CSCI205 Computer Science III
Lab Exercise: Quick Sort Experiments (and a small Merge Sort experiment)

Feel free to use Vectors for this assignment and do not worry about implementing type flexibility. Focus on the logic of the algorithm using a primitive type like: int

Objectives:

- Experiment with various approaches to the quicksort
- Experiment with various approaches to the pivot choice
- Compare call memory consumption between Merge and Quick sort
- Tally and graph algorithm cost
- Extrapolate conclusions

A big limitation of quicksort is that it has $O(n^2)$ worst-case running time. The following are two common optimizations

- ***The cutoff to insertion sort***. Switch to insertion sort for arrays with size less than a pre-decided limit. Once the size of the sub-array goes lower than the limit, apply insertion sort on that sub-array. Limit varies from system to system and typically it is between 5 to 27. This approach limits the recursive depth spawned by increasingly small list ranges. Your insertion sort will need to be provided with ranges (beginning and ending points) to sort. Pass it the vector (or array) and two ints **start** and **end.**

  Another cutoff approach is to avoid **many small sorts** by simply stopping at the cutoff. After the entire collection has been processed you can insertion sort in one step. This would give you one insertion sort pass that would execute in ***O(kn)*** time where ***k = cutoff point***

- ***Median-of-three partitioning***. Use the median of a small sample of items taken from the array as the partitioning item. Doing so will give a slightly better partition but at the cost of computing the median.

  Choose three items from the array: **array[0], array[length / 2], array[length – 1].** Sort those items while you have them. Choose the middle item as the pivot.

A better way of choosing the pivot? Median of three is certainly an improvement on the lazy pivot choice, but is there a better way? Cue *Tukey's ninther* . . . A more robust estimator in which a median is computed following the median of three rule with limited recursion:

if A is the sample array, and

$$\text{med3}(A) = \text{median}(A[1], A[\tfrac{n}{2}], A[n]),$$

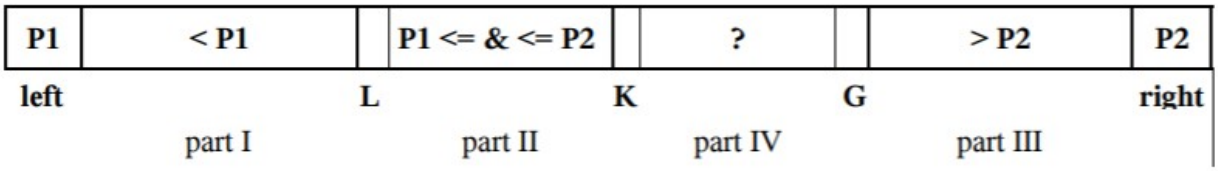
then

$$\text{ninther}(A) = \text{med3}(\text{med3}(A[1 \ldots \tfrac{1}{3}n]), \text{med3}(A[\tfrac{1}{3}n \ldots \tfrac{2}{3}n]), \text{med3}(A[\tfrac{2}{3}n \ldots n]))$$

Notice that *ninther* applies the median of three rule to thirds of the array and then applies the median of three rule to those medians.

Dual Pivot Quicksort: Arrays.sort() in the Java API uses this approach

This Dual-Pivot Quicksort algorithm partitions a source array **T [ ] a** to three parts, defined by two pivot elements **P1** and **P2** (and therefore, there are three pointers **{L, K, G}** and **left** and **right** — indices of the first and last elements respectively) shown in the following figure.



The algorithm consists of the following steps:

1. For small arrays **(length < 27)**, use the Insertion sort algorithm.
2. Choose two pivot elements **P1** and **P2**. We can get, for example, the first element **a[left]** as **P1** and the last element **a[right]** as **P2**.
3. **P1** must be less than **P2**, otherwise they are swapped. This reslts in the following parts:
    a. **part I** with indices from **left+1** to **L−1** with elements, which are less than **P1**,
    b. **part II** with indices from **L** to **K−1** with elements, which are greater or equal to **P1** and less or equal to **P2**,
    c. **part III** with indices from **G+1** to **right−1** with elements greater than **P2**,
    d. **part IV** contains the rest of the elements to be examined with indices from **K** to **G**.
4. The next element **a[K]** from the **part IV** is compared with two pivots **P1** and **P2**, and placed to the corresponding **part I, II, or III**.
5. The pointers **L, K,** and **G** are changed in the corresponding directions.
6. The steps 4 - 5 are repeated while **K ≤ G**.

7. The pivot element **P1** is swapped with the last element from **part I**, the pivot element **P2** is swapped with the first element from **part III**.
8. The steps 1 - 7 are repeated recursively for every **part I, part II, and part III.**

Task One: Create working implementations of the following algorithms (some of these have been provided for you in the lecture materials). Add the ability to tally relevant operations in each (comparisons, swaps, memory). Decompose the relevant tasks into functions for ease of development and analysis. It is perfectly fine to have multiple quicksort functions.

1. Quicksort with lazy pivot choice of either right most or left most element
2. Quicksort with median of three pivot choice
3. Cutoff to insertion sort
4. Quicksort with *Tukey's ninther* pivot choice
5. Dual pivot quicksort

Be sure to provide yourself a convenient way to switch between the various implementations, particularly the pivot choice and the *cutoff-to-insertion*. You will be experimenting with each.

Quick Sort Experiments

Experiment with the various implementations of the quicksort. Throughout this exercise I will be asking you to plot data sets.

Experiment One (Graph One): Quicksort degeneration

- This graph will illustrate one quicksort approach to prove that quicksort can degenerate to $O(n^2)$ running time
- Run quicksort with ***lazy pivot choice***
- Graph the results with clearly marked axis
- ***Provide detailed conclusions that are not generated by AI. It is in your best interest to learn what is going on here***

Experiment Two (Graph Two and Three): Worst case

- This graph will illustrate three quicksort approaches to a worst case scenario of inverse ordering
- Run quicksort with *lazy pivot*, *median of three pivot* and quicksort with *Tukey's ninther*
- Graph Two: The results showing **comparisons** with clearly marked axis
- Graph Three: the results showing **swaps** with clearly marked axis
- ***Provide detailed conclusions that are not generated by AI. It is in your best interest to learn what is going on here***

Experiment Three (Graph Four and Five): Call Stack Memory Consumption

- This experiment will take the average case approach to random ordering.
- Tally and graph **(graph 4)** the amount of call stack memory consumed by each of the following sorts. Remember, this correlates to the recursive depth and not **total** number of recursive calls.
    - Dual Pivot Quicksort
    - Median of Three Pivot
    - Tukey's Ninther Pivot
    - Merge Sort
- Apply the *cutoff-to-insertion* for **ranges <= 15** elements in size
- Tally and graph **(graph 5)** the amount of memory consumed by each of the following quicksorts using the *cutoff-to-insertion*
    - Dual Pivot Quicksort
    - Median of Three Pivot
    - Tukey's Ninther Pivot
    - Merge Sort
- ***Provide detailed conclusions that are not generated by AI. It is in your best interest to learn what is going on here***

In General: If you have relevant console display, please make it legible and provide context. If you have console display that is irrelevant, please comment it out.

Do you feel like I left anything out? Add it to your work.

Submission: Push all relevant files including the make file. All code should be leak free. If there are any special instructions, please include them. Review the instructions and ensure that you have everything covered.