**Lab 4**

1. Improve the BubbleSort implementation so that in the best case (which means here that the input is already sorted), the algorithm runs in O(n) time. Explain why your new version works --- in other words, prove that the best case running time of your code is O(n). Call your new Java file BubbleSort1.java.

**Solution**:

My approach to solve this problem is creating a separate method that checks whether a given array is sorted. So, this logic is invoked before we begin our sorting routing. The method body given below (isSorted()), shows how it’s implemented. Inside a while loop, we compare adjacent elements beginning from the right most and try to see if they are in sorted order. If adjacent elements are not in proper sorted order, a false is returned immediately. Otherwise, we continue to check all the remaining elements.

**boolean** isSorted(){

**int** index = arr.length - 1;

**while**(index > 0){

**if**(arr[index - 1] > arr[index]){

**return** **false**;

}

index--;

}

**return** **true**;

}

Assuming that the probability of adjacent elements being in sorted order is 1/2, the average running time of this loop is 2. (Let Y be random variable denoting the number of iterations or loop runs.). Worst case running time of this loop is O(n), and best case is O(1). Therefore, addition of this logic improves the bubble sort algorithm as it saves running O(n2) iterations for some of the arrays that are already sorted.

E[Y] = ≈ 2

The additional cost of including this logic is worth it if there is some chance of having sorted arrays as inputs.

1. Recall that in BubbleSort, at the end of the first pass through the outer loop, the largest element of the array is in its final sorted position. After the next pass, the next largest element is in its final sorted position. After the ith pass (i=0,1,2,…), the largest, second largest,…, i+1st largest elements are in their final sorted position. Use this observation to cut the running time of BubbleSort in half. Implement your solution in code, and prove that you have improved the running time in this way. Call your new Java file, which contains the improvements from this problem and the previous problem, BubbleSort2.java.

**Solution**:

As clearly explained in the question itself, i number of elements counting from the right most element to be in sorted order after the ith iteration of the outer loop has executed. So, we don’t need to check them in the inner loop. The code given below improves the BubbleSort algorithm in that respect.

**for**(**int** i = 0; i < len; ++i) {

**for**(**int** j = 0; j < len-1-i; ++j) {

**if**(arr[j]> arr[j+1]){

swap(j,j+1);

}

}

}

As explained in the lecture slide for lesson 4, this algorithm cuts the running time of the original BubbleSort algorithm by half.

1. In this lab folder, I have given you an environment for testing sorting routines. Insert into this environment the original BubbleSort file along with your new BubbleSort1 and BubbleSort2 classes, and run the SortTester class. What are the results? Are the results what you expected? Explain why the running times turned out in the way that they did.

**Solution**:

I modified the given BubbleSort.java code as described in (1) and (2) above to create two additional java classes, namely BubbleSort1.java and BubbleSort2.java. Then I run the the SortTester classes after including the names of these new classes in sorters\_to\_be\_run.txt. The results I found from performing several such runs support the claims that I put forward in my solutions for questions (1) and (2) above.