CMPSC 250
Analysis of Algorithms
Fall 2012
Bob Roos

Lab 4
Tues., 25 September 2012
Hand In by 2:45pm Tues., 2 Oct.

I strongly suggest that you create a new subdirectory called "lab4" or something similar and place all your files from this lab in that directory. At the end of the assignment I'll ask you to create a "zip" file to upload to the Sakai drop box.

Hybrid Sorts

A "hybrid" sorting algorithm is one that combines several different sorting algorithms into one. In today's lab you will explore the modification described on page 296, "Cutoff to insertion sort," and in exercise 2.3.25, page 307.

For simplicity we will limit ourselves to arrays of integers.

1. Write a method named "`insertion`" that has three parameters, an `int` array `a` and two `int` values named `lo` and `hi`. The method should perform an insertion sort on array `a` between indices `lo` and `hi`, inclusive. For example, if

   `a` =

8	6	7	1	5	4	2	3
0	1	2	3	4	5	6	7

   then a call to "`insertion(a,2,5)`" should result in:

8	6	**1**	**4**	**5**	**7**	2	3
0	1	2	3	4	5	6	7

   Your code should be a modification of the `sort` method in program `SimpleInsertion.java` in the "Programs/sep19" folder on Sakai. Some modifications are obvious (for instance, using `int` data rather than `String` data); others are less obvious (for instance, you will have to figure out where the "`lo`" and "`hi`" variables have to be used). *Please don't use insertion sort code from some other source*—I want to see a clear resemblance to the `sort` code in `SimpleInsertion`. Try to give a short description of the changes you made in the comments preceding your `insertion` method. (What?? You hadn't planned to include any comments??!! Shame, shame!)

2. Create a short, simple `main` method to test your `insertion` method. I will run this program, so make it easy for me to use! I suggest something like the following:

```
// header comments--name, lab #, description, usage, etc.
public class TestInsertion {
    public static void main(String[] args) {
        // read in array size, either from args or by prompting user
        // create a random int array of the given size
        // read in lo and hi values, either from args or by prompting user
        // print unsorted array
        // call "insertion" with the array and the given lo and hi values
        // print sorted array
    }

    // header comments--see lab handout
    public static void insertion(int[] a, int lo, int hi) {
        // your code here
    }
}
```

3. Modify your test program so that it has a global `int` variable named `count` (i.e., "`public static int count`"). Initialize this to zero just before calling `insertion`. Within method `insertion`, add 1 to `count` each time an array element is compared to another value. Don't include comparisons other than those involving array elements `a[...]`. Print the value of `count` when sorting is finished.

   Use your understanding of how insertion sort works and a few small arrays to verify that it is counting correctly. For example, sorting the array | 5 | 4 | 3 | 2 | 1 | should require exactly 10 comparisons, while sorting | 1 | 2 | 3 | 4 | 5 | should require exactly 4. (It's easy to miss the comparison that causes the innermost loop to terminate.)

4. Download program "`QuickIns.java`." Modify it so that, inside the second `sort` method (the one that has three parameters), it checks to see if the size of the subarray to be sorted is $\leq$ the value of `cutoff` (read the comments at the top of the program to see what this means). Verify that it is sorting correctly (for instance, use a small n and add code to print out the array before and after sorting for small n). Once you're sure it works, comment out the code that prints the arrays and run it with larger values of n.

5. Create a second program named "`QuickNoIns.java`." It is exactly the same as `QuickIns` except there is no call to `insertion`—the quicksort method keeps calling itself recursively all the way down to arrays of sizes 0 and 1. (Of course, you don't need a `cutoff` value for this version.)

6. Determine the best value of `cutoff` where it pays to switch to insertion sort (best in terms of reducing the number of comparisons performed). Explain in an accompanying document how you made this determination and provide supporting evidence in the form of experimental data collected from `QuickIns` and `QuickNoIns`. Graphs or tables showing numbers of comparisons for different values of n and different values of `cutoff` should be provided if possible.

   Your document should also answer the following questions (you might need to study the code and comments in `QuickIns.java` to answer some of them):

   (a) Why is the call to `StdRandom.setSeed` included?

   (b) (Related to previous item:) Why does the program print out the values of `a[3]`, `a[6]`, and `a[9]`?

   (c) Why was the "shuffle" step disabled in the quicksort method?

7. Move up one level from your `lab4` directory and type "`zip -r lastnamelab4.zip lab4`" (if you named it something else, use the something else). Now upload this to your drop box.

I will do the following with your programs:

- I'll run your test program for insertion using small values of n to make sure that it correctly sorts between values of `lo` and `hi` and to make sure that it is correctly counting number of comparisons.
- I'll run `QuickIns` with several different values of `n`, `numexp`, and `cutoff`, then do the same with `QuickNoIns`, and I'll see if the values seem reasonable and seem consistent with your explanation in your accompanying document.
- I'll look at your code, paying attention to correct use of `count`, correct placement and use of call to `insertion` (in `QuickIns`, proper formatting of code, and documentation.