Robert Ottolia
Myles Chatman
Andrew Brown
CS 325 Algorithms
Group 26 Winter 2016

Project 1: Maximum Sum Subarray

1. Theoretical Run-Time Analysis

1.1 **Enumeration Algorithm**

Pseudocode:

```
maxSum ← 0
for i = 0 to size of array do
        for j = i to size of array do
                newSum ← sum of array from index i to index j
                if newSum > maxSum then
                        maxSum ← newSum
                end if
        end for
end for
return maxSum
```

Analysis:

There is no recursion in this algorithm. There is n*n computation done for the nested for-loops. Additionally, there is a summing up of n items within the innermost for-loops. Thus, the running time is $\theta(n^3)$. This is a tight bound because we must sum up all elements in the array in order to accurately gauge the greatest sum.

1.2 **Better Enumeration Algorithm**

Pseudocode:

```
maxSum ← 0
for i = 0 to size of the array do
        oldSum ← 0
        for j = i to size of array do
                if j = 0 then
                        newSum ← content of the array at index i
                else
                        newSum ← oldSum + content of the array at index j
                end if
                if newSum > maxSum then
                        maxSum ← newSum
                end if
                oldSum ← newSum
        end for
end for
return maxSum
```

Analysis:
This algorithm removes the implicit third loop adding up the array elements from index *i* to index *j* in the first algorithm. Thus, this algorithm has a better complexity of $\boldsymbol{\theta}(n^2)$.

1.3 **Divide and Conquer Algorithm**

Pseudocode:

```
function maxSuffix(array):
        reverseArray ← array is reversed
        max ← first element of reversed array
        sum ←  0
        for i = 0 to end of reverseArray do
                sum← sum + i
                if sum > max then
                        max← sum
                endif
        end for
return max

function maxPrefix(array):
        max← first element of array
        sum← 0
        for i = 0 to end of array do
                sum← sum + i
                if sum > max then
                        max← sum
                end if
        end for
return max

function divideAndConquer(array):
        if size of array <= 1 then
                return first element of array
        else
                firstHalf ←first half of array
                secondHalf← second half of array
                first← divideAndConquer(firstHalf)
                last← divideAndConquer(secondHalf)
                middle← maxSuffix(firstHalf) + maxPrefix(secondHalf)
                return max(first, last, middle)
        end if
```

Analysis:
Assuming the size of the input array is > 1, the division of the array into two halves takes $\boldsymbol{\theta}(1)$ time. From there, the algorithm has to solve two subarrays of $n/2$. This takes $2T(n/2)$ time. We then must look at the case when the maximum subarray could be contained in the suffix of the first half and a prefix of the second half. The two helper algorithms in the code determine the max array sum in these two pieces. It is clear that these two algorithms have to loop through n elements, adding a complexity of $\boldsymbol{\theta}$ (n), and then the calling function has to combine the results which is a $\boldsymbol{\theta}(1)$ operation.

Therefore, the overall recurrence is:

$T(n) = 2T(n/2) + \boldsymbol{\theta}(n) + \boldsymbol{\theta}(n) + \boldsymbol{\theta}(1)$

Combining and dropping constants/low order terms we get the recurrence:
$T(n) = \boldsymbol{\theta}(1)$, if $n = 1$, and
$T(n) = 2T(n/2) + \boldsymbol{\theta}(n)$ if $n > 1$

By using the master theorem, the solution to the recurrence is $\boldsymbol{\theta}(n \log n)$.

1.4 **Linear-Time Algorithm**

Pseudocode:

bestSum ← *–MAX* (lowest possible value that can be stored in type int)
bestStart ← *bestEnd* = -1
localStart ← *localSum* = 0
for *i* = 0 to size of array - 1 **do:**
    localSum ← *localSum* + value at index *i*
    if *localSum* > *bestSum* **then**
        bestSum ← *localSum*
        bestStart ← *localStart*
        bestEnd ← *i*
    end if
    if *localSum* <= *0* **then**
        localSum ← *0*
        localStart ← *i* + 1

Analysis:

This algorithm iterates through the array, keeping a tally of the current subarray. If the current subarray sum is greater than zero, it will contribute to future subset sums, so it is kept. Otherwise, if the current subset sum is less than or equal to zero, it will not contribute to any future subarray sums so it is discarded and the algorithm starts over with a new sum. From there, all that must be done is update the maximum sum that has been encountered.

The algorithm has one for-loop, looping over all of the elements of the array. The addition of variables are all of complexity $\boldsymbol{\theta}(1)$, so the overall complexity of this algorithm is $\boldsymbol{\theta}(n)$.