CS 325 Project 1

Group Members
Stephen McGuckin
Ryan Peterson
Brandon Swanson

Theoretical Runtime Analysis

Algorithm 1
Pseudo code:

```
maxsum = -INFINITY
for i from 0 to end of array index
        for J = i to the end of array index
                 currentsum = 0
                for k = i while k <= J
                        current sum = currentsum + array[k]
                if currentsum > maxsum then maxsum = currentsum
```

The statement in the outermost loop is computed exactly N times (N = size of the array). The middle loop is also executed at most N times in each iteration of the outer loop. The inner loop is never executed more than n times. Multiplying these 3 together we get that this algorithm's cost is Θ(n^3).

Algorithm 2
Pseudo code:

```
maxsum = -INFINITY
for  i from 0 to end of array
        currentsum = 0
        for J = i till the end of array
                currentsum = currentsum + array[ J ]
        if currentsum > maxsum then maxsum = currentsum
```

The statement in the outer loop is executed exactly n times. And the inner loop is executed at most n times therefore this algorithm is Θ(n^2).

The inner loop will run n times on its first iteration then n-1, n-2, until it runs 1 time. the sum of these iterations is the sum of numbers 1 through n sequence which is equal to n(n+1)/2 = (n^2 + n) /2 which asymptotically is Θ(n^2)

Algorithm 3

Pseudo code:

```
maxmiddle(ARRAY[],low, mid, end) {

        leftsum = -INFINITY
        For i = mid downto low.
                sum=sum+ARRAY[i]
                if sum>leftsum
                        lefsum = sum
                        max-left = i

        rightsum = -INFINITY
        For i = mid + 1 upto high.
                sum=sum+ARRAY[i]
                if sum>rightsum
                        rightsum = sum
                        max-right = i

        return maxl-left,max-right, leftsum+rightsum

 }

maxsubarray3(ARRAY[],low,high) {
        if low = high we have one element return indexes low  mid high

        else
                mid = middle of array
                return the biggest value of the 3(
                        1. max_middle(array[], low,mid,high) //max of anything in the middle
                        2. maxsubarray3(array[],low,mid) // max from the beginning  to middle
                        3. maxsubarray3(array[],mid+1,high) // max from the beginning +1 to end
                )
}
```

by examining the recursion tree we can evaluate that this algorithm does Θ((n) for each level of recursion. there are lg n levels of recursion. Therefore we get Θ(n lg n). We can further prove this by examining the recurrence and applying the master theorem.
Because each call to the algorithm makes a recursive calls on one half of the subarray each, and calls a function that sums all the values of the subarray received by the algorithm, The recurrence can be stated thusly:

$T(n) = 1 \quad for\ n = 1$
$T(n) = 2T(n/2) + n$

This fits the masters theorem format and
$f(n) = \Theta(n^{log_2 2} * log^0 n) = \Theta(n)$ therefore T(n) = Θ(nlgn)

Algorithm 4

Pseudo code:

for i from 0 to size of array

max = max + the value of element in array at index i

if current max <= 0

current max = 0

keep track of greatest max seen so far

This loop only goes through array exactly n times. This is a linear time algorithm and cost Θ(n).

Proof of Correctness

Problem: Find the subarray of array a that has the greatest sum

Input: a, a start and end point

Output: the start point, end point and sum of the max subarray

Proving max

Precondition: a and b have valid max values

Postcondition: return value is whichever of a and b has the highest max value

Proof:

1. a.max > b.max is true if a has a greater max value than b
2. if (a.max > b.max) return a returns a if a has a greater max value than b
3. else return b returns be if b has a greater max value than a

Proving threemax

Precondition: a, b and c all have valid max values

Postcondition: return value is whichever maxij has the greatest max value

Proof:

1. max(a,b) returns a or b with greatest max
2. max(max(a,b),c) returns greatest between c and max(a,b);
3. return max(max(a,b),c) returns greatest between a,b and c

Proving max_middle

Precondition: a is a valid array, start = x, mid = y, end = z

Postcondition: result represents the subarray with the largest sum that crosses the midpoint

Proof:

Proving for (i = mid-1; i >= start; i--)

Input: an array a, int sum, int left_sum, int start, maxij result

Output: result.i = low bound for the subarray with the greatest sum with mid as it's upper

bound and left_sum = sum of subarray bound by result.i and mid

Loop Invariant: At step k at the start of each loop sum equals the sum of the values of a from
mid to mid-k, left_sum equals the greatest sum found between i=mid and i=mid-k and result.i = the i corresponding to left_sum
Initialization:
k = 0, sum = a[mid], left_sum = a[mid], result.i = 0. Everything is still where it was at initialization
Maintenance:
i = mid-(k+1), sum = sum(from k) + a[i], left_sum = left_sum(from k) unless sum is greater
then left_sum =sum, the same goes for result.i - if sum is less than left_sum(from k) then
result.i = result.i(from k) otherwise result.i = i
Termination:
The loop terminates when mid-k <= start

Proving for (i = mid+1; i <= end; i++)

Input: an array a, int sum, int left_sum, int end, maxij result
Output: result.j = high bound for the subarray with the greatest sum with mid as it's lower
bound and left_sum = sum of subarray bound by result.j and mid
Loop Invariant: At step k at the start of each loop sum equals the sum of the values of a from
mid+1 to mid+k when k > 0, right_sum equals the greatest sum found between i=mid and
i=mid-k and result.j = the i coresponding to right_sum
Initialization:
k = 0, sum = 0, right_sum = 0, result.j = 0.
Maintenance:
i = mid+1+(k+1), sum = sum(from k) + a[i], right_sum = right_sum(from k) unless sum is
greater then right_sum =sum, the same goes for i - if sum is less than right_sum(from k) then
result.j = result.j(from k) otherwise result.j = j
Termination:
The loop terminates when mid+1+k >= end

The two loops (proved above) set result.j and result.i and set left_sum and right_sum
The final step sets result.max to left_sum + right_sum.
left_sum = greatest sum begin to mid with mid as upper bound
right_sum = greatest sum mid to end with mid as lower bound
result.max = greatest sum including mid

Proving r_maxsubarray_3

Input: array a, int low and int high

Output: a maxij representing the subarray that sums to the greatest amount
Base Case: If low = high then the subarray is of size one and a maxij is returned in the form {low,high,a[low]}
Inductive Hypothesis: r_maxsubarray_3 finds the maximum subarray of an array (or subarray) where (high - low) = k
Inductive Step: (high - low) = k + 1
First recursive call takes the array a, int low and the calculated middle (as int high) as input and outputs max subarray of low+1 to mid
Second recursive call takes the array a, the calculated middle plus one (as int low) and int high as input and outputs max subarray of mid+1 to high
max_middle takes array a, int low, the calculated middle as mid and int high as inputs and outputs max subarray including mid
threemax outputs max subarray of the three
Greatest possible subarray is returned
Termination:
Every recursive call halves (high - low) and the recursion terminates at (high - low = 0)

Testing

We tested the algorithms by creating a compiled program that would output the result of each algorithm to a different file each. reading from the MSS_Problems.txt file provided and outputting in the same format as the provided MSS_Results.txt file. Each output could then be automatically compared to the results file using diff. Algorithm 3 sometimes found a different optimal solution but on these small samples could still be verified to be a correct solution. After testing was concluded on these provided tests the algorithms were then tested on larger randomly generated values and then their outputs were compared with each other.

Experimental Analysis

Data for 100 values of n for algorithms 2,3,and 4 and 37 values of n for algorithm 1 are attached as well as plots and fit lines.

The only algorithm with any significant discrepancy from the expected trend line was the 4th algorithm, the linear time algorithm. We believe this was caused by the two conditionals within this algorithm's loop used for tracking the best sum discovered so far and the beginning and end points of that subarray. Each of these conditionals involve variable assignments but are not always triggered on each iteration of the loop. Therefore different arrays with different distributions of positive and negative numbers would cause more variable assignments therefore increasing the time the algorithm takes to finish. This accounts for the distribution of the measured time across both sides of the linear trendline.

Regression model and 10 minute size of N
(functions generated and solved using Matlab and Wolfram Alpha)

Algorithm 1:
Regression Function:

(1351074808815051*n^3)/2417851639229258349412352 -
(4764799537088881*n^2)/1888946593147858085478 4 +
(5047297659782551*n)/18446744073709551616 - 1089898065277315/18014398509481984

Largest value of N under 10 minutes:
10377

Algorithm 2:
Regression Function:
(7022048938385443*n^2)/4835703278458516698824704 -
(1995149045137059*n)/75557863725914323419136 +
8321871326131415/295147905179352825856

Largest value of N under 10 minutes:
642805

Algorithm 3:
Regression Function:
(1355390088139563*n^2)/6189700196426901374495621 12 +
(1535378422870539*n)/1888946593147858085478 4 +
2841859988135817/73786976294838206464

Largest value of N under 10 minutes:
16534510

Algorithm 4:
Regression Function:
(2154593805482993*n)/6044629098073145873530 88 +
4781038308868933/590295810358705651712

Largest value of N under 10 minutes:
168327663463