CAB302 - Assignment 2

Luke Josh, Jason Queen

May 24, 2016

Contents

1 Summary **2**

2 Description of the Algorithms **2**
- 2.1 Brute Force . . . 2
  - 2.1.1 The Algorithm . . . 2
- 2.2 Median Selection Algorithm . . . 2
  - 2.2.1 The Algorithm . . . 4

3 Theoretical Analysis of the Algorithms **4**
- 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 Average Case Efficiency . . . 5

4 Methodology, Tools and Techniques **6**
- 4.1 Programming Environment . . . 6
- 4.2 Implementation of the Algorithms . . . 6
- 4.3 Generating Test Data and Running the Experiments . . . 6

5 Experimental Results **6**
- 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 . . . 7
  - 5.3.1 Number of operations . . . 7
  - 5.3.2 Computation time . . . 7
- 5.4 Comparison . . . 7

6 Appendix **7**
- 6.1 Source Code . . . 7
  - 6.1.1 Brute Force Median . . . 7
  - 6.1.2 Selection Median . . . 8
  - 6.1.3 Main . . . 9
  - 6.1.4 Unit Tests . . . 11
- 6.2 Figures . . . 13
  - 6.2.1 Brute Force . . . 13
  - 6.2.2 Selection . . . 14
  - 6.2.3 Comparison . . . 15

7 Bibliography **16**

References **16**

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 element's position in a sorted array is in the median position of $\frac{n}{2}$. 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 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

```
function BruteForceMedian(A[0..n − 1])
    k ← ‖n/2‖
    for i ← 1 to n − 1 do
        numsmaller ← 0
        numeqal ← 0
        for j ← 0 to n − 1 do
            if A[j] < A[i] then
                numsmaller ← numsmaller + 1
            else
                if A[j] = A[i] then
                    numequal ← numequal + 1
                end if
            end if
        end for
        if numsmaller < k and k ≤ (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 derives much of its logic from the QuickSelect algorithm. QuickSelect implements a random selection of a pivot element, which is used to partially sort the array around that pivot - which it then recursively continues on the subarray containing the median. QuickSelect has been quoted to have an average running complexity of Θn (Balkcom, n.d.) - which is a value we can expect to get for the median selection 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 (starting with element 0 as the first pivot) 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 pivot will have a value that is lower than that of the pivot element and all elements with an index greater than that of the pivot element will have values that are also greater than the value of the median. Furthermore, in contract 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 $A = [3, 5, 4, 2, 1]$:

$A = [3, 5, 4, 2, 1] \rightarrow$ 5 and 4 are both greater than 3, do nothing

$A = [3, 2, 4, 5, 1] \rightarrow 2 < 3$, swap 5 with 2

$A = [3, 2, 1, 5, 4] \rightarrow 1 < 3$, swap 4 with 1

$A = [1, 2, 3, 5, 4] \rightarrow$ 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 ← 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 ← A[l]
    pivotloc ← l
    for j ← l + 1 to h do
        if A[j] < pivotval then
            pivotloc ← 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 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 $A = [a_1, a_2, ..., a_n]$ 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 $\Theta(n^2)$.
The above caluculations are assuming the median value for the array only occurs once. In other cases, an array may take the form $[1, 2, 3, 3, 3, 4, 5]$, which will skew the results for the average number of comparisons. For example, an array $A = [1, 2, 4, 5, 3, 3, 3]$, the median value is 3, but 3 appears more than once. This means the algorithm will find the median at the 5th position, but it must check each value against 7 other values.

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 $A[j] < pivotval$ 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 if the median value is in the first position of the array, and a maximum of $(n-1)^2$ times, if the median value is in it's correct position at the centre of the array.

3.3.2 Average Case Efficiency

The average case number of comparisons performed in the selection algorithm is considerably more complex than that of the brute force implementation. It has been stated that the selection algorithm is extremely similar to the QuickSort algorithm - and as such, a similar computational complexity is expected. Berman and Paul state that the number of comparisons to be expected for an array of size n is less than $4 \cdot n$ (Paul, 2004).

4 Methodology, Tools and Techniques

4.1 Programming Environment

All testing and computation was executed on a Microsoft Surface running Windows 8.1 64 bit, Intel Core i5 Processor (1.9GHz), 4GB DDR4 RAM.

4.2 Implementation of the Algorithms

The algorithms and their tests were both implemented in C++, compiled using the GCC GNU Compiler in the Code Blocks IDE. The source code for these algorithms is included in Appendix 1.

In general, operation counts were performed by running the algorithm a large number of times on randomly generated arrays, appending the number of operations for each iteration to a pointer to a integer running total, and then calculating the average. The computation time for algorithms was generated in much the same way - using clock types to find the time between two points in the code (namely, the start and end of the algorithm), and then repeating it and taking the average.

4.3 Generating Test Data and Running the Experiments

I think I've already covered this, read again when you're not tired

5 Experimental Results

All graphs referenced in this section have been included in section 2 of the appendix.

5.1 Functional Testing

To ensure the functional correctness of the implemented algorithms - a number of unit tests have been implemented. These tests are included the the test.cpp file, which is also included in appendix 1 (section 6.1.1). These unit tests ensure that the algorithms produce the expected results for a number of predefined test cases.

5.2 Brute Force

5.2.1 Number of operations

The number of operations of the brute force median algorithm was calculated by running the algorithm on a random array of various sizes, and counting the number of times the comparison $A[j] < A[i]$ is performed. This expected number of comparisons was calculated above to be $\frac{n^2}{2}$ for odd sized arrays, and $\frac{n^2+n}{2}$ for even sized arrays. This function has been graphed along the data that was collected from the tests - and clearly shows that the expected trend holds true.

It must be noted that the data sets used to test the number of operations is not tested for uniqueness - a single value may appear more than once. This causes some amount of discrepency between the expected result and the actual result, but can mostly be corrected by increasing the maximum value a value can take. This can be seen in the source code in section 6.1.4, on line 97.

5.2.2 Computation time

The computation for the brute force algorithm was expected to run in Θn^2 time. In other words, the time for the algorithm to run is determined quadratically by the number of elements in the array. It can be seen in the included figure that the collected data follows a quadratic trend. Included on this graph also is the operation count again, which allows us to see that the average case running time is indeed correllated to the number of basic operations performed.

5.3 Selection

5.3.1 Number of operations

The selection algorithm was calculated to perform [a number of operations that it doesn't actually perform], which clearly [cannot be seen because it's fucked].

5.3.2 Computation time

An operating complexity of Θn was expected for the median selection. While the included graph of the data makes the linear trend clear - it is also apparent that the data is neither pretty nor smooth. This is due to the fact that the algorithm runs in a very short amount of time, and the number of averages was not high enough to accurately depict a trend for each array size.

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 Θn^2 and Θn respectively. In section 5, we have shown that these theoretical models fit the data that was collected quite well - which allows us to confidently compare the two algorithms.

It is clear that the selection algorithm is superiour to the brute force method for an arbitrarily sized array - shown computationality and [lol] analytically. Included in the figures are graphs comparing the computation time and operation count for both algorithms, these allow us to see the drastic difference between these two (result wise) equivalent algorithms. It can be seen that the difference in linear and quadratic running time causes the brute force method to reduce the selection method to less than a percent of the graphs y axis.

6 Appendix

6.1 Source Code

6.1.1 Brute Force Median

```
#include "bruteForceMedian.h"

int bruteForceMedian(int *a, int num_elements, float *time_taken, int *num_operations){
    int num_smaller, num_equal, operations_counter = 0, k;

    clock_t start = clock();

    k = ceil((float)num_elements/2.0);

    for(int i = 0; i < num_elements; i++){
        num_smaller = 0;
        num_equal = 0;
        for(int j = 0; j < num_elements; j++){
                if (a[j] < a[i] & (operations_counter++ | 1)){
                    num_smaller++;
                }else{
                    if(a[j] == a[i]){
                    num_equal++;
                    };
                };
        };

        if (num_smaller < k && k <= (num_smaller + num_equal)){
            *num_operations = *num_operations + operations_counter;
```

```
            *time_taken = (float)*time_taken + ((float)clock() - (float)start);

            return a[i];
        };
    }
}
```

6.1.2 Selection Median

```
#include "selectionMedian.h"

int selectionMedian(int *a, int num_elements, float *time_taken, int *num_operations){
    int answer, operations_counter = 0;

    clock_t start = clock();

    if (num_elements == 1){
        answer = a[0];
    } else {
        answer = select(a, 0, floor(num_elements/2.0) , num_elements -1, &operations_counter);
    }

    *num_operations = *num_operations + operations_counter;
    *time_taken = *time_taken + ((float)clock() - (float)start);

    return answer;
}

int select(int *a, int lower, int middle, int upper ,int *operations_counter){

    int pos = partitionSort(a,lower,upper,operations_counter);

    //*operations_counter =  *operations_counter + 1;
    if (pos == middle){
        return a[pos];
    }else{
        //*operations_counter =  *operations_counter + 1;
        if (pos > middle){
            return select(a, lower, middle , pos-1, operations_counter);
        } else if (pos < middle){
            return select(a, pos+1, middle , upper, operations_counter);
        };
    };
}

int partitionSort(int *a, int lower, int upper ,int *num_operations){
    int pivot_val = a[lower];
    int pivot_loc = lower;
    int operations_counter = 0;

    for (int i = lower+1; i <= upper; i++){
        if (a[i] < pivot_val & (operations_counter++ | 1)){
            pivot_loc++;
            swapValues(&a[pivot_loc], &a[i]);
        }
    }
    swapValues(&a[lower], &a[pivot_loc]);

    *num_operations = *num_operations + operations_counter;
    return pivot_loc;
}

void swapValues(int *val_1, int *val_2){
    int *temp = val_1;
```

```
    val_1 = val_2;
    val_2 = temp;
}
```

6.1.3 Main

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdio>
#include <stdlib.h>
#include "bruteForceMedian.h"
#include "selectionMedian.h"
#include "tests.h"

#define MIN_INPUT_SIZE 990
#define MAX_INPUT_SIZE 1003
#define STEP_INPUT_SIZE 3
#define NUM_AVERAGES 2000

#define MS_PER_SEC 1000

using namespace std;

// Creates an array of random numbers for a given size
int *createRandomArray(int num_elements);

// writes the results to a csv file
int writeResultsToFile(int num_elements, int *input_size, float *brute_time, int *brute_operations, float *select_time, int *select_operations);

int main()
{
    int do_test;
    int num_datapoints = (MAX_INPUT_SIZE - MIN_INPUT_SIZE)/STEP_INPUT_SIZE;

    srand(clock());

    cout << "Would you like to run the tests? (0 = No, 1 = Yes)"<< endl;
    cin >> do_test;
    if (do_test > 0){
        run_tests();
        return 0;
    };

    int *input_size = new int[num_datapoints];
    float *brute_time = new float[num_datapoints];
    float *select_time = new float[num_datapoints];
    int *brute_operations = new int[num_datapoints];
    int *select_operations = new int[num_datapoints];

    // #pragma omp parallel for
    for (int i=0; i < num_datapoints; i++) { // for each test
            input_size[i] = MIN_INPUT_SIZE+(i*STEP_INPUT_SIZE);

            cout << "Input Size = " << input_size[i] << endl;

            brute_time[i] = 0;
            select_time[i] = 0;
            brute_operations[i] = 0;
            select_operations[i] = 0;

        for (int j = 0; j < NUM_AVERAGES; j++){
            int *a = createRandomArray(input_size[i]);

```

```
            bruteForceMedian(a, input_size[i], &brute_time[i], &brute_operations[i] );
            selectionMedian(a, input_size[i], &select_time[i], &select_operations[i] );

            delete[] a;
        };

        brute_time[i] = (brute_time[i]*(float)MS_PER_SEC)/((float)NUM_AVERAGES*(float)CLOCKS_PER_SEC);
        select_time[i] = (select_time[i]*(float)MS_PER_SEC)/((float)NUM_AVERAGES*(float)CLOCKS_PER_SEC);

        brute_operations[i] = (int)brute_operations[i]/NUM_AVERAGES;
        select_operations[i] = (int)select_operations[i]/NUM_AVERAGES;

        cout << "   Brute Force Algorithm" << endl;
        cout << "       Time Taken(ms) = " << brute_time[i] << endl;
        cout << "       Num Operations = " << brute_operations[i] << endl;

        cout << "   Selection Algorithm" << endl;
        cout << "       Time Taken(ms) = " << select_time[i] << endl;
        cout << "       Num Operations = " << select_operations[i] << endl << endl;


    };

    writeResultsToFile(num_datapoints, input_size, brute_time, brute_operations, select_time, select_operations);

    delete[] input_size;
    delete[] brute_time;
    delete[] select_time;
    delete[] brute_operations;
    delete[] select_operations;

    return 0;
}

int *createRandomArray(int num_elements){
    int *a = new int[num_elements];

    for ( int i = 0 ; i < num_elements ; i++) {
        a[i] = rand() % 100000000 + 1;
    };

    return a;
}

int writeResultsToFile(int num_elements, int *input_size, float *brute_time, int *brute_operations, float *select_time, int *select_operations){

    ofstream myfile ("results.csv");
    if (myfile.is_open())
    {
        myfile << "input size" << "," << "brute force - time taken" << "," << "brute force - number operations";
        myfile << "," << "select - time taken" << "," << "select - number operations" << endl;
        for ( int i = 0 ; i < num_elements ; i++) {
            myfile << input_size[i] << "," << brute_time[i] << "," << brute_operations[i];
            myfile << "," << select_time[i] << "," << select_operations[i]<< endl;
        };
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

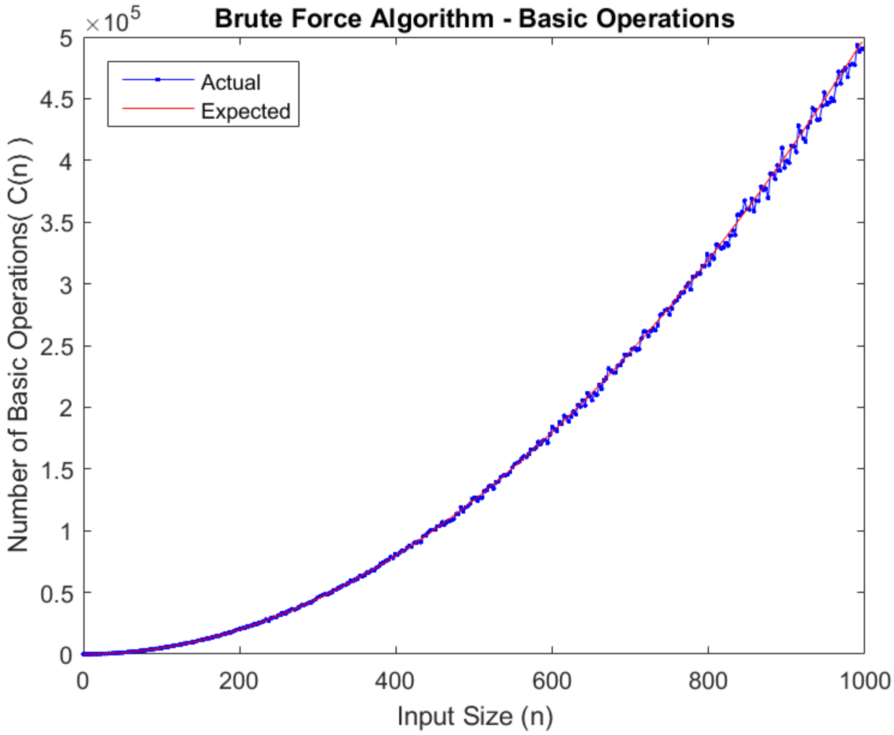
6.1.4 Unit Tests

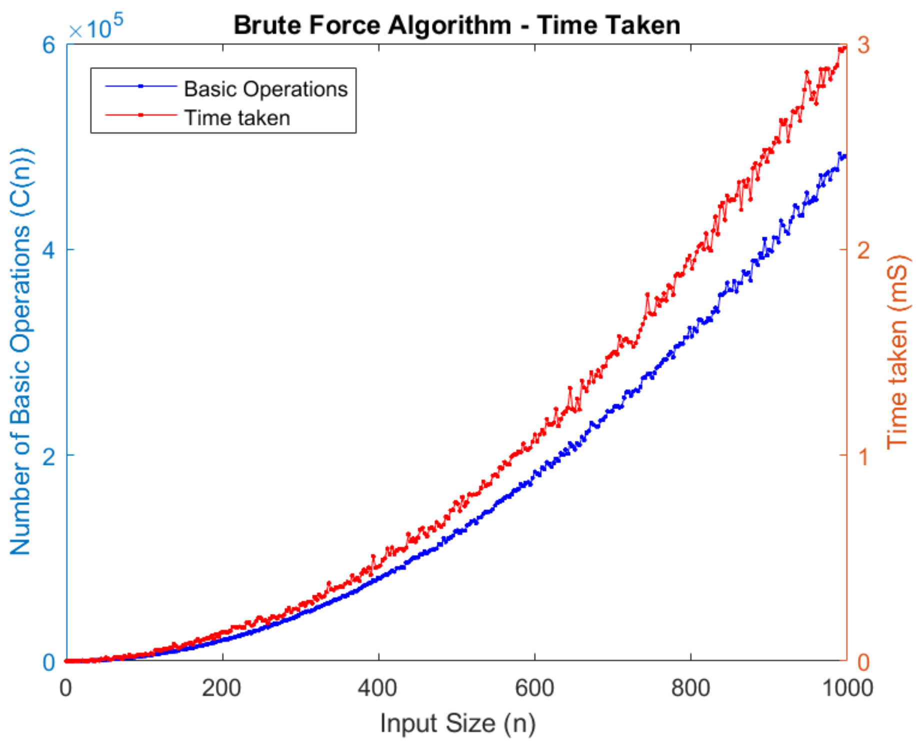
```
#include "tests.h"

void run_tests(){
    int test_size;

    cout << "Test 1" << endl;
    test_size = 5;
    int test1 [5] = { 1, 2, 3, 4, 5 };
    print_array(test1, test_size);
    cout << "    Passed = " << get_results(test1, test_size, 3, 3) << endl;
    delete[] test1;

    cout << "Test 2" << endl;
    test_size = 6;
    int test2 [6] = { 1, 2, 3, 4, 5, 6 };
    print_array(test2, test_size);
    cout << "    Passed = " << get_results(test2, test_size, 3, 4) << endl;
    delete[] test2;

    cout << "Test 3" << endl;
    test_size = 5;
    int test3 [5] = { 1, 2, 3, 3, 5};
    print_array(test3, test_size);
    cout << "    Passed = " << get_results(test3, test_size, 3, 3) << endl;
    delete[] test3;

}

bool get_results(int* a, int num_elements, int brute_expected,int selection_expected){

    bool brute_result =  get_brute_results(a,num_elements,brute_expected);

    bool selection_result =  get_selection_results(a,num_elements,selection_expected);

    return (brute_result && selection_result);

}


bool get_brute_results(int* a, int num_elements, int expected){
    int operations = 0;
    float time = 0;
    int ans;
    bool result;

    ans = bruteForceMedian(a, num_elements, &time, &operations);

    result = (ans == expected);

    cout << "        Brute Force Algorithm" << endl;
    cout << "            Num Operations = " << operations << endl;
    cout << "            Answer = " << ans << endl;
    cout << "            Expected = " << expected << endl;
    cout << "            Passed = " << result << endl;


    return result;
}

// Returns true if the select median returns the correct expected result
bool get_selection_results(int* a, int num_elements, int expected){
    int operations = 0;
    float time = 0;
    int ans;
```

```
    bool result;

    ans = selectionMedian(a, num_elements, &time, &operations);

    result = (ans == expected);

    cout << "        Selection Algorithm" << endl;
    cout << "            Num Operations = " << operations << endl;
    cout << "            Answer = " << ans << endl;
    cout << "            Expected = " << expected << endl;
    cout << "            Passed = " << result << endl;

    return result;
}

void print_array(int* a, int length){
    cout << "    Array = [ ";
    for (int i = 0; i < length; i++){
        cout << a[i] << " ";
    };
    cout << "]" << endl;
}
```

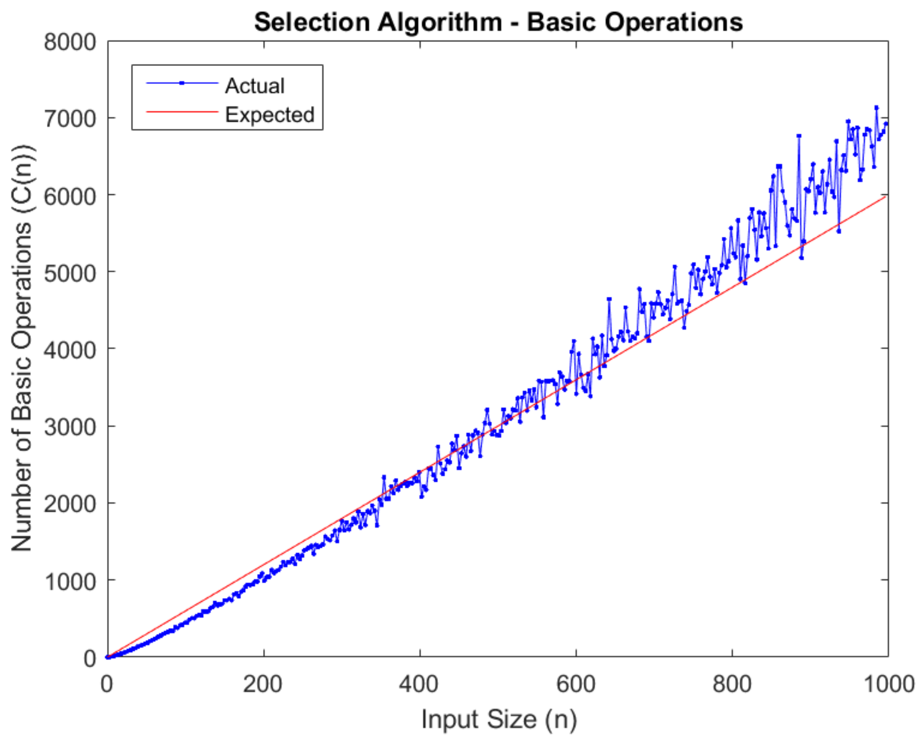
6.2 Figures

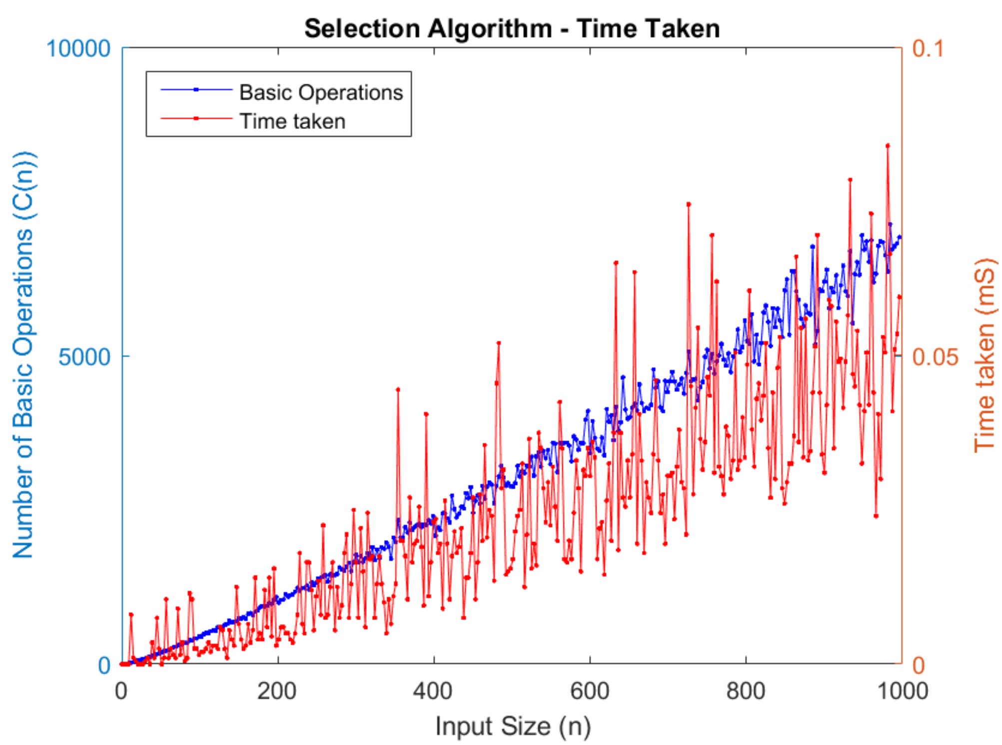
6.2.1 Brute Force



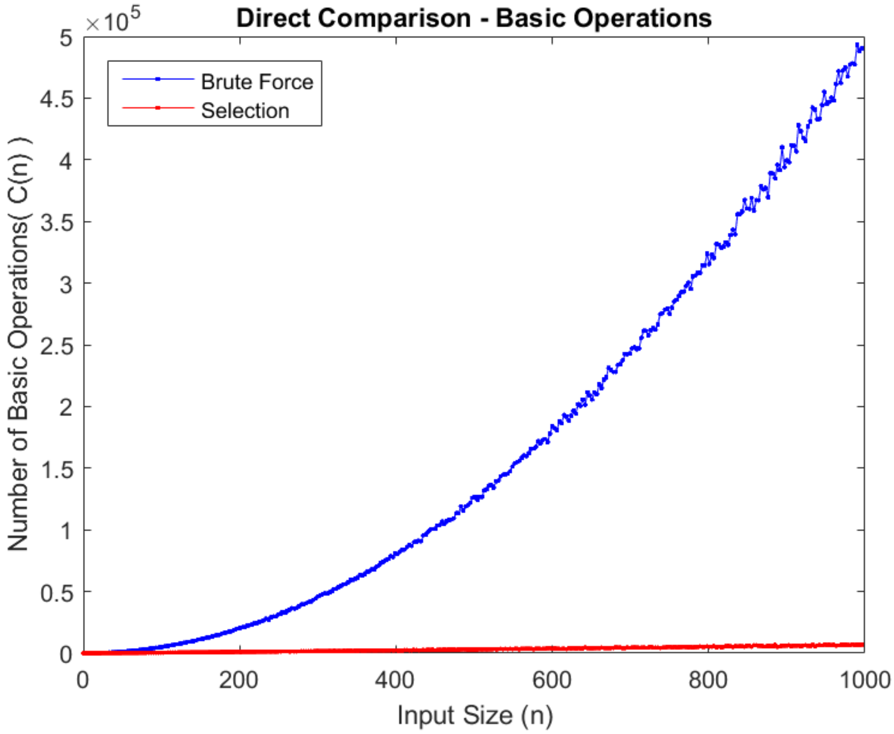


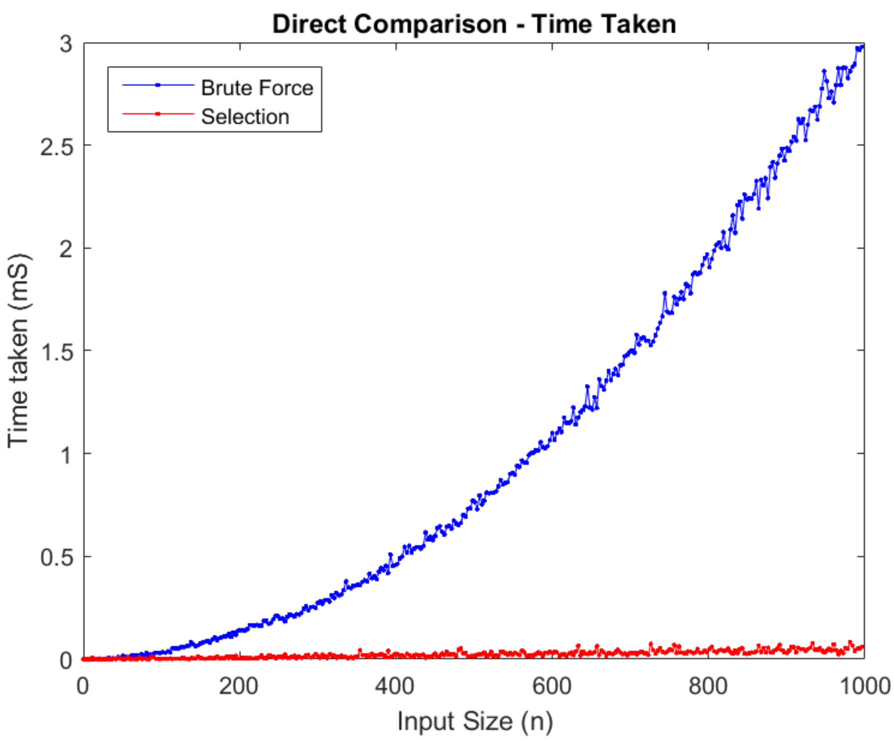
6.2.2 Selection





6.2.3 Comparison





7 Bibliography

References

Balkcom, T. C. . D. (n.d.). *Analysis of quicksort.* https://www.khanacademy.org/computing/computer-science/algorithm
(Online; accessed 15-May-2016)

Paul, K. A. B. . J. L. (2004). *Algorithms: Sequential, parallel, and distributed* (1st ed., Vol. 1). Course Technology.