

## Practice Exercise #39: Maximum Contiguous Subsequence Sum

[http://www.comp.nus.edu.sg/~cs1020/4\\_misc/practice.html](http://www.comp.nus.edu.sg/~cs1020/4_misc/practice.html)

Prepared by Aaron Tan

### Objectives:

- Writing efficient algorithm
- Knowing how to analyse algorithm

### Task statement:

(Taken from “Data Structures and Algorithm Analysis” by Mark Allen Weiss.)

A *contiguous subsequence* of a list is a subsequence that is made up of consecutive elements of the list. For example, for the list {6, 3, 9, 1, 2}, some examples of contiguous subsequence are {} (empty subsequence), {1}, {6, 3, 9}, {3, 9, 1, 2}, and {9, 1}. The following are not contiguous subsequences of the given list: {3, 1}, {6, 9, 1, 2}.

Here, we shall refer to a contiguous subsequence simply as a subsequence to keep the description short.

The **Maximum Subsequence Sum** problem is defined as follows:

Given a list  $A$  with  $n$  elements  $a_0, a_1, \dots, a_{n-1}$ , find the maximum value of

$$\sum_{k=i}^j a_k$$

For example, for the list  $A = \{-2, 11, -4, 13, -5, -2\}$ , the maximum subsequence sum is 20 ( $a_1$  through  $a_3$ ).

For convenience, the maximum subsequence sum is defined to be 0 if all the integers in the subsequence are negative. You may assume that there are at least two elements in the list.

You are given 2 algorithms which are inefficient.

### Algorithm 1

```
public static int maxSubSumV1(int[] arr) {           // line 1
    int maxSum = 0;                                   // line 2
    for (int i = 0; i < arr.length; i++) {           // line 3
        for (int j = i; j < arr.length; j++) {       // line 4
            int thisSum = 0;                           // line 5
            for (int k = i; k <= j; k++)               // line 6
                thisSum += arr[k];                     // line 7
            if (thisSum > maxSum)                       // line 8
                maxSum = thisSum;                       // line 9
        }                                              // line 10
    }                                                  // line 11
    return maxSum;                                     // line 12
}
```

### Analysis of Algorithm 1

Let  $n$  be the length of array `arr`. At a quick glance, the running time of the algorithm is  $O(n^3)$ , by observing that the statement in **line 7** is inside a triply nested loop, each loop running in  $O(n)$ . It happens that this rough estimate gives the correct big-O running time.

A more precise analysis is obtained by determining how many times line 7 is actually executed, given by:

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^j 1$$

Working from inside out,

$$\sum_{k=i}^j 1 = j - i + 1$$

Next, we evaluate

$$\sum_{j=i}^{n-1} (j - i + 1) = \frac{(n - i)(n - i + 1)}{2}$$

using equality (1) in the handout “Some Equalities” on the CS1020 “Lectures” web page, recognising that it is the sum of the first  $n - i$  integers.

Finally,

$$\sum_{i=0}^{n-1} \frac{(n - i)(n - i + 1)}{2} = \frac{n^3 + 3n^2 + 2n}{6} = O(n^3)$$

Hence algorithm 1 has a running time of  $O(n^3)$ .

It is quite apparent that algorithm 1 is very inefficient, due to repeated computations in the inner-most loop. Removing the inner-most loop would improve the efficiency drastically.

(Try to do this yourself before reading on!)

## Algorithm 2

```
public static int maxSubSumV2(int[] arr) {    // line 1
    int maxSum = 0;                          // line 2
    for (int i = 0; i < arr.length; i++) {    // line 3
        int thisSum = 0;                     // line 4
        for (int j = i; j < arr.length; j++) { // line 5
            thisSum += arr[j];                // line 6
            if (thisSum > maxSum)              // line 7
                maxSum = thisSum;             // line 8
        }                                    // line 9
    }                                        // line 10
    return maxSum;                           // line 11
}
```

## Analysis of Algorithm 2

The ‘for j’ loop now is the inner-most loop, so we shall calculate the number of times **line 6** is executed, as follows:

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1} (n-i) = n^2 - \sum_{i=0}^{n-1} i = n^2 - \frac{(n-1)n}{2} = \frac{n^2 + n}{2} = O(n^2)$$

Hence algorithm 2 has a running time of  $O(n^2)$ , a huge improvement over algorithm 1.

## Algorithm 3

Now, your challenge is to find the fastest algorithm, with a running time of  $O(n)$ , for this problem. This linear-time complexity is another huge improvement over the quadratic time complexity of algorithm 2.

A BIG hint is given at the end of this write-up, but refer to it only after you have tried it out on your own first.

## Recursive solution

You may attempt a recursive solution on your own. However, the recursion solution has a running time of  $O(n \lg n)$ . Hence the linear-time non-recursive solution is still superior.

### Skeleton Program

The skeleton program **MaxSubSum.java** is given. It contains the first two algorithms.

### Test Data

Eight test data sets are given. Algorithm 1 could only work in a reasonable time for the first 3 sets, and algorithm 2 for the first 5 sets. Only an algorithm with running time of  $O(n)$  would be able to solve the last 3 sets in a reasonably short time.

### Sample Run #1

```
Enter size of array: 6
Enter 6 integers: -2 11 -4 13 -5 -2
Answer = 20
```

### Sample Run #2

```
Enter size of array: 10
Enter 10 integers: -1362 4751 4614 -1353 147 -1155 163
-2502 -1727 3328
Answer = 9365
```

### Hint for algorithm 3:

If the subsequence sum is negative, what should you do?