Lab 09: Improving Bubble Sort

CSci 515 Spring 2015

<2015-02-18 Wed>

Dates:

Due: In Lab, Wednesday April 1, by 4:05 pm (lab end time)

Objectives

- Practice using arrays.
- Become familiar with details of sorting algorithms.
- Learn more about analyzing the time complexity of algorithms.

Description

There are several improvements / tweaks that can be done to the basic bubble sort sorting algorithm we looked at last week that improve it somewhat. Start this lab by copying the final version of the bubble sort (P03-bubble-sort.cpp) from last weeks lectures. Then make the following two improvements to the basic bubble sort.

First of all, the best case behavior of the basic bubble sort is $O(n^2)$. Basically, even if the values are already sorted, as currently implemented, the outer and inner loops will both always run approximately n times. However, it is unnecessary to continue running the outer loop if/when the array of values has become sorted. An easy way to determine if the values are sorted is to keep track in the inner loop of weather or not any swaps occur. In other words in the inner loop simply keep a flag or count of the number of swaps. If it ever happens that no swaps occur, then it must mean that the values are sorted, and we can stop. In the best case, after adding this improvement, if you give bubble sort an already sorted list, it will simply go through the inner loop 1 time, see the values are sorted, and then stop.

For your second improvement you will perform something related, but a little bit trickier. It can happen that large portions of the end of the array have become sorted during the bubble sort. Our current implementation does take advantage of the fact that on the first pass the largest item will be bubbled up to the last position, so it only performs the necessary iterations in the inner loop keeping track of which items have been sorted. But, as we did before, we can try and keep track of the location where the last swap occurs during the inner loop. For example, let `values[idx]` and `values[idx+1]` be the last two values swapped on a pass. Then surely the values `values[idx+1]` to `values[size-1]` are already in their final positions. In the next pass we only need to consider numbers `values[0]` to `values[idx-1]`. If you keep track of the last swap location, you can use this to be the maximum location to iterate up to on the next pass.

Perform the following tasks:

1. Implement the first optimization described. Start by copying the

first version of the `bubbleSort()` function to a new function named `bubbleSort2()`. In your `bubbleSort2()` function, implement the optimization that causes the sorting to stop anytime the inner loop detects it makes a pass without any swaps occurring. As a hint, and to prepare to do the next task, you might want to try counting the number of swaps that occur, and/or keep track of the last item swapped. You would start this count/location out at 0 in both cases, and if it is 0 after the inner loop iteration, it means no swaps occurred.

1. Implement the second optimization described. Start by copying

the clean first version of the `bubbleSort()` function to a new function named `bubbleSort3()` (don't use your modification for part 1 here, do this optimization from scratch). Implement the idea that you keep track of the location where the last swap occurs, and on the next pass in the inner loop you only bubble up to this location. If you check when this location becomes 0, to get out of the outer loop you will then also have the equivalent of the first optimization you worked on.

In your `main` function, show an example of calling your versions of the `bubbleSort()` functions. To test that your optimizations are working, demonstrate passing in a small, already sorted array, and check using the debugger that the function is performing as you expect.

NOTE: Now that our programs have more functions than just the `main()` function, the use of the function headers becomes meaningful and required. Make sure that all of your functions have function headers preceding them that document the purpose of the functions, and the input parameters and return value of the function.

Lab Submission

An eCollege dropbox has been created for this lab. You should upload your version of the lab by the end of lab time to the eCollege dropbox named `Lab 09 Improving Bubble Sort`. Work submitted by the end of lab will be considered, but after the lab ends you may no longer submit work, so make sure you submit your best effort by the lab end time in order to receive credit.

Requirements and Grading Rubrics

Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. 0 if not satisfied.

2. 50+ pts. Your program must have the required 2 named function, that accepts the required input parameters and return the required values (if any).

3. 50+ pts. The functions must implement the described improvements to the bubble sort algorithm and work.

Program Style

Your programs must conform to the style and formatting guidelines given for this course. The following is a list of the guidelines that are required for the lab to be submitted this week.

1. The file header for the file with your name and program information and the function header for your main function must be present, and filled out correctly.

2. A function header must be present for all functions you define. You must document the purpose, input parameters and return values of all functions. Your function headers must be formatted exactly as shown in the style guidelines for the class.

3. You must indent your code correctly and have no embedded tabs in your source code. (Don't forget about the Visual Studio Format Selection command).

4. You must not have any statements that are hacks in order to keep your terminal from closing when your program exits (e.g. no calls to system()).

5. You must have a single space before and after each binary operator.

6. You must have a single blank line after the end of your declaration of variables at the top of a function, before the first code statement.

7. You must have a single blank space after , and ; operators used as a separator in lists of variables, parameters or other control structures.

8. You must have opening `{` and closing `}` for control statement blocks on their own line, indented correctly for the level of the control statement block.

9. All control statement blocks (if, for, while, etc.) must have `{ }` enclosing them, even when they are not strictly necessary (when there is only 1 statement in the block).

10. You should attempt to use meaningful variable and function names in your program, for program clarity. Of course, when required, you must name functions, parameters and variables as specified in the assignments. Variable and function names must conform to correct `camelCaseNameingConvention` .

Failure to conform to any of these formatting and programming practice guidelines for this lab will result in at least 1/3 of the points (33) for the assignment being removed for each guideline that is not followed (up to 3 before getting a 0 for the assignment). Failure to follow other class/textbook programming guidelines may result in a loss of points, especially for those programming practices given in our Deitel textbook that have been in our required reading so far.