Project 1 - Max Sum Subarray Report

Project Group 7: Larissa Hahn, Allison Barnett, and Nathan Thunem

July 12, 2015

I. Theoretical Runtime Analysis

Algorithm 1: Enumeration

```
max_enumeration_subarray(array, count) {
  for i=0 up to count
      for j=i up to count
          sum = 0
          for k=i up to and including j
              sum += array[k]
          if sum > best
              best = sum
              subarray.leftIdx = i
              subarray.rightIdx = j

  subarray.maxSum = best
  return subarray
}
```

Asymptotic Analysis for Algorithm 1:

The outer loop for this algorithm takes n amount of work to compute an array of size n. The first inner loop takes n-i amount of work. The third loop takes n-i-j.

Therefore,

T(n)=n(n-i)(n-j-i)

=(n^2-ni)(n-j-i)

=n^3-n^2j-n^2i-n^2i+nij+ni^2

Therefore,

T(n) = **Θ(n^3)**

Algorithm 2: Better Enumeration

```
SubArray max_ better_enumeration_subarray(array, count) {
  for i=0 up to count
      sum = 0
      for j=i up to count
```

```
            sum += array[j]
            if sum > best
                  best = sum
                  subarray.leftIdx = i
                  subarray.rightIdx = j
     subarray.maxSum = best
   return subarray
}
```

Asymptotic Analysis for Algorithm 2:
The outer loop takes n amount of work for an array of size n. The inner loop takes a total of n-i amount of work.

Therefore, the total amount of work is:

T(n)=n(n-i)
     =n^2-ni

Therefore, this algorithm has a complexity of **Θ(n^2)**.

Algorithm 3: Divide and Conquer

```
SubArray find_max_crossing_subarray(array, low, mid, high) {
          //sources: "Intro. to Algorithms 3rd ed." (Cormen, Leiserson, Rivest, Stein) p.71

          SubArray subarray;   //make a struct to store results

         left_sum = -infinity
         sum = 0
          for i=mid downto low
                 sum = sum + array[ i ]
                 if sum > left_sum
                        left_sum = sum
                       subarray.leftIdx = i
          right_sum = -infinity
          sum = 0
          for j= mid+1 to high
                   sum = sum + array[ i ]
                   if sum > right_sum
                          right_sum = sum
                          subarray.rightIdx = j
          subarray.maxsum = left_sum + right_sum

          return subarray
}

SubArray maxSubArraySum (array, low, high){
```

```
        // Base Case: Only one element
        if (low == high)
                SubArray oneElement;
                oneElement.leftIdx = low;
                oneElement.rightIdx = low;
                oneElement.maxSum = array[low];
               return oneElement;

        //find middle point
        mid = (low + high)/2;

        SubArray leftHalf = maxSubArraySum(array, low, mid);
        SubArray rightHalf = maxSubArraySum(array, mid+1, high);
        SubArray crossing = find_max_crossing_subarray(array, low, mid, high);

   if (leftHalf.maxSum >= rightHalf.maxSum && leftHalf.maxSum >= crossing.maxSum)
                return leftHalf;
   else if (rightHalf.maxSum >= leftHalf.maxSum && rightHalf.maxSum >= crossing.maxSum)
                return rightHalf;
   else
                return crossing;
}
```

<u>*Asymptotic Analysis for Algorithm 3*</u>:

For the base case, when n=1, the division of the subarray takes Θ(1) time. When n>1, the recursive case occurs. Solving two (leftHalf and rightHalf) subproblems (of size n/2) takes T(n/2) time for each problem, so it takes 2T(n/2) time to solve both. To find the max subarray from when it crosses (crossing), it takes Θ(n) time to go through all elements in the left half and then in the right half. It also takes a constant Θ(1) to perform the comparisons. So, in total to find the max subarray from when it crosses, it takes Θ(n) + Θ(1).

Therefore,

Recurrence for recursive case is: $T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1)$

$= 2T(n/1) + \Theta(n)$

If we combine both the base case and recursive case:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2)+ \Theta(n) & \text{if } n>1 \end{cases}$$

Using the master method (a=2, b=2, f(n) = n) :

T(n) = Θ(n lg n)

Algorithm 4: Linear-time

```
SubArray max_linear_subarray(int arr[], int count) {
        max_sum = -infinity
        ending_here_sum = -infinity

        ending_here_high

        ending_here_low

        for i in range(count)
                ending_here_high = i
                if ending_here_sum > 0:
                        ending_here_sum += arr[i]
                else:
                        ending_here_low = i
                        ending_here_sum = arr[i]
                if ending_here_sum > max_sum
                        max_sum = ending_here_sum
                        leftIndex = ending_here_low
                        rightIndex = ending_here_high
        subarray.maxSum = max_sum
        return subarray
}
```

Asymptotic Analysis for Algorithm 4: With one loop, the running time is **Θ(n)**. Even if the max subarray has already been computed, the algorithm continues until the end of the array.

II. Proof of Correctness

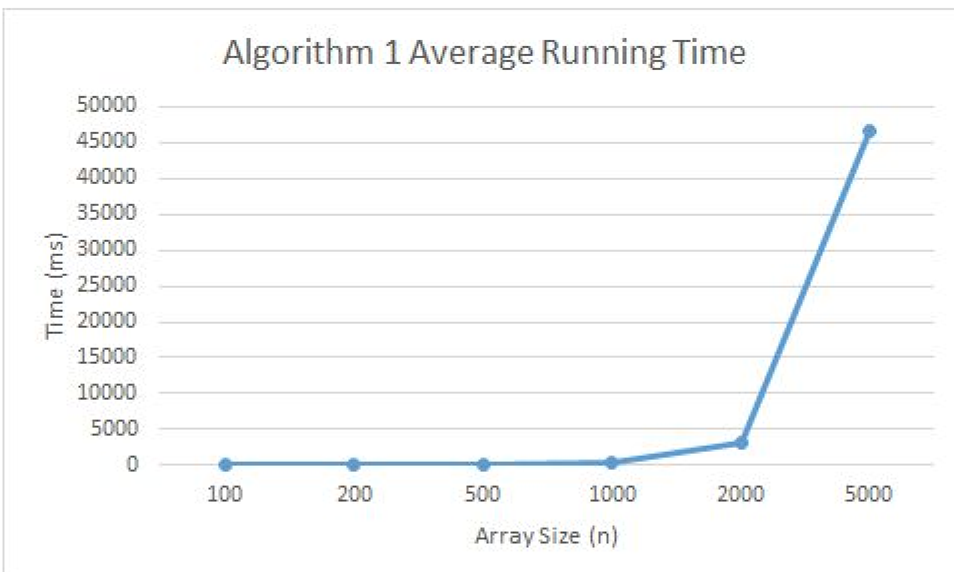
Assuming that find_max_crossing_subarray is correct, the correctness of maxSubArraySum is readily apparent. The comparison to find the maximum sum to the right determines the best-possible subsequence among the three possibilities (ie. left, right, or spanning subarrays).

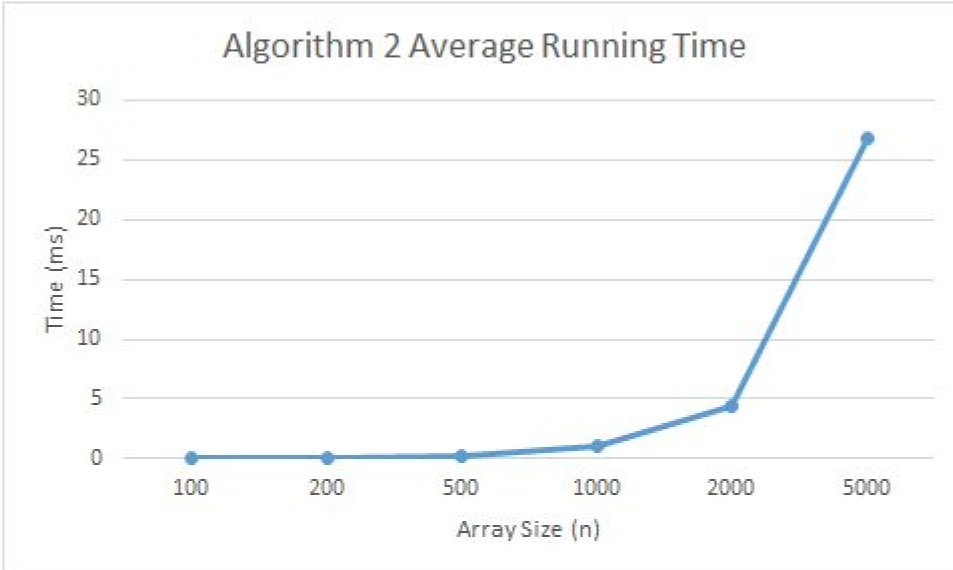
We can prove the correctness of find_max_crossing_subarray directly. It's separated into two parts. First, we determine the MSS (Max Sum Subarray) beginning at mid. Then we determine the MSS beginning at mid + 1. Either way, we evaluate the MSS by looping over the array elements and storing the current MSS estimate. We initialize the left index to i = mid, the MSS to left_sum = $-\infty$, and the “accumulator” to sum = 0. As we loop over the array elements, we increment the accumulator sum. If the running sum surpasses the estimated MSS left_sum, then we update our estimate of the endpoint i and left_sum. A similar process is applied to the right side to determine j and right_sum. In conclusion, the correct MSS for the “spanning” subarray is given by subarray ← left_sum + right_sum.

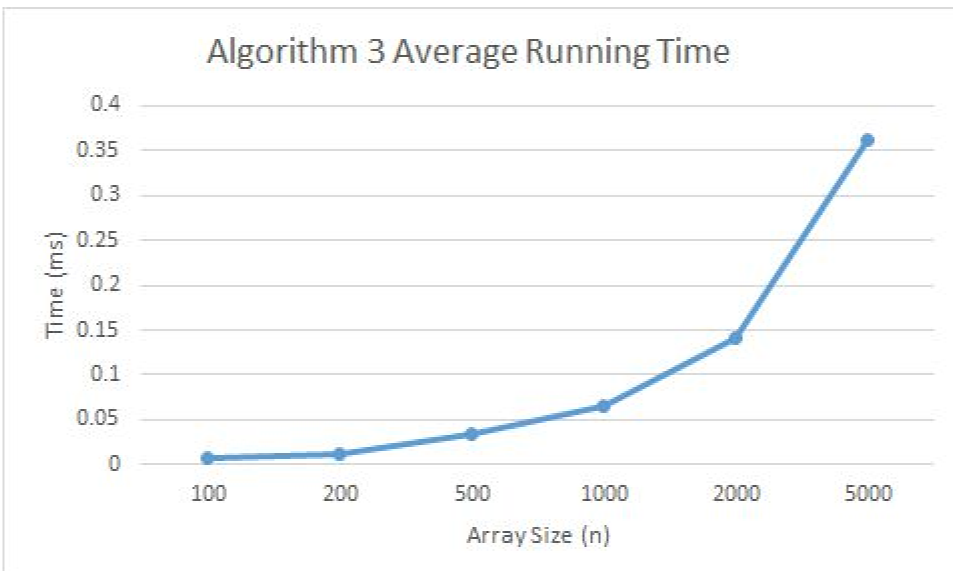
III. Testing

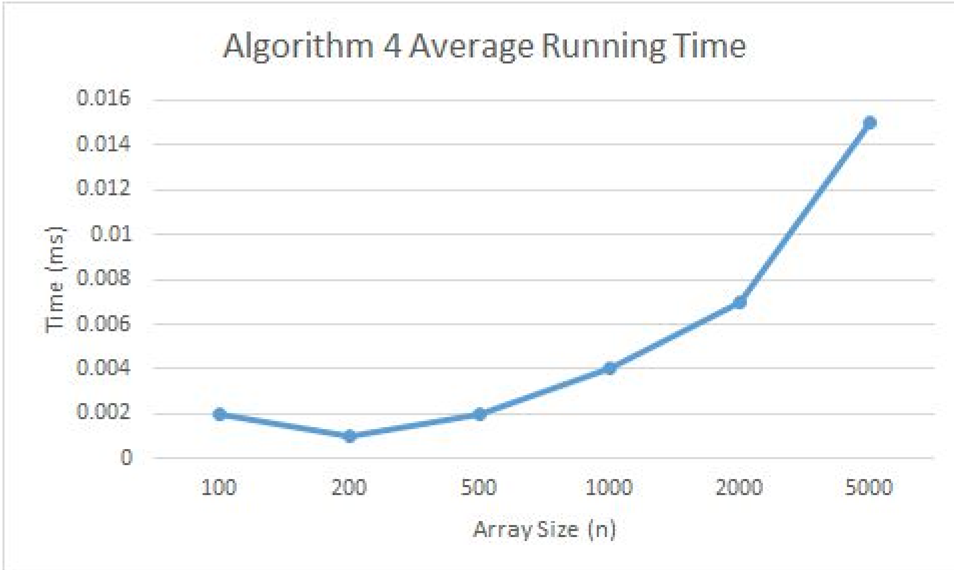
During the testing phase of our project, we ran into some discrepancies with a couple of the algorithms. We found that algorithm 3 worked the best and was correct in all cases. The other algorithms produced correct results about half the time. We compared algorithm 3 with the other algorithms, and went one by one fixing the errors. The biggest problems in the algorithms were small logic errors, and were easily fixed. In order to verify that everything worked properly, we used the provided MSS_Problems.txt files.

IV. Experimental Analysis




Algorithm 2 Average Running Time
30
25
20
15
10
5
0
Time (ms)
100
200
500
1000
2000
5000
Array Size (n)


Algorithm 3 Average Running Time
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Time (ms)
100
200
500
1000
2000
5000
Array Size (n)



Regression model:

Algorithm #1: $0.000000551 * x^{2.9517}$
Algorithm #2: $0.00000179 * x^{1.9320}$
Algorithm #3: $0.000008913 * x * \log(x)$
Algorithm #4: $0.0000028133x + 0.0010$

Discuss discrepancies between experimental and theoretical running times.

There are slight discrepancies between the theoretical running time and the experimental running times. These discrepancies are negligible and are expected during real world analysis. The largest discrepancy (Algorithm #2) is less than 4% of the theoretical running times, and was slower than theoretical. Algorithm #1 was also slower than theoretical by less than 2%.

Regression model: Largest input for the algorithm that can be solved in 10 minutes.

Algorithm #1: 164,594,625
Algorithm #2: 546,900,708
Algorithm #3: 3,081,090,000
Algorithm #4: 213,276,000,000


Log Log Plot
100000
10000
1000
100
10
1
0.1
0.01
0.001
Time (ms)
0
1000
2000
3000
4000
5000
6000
Array Size (n)
1
2
3
4