COE428 Lab 3: Sorting

1. IMPORTANT: Two week lab

Note that you have two weeks to complete this lab. This lab must be submitted at least 48 hours before the beginning of your next lab period.

2. Prelab preparation

Before coming to the lab you should:

- Read the lab. Try to prepare any questions you may have about the lab.
- Refer to **Lab Guide**.
- Create your lab directory for lab3. (i.e. use **mkdir lab3** within your coe428 directory.)
- Change to your coe428/lab3 and unzip the lab3.zip file with the command:
  unzip /home/courses/coe428/lab3/lab3.zip
- Ensure that you have downloaded properly by entering the command: **make**
  No errors should be reported..

3. Requirements

The requirements to complete the lab are summarized below.

1. Implement *InsertionSort* using the `mySort()` function in the file: `insertionSort.c`.
2. Implement *MergeSort* using the `mySort()` function in the file `mergeSort.c`.
3. Edit your `README` file to include:
    1. A brief summary of what you accomplished and what (if any) parts you did not complete or bugs that you are aware of but have not fixed.
    2. Analysis (including equations for number of moves, swaps and compares as a function of *n*) for the best-, average- and worst-case behaviors of *InsertionSort* and *MergeSort*,

4. Introduction

This lab revisits *sorting* algorithms. You will:

1. Implement and analyze two sort algorithms:
    a. *InsertionSort*
    b. *MergeSort*
2. Using programming conventions (or patterns) that allow the performance of your implementations of the algorithms to be measured.
3. Use the `make` development tool to keep your project up to date.

5. Theory

5.1. The Insertion Sort Algorithm

You are required to analyze and implement another sorting strategy: *InsertionSort*. The *InsertionSort* algorithm applied to a deck of cards can be described as:

Step 1:

If there are no cards to sort, then **`Stop`**.

Step 2:

Otherwise, remove the top card from the unsorted pile, figure out where it should go in the sorted pile and insert it there.

Step 3:

Go back to step 1.

The important characteristics of the algorithm are:

- As it runs, it divides the deck into a sorted portion (initially empty) and an unsorted portion (initially the whole deck).
- Each time Step 2 is performed, the sorted portion has one more card and the unsorted portion has one less card.
- `Step 2` is performed *n* times (where *n* is the number of cards in the deck).
- `Step 2` describes how this is done: in this case, get the first unsorted card and insert it into the proper position in the sorted part.

It is useful to compare this algorithm with the similar *SelectionSort* algorithm.

Step 1:

If there are no cards to sort, then **`Stop`**.

Step 2:

Otherwise, find the smallest card, remove it and place it on top of the sorted card pile.

Step 3:

Go back to step 1.

It should be evident that both algorithms share some basic characteristics. Both increase the sorted portion by one each time `Step 2` is performed. They differ only in the way this is done:

- *SelectionSort* has to examine all the cards in the unsorted portion, remove the smallest one and then place it at the end of the sorted portion.
- *InsertionSort* takes the first card from the unsorted portion (i.e. it does not have to scan through the unsorted cards) and inserts it into the sorted portion (and it has to scan the sorted portion to figure out where it should go).

5.2. Implementing sort algorithms with metrics

The algorithms described previously use the *deck-of-cards* metaphor. This may be useful for human understanding of the algorithm, but is incomprehensible to computers.

In this lab, we require that all sort algorithms be implemented in C using a function `mySort` that sorts an array (or a subarray) of `int`'s. The signature for `mySort()` is:

```
void mySort(int data[], unsigned int first, unsigned int last);
```

Note

The signature for `mySort` is in the file `mySort.h`.

An implementation of the `mySort` function must modify the `data` array such that all elements in the range `data[first], data[first+1]...data[last]` are in sorted order.

5.2.1. Example: SelectionSort implementation

The *SelectionSort* algorithm is described in Chapter 2 of *Introduction to Algorithms*, exercises 2.2-2 on page 27. It is **strongly** recommended that you review the idea of *SelectionSort*.

5.2.1.1. The initial implementation

For reference, the initial implementation of *SelectionSort* is shown below:

```
void mySort(int array[], unsigned int first, unsigned int last)
{
   int i;
   /* Step 1: Is there nothing to sort? */
   while (first < last)
      /* Step 2: Swap... */
      for(i = first+1; i <= last; i++) {
         /* Find smallest one in rest of array */
         if(array[first] > array[i])) {
         /Step 2..continued...swap them */
            int tmp;
            tmp = array[first]
            array[first] = array[i];
            array[i] = tmp;
         }
         first++;
      }
return;
```

5.2.2. Using metrics

All sort algorithms require that elements in the array to be sorted can be compared with something else of the same data type. Indeed, the number of times an algorithm needs to compare a data item with something else (of the same type) is fundamental metric (i.e. *measure-of-performance* or *benchmark*) that is often used to evaluate different sort algorithms.

You are provided with a framework that counts the total number of times that data elements are compared. Although you do not need to write (or even understand) this *metrics* framework, you **must** respect its contract.

In order to respect the contact so that the total number of comparisons is tracked, it is forbidden to compare a data element with something else using C comparison operators (such as `==`, `<`, `>`, etc.) Instead, you must use the `myCompare()` function.

Note: Replacing “(data[j]<foo)” with myCompare()

Consider the following code that directly compares a data element with something else:

```
if (data[j] < foo) {
```

The `myCompare(x, y)` function returns a negative number if and only if `x < y`. Hence, the previous code can

transformed to:

```
if (myCompare(data[i], foo) < 0) {
```

In addition to comparing elements in the data array, all sort algorithms also have to change the positions of individual data elements. In order to be able to keep count of these operations, we require that elements be moved using the `myCopy()` or `mySwap()` functions.

Note: Replacing "data[i] = tmp" with myCopy()

For example, instead of writing something like:

```
   data[i] = tmp;
```

you would use the `myCopy()`:

```
   myCopy(&tmp, &data[i]);
```

Note: Using mySwap() to interchange elements

`Step 2` of *SelectionSort* was initially implemented in C (as shown previously) with:

```
   int tmp;
   tmp = array[first]
   array[first] = array[i];
   array[i] = tmp;
```

The effect of this code was to interchange (or swap) the elements `array[first]` and `array[i]`. The same result can be achieved with the `mySwap()` function:

```
   mySwap(&array[first], &array[i]);
```

5.3. Using make

Unlike previously labs where you had to type in the compile and link commands, you can perform these steps efficiently with the single command `make`.

6. Suggested approach

There are several requirements in the lab. A suggested approach is outlined below.

6.1. Use make

The files furnished to you include *stub* implementations of `mySort()` and a real `main()` function.

By default, the command `make` ensures that the commands `insertionSort` and `mergeSort` exist and correspond to the most recent modifications to source code files. The command `make` works "right out of the box" (which you should have verified in your prelab).

Although the "out-of-the-box" commands are compiled without error, they do not really work! (Actually, they do work if they are asked to sort *nothing*. Try it! (i.e. the command `insertionSort < /dev/null` will "sort nothing", write "nothing" to *stdout* and write statistics to *stderr* indicating that no comparisons, move, or swaps were needed to "sort nothing".)

6.2. Implement InsertionSort without metrics

As described previously, the *InsertionSort* sort algorithm is quite similar to *SelectionSort*.

6.3. Modify InsertionSort to include metrics

Once you are reasonably confident that the command `insertionSort` works, modify the implementation so that the metric functions are used.

The most important measurement is the number of comparisons; start by using `myCompare()` instead of direct comparisons. When this works, the output should still be sorted and the statistics (displayed to *stderr*) will indicate the number of comparisons required.

Finally, replace data element movements with `myCopy()` and/or `mySwap()`. The statistics should now reflect the number of copies/swaps performed by the algorithm.

6.4. Complete InsertionSort theoretical analysis

You should now attempt to complete a theoretical analysis of the number of compares/copies/swaps that your implementation performs for worst-, best- and average-case inputs of *n*. Ideally, you should develop an equation (a function of *n*) that gives these metrics.

6.5. Implement and analyze MergeSort

You should aim to complete the implementation and analysis of *InsertionSort* in the first week of the lab.

MergeSort is a bit trickier to implement and analyze. Note, in particular, that the algorithm itself is actually *very easy* to implement in C (although, of course, it does use recursion). The tricky part is the *merge* sub-algorithm.

Although the *merge* is not itself recursive, it does require at least one temporary array and it does require a fair amount of copying of elements to an from the temporary. Note, however, that it is **not** necessary to use dynamic memory allocation functions (such as `malloc()`); you can simply declare a local variable for your temporary array as (for example):

```
int temp[MAX_SIZE_N_TO_SORT};
```

Submit your lab

1. Go to your **coe428** directory
2. Zip your **lab3** directory by using the following command:
   zip -r lab3.zip lab3/

3. Submit the lab3.zip file using the following command:

```
submit   coe428   lab3   lab3.zip
```

by Ken Clowes, revised by Ivan Lee, revised by Olivia Das