Programmeren (FEB22012)

2. Exercise

Deadline for submission: 2011-09-14 23:59 CET

Instructions / bubble sort

In the second lecture you were introduced to insertion sort with complexity $O(n^2)$. In this exercise you will first implement bubble sort for sorting the contents of an array[1]. Then you will compare the running times of bubble sort with those of insertion sort and Matlab's built-in sorting algorithm, construct *computational tests*, and plot the running times of the 3 sorting algorithms with inputs of different sizes of random arrays. Afterwards you should be able to say which of these sorting algorithms is the fastest on average. The last part (#4) of the exercise considers separating the different components in IEEE754 floating point numbers.

The idea of insertion sort presented in the lecture (and in lecture notes) is to iteratively go through every element of an array and put them into the correct places in the currently sorted array (see LN-TT-22012-2 for details). The idea in bubble sort is slightly different: the array will be passed through repetitively, and in each repetition, each pair of elements is compared with each other and swapped if they are in an incorrect order. This traversal of the complete array is repeated until no more swaps occur. For more information about bubble sort and a sample implementation in pseudo-code (which is an abstract, imaginary programming language) can be found at `http://en.wikipedia.org/wiki/Bubble_sort`. For a nice visualization, see `http://www.youtube.com/watch?v=lyZQPjUT5B4`.

Exercise / part #1

1. Implement bubble sort as a matlab function taking as a parameter an array of numbers and returning that array sorted in an increasing order.
2. Implement insertion sort in a similar manner. You can copy-paste the code directly from LN-TT-22012-1, page 8.

After completing the first part you should have the two sorting algorithms performing the same functionality. Try it in the command prompt with e.g. bubbleSort([2 4 5 4]) and insertionSort([2 4 5 4]). Matlab has also a built-in sorting function `sort`. Check its documentation and make sure you know how to use it to sort contents of an array in an ascending order.

Exercise / part #2

Now we want to assess the running times of the different sorting algorithms (bubbleSort, insertionSort, and Matlab's built-in sort) for sorting arrays of sizes from 10 to 1000. For this you should make a script that:

1. Iterates for the input sizes n from 10 to 1000 with a step size of 10 (so n=10, 20, 30, ...).

[1] arrays are called vectors in the Matlab slang

2. For each input size, construct an array of length n consisting of integers randomly generated from the interval (1, n) (e.g. for n=10, a random array could be [2 3 4 1 6 5 7 9 2 3]). See Matlab's built-in function `rand` for this. Then for each of these arrays, assess how *long* it takes to run (by using functions `tic` and `toc`):

   (a) insertionSort

   (b) bubbleSort

   (c) Matlab's built-in sort

3. Plot the running times of the three sorting algorithms in one graph. On the x-axis should be n, and on the y-axis T(n). Plot the series of running times as lines with different colours. Give the axes and the graph meaningful titles, and include a legend in the graph explaining what the three different series represent.

Exercise / part #3

Add to your graph of running times plots of the mathematical functions $f(n) = n^3$, $f(n) = n^2$, $f(n) = n$ and $f(n) = n \log n$. Now by looking at the graph you should have an idea of asymptotic growth rates of the three sorting algorithms. Which complexity class does each sorting algorithm belong to? (just answer question this for yourself, you do not need to include it in the exercise answer)

Exercise / part #4

Write a function that takes as a parameter a floating point number and returns the components of its IEEE 754 representation (sign, fraction, and exponent). Note that each of these component is an unsigned integer (clearer to represent as bit strings). For the fraction, the value to be returned is the series of bits that are the multipliers (0 or 1) of the 2^{-i}'s in the IEEE754 double precision (64-bit) floating point representation.

Hint: check functions `num2hex` and `bitget`.