Anthony Miller (milleant)
Taylor Fahlman (fahlmant)
Canvas Group #3

Group Assignment #3

Introduction

The problem is asking for the indices from a given array that signify the left and right bounds on the subarray whose sum is closest to 0. A divide and conquer algorithm is a great option for this because we know that the indexes will either be in the left half, right half, or one in each.

Method 1

```
min_num = infinity
for x in range len(array1)
  for y in range len(array2)
    min_num = min(abs(array1[x] + array2[y]), abs(min_num))
return min_num
```

Since we are looping over every element exactly twice we have n^2 iterations. There are two constant time operations of adding two array elements and finding the minimum between that sum and the previous minimum sum. The concluding function would be $f(n) = n^2 + 2$ or $\theta(n^2)$.

Method 2

```
sort(array1)    // $O(n \log n)$
sort(array2)    // $O(m \log m)$
len1 = len(array1)
len2 = len(array2)
best = (0, 0)
i = 0
k = len2
sum = infinity




//Next page
```

```
while (i < len1 and r >= 0)
    if abs(array1[i] - array2[k] < sum)
        best = (i, k)
        sum = abs(array1[i] - array2[r])

    if array1[i] - array2[k] > 0
        k = k - 1
    else
        i = i + 1

return best
```

We first need to sort the 2 arrays within the method. The asymptotic runtime is based of off Python's implementation of the *.sort()* function which runs in $O(n \log n)$. Since we have 2 arrays with different sizes this adds $2O(n \log n)$ to the total runtime. The main while loop will stop when either the 1st index is greater than n or the 2nd index is less than 0. Theoretically the correct answer could be the last element of the prefix and the first element in the suffix. To find that we would have to iterate m + n times. The final function would be: $f(n) = 2(n \log n) + 2n + 2$ or $O(n \log n + n)$

Method 3

```
array1 = (array1, 1) for each element in array1
negarray = -(array2)
negarray = (negarray, 2) for each element in negarray
biglist = array1 + negarray
sort(biglist by x value)
min_num = infinity
for x in range len(biglist)
  if(biglist[x][0] != biglist[x+1][0])
    min_num = min(abs(min_num), abs(biglist[x] - biglist[x+1])

return min_num
```

We first have to sort the 2 summed arrays. After sorting, we take the first half and put it into another array, but with a 1 associated with it. We then take the negative of the second array, associate a 2 with it, and combine the lists and sort it. Then we look at each element, and if that element and the next are each from a different list, we find the difference. If the distance is lower than the current minimum value, we replace it. Given this, our function would be:
$f(n) = n \log n + 2n + 2$ or $O($n log n + n)

Divide and Conquer

```
method = 3

divide_conquer(array):
       if length(array) < 2
              return array[0]
       left_sums = prefix_sum(array[:½])
       right_sums = suffix_sum(array[½:])
       i = dc(array[:½])
       j = dc(array[½:])
       k = method(left_sums, right_sums)

       return min(i, j, k)
```

Recursive Relations

Method 1

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

Solution: T(n) = 2^kT(n/2^k) + n/2^k-1 + n^2 + 2

k=log[2] n

T(n) = nT(1) + 1 + 2 = n^2 + 3

O(n^2)

Method 2

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

Solution:

T(n) = 2^kT(n/2^k) + 2^kn log n + 2^kn + 2^k

k = log[2]n

T(n) = nT(1) + n logn + n^2 + n

O(n^2)

Method 3

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

Solution:

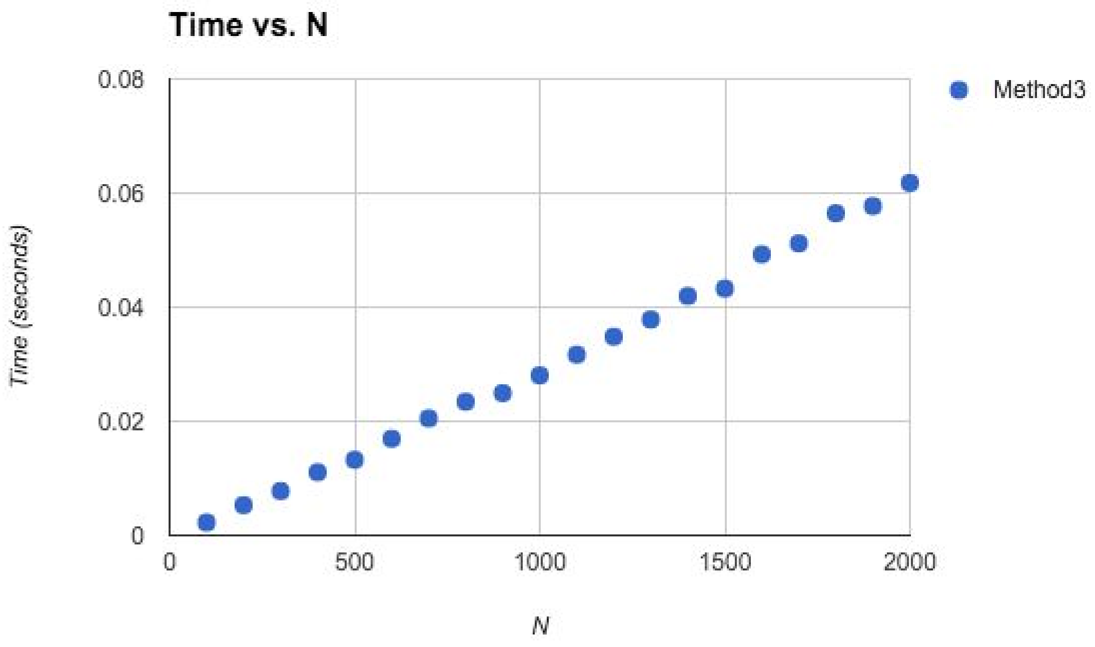
T(n) = 2^kT(n/2^k) + 2^(k-1)n*logn

k = log[2]n

T(n) = nT(1) + n^2 * logn

O(n^2 * log(n))

Run Time vs. Input Size


Time vs. N
Time (seconds)
N
● Method3