Project One

Group 2

Evan Schallerer
Eli Rademacher
Krisna Irawan

Theoretical Run-Time Analysis

Algorithm 1: Enumeration

```
maxSum(array)
{
	subarrayLength = 1
	maxSum = 0, startIndex = -1, endIndex = -1
	while (subarrayLength <= length(array))
	{
		i = 0
		while (i <= (length(array) - subarrayLength))
		{
			sum = 0
			for (j to length(subarray))
			{
				sum  = sum + array[i + j]
			}
			if(sum > maxSum)
			{
				maxSum = sum
				startIndex = i
				endIndex = j + i
			}
			i = i + 1
		}
		subarrayLength = subarrayLength + 1
	}
	return (maxSum, array[start:end + 1])
}
```

Analysis of the Asymptotic running time: $O(n^3)$

In this algorithm, we can see that there are three loops nested inside of each other. The outer loop runs exactly n times, as does the next one. The very inner loop runs proportionally to n, so we have the runtime as:

$$n * n * n = n^3.$$

Algorithm 2: Better Enumeration

```
maxSubArray(array)
{
	end = length(array)
	maxSum = -1
	start = 0

	for (j from 0 to length(array))
	{
		i = j
		tempSum = 0

		while (i from j to length(array))
		{
			tempSum = tempSum + array[i]
			if (tempSum > maxSum)
			{
				maxSum = tempSum
				start = j
				end = i
			}
			i = i + 1
		}
	subArray = array[start:end + 1]

	return (maxSum, subArray)
}
```

Analysis of the Asymptotic running time: $O(n^2)$

We can see that this algorithm is $O(n^2)$ because it has a loop nested within another, both of which run exactly n times.

Algorithm 3: Divide and Conquer

```
subroutine(array, low, mid, high)
{
	maxLeft = mid
	leftSum = -∞
	sum = 0
	i = mid
	while (i >= low)
	{
		sum = sum + array[i]
		if (sum > leftSum)
		{
			leftSum = sum
			maxLeft  = i
		}
		i = i - 1
	}

	maxRight = mid
	rightSum = -∞
	sum = 0
	j = mid + 1
	while (j <= high)
	{
		sum = sum + array[j]
		if (sum > rightSum)
		{
			rightSum = sum
			maxRight = j
		}
		j = j + 1
	}
	return (leftSum + rightSum, maxLeft, maxRight)
}


divideAndConquer(array, low, high)
{
	if (high == low)
	{
		return (array[low], low, high)
	} else {
		mid = int((low+high)/2)
		(leftSum, leftLow, leftHigh) = divideAndConquer(array, low, mid)
		(rightSum, rightLow, rightHigh) = divideAndConquer(array, mid+1,
high)
		(crossSum, crossLow, crossHigh) = subroutine(array, low, mid, high)

		if (leftSum >= rightSum and leftSum >= crossSum)
		{
			return	(leftSum, leftLow, leftHigh)
		}
		elif (rightSum >= leftSum and rightSum >= crossSum)
```

```
            {
                  return (rightSum, rightLow, rightHigh)
            } else{
                  return (crossSum, crossLow, crossHigh)
            }
      }
}
```

Analysis of the Asymptotic running time: O(nlg(n))

Proof by the Master Theorem:

We can write this recursion as:

$$T(n) = 2T(\tfrac{n}{2}) + cn$$

The $2T(\ )$ part of this equation comes from the two recursive calls that are made, and the $T(\frac{n}{2})$ comes from the fact that each recursive call passes half of the array into itself each time.

The cn comes from the subroutine that iterates over the array passed into it twice. For generality, we said it iterates c times.

From the master method, we get that:

$$a = 2,\ b = 2,\ f(n) = cn$$

Comparing $f(n)$ to $O(n^{log_2(2)}) = O(n)$, we can see that:

$$cn = \theta(n)$$

Therefore we are in the second case of the master theorem, where

$$T(n) = \theta(n^{log_2(2)} * lg(n)) = \theta(nlg(\ n\ ))$$

Algorithm 4: Linear-Time

```
maxSubArray(array)
{
	n = length(array)
	maxSum = -∞
	endingHereSum = -∞
	i = 0
	while (i < n):
	{
		endingHereHigh = i
		if (endingHereSum > 0)
		{
			endingHereSum = endingHereSum + array[i]
		}
		else
		{
			endingHereLow = i
			endingHereSum = array[i]
		}
		if (endingHereSum > maxSum)
		{
			maxSum = endingHereSum
			low = endingHereLow
			high = endingHereHigh
		}
		i = i + 1
	}
	return (maxSum, array[low:high + 1])
}
```

Analysis of the Asymptotic running time: O(n)

We can see that this algorithm runs in linear time because there is only one loop in the program which runs exactly n times.

Proof of Correctness

Max Subarray Problem

Divide and Conquer Algorithm

Pre-Condition:

- Arrays with at least one positive number

Post-Conditions:

- The sum of the largest subarray in the original array
- The maximum subarray contained in the original array

Proof by Induction:

Base Case:

Assume $n = 1.$ This means that the array will be of size one, therefore the maximum subarray of the original array, *A*, is made up of the only element of *A* and therefore is equal to *A*. The value of the sum of this subarray is also equal to the value of *A[0]*, so this will be returned as the sum of the largest subarray. The single element array *A* is returned in this case when $high == low$, where *high* is the last index in the array, and *low* is the first index of the array.

Inductive Step:

Assume that the algorithm returns the largest subarray and the sum of this subarray for any array, *A*, of size *n*.

Now assume that the algorithm was passed an array of size *n+1* , where *n* is an integer greater than 0.

Because this algorithm is a Divide and Conquer algorithm, the recursive portion of the algorithm splits the input into two arrays of $\frac{n+1}{2}$, so $n_1 = \frac{n+1}{2}$ and $n_2 = \frac{n+1}{2}$. Because we know that the algorithm returns the largest subarray for any array *A* of size *n*, and *n* > 0, then we know that $\frac{n+1}{2} \leq n$, and therefore is true for any integer value of *n* > 0.

This proves the recursive portion of this Divide and Conquer algorithm, and now we must examine the iterative part that determines whether the maximum subarray is contained in the middle of the portion the halves that got split in the recursive part of this algorithm.

Iterative proof of the left side sum of the array

Induction hypothesis:

- *maxLeft* is the maximum left index of the largest subarray
- *leftSum* is maximum sum of the left side of the subarray
- *sum* is the sum of the left side of the subarray

1) Initialization: Prior to the first loop we have 3 different variables:
    - *maxLeft* = the middle index of the array
    - *leftSum* = negative infinity
    - *sum* = 0

2) Maintenance : To show that the hypothesis is true for step k, then it will be true for step k+1

At the start of step *k*: Assume that sum is the sum of the subarray from the middle of the array to the middle - *k* elements of the array. At step *k*+1 the *sum* will equal to the sum of the subarray from the middle of the array to the middle - (*k*+1) elements of the array. After the sum is calculated, if *sum* is greater than *leftSum* then we assign the sum to *leftSum* and *k* to *maxLeft*.

3) Termination: The loop terminates when *k* is equal to *low* which is the beginning index of the array. At this point, *leftSum* is equal to the maximum sum of the left side of the subarray, and the *maxLeft* is equal to the first index of the largest subarray.

Iterative proof of the right side sum of the array

Induction hypothesis:

- *maxRight* is the maximum right index of the subarray
- *rightSum* is maximum sum of the right side of the subarray
- *sum* is the sum of the right side of the subarray

2) Initialization: Prior to the first loop we have 3 different variables:
    - *maxRight* = the middle index of the array
    - *rightSum* = negative infinity
    - *sum* = 0

2) Maintenance : To show that the hypothesis is true for step k, then it will be true for step k+1

At the start of step *k*: Assume that sum is the sum of the subarray from the (middle + 1) of the array to the (middle + 1) + *k* elements of the array. At step *k*+1 the sum will equal to the sum of the subarray from the (middle + 1) of the array to the (middle + 1) + (*k*+1) elements of the array. After the sum is calculated, if the sum is greater than *rightSum* then we assigning the sum to *rightSum* and *k* to *maxRight*.

3) Termination: The loop terminates when k is equal to the high which is the end index of the array. At this point *rightSum* is equal to the maximum sum of the right side of the subarray and the *maxRight* is equal to the last index of the largest subarray.

Testing

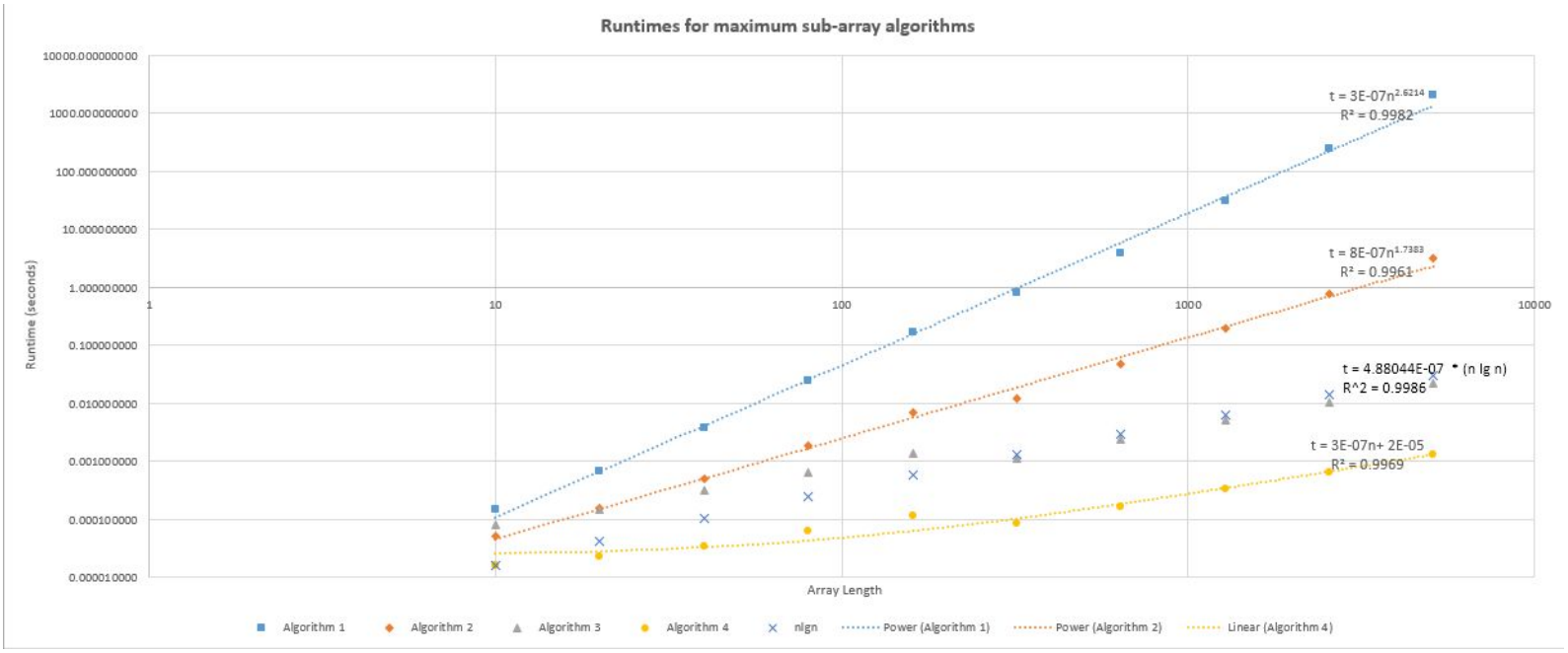
To test our algorithms, we used test-driven development and built a testing suite before each algorithm was completed. The tests execute each algorithm on a set of arrays, and each algorithm is compared to the last that ran to ensure that they all terminate with the same results. This alone, however, does not guarantee their correctness, as if they all had the same bug, the test suite would not catch it. To account for this, we manually checked the test suite's output against the provided web-application that we are assuming is correct.

Experimental Analysis

Average running time for each n :

Array Length	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
10	0.000146151	0.000051975	0.000081062	0.000015974
20	0.000677109	0.000154972	0.000152111	0.000023127
40	0.003838062	0.00049901	0.000312805	0.000035048
80	0.024200916	0.001832008	0.000645161	0.000062943
160	0.169660091	0.007081032	0.001364946	0.00011611
320	0.807360888	0.011959076	0.001124859	0.000085115
640	3.895560026	0.047562122	0.002374887	0.000164986
1280	31.423594	0.192635059	0.005053997	0.000329971
2560	253.224937	0.773458004	0.010546923	0.000654936
5120	2055.36006	3.156152964	0.022043943	0.001335144

Log-log plot of the running time of the four algorithms



Algorithm 1: $y = 3 \times 10^{-7} n^{2.6214}$, which is close to cubic

Algorithm 2: $y = 8 \times 10^{-7} n^{1.7383}$, which is close to quadratic

Algorithm 3: $y = 4.88044 \times 10^{-7}(nlg(n))$, which is $nlg(n)$

Algorithm 4: $y = 3 \times 10^{-7} n + (2 \times 10^{-5})$, which is linear

We can see that the equations obtained from the line of best fit for each algorithm all vary from the theoretical running times. This can be attributed to many factors. The largest factor is probably that the machine that the program was running on has many users and was used over a wide time range due to the long running time of some of the algorithms. This could cause it to execute certain parts faster or slower, which would change the times. There is also the issue that theoretical runtimes generalize many of the smaller details of the algorithm, which causes it to stray more from the actual times as well.

Theoretical maximum array size that can be solved in ten minutes			
Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
3,533	127,526	48,170,800	1,999,999,933

*Note: This is assuming a calculation time of one second.