Group Assignment 3

Kevin Stine

Introduction:

The problem we have been tasked to solve is the closest to zero problem. Given an array of both positive and negative values, we are to find an algorithm which is able to find a subarray of that initial array which has a sum of values closest to zero. We have also been tasked to use a divide and conquer approach to solving this algorithm.

Prefix-Suffix Subroutine, Method 1:

```
methodOne(suffix[0,..., n], prefix[0,..., m])
              sumOfSuffix(suffix)
              sumOfPrefix(prefix)
              bestSuffix = n
              bestPrefix = 0
              for (int i = 0; i < n; i++) {
                     for (int j = 0; j < m; j++) {
                            best = abs(suffix[bestSuffix] + prefix[bestPrefix])
                            curr = abs(suffix[i] + prefix[j])
                            if curr < best
                                   bestPrefix = i
                                   bestSuffix = j
                     }
              }
              return bestSuffix, bestPrefix
```

The runtime for this algorithm should be $\Theta(n^2)$ because it has to iterate through the full size of both the suffix and the prefix. It will compare against each element in the array.

Prefix-Suffix Subroutine, Method 2:

```
methodTwo(suffix[0,...,n], prefix[0,...,m])
              sumOfSuffix(suffix)
              sumOfPrefix(prefix)
              bestSuffix = n
              bestPrefix = 0
              sortedSuffix = sort(suffix)
              sortedPrefix = sort(prefix)
              minSum = INT_MAX
              for (int i = 0; i < n; i++) {
                     for (int j = 0; j < m; j++) {
                            sum = sortedSuffix[i] + sortedPrefix[j]
                            if (abs(sum) < abs(minSum)) {
                                   minSum = sum
                                   minSuffix = i
                                   minPrefix = j
                            }
                     }
              }
```

The runtime for this algorithm should be $\Theta(n^2)$ because it has to iterate through the full size of both the suffix and the prefix. It will compare against each element in the array.

Prefix-Suffix Subroutine, Method 3:

```
methodThree(testArray)
                sumOfSuffix(suffix)
               sumOfPrefix(prefix)
               bestSuffix = n
               bestPrefix = 0
               for (int i = 0; i < m; i++) {
                       prefix[i] = -abs(prefix[i])
               }
               list = suffix + prefix
               return list
```

The runtime for this algorithm should $\Theta(n)$ because it combines the suffix and prefix so it only needs to loop through the entire array once.

Divide and Conquer Algorithm:

Pseudocode:

```
def compute(suffix, prefix):
       methodThree(suffix, prefix)
       for i in range(0, len(combinedArray)):
    for j in range (i+1, len(combinedArray)):
        tempSum = arrSum
        initVal = combinedArray[j]
        secondVal = combinedArray[i]

        if secondVal < 0:
            if initVal > 0:
               arrSum = abs(initVal) - (secondVal)
            else:
                arrSum = abs(initVal) - abs(secondVal)
        else:
            arrSum = abs(initVal) - abs(secondVal)
        if  abs(arrSum) < minSum:
            minSum = abs(arrSum)
            start = i
            startVal = secondVal
            finish = j
            finishVal = initVal
        else:
            arrSum = tempSum
        if smallestVal < minSum:
            minSum = smallestVal
```

Recurrence Relations:

Algorithm 1 Recurrence Relation:

Since this algorithm divides the initial array in half, and has to iterate through each full half from 0 to n, we have the following recurrence relation:

T(n) = 2(n/2) + n where a = 2, b = 2, f(n) = n and c = 1
logbase(2)2 = 1 = c
T(n) = big theta (nlogn)

Algorithm 2 Recurrence Relation:

Since this algorithm divides the initial array in half, and has to iterate through each full half from 0 to n, we have the following recurrence relation:

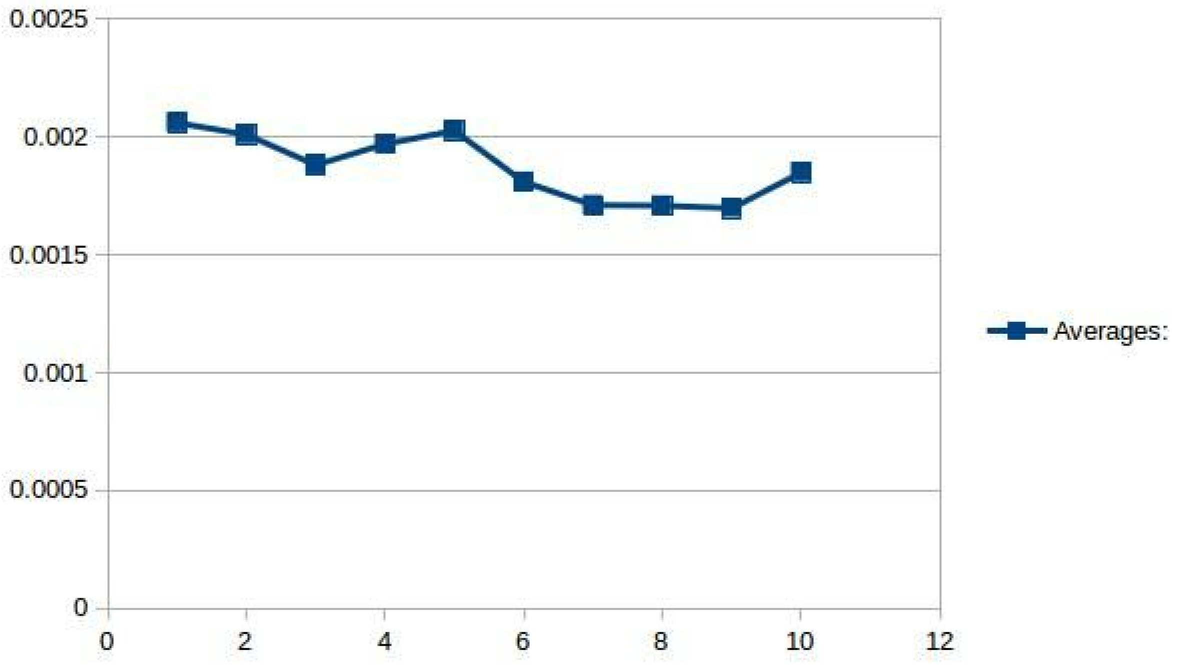
T(n) = 2(n/2) + n where a = 2, b = 2, f(n) = n and c = 1
logbase(2)2 = 1 = c
T(n) = big theta (nlogn)

Algorithm 3 Recurrence Relation:

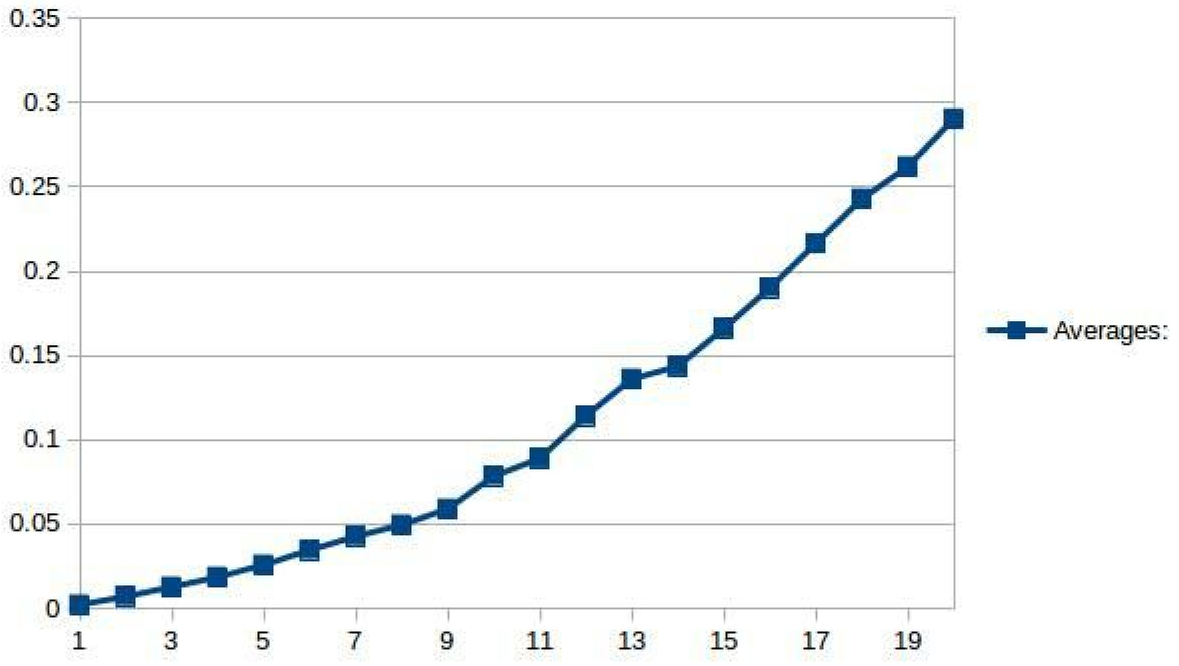
Since this algorithm divides the initial array in half, and has to iterate through each full half from 0 to n, we have the following recurrence relation:

T(n) = 2(n/2) + n where a = 2, b = 2, f(n) = n and c = 1
logbase(2)2 = 1 = c
T(n) = big theta (nlogn)

Programming Implementation:



Averages from run time of test_cases_with_solutions



Averages from run time of test_cases_without_solutions