CAB301 Algorithms and Complexity

Assignment 1 — Empirical Analysis of an Algorithm

Due date: 15th April 2018

Weight: 30%

Group or individual: Individual

Summary

In this assignment, you will analyse the average-case efficiency of a given algorithm experimentally. First you will identify the basic operation of the algorithm and count the number of times the basic operation is performed by the algorithm, to confirm its predicted order of growth. Then you will measure its actual execution time, to determine whether the implementation introduces additional overheads (or optimisations!) not allowed for in the theoretical analysis. Finally, you must produce a detailed report describing your findings. This must all be done with a high degree of professionalism, as would be required when analysing an algorithm to be used in some critical application.

Sample Assignment

To illustrate the kind of report required for this assignment, a sample report is available on the CAB301 BlackBoard site. The sample report analyses a linked list algorithm. NB: The sample assignment is intended as a guide only. It has several features that differ from your assignment. In particular, it analyses the best-case, worst-case and 'aggregate' efficiency of the algorithm of interest. You are not required to do any of these for your assignment; you are required to analyse your algorithm's average-case efficiency only.

Tasks

To complete this assignment you must submit a written report, your implementation of a given algorithm in C#, C, C++ or Java, and a file detailing the results of your experiments to measure a given algorithm's average-case efficiency. The steps you must perform, and the corresponding (brief) summaries required in your written report, are as follows.

1. You must ensure you understand the algorithm to be analysed.
   - Your report must briefly describe the algorithm.
2. You must ensure you understand the algorithm's predicted (theoretical) average-case efficiency with respect to its 'basic operation'.
   - You must explain clearly the choice of basic operation for the particular algorithm of interest.
   - Your report must summarise the expected time efficiency of the algorithm with respect to the size of its input(s). This should be expressed as the algorithm's predicted average-case efficiency and/or order of growth.
3. You must decide on an appropriate methodology, tools and techniques for performing the experiments.
   - Your report must describe your choice of computing environment. You must also say how you produced test data for the experiments, or chose cases to test, as appropriate.
4. You must implement the given algorithm in C#, C, C++ or Java, and verify its functional correctness.

- Your report must describe your programming language implementation of the given algorithm. You must ensure that the correspondence between features of the algorithm and the program code is clear, to confirm the validity of the experiments. (For instance, implementing a recursive algorithm iteratively would not be acceptable because the time efficiency of the program may be very different from that of the algorithm. Similarly, certain code refactorings or optimisations may influence the experiments in undesirable ways.) The program code should replicate the structure of the algorithm as faithfully as possible.

- Your report must explain how you showed that your program works correctly. (Thorough testing would normally be sufficient, although you may prefer to give a formal proof of correctness.)

5. You must count the number of basic operations performed by the program on a range of input values, and present the results in a form that can be compared easily against the theoretical efficiency predictions. To do this you will need to: devise a way of counting basic operations, typically by incrementing a counter variable at the relevant point(s) in the code; run several tests, using appropriately chosen test data; plot the test outcomes on a graph; and state briefly how your experimental results compare to the predictions.

    - Your report must explain clearly how you counted basic operations, e.g., by highlighting the relevant statements inserted into the program. In particular, it should be easy to see that the method used is accurate with respect to the original algorithm.

    - You must perform enough experiments to produce a clear 'trend' in the outcomes. Your report must explain how you produced test data. Depending on the kind of algorithm involved, you may need to produce sets of 'random' values (so that you can produce average-case results for a particular size of input), or an ordered sequence of test values (so that you can show how the algorithm grows with respect to the input's size). In either case you may choose to create test data manually (which may be very tedious) or automatically (which may require some programming).

    - You must present your experimental results as a graph. NB: You must state clearly how many data points contribute to the line(s) on the graph and what each data point represents. If possible, you should use a graph drawing tool that displays each data point as a distinct symbol.

    - You must state whether or not the experimental results matched the predicted number of operations. If they do *not* match then you must offer some explanation for the discrepancy. (Normally we would expect that counting basic operations produces results that closely match the theoretical predictions, but it is possible that there is some peculiarity of your experimental set-up that skews the results, or even that the theoretical predictions are wrong.)

6. You must measure the execution time for your program on a range of input values, and present the results in a form that can be compared easily against the theoretical efficiency predictions. To do this you will need to: devise an accurate way of measuring execution times, typically by recording the system 'clock' time at relevant points in the code; run several tests, using appropriately chosen test data; plot the test outcomes on a graph; and state briefly how your experimental results compare to the predictions.

    - Your report must explain clearly how you measured execution times, e.g., by showing the relevant test program. (Alternatively, you may even choose to time your program with a stopwatch, although this is unlikely to produce accurate results.) It is often the case that small program fragments execute too quickly to time accurately. Therefore, you may need to time a large number of identical tests and divide the total time by the number of tests to get useful results.

    - You must perform sufficient experiments to produce a clear 'trend' in the outcomes. Your report must make clear how you produced test data (as per the discussion above on counting basic operations).

- You must present your experimental results as a graph. NB: You must state clearly how many data points contribute to the results on the graph and what each data point represents. If possible, you should use a graph drawing tool that displays each data point as a distinct symbol.
- You must state whether or not the experimental results matched the predicted order of growth. It is possible that your measured execution times may not match the prediction due to factors other than the algorithm's behaviour, and you should point this out if this is the case in your experiments. For instance, an algorithm with an anticipated linear growth may produce a slightly convex scatterplot due to operating system and memory management overheads on your computer that are not allowed for in the theoretical analysis. (However, a concave or totally random scatterplot is more likely to be due to errors in your experimental methodology in this case!)

7. You must produce a written report describing all of the above steps.
   - Your report should be prepared to a professional standard and must not include errors in spelling, grammar or typography.
   - You are free to consult any legitimate reference materials such as textbooks, web pages, etc, that can help you complete the assignment. However, you must appropriately acknowledge all such materials used either via citations to a list of references, or using footnotes. (Copying your assignment from another student is *not* a legitimate process to follow, however. Note the comments below concerning plagiarism.)
   - Your report must be organised so that it is easy to read and understand. The main body of the report should summarise what you did and what results you achieved as clearly and succinctly as possible. Supporting evidence, such as program listings or execution traces, should be relegated to appendices so that they do not interrupt the 'flow' of your overall argument.
   - There should be enough information in your report to allow another competent programmer to fully understand your experimental methodology. This does not mean that you should include voluminous listings of programs, execution traces, etc. Instead you should include just the important parts of the code, and brief, precise descriptions of your methodology.
8. You must provide all of the essential information needed for someone else to duplicate your experiments in your submission, including complete copies of program code (because the written report will normally contain code extracts only). As a minimum the electronic submission must include:
   - An electronic copy of your report;
   - The complete source code for your implementation of the algorithm; and
   - The complete source code for the test procedures used in your experiments; and
   - Electronic versions of the data produced by the experiments.

   In all cases, choose descriptive folder and file names.

Algorithms for CAB301 Assignment 1

Overview

There are four algorithms, all of which 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 sufficient 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 4, as per the following allocation scheme, based on the final digit in your student number. (Check carefully. 25% marks will be deducted for analysing the wrong algorithm.)
  - Do Algorithm 1 if your student number ends with the digits 00-24.
  - Do Algorithm 2 if your student number ends with the digits 25-49.
  - Do Algorithm 3 if your student number ends with the digits 50-74.
  - Do Algorithm 4 if your student number ends with the digits 75-99.

Algorithm 1: 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?

Algorithm 2: Minimum and Maximum Numbers

Berman and Paul propose the following algorithm as a way of finding both the largest and smallest numbers in an array [p. 39]. A naïve algorithm to do this would compare every number in the array with the largest and smallest values encountered so far, thus requiring $2n - 2$ comparisons. In the following algorithm the nested '**if**' statements aim to make the algorithm more efficient by reducing the number of comparisons performed on average.

ALGORITHM *MaxMin2*(*A*[0..*n* − 1], *MaxValue*, *MinValue*)

// Finds the maximum and minimum numbers in an array

// Input parameter: An array *A* of *n* numbers, where $n \geq 1$

// Output parameters: The largest and smallest numbers in the given

// array, *MaxValue* and *MinValue*, respectively

MaxValue ← *A*[0]

MinValue ← *A*[0]

for *i* ← 1 **to** *n* − 1 **do**

  if *A*[*i*] > *MaxValue*

    MaxValue ← *A*[*i*]

  else

    if *A*[*i*] < *MinValue*

      MinValue ← *A*[*i*]

Berman and Paul then perform a careful analysis of the algorithm's average-case behaviour to see how much better it is than the worst case [Sect. 6.6]. Unfortunately, they conclude that *MaxMin2* still requires $2n - \ln n - 1$ comparisons on average (where 'ln' denotes the natural logarithm) meaning that its average-case behaviour is little better than its worst-case behaviour. Does your analysis confirm this disappointing outcome?

Algorithm 3: Negatives Before Positives (and Zeroes)

Levitin describes an interesting sorting challenge in which an array of numbers is to be ordered so that all the negative numbers occur first [Ex. 4.2(8)]. The following solution does this by examining the array from both ends.

Algorithm *NegBeforePos*$(A[0..n-1])$
//Puts negative elements before positive (and zeros, if any) in an array
//Input: Array $A[0..n-1]$ of real numbers
//Output: Array $A[0..n-1]$ in which all its negative elements precede nonnegative
$i \leftarrow 0$; $j \leftarrow n-1$
while $i \leq j$ **do** //$i < j$ would suffice
  if $A[i] < 0$ //shrink the unknown section from the left
    $i \leftarrow i+1$
  else //shrink the unknown section from the right
    swap$(A[i], A[j])$
    $j \leftarrow j-1$

In fact, this is a 'two-colour' version of a well-known algorithm called the 'Dutch national flag problem'. The basic operation of interest is usually that of swapping items. A particular solution to the two-colour Dutch flag problem was claimed to require $(n - 1)/4$ swaps in the average case [Colin McMaster, *An Analysis of Algorithms for the Dutch National Flag Problem*, Communications of the ACM, 21(10):842–846, October 1978]. We suspect that Levitin's algorithm above is not as efficient as McMaster's because it does not bother to check if item $A[j]$ needs to be moved, and may thus swap items unnecessarily. Does your analysis confirm this suspicion?

Algorithm 4: Binary Search

Binary search is a well-known algorithm for efficiently finding an item in a previously-sorted array. Levitin presents the following version [Sect. 4.3]. Berman and Paul present a similar one [Sect. 2.6.2], as do Johnsonbaugh and Schaefer [p. 166].

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//        or −1 if there is no such element
l ← 0; r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Levitin observes that the algorithm's *approximate* average-case complexity is $\log_2 n$ in general, which is not much better than its worst case. More specifically he notes that the average complexity is slightly different depending on whether or not the item being searched for is in the array. In the successful case the *approximate* average-case complexity is $\log_2 n - 1$ and in the unsuccessful case it is $\log_2(n + 1)$. (Berman and Paul do a more detailed analysis of their version [Sect. 6.7].) Can you confirm Levitin's claims for both successful and unsuccessful searches?

Assessment Criteria

The specific criteria by which your assignment will be assessed are detailed in the Marking Schema and Feedback Sheet for this assignment. Assessment will be based primarily on your written report. However, if there is some concern about the originality of your work or the quality of your experimental results you may be asked to give a practical demonstration of your program.

Submission Process

Your assignment solution should be submitted electronically via Blackboard before 11:59pm on 15th April 2018. Your submission should be a zip file. The file name should contain your student number and include the following items:

- An electronic copy of your report;
- The complete source code for your implementation of the algorithm; and
- The complete source code for the test procedures used in your experiments; and
- Electronic versions of the data produced by the experiments.