**NATIONAL TECHNICAL UNIVERSITY   
OF UKRAINE "KPI IM. Igor Sikorsky »**

FACULTY OF APPLIED MATHEMATICS

**Department of system programming and specialized computer systems**

**COURSEWORK**

***from the discipline "Data Structures and Algorithms"***

Performed by: Bitlian A.V.

Group: KB-91

Record book number: KV-9102

Admitted to defense

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2nd semester of the 2019/2020 academic year

**NATIONAL TECHNICAL UNIVERSITY   
OF UKRAINE "KPI IM. Igor Sikorsky"**

FACULTY OF APPLIED MATHEMATICS

**Department of system programming and specialized computer systems**

Agreed PROTECTED "\_\_"\_\_\_\_\_\_2020 .

Head of work with evaluation\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_/Marchenko O.I./ \_\_\_\_\_\_\_\_\_ /Marchenko O.I./

***Research on the effectiveness of sorting methods (sorting algorithm No. 5 of the direct selection method,***

***Sorting algorithm No. 8 of the direct selection method,***

***Algorithm of the "quick sort" method (Hoare sort)) on multidimensional arrays***

Executor of the work:\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(signature)

Andrii Vitalyovych Bitlyan

\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020

**TECHNICAL ASSIGNMENT FOR THE COURSE WORK**

**I.** Describe the principle and scheme of operation of each of the researched methods of sorting or searching for a one-dimensional array.

**II.** Compose algorithms for sorting or searching in a multidimensional array using the given methods, according to the option, and write a corresponding program in a programming language.

The program must meet the following requirements:

1. All algorithms must be implemented within ONE program with a dialog interface for choosing testing options and measuring the time of each algorithm.
2. One of the options for starting the program should be the mode of starting time measurement of all algorithms in batch mode, that is, starting all algorithms for all cases and building the resulting table according to the sample below for an array with given geometric dimensions.
3. ( unit ) must be used .
4. The program must have comments for all data structures, procedures and functions, as well as for the main semantic fragments of algorithms.

**III.** Debugging and testing the correctness of the written program.

**IV.** Conduct practical research on the performance of complex algorithms.

**V.** Based on the results of research, make comparative tables according to various characteristics.

To carry out a thorough analysis of the algorithms, it is necessary to perform time measurements and build tables for several arrays with different geometric dimensions, as well as for the standard case of a one-dimensional array. The number of necessary tables for arrays with different geometric dimensions depends on the task of a specific version of the course work and is chosen independently by the student in order to perform a comprehensive and thorough comparative analysis of the given algorithms. Based on the obtained results, graphs can also be constructed for clarity of information presentation.

**VI.** Perform a comparative analysis of the behavior of the given algorithms according to the obtained results:

* for a one-dimensional array relative to the well-known theory;
* for multidimensional arrays relative to the results for a one-dimensional array;
* for given algorithms on multidimensional arrays among themselves;
* to investigate the influence of different geometric dimensions of multidimensional arrays on the behavior of algorithms and their interrelationship;
* for all the above points of the comparative analysis, explain WHY the algorithms in the considered situations behave in this way, and not otherwise.

**VII.** Draw conclusions based on the performed comparative analysis.

**VIII.** The coursework program during its defense MUST be carried with you on an electronic data carrier.

**Option No. 67**

**Task 5**

Arrange the three-dimensional array A rr3D [ P , M , N ] as follows:

rearrange the sections of the array without decreasing the sums of their elements.

**Researched methods and algorithms**

1. Sorting algorithm No. 5 of the direct selection method
2. Sorting algorithm No. 8 of the direct selection method
3. Algorithm of the "quick sort" method (Hoare sort)

**Workaround methods**

As the first stage of sorting, form an additional vector Sum, the length of which is equal to the number of sections and the values of which are the sums of the elements of the corresponding sections. Using the elements of the Sum vector as the sort keys, rearrange the corresponding sections each time the keys need to be rearranged.

**Case studies**

* The elements of the initial array are ordered according to the given characteristic.
* The elements of the initial array are unordered.
* The elements of the initial array are ordered according to the oppositely specified sign.

**Principles of operation and schemes of algorithms**

1) Sorting algorithm No. 5 of the direct selection method

*General Information:*

In this algorithm the array consists of two parts: sorted and unsorted. At the beginning, the sorted one consists of one element (which is at position 0), all others are unsorted.

Working principle:

1. We are looking for an element smaller than the one selected from the sorted part , that is, the minimum element in the unsorted part.
2. If it is found, then we exchange it with the one with which we are comparing, on the first iteration it is the 0 position, on each subsequent position the number increases by one.
3. We repeat steps 1 and 2 until we run out of elements in the unsorted part.

*Scheme :*

The red color is the element with which we compare 0 1 2 3 4 5 6

Blue color – found element (smallest) **1 2 5 3 6 0 4**

Green background – sorted part **0 2 5 3 6 1 4**

**0 1 5 3 6 2 4**

**0 1 2 3 6 5 4**

**0 1 2 3 6 5 4**

**0 1 2 3 4 5 6**

**0 1 2 3 4 5 6**

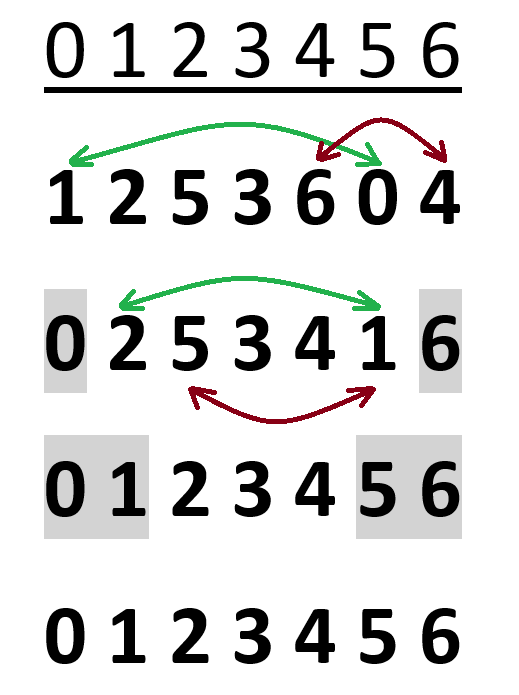
2) Sorting algorithm No. 8 of the direct selection method

*General Information:*

In this algorithm the array consists of two parts: sorted and unsorted. At the beginning, the sorted one consists of one element (which is at position 0), all others are unsorted.

Working principle:

1. We are looking for the minimum and maximum element in the array.
2. We exchange the minimum with the first, the maximum with the last.
3. Now the first and the last are in their places. So, we narrow the search circle - we repeat 1-2 steps for the array from the second to the penultimate and so on. Until the borders cross.

Note.

Green arrows

the exchange of minimal elements is specified.

And red color - maximum.

Exchange of minimal elements

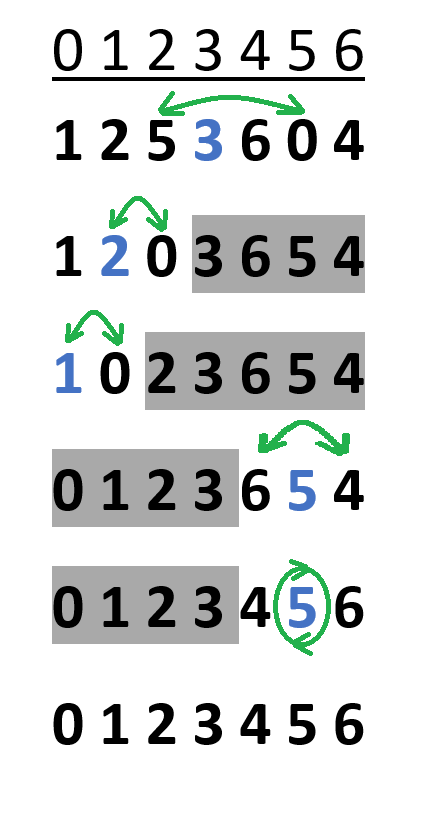
occurs before the exchange of maximums.

3) FAST SORTING (Hoare sort)

This algorithm was created by Professor Hoare C.E.R.

*Working principle:*

1. We choose an arbitrary element from the array, usually, for convenience, we take the central one.
2. We are looking for an element larger than the selected (central) element from left to right.
3. We are looking for an element smaller than the selected (central) one from right to left.
4. If the desired element was found on both sides, then they switch places.
5. If the indices cross, it means that all the numbers to the left of the selected element are less than it, and to the right - more. The cycle ends.
6. We divide the obtained parts again, choose the central element, etc. We repeat steps 1-5.
7. When all parts are equal to one element, the array is sorted.

****

*Scheme :*

The selected element is marked in blue

On the gray background is the part of the array that is not involved in sorting at this stage

**Description of the purpose of procedures and functions**

**Main module**

menu – menu of the dialog interface of the program

All - starting the time measurement of all algorithms in batch mode, i.e. starting all algorithms for all cases and building the resulting table

D3 – calculation of the speed of the algorithm in a three-dimensional array, sorting and rearranging the sections by sum

vector – calculating the speed of the sorting algorithm on a one-dimensional array

**Module fill**

fill\_sorted – filling of a three-dimensional array with numbers strictly in ascending order

fill\_reversed – filling of a three-dimensional array with numbers strictly in descending order

fill\_random – filling a three-dimensional array with random numbers

**sum module**

sum - a function for finding the sum of the cross-section of a three-dimensional array

**Sort module**

Swap is a function for swapping two sections

timeQuickSort - a function for finding the running time of a quick sort algorithm in a three-dimensional array

quickSort - an algorithm for quick sorting of a given task

Select5 - algorithm for selecting number 5 of a given task

Select8 - the algorithm for selecting number 8 of a given task

**Vector module**

Vtime\_quick\_sort - a function for finding the operating time of the quick sort algorithm in a one-dimensional array

Vquick\_sort - a quick sorting algorithm in a one-dimensional array

V Select5 - algorithm for selecting number 5 in a one-dimensional array

V Select8 - algorithm for selecting number 8 in a one-dimensional array

**Program text**

**Main**

//connection of modules

#include "print.h"

#include "fill.h"

#include "sort.h"

#include "sum.h"

#include "vector.h"

using namespace std;

void vector()

{

//array declaration

int\* A;

int N;

cout << "Input size: ";

cin >> N;

A = new int[N];

//choosing how to populate the array

cout << "Choose the option(type the digit):" << endl;

cout << "1. Sorted" << endl;

cout << "2. Random" << endl;

cout << "3. Reversed" << endl;

int options;

cin >> option;

switch (option)

{

//ordered

case 1: for(int i = 0; i < N; i++)

{

A[i] = i;

}

break

//unordered

case 2: for (int i = 0; i < N; i++)

{

A[i] = rand() % N;

}

break

//obviously ordered

case 3: for (int i = 0; i < N; i++)

{

A[i] = N - i;

}

break

default: cout << "Incorrect data!" << endl;

}

//selection of sorting algorithm

cout << "Choose the option(type the digit):" << endl;

cout << "1. Quick sort" << endl;

cout << "2. Select 5" << endl;

cout << "3. Select 8" << endl;

cin >> option;

switch (option)

{

case 1: cout << "time = " << Vtime\_quick\_sort(N, A, 0, N - 1) << endl;

break

case 2: cout << "time = " << VSelect5(A, N) << endl;

break

case 3: cout << "time = " << VSelect8(A, N) << endl;

break

default: cout << "Incorrect data!" << endl;

}

}

void D3()

{

//array declaration

int P, M, N;

int\*\*\* Arr3D;

cout << "Input size:" << endl;

cout << "P: ";

cin >> P;

cout << "M: ";

cin >> M;

cout << "N: ";

cin >> N;

//allocate memory for the array

Arr3D = new int\*\* [P];

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

//choosing how to populate the array

cout << "Choose the option(type the digit):" << endl;

cout << "1. Sorted" << endl;

cout << "2. Random" << endl;

cout << "3. Reversed" << endl;

int options;

cin >> option;

switch (option)

{

//ordered

case 1: fill\_sorted(Arr3D, P, M, N);

break

//unordered

case 2: fill\_random(Arr3D, P, M, N);

break

//obviously ordered

case 3: fill\_reversed(Arr3D, P, M, N);

break

default: cout << "Incorrect data!" << endl;

}

//selection of sorting algorithm

cout << "Choose the option(type the digit):" << endl;

cout << "1. Quick sort" << endl;

cout << "2. Select 5" << endl;

cout << "3. Select 8" << endl;

cin >> option;

switch (option)

{

case 1: cout << "time = " << timeQuickSort(0, P-1, Arr3D, P, M, N) << endl;

break

case 2: cout << "time = " << Select5(Arr3D, P, M, N) << endl;

break

case 3: cout << "time = " << Select8(Arr3D, P, M, N) << endl;

break

default: cout << "Incorrect data!" << endl;

}

//clear memory

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

delete[]Arr3D;

}

void All()

{

int P = 50000, M, N;

int\*\*\* Arr3D;

Arr3D = new int\*\* [P];

//array in which we write data from all cases

int tmp[3][3]{};

//calculate the values and write them into the tmp array

for (int i = 1; i <= 16; i += i)

{

M = i;

N = i;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

tmp[0][0] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[0][1] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[0][2] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int i = 1; i <= 16; i += i)

{

M = i;

N = i;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

tmp[1][0] += Select5(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[1][1] += Select5(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[1][2] += Select5(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int i = 1; i <= 16; i += i)

{

M = i;

N = i;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

tmp[2][0] += Select8(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[2][1] += Select8(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[2][2] += Select8(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

P = 32000;

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = s;

N = r;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

tmp[0][0] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[0][1] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[0][2] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = r;

N = s;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

tmp[0][0] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[0][1] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[0][2] += timeQuickSort(0, P - 1, Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = s;

N = r;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

fill\_sorted(Arr3D, P, M, N);

tmp[1][0] += Select5(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[1][1] += Select5(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[1][2] += Select5(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = r;

N = s;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

fill\_sorted(Arr3D, P, M, N);

tmp[1][0] += Select5(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[1][1] += Select5(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[1][2] += Select5(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = s;

N = r;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

fill\_sorted(Arr3D, P, M, N);

tmp[2][0] += Select8(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[2][1] += Select8(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[2][2] += Select8(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

for (int s = 2, r = 800; s <= 16; s += s, r /= 2)

{

M = r;

N = s;

for (int i = 0; i < P; i++)

Arr3D[i] = new int\*[M];

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

Arr3D[i][j] = new int[N];

fill\_sorted(Arr3D, P, M, N);

fill\_sorted(Arr3D, P, M, N);

tmp[2][0] += Select8(Arr3D, P, M, N);

fill\_random(Arr3D, P, M, N);

tmp[2][1] += Select8(Arr3D, P, M, N);

fill\_reversed(Arr3D, P, M, N);

tmp[2][2] += Select8(Arr3D, P, M, N);

for (int i = 0; i < P; i++)

for (int j = 0; j < M; j++)

delete[]Arr3D[i][j];

for (int i = 0; i < P; i++)

delete[]Arr3D[i];

}

//divide the obtained values by 13 (number of tests) to get the arithmetic mean

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

tmp[i][j] /= 13;

//display the table with the results

cout << "------------------------------------------" << endl;

cout << "| | Sorted | Random | Reversed |" << endl;

cout << "------------------------------------------" << endl;

cout << "| Quick sort |" << setw(5) << tmp[0][0] << setw(4) << "|" << setw(5) << tmp[0][1] << setw(4) << "|"<< setw(6) << tmp[0][2] << setw(5) << " |" << endl;

cout << "------------------------------------------" << endl;

cout << "| Select 5 |" << setw(5) << tmp[1][0] << setw(4) << "|" << setw(5) << tmp[1][1] << setw(4) << "|" << setw(6) << tmp[1][2] << setw(5) << "|" << endl;

cout << "------------------------------------------" << endl;

cout << "| Select 8 |" << setw(5) << tmp[2][0] << setw(4) << "|" << setw(5) << tmp[2][1] << setw(4) << "|" << setw(6) << tmp[2][2] << setw(5) << "|" << endl;

cout << "------------------------------------------" << endl;

delete[]\*\*Arr3D;

}

void menu()

{

int options;

//loop to re-enter data

while (true)

{

cout << "--------------------------------" << endl;

cout << "Choose the option(type the digit):" << endl;

cout << "1. Vector" << endl;

cout << "2. 3D" << endl;

cout << "3. All" << endl;

cout << "4. Exit" << endl;

cin >> option;

switch (option)

{

//first case one-dimensional array

case 1: vector();

break

//the second case is a three-dimensional array

case 2: D3();

break

//test all cases at once

case 3: All();

break

//exit the menu

case 4: return;

break

default: cout << "Incorrect data!" << endl;

break

}

}

}

int main()

{

srand(time(0));

//menu for selecting program functions

menu();

}

**Fill.h**

#pragma once

#include <ctime>

#include <cstdlib>

//a module for filling a three-dimensional array with numbers

void fill\_ sorted( int \*\*\* Arr3D , int P , int M , int N );

void fill\_ reversed( int \*\*\* Arr3D , int P , int M , int N );

void fill\_random ( int \*\*\* Arr3D , int P , int M , int N );

**Fill.cpp**

#include " fill.h "

// ordered

void fill\_ sorted( int \*\*\* Arr3D , int P , int M , int N )

{

int number = 0;

for ( int k = 0; k < P ; k++)

for ( int j = 0; j < N ; j++ )

for ( int i = 0; i < M ; i ++)

Arr3D [k][ i ][j] = number++;

}

// carefully orderly

void fill\_ reversed( int \*\*\* Arr3D , int P , int M , int N )

{

int number = P \* M \* N ;

for ( int k = 0; k < P ; k++)

for ( int j = 0; j < N ; j++ )

for ( int i = 0; i < M ; i ++)

Arr3D [k][ i ][j] = number--;

}

// unordered

void fill\_random ( int \*\*\* Arr3D , int P , int M , int N )

{

for ( int k = 0; k < P ; k++)

for ( int j = 0; j < N ; j++ )

for ( int i = 0; i < M ; i ++)

Arr3D [k][ i ][j] = rand( ) % P \* M \* N ;

}

**Sum. h**

#pragma once

//module for searching Sumy everyone sections of the given array

void sum( int \*\*\* Arr3D , int \* Sum , int P , int M , int N );

**Sum.cpp**

# include "sum.h "

void sum( int \*\*\* Arr3D , int \* Sum , int P , int M , int N )

{

int tmp \_

for ( int i = 0; i < P ; i ++)

{

tmp = 0;

for ( int j = 0; j < M ; j++ )

{

for ( int k = 0; k < N ; k++)

{

tmp += Arr3D [ i ][j][k];

}

}

Sum [ i ] = tmp ;

}

}

**Sort.h**

#pragma once

#include < time.h >

#include < windows.h >

#include " sum.h "

//module for sorting a three-dimensional array

// function to permute sections in places

void Swap( int \*\*\* Arr3D , int M , int N , int imin , int s );

// function to search time algorithm fast sorting

clock\_t timeQuickSort ( int L , int R , int \*\*\* Arr3D , int P , int M , int N );

// algorithm fast sorting

void quickSort ( int \* Sum , int L , int R , int \*\*\* Arr3D , int P , int M , int N );

// algorithm of choice number 5

clock\_t Select5( int \*\*\* Arr3D , int P , int M , int N );

// algorithm of choice number 8

clock\_t Select8( int \*\*\* Arr3D , int P , int M , int N );

**Sort.cpp**

#include " sort.h "

void Swap( int \*\*\* Arr3D , int M , int N , int imin , int ) \_

{

int tmp \_

for ( int i = 0; i < M ; i ++)

{

for ( int j = 0; j < N ; j++ )

{

tmp = Arr3D [ imin ][ i ][j];

Arr3D [ imin ][ i ][j] = Arr3D [ s ][ i ][j];

Arr3D [ s ][ i ][ j] = tmp ;

}

}

}

clock\_t timeQuickSort ( int L , int R , int \*\*\* Arr3D , int P , int M , int N )

{

clock\_t time\_start , time\_stop ;

time\_start = clock( );

int \* Sum = new int [ P ];

sum( Arr3D , Sum, P , M , N );

quickSort ( Sum, 0, P - 1, Arr3D , P , M , N );

time\_stop = clock( );

delete [ ]Sum;

return time\_stop - time\_start ;

}

void quickSort ( int \* Sum , int L , int R , int \*\*\* Arr3D , int P , int M , int N )

{

int B, tmp , i , j;

B = Sum [ ( L + R ) / 2]; // central element

i = L , j = R ;

// procedure separation

while ( i <= j)

{

while ( Sum [ i ] < B) i ++;

while ( Sum [j] > B) j--;

if ( i <= j) {

tmp = Sum [ i ];

Sum [ i ] = Sum [ j ];

Sum [j] = tmp ;

Swap( Arr3D , M , N , i , j);

i ++;

j--;

}

}

// recursive challenges , if any, that sort

if ( L < j) quickSort ( Sum , L , j, Arr3D , P , M , N );

if ( i < R ) quickSort ( Sum , i , R , Arr3D , P , M , N );

}

clock\_t Select5( int \*\*\* Arr3D , int P , int M , int N )

{

int Min, imin ;

clock\_t time\_start , time\_stop ;

time\_start = clock( );

int \* Sum = new int [ P ];

sum( Arr3D , Sum, P , M , N );

for ( int s = 0; s < P - 1; s++)

{

Min = Sum[s];

imin = s;

for ( int i = s + 1; i < P ; i ++)

if (Sum[ i ] < Min)

{

Min = Sum[ i ];

imin = i ;

}

if ( imin ! = s)

{

Sum[ imin ] = Sum[s];

Sum[s] = Min;

Swap( Arr3D , M , N , imin , s);

}

}

delete [ ]Sum;

time\_stop = clock( );

return time\_stop - time\_start ;

}

clock\_t Select8( int \*\*\* Arr3D , int P , int M , int N )

{

int L, R, imin , imax , tmp ;

clock\_t time\_start , time\_stop ;

time\_start = clock( );

int \* Sum = new int [ P ];

sum( Arr3D , Sum, P , M , N );

L = 0; R = P - 1;

while (L < R)

{

imin = L;

imax = L;

for ( int i = L + 1; i < R + 1; i ++)

if (Sum[ i ] < Sum[ imin ])

imin = i ;

else if (Sum[ i ] > Sum[ imax ])

imax = i ;

if ( imin ! = L)

{

tmp = Sum[ imin ];

Sum[ imin ] = Sum[L];

Sum[L] = tmp ;

Swap( Arr3D , M , N , imin , L);

}

if ( imax ! = R)

if ( imax == L )

{

tmp = Sum[ imin ];

Sum[ imin ] = Sum[R];

Sum[R] = tmp ;

Swap( Arr3D , M , N , imin , R);

}

else

{

tmp = Sum[ imax ];

Sum[ imax ] = Sum[R];

Sum[R] = tmp ;

Swap( Arr3D , M , N , imax , R);

}

L = L + 1;

R = R - 1;

}

delete [ ]Sum;

time\_stop = clock( );

return time\_stop - time\_start ;

}

**Vector.h**

#pragma once

#include <iostream>

#include < time.h >

#include < windows.h >

#include < iomanip >

//module for sorting one-dimensional array

// function to search time work algorithm fast sorting

clock\_t Vtime\_quick\_ sort( int n , int \* A , int l , int r );

// algorithm fast sorting

void Vquick\_ sort ( int n , int \* A , int l , int r );

// algorithm of choice number 5

clock\_t VSelect5( int \* A , int N );

// algorithm of choice number 8

clock\_t VSelect8( int \* A , int N );

**Vector.cpp**

# include "vector.h "

clock\_t Vtime\_quick\_ sort( int n , int \* A , int L , int R )

{

clock\_t time\_start , time\_stop ;

time\_start = clock( );

Vquick\_ sort ( n , A , L , R );

time\_stop = clock( );

return time\_stop - time\_start ;

}

void Vquick\_ sort ( int n , int \* A , int L , int R )

{

int b, i , j;

b = A [ ( L + R ) / 2];

i = L ; j = R ;

while ( i <= j)

{

while ( A [ i ] < b) i ++;

while ( A [j] > b)j --;

if ( i <= j)

{

int tmp = A [ i ];

A [ i ] = A [ j ];

A [j] = tmp ;

i ++;

j--;

}

}

if ( L < j)

Vquick\_ sort ( n , A , L , j);

if ( i < R )

Vquick\_ sort ( n , A , i , R );

}

clock\_t VSelect5( int \* A , int N )

{

int Min, imin ;

clock\_t time\_start , time\_stop ;

time\_start = clock( );

for ( int s = 0; s < N - 1; s++)

{

Min = A [s];

imin = s;

for ( int i = s + 1; i < N ; i ++)

if ( A [ i ] < Min)

{

Min = A [ i ];

imin = i ;

}

if ( imin ! = s)

{

A [ imin ] = A [s];

A [s] = Min;

}

}

time\_stop = clock( );

return time\_stop - time\_start ;

}

clock\_t VSelect8( int \* A , int N )

{

int L, R, imin , imax , tmp ;

clock\_t time\_start , time\_stop ;

time\_start = clock( );

L = 0; R = N - 1;

while (L < R)

{

imin = L;

imax = L;

for ( int i = L + 1; i < R + 1; i ++)

if ( A [ i ] < A [ imin ])

imin = i ;

else if ( A [ i ] > A [ imax ])

imax = i ;

if ( imin ! = L)

{

tmp = A [ imin ];

A [ imin ] = A [ L ];

A [L] = tmp ;

}

if ( imax ! = R)

if ( imax == L )

{

tmp = A [ imin ];

A [ imin ] = A [ R ];

A [R] = tmp ;

}

else

{

tmp = A [ imax ];

A [ imax ] = A [ R ];

A [R] = tmp ;

}

L = L + 1;

R = R - 1;

}

time\_stop = clock( );

return time\_stop - time\_start ;

}

**Information about the computer**

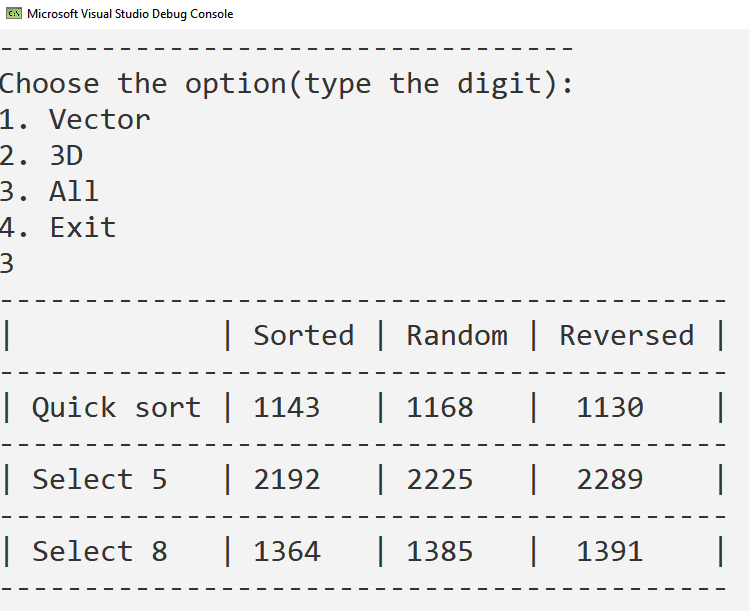
Operating system – Windows 10 ( x64)

The processor is AMD Ryzen 5 2600

RAM – 16 GB

The compiler is Microsoft Visual Studio 2019

**Tests**

****

**The results**

Table # 1 for vector A[N], where N = 1 50,000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 8 | 19 | 8 |
| Choice #5 | 25498 | 25466 | 27676 |
| Selection No. 8 | 13719 | 13782 | 14188 |

**Research case I. Dependence of algorithm operation time on the size of array sections**

Recommended array sizes for research:

Number of keys (sections) P = const = 1,500,000

The shape of the section is the same (square)

M = var, N = var, M = N

P = 50000 was chosen , that is, 30 times less than the recommended value, because higher values caused a stack overflow error when using the quicksort algorithm.

Table No. 2 for the array A[ P, M, N], where P = 50,000; M = 1; N = 1

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 4 | 13 | 5 |
| Choice #5 | 2830 | 2835 | 3073 |
| Selection No. 8 | 1524 | 1543 | 1585 |

Table No. 3 for the array A[ P, M, N], where P = 50,000; M = 2; N = 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 7 | 20 | 9 |
| Choice #5 | 2827 | 2845 | 3074 |
| Selection No. 8 | 1520 | 1545 | 1582 |

Table No. 4 for the array A[ P, M, N], where P = 50,000; M = 4; N = 4

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 14 | 44 | 21 |
| Choice #5 | 2839 | 2855 | 3082 |
| Selection No. 8 | 1525 | 1558 | 1590 |

Table No. 5 for the array A[ P, M, N], where P = 50,000; M = 8; N = 8

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 36 | 131 | 51 |
| Choice #5 | 2838 | 2885 | 3098 |
| Selection No. 8 | 1501 | 1590 | 1613 |

Table No. 6 for the array A[ P, M, N], where P = 50,000; M = 16; N = 16

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 254 | 529 | 156 |
| Choice #5 | 2961 | 2990 | 3052 |
| Selection No. 8 | 1702 | 1703 | 1627 |

**Case study I I . Algorithms' time dependence on the shape of the cross-sections of the array**

Recommended array sizes for research:

Number of keys (sections) P = const = 128,000

The shape of the section is the same (square)

M = var, N = var, M\*N = const

Total number of elements P\*M\*N = const

P = 32 ,000 was chosen , that is, 4 times less than the recommended value , because higher values caused a stack overflow error when using the quicksort algorithm.

Table No. 7 for the array A[ P, M, N], where P = 32,000 ; M = 2 ; N = 800

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1317 | 1345 | 1302 |
| Choice #5 | 1563 | 1675 | 1689 |
| Selection No. 8 | 1028 | 1058 | 1028 |

Table No. 8 for the array A[ P, M, N], where P = 32,000 ; M = 4 ; N = 400

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1442 | 1395 | 1329 |
| Choice #5 | 1592 | 1572 | 1560 |
| Selection No. 8 | 1099 | 1056 | 1062 |

Table No. 9 for the array A[ P, M, N], where P = 32,000 ; M = 8 ; N = 200

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1607 | 1391 | 1442 |
| Choice #5 | 1563 | 1592 | 1558 |
| Selection No. 8 | 1068 | 1060 | 1054 |

Table No. 10 for the array A[ P, M, N], where P = 32,000 ; M = 16 ; N = 100

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1419 | 1507 | 1400 |
| Choice #5 | 1619 | 1730 | 1611 |
| Selection No. 8 | 1091 | 1087 | 1103 |

Table No. 11 for the array A[ P, M, N], where P = 32,000 ; M = 100 ; N = 16

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1624 | 1734 | 1859 |
| Choice #5 | 1778 | 1735 | 1806 |
| Selection No. 8 | 1275 | 1207 | 1255 |

Table No. 12 for the array A[ P, M, N], where P = 32,000 ; M = 200 ; N = 8

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 1926 | 1786 | 1935 |
| Choice #5 | 1840 | 1798 | 1850 |
| Selection No. 8 | 1339 | 1312 | 1325 |

Table No. 13 for the array A[ P, M, N], where P = 32,000 ; M = 400 ; N = 4

|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 2263 | 2420 | 2266 |
| Choice #5 | 2051 | 1978 | 1981 |
| Selection No. 8 | 1504 | 1494 | 1506 |

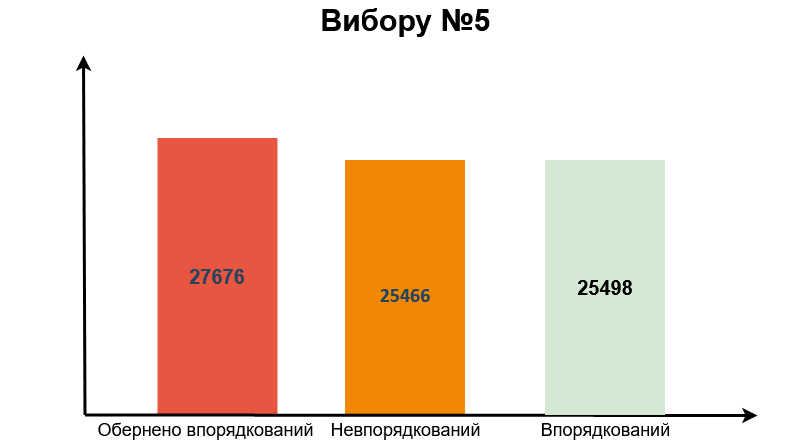
Table No. 14 for the array A[ P, M, N], where P = 32,000 ; M = 800 ; N = 2

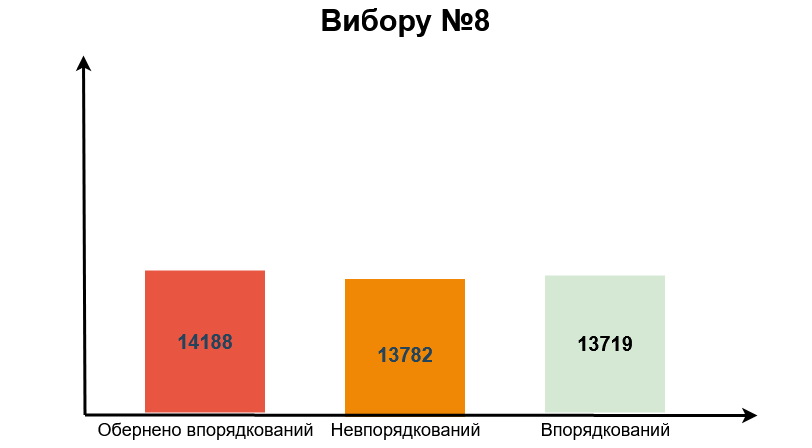
|  |  |  |  |
| --- | --- | --- | --- |
|  | Orderly | Disorganized | Reverse ordered |
| Quick sorting | 2529 | 2546 | 2586 |
| Choice #5 | 2236 | 2238 | 2212 |
| Selection No. 8 | 1712 | 1708 | 1718 |

*Graphs*

**Vector**







**Comparative analysis of algorithms**

The results of the tests agree with the numbers given in the lecture:

For direct selection time sorting:

Minimum Average Maximum

0.94 0.96 1.18

For quick sorting time:

Minimum Average Maximum

0.08 0.12 0.08

**Vector**

As can be seen from the graphs, sorting algorithm No. 5 and No. 8 *of the direct method* *selection* practically do not depend on the orderliness of the array. This can be explained by the fact that these algorithms have the same number of cycle repetitions regardless of the order of the array.

However, for an inversely ordered array, these sorting algorithms, when searching for the minimum element, find a new one at each iteration - that is, they spend time on assignments. And with ordered or not ordered, this happens much less often. Also, for a reverse-ordered array, at each iteration of the outer loop, elements are guaranteed to be exchanged between the sorted and unsorted parts of the array.

I also want to note that algorithm number 8 works about 50-100% faster than 5. This was achieved thanks to the permutation of two, instead of one, elements in one iteration.

*Quicksort* works at approximately the same speed for a sorted and reverse-sorted array, because on the first recursive call, all elements that are smaller than the selected one (in our case, the central one) are placed on the left, and larger ones on the right. So, with a sorted array, quicksort only needs one recursive call to check this. And with an inversely sorted array, only one recursive call is also required, because when searching for each smaller element on the right, a larger one is found on the left.

On the other hand, with an unordered array, the number of recursive calls is unknown - so the running time can also be different, usually longer than in the first two cases.

**A multi-layered massif**

A multidimensional array is located in memory in the same way as a one-dimensional array. It is made only for ease of use. That is, sorting in a multidimensional array occurs in the same way as in a vector.

**Research case I. Dependence of algorithm operation time on the size of array sections**

It can be seen from the graphs that with the increase in the size of the section, the operating time of the selection algorithms almost does not increase. This is due to the fact that these algorithms rearrange the sections immediately to the correct places, that is, with a minimum number of assignments. Since the number of slices is much larger than their size, the selection sort works at about the same speed.

The quicksort algorithm initially also works at approximately the same speed on cross-sections of different sizes. But when the cross-section size increases significantly, the quick sort algorithm slows down a lot. This is due to the fact that this algorithm does not always immediately place sections in the correct position, therefore it performs unnecessary permutations, which, as the section increases, have a stronger and stronger effect on the speed of the algorithm.

So, for arrays with large sections, the selection sort algorithm is better suited.

**Case study I I . Algorithms' time dependence on the shape of the cross-sections of the array**

After measuring the runtime of the Swap function , I realized that everything relies on it. For some reason, this function exchanged elements longer in the cross-section form, when M is greater than N. That is, in sorting with cross-section permutation, everything relies only on the speed of the exchange function. That's why quicksort became so slow after changing the shape of the section, even slower than the selection algorithm 5.

**Conclusions**

Thanks to this coursework, I learned to correctly and quickly evaluate algorithms, choose the most effective one for each situation. I ran 3 algorithms, two of which were direct selection, that is, I could compare two different modifications of the selection method. It turned out that algorithm number 8 works 50-100% faster than the fifth algorithm. Also, both algorithms of the selection method performed worse with the inversely ordered array.

The quicksort algorithm proved to be the fastest in most situations, except when large cross-sections need to be rearranged. Hoare's sorting algorithm works fastest with sorted and reverse-sorted arrays - it works well for sorting those.

It was found that the exchange of elements in the Swap function is slower when M>N cross-section shape. Accordingly, it affected the speed of sorting algorithms.

**References**

1. Synopsis of lectures from the course "Data Structures and Algorithms".
2. Methodical materials for course work.