Name: Parth Tusham
UIN: 735008700
Course: CSCE 629

Problem 1

Input : A sequence of numbers S = (s1, s2, · · · , sn)
Output: A contiguous subsequence of S whose sum is maximized

1) **Main idea of the algorithm**
2) **Pseudo-code**
3) **Proof of correctness**
4) **Time Complexity**

Main Idea

The key idea is that for each position i, we calculate the maximum sum of a contiguous subarray that ends at the i^{th} element.

For example: let the array be S = (6, 16, −29, 11, −4, 41, 11)
(The size of S) n = 7

Then, $S_1 = 6$, $S_2 = 16$, $S_3 = -29$, $S_4 = 11$, $S_5 = -4$, $S_6 = 41$, $S_7 = 11$

So, for i = 4 all the possible contiguous subsequences ending at i are:

- S[1, 2, 3, 4] = 6 + 16 + (-29) + 11 = 4
- S[2, 3, 4] = 16 + (-29) + 11 = -2
- S[3, 4] = (-29) + 11 = -18
- S[4] = 11

From the above possible contiguous subsequences, we can see that S[4] is the **maximum possible sum (MPS).**

Like this, we will calculate MPS for each i 0< i <=n which we will call DP[i]

So, our answer for the maximum possible sum would be max(DP[i])

for $0 < i <= n$

This is because the contiguous subsequence with the max sum would have to end somewhere and we will consider all its endpoints i.e. from 1 to n (size of S)

We've come up with a brute-force solution that exhaustively calculates all possible sums for a contiguous subsequence ending at i^{th} position and then takes the maximum possible sum over all values of $0<i<=n$ after calculating each DP[i] value

This has a time complexity of $O(n^3)$

- The top loop goes from 1 -> n to mark the endpoints => $O(n)$
- The two loops inside would calculate all possible sums => $O(n^2)$
    - The first loop marks the start (j) which is from 1 -> i = O(n)
        - The second loop adds all elements from S[j…i] and saves the max value in DP[i] => O(n)
- We can use prefix sums to reduce the complexity to $O(n^2)$

Find the maximum possible contiguous subsequence sum

1. Initialize DP[1.....n] = - Inf (n is the size of the given array S)
2. For i = 1 to n
    - 2.1. For j = 1 to i
        - 2.1.1. Sum = 0
        - 2.1.2. For k = j to i
            - 2.1.2.1. Sum = sum + S_k
        - 2.1.3. Dp[i] = max(DP[i],Sum)
3. Ans = max{DP[0], DP[1], DP[2].....DP [n]}

So, to make this exhaustive search approach better in time complexity we will use the observation that

- Let's say we have DP[i] computed then how can we efficiently compute the subsequence with MPS ending at i +1 position i.e. the DP[i + 1] value
- The subsequence with MPS ending at i + 1 position will have S_{i+1} for sure but S_i depends on:
    - If we take Si then we add the DP[i] to DP[i+1] which means we are extending the subsequence ending at the i^{th} position with MPS to

end at i+1th position by adding the S_{i+1} element
=> DP[i+1] = DP[i] + S_i
    - If we don't take Si then our MPS only contains S_{i+1}
      => DP[i+1] = S_{i+1}
    - Now we take the maximum of both approaches
- For i==1 the contiguous subsequence with MPS ending at i = S_1
- **NOTE: But this isn't it, we try to backtrack using the DP values to get the elements that make up the contiguous subsequence with the maximum sum (as mentioned in the Problem)**

Therefore, for i = 1 to n (size of the array S) this is the function we get:

- DP [i] = Si , if i ==1
- DP [i] = Max { DP[i-1] + Si , Si } , otherwise

Pseudo Code

Find the maximum possible contiguous subsequence sum

1. Initialize DP[1.....n] (n is the size of the given array S)
2. For i = 1 to n
    2.1. If i ==1
        2.1.1. DP[i] = S_i
    2.2. If i > 1
        2.2.1. DP[i] = max {DP[i-1] + S_i , S_i}
3. Ans = max{DP[1], DP[2].....DP [n]}

Backtrack to generate the contiguous subsequence having the maximum sum

1. Initialize empty vector v (it will contain the elements)
2. EndPoint = n
3. For i = n to 1
    3.1. If DP[i] == Ans
        3.1.1. EndPoint = i
        3.1.2. break
4. RemainingSum = Ans
5. For i = Endpoint to 1
    5.1. If RemainingSum == 0
        5.1.1. break
    5.2. If DP[i] == RemainingSum
        5.2.1. v.push_back(S_i)
        5.2.2. RemainingSum -= S_i
6. v.reverse()

Proof of Correctness

Let the max contiguous sum ending at i be denoted as DP(i) and the Sum of elements of the given array from i to j be denoted as Sum(i -> j)
We know that, DP(i) = Max { Sum(1 -> i), Sum(2 -> i) ……. Sum (i -> i) }
Then,

1. DP(i +1) = Max { Sum(1 -> i +1), Sum(2 -> i+1) ……. Sum (i +1 -> i+1) }

2. DP(i +1) = Max { Sum(1 -> i) + S_{i+1} , Sum(2 -> i) + S_{i+1} …. Sum(i -> i) + S_{i+1} , S_{i+1} }

3. Dp (i+1) =Max { Sum(1 -> i), Sum(2 -> i) …… Sum(i -> i) , 0 } + S_{i+1}

4. Dp (i+1) =Max { Max **{** Sum(1 -> i), Sum(2 -> i) …… Sum(i -> i) **}** , 0 }

5. Dp (i+1) = Max {DP(i) , 0} + S_{i+1}

6. Dp (i+1) = Max {DP(i) + S_{i+1} , S_{i+1} }

Notes:

- In step 2, we separate the Sum(j -> i+1) into Sum(j->i) + S_{I+1}
- In step 3, we take the common value of S_{i+1} out of the Max { } operator
- In step 4, we plug in the value of DP (i)
- In Step 6, we push the value of S_{i+1} into the Max { } operator
- Max {x1, x2,x3,x4 } = Max { Max {x1, x2, x3 } , x4 }
- Max {x1 + x2, x1 + x3 } = Max {x2, x3 } + x1

The equation obtained in step 6 is equivalent to

- DP [i] = Si , if i ==1
- DP [i] = Max { DP[i-1] + Si , Si } , otherwise

This is the equation we came up with in the Main Idea Part

Hence, Proved that our algorithm gives the correct output for the maximum contiguous subsequence problem

For the backing tracking part, we're simply using the fact that when a DP(i) is equal to the Ans (max possible sum) then the optimal contiguous subsequence ends there. So, we just iterate from there towards i = 1 adding elements to our collection, and subtracting the value of those elements from RemainingSum while our RemainingSum is greater than 0. If it becomes zero that means we've reached the start point of our contiguous subsequence and we break out of the loop

Time Complexity

The time complexity for both the sum finding and backtracking parts is O(n)
As there are no nested loops, and loops run most of the n iterations while doing 0(1) calculations inside.

Therefore, the total Time complexity is O(n)