Project Report

Grant Nakashima, Ben Beckerman, and Jessica Siano

Group 14

Theoretical Run-Time Analysis

Algorithm 1:

Pseudocode

```
max_sum = -infinity
for i = 0; i < n; i++
   for j = i; j < n; k++
      current_sum = 0
      for k = i; k <= j; j++
         current_sum += array[k]
      if current_sum > max_sum
         max_sum = current_sum
return max_sum
```

Run Time Analysis

The run time of this algorithm is easy to calculate. There are three loops that potentially iterate through all of n. Each loop is nested inside of each other, resulting in O(n * n * n) or O(n^2) time

Algorithm 2:

Pseudocode

```
max_sum = -infinity
for i = 0; i < n; i++
   current_sum = 0
   for j = i; j < n; k++
      current_sum += array[j]
      if current_sum > max_sum
         max_sum = current_sum
return max_sum
```

Run Time Analysis

The run time of this algorithm is similar to that of algorithm one. However, it eliminates the inner most loop, resulting in time of O(n * n) or O(n^2) time.

Algorithm 3:

Pseudocode

```
maxSumWithMid(vector <int> num, int left, int right, int*total, int *low, int* high)
sum = 0
sumL = 0
sumR = 0
mid = (left+right) / 2

for I = mid down to left
```

```
        tempSum = num.at(i)
                if sum > sumL
                        then low = I, sumL = tempSum
sum = 0

for I = mid+1 up to right
        tempSum = num.at(i)
                if sum >SumR
                        then high = I, sumR = tempSum
total = sumL + sumR

hleperAlg3(vector<int>num, int left, int right, int* total, int *leftIndex, int *rightIndex)

if left == right
    total = num.at(left)
    leftIndex = left
    rightIndex = right

else
    mid = (left+right) /2
leftLow, leftHigh, leftSum, rightLow, rightHigh, rightSum, midLow, midHigh, midSum

call helperAlg3(num,left,mid,&leftSum,&leftLow,&leftHigh)
call helperAlg3(num,mid+1,right,&rightSum,&rightLow,&rightHigh)
call maxSumWithMid(num,left,right,&midSum,&midLow, &midHigh)

if (leftSum > rightSum and leftSum > midSum)
    total = leftSum
    leftIndex = leftLow
    rightIndex = leftHigh
if (leftSum < rightSum and rightSum > midSum)
    total = rightSum
    leftIndex = rightLow
    rightIndex = rightHigh
if (midSum > rightSum and leftSum < midSum)
    total = midSum
    leftIndex = midLow
    rightIndex = midHigh

alg3(vector<int>num)


    tempVector, low, high, total, leftIndex,rightIndex

helperAlg3(nums,low,high,&total,&leftIndex,&rightIndex)

for(I =leftIndex to right index)
        push into tempVector
```

return tempVector

Run Time Analysis

The functions has two loops and it runs over the entire vector's elements. All of it involves constant time operations so the worst case running time for maxSumWithMid is theta(n). Finding the recurrence relationship of helperAlg3 we get T(n) = theta(1) if n =1 and 2T(n/2) + theta(n) where n >= 2. The recursive part of the function is 2T(n/2) and the remaining part of the function is theta(n).

Solve 2T(n/2) + theta(n).

Guess T(n) = O(n log n)

<= 2 [c(n/2)log(n/2)] + n *2 cancel

<= cnlog(n/2) + n

= cn log(n) – cn log(2) + n

If c >=1 then T(n) <= cnlogn

Algorithm 4:

Pseudocode

```
vector<int> alg4(vector<int> nums)
    int sum, maxSum, endHereSum, endHereLow, endHereHigh, start, end
    for(every element in nums)
        endHereHigh = current index
        if endHereSum > 0
            endHereSum += current element
        else
            endHereLow = current index
            endHereSum = current element
        if endHereSum > maxSum
            maxSum = endHereSum
            start = endHereLow
            end = endHereHigh

        vector<int> maxSub
        for(starting at start, ending at end)
        maxSub.pushBack(current element in original array)
        return maxSub
```

Run Time Analysis

Since this algorithm loops over the given array once, it runs in O(n) time.

Proof of Correctness

Hypothesis: suppose at any iteration of the loop, leftIndex and rightIndex still enclose the target elements
Inductive case: If leftIndex+1 < rightIndex we update leftIndex and go to the next element. If leftIndex+1 > rightIndex we terminate the loop. As leftIndex increases, it gets closer to rightIndex until leftIndex=rightIndex. In other words, leftIndex <= target <= rightIndex.
Termination: loop terminates when leftIndex = rightIndex.

maxSumWithMid

maxSumWithMid is separated into two parts, mid to left and from mid to right. We loop through both cases to find and store the maximum subarray sum. In the first loop for example, we initialize the left side of I to equal mid and decrement to left. As it loops through the first half of the array/vector, tempSum will get summed up. If tempSum is greater than leftSum, the old estimate for the endpoint and leftSum are updated. This is done on the right side of the array/vector. We evaluate the sum by looping over the vector's elements and storing the current estimate. Once both sums are found, we can find the correct maximum subarray sum by adding them together. Once the correct sum is found, we will reach the end of the function and will terminate.

helperAlg3

Precondition: array has at least on element between left and right.
Post condition: indexes of the max sum given
Base case: N = 1, contains a single element
Inductive hypothesis: assume helperAlg3 correctly determines the max in it's set of elements.
Inductive step:

- 1st recursive call: from subarray a[p..q] max found
- 2nd recursive call: from subarray a[q+1 …r] max found
- Maxsumwithmid called to get 3rd max
- Now we apply the inductive hypothesis and determine what the maximum is.
- Compare max of all three sides with each other. For example, compare if left> mid and left > right. If it passes this condition it will check if it is the highest total. The greatest value will get stored (indexes)

Termination: There are two ways for the function to terminate.

- If the left and right values are equal, the function will return the value at left and terminate.
- If the functions reaches the conditions part of the function, it will perform the if/else conditions and terminated once it has completed the evaluations.

Show n = k + 1
Inductive Hypothesis: helperAlg3 will find the indexes of max sum.
Inductive Step: Show helperAlg3 correctly finds the indexes.

- If the left and right are equal the total will get returned and the function will terminate. This is one of the ways the function terminates.

Lets make the first sub array be L{low, low+1…mid} and the second sub array be R{mid+1, mid +2 … high} . We see that L < n and R < n.

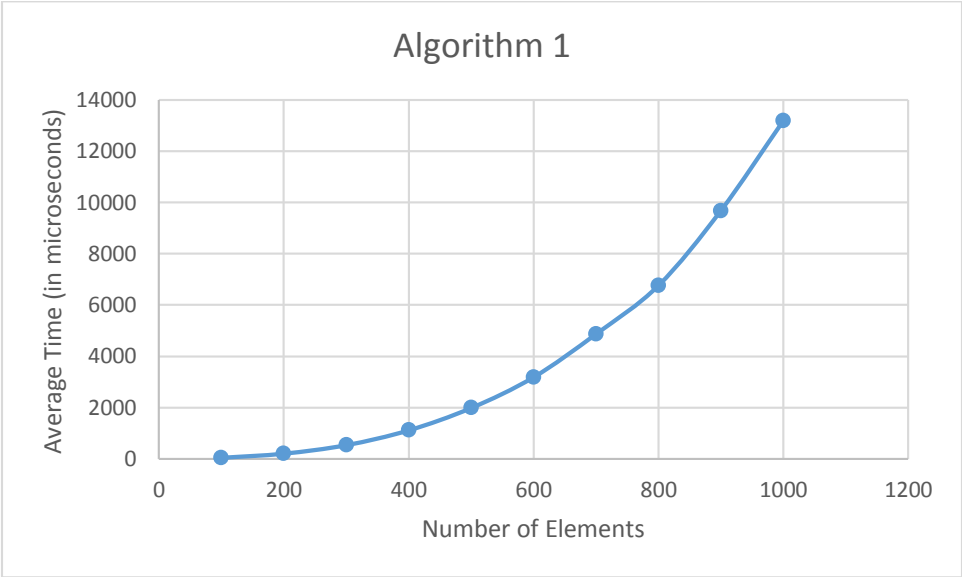
Testing

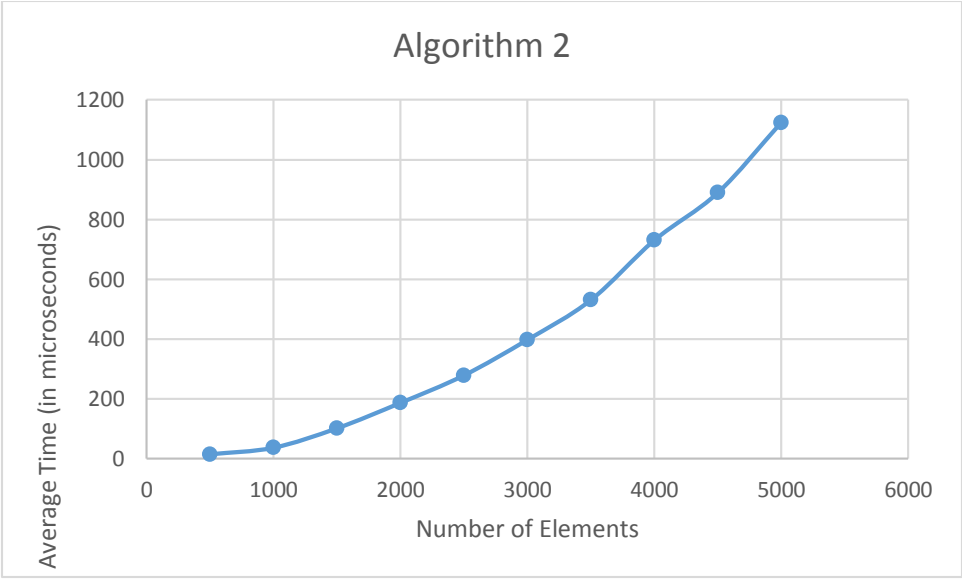
All algorithms were tested for correctness on the MSS_Problems.txt file, and produced the correct results for those arrays. At first, we noticed that we had some issues with the output for Algorithms 3 and 4. In Algorithm 3, there were some small problems in the code that were easily fixable. In algorithm 4, we were originally going off of pseudo code from a different website, but that code was not running properly. We tweaked it once or twice but it still wasn't working, so we scrapped it and went with a different algorithm. Once we did that, it was up and running with no problems. After that, all of the algorithms were run several times and by each member to make sure the results were consistent.

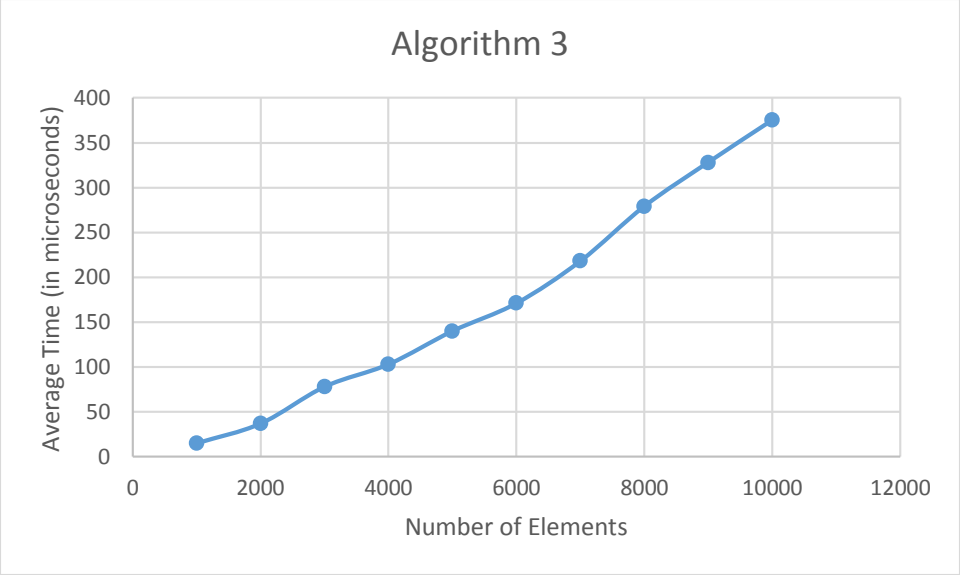
The algorithms were also tested on randomly generated arrays of sizes 100 and 1000. The algorithms also produced the correct results for those array once we got the bugs worked out of our code.

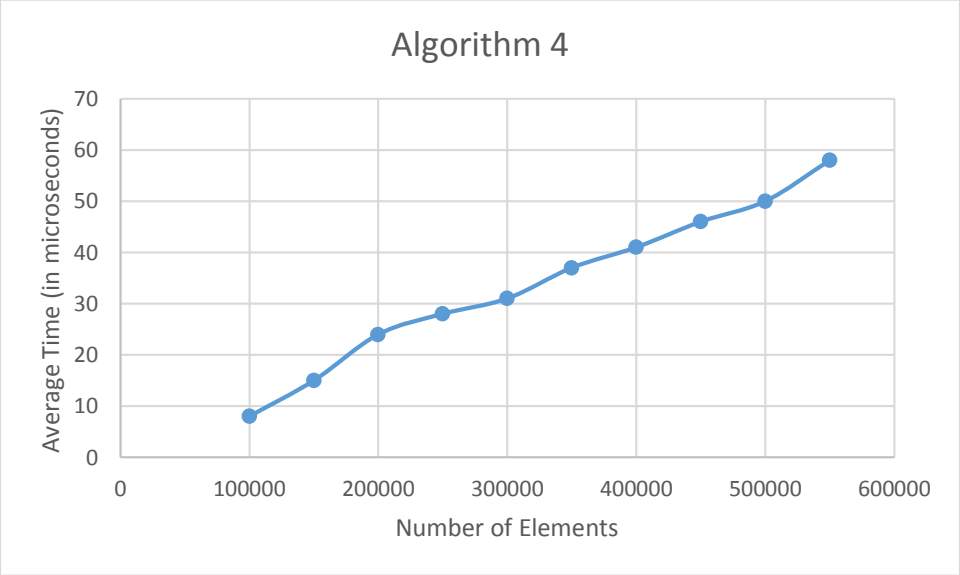
Experimental Analysis

1) The average run time for algorithm 1 with 1000 elements was 13188 microseconds, for algorithm 2 with 5000 elements was 1125 microseconds, for algorithm 3 with 10000 elements was 375 microseconds, and for algorithm 4 with 200000 elements was 24 microseconds.
2) Plots of the data:

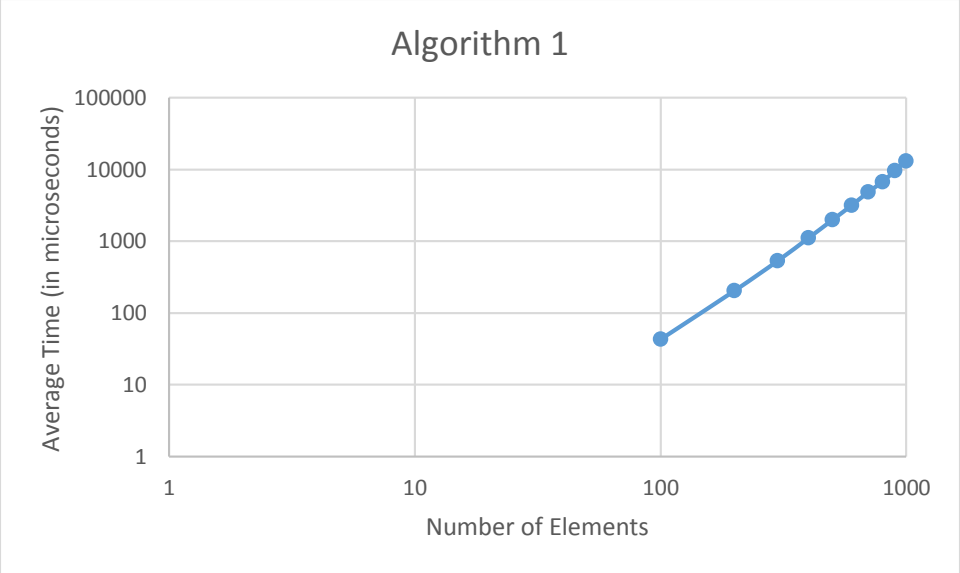


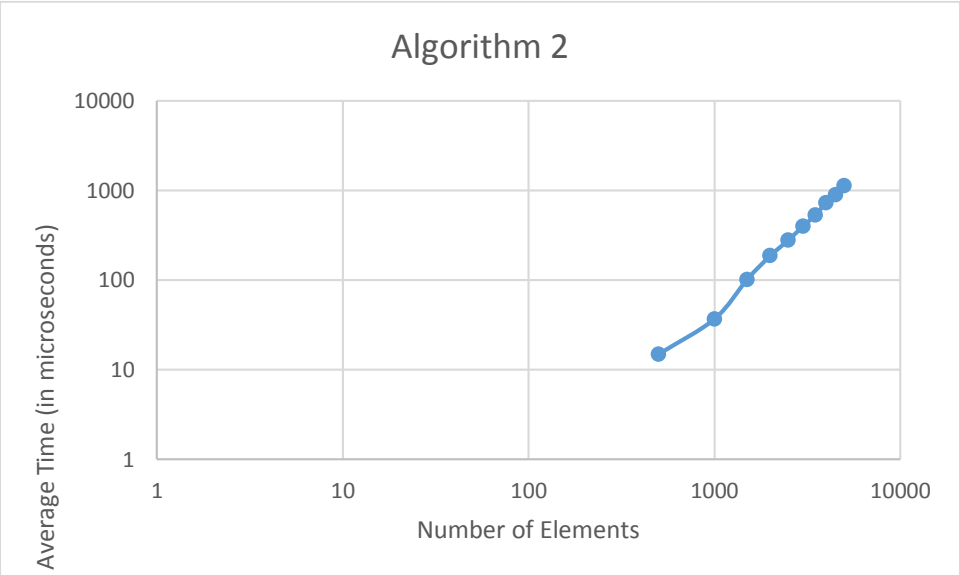


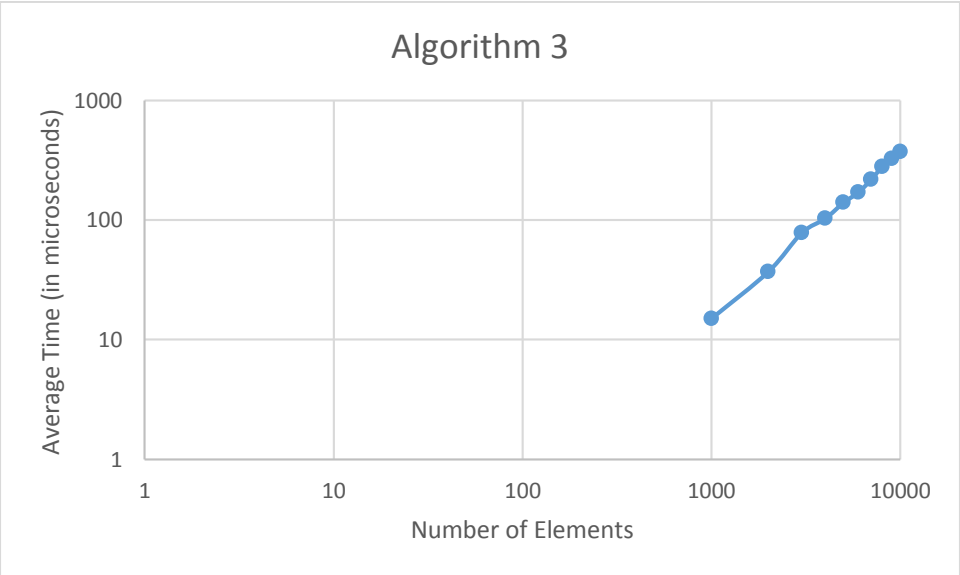


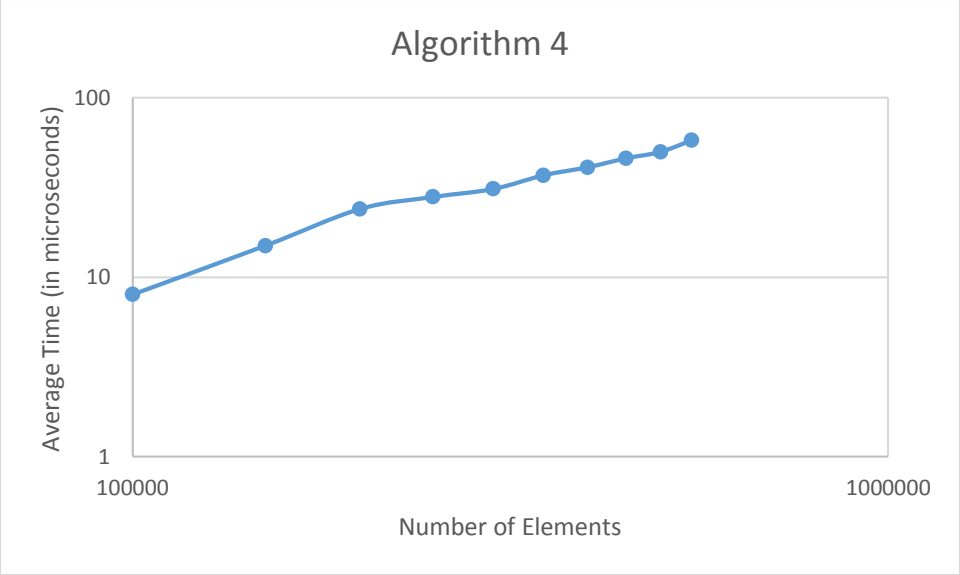


3) The cubic regression for algorithm 1 is: $y = .000013737x^3 - .0025x^2 + 2.1x - 192.7$
   The quadratic regression for algorithm 2 is: $y = .00004544x^2 - .00434x + 4.183$
   The linear regression for algorithm 3 is: $y = .04035x - 47.53$
   The logarithmic regression for algorithm 4 is $y = -23.452 + 4.64 \ln x$
4) Algorithms 1 and 2 did pretty much exactly what we were expecting them to, but 3 and 4 were a little more scattered. If you look at the graphs you can see that the curves aren't as smooth, indicated that sometimes the average running times were a little higher or lower than expected, but they were not that far off. Overall, our experimental run times were in line with what we expected to get, theoretically.
5) Based on the data above and a run time of 10 minutes, algorithm 1 could process approximately 35276 elements, algorithm 2 could process approximately 3.63381×10^6 elements, algorithm 3 could process approximately 1.48699×10^{10} elements, and algorithm 4 could process approximately $2.55 \times 10^{56158771}$ elements.
6) The log-log graphs are:

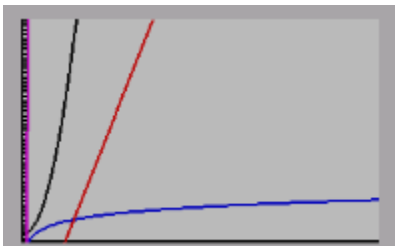

Algorithm 1
Average Time (in microseconds)
100000
10000
1000
100
10
1
1
10
100
1000
Number of Elements


Algorithm 2
Average Time (in microseconds)
10000
1000
100
10
1
1
10
100
1000
10000
Number of Elements


Algorithm 3
Average Time (in microseconds)
1000
100
10
1
1
10
100
1000
10000
Number of Elements



7) All 4 run times, on one graph, would look like:



Where algorithm 1 is in pink, 2 is in black, 3 is in red, and 4 is in blue.