Cab 301 Assignment 2

Empirical Analyses of an Algorithm

for BruteForceMedian and Median

Arjai Reynolds

Student Number: N9378600

Connor Jackson

Student Number: HERE

Date Submitted: FIND OUT

**Summary**

**Insert summary here**

1. **Description of the Algorithms**

Both algorithms take on different approaches of accomplishing the same result. The algorithms simply find the median of a list of numbers. They both take in lists (A[0..n-1] and return the median.

**BruteForceMedian Algorithm**

The BruteForceMedian algorithm follows the brute force methodology. The variable k is assigned the length of the list divided by 2. The algorithm also takes into account that a list may have duplicate values, hence, the more complex the implementation. There is a nested loop that compares all elements A[j] to A[i]. The numsmaller variable counts how many elements are smaller than A[i] and the numequal counts how many are equal to A[i]. At each iteration of the outside loop, there is a check whether numSmaller is less than k and k is less than or equal to numSmaller + numEqual. If these conditions are met, the algorithm returns A[i], because that is the median of the list.

Example:

The List {4 ,8, 10, 9, 7, 12, 1, 2, 15}

The variable K will be assigned 9/2 +1. 9/2 is rounded up to be 5.

The first iteration will compare 4 to each element of the list.

NumSmaller will be 2 and numEqual 0.

The comparison at the end of the outside loop is then executed. NumSmaller is less than k but k is not less than or equal to numSmaller + numEqual. This means that A[i] is not the median

The same process is repeated for the next element in the list. numSmaller for the element 8 will be 4 and numEqual 0. This satisfies the comparison statement at the end of the first for loop and returns A[i], since element 8 is the median of the list.

Since finding the median of a list is closely related to sorting a list, issues arise related to the selection problem. This issue is caused by comparing each element of the list to each other. To exploit this, the principles of QuickSort are used in the following algorithm.

**Median Algorithm**

1. **Theoretical Analysis of the Algorithms** 
   1. **Choice of Basic Operation**

**BruteForceMedian**

The basic operation is chosen based on what influences the algorithm the most. There are several basic operations of this algorithm, however, one particular comparison stands out as it varies depending on the size of input and does the most work to solve the problem.

The comparison A[j] < A[i] has the greatest influence on the algorithm because it is used to compare each element to each other. The comparison A[j] == A[i] is only executed if A[j] < A[i] comparison fails, and is a special case that there are duplicates in the list. The comparison numSmaller <k and k<= (numSmaller + numEqual) is not considered as it is all performed at the end of each iteration of the outside loop and does not have a significant effect on the performance of the algorithm as this comparison uses the results of the A[j] < A[i] process.

**2.2 Choice of Problem Size**

The performance of the algorithms are relatively fast with small input sizes. To produce a trend, so that a meaningful comparison can be made, the chosen problem size was a list of unsorted elements with duplicate values that ranged from 2 elements up to 1000.

* 1. **Predicted Theoretical Time Efficiency**

The predicted efficiency for BruteForceMedian follows Θ() because for each element of the list, it is compared to all the elements in the list. For each i element, it will be compared to j elements [0-n-1] times. So this means there are n \* n comparisons.

For the best case, the algorithm only needs to compare each element n-1 times, because if the median of the list is at the beginning, it only needs to compare that element as the algorithm will return it straight away. The worst case is the element is at the end of the list. This means that for each element of the list, it compares n times, hence the quadratic efficiency.

= = =

For an average case, the algorithm will not make n\*n comparisons because the median will not always appear at the very end of the list. The median will not always appear at the beginning of the list and perform n-1 comparisons.

Our intuition says that an element will appear between the best and worst case efficiency. This means on average, it will make half the amount of comparisons as it would exit early once the median is found around the middle of the list. This means the the efficiency will be

**3 Methodology, Tools and Techniques**

**3.1 Programming Environment**

1. The algorithms and test data to produce graphs were both implemented in C++
2. The experiments were performed on an Apple MacBook Pro, running a UNIX based OS. For the time to be accurately measured, all programs on the test computer were closed to avoid interruptions to the run time of the algorithm. To make a meaningful comparison, the same instance of the datasets were used to measure results for both algorithms. The data was random, to create a scenario where the median could be anywhere in the list. The data had natural numbers ranging from 0 to 250. Duplicate values were allowed.
3. To graph results, Microsoft Excel was used, in conjunction with the graphing tools. The program produced text files that were later imported into Excel using CSV. The graphs were then copy and pasted into the report.

**3.2 Implementation of the Algorithms**

**BruteForceMedian**

**Testing Programs**

1. To test the programs, datasets were manually created with the knowledge of what the expected answer is in mind. They were then fed into the program, to see what the algorithm would return as the median. This was then compared with what we expected.

int test1[9]= {4, 1, 10, 9, 7, 12, 8, 2, 15}

This dataset is a normal case with unique numbers.

int test2[9] = {1,2,3,4,5,6,7,8,9}

This dataset is numbers that are in ascending order.

int test3[9] = {9,8,7,6,5,4,3,2,1}

Numbers are in descending order.

int test4[8] = {4,1,10,9,7,12,8,2}

Numbers are out of order.

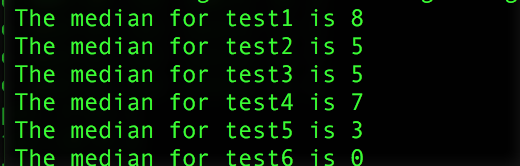
int test5[9] = {1,1,1,1,4,3,3,3,9};

Set contains duplicates .

int test6[9] = {-1,-2,-3,3,4,5,6,3,1};

Set contains negative numbers.

The results are in the console screenshot below.

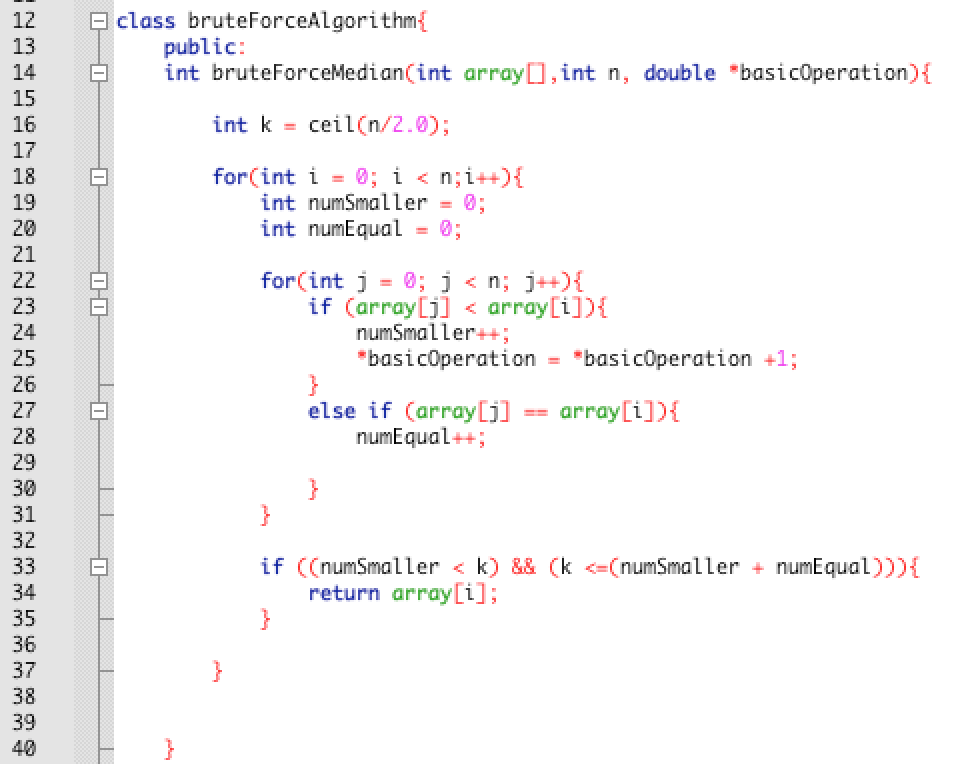


**Basic Operation Count**

**BruteForceMedian**

The basic operation was counted in BruteForceMedian by placing a count variable within the inner comparison of the loop (Appendix A). Since the algorithm returns a variable already, the count can’t simply be returned. To overcome this obstacle, a variable was declared in main and it’s reference was passed to BruteForceMedian.

**Appendix A**

****