**COT 5405 ANALYSIS OF ALGORITHMS**

**PROGRAMMING PROJECT**

**Spring 2022**

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

* Problem Definition
* Design and Analysis of Algorithms

(For each alg)

* Algorithm
* Correctness
* Mathematical Recursive Formulation expressing optimal substructure property
* Time Complexity
* Space Complexity
* Experimental Comparative Study
* Graphs etc.
* Conclusion

(For each prog task)

* Ease of implementation
* Other potential technical challenges

**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(n3).

**Space Complexity**

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

**ALG2 – Dynamic Programming ( O(n2) )**

**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 [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(n2).**

**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
20. return maxSumSoFar, L, R

**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(n6) )**

**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. for r1 = 0 to rows
9. for c1 = 0 to cols
10. for r2 = r1 to rows
11. for c2 = c1 to cols
12. currSum = 0
13. for m = r1 to r2
14. for n = c1 to c2
15. currSum += MAT[m,n]
16. if maxSumSoFar < currSum
17. maxSumSoFar = currSum
18. xLeft = r1
19. yLeft = c1
20. xRight = r2
21. yRight = c2
22. 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(n6).**

**Space Complexity**

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

**ALG5 – DYNAMIC PROGRAMMING ( O(n4) )**

**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. for r1 = 0 to rows
10. For r2 = 0 to cols
11. if r1 == 0 || r2 == 0
12. SUB[r1,r2] = 0
13. else
14. SUB[r1,r2] = SUB[r1-1,r2] + SUB[r1,r2-1]

-SUB[r1-1,r2-1]+SUB[r1-1,r2-1]

2. for r1 = 0 to rows
3. for r2 = r1 to rows
4. for c1 = 0 to cols
5. for c2 = c1 to cols
6. submatrix\_sum = SUB[r2+1,c2+1] - SUB[r2+1,c1]- SUB[r1,c2+1]+ sub[r1,c1]
7. If submatrix\_sum > maxSumSoFar
8. maxSumSoFar = submatrix\_sum
9. xLeft = r1
10. yLeft = c1
11. xRight = r2
12. yRight = c2
13. 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(n4).**

**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(n3) )**

**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. for i = 0 to rows
14. for j = 0 to cols
15. if j == 0
16. prefix[i,j] = MAT[i,j]
17. else
18. prefix[i,j] = prefix[i,j] + prefix[i,j-1]
20. maxSum = -INF
21. for i = 0 to n
22. for j = i to n
23. V[0…rows]
24. for k = 0 to m
25. ele = 0
26. if i == 0
27. ele = prefix[k,j]
28. else
29. ele = prefix[k,j] – prefix[k,i-1]
30. V[k] = ele
32. maxTempSum = HELPER-ALG3B(V)
34. If maxSum < maxTempSum
35. maxSum = temp
36. tt = i
37. bb = j
38. xLeft = ll
39. xRight = rr
41. yLeft = tt
42. yRight = bb
43. return maxSum, xLeft, yLeft, xRight, yRight

**Proof of Correctness**

**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.

Chart, line chart

Description automatically generated

For Task 1, as the input size increases, we can clearly see the graph trend like the O(n3) graph. Below is the logarithmic representation of the graph to get a better understanding.

Chart, line chart

Description automatically generated

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.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Task 2 running at O(n2) will gradually have an increasing curve and be above both the Task 3 algorithms.

Chart, line chart

Description automatically generated

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.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

The Dynamic Programming approach in Task 6 (using Task 3b) proves to be the fastest with O(n3). However, the trend is not visible for these readings as they are too small.

**CONCLUSION**

Ease of Implementation

Because of the means of implementing Inheritance through Abstract Classes and Interfaces the Java Implementation is straightforward.

Every class implements the following methods of Problem Interface:

1) getInput() - Gets the input for the task

2) solve() - Contains the actual algorithm for the corresponding task

3) displayResult() - Displays output of the program to the console

The appropriate class implementation is chosen at runtime 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.

The programs mostly use primitive data structures like Arrays & 2-Dimensional Arrays, along with some Complex Data Structures like Vectors at few places. Below is the architecture of the implementation for our project:

Diagram

Description automatically generated