Lab: Sorting by Insertion

CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
April 5, 2016

In this lab, we’ll study an algorithm for sorting an array and a mechanism for visually depicting the progress of the algorithm.

Pairs

- Anna Blindermann and Colin Greenman
- Gemma Nash and Ying Zhang
- Ella Nicolson and Dennis Chan
- Jong Hoon Bae and Sanjay Sudhir
- Faizaan Ali and Saung Thuya
- Mattori Birnbaum and Zachary Susag
- Tanner Tufto and Yuyin Sun
- Cory McCartan and Erhaan Ahmad
- Lilly Webster and Jae Hong Shin
- Lex Martin and Tyler Williams
- Sophie Gaschott and Josh Lavin

Eli Salm will take the place of anyone who is absent.

Exercises

Suppose that we have an array \( \text{arr} \) of \( \text{ARRAY} \text{SIZE} \) integer values, and that we have already confirmed that all but the last of the elements of the array are arranged from least to greatest, so that, for any indices \( i \) and \( j \), where \( 0 \leq i < j < \text{ARRAY} \text{SIZE} - 1 \), \( \text{arr}[i] \leq \text{arr}[j] \). The array is almost sorted, needing only to have the element at position \( \text{ARRAY} \text{SIZE} - 1 \) placed correctly among the other elements.

One way to do this efficiently is to examine the elements in the sorted part of the array, beginning with the one at position \( \text{ARRAY} \text{SIZE} - 1 \) and moving through successively lower-numbered positions until we either find an element that is less than or equal to the one we are trying to place or reach the beginning of the array without having found such an element. As we go, we can shift each element that is greater than the one we are trying to place to the next higher-numbered position in the array so as to make room for the one we are trying to place. We can do this effectively with a loop.

If we precede the loop with the statement

\[
\text{situend} = \text{arr}[\text{ARRAY} \text{SIZE} - 1];
\]

then we won’t need to worry about losing elements of the array as they are shifted to adjacent positions. We’ll have (in the variable \( \text{situend} \)) a copy of the last element of the array, and as we work our way towards the beginning of the array we can make sure to copy each element that needs to be shifted to the next higher-numbered position before there is any possibility of overwriting it.

Exercise 000: In the case where \( \text{ARRAY} \text{SIZE} \) is 5 and the successive elements of \( \text{arr} \) are 17, 29, 48, 62, and 34, in that order, draw a diagram showing what the array should look like before and after the execution of the loop and indicating which elements of the array will have to be repositioned in order to achieve this result. When the loop exits, what additional statement would one execute so that \( \text{arr} \) would end up as a completely sorted version of the given array?
Exercise 001: Repeat the preceding exercise, but with the initial elements of \texttt{arr} being 28, 33, 47, 61, and 16, in that order.

Exercise 010: Write the \texttt{for}-loop that examines each element of \texttt{arr} in turn, starting with the element in position \texttt{ARRAY\_SIZE} – 2, and shifts it to the next higher-numbered position if it is greater than \texttt{situend}, but executes a \texttt{break} statement if it is not. What additional statement would one execute so that \texttt{arr} would end up as a completely sorted version of the given array, regardless of whether the loop exited because of the \texttt{break} or because the re-entry condition in the loop header failed?

Next, consider the more common situation in which the elements of the array might initially be in any order and our goal is to arrange them in the ascending order described above. We can adapt the single-insertion loop that we just wrote for this purpose if we think of the array at all times as being notionally divided into a sorted segment (initially consisting of just the element in position 0) and an unsorted segment (initially comprising all of the other elements). The single-insertion loop can be used to extract the first element in the unsorted segment and insert it into the sorted segment, thus increasing the size of the sorted segment by 1 and decreasing the size of the unsorted segment. If we repeat this process \texttt{ARRAY\_SIZE} – 1 times, the size of the unsorted segment will be reduced all the way to 0 and the array will be completely sorted.

(In its last iteration, this algorithm will do exactly what the solution to exercise 010 does, because by then the earlier steps of the algorithm will have established the truth of the precondition in that exercise: The elements in positions less than \texttt{ARRAY\_SIZE} – 1 will all have been correctly placed relative to one another, and only the last array element will remain in the unsorted segment.)

Exercise 011: Write a C function that takes as arguments an array \texttt{arr} of \texttt{int} values and the number \texttt{size} of elements in that array, and rearranges the elements of the array so that they are sorted from least to greatest. (The exact postcondition is that, for any indices \texttt{i} and \texttt{j}, where \(0 \leq i < j < \texttt{size}\), \texttt{arr[i]} \leq \texttt{arr[j]}.)

Exercise 100: Test your function by generating an array of size 192 containing randomly selected integers in the range from 0 up to, but not including, 256 and then sorting that array. Confirm that the elements of the array are not sorted before you call the function and are sorted afterwards.

Exercise 101: We can get a visual depiction of the array from exercise 100 by interpreting the elements as grayscale values for \texttt{Pixel}s. Write a C function that takes two arguments, an \texttt{int} value in the range from 0 up to, but not including, 256 and a pointer to a \texttt{Pixel}, and overwrites all three components of the \texttt{Pixel} to which the pointer points with the given \texttt{int} value. This turns the \texttt{Pixel} gray and gives it a brightness level depicting the given \texttt{int}.

Exercise 110: Test your function by writing a C program in which you first declare and initialize a \texttt{Picture} of height 1 and width 192, then (as in exercise 100) generate an array of size 192 containing randomly selected integers in the range from 0 up to, but not including, 256, and finally fill in the first 192 elements of row 0 of the \texttt{Picture} to contain grayscale \texttt{Pixel}s produced by your function from the values in the randomly generated array.

Exercise 111: Adapt your solution to the preceding exercise so that the \texttt{Picture} has a height of 192 rather than 1. Add code to sort the array of integers and to fill in a new row of \texttt{Pixel}s within the \texttt{Picture} at the end of each iteration of the outer loop in the sorting function, so that that row will reflect that data movements within the array. Finally, once the sort is completed, use \texttt{rDisplayPicture} to show the depiction of the
sorting algorithm. How does the notional dividing line between the sorted and unsorted segments of the array show up in the Picture?