Precept Outline
- Review of Lectures 5 and 6:
  - Comparators and Comparables
  - Elementary sorts
  - Mergesort

Relevant Book Sections
- Book chapters: 2.1, 2.2 and 2.5

A. Review: Elementary Sorts + Mergesort

Your preceptor will briefly review key points of this week's lectures.

B. Comparable & Comparator

Solve the exercises in the "Comparable & Comparator" Ed lesson.

C. Mergesort

Part 1: Three-way Mergesort

(Two-way) Mergesort is quite a simple algorithm to describe: to sort n elements, divide the array in half, (recursively) sort each then merge the two halves together. In this exercise, we will study a variant of it: the three-way Mergesort, we divide an array of length n into 3 subarrays of length $\frac{n}{3}$, sort each of them and then perform a 3-way merge.

Given 3 **sorted** subarrays of size $\frac{n}{3}$, how many comparisons are needed (in the worst case) to merge them to a sorted array of size n? Provide your answer in tilde notation.

What is the order of growth of the number of compares in 3-way Mergesort as a function of the array size n? (Here we're counting the total number, including all recursive calls.)

Given a choice, would you choose 3-way or 2-way mergesort? Justify your answer.

Challenge Problem (optional): In an array h of n numbers, an *inversion* is a pair of elements that isn't sorted; that is, two indices i and j such that $i < j$ and $h[i] > h[j]$.

Describe an algorithm to compute the total number of inversions of an array of length n in time $\Theta(n \log n)$. *Hint: think about how you can modify the merge sort algorithm to achieve this.*

D. Assignment Overview: Autocomplete

Your preceptor will introduce and give an overview of your third assignment. Please don't hesitate to ask questions!

Summary of the assignment.

- Implement a `Term` class, which stores a word (as a string) and a numeric weight, and also implements comparators for comparing terms in natural order, in decreasing order of weight, and lexicographically based on the first r characters.
- Create a data type `Autocomplete` that initializes with given arrays of terms and weights, and supports methods to return the weight of a term, the top matching term, and the top k matching terms in descending order of weight.
- Implement a `BinarySearchDeluxe` class, which should use binary search to find the first and last index of a given key in a sorted array (these are important primitives to the `Autocomplete` class).
- Determine the theoretical runtime of the main methods to implement, and report it is using big Theta notation. This will be done in `readme.txt`.