COMP 250, Winter 2009
Assignment 2
Prof. Michael Langer
Posted Friday Feb. 13
Due Wednesday Mar. 4 at 11:59 PM

Prepared by Mansoor Siddiqui, Neeraj Tickoo, Julien Villemure.
Questions about this assignment should be directed to the TAs above.

Question 1 – (30 points) – Asymptotics

Use the definition of big-Oh to:

a) Prove that $(n+10)^{1.3} + n + 1$ is $O(n^{1.3})$
b) Prove that n^3 is **not** $O(n^2)$

Use the definition of big-Omega to:

c) Prove that $(n \log(n))$ is $\Omega(\log(n!))$

Question 2 – (30 points) - Recursion

You are given two sorted arrays of integers, A and B, each with $N > 0$ elements. Consider the sorted union of A and B, namely the sorted array C of size 2N containing all elements of A and all elements of B. Your goal is to **find the median** of C.

Recall that the median is defined to be the middle element of an array. If the array has an odd number of elements, this definition is clear. However, C has an even number of elements, namely 2N, and thus has two middle elements. For this question we define the median to be **the least of the two elements**.

Example where N = 4 and the median is 3:

A = {2, 3, 7, 9}
B = {0, 1, 4, 9}
C = {0, 1, 2, **3**, **4**, 7, 9, 9}

It is easy to solve this problem in time O(N) by using the merge step of *mergesort* to create C, then returning the least of the middle elements.

a) (20 points) Write a recursive algorithm, with running time O(log(N)), that solves this problem.

You are provided with a Java program, Median.java. The arrays A and B are generated for you and are available as static variables. Your task is to write the contents of the following recursive method:

```
int fastMedian (int s, int t, int p, int q)
```

This method should return the median of the union of A[s,t] and B[p,q], i.e. indices s to t (inclusive) of A, and indices p to q (inclusive) of B.

For the example shown earlier, fastMedian(0, N-1, 0, N-1) should return 3, because it looks at A[0, N-1] and B[0, N-1], i.e. the whole arrays. Similarly, fastMedian(2, 3, 1, 2) should consider the subsets {7, 9} and {1, 4}, returning the median 4.

A simple $O(N^2)$ algorithm is already implemented in slowMedian(). You may use it, for example, to test your solution.

b) (10 points) Write the recurrence relation for your algorithm. Show that the running time is indeed O(log(N)) by deriving an explicit formula for the recurrence relation. Use the substitution method.

Hints:

- Consider C as defined earlier. Suppose its middle two elements are i and j. Then the median is i. If you ignore an element smaller than both i and j, and you also ignore an element bigger than both i and j, the middle two elements of the resulting array are still i and j. Example:

  C = {2, 5, 6, 7, 8, 9} with {i,j} = {6,7}
  C' = {5, 6, 7, 9} with {i,j} = {6,7}

  Note that you can repeat this process more than one time and the median will still be unaffected.

- Remember that integer division disregards fractionals. You will have to be careful. Example: 7 / 2 = 3

- A valid implementation of fastMedian() can be very short. Ours is 10 to 15 lines of code.

Question 3 – (40 points) – Abstract Classes, Interfaces & Generics

In this question, you will be using abstract classes, interfaces, and generics to implement a song playlist. You are provided with a Song class which consists of three fields: the song's title, the artist, and the year of release. The "natural ordering" (i.e. the default sorted order) of songs is in alphabetical order by song title. (This is already implemented for you. It might be helpful to

read about Java's Comparable interface in the Java 6 API to understand how objects are given a natural ordering.)

You are also provided with an abstract class called Playlist, and an incomplete class called SongPlaylist. The SongPlaylist class maintains a list of songs which can be sorted by song title, artist, or year of release.

For both questions below, you should avoid casting wherever possible. Once you've completed your implementation, you can uncomment and run the SongPlaylist.main() method to test your implementation.

a) Modify the SongPlaylist class so that it extends the Playlist class. Implement all inherited methods.

b) Implement the methods sortByArtist() and sortByYear() in the SongPlaylist class. You should not be implementing the sorting algorithm yourself -- rather, you should use one of the Collections.sort() methods. Since songs are not naturally ordered by artist or year, your implementation may not look exactly like the implementation of the SongPlaylist.sortByTitle() method (which is provided for you). In particular, instead of using the Collections.sort(List<T>) method, you can sort the songs without using their natural ordering by calling the Collections.sort(List<T>, Comparator<T>) method*. This will require you to write your own Comparator classes.

Hints:

Be careful not to confuse the Collection interface and the Collections class in the Java API. The Collection interface is implemented by any classes that represent a collection of items (e.g. lists, sets, and so forth). The Collections class, however, is a concrete class with static operations (such as the sort() method) that can be performed on classes which implement the Collection interface. You can find further background info in the Java 6 API:

http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html
http://java.sun.com/javase/6/docs/api/java/util/Collection.html
http://java.sun.com/javase/6/docs/api/java/util/Collections.html
http://java.sun.com/javase/6/docs/api/java/lang/Comparable.html
http://java.sun.com/javase/6/docs/api/java/util/Comparator.html

*Note: If you read the API documentation carefully, you will see that this method actually takes two arguments of type List<T> and *Comparator<? super T>*. The latter simply means that you can pass it a Comparator<S> of any class S such that S = T or S is a superclass of T. You do not need to worry about this and you can assume the required argument is of type Comparator<T>.