CAB301

Empirical Analysis of An Algorithm

ASSIGNMENT 1

Arik Intenam Mir n9637567
12-4-2019

Abstract:

This academic report is intended to identify basic operations and execution time while empirically analyzing the average case efficiency and the time complexity of the Brute Force Median Algorithm. The aforementioned algorithm was implemented in C# on a Windows environment. The number of basic operations and the execution time of this algorithm, in the case of average efficiency, was measured by comparing against arrays of varying lengths. And the functionality of the algorithm was also proved by using various testing cases. The experimental results were analyzed and graphed using Microsoft Excel. The average case efficiency for both basic operations and the execution time of this algorithm was found to be in order with the expected theoretical predictions.

Table of Contents

1. Description of the Algorithm

The basic functionality of the Brute Force Median Algorithm is to take an array of integers of any length, and then return the median value from the inputs of the entire array by taking each value and comparing it against the sequential values of the array, from the 1st index of the array to $\boldsymbol{n} - \mathbf{1}$. This is an iterative algorithm, thus it will return the value of the median, if it's found after each iteration. Two variables $\boldsymbol{numSmaller}$ and $\boldsymbol{numSmaller}$ are used to determine when the median has been found, while another variable called $\boldsymbol{k}$ is equal to the half of the length of the array, if the array is of length $\boldsymbol{n}$. The factor that affects the efficiency of the algorithm is the distance between the first value of the array to the index of the median value. Thus, the algorithm's efficiency will increase if the range of the lowest and the highest integer is kept to a minimum and also if subsequently the length of the array is longer. **Appendix A** shows the implementation of the algorithm in code.

2. Implementation of the Algorithm and Experiments

- The algorithm itself and all the relevant experiments were implemented in the C# (C Sharp) programming language using Visual Studio 2017. C# is a free, general purpose and a multi-paradigm programming language, with simple and user-friendly syntax (Wikipedia, n.d.).
- All of the experiments were implemented on an Asus laptop computer, running on the operating system Windows 10, with the system running at 1.99 GHz.
- The test data required for the experiments were produced using the C# built-in **.Next()** method of the **Random** class. The execution time of the experiments were calculated using an instance of the default **Stopwatch** Class. For using the **Stopwatch class,** the library **System.Diagnostics** had to be declared before the namespace section. The unit of the execution time is in milliseconds. While measuring the execution times, other concurrent running concurrent software applications were minimized.

- The results of the experiments were exported from Virtual Studio to Microsoft Excel 2013 as a .CSV format file for further analysis and graphing by converting the datapoints into a Table. The Method **File.WriteAllText()** was used to convert the gathered data from C# to a CSV format. **File.WriteAllText()** takes the file path in the form of an integer.
- **Figure 1** and **Figure 2** were all prepared in Excel. Further knowledge required for exporting data from C# to a .CSV file were learnt by thorough research on StackOverflow(vc 74, 2011). The report was prepared using Microsoft Word 2013.

3. Design of Experiments

This section briefly explains the implementations and techniques used to formulate the experiments required for calculating the number of average-case basic operations, average-case execution time and the functionality testing respectively.

3.1 Average-case Basic Operations and Execution Time

1. For calculating both, the number of average-case basic operations and the execution time of the Brute Force Median algorithm, the experimental data was gathered in the form of arrays, with each array length ranging between 1000 and 20000, in increment of 1000.
2. The pre-defined class **Random** of C# was used to produce random numbers for the experimental arrays, where the numbers ranged from -2,147,483,647 to 2,147,483,647, which in other words is the range of **int32** (Microsoft .NET,n.d.)
3. The average values of basic operations was stored in the variable **double averageCount** . And in the similar fashion, for the execution time, the variable **double averageMilliSecs** was used to store the data.
4. The method used for calculating the basic operations of the algorithm is shown in **Appendix B.** The obtained data was then compared against the theoretical prediction of the basic operations, which was figured out using the theoretical average case efficiency (n^2) of the algorithm, in the following way:

$$c_1 \times n^2 = No. of\ basic\ operations$$

Here, *n* is the array length, c_1 is the constant for all the arrays.

5. The source-code for the calculation of the number average-case basic operation and the execution time was implemented in two separate C# files named ***Basic_operations*** and ***Execution_time*** respectively.

3.2 Functionality Testing

1. The functional correction testing of the algorithm was conducted by thorough tests using the test method named **FunctionalTesting().**This method displays the output of the test arrays that was inserted through the brute force median algorithm.
2. Varying sorts of arrays were used for testing the correctness, including Sorted, Unsorted, Reverse, Partially sorted, Random, Consecutive and array of Odd length.
3. All the arrays are of length 10, except the odd length array, where the length of the array is set at 11.
4. The source-code for the functionality testing was performed in the C# file **Functionality_test.**

4. Experimental Results

4.1 Basic Operations

Through thorough observations and research, the **for** loops of the algorithm were identified as the basic operation. It was observed that, the **for** loops takes the most substantial amount of time to execute, compared to the **if** and **else if** statements and all the increment statements. In the inner **for** loop, the every items in the array are compared against the first element, and consequently against the second item, then the third, and so on. The basic operations counter for the algorithm is placed inside the inner **for** loop. Which increments every time, an item is compared against the whole length of the array. A code-snippet of the algorithm, with the basic operation counter is given in the **Appendix A.**

After comparing the theoretical predictions and the against the obtained result, it was found that there are minor discrepancies among the results. While, the theoretical prediction follows a steady path following the average-case efficiency order of growth(n^2), the experimental result doesn't for the whole experiments.

4.2 Execution Time

After experimenting, it was observed that the average execution time of the Brute Force Median Algorithm is accurate and produced a clear trend, when compared to the algorithm's predicted growth (n^2). The graph in **section 5.2** demonstrates the exponential rate of growth in execution time as the array length n increases.

4.3 Functionality Correctness

Functionality correctness of the algorithm was proved after testing the algorithm with numerous instances of different sorts of arrays. **Appendix D** demonstrates the implementation of the method used for testing. For all of testing instances, the desired output matched the actual outputs. Table in **Section 5.3** shows represents all the testing instances with their expected and actual outputs. The results of code-snippets of **Appendix D** is shown in **Appendix E.**

5. Analysis of Experimental Results

This section shows the visual outcomes of all the experimental results in the form of graphs, tables and figures. The outcomes were compared with theoretical predictions to better understand the results.

5.1 Basic operations

The graph below shows the average-case basic operations compared against its theoretical counterpart. In the graph, the output of the experiments (**red**) are graphed with the average-case efficiency theoretical predictions (**green**). The x-axis of the graph shows the array length from 1000-20000, with the incrementing order of 1000, while the y-axis shows the number of average-case base operations, starting from 0-250 millions with an incrementing order of 50 million.

It was seen that the experimental results seem to deviate a bit from the predictions, but still at the end of the experiment concludes in the same manner. The reason for the slight deviation is due to the properties of the operating system and the computing environment.

The graph is represented in the **following page.**

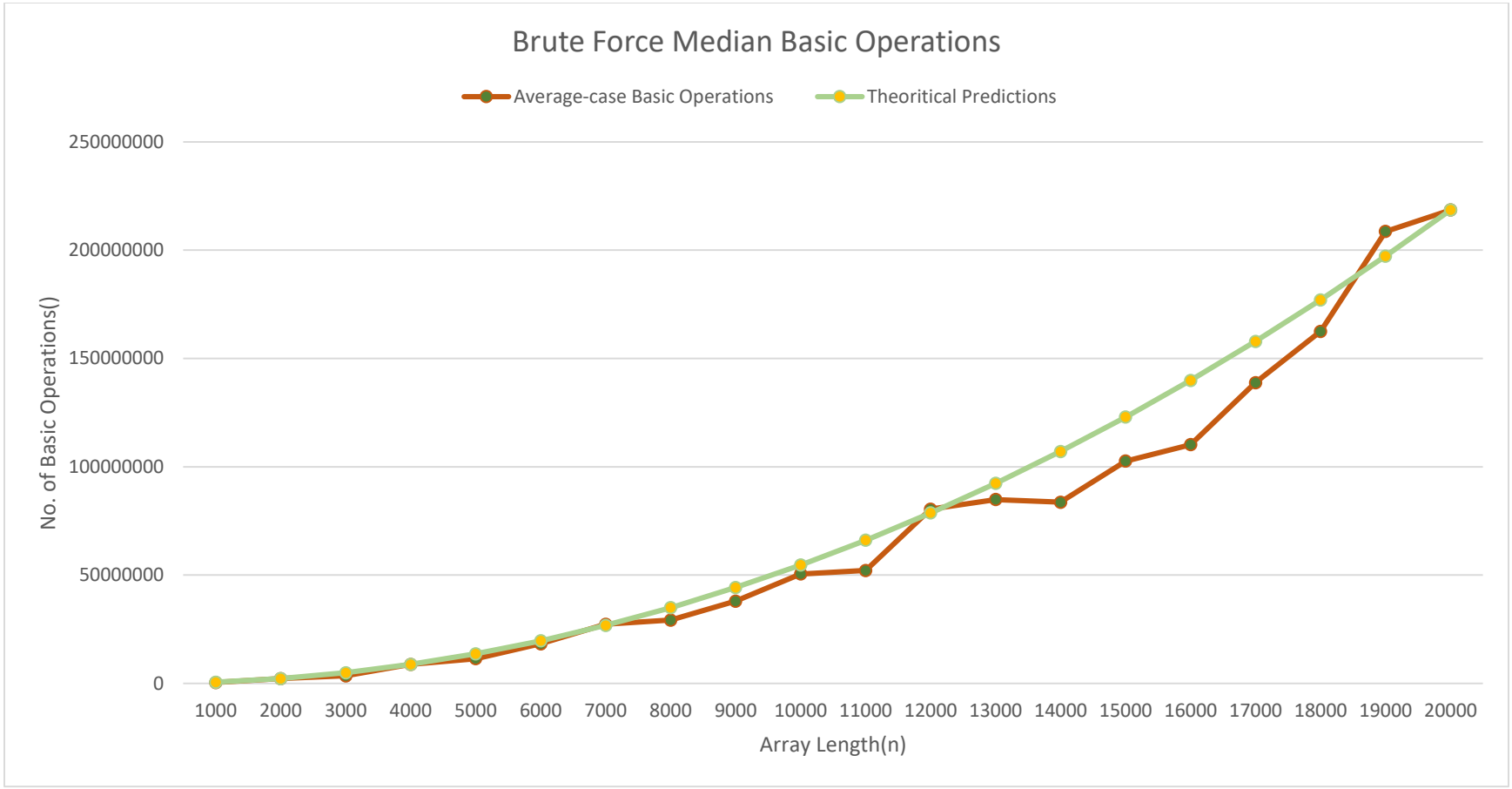

Figure 1: Graph of average-case basic operations compared against theoretical predictions.

5.2 Execution Time

It is seen that the observed execution time follows the same trend as the theoretical prediction. The visible deviation between them is due to the contrasting properties of the operating system and the computing environment used to produce these data. The graph below shows the execution time required for the brute force median algorithm to perform. The unit used for representing is milliseconds.

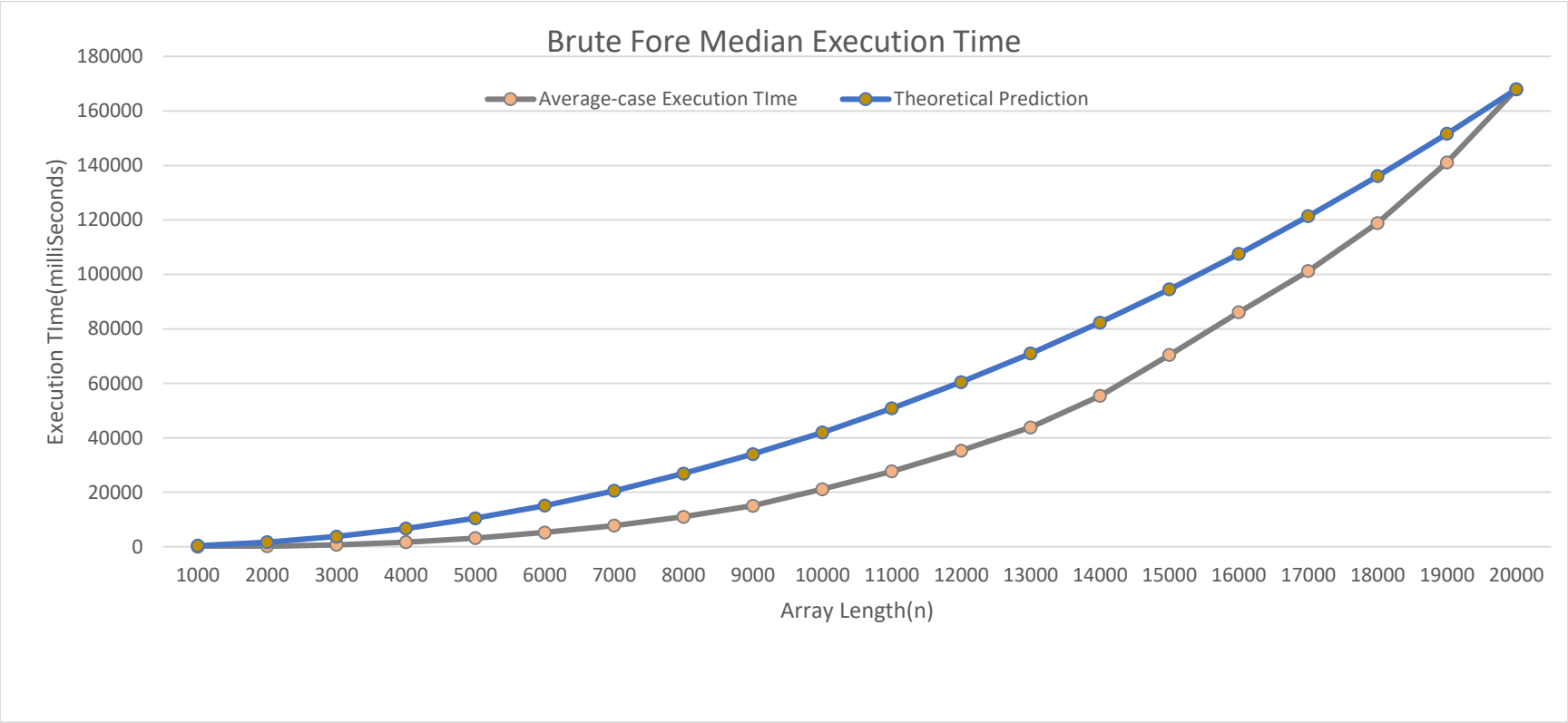

Figure 2: Graph of average-case execution time compared against theoretical predictions

5.3 Functionality Correctness

The table given below demonstrates the functional correctness of the brute force algorithm for different arrays of varying lengths:

TEST CASE	TEST INSTANCE	EXPECTED OUTPUT	ACTUAL OUTPUT	TEST RESULT
Array of Consecutive Numbers	`sorted_arr =[ 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]`	26	26	**Successful**
Array of Sorted Numbers	`sorted_arr= [5, 7, 11, 13, 15, 17, 20, 23, 27, 29]`	15	15	**Successful**
Array of Unsorted Numbers	`unsorted_arr = [20, 23, 7, 5, 11, 15, 29, 27, 13, 17]`	15	15	**Successful**
Reverse Sorted array	`reverse_arr = [29, 27, 23, 20, 17, 15, 13, 11, 7, 5]`	15	15	**Successful**
Array of Random Numbers	`random_arr = [44, 12, 32, 22, 1, 23, 6, 49, 11, 43]`	22	22	**Successful**
Array of Partially Sorted Numbers	`partial_arr = [13, 15, 17, 19, 20, 45, 11, 34, 31, 14]`	17	17	**Successful**
Array of Odd length	`odd_arr = [12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32]`	20	20	**Successful**
Array of both positive and negative integers	`mixed_arr = [-12,34,54,-99,-34,-7,89,234,-119,37]`	-7	7	**Successful**

Table 1: Functionality testing of Brute Force Median Algorithm.

REFERENCES

1. C Sharp(Programming Language).(n.d.).
   Retrieved on 6th April, 2019 from
   https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

2. Vc 74. (2011). *C# datatable to csv.*
   Retrieved on 6th April, 2019 from
   https://stackoverflow.com/questions/4959722/c-sharp-datatable-to-csv

3. Microsoft .NET. (n.d.) *Int32 Struct.*
   Retrieved on 7th April, 2019 from
   https://docs.microsoft.com/en-us/dotnet/api/system.int32?view=netframework-4.7.2

APPENDICES

Appendix A

Implementation of the Brute Force Median Algorithm.

```
public static int Median(int[] A)
{
    int n = A.Length;
    int k = n / 2;

    for (int i = 0; i <= (n - 1); i++)
    {
        int numsmaller = 0;
        int numequal = 0;

        for (int j = 0; j <= (n - 1); j++)
        {

            if (A[j] < A[i])
            {

                numsmaller = numsmaller + 1;
            }
            else if (A[j] == A[i])
            {

                numequal = numequal + 1;
            }
        }
        if ((numsmaller < k) && (k <= (numsmaller + numequal)))
        {
            return A[i];
        }
    }
    return -1;
}
```

Appendix B

Implementation of counting the total number average-case basic operations.

```
static void Main(string[] args)
{
    int noOfRuns = 20;
    var csv = new StringBuilder();
    for (int size = 1000; size <= 20000; size += 1000)
    {

        long totalCounts = 0;
        double averageCount = 0;
        for (int i = 1; i <= noOfRuns; i++)
        {
            int[] X = GenerateRandomArray(size);
            int count = Median(X);
            totalCounts = totalCounts + count;

            averageCount = totalCounts * 1.0 / noOfRuns;

        }

        Console.WriteLine("Size = " + size + "; Average Count = " + averageCount);


        var first = size.ToString();
        var second = averageCount.ToString();

        var newLine = string.Format("{0},{1}", first, second);
        csv.AppendLine(newLine);

    }

    File.WriteAllText(@"C:\Users\Arik\Desktop\CAB 301\Assesment 1\basic_operationsmir.csv", csv.ToString());
    Console.ReadKey();
}
```

Appendix C

Implementation of counting the total execution time of the algorithm

```
static void Main(string[] args)
{
    int totalRunningTimes = 20;
    Stopwatch sw = new Stopwatch();
    var csv = new StringBuilder();
    for (int size = 1000; size <= 20000; size += 1000)
    {
        long totalMilliSecs = 0;
        double averageMilliSecs = 0;
        for (int i = 1; i <= totalRunningTimes; i++)
        {
            long milliSecs = 0;
            int[] A = GenerateRandomArray(size);
            sw.Start();
            Median(A);
            sw.Stop();
            milliSecs = sw.ElapsedMilliseconds;
            totalMilliSecs = totalMilliSecs + milliSecs;
        }

        averageMilliSecs = totalMilliSecs * 1.0 / totalRunningTimes;
        Console.WriteLine("Size: " + size + "; Average Running Time (MilliSec)= " + averageMilliSecs);
        var first = size.ToString();
        var second = averageMilliSecs.ToString();
        var newLine = string.Format("{0},{1}", first, second);
        csv.AppendLine(newLine);

    }
    File.WriteAllText(@"C:\Users\Arik\Desktop\CAB 301\execution_time.csv", csv.ToString());
    Console.ReadKey();
}
```

Appendix D

Implementation of the method used for functionality correctness testing.

```
static void FunctionalTesting()
{

    Console.WriteLine("Running Tests");
    Console.WriteLine("----------------------");

    // Console.WriteLine("");
    int n = 10;
    int k = (int)Math.Ceiling(n / 2.0);

    //Array of Consecutive Numbers
    int[] consecutive_arr = new int[] { 22, 23, 24, 25, 26, 27, 28, 29, 30, 31 };
    int consec_median = Median(consecutive_arr, n, k);
    Console.WriteLine("An array of CONSECUTIVE NUMBERS: { 22, 23, 24, 25, 26, 27, 28, 29, 30, 31 }");
    Console.WriteLine("Median of the array is: {0}", consec_median);

    Console.WriteLine();

    //Array of Sorted Numbers
    int[] sorted_arr = new int[] { 5, 7, 11, 13, 15, 17, 20, 23, 27, 29 };
    int sorted_median = Median(sorted_arr, n, k);
    Console.WriteLine("An array of SORTED NUMBERS: { 5, 7, 11, 13, 15, 17, 20, 23, 27, 29}");
    Console.WriteLine("Median of the array is: {0}", sorted_median);

    Console.WriteLine();

    //Array of Unsorted Numbers
    int[] unsorted_arr = new int[] { 20, 23, 7, 5, 11, 15, 29, 27, 13, 17 };
    int unsorted_median = Median(unsorted_arr, n, k);
    Console.WriteLine("An array of UNSORTED NUMBERS: { 20, 23, 7, 5, 11, 15, 29, 27, 13, 17 }");
    Console.WriteLine("Median of the array is: {0}", unsorted_median);

    Console.WriteLine();

    //Array of Reverse Numbers
    int[] reverse_arr = new int[] { 29, 27, 23, 20, 17, 15, 13, 11, 7, 5 };
    int reverse_median = Median(reverse_arr, n, k);
    Console.WriteLine("An array of REVERSE NUMBERS: { 29, 27, 23, 20, 17, 15, 13, 11, 7, 5 }");
    Console.WriteLine("Median of the array is: {0}", reverse_median);

    Console.WriteLine();

    //Array of Random Numbers
    int[] random_arr = new int[] { 44, 12, 32, 22, 1, 23, 6, 49, 11, 43 };
    int random_median = Median(random_arr, n, k);
    Console.WriteLine("An array of RANODM NUMBERS: { 44, 12, 32, 22, 1, 23, 6, 49, 11, 43 }");
    Console.WriteLine("Median of the array is: {0}", random_median);

    Console.WriteLine();

    //Array of Partially Sorted Numbers
    int[] partial_arr = new int[] { 13, 15, 17, 19, 20, 45, 11, 34, 31, 14 };
    int partial_median = Median(partial_arr, n, k);
    Console.WriteLine("An array of PARTIALLY SORTED NUMBERS: { 13, 15, 17, 19, 20, 45, 11, 34, 31, 22 }");
    Console.WriteLine("Median of the array is: {0}", partial_median);
```

Appendix E

Output of method used for functionality correctness testing.

```
FUNCTIONALITY CORRECTNESS TESTING
---------------------
An array of CONSECUTIVE NUMBERS: { 22, 23, 24, 25, 26, 27, 28, 29, 30, 31 }
Median of the array is: 26

An array of SORTED NUMBERS: { 5, 7, 11, 13, 15, 17, 20, 23, 27, 29}
Median of the array is: 15

An array of UNSORTED NUMBERS: { 20, 23, 7, 5, 11, 15, 29, 27, 13, 17 }
Median of the array is: 15

An array of REVERSE NUMBERS: { 29, 27, 23, 20, 17, 15, 13, 11, 7, 5 }
Median of the array is: 15

An array of RANODM NUMBERS: { 44, 12, 32, 22, 1, 23, 6, 49, 11, 43 }
Median of the array is: 22

An array of PARTIALLY SORTED NUMBERS: { 13, 15, 17, 19, 20, 45, 11, 34, 31, 22 }
Median of the array is: 17

An array of ODD LENGTH: { 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32 }
Median of the array is: 20

An array of PARTIALLY SORTED NUMBERS: { -12,34,54,-99,-34,-7,89,234,-119,37}
Median of the array is: -7
```