UNIVERSITY of INFORMATION TECHNOLOGY and MANAGEMENT
in Rzeszow, POLAND

Project

Measuring Time Complexity

Lecturer:
Dr. Władysław Homenda

Student:
Diana Levchenko w64733

Class:
Algorithms and Data Structures

Field of study:
Programming

Rzeszów 2021

Content

Introduction

In this project, the peculiarities of measuring the time complexity of several sorting as well as searching algorithms will be investigated.

The following sorting algorithms will be presented:
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick sort

as well as the following sorting algorithms will be investigated:
- Linear Search
- Binary Search (used for sorted data only).

Thus, with the help of the program presented, the time of execution of the above algorithms will be measured and, with the help of Microsoft Excel, the graphs of dependence between the number of array elements and the algorithms' execution time (in nanoseconds) will be drawn.

Moreover, the data for sorting algorithms will be presented in three forms: sorted, sorted in reversed order (within this project for convenience purposes it will be called unsorted) and random. Thus, the sorting algorithms will be analyzed with respect to each of the three types of data.

For each experiment six tests will be conducted, each time the size of the array will be multiplied by the number of the test run (e.g. having array size of 1000 elements for the first test, there will be the size of 2000 elements for the second test, 3000 for the third one, etc.)

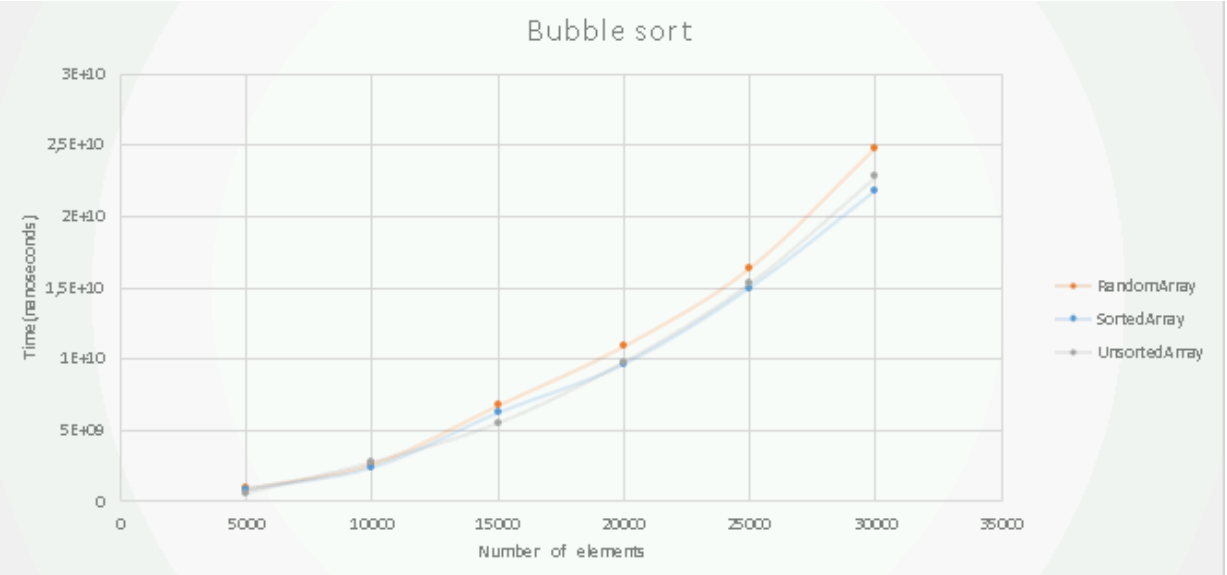
1. Sorting algorithms implemented

1.1. Bubble sort

Bubble sort is a stable in-place sorting algorithm which repeatedly swaps the adjacent elements of a sequence in case they are in the wrong order.

Time complexity: O(n^2) in all cases. Thus, the algorithm gets really slow as the amount of input data increases.

	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
5000	935446700	894657200	614269300
10000	2646104900	2397284700	2746635900
15000	6759328300	6238195500	5504702300
20000	10899914100	9647834800	9780220200
25000	16357113600	14995954900	15285290800
30000	24781502300	21835263100	22779432500

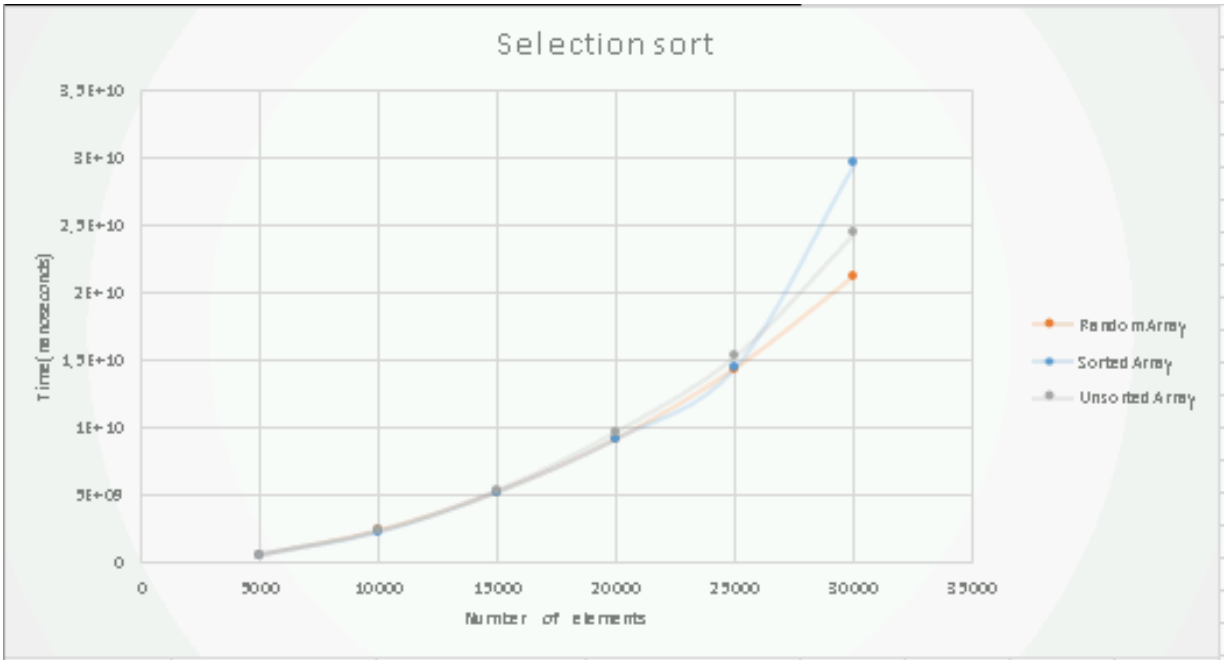


1.2. Selection Sort

Selection sort is an in-place unstable (by default) algorithm sorting an array by repeatedly finding the minimum element (for ascending order) and placing it at the array's beginning. Thus, virtually we have two parts of the array: sorted and unsorted. Each iteration the minimum element is picked from the unsorted part of the array and then moved to the sorted one.

Time complexity: O(n^2) in all cases. Thus, the algorithm gets really slow as the amount of input data increases.

	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
5000	558288200	585063300	587167800
10000	2344981900	2275264300	2356390700
15000	5243892100	5263250300	5397580500
20000	9194297400	9225875800	9734408300
25000	14411331800	14521952000	15330916800
30000	21270653900	29660199300	24435618200

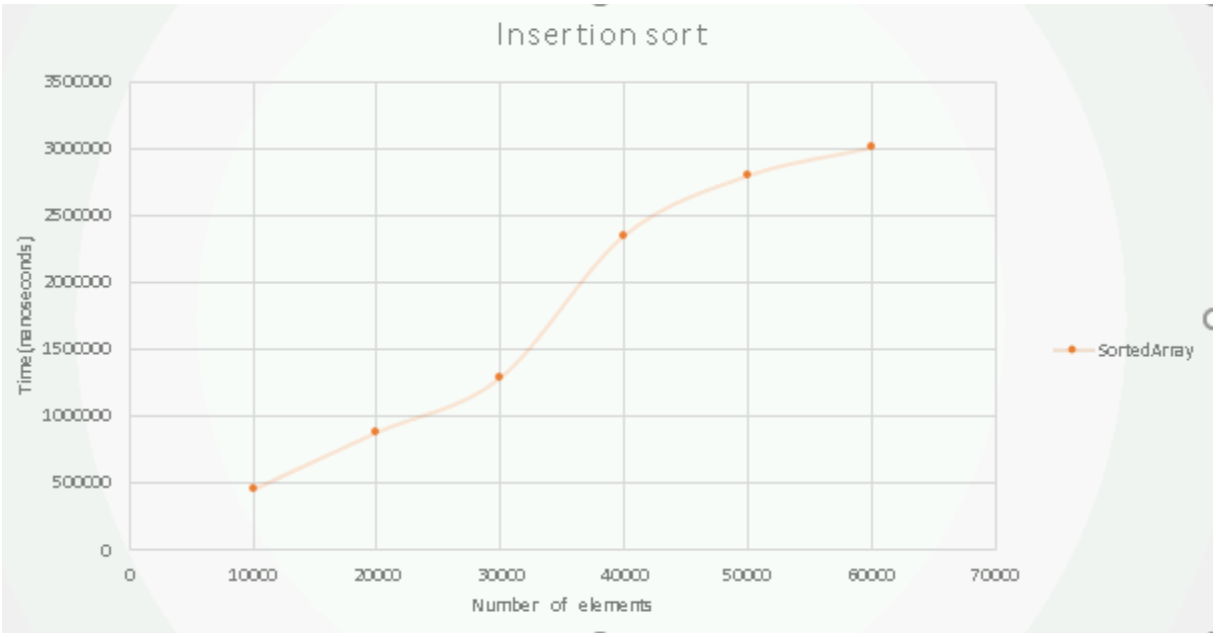


1.3. Insertion sort

Insertion sort is a stable in-place sorting algorithm (usually used when the number of elements is small). The array that is being sorted is virtually split into two parts: sorted and unsorted. Thus, the elements of the unsorted part are taken and placed at the correct position of the sorted part of the array.

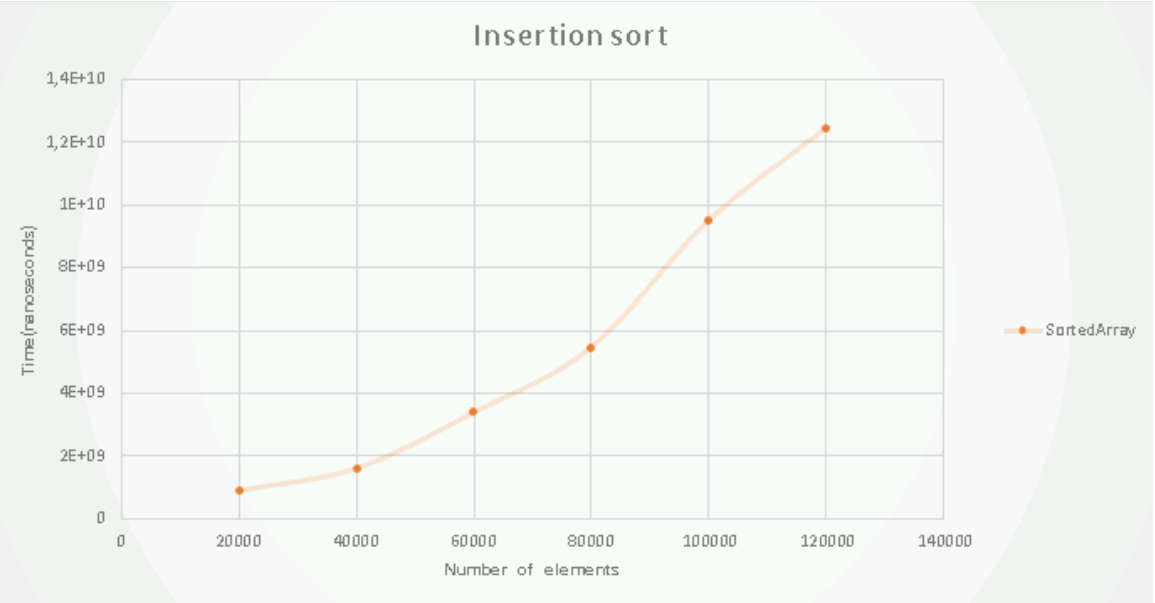
Time Complexity: takes most time when sorting an array in the reversed order with O(n^2) and minimum time with O(n) when the array is already sorted.

	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
10000	1677104700	454000	2471484100
20000	5245468000	881900	9729412700
30000	11236265800	1289200	23470501800
40000	19684964100	2351700	40628446900
50000	31433446500	2799800	62528169100
60000	44401133100	3010300	91357906000



One can observe that for bigger sizes of the SORTED arrays investigated the linear optimistic complexity is more explicit:

	time(nanoseconds)
sizeOfArray	SortedArray
20000	901870200
40000	1608531400
60000	3391938500
80000	5462145900
100000	9500804600
120000	12439520800



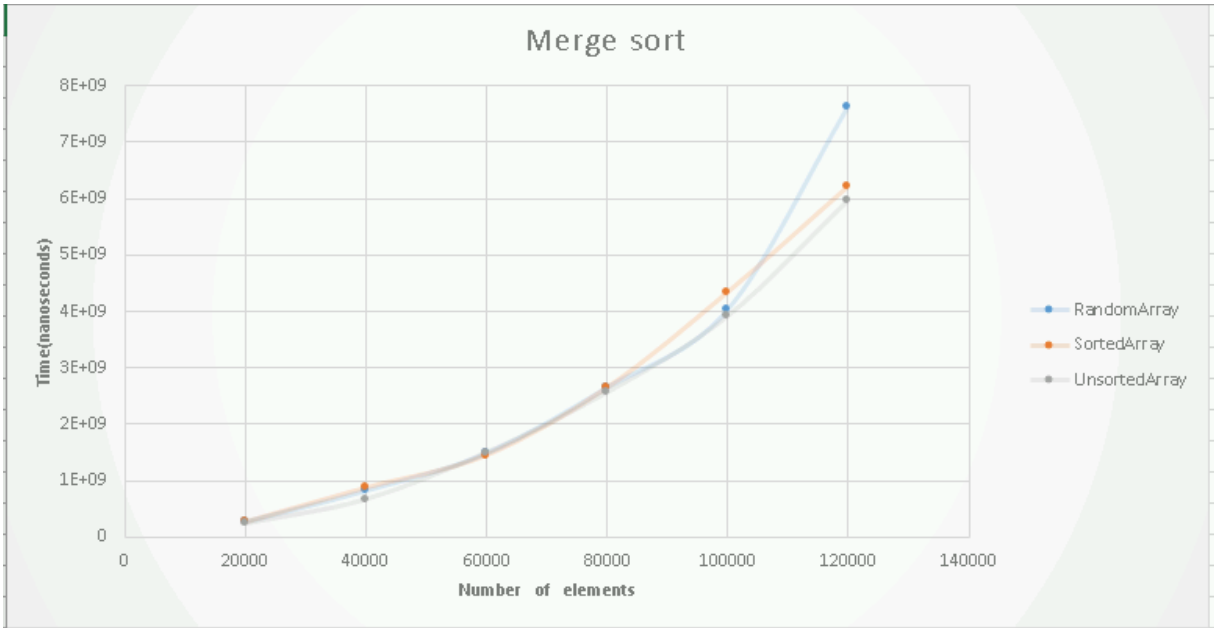
1.4. Merge Sort

Merge sort is a stable Divide and Conquer algorithm. In a typical implementation, not in-place.

Merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. After the achievement of atomic elements that may no longer be broken, they are combined in a sorted manner. Thus, a sorted array is obtained.

Time Complexity: O(n*log n) - average and pessimistic.

	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
20000	281824500	262814100	240141000
40000	830321200	868439000	669803400
60000	1491092100	1444646400	1498799000
80000	2655005000	2639472500	2563023700
100000	4026573000	4327088700	3913201100
120000	7615967700	6200267700	5963855900



1.5. Quick Sort

QuickSort is an unstable in-place algorithm, which picks an element as pivot (might be beginning of the array, median, end or random element) and partitions the given array around the picked pivot.

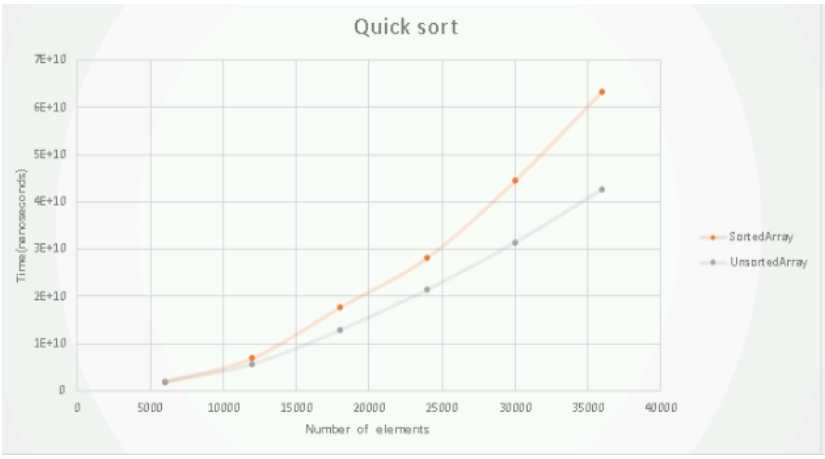
An element x of array (pivot), is put at its correct position in the array, all smaller elements (smaller than x) are put before x, and all greater elements (greater than x) - after x.

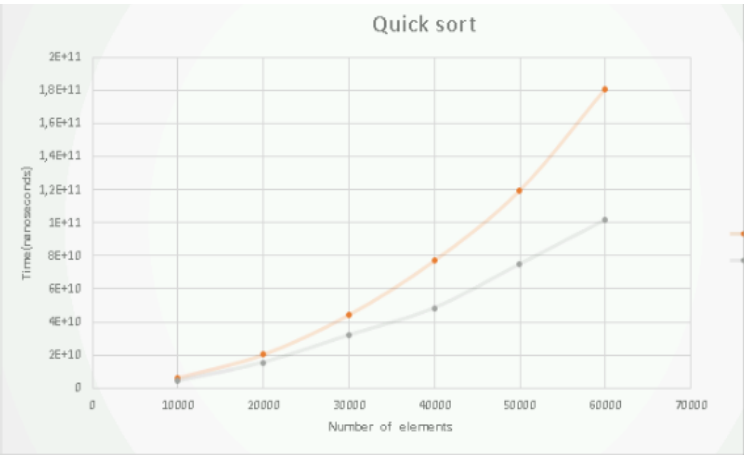
Thus, the array is sorted until all elements are put in the right positions.

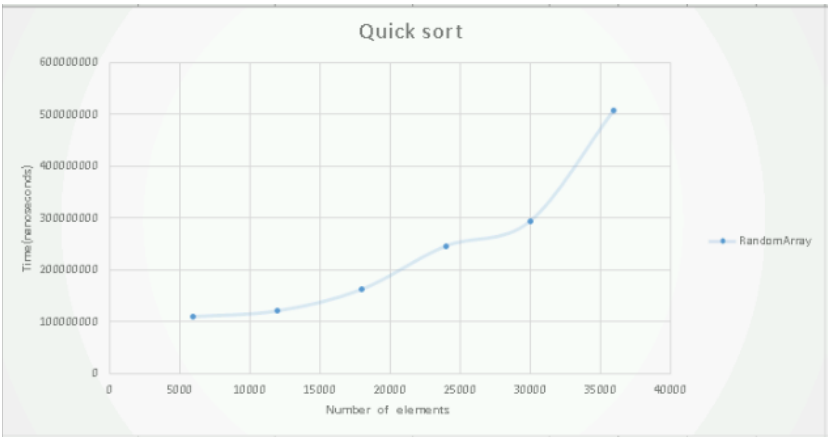
Time Complexity: O(n*log n) - average, O(n^2) - pessimistic.

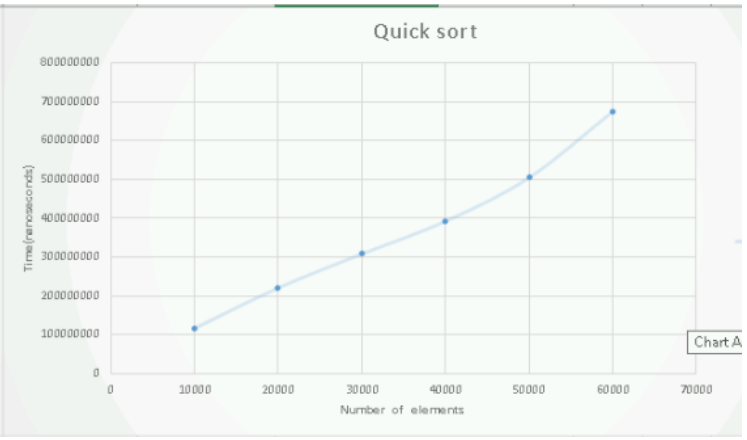
	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
6000	110387200	1831412700	1775742100
12000	121272100	6838189800	5596905100
18000	162504700	17584371800	12776311200
24000	246077000	28141380700	21378091000
30000	293437200	44427978200	31326326100
36000	507758000	63459753400	42657508800

	time(nanoseconds)		
sizeOfArray	RandomArray	SortedArray	UnsortedArray
10000	115961300	6170723900	4068472100
20000	220283200	20287920600	15302162300
30000	307716800	44247257700	31726586900
40000	391875000	77022531500	48154603800
50000	504042200	1,1958E+11	74604038300
60000	674407300	1,81047E+11	1,0143E+11









Thus, we can observe that for random data the graph is more vivid when the method is applied to the bigger input.

2. Searching algorithms implemented

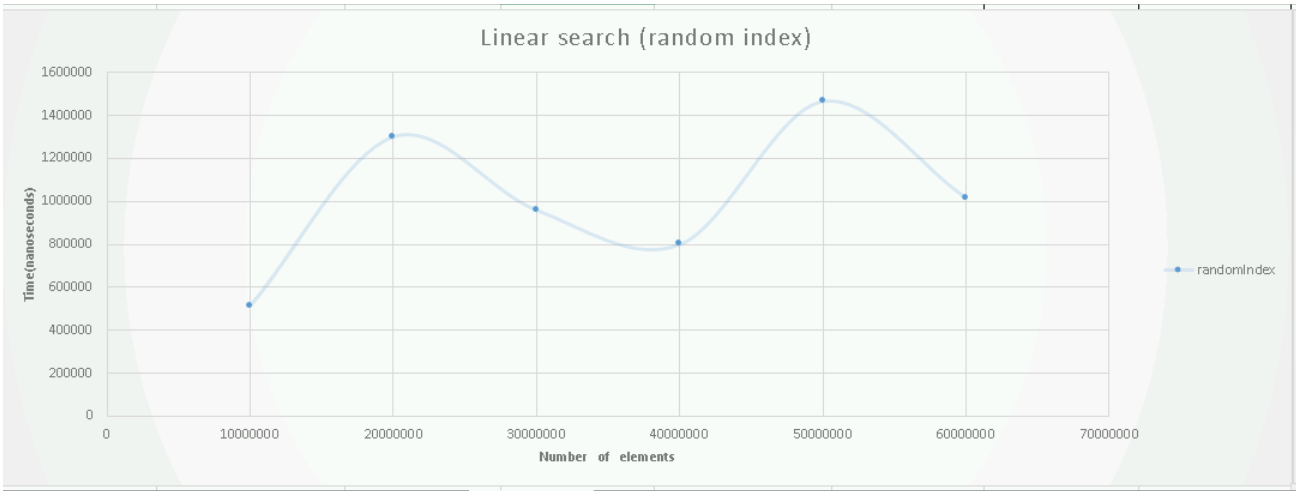
2.1. Linear Search

Method:

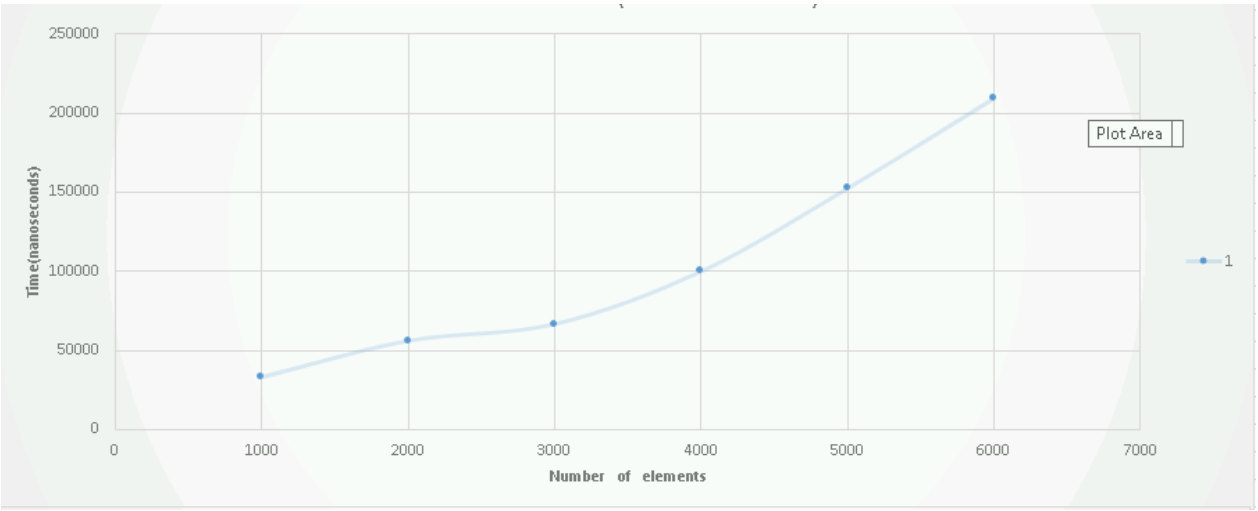
Starting from the first element of an array, one by one the searched element is compared to the current element of the array. If they are equal, the index of the current element is returned.

Time Complexity: O(n) - pessimistic and average.

	time(nanoseconds)
sizeOfArray	randomIndex
10000000	515600
20000000	1301400
30000000	961200
40000000	800700
50000000	1466000
60000000	1017900

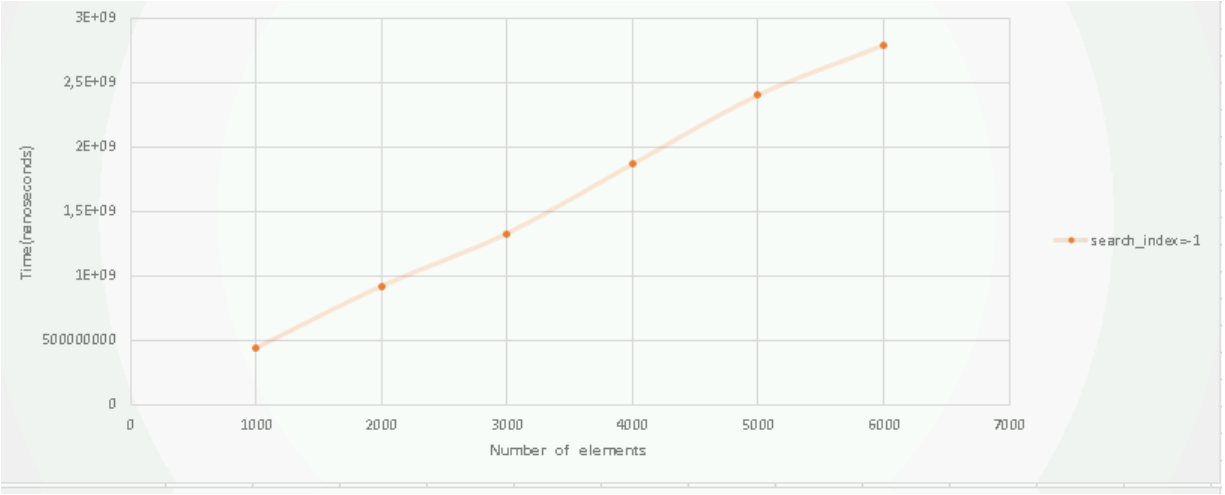


Thus, we obtain the following result when using random indexes. To observe linear dependence, one should investigate pessimistic time complexity of the method. For that purpose, let's assume (simulate) that there is no match in the sequence and the function, after going through all of the elements, returns -1.



One can notice that for the bigger size of array linear dependence can be observed more explicitly:

sizeOfArray	search_index=-1
10000000	439986200
20000000	921094200
30000000	1330258300
40000000	1869501500
50000000	2405013600
60000000	2793330400



2.2. Binary Search

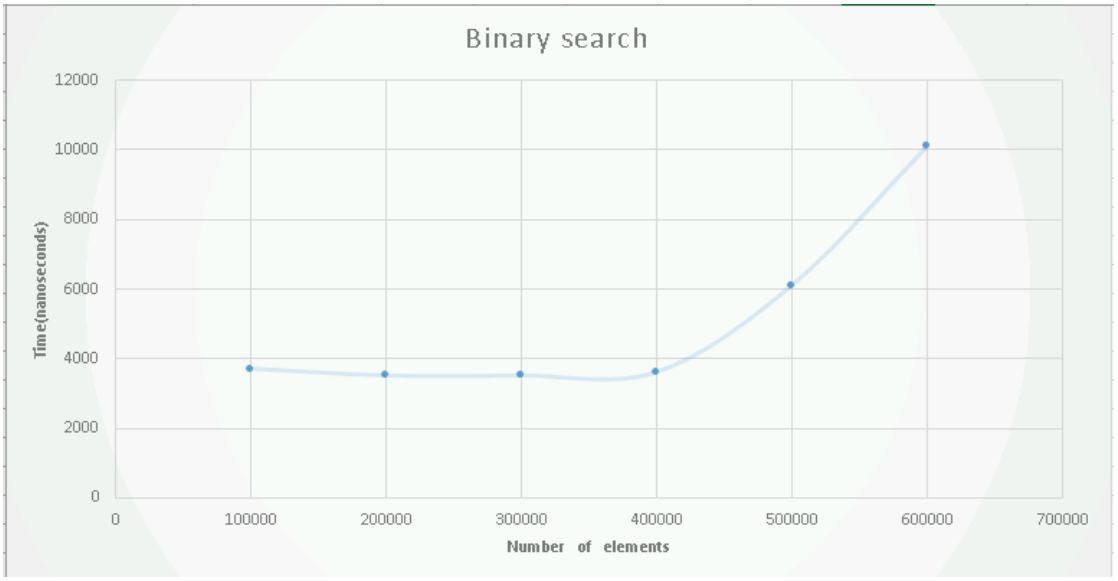
Method:

A sorted array is repeatedly divided in half. Firstly, the whole array is divided, then it is checked whether the value searched is less than or more than the

middle element of the interval. The search is continued within the interval of the presumed location of the element searched. Thus, the searched interval is narrowed by half. The operation is repeated until the value is found or the interval is empty.

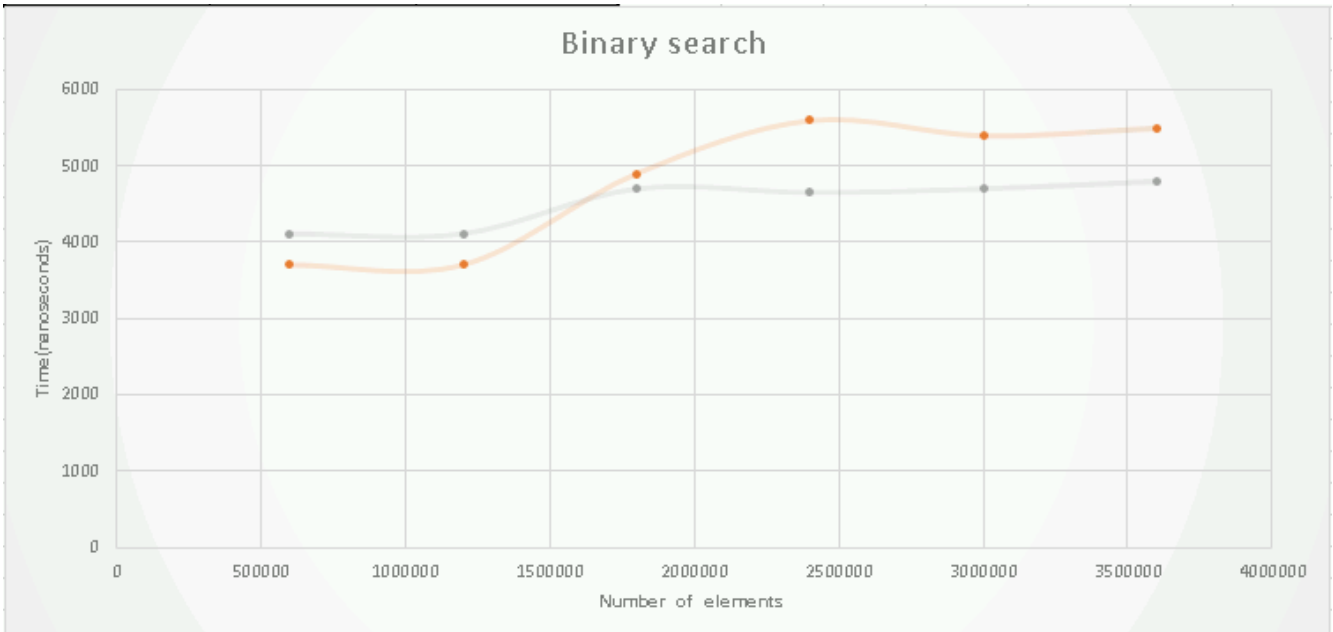
Time Complexity: O(log n) - pessimistic.

sizeOfArray	time(nanoseconds)
100000	3700
200000	3500
300000	3500
400000	3600
500000	6100
600000	10100



On the graph below, the logarithmic dependence is more explicit due to the larger scale of the data used:

sizeOfArray	time(nanoseconds)	
600000	3700	4100
1200000	3700	4100
1800000	4900	4700
2400000	5600	4650
3000000	5400	4700
3600000	5500	4800



3. Project structure

The project is broken into 4 folders:
- “1_sources” containing all project files.
- “2_exe” containing the MeasuringTimeComplexity.exe file.
- “3_example_data” – empty.
- “4_description” containing description of the project as well as .xlsx file, containing the analysis of the experiments’ results.

4.Program usage

To use the program with default parameters, run .exe file from the “2_exe” folder. To launch the program with custom parameters, one will have to change them in MeasurinTimeComplexity.cpp in the “1_sources” folder.

```
E:\Algorithms and data structures\MeasuringTimeComplexity\1_sources\x64\Debug\MeasuringTimeComplex...
        >>>>>> QUICK SORT DEMO <<<<<<

Quick sort. Test #1.
       -size of array: 10000
       -random array sort time: 115961300 nanoseconds
       -sorted array sort time: 6170723900 nanoseconds
       -reversed array sort time: 4068472100 nanoseconds


Quick sort. Test #2.
       -size of array: 20000
       -random array sort time: 220283200 nanoseconds
       -sorted array sort time: 20287920600 nanoseconds
       -reversed array sort time: 15302162300 nanoseconds


Quick sort. Test #3.
       -size of array: 30000
       -random array sort time: 307716800 nanoseconds
       -sorted array sort time: 44247257700 nanoseconds
       -reversed array sort time: 31726586900 nanoseconds


Quick sort. Test #4.
       -size of array: 40000
       -random array sort time: 391875000 nanoseconds
       -sorted array sort time: 77022531500 nanoseconds
       -reversed array sort time: 48154603800 nanoseconds
```

To change the maximum possible number occurring in the arrays created, the variable
MAX_RANDOM_VALUE should be referred to.
RANDOM_PRECISION_NUMBERS specifies the amount of digits after the floating point.
To set the size of the initial array the variable ARRAY_SIZE is to be changed.
COUNT_OF_TESTS lets the user set the amount of tests run for each algorithm.
Every test, following after the first one, increases the ARRAY_SIZE by the factor equal to the number of the test running (e.g. TEST 3 will investigate the array of size that is equal to 3*ARRAY_SIZE of the initial array.

MeasuringTimeComplexity.cpp

```cpp
#include <chrono>
#include <iostream>
#include <vector>
#include <queue>

using namespace std;
using namespace std::chrono;


const int MAX_RANDOM_VALUE = 1000000;
const int RANDOM_PRECISION_NUMBERS = 6;
const int ARRAY_SIZE = 10000;
const int COUNT_OF_TESTS = 10;

template <typename TElement>
TElement* generate_array(int size, TElement(*generate)()) {
    TElement* buffer = new TElement[size];

    for (int i = 0; i < size; i++) {
        buffer[i] = generate();
    }

    return buffer;
}

int int_number_generator(int max) {
    return rand() % max;
}

float float_number_generator() {
    int precision = pow(10, RANDOM_PRECISION_NUMBERS);
    return rand() % (MAX_RANDOM_VALUE * precision) / (1.0 * precision);
```

Conclusion

Overall, the complexity obtained via investigation of the above algorithms coincides with the one claimed. However, for most of the algorithms the dependence gets more explicit with the growth of the input data. Thus, for smaller input data it may not be obvious with the graph being illegible, whereas on larger scales the function will depict a dependence that is more vivid.

Sources used

The information presented as well as the algorithms used were implemented and presented on the base of the materials given during the lecture, as well as where modified and adapted for the needs of the experiment.