I. Introduction

Given an array A of n real numbers, the threeWayMergesort problem sorts the array in a nondecreasing order. For example, consider the following array A indexed from 0 to 7:

Index	0	1	2	3	4	5	6	7
A	27	10	12	20	25	13	15	22

The sorted array of A is [10, 12, 13, 15, 20, 22, 25, 27]. Divide-and-conquer is an algorithm design paradigm in which a problem is recursively broken down into two or more subproblems of the same type. The solutions to the subproblems are then combined to produce a solution to the original problem. As we will see, this approach applied to the *threeWayMergesort* problem yields an efficient algorithm by modifying a standard Mergesort algorithm to sort an array of n items into three subarrays of approximately n/3 items.

II. Overview of Algorithms

The divide-and-conquer approach to the threeWayMergesort problem involves splitting the array in three subarrays and making recursive calls on each of the three subarrays with n/3 items. The *conquer* phase sorts each subarray and then, the combine phase will combine the solution by merging those split subarrays into a single sorted array. The pseudocode for the divide-and-conquer algorithm is divided into two separate functions, threeWayMergesort(.) and Merge(.), where the first function is the main divide-and-conquer algorithm and the second function performs the combine phase. Given an array A indexed from 0 to n-1, the initial function call should be `threeWayMergesort(A)`

```
Algorithm: threeWayMergesort(A[0..n-1])
// Sorts array A by recursive mergesort
// Input: An array A of orderable elements
// Output: Array A sorted in nondecreasing order
if n == 1 then
      // The array is already sorted.
      return
else
      if n == 2 then
            if A[0] > A[1] then
                  // swap
                  swap(A[0], A[1])
      else
            E ← E[0..2n/3 - 1]
            copy A[0..⌊n/3⌋-1] to B[0..⌊n/3⌋-1]
            copy A[⌊n/3⌋..⌊2n/3⌋-1] to C[0..(⌊2n/3⌋-1-⌊n/3⌋+1)-1]
            copy A[⌊2n/3⌋..n-1] to D[0..((n-1)-⌊2n/3⌋+1)-1]
            Mergesort(B[0..⌊n/3⌋-1])
            Mergesort(C[0..(⌊2n/3⌋-1-⌊n/3⌋+1)-1])
            Mergesort(D[0..((n-1)-⌊2n/3⌋+1)-1])
            Merge(B,C,E)
            Merge(E,D,A)
```

```
      end-if
end-if
```

In the algorithm below,

```
i = current location in B
j = current location in C
k = current location in newly constructed A
```

```
Algorithm: Merge(B[0..p-1],C[0..q-1],A[0..p+q-1])
// Merges two sorted arrays into one sorted array
// Input: Arrays B and C both sorted
// Output: Sorted array A of the elements of B and C
i←0; j←0; k←0
while i < p and j < q do
  if B[i] ≤ C[j] then
    A[k]←B[i]
    i←i+1
  else
    A[k]←C[j]
    j←j+1
  end-if
  k←k+1
end-while
if i==p then
  copy C[j..q-1] to A[k..p+q-1]
else
  copy B[i..p-1] to A[k..p+q-1]
end-if
```

Have not completely exhausted B or C

Have exhausted B

We first noted that our threeWayMergesort(.) algorithm assumed that the number of elements is within threeWayMergesort(.), the algorithm runs a divide-and-conquer algorithm by initially checking if the subarray contains only one element. This case belongs to the base-case, and we do not further continue our algorithm since the array is already sorted. We then check if there are only 2 elements with the second element having a smaller value than the first element. If this is the case, then we swap the order of these two elements. Otherwise, we will recursively split the array into three subarrays until all the subarrays either have 1 or 2 elements left. In our pseudo code, we call this function three times in order to ensure the recursive splitting of the array A into three subarrays until it hits the base case, which is when the subarray elements only have 1 or 2 elements. Then, we will execute the combine phase by calling a function Merge(.).

In our pseudocode, the Merge(.) algorithm will combine two subarrays given two sorted arrays as inputs. Then, we have i and j cursors to keep track of the element to be compared in each input array from both input arrays. Then, the smaller one will be copied to the merged array and update the cursor. This process will be repeated until one of the input arrays is exhausted. When one of those arrays is exhausted, the rest of the elements from non-exhausted array will be copied into the merged array.

III. Time and Space Complexity

To simplify our analysis, we assume that size of the array n is a power of 3 where there exist at least 3 elements in the array. Our analysis will focus on the number of comparisons which is our basic operations. First, let T(n) be the total number of comparisons needed by our divide-and-conquer algorithm for an array of size n. For the base case (n=1), the algorithm does not require any such comparison since the array is already sorted. Also, we will operate one comparison if there are only two elements in the array. Then, we will compare those two elements and swap if the former element has a smaller value than that of the latter one. Otherwise, we recursively call threeWayMergesort(.) on three-way splitted array and then call merge(.) as part of the combine phase. Thus, the recurrence relation for T(n) is given by

$$T(n) = 3T\left(\frac{n}{3}\right) + T_{merge}\left(\frac{2n}{3}\right) + T_{merge}(n),\ T(1) = 0,\ T(2) = 1$$

where $T_{merge}(n)$ is the number of comparisons within $Merge(.)$. Now, let's consider the worst case of merge operations, denoted $T_{merge}(p, q, n = p + q)$ at which *p, q,* and *r* are sizes of the first, second, and third subarrays. Notice that the merge operation takes k-1 times assuming that k is the number of elements that are in the merged array. Then, the worst case for merging operations is going to be as follows.

$$T_{merge}(p, q, n = p + q) = T_{merge}\left(\frac{2n}{3}\right) + T_{merge}(n) = \left(\left(\frac{2n}{3}\right) - 1\right) + (n - 1) = \frac{5n}{3} - 2$$

Now, notice that worst case for *threeWayMergesort* occurs when smaller elements come from alternating arrays. This implies that

$$T_{worst}(n) = 3T_{worst}\left(\tfrac{n}{3}\right) + \left(\tfrac{5n}{3}\right) - 2 \text{ for } n > 1, T_{worst}(1) = 0.$$

By Master Theorem, note that $a = 3, b = 3, f(n) = \frac{5n}{3} - 2$ with $d = 1$.
Note that $a = 3 = 3 = b^d$. Hence, by Master Theorem,

$$T_{worst}(n) \in \theta(n\, logn).$$

At the top level recursive call, the sum of the number of items in the three arrays is about $n/3$. In the recursive call at the next level, the sum of the number of the items in the three arrays is approximately $n/9$, and in general, the sum of the number of items in the three arrays at each recursion level is about one-third of the sum at the previous level. Given that there are $\log_3 n$ levels, the total number of extra array items is about

$$n + \frac{1}{3}n + \frac{1}{9}n + \cdots = n + \sum_{i=1}^{\log_3 n} \frac{1}{3^i} \leq n + \sum_{i=1}^{\infty} \frac{1}{3^i} = n + \frac{1}{2}n = \frac{3n}{2}.$$

Therefore, the space complexity of the *threeWayMergesort* is approximately $\frac{3n}{2}$ or O(n).

Appendix

IV. Java Implementation

Sorts.java

```
/* Java program for Merge Sort */
public class Sorts
{
    public static void merge(int[] B, int[] C, int[] A)
    {
        int p = B.length;
        int q = C.length;
        // int i = 0, j = 0, k = 0;
        int i = 0;
        int j = 0;
        int k = 0;

        while (i < p && j < q)
        {
            if (B[i] <= C[j])
            {
                A[k] = B[i];
                i++;
            }
            else
            {
                A[k] = C[j];
                j++;
            }

            k++;
        }

        if (i == p)
            System.arraycopy(C, j, A, k, q - j);
        else
            System.arraycopy(B, i, A, k, p - i);

    } // end of merge method

    public static void threeWayMergesort(int[] A)
    {
        int n = A.length;
        int B[] = new int[n / 3];
        int C[] = new int[n / 3];
        int D[] = new int[n - B.length - C.length];
        int E[] = new int[B.length + C.length];
```

```
        if (n == 1){
            return;
        }else {
            if(n == 2) {
                  if(A[0] > A[1]) {
                        // swap
                        int temp = A[0];
                        A[0] = A[1];
                        A[1] = temp;
                  }
            }else {
                  System.arraycopy(A, 0, B, 0, n/3);
                System.arraycopy(A, n/3, C, 0, n/3);
                System.arraycopy(A, 2*(n/3), D, 0, D.length);
                threeWayMergesort(B);
                threeWayMergesort(C);
                threeWayMergesort(D);
                merge(B, C, E);
                merge(E, D, A);
            }
        }



    } // end of mergesort method

    /* A utility function to print array */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

sortDriver.java

```
import java.io.*;
import java.util.Random;

public class sortDriver {

    public static void main(String args[])
    {
        int arr1[] = {12, 11, 13, 5, 6, 7};

        System.out.println("Given Array 1");
        Sorts.printArray(arr1);

        Sorts.threeWayMergesort(arr1);
```

```
        System.out.println("\nSorted Array 1");
        Sorts.printArray(arr1);

        // generate a uniformly distributed int random numbers
        Random r = new Random();
        int[] arr2 = new int[20];
        for (int i = 0; i < arr2.length; i++) {
            arr2[i] = r.nextInt() % 20;
        }

        System.out.println("\nGiven Array 2");
        Sorts.printArray(arr2);

        Sorts.threeWayMergesort(arr2);
        System.out.println("\nSorted Array 2");
        Sorts.printArray(arr2);

    }

}
```