IFN 505 – Analysis of Programs

Algorithms for Assignment 2

Overview

This document provides brief descriptions of the algorithms you are required to analyse for Assignment 2. There are five pairs of algorithms listed. In your experiments you should analyse the performance of the two algorithms under identical conditions, so that their efficiencies can be compared meaningfully.

Choice of algorithms

You should analyse a pair of algorithms according to the following allocation scheme. Check carefully—some or all marks will be deducted for not analysing the algorithms allocated.

- Do algorithm pair 1, the *Searching Algorithms*, if your student number ends with the digit 5 or 6.
- Do pair 2, the *Sorting Algorithms*, if your student number ends with the digit 3, 7 or 9.
- Do pair 3, the *Minimum Distance Algorithms*, if your student number ends with the digit 0, 1 or 2.
- Do pair 4, the *Median Algorithms*, if your student number ends with the digit 4 or 8

1. Searching algorithms

In this assignment you will compare the efficiency of two algorithms for finding items in a *sorted* array. The first algorithm is a modified version of the simple sequential or linear search algorithm.

```
ALGORITHM SearchSorted(A[0..n − 1], K)
  // Searches for a given value K in a given array A of n numbers which
  // is sorted into nondecreasing order. The index of the first item in A that
  // matches K is returned, or −1 if there are no such items.
  i ← 0
  while i < n do
    if K < A[i]
      return −1
    else
      if K = A[i]
        return i
      else
        i ← i + 1
  return −1
```

Clearly this algorithm has a linear order of growth with respect to the length n of the array. However, because the array is sorted, the algorithm should perform better than a simple sequential search [Levitin, p. 47][Berman and Paul, p. 34] on average. When search key K is not in the array, a sequential search on unordered data must check all n items. However, the algorithm above can stop as soon as it encounters a value larger than the search key, which will be about half-way through the array on average.

The second algorithm is the well-known binary search 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. 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].) In any case, however, the average efficiency of binary search is predicted to be much better than a sequential search.

To directly compare the efficiency of these two algorithms you must produce appropriate test data. This means creating arrays of values sorted into non-decreasing order, where you know whether or not the search key is in the array. Are the two algorithms comparable if the array contains duplicate items? Also, because the efficiency of searching algorithms is different for successful and unsuccessful searches, you must analyse these two cases separately.

2. Sorting algorithms

In this assignment you will compare the efficiency of two sorting algorithms with different predicted behaviours. The first algorithm is the simple selection sort [Levitin, p. 99].

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

The number of comparisons performed by this algorithm is of order n^2.

The second algorithm is the more sophisticated 'sorting by counting' algorithm. The following version is presented by Levitin [p. 253]. (Johnsonbaugh and Schaefer present a similar version [pp. 258–9].) The algorithm requires us to know in advance the range of numbers that will appear in the array to be sorted, and this range is used as the index to a temporary array *D* of the frequency of occurrence of each value.

ALGORITHM *DistributionCounting*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

Levitin notes that this algorithm is linear in the size *n* of array *A* because it makes just two passes through the array. More precisely, Johnsonbaugh and Schaefer note that the range of values that may appear in the array also affects the algorithm's efficiency. For an array of length *n* whose elements may have *m* distinct values, where $m = u - l + 1$, they note that their version of the algorithm's efficiency is of order $\Theta(m+n)$. This is still linear, however.

Given that Levitin analyses the number of 'passes' through the array, you will need to choose carefully a 'basic operation' that is applicable to both algorithms.

3. Minimum Distance Algorithms

Levitin presents the following algorithm for finding the distance between the two closest elements in an array of numbers [Ex. 1.2(9), p. 18].

Algorithm *MinDistance*$(A[0..n-1])$
//Input: Array $A[0..n-1]$ of numbers
//Output: Minimum distance between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if $i \neq j$ **and** $|A[i] - A[j]| < dmin$
      $dmin \leftarrow |A[i] - A[j]|$
return $dmin$

He then asks if there is a more efficient algorithm to do this task. One possible answer is as follows.

Algorithm *MinDistance2*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance d between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    $temp \leftarrow |A[i] - A[j]|$
    if $temp < dmin$
      $dmin \leftarrow temp$
return $dmin$

This version would appear to be more efficient because it compares the same pair of numbers only once, and avoids recomputing the same expression in the innermost loop. Can your empirical analysis confirm the claim that *MinDistance2* is more efficient than *MinDistance* on average?

Since the optimisation in *MinDistance2* is motivated by a desire to reduce the number of expressions that access elements of array *A*, you may want to use this principle to justify your choice of basic operation. (When counting basic operations for algorithm *MinDistance*, be careful to consider whether or not the second conjunct is evaluated when index *i* equals index *j*.)

Unfortunately, Levitin does not give an analysis of average-case efficiency for either algorithm. Neverthless, it is obvious from inspection that the best- and worst-case behaviours for both algorithms are quadratic with respect to the length of the array. (Why?) Thus their average-case efficiencies must be quadratic as well. (Why?) However, although both algorithms belong to the same efficiency class, your experiments should reveal that algorithm *MinDistance*'s average-case efficiency function has a significantly larger 'constant multiplier' than that of *MinDistance2*.

4. Median 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 kth element, where  k = ⌊ n/2 ⌋, if the array was sorted.
  k ← ⌊ n/2 ⌋
  for i in 0 to n − 1 do
    numsmaller ← 0   // How many elements are smaller than A[i]
    numequal ← 0     // How many elements are equal to A[i]
    for j in 0 to n − 1 do
      if A[j] < A[i] then
        numsmaller ← numsmaller + 1
      else
        if A[j] = A[i] then
          numequal ← numequal + 1
    if numsmaller < k and k ≤ (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, ⌊n/2⌋, 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 is 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.