The standard partition algorithm in Quicksort has one pivot and two groups of elements (the small elements and the large elements). We are going to investigate Quicksort when the partition algorithm has more than one pivot. In general, if there are k pivots, there will be $k+1$ groups of elements. We will count comparisons and moves as a function of n and k.

PART 1.

Write a program for this new version of Quicksort. The quality of your code counts. For the program you hand in, you can assume that $n \leq 1000$ and $k \leq 100$.

- Use the last k elements in the list as the pivot elements for partition. Sort them using Insertion Sort.
- End the recursion when the list has size at most $2k$. At that point, sort the list using Insertion Sort.
- As you process the elements during partition there will be the $k+1$ groups, followed by the unprocessed elements, and finally the k pivots. You need to figure out what group the next unprocessed element belongs to, rotate the groups of larger elements right, and put the element into its proper group. If the next unprocessed element belongs to the ith largest group this should take exactly $i+1$ moves.

PART 2.

We want to experimently determine how many COMPARISONS and MOVES the algorithm does as a function of n and k. For this part, you will need to do experiments with a variety of sizes of k and n (without restriction on their values). Exactly how to do the experiment is up to you.

You will need to create random lists of n distinct values, which might as well be the numbers from 1 to n (since only their relative value matters). For each experiment, create a random permutation of the n numbers. You must write the permutation code yourself, i.e. do not use a library routine; you may use the random number generator provided. (Note: For the submit server tests to run correctly, you must use the numbers 1 to n, not 0 to $n-1$.)

Let $C(k,n)$ and $M(k,n)$ be the average number of comparisons and moves, respectively, for your experiments with n elements and k pivots. For fixed k, we are expecting $C(k,n)$ and $M(k,n)$ to grow as $n \log n$, so we expect $C'(k,n) = C(k,n)/(n \log n)$ and $M'(k,n) = M(k,n)/(n \log n)$ to look like constants. Using these values, we can estimate the exact high order terms for $C(k,n)$ and $M(k,n)$. The behavior of $C(k,n)$ and $M(k,n)$ as a function of k is less obvious. For the following you should choose an appropriate base for the logarithm.

- Make a table with k, n, $C(k,n)$, $M(k,n)$, $C'(k,n)$, and $M'(k,n)$.
- Graph n versus $C(k,n)$ and $M(k,n)$.
- Graph n versus $C'(k,n)$ and $M'(k,n)$.
- Give a formula that estimates the number of comparisons and moves as a function of n for each fixed k. If your experiments do not seem to converge, do the best you can.

If you are not sure what data to collect or how to present it, think of yourself as a scientist trying to understand Quicksort with this generalized partition. Imagine that the government is interested in your results. You will be called up in front of a congressional committee and mercilessly grilled while the whole country is watching.

BONUS POINTS.

There are bonus points available for bells and whistles. There are many ways to extend the program or the experiments. Do so only if it interests you, not for the bonus points. Make sure sure to clearly state what you have done. For example:

- Do a binary search to determine which group the new element belongs in.
- If the next unprocessed element belongs in the group with the largest elements, don't do any moves (especially if k is "small"). If possible, you should do this in a clever way that does not slow down the program when the next unprocessed element belongs in a different group.
- Build the $k/2$ groups of small values from left to right, and the $k/2$ groups of large values from right to left. (Why is this a good idea?)
- Make the algorithm work efficiently if there are duplicate values. If, for example, all of the elements have the same value, quicksort will put all of the elements (except for the pivot) into one group and make little progress (giving a quadratic runtime). You would like values that are equal to the pivot to be split evenly between two groups. If possible, you should do this in a clever way that does not slow down the program when the values are distinct.
- Do actual timings.
- Do experiments to figure out what size the recursion should end as a function of k and n. This is more interesting with actual timings than with counting moves and comparisons. (Why?)
- Estimate the number of comparisons and moves as functions of both k and n.

Ideally, you should write the code for ideas that modify the algorithm. You only have to implement the modified algorithm (not the basic algorithm). However, it is not absolutely necessary to write code for your modified algorithm. If you have a novel idea you can write it up. You can even flesh out one of the example ideas listed above, if you can say something nonobvious. The more you do, the better.

As an example, if your idea is to use Selection Sort rather than Insertion Sort, that by itself in not interesting, since Selection Sort is not clearly an improvement and is no more difficult to implement. However, it would be interesting to run experiments comparing the use of Selection Sort versus Insertion Sort in the algorithm, since Selection Sort uses fewer moves but, on average, more comparisons.

PROGRAMMING RULES.

You must write this program yourself. You may discuss general ideas with other people. You may not look at any other *code* or *pseudo-code* (even if it is on the internet), other than what is on our website or in our book.

SUBMISSION GUIDELINES.

You will implement programs for both parts in JAVA and submit it on Gradescope. A skeleton code is provided on the course website and you must build you own program upon it.

To start, if you use Eclipse, you can create a new project and copy the folder **cmsc351s19** into src under your project folder. If you use commandline tools, you can compile the code by **javac cmsc351s19/*.java** and execute Main by **java cmsc351s19.Main**

You should not change the name of **ModifiedQuicksorter.java** and **PermutationGenerator.java.** You can, however, create new classes and new JAVA files as you see appropriate. When creating new classes, make sure all of them lie in the same package **cmsc351s19** by adding **package cmsc351s19;** at the top of each file. Moreover, you should not change the name and signature of methods that are already in the skeleton code, but you can create your own helper functions and class members.

To help you test your code, we've included a **Main.java** to read space-separated integers that set different parameters in **ModifiedQuicksorter.java**.

When you implement **PermutationGenerator.java**, you should use **m_random** defined in the constructor as your source of randomness.

After you finish, pack all your .java files in a zip archive and submit it under 'Modified Quicksort - Code'. After you submit the code, make sure that your code compiles by checking the autograder output on Gradescope. In addition, submit a pdf file containing experiment results from PART 2 under 'Modified Quicksort - Writeup'.