Lab9: Sorting Lab

CSSSKL162: Programming Methodology Skills,

Summary

The purpose of this lab is to practice implementing the sorts we've covered in class. We'll start by building a simple Bubble Sort, which can be accomplished in 9 lines of code (LOC) or less, and then move on to more advanced sorting techniques like the Selection sort and Insertion sort. Also, well make minor improvements to the sorts to obtain better average-case execution times, but it is left as an exercise to the reader to demonstrate that these optimizations will have no effect on the (worst-case) Big O analysis.

Beginning with the Bubble Sort

The Bubble Sort is named due to how its sorting mechanism moves data items around during its sorting procedure. Heavy items "sink" to the bottom while smaller items "bubble" their way to the top by comparing neighbors and checking for ***inversions*** – that is, two elements out of order that require a swap. By comparing successive neighbors, we are guaranteed the max element in the last place after the first pass, so we are able to optimize on this search by only inspecting neighbors in the unsorted part of the array (the ***unsorted partition***). This sort grows the sorted partition by one element (the largest) each pass, and shrinks the unsorted partition accordingly.

(1) Download the provided MyArrayList class, and inside of it declare a method called intBubbleSort()
   a. Note that MyArrayList has 3 arrays as its class fields; int[], char[], and String[]
   b. public void intBubbleSort() //needs no input, as the list is instance data
      i. See the pseudocode in the book or the slides for help on how to implement the bubble sort method
      ii. Implement a helper method called "public void swapInts (int[] intList, int j)" to call from your intBubbleSort() to swap the elements for you

(2) Download the SortDriver.java which contains the main method to test your code.
   a. The arrays get populated with random values in constructor.
   b. Print the list out using toString(); it should appear unsorted.
   c. In driver, uncomment the call to your intBubbleSort () method and output the list, this time in sorted order. Be sure to get the correct output.

(3) Now, change your code so that it sorts an ArrayList of Comparable objects. To do this, you'll need to declare the ArrayList class as "public class MyArrayList implements Comparable <MyArrayList>" and implement the compareTo() method to determine if one object is "less than" a second object.

(4) In your compareTo() use the following logic to compare the objects:

```
 if(this.IntList[0] < other.IntList[0])     return -1;
else if(this.IntList[0] > other.IntList[0])    return 1;
else                        return 0;
```

Explain in your code what does each one of these return values mean? Why 1, or -1, or 0?

(5) Uncomment the driver, and test your compareTo().

(6) Optimizations
   a. Can you reduce the inner loop relative to the outer loop? Try it.
   b. Can you rewrite the outer loop so that it *never* executes additional times if the list is sorted "early"?
      i. For example, the list {3 1 2} will need the outer loop to execute only one time, and not n(==3) times

(7) Now, refer to the method headings in MyArrayList class and implement bubble sort for chars and Strings, as well. Do we need a separate swap() for Strings, since Strings are compared differently than primitive types?

(8) Uncomment the driver, and test these newly created methods too.

The Selection Sort

The selection sort makes progress by growing a sorted partition and shrinking the unsorted remainder. It does so by traversing the unsorted partition looking for the least element in that partition. The first pass simply looks for the least element and swaps that with the element at the head of the list (which is now considered sorted). The second pass would scan the remainder n-1 elements looking for the next minimum element, which then becomes the second element to the right of the head (now a 2-element sorted list), and so on until sorted.

(a) Use the following pseudo code to implement your selection sort inside MyArrayList class :

Construct an outer for loop that traverses the array up to (n-1) element {
   1. Construct an inner for loop that traverses the array up to the last element (n)
   2. Does the inner loop iterator start at 0 or 1? Why?

3. Inside your loop check to see element at array[j] < array[i];
4. Assign the index number for the smallest value to a variable.
5. Keep updating the variable that holds the index for the smallest element as you continue traversing the array
5. When inner loop is completed, you will have the index for the smallest element in the array
6. Now, all you have to do it use the swap() and swap array[smallest index] with array[i]
7. Your loop will now go back to outer loop and will continue its work

(b) Now, use the following pseudo code to implement the two functions that the selection sort will need to iterate:

1. void swap(arr[], int index1, int index2)
2. int findSmallest(arr[], int begin, int end) //returns the index of the smallest element
    i. minIndex = begin; //hint
    ii. for i = start to end

(c) Finally, let's make an improvement to the Selection Sort. In findSmallest, since we're walking the length of the unsorted partition, why not track both the smallest *and* the largest items we've seen? It's a few more compares per iteration, but we know from our Big O series that this won't make the time complexity any worse. In our new version, we may need to change the outermost loop so that it calls findSmallest() and findLargest().

(d) Now compare the two sorts. The improvement we made in (3) should speed up the average-case execution of this algorithm, but does this improve the Big O for this algorithm? Build a Big O estimate using the estimation techniques covered in class, tracking the number of compares the new algorithm uses versus the standard selection sort.

InsertionSort

In this section, we will explore a new sorting implementation that grows a sorted partition by one at each step. As we expand our sorted partition to include our "nearest neighbor", our partition becomes unsorted until we put the newest neighbor in the correct spot. So, our strategy will be outlined and developed below by our pseudocode below; if you think you've got a handle on the mechanism, try to implement the code after looking at just the first iteration of the pseudocode below. Look at the second iteration (spoiler alert!) if stumped.

InsertionSort Pseudocode, Iteration 1

Starting with the first node, ending with the last

Get its neighbor (i+1)

While the neighbor is out of place, move it backwards through the sorted partition

Once the neighbor is in the right place, we're (locally) sorted and its time to repeat this process

InsertionSort Implementation

(1) In the same ArrayList class, declare a method called insertionSort();
(2) Implement the above (or below) pseudocode logic in your code.
(3) In your main test driver, change the code so that it invokes the InsertionSort.
    a. Test your code.

InsertionSort Pseudocode, Iteration 2 (Spoiler Alert!)

```
for(int a = 0; a < length – 1; a++)  {//note -1 here since we're dealing with neighbors (a, a+1)

        int current = data[a];

        int hole = a;

        while( hole >0 && data[hole-1] < current ) {  //while "out of place"

                //slide data to the left moving the "hole" left

        }

}
```