Robert Ottolia

Myles Chatman

Andrew Brown

CS 325 Algorithms

Group 26 Winter 2016

Project 1: Maximum Sum Subarray

**1. Theoretical Run-Time Analysis**

* 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 θ(n3). This is a tight bound because we must sum up all elements in the array in order to accurately gauge the greatest sum.

* 1. **Better Enumeration Analysis**

**Pseudocode:**

*maxSum* 0

**for** i = 0 to size of array **do**

**for** j = i to size of array **do**

*tempSum += array[j]*

**if** *maxSum > tempSum* **then**

*tempSum**maxSum*

**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. In side the loop we have a constant which is tempSum += array[j], and then the max of tempSum and maxSum is kept. Since this is a constant, the running time for this algorithm is O(n^2).

* 1. **Divide and Conquer**

**divideConquer(array, left, right)**

**divideConquer(array[1…n/2])**

**divideConquer{array[n/2…n])**

**loop for n to find max sub of both left and right**

**return max(leftsub, rightsub, leftsub+rightsub)**

**Analysis:**

This algorithm is recursive, but you are still looping for n. So it is O(n) for the loop X O(log n) for the recursion, coming out to a running time of O(n log n)

* 1. **Linear Time**