CS 325 Project 1: Maximum Sum Subarray

Group 26: Aaron Boutin, Edward Francis, Jessica Spokoyny

Project requirements

For this project, you will design, implement and analyze (both experimentally and mathematically) four algorithms for the maximum subarray problem: Given an array of small integers A[1,…,n], compute max $i \leq j \sum A[k]\ j\ k = i$ For example, MAXSUBARRAY([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])=187.

You may use any language you choose to implement your algorithms. All algorithms will take as input an array and output the subarray with a maximum sum along with the sum.

Design and implementation of four algorithms for finding maximum subarray within an input array

Algorithm 1: Enumeration

Pseudocode

A simple, “brute force” approach to finding a maximum subarray within an input array could be based on finding the size of each possible subarray, with each subarray identified by a unique pair of start and end indexes, and then returning the maximum sum and the pair of indexes that produced it. The following pseudocode describes this algorithm.

```
Function findmaxsubarray(array){
        Let maxlowerindex = maxupperindex = 0
        Let maxsum = array[0]
        For each i in i = 0… i = array.size - 1
                For each j in j = i… array.size - 1
                        Let currentsum = 0
                        For k = i… j
                                Currentsum += array[k]
                        If i = j
                                Currentsum -= array[j] (prevents double-counting)
                        If Currentsum > maxsum
```

Maxsum = currentsum
Maxlowerindex = i
Maxupperindex = j
Return (maxlowerindex, maxupperindex, maxsum)}

C++ implementation

The following C++ function implements the enumeration algorithm for finding a subarray with maximum sum within an array. It takes an array as its first parameter, along with references to integers that will hold the lower and upper subarray indexes and subarray sum, when the function returns.

```
void findmax_enum(int* array, int size, int* lower, int*upper, int*maxarraysum){
   int ml=0;
   int mu=0;
   int i=0;
   int j=0;
   int max = array[0];
   for (i = 0; i < size; i++){
      for (j = i; j < size; j++){
         int total=0;
         for (int k = i; k <= j; k++){
            total+= array[k];
         }
         if (i == j)
            total -= array[j];
         if (total > max){
            max = total;
            ml=i;
            mu=j;
         }
      }
   }
   *lower = ml;
   *upper = mu;
   *maxarraysum = max;
   return;
}
```

Theoretical performance analysis

The following discussion explains why the simple enumeration algorithm is $\Theta(n^3)$.
Let n = the size of the array
The outer loop (i) runs from 1 to n = $\Theta(n)$

The first nested loop (j) runs from i to n = Θ(n)
And the second nested loop (k) runs from i to j = Θ(n)
The operations within the loops run in constant time = Θ(1)
This means that the run-time of the entire algorithm is $\Theta(n^3)$

Algorithm 2: Better enumeration

Pseudocode

A slightly better, yet still “brute force,” approach to finding a maximum subarray within an input array could be based on the preceding enumeration algorithm, with an adaptation to avoid recalculating sums quite to repetitively. The following pseudocode describes this algorithm.

```
Function findmaxsubarray_better(array){
        Let maxlower = maxupper = 0;
        Let maxsum = array[0];
        For i = 0… array.size -1
                CurrentSum = 0;
                For j = i… array.size - 1
                        CurrentSum += array[j]
                        If CurrentSum > maxsum
                                Maxsum = CurrentSum
                                Maxlower = i
                                Maxupper = j
        Return (maxlower, maxupper, maxsum)
```

C++ implementation

```
void findmax_enum_better(int* array, int size, int* lower, int*upper, int*maxarraysum){
   int ml=0;
   int mu=0;
   int i=0;
   int j=0;
   int max = array[0];
   for (i = 0; i < size; i++){
      int total = 0;
      for ( j = i; j < size; j++) {
         total+= array[j];
         if (total > max){
            max = total;
            ml=i;
            mu=j;
         }
      }
```

```
    }
    *lower = ml;
    *upper = mu;
    *maxarraysum = max;
    return;
}
```

Theoretical performance analysis

The following theoretical analysis explains why the improved enumeration algorithm is $\Theta(n^2)$.

Let n = the size of the array
The outer loop (i) runs from 1 to n = $\Theta(n)$
The inner loop (j) runs from i to n = $\Theta(n)$
The operations within the loops run in constant time = $\Theta(1)$
This means that the run-time of the entire algorithm is $\Theta(n^2)$

Algorithm 3: Divide and conquer (recursive) algorithm

Pseudocode

A divide and conquer (recursive) algorithm may seem more difficult to implement than a brute force enumeration algorithm, but it offers performance advantages, particularly as the size of the input array grows large. The following pseudocode explains the basic approach to a divide and conquer algorithm for finding the maximum subarray within an input array. The function calls itself recursively to find the max subarray within its left and right halves, and it users a helper function to find the maximum subarray that crosses the array midpoint, then returns the maximum subarray found out of these three possible choices.

```
Function findmaxsubarray_recursive(array, int arraystart, int arrayend){
        As the base case, if arraystart == arrayend (single element array),
                maxlower = arraystart
                maxupper = arrayend
                Maxsum = array[arraystart]
                Return (maxlower, maxupper, maxsum)
        If base case not reached, use recursion
                Let midpoint = arraystart + arrayend  / 2
                Maxleft = findmaxsubarray_recursive(array, arraystart, mid)
                Maxright = findmaxsubarray_recursive(array, mid + 1, arrayend)
                MaxCrossing = _findmaxcrossingsubarray(array, arraystart, mid, arrayend)
                Return max (maxleft, maxright, maxcrossing)}

Function _findmaxcrossingsubarray(array, int arraystart, int arraymidpoint, int arrayend){
```

```
        Let leftsum = array[arraymidpoint]
        Let rightsum = array[arraymidpoint + 1]
        Let maxleft = arraymidpoint
        Let maxright = arraymidpoint + 1
        Let currentSum = 0
        For i = arraymidpoint… arraystart
                currentSum += array[i]
                If currentSum > leftsum
                        Leftsum = currentSum
                        Maxleft = i
        Let currentSum = 0
        For j = arraymidpoint + 1… arrayend
                currentSum += array[j]
                If currentSum > rightsum
                        rightsum = currentSum
                        maxright = j
        Let crosslow = maxleft
        Let crosshigh = maxright
        Let crosssum = leftsum + rightsum
        Return (crosslow, crosshigh, crosssum)}
```

C++ implementation

```
void findmax_recursive(int* array, int lowerbound, int upperbound, int* lower, int*upper,
int*maxarraysum){
   if (lowerbound == upperbound){
      *lower = lowerbound;
      *upper = upperbound;
      *maxarraysum = array[lowerbound];
      return;}
   else{
      int mid = (lowerbound + upperbound ) / 2;
      int leftlow, lefthigh, leftsum;
      findmax_recursive(array, lowerbound, mid, &leftlow, &lefthigh, &leftsum);
      int rightlow, righthigh, rightsum;
      findmax_recursive(array, mid+1, upperbound, &rightlow, &righthigh, &rightsum);
      int crosslow, crosshigh, crosssum;
      _findmaxcrossingsubarray(array, lowerbound, mid, upperbound, &crosslow, &crosshigh,
&crosssum);
      if ((leftsum >= rightsum) && (leftsum >= crosssum)){
         *lower = leftlow;
         *upper = lefthigh;
```

```
            *maxarraysum = leftsum;
            return;
        }
        else if ((rightsum >= leftsum) && (rightsum >= crosssum)){
            *lower = rightlow;
            *upper = righthigh;
            *maxarraysum = rightsum;
            return;
        }
        else{
            *lower = crosslow;
            *upper = crosshigh;
            *maxarraysum = crosssum;
            return;
        }
    }
    return;
}

void _findmaxcrossingsubarray(int* array, int lowerbound, int mid, int upperbound, int* crosslow,
int* crosshigh, int* crosssum){
    int leftsum = array[mid];
    int rightsum = array[mid + 1];
    int maxleft = mid;
    int maxright = mid + 1;
    int i;
    int j;
    int sum = 0;
    for (i = mid; i >= lowerbound; i--){
        sum = sum + array[i];
        if (sum > leftsum){
            leftsum = sum;
            maxleft = i;
        }
    }
    sum = 0;
    for (j = mid + 1; j <= upperbound; j++){
        sum = sum + array[j];
        if (sum > rightsum){
            rightsum = sum;
            maxright = j;
        }
    }
```

```
    *crosslow = maxleft;
    *crosshigh = maxright;
    *crosssum = leftsum + rightsum;
    return;
}
```

Theoretical performance analysis

The following discussion explains why the running time for the divide and conquer algorithm is expected to be on the order Θ(nlg(n)).

We can find the recurrence of this algorithm and solve to get the run-time.
The findmaxsubarray function contains 2 recursive calls and each call divides the array in half. It also calls findmaxcrossingsubarray which runs 2n times.

Thus, we get the recurrence:

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ for all n > 1

$T(n) = \Theta(1)$ for n = 0 or n = 1

Using the Master Method,

- $T(n) = aT(\frac{n}{b}) + f(n)$
- $a = 2,\ b = 2,\ f(n) = \Theta(n)$
- $\log_b(a) = \log_2(2) = 1$
- $\Theta(n^1) = \Theta(n)$ (case 2)
- $T(n) = \Theta(n^1 \times lg(n)) = \Theta(nlg(n))$

Thus, the run-time is Θ(nlg(n)).

Algorithm 4: Linear-time

Pseudocode

A linear time algorithm can in theory provide even better running time performance with large n, than an algorithm that is O(n log(n)). The following pseudocode describes such an algorithm, which uses a temporary array to store the maximum subarray found up to each position in the array, and then reviews the elements of this subarray one by one to find the optimum solution.

Let a structure “maxsubarray” have three integer member variables, left, right, and sum.

```
Function findmax_linear(array, arraysize){
        Let s be an array of struct “maxsubarray” with “count” elements
        Let s[0].left = 0;
        Let s[0].right = 0;
```

```
        Let s[0].sum = array[0];

        For i = 1… count -1
                If s[i - 1].sum < 0
                        S[i].left = i
                        S[i].right = i
                        S[i].sum = array[i]
                Else
                        S[i].left = s[i-1].left
                        S[i].right = i
                        S[i].sum = s[i-1].sum + array[i]
        Let maxsubarray * max = &s[0]
        For i = 1… count - 1
                If s[i].sum > max.sum
                        Max = &s[i]
        Return (max.left, max.right, max.sum)}
```

C++ implementation

```
typedef struct {
        unsigned left;
        unsigned right;
        int sum;
} maxsubarray;

void findmax_linear(int array[], int count, int* lowerlim, int*upperlim, int*maxarraysum) {
   maxsubarray* s = new maxsubarray[count];

   s[0].left = 0;
   s[0].right = 0;
   s[0].sum = array[0];

   if(count == 1){
      *lowerlim = s[0].left;
      *upperlim = s[0].right = 0;;
      *maxarraysum = array[0];
      delete [] s;
      return;
   }

   for (int i = 1; i < count; i++) {
      if (s[i - 1].sum < 0) {
         s[i].left = i;
```

```
            s[i].right = i;
            s[i].sum = array[i];
            }
        else {
            maxsubarray *previous = &s[i - 1];
            s[i].left = previous->left;
            s[i].right = i;
            s[i].sum = previous->sum + array[i];
        }
    }

    maxsubarray *max = &s[0];

    for (int i = 1; i < count; i++) {
        if (max->sum < s[i].sum)
            max = &s[i];
    }

    *lowerlim = (*max).left;
    *upperlim = (*max).right;
    *maxarraysum = (*max).sum;

    delete [] s;
    return;
}
```

Testing for correctness

The four algorithms described above were all tested for correctness using sample array data provided for the assignment in the form of a text file, “MSS_TestProblems.txt.” We wrote a C++ driver function for our algorithms that handled file input and output. This “main” function opened MSS_TestProblems.txt and read each line of data, representing an array of numbers, into a buffer. The buffer string was then parsed and read into an array of integers large enough to contain the largest number of values provided in a single line in the text file, and the size of the array was entered into the last element in the array. The driver function then called each algorithm function to find the maximum subarray within the current input array, and output the resulting lower bound, upper bound, and maximum subarray sum to both standard output and to an output text file. The results of all of the algorithms were compared against each other for each input array, as a means of ensuring that the algorithms were operating correctly and were using the same definitions for upper bound, lower bound, and sum.

Experimental Analysis

In addition to designing the four algorithms for finding a maximum subarray within an array and demonstrating their effectiveness on the sample data provided, we also tested each algorithm's running time performance for handling arrays of different input sizes. For each algorithm, we compiled data for ten input arrays, of ten different sizes. The arrays were all arrays of integers between -100 and 100, and each array was randomly generated using the C++ library function rand(). We used the C++ timing library function chrono() to calculate the average time it took for each algorithm to process each array and return a maximum sum array. After compiling the data in Excel, we used Excel and MatLab to plot our data and used these programs' automatic regression analysis (curve fitting) functions to determine a function corresponding to our observed data.

For each regression analysis using Excel, the dotted line indicates a best-fit curve produced by our regression analysis, while the solid line simply shows a path connecting our experimental data.

The results of this experimental analysis are summarized for each algorithm below, followed by a chart comparing the performance of all four functions directly, and another chart comparing the performance of all four functions directly, using a log-log coordinate system to provide a clear picture of the algorithms' performance.

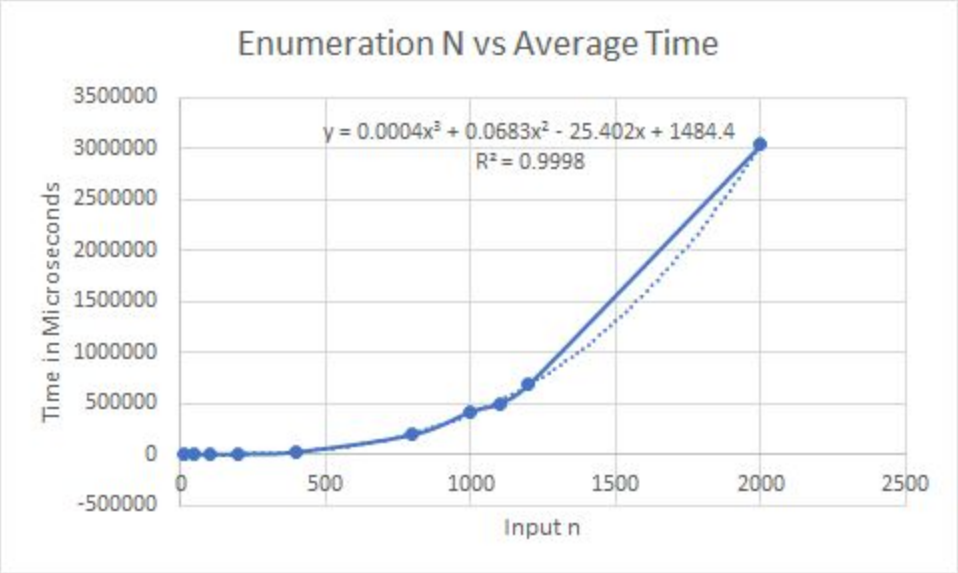
Algorithm 1: Enumeration

Experimental data, showing average run time for arrays of size n=10… n=2000

Array size (n)	Average running time (microseconds)
10	0.9
50	80.5
100	427.2
200	3553.3
400	26364.4
800	195147.7
1000	421506
1100	498370.1

1200	686914.1
2000	3039746

Results of MS Excel regression analysis



The best fit for the data was third-degree polynomial, with the regression equation being y = 0.0004x^3 + 0.0683x^2 - 25.402x + 1484.4.

Comparison to theoretical running time analysis

Let n = the size of the array

The outer loop (i) runs from 1 to n = Θ(n)
The first nested loop (j) runs from i to n = Θ(n)
And the second nested loop (k) runs from i to j = Θ(n)

The operations within the loops run in constant time = Θ(1)

This means that the run-time of the entire algorithm is $\Theta(n^3)$

This matches the fit we found for our experimental data. No discrepancies found.

Maximum “n” for given run time

Using the regression equation y = 0.0004x^3 + 0.0683x^2 - 25.402x + 1484.4 as our model, the maximum array sizes n as a function of running time limit T, for T = 30, 60, and 90 seconds are summarized in the following table.

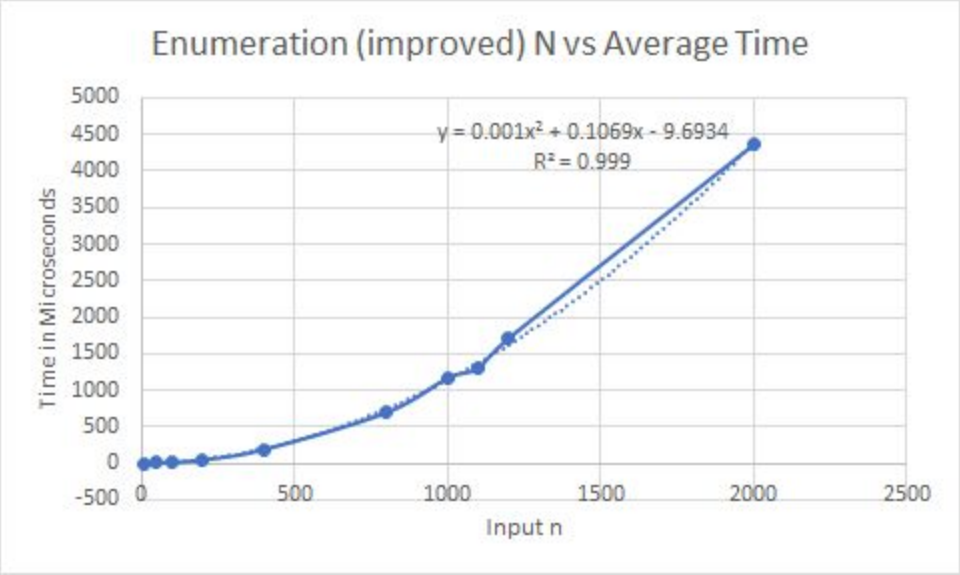
T in seconds	T in microseconds	N, max number of elements in array
10	10,000,000	2,875
30	30,000,000	4,165
60	60,000,000	5,260
90	90,000,000	6,029

Algorithm 2: Enumeration (better algorithm)

Experimental data, showing average run time for arrays of size n=10… n=2000

Array size (n)	Average running time (microseconds)
10	0.7
50	6.2
100	14.1
200	48.5
400	189.5
800	697.2
1000	1171
1100	1302.5
1200	1716
2000	4367.6

Results of MS Excel regression analysis



The best fit for the data was quadratic, with the regression equation being y = 0.001x^2 + 0.1069x - 9.6934.

Comparison to theoretical running time analysis

Let n = the size of the array
The outer loop (i) runs from 1 to n = Θ(n)
The inner loop (j) runs from i to n = Θ(n)

The operations within the loops run in constant time = Θ(1)
This means that the run-time of the entire algorithm is $\Theta(n^2)$
This matches the fit we found for our experimental data. No discrepancies found.

Maximum “n” for given run time

Using the regression equation y = 0.001x^2 + 0.1069x - 9.6934 as a model, the maximum array sizes n as a function of running time limit T, for T = 30, 60, and 90 seconds are summarized in the following table.

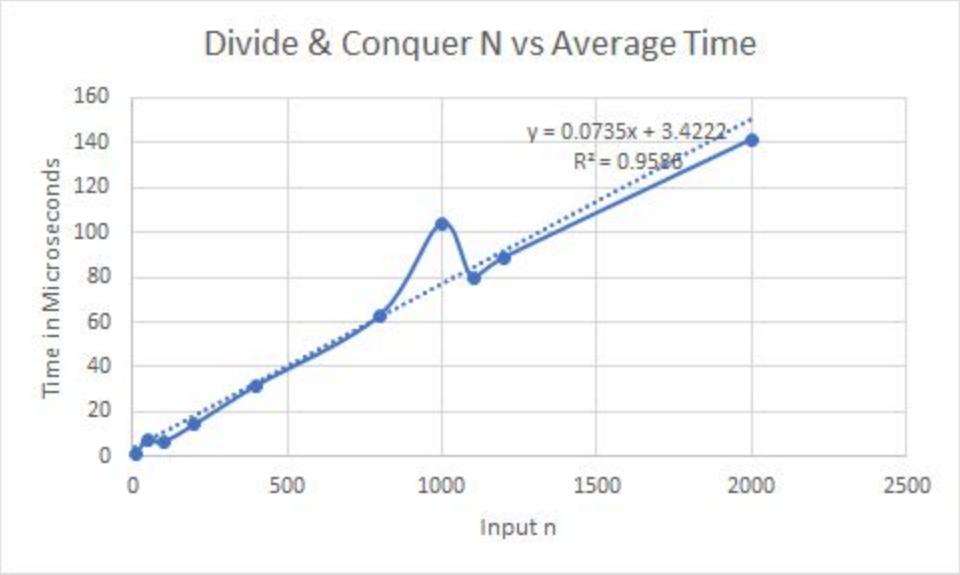
T in seconds	T in microseconds	N, max number of elements in array
10	10,000,000	99,946
30	30,000,000	173,152
60	60,000,000	244,896
90	90,000,000	299,947

Algorithm 3: Divide and Conquer (Recursive)

Experimental data, showing average run time for arrays of size n=10… n=2000

Array size (n)	Average running time (microseconds)
10	0.8
50	7.6
100	6.7
200	14.4
400	31.7
800	63
1000	104
1100	80
1200	88.8
2000	141.4

Results of MS Excel regression analysis



Somewhat surprisingly, the best fitting curve for the data was linear, with the regression equation being y = 0.0735x + 3.4222. This curve fit with an R^2 value of 0.9586.

Comparison to theoretical running time analysis

We can find the recurrence of this algorithm and solve to get the run-time.
The findmaxsubarray function contains 2 recursive calls and each call divides the array in half. It also calls findmaxcrossingsubarray which runs 2n times.

Thus, we get the recurrence:

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ for all n > 1

$T(n) = \Theta(1)$ for n = 0 or n = 1

Using the Master Method,

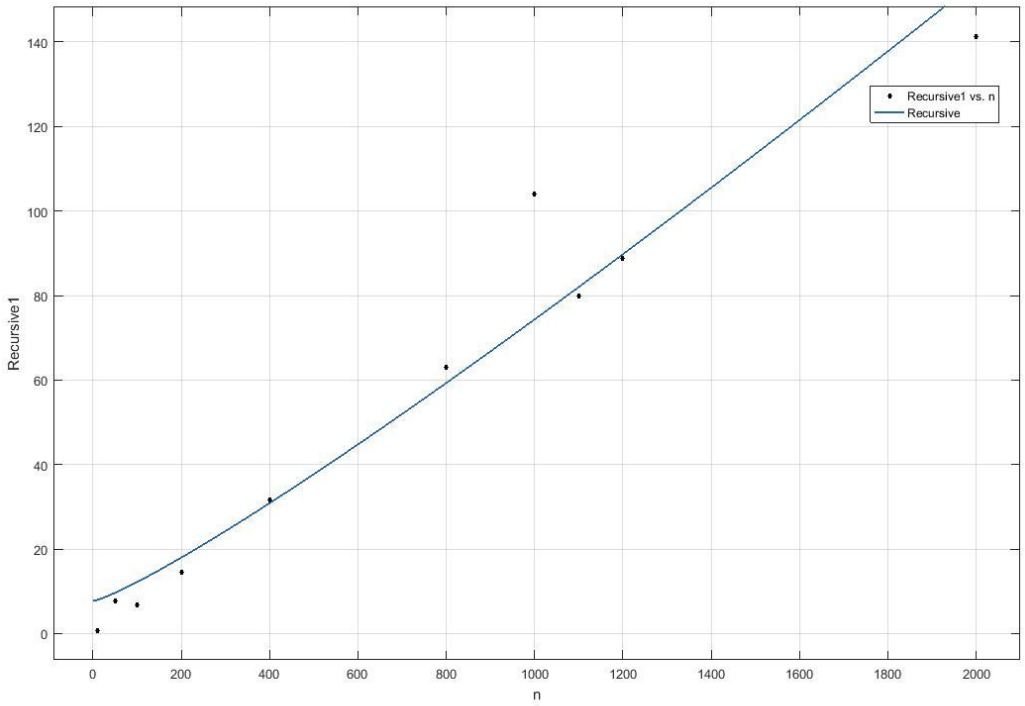
- $T(n) = aT(\frac{n}{b}) + f(n)$
- $a = 2,\ b = 2,\ f(n) = \Theta(n)$
- $\log_b(a) = \log_2(2) = 1$
- $\Theta(n^1) = \Theta(n)$ (case 2)
- $T(n) = \Theta(n^1 \times lg(n)) = \Theta(nlg(n))$

Thus, the run-time is Θ(nlg(n))

Based on our regression analysis with Excel and Matlab, our experimental running time for the divide and conquer algorithm was $\Theta(n)$. This was surprising, because it seemed to

contradict our theoretical analysis, which predicted O(nlg(n)) run time. There are several possible reasons for this discrepancy. It could either be a flaw in our code in how it implements the procedure, or a miscalculation in our theoretical assessment. Additionally, if neither of the aforementioned, it could be the way the computer compiled and processed the code.

Because of the difference between our predicted running time and our experimental results, we double checked our work by fitting an n*logn curve to the data using Matlab. Matlab generated the curve 0.006681*n*log2(n) + 7.763, which fit the data somewhat closely. However, this curve's R^2 value is 0.9456, which is slightly worse that the R^2 value of 0.9586 for our linear curve. The nlogn curve is shown below.



Maximum "n" for given run time

Using our best-fit regression equation y = 0.0735x + 3.4222 as a model, the maximum array sizes n as a function of running time limit T, for T = 30, 60, and 90 seconds are summarized in the following table. We also summarize the maximum array size with our nlogn curve 0.006681*n*log2(n) + 7.763, which did not fit our experimental data as well but was a closer match for our theoretical analysis, for comparison.

T in seconds	T in microseconds	N, max number of	N, max number of

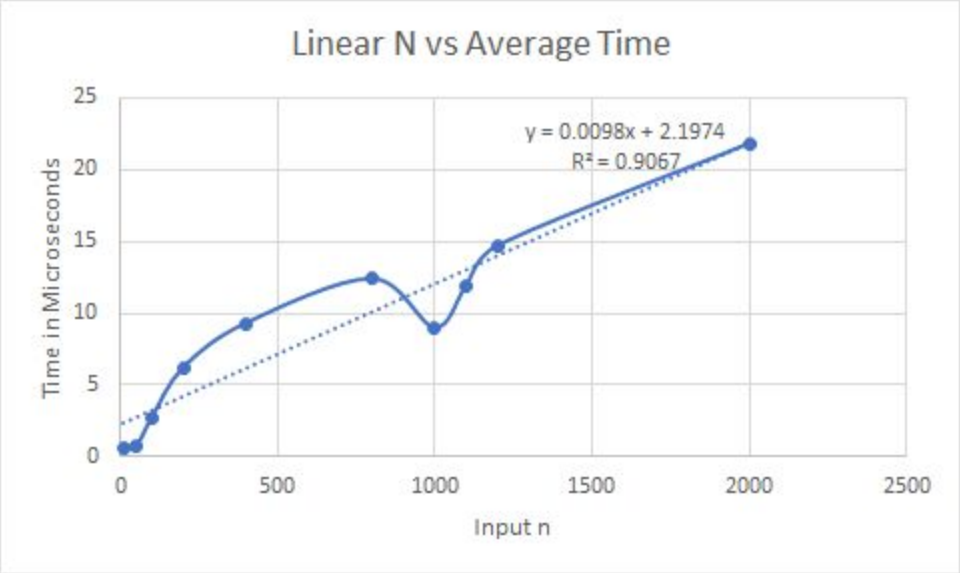
		elements in array, with linear equation	elements in array, with nlogn equation
10	10,000,000	136,054,000	58,036,200
30	30,000,000	408,163,000	164,520,000
60	60,000,000	816,326,000	317,965,000
90	90,000,000	1,224,490,000	467,727,000

Algorithm 4: Linear Algorithm

Experimental data, showing average run time for arrays of size n=10… n=2000

Array size (n)	Average running time (microseconds)
10	0.6
50	0.7
100	2.7
200	6.2
400	9.3
800	12.4
1000	9
1100	11.9
1200	14.7
2000	21.8

Results of MS Excel regression analysis



The best fit for the data was linear, with the regression equation being y = 0.0098x + 2.1974. While the data may seemingly start out as logarithmic, regression using a logarithmic model produced an r^2 value of 0.7928, while our linear regression resulted in an r^2 value of 0.9067.

Comparison to theoretical running time analysis

Let n = the size of the array
The loop (i) runs twice from 1 to n = 2n = Θ(n)

The operations within the loops run in constant time = Θ(1)
This means that the run-time of the entire algorithm is Θ(n)
This matches the fit we found for our experimental data. No discrepancies found.

Maximum “n” for given run time

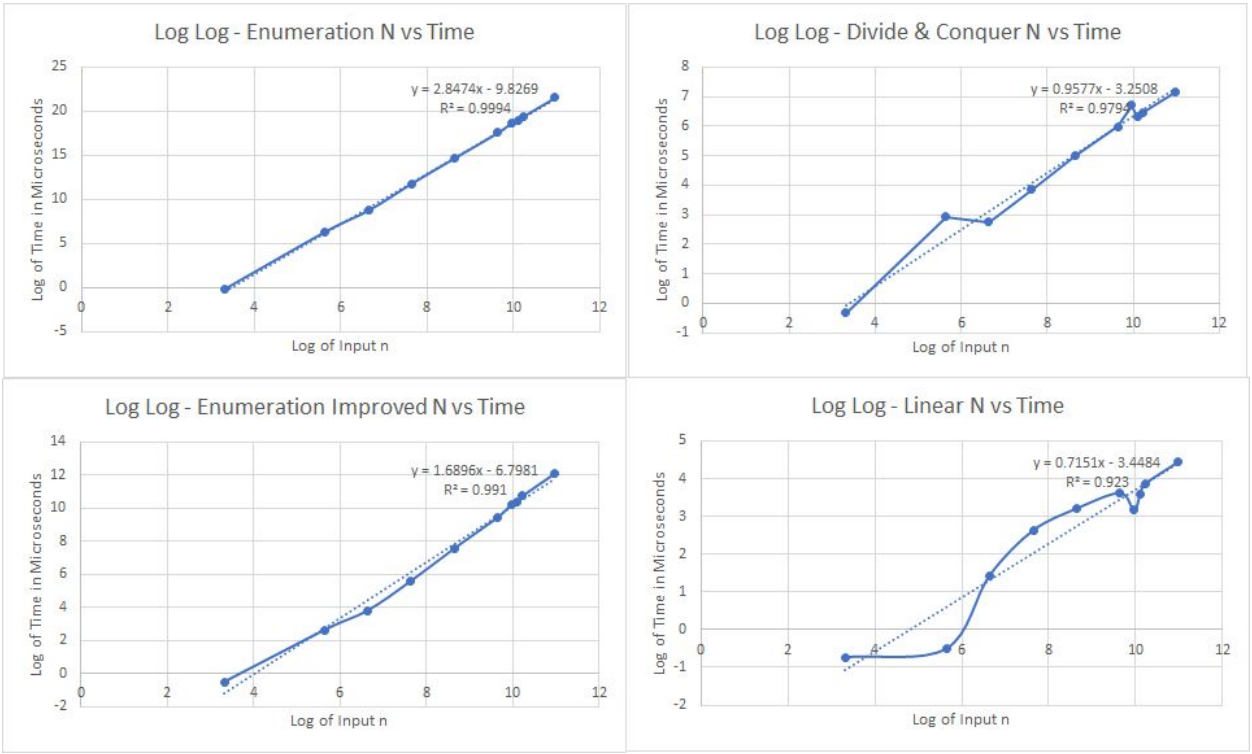
Using the regression equation y = 0.0098x + 2.1974 as a model, the maximum array sizes n as a function of running time limit T, for T = 30, 60, and 90 seconds are summarized in the following table.

T in seconds	T in microseconds	N, max number of elements

		in array
10	10,000,000	1,020,410,000
30	30,000,000	3,061,220,000
60	60,000,000	6,122,450,000
90	90,000,000	9,183,670,000

Log-Log Plots and All-in-One Graph

Individual log-log plots for each algorithm

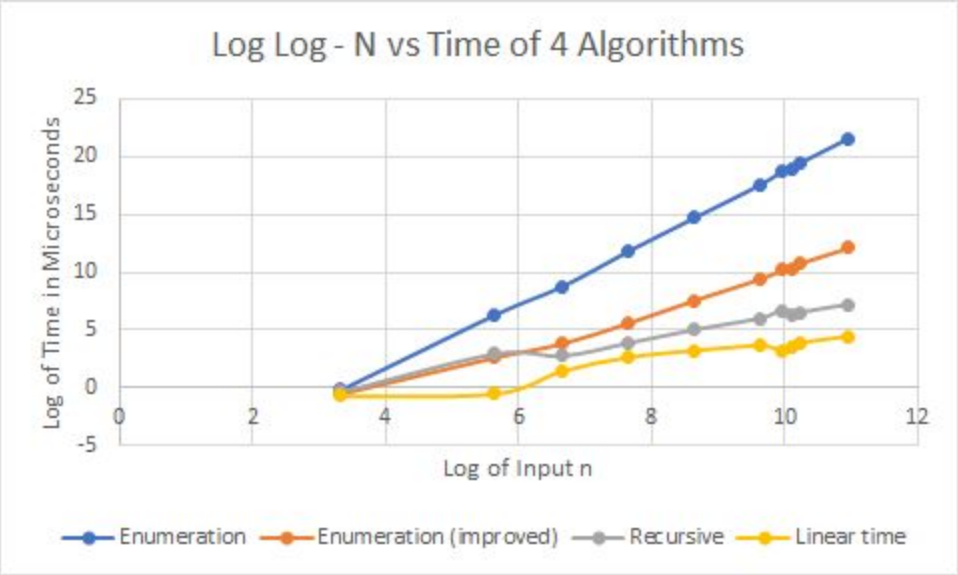


You can see in the first two log log plots that both enumeration techniques have a linear slope, indicating they are in O(x^m). The results in are in line with our analysis, in that the first algorithm is $\Theta(n^3)$ in theory and has a slope of 2.85, which is close to 3. The second, improved enumeration, is in theory $\Theta(n^2)$, which is reflected by a slope of 1.69, close to 2, in the plot of experimental data.

The log log plot for the Divide & Conquer algorithm indicates an exponential factor of 0.9577, or slightly less than 1. This is in line with our experimental finding that the data has a linear trend line as n increases, even though these experimental results are not quite in alignment with our theoretical expectations that the algorithm would be $\Theta(n \log n)$.

The log log plot of the linear algorithm is a little bit strange looking, and it is not clear that the best fitting curve would be a linear one to support our expectation that the algorithm is Θ(n). A linear curve fit to the data on the log-log plot has an x-coefficient of 0.7151, which suggests that the execution time will in fact grow less slowly than n. The R^2 value is 0.923, which reflects the fact that the log-log plot for this algorithm is somewhat scattered compared to the plots for our other three algorithms.

All-together log-log plot showing all four algorithms



The log log plot of all 4 algorithms gives a clear indication of their performance in relation to each other, which matches our theoretical data. The worst is Enumeration, then Enumeration Improved, Divide & Conquer, and finally the best at Linear, with theoretical complexities of $\Theta(n^3)$, $\Theta(n^2)$, $\Theta(n\lg(n))$, and $\Theta(n)$, respectively.