Instructions:

- Review the requirements given below and complete your work. Please submit all files through Canvas (including all input and output files).
- Grading: 100 points (50 + 50) + Bonus 20 points

Part 1: Sorting

Implement the following sorting algorithm.

- First split the array into non-decreasing and decreasing segments (i.e., segments such that a[i] ≤ a[i+1] ≤ … ≤ a[i+k] or a[j] > a[j+1] > … > a[j+m]).
- Reverse all decreasing segments. Now the array is partitioned into non-decreasing segments.
- Merge any two segments with each other by comparing its elements and then the next two segments and so on, resulting in a non-decreasing array.
- Repeat this process until all segments are merged.

Add appropriate **Junit** tests.

Example:
{2,3,6,1,9,7,5,4}
Split into {2,3,6}, {1,9}, {7,5,4}
Reverse {7,5,4} to {4,5,7}
Segments will now look like: {2,3,6}, {1,9}, {4,5,7}
Merge: 1st Iteration: {1,2,3,6,9}; 2nd Iteration: {1,2,3,4,5,6,7,9}

Part 2: Modified Quick Sort (by Bentley and McIlroy)

Quicksort algorithm is discussed in Special Topic 14.3 in the textbook. Implement the following modification of the quicksort algorithm. Instead of using the first element as the pivot, use an approximation of the median.
Case 1: If n ≤ 7, use the middle element.
Case 2: If n ≤ 40, use the median of the first, middle, and last element.
Case 3: Otherwise, compute the "pseudomedian" of the nine elements a[i * (n - 1) / 8], where i ranges from 0 to 8. The pseudomedian of nine values is *med(med(v0, v1, v2), med(v3, v4, v5), med(v6, v7, v8))*, i.e., the median of medians of the array. To compute that, divide the array into 3 equal parts, compute the median of each part, and then compute the median of the 3 medians.

Compare the running time of this modification with that of the original quicksort algorithm on sequences that are nearly sorted or reverse sorted, and on sequences with many identical elements. What do you observe? Provide your observations as comments in the code.

Part 3: 2D Binary Range Search (Bonus)

Given an array, perform Binary Search and return the indices of the range of values in the array.

Create a search method which takes in an integer array and the range of keys to be searched. If the range is valid, return the indices of all the numbers that belong to the range. Your key should be a range. If the range is {5, 10}, you must print the indices of all the numbers between 5 and 10 ***both non-inclusive*** (if present). ***The range does not have to be sorted*** (i.e., user may provide a range of {10, 5} which will be equivalent to {5, 10}. You array and key may have positive or negative numbers.

Note that there may be multiple correct answers (in an event of duplicates in the array), you must return **all the indices**. If there is no valid answer, return -1. *Notice that the indices are for the original array, not the sorted array*!

Test your code with all the possible cases using Junit.

Algorithm:

1) Sort the array first and then perform binary search. **Sorting of the array should be done without using Arrays.sort( ).** Use your own sort method (*any*).
2) Sort the key (if needed).
3) Check if there are elements between the lower limit and the upper limit (of the range / key). If there are, find the indices of the original array and return them.
4) If the range is invalid or no values are found, return -1. Invalid range may mean that range has no numbers to search (e.g., [4,4] would search nothing).

Example 1:

int M[ ] = { 1,5,4,0,9,9,6,7,8,1,1,3,5,0,9,7,3,0,8,6};
int key = {5, 10};

You may have to first sort the array
M[ ] would now look like : {0,0,0,1,1,1,3,3,4,5,5,6,6,7,7,8,8,9,9,9};
Output: "The elements between 5 and 10 are found at position:
4,5,6,7,8,14,15,18,19"

Example 2:

int M[ ] = { -1,5,4,0,9,9,6,7,8,1,1,3,5,0,9,7,3,0,8,6};
int key = {2, -2};
Output: The elements between -2 and 2 are found at position:
0,3,9,10,13,17

Example 3:

int M[ ] = { -1,5,4,0,9,9,6,7,8,1,1,3,5,0,9,7,3,0,8,6};
int key = {0, 1} or {4, 4};
Output: -1