This report explores the implementation, testing and analysis of the BruteForceMedian algorithm. The predicted order of growth of the algorithm is compared to the number of basic operations and processing time of the implemented program. Finally, the discrepancies between the excremental results and the predicted outcomes are discussed.

Empirical Analysis of BruteForceMedian Algorithm

CAB301 Algorithms and ComplexityAssignment 1

Danial Baharvand - n10084983

Table of Contents

Description of the Algorithm

In this report, the *BruteForceMedian* algorithm will be analyzed. Median is the element in the middle position of an array after it is sorted (Taylor, 2018). The algorithm takes an array and returns its median value by applying the following method.

- The middle position of the array will be stored in the variable k. To do so the size of the array will be divided by 2. If the result is not a whole number, the outcome of the division will be rounded up.
- Using two loops, of which one is nested, each number in the array will be examined for the desired condition. The outer loop takes each element in the array and the inner loop compares it against all elements in the array. This is to record the number of smaller and equal numbers to the selected element.
- For any examined element in the array, if the number of smaller elements in the array is less than k and the number of smaller and equal elements in the array is greater or equal to k, the element will be returned as the median value of the array.

The algorithm works because the median value in an array would have k-1 elements before its occurrence. In addition, there will be at least one equal number to the chosen element as the inner loop does not skip over the selected element in the outer loop. This is also beneficial as counting the equal numbers also takes into consideration any other elements with the same value. The instructions in the outer loop are executed n times where n is the number of elements in the array. These instructions are repeated for each number in the array until the desired element is found. Moreover, the instructions in the inner loop are repeated n^2 times. Therefore, the calculations within the inner loop will be performed for all sets of {A[x], A[y]} where x, y ∈ A. The inner loop checks for each element in the array, if the number at the position provided by the outer loop is greater than the current value. If the test succeeds the ***numsmaller*** variable is incremented by 1. If the test fails, it is then examined if the two numbers are the same value. If so, the ***numequal*** variable is incremented by 1. Ultimately, at any point, if the conditions of the final if statement becomes true, the loop is interrupted and the element at the position provided by the first loop will be returned.

Basic operation

Basic operation is the most expensive arithmetic operation in the algorithm (Tang, 2019). To find the basic operation, the number of times each instruction will be performed should be studied.

- In the BruteForceMedian(A[0..n − 1]) algorithm, with the input size **n** of array **A**, *the* **k ← ⌊n/2⌋** and **return A[i]** instructions will be performed exactly one time.
- In the best case where the median value is the first element of the array, all commands besides the inner loop will be performed once.
- ***numsmaller ← 0*** and ***numequal ←0*** operations will be performed **n** times in the worst case and **i+1** time(s) depending on the position of the median value in the array.
- The comparison process within **if A[j] < A[i] then** will be executed **n** times in the best case and $\mathbf{n^2}$ times in the worst case and **n(i+1)** times in all other cases.
- As the cases within the if statements are condition dependent, it is guaranteed that they will be executed less than the first condition.

As a result, considering that the comparison operation in the inner loop will be executed the most times given an array of any size with any order, this process can be considered the basic operation. This has also been tested using the software implementation with the same conclusion (see Appendix B and BruteForceMedianBasicFind.java).

Methodology, Tools and Techniques

- The algorithm has been developed using the Java programming language in IntelliJ integrated development environment. Java is a general-purpose language focused on object-oriented programming (Leahy, 2019). It has been developed to run on multiple platforms and is offered at no cost by Oracle (Oracle, 2019).
- The experiments were performed on a Windows 10 computer equipped with a quad-core CPU capable of running at 4.0 GHz clock speed. All non-essential processes had been terminated at the time of testing. System memory was consistently monitored to avoid any bottlenecks.
- Java's **Math** class was used to source pseudorandom numbers to create arrays with random numbers. Additionally, the **System** class in Java was used to record the duration of the executions of the program. As the code would be executed in a very small amount of time, all execution times were recorded in Nanoseconds. Finally, all data was formatted and saved to corresponding text files using the **io** class in a way that can be copied to Microsoft Excel.
- The graphs of the experiment results were produced using Microsoft Excel. The information was copied from the output text file and imported to Excel using the "special paste" function.

Implementation of the Algorithm

The Java language implementation strictly adheres to the algorithm's specification (see Appendix A and BruteForceMedian.java).

- The algorithm has been implemented in the method **BruteForceMedian** which accepts an array of integers (**arr**) and returns the median value in the array.
- The integer **k** stores the index of the middle position in the array. **arr.length/2d** divides the length of the array by two and returns it as a double. The **d** modifier is used to keep the number after the decimal point so that the **Math.ceil** method can round up the result. In the end, the final value is cast to an int and stored in **k**.
- Two **for** loops have been utilized to implement the loops in the algorithm.
- The first loop uses the integer **i** as an index which starts at 0 and is incremented at each iteration of the loop by 1. The loop continues to execute as long as **i** is smaller than the length of the array – 1. This is so that every element in the array can be examined. Moreover, **numSmaller** and **numEqual** integers are initialized to 0.
- The second loop is nested within the first loop. The integer **j** is used as an index similar to the first loop. Within the loop, the number of smaller and equal numbers in relation to the value at the indexed position is calculated and stored. Then the results are checked against the conditions as specified by the algorithm. If the conditions are met, the median number will be returned.
- A return statement has been added to the end of the code to avoid compilation issues.

Testing Data Sets

In order to test different aspects of the program, the **Data** class was developed. This class produces sets of data with desired configurations. To modify the contents of the data, values of integers, **dataRules** and **dataSize,** can be set according to the comments found in the class(see Data.java). All tests performed in this report use data sets provided by this class. The Data class can produce the following sets:

- Random, ascending and descending sets with the desired number of elements within them.
- Data sets where the median number is in the first or last position.
- An array where all elements are the same value.
- Sets of data where elements before or after the median are repeating.

Functional Testing of the Implementation

In order to test the correctness of the implementation of the algorithm, 16 test cases were performed (see Appendix C and BruteForceMedian.java).

- These test cases were chosen specially to represent all possible types of data sets.
- The **correctMedian** method was developed to calculate the expected result. This method simply sorts the array and chooses the value at the middle position.
- The output of the **BruteForceMedian** method is compared to the expected output.
- The code was executed for each test case and the results were recorded.
- In all cases, the result produced by the implemented algorithm is equal to the expected outcome. Therefore, it is safe to conclude that the program works correctly.

Counting Basic Operation Executions

The number of basic operations performed by the program can be recorded with the **BruteForceMedianBasicCount** class (see BruteForceMedianBasicCount.java). In this class,

- the variable **t1** is strategically added to the original algorithm to count the performed basic operations.
- It is incremented every time the basic operation is executed.
- The index of the first loop of the algorithm has been initialized outside of its method to be easily accessible later to calculate the expected basic operations.
- No other aspect of the method was changed, and it is accurate with respect to the original algorithm.
- The results will be formatted and stored in the text file **basicCountResults**.
- Two different sets of data sets were used to count the number of basic operations.

Testing the Average Case

The developed program was used to produce average case results.

- Using the **Data** class, 100 sets of 100 pseudorandom numbers were generated.
- The sets were used as input for the algorithm.
- The results were formatted and saved to the output file (see Appendix D).
- Figure 1.1 shows the number of basic operations performed in the calculation of the median number for each set.
- The efficiency of the algorithm belongs to $O(n^2)$ and $\Omega(n)$.
- Using this knowledge, the theoretical maximum and minimum number of basic operations for the given data size has been shown in the graph.

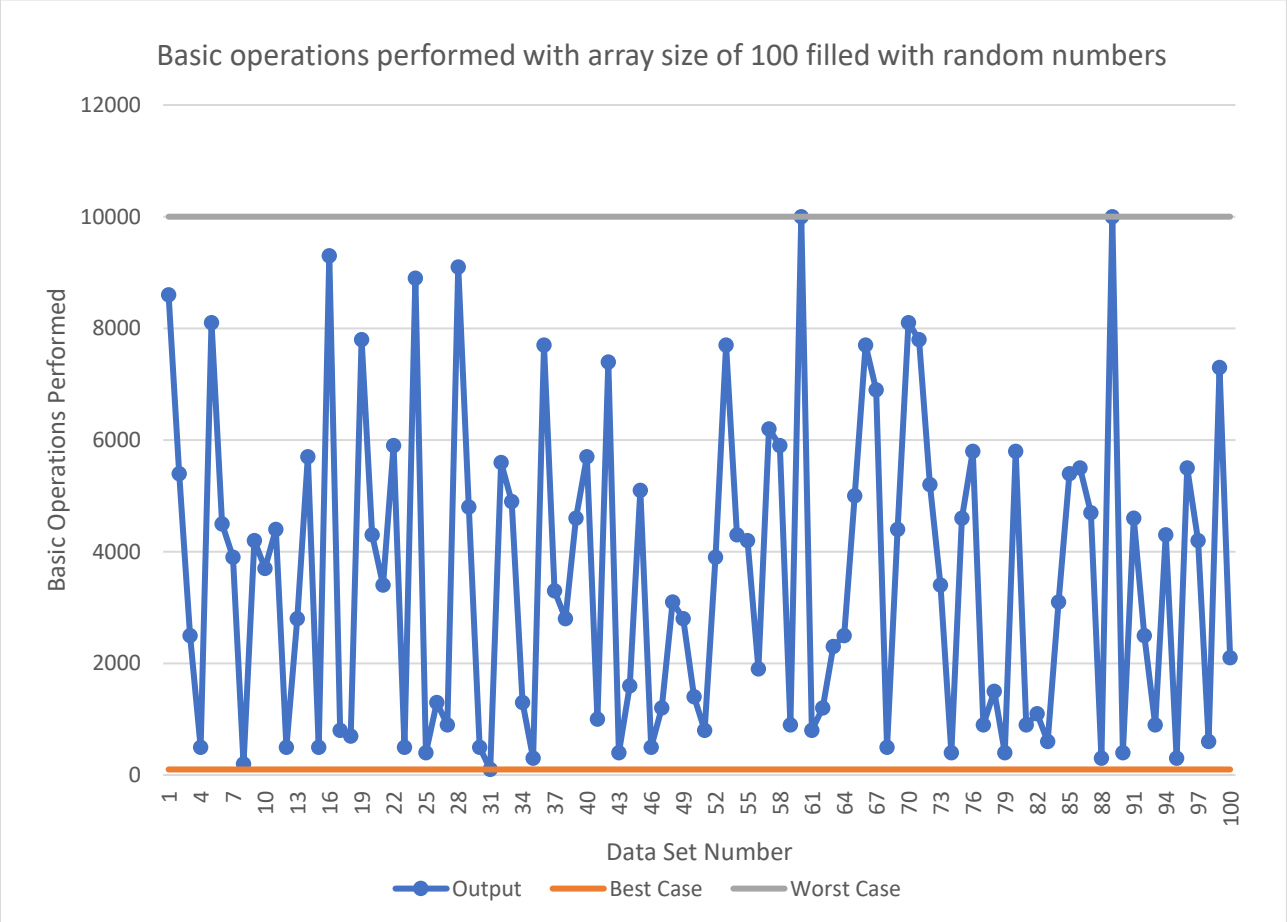

Figure 1.1

Average Case Analysis

The number of basic operations performed in all 100 cases was within the predicted best and worst cases. The theoretical average position of the median number for the algorithm can be calculated as $\left(\frac{1}{n}\right)(1+2+3+\cdots+n)=\left\lceil\frac{n+1}{2}\right\rceil$. Therefore, the theoretical average number of basic operations for an array of 100 numbers is $n\left(\left\lceil\frac{n+1}{2}\right\rceil+1\right)=100\left(\left\lceil\frac{100+1}{2}\right\rceil+1\right)=5200.$ Using the generated table, the actual average number of performed basic operations can be calculated to 3602. while the expected and actual number of average basic operations are not the same, the difference is within the expected range considering the following factors.

- Elements in the arrays were positioned randomly and the median could have been in any position.
- The efficiency and as an extension, the number of basic operations, of the array is heavily dependant on the position of the element and less dependant on the array's size.
- Limited tests have been performed. While working with random numbers, the sample size has a direct relationship with the even distribution of the numbers (Hoeffding, 1992). Therefore, with a sample size of greater than 100 arrays, a more accurate reproduction of the prediction can be expected.

Testing Algorithm Growth

The developed program was used to test how the algorithm grows with respect to input size.

- 21 sorted sets of ascending arrays were put through the algorithm.
- The number of elements in the test data sets started at 1 and then 5 and was incremented by 5 until the set of 100 elements.
- The results were then exported to the output file (see Appendix E).

The relation of basic operations to the size of the array is represented in Figure 1.2.

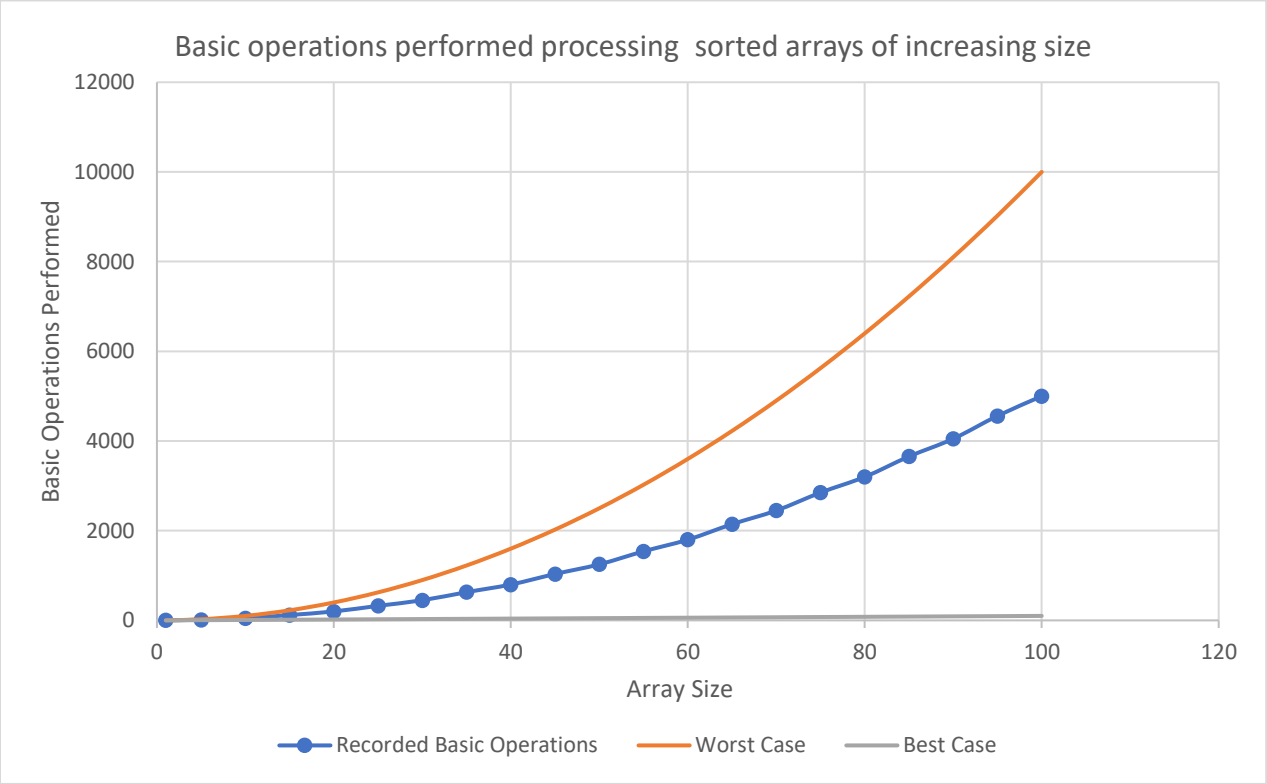

Figure 1.2

Algorithm Growth Analysis

The expected basic operations using the **BruteForceMedian** algorithm for a sorted array with size n is $n(\lceil \frac{n}{2} \rceil)$. The output of the program exactly follows the predicted efficiency. The theoretical average efficiency of the algorithm belongs to Θ(n2) which can be observed by in the convex growth in the basic operations compared to the array size in the graph.

Program Execution Time

The execution time of the program can be recorded using the **BruteForceMedianTimeCount** class (see BruteForceMedianTimeCount.java).

- **Start** variable will record the system time in Nanoseconds at the beginning of the algorithm.
- After the instructions within the algorithm have been processed the start time is deducted from the current system time and stored in the **time** variable.
- Considering that small program fragments often execute too quickly to time accurately, the method has been developed to record 1 million runs of the program and divide that time by the number of executions.
- Through testing, it was discovered that CPU optimizations and cashing can help in the efficiency of processing the instructions. Therefore, to achieve consistent results, 1 million "warm-up" runs are executed before the testing begins.
- While the added code to the original algorithm will have some operational over-head, all possible options were explored to limit additional instructions in the algorithm.

Testing Program Execution Time

- Sets of ascending numbers provided by the **Data** class was used in the experiments. The number of elements in the array started at 2, then 5 and incremented by 5 until the array of 100 elements.
- The length of the array and the average time of each operation is stored in the **timeResults** text file (see Appendix F).

The result of the experiment is shown in Figure 2.1

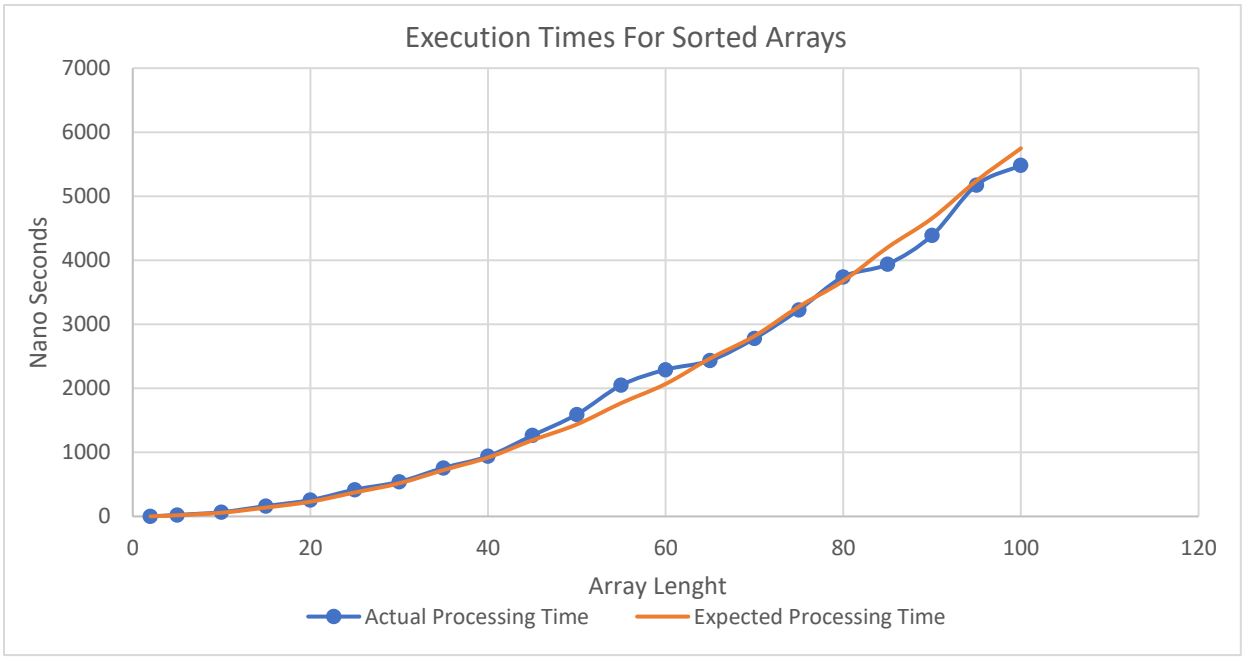

Figure 2.1

Program Execution Time Analysis

Figure 2.1 Shows the experimental results compared to expected times of execution of the algorithm over a range of input sizes. As the algorithm includes a nested loop, its efficiency belongs to $\Theta(n^2)$. Therefore, the number of basic operations can be predicted to grow in a quadratic manner.

- The result of the experiment was extracted from the text file and translated into a graph
- A prediction model of the processing time of the algorithm on the test computer was developed
    - The duration of each basic operation was measured in Nanoseconds by dividing the total time spent by the total number of basic operations performed.
    - It was discovered that each operation took an average of 1.149504512 Nanoseconds to be executed.
    - Using this, and the known number of basic operations for each array size, a prediction model was mapped to the graph.
- To compare the results with the predicted order of growth, the gradients of two lines can be analysed.
    - The gradient of a line can be calculated using $m = \frac{y_2 - y_1}{x_2 - x_1}$.
    - The gradient of the predicted line for the testing computer equals to $\frac{5747.52 - 1.14}{100 - 2} = 58.63$.
    - The gradient of the actual processing time equals to $\frac{5482 - 1}{100 - 2} = 55.92$.
    - This means that when comparing the sets of 1 and 100 elements. On average, for each element added to the array, the processing time is predicted to increase by 58.63 Nanoseconds. However, the experiments show that it is actually increasing by 55.92 Nanoseconds.

Comparing the actual and expected results, most experiments have returned an extremely close result to the predicted value. However, some results deviate from the prediction more significantly. This can be explained by exploring the factors below:

- The exact duration of the basic operation's execution time is needed to achieve a precise prediction of processing time for an array. As the average execution time of basic operations was used, be it over a large sample size, the prediction cannot be completely accurate.
- The same operation on the same hardware can take different amounts of time to complete. Due to the physical architecture of the hardware, processing techniques and operating system optimizations the consistency of the operation time cannot be guaranteed.

Therefore, it can be concluded that the achieved results closely follow the predicted execution times for the algorithm when accounting for variations in the handling of instructions by the operating system.

References

Hoeffding, W. (1992). A Class of Statistics with Asymptotically Normal Distribution. In S. Kotz & N. L. Johnson (Eds.), *Breakthroughs in Statistics: Foundations and Basic Theory* (pp. 308–334). New York, NY: Springer New York.

Leahy, P. (2019, January 16). What Is Java? Retrieved April 1, 2019, from https://www.thoughtco.com/what-is-java-2034117

Oracle. (2019). Java SE Development Kit 11- - Downloads. Retrieved April 1, 2019, from https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html

Tang, M. (2019). *CAB301 Algorithms and Complexity Lecture 3*.

Taylor, C. (2018, February 19). How Do You Find the Median in Statistics? Retrieved April 1, 2019, from https://www.thoughtco.com/what-is-the-median-3126370

Appendix A

Implementation of the BruteForceMedian algorithm

```
private static int BruteForceMedian(int[] arr){//Implementation of the algorithm
    int k =(int)  Math.ceil(arr.length/2d);//Find middle position
    for (int i = 0; i <= arr.length-1; i++){//Loop through all elements
        int numSmaller = 0;//Stores number of smaller elements
        int numEqual = 0;//Stores number of equal elements
        for (int j = 0; j <= arr.length-1; j++){//Loop through all elements
            if(arr[j] < arr[i]) {
                numSmaller = numSmaller + 1;
            } else if (arr[j] == arr[i]) {
                numEqual = numEqual + 1;
            }
        }
        if (numSmaller < k && k <= (numSmaller + numEqual)) {
            return arr[i];//Returns median
        }
    }
    return -1;//To avoid java:missing return statement
}
```

Appendix B

Finding basic operation by testing

	Ascending set	Descending set	Random set	first element is median	last element is median	1 element in set
dataRule	1	2	0	3	4	0
array length	65	6	150	5	5	1
Median	32	2	48	40	40	83
k assignment	**1**	**1**	**1**	**1**	**1**	**1**
first for loop condition	**33**	**4**	**57**	**5**	**1**	**1**
numSmaller and numEqual assignment	**33**	**4**	**57**	**5**	**1**	**1**
for loop and if statement condition check	**2145**	**24**	**8550**	**25**	**5**	**1**
arr[j] is smaller than arr[i] check	**528**	**14**	**4045**	**10**	**2**	**0**
arr[j] is is equal to arr[i] check	**33**	**4**	**142**	**5**	**1**	**1**
return condition check	**33**	**4**	**57**	**5**	**1**	**1**
result is returned	**1**	**1**	**1**	**1**	**1**	**1**

The table shows the number of executions of each instruction based on a range of tested inputs. Bold fields indicate operations whereas other fields provide more information about the test. The highlighted row shows the number of executions for the chosen basic operation. This table confirms the choice of basic operation.

Appendix C

Results of functional testing of the implementation

Test Case	Data Set	Expected Output	Actual Output	Test Passed
single element in the array	[16]	16	16	TRUE
2 random elements in the array	[34, 36]	34	34	TRUE
Odd number of random elements	[99, 65, 25, 27, 78]	65	65	TRUE
even number of random elements	[94, 81, 52, 38, 61, 23, 1, 54, 97, 16]	52	52	TRUE
Large number of random elements	[95, 26, 92, 47, 60, 31, 75, 89, 3, 40, 26, 21, 61, 42, 64, 69, 63, 65, 93, 88]	61	61	TRUE
Odd number of ascending elements	[0, 1, 2, 3, 4]	2	2	TRUE
even number of ascending elements	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	4	4	TRUE
Large number of ascending elements	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]	9	9	TRUE
Odd number of descending elements	[4, 3, 2, 1, 0]	2	2	TRUE
even number of descending elements	[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]	4	4	TRUE
Large number of descending elements	[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]	9	9	TRUE
Median is the last element in the array	[9, 28, 61, 66, 40]	40	40	TRUE
Median is the first element in the array	[40, 9, 28, 61, 66]	40	40	TRUE
All element in the array are equal	[5, 5, 5, 5, 5]	5	5	TRUE
Repeating elements and changing after median	[5, 5, 5, 6, 6]	5	5	TRUE
Repeating elements and changing before median	[5, 5, 6, 6, 6]	6	6	TRUE

Appendix D

Number of basic operations performed in arrays of 100 random elements

Median position	Array size	Basic operations	median position	Array size	Basic operations	Median position	Array size	Basic operations
85	100	8600	2	100	300	4	100	500
53	100	5400	76	100	7700	43	100	4400
24	100	2500	32	100	3300	80	100	8100
4	100	500	27	100	2800	77	100	7800
80	100	8100	45	100	4600	51	100	5200
44	100	4500	56	100	5700	33	100	3400
38	100	3900	9	100	1000	3	100	400
1	100	200	73	100	7400	45	100	4600
41	100	4200	3	100	400	57	100	5800
36	100	3700	15	100	1600	8	100	900
43	100	4400	50	100	5100	14	100	1500
4	100	500	4	100	500	3	100	400
27	100	2800	11	100	1200	57	100	5800
56	100	5700	30	100	3100	8	100	900
4	100	500	27	100	2800	10	100	1100
92	100	9300	13	100	1400	5	100	600
7	100	800	7	100	800	30	100	3100
6	100	700	38	100	3900	53	100	5400
77	100	7800	76	100	7700	54	100	5500
42	100	4300	42	100	4300	46	100	4700
33	100	3400	41	100	4200	2	100	300
58	100	5900	18	100	1900	99	100	10000
4	100	500	61	100	6200	3	100	400
88	100	8900	58	100	5900	45	100	4600
3	100	400	8	100	900	24	100	2500
12	100	1300	99	100	10000	8	100	900
8	100	900	7	100	800	42	100	4300
90	100	9100	11	100	1200	2	100	300
47	100	4800	22	100	2300	54	100	5500
4	100	500	24	100	2500	41	100	4200
0	100	100	49	100	5000	5	100	600
55	100	5600	76	100	7700	72	100	7300
48	100	4900	68	100	6900	20	100	2100
12	100	1300						

Appendix E

Number of basic operations performed processing sorted arrays of increasing size

Median position	Array size	Number of basic operation
0	1	1
2	5	15
4	10	50
7	15	120
9	20	200
12	25	325
14	30	450
17	35	630
19	40	800
22	45	1035
24	50	1250
27	55	1540
29	60	1800
32	65	2145
34	70	2450
37	75	2850
39	80	3200
42	85	3655
44	90	4050
47	95	4560
49	100	5000
499	1000	500000
4999	10000	50000000

Appendix F

Array size and execution times for sets of ascending numbers

Array size	Execution time
2	1
5	23
10	67
15	162
20	255
25	418
30	540
35	758
40	939
45	1264
50	1593
55	2050
60	2292
65	2434
70	2778
75	3227
80	3741
85	3939
90	4389
95	5175
100	5482