

CSC 112 Homework 3

Due Date: 5:15PM, 10/22/2009

Name _____

1. Trace (by showing the progress of the array as each item is put into place) both the selection sort and insertion sort algorithms on the following array. Implemented versions of the algorithms can be found on the last page of the homework.

[1 | 4 | 3 | 9]

Trace for selection sort:

Trace for insertion sort:

What are the total number of item-to-item comparisons required for each algorithm on the given array dataset? Show your work (or at least point back to your trace).

2. Augment insertion sort with an additional check done before the actual insertion sort algorithm to handle this case: if the array is reverse sorted, reverse the data in the array and return immediately.

3. The selection sort algorithm repeats the same process (written below) over and over after initially setting the “target” position (where the minimum element should go) to be 0:

- Find the smallest element of the unsorted part of the array
- Put that element in the “target” position
- Update the target position by +1

This process stops when we are down to one element, which by default is already in the target position. This thought process suggests there is a natural **recursive** implementation of selection sort. Change the *selectionSort* code (not *minimumIndex* or *swap*) to be recursive and explain why you chose your recursive approach. You should keep the current selectionsort interface to the user: a *selectionSort* function taking as parameters (int* array, int size). In preserving the interface, your best bet is to use the *selectionSort* function to call a helper function which is the actual recursive component.

4. A sorting algorithm is called a *stable sorting algorithm* if it ensures the following property about the data: if there is a tie in the data, when sorted, the tied values will appear in the same physical order they were in as part of the original array. Indicate whether or not you believe the selection sort and insertion sort algorithms outlined on the following pages are *stable* and support your answer. If your answer is “no, sort X is not stable”, the best argument is to provide a counter-example case which shows the sort doesn’t preserve this property. If your answer is “yes, sort X is stable”, attempt to provide a general answer that shows the stable sorting property is never violated given the steps executed by the algorithm.

Selection Sort:

```
int minimumIndex(int* array, int first, int last)
{
    int j;
    int minIndex = first;
    for (j = first + 1; j <= last; j = j + 1)
    {
        if (array[j] < array[minIndex]) minIndex = j;
    }
    return minIndex;
}

void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void selectionSort(int* array, int size)
{
    int index, k;
    for (k = 0; k < (size-1); k++)
    {
        index = minimumIndex(array, k, size-1);
        swap(array, k, index);
    }
}
```

Insertion Sort:

```
void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void insertionSort(int* array, int size)
{
    int k;
    int temp;
    int position;

    // first item already "sorted"
    // start with second (k = 1 instead of 0)
    for (k = 1; k < size; k++)
    {
        temp = array[k];
        position = k;

        // compare item against already sorted front items,
        // moving past those bigger than you until hit front
        // of array or hit one smaller than you
        while ((position > 0) && (array[position-1] > temp))
        {
            // push them back as you move forwards
            array[position] = array[position-1];
            position = position - 1;
        }
        // insert yourself at the right spot
        array[position] = temp;
    }
}
```