Group Assignment #3

Kyle Nichols, Justin Pham, Alex Bailey

**Introduction**

This problem had us look at designing three different methods of using divide and conquer to find the sub array whose sum is closest to 0. We ultimately used our design of the second method, as it would on average perform better than n2 and because the sorting for our design of method 3 required a little bit more overhead.

**Pseudo code**

Method1(int arr1, int arr2, int n1, int n2){

int smallestD = infinity;

for i = 0 to n1

for j = 0 to n2

temp = abs( arr1[i] + arr2[j] )

if(temp < smallestD){

smallestD = temp;

store location of the number in the first list

store location of the number in the 2nd list

}

end

end

return location of the numbers and the smallestD to 0

}  
  
Method2(A[n], B[n])  
Sort(A);  
Sort(B);  
min=65535  
prev=65535  
for i=0 to n-1  
 for j=0 to n-1  
 sum=abs(A[i] + B[j])  
 if sum > prev // The sum will now be going up because it they are sorted  
 break  
 else  
 if sum < min  
 min=sum  
 iFinal=i  
 jFinal=j  
return min

Times every number in the A2 by -1  
Append the A2 to the end of the A1, making A3  
Create A4 array the same length as the A3 array  
Put 0s in the first half of A4 and 1s in the second  
Sort the A3 such that every time a switch is made, the A4 is switched as well  
For element 0 to n-1 of A4, i  
 If i != i+1  
 If abs( abs(A3[i]) – abs(A3[i+1]) )   
 best = abs( abs(A3[i]) – abs(A3[i+1]) )  
return best  
  
DivAndCon(int arr[], int n){

int nHalf = n/2;

int firstHalf, secondHalf

int arr1, arr2

int count = nHalf - 1;

//suffix array, calculate sum for the first half of the array

for i = nHalf to 0

firstHalf[i] = arr[i]

if(i == count)

arr1[i] = arr[count ]

else

arr1[i] = arr1[i] + arr[i+1]

end

//prefix array, calculate sum for the second half of the array

count = 0

for i = nHalf to n

secondHalf[i] = arr[i]

if( i == nHalf)

arr2[i] = arr[i]

else

arr2[i] = arr2[i-1] + arr[i]

end

call method1(arr1, arr2, size1, size2) or method2(arr1, arr2, size1, size2) or method3(arr1, arr2, size1, size2)

}

**Recurrence**

*Method1*

T(n) = 2T(n/2) + n2

= T(n/4) + 2(n/2)2 + n2

...

= n2 + n2/2 + … + n2/2k

This makes this a runtime of n2.

*Method2*

T(n) = 2T(n/2) + nlogn [quicksort] + n2 [worst case]

= T(n/4) + 2(n/2)log(n/2) + nlogn + 2(n/2)2 + n2

= n2 + n2/2+ … + n2/2k + nlogn + nlog(n/2) + … + nlog(n/2k)

Note: This is only the worst case. The average case will not run the full loops each time and will perform half as quickly. Therefore it is only upper bounded by n2.

*Method3*

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

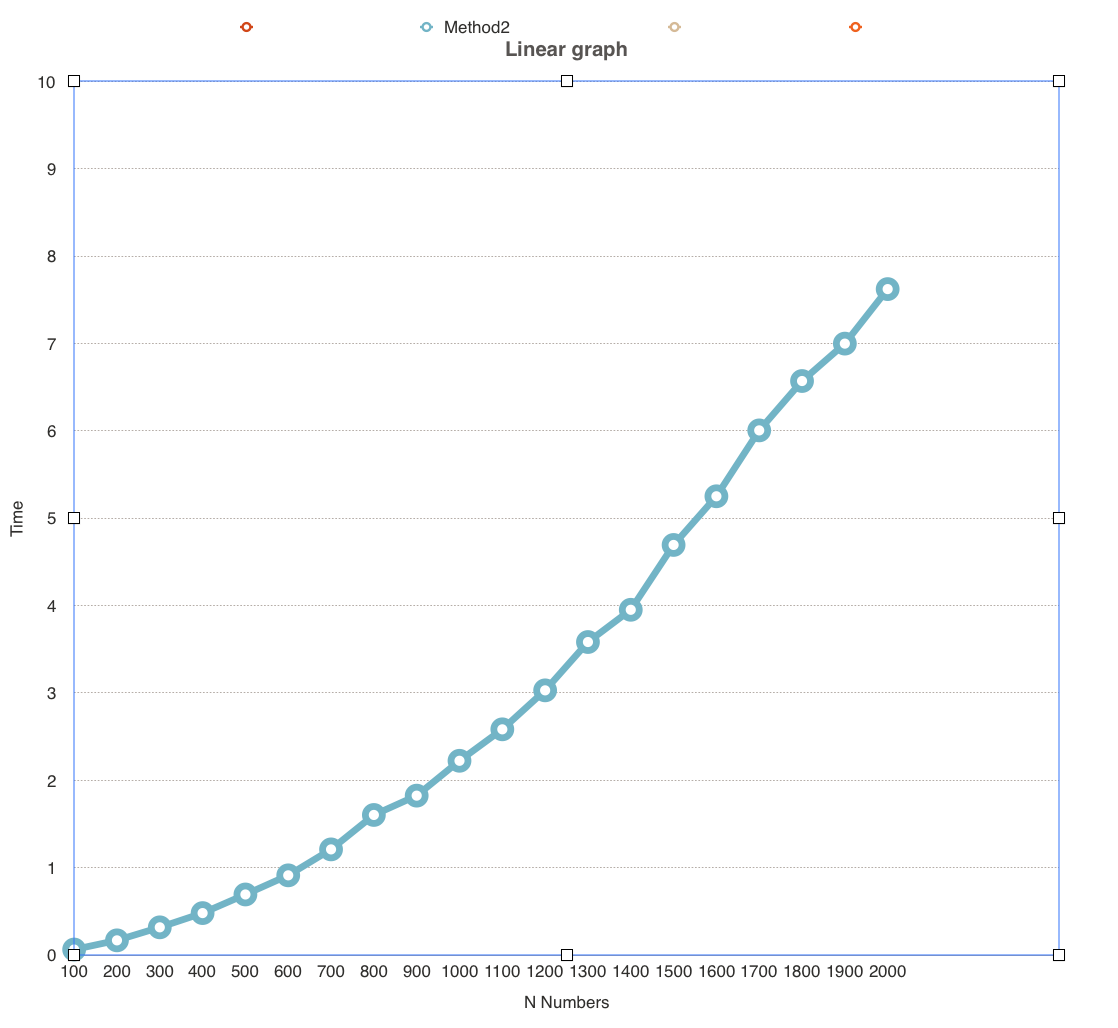
= 2T(n/4) + 3n

…

= (2k-1)n

This makes this a runtime of n, on its own, though the sort used will overtake it (if the sort used is quicksort, for example, it will be nlogn).

**Plot of Runtime**

**