**605.620.82: Algorithms for Bioinformatics**

**Thomas Muamar**

**Josie Lu**

**Libby Lewis**

**Shadi Mohammadabadi**

**Collaborative Assignment**

**Due Date: September 13, 2021**

Merge(L,R,A){

nL = length(L)

nR = length(r)

i=j=k=0 #starting index, i for L, j for R, k for A

while i < nL && j < nr

{

if L[ i ] <= R[ j ]

A[k] = L[i]

i = i + 1

else

A[k] = R[j]

j = j + 1

k = k + 1

}

while ( i < nL)

{

A[k] = L[i]

i = i +1

k = k + 1

}

while ( k < nL)

{

A[k] = R[j]

j = j +1

k = k + 1

}

MergeSort(A){

n = length(A)

if (n < 2)

return

mid = n/2

left = arraysize(mid)

right = arraysize(n-mid)

for i=0 to (mid-1)

left[i] = A[i]

for i=mid to (n-1)

right[i-mid] = A[i]

MergeSort(left)

MergeSort(right)

Merge(left,right, A)

#return left, right}

Split(A){

mid = n/2

left = arraysize(mid)

right = arraysize(n-mid)

for i=0 to (mid-1)

left[i] = A[i]

for i=mid to (n-1)

right[i-mid] = A[i]

return left, right

}

main

// start by sorting the array by increasing order

int arr[] = {x1,x2,x3,…xn}

merge(arr)

//once sorted separate into two array subsets of n/2

Int[] a = [n+1/2]

Int [] b = [n - a.length]

For int I = 0; i<n; i++

If I < a.length

A[i] = arr[i]

Otherwise

B[i-a.length] = arr[i]

//the two separated arrays a and b will then calculate sum and subtract both to produce difference of sum

For I = 0 I <a.length i++

Sum = sum+ a[i]

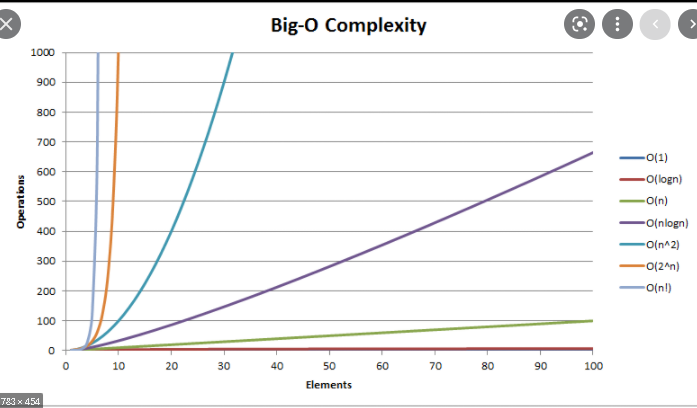
For I = 0 I <b.length I++

Sum2 = sum2+b[i]

Difference = sum2-sum

We as a group decided to go for merge sort because it provided o(nlogn) for both best- and worst-case scenario. This is important because when sorting the array, the values are most likely going to be a random set of values. The reason we are choosing to use a sort to allow the array to be in ascending order before we separate them in to two subsets and calculate the difference. Sorting it this way will allow small values on one side of the array and the second will have all the high values producing the maximized difference of the subsets. Since were always assuming that the array will be the same size this provides the most efficient solution in producing the sums of the two subsets being maximized.

In terms of space complexity here the merge sort does have a problem but as we have learned that we can easily buy memory.



Looking at this graph gives us the reason for are choice. Even though insertion sort might have o(n) for best case it struggles on the average and worst-case scenarios where it goes to O(n^2). Having o(nlogn) allows for more stabilized choice when delaying with a data set that can be randomized.

B)

**Josie's thought:**

I do not know how to approach this question but since the subset sizes do not need to be equal, I believe the cost can vary depending on how we approach this. I think the cost will be more than the cost of part a. Since the algorithm has to check if the differences between the two subset are as minimal as possible. The algorithm must check the differences of the sums for each subset it can create. I think if we can make the subsets the same size, it would be much easier to approach this algorithm, but that is not the case.

**Thomas' thoughts:**

The cost of this version would be a lot worse than the first problem. The reason for this is because you would need to have the algorithm be able to test each of the different sorted subsets and then compare which would give the minimum value.

**Libby's thoughts:**

My initial reaction to the minor change is that the cost would be much greater. Since the subsets do not have to be the same size, the algorithm would have to check every combination of sizes of subsets and every combination of sorting the array contents into each combination of subset. This must be done in order to find which configuration produces the minimum difference between the sums of the subsets. Whereas the initial algorithm just had to sort the array contents in place at O(nlogn), the new algorithm would have to try multiple sorting options into each of many options of sized subsets. The cost, therefore, would be much greater.

C)

Started off by thinking doing an indexing every other number would work but wasn’t producing the minimum value on different set of data. The best way I could think of when approaching this problem is through a recursive solution where it checks for the most optimal solution for the minimum values. Then using the math.min function in java that takes the values of two numbers.

static int minimumValue(int arr[], int i, int sub1, int sub2)

int diff = 0;

int first = 0;

int second = 0;

//base case of to check if at end of array

while(i == arr.length)

diff = Math.abs(sub1 - sub2);

return diff;

while(i < arr.length)

// when there are values to choose from it will take that value and add it to the first subset and then it will also choose to add it to the second subset.

// includes the element

first = minimumValue(arr, i+1, sub1+arr[i], sub2);

second = minimumValue(arr, i+1, sub1, sub2+arr[i]);

return Math.min(first, second);

// starts the the values at 0

return minimumValue(arr, 0, 0, 0);

The time complexity in this problem is definitely a lot worse than the first problem. The reason for that is we have two partitions here subset 1 and subset 2. When making a choice for the first index it will either choose between group 1 and group2. This for every index within the array. So the cost of this algorithm would O(2^n) because each time its making a choice for a number it is 2\*2\*2….\*2 and so on. The space complexity for this algorithm would be O(n) because we wont be adding any more values to the array so once it reaches the base case of the length of the array it will end.

Applications to bioinformatics:

This can apply to bioinformatics because sorting can end up making it easier for sequence alignments.