Project 1

CS325 — Spring 2015

---

by Group 2

Vedanth Narayanan

Jonathan Merrill

Tracie Lee

April 26, 2015

1 Theoretical Run-time Analysis

1.1 Algorithm 1

```
maxSubarray(a[1,...,n])
    max = a[0]
    for i = [0...n]
        for j = [i,n]
            sum = 0
            for each pair(i,j) with 1<=i<=j<=n
             compute a[i]+a[j+1]+...+a[j-1]+a[j]
        keep max sum found so far
    return max sum found
```

Asymptotic Analysis
We have $O(n^2)$ pairs * $O(n)$ time to compute each sum = $O(n^3)$.

1.2 Algorithm 2

```
maxSubarray(a[1,...,n])
    for i = 1, ...., n
        sum = 0
        for = i, ...., n
            sum = sum + a[j]
            keep max sum found so far
    return max sum found
```

Asymptotic Analysis
We have $O(n)$ i-iterations (outer loop) * $O(n)$ j-iterations (inner loop) * $O(n)$ for the time to update = $O(n^2)$.

1.3 Algorithm 3

```
maxSubarray(a[1,...,n], initial array length)
    length = len(a)

    if length > 1:
        left = left half of array
        right = right half of array
        first = maxSubarray(left, 0)
```

```
            last = maxSubarray(right, 0)
            reverse left
            center = helper(left) + helper(right)
        else:
            first = last = center = a[0]

        if initial array length == len(a):
            PrintResults(max([first, last, center]), a, [first, last, center])

        return max([first, last, center])

    helper(a):
        max = a[0]
        sum = 0
        for i in range(0, len(a)):
            sum += a[i]
            if sum > max:
                max = sum
        return max
```

Asymptotic Analysis
We have $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. This falls within Case 2 of the Master Method, and therefore yields a solution of $\Theta(nlgn)$.

1.4 Algorithm 4

```
maxSubarray(a[1,...,n])

maybeStart = 0
start = 0
end = 0
i = a[0]
sum = a[0]
small = minimum of (0, i)

for j in range(1,len(a)):
    i = i + a[j]
    if (i - small) > sum:
        start = maybeStart
```

```
            end = j+1
            sum = (i - small)
        if i < small:
            maybeStart = j+1
            small = i
    return (sum, a, a[start:end])
```

Asymptotic Analysis
We have $O(n)$ things to compute, therefore this takes $O(n)$ time.

2 Proof of Correctness: Algorithm 3

Base Case

We pass in an array consisting of 1 element, in which case the algorithm has a check in place for an array of length greater than one, and consequently returns the same array that had been passed in since it is the max subarray within that array.

Inductive Hypothesis

Assume that algorithm 3 correctly returns a maximum contiguous sum of elements s from an array A of $n > 1$ elements.

Inductive Step

If we split A into 2 separate arrays, let L represent the new array from the left side and let R represent the new array from the right side. Then let $s_{i...j}$ represent any sequence of numbers with the largest sum that lies with in S. If the sum of $s_{i...j} = x$, then $max(first, last, center) \leq x$. There are then 3 possibilities:

1. $s_{i...j}$ lies completely within L. In this case, it would follow that the max subarray of L is equal to x which means that $max(first, last, center) \geq first = x$. Because of this, we know that the answer returned is exactly x.

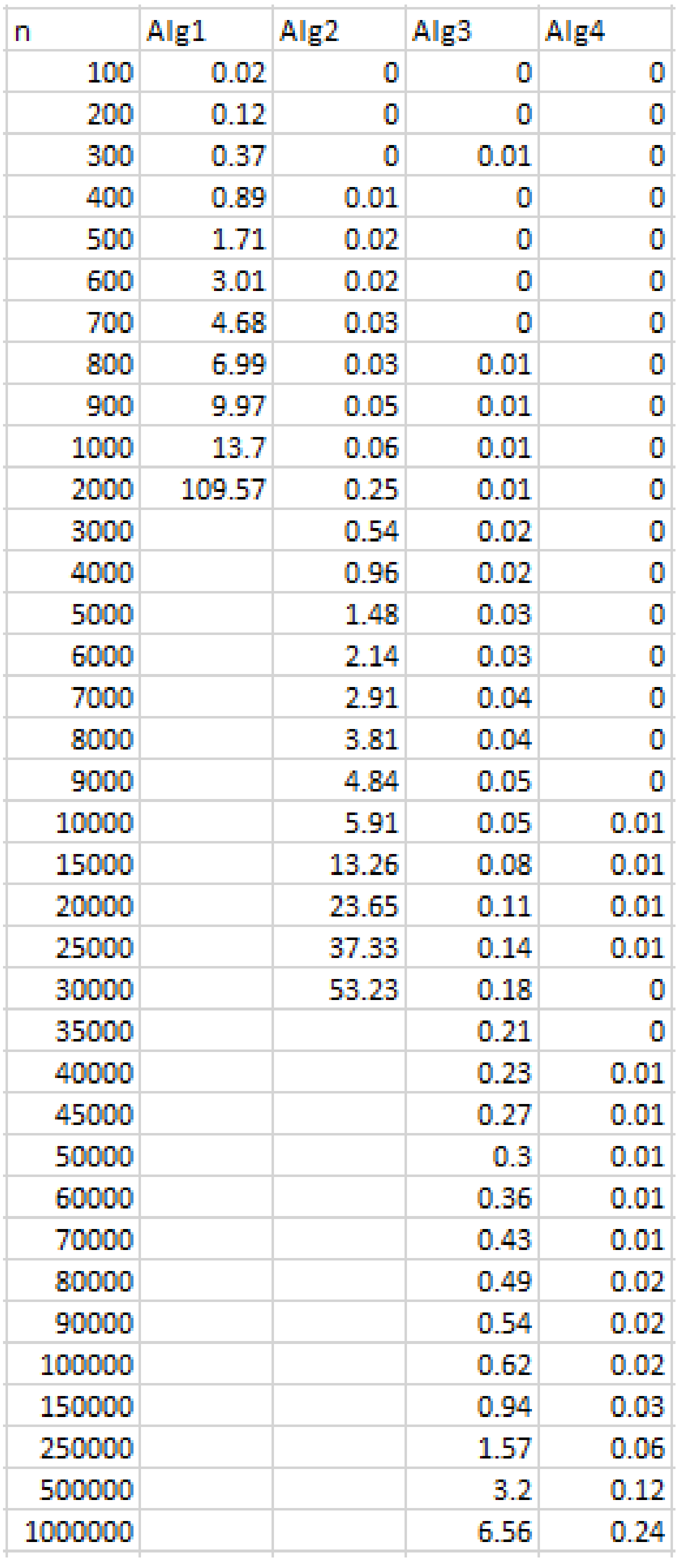
2. $s_{i...j}$ lies completely within R. In this case, it would follow that the max subarray of R is equal to x which means that $max(first, last, center) \geq last = x$. Because of this, we know that the answer returned is exactly x.

3. If $s_{i...j}$ does not lie completely within L or R, then it must start in L and end in R.

3 Testing

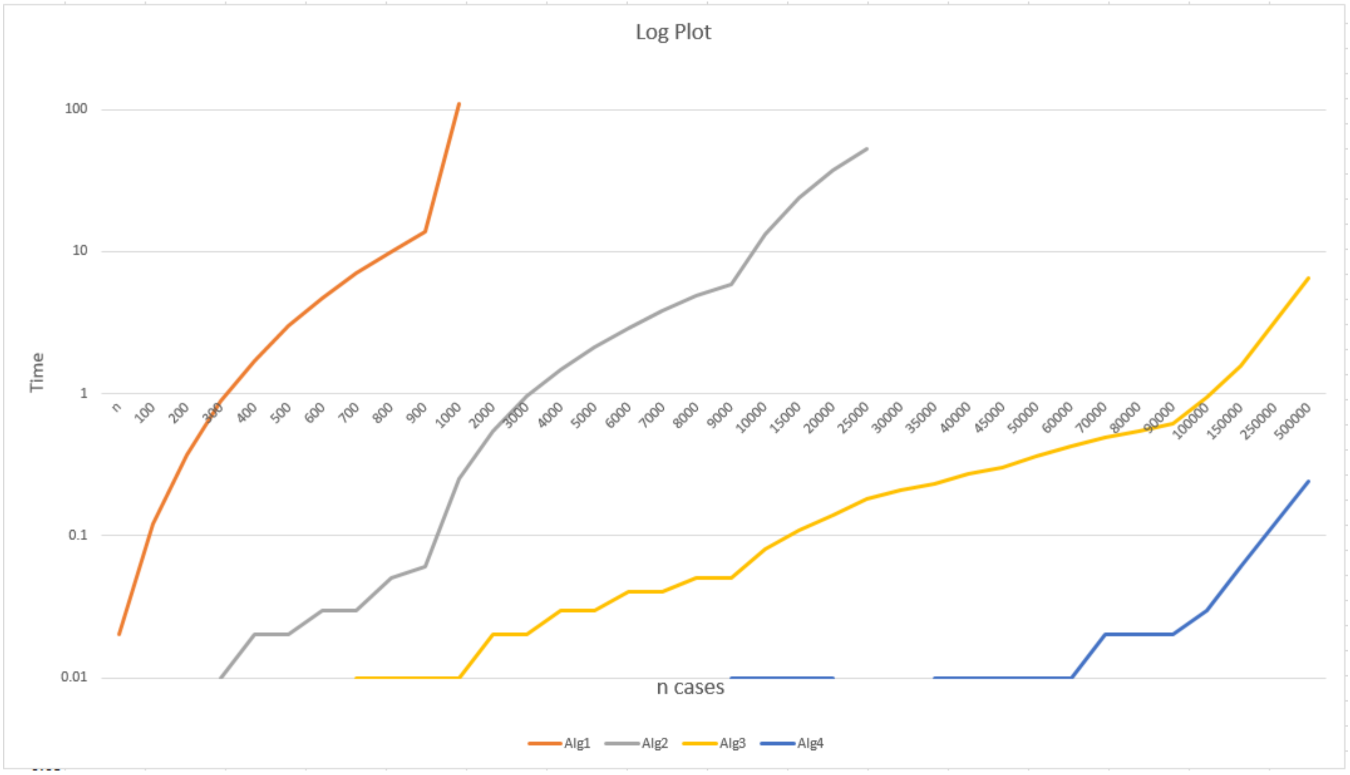
Our testing was quite thorough through out the completion of the project. The first measures taken for testing were for the correctness of our algorithms. We had written pseudo-code, and we needed to make sure that they were implemented properly. While not exactly writing unit tests, we performed manual tests. We gave the program a set list of integers, and knew what to expect as the result. The other side of our manual tests were running through the code by hand. In the beginning either we came across some indexing issues, or infinite loops, and similar issues. It was really helpful to go through by hand because it was our version of debugging, and more often than not the problems were identified and solved. In the end, we used the test cases mentioned on the project description, as well as on Canvas, and we were successfully able to pass the tests.

We needed to perform some sort of ”regression testing” next. This was to check the performance of our algorithms and the amount of time it took for them to run. We ended up writing a little function that generated random numbers, put them in a list, and wrote the list into the MSS_Problems.txt file. This function still exists in our script, if it need be checked. We always wanted to write the MSS_Problems.txt file, so we had our script accept integer values that defined the number of n's. Accordingly, we created a Makefile from where we were able to call our script with different arguments and append the results to MSS_Results.txt. The first arguments were ones that were mentioned in the project description, but we went beyond that. For the first 2 algorithms, we found out their limits, and at that point, we decided to comment out the function calls for them, and then continue the tests for the other 2 algorithms. The last n case for Algorithm 1 was 2000. The last n case for Algorithm 2 was 30000. Both Algorithm 3 and 4 ran till n was 1000000. All this testing was done on the Flip servers, for consistent results.



n	Alg1	Alg2	Alg3	Alg4
100	0.02	0	0	0
200	0.12	0	0	0
300	0.37	0	0.01	0
400	0.89	0.01	0	0
500	1.71	0.02	0	0
600	3.01	0.02	0	0
700	4.68	0.03	0	0
800	6.99	0.03	0.01	0
900	9.97	0.05	0.01	0
1000	13.7	0.06	0.01	0
2000	109.57	0.25	0.01	0
3000		0.54	0.02	0
4000		0.96	0.02	0
5000		1.48	0.03	0
6000		2.14	0.03	0
7000		2.91	0.04	0
8000		3.81	0.04	0
9000		4.84	0.05	0
10000		5.91	0.05	0.01
15000		13.26	0.08	0.01
20000		23.65	0.11	0.01
25000		37.33	0.14	0.01
30000		53.23	0.18	0
35000			0.21	0
40000			0.23	0.01
45000			0.27	0.01
50000			0.3	0.01
60000			0.36	0.01
70000			0.43	0.01
80000			0.49	0.02
90000			0.54	0.02
100000			0.62	0.02
150000			0.94	0.03
250000			1.57	0.06
500000			3.2	0.12
1000000			6.56	0.24

4 Experimental Analysis



5 Extrapolation and Interpretation

In general, our test data confirmed that our asymptotic analysis for each algorithm was very close. Algorithm was predicted to be $O(n^3)$, algorithm 2 $O(n^2)$, algorithm 3 $O(nlgn)$ and algorithm 4 $O(n)$, and all 4 algorithm plots have curves that are roughly equivalent.

To find the largest instance that can be solved within an hour for each algorithm, we had to do some rough estimating. Assuming that algorithm 1 is indeed $O(n^3)$, we used our largest testing trial for algorithm 1 ($n = 2000$; completing in 109 seconds) to find the approximate number of operations the computer can do each second:

$2000^3/109 = 73394495$ operations per second

Then, using the complexities we found for each algorithm, we could solve for n using a time restraint of one hour, and the operations per second found above:

- Algorithm 1: $n = 6400$
- Algorithm 2: $n = 514023$
- Algorithm 3: $n = 8.02 * 10^9$

- Algorithm 4: $n = 2.64 * 10^{11}$