CS325 Project 1 – Maximum Sub-Array

Project Team: Eric Rouse

Resources:

1) Wikipedia Maximum Subarray problem:
http://en.wikipedia.org/wiki/Maximum_subarray_problem

2) Wikipedia Divide and Conquer Algorithm:
http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm

3) Programming Pearls by Jon Bently:
http://www.akira.ruc.dk/~keld/teaching/algoritmedesign_f03/Artikler/05/Bentley84.pdf

4) Merge Sort - divide and conquer: http://www.shaunlippy.org/blog/?p=77

5) Notes on maximum subarray problem:
http://cs.slu.edu/~goldwasser/courses/slu/csci314/2012_Fall/lectures/maxsubarray/

6) divide and conquer proof by induction: www.cs.oberlin.edu%2F~asharp%2Fcs280%2F2012sp%2Fhandouts%2Fdc.pdf

Mathematical Analysis

Algorithm 1 – O(n^2)

Psuedocode:

```
for each item in the array starting at element 0:
    current_sum = value of item
    current_maxum = max(maximum_sum, current_sum)
    for each item in the array starting at the current element position plus 1:
        current_sum += value of item
        current_maxum = max(maximum_sum, current_sum)
return maximum_sum
```

Analysis of Running Time:

In algorithm 1, each array element is summed with every subsequent array element. Each array element after the first is traversed the same number of times as there are elements remaining. If we denote array elements as n then the run time as a function of elements is n*(n-1). That evaluates to n^2-n, and simplifying to big-O notation the n^2 term dominites so, O(n^2).

Algorithm 2 – O(n)

Psuedocode:

```
for each item in the array starting at element 0:
    current_sum = max(item, summation + item)
    current_maxum = max(maximum_sum, summation)
return maximum_sum
```

Analysis of Running Time:

In algorithm 2 each array element is summed exactly one time. We keep a rolling sum that is reset if the current array element is larger than that element plus the sum, thus we don't traverse every possible linear combination. Again denoting the number of elements as n, we visit each element only once. So, in big-O: O(n).

Algorithm 3 – O(n log n)

Psuedocode:

```
#BASE CASE:
else if ending_element == beginning_element + 1:
   return array[beginning_element], beginning_element, ending_element)
#RECURRSIVE CASE:
# two or more elements
else:
   middle_element = floor((beginning_element + ending_element) / 2)

   # work from middle down to beginning, find max.
   for each element in the array from middle down to begging:
      current_sum += element
      if current_sum > left_max:
         left_max = current_sum

   # work from middle up to ending_element, find max.
   for each element in the array from begging up to end:
      current_sum += element
      if current_sum > right_max:
         right_max = current_sum
   #build results for max testing
   joiningmax_results = (left_max+right_max)

   #the max will either be the left arrayray, the right arrayray or the middle array
   return max(recurse(array, beginning_element, middle_element),
           recurse(array, middle_element, ending_element),
           joiningmax_results)
```

Analysis of Running Time:

Algorithm 3 is more difficult to characterize. At it's core we have a pretty simple idea: divide an array in half and check the left and right half maxes against each other.This would simplify to O(log n) because every operation divides the operation in two.

But there is a third case that we must check. The maximum might occur in some arbitrary overlap between the right and left arrays. So for each division we also perform a linear calculation on each half of the array. On the left half we start at the end and work our way down. On the right half we start at the beginning and work our way to the end. We sum the whole array iteratively and keep only the largest continuous sum.

Thus the run time is (n/2 + n/2) for each iteration and log n iterations, or O(n log n).

Theoretical Analysis

Given an array A of n elements, let recurrsive_algorithm return the sum of the maximum sub-array.

Claim

Recurrsive_algorithm(A) correctly returns the maximum sum of the sub-elements of A.

Proof

For an array A, let P(A) be the statement that recurrsive_algorithm(A) correctly returns the maximum sum of all sub-arrays of A.

As a base case, consider when |A| = 1. This one-element array is already the maximum possible sub array and the algorithm correctly returns the value of A as the maximum sum.

For the induction hypothesis, suppose that P(A) is true for all array of length < n; that is, suppose that for any array A of length < n, recurrsive_algorithm(A) correctly sums A. Now consider an array A of length n. Our algorithm divides A into two halves LEFT and RIGHT of size ~ n/2 which is less than n; therefore, LEFT and RIGHT are summed properly by the induction hypothesis.

The maximum sum is either entirely contained in LEFT or RIGHT or in the overlap between LEFT and RIGHT. So the maximum of LEFT, RIGHT or overlap is returned.

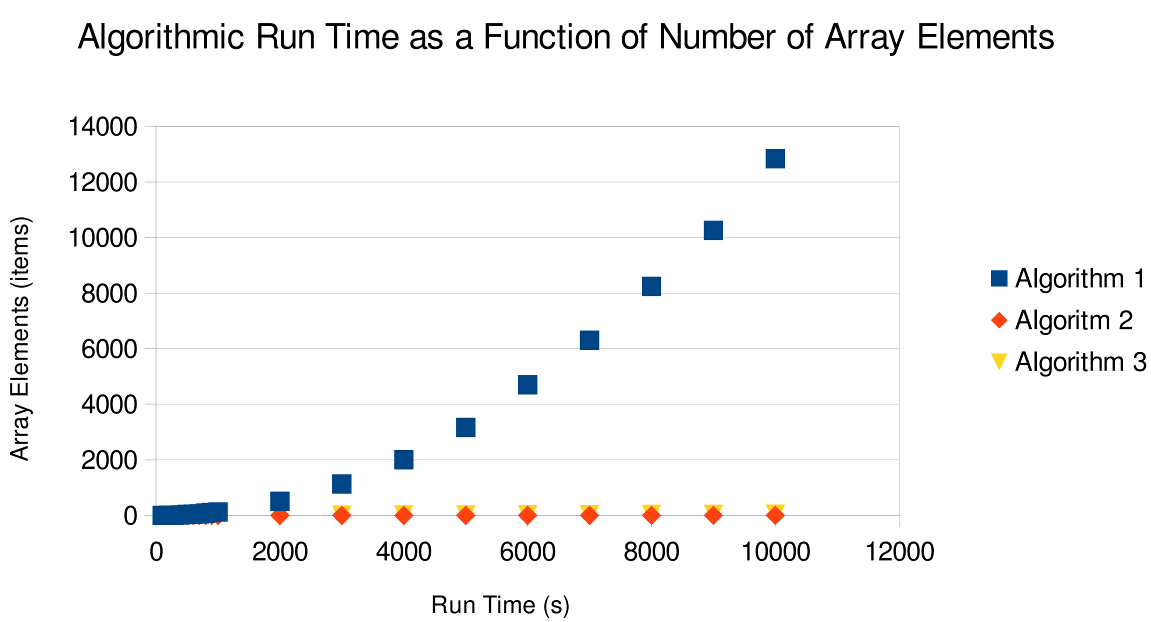
This is repeated until the base case is reached. Therefore, by induction, recurrsive_algorithm(A) correctly sums any sub-array in any array.

Experimental Analysis

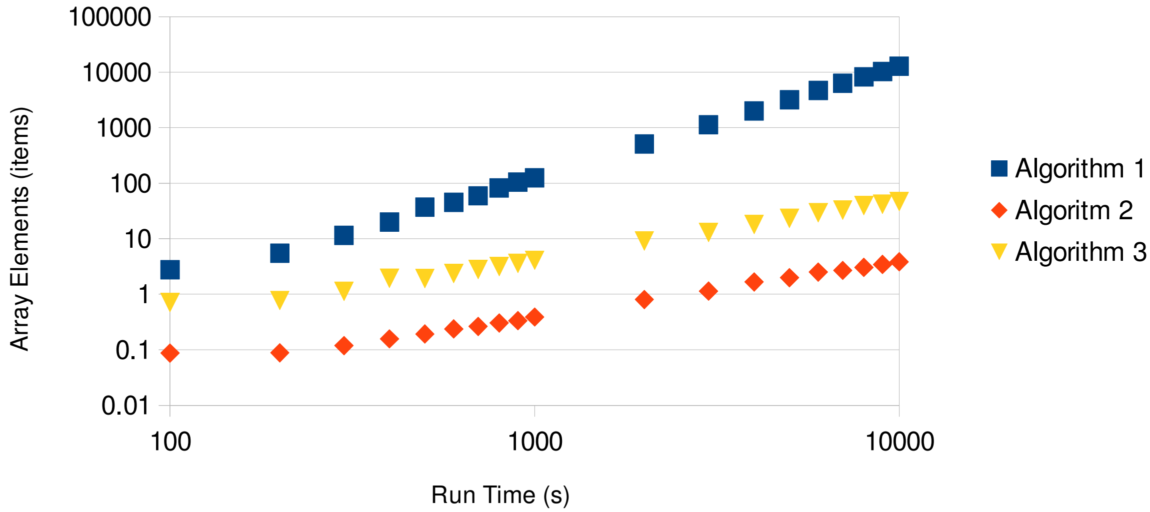
Tabulated data:

length of array (items):	Algorithm 1	Algoritm 2	Algorithm 3
100	2.7716159821	0.0872612	0.7102489471
200	5.4697990417	0.0882148743	0.7722377777
300	11.5578174591	0.1192092896	1.1301040649
400	20.049571991	0.1575946808	1.9469261169
500	37.1966362	0.1928806305	1.9254684448
600	45.1765060425	0.2381801605	2.3641586304
700	59.3752861023	0.2632141113	2.8069019318
800	82.5231075287	0.3051757813	3.1435489655
900	104.8910617828	0.3345012665	3.5936832428
1000	124.716758728	0.3890991211	4.0917396545
2000	506.4375400543	0.8039474487	9.1133117676
3000	1130.9654712677	1.1386871338	12.9764080048
4000	2003.3891201019	1.6663074493	18.19896698
5000	3167.5834655762	1.9881725311	23.2424736023
6000	4702.143907547	2.5098323822	29.3796062469
7000	6300.3859519959	2.6910305023	32.7661037445
8000	8240.9496307373	3.0426979065	39.7121906281
9000	10258.1548690796	3.4453868866	41.8004989624
10000	12838.2110595703	3.8516521454	46.7867851257

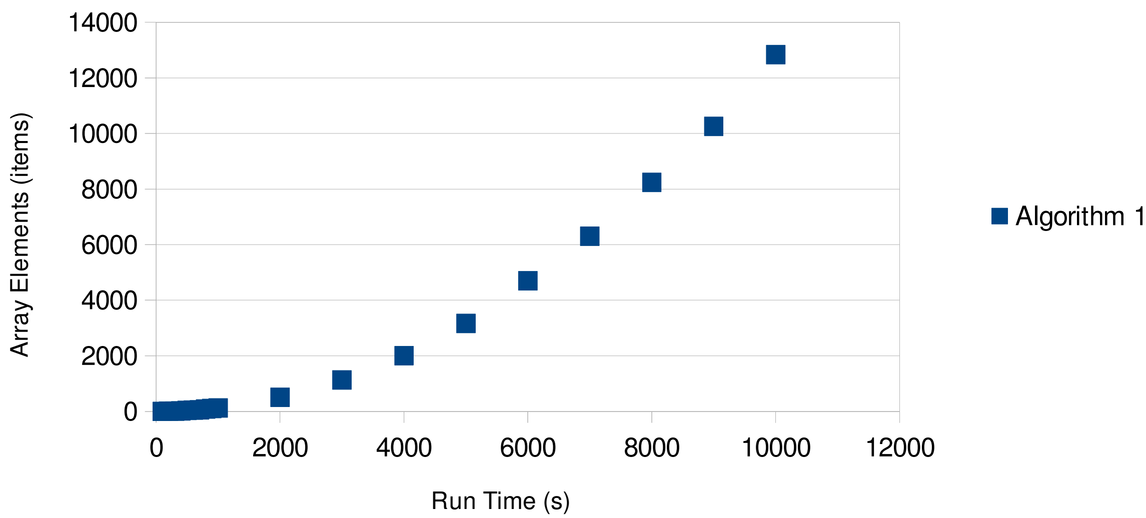
Plots

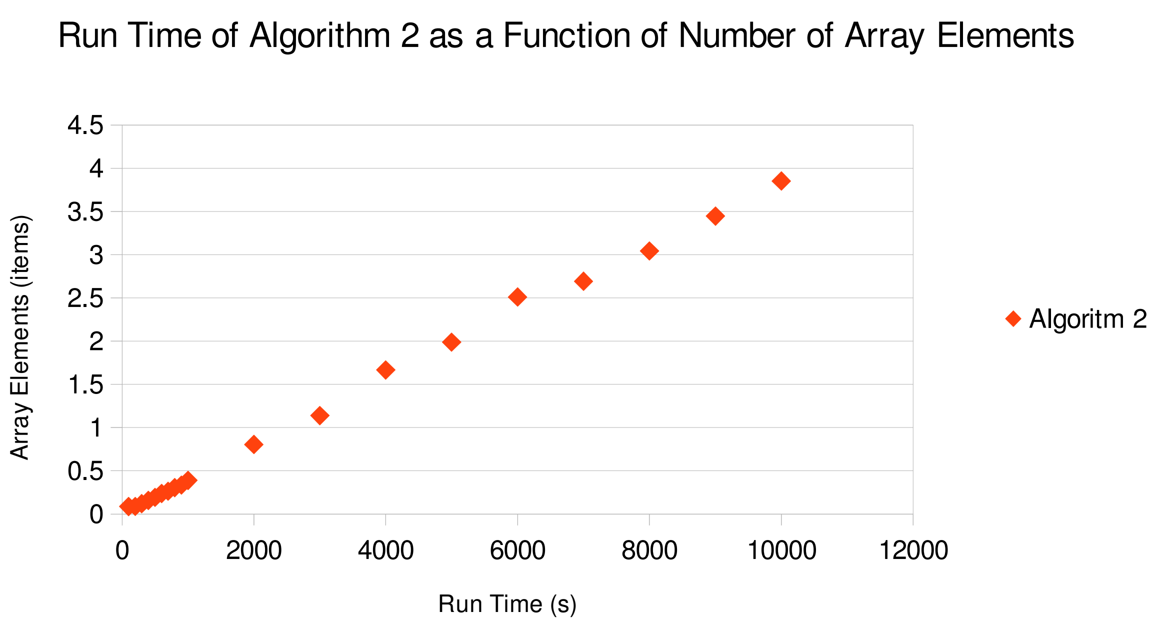


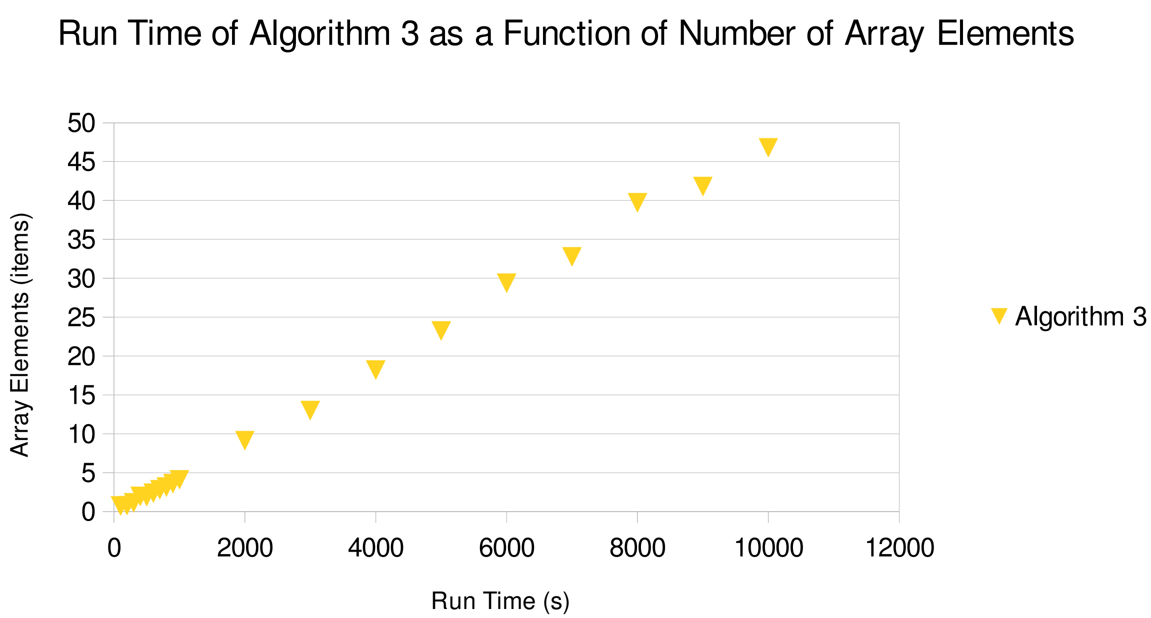
Algorithmic Run Time as a Function of Number of Array Elements, Log-Log



Run Time of Algorithm 1 as a Function of Number of Array Elements







Extrapolation and Interpretation:

Question 1:

The largest array size that can be solved in an hour can be determined by the trendline of the data of the best fit of each curve.

Algorithm 1 – best fit curve: $f(n) = 0.0002132125\ n^{1.9361128816}$, R^2 of 0.9981, slove for n:

$n = (f(n)/0.0002132125\ )^{1/1.9361128816}$

let f(n) = 1 hour = 3600 seconds

$n = (3600/0.0002132125\ )^{1/1.9361128816}$

n = 283,443

Algorithm 2 – best fit curve: $f(n) = 0.0003864076n + 0.0169365923$, R^2 of 0.9967, slove for n:

$n = (f(n) – 0.0169365923)/0.0003864076$

let f(n) = 1 hour = 3600 seconds

$n = (3600 – 0.0169365923)/0.0003864076$

n = 9,316,543

Algorithm 3 – lower bound best fit curve, linear fit: $f(n) = 0.004780369n – 0.419488129$, R^2 of 0.9983, slove for n:

$n = (f(n) - 0.419488129)/0.004780369$

let f(n) = 1 hour = 3600 seconds

$n = (3600 - 0.419488129)/0.004780369$

n = 752,992

Algorithm 3 – upper bound best fit curve, logrithmic fit: $f(n) = 10.170529441 \ln(n) – 59.5499120322$, R^2 of 0.8082, solve for n:

$n = e^{(f(n) + 59.5499120322)/10.170529441}$

let f(n) = 1 hour = 3600 seconds

$n = e^{(3600 + 59.5499120322)/10.170529441}$

n ~ e^{366} which is HUGE!

Question 2:

Log – log slopes, calculated using $m = \log(f(n)_1/f(n)_0)/\log(n_1/n_0)$:

log log slope	n1	n0	f(n)1	f(n)0	slope
Algorithm 1	10000	100	12838.2110595703	2.7716159821	1.8328857271
Algorithm 2	10000	100	3.8516521454	0.0872612	0.8224129385
Algorithm 3	10000	100	46.7867851257	0.7102489471	0.9093563029

The slope of algorithm 1 is approximately 2, making it a 2nd order quadradic. Algorithm 1 and 2 are near a slope of 1. They are nearly linear, at least according to the data collected.

At first glance it seems that the third algorithm should have had the highest performance. But it was outperformed by the second algorithm. It acted as a linear algorithm, I think this is because the linear time operation of finding the overlap is actually pretty costly, so it dominiates. I tried testing huge numbers of array elements (in the billions) but I used all my memory (16GiB!) in stack generation.

Another possibility is that algorithm 3 should be optimized for tail recurrsion, this might reduce the number of overlap operations that need to take place.