Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 2:
Sortarile. Tehnica Devide și Stăpânește.

Elaborated:
st. gr. FAF-211 Kalamaghin Arteom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyze different sorting algorithms.

Tasks

1. Implement at least 4 sorting algorithms;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes

An alternative to mathematical analysis of complexity is empirical analysis. This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established;
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm;
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties);
4. The algorithm is implemented in a programming language;
5. Generating multiple sets of input data;
6. Run the program for each input data set;
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction

In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output.

Formally, the output of any sorting algorithm must satisfy two conditions:

1. The output is in monotonic order (each element is no smaller/larger than the previous element, according to the required order).
2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

For optimum efficiency, the input data should be stored in a data structure which allows random access rather than one that allows only sequential access.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations). Within this laboratory, we will be analyzing these 4 naive algorithms empirically.

Comparison Metric

The time of execution of each algorithm (T(n)) will be used as the comparison metric for this laboratory work. Each algorithm will be tested twice on the same set of input values; these values will be sorted in ascending and descending order, and the average runtime will be considered the execution time of the algorithm.

Input Format

The first two algorithms will be tested on a dynamically allocated array of random integers of size n = {10000, 20000, 30000, … , 100000}. The other two on values of n = {100000, 200000, 300000, … , 1000000}.

IMPLEMENTATIONS

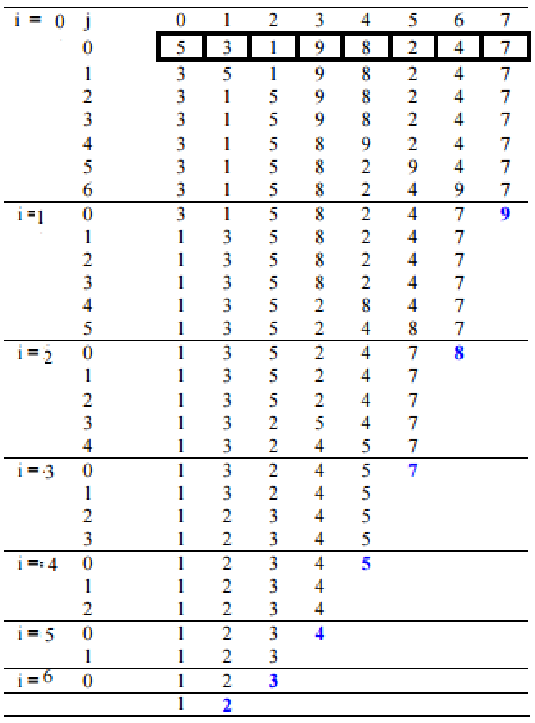
All four algorithms will be implemented in their naive form in C++ and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used. The runtime calculation will be tied to UNIX Epoch Time.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This approach requires us to traverse the sequence of size n - n times, thus the array (random access structure) will be accessed n*n times - **time complexity of the algorithm - O(n^2)**.

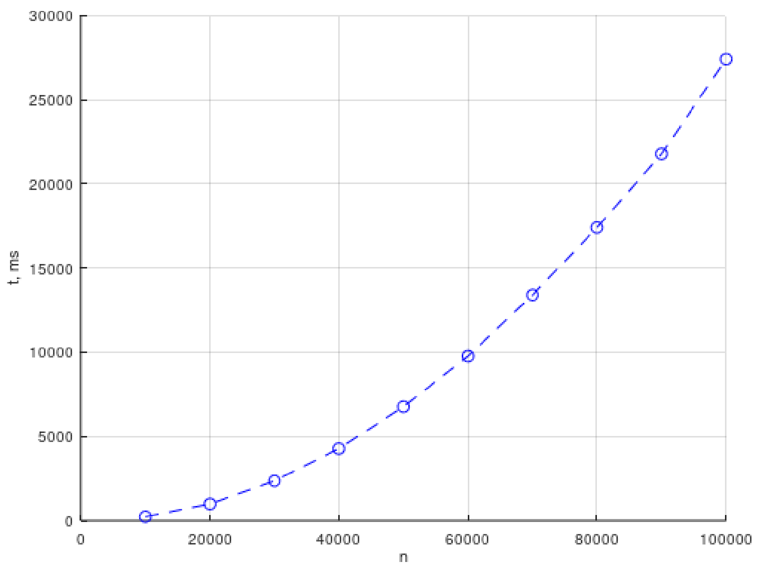
Implementation:

```cpp
class BubbleSort: public Sort {
    public:
        BubbleSort(int order_val) {
            order = order_val;
        };

        void sort(int *arr, int n) {
            for(int i = 0; i < n-1; i++) {
                for(int j = 0; j < n-i-1; j++) {
                    if(!order) {
                        if(*(arr+j) > *(arr+j+1)) {
                            swap(arr+j, arr+j+1);
                        }
                    } else if(order) {
                        if(*(arr+j) < *(arr+j+1)) {
                            swap(arr+j, arr+j+1);
                        }
                    }
                }
            }
        };

        void swap(int *v1, int *v2) {
            int temp = *v1; *v1 = *v2; *v2 = temp;
        };
};
```



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

n	1e4	2e4	3e4	4e4	5e4	6e4	7e4	8e4	9e4	41e5
t, ms	237	997	2365	4280	6774	9774	13413	17411	21791	27411



Polynomial growth of degree 2 is unambiguously expressed.

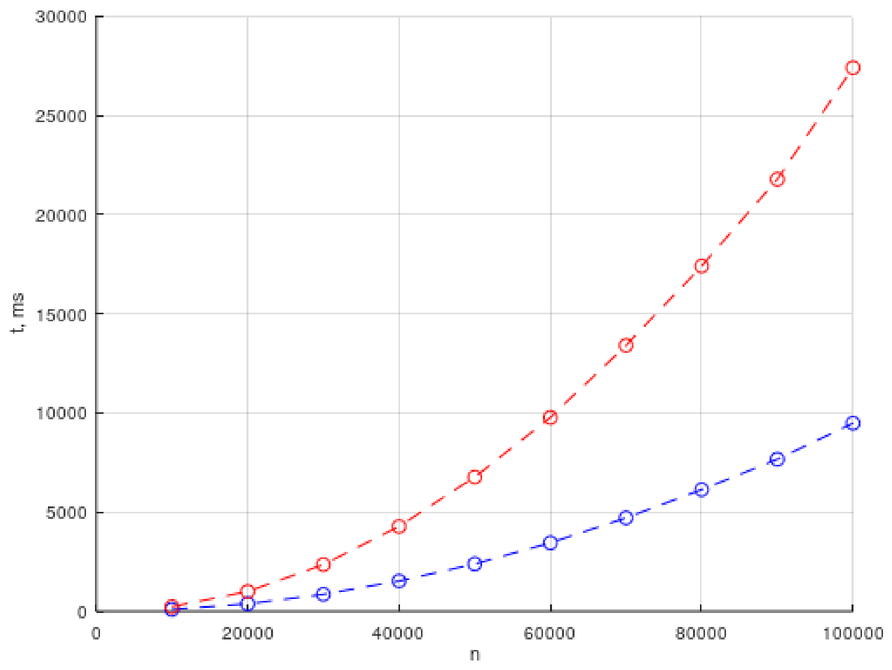
Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons. It is much less efficient on large lists than more advanced algorithms such as merge sort etc. However, insertion sort provides several advantages: Simple implementation; Efficient for (quite) small data sets; More efficient in practice than most other simple quadratic algorithms; Stable; **In-place (only requires a constant amount O(1) of additional memory space)**. **Time complexity for this method is also O(n^2).**

Implementation:

```
void sort(int *arr, int n) {
    int i, j, key;

    for(i = 1; i < n; i++) {
        key = *(arr+i); j = i - 1;

        if(!order) {
            while(j >= 0 && *(arr+j) > key) {
                *(arr+j+1) = *(arr+j); j = j - 1;
            }
        } else if(order) {
            while(j >= 0 && *(arr+j) < key) {
                *(arr+j+1) = *(arr+j); j = j - 1;
            }
        }

        *(arr+j+1) = key;
    }
};
```

n	1e4	2e4	3e4	4e4	5e4	6e4	7e4	8e4	9e4	1e5
t, ms	96	379	853	1533	2397	3457	4714	6136	7675	9487



From this comparison we can clearly see that nevertheless in theory the Bubble Sort (red) and the Insertion Sort (blue) have the same time complexity, in practise the Insertion Sort (as was mentioned above cause of its stability and adaptiveness) is way faster than the Bubble Sort.

Merge Sort

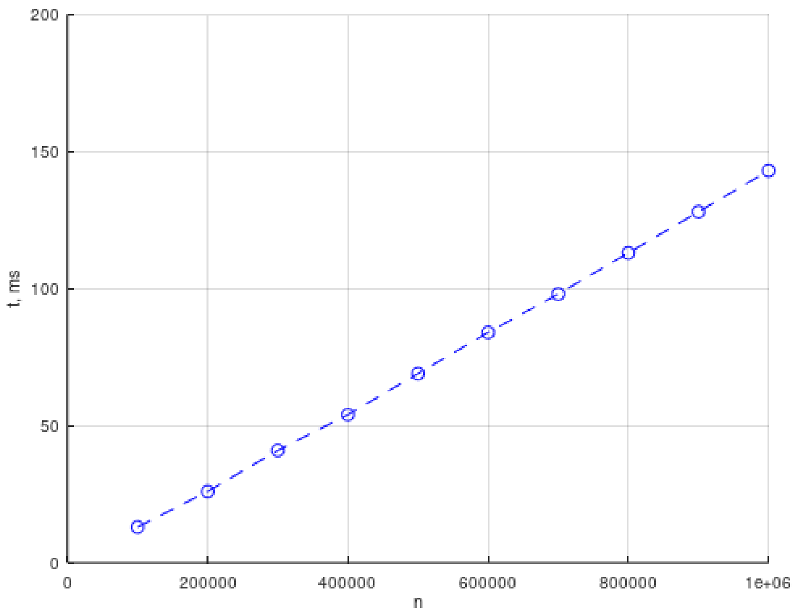
In computer science, divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted. The recursive calls continue until we have to sort n/2 arrays of size 2 - trivial case of determining which of 2 numbers are bigger. **This is a very efficient sorting algorithm - time complexity is O(nlog(n)), BUT it is not an in-place algorithm - requires O(n) of auxiliary space in its basic implementation.**

Implementation:

```
void merge(int *arr, int left, int mid, int right) {
    int subArr1 = mid - left + 1;
    int subArr2 = right - mid;

    int *tmpLeft = (int*)malloc(subArr1*sizeof(int));
    int *tmpRight = (int*)malloc(subArr2*sizeof(int));

    for(int i = 0; i < subArr1; i++) { *(tmpLeft+i) = *(arr+left+i); }
    for(int j = 0; j < subArr2; j++) { *(tmpRight+j) = *(arr+mid+1+j); }

    int idxSubArr1 = 0; int idxSubArr2 = 0; int idxMerged = left;

    while(idxSubArr1 < subArr1 && idxSubArr2 < subArr2) {
        if(!order) {
            if(*(tmpLeft+idxSubArr1) <= *(tmpRight+idxSubArr2)) {
                *(arr+idxMerged) = *(tmpLeft+idxSubArr1);
                idxSubArr1++;
            } else {
                *(arr+idxMerged) = *(tmpRight+idxSubArr2);
                idxSubArr2++;
            }
        } else if(order) {
            if(*(tmpLeft+idxSubArr1) >= *(tmpRight+idxSubArr2)) {
                *(arr+idxMerged) = *(tmpLeft+idxSubArr1);
                idxSubArr1++;
            } else {
                *(arr+idxMerged) = *(tmpRight+idxSubArr2);
                idxSubArr2++;
            }
        }

        idxMerged++;
    }

    while(idxSubArr1 < subArr1) {
        *(arr+idxMerged) = *(tmpLeft+idxSubArr1);
        idxSubArr1++; idxMerged++;
    }

    while(idxSubArr2 < subArr2) {
        *(arr+idxMerged) = *(tmpRight+idxSubArr2);
        idxSubArr2++; idxMerged++;
    }

    delete tmpLeft; delete tmpRight;

};

void sort(int *arr, int begin, int end) {
    if(begin >= end) {
        return;
    }

    int mid = begin + (end - begin) / 2;

    sort(arr, begin, mid);
    sort(arr, mid+1, end);

    merge(arr, begin, mid, end);
};
```

n	1e5	2e5	3e5	4e5	5e5	6e5	7e5	8e5	9e5	1e6
t, ms	13	26	41	54	69	84	98	113	128	143



Quick Sort

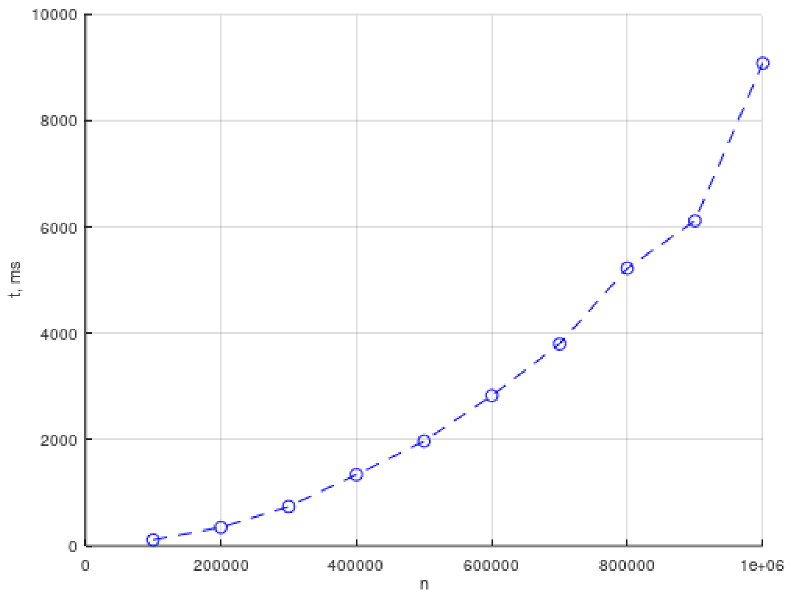
Quicksort is an efficient, general-purpose sorting algorithm. Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting. Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined.

Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved. **Mathematical analysis of quicksort shows that, on average, the algorithm takes O(nlog(n)) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons; auxiliary space - O(log(n)).**

Implementation:

```
int partition(int *arr, int low, int high) {
    int pivot = *(arr+high);
    int i = low - 1;

    for(int j = low; j <= high-1; j++) {
        if(!order) {
            if(*(arr+j) < pivot) {
                i++;
                swap(arr+i, arr+j);
            }
        } else if(order) {
            if(*(arr+j) > pivot) {
                i++;
                swap(arr+i, arr+j);
            }
        }
    }

    swap(arr+i+1, arr+high);

    return i+1;
};

void sort(int *arr, int start, int end) {
    if (start >= end) { return; }

    int p = partition(arr, start, end);
    sort(arr, start, p-1);
    sort(arr, p+1, end);
};

void swap(int *v1, int *v2) {
    int temp = *v1; *v1 = *v2; *v2 = temp;
};
```

n	1e4	2e4	3e4	4e4	5e4	6e4	7e4	8e4	9e4	1e5
t, ms	112	348	736	1340	1968	2826	3799	5222	6118	9081



CONCLUSION

Through Empirical Analysis, within this paper, four sorting algorithms have been tested in their efficiency at both their providing of correct results , as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

Thus, the results for each algorithm will be summarized below in order of increasing performance.

The Bubble Sort showed the worst results, but it is straightforward, easy and fast to implement so could be used to sort relatively small sets of data.

The Insertion Sort has an impressive performance on a small scale and also is relatively easy to implement, so it could be recommended to boost the algorithms based on the D&C technique: for relatively small partitions stop recursive calls before reaching the trivial case and apply the Insertion Sort.

Although usually the Quick Sort shows better performance than the Merge Sort, this particular implementation wasn't very stable and did not use the most efficient pivot finding approaches, so its runtime was larger than the Merge Sort, but nevertheless it came out to be applicable for big sets of data.

During this laboratory work the Merge Sort manifested the best performance. Personally, I find it easier and more straightforward to implement among efficient algorithms; however, the undeniable disadvantage is the relatively big auxiliary space it requires.

* On the next page you can see the format in which I got the output to the console. All the source code you can find by accessing this link: *https://github.com/Starlight-Crusader/AA-Lab-II*.

```
+====================+
RESULTS FOR N = 10000:
BUBBLE SORT: 381.5 ms
INSERTION SORT: 156 ms
+====================+
RESULTS FOR N = 20000:
BUBBLE SORT: 1669 ms
INSERTION SORT: 624 ms
+====================+
...
+====================+
RESULTS FOR N = 50000:
BUBBLE SORT: 6707.5 ms
INSERTION SORT: 2370.5 ms
+====================+
RESULTS FOR N = 60000:
BUBBLE SORT: 9759.5 ms
INSERTION SORT: 3417 ms
+====================+
...
+====================+
RESULTS FOR N = 80000:
BUBBLE SORT: 17485.5 ms
INSERTION SORT: 6190 ms
+====================+
RESULTS FOR N = 90000:
BUBBLE SORT: 22200 ms
INSERTION SORT: 7714 ms
+====================+
...
+====================+
RESULTS FOR N = 100000:
MERGE SORT: 12.5 ms
QUICK SORT: 103.5 ms
+====================+
...
+====================+
RESULTS FOR N = 300000:
MERGE SORT: 40.5 ms
QUICK SORT: 859 ms
+====================+
RESULTS FOR N = 400000:
MERGE SORT: 54 ms
QUICK SORT: 1457 ms
+====================+
RESULTS FOR N = 500000:
MERGE SORT: 68.5 ms
QUICK SORT: 1955.5 ms
+====================+
RESULTS FOR N = 600000:
MERGE SORT: 83.5 ms
QUICK SORT: 3270 ms
+====================+
...
+====================+
RESULTS FOR N = 800000:
MERGE SORT: 112.5 ms
QUICK SORT: 5701.5 ms
+====================+
RESULTS FOR N = 900000:
MERGE SORT: 127.5 ms
QUICK SORT: 6094.5 ms
+====================+
RESULTS FOR N = 1000000:
MERGE SORT: 142 ms
QUICK SORT: 8740.5 ms
+====================+
```