Writing efficient algorithms
Maximal Contiguous Subsequent Sum Problem

Maximum Contiguous Subsequence Sum: given (a possibly negative) integers A_1, A_2, …, A_N, find (and identify the sequence corresponding to) the maximum value of

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

For the degenerate case when all of the integers are negative, the maximum contiguous subsequence sum is zero.

Examples:

If input is: {-2, <u>11, -4, 13</u>, -5, 2}. Then the output is: 20.

If the input is {1, -3, <u>4, -2, -1, 6</u>}. Then the output is 7.

In the degenerative case, since the sum is defined as zero, the subsequence is an empty string. An empty subsequence is contiguous and clearly, 0 > any negative number, so zero is the maximum contiguous subseqeunce sum.

The $O(N^3)$ Algorithm (brute force method)

```java
public static int MCSS(int [] a) {

   int max = 0, sum = 0, start = 0, end = 0;

   // Cycle through all possible values of start and end indexes
   // for the sum.
   for (i = 0; i < a.length; i++) {
      for (j = i; j < a.length; j++) {
         sum = 0;

         // Find sum A[i] to A[j].
         for (k = i; k <= j; k++)
            sum += a[k];
         if (sum > max) {
            max = sum;
            start = i; // Although method doesn't return these
            end = j;  // they can be computed.
         }
      }
   }
   return max;
}
```

General Observation Analysis

Look at the three loops: the *i* loop executes *SIZE* (or N) times. The *j* loop executes *SIZE-1* (or N-1) times. The *k* loop executes *SIZE-1* times in the worst case (when i = 0). This gives a rough estimate that the algorithm is $O(N^3)$.

Precise Analysis Using Big-Oh Notation

In all cases the number of times that, sum += a[k], is executed is equal to the number of ordered triplets (i, j, k) where $1 \leq i \leq k \leq j \leq N^2$ (since i runs over the whole index, j runs from i to the end, and k runs from i to j). Therefore, since i, j, k, can each only assume 1 of n values, we know that the number of triplets must be less than n(n)(n) = N^3 but $i \leq k \leq j$ restricts this even further. By combinatorics it can be proven that the number of ordered triplets is n(n+1)(n+2)/6. Therefore, the algorithm is $O(N^3)$.

A Simple Big-Oh Rule

A Big-Oh estimate of the running time is determined by multiplying the size of all the nested loops together. BUT, THERE ARE EXCEPTIONS TO THIS RULE!!!

The O(N²) Algorithm

<u>Algorithm</u>
```
public static int MCSS(int [] a) {
    int max = 0, sum = 0, start = 0, end = 0;
    // Try all possible values of start and end indexes for the sum.
    for (i = 0; i < a.length; i++) {
        sum = 0;
        for (j = i; j < a.length; j++) {
            sum += a[j]; // No need to re-add all values.
            if (sum > max) {
                max = sum;
                start = i; // Although method doesn't return these
                end = j;  // they can be computed.
            }
        }
    }
    return max;
}
```

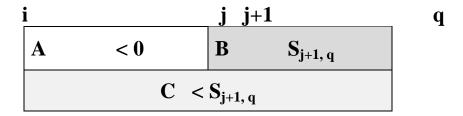<u>Discussion of the technique and analysis</u>
We would like to improve this algorithm to run in time better than $O(N^3)$. To do this we need to remove a loop! The question then becomes, "how do we remove one of the loops?" In general, by looking for uncessary calculations, in this specific case, uncessary calculations are performed in the innerloop. The sum for the subsequence extending from i to $j – 1$ was just calculated – so calculating the sum of the sequence from i to j shouldn't take long because all that is required is that you add one more term to the previous sum (i.e., add A_j). However, the cubic algorithm throws away all of this previous information and must recompute the entire sequence!

Mathematically, we are utilizing: $\sum_{k=i}^{j} A_k = (\sum_{k=i}^{j-1} A_k) + A_j$.

The O(N) Algorithm (A linear algorithm)

Discussion of the technique and analysis

To further streamline this algorithm from a quadratic one to a linear one will require the removal of yet another loop. Getting rid of another loop will not be as simple as was the first loop removal. The problem with the quadratic algorithm is that it is still an exhaustive search, we've simply reduced the cost of computing the last subsequence down to a constant time (O(1)) compared with the linear time (O(N)) for this calculation in the cubic algorithm. The only way to obtain a subquadratic bound for this algorithm is to narrow the search space by eliminating from consideration a large number of subsequences that cannot possibly affect the maximum value.

How to eliminate subsequences from consideration

If A < 0 then C < B

If $\displaystyle\sum_{k=i}^{j} A_k < 0$, and if q > j, then $A_i...A_q$ is not the MCSS!

Basically if you take the sum from A_i to A_q and get rid of the first terms from A_i to A_j your sum increases!!! Thus, in this situation the sum from A_{j+1} to A_q must be greater than the sum from A_i to A_q. So, no subsequence that starts from index i and ends after index j has to be considered.

So – if we test for sum < 0 and it is – then we can break out of the inner loop. However, this is not sufficient for reducing the running time below quadratic!

Now, using the fact above and one more observation, we can create a O(n) algorithm to solve the problem.

If we start computing sums $\sum_{k=i}^{i} A_k$, $\sum_{k=i}^{i+1} A_k$, etc. until we find

the first value j such that $\sum_{k=i}^{j} A_k < 0$, then immediately we

know that either

1) The MCSS is contained entirely in between A_i to A_{j-1} OR
2) The MCSS starts before A_i or after A_j.

From this, we can also deduce that unless there exists a subsequence that starts at the beginning that is negative, the MCSS MUST start at the beginning. If it does not start at the beginning, then it MUST start after the point at which the sum from the beginning to a certain point is negative.

So, using this how can we come up with an algorithm?

1) We can compute intermediate sums starting at i=0.
2) When a new value is added, adjust the MCSS accordingly.
3) If the running sum ever drops below 0, we KNOW that if there is a new MCSS than what has already been calculated, it will start AFTER index j, where j is the first time the sum dropped below zero.
4) So now, just start the new running sum from j+1.

Algorithm

```
public static int MCSS(int [] a) {

    int max = 0, sum = 0, start = 0, end = 0, i=0;

    // Cycle through all possible end indexes.
    for (j = 0; j < a.length; j++) {

        sum += a[j]; // No need to re-add all values.
        if (sum > max) {
            max = sum;
            start = i; // Although method doesn't return these
            end = j;  // they can be computed.
        }
        else if (sum < 0) {
            i = j+1; // Only possible MCSSs start with an index >j.
            sum = 0; // Reset running sum.
        }
    }
    return max;
}
```

Discussion of running time analysis

The *j* loop runs N times and the body of the loop contains only constant time operations, therefore the algorithm is O(N).

MCSS Linear Algorithm Clarification

Whenever a subsequence is encountered which has a negative sum – the next subsequence to examine can begin after the end of the subsequence which produced the negative sum. In other words, there is no starting point in that subsequence which will generate a positive sum and thus, they can all be ignored.

To illustrate this, consider the example with the values

5, 7, -3, 1, -11, 8, 12

You'll notice that the sums

5,   5+7,   5+7+(-3) and   5+7+(-3)+1   are positive, but

5+7+(-3)+1+(-11) is negative.

It must be the case that all subsequences that start with a value in between the 5 and -11 and end with the -11 have a negative sum. Consider the following sums:

7+(-3)+1+(-11)   (-3)+1+(-11)   1+(-11)   (-11)

Notice that if any of these were positive, then the subsequence starting at 5 and ending at -11 would have to be also. (Because all we have done is stripped the initial positive subsequence starting at 5 in the subsequences above.) Since ALL of these are negative, it follows that NOW MCSS could start at any value in between 5 and -11 that has not been computed.

Thus, it is perfectly fine, at this stage, to only consider sequences starting at 8 to compare to the previous maximum sequence of 5, 7, -3, and 1.