Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

По лабораторной работе №2

По курсу: «Анализ алгоритмов»

Тема: «Алгоритмы умножения матриц»

Студент:	Пронин А. С.
Группа:	ИУ7-52Б
Преподаватель:	Волкова Л. Л.
Оценка:	________________

Москва

2021

Содержание

Введение

Цель работы – изучение алгоритмов умножения матриц, оценка их трудоёмкости и получение навыков в улучшении алгоритмов.

Задачи работы:

- изучить алгоритмы умножения матриц – стандартный и алгоритм Винограда;
- оптимизировать алгоритм Винограда;
- реализовать три алгоритма умножения матриц – классический, алгоритм Винограда и улучшенный алгоритм Винограда;
- оценить время выполнения реализованных алгоритмов;
- рассчитать трудоемкость каждого алгоритма умножения.

1 Аналитический раздел

Результатом умножения матриц Am×n и Bn×k будет матрица Cm×k такая, что элемент матрицы C, стоящий в i-той строке и j-том столбце (cij), равен сумме произведений элементов i-той строки матрицы A на соответствующие элементы j-того столбца матрицы B:

cij = ai1 · b1j + ai2 · b2j + ... + ain · bnj [1]

В данной лабораторной работе рассматриваются следующие алгоритмы стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда.

1.1 Умножение матриц

Матрицей A размера $[m \times n]$ называется прямоугольная таблица чисел, функций или алгебраических выражений, содержащая m строк и n столбцов. Числа m и n определяют размер матрицы.[2] Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

Пусть даны две прямоугольные матрицы A и B размеров $[m \times n]$ и $[n \times k]$ соответственно. В результате произведения матриц A и B получим матрицу C размера $[m \times k]$. Тогда матрица C (1.1)

$$C_{mk} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mk} \end{pmatrix}, \tag{1.1}$$

где элементы матрицы равны (1.2)

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} \quad (i = \overline{1, m}; j = \overline{1, k}) \tag{1.2}$$

будет называться произведением матриц A и B [2].

1.2 Классический алгоритм умножения матриц

Реализация классического алгоритма умножения двух матриц заключается в реализации вычисления элементов итоговой матрицы по формуле 1.2. То есть по определению.

1.3 Алгоритм Винограда

Подход алгоритма Винограда является иллюстрацией общей методологии, начатой в 1979 году на основе билинейных и трилинейных форм, благодаря которым большинство усовершенствований для умножения матриц были получены [3].

Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. Их скалярное произведение равно (1.3)

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4 \tag{1.3}$$

Равенство (1.3) можно переписать в виде (1.4)

$$V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) - v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4 \tag{1.4}$$

Менее очевидно, что выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. В случае нечетных размеров векторов, после всех вычислений добавим недостающую сумму элементов $v_5 + w_5$ в цикле по элементам результирующей матрицы.

Вывод

Были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда, основное отличие которых — наличие предварительной обработки, а также количество операций умножения.

2 Конструкторский раздел

В данном разделе представлены требования к разрабатываемому ПО и схемы алгоритмов умножения матриц.

2.1 Требования к ПО

К программе предъявляется ряд требований:

- корректное умножение матриц размером до $[N \times N]$, где $N \in [0 : 2000]$;
- при матрицах неправильных размеров программа не должна аварийно завершаться.

2.2 Схемы алгоритмов

Ниже представлены схемы следующих алгоритмов сортировки:

- схема классического алгоритма умножения матриц (Рисунок **??**);
- схема алгоритма Винограда (рисунки **??** - **??**);

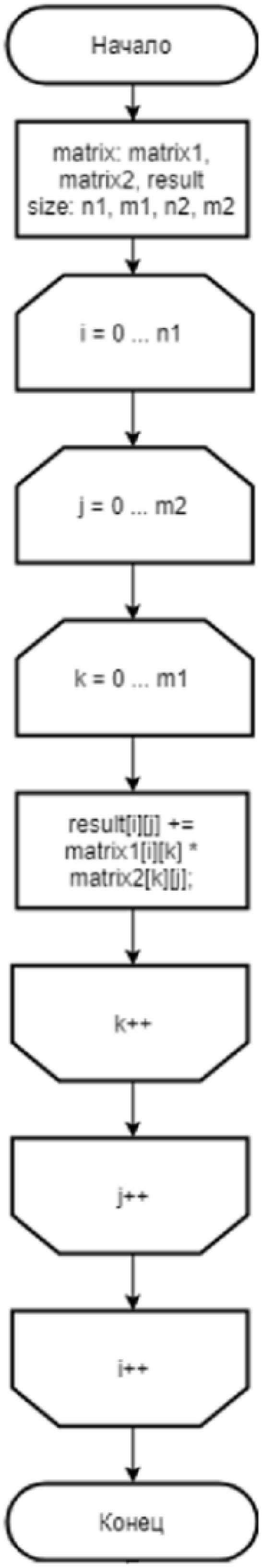


Рисунок 2.1 – Схема классического алгоритма умножения матриц

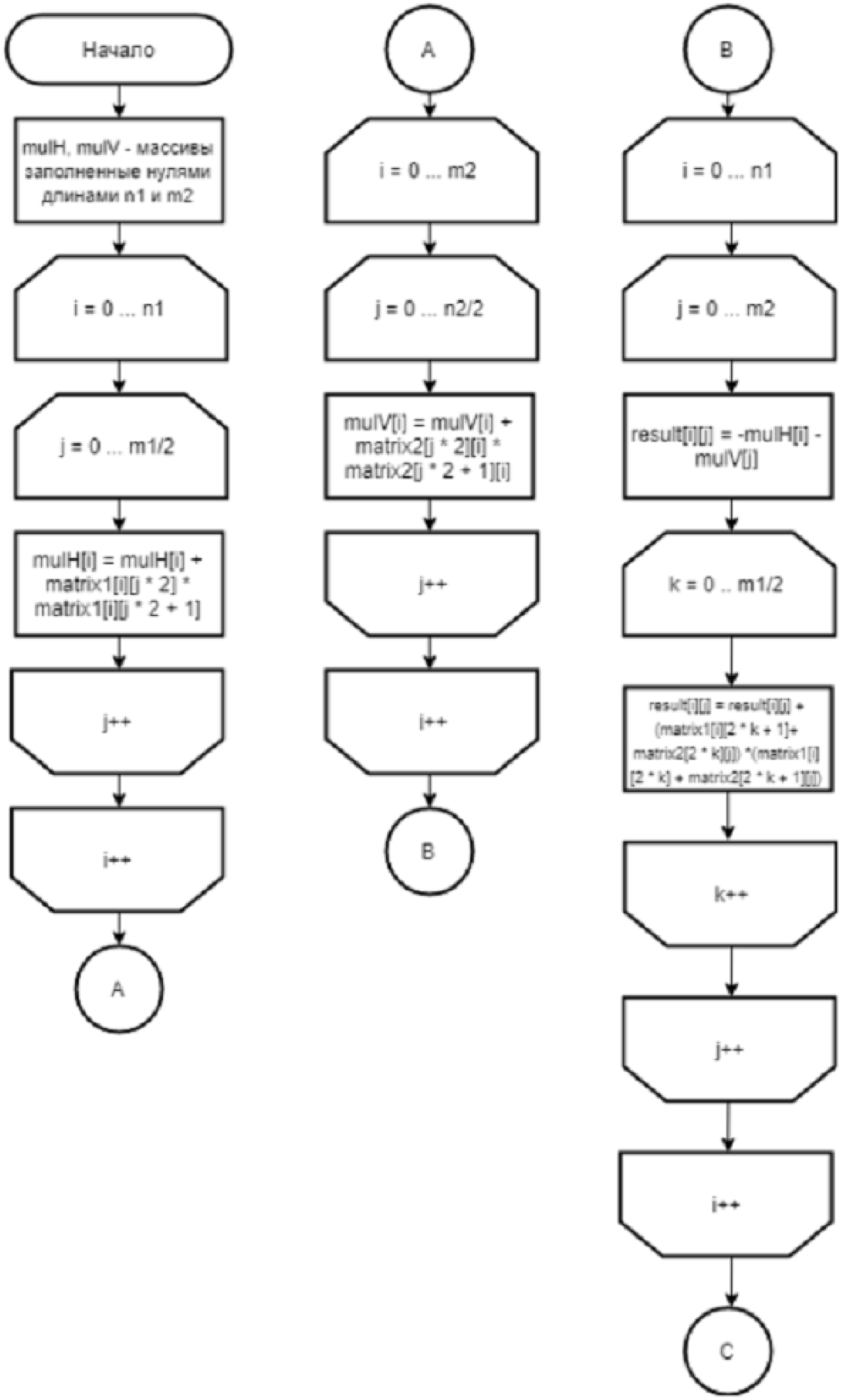


Рисунок 2.2 – Схема алгоритма Винограда

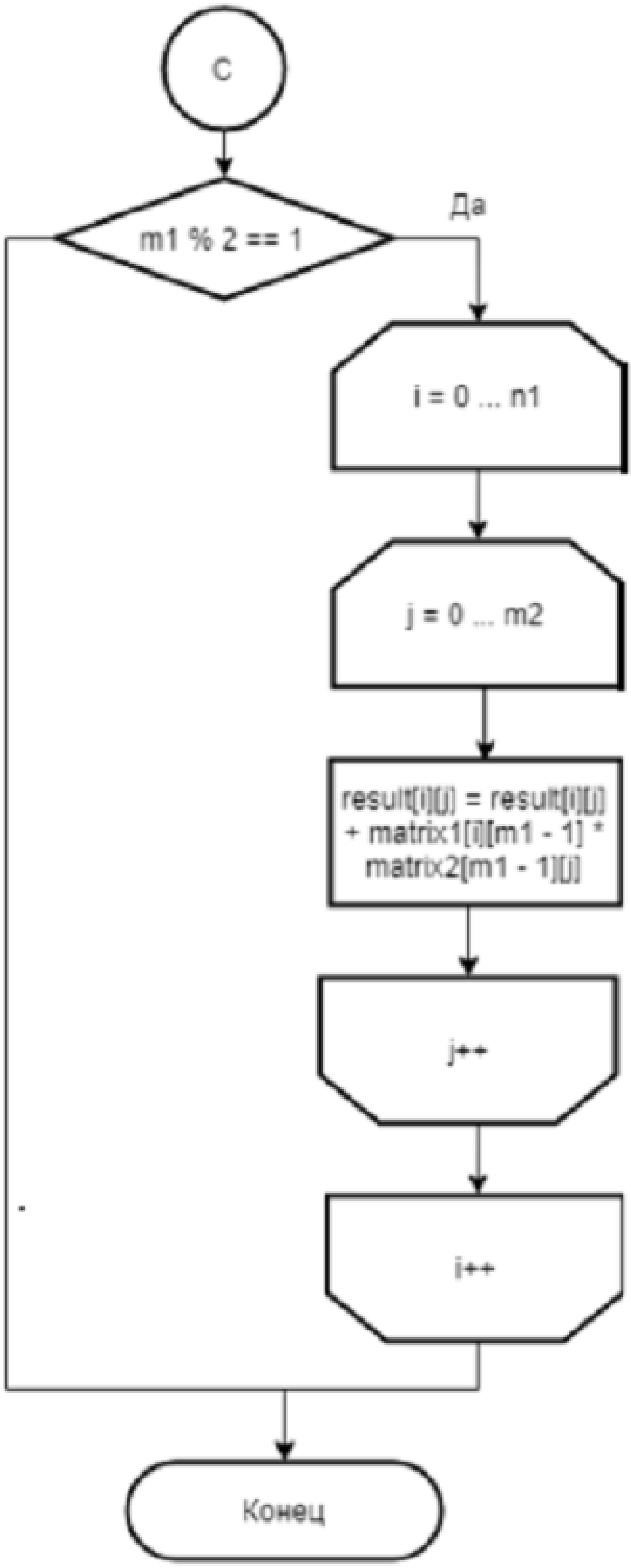


Рисунок 2.3 – Схема алгоритма Винограда (продолжение)

2.3 Вывод

В данном разделе были представлены требования к разрабатываемому ПО и разработаны схемы алгоритмов умножения матриц.

3 Технологический раздел

В данном разеделе представлены выбор инструментов для реализации и оценки алгоритмов, а также листинги полученного кода.

3.1 Выбор инструментов

По-скольку наиболее освоенным языком для разработчика является c++, для реалищзации алгоритмов был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответсвенно для компиляции кода будет использоваться компилятор G++.

Чтобы оценить время выполнения программы будет замерятся процессорное время, т.к. таким образом будут получены данные подходящие для целесообразного сравнения алгоритмов. Для замера процессорного времени программы используется функция GetProcessTimes() т.к. программа тестируется на компьютере с установленной ОС Windows. [4]

Кроме этого, необходимо отключить оптимизации компилятора для более честного сравнения алгоритмов. В моём случае это делается с помощью ключа $-O0$ т.к. используется компилятор G++. [5]

3.2 Реализация алгоритмов

На листингах 3.1–3.3 представлены реализации алгоритмов умножения матриц.

Листинг 3.1 – Классический алгоритм

```
int standartMult(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrx2, int n2, int m2)
{
    if (m1 != n2)
        return SIZE_ERROR;

    for (int i = 0; i < n1; i++)
       for (int j = 0; j < m2; j++)
       {
           rez[i][j] = 0;
           for (int k = 0; k < n2; k++)
               rez[i][j] = rez[i][j] + mtrx1[i][k] * mtrx2[k][j];
       }
    return 0;
}
```

Листинг 3.2 – Алгоритм Винограда

```
int vinograd(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrx2, int n2, int m2)
{
    if (m1 != n2)
        return SIZE_ERROR;
    vector<int> mulH(n1, 0);
    for (int i = 0; i < n1; i++)
        for (int j = 0; j < m1 / 2; j++)
            mulH[i] = mulH[i] + mtrx1[i][j * 2] * mtrx1[i][j * 2 + 1];

    vector<int> mulV(n1, 0);
    for (int i = 0; i < m2; i++)
        for (int j = 0; j < n2 / 2; j++)
            mulV[i] = mulV[i] + mtrx2[j * 2][i] * mtrx2[j * 2 + 1][i];

    for (int i = 0; i < n1; i++)
        for (int j = 0; j < m2; j++)
        {
            rez[i][j] = -mulH[i] - mulV[j];
            for (int k = 0; k < n2 / 2; k++)
                rez[i][j] = rez[i][j] + (mtrx1[i][2 * k + 1] + mtrx2[2 * k][j]) *
                     (mtrx1[i][2 * k] + mtrx2[2 * k + 1][j]);
        }

    if (n2 % 2 == 1)
        for (int i = 0; i < n1; i++)
            for (int j = 0; j < m2; j++)
                rez[i][j] = rez[i][j] + mtrx1[i][n2 - 1] * mtrx2[n2 - 1][j];

    return 0;
}
```

Листинг 3.3 – Оптимизированный алгоритм Винограда

```
int optimizedVinograd(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrrx2, int n2, int m2)
{
    if (m1 != n2)
        return SIZE_ERROR;

    vector<int> mulH(n1, 0);
    for (int i = 0; i < n1; i++)
        for (int j = 0; j < m1 - 1; j += 2)
            mulH[i] -= mtrx1[i][j] * mtrx1[i][j + 1];

    vector<int> mulV(n1, 0);
    for (int i = 0; i < m2; i++)
        for (int j = 0; j < n2 - 1; j += 2)
            mulV[i] -= mtrrx2[j][i] * mtrrx2[j + 1][i];

    bool flag = false;
    if (n2 % 2 == 1)
        flag = true;

    for (int i = 0; i < n1; i++)
        for (int j = 0; j < m2; j++)
        {
            rez[i][j] = mulH[i] + mulV[j];
            for (int k = 0; k < n2 - 1; k += 2)
                rez[i][j] += (mtrx1[i][k + 1] + mtrrx2[k][j]) * (mtrx1[i][k] + mtrrx2[k +
                    1][j]);

            if (flag)
                rez[i][j] += mtrx1[i][n2 - 1] * mtrrx2[n2 - 1][j];
        }

    return 0;
}
```

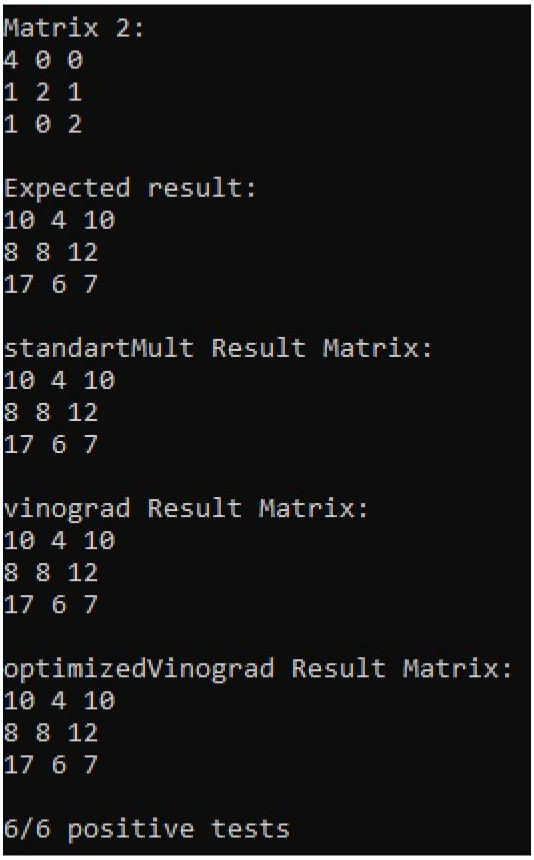
3.3 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов умножения матриц. Тестирование проводилось методом чёрного ящика. Все тесты пройдены успешно для всех алгоритмов.

Таблица 3.1 – Функциональные тесты

Матрица 1	**Матрица 2**	**Ожидаемый рез.**	**Фактический рез.**
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 4 \\ 0 & 4 & 4 \\ 3 & 3 & 2 \end{pmatrix}$	$\begin{pmatrix} 4 & 0 & 0 \\ 1 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 10 & 4 & 10 \\ 8 & 8 & 12 \\ 17 & 6 & 7 \end{pmatrix}$	$\begin{pmatrix} 10 & 4 & 10 \\ 8 & 8 & 12 \\ 17 & 6 & 7 \end{pmatrix}$

На рисунке 3.1 приведены результаты тестирования.



```
Matrix 2:
4 0 0
1 2 1
1 0 2

Expected result:
10 4 10
8 8 12
17 6 7

standartMult Result Matrix:
10 4 10
8 8 12
17 6 7

vinograd Result Matrix:
10 4 10
8 8 12
17 6 7

optimizedVinograd Result Matrix:
10 4 10
8 8 12
17 6 7

6/6 positive tests
```

Рисунок 3.1 – Результаты функционального тестирования

Как видно по рисунку, функциональные тесты пройдены.

3.4 Вывод

В данном разделе были выбраны инструменты для реализации алгоритмов, представлены листинги их реализации, а также проведено функциональное тестирование.

4 Исследовательский раздел

В данном разделе представлены примеры работы программы, сравнительный анализ реализованных алгоритмов и оценка их трудоёмкости.

5 Исследовательская часть

5.1 Пример работы

Демонстрация работы программы приведена на рисунке 5.1.

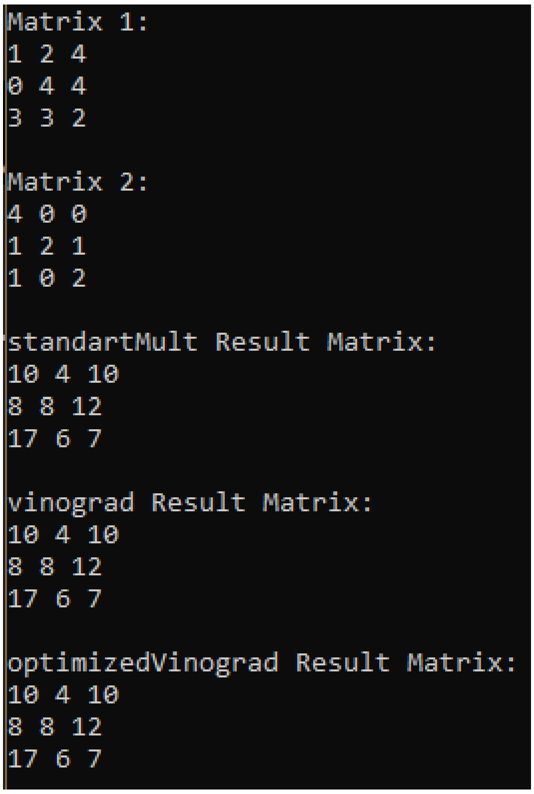


Рисунок 5.1 – Пример работы программы

STOPPED HERE!!

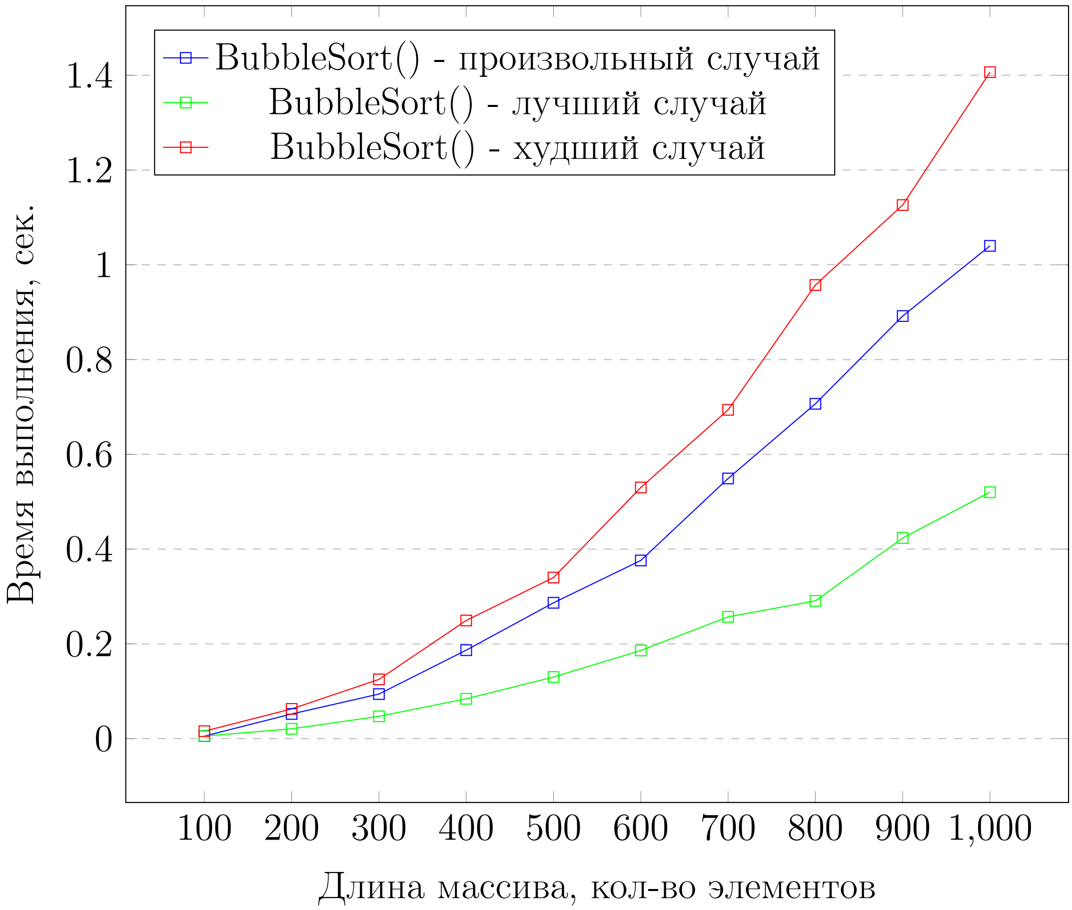


Рисунок 5.2 – Зависимость времени выполнения сортировки пузырьком от длины массива в разных случаях

5.2 Сравнительный анализ времени выполнения алгоритмов

Чтобы провести сравнительный анализ времени выполнения алгоритмов замерялось процессорное время для массивов с 100, 200, ... 1000 элементами. Чтобы оценить время выполнения сортировки для массива размера N, он заполнялся цислами от 0 $N^{10} - 1$, замерялось процессорное время для части кода, которая сортировала массивы 500000/N раз, после чего результат делился на кол-во итераций.

Сравнительный анализ проводилось на компьютере с процессором AMD Ryzen 5 5600H.

Для сортировки пузырьком наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.4 изображены зависимости времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [6]

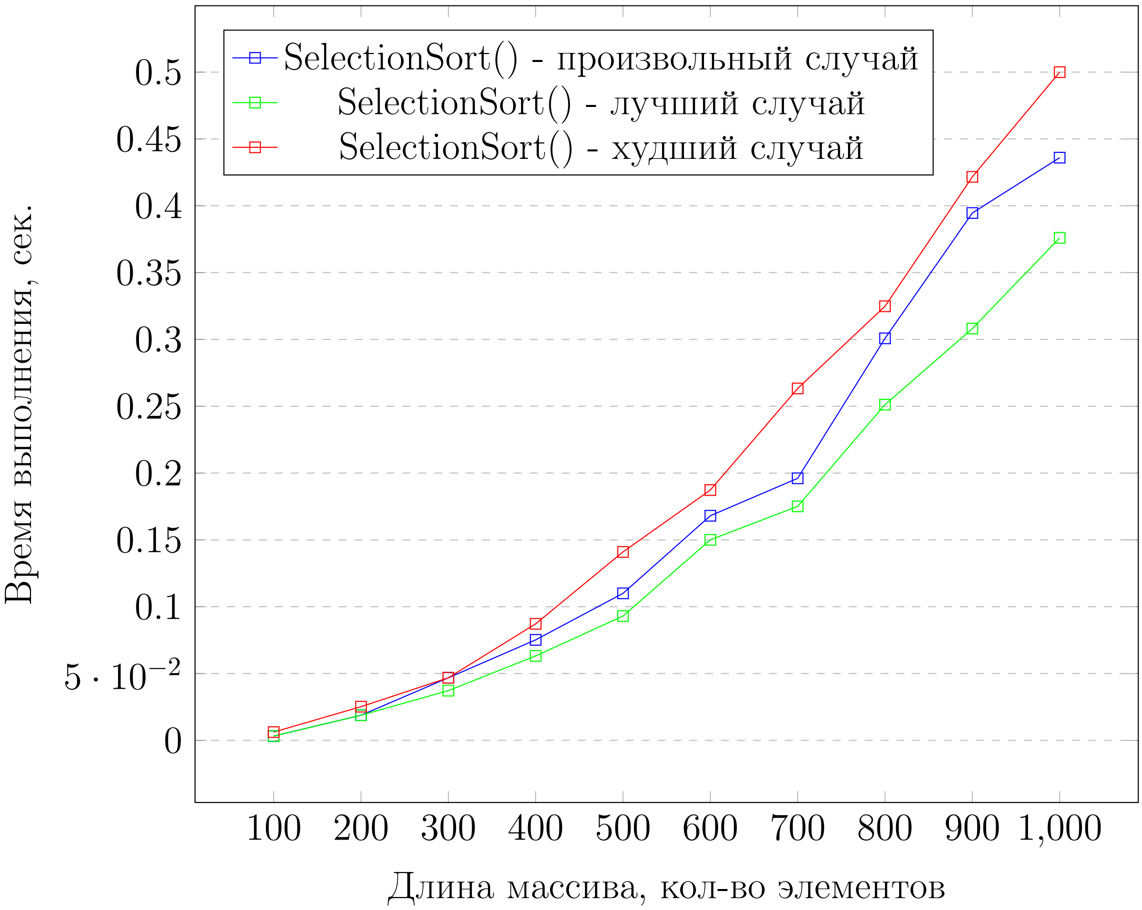


Рисунок 5.3 – Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

Для сортировки выбором наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.5 изображены зависимости времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [6]

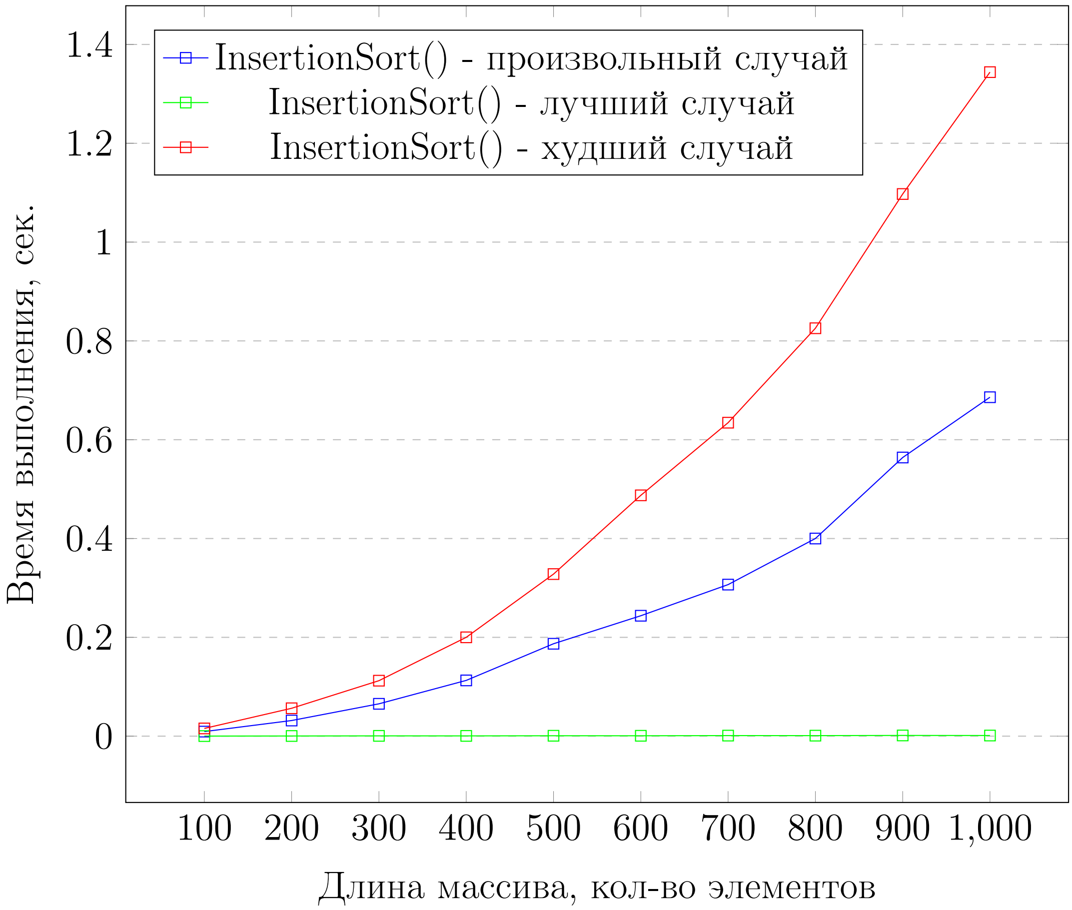


Рисунок 5.4 – Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

Для сортировки вставками наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.6 изображены зависимости времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [6]

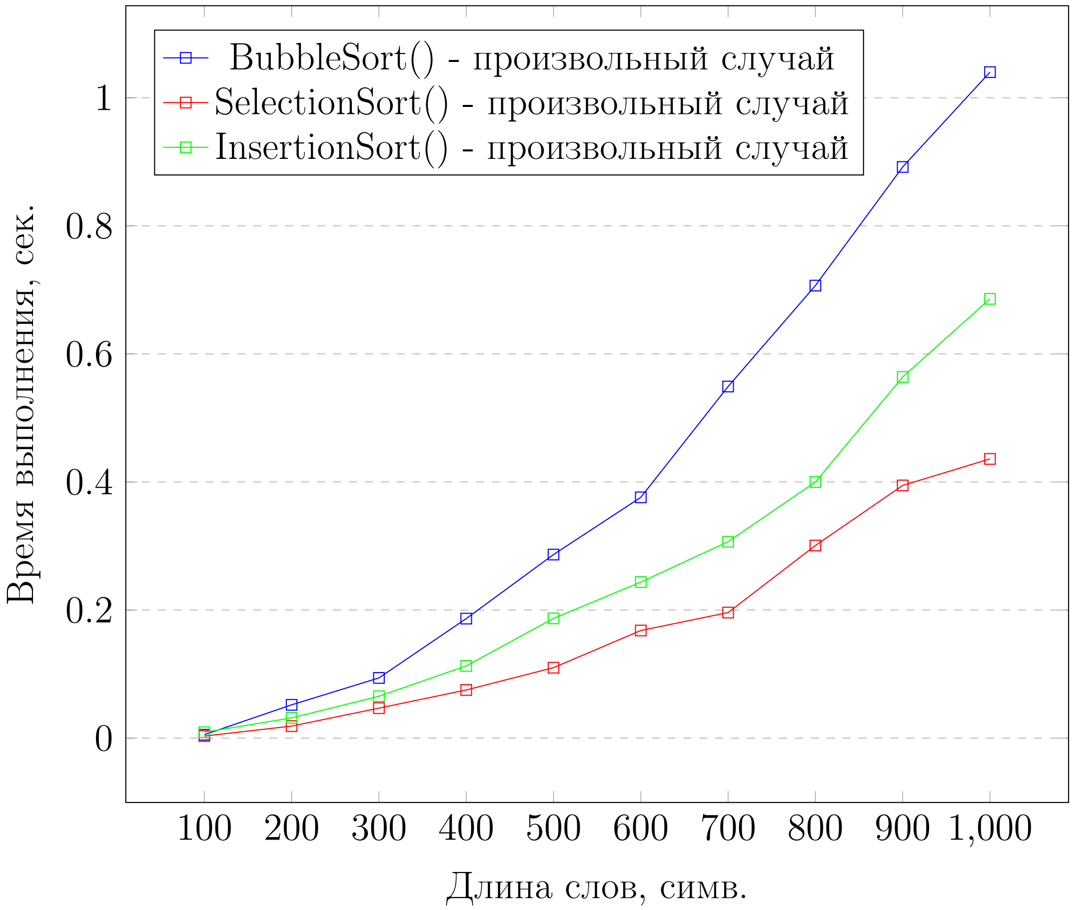


Рисунок 5.5 – Зависимость времени выполнения алгоритмов сортировок от длины массива в произвольном случае

Также приведены графики (рисунки 4.7-4.9) для сравнения алгоритмов сортировок между собой в произвольном, лучшем и худшем случаях.

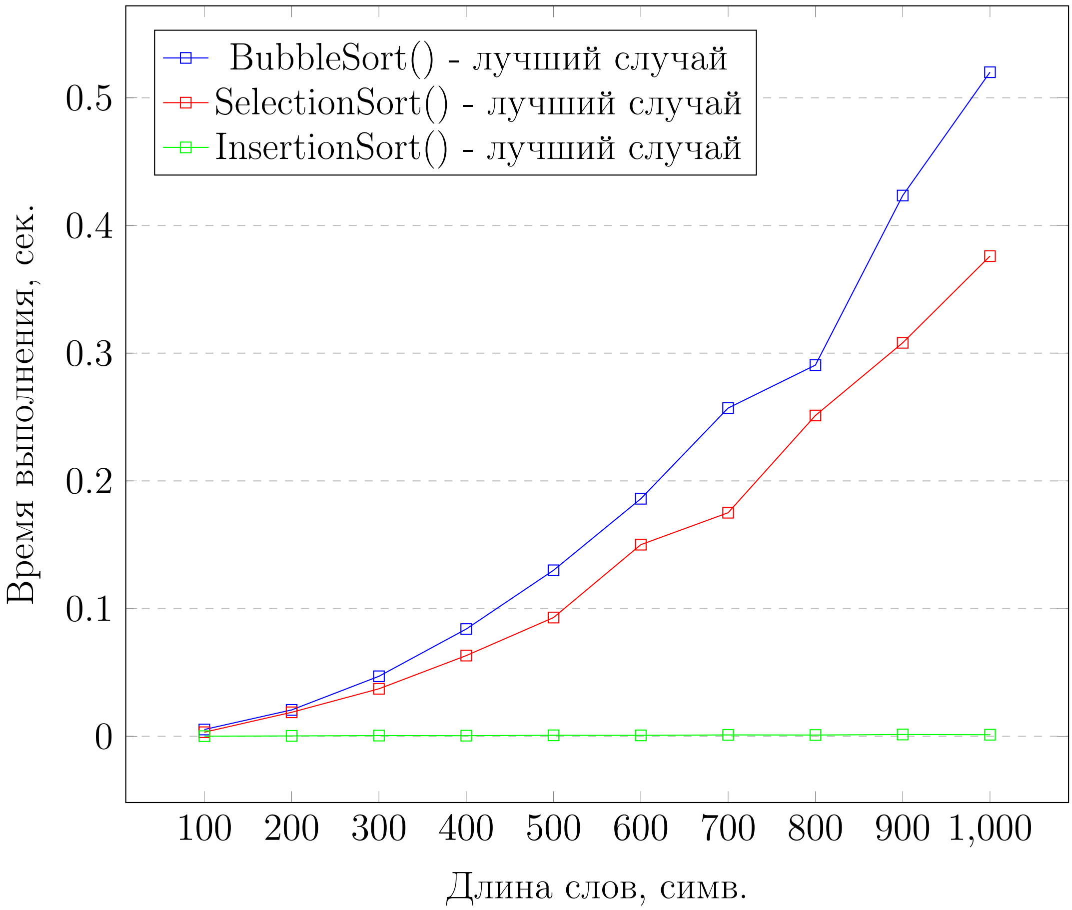


Рисунок 5.6 – Зависимость времени выполнения алгоритмов сортировок от длины массива в лучшем случае

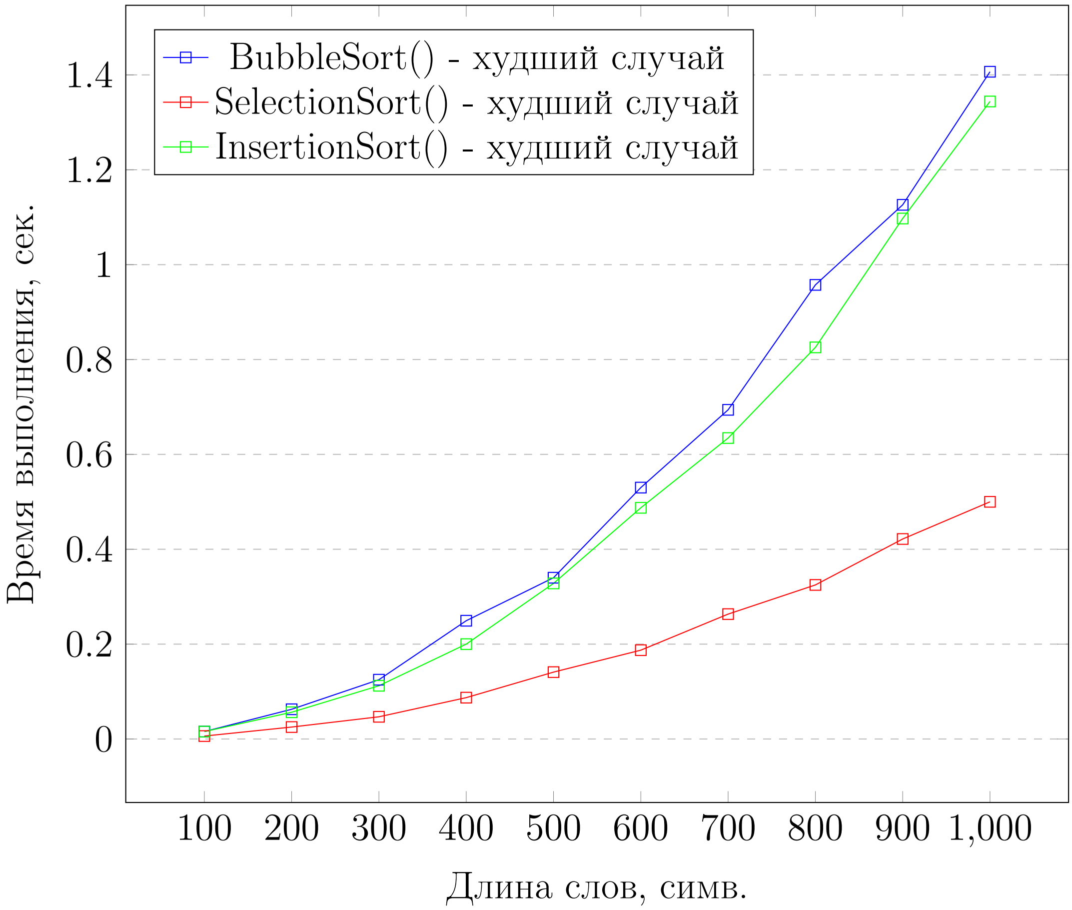


Рисунок 5.7 – Зависимость времени выполнения алгоритмов сортировок от длины массива в худшем случае

5.3 Оценка трудоёмкости

Для оценки трудоёмкости использовалась следующая модель вычислений: [7]

- Трудоёмкость следующих операций единична: +, -, =, +=, -=, ==, !=, <, >, -=, >=, «, », [];
- Трудоёмкость следующих операций = 2: *, /, %, /=, *=.

Трудоёмкость выбранных алгоритмов сортировок рассчитывалась по написанному коду.

На листинге 4.1 представлена программа для вычисления трудоёмкости алгоритма сортировки пузырьком для худшего случая.

Листинг 5.1 – Вычисление трудоёмкости алгоритма сортировки пузырьком

```
int getBubbleSort(int *l, int *r)
{
        int rez = 3; //init+srav
        for (int i = 0; i < r-l; i++)
        {
                rez += 3; //init+srav
                for (int *j = l; j < r-i; j++)
                {
                        if (*j > *(j+1))
                                swap(j, (j+1));
                        rez+=5; //telo j
                        rez++; //increment
                        rez+=2; //srav
                }
                rez++; //increment
                rez+=2; //srav
        }
        return rez;
}
```

Соответственно получается следующая формула трудоёмоксти:

$Fbubblesort = 3 + (3 * (N - 1) + ((N - 1) * N/2) * 8) + 3 * (N - 1)$, где N - размер массива

В лучшем же случае, не надо будет менять элементы местами, а значит трудоёмкость тела цикла по j уменьшится на 3 и формула примет вид:

$Fbubble\_sort = 3 + (3 * (N - 1) + ((N - 1) * N/2) * 5) + 3 * (N - 1)$

На листинге 4.2 представлена программа для вычисления трудоёмкости алгоритма сортировки выбором для худшего случая.

Листинг 5.2 – Вычисление трудоёмкости алгоритма сортировки выбором

```
int getSelectionSort(int *l, int *r)
{
	int rez = 2; //init+srav
	for (int *i = l; i <= r; i++)
	{
		rez += 2; //double = (assignment)
		int minz = *i, *ind = i;
		rez += 3; //init+srav
		for (int *j = i + 1; j <= r; j++)
		{
			if (*j < minz)
			{
				minz = *j;
				ind = j;
			}
			rez += 3; //telo j
			rez++; //increment
			rez++; //srav
		}
		rez+=3; //swap
		swap(i, ind);
		rez++; //increment
		rez++; //srav
	}
	return rez;
}
```

Следовательно формула трудоёмкости будет следующей:

$Fselection\_sort = 2 + 10 * N + ((N - 1) * N/2) * 5$

А в лучшем случае не будет выполнятся условие if и формула станет такой:

$$Fselection\_sort = 2 + 10 * N + ((N - 1) * N/2) * 3$$

На листинге 4.3 представлена программа для вычисления трудоёмкости алгоритма сортировки вставками для худшего случая.

Листинг 5.3 – Вычисление трудоёмкости алгоритма сортировки вставками

```
int getInsertionSort(int* l, int* r)
{
        int rez = 3; //init+srav
        for (int *i = l + 1; i <= r; i++)
        {
                rez++; //assigment
                int* j = i;
                rez+=3; //srav
                while (j > l && *(j - 1) > *j)
                {
                        rez+=4; //swap
                        swap((j - 1), j);
                        j--;
                        rez++; //deccrement
                        rez+=3; //srav
                }
        }
        return rez;
}
```

Формула трудоёмкости:

$Finsertion\_sort = 3 + 4 * (N - 1) + 8 * ((N - 1) * N/2)$

В лучшем случае полностью пропадает тело цикла while, а значит формула изменится на следующую:

$Finsertion\_sort = 3 + 4 * (N - 1)$

5.4 Вывод

По итогу иследования выяснилось, что разработанная программа работает верно, то-есть сортирует массивы по возрастанию. Кроме этого, смотря на время выполнения каждого алгоритма, логично сделать вывод, что наиболее быстрым в произвольном случае, является алгоритм сортировки выбором и судя по оценке трудоёмкости, наименее трудоёмким является также алгоритм сортировки выбором.

Заключение

По итогу проделанной работы была достигнута цель - изучены алгоритмы сортировки и получены навыки оценки трудоемкости алгоритмов.

Также были решены все поставленные задачи, а именно:

- реализованы 3 выбранных алгоритма сортировки;
- выполнена оценка времени выполнения алгоритмов сортировки;
- рассчитана трудоемкость каждого из алгоритма сортировки.

Список использованных источников

[1] Умножение матриц. [Электронный ресурс]. // URL: https://ru.onlinemschool.com/math/library/matrix/multiply/.

[2] И. В. Белоусов. *Матрицы и определители. Учебное пособие по линейной алгебре*, pages 1–16. – Институт прикладной физики, г. Кишинёв, 2006.

[3] F Le Gall. *Faster algorithms for rectangular matrix multiplication*, pages 514–523. – Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), 2012.

[4] Getprocesstimes function (processthreadsapi.h) [Электронный ресурс]. // URL: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocesstimes#syntax.

[5] Как применить настройки оптимизации gcc в qt? // URL: http://blog.kislenko.net/show.php?id=1991.

[6] Опанасенко М. Описание алгоритмов сортировки и сравнение их производительности [Электронный ресурс]. // URL: https://habr.com/ru/post/335920/.

[7] Ульянов М.В. *Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ.* Москва: ФИЗМАТЛИТ, 2008. 304 с.