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: O/Ω Notation + Elementary Sorts + Mergesort + Comparable/Comparator

Your preceptor will briefly review key points of this week's lectures. They may refer to the warm-up exercise and the code snippet shown below.

Warm-up: Let $f(n) = 3n + 4n\log_2 n + 8\sqrt{n}\log_2 n$. Select all that apply.

( ) $f(n) = O(n)$
( ) $f(n) = \Omega(n)$
( ) $f(n) = O(\sqrt{n}\log n)$
( ) $f(n) = \Omega(\sqrt{n}\log n)$
( ) $f(n) = O(n\log n)$
( ) $f(n) = \Omega(n\log n)$
( ) $f(n) = O(n^2)$
( ) $f(n) = \Omega(n^2)$
( ) $f(n) = O(\log n)$
( ) $f(n) = \Omega(\log n)$
( ) $f(n) = O(2^n)$
( ) $f(n) = \Omega(2^n)$

```
 public class YourClass implements Comparable<YourClass> {
   public int compareTo(YourClass that) {
       // returns int > 0 if this > that
       // returns int < 0 if this < that
       // returns 0 otherwise
   }

   private static class YourComparator implements Comparator<YourClass> {
           public int compare(YourClass obj1, YourClass obj2) {
           // returns int > 0 if obj1 > obj2
           // returns int < 0 if obj1 < obj2
           // returns 0 otherwise
       }
   }
   public static Comparator<YourClass> yourComparison() {
       return new YourComparator();
   }
   ...
}
```

B. Comparable & Comparator

The code snippet below shows the instance variables of a class `Movie`, and partially filled instance methods that should support comparing elements of this class in three ways:

- by alphabetical order of `title` (the default order);
- by release `year`; and
- by `rating` (0-5 stars).

Fill in the blanks numbered 1 to 6.

```
public class Movie implements ______________(1)______________ {
    private String title;
    private int    year;
    private int    rating;

    public int compareTo(Movie m) {
        return ______________(2)______________;
    }

    public static Comparator<Movie> byYear() {
        return new YearComparator();
    }

    private static class YearComparator implements ______________(3)______________ {
        public int compare(Movie m1, Movie m2) {
            return ______________(4)______________;
        }
    }

    public static Comparator<Movie> byRating() {
        return new RatingComparator();
    }

    private static class RatingComparator implements ______________(5)______________ {
        public int compare(Movie m1, Movie m2) {
            return ______________(6)______________;
        }
    }
    ...
}
```

C. Sorting Algorithms

Part 1: Spring'24 Midterm Problem

Given two integer arrays, `a[]` and `b[]`, the *symmetric difference* between `a[]` and `b[]` is the set of elements that appear in exactly one of the arrays. Design an algorithm that receives two *sorted arrays*, each consisting of n *distinct elements*, and outputs the size of their symmetric difference.

For full credit, it must use $\Theta(1)$ extra memory and its running time must be $\Theta(n)$ in the worst case (the arrays `a[]` and `b[]` should not be modified).

Part 2: Comparison-Based Lower Bounds

The $\sim n \log_2 n$ lower bound for compare-based sorting algorithms is obtained by constructing a *comparison tree*, counting its number of leaves, and comparing this count against the number of possible outputs (permutations of the array).

Adapt this argument to prove that *binary search* is an optimal algorithm for searching in a sorted array: Assuming n is a power of 2, prove that any compare-based algorithm must make at least $1 + \log_2 n$ compares to solve the following problem.

- **Input:** a sorted array a of length n and a search key k.
- **Output:** the index of an element in the array that matches the key, or `-1` if there is no match.

(If you can't prove the exact bound, try with $\sim \log_2 n$.)

Part 3: Finding the Missing Element

Suppose that you are given a sorted array `a[]` with $n - 1$ distinct integers between 0 and $n - 1$. In other words, you are given the array `[0, 1, ..., n - 1]` but with one of the elements missing. Design an algorithm with $\Theta(\log n)$ worst-case running time that outputs the missing element.

For example, if the array is `a[] = [0, 1, 2, 3, 5, 6, 7]`, then $n = 8$ and the missing element is 4.

Part 4: Stability

A sorting algorithm is called *stable* if it maintains the relative order of equal elements. For example, if a deck of cards is pre-sorted by rank, running a stable sort by suit results in a deck in suit-major order (all clubs in sorted order, then all diamonds, hearts and spades).

Run insertion, selection and mergesort (by rank) on the following sequence of cards: `[2♠, 3♠, 4♢, 5♣]`. Which of these algorithms are *not* stable?

D. Optional Bonus Problems

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: in 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.

Part 2: Counting Inversions

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 mergesort to achieve this.*