Knuth Group
Ryan Fitch
Karen Chapman
Steven Dea
Nadja Dimartino

605.620.81.SP20 Algorithms for Bioinformatics
Group Project 1

A)

Our algorithm capitalized on the constraint of the two returned arrays having to be equal in size. Due to this, we can infer that the maximum difference must come from one array containing values that are all larger than the other array. Rather than infer and impute relative size of values as they are encountered, we chose to first sort the array in ascending order and proceed with populating the new arrays based on expected order. In this scenario we should expect each value encountered up to the midpoint of our sorted array to be smaller than the latter half.

Since we are proceeding with a single and simple iterative loop over the length of our sorted array, the processing time should follow an O(n) behavior. However, our algorithm relies on the array being sorted first, and our use of Merge Sort has a limiting behavior of θ(nlogn). This would put the overall limiting behavior and efficiency of our algorithm on the performance of our sorting algorithm. We chose Merge Sort in this case as its performance is always nlogn, regardless of the initial configuration of the array. Merge sort is not an in place sorting algorithm, so our space efficiency leaves much to be desired as the remainder of our algorithm also increases space/memory usage by n as well. However, our algorithm is very time efficient and will always perform at a predictable level of efficiency.

`Maximum_Difference_Equal_Length(Array_A):`

```
// Returned Array_A sorted in ascending order
sorted_A = Mergeort(Array_A)                                    // (nlogn)
subarray_length = Array_A.length/2                              // (c)
// Initialize 2 new arrays:
A1[] and A2[]                                                   // (c)
For i = 0 until i = sorted_A.length:                            // (n)
     // populate A1 with the lowest half of our sorted_A values
     if i < subarray_length:                                    // (c)
          Add sorted_A[i] to A1[]                               // (c)
     // populate A2 with the remaining (larger) values from sorted_A
     else:
          Add sorted_A[i] to A2[]                               // (c)
     i = i + 1                                                  // (c)
Return A1 and A2 arrays
```

Expected Input/Output example sets:

Input = [1, 5, 2, 4, 3]
Output:
Array 1 = [1, 2]
Array 2 = [3, 4, 5]

```
Array at start = [1, 5, 2, 4, 3]
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [1, 2, 3, 4, 5]
// Step: 0
Array 1 = []
Array 2 = []
// Step: 1
Array 1 = [1]
Array 2 = []
// Step: 2
Array 1 = [1, 2]
Array 2 = []
// Step: 3
Array 1 = [1, 2]
Array 2 = [3]
// Step: 4
Array 1 = [1, 2]
Array 2 = [3, 4]
// Step: 5
Array 1 = [1, 2]
Array 2 = [3, 4, 5]
```

Input = [10000, 1]
Output:
Array 1 = [1]
Array 2 = [10000]

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [1, 10000]
// Step: 0
Array 1 = []
Array 2 = []
// Step: 1
Array 1 = [1]
Array 2 = []
```

```
// Step: 2
Array 1 = [1]
Array 2 = [10000]
```

Input = [1, 1, 1]
Output:
Array 1 = [1]
Array 2 = [1, 1]

```
Array at start = [1, 1, 1]
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [1, 1, 1]
// Step: 0
Array 1 = []
Array 2 = []
// Step: 1
Array 1 = [1]
Array 2 = []
// Step: 2
Array 1 = [1]
Array 2 = [1]
// Step: 3
Array 1 = [1]
Array 2 = [1, 1]
```

B & C)

Our minimum difference array also relies upon us first sorting our initial array, however we have it returned in descending order. This ensures that should our data be populated with extreme outliers, they will be handled appropriately first. The notion being that large outliers are going to have a much larger impact on a differential than small outliers will; (5x10)-20 is a bigger differential than (5x10)-1.

Our algorithm will utilize local variables to track the sum of either array adding only a constant cost association in both the initial declaration and within the loop algorithm. Similar to our maximum difference algorithm, we rely on an iterative approach to binning values as we encounter them in the sorted array with a performance of O(n). We take the difference between the sum of either array to determine which array to place the current value, adding a constant cost.

Much like our maximum difference algorithm, the expected performance is still limited by our sorting algorithms performance of θ(nlogn). We could instead use quick sort in either algorithm and potentially obtain O(n) performance, however we could also get O(n2) performance. Another alteration we considered was to still use a normal ascending ordered sort

and just iterate in reverse on our sorted algorithm, but it seems slightly more direct to start outright with descending order and work in usual loop direction.

`Minimum_Difference(Array_A):`

```
// Returned Array_A sorted in descending order
sorted_A = Mergesort_descending(Array_A)                          // (nlogn)

// Initialize array 1 with the largest value
// and a value to track total sum of array 1
A1[0] = sorted_A[0]                                               // (c)
sum_A1 = sorted_A[0]                                              // (c)

// Initialize array 2 empty with a sum of 0
A2[]                                                              // (c)
sum_A2 = 0                                                        // (c)

// iterate over our array starting at position 1 since
// position 0 is already in A1
For i = 1 until i = sorted_A.length:                              // (n)
     if sum_A1 - sum_A2 is negative:                              // (c)
          Add sorted_A[i] to A1[]                                 // (c)
          Increase sum_A1 by sorted_A[i] value                    // (c)
     else:
          Add sorted_A[i] to A2[]                                 // (c)
          Increase sum_A2 by sorted_A[i] value                    // (c)
     i = i + 1                                                    // (c)

Return A1 and A2 arrays
```

Expected Input/Output example sets:

Input = [10, 22, 15, 60, 31]
Output:
A1 = [60, 10]
A2 = [31, 22, 15]
Difference = 2

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [60, 31, 22, 15, 10]
// Step: 1
Array 1 = [60]
```

```
Array 2 = []
Difference = 60
// Step: 2
Array 1 = [60]
Array 2 = [31]
Difference = 29
// Step: 3
Array 1 = [60]
Array 2 = [31, 22]
Difference = 7
// Step: 4
Array 1 = [60]
Array 2 = [31, 22, 15]
Difference = -8
// Step: 5
Array 1 = [60, 10]
Array 2 = [31, 22, 15]
Difference = 2
```

Input = [1, 2, 3, 4, 5]
Output:
A1 = [5, 2]
A2 = [4, 3, 1]
Difference = 1

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [5, 4, 3, 2, 1]
// Step: 1
Array 1 = [5]
Array 2 = []
Difference = 5
// Step: 2
Array 1 = [5]
Array 2 = [4]
Difference = 1
// Step: 3
Array 1 = [5]
Array 2 = [4, 3]
Difference = -2
// Step: 4
Array 1 = [5, 2]
Array 2 = [4, 3]
```

```
Difference = 0
// Step: 5
Array 1 = [5, 2]
Array 2 = [4, 3, 1]
Difference = -1
```

Input = [1]
Output:
A1 = [1]
A2 = []
Difference = 1

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [1]
// Step: 1
Array 1 = [1]
Array 2 = []
Difference = 1
```

Input = [1, 10000]
Output:
A1 = [10000]
A2 = [1]
Difference = 9999

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [10000, 1]
// Step: 1
Array 1 = [10000]
Array 2 = []
Difference = 10000
// Step: 2
Array 1 = [10000]
Array 2 = [1]
Difference = 9999
```

Input = [1, 1, 1, 1]
Output:
A1 = [1, 1]
A2 = [1, 1]
Difference = 0

```
// Initial array is sorted, then proceeds through algorithm.
```

```
sorted_A = [1, 1, 1, 1]
// Step: 1
Array 1 = [1]
Array 2 = []
Difference = 1
// Step: 2
Array 1 = [1]
Array 2 = [1]
Difference = 0
// Step: 3
Array 1 = [1]
Array 2 = [1, 1]
Difference = -1
// Step: 4
Array 1 = [1, 1]
Array 2 = [1, 1]
Difference = 0
```

Input = [1, 2, 3, 4, 10]
Output:
A1 = [10]
A2 = [4, 3, 2, 1]
Difference = 0

```
// Initial array is sorted, then proceeds through algorithm.
sorted_A = [10, 4, 3, 2, 1]
// Step: 1
Array 1 = [10]
Array 2 = []
Difference = 10
// Step: 2
Array 1 = [10]
Array 2 = [4]
Difference = 6
// Step: 3
Array 1 = [10]
Array 2 = [4, 3]
Difference = 3
// Step: 4
Array 1 = [10]
Array 2 = [4, 3, 2]
Difference = 1
```

```
// Step: 5
Array 1 = [10]
Array 2 = [4, 3, 2, 1]
Difference = 0
```