Programming Assignment 3

COP3502 Computer Science I – Fall 2020

Overview

This assignment is intended to make you implement several different sorting algorithms, and the means to analyze their performance. **This assignment is in two parts: a program, and an analysis.**

Details

You've been given a scattered list of small monsters present in the Knightrola region. You need to evaluate six means of comparison-based sort to get these monsters in order.

You need to implement six sorts for the monster structure:

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Merge sort
- Merge sort, switching to insertion sort at $n \leq 25$

Data Structures and Prototypes

We've given you two template C files that will help you with implementation.

- **integer-sorts-template.c**: Contains implementation code for bubble, selection, and quick sorts against integer arrays. ***IMPLEMENT MERGE SORT AND INSERTION SORT HERE FIRST, AND GET THEM WORKING, TO DEVELOP YOUR UNDERSTANDING OF THE ALGORITHMS.***
- **monster-sorts-template.c**: Contains the shell for sorting monsters via all six sorts. You may use your code from **integer-sorts-template.c** directly, modifying it as necessary to work on the more complex data structures. Don't change the structures, the function headers, or any of the helper functions.

So your order of operations should look like this:

- Read and understand the attached documentation for insertion sort and merge sort.
- Implement insertion sort, merge sort, and merge-insertion sort against integers, within your own copy of **integer-sorts-template.c**.
  - Implement the six sorts against monsters, within your own copy of **monster-sorts-template.c**.
    - Implement compare_monsters() **first** to avoid implementing everything twice!
    - Remember to use memmove() instead of memcpy() when structures can overlap. (This will come up in insertion sort.)

You aren't required to deal with file input, file output or complex memory management in this assignment (though you will need to do some minor memory allocation for merge sort). This is intentional.

Once you have everything in **monster-sorts-template** working, the output to the terminal should look something like the sample output you've been provided. ***The list is randomized, so it will not be exactly like what you see there.***

The New Sorts

Merge Sort

In merge sort, you repeatedly sort and merge the smallest sublists of a list, working your way up to sorting and merging the entire list. You'll implement the basic recursive version, which works as follows:

Merge sort:

- **Recursive merge sort** the entire list.

Recursive merge sort:

- If the size of the list is one, it's already sorted.
- Otherwise:
    - **Recursive merge sort** the front half of the list and the back half of the list.
    - **Merge** the now-sorted front and back halves of the list.

Merge adjacent sub-lists:

- Create a temporary list large enough to hold all the elements in both lists. Set a pointer to the first element in both lists, and to the first space in the temporary list.
- Iterate until you run out of elements in both lists:
    - Look at the current element in both lists.
    - Copy the smaller one into the temporary list. (If you're out of elements in one list, use the other.)
    - Increment the temporary list pointer, and the list pointer for the list you copied from.
- Copy the temporary list back into the lists you were merging, writing over both. (If the lists you were merging were not adjacent, this will cause bad things to happen.)

Insertion Sort

In insertion sort, you divide the list into the elements you've already sorted and the elements you haven't yet, iterate over the elements you haven't yet, and insert them into the sorted sub-list. The basic array version, which you'll implement, works as follows:

- Iterate i from the second to the last element of the list. i is always the first element you *have not yet sorted*.
    - Grab element i.
    - Iterate j from the front of the list, until you either reach i or find an element with a value higher than element i. You are using j to look through *your already-sorted list.*
    - If you reached i, do nothing. Otherwise:
        - Move every element *to the right* (inclusive) of your final element j and *to the left* (exclusive) of element i right by one, "squashing" element i and creating an empty space before element j.
        - Put your copied element i in the empty space.
    - Increment i and keep going.

Cleanup

Specific Requirements

- You do not need to comment line by line, but comment every function and every "paragraph" of code.
- You don't have to hold to any particular indentation standard, but you must indent and you must do so consistently within your own code.
- You may not use global variables.

Analysis, Reflection and Submission

Name your file **cop3502-as3-<yourlname>-<yourfname>.c**. For example, mine will be named **cop3502-as3-gerber-matthew.c**.

Along with your C file, submit a short report in PDF format answering:

- Did the results of your program confirm what you already knew or suspected about these sorting algorithms? How?
- What differences did you see between and among the slow (bubble, selection, insertion) and fast (quick, merge, merge-insertion) algorithms?