Assignment 9 – Sorting

Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail.

1. Let's sort using a method not discussed in class. Suppose you have *n* data values in in array *A*. Declare an array called *Count*. Look at the value in *A[i]*. Count the number of items in *A* that are smaller than the value in *A[i]*. Assign that result to *count[i]*. Declare an output array *Output*. Assign *Output[count[i]] = A[i]*. Think about what the size of *Output* needs to be. Is it *n* or something else? Write a method to sort based on this strategy.

ANSWER:

```
public int[] countSort(int[] A) {
int[] Count = new int[A.size()]; // set Count empty of same size as A (assume int's; can do double, float)

//!!!
for(int j = 0; j < A.size(); j++){
   int num_less = 0;
   for(int k = 0; k < A.size(); k++) {
      if(A[k] < A[j]) {
         num_less++; // increment value for each item less than current
      }
   }
   Count[j] = num_less; // assign value to Count
}
//!!!

int[] Output = new int[A.size()]
for(int j = 0; j < A.size(); j++) {
   Output[Count[j]] = A[j];
}
return Output;
}
```

Output is of size n, since Output is the ordered version of the array A and A is of size n.

2. Analyze the cost of the sort you wrote in the previous problem. What is the impact of random, ordered, or reverse ordered data?

The cost of the sort in the previous problem is the sum of the cost of two operations. The first populates an array Count, (this is the region of code between //!!! and //!!!). This operation takes O(n) to populate the count array * O(n) to compare values in array A. Therefore the cost of this is O(n^2). Then the cost of assigning values to the Output array takes O(n) time. O(n^2) + O(n) = O(n^2), therefore a sort of this type takes O(n^2) time.

It should be noted that this is a limited search algorithm, because it requires all elements in the array to be distinct (i.e. does not work if there are two values that are equal).

Using this algorithm there is no performance enhancement for ordered data vs. random data vs. reverse ordered data. If you had ordered data, you could perform a binary search rather than a linear search in the region between //!!! and //!!!, which would reduce the complexity to O(nlgn), but if you already had ordered or reverse ordered data, you would not need/want to do a countSort algorithm.

3. How many comparisons are necessary to find the largest and smallest of a set of n distinct elements? Do not assume the answer must involve sorting. It could but does not need to do so. Try to be as efficient as you can.

The cost to find the largest and smallest of a set of n distinct elements requires 2n-3 comparisons. The method would require you to first compare the first and second values. The larger gets set to a variable called largest, and the smaller gets assigned to a variable called smallest. You can then traverse the list (so the next item to compare to would the third item). Each item in the list gets compared to the smallest value (and if it is smaller that value is assigned to smallest) and the largest value (and if it is larger that value is assigned to largest). This assumes that the list is not sorted in any way and only takes 2n-3 comparisons. This solution does not use sorting, but is faster than a sorting algorithm (O(n)).