НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



**ЗВІТ**

**Про виконання лабораторної роботи № 7**

**з дисципліни «Алгоритми і структури даних»**

**Лектор:**

Доцент кафедри ПЗ

Коротєєва Т.О.

**Виконав:**

студ. групи ПЗ-23

Михалевич П.-І.В.

**Прийняла:**

Доцент кафедри ПЗ

Мельник Н.Б.

«\_\_\_» \_\_\_\_\_\_ 2021 р.

∑ = \_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

Львів – 2021

**Тема роботи**: Порівняння методів сортування.

**Мета роботи**: Порівняти вивчені раніше [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)и сортування. Побудувати таблицю і графік швидкодії таких [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ів сортування. Зробити висновки щодо застосовності цих [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ів.

**Теоретичні відомості**

[Алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary) сортування — це [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary), що розв'язує задачу сортування, тобто здійснює впорядкування лінійного списку ([масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)у) елементів.

На практиці елементи, що впорядковуються, рідко бувають просто числами. Набагато частіше, кожен такий елемент є записом (англійською «Record»). В кожному записі є ключ (англійською «Key»), по якому власне і здійснюється впорядкування, в той же час є й інша супутня інформація. [Алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary) сортування на практиці має бути реалізований так, щоб разом з ключами переміщати і супутню інформацію. Якщо кожен запис містить супутню інформацію великого об’єму, то з метою звести до мінімуму переписування великих об’ємів інформації, впорядкування відбувається не у самому [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)і елементів, а в [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)і вказівників на елементи.

Сам метод сортування не залежить від того, чи впорядковуються тільки числа, чи також і супутня інформація, тому при описі[алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ів для простоти припускають, що елементи є числами.

Для [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)у сортування (як і для будь-якого іншого сучасного [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)у) основними характеристиками є обчислювальна та ємнісна складність. Крім цих двох характеристик, сортування поділяють на стабільні та нестабільні, з використанням додаткової інформації про елементи, чи без використання.

Стабільним (англійською «Stable») називається такий [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary) сортування, що не змінює порядок елементів з однаковим ключем.

Для значної кількості [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ів середній і найгірший час впорядкування [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)у з *n*елементів є *O*(*n*2), це пов’язано з тим, що в них передбачені перестановки елементів, що стоять поряд (різниця між індексами елементів не перевищує деякого заданого числа). Такі [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)и зазвичай є стабільними, хоча і не ефективними для великих [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)ів.

Інший клас [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ів здійснює впорядкування за час *O*(*n*∙log(*n*)). В цих [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)ах використовується можливість обміну елементів, що знаходяться на будь-якій відстані один від одного.

Теорема про найкращий час сортування стверджує, що якщо [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary) сортування в своїй роботі спирається тільки на операції порівняння двох об’єктів (≤) і не враховує жодної додаткової інформації про елементи, то він не може впорядкувати [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary) елементів швидше ніж за *O*(*n*∙log(*n*)) в найгіршому випадку.

**Постановка завдання**

Запустити на виконання кожну з написаних раніше програм щонайменше сім разів, отримати таким чином значення часу сортування [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)ів щонайменше семи різних розмірів кожним з шести вивчених методів. В якості набору значень розмірів [масив](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19129&displayformat=dictionary)ів використати таку послідовність чисел:

1)     1024;

2)     4096;

3)     16384;

4)     65536;

5)     262144;

6)     1048576;

7)     4194304 (в разі якщо сортування відбувається довше, ніж 5 хвилин — переривати роботу програми та вважати час сортування нескінченно великим).

**Код програми**

**algorithms.cpp**

#include "algotithms.h"

#include <iostream>

#include <fstream>

#include <map>

#include <algorithm>

#include <sstream>

#include <chrono>

double **BubbleSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Bubble sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> bubbleSortedVector = inputVector;

*for* (size\_t i = 0; i < bubbleSortedVector.size()-1; i++){

bool F = *false*;

*for* (size\_t j = 0; j < bubbleSortedVector.size()-i-1; j++) {

*if* (bubbleSortedVector[j] > bubbleSortedVector[j+1]) {

std::swap(*bubbleSortedVector[j]*, *bubbleSortedVector[j+*1*]*);

F = *true*;

}

}

*if*(F == *false*) *break*;

*auto* end\_loop = std::chrono::high\_resolution\_clock::now();

*auto* elapsed\_loop = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_loop - begin);

*if*((elapsed\_loop.count() \* 1e-3) > std::chrono::seconds{300}.count()){

*return* 300;

}

}

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputBubbleSort\_1",bubbleSortedVector);

*return* elapsed.count() \* 1e-6;

}

double **SelectionSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Selection sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> selectionSortedVector = inputVector;

int min\_idx;

*for* (size\_t i = 0; i < selectionSortedVector.size()-1; i++)

{

min\_idx = i;

*for* (size\_t j = i+1; j < selectionSortedVector.size(); j++)

*if* (selectionSortedVector[j] < selectionSortedVector[min\_idx])

min\_idx = j;

std::swap(*selectionSortedVector[min\_idx]*, *selectionSortedVector[i]*);

*auto* end\_loop = std::chrono::high\_resolution\_clock::now();

*auto* elapsed\_loop = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_loop - begin);

*if*((elapsed\_loop.count() \* 1e-3) > std::chrono::seconds{300}.count()){

*return* 300;

}

}

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputSelectionSort\_1",selectionSortedVector);

*return* elapsed.count() \* 1e-6;

}

double **ShellSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Shell sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> shellSortedVector = inputVector;

int d = (shellSortedVector.size()-1 - 0) / 2;

*while*(d >= 1){

*for* (size\_t i = 0 + d; i < shellSortedVector.size()-1; i++) {

*for* (int j = i; j >= 0; j -= d ) {

int min\_Index = j;

*for* (int k = j-d; k >= 0; k -= d ) {

*if*(shellSortedVector[k]<shellSortedVector[min\_Index]){

min\_Index = k;

}

*auto* end\_loop = std::chrono::high\_resolution\_clock::now();

*auto* elapsed\_loop = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_loop - begin);

*if*((elapsed\_loop.count() \* 1e-3) > std::chrono::seconds{300}.count()){

*return* 300;

}

}

*if*(min\_Index != j){

int tmp = shellSortedVector[min\_Index];

shellSortedVector[min\_Index] = shellSortedVector[j];

shellSortedVector[j] = tmp;

}

}

}

d /= 2;

}

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputShellSort\_1",shellSortedVector);

*return* elapsed.count() \* 1e-6;

}

double **QuickSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Quick sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> quickSortedVector = inputVector;

quickSort(*quickSortedVector*,0,quickSortedVector.size()-1);

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputQuickSort\_1",quickSortedVector);

*return* elapsed.count() \* 1e-6;

}

int **partition** (std::vector<int>& inputVector, int low, int high)

{

int pivot = inputVector[high];

int i = (low - 1);

*for* (int j = low; j <= high - 1; j++)

{

*if* (inputVector[j] < pivot)

{

i++;

int t = inputVector[j];

inputVector[j] = inputVector[i];

inputVector[i] = t;

}

}

int t = inputVector[i + 1];

inputVector[i + 1] = inputVector[high];

inputVector[high] = t;

*return* (i + 1);

}

void **quickSort**(std::vector<int>& inputVector, int low, int high)

{

*if* (low < high)

{

int pi = partition(*inputVector*, low, high);

quickSort(*inputVector*, low, pi - 1);

quickSort(*inputVector*, pi + 1, high);

}

}

double **MergeSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Merge sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> mergeSortedVector = inputVector;

mergeSort(*mergeSortedVector*,0,mergeSortedVector.size()-1);

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputMergeSort\_1 ",mergeSortedVector);

*return* elapsed.count() \* 1e-6;

}

void **merge**(std::vector<int>& inputVector,*const* int left,*const* int mid,*const* int right)

{

*const* int subArrayOneCount = mid - left + 1;

*const* int subArrayTwoCount = right - mid;

int \*leftArray = *new* int[subArrayOneCount],

\*rightArray = *new* int[subArrayTwoCount];

*for* (int i = 0; i < subArrayOneCount; i++)

leftArray[i] = inputVector[left + i];

*for* (int j = 0; j < subArrayTwoCount; j++)

rightArray[j] = inputVector[mid + 1 + j];

int indexOfSubArrayOne = 0,

indexOfSubArrayTwo = 0;

int indexOfMergedArray = left;

*//* *Merge*

*while* (indexOfSubArrayOne < subArrayOneCount && indexOfSubArrayTwo < subArrayTwoCount) {

*if* (leftArray[indexOfSubArrayOne] >= rightArray[indexOfSubArrayTwo]) {

inputVector[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

}

*else* {

inputVector[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

}

indexOfMergedArray++;

}

*while* (indexOfSubArrayOne < subArrayOneCount) {

inputVector[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

indexOfMergedArray++;

}

*while* (indexOfSubArrayTwo < subArrayTwoCount) {

inputVector[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

indexOfMergedArray++;

}

*delete* [] leftArray;

*delete* [] rightArray;

}

void **mergeSort**(std::vector<int>& inputVector, *const* int begin,*const* int end)

{

*if* (begin >= end)

*return*;

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

mergeSort(*inputVector*, begin, mid);

mergeSort(*inputVector*, mid + 1, end);

merge(*inputVector*, begin, mid, end);

}

double **CountingSort**(*const* std::vector<int>& inputVector)

{

*auto* begin = std::chrono::high\_resolution\_clock::now();

std::cout << "Counting sort for " << inputVector.size() << " elements starts."<< std::endl;

std::vector<int> countingSortedVector = inputVector;

std::map<int,int> keys\_values;

*for*(size\_t i = 0; i < countingSortedVector.size(); i++)

{

keys\_values[countingSortedVector[i]]++;

}

int ind = 0;

*for*(*auto* it : keys\_values){

int val = it.first;

int freq = it.second;

*for* (int j = 0;j < freq ;j++ ) {

countingSortedVector[ind] = val;

ind++;

}

}

*auto* end = std::chrono::high\_resolution\_clock::now();

*auto* elapsed = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);

WriteInFile("outputCountingSort\_1",countingSortedVector);

*return* elapsed.count() \* 1e-6;

}

void **WriteInFile**(std::string path, *const* std::vector<int> outputVecor){

std::ofstream file;

*if*(outputVecor.size() == 1024){

file.open(path);

}*else*{

file.open(path, std::ios::app);

}

std::string newstr;

file << "\n\n\n" <<"Vector of " << outputVecor.size() << "elements" << "\n\n" << std::endl;

int myIndex = 0;

*for*(*const* *auto*& i : outputVecor){

*if*(myIndex != 0 && myIndex % 5 == 0){

file << std::endl;

myIndex = 0;

}

file.width(15);

file << i;

file.width(0);

file << "//";

myIndex++;

}

}

**algorithms.h**

#ifndef ALGOTITHMS\_H

#define ALGOTITHMS\_H

#include <vector>

#include <string>

double **BubbleSort**(*const* std::vector<int>& inputVector);

double **SelectionSort**(*const* std::vector<int>& inputVector);

double **ShellSort**(*const* std::vector<int>& inputVector);

double **QuickSort**(*const* std::vector<int>& inputVector);

int **partition** (std::vector<int>& inputVector, int low, int high);

void **quickSort**(std::vector<int>& inputVector, int low, int high);

double **MergeSort**(*const* std::vector<int>& inputVector);

void **merge**(std::vector<int>& inputVector,*const* int left,*const* int mid,*const* int right);

void **mergeSort**(std::vector<int>& inputVector, *const* int begin,*const* int end);

double **CountingSort**(*const* std::vector<int>& inputVector);

void **WriteInFile**(std::string path, std::vector<int> outputVecor);

#endif *//* *ALGOTITHMS\_H*

**mainwindow.cpp**

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <ctime>

MainWindow::**MainWindow**(QWidget \*parent)

: QMainWindow(*parent*)

, ui(*new* Ui::MainWindow)

{

ui->setupUi(*this*);

*for*(int i = 0; i < ui->tableWidget->rowCount(); i++)

{

*for*(int j = 0; j < ui->tableWidget->columnCount(); j++)

{

QTableWidgetItem\* item = *new* QTableWidgetItem;

item->setText("");

item->setTextAlignment(Qt::*AlignCenter*);

ui->tableWidget->setItem(i,j,*item*);

}

}

connect(ui->pushButton\_start,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_2,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_3,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_4,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_5,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_6,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_7,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_start\_8,SIGNAL(clicked()),*this*,SLOT(MySlot()));

connect(ui->pushButton\_clear,SIGNAL(clicked()),*this*,SLOT(MySlot()));

}

void MainWindow::**SortVector**(int size, QTableWidget\* tableWidget)

{

std::vector<int> startVect(size,0);

srand(time(*nullptr*));

*for* (*auto*& i : startVect) {

i = rand()%10001;

}

int i = 0;

*switch*(size){

*case* 1024:

i = 0;

*break*;

*case* 4096:

i = 1;

*break*;

*case* 16384:

i = 2;

*break*;

*case* 65546:

i = 3;

*break*;

*case* 262144:

i = 4;

*break*;

*case* 1048576:

i = 5;

*break*;

*case* 4194304:

i = 6;

*break*;

}

tableWidget->item(i,0)->setText(QString::number(BubbleSort(startVect),'f',6));

tableWidget->item(i,1)->setText(QString::number(SelectionSort(startVect),'f',6));

tableWidget->item(i,2)->setText(QString::number(ShellSort(startVect),'f',6));

tableWidget->item(i,3)->setText(QString::number(QuickSort(startVect),'f',6));

tableWidget->item(i,4)->setText(QString::number(MergeSort(startVect),'f',6));

tableWidget->item(i,5)->setText(QString::number(CountingSort(startVect),'f',6));

}

void MainWindow::**MySlot**()

{

std::vector<int> sizes = {1024, 4096, 16384,65546, 262144, 1048576, 4194304};

QPushButton\* btn = (QPushButton\*)sender();

*if*(btn->text() == "Start 1024")

{

SortVector(sizes[0],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 4096")

{

SortVector(sizes[1],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 16384")

{

SortVector(sizes[2],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 65546")

{

SortVector(sizes[3],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 262144")

{

SortVector(sizes[4],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 1048576")

{

SortVector(sizes[5],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start 4194304")

{

SortVector(sizes[6],*ui->tableWidget*);

}*else* *if*(btn->text() == "Start")

{

SortVector(sizes[0],*ui->tableWidget*);

SortVector(sizes[1],*ui->tableWidget*);

SortVector(sizes[2],*ui->tableWidget*);

SortVector(sizes[3],*ui->tableWidget*);

SortVector(sizes[4],*ui->tableWidget*);

SortVector(sizes[5],*ui->tableWidget*);

SortVector(sizes[6],*ui->tableWidget*);

}*else* *if*(btn->text() == "Clear")

{

*for*(int i = 0; i < ui->tableWidget->rowCount(); i++)

{

*for*(int j = 0; j < ui->tableWidget->columnCount(); j++)

{

ui->tableWidget->item(i,j)->setText("");

}

}

}

}

MainWindow::~***MainWindow***()

{

*delete* ui;

}

**mainwindow.h**

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QTableWidget>

#include "algotithms.h"

QT\_BEGIN\_NAMESPACE

*namespace* Ui { *class* MainWindow; }

QT\_END\_NAMESPACE

*class* MainWindow : *public* QMainWindow

{

Q\_OBJECT

*public*:

MainWindow(QWidget \*parent = *nullptr*);

~MainWindow();

*private*:

void SortVector(int size, QTableWidget\* tableWidget);

*private* *slots*:

void MySlot();

*private*:

Ui::MainWindow \*ui;

};

#endif *//* *MAINWINDOW\_H*

**main.cpp**

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(*argc*, *argv*);

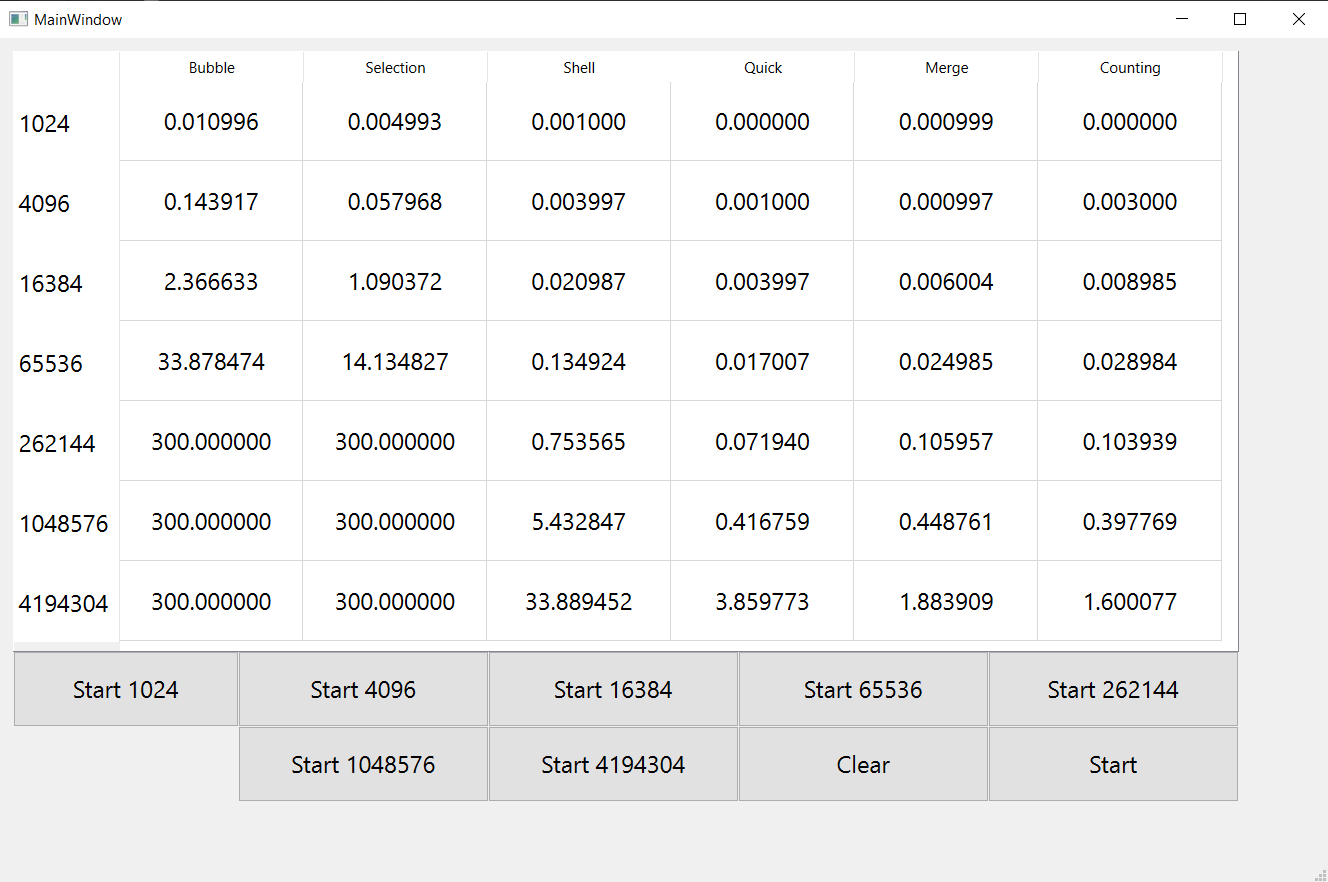
MainWindow w;

w.show();

*return* a.exec();

}

**Результат програми**



### Висновок

На даній лабораторній роботі я порівняв швидкодію вивчених на попередніх лабораторних роботах алгоритмів сортування, а саме алгоритмів бульбашки, вибірки, швидкого сортування, Шелла, сортування злиттям та підрахунком. Реалізував ці методи у програмі, за допомогою якої виміряв їхній час роботи та побудував графік залежності часу роботи алгоритмів від кількості елементів. Дослідження швидкодії було проведено на масивах, розмірами 1024, 4096, 16384,65536, 262144, 1048576 та 4194304 елементів.

Найповільнішими виявилися методи сортування бульбашкою та вибіркою, які всередньому посортували масив на 65536 елементів за відповідно 34 с і 14 с. Вони значно сповільнювалися при зростанні кількості елементів.

Найшвидшим методом сортування виявився метод сортування підрахунком, який сортував масив на 4096 елементів всередньому за 3 мс, на 65536 елементів – за 29 мс, і на 4194304 - за 1.6 с. Така швидкодія обумовлюється тим, що всі числа масиву лежали в межах між 0 і 10000, і в масивах було багато повторювальних елементів. Хоч цей метод і виграє у швидкодії, він використовує значну кількість оперативної пам’яті для роботи.