CAB301 Algorithms and Complexity
Algorithms for Assignment 1

Overview

This document provides brief descriptions of the algorithms you are required to analyse for CAB301 Assignment 1. There are two sorting algorithms. All of them involve manipulating data in arrays. Typically this means that the problem's 'size' corresponds to the array's length. Therefore, your experiments should involve analysing the algorithm's performance for arrays of different lengths and plotting the outcomes. For each length of array you should run several experiments in which the array is filled with different 'random' values and then average the results.

Which Algorithm Should You Analyse?

- You should do one of Algorithms 1 to 2, as per the following allocation scheme, based on the final digit in your student number. (Check carefully. Marks will be deducted for analysing the wrong algorithm.)
  - Do Algorithm 1 if your student number ends with the digit 0-4.
  - Do Algorithm 2 if your student number ends with the digit 5-9.

Algorithm 1: Insertion Sort

The following version of the well-known insertion sorting algorithm is presented by Levitin [p. 161]. (Equivalent descriptions and given by Berman and Paul [p. 42] and Johnsonbaugh and Schaefer [p. 241].)

ALGORITHM *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Levitin notes that inequality $A[j] > v$ is performed $n^2/4$ times on average [p. 162], as do Berman and Paul [Sect. 6.4]. Do your experiments confirm or refute this claim?

Algorithm 2: Bubble Sort (Efficient Version)

You should be able to find a lot of information about this well-known sorting algorithm on the web. Be careful, however, because there are several different versions. Levitin presents an inefficient version of the algorithm [pp. 100–101] and leaves the 'efficient' (or 'modified' or 'improved' or 'better') version as an exercise [Ex. 3.1(9b)]. The difference is that the efficient version can stop iterating earlier than the inefficient one on average. Below is the efficient version of the algorithm that you should analyse. (In fact, even an 'efficient' bubble sort is very *in*efficient compared to other sorting algorithms!)

Algorithm *BetterBubbleSort*$(A[0..n-1])$
//The algorithm sorts array $A[0..n-1]$ by improved bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
$count \leftarrow n-1$ //number of adjacent pairs to be compared
$sflag \leftarrow$ **true** //swap flag
while *sflag* **do**
    $sflag \leftarrow$ **false**
    for $j \leftarrow 0$ **to** $count-1$ **do**
        if $A[j+1] < A[j]$
            swap $A[j]$ and $A[j+1]$
            $sflag \leftarrow$ **true**
    $count \leftarrow count - 1$

The average-case efficiency of versions of this algorithm without the 'swap flag' variable used above—which allows the program to stop as soon as the array is sorted—is usually cited as approximately $n^2/2$ comparisons. (Confirm this by finding references to it on the web or in textbooks.) Do your experiments show that the improved version of bubble sort above does significantly better than this?