

# CAB302 - Assignment 2

Luke Josh, Jason Queen

May 23, 2016

## Contents

<b>1</b>	<b>Summary</b>	<b>1</b>
<b>2</b>	<b>Description of the Algorithms</b>	<b>2</b>
2.1	Brute Force . . . . .	2
2.1.1	The Algorithm . . . . .	2
2.2	Median Selection Algorithm . . . . .	2
2.2.1	The Algorithm . . . . .	4
<b>3</b>	<b>Theoretical Analysis of the Algorithms</b>	<b>4</b>
3.1	Choice of Problem Size . . . . .	4
3.2	Brute Force Median . . . . .	4
3.2.1	Choice of Basic Operations . . . . .	4
3.2.2	Average Case Efficiency . . . . .	5
3.3	Selection Median . . . . .	5
3.3.1	Choice of Basic Operations . . . . .	5
3.3.2	Choice of Problem Size . . . . .	5
3.3.3	Average Case Efficiency . . . . .	5
<b>4</b>	<b>Methodology, Tools and Techniques</b>	<b>6</b>
4.1	Programming Environment . . . . .	6
4.2	Implementation of the Algorithms . . . . .	6
4.3	Generating Test Data and Running the Experiments . . . . .	6
<b>5</b>	<b>Experimental Results</b>	<b>6</b>
5.1	Functional Testing . . . . .	6
5.2	Brute Force . . . . .	6
5.2.1	Number of operations . . . . .	6
5.2.2	Computation time . . . . .	6
5.3	Selection . . . . .	6
5.3.1	Number of operations . . . . .	6
5.3.2	Computation time . . . . .	6
5.4	Comparison . . . . .	6

## 1 Summary

The purpose of this report is to analyse and compare the average complexity of the selection median algorithm against a brute force solution. This report uses algorithm analysis techniques to experimentally determine the average case efficiency of both the selection median and brute force median algorithm. The expected theoretical efficiencies of the algorithms are evaluated using mathematical analysis and are contrasted against the computed experimental results. The results for each algorithm are also compared to each other and the most efficient algorithm is determined.

## 2 Description of the Algorithms

### 2.1 Brute Force

The brute force median algorithm works by manually checking each value of the array, and determining if the elements position in a sorted array is in the median position of  $jk=2j$ . 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 element that occurs in center position (rounded up for the purpose of this algorithm), the algorithm tests if a value is the median value by testing if half the elements in the array are less than the value. This can be seen in the pseudocode included in the next section

#### 2.1.1 The Algorithm

---

**Algorithm 1** Brute Force Median

---

```
1: function BRUTEFORCEMEDIAN( $A[0..n-1]$ )
2:    $k \leftarrow \lceil n/2 \rceil$ 
3:   for  $i \leftarrow 1$  to  $n-1$  do
4:      $numsmaller \leftarrow 0$ 
5:      $numequal \leftarrow 0$ 
6:     for  $j \leftarrow 0$  to  $n-1$  do
7:       if  $A[j] < A[i]$  then
8:          $numsmaller \leftarrow numsmaller + 1$ 
9:       else
10:        if  $A[j] = A[i]$  then
11:           $numequal \leftarrow numequal + 1$ 
12:        end if
13:      end if
14:    end for
15:    if  $numsmaller < k$  and  $k \leq (numsmaller + numequal)$  then
16:      return  $A[i]$ 
17:    end if
18:  end for
19: end function
```

---

### 2.2 Median Selection Algorithm

The selection algorithm for finding the median value of the array derives much of its logic from the Quicksort algorithm. The selection algorithm uses a recursive method to determine the median of the array. The recursive function first performs a partition on the array where all elements less than a pivot element are swapped until the pivot element swapped into its correct position. All elements with an index lower than that of the pivot, now also have a value which is less than that of the pivot. If the new position of the pivot is the middle of the array then the value of the pivot is returned as it is the median of the array. This check is the base case of the recursive function. Otherwise, if the position of the pivot is less than or greater than that of the middle index then the recursive function is called again and then performs the partition on either the lower or upper partition of the array respectively. This is performed until the base case is met and the element in the middle of the array is returned. It should be noted that there is no guarantee that the whole array will be completely sorted by this process. However, all elements at a lower index than that of the median will have a value that is lower than that of the median element and all elements with an index greater than that of the median element will have values that are also greater than the value of the median. Furthermore, in contrast to the brute force algorithm, when this algorithm is performed on an even array size, returns the value which is towards the lower section of the sorted array. 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 possibilities:

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

---

```
1: function MEDIAN( $A[0..n-1]$ )
2:   if  $n = 1$  then
3:     return  $A[0]$ 
4:   else
5:      $\text{Select}(A, 0, \lfloor n/2 \rfloor, n-1)$ 
6:   end if
7: end function
8:
9: function SELECT( $A[0..n-1], l, m, h$ )
10:   $pos \leftarrow \text{Partition}(A, l, h)$ 
11:  if  $pos = m$  then
12:    return  $A[pos]$ 
13:  end if
14:  if  $pos > m$  then
15:    return  $\text{Select}(A, l, m, pos-1)$ 
16:  end if
17:  if  $pos < m$  then
18:    return  $\text{Select}(A, pos+1, m, h)$ 
19:  end if
20: end function
21:
22: function PARTITION( $A[0..n-1], l, h$ )
23:   $pivotval \leftarrow A[l]$ 
24:   $pivotloc \leftarrow l$ 
25:  for  $j \leftarrow l+1$  to  $h$  do
26:    if  $A[j] < pivotval$  then
27:       $pivotloc \leftarrow pivotloc + 1$ 
28:      swap( $A[pivotloc], A[j]$ )
29:    end if
30:  swap( $A[l], A[pivotloc]$ )
31: end for
32: return  $pivotloc$ 
33: end function
```

---

## 3 Theoretical Analysis of the Algorithms

### 3.1 Choice of Problem Size

The algorithms are tested using randomly generated arrays of sizes 1 through to 999 in steps of 3. Additionally, the test for each array size is repeated 2000 times in order to normalize the results.

The upper limit was chosen as array sizes greater than 1000 have brute force time complexities which are significantly large. This limits the number of averages that can be taken for each array size as the total time to run all tests increases to a value that is impractical to perform repeatedly.

The odd step size of 3 is chosen so that tests are performed for both odd and even array sizes.

### 3.2 Brute Force Median

#### 3.2.1 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.

### 3.2.2 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  $A = [a_1, a_2, \dots, a_n]$  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 pairty 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  $M = A[k] \forall k \in [0, 1, \dots, n]$ :

$$c_{average} = \frac{\sum_{j=1}^n j \cdot n}{n} \quad (1)$$

$$c_{average} = \sum_{j=1}^n j \quad (2)$$

$$c_{average} = \frac{n^2 + n}{2} \quad (3)$$

To account for the issue of pairty, consider two sorted arrays,  $A = [a_1, a_2, \dots, a_n]$  and  $B = [a_1, a_2, \dots, a_{n+1}]$ , where  $n$  is odd. In each of these arrays, the median value will be the same,  $M = a_{\lfloor \frac{n}{2} \rfloor}$ , 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:

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

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

## 3.3 Selection Median

### 3.3.1 Choice of Basic Operations

The operation that best defines the complexity and running time of the selection median algorithm is the comparison  $A[j] < pivot_{val}$  which is performed by the partition sort logic borrowed from the Quicksort algorithm. 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.

### 3.3.2 Average Case Efficiency

Words

## 4 Methodology, Tools and Techniques

### 4.1 Programming Environment

Something something compile command something something ram something something GHz something something ubuntu 14.04 something something.

## 4.2 Implementation of the Algorithms

The algorithms and their tests were both implemented in C++ version TODO

## 4.3 Generating Test Data and Running the Experiments

# 5 Experimental Results

## 5.1 Functional Testing

Yada yada unit tests

## 5.2 Brute Force

### 5.2.1 Number of operations

Some operations

### 5.2.2 Computation time

Some time

## 5.3 Selection

### 5.3.1 Number of operations

Some operations

### 5.3.2 Computation time

Some time

## 5.4 Comparison

As the two algorithms perform the same operation, they are able to be quantitatively compared. In section 3, we have theoretically shown that the average computational complexity of the brute force and selection algorithms are  $\Theta n^2$  and  $\Theta n$  respectively. In section 5, we have shown that these theoretical models fit the data that was collected quite well TODO r2?? - which allows us to confidently compare the two algorithms. It is clear that the selection algorithm is superior to the brute force method for an arbitrarily sized array - shown computationally and analytically.