Learning Goals

During this lab, you will cover:

- Mergesort
- Quicksort
- Insertion sort

General Problem Statement

Given an array of length $n$, sort it in ascending order (could also be descending). You cannot assume anything about the contents of the array.

Mergesort

We can apply the principles of Divide and Conquer when thinking about approaching this problem.

**Divide:** Can we divide this into equivalent subproblems? Yes, we can divide this array into two halves each with $\frac{n}{2}$ elements. Thus, each is an equivalent subproblem.

**Conquer:** How can we recursively sort the two halves? That’s easy! Since we already broke it into subproblems, we will recurse using mergesort on the two halves until we hit the base case of a singleton element (we trivially know that a singleton element is sorted).

**Combine:** Once we have two sorted arrays we can combine them in $O(n)$ time by interleaving the halves!

**Runtime:** We will discuss the worst case of the algorithm. We know that the runtime of mergesort, which we will denote $T(n)$, depends on two things. First, we consider the recursive calls. We are constantly splitting the array in halves, so we have a $2T\left(\frac{n}{2}\right)$ term in the recurrence relation. Finally, we have to consider the interleaving of arrays, which we claimed earlier was a $O(n)$ operation. Thus, our runtime becomes $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Solving this out with whichever method you choose, $T(n) = \Theta(n \log n)$.

Note: Mergesort is a stable sort.

Quicksort

Recall the quicksort algorithm:

1. Pick an element, called a pivot, from the array.

2. Perform the partition operation on the array: Reorder all the elements such that all values less than the pivot come before it, and all values greater or equal to the pivot come after. After this operation, the pivot is in the final position.

3. Recursively apply the above steps to the generated sub-arrays. Once again, the singleton base case is trivially sorted.

**Runtime:** We will discuss the worst case of the algorithm. If we have a very poor choice of pivot, our algorithm will partition the array into two arrays of size 0 and $n - 1$. If we do this poor selection each time, we have a runtime of $T(n) = \sum_{i=0}^{n} (n - i)$ which evaluates to $O(n^2)$. In general, however, the average case of quicksort will run in $O(n \log n)$ time. We will defer this analysis to CIS 320.

Note: An efficient implementation of quicksort is not stable.
**Insertion Sort**

Insertion sort is a simple solution to sorting. Useful for smaller data sets and implemented very easily, insertion sort is a stable sort, is done in-place, and can be considered an “online” algorithm, meaning it can sort a list as it receives it!

Recall:

1. We recall the invariant in place: We will keep a sorted side of the array and have some function to *insert* a value into the sorted sequence at the beginning of the array. Essentially, it will begin at the end of the sorted sequence and shift each element one place to the right until a suitable position is found for the element.

2. We begin at the leftmost element in the array, and call *insert* in order to position each element in its proper position in the array. The ordered sequence is built up from the left to the right.

Refer to the java implementation below:

```java
for (int i = 1; i < A.length; i++) {
    int j = i;
    while (j > 0 && A[j - 1] > A[j]) {
        swap(A[j], A[j - 1]);
        j--;
    }
}
```

**Runtime:** We can see that the outer loop executes $n - 1$ times. In the worst case, the inner while loop makes $n$ swaps. We can trivially see that the worst case runtime for this algorithm is $O(n^2)$. 
Problems

Problem 1

Banks often record transactions on an account in order of the timestamp of the transactions. However, many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. Thus, the problem of converting timestamp ordering to check number ordering is essentially a problem of sorting almost sorted input. You are the manager of the bank, and a new software developer asks if he should solve this problem with insertion sort or quicksort. Being the accomplished computer scientist that you are, you instantly tell him which procedure trumps the other. What did you tell him and why?

Problem 2

Implement a \texttt{merge()} function that might be utilized in mergesort. Assume you are given the correct length of the output array parameter, \texttt{merged}.

\footnotesize{
\begin{verbatim}
private static void merge(int [] A, int [] B, int [] merged) {...}
\end{verbatim}
}

Problem 3

Assume you somehow generate a pivot for your quicksort function. Implement an in-place \texttt{partition()} function, using this pivot to partition against. Return the partitioning index, i.e., the first index \( i \) such that \( \texttt{nums}[i] \geq k \).

\footnotesize{
\begin{verbatim}
private static int partition(int [] nums, int k) {...}
\end{verbatim}
}

Problem 4

Assume you have access to the following functions:

\footnotesize{
\begin{verbatim}
private void swap(int [] A, int i, int j) { /* ... */ }
private int partition(int [] A, int firstIndex, int lastIndex) { /* ... */ }
\end{verbatim}
}

Using the above, implement an algorithm to find the \( k \)-th smallest value in a given array. This algorithm is known as \texttt{quickselect}.
