# CSE374 – Spring 2021 – Assignment 4

This assignment is done individually. You can search for information online, use your textbook, ask the instructor or Teaching Assistant for help. Working with anyone else is a non-authorized collaboration, that is, an act of academic dishonesty.

*Objective*: Leverage the property of a sorted input to create efficient algorithms.

*Due date*: **Friday April 9th end of day (11:59pm)**.
*You can do at least one exercise every day. A little bit regularly may help, but it does not take long to do the assignment so if you want to submit before and be done, you certainly can.*

*How to submit*: A ZIP file sent through the course website (no emails accepted).
ZIP your whole Eclipse project, name it "*Firstname_Lastname*_Assignment4.zip"
You are responsible for ensuring that your submission was correctly submitted.

*Grading (10 pts)*: Each exercise is worth one point. **Exercises are independent and can be done in any order**. You must **use the class names specified**, so that the TA can grade you.

## Overview

The lectures showed you how to decide whether sorting a list or not. Through practice problems, you have also designed algorithms that either did not sort the list, or leveraged the fact that it was sorted when that led to a better (i.e. smaller) order of time complexity. This assignment will give you additional practice on design and implementation. In particular, you will:

- Use binary search, as one of the ways to leverage the fact that a list is ordered
- Write your own comparators to guide Java in sorting your lists
- Propose algorithm design, both by revisiting problems from class and through new ones

The sections in this assignment can be done in any order. Do not just read the first section and get stuck on it: read the whole assignment and start with the sections that speak to you the most.

## Exercise 1. Sometimes you just have to use iterators

When writing a function taking a list as input, we cannot use 'get'. It's inefficient if the list turned out to be a LinkedList, it may cause constipation, and it may curse your descendants for 2.5 generations. The new(-ish) Java loops are a nice way to go through the *elements* of a list, but sometimes we need to better control the flow of the loop and manually decide when to move. Iterators are good for this. Rewrite the algorithm from slide 40 with iterators. As a result, we should not see 'get' anymore.
Your result should be stored in a class named `Exercise1Shared`.

## Exercise 2. Counting Sort

We saw an implementation of counting sort that created a secondary (storage) array as large as the maximum value in the input array (slides 23-28). However, this may lead to leaving a large chunk of the secondary array filled in with zeros on the left, hence wasting a lot of space.
Rewrite the nonComparisonSort (slide 28) so that it does not have any 0's on the left, ever.
Your result should be stored in a class named `Exercise2CountingSort`.

## Exercises 3-5. Binary search

We can *consider* sorting a list when the time complexity of a first solution is greater than the cost of sorting. But sorting by itself does not give us the solution: it simply introduces a new property into the input (i.e. ordering). Thus, we need to figure out how to leverage this property. One method that requires this property is the *binary search*.
While you may have been exposed to binary search in earlier courses, it can be useful to have a fresh reminder of how it works and what it looks like in Java. You can start by watching this short tutorial:

https://www.youtube.com/watch?v=P3YID7liBug

Note that:

• one binary search (i.e. searching for one element) costs O(log $n$). Hence if you search for an element $n$ times, the total will be O($n$ log $n$). This is better than an O($n^2$) solution.

• Java already implements a binary search so you should not write it from scratch. Just call the methods in either the Arrays or Collection (which works for List) classes.

***Exercise 3 (use a binary search to make it efficient!)***
Our solution for the 'shared' function used sorting followed by going through both lists in parallel (slides 37-40). An alternative is to sort the smaller list and, for each element of the larger list, use a binary search to check if it's also in the smaller list. Implement this in the Exercise3FindShared class.

***Exercise 4 (use a binary search to make it efficient!)***
We have an array of sorted elements without repeats, and a target number *x*. We want to return the pair (a, b) where *a* is how many numbers are less or equal to *x*, and b is how many numbers are strictly larger than *x*. Your code will be in the class Exercise4Pair.

***Exercise 5 (use a binary search to make it efficient!)***
An array was sorted but wasn't necessarily stored starting from the first element. Mathematically, the array has *n* sorted elements written in the order $x_k\, x_{k+1}\, x_{k+1} \ldots x_n\, x_1\, x_2 \ldots x_{k-1}$

$x_1$ is the first element, but as you can see it does not necessarily come first. Here are two examples:
Example 1: 15 21 35 60 3 7 9 10
Example 2: 5 7 9 1 2 3 4
We want to first the minimum in this array. That is, the place where the ordering breaks. In example 1 it would be 3, and in example 2 it would be 1.
Obviously we could just scan the array and figure out where one element is bigger than the next… but that would be Θ(n). Remember we want Θ(log n). Write the code in the class Exercise5SortedMin.

**Exercises 6-7. Comparators**
Start by going through the slide "Crash Intro on Comparing" for a few examples on how you can write Java code that executes your custom comparisons to sort/order data. We also recommend going through the video walkthrough at https://www.youtube.com/watch?v=oAp4GYprVHM
*(Interview questions on Java-related jobs may ask for how to implement Comparable or how to write a Comparator. So you should not overlook these two notions.)*

***Exercise 6***
• Write the Exercise6Comparators class, which uses the following fields: name (*String*), age (*Float*), years of experience (*Integer*), punctuality (*Boolean*), collegial (*Boolean*), disciplined (*Short* where higher values indicate more disciplined employees), dependable (*Boolean*), and energetic (*Boolean*).
• Write Java code such that two employees can be compared, to figure out is one is better (+1), worst (-1) or about the same (0) as another. Use your own philosophy of what makes a good employee, but (*i*) do not use names or age, because it's discriminatory; and (ii) use at least 4 fields.

***Exercise 7***
Write a class Exercise7Diff with a method named compare that takes in one list and two comparators. It returns the number of elements that are at the same place when sorted with one comparator or another. For instance, "3 1 2" will give the list "1 2 3" when the comparator is < but "3 2 1" when the comparator is >. Element 2 is at the same place in both cases, so your function should return 1.

***Exercise 8***. Answer question 2.5.4 of the textbook in a class Exercise8Dedup.
***Exercise 9***. Answer question 2.5.18 of the textbook in a class Exercise9Force.
***Exercise 10***. Answer question 1.4.17 of the textbook in a class Exercise10Farthest.
*If you want more midterm practice, also consider questions 1.4.5, 1.4.6, 1.4.19, and 1.4.20.*