Moldova State University

Faculty of Mathematics and Informatics

Department of Informatics

Elementary sorting algorithms

Laboratory Report I

Author Ciobanu Stanislav

Group I2302

Date 05.10.23

Table of contents

Role of sorting

Sorting algorithms play crucial role in data analysis. Their main goal is to arrange a collection of data in a specific order, to make it easier to analyze and search.

1. **Presentation of data**

   Sorting usually makes presentation of data easy to understand. It allow you to exclude extra data and sort remaining data in comfortable order.

2. **Management of big amount of data**

   Manual working with a lot of data is hard and can cause unlikable errors. That is why this task is better to be done by computer. Sorting is a tool, that allows computer working with these data.

3. **Data analysis**

   Analyzing data is faster and easier when data are ordered and similar. Sorting's role is to make data be prepared for analysis.

Description of sorting algorithms

1. **Bubble sort**
   We start with an unsorted array ar [ar_0, ar_1, … , ar_{n-1}] with size n and start comparing all the neighboring elements of the array. If $ar_i > ar_{i+1}$ we swap them. We start again when all elements were compared. Doing an iteration, we place the greatest element on it's place. That's why there is no need to compare all elements always. We compare only unsorted elements. For each new iteration from i = 0 to n – 2 we do from j = 0 to n – i – 1 iterations of comparing.
   Complexity of this sort is $O(n^2)$, because number of comparisons and swaps is proportional to n^2.

2. **Selection sort**
   We start with an unsorted array ar { ar_0, ar_1, … , ar_{n-1} } with size n and consistently choose one of the elements. For each chosen element we start finding out where is that element's place. If chosen element $ar_i > ar_{i+1}$ we swap them and continue comparing ar_i with others. We repeat this steps while j = n – 1 > 0.
   Complexity of this sort is $O(n^2)$ in all the cases.

3. **Insertion sort**
   We start with an unsorted array ar { ar_0, ar_1, … , ar_{n-1} } with size n. We separate this array on two sides. Sorted (Left) and unsorted (Right). We start consistently choose each element (Except first) and comparing it to all previous elements. If $ar_i > ar_{i+1}$ we swap them.
   Complexity of this sort is $O(n^2)$ in most of the cases. But in the best cases it is $O(n)$.

4. **Insertion binary search sort**
   We start with an unsorted array ar { ar_0, ar_1, … , ar_{n-1} } with size n. Using binary search, we reduce number of comparisons for finding position of element ar_i. When we found element's position, we just shift some elements to make space for element ar_i. We perform this for n – 1 iterations.

Complexity of this sort is $O(n^2)$ in most of the cases. However it is really effective in work with large arrays.

Pseudocode

Additional functions

```
        function Switch (x1, x2)  // All operations have to be performed with
{                                 // pointers. That's why we don't need to
    int t = *x1;                  // return value
    *x1 = *x2;                    // Other pseudocode examples are
    *x2 = t;                      // presented without use of pointers
}
```

Bubble sort pseudocode

```
        function BubbleSort (ar, size)          //Should return time
{                                               // of processing
    for i = 0 to size do
    {
        for j = 0 to size – 1 – 1 do
        {
            If ar[j] > ar[j + 1] then
                Swap(ar[j], ar[j+1]);
        }
    }
}
```

Selection sort code

```
        function BubbleSort (ar, size)          //Should return time
{                                               // of processing
```

```
    for i = 0 to size do
    {
        for j = 0 to size – 1 – 1 do
        {
            If ar[j] > ar[j + 1] then
                Swap(ar[j], ar[j+1]);
        }
    }
}
```

Insertion sort pseudocode

```
        function InsertionSort (ar, size)            //Should return time
{                                                    // of processing
    Int min;
    for j = 0 to size do
    {
        min = ar[i];
        int t;
        for i = 0 to size do
        {
            if min > ar[i]
            {
                Min = ar[i];
            }
        }
        Swap(ar[j], min);
```

```
        }
}
```

Insertion binary search sort pseudocode

```
        function InsertionBinarySearchSort (ar, size) //Should return time
{                                                    // of processing
    int i, loc, j, k, selected;
    for i = 1 to size do
    {
        j = i – 1;
        Selected = ar[i];
        loc = binarySearch(ar, selected, 0, j);
        While j >= loc
        {
            ar[j + 1] = ar[j]
            j--;
        }
        ar[j + 1] = selected;
    }
}

function BinarySearch (ar, item, low, high) : int
{
    if high <= low
    {
        If item > ar[low]
        {
```

```
                return low + 1;
            }
            Else
            {
                return low;
            }
        }
        int mid = (low + high) / 2;
        if item == ar[mid]
            return mid + 1;
        if item > ar[mid]
            return BinarySearch(ar, item, mid + 1, high);
        return BinarySearch(ar, time, low, mid - 1);
}
```

Code

Additional functions

```
void Swap(int* i, int* j)
{
	int t = *i;
	*i = *j;
	*j = t;
}



void PrintArray(int** ar, int size)
{
	for (int i = 0; i < size; i++)
	{
		std::cout << *ar[i] << std::endl;
	}
	std::cout << "" << std::endl;
}


void RandomizeArray(int** ar, int size)
{
	for (int i = 0; i < size; i++)
	{
		*ar[i] = rand() % 100000;
	}
}
```

Bubble sort code

```
double BubbleSort(int** ar, int size)
{
	clock_t c;
	c = clock();

	for (int i = 0; i < size - 2; i++)
	{
		for (int j = 0; j < size - i - 1; j++)
		{
			if (*ar[j] > *ar[j + 1])
			{
				Swap(ar[j], ar[j + 1]);
			}
		}
	}

	return (float)(clock() - c) / 1000;
}
```

Selection sort code

```
double SelectionSort(int** ar, int size)
{
	clock_t c;
	c = clock();  // Get time when algorithm starts


	int* min;
	for (int j = 0; j < size; j++) // Sorting itself

	{
		min = ar[j];
		int t;

		for (int i = 0 + j; i < size; i++)
		{
			if (*min > *ar[i])
			{
				min = ar[i];
			}
		}
		Swap(ar[j], min);
	}

	return (float)(clock() - c) / 1000; // Returning time of execution
}
```

Insertion sort code

```
double InsertionSort(int** ar, int size)
{
	clock_t c;
	c = clock();  // Get time when algorithm starts

	int i, key, j;
	for (i = 1; i < size; i++) {
		key = *ar[i];
		j = i - 1;

		while (j >= 0 && *ar[j] > key) {
			*ar[j + 1] = *ar[j];
			j = j - 1;
		}
		*ar[j + 1] = key;
	}

	return (float)(clock() - c) / 1000;
}
```

Insertion binary search sort code

```
int binarySearch(int** ar, int item,
	int low, int high)
{
```

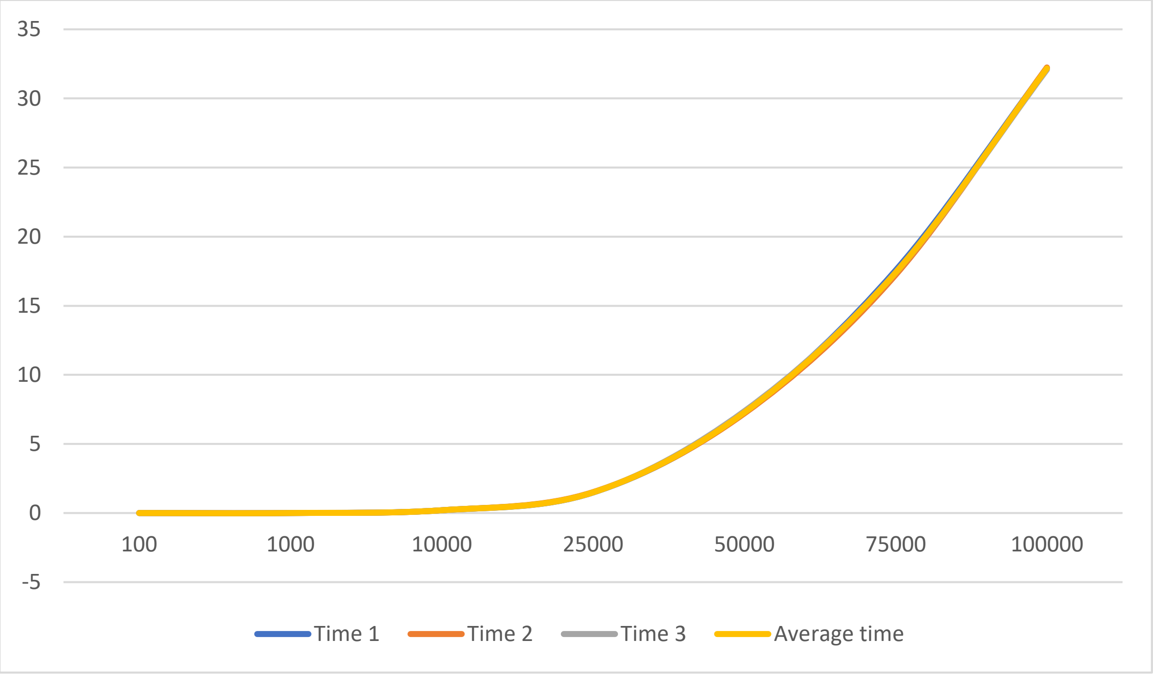
```
	if (high <= low)
		return (item > *ar[low]) ?
		(low + 1) : low;

	int mid = (low + high) / 2;

	if (item == *ar[mid])
		return mid + 1;

	if (item > *ar[mid])
		return binarySearch(ar, item,
			mid + 1, high);
	return binarySearch(ar, item, low,
		mid - 1);
}

double InsertionBinarySort(int** ar, int size)
{
	clock_t c;
	c = clock();

	int i, loc, j, k, selected;

	for (i = 1; i < size; ++i)
	{
		j = i - 1;
		selected = *ar[i];

		loc = binarySearch(ar, selected, 0, j);

		while (j >= loc)
		{
			*ar[j + 1] = *ar[j];
			j--;
		}
		*ar[j + 1] = selected;
	}

	return (float)(clock() - c) / 1000;
}
```

Analysis

1) Bubble sort

Results:

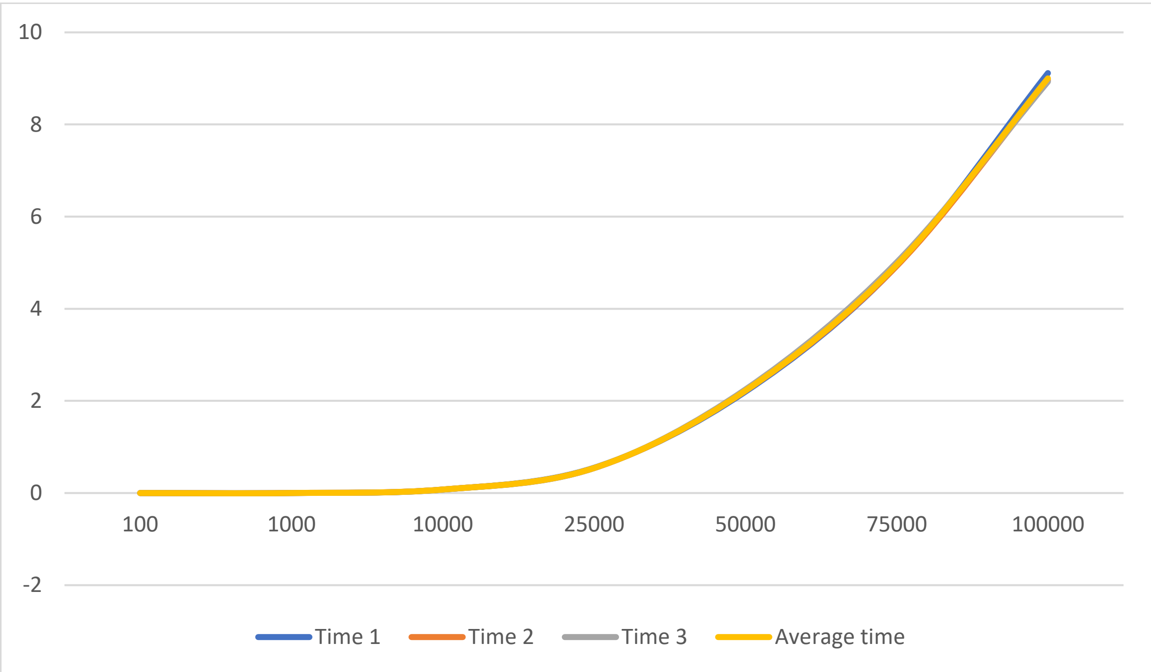
Size of array	Time 1	Time 2	Time 3	Average time
100	0 sec	0 sec	0 sec	0 sec
1000	0.002 sec	0.003 sec	0.002 sec	0.002333 sec
10000	0.201 sec	0.202 sec	0.202 sec	0.201667 sec
25000	1.486 sec	1.512 sec	1.488 sec	1.495333 sec
50000	7.248 sec	7.208 sec	7.364 sec	7.2766667 sec
75000	17.557 sec	17.232 sec	17.392 sec	17.39367 sec
100000	32.204 sec	32.24 sec	32.085 sec	32.17633 sec



2) Selection sort

Results:

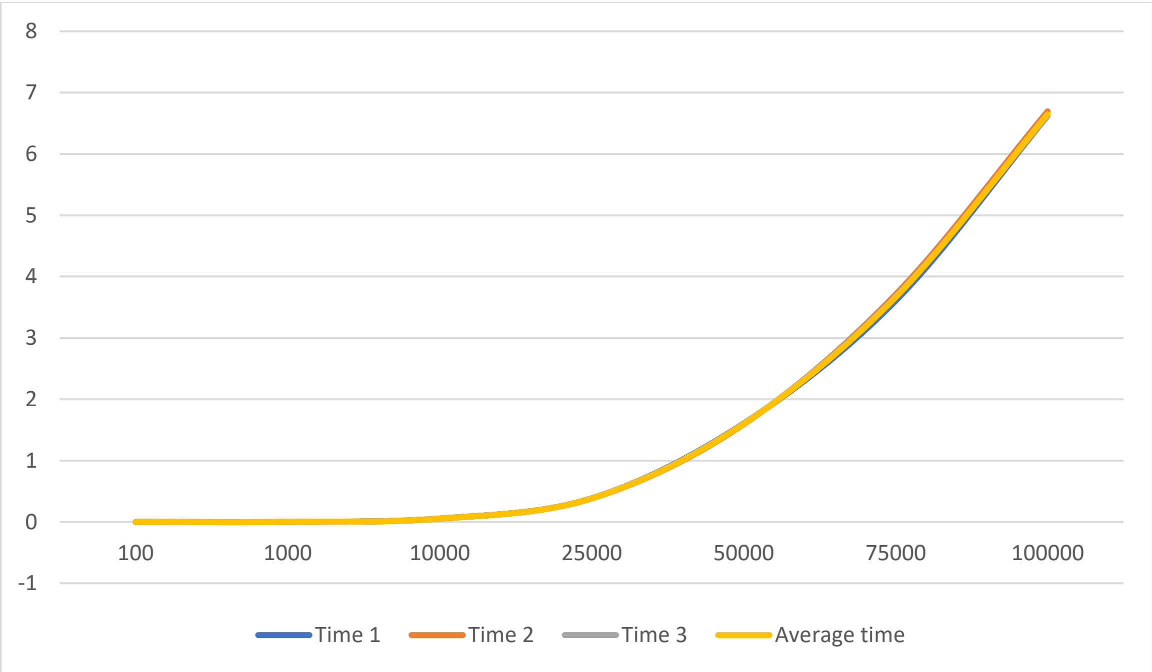
Size of array	Time 1	Time 2	Time 3	Average time
100	0 sec	0 sec	0 sec	0 sec
1000	0.001 sec	0.001 sec	0.000 sec	0.000667 sec
10000	0.077 sec	0.075 sec	0.076 sec	0.076 sec
25000	0.548 sec	0.541 sec	0.541 sec	0.543333 sec
50000	2.199 sec	2.219 sec	2.246 sec	2.221333 sec
75000	4.932 sec	4.927 sec	5.006 sec	4.955 sec
100000	9.116 sec	8.958 sec	8.929 sec	9.001 sec



3) Insertion sort

Results:

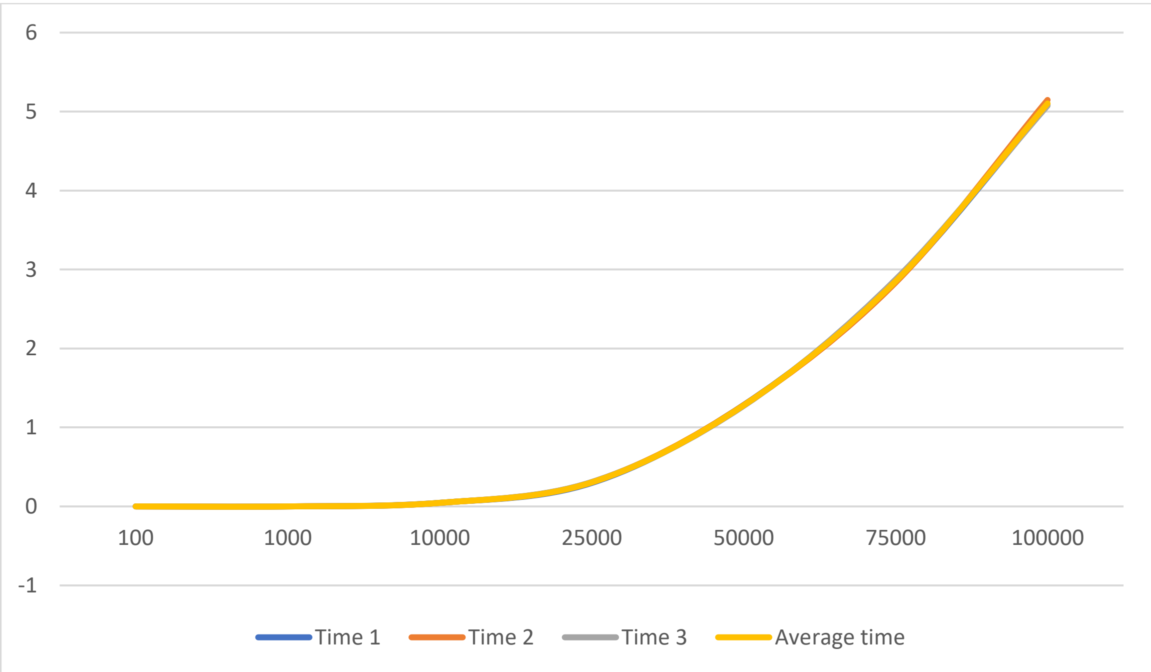
Size of array	Time 1	Time 2	Time 3	Average time
100	0 sec	0 sec	0 sec	0 sec
1000	0.001 sec	0.001 sec	0.001 sec	0.001 sec
10000	0.054 sec	0.056 sec	0.055 sec	0.055 sec
25000	0.386 sec	0.384 sec	0.384 sec	0.384667 sec
50000	1.605 sec	1.592 sec	1.6 sec	1.599 sec
75000	3.605 sec	3.708 sec	3.676 sec	3.663 sec
100000	6.622 sec	6.695 sec	6.626 sec	6.647667 sec



4) Insertion binary search sort

Results:

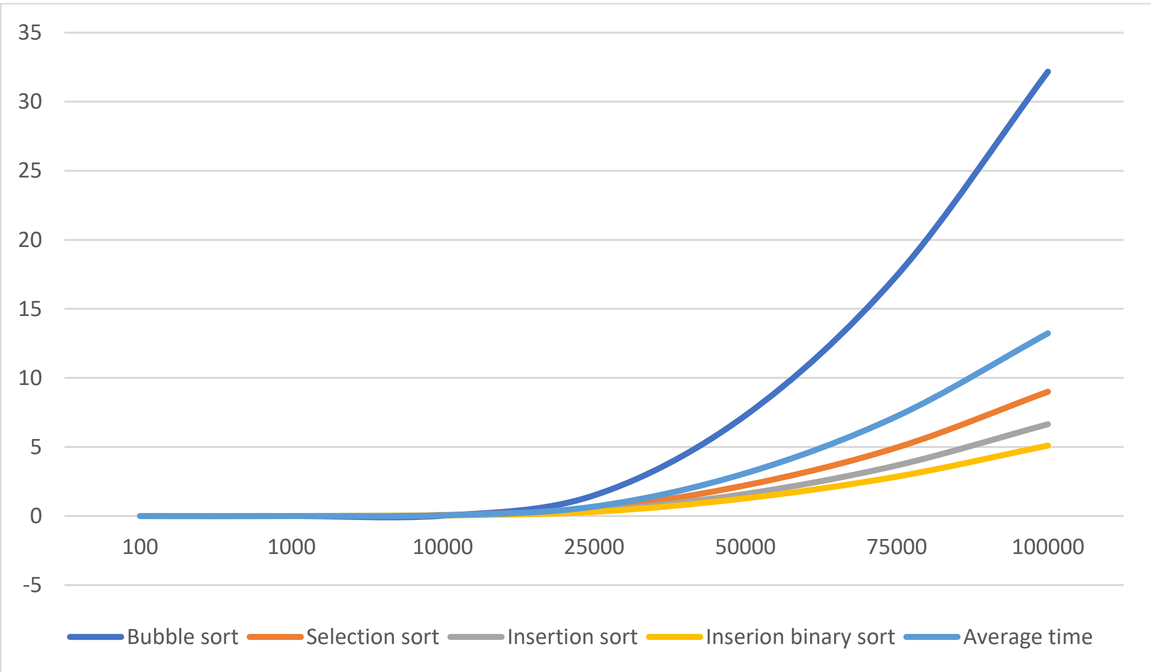
Size of array	Time 1	Time 2	Time 3	Average time
100	0 sec	0 sec	0 sec	0 sec
1000	0 sec	0 sec	0 sec	0 sec
10000	0.046 sec	0.045 sec	0.045 sec	0.045333 sec
25000	0.297 sec	0.307 sec	0.309 sec	0.304333 sec
50000	1.248 sec	1.282 sec	1.273 sec	1.279667 sec
75000	2.844 sec	2.835 sec	2.873 sec	2.850667 sec
100000	5.085 sec	5.147 sec	5.075 sec	5.102333 sec



5) Comparison of average values

Results:

Size of array	Bubble sort	Selection sort	Insertion sort	Insertion binary sort
100	0 sec	0 sec	0 sec	0 sec
1000	0.002333 sec	0.000667 sec	0.001 sec	0 sec
10000	0.201667 sec	0.076 sec	0.055 sec	0.045333 sec
25000	1.495333 sec	0.543333 sec	0.384667 sec	0.304333 sec
50000	7.2766667 sec	2.221333 sec	1.599 sec	1.279667 sec
75000	17.39367 sec	4.955 sec	3.663 sec	2.850667 sec
100000	32.17633 sec	9.001 sec	6.647667 sec	5.102333 sec



As we can see, bubble sort is the most uneffective, when classic and improved with binary search insertion searching algorithm are the most profitable.