

Algorithm & Data Structures

Assignment: MaxSubsequenceArray optimized
Implementation.

Problem Statement : Implementing Maximum Subsequence Array
Algorithm using Divide and Conquer inorder to reduce the running
time to $O(N\log N)$.

Report by Sayali More(801076191)

The Maximum Subsequence Array algorithm sums each element with the continuous subsequent element in order to find a subarray that gives the maximum sum. In general implementation it takes $O(n^2)$ running time.

The aim of this assignment is to implement the Maxsubsequence problem using Divide and Conquer method and thus reduce its running time to $O(n \log n)$

Brief Description of the divide and conquer method implementation :

Pseudocode:

Find-MSS($A, low, high$)

```
1  if  $low = high$ 
2    then return  $(low, low, A[low])$ 
3  else if  $low < high$ 
4     $mid \leftarrow \text{floor}((low + high)/2)$ 
5     $(i_l, j_l, T_l) \leftarrow \text{Find-MSS}(A, low, mid)$ 
6     $(i_r, j_r, T_r) \leftarrow \text{Find-MSS}(A, mid + 1, high)$ 
7     $(i_s, j_s, T_s) \leftarrow \text{Max-Span}(A, low, mid, high)$ 
8     $T \leftarrow \max(T_l, T_s, T_r)$ 
9  switch
10   case  $T = T_l$ :
11     return  $(i_l, j_l, T_l)$ 
12   case  $T = T_r$ :
13     return  $(i_r, j_r, T_r)$ 
14   case  $T = T_s$ :
15     return  $(i_s, j_s, T_s)$ 
16 else  $low > high$ 
17   return  $(A, low, high + 1)$ 
```

Max-Span ($A, low, mid, high$)

```
1  leftsum = -inf; sum = 0
2  for  $i = mid$  downto  $low$  // Find max-subarray of  $A[i..mid]$ 
3    sum = sum +  $A[i]$ 
4    if sum > leftsum
5      leftsum = sum
6      maxleft =  $i$ 
7  rightsum = -inf; sum = 0
8  for  $j = mid+1$  to  $high$  // Find max-subarray of  $A[mid+1..j]$ 
```

```

9      sum = sum + A[j]
10     if sum > rightsum
11         rightsum = sum
12         maxright = j
13     return (maxleft,maxright,leftsum + rightsum) // Return the indices i and j and the sum of
        two subarrays

```

Explanation:

The divide and conquer approach divides the array into two halves i.e. Left subsequence and Right subsequence. The left subsequence is the subarray starting from array start position 0 to mid and the right subsequence is the subarray starting from mid+1 position to the end of the array i.e. n. Then we scan both the subsequences (left and right) to find their corresponding subsequences from their subarrays.(Find-MSS()) These independent maxsum function returns the start and the end position of the max subarray in their respective subarrays with their maxsum. This can be carried out recursively till we get the base case. Since the subsequence can lie on any side left to mid or mid+1 to right of in the middle elements we scan the middle elements too to find the subsequence that gives max sum(MaxSpan()). Then the maximum of all the three values returned by the function is taken as the final maximum subsequence.

The approach thus divides the work recursively and scans the subsequences reducing the running time to $O(n\log n)$.

Time Complexity:

As the algorithm divides the array into two halves and scans them recursively the time $T(n)$ taken by the Algorithm can be given as follows:

WORLD STAR
Date: _____
Page: _____

For dividing into subarrays constant time for combining max sub array.

$$T(n) = 2T(n/2) + c(n)$$

Solving, the equation using Master Method we get,

$$a=2 \quad b=2 \quad f(n)=n$$

$$\therefore g(n) = \frac{\log^a n}{n \log b} = \frac{\log^2 n}{n \log 2} = \frac{1}{n} = n$$

Hence, it is a case 2

$$\therefore T(n) = O(n^{\log_b a} \log n)$$

$$= O(n^{\log_2 2} \log n)$$

$$= O(n \log n)$$

Hence, the running time for this approach is $O(n \log n)$

Code Explanation and Correctness Analysis:

- i. Indices i & j can never reference A outside the interval $A[p, \dots, r]$ when such condition occur when $low > high$ the algorithm resorts to the previous stable condition when $low == high$ and returns the max sum of that subarray.

ii. This approach tests for all possible positions of the max subsequence by scanning left sub array, right sub array and the middle elements thus assuring us checking of all the possible max subarray and then giving the maximum of all the three results of MSS function. This, ensures that after every function call we get the max subsequence of that subarray.

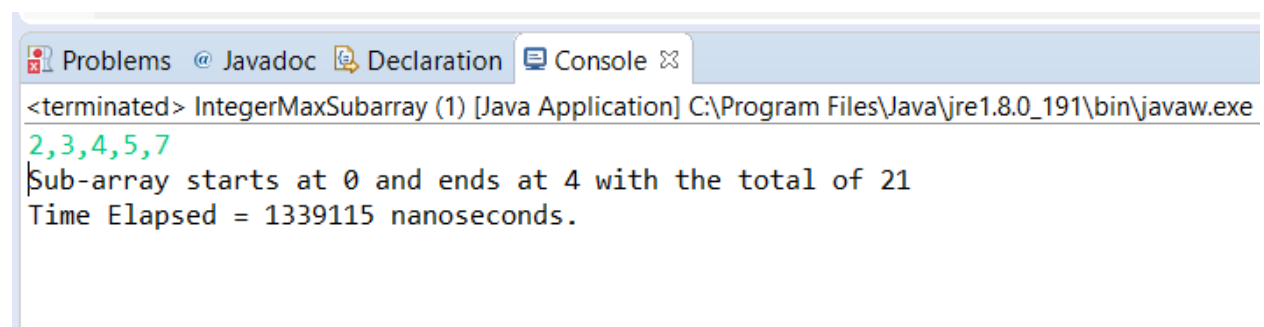
iii. Extreme cases:

The extreme cases for the MaxSubsequence problem can be when all the input integers are positive or negative. When all the integers happen to be positive it should return the entire array as the maximum array.

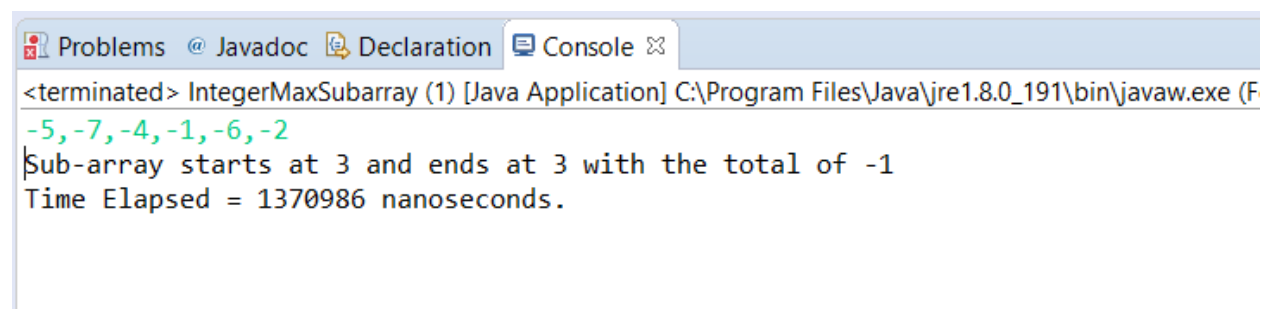
Also, for the array having all the negative integers it should return the smallest negative integer as the maximum subsequence.

Case when all the input elements are zeros. It returns the max sum as zero and the start and end index as the starting element index.

The algorithm implemented follows this as can be seen from the results:



```
Problems @ Javadoc Declaration Console
<terminated> IntegerMaxSubarray (1) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
2,3,4,5,7
Sub-array starts at 0 and ends at 4 with the total of 21
Time Elapsed = 1339115 nanoseconds.
```



```
Problems @ Javadoc Declaration Console
<terminated> IntegerMaxSubarray (1) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (F
-5,-7,-4,-1,-6,-2
Sub-array starts at 3 and ends at 3 with the total of -1
Time Elapsed = 1370986 nanoseconds.
```

```
Problems @ Javadoc Declaration Console
<terminated> IntegerMaxSubarray (1) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\ja
0,0,0,0,0,0
Sub-array starts at 0 and ends at 0 with the total of 0
Time Elapsed = 2098889 nanoseconds.
```

iv. Checking for the base case correctness:

The base case or the termination condition of the recursive call that is when there's only one element in the subarray, should always return the correct results at the termination of the call or loop. In the above algorithm when ($A.length == 0$ (empty)) returns null or when ($i == j$) the call returns the calculated max subsequence for that subarray. Thus, terminates the function for each function call correctly. When the problem is sufficiently small like may be just when $n=1$ it returns trivial results ($A[0]$ value as the return value for that function call)

v. Loop Invariant:

Loop invariant is a condition which remains unchanged or is satisfied for every recurrence call. It varies from algorithm to algorithm and for the above code this condition can be thought of as the Max sum condition which is always greater than or equal to the total sum of its subarray.

At the termination of the calls the above algorithm it always **guarantees** to return the maximum of the entire series as it compares result from all the sub arrays and selects the maximum among them to return as output.

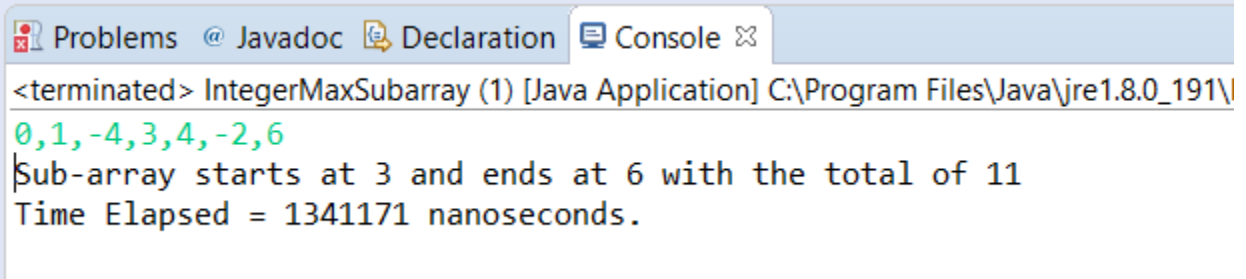
vi. Assumption step:

As can be seen from the above algorithm it divides the array into smaller problem and divides it to single elements thus returning the individual elements and then summing them one by one by combining two sub arrays and returning the maximum of the combined arrays at the end of each function call.

Algorithm Output:

Input Sequence: {0,1,-4,3,4,-2,6}

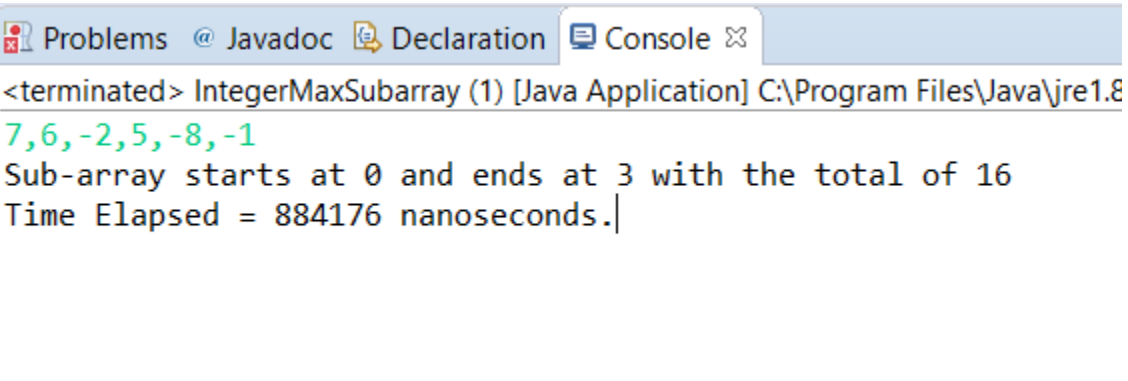
Result:



```
<terminated> IntegerMaxSubarray (1) [Java Application] C:\Program Files\Java\jre1.8.0_191\
0,1,-4,3,4,-2,6
Sub-array starts at 3 and ends at 6 with the total of 11
Time Elapsed = 1341171 nanoseconds.
```

Input Sequence: {7,6,-2,5,-8,-1}

Result:



```
<terminated> IntegerMaxSubarray (1) [Java Application] C:\Program Files\Java\jre1.8
7,6,-2,5,-8,-1
Sub-array starts at 0 and ends at 3 with the total of 16
Time Elapsed = 884176 nanoseconds.
```

Optional Question:

Can we design a divide-and-conquer solution with a linear running time?

Yes, We can design a divide and conquer algorithm with a linear time. As can be seen due to the combine step in the above algorithm a linear term to the recurrence term is added.

$$T(n) = 2T(n/2) + cn$$

If we modify this combine step of the above divide and conquer algorithm and make the linear time to constant time we will be able to achieve linear running time. As shown in the paper Divide & Conquer strikes back: maximum-subarray in linear time by Ovidiu Daescu and Shane St. Luce at UT, Dallas we can achieve this by changing the combine to series of comparison by sending some more information in the output.

```
function FMS-COMPARE(A; low; high)
```

```
2: if low = high then
```

```
3:   return (A[low];A[low];A[low];A[low])
```

```
4: else
```

```
5:   mid = (low+high)/2
```

```
6: Left = FMS-COMPARE(A; low; mid)
```

```
7: Right = FMS-COMPARE(A; mid + 1; high)
```

```
8: return COMPARE(A; Left;Right)
```

```
9: end if
```

```
10: end function
```

```
11: function COMPARE(A; L;R)
```

```
12:   totalSum = L:totalSum + R:totalSum
```

```
13:   maxPrefix = MAX(L:maxPrefix;
```

```
14:   L:totalSum + R:maxPrefix)
```

```
15:   maxSuffix = MAX(R:maxSuffix;
```



```

16: R:totalSum + L:maxSuffix)
17: maxSum = MAX(L:maxSum;R:maxSum;
19: L:maxSuffix + R:maxPrefix)
20: return (totalSum; maxSum; maxPrefix;
21: maxSuffix)
22: end function

```

Following the above pseudocode, I have implemented it in Python and am able to get the correct results from the program.

Algorithm Output:

Input Sequence: {0,1,-4,3,4,-2,6}

Result:

```

35 print("Maximum Sum: ", ans[1])

```

```

Enter the number of elements you want:7
Enter numbers in array:
number :0
number :1
number :-4
number :3
number :4
number :-2
number :6
Total time taken in seconds : 0.0

Total Sum: 8
Maximum Sum: 11

```

Input Sequence: {7,6,-2,5,-8,-1}

Result:

```
35 print("Maximum Sum: ", ans[1])
```

Enter the number of elements you want:6

Enter numbers in array:

number :7

number :6

number :-2

number :5

number :-8

number :-1

Total time taken in seconds : 0.0

Total Sum: 7

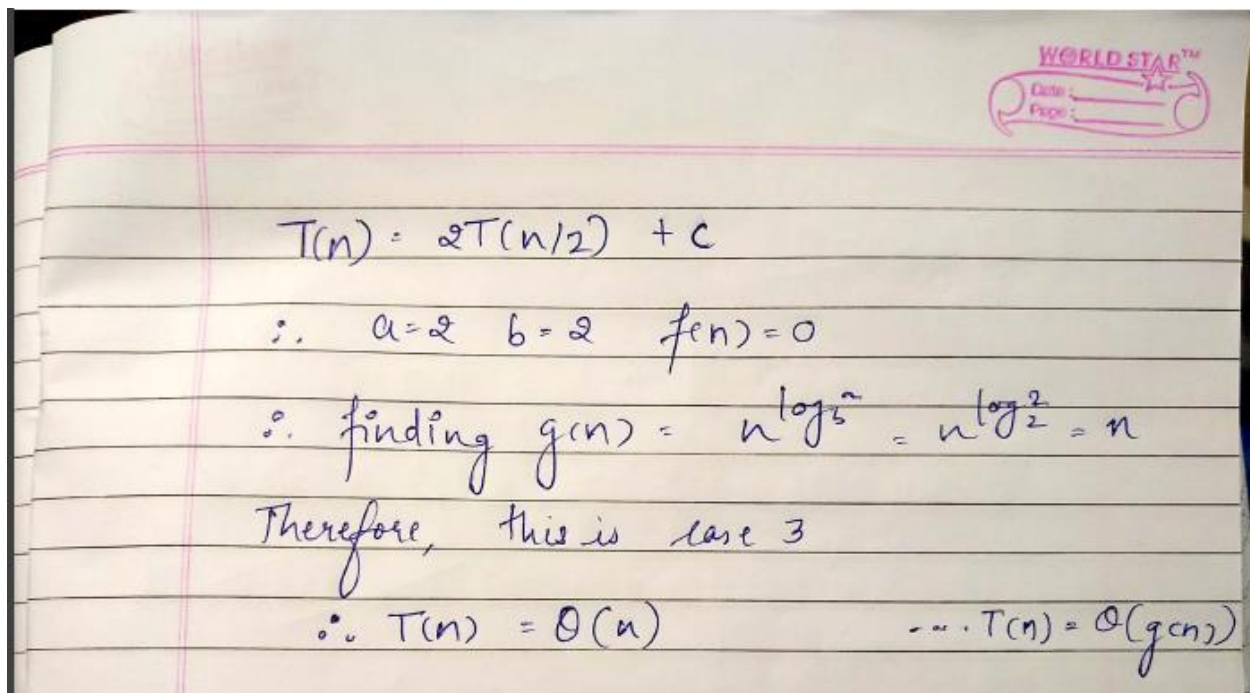
Maximum Sum: 16

Time Complexity:

As in this algorithm, the calculation after the recursive calls involves a constant number of comparisons and additions, the overall time to find the totalSum, maxSum, maxPrefix, maxSuffix tuple the Time required by this algorithm becomes:

$T(n) = 2T(n/2) + c$, where c is a constant.

Solving it ahead we get,



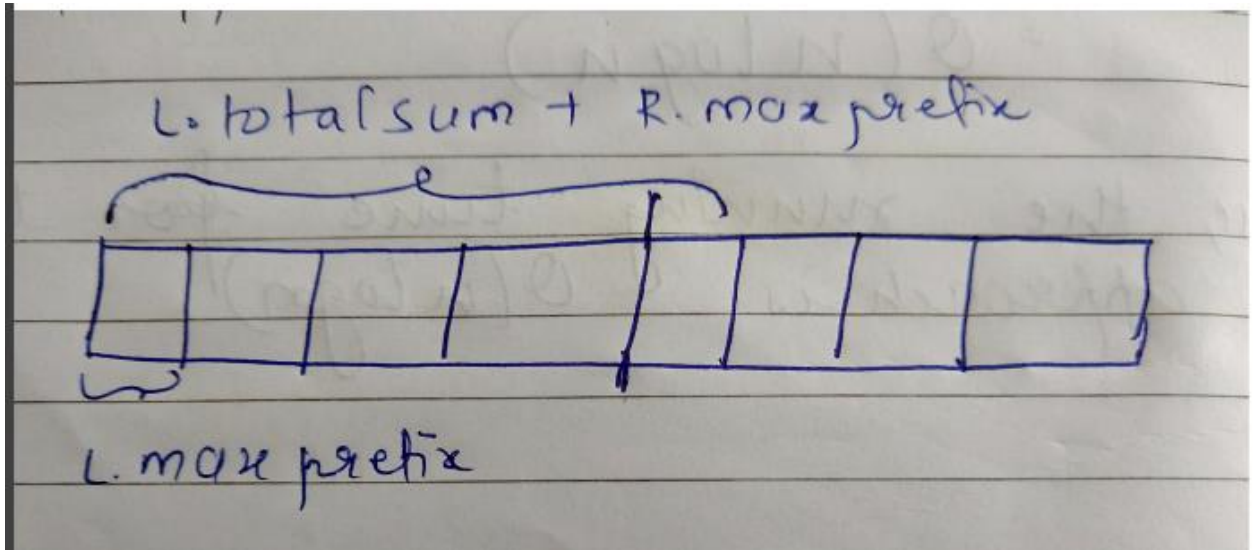
Code Explanation and Correctness Analysis:

The alternate linear time complexity algorithm follows all the correctness rules mentioned above.

1. The low and high indexes never go out of bound. The function calls with high and low as mid and mid+1 resp. takes care of it.
2. For the base case when $i=j$ it returns the single element as the totalsum, maxsum, maxprefix and maxsuffix as expected when there are more than one element in the subarray it calculates the totalsum of the elements in the array, maximum sum of the sub array, max prefix elements sum and max suffix sum of the array.
3. Assumption and Inductive step:
In this algorithm, Considering u as a node having v and w as its left and right children return their TotalSum, MaxSum, MaxPrefix and MaxSuffix to u which is stored in L & R respectively. So, the total sum at u is just the sum of $L.\text{totalsum} + R.\text{totalsum}$. For maxprefix at u there are two cases:
 - a. If the maximum prefix of the array at u does not cross the middle of the array then, the maximum prefix of that array must be the same as the maximum prefix of the left subarray, stored at v .

- b. If the max prefix crosses the middle, then it must include the entire left subarray as well as the maximum prefix of the right subarray, stored at w.

Then, the maximum prefix of the array stored at u is the maximum of $L(\text{maxPrefix})$ and $L(\text{totalSum} + R:\text{maxPrefix})$. As depicted below in the image thus it assures to always return the maximum sum of the subarray.



Similarly, the maximum suffix at u is the maximum of $R:\text{maxSuffix}$ and $R(\text{totalSum}) + L(\text{maxSuffix})$. Lastly, the maxSum of the array at u is determined by comparing $L(\text{maxSum})$ and $R(\text{maxSum})$ with the maximum crossing sum, $L(\text{suffix}) + R(\text{prefix})$. This is followed even for the combining step when only the maximum of the left and right sub array is taken. Thus, ensuring the maxsum to be returned at the end of the algorithm.

Proof of Linearity:

As can be seen from above explanation, since the calculation after the recursive calls involves a constant number of comparisons and additions, the overall time to find the totalSum, maxSum, maxPrefix, maxSuffix tuple at u is constant. Thus, the total time taken is just the sum of the division calls and a constant time c. which gives the time complexity as $O(n)$ as shown above.

