EXERCISE 1: Analysis of Sorting Algorithms

Suppose that you have an array of length $2n$ consisting of n B's followed by n A's. Below is the array when $n = 8$.

B B B B B B B B A A A A A A A A

(a) How many compares, as a function of n, does it take to sort the array in ascending order using **Selection Sort**? Use tilde notation.

(b) How many compares, as a function of n, does it take to sort the array in ascending order using **Insertion Sort**? Use tilde notation.

(c) How many compares, as a function of n, does it take to sort the array in ascending order using **Merge Sort**? Use tilde notation.

EXERCISE 2: Three-Way Merge Sort

3-way Merge sort is a variant of the Merge sort algorithm that considers 3 “equal” subarrays instead of 2 subarrays.

(a) Given 3 sorted subarrays of size $n/3$, how many comparisons are needed to merge them to a sorted array of size n? Provide your answer in tilde notation.

(b) What is the *order of growth* of the number of compares in 3-way Merge Sort as a function of the array size n?

(c) Given a choice, would you choose 3-way or 2-way merge sort? Justify your answer.

OPTIONAL: Algorithm Design (*Midterm Spring 2015*)

Let $a = a_0, a_1, \ldots, a_{n-1}$ be an array of length n. An array b is a circular shift of a if it consists of the subarray $a_k, a_{k+1}, \ldots, a_{n-1}$ followed by the subarray $a_0, a_1, \ldots, a_{k-1}$ for some integer k. In the example below, b is a circular shift of a (with $k = 7$ and $n = 10$).

sorted array a[]

1	2	3	5	6	8	9	34	55	89

circular shift b[]

34	55	89	1	2	3	5	6	8	9

Suppose that you are given an array b that is a circular shift of some sorted array (but you have access to neither k nor the sorted array). Assume that the array b consists of n comparable keys, no two of which are equal. Design an efficient algorithm to determine whether a given key appears in the array b. The order of growth of the running time of your algorithm should be $lg(n)$ (or better) in the worst case, where n is the length of the array.

ASSIGNMENT TIPS: Autocomplete

(1) Given an array of elements with duplicates, can we use the book implementation of Binary Search to find the ***first occurence*** of an element?

- The standard implementation of Binary Search finds *an* occurrence, which is not necessarily the *first* occurence.
- Finding the element and then scanning left to find the first occurence yields a linear running time (in the worst case), which is not good!
- In this assignment, you will have to modify Binary Search to find the first (and last) occurence of an element in a sorted array in logarithmic time (in the worst case).
- For full credit, your algorithm has to make at most $1 + \lceil log_2 n \rceil$ compares. However, if your algorithm has a logarithmic order of growth but makes more than $1 + \lceil log_2 n \rceil$ compares, you will lose *only* 1 point.

(2) What is the difference between a **`Comparable`** and a **`Comparator`**?

- A **`Comparable<T>`** is an object of a class that has the method **`compareTo(T other)`**. This method allows the object to compare itself to other objects.
- A **`Comparator<T>`** is an object that can be used to compare two given objects. It has the method **`compare(T obj1, T obj2)`**.
- Making an object **`Comparable`** makes it comparable with other objects using the logic provided in the **`compareTo`** method. However, if we want to implement multiple ways of comparison (for e.g. compare files by name, date created, date modified, etc.), then we need to have multiple Comparators.
- A good example of the use of **`Comparable`** and **`Comparator`** is **`Point2D.java`**, which is available at: https://algs4.cs.princeton.edu/code/. You can use this as a guide when working on the assignment.
- Note that a **`Comparator`** class can have a constructor that takes arguments. This may be needed in the assignment!

(3) What is the order of growth of the **`substring`** method?

- Creating a substring of length r takes time proportional to r.
- Note that the string comparison functions in the assignment should take time proportional to the number of characters needed to resolve the comparison.

  Example: The comparison between X="AAAAAAA" and Y="AABBB" can be resolved when the first "B" in Y is reached. The comparison function should *not* take time proportional to the size of X or the size of Y. It should take time proportional to the number of characters needed to resolve the comparison!
- Most uses of the **`substring`** method in the compare functions do not meet the above time constraint. So, be careful!

(4) A video that provides some tips for the assignment is available on the assignment Checklist page. The video was made in 2014, so a few things are outdated, but most of it still useful!