ISSN 2181-8622

Manufacturing technology problems

Scientific and Technical Journal Namangan Institute of Engineering and Technology

INDEX COPERNICUS INTERNATIONAL

Google Scholar

Crossref doi

SLIB.UZ Scientific library of Uzbekistan

Volume 9
Issue 1
2024

ANALYSIS OF USE OF SORTING ALGORITHMS IN DATA PROCESSING

TUYCHIBAEV HAMIDULLO
Master student of Namangan Engineering-Construction Institute, Namangan, Uzbekistan
Tel: (0890) 067-2273, E-mail: tuychibaevhamidullo1999@gmail.com

Abstract. This article aims to identify the most efficient sorting algorithm for solving array problems in the programming language C++. By analyzing existing sorting algorithms and conducting performance tests, we can determine the algorithm that best meets the requirements based on the analysis of test results. Analyzing the processor load data obtained during the execution of the test tasks performed using the bubble-sort method and the selection method; we can consider the selection-sorting algorithm as the most optimal algorithm for specific test tasks. Tasks performed on it, the processor used the least resources. It is not possible to speak for sure about the efficiency of this method, since the number of used test tasks was relatively small, but the obtained experimental results allow us to conclude that the selection-sorting algorithm is more efficient in terms of the use of processor resources. After careful analysis and performance testing, we have successfully identified the most efficient sorting algorithm for solving array tasks in C++. By choosing this algorithm, developers can optimize their code and improve the overall performance of their applications. It is important to consider specific requirements and constraints when choosing a sorting algorithm, but this study provides a solid basis for making an informed decision.

Keywords: programming, programming languages, data structures, sorting algorithms, data, testing tasks, the results of test tasks, optimization, processor, the efficiency of the method.

Introduction. Correct and effective use of sorting algorithms in data processing is very important. When working with a large amount of data, the correctly chosen method ensures fast and efficient operation of the software product. In addition to choosing a programming language for the qualitative formation of algorithmic thinking and obtaining competencies in the specialty, the students of the specialized specialty are taught the basics of algorithmization and various algorithms for solving problems. The most common and important are data sorting algorithms, which are often used in solving computational problems, including working with data arrays. Today, there are several data sorting algorithms. The most common and most used ones include Bubble Sort Algorithm, Quick Sort Algorithm, Shell Sort Method, and Selection Sort Algorithm.

Research methodology. The purpose of this work is to analyze two sorting algorithms ("bubble sort" algorithm and selection sort algorithm) and determine the most optimal one for solving the tasks of computing data arrays in the C++ programming language. In order to evaluate the effectiveness of the selected algorithms ("bubble" and selection), it was decided to use them in solving a number of test tasks. In addition, based on the obtained results of the processor load in the calculation of these tasks, a conclusion was drawn about the effectiveness of a certain sorting algorithm.

Analysis and results. The following tasks were selected as test tasks:

Task 1. Given a one-dimensional array. Sort the array in ascending order.

Task 2. Sort a two-dimensional array in ascending order.

The basic code structure for Task 1 using the bubble sort algorithm is as follows:

```
int main(int argc, char* argv[])
{
        srand(time(NULL));
```

```
cout << " Enter the size of the array: ";
int D;
cin >> D; // Entering the size of a function
int *massiv = new int [D];
for (int i = 0; i < D; i++) // Filling an array with random data
{
	massiv[i] = rand() % 100 + 1;
	cout << setw(2) << massiv[i] << " ";
}
cout << "\n\n";
cout << " Sorted array: "<<endl;
for (int i=D-1;i>0;i--) // Bubble-sorting
{
	for (int j=0;j<i;j++)
	{
		if(massiv[j]>massiv[j+1])
		{
			int t=massiv[j];
			massiv[j]=massiv[j+1];
			massiv[j+1]=t;
		}
	}
}
for (int i = 0; i < D; i++)// Array output
{
	cout << setw(2) << massiv[i] << " ";
}
cout << "\n\n";
system("pause");
return 0;
}
```

The following indicators of processor load were obtained when executing the program code of task 1, written in the C++ programming language, using the above-mentioned bubble-sorting algorithm (Fig. 1):

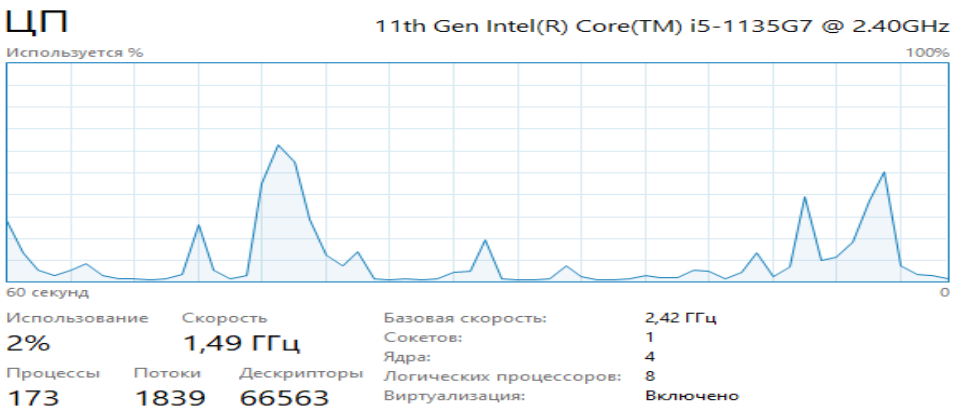

Figure 1. CPU load when performing task No. 1 using the sorting algorithm via the bubble-sorting algorithm.

The text of the program for the implementation of the first task using the sorting algorithm with the selection method:

```
void choicesSort(int*, int);
int main(int argc, char* argv[])
{
	srand(time(NULL));
	cout << " Enter the size of the array: ";
	int D;
	cin >> D; // Entering the size of a function
	int *massiv = new int [D];
	for (int i = 0; i < D; i++) // Filling an array with random data
	{
		massiv[i] = rand() % 100 + 1;
		cout << setw(2) << massiv[i] << " ";
	}
	cout << "\n\n";
	choicesSort(massiv, D); // Calling the sort function
	for (int i = 0; i < D; i++)// Display an array on the screen
	{
		cout << setw(2) << massiv[i] << " ";
	}
	cout << "\n";
	system("pause");
	return 0;
}
void choicesSort(int* arrayPtr, int length_array)// Implementation of the selection function
{
	for (int repeat_counter = 0; repeat_counter < length_array; repeat_counter++)
```

```
    {
        int temp = arrayPtr[0];
        for (int element_counter =
            repeat_counter + 1; element_counter <
            length_array; element_counter++)
            {
                if (arrayPtr[repeat_counter] > arrayPtr[element_counter])
                {
                    temp = arrayPtr[repeat_counter];
                    arrayPtr[repeat_counter] =
                    arrayPtr[element_counter];  arrayPtr[element_counter]
                    = temp;
                }
            }
     }
 }
```

The following processor load indicators were obtained when executing the program code of the first task written using the sorting algorithm with the selection method (Fig. 2).

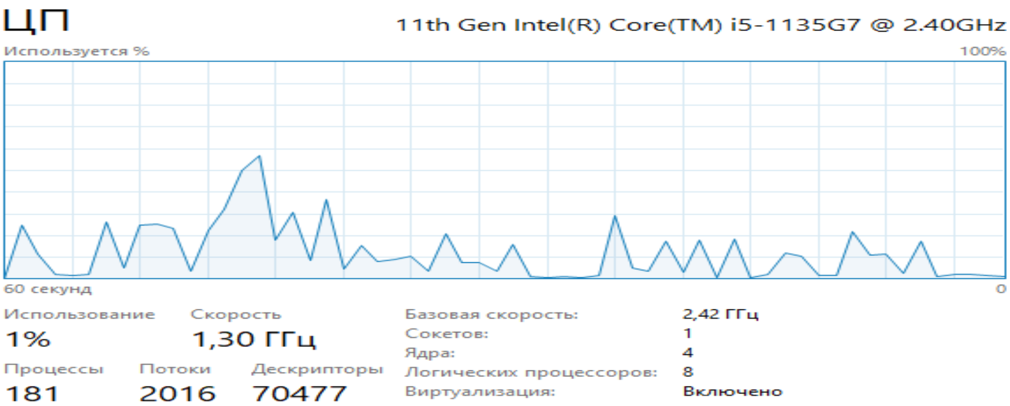

Figure 2. CPU load when performing task No. 1 using the sorting algorithm via the selection method.

Based on the obtained data, it is possible to construct a graph of the processor load when calculating task No. 1 using different sorting algorithms (Fig. 3). Based on the obtained data, we can conclude that the selection method algorithm is more suitable for solving task 1.

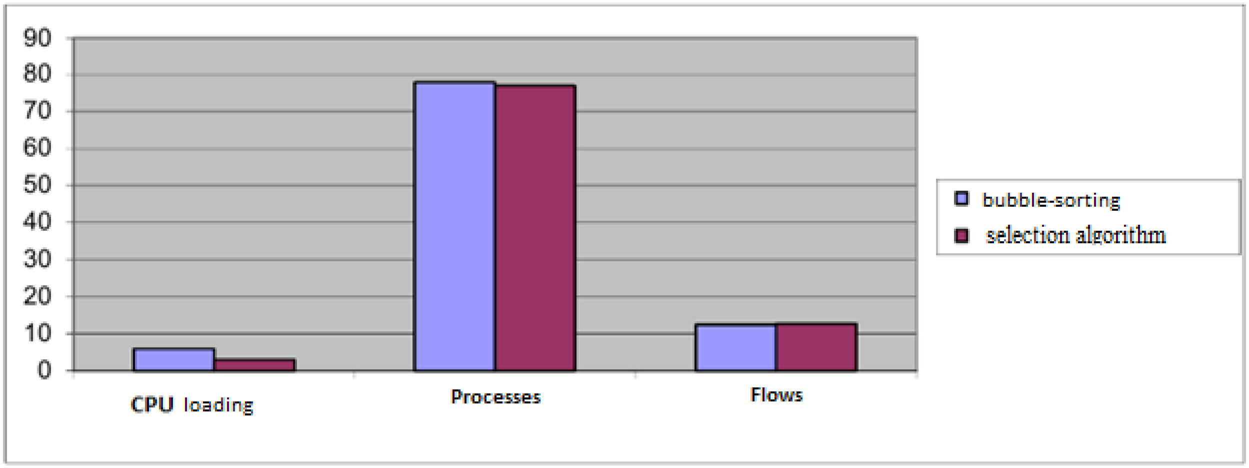

Figure 3. Comparison graph of CPU load when computing task No. 1 using different sorting algorithms.

Two sorting algorithms were also used in the implementation of test task No. 2: sorting by "bubble" and selection methods. Below is the basic code structure for Task 2 using the bubble-sorting algorithm:

```
int main()
{
	const int D = 5;
	unsigned short k;
	int i, j, i_min, sum;
	int massiv[D][D];
	srand(time(NULL));
	for(i = 0; i < D; ++i) // Filling a two-dimensional array with random data
	{
		for(j = 0; j < D; ++j)
		{
			massiv[i][j] = rand()%50 + 1;
		}
	}
	cout<<" Initial matrix: "<<endl;
	for(i = 0; i < D; ++i) // Display the initial matrix on the screen
	{
		for(j = 0; j < D; ++j)
		{
			cout << massiv[i][j] << '\t';
		}
		cout << endl;
	}
	cout << " Sorted array: "<<endl;
```

```
    for(int c=0;c<=(i*j);c++)// A bubble to sort the matrix in ascending order
    {
        for(int i2=0;i2<i;i2++)
        {
            for(int j2=0;j2<j-1;j2++)
            {
                if(massiv[i2][j2]>massiv[i2][j2+1])
                {
                    int temp=massiv[i2][j2];
                    massiv[i2][j2]=massiv[i2][j2+1];
                    massiv[i2][j2+1]=temp;
                }
            }
        }
        for(int x=0;x<j;x++)
        {
            for(int z=0;z<i-1;z++)
            {
                if(massiv[z][x]>massiv[z][x+1])
                {
                    int temp=massiv[z][x];
                    massiv[z][x]=massiv[z][x+1];
                    massiv[z][x+1]=temp;
                }
            }
        }
    }
    for(int i2=0;i2<i;i2++) // Display the matrix on the screen
    {
        for(int j2=0;j2<j;j2++)
        {
            cout<<massiv[i2][j2]<<"\t";
        }
        cout<<"\n";
    }
    return 0;
}
```

The following indicators of processor load were obtained when executing the program code of task 2 written in the C++ programming language using the bubble-sorting algorithm (Fig. 4).

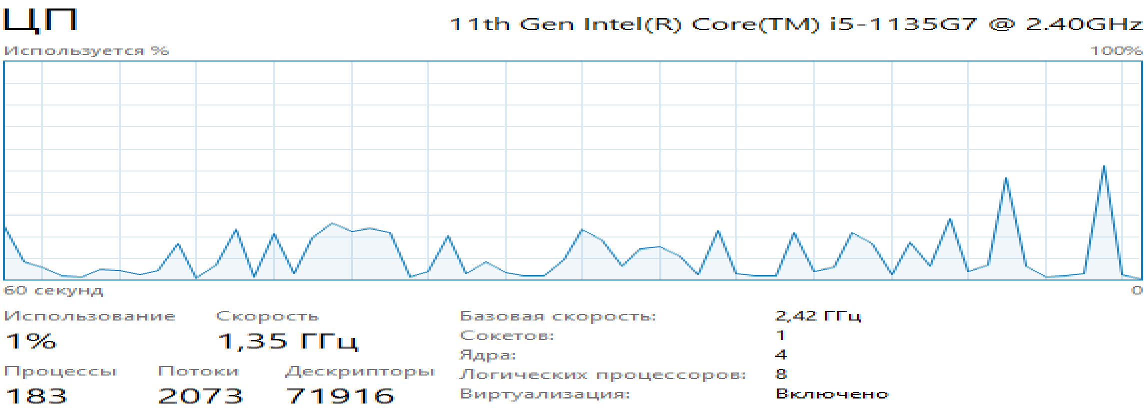

Figure 4. CPU load when performing task No. 2 using the bubble-sorting algorithm.

The basic code structure for the implementation of task No. 2 using the choice sorting algorithm:

```
int main()
{
	const int D = 5;
	unsigned short k;
	int i, j, i_min, sum;
	int massiv[D][D];
	srand(time(NULL));
	for(i=0;i<D;++i)// Filling a two-dimensional array with random data
	{
		for(j = 0; j < D; ++j)
		{
			massiv[i][j] = rand()%50 + 1;
		}
	}
	cout<<" Initial matrix: "<<endl;
	for(i = 0; i < D; ++i) // Display the initial matrix
	{
		for(j = 0; j < D; ++j)
		{
			cout << massiv[i][j] << '\t';
		}
		cout << endl;
	}
	int t1;
	unsigned short minind,
	mm = D - 1;
	for(i=0;i<D;i++)// Sorting the matrix in ascending order
	{
```

```
        for (j = 0; j < mm; j++)
        {
            minind = j;
            for (k = j+1; k < D; k++)
            if (massiv[i][minind] > massiv[i][k])
                minind = k;
            t1 = massiv[i][j];
            massiv[i][j] = massiv[i][minind];
            massiv[i][minind] = t1;
        }
    }
    cout << " Sorted array: " <<endl;
    for(i  =  0;  i  <  D;  ++i)  // Display a sorted matrix
    {

        for(j = 0; j < D; ++j)
        {
            cout << massiv[i][j] << '\t';
        }
        cout << endl;
    }
    return 0;
}
```

The following indicators of the processor load were obtained when executing the program code of the second task written using the sorting algorithm by the selection method (Fig. 5).

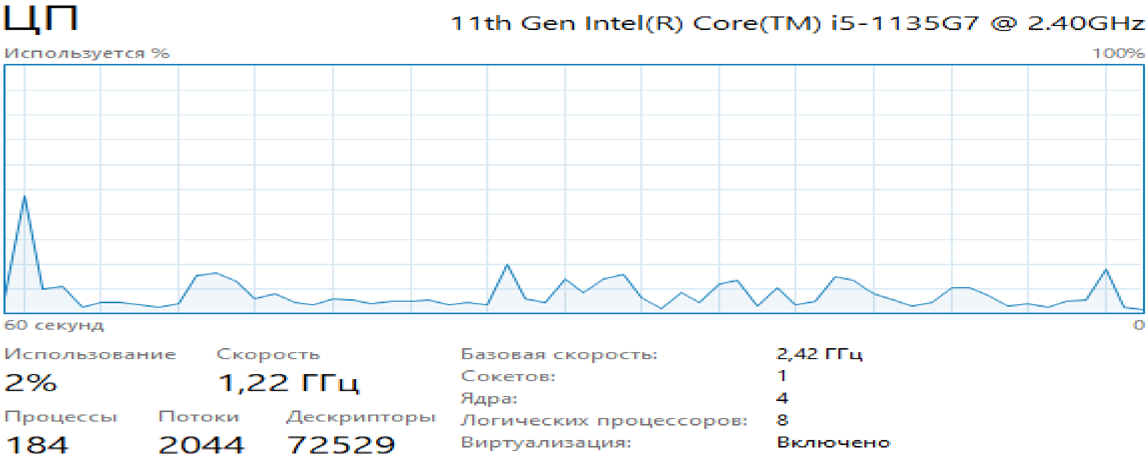

Figure 5. CPU load when performing task No. 2 using the selection algorithm.

Based on the obtained data, we can construct a graph of the processor load when calculating test task No. 2 using different sorting algorithms (Fig. 6).

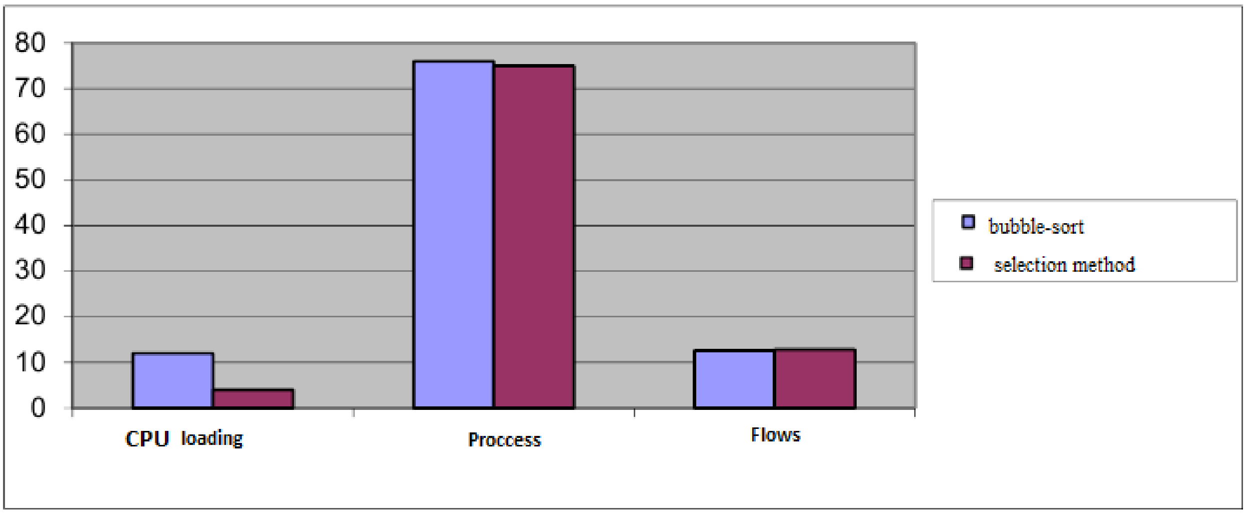

Figure 6. Comparison graph of CPU load when computing task No. 2 using different sorting algorithms.

Based on the obtained data, we can conclude that the selection method is the most effective for solving this task.

REFERENCES

1. Shildt, G. Self-instruction manual C++ / trans. from English – St. Petersburg: BHV-St. Petersburg, 1998. - 620 p.
2. Jamsa, K. Learning to program in C++ / translated. from English – M.: Mir, 1997. - 320 p.
3. Stephen Prata C++ programming language. Lectures and exercises. 6th ed. - M.: Williams, 2012.
4. Savitch, W. C++ in examples / trans. from English – M.: EKOM, 1997. - 736 p.
5. Deitel, H. Deitel, P. How to program in C++ / trans. from English – M.: BINOM, 1998. - 1024 p.
6. Rao, S. Master C++ on your own in 21 days. - 7th ed. – M.: Williams, 2013.
7. Stroustrup, Bjorn. C++ programming language. Special edition / trans. from English – M.: Binom, 2011. - 1136 p.
8. Stroustrup, Bjorn. Programming. Principles and practice of using C++ / trans. from English – M.: Williams, 2011. - 1248 p.
9. Laforet, R. Object-oriented programming in C++ / trans. from English – St. Petersburg: Peter, 2012. - 900 p.

CONTENTS