CCS245-03, Spring 18
Prof. Karpenko

Project 3, Part 1: Sorting algorithms I[1]

Due: 03/28/2018, 11:59pm

For part 1 of project 3, you need to

1. Implement the following sorting methods:
   - Insertion sort
   - Iterative merge sort
   - In-place Heap Sort
   - Randomized Quick Sort
   - Hybrid sort (that uses both quicksort and insertion sort)

You *may* use the sorting code that I posted on github (assuming you understand the code completely and can explain it), but you may **not** copy any code from any other source (even partially). You may **not** use any built-in sorting methods (or in-built classes from the Collections framework such as ArrayList, HashMap etc.) for this assignment.

Implementation Details

You will need to write a class called `SortingAlgorithms` that implements the following `SortInterface` (it's important that you do **not** modify the signatures of any methods):

```
public interface SortInterface {

        public void insertionSort(Comparable[] array, int lowindex, int highindex,
        boolean reversed);

         public void iterativeMergeSort(Comparable[] array);

        public void heapSort(Comparable[] array, int lowindex, int highindex, boolean
        reversed);

        public void randomizedQuickSort(Comparable[] array, int lowindex, int
        highindex);

        public void hybridSort(Comparable[] array, int lowindex, int highindex);

}
```

Your `SortingAlgorithms` class must implement the interface above. You should **not** use any instance variables for this assignment apart from constants, only local method variables. You may write private helper methods (you would need them for several sorts such as a heap sort).

To test your code, create an array of `Comparable`-s (such as integers) and test all the sorting methods. All sorting algorithms you will write should sort a list *in ascending order* (from

[1] Some *parts* of this assignment are modified from the assignment developed by Professor Galles.

smallest to largest).

We describe each of the sorting algorithms below:

1. Insertion Sort

```
public void insertionSort(Comparable[] array, int lowindex, int    highindex,
boolean reversed);
```

Modify the code of the insertion sort we discussed in class (posted on github) so that:

- It sorts all elements in the array with indices in the range from `lowindex` to `highindex` (inclusive). You should not touch any of the data elements outside the range `lowindex` to `highindex`.
- If reversed is false, the list should be sorted in ascending order. If the reversed flag is true, the list should be sorted in descending order.
- Can sort the array of any Comparable objects, not just integers.

2. Iterative Merge Sort

```
public void iterativeMergeSort(Comparable[] array);
```

This is a variation of merge sort that is **non-recursive**. Note that recursive solutions will not be given any credit.
You may assume the number of elements in the array is $\mathbf{n = 2^k}$ (some power of 2). Here is an example of how iterative merge sort works:

First, we treat the array as an array of sublists of size 1 that are already sorted:
15 3 7 10 9 1 18 6 4 8 12 67 17 13 8 2
Merge pairs of adjacent sublists of size 1 into sublists of size 2:
3 15 7 10 1 9 6 18 4 8 12 67 13 17 2 8

Merge pairs of adjacent sublists of size 2 into sublists of size 4
3 7 10 15 1 6 9 18 4 8 12 67 2 8 13 17

Merge pairs of adjacent sublists of size 4 into sublists of size 8
1 3 6 7 9 10 15 18 2 4 8 8 12 13 17 67
....

Keep going until you sort the whole list. In this example, we just need to merge two sublists of size 8:
1 2 3 4 6 7 8 8 9 10 12 13 15 17 18 67

You are allowed to use one temp array of size n.

3. In-place Heap Sort

```
public void heapSort(Comparable[] array, int lowindex, int highindex, boolean
reversed)
```

Modify the methods of class MinHeap (posted on github) to implement the in-place Heap Sort algorithm we discussed in class (that uses a *max* heap and repeatedly removes the

largest element from the heap and adds it to the end of the array). The algorithm should run **in place** (do **not** use a temporary array). When you first build a heap from the given array, build it **from the bottom up** (that takes O(n)) as opposed to using n insertions into an empty heap (takes O(n log n)).
The method should sort all elements in the array with indices in the range from `lowindex` to `highindex` (inclusive). If `reversed` is true, the list should be sorted in descending order, otherwise in the ascending order.

4. Randomized Quick Sort

```
public void randomizedQuickSort(Comparable[] array, int lowindex, int
highindex);
```

Change the code of the quick sort we discussed in class (that is posted on github) so that:

- It sorts the sub-list of the original list (from lowindex to highindex)
- At each pass, it picks **three random elements** of the sub-list, chooses the **median** of these three elements and uses it as a pivot. How do you compute a median of three values? If we "sort" three elements, the median is the element in the middle (for instance, if the three elements are 5, 2, 19, the median is 5); note that it is different from *mean*! Such algorithm is called a *randomized quick-sort.* The expected running time of a randomized quick sort is O(n log n).

Example: Consider the following array 5, 2, 9, 12, 6, 8, 3, 7 and assume we want to sort the whole array (so lowindex = 0, highindex = 7). We first generate three random integers from 0 to 7 (assuming a uniform distribution) and we get indices 1, 7, 4. These indices correspond to elements 2, 7, 6 of the array. The median of (2, 7, 6) is a 6 (because 6 is in-between 2 and 7). So our pivot for the first pass is a 6. We then run quicksort as usual:
5, 2, 9, 12, 7, 8, 3, 6 (swap the pivot with the last element)
Move i until it points at 9. j points at 3. Swap them:
5, 2, 3, 12, 7, 8, 9, 6
i now points at 12, j moves until it crosses i and points at 3 (because elements 8, 7, 12 are all larger than the pivot 6). We swap the pivot with the element at i and get:
5, 2, 3, 6, 7, 8, 9, 12
So the first pass split the list into elements < 6, 6 and elements > 6. We now need to recursively run randomized quicksort on sublists 5, 2, 3 and 7, 8, 9, 12. For each sublist, we would again pick three random elements of the sublists and choose a median as a pivot. If the sublist contains only two elements, randomly pick one of the two as the pivot. Finish this example before you start coding randomized quick sort.

5. Hybrid Sort

```
public void hybridSort(Comparable[] array, int lowindex, int highindex);
```

For large lists, quicksort tends to be the fastest general-purpose (comparison) sorting algorithm in practice. However, it runs slower than some of the $\Theta(n^2)$ algorithms on small lists. Since it's a recursive divide and conquer algorithm, it needs to sort many small sublists. You need to design a hybrid sorting algorithm that **combines quicksort with insertion sort** to make quicksort faster. Hint: Run quicksort until the sublists become small (say, when the number of elements in a sublist is 10), and then use insertion sort to sort the small lists. If you want to implement an alternative hybrid sort, discuss it with the instructor to confirm it's a good alternative to the ones proposed above.

For this algorithm, you also need to **design tests and run them on both the randomized quick sort and on hybrid sort** to see if your hybrid sort is faster. Make sure your hybrid sort is fast on all kinds of lists, including random, sorted and inverse sorted lists.
Submit the java file with the tests and a README that describes your hybrid algorithm and the results of the tests.

Submission

Submit project3 to your private github repo by the deadline. Only github submissions are accepted. You need to have at least 5 meaningful commits before the deadline.

Code Style

Starting this project, your code needs to adhere to the code style described in the *CodeStyle.pdf* document on Canvas.

Interactive Code Reviews:

I will invite several students for an interactive code review for this project. Please come prepared to answer any questions about your code. If you are not able to explain your code, you may **not** get **any** credit for it.

Please do **not** copy any code from the web, and do **not** discuss low-level implementation details with anybody apart from the instructor, the TAs and the CS tutors. Sorting-related questions often come up during job interviews; this project is a great chance for you to practice writing these algorithms.