Project Report

Group 41

Anthony De La Torre

Kaiyuan Fan

Cory Melendez

Theoretical Run time Analysis

Algorithm 1 Pseduocode

Enumeration(A[0…n])

```
for i = 0  to n-1
      for j = i to n-1
            sum = 0;
            aLength = 0;
            //clear the subA array
            j++

            for k = i to j
                  subA[aLength] = a[k];
                  sum += a[k];
                  aLength++;
            k++

      if sum > maxSum
            maxSum = sum;
            maxLength = aLength;
            //copy subA array into maxSubA array
      i++
return (A[0…maxLength],sum)
```

Complexity

First for loop takes n steps, while the second does n steps as well, as well as the third

T(n) = n^3 or T(n) = O(n^3)

Algorithm 2 Pseduocode

```
BetterEnumeration(A[0…n])

SubA[]

maxSubA[]

maxSum = 0

For i = 0 to n-1

      Sum = 0

      aLength = 0

      Clear array SubA[0…n]

      For j = i to n-1

            subA[aLength] = a[j]

            sum+=a[j]

            aLength++

            if sum > maxSum

                  maxSum = sum

                  maxLength = aLength

                  Copy SubA into maxSubA

            j++

      i++

return (a[0…maxLength],sum)
```

Complexity

We only have two for loops that run n times,

T(n) = n^2 or T(n) = O(n^2)

Algorithm 3 Pseudocode

```
DivideAndConquer(A[1…n])

If n=1
```

```
      return (A[1…n],sum(A))
Lhs_subArray = DivideAndConquer(A[1…n/2])
Rhs_subArray = DivideAndConquer(A[n/2+1…n])

checkRhsArray = -inf
SumRhsArray = 0
For j = n/2 +1 to n
      sumRhsArray = sumRhsArray + A[j]
      if(sumRhsArray > checkRhsArray)
            checkRhsArray = sumRhsArray
            rhs_checkLength = j+1 – (n/2)
            rhs_checkEndIndex = j
            sumToRhsEnd = sumRhsArray
      j = j+1
if Sum(Rhs_subArray) < checkRhsArray
      Rhs_subArray = Rhs_subArray[n/2 + 1…j]
      Sum(Rhs_subArray) = checkRhsArray

checkLhsArray = -inf
SumLhsArray = 0
For i = n/2 to 1
      sumLhsArray = sumLhsArray + A[i]
      if(sumLhsArray > checkLhsArray)
            checkLhsArray = sumLhsArray
            rhs_checkLength = n/2 - i
            rhs_checkBegIndex = j
            sumToLhsEnd = sumLhsArray
      i = i+1
if Sum(Lhs_subArray) < checkLhsArray
      Lhs_subArray = Lhs_subArray[i…n/2]
```

```
    Sum(Lhs_subArray) = checkLhsArray
```

```
MaxsubArr =
max(max(Lhs_subArray[i…n/2],Rhs_subArray[n/2…j]),A[checkBegIndex…check
EndIndex])

Return (MaxsubArr[],sum(MaxSubArr))
```

Complexity

The recursive calls require 2T(n/2) complexity, while the rest requires c_1* n since we are checking to see if there is a greater sub array in the left hand side then the right hand side, so we do n calculations.

$T(n) = 2T(n/2) + c_1 n$

Apply master method:

a=2,b=2 => $n^{\log 2(2)}$=n,f(n) = n

f(n) = θ(n) therefore T(n) = θ(n*lg(n))

Algorithm 4 Pseudocode

```
linear_time(a[1…n])
    int b[] = a[];//b is a clone of a
    int max=a[0];
    for i from 0 to n
        if(b[i-1]>0)
            b[i]=b[i-1]+a[i];
    maxsum=max value in array b
return maxsum
```

Complexity

The i loop track the input array elements from 1 to n. $\sum_{i=1}^{n} O(1) = n * O(1) = O(N)$, the theoretical run-time is O(N).

Testing

For testing we tested our algorithms against the given test text file, we also used the online tool provided and checked irregular arrays and any arrays that we thought may give us trouble, such as all negatives in the beginning and at the end with a large positive number in the middle, a single positive number, and several larger arrays

Analysis

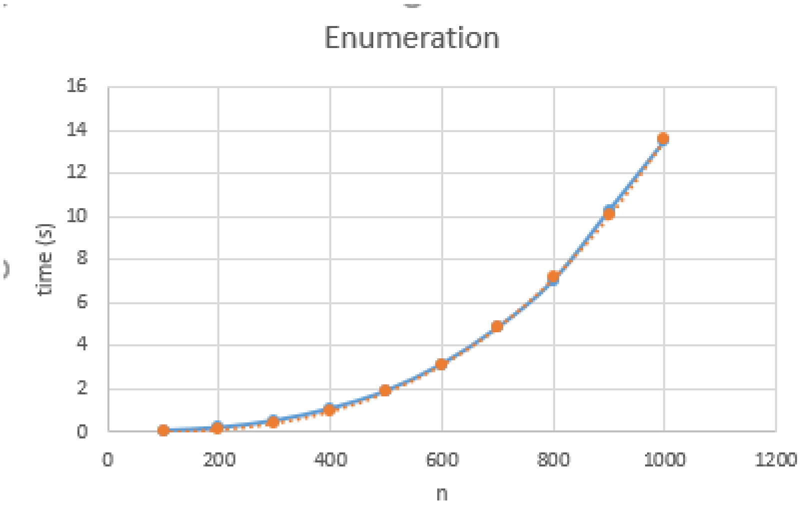
To get more accurate run time across all algorithms, we ran the following tests on a single computer:

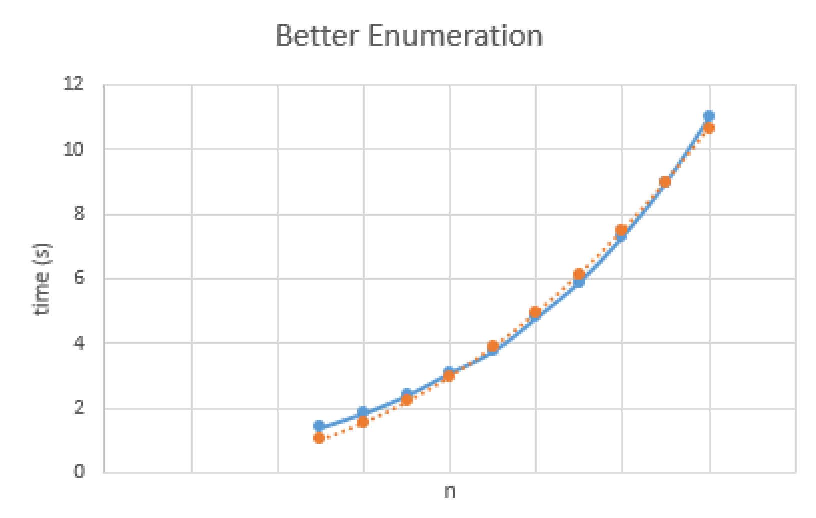
enumeration			better enumeration		
n	time	regression	n	time	regression
100	0.025	0.01823	5000	1.39279	1.0215313
200	0.1627	0.13346	6000	1.8322	1.54692121
300	0.48942	0.42765	7000	2.3817	2.19704568
400	1.04523	0.97705	8000	3.08695	2.97734223
500	1.8723	1.85456	9000	3.74957	3.89270864
600	3.1159	3.13076	10000	4.78972	4.94761557
700	4.81477	4.8744	11000	5.89917	6.14618599
800	7.00556	7.15277	12000	7.30852	7.4922535
900	10.2581	10.0319	13000	8.97589	8.98940645
1000	13.5017	13.5769	14000	10.9915	10.6410224
(3.287*10^-8)*(x^2.872)			(3.894*10^-9)*(x^2.276)		
should be x^3			should be x^2		

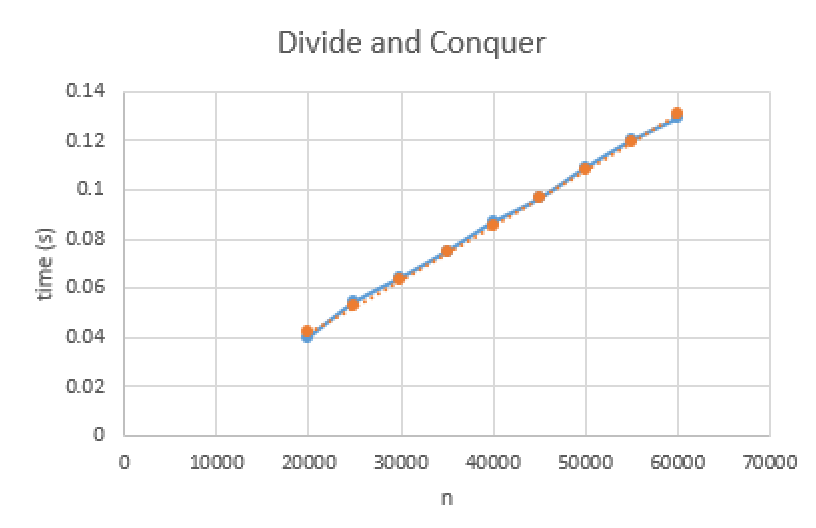
divide and conquer			linear-time		
n	time	regression	n	time	regression
20000	0.03994	0.04224	20000	0.00283	0.0030798
25000	0.05429	0.05277	25000	0.00377	0.0038068
30000	0.06421	0.06349	30000	0.00464	0.0045338
35000	0.07497	0.07437	35000	0.0054	0.0052608
40000	0.08687	0.08539	40000	0.00615	0.0059878
45000	0.09646	0.09652	45000	0.00683	0.0067148
50000	0.10901	0.10777	50000	0.00746	0.0074418
55000	0.12026	0.11911	55000	0.00809	0.0081688
60000	0.12915	0.13053	60000	0.00873	0.0088958
(4.4*10^-7)*x*(LOG(x))+0.004391			(1.454*10^-7)*x+0.0001718		
Rsquared = 0.99 excellent			Rsquared = 0.99 excellent		

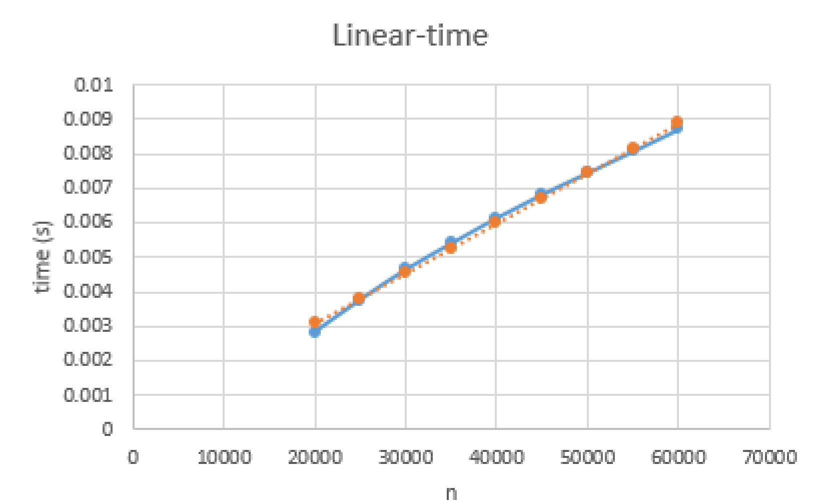
Times(s)	Enumeration	Better_enumeration	Divide and conquer	Linear_time
10	8.99*10^2	1.36*10^4	1.59*10^6	6.88*10^7
30	1.32*10^3	2.21*10^4	4.45*10^6	2.06*10^8
60	1.68*10^3	2.99*10^4	8.54*10^6	4.12*10^8

The functions at the bottom are the functions that were the best fit, as you can see we got very close on every algorithm. Also, you can see that we had to use widely different sized algorithms, we had to do this since some of the algorithms would require much less work than the others. The following are the graphs of the above data set with the orange as the best fit:


Enumeration
time (s)
n
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200


Better Enumeration
time (s)
n
0
2
4
6
8
10
12


Divide and Conquer
time (s)
n
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0
10000
20000
30000
40000
50000
60000
70000



Since we had widely different array sizes, we did a log-log plot of all four on the same graph