**COT 5405 ANALYSIS OF ALGORITHMS**

**PROGRAMMING PROJECT**

**Spring 2022**

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

Table of Contents

* 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

**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

**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

**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

**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. maxSum = -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 > maxSum
8. maxSum = submatrix\_sum
9. xLeft = r1
10. yLeft = c1
11. xRight = r2
12. yRight = c2
13. return maxSum, xLeft, yLeft, xRight, yRight

**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

**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