# Assignment 1: Introduction to Systems Programming

Kevin Rich, Francis Vo, Soo-Hyun Yoo

October 15, 2012

## 1 Mathematical Analysis

### 1.1 Algorithm 1

**Data**: Integer array A of size N
**Result**: Greatest Sum of Subarray

```
for i ← 0 to N do
    for j ← i to N do
        s ← 0
        for k ← i to j do
            s ← s + A[k]
        end
        if s > max then
            max ← s
        end
    end
end
```

**Algorithm 1**: Pseudocode for Basic Enumeration

This Algorithm has 3 for-loops so it is O($n^3$).

### 1.2 Algorithm 2

**Data**: Integer array A of size N
**Result**: Greatest Sum of Subarray

```
for i ← 0 to N do
    s ← 0
    for j ← i to N do
        s ← A[j]
        if s > max then
            max ← s
        end
    end
end
```

**Algorithm 2**: Pseudocode for Better Enumeration

This Algorithm has 2 for-loops so it is O($n^2$).

### 1.3 Algorithm 3

**Data**: Integer array A of size N
**Result**: Greatest Sum of Subarray

**Algorithm 3**: Pseudocode for Divide and Conquer - Starting function

```
Data: Integer array A of size N
Result: Integer array of size 4
if A.size() <= 1 then
    return sum = A[0], sum_left = A[0], sum_right = A[0], MAX = A[0]
end
Left_results ← MaxSubarray_recursion(A.Left_Side)
Right_results ← MaxSubarray_recursion(A.Right_Side)
sum ← Left_results.sum + Right_results.sum
sum_left ← Left_results.sum + Right_results.sum_left
sum_right ← Left_results.sum_right + Right_results.sum
MAX ← Left_results.sum + Right_results.sum
sum_left ← Greater(sum_left, left_results.sum_left)
sum_right ← Greater(sum_right, Right_results.sum_right)
MAX ← Greater(MAX, Right_results.MAX, Right_results.MAX)
return sum, sum_left, sum_right, MAX
```

**Algorithm 4**: Pseudocode for Divide and Conquer - Recursive function

This algorithm is recurvsive and decreases by half every step. Each lower step has double the ammount of calls. This algorithm is O($n\log(n)$)

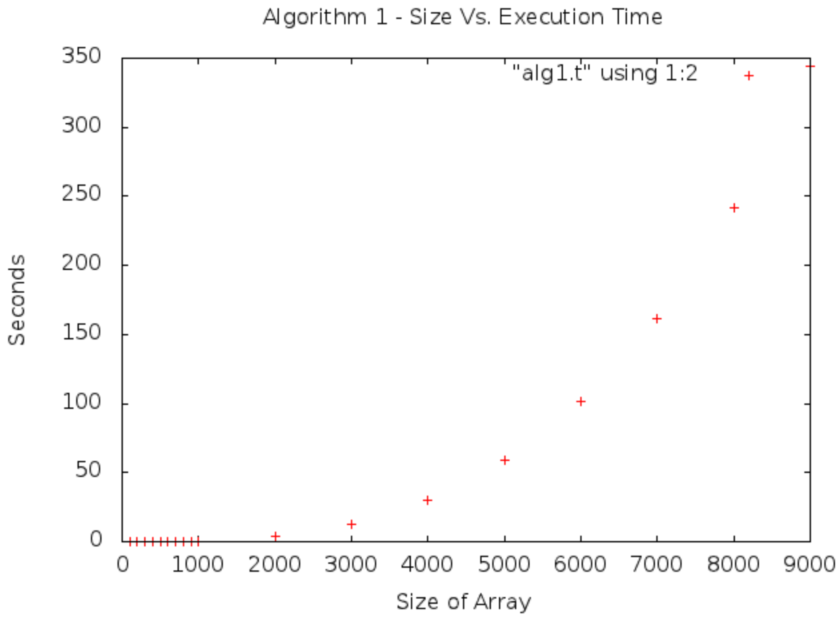
# 2 Theoretical Correctness

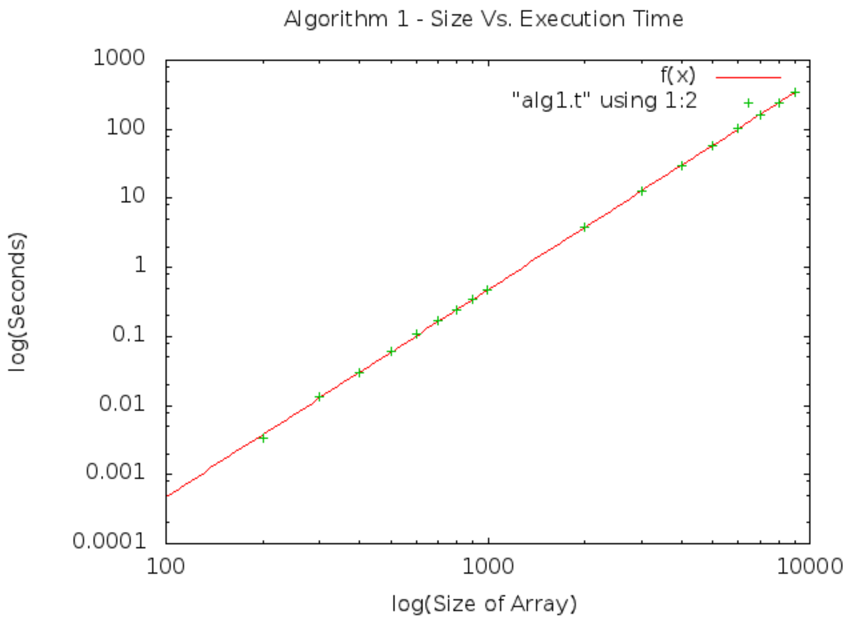
# 3 Testing

| Student ID | Answer |
|---|---|
| 931678074 | 5703 |
| 930569466 | 8184 |
| 932086449 | 4949 |

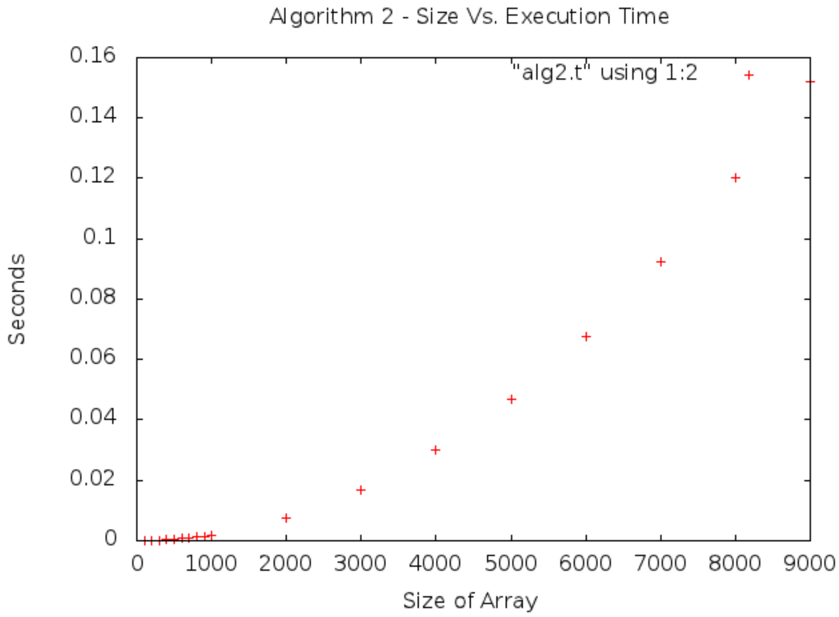
# 4 Experimental Analysis

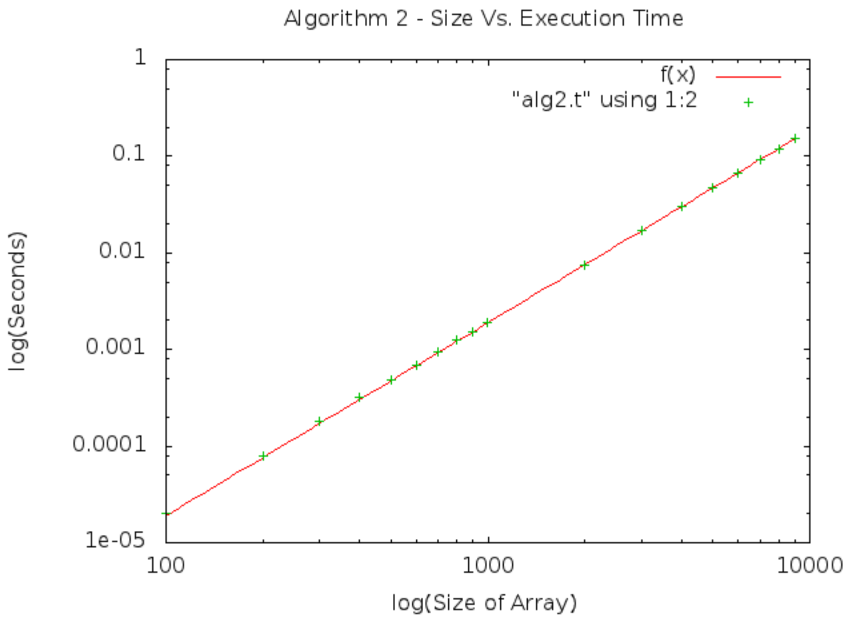
## 4.1 Algorithm 1



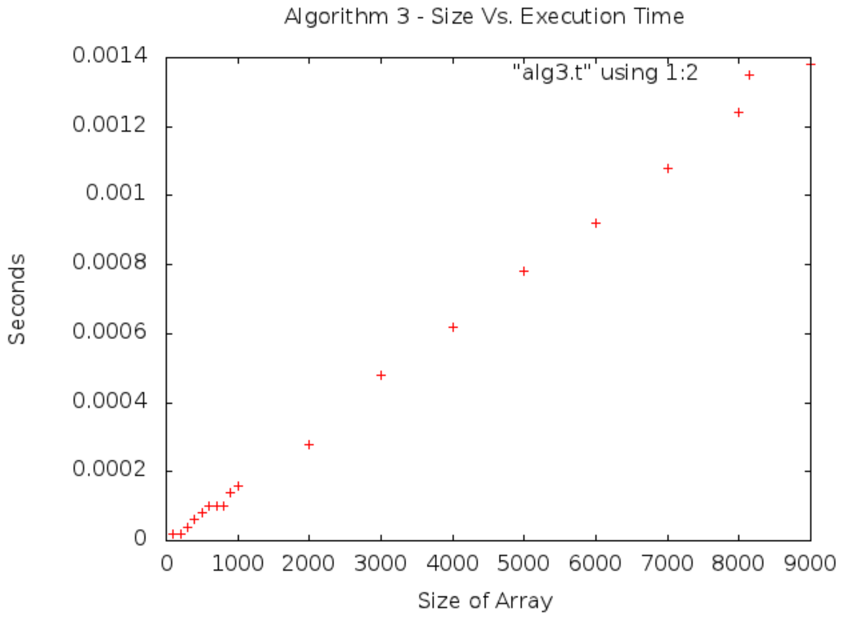



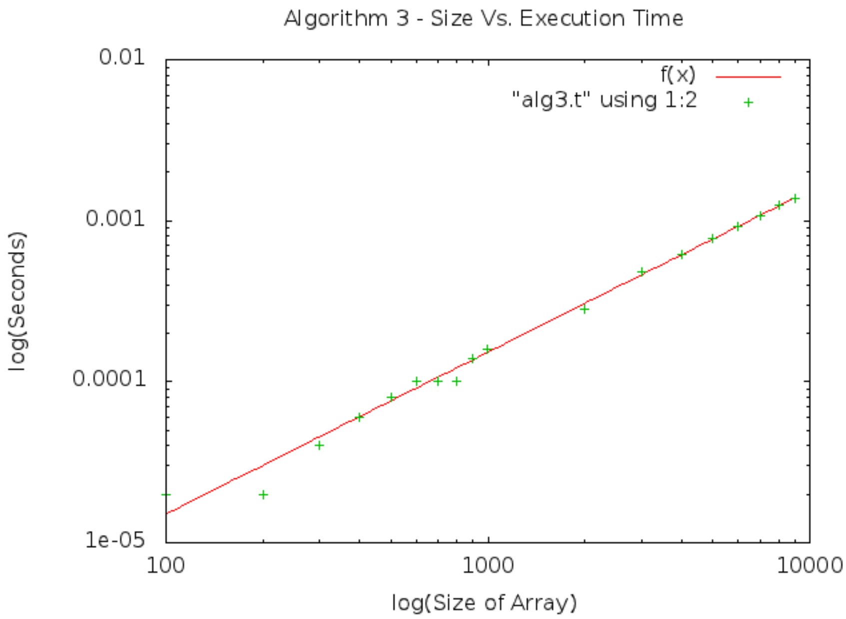
## 4.2 Algorithm 2



## 4.3 Algorithm 3



# 5 Extrapolation and Interpretation

## 5.1 Extrapolation

The functions were calculated using gnuplot's fit function.

## 5.2 Interpretation

The functions were calculated using gnuplot's fit function and fitting the data to $f(n) = 10^{m*log_{10}(n)+c}$ The slopes for each algorithm is a little lower than the actual power because of the creation overhead of the function has a larger affect on arrays with small sizes. This will cause the left side higher and therefore decreases slope.

## 5.3 Algorithm 1

### 5.3.1 Extrapolation

$f(n) = 4.71599 \times 10^{-10} \times n^3$
$f(n) = 3600 \rightarrow n = 19690$

### 5.3.2 Interpretation

Slope = 2.99734

## 5.4 Algorithm 2

### 5.4.1 Extrapolation

$f(n) = 1.87761 \times 10^{-9} \times n^2$
$f(n) = 3600 \rightarrow n = 1384678$

### 5.4.2 Interpretation

Slope = 1.99602

## 5.5 Algorithm 3

### 5.5.1 Extrapolation

$f(n) = 1.74832 \times 10^{-8} \times n \times log(n)$
$f(n) = 3600 \rightarrow n = 8984428998 = 8.98 \times 10^9$

### 5.5.2 Interpretation

Slope = 1.00506

# 6 Code

## 6.1 Files

alg1.cpp - function for algorithm 1
alg2.cpp - function for algorithm 1
alg3.cpp - function for algorithm 1
analysis.cpp - code to run algorithm and measure times for the number of array then outputs .t file
makefile - to compile files
maxSubarray.pdf -
maxSubarray.tex - to create pdf filename
test.cpp - allows input of file, and runs algorithm on input file
analysis/ - hold compiled executables for running analysis
test/ - hold compiled executables for running tests on code, and test array files
timingfiles/ - holds files for creating plots
timingfiles/*.t - files that holds rum times for different array sizes
timingfiles/*.gp - code for gnuplot. 2 plots of each algorithm: 1 normal plot, and 1 log-log plot

## 6.2 Algorithm 1

```
/*
 * Enumeration
 * Loop over each pair of indices i; j and compute the sum from k=i to j of a[k].
 * Keep the best sum you have found so far.
 */

using namespace std;


int MaxSubarray(int a[], int n){

  int i,j,k;
  int max = a[0];
  int sum;
  for(i = 0; i < n; ++i ){
    for (j = i; j < n; ++j){
      sum = 0;
      for (k = i; k <=j; ++k){
        sum += a[k];
      }
      if(max < sum){
        max = sum;
      }
    }
  }
  return max;
}
```

alg1.cpp

### 6.3 Algorithm 2

```cpp
/*
 * Better Enumeration
 * Notice that in the previous algorithm, the same sum is computed many times.
 * In particular, notice that sum from k=i to j of a[k] can be computed from sum from k=i to j - 1 of a[k] in O(1) time, rather than starting from scratch.
 * Write a new version of the frst algorithm that takes advantage of this observation.
 */


using namespace std;


int MaxSubarray(int a[], int n){

  int i,j,k;
  int max = a[0];
  int sum;
  for(i = 0; i < n; ++i ){
    sum = 0;
    for (j = i; j < n; ++j){
      sum += a[j];
      if(max < sum){
        max = sum;
      }
    }
  }
  return max;
}
```

alg2.cpp

### 6.4 Algorithm 3

```cpp
/*
 * Divide and Conquer
 * If we split the array into two halves, we know that the maximum subarray will either be
 *      * contained entirely in the frst half,
 *      * contained entirely in the second half, or
 *      * made of a suffix of the frst half of maximum sum and a prefix of the second half of maximum sum
 * The frst two cases can be found recursively. The last case can be found in linear time.
 */

#define ALL     0
#define LEFT    1
#define RIGHT   2
#define OVERALL 3

using namespace std;

void MaxSubarray_h(int array[], int size, int sums[]){
  // Base case.
  if(size <= 1){
    sums[ALL]     = array[0];   // Sum of entire array
    sums[LEFT]    = array[0];   // Largest sum from left end of array
    sums[RIGHT]   = array[0];   // Largest sum from right end of array
    sums[OVERALL] = array[0];   // Largest sum found so far
    return;
  }
  int i = size/2;   // Index of middle element

  // Recurse.
  int *left  = new int[4];
  int *right = new int[4];
  MaxSubarray_h(array,i, left);
  MaxSubarray_h(array+i, size-i, right);

  // Calculate various possible maximum sums.
  int a = left[ALL]   + right[ALL];    // Sum of everything
  int l = left[ALL]   + right[LEFT];   // Possible max sum from the left
  int r = left[RIGHT] + right[ALL];    // Possible max sum from the right
  int m = left[RIGHT] + right[LEFT];   // Possible max sum straddling both branches

  // Check for and find new maximums.
  l = l > left[LEFT] ? l : left[LEFT];    // Is the new left sum larger?
  r = r > right[RIGHT] ? r : right[RIGHT];    // Is the new right sum larger?
  int overall = left[OVERALL] > right[OVERALL] ? left[OVERALL] : right[OVERALL];
  overall = overall > m ? overall : m;

  // Final answers!
  sums[0] = a;
  sums[1] = l;
  sums[2] = r;
  sums[3] = overall;
}


int MaxSubarray(int a[], int n){
  int *p = new int[4];
  MaxSubarray_h(a,n,p);
  int s1 = p[0] > p[1] ? p[0] : p[1];
  int s2 = p[2] > p[3] ? p[2] : p[3];
  s1 = s1 > s2 ? s1 : s2;
  return s1;
}
```

alg3.cpp