Group 11: Jacob Karcz, David Haskell, Keisha Arnold
CS 325-400: Project 1 Report
4/23/2017

Theoretical Run-time Analysis

Algorithm 1: Enumeration

```
cubicMSS(A[0...n-1], n) {
	maxSum = 0;
	sum = 0;
	for (i to n) {    // starting index
		for(j = i to n) {  // ending index
			sum = 0;
			for (k = i to k <=j) {    // compute sum from i, j
				sum += A[k]
				If (sum > maxSum) {
					maxSum = sum;
					startIndex = i;		// keep this startIndex
					endIndex = j;		// keep this endIndex
				}
			}
		}
	}
	subArrSize = (endIndex - startIndex) + 1;    // calculate sub array size
	maxSubArr[subArrSize];			// make new array of appropriate size
	for (i to subArrSize) {			// fill new sub array
		maxSubArr[i] = A[startIndex];
		increment startIndex;
	}
	return maxSum, maxSubArr;		// return max subarray sum and subarray
}
```

This algorithm uses brute-force, trying every possible i, j pair and computing the sum of entries in A[i...j]. There are $O(n^2)$ pairs and it takes $O(n)$ time to compute each sum of pairs.

$O(n^2) * O(n) = O(n^3)$ run time

Algorithm 2: Better Enumeration

```
quadMSS(A[0...n-1], n) {
	maxSum = 0;
	sum = 0;
	for (i to n) {			//O(n) iterations for outer loop (n possible start positions)
		sum = 0
		for (for j = i to n) {	//O(n) iterations for inner loop (n possible end positions)
			sum += A[j];	//O(1) constant time to update sum
			if (sum > maxSum) {
				maxSum = sum;
				startIndex = i;		// keep this startIndex
				endIndex = j;		// keep this endIndex
			}
		}
	}
	subArrSize = (endIndex - startIndex) + 1;	// calculate sub array size
	maxSubArr[subArrSize];			// make new array of appropriate size
	for (i to subArrSize) {			// fill new sub array
		maxSubArr[i] = A[startIndex];
		increment startIndex;
	}
	return maxSum, maxSubArr;		// return max subarray sum and subarray
}
```

This algorithm also uses brute-force using iteration to go through every possible subarray until the max sum is found. Similar to algorithm 1, we have an outer for loop which loops through n possible start positions taking O(n) time and an inner for loop which loops through n possible end positions, also taking O(n) time. However rather than computing the sum from scratch every time, we keep the previous sum from A[i] to A[j-1] and just add the extra A[j] to that previous sum, eliminating the need for the third for loop as we saw in algorithm 1 and making the time to compute the sum a constant O(1) rather than O(n).

(O(n) outer for loop) * (O(n) inner for loop) * (O(1) updating sum) = O(n^2) run time

Algorithm 3: Divide and Conquer // adapted from textbook: Introduction to Algorithms, 3rd ed.

```
findMaxSubArrCrossingMid(A[1….n-1], left, mid, right) {
        leftSum = - infinity;
        sum = 0;
        // find max subarray of the left half
        for (i = mid to left) {    // mid down to left (decreasing indices)
                sum = sum + A[i];
                if (sum > leftSum) {
                        leftSum = sum;
                        maxLeft = i;       // starting index of max subarray
                }
        rightSum = - infinity;
        sum = 0;
        // find max subarray of the right half
        for (j = mid + 1 to right) {    // mid up to high (increasing indicies)
                sum = sum + A[j];
                if (sum > rightSum) {
                        rightSum = sum;
                        maxRight = j;    // ending index of max subarray
                }
        }
        return(maxLeft, maxRight, leftSum + rightSum);
}

findMaxSubArr(A[0...n-1], left, right) {
        if (right == left) {    //base case (only one element)
                return (left, right, A[left]);
        }
        else mid = (left + right) / 2;    // calculate middle
        // return max subarray (recursive calls)  in one of the following:
                // a) left half
                (minLeft, maxLeft, leftSum) = findMaxSubArr(A, left, mid);
                // b) right half
                (minRight, maxRight, rightSum) = findMaxSubArr(A, mid+1, right);
                // c) subarray that crosses the midpoint
                (crossLeft, crossRight, crossSum) = findMaxSubArrCrossingMid(A, left, mid, right);
                // leftSum is the mss
                if (leftSum >= rightSum and leftSum >= crossSum)
                        return (minLeft, maxLeft, leftSum);
                // rightSum is the mss
                else if (rightSum >= leftSum and  rightSum >= crossSum)
                        return (minRight, maxRight, rightSum);
                // crossSum is the mss
                else
                        return (crossLeft, crossRight, crossSum);
        }
```

The above algorithm also uses the divide and conquer concept using the observation that (1) the max subarray can only be contained in the first half, (2) only contained in the second half, or (3) composed of a suffix of the first half and a prefix of the second half. (1) and (2) are solved by recursion. Therefore, we are repeatedly dividing the length of our array in half every time we go down a level in our tree until we finally reach the base case (an array containing a single element), giving us a depth of O(lg n). The non-recursive work for (3) is done in linear O(n) time because we are simply finding the max suffix of the first half and the max prefix of the second half and combining them together.

(O(n) non-recursive work) * (O(lg n) tree depth) = O(n lg n) run time

<u>Algorithm 4: Linear-time</u>

```
linearMSS(A[0...n-1], n) {
        maxSoFar = 0;
        maxEndingHere = 0;
        startIndex, endIndex, s = 0

        for(i to n) {
                maxEndingHere += A[i];
                if(maxSoFar < maxEndingHere) {
                        maxSoFar = maxEndingHere;
                        startIndex = s;
                        endIndex = i;
                }
                if(maxEndingHere < 0) {
                        maxEndingHere = 0;
                        s = i + 1;
                }
        }
        subArrSize = (endIndex - startIndex) + 1;    // calculate sub array size
        maxSubArr[subArrSize];                       // make new array of appropriate size
        for (i to subArrSize) {                      // fill new sub array
                maxSubArr[i] = A[startIndex];
                increment startIndex;
        }
        return maxSoFar, maxSubArr;                  // return max subarray sum and subarray
}
```

The above algorithm uses dynamic programming by using a bottom-up approach to find our max sum subarray. This means we start from our smallest problem (1 element) and work our way up to the biggest problem (the entire array). The max subarray we are looking for either uses the last element in our subset or it doesn't. The work of finding our max subarray and max suffix takes constant O(1) time. There are O(n) elements to compute in our array, therefore our running time is O(n).

(O(n) elements to compute) * (O(1) compute max array and max suffix) = O(n) run time

Testing
Given that we are to assume all elements in the input arrays will be integers with at least one positive element in each array, we used the following guidelines for test sets against our program:

Test Input	**Example**
Array of size 1, positive	[5]
Array of all negative values with one positive value	[-1 -10 -9 -2 -8 2]
Array of all positive values	[7 3 6 12 100 0 2]
Array of even size, mixed pos/neg values	[-100 0 -12 20 -50 30 5 -7 9 100]
Array of odd size, mixed pos/neg values	[3 8 -25 150 -10 0 1 2 -40]
Array containing all pos values in the first half and all negative values in the second half	[1 1 1 1 1 -1 -1 -1 -1 -1]
Array containing all neg values in the first half and all pos values in the second half	[-100 -100 -100 -100 100 100 100 100]

We also tested our algorithms against the MSS_Problems.txt and MSS_TestProblems.txt files. While MSS_Problems.txt produced the expected results, MSS_TestProblems.txt contained a bug where it would sometimes duplicate the last number of our subset array. After some investigation, we determined that a few lines in MSS_TestProblems.txt contained a trailing space at the end, causing the duplicate number to show up in our results. The MSS_Problems.txt file did not contain these trailing spaces. After removing the trailing spaces from the MSS_TestProblems.txt file our code produced the correct results. Since we are using MSS_Problems as the file we are submitting with our code, we used it's format as a "standard" and therefore did not leave trailing spaces at the end of the lines of input when testing our code.

Conclusion

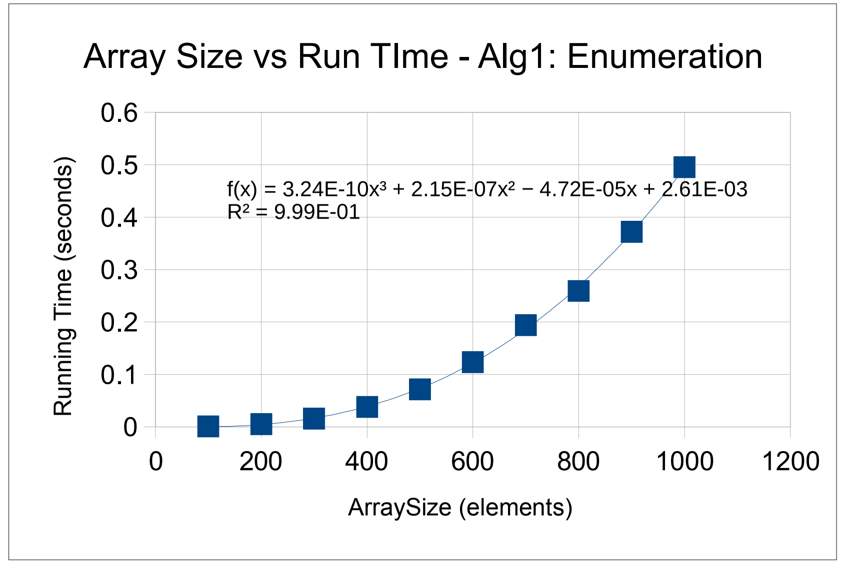
For the problem where we are given an array of small integers and are to compute the max sum and corresponding subarray, we designed, implemented, and analyzed four algorithms to solve this problem. From this analysis, we have come to the conclusion that algorithm 4, using dynamic programming and a bottom-up approach was the most efficient. This means that algorithm design does indeed matter- especially when dealing with large data sets. To illustrate the difference we just have to look at our experimental analysis to see that algorithm 1 can only handle an input of 5,490 in 60 seconds compared to algorithm 4 which handles 1.41E+10 in the same amount of time! In addition, since algorithm 1 follows a cubic curve, it's efficiency will taper as our data set grows larger, but algorithm 4 follows a linear curve, so it's efficiency will remain linear to the size of the input.

	Theoretical RT	**Curve fit to data**	**R^2**	**Largest input in 60 secs**
Algorithm 1	$O(n^3)$	Cubic	.999	5,490
Algorithm 2	$O(n^2)$	Quadratic	.995	229,000
Algorithm 3	O(nlgn)	nlgn/linear	.980	1.35E+09
Algorithm 4	O(n)	linear	.995	1.41E+10

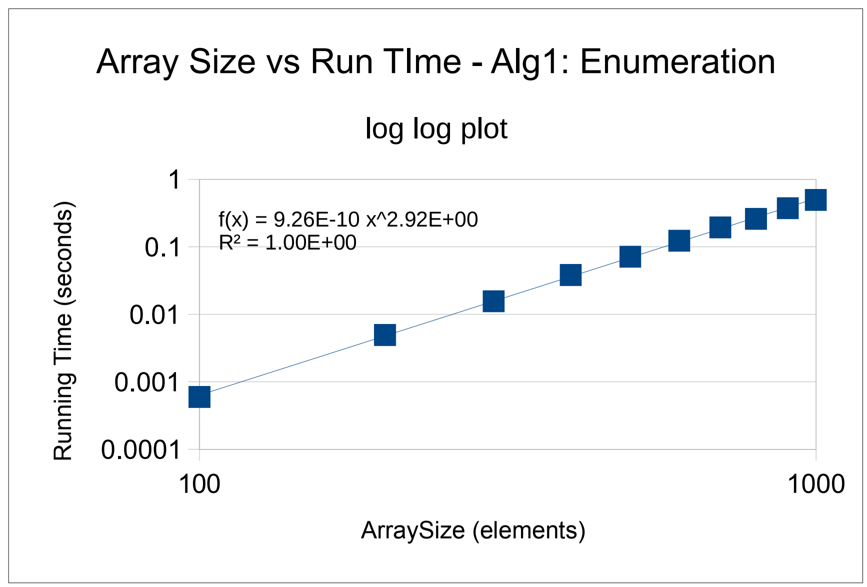
Alg1 : Enumeration	
n	Avg run time (s)
100	0.0005969
200	0.0049087
300	0.0155972
400	0.0381729
500	0.0715286
600	0.1233996
700	0.1936407
800	0.2591869
900	0.3719375
1000	0.4951969

To predict the largest input (array size) that could be solved in a given time, we set the regression equation equal to the given times:

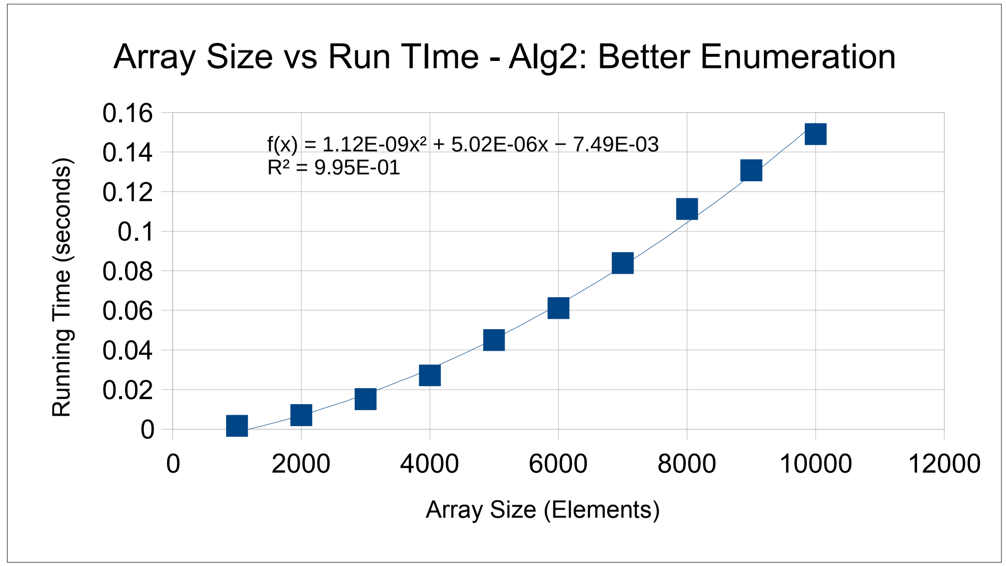
Time (s)	Array Size (n)
5	2290
10	2930
60	5490



The regression curves show a polynomial (cubic) relationship, which is predicted theoretically.



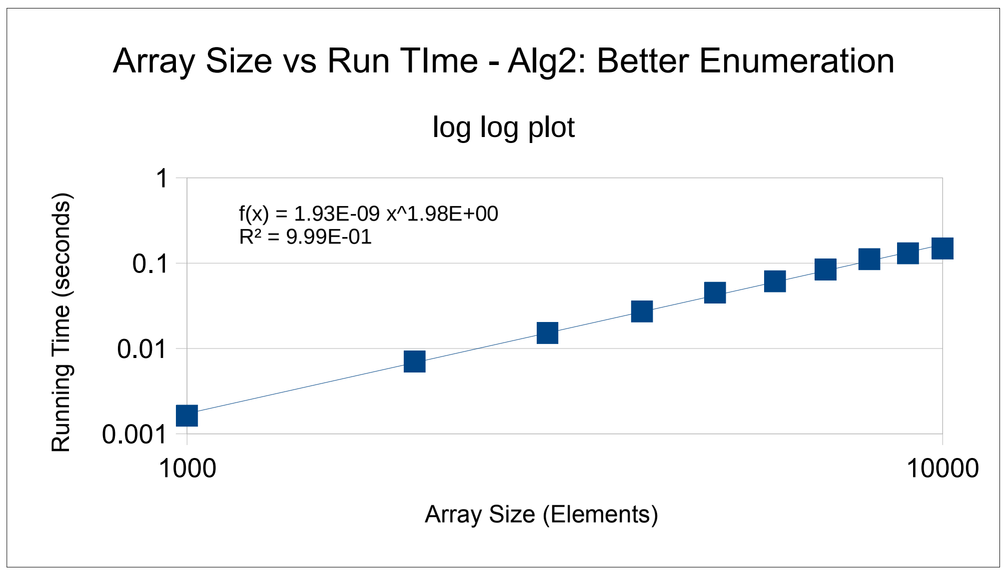
Alg2: Better Enumeration	
n	Avg run time (s)
1000	0.001633
2000	0.0070292
3000	0.0151424
4000	0.0271719
5000	0.0450681
6000	0.0610665
7000	0.0839139
8000	0.1112906
9000	0.1307949
10000	0.1490488



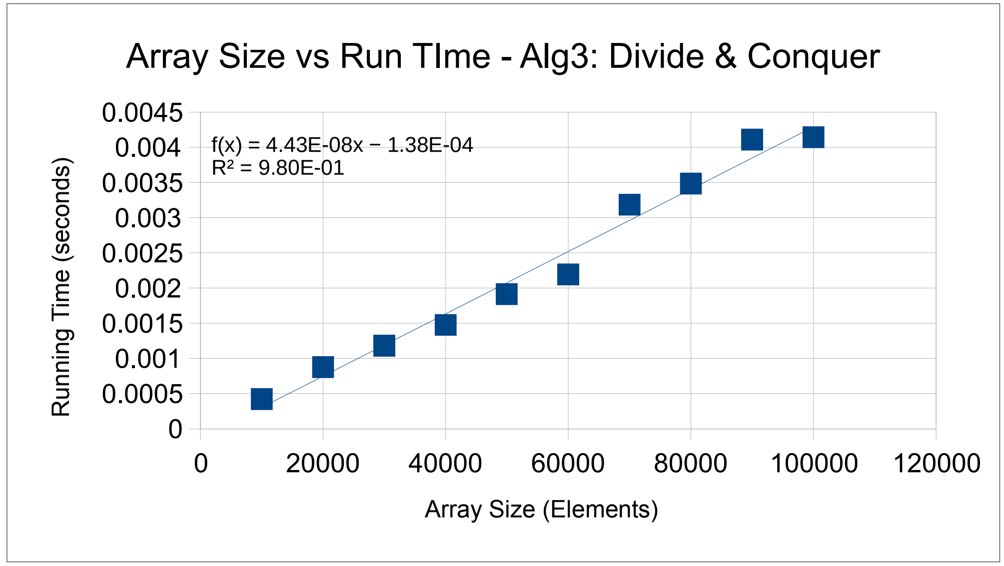
The regression curves show a polynomial (quadratic) relationship, which is predicted theoretically.

To predict the largest input (array size) that could be solved in a given time, we set the regression equation equal to the given times:

Time (s)	Array Size (n)
5	65000
10	92000
60	229000



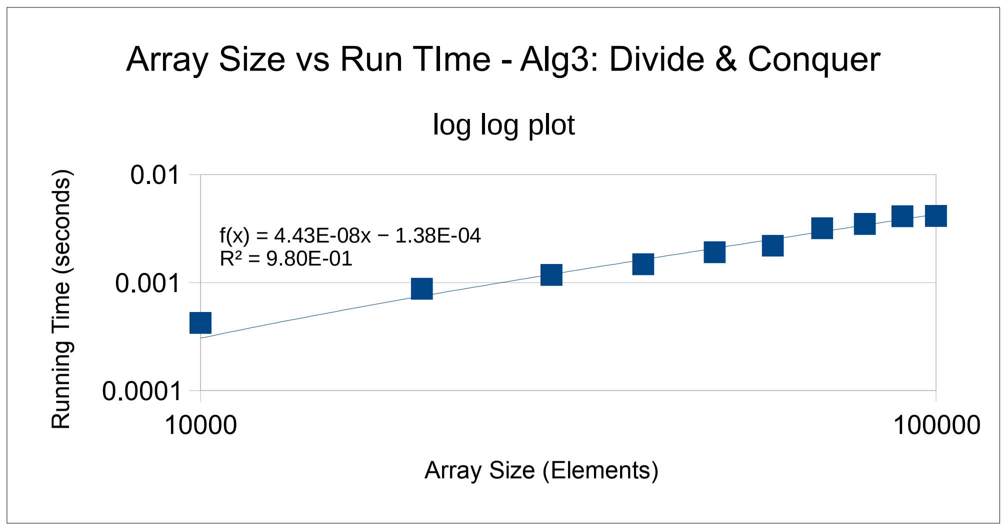
Alg3: Divide and Conquer	
n	Avg run time (s)
10000	0.0004214
20000	0.0008745
30000	0.0011774
40000	0.0014758
50000	0.0019137
60000	0.0021938
70000	0.0031834
80000	0.003483
90000	0.0041069
100000	0.0041421



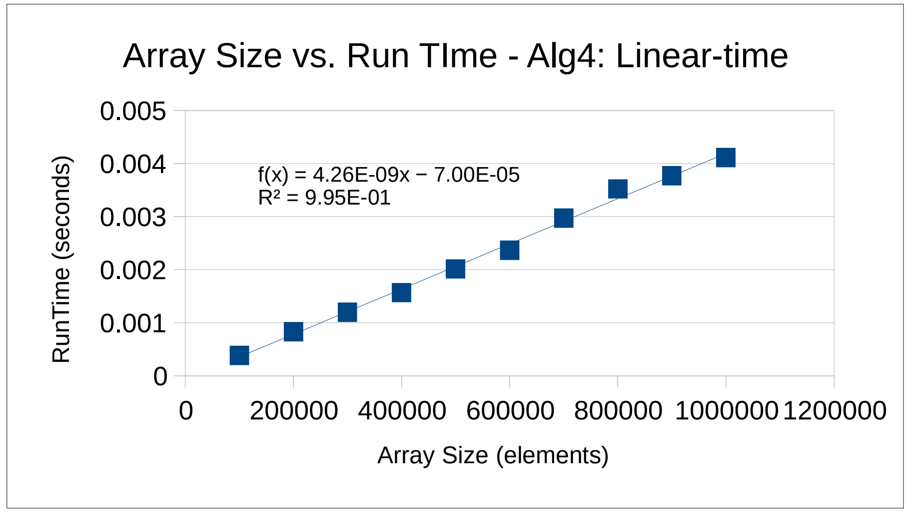
The regression curves show a close match to a linear relationship. Theoretical predictions indicate a nlgn relationship. However, the variation due to lgn is not significant compared to n, so this graph appears linear.

To predict the largest input (array size) that could be solved in a given time, we set the regression equation equal to the given times:

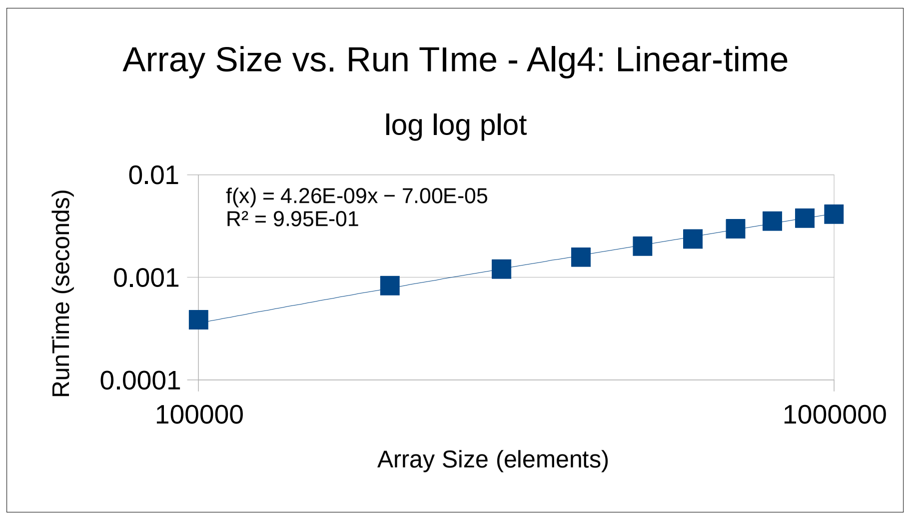
Time (s)	Array Size (n)
5	1.13E+08
10	2.26E+08
60	1.35E+09



Alg4: Linear-time	
n	Avg run time (s)
100000	0.0003848
200000	0.0008302
300000	0.0012008
400000	0.0015683
500000	0.0020156
600000	0.0023643
700000	0.0029723
800000	0.0035241
900000	0.0037737
1000000	0.0041141



The regression curves show a linear relationship, which is predicted theoretically.



To predict the largest input (array size) that could be solved in a given time, we set the regression equation equal to the given times:

Time (s)	Array Size (n)
5	1.17E+09
10	2.35E+09
60	1.41E+10


Array Size vs. Run Time
log log plot
Alg1 : Enumeration
Alg2: Better Enumeration
Alg3: Divide and Conquer
Alg4: Linear-time
RunTime (seconds)
1
0.1
0.01
0.001
0.0001
100
1000
10000
100000
1000000
Array Size (elements)