Assignment 1: Introduction to Systems Programming

Kevin Rich, Francis Vo, Soo-Hyun Yoo

October 17, 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

```
def MaxSubarray:
    sums = MaxSubarray_recursive(A)
    return max(sums)
```

Algorithm 3.1: Starting function pseudocode for Divide and Conquer

Data: Integer array A of size N
Result: Integer array of size 4

```
def MaxSubarray_recursive:
    if A.size <= 1:
        sums.all = A[0]
        sums.left = A[0]
        sums.right = A[0]
        sums.overall = A[0]
        return sums

    left_sums = MaxSubarray_recursive(A, left_branch)
    right_sums = MaxSubarray_recursive(A, right_branch)

    sums.all = left_sums.all + right_sums.all
    sums.left = max(left_sums.left, left_sums.all + right_sums.left)
    sums.right = max(right_sums.right, left_sums.right + right_sums.all)
    m = left_sums.right + right_sums.left
    sums.overall = max(sums.all, sums.left, sums.right, m)

    return sums
```

Algorithm 3.2: Recursive function pseudocode for Divide and Conquer

This algorithm is recursive and decreases by half every step. Each lower step has double the number of calls. Thus, this algorithm is O($n \log n$).

2 Theoretical Correctness

Claim 1: Given an array a containing n positive integers $a_0, a_1, \ldots, a_{n-1}$ for $n > 0$, the divide-and-conquer algorithm (algorithm 3) correctly calculates the sum of the maximum subarray, $s = \max_{i \leq j} \left(\sum_{k=i}^{j} a_k \right)$, for positive integers $i, j < n$.

Proof: As a base case, consider when $n = 1$. Then `MaxSubarray_recursive`$(n) = a_0$, which is true.

For the inductive hypothesis, assume that for $n > 1$ and $n \leq q$ for some positive integer $q > 1$, the algorithm correctly computes the sum of the maximum subarray.

Consider an array of size $n = q + 1$. Then we can consider one of four cases regarding the location of the maximum subarray within the whole array.

Case 1: $s = a$. We correctly capture s in `sums.all`.

Case 2: $s = \sum_{k=0}^{j} a_k$, for $j < q$. We correctly capture s in `sums.left`.

Case 3: $s = \sum_{k=i}^{q} a_k$, for $i > 0$. We correctly capture s in `sums.right`.

Case 4: $s = \sum_{k=i}^{j} a_k$, for $0 < i \leq j < q$. We correctly capture s in `m`.

In all four cases, we correctly select the maximum sum among `sums.all`, `sums.left`, `sums.right`, and `m` as the sum of the maximum subarray of a.

$\square$

Claim 2: The algorithm terminates.

Proof: Since $n > 0$ per the problem statement, n must be at least 1, and the algorithm returns. This proves the base case.

For the inductive hypothesis, assume that the algorithm returns for an array of length $n \leq q$ for some positive integer $q > 1$. Consider $n = q + 1$. The array will be split up into two branches of positive lengths, which means the branches will have lengths less than or equal to q. Thus, the algorithm will return for each branch, and the algorithm returns right afterwards.

$\square$

Claim 3: The divide-and-conquer algorithm computes the sum of the maximum subarray in $\mathrm{O}(n \log n)$ time.

Proof: Let n be the size of the array of integers, a. For $n > 1$, the recurrence for the recursive step of the algorithm can be found to be

$$\begin{aligned} T(n) &= \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(1) \\ &= 2T\left(\frac{n}{2}\right). \end{aligned}$$

In its entirety,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) & \text{if } n > 1 \end{cases}.$$

Suppose $T(n) \leq n \log n = \mathrm{O}(n \log n)$.

As a base case, let $n = 1$. Then $T(1) = 1 \log 1 = \Theta(1)$, as desired.

For the inductive hypothesis, assume $T(m) \leq m \log m$ for $m < n$. Then if $m = \frac{n}{2}$,

$$\begin{aligned} T\left(\frac{n}{2}\right) &\leq \frac{n}{2} \log \frac{n}{2} \\ &\leq n \log n \\ &= \mathrm{O}(n \log n). \end{aligned}$$

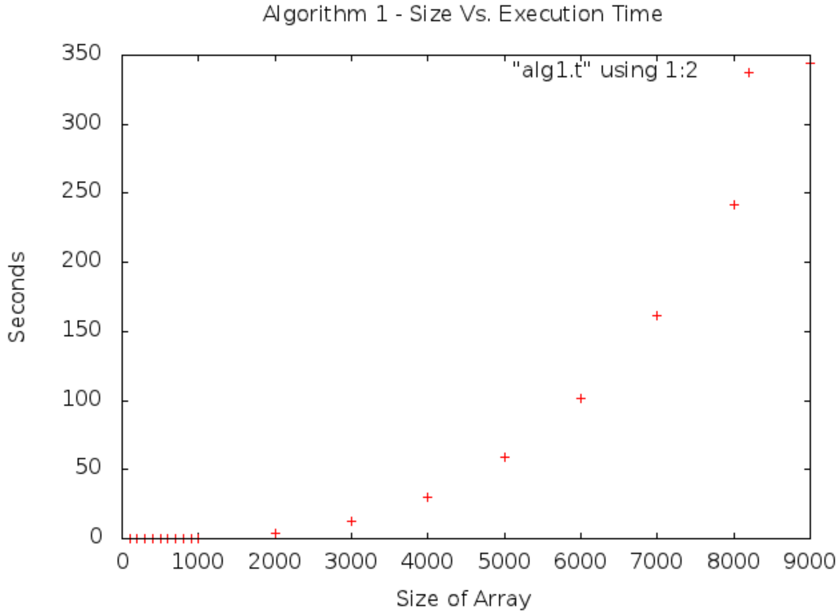
$\square$

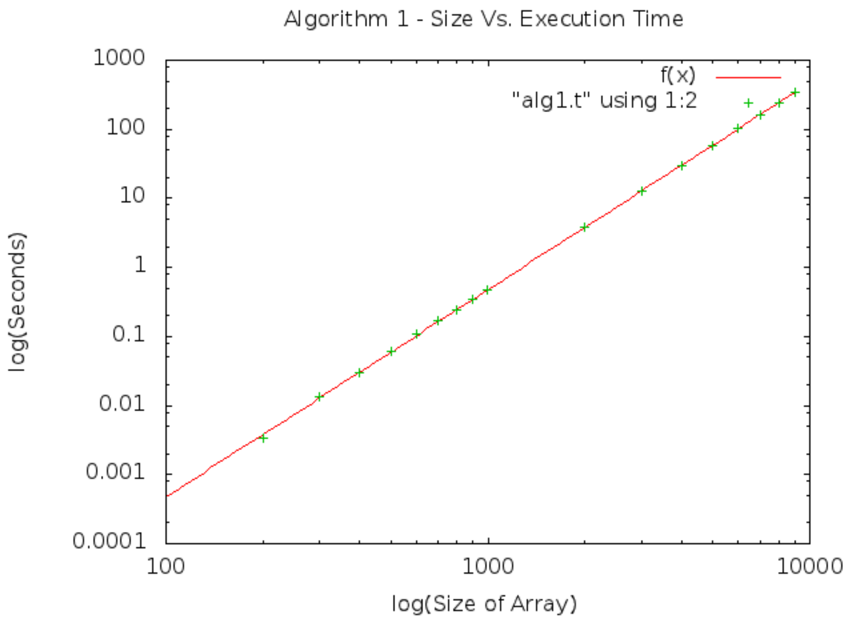
3 Testing

Student ID	Answer
931678074	5703
930569466	8184
932086449	4949

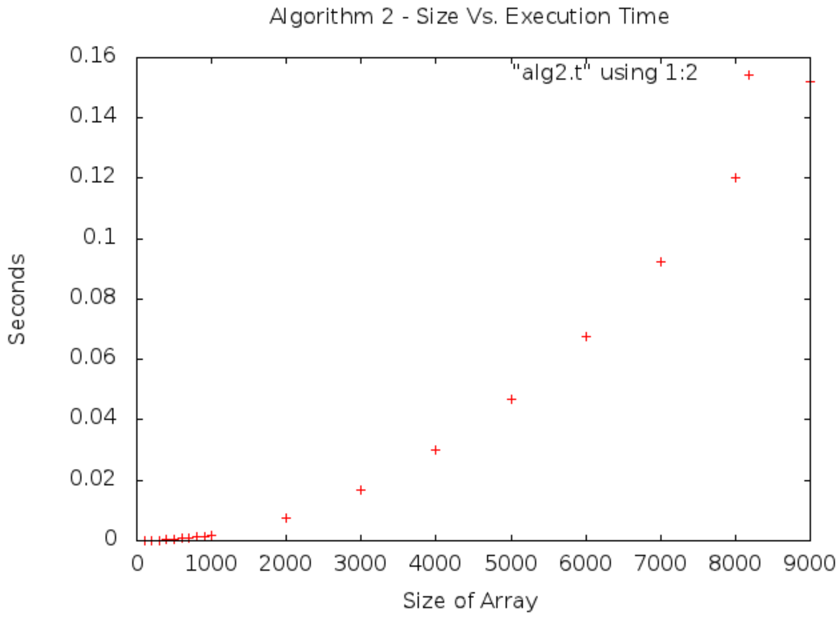
4 Experimental Analysis

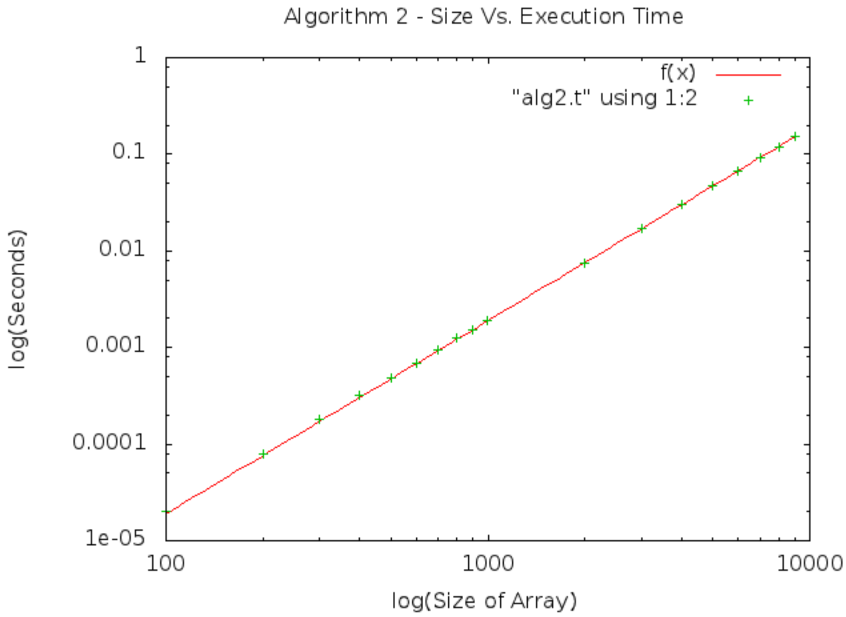
4.1 Algorithm 1



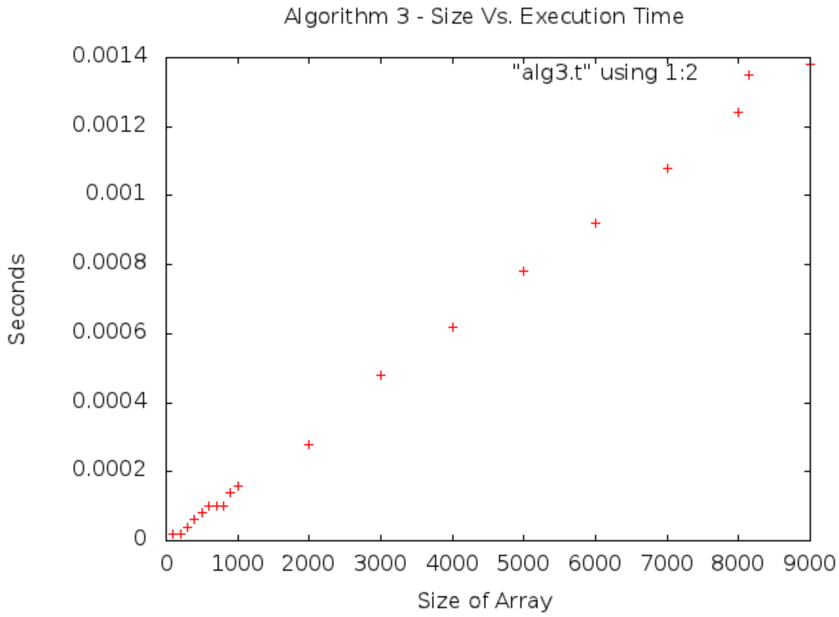


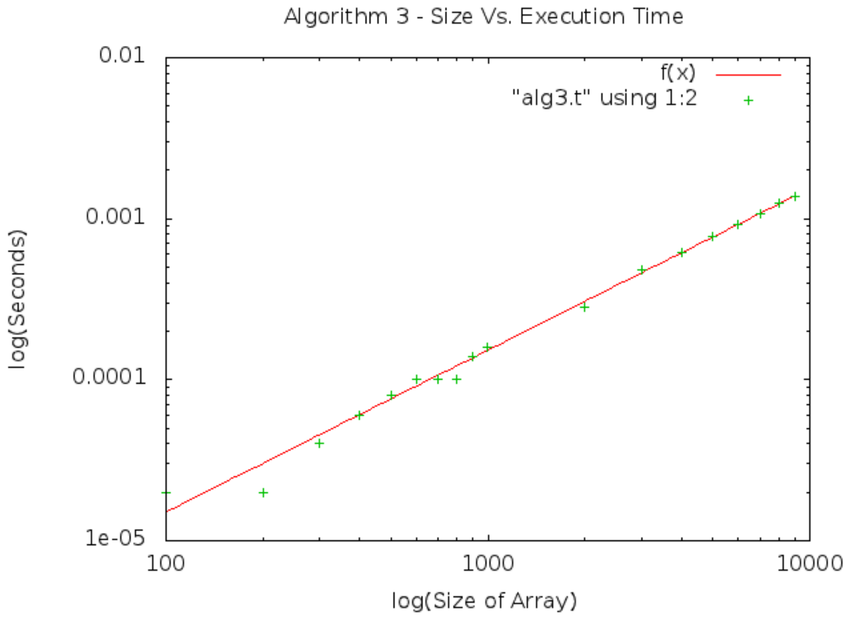
4.2 Algorithm 2





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 to be higher and therefore decreases slope.

5.3 Algorithm 1

5.3.1 Extrapolation

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

5.3.2 Interpretation

Slope = $\boxed{2.99734}$

5.4 Algorithm 2

5.4.1 Extrapolation

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

5.4.2 Interpretation

Slope = $\boxed{1.99602}$

5.5 Algorithm 3

5.5.1 Extrapolation

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

5.5.2 Interpretation

Slope = $\boxed{1.00506}$

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 - This writeup
maxSubarray.tex - TEXfile for PDF
test.cpp - Allows input of file and runs algorithm on input file
analysis/ - Contains compiled executables for running analysis
test/ - Contains compiled executables for running tests on code, and test array files
timingfiles/ - Contains files for creating plots
timingfiles/*.t - Log of runtimes 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