

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

```
1 for i ← 0 to N do
2   for j ← i to N do
3     s ← 0
4     for k ← i to j do
5       s ← s + A[k]
6     end
7     if s > max then
8       max ← s
9     end
10  end
11 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

```
1 for i ← 0 to N do
2   s ← 0
3   for j ← i to N do
4     s ← A[j]
5     if s > max then
6       max ← s
7     end
8   end
9 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 integers a_0, a_1, \dots, 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 integers $i, j < n$.

We propose two methods of inductive proof: top-down and bottom-up.

The sum of all the elements in array denoted as `sums.all`
 The largest sum starting from the left denoted as `sums.left`
 The largest sum starting from the right denoted as `sums.right`
 The overall max sum denoted as `sums.overall`

Proof (top-down): As a base case, let $n = 1$. Then `sums.all` = $A[0]$, `sums.left` = $A[0]$, `sums.right` = $A[0]$, `sums.overall` = $A[0]$

For the inductive hypothesis, consider:

`left_sums` = `MaxSubarray_recursive`($A[0:\frac{n}{2}-1]$)
`right_sums` = `MaxSubarray_recursive`($A[\frac{n}{2}:n]$)
`sums.all` = `left_sums.all` + `right_sums.all`
`sums.left` = $\max(\text{left_sums.left}, \text{left_sums.all} + \text{right_sums.left})$
`sums.right` = $\max(\text{right_sums.right}, \text{left_sums.right} + \text{right_sums.all})$
`sums.overall` = $\max(\text{sums.all}, \text{sums.left}, \text{sums.right}, \text{left_sums.right} + \text{right_sums.left})$

We can consider three cases:

Case 1: Contained entirely in the first half

This will be returned as `left_sums.overall` from the recursive call on `left_sums`.

Case 2: Contained entirely in the second half

This will be returned from the recursive call on `right_sums`.

Case 3: Made of a suffix of the first half of maximum sum and the prefix of the second half of the maximum This will be found using `left_sums.right` + `right_sums.left`

□

Proof (bottom-up): 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 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 .

□

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.

□

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

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

where

- The base case takes $\Theta(1)$,
- The recursive calls take $2T\left(\frac{n}{2}\right)$,
- The `max()` calculations take $\Theta(n)$, and
- The final `return` takes $\Theta(1)$.

In its entirety,

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

Suppose $T(n) \leq cn \log n + n = O(n \log n)$. Then

$$\begin{aligned} T(n) &\leq 2\left(c \cdot \frac{n}{2} \log \frac{n}{2}\right) + n \\ &\leq cn \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &\leq cn \log n \\ &= O(n \log n), \end{aligned}$$

as desired.

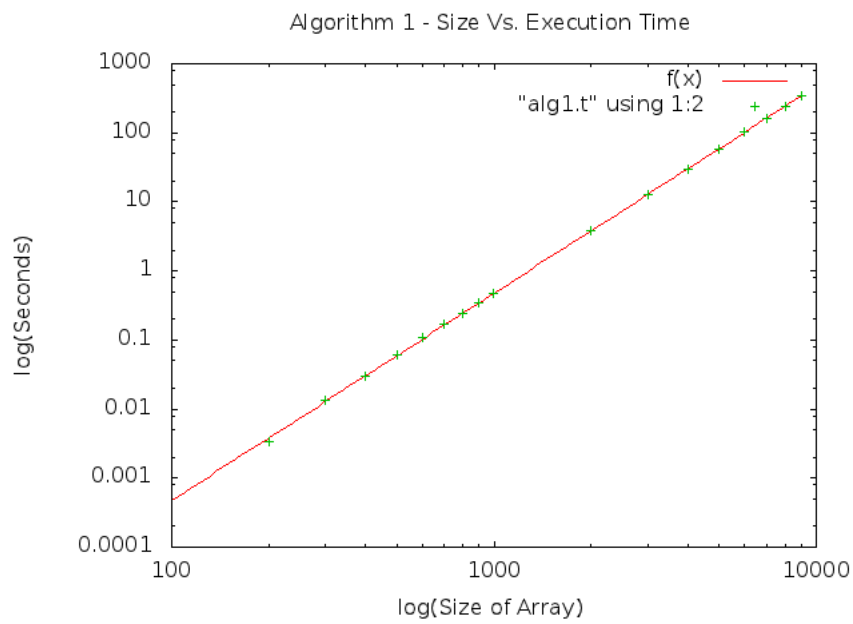
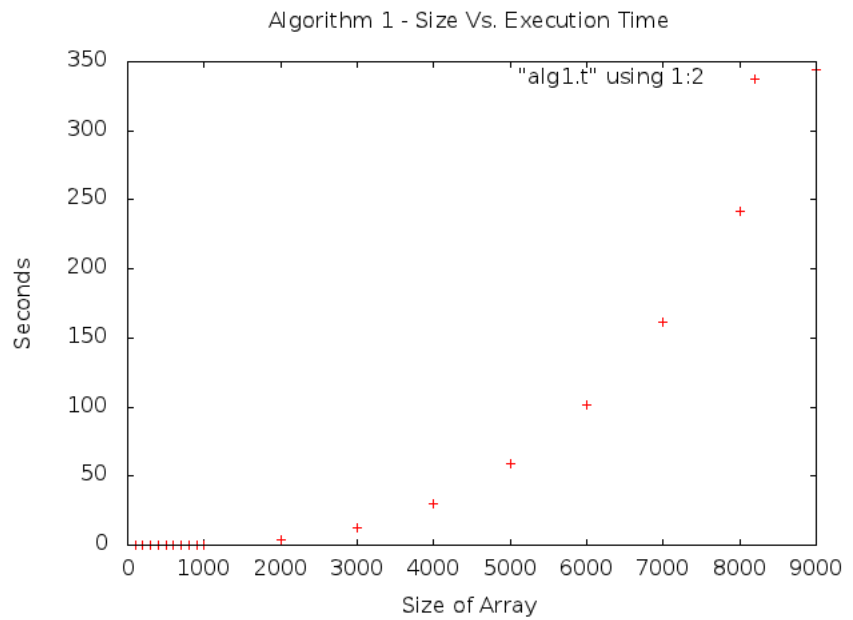
□

3 Testing

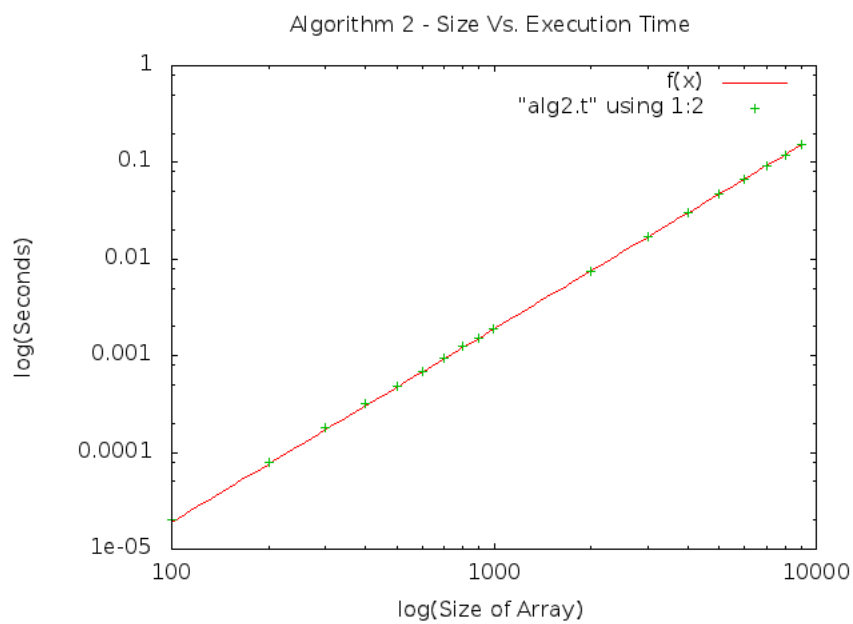
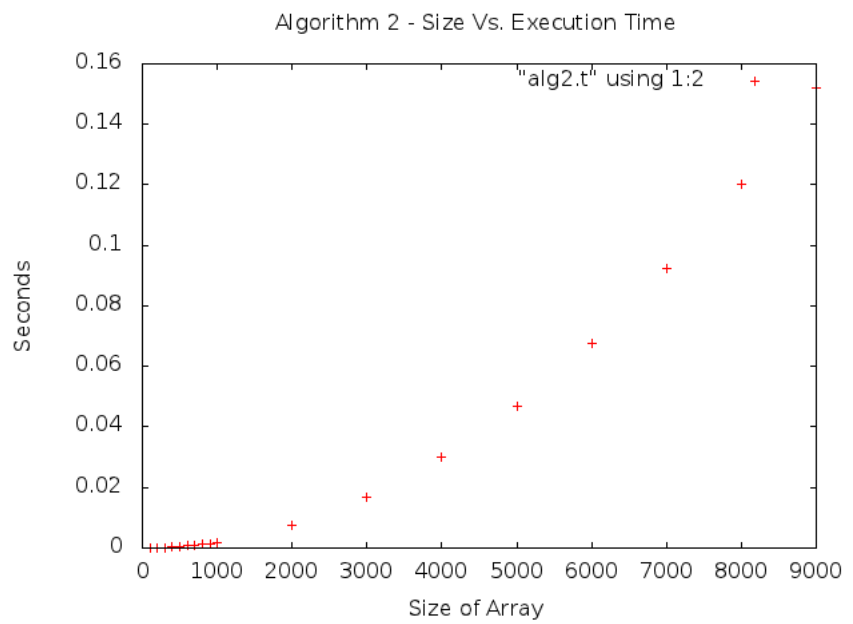
Student ID	Answer
931678074	5703
930569466	8184
932086449	4949

4 Experimental Analysis

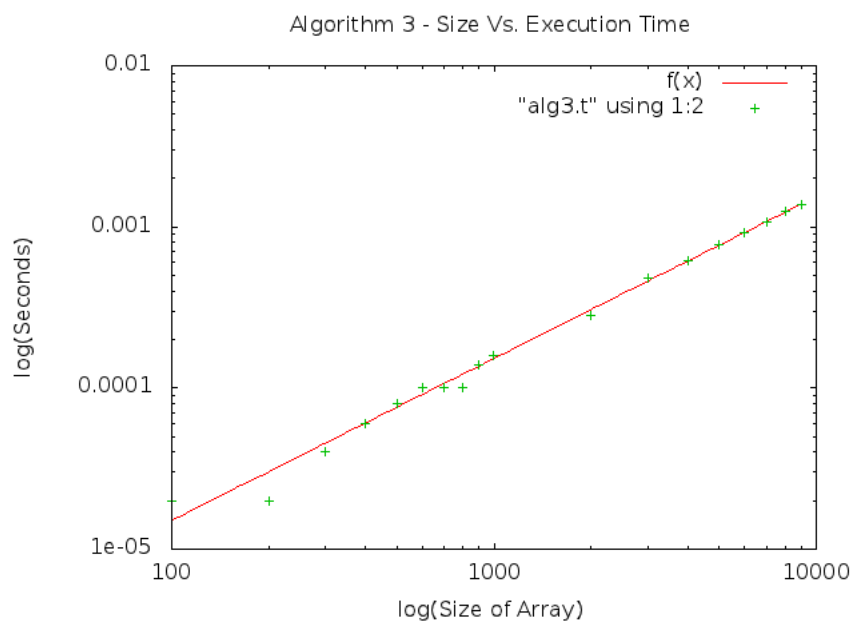
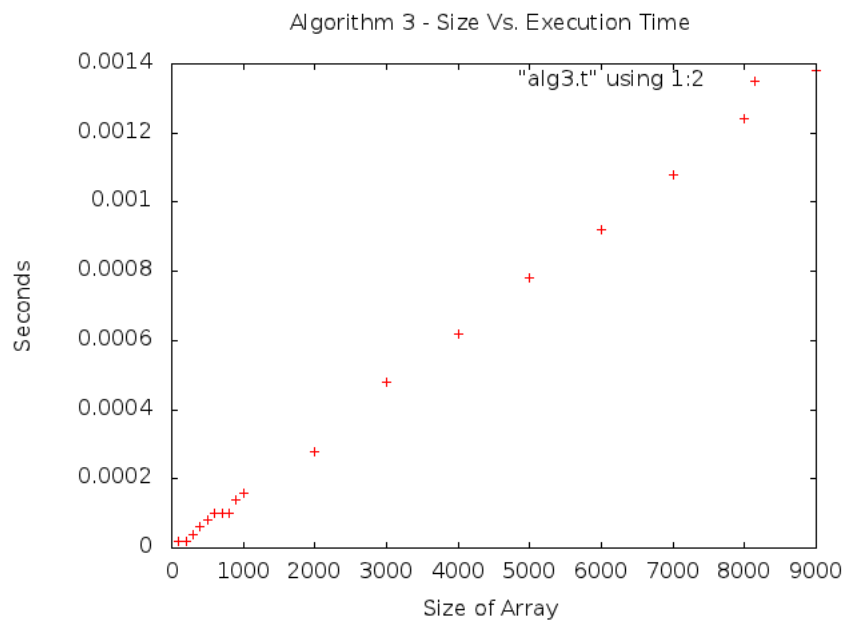
4.1 Algorithm 1



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

$$\text{Slope} = \boxed{2.99734}$$

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

$$\text{Slope} = \boxed{1.99602}$$

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

$$\text{Slope} = \boxed{1.00506}$$

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 - T_EXfile 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

```
1  /*
2  * Enumeration
3  * Loop over each pair of indices i; j and compute the sum from k=i to j of a[k].
4  * Keep the best sum you have found so far.
5  */
6
7  using namespace std;
8
9
10 int MaxSubarray(int a[], int n){
11
12     int i, j, k;
13     int max = a[0];
14     int sum;
15     for (i = 0; i < n; ++i) {
16         for (j = i; j < n; ++j) {
17             sum = 0;
18             for (k = i; k <= j; ++k) {
19                 sum += a[k];
20             }
21             if (max < sum) {
22                 max = sum;
23             }
24         }
25     }
26     return max;
27 }
```

alg1.cpp

6.3 Algorithm 2

```
1  /*
2  * Better Enumeration
3  * Notice that in the previous algorithm, the same sum is computed many times.
4  * In particular, notice that sum from k=i to j of a[k] can be computed from sum from k=
5  * i to j - 1 of a[k] in O(1) time, rather than starting from scratch.
6  * Write a new version of the first algorithm that takes advantage of this observation.
7  */
8
9  using namespace std;
10
11
12  int MaxSubarray(int a[], int n){
13
14      int i, j, k;
15      int max = a[0];
16      int sum;
17      for (i = 0; i < n; ++i) {
18          sum = 0;
19          for (j = i; j < n; ++j) {
20              sum += a[j];
21              if (max < sum) {
22                  max = sum;
23              }
24          }
25      }
26      return max;
27  }
```

alg2.cpp

6.4 Algorithm 3

```

1  /*
2  * Divide and Conquer
3  * If we split the array into two halves, we know that the maximum subarray will either
4  *   be
5  *   * contained entirely in the first half,
6  *   * contained entirely in the second half, or
7  *   * made of a suffix of the first half of maximum sum and a prefix of the second
8  *     half of maximum sum
9  * The first two cases can be found recursively. The last case can be found in linear
10 * time.
11 */
12
13 #define ALL      0
14 #define LEFT     1
15 #define RIGHT    2
16 #define OVERALL  3
17
18 using namespace std;
19
20 void MaxSubarray_h(int array[], int size, int sums[]) {
21     // Base case.
22     if (size <= 1) {
23         sums[ALL] = array[0];    // Sum of entire array
24         sums[LEFT] = array[0];   // Largest sum from left end of array
25         sums[RIGHT] = array[0];  // Largest sum from right end of array
26         sums[OVERALL] = array[0]; // Largest sum found so far
27         return;
28     }
29     int i = size/2;    // Index of middle element
30
31     // Recurse.
32     int *left = new int[4];
33     int *right = new int[4];
34     MaxSubarray_h(array, i, left);
35     MaxSubarray_h(array+i, size-i, right);
36
37     // Calculate various possible maximum sums.
38     int a = left[ALL] + right[ALL];    // Sum of everything
39     int l = left[ALL] + right[LEFT];   // Possible max sum from the left
40     int r = left[RIGHT] + right[ALL];  // Possible max sum from the right
41     int m = left[RIGHT] + right[LEFT]; // Possible max sum straddling both branches
42
43     // Check for and find new maximums.
44     l = l > left[LEFT] ? l : left[LEFT]; // Is the new left sum larger?
45     r = r > right[RIGHT] ? r : right[RIGHT]; // Is the new right sum larger?
46     int overall = left[OVERALL] > right[OVERALL] ? left[OVERALL] : right[OVERALL];
47     overall = overall > m ? overall : m;
48
49     // Final answers!
50     sums[0] = a;
51     sums[1] = l;
52     sums[2] = r;
53     sums[3] = overall;
54 }
55
56 int MaxSubarray(int a[], int n) {
57     int *p = new int[4];
58     MaxSubarray_h(a, n, p);
59     int s1 = p[0] > p[1] ? p[0] : p[1];
60     int s2 = p[2] > p[3] ? p[2] : p[3];
61     s1 = s1 > s2 ? s1 : s2;
62     return s1;
63 }

```

alg3.cpp