1 Formal definition of maximum contiguous array(MCS) problem

Definition of $V(i,j)$: Let $R[1..n]$ be an array of n real numbers. The value of subarray $R[i...j]$ is

$$V(i,j) = \Sigma_{x=i}^{j} R[x]$$

Definition of maximum contiguous array problem:

- **Input:** An array $R[1..n]$ of n real numbers.
- **Output:** $V(i,j)$ such that $\forall(i',j'), V(i',j') \leq V(i,j)$, output i, j and $V(i,j)$'s value.

2 The recurrence relation of your dynamic programming algorithm

Definition: $MCSsuffix[i...j]$: maximum contiguous array which contain the suffix j in subarray $R[i...j]$
Buttom up computation of $MCSsuffix[i..j]$:
If $i = j$: only one elements, maximum contiguous subarray of $R[i..j]$ is itself.
Eles $i < j$:

$$MCSsuffix[i...j] = \begin{cases} V[j...j],\ if\ MCSsuffix[i...j-1] \leq 0 \\ MCSsuffix[i...j-1] + R[j],\ if\ MCSsuffix[i...j-1] > 0 \end{cases}$$

Compute $MCSsuffix$ from i=1 to length, and maximum contiguous array result is the biggest value in $MCSsuffix$, use a variable $maxValue$ to record it.

3 definitions of the notations in your recurrence relation

Variable	Description
$input[]$	the input array R
$length$	the length of the input array R
$mcsSuffix$	An array, represent the table used for the dynamic programming to store the maximum contiguous subarray result of $R[i...j]$, the suffix of recorded maximum contiguous subarray should be j, and it will also store the start index of maximum contigous subarray.
$maxValue$	the maximum value of $MCSsuffix$
i	index of array for traversal
$V[i...j]$	$\Sigma_{x=i}^{j} R[x]$, the sum value of subarray $R[i...j]$
$indexPointer$	pointer help to record multiple optimal solution
$count$	count how many optimal solution
$startIndex$	array to record start index of all optimal solution
$endIndex$	array to record end index of all optimal solution

4 Pseudocode of the MCS algorithm

Algorithm 1 Maximum contiguous subarray algorithm

1: **function** MCS($input[], length$)
2: **for** i=0 to length **do**
3: intialize $startIndex[i]$ and $endIndex[i]$ to -1 ▷ -1 represent INVALID
4: **end for**
5: $indexPointer = 0$
6: create a new array $mcsSuffix[length][2]$ and initialize all elements to INVALID ▷ mcsSuffix[i][0] record the mcs result, ncsSuffix[i][1] record the start index of mcs result
7: $maxValue$=$input[0]$, $mcsSuffix$=$input[0]$ ▷ buttom-up computation
8: **for** i=1 to length **do**
9: $mcsSuffix[i]$=$input[i]$
10: **if** $mcsSuffix[i-1]$>0 **then**
11: $mcsSuffix[i]$=$mcsSuffix[i-1]$+$input[i]$ ▷ Both mcs value and start index will bre recorded in mcsSuffix array
12: **end if**
13: **if** $mcsSuffix[i]$=$maxValue$ **then**
14: record index to $startIndex$ and $endIndex$
15: **end if**
16: **if** $mcsSuffix[i]$>$maxValue$ **then**
17: $maxValue$=$mcsSuffix[i]$
18: record index to $startIndex[0]$ and $endIndex[0]$ ▷ Start index can get from mcsSuffix
19: set INVALID to $startIndex[1]$ and $endIndex[1]$
20: **end if**
21: **end for**
22: $count$=1
23: **for** i=0 to length **do**
24: **if** $startIndex[i]$=INVALID or $startIndex[i]$=0 **then**
25: break
26: **end if**
27: $count++$
28: **end for**
29: **return** $maxValue$, all start and end index of optimal solution in $startIndex$ and $endIndex$, count of optimal solutions $count$
30: **end function**
31: **function** MAIN
32: read array from input file and store it in to a fixed size array $input[]$
33: $mcsResult$ = Mcs($input$, $length(input)$)
34: write $mcsResult$ to output file
35: **end function**

5 Time cost analysis of the algorithm

Only three loops are used in the algorithm, and each loop has a constant time complexity, index takes on $\leq$ length size values,without nested loop. Therefore, the time complexity is $T(n) = O(n)$
Note that the algorithm can return many optimal solutions, but it will not return all optimal solutions, because the $mcsSuffix$ array only store one index of previous optimal solution. If previous optimal solution has many start index which have the same maximum value, previous algorithm will not load it. And when we want to achieve the second functionality bonus, we need to improve the algorithm, load all start index of previous optimal solution into current mcsSuffix array, which takes $O(n)$ time, and it is inside the procsss input loop, so the worst case time complexity will be $O(n^2)$, but it seldom happens. Detail will be explained in section 6.2.

6 Implement of functionality bonus

6.1 Write the documentation in markdown files or LaTeX:

Achieve it.

6.2 Output all optimal solutions (both on the screen and to the output file):

1. Use a startIndex array and endIndex array to record all optimal solutions, and output them both on the screen and the output file. And to ouput all optimal solutioncan be little bit difficult because in previous algorithm, $mcsSuffix$ array will only store one index of previous optimal solution to current dynamic optimal solution table, if we need to handle the situation that current start index is same as the start index of previous, improve it using a array to and load all previous start index of solution into current start index,store in the array, similar to current solution inherit all start index of previous, rather than using $mcsSuffix[length][2]$ array,

2. Use a one dimensional array $mcsSuffix[length]$ as the dynamic programming table to store all optimal solution values, $mcsSuffix[length]$ is the optimal solution value of $R[0...i]$. And use a two dimensional array $mcsStartIndex[length][length]$ as the dynamic programming table to store all start index of optimal solution, $mcsStartIndex[i][...]$ rececord all start index of solution of $R[0...i]$,.

3. Size of startIndex and endArray can be larger than the length of input array. It will be at most $0.5 * (length * length) + 1$ because there are at most $0.5 * (n * (n + 1))$ pair of n size subarray

    - **Example:** The array input size is 5, total it has $(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4)$

$(4, 4)$ pairs of subarray,total $5 + 4 + 3 + 2 + 1 = 15$ pairs of subarray, equal to $0.5 * (5 * (5 + 1)) = 15$.

So if size is n it will have at most $0.5 * (n * (n + 1))$ pairs of subarray, and it is also the maximum optimal solution number.

4. And because we need to load all previous start index of previous optimal solution into current start index of current optimal solution, It takes maximum $O(n)$ time to do once load all previous start index. So the algorithm of output all optimal solution will be $O(n^2)$. But it seldom happens, because in most cases, the start index of previous optimal solution will only have start value. Only in some cases like there are many 0 in the array or sum of subarrau is 0 will cause the problem.

   - **Example:** Input array is $R = [2, -2, 2, -2, 2]$, so the start index of $mcsSuffix[4]$ can be $0, 2, 4$, because both $V(0, 4), V(2, 4), V(4, 4)$ are the optimal solution 2.
   - **Example:** Input array is $R = [0, 0, 0, 0, 0]$, the optimal solution is 0, start index of $mcsSuffix[4]$ can be $0, 1, 2, 3, 4$, and optimal subarray can be all subarray of R because all $V(i, j)$ are 0.

   Overall, in most cases, the time complexity can be $O(n)$, but in these kinds of cases with many 0 or sum is 0, worst time complexity is $O(n^2)$.

5. The submitted code is the code that can output all optimal solutions, with worst time complexity $O(n^2)$ and larger space complexity. It might have some difference of the variable name between Pseudocodeand actual code, like mcsSuffix is dp in actual code and mcsStartIndex is dpStartIndex in actual code, the msSuffix and mcsStartIndex is used for simple presentation in pseudocode step and know what does it store, and dp and dpStartIndex are more convenient for actual coding and inplementation step, we can have a better understanding of dynamic programming thought during coding. And it doesn't a big problem between it.

6. Improved pseudcode:

Algorithm 2 Maximum contiguous subarray algorithm

1: **function** MCS($input[], length$)
2: **if** length=1 **then return** $input[0]$ as corrersponding MCS return
3: **end if**
4: $resultSize = 0.5 * (length * (length + 1))$
5: **for** i=0 to resultSize **do**
6: intialize $startIndex[i]$ and $endIndex[i]$ to -1, and startIndex[0] and endIndex[0] is 0 ▷ -1 represent INVALID
7: **end for**
8: $indexPointer = 1$
9: create a new array $mcsSuffix[length]$ and initialize all elements to 0
10: create a new array $mcsStartIndex[length][lengthj]$ and initialize all elements to INVALID
11: $maxValue$=$input[0]$, $mcsSuffix$=$input[0]$
12: **for** i=1 to length **do**
13: $mcsSuffix[i]$=$mcsSuffix[i-1]$+$input[i]$
14: **if** $mcsSuffix[i-1]$>0 **then**
15: $mcsSuffix[i]$=$mcsSuffix[i-1]$+$input[i]$
16: load all start index inside $mcsStartIndex[i-1][...]$ into $mcsStartIndex[i][...]$, and end with INVALID ▷ Takes O(n) time
17: **end if**
18: **if** $mcsSuffix[i-1]$=0 **then**
19: load all start index inside $mcsStartIndex[i-1][...]$ into $mcsStartIndex[i][...]$, and end with INVALID ▷ Takes O(n) time
20: load current index into the $mcsStartIndex[i][...]$ and set it end with INVALID after all optimal start index
21: **end if**
22: **if** $mcsSuffix[i]$=$maxValue$ **then**
23: $maxValue$=$mcsSuffix[i]$
24: record all start index of current elements into $startIndex$
25: recode current index to $endIndex$, startIndex and endIndex array's corner makers should be correspond
26: set INVALID to end of optimal start index in $endIndex$ and $endIndex$ ▷ Takes O(n) time
27: **end if**
28: **if** $mcsSuffix[i]$>$maxValue$ **then**
29: $maxValue$=$mcsSuffix[i]$
30: record all start index of current elements into $startIndex$
31: recode current index to $endIndex$, startIndex and endIndex array's corner makers should be correspond
32: set INVALID to end of optimal start index in $endIndex$ and $endIndex$
33: **end if**
34: **end for**
35: $count$=indexPointer
36: **return** $maxValue$, all start and end index of optimal solution in $startIndex$ and $endIndex$, count of optimal solutions $count$
37: **end function**

6.3 The number of integers from the input is unknown. So, you cannot assume the maximum length of the input array. Thus, to read an arbitrary length array, you need to use dynamic memory allocation. The time cost of reading the input needs to be in $O(n)$.

Use malloc function and remalloc to do the dynamic memory allocation. If the input file size is larger than the current allocated memory, then reallocate twice as much memory. The time cost of reading the inputs fixed $O(n)$.