Garrett Bauer
Brendan Byers
Eddie Fox
Michael Gillett

Group 42

CS 325 Project 1: Maximum Sum Subarray

Theoretical Run-time Analysis:

Algorithm 1: Enumeration

Pseudocode:

```
Function enumeration(arr[]):
        Var sum, x,y, holder[][] (2 dimensional array)

        //Get sum of every sub-array

        For each element in arr var i:
                For each element in arr from i the arr size var j:
                        Sum := sum + arr[j]
                        Holder[i][j] := sum
                Sum := 0

        Sum  := holder[0][0]

        //Find largest sum in holder

        For each element in holder[]
                For each element in holder[][]
                        If holder[i][j] > sum
                                Sum = holder[i][j]
                                X := i
                                Y := j
        //sum holder largest sum
        //x holder beginning index of sub-array
        //y holder ending index of sub-array

        Var sub-array = values(x to y)
        Return sub-array
```

Asymptotic Runtime Analysis:

The asymptotic runtime of the Enumeration algorithm is $T(n) = O(n^2)$. This is because the algorithm has a nested loop. An inner loops proceeds through the rest of the array for each element of the outer loop. It creates an a array of the sum of every single sub array, with the index of the sums equaling the start and end indexes of the sub-array. For example, the sum at [3][6] is the sum of the numbers from indices 3 to 6 in the parent data array.

Algorithm 2: Better Enumeration

Pseudocode:

```
Function better_enumeration(arr[]):
    Var sum, x, y, best_sum

    For each element in arr var i:
        Sum = 0
        For each element in j = i to length(arr):
            Sum = sum + arr[j]

            If (sum > best_sum)
                Best_sum = sum
                X = i
                Y = j


    Var subarray = arr(x to y)

    Return subarray
```

Asymptotic Runtime Analysis:

The asymptotic runtime of the Better Enumeration algorithm is $T(n) = O(n^2)$. It uses a nested loop of 2 layers deep to go through each element of the array and compare the sum of each sub array with the best sum. This algorithm avoids recomputing the same sum many times and has one less nested for loop, so it should be better than the first algorithm by a constant factor, but still grow at the same rate because both have nested loops of 2 layers deep.

Algorithm 3: Divide and Conquer:

Pseudocode on next page:

```
Function divide(arr[], left, right):
        If right == left   //if the index is the same...
                Return (arr[left], left, right]
        Var middle := (left + right)/2    //get true middle of two indices

        //Get result of left and right halves to check the middle
        Var leftans := divider(arr[], left, middle)
        Var rightans := divider(arr[], middle, right)

        //variables to hold max of left and right
        Var leftmax, lx, ly
        Var rightmax, rx, ry

        //Initialize side maxs
        Leftmax := arr[middle]
        Rightmax := arr[middle+1]

        //Get max of left side
        Var max := 0
        For each element from middle to right in arr var i:
                max := max+ arr[i]
                If max > rightmax
                        Leftmax := max
                        lx := middle+1
                        ly := i

        //Get maximum of right side
        Var max := 0
        For each element from middle to right in arr var i:
                max := max + arr[i]
                If max > rightmax
                        Rightmax := max
                        rx := middle+1
                        Ry := i

        //setup middle maximum to compare
        Var midmax, mx, my
        Midmax := leftmax + rightmax
        Mx := lx
        My := ry

        Return max(leftans, rightans, midmax)
```

Asymptotic Runtime Analysis:

This algorithm splits the array into two halves, then finds the max of the left half, right half, and the middle array that goes across the divide between the two halves. Finding the max of the left and right is done with recursive calls, then the middle sum is found, The algorithm splits the initial array into small chunks and goes through and shifts the maximum from the union of the different chunk, eventually arriving back at the full size array with the max-sum array. It runs with a time complexity of T(n) = 2T(n/2) + Θ(n). After solving the recursion, this comes out to O(n (log n)).

Algorithm 4: Linear

Pseudocode:

```
Function linearTime(arr[])
    Var max_so_far := 0, max_ending_here := 0, temp[], max[]
    For each element in arr[]:
        Max_ending_here := max_so_far + arr[i]
        Append arr[i] to temp[]
        If max_ending_here < 0:
            Max_ending_here := 0
            Clear temp[]
        Else if max_so_far < max_ending_here:
            Max_so_far := max_ending_here
            Max[] := temp[]
    Return max[]
```

Asymptotic Runtime Analysis:

The asymptotic run-time for linearTime() is $O(n)$. The for loop goes through each element of the array $A[1, ..., n]$, performing constant-time calculations that take $O(1)$ to complete. Because it looks at each element in the array exactly once, we can say that linearTime() grows at the same rate that n does and is therefore $O(n)$.

Testing methodology of algorithms on next page.

Testing Methodology:

We split all the programs into individual programs to aid with validation. The programs took input files with lists of space-separated randomly generated values between -50 and 50. Varying ranges were also generated to make sure that result accuracy didn't depend on a minimum or maximum input size. Each algorithm was tested against the same list for agreement, as it would be very improbable for every algorithm to agree with each other if they did not produce the correct results. To further strengthen the confidence in our testing, we used the provided test site. In all cases, the results were as expected, giving us a reasonable level of confidence in the validity of our algorithms.

Experimental Analysis:

Averages of running 5 times each

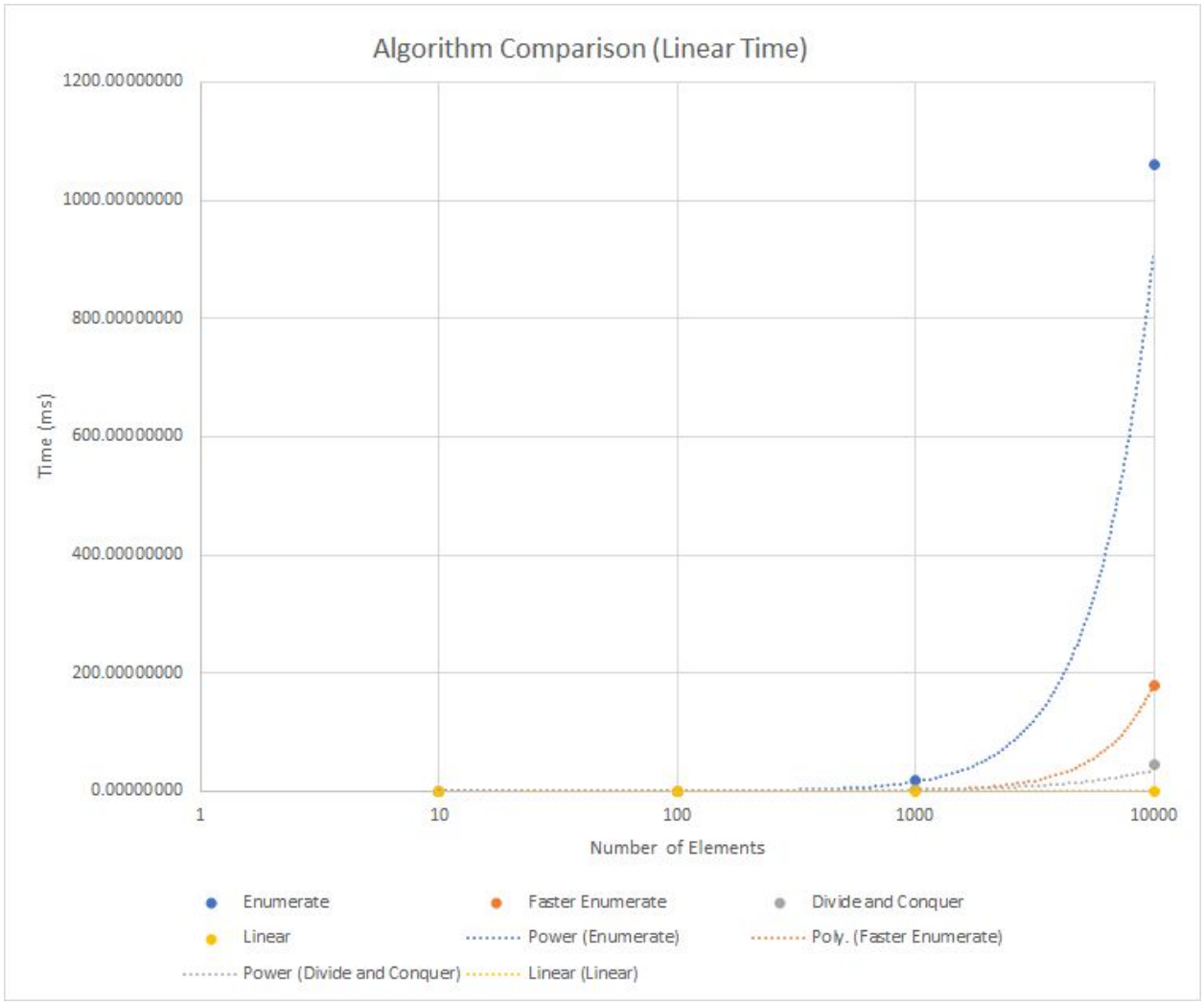
(ms)	10 Elements	100 Elements	1,000 Elements	10,000 elements
Enumerate	.0070758	.174202	14.8309	1,100.87
Faster Enumerate	.00282	0.017893	1.73541	174.827
Divide and Conquer	0.0108408	0.126345	1.63283	47.819
Linear	.002099	.005951	0.034022	.314219

These are the times we got when we ran timer.sh.
Ran on a Ubuntu VM.

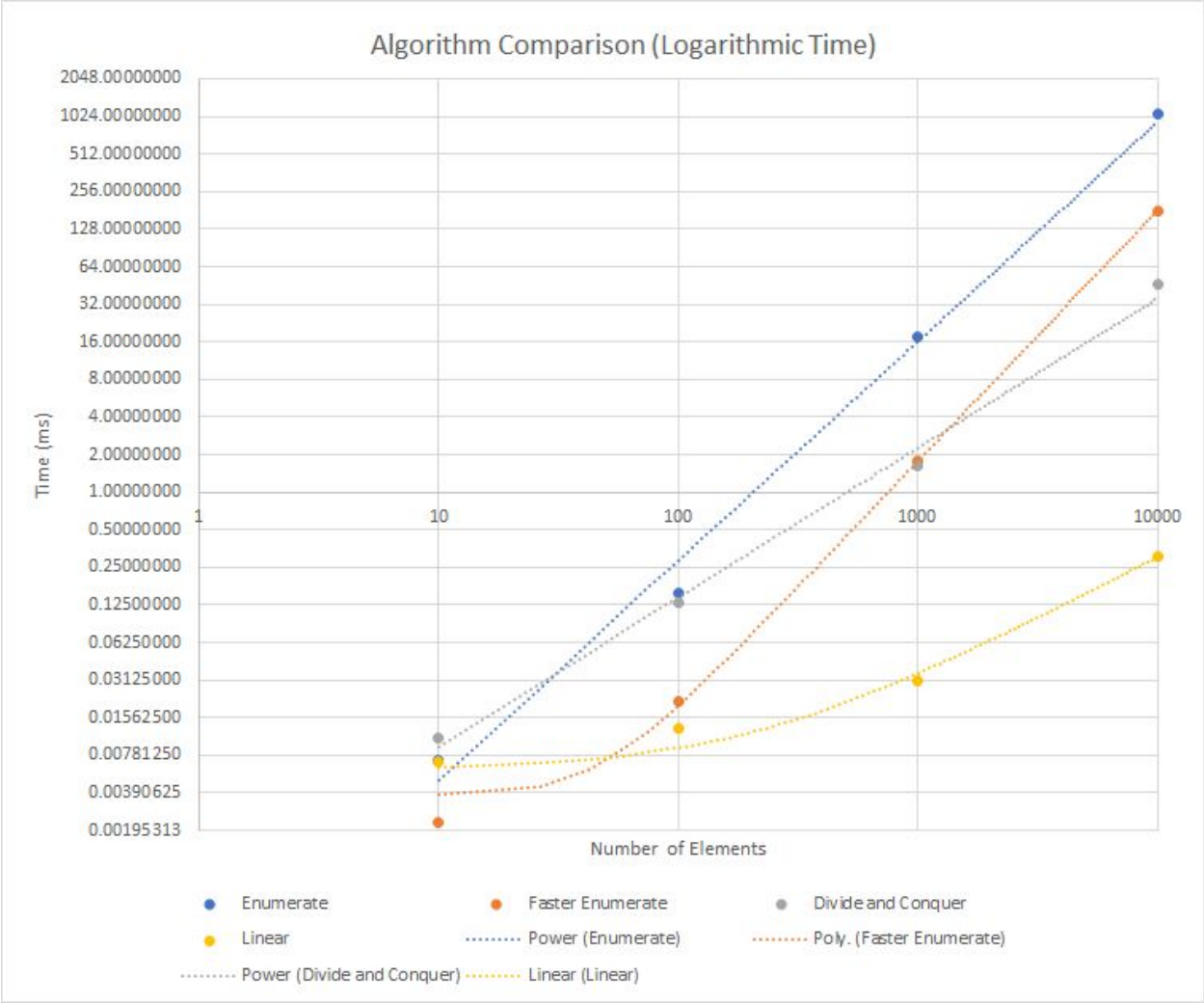
Averages of running total of 30 times:

(ms)	10 Elements	100 Elements	1,000 Elements	10,000 Elements
Enumerate	0.00715967	0.15427583	17.35250000	1061.67666667
Faster Enumerate	0.00229783	0.02097210	1.77722333	179.57516667
Divide and Conquer	0.01086660	0.12878933	1.62351167	46.09163333
Linear	0.00686133	0.01264913	0.03088607	0.30203633

-


Algorithm Comparison (Linear Time)
1200.00000000
1000.00000000
800.00000000
600.00000000
400.00000000
200.00000000
0.00000000
Time (ms)
1
10
100
1000
10000
Number of Elements
Enumerate
Faster Enumerate
Divide and Conquer
Linear
Power (Enumerate)
Poly. (Faster Enumerate)
Power (Divide and Conquer)
Linear (Linear)

Or:



Raw data used in plotting log-log graph:

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.0758e-06,0.000174202,0.0148309,1.10087,
2.282e-06,1.7893e-05,0.00173541,0.174827,
1.08408e-05,0.000126345,0.00163283,0.047819,
2.0996e-06,5.951e-06,3.4022e-05,0.000314219,

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.1076e-06,0.000142059,0.0203271,1.05364,
2.3674e-06,1.84868e-05,0.0016947,0.186013,
1.09282e-05,0.000127184,5.07401e+150,0.0452301,

2.2212e-06,5.9064e-06,2.89694e-05,0.000286987,

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.0928e-06,0.000157893,0.01542,1.06999,
2.5296e-06,1.84856e-05,0.00187183,0.175931,
1.07528e-05,0.000126162,5.07401e+150,0.0448426,
2.0286e-06,7.5934e-06,3.0781e-05,0.000300494,

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.0872e-06,0.000142281,0.0185443,1.03005,
2.177e-06,1.82198e-05,0.00181938,0.183726,
1.09346e-05,0.000125544,0.0016066,0.0452667,
2.1808e-06,4.21102e-05,2.91098e-05,0.000273322,

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.579e-06,0.000167257,0.0162531,1.0636,
2.2674e-06,1.80952e-05,0.00173419,0.18004,
1.08068e-05,0.000125292,5.07401e+150,0.0449883,
1.9966e-06,8.5274e-06,3.1014e-05,0.000283861,

ubu@ubu-VirtualBox:~/Documents/Git/CS325_Project1$ ./timer.sh
7.0156e-06,0.000141963,0.0187396,1.05191,
2.1636e-06,3.46522e-05,0.00180783,0.176914,
1.09364e-05,0.000142209,0.00162141,0.0484031,
3.06412e-05,5.8064e-06,3.14202e-05,0.000353335,

Equations Derived via Curve Regression models:

Enumeration: $0.0000105661\,x^2 + 0.00444542\,x - 0.198419$

Better Enumeration: $1.74995\times10^{-6}x^2 - 0.0000170543\,x + 0.00247591$

Divide and Conquer: 54.09x * log(x)

Linear: $0.0000295601\,x + 0.00600511$

Largest input algorithms can solve in 5 seconds, 10 seconds, and 1 minute:

Enumeration: 521 in 5 seconds, 794 in 10 seconds, 2,186 in 1 minute

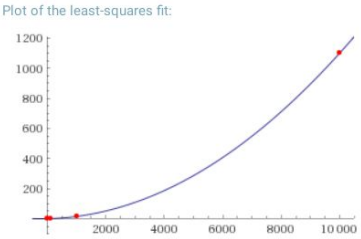
Better Enumeration: 1694 in 5 seconds, 2395 in 10 seconds, 5860 in 1 minute.

Divide and Conquer: 627 in 5 seconds. 1,796 in 10 seconds. 19,170 in 1 minute.

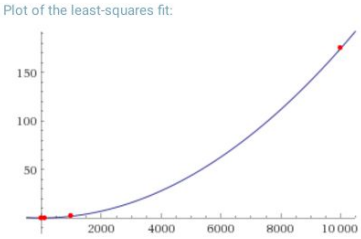
Linear: 168,943 in 5 seconds. 338,090 in 10 seconds. 2,029,559 in 1 minute.

Graphed lines of best fit over tested times:

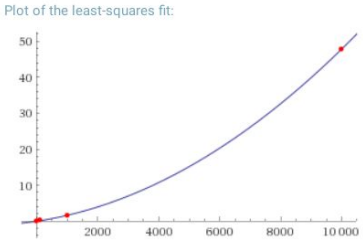
Enumeration:



Better Enumeration:



Divide and Conquer:



(Done in matlab)

https://puu.sh/vunZw/62a6034159.png
Linear:

Comparison of Theoretical and Actual Runtimes:

The actual runtimes of Enumeration and Better Enumeration are consistent with our theoretical projection. That is, they ran roughly with a curve of X^2, and followed the prediction that better enumeration would have a lower constant than enumeration and be able to process a greater amount of elements in the same timeframe. With divide and conquer, we could see how the algorithm became increasingly effective with as the time to process elements increased. This differentiated it from the previous algorithms, which were X^2's and grew at the same rate, but differed by a constant factor. With this algorithm, the number of elements processable as a function of time grew disproportionately fast, as we would expect from an algorithm that was actual O(n(log n)). Linear scaled as expected, but it was impressive to see just how big of a difference there was in the runtimes between O(n) and O(n^2) as the input size increased.