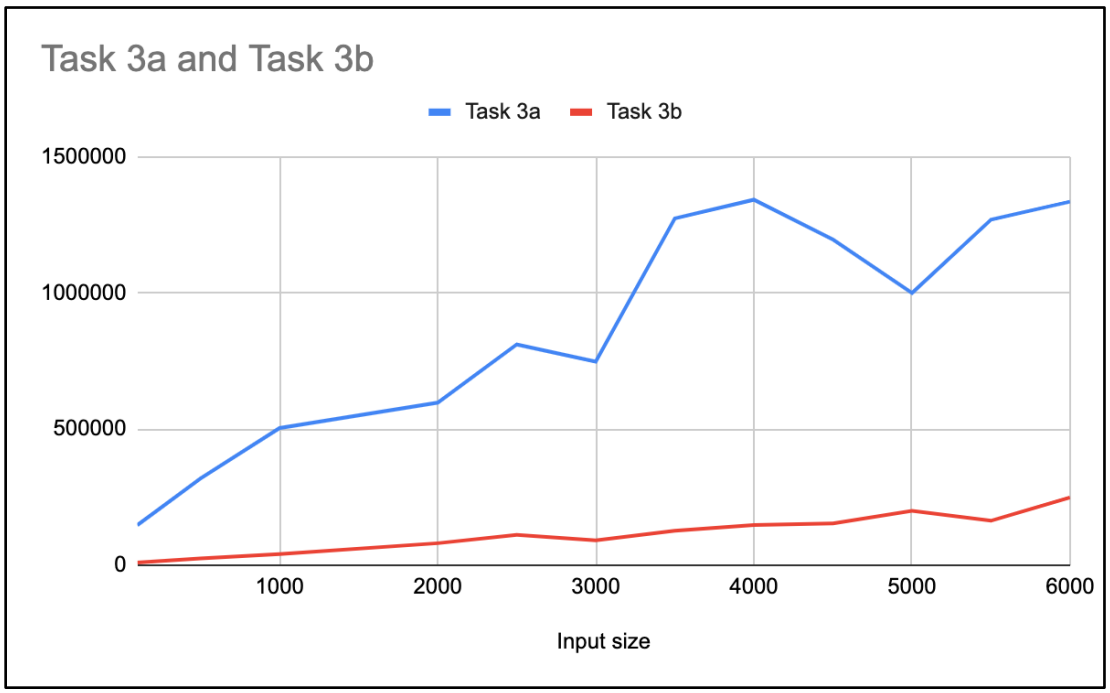


Below are two more snapshots of the same graph zoomed out in order to understand the relative difference of Task 1, Task 2 and Task 3 time.





Task 2 running at $O(n^2)$ will gradually have an increasing curve and be above both the Task 3 algorithms.

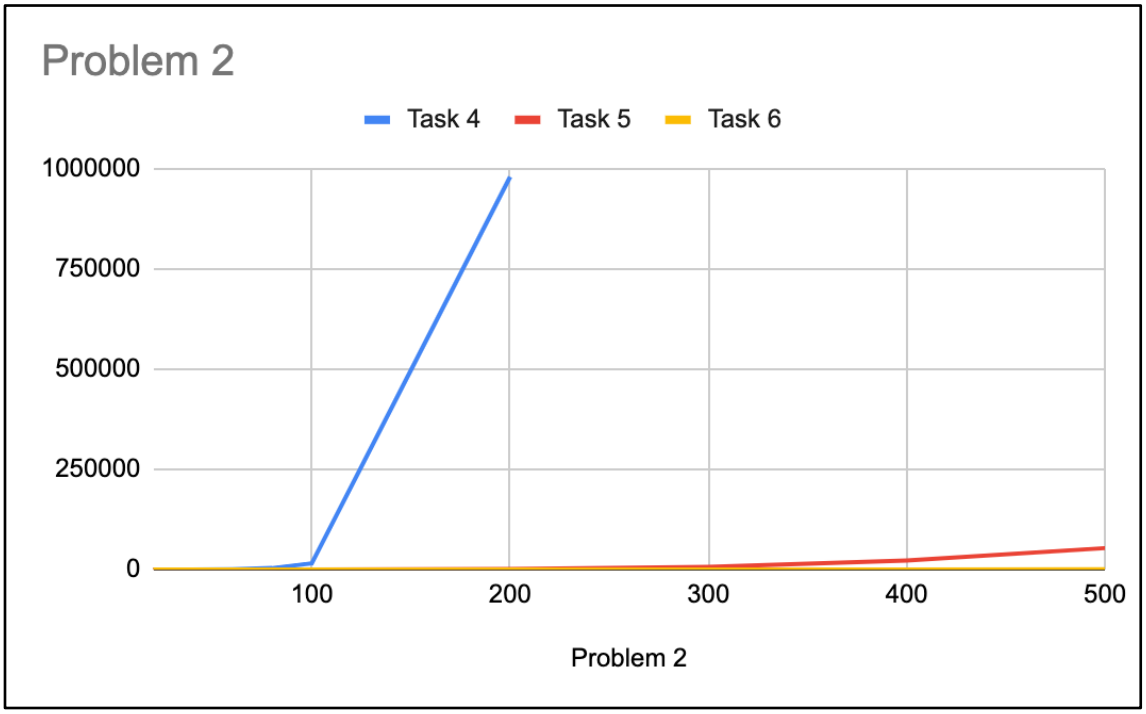


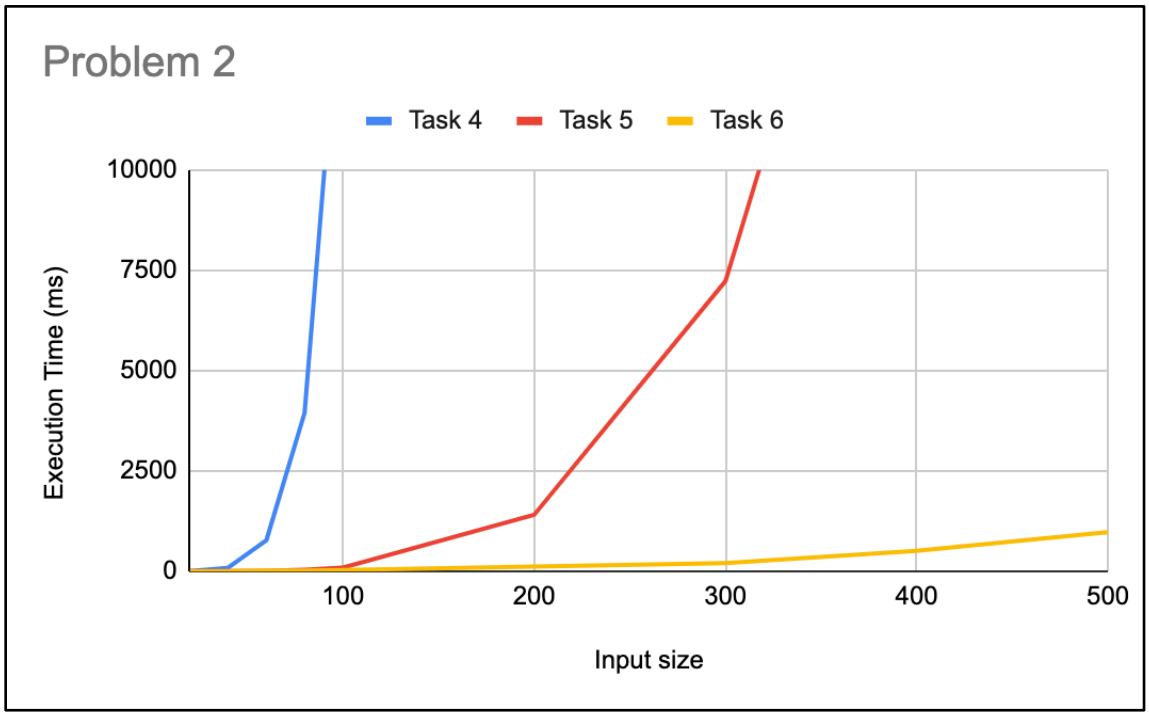
Both Task 3a and Task 3b implementation of the Kadane's algorithm using Dynamic Programming have the running time of O(n). However, we can see that the Task 3a implementation takes a longer time than 3b. This is essentially because of the *recursive* implementation of the same algorithm, that adds an extra overhead of internal function calling and stack handling. However, it is still O(n) and hence the graphs run very closely to the 3b plot.

The Task 3b (Kadane's) algorithm proves to be the fastest in terms of execution time and has a time complexity of O(n).

Problem 2

Similar plot for Problem 2 is below, with Task 4, Task 5 and Task 6 and their running times.



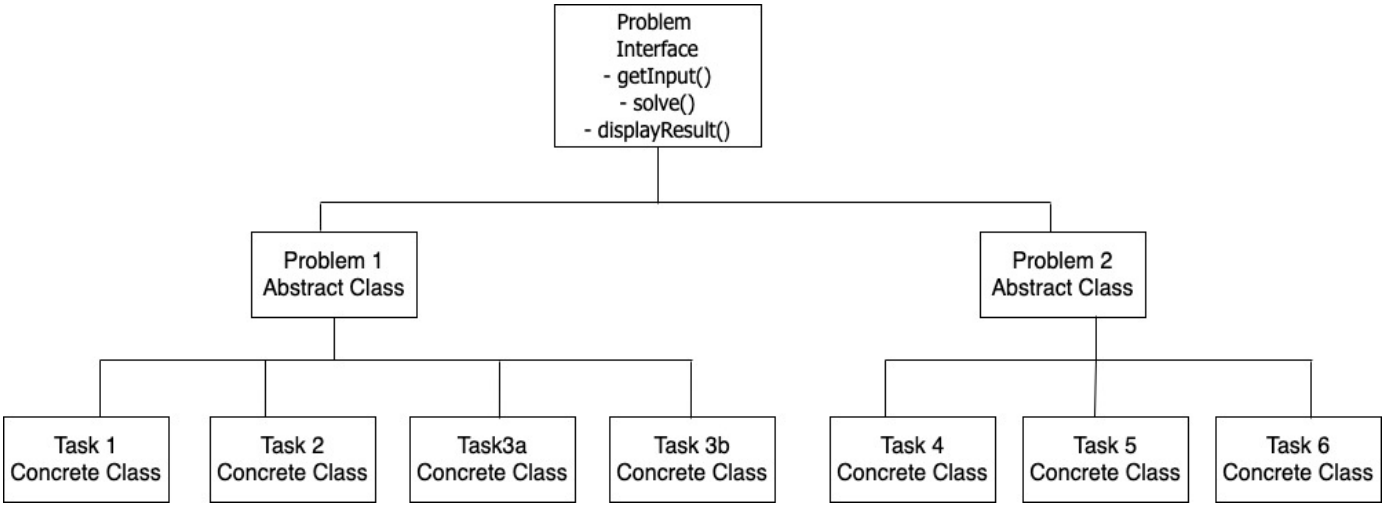


The Dynamic Programming approach in Task 6 (using Task 3b) proves to be the fastest with $O(n^3)$. However, the trend is not visible for these readings as they are too small.

OBSERVATIONS AND CONCLUSION

Ease Of Implementation -

- Because of the means of Implementing Inheritance through Abstract Classes and Interfaces the Java Implementation is Pretty Straightforward. Every class implements the following methods of Problem.java Interface i.e.
  1. getInput() - Gets the input for the task
  2. solve() - Contains the actual algorithm for the particular task
  3. displayResult() - Displays output of the task to the console.
- The appropriate class implementation (i.e. Task1.java, Task2.java, ... Task 6.java etc.) is chosen with the help of Polymorphism, based on the command line argument that is supplied while executing the program and then the above-mentioned methods are executed one after the other.
- For example: for Task1, Task2 & Task3 the getInput() method is designed to take only array as input. Whereas getInput() method for Task4, Task5 & Task6 the get Input() method is designed to read a matrix from standard input. All this has been achieved while keeping the method signature same which allowed us to abstract the details of the implementation.
- Moreover, the data structures and custom variables needed for each task has been isolated in their respective classes, rather than keeping them at some global location or class.
- The programs mostly use primitive data structures like Arrays & 2 Dimensional Arrays, along with some Complex Data Structures like Vectors at few places.
- We also have included the scripts we used to generate our test inputs.



Observations -

After implementing solutions for the given tasks through different paradigms, we can observe that:

- Brute Force Implementations are although easy to implement as well as understand. They also consumed considerably less space as compared to the other tasks that we implemented.
- However, the big downside of this approach is that these programs were highly inefficient. These programs took 10-100 times more time to solve the same problem than the other algorithms. This time difference was evidently visible for large input sizes as described in the experimental studies.
- DP Implementations are somewhat complex and less memory efficient than the brute force algorithm. As their implementation was aided by another data structure of similar or little, larger size than the actual input. However, the upside of this implementation was that their runtime was considerably lower than their brute force counterparts.
- The recursive implementation of the DP is multiple times faster than the brute force even after the overhead of stack maintenance it requires.