# CAB301 Algorithms and Complexity
# Assignment 2 — Empirical Comparison of Two Algorithms

## Due: Monday, 30th May 2016

## Weight: 40%

## Group or individual: Group of two students

## Summary

In the first assignment you learned how to analyse an algorithm experimentally, by implementing it as an executable program, measuring the program's run-time characteristics, and comparing the results with theoretical predictions for the algorithm. In this assignment you will undertake a similar exercise, but this time the aim is to directly compare two different algorithms that both perform the same function but may have different efficiencies. In particular, you must ensure that both programs are analysed under exactly the same conditions, to ensure that the results are meaningfully comparable. As in the first assignment, you will be required to count the number of basic operations performed, measure execution times, and produce a clear report describing your findings. (An example report is available on the CAB301 BlackBoard site.)

## Tasks

To complete this assignment you must submit a written report, and accompanying evidence, describing the results of your experiments to compare the efficiency of two given algorithms. 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 algorithms to be analysed.
    - Your report must briefly describe the two algorithms.
2. You must define a common basis for meaningful comparison of the two algorithms, in terms of basic operations and the size of inputs.
    - Your report must describe your choice of 'basic operation' applicable to both algorithms.
    - Your report must describe your choice of 'problem size' applicable to both algorithms.
    - Your report must summarise the predicted (theoretical) time efficiency of the two algorithms, as far as possible. For the purposes of this assignment we are interested in average-case efficiency only.
3. You must describe the methodology you used for performing the experiments.
    - Your report must describe your choice of computing environment.
    - Your report must describe your C++ implementation of the given algorithms. You must ensure that the correspondence between features of the algorithms and the corresponding program code is clear, to confirm the validity of the experiments. The program code should replicate the structure of the algorithms as faithfully as possible.
    - Your report must explain how you showed that your programs work correctly. Thorough testing would normally be sufficient.
    - Your report must explain clearly how you produced test data for the experiments, or chose cases to test, as appropriate. Usually this will involve generating random data for different sizes of input. In particular, it is important that both algorithms are tested using the same data, to ensure that the comparative results are meaningful.
    - Your report must explain clearly how you counted basic operations, e.g., by showing the relevant program code for maintaining 'counter' variables. In particular, it should be easy to see that the method used is accurate with respect to the original algorithms.

- Your report must explain clearly how you measured execution times, e.g., by showing the relevant program code augmented with calls to 'clock' or 'time' procedures. Since it is often the case that small program fragments execute too quickly to measure accurately, 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.

4. You must describe the results of your experiments.

   - Your report must *briefly* explain how you proved that your programs work correctly. This would normally be done through testing.

   - Your report must show in detail the results of comparing the number of basic operations performed by the two algorithms for different sizes of inputs. The results should be plotted as one or more graphs which make it easy to see how the two algorithms compare. You must produce enough data points to show a clear 'trend' in the outcomes. It must be clear how many individual data points contributed to the lines shown and how many individual tests contributed to each data point. You must briefly explain what the results reveal about the comparative efficiency of the two algorithms.

   - Your report must show in detail the results of comparing the execution times of the two programs for different sizes of inputs. The results should be plotted as one or more graphs which make it easy to see how the two algorithms compare. You must produce enough data points to show a clear 'trend' in the outcomes. It must be clear how many individual data points contributed to the lines shown and how many individual tests contributed to each data point. You must briefly explain what the results reveal about the comparative efficiency of the two programming language implementations of the algorithms.

5. You must produce a brief 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 one of your classmates 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 results. This does not mean that you should include voluminous listings of programs, execution traces, etc. However, you should include the important parts of the code, and brief, precise descriptions of your experimental methodology.

6. You must provide all of the essential information needed for someone else to duplicate your experiments in electronic form, including complete copies of program code (because the written report will normally contain code extracts only). As a minimum this data must include:

   - An electronic copy of your report;

   - The complete source code for your C++ implementations of the algorithms; and

   - The complete source code for the test procedures used in your experiments.

   Optionally you may also include electronic versions of the data produced by the experiments. In all cases, choose descriptive folder and file names.

# Algorithms

The median of a list of numbers is the value that would be in the middle if the list were sorted. It is one of the most important values in statistics. (Be careful not to confuse the median with the mean, or average, of a set of numbers.) For example, imagine that you are given the following list of numbers: 4, 1, 10, 9, 7, 12, 8, 2, 15. Since there are 9 numbers we assume the 'middle' element is the one at position $k = \lceil 9/2 \rceil = 5$, where the ceiling brackets round up to the nearest integer, if the list was sorted. In other words, the median is the 5th element in the sorted list, which is 8 in this case.

One way of finding the median of an array of elements is the following brute force algorithm. (The algorithm is complicated by the need to allow for the possibility that the median value appears more than once in the array. An even simpler version is possible if the array elements are all unique.)

**ALGORITHM** *BruteForceMedian*($A[0..n-1]$)
// Returns the median value in a given array $A$ of $n$ numbers. This is
// the $k$th element, where $k = \lceil n/2 \rceil$, if the array was sorted.
$k \leftarrow \lceil n/2 \rceil$
**for** $i$ **in** 0 **to** $n-1$ **do**
  *numsmaller* $\leftarrow 0$ // How many elements are smaller than $A[i]$
  *numequal* $\leftarrow 0$ // How many elements are equal to $A[i]$
  **for** $j$ **in** 0 **to** $n-1$ **do**
    **if** $A[j] < A[i]$ **then**
      *numsmaller* $\leftarrow$ *numsmaller* + 1
    **else**
      **if** $A[j] = A[i]$ **then**
        *numequal* $\leftarrow$ *numequal* + 1
  **if** *numsmaller* $< k$ **and** $k \leq$ (*numsmaller* + *numequal*) **then**
    **return** $A[i]$

This algorithm appears to have a quadratic efficiency. (Why?)

However, it is obvious that the problem of finding the median of an array is closely related to that of sorting the array. In fact, finding the median value of an array is a special case of the general problem of finding the $k$th value by size in an array. This is known as the 'selection problem', and a well-known solution to it exploits the principles underlying the Quicksort algorithm. Levitin discusses the general selection problem briefly [Section 5.6, pages 188–189], whereas Berman and Paul explore it in depth [Section 8.5, pages 246–253], as do Johnsonbaugh and Schaefer [Section 6.5, pages 262–267].

To solve the median problem efficiently, we can use the following special case of Johnsonbaugh and Schaefer's version of the selection problem algorithm. There are three separate procedures. The main one simply deals with the special case of an array containing only one item, and otherwise initialises the recursive procedure.

**ALGORITHM** *Median*($A[0..n-1]$)
// Returns the median value in a given array $A$ of $n$ numbers.
**if** $n = 1$ **then**
  **return** $A[0]$
**else**
  *Select*($A$, 0, $\lfloor n/2 \rfloor$, $n-1$) // NB: The third argument is rounded down

The next procedure recurs until the desired index value is reached and then returns the value at that location in the (now partially sorted) array. Note that at each recursive call the array slice of interest, between indices *l* and *h*, is reduced.

```
ALGORITHM Select(A[0..n − 1], l, m, h)
  // Returns the value at index m in array slice A[l..h], if the slice
  // were sorted into nondecreasing order.
  pos ← Partition(A, l, h)
  if pos = m then
    return A[pos]
  if pos > m then
    return Select(A, l, m, pos − 1)
  if pos < m then
    return Select(A, pos + 1, m, h)
```

The remaining work is performed by the *Partition* procedure which is also used in the Quicksort algorithm. (Unlike Quicksort, however, the *Select* procedure only needs to process one 'half' of each partition.) The version below is based on Johnsonbaugh and Schaefer's relatively simple version [Algorithm 6.2.2, page 245]. More complicated versions are presented by Berman and Paul [pages 50–51] and Levitin [page 131].

```
ALGORITHM Partition(A[0..n − 1], l, h)
  // Partitions array slice A[l..h] by moving element A[l] to the position
  // it would have if the array slice was sorted, and by moving all
  // values in the slice smaller than A[l] to earlier positions, and all values
  // larger than or equal to A[l] to later positions. Returns the index at which
  // the 'pivot' element formerly at location A[l] is placed.
  pivotval ← A[l]  // Choose first value in slice as pivot value
  pivotloc ← l     // Location to insert pivot value
  for j in l + 1 to h do
    if A[j] < pivotval then
      pivotloc ← pivotloc + 1
      swap(A[pivotloc], A[j])  // Swap elements around pivot
  swap(A[l], A[pivotloc])  // Put pivot element in place
  return pivotloc
```

The worst-case efficiency of the *Median* algorithm is known to be quadratic [Berman and Paul, page 248; Johnsonbaugh and Schaefer, page 266; Levitin, page 189]. Nevertheless, it can have a much better average-case behaviour if 'good' partitions occur, i.e., those that allow the *Select* procedure to eliminate large slices of the array at each recursive step.

Sophisticated analyses have shown that the average-case efficiency of the *Median* algorithm can be linear, which is much better than our *BruteForceMedian* algorithm above. For instance, Levitin explains briefly why the *Median* algorithm should have a linear efficiency, by analogy with the efficiency argument for Quicksort [page 189]. Berman and Paul do a detailed analysis of their version of the algorithm, using array comparisons in the *Partition* procedure as the basic operation. They show that it belongs to efficiency class $\Theta(n)$ [pages 247–250]. Johnsonbaugh and Schaefer argue that a slightly different version of the algorithm above, using randomly chosen pivot elements, also has linear average-case efficiency [pages 265–266].

Do your experiments confirm the claimed efficiency advantages of *Median* over *BruteForceMedian* in the average case? (When writing your programs and setting up your comparative experiments, note that the *Partition* algorithm rearranges the values in array *A*.)

NB: The two algorithms have different functional behaviour if there are an even number of items in the array, i.e., when there is no 'middle' item. The first algorithm (arbitrarily) returns the value to the left of the midpoint and the second algorithm (arbitrarily) returns the value to the right. (In fact, the usual mathematical solution to this dilemma is to return the average of the middle two values.) For large arrays this small functional difference between the two algorithms has no significant impact on their overall comparative efficiency.

## 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.

## 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

Your assignment solution should be submitted electronically via Blackboard before 11:59 pm on 30th May 2016. Your submission should be a zip file. The file name should contain your student number and include the following items:

- All source code of your algorithm implementations
- A detailed report on your empirical analysis of both algorithms - the structure of your detailed report must be the same with that of the sample assignment
- Statement of completeness including the percentage contribution of each student