DATA STRUCTURE THEORETICAL ASSIGNMENT 1

Maximum Sum Submatrix Problem

Problem: Given a matrix A storing n^2 numbers, compute the submatrix of A whose sum is maximum among all possible submatrix of A.

Solution:

Let *tempmaxsum* denote the sum of the largest-sum submatrix starting at (and including) *ith* row of A and ending at row with index such that i <=index<= n.So to solve our problem, it suffices if we compute *tempmaxsum* for all i and update *maxsum* at each iteration. Let us now try to find *tempmaxsum*. Let $T_{ij}(n)$ represnt the maximum sum sub array of the array C formed by adding all rows from i to j , $i < j$ with number of columns n ranging from 0 to n.So it is clear that *tempmaxsum* is the maximum of all $T_{ij}(n)$ with j varying from i to n. We state the following lemma which establishes a relationship between $T_{ij}(k)$ and $T_{ij}(k-1)$ for any $k \geq 1$.

Lemma 1. *If $T_{ij}(k-1) \leq 0$ then $T_{ij}(k) = C[k]$; otherwise $T_{ij}(k) = T_{ij}(k-1) + C[k]$.*

Proof. The key insight to prove the lemma lies in the following relationship between the subarrays associated with $T_{ij}(k)$ and $T_{ij}(k-1)$. Notice that the subarray associated with $T_{ij}(k-1)$ if appended with $C[k]$ gives us a subarray of C which terminates at $C[k]$. Hence, $T_{ij}(k) \geq T_{ij}(k-1) + C[k]$.

Now $T_{ij}(k-1) < 0$ implies that every subarray terminating at $C[k-1]$ has a negative sum. In other words, it is futile to append any subarray terminating at $k-1$ with $C[k]$ to get a better sum. Hence $T_{ij}(k) = C[k]$ in this case. Likewise, if $T_{ij}(k-1) > 0$, then by definition, the subarray corresponding to $T_{ij}k-1$ is the subarray of maximum sum terminating at $C[k-1]$. This subarray when appended with $C[k]$ will give us a subarray terminating at $C[k]$ and having maximum sum. This is exactly the subarray corresponding to $T_{ij}(k)$. Hence $T_{ij}(k) = T_{ij}(k-1) + C[k]$ in this case. □

Algorithm 1 computes maximum sum submatrix based on the above discussion. Algorithm 2 computes maximum sum subarray based on the lemma 1.

Proving correctness of the algorithm:

We need to prove that Algorithm 1 is correct. In order to prove that first we need to prove that Algorithm 2 is correct. So, lets prove that first. It follows from the discussion preceding Lemma 1 that it suffices if we can show that Algorithm 2 correctly computes *maxsum*.

Lemma 2. *At the end of the "for" loop in Algorithm 2, $T[i]$ stores maxsum for each $i < n$.*

Proof. We give a proof by induction of i. In particular, we show that assertion $\mathcal{C}(i)$, defined below is true for all $i < n$.
$\mathcal{C}(i)$: At the end of ith iteration, $T[i] = maxsum$ till i.
The base case ($i = 0$) is trivially true since we explicitly set $T[0]$ to $C[0]$ (which is indeed *maxsum* till that point) in the first statement of the algorithm. Hence $\mathcal{C}(0)$ holds true.

Let us prove that $\mathcal{C}(j)$ is true for some $j > 0$, assuming, as induction hypothesis, that $\mathcal{C}(i)$ is true for all $i < j$. Consider the beginning of jth iteration. Firstly note that by induction hypothesis, $T[j-1] = maxsum$ till j-1. Now Lemma 1 states that $T[j]$ is $C[j]$ if $T[j-1] < 0$ and $T[j-1]+C[j]$ otherwise. Since $T[j-1]$ stores *maxsum* till that point, it follows from the "if" statement executed during jth iteration of "for" loop that $T[j]$ stores *maxsum*.

Hence by principle of mathematical induction, $T[i]$ stores *maxsum* for each $i < n$.

Data: Input will be a matrix $A[0..n-1][0..n-1]$. However, the algorithm will use additional array $C[0..n-1]$ such that $C[k]$ will store sum of all elements in *ith* column from row i , j at any instant. Variable maxsum is used to store the sum of maximum sum submatrix. The variables x1,x2,y1,y2 are used to track the maximum sum submatrix.

Result: The sum of the maximum sum submatrix along with its location in matrix A.

$maxsum \leftarrow A[0][0]$;
$x1, x2, y1, y2 \leftarrow 0$;
for $i \leftarrow 0$ **to** $n-1$ **do**
    $C[0..n] \leftarrow 0$ **for** $j \leftarrow i$ **to** $n-1$ **do**
        $C[0..n-1] \leftarrow C[0..n-1] + A[j][0..n-1]$
        $temp \leftarrow$maximumsumsubarray($C[0..n-1]$) **if** $temp > maxsum$ **then**
            $maxsum \leftarrow temp$ $y1 \leftarrow i$ $y2 \leftarrow j$
        end
    end
end
Output is submatrix from row $y1$ to $y2$ and column $x1$ to $x2$.The maximum sum of thissubmatrix is $maxsum$.

Algorithm 1: Algorithm for finding Maximum-Sum Sub Matrix

Data: Input will be an array $C[0..n-1]$. However, the algorithm will use additional array $T[0..n-1]$ such that $T[i]$ will store maximum sum of maximum sum subarray till i including i.

Result: The sum of the maximum sum subarray along with its leftmost and rightmost indices.

$T[0] \leftarrow C[0]$;
$x1, x2 \leftarrow 0$;
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $T[i-1] \leq 0$ **then**
        $T[i] \leftarrow C[i]$;
        $x1, x2 \leftarrow i$;
    else
        $T[i] \leftarrow T[i-1] + C[i]$;
        $x2 \leftarrow i$;
    end
end
Return maximum of T[0..n-1] and x1,x2.

Algorithm 2: Algorithm for finding Maximum-Sum Sub Array

So we have proved that Algorithm 2 returns correct maxsum to Algorithm 1. Now, we can say that at end of each inner "for" loop in Algorithm 1 we get correct value of $maxsum$ because Algorithm 2 returns correct tempmaxsum and it follows from the "if" statement executed during jth iteration of "for" loop that $maxsum$ stores correct $maxsum$ till that point. Hence we can say that $maxsum$ is correct at end of each jth iteration. Since i varies over all rows and so the i and j hovers over all submatrices possible for A so $maxsum$ is correct $maxsum$ found from all submatrices.

Hence the $maxsum$ stores the correct answer. $\square$

Analysis of Time and Space complexity the Algorithm

The Algorithm 1 executes a "for" loop $n-1$ times and in each iteration this "for" loop executes a statement of $O(1)$ time and another "for" loop in which it executes few statements of $O(1)$ time and calls Algorithm 2 which executes "for" loop $n-1$ times, and in each iteration it spends$O(1)$ time. Thereafter, it spends a total of $O(n)$ time in the "for" loop. Thereafter, it scans array S to find the maximum element. This also takes $O(n)$ time. So overall time complexity of the Algorithm 2 is $O(n)$. It implies that overall complexity of our Algorithm 1 is $O(n^3)$. The algorithm uses two additional arrays C and T in addition to a few variables. Hence, the algorithm uses $O(n)$ extra space.