CAB302 - Assignment 2

Luke Josh, Jason Queen

May 12, 2016

Contents

1 Summary

The purpose of this report is to analyse algorithms yada yada average complexity yada yada c++ yada yada execution time yada yada.

2 Description of the Algorithms

2.1 Brute Force

The brute force median algorithm works by manually checking each value of the array, and seeing if it's position in a sorted array is in the median position of $|k/2|$. It does this by checking each element of the array against every other element of the array, and counting the number of elements that are less than the value, and the number of elements that are greater than the value.

Recalling that the median value of a list of numbers is the one that were to occur in centre value (rounded up for this purpose), we can test if a value is the median value by seeing if it has half the arrays values smaller than it. This can be seen in the psuedocode included in the next section

2.1.1 The Algorithm

Algorithm 1 Brute Force Median

```
function BruteForceMedian($A[0..n-1]$)
    $k \leftarrow \|n/2\|$
    for $i \leftarrow 1$ to $n-1$ do
        $numsmaller \leftarrow 0$
        $numeqal \leftarrow 0$
        for $j \leftarrow 0$ to $n-1$ do
            if $A[j] < A[i]$ then
                $numsmaller \leftarrow numsmaller + 1$
            else
                if $A[j] = A[i]$ then
                    $numequal \leftarrow numequal + 1$
                end if
            end if
        end for
        if $numsmaller < k$ and $k \leq (numsmaller + numequal)$ then
            return $A[i]$
        end if
    end for
end function
```

2.2 Median Selection Algorithm

The selection algorithm for finding the median value of the array borrows much of it's logic from the Quicksort algorithm. The algoritm works by partitioning the array into subarrays, searching for the median value stage by stage, sorting as it goes.

First, the index of the median value is determined - being half the length of the array, here being rounded up. If the array is of length one, this is treated as a special case, and the value is simply returned, otherwise, it begins to partition.

It begins by finding the index of the first element in the array (hereby reffered to as the primary element), if the array were to be sorted. It loops through each value other than the primary value (hereby reffered to as the secondary element), and checks to see whether the secondary value is less than the primary value. If a secondary value is less than a primary value, the secondary value is swapped with the element in front of the primary element (or the next unswapped value after the primary value, if multiple values are found). At the conclusion of the loop, the primary value is swapped with the value in the position that the last secondary element that was swapped. It should be noted that there is no gaurentee that the array will be sorted by this process - it is just ensuring that all values to the left of the first element are lesser, and all values to the right are greater. An example of this process is as follows, on an array $A = [35421]$:

$A = [35421]$ 5 and 4 are both greater than 3, do nothing

$A = [32451]$ $2 < 3$, swap 5 with 2

$A = [32154]$ $1 < 3$, swap 4 with 1

$A = [12354]$ Finally, swap 3 with 1

If the first value happens to be the median value of the array, the algorithm stops. However, if the index returned is not the median index, there are two possibilites:

1. If the index is less than the median index, the same process as above is performed, ignoring the first element of the array.
2. If the index is greater than the median index, the same process as above is performed, ignoring the last element of the array.

This process continues until the value with an index equal to the median index is found

2.2.1 The Algorithm

Algorithm 2 Selecion Median

```
function MEDIAN($A[0..n-1)$)
    if $n = 1$ then
        return $A[0]$
    else
        Select$(A, 0, |n/2|, n-1)$
    end if
end function

function SELECT($A[0..n-1], l, m, h$)
    $pos \leftarrow$ Partition$(A, l, h)$
    if $pos = m$ then
        return $A[pos]$
    end if
    if $pos > m$ then
        return Select$(A, l, m, pos-1)$
    end if
    if $pos < m$ then
        return Select$(A, pos+1, m, h)$
    end if
end function

function PARTITION($A[0..n-1], l, h$)
    $pivotval \leftarrow A[l]$
    $pivotloc \leftarrow l$
    for $j \leftarrow l+1$ to $h$ do
        if $A[j] < pivotval$ then
            $pivotloc \leftarrow pivotloc + 1$
            swap$(A[pivotloc], A[j])$
        end if
        swap$(A[l], A[pivotloc])$
    end for
    return $pivotloc$
end function
```

3 Theoretical Analysis of the Algorithms

3.1 Brute Force Median

3.1.1 Choice of Basic Operations

The operation that best defines the complexity and running time of the brute force median algorithm is the comparison $A[j] < A[i]$. This comparison operation is performed more than any other operation in the algorithm - a minimum of $n-1$ times, and a maximum of $(n-1)^2$ times.

3.1.2 Choice of Problem Size

Words

3.1.3 Average Case Efficiency

The average case efficiency of the algorithm can be derived by considering then number of operations required to determine the median of an arbitrarily sized array. Consider an array $A = [a_1, a_2, ..., a_n]$ with it's median value placed at position k. The algorithm must check each element in the array up to and including $A[k]$ against each element in the array, including itself. Thus, the algorithm must perform $k \cdot n$ comparisons to determine that $A[k]$ is the median value of the array.

To determine the average number of operations for an array of size n, we must consider the pairity of the size of the array, as the algorithm chooses the value to the left of the midpoint when there are an even number of elements. We will first assume that each value in the array has an equivalent chance of being the median. In this case, the average case number of operations to determine the median is average number of operations for each $M = A[k]\ \forall k \in [0, 1, ..., n]$:

$$c_{average} = \frac{\sum_{j=1}^{n} j \cdot n}{n} \tag{1}$$

$$c_{average} = \sum_{j=1}^{n} j \tag{2}$$

$$c_{average} = \frac{n^2 + n}{2} \tag{3}$$

To account for the issue of pairty, consider two sorted arrays, $A = [a_1, a_2, ..., a_n]$ and $B = [a_1, a_2, ..., a_{n+1}$, where n is odd. In each of these arrays, the median value will be the same, $M = a_{floor(\frac{n}{2})}$, and will have to check the same number of elements to find it, however, in B, each element is checked against $n+1$ other elements, as opposed to A's n other elements. We can use this to restate the above value for the average number of comparisons as $c_{oddaverage} = \frac{n^2+n}{2}$, and conversely, $c_{evenaverage} = \frac{n^2}{2}$. These two statements can be combined using a modulo operator to give the true average number of comparisons for an arbitrarily sized array as:

$$c_{average} = \frac{n^2 + (n \mod 2) \cdot n}{2} \tag{4}$$

Which gives the algorithm an average case complexity of $\Theta(n^2)$.

3.2 Selection Median

3.2.1 Choice of Basic Operations

Words

3.2.2 Choice of Problem Size

Words

3.2.3 Average Case Efficiency

Words

4 Methodology, Tools and Techniques

4.1 Programming Environment

4.2 Implementation of the Algorithms

4.3 Generating Test Data and Running the Experiments

5 Experimental Results

5.1 Functional Testing

5.2 Average-Case Number of Basic Operations for an Item in the Set

5.3 Average-Case Number of Basic Operations for an Item not in the Set

5.4 Average-Case Execution Time for an Item in the Set

5.5 Average-Case Execution Time for an Item not in the Set