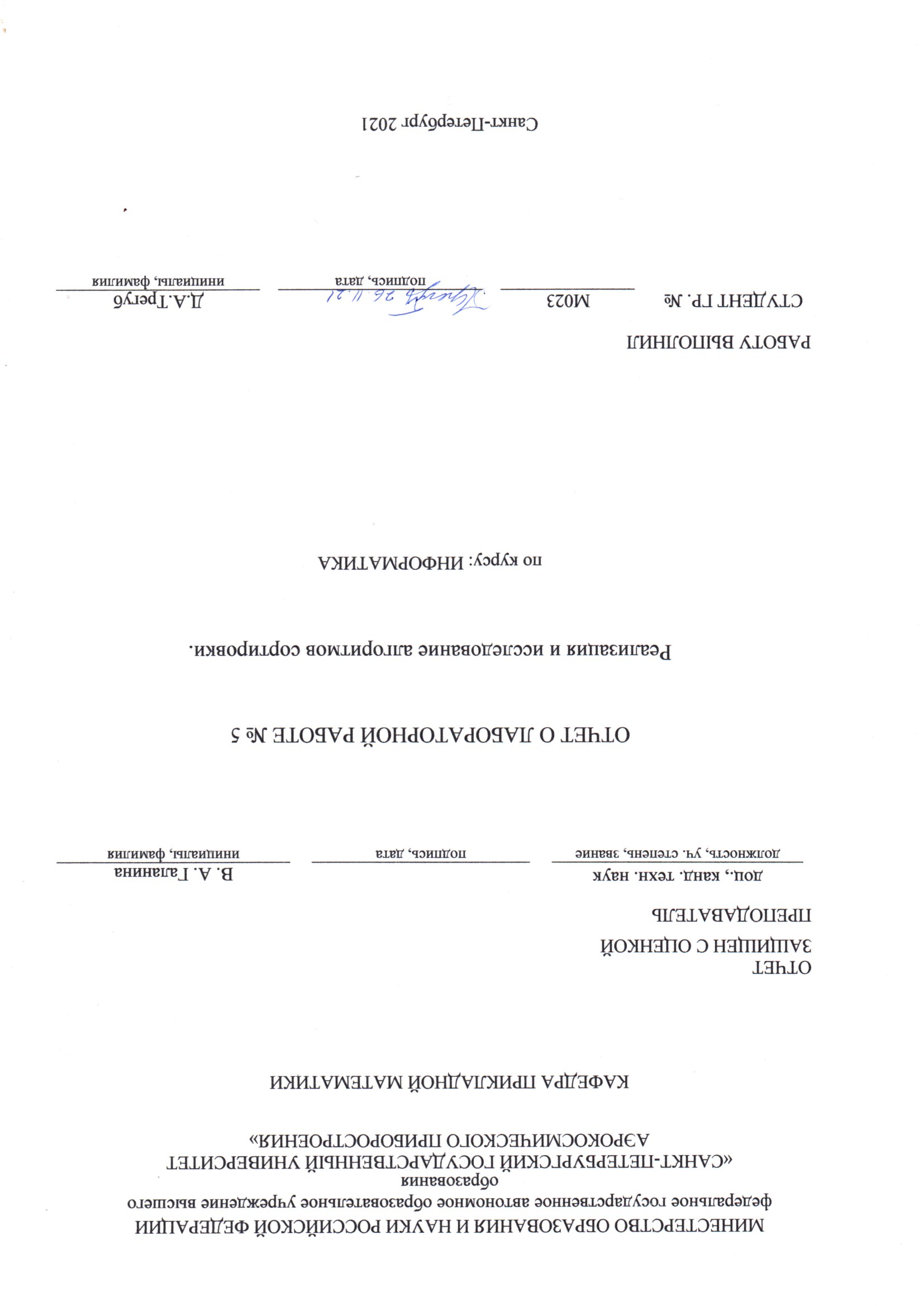
****

**1)Цель работы:**

Используя разработанный в лабораторной работе 4 класс Vector реализовать два алгоритма сортировки, согласно заданию. Исследовать асимптотические зависимости количества сравнений, перестановок и времени выполнения алгоритма в зависимости от количества элементов в массиве (не менее 3-х случаев) и степени упорядоченности ( лучший, худший и средний случаи).

**2)Задание**:

|  |  |
| --- | --- |
| *№ варианта* | *Алгоритмы сортировки* |
| 19 | Быстрая сортировка; Сортировка вставками |

**3)Словесное описание алгоритмов:**

Быстрая сортировка:  
Быстрая сортировка использует алгоритм "разделяй и властвуй". Она начинается с разбиения исходного массива на две области. Эти части находятся слева и справа от отмеченного элемента, называемого опорным. В конце процесса одна часть будет содержать элементы меньшие, чем опорный, а другая часть будет содержать элементы больше опорного.

Сортировка вставками:  
При сортировке вставками массив разбивается на две области: упорядоченную и неупорядоченную. Изначально весь массив является неупорядоченной областью. При первом проходе первый элемент из неупорядоченной области изымается и помещается в правильном положении в упорядоченной области.

**4)Алгоритмы сортировки:**

Быстрая сортировка:

void quickSort(int left, int right)

{

int pivot; // разрешающий элемент

int l\_hold = left; //левая граница

int r\_hold = right; // правая граница

pivot = data[left];

while (left < right) // пока границы не сомкнутся

{

while ((data[right] >= pivot) && (left < right)) {

right--; // сдвигаем правую границу пока элемент [right] больше [pivot]

s++;//сравнения

}

if (left != right) // если границы не сомкнулись

{

data[left] = data[right]; // перемещаем элемент [right] на место разрешающего

left++; // сдвигаем левую границу вправо

p++;//перестановки

}

while ((data[left] <= pivot) && (left < right)){

left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]

s++;//сравнения

}

if (left != right) // если границы не сомкнулись

{

data[right] = data[left]; // перемещаем элемент [left] на место [right]

right--; // сдвигаем правую границу вправо

p++;//перестановки

}

}

data[left] = pivot; // ставим разрешающий элемент на место

pivot = left;

left = l\_hold;

right = r\_hold;

if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива

quickSort(left, pivot - 1);

if (right > pivot)

quickSort(pivot + 1, right);

}

Сортировка вставками:

void insertsort() {

for (int i = 1; i < len; i++) {

s++;

for (int j = i; j > 0 && data[j - 1] > data[j]; j--) {

swap(j - 1, j);

p++;

}

}

}

**5)Текст программы:**

#include "iostream"

#include <cassert>

#include <ctime>

#include <chrono>

using namespace std;

class Timer

{

private:

// Псевдонимы типов используются для удобного доступа к вложенным типам

using clock\_t = std::chrono::high\_resolution\_clock;

using second\_t = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<clock\_t> m\_beg;

public:

Timer() : m\_beg(clock\_t::now())

{

}

void reset()

{

m\_beg = clock\_t::now();

}

double elapsed() const

{

return std::chrono::duration\_cast<second\_t>(clock\_t::now() - m\_beg).count();

}

};

template < typename T >

class Vect {

private:

T\* data;

int len = 0, p, s;

public:

Vect() {

len = 0;

data = nullptr;

}

Vect(int q, int val = 0) {

len = q;

p = 0; s = 0;

assert(q >= 0);

if (len >= 0) {

data = new T[len];

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

data[i] = val;

}

else

data = nullptr;

}

~Vect() {

delete[] data;

data = nullptr;

len = 0;

}

int& operator[](int index)

{

return data[index];

}

int leng() {

return len;

}

void create(int q, int val = 0) {

len = q;

if (len >= 0) {

data = new T[len];

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

data[i] = val;

}

}

void clear() { //Полная отчистка массива

delete[] data;

data = nullptr;

len = 0;

}

int front() { //Просмотр первой ячейки

return (data[0]);

}

int back() { //Просмотр последней ячейки

return (data[len - 1]);

}

void resize(int n, int val = 0) {//измененяет размер массива на n элементов и добавляет новые со значение val

if (n == len) //Если совпадает

return;

if (n <= 0) { //Если нужно обнулить массив

clear();

return;

}

T\* new\_data = new T[n];

if (n > len) { //Если новый размер масссива больше

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

new\_data[i] = data[i];

for (int i = len; i < n; i++)

new\_data[i] = val; //изменение

clear();

len = n;

data = new\_data;

}

if (n < len) { //Если новый размер массива меньше

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

new\_data[i] = data[i];

clear();

len = n;

data = new\_data;

}

new\_data = nullptr;

}

void empty() { //если контейнер пуст возвращает true, если нет - false

if (len == 0) {

cout << "Вектор не содержит данных" << endl;

return;

}

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

if (data[i] != 0) {

cout << "Вектор содержит данные" << endl;

return;

}

cout << "Вектор не содержит данных" << endl;

return;

}

void push\_back(int val = 0) {//добавляет заданный элемент в конец вектора

len = len + 1;

T\* new\_data = new T[len];

for (int i = 0; i < len - 1; i++)

new\_data[i] = data[i];

new\_data[len - 1] = val;

clear();

data = new\_data;

new\_data = nullptr;

};

void pop\_back() {//удаляет элемент из конца вектора

len = len - 1;

T\* new\_data = new T[len];

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

new\_data[i] = data[i];

clear();

data = new\_data;

new\_data = nullptr;

}

void insert(int val = 0, int n = 1, int it = 0) {//добавляет элементы в начало вектора

len = len + n;

T\* new\_data = new T[len + 1];

for (int i = it; i < n; i++)

new\_data[i] = val;

for (int i = n; i < len; i++)

new\_data[i] = data[i - n];

clear();

data = new\_data;

new\_data = nullptr;

}

void erase(int n) { //удаляет выбранный элемент

len = len - 1;

T\* new\_data = new T[len + 1];

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

new\_data[i] = data[i];

for (int i = n + 1; i < len + 1; i++)

new\_data[i - 1] = data[i];

clear();

data = new\_data;

new\_data = nullptr;

}

void swap(int n, int m) { //меняет два элемента местами

int q = data[n];

data[n] = data[m];

data[m] = q;

}

void show() {

if (len == 0)

cout << "er";

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

cout << data[i] << " ";

cout << endl;

}

void show\_char() {

cout << "Количество перестановок:" << p << endl << "Количество сравнений:" << s << endl;

p = 0;

s = 0;

}

void quickSort(int left, int right)

{

int pivot; // разрешающий элемент

int l\_hold = left; //левая граница

int r\_hold = right; // правая граница

pivot = data[left];

while (left < right) // пока границы не сомкнутся

{

while ((data[right] >= pivot) && (left < right)) {

right--; // сдвигаем правую границу пока элемент [right] больше [pivot]

s++;//сравнения

}

if (left != right) // если границы не сомкнулись

{

data[left] = data[right]; // перемещаем элемент [right] на место разрешающего

left++; // сдвигаем левую границу вправо

p++;//перестановки

}

while ((data[left] <= pivot) && (left < right)){

left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]

s++;//сравнения

}

if (left != right) // если границы не сомкнулись

{

data[right] = data[left]; // перемещаем элемент [left] на место [right]

right--; // сдвигаем правую границу вправо

p++;//перестановки

}

}

data[left] = pivot; // ставим разрешающий элемент на место

pivot = left;

left = l\_hold;

right = r\_hold;

if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива

quickSort(left, pivot - 1);

if (right > pivot)

quickSort(pivot + 1, right);

}

void insertsort() {

for (int i = 1; i < len; i++) {

s++;

for (int j = i; j > 0 && data[j - 1] > data[j]; j--) {

swap(j - 1, j);

p++;

}

}

}

};

int main() {

setlocale(LC\_ALL, "rus");

int n, k, j;

bool t = false;

cout << "Введите длинну массива:" << endl;

cin >> n;

Vect <int> q(n);

do {

cout << "Выберите вариант массива:" << endl;

cout << "1)Отсортированный" << endl;

cout << "2)Отсортированный в обратном порядке" << endl;

cout << "3)Рандомные" << endl;

cout << "4)Выход" << endl;

cin >> k;

if (k == 1) {

for (int i = 0; i < q.leng(); i++)

q[i] = i;

t = true;

}

if (k == 2) {

j = n;

for (int i = 0; i < q.leng(); i++) {

q[i] = j;

j--;

}

t = true;

}

if (k == 3) {

srand(time(NULL));

for (int i = 0; i < q.leng(); i++)

q[i] = rand() % 101;

t = true;

}

if (k == 4) {

return 0;

break;

}

system("pause");

system("cls");

if (t == true) {

do {

cout << "Выберите действие:" << endl;

cout << "1)Быстрая сортировка" << endl;

cout << "2)Сортировка вставками" << endl;

cout << "3)Назад" << endl;

cin >> k;

if (k == 1) {

cout << "Первоначальный массив: " << endl;

q.show();

Timer t;

q.quickSort(0, q.leng() - 1);

cout << "Время работы сортировки: " << fixed << t.elapsed() << '\n';

q.show\_char();

cout << "Отсортированный массив: " << endl;

q.show();

}

if (k == 2) {

cout << "Первоначальный массив: " << endl;

q.show();

Timer t;

q.insertsort();

cout << "Время работы сортировки: " << fixed << t.elapsed() << '\n';

q.show\_char();

cout << "Отсортированный массив: " << endl;

q.show();

}

if (k == 3) {

t = false;

}

system("pause");

system("cls");

} while (t == true);

}

} while (t == false);

}**6)Результаты работы программы:**

В зависимости от количества элементов при быстрой сортировке:

1000:



2000:



3000:



В зависимости от количества элементов при сортировке вставками:

1000:



2000:



3000:



В зависимости от степени упорядоченности при быстрой сортировке (кол-во элементов – 3000):

Худший:



Средний:



Лучший:



В зависимости от степени упорядоченности при сортировке вставками (кол-во элементов – 3000):

Худший:



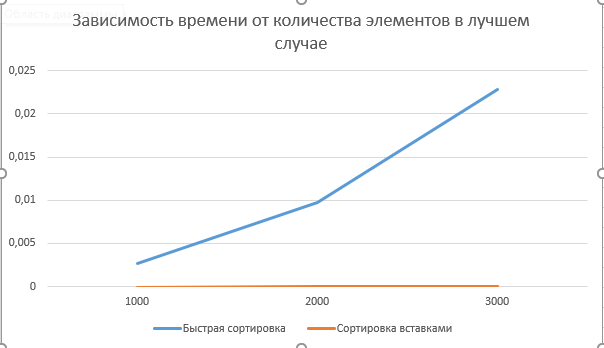
Средний:

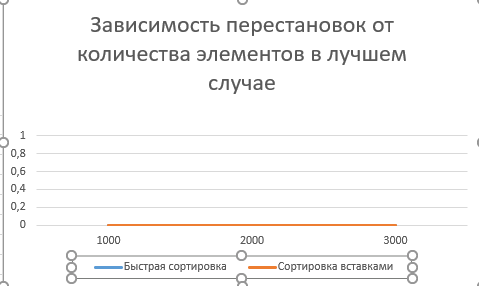


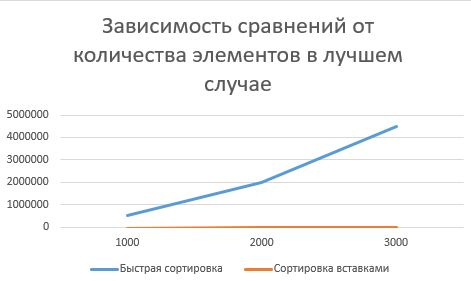
Лучший:



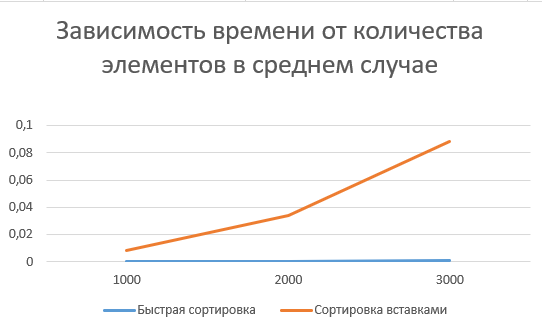
**7)Графики полученных зависимостей:**

Лучший случай:  


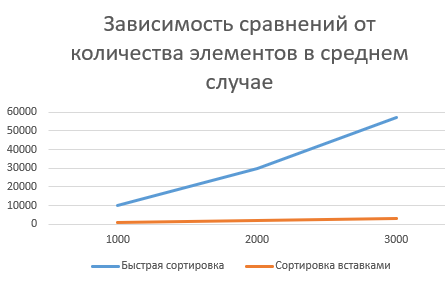




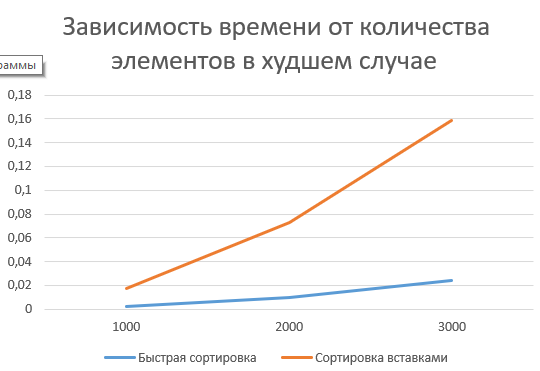
Средний случай:

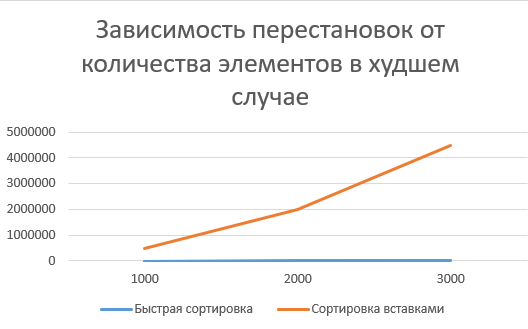


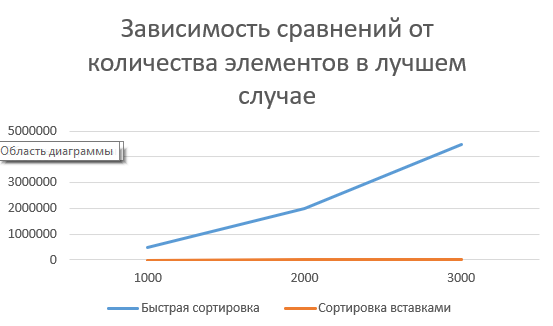




Худший случай:







**8)Выводы из полученных зависимостей:**

Быстрая сортировка: при увеличении количества элементов все показатели линейно зависят от количества элементов. При изменении степени упорядоченности максимальное быстродействие достигается в среднем случае, что говорит об эффективности данного алгоритма

Сортировка вставками: при увеличении количества элементов все показатели линейно зависят от количества элементов. При изменении степени упорядоченности время работы оказывается худшим при среднем случае, максимально эффективном при лучшем случае и среднем в худшем случае. Эти данные говорят о неэффективности этого алгоритма для большинства наборов данных.

**9)Вывод:** Используя разработанный в лабораторной работе 4 класс Vector реализовал два алгоритма сортировки(быстрая сортировка и сортировка вставками), а также провёл анализ количества сравнений, перестановок и времени выполнения алгоритма в зависимости от количества элементов в массиве (при 1000, 2000 и 3000 ) и степени упорядоченности ( лучший, худший и средний случаи).