



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 | 3 |
| 1 Аналитический раздел | 4 |
| 1.1 Умножение матриц | 4 |
| 1.2 Классический алгоритм умножения матриц | 5 |
| 1.3 Алгоритм Винограда | 5 |
| 2 Конструкторский раздел | 7 |
| 2.1 Требования к ПО | 7 |
| 2.2 Схемы алгоритмов | 7 |
| 2.3 Вывод | 11 |
| 3 Технологический раздел | 12 |
| 3.1 Выбор инструментов | 12 |
| 3.2 Реализация алгоритмов | 12 |
| 3.3 Тестирование | 14 |
| 3.4 Вывод | 16 |
| 4 Исследовательский раздел | 17 |
| 4.1 Примеры работы программы | 17 |
| 4.2 Сравнительный анализ времени выполнения алгоритмов . | 18 |
| 4.3 Оценка трудоёмкости | 25 |
| 4.4 Вывод | 27 |
| Заключение | 28 |
| Список использованных источников | 29 |

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 $A_{m \times n}$ и $B_{n \times k}$ будет матрица $C_{m \times k}$ такая, что элемент матрицы C , стоящий в i -той строке и j -том столбце (c_{ij}), равен сумме произведений элементов i -той строки матрицы A на соответствующие элементы j -того столбца матрицы B :

$$c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \dots + a_{in} \cdot b_{nj} [1]$$

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)

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

где элементы матрицы равны (1.2)

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

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 $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно (1.3)

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4 \quad (1.3)$$

Равенство (1.3) можно переписать в виде (1.4)

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

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 ?? - ??);

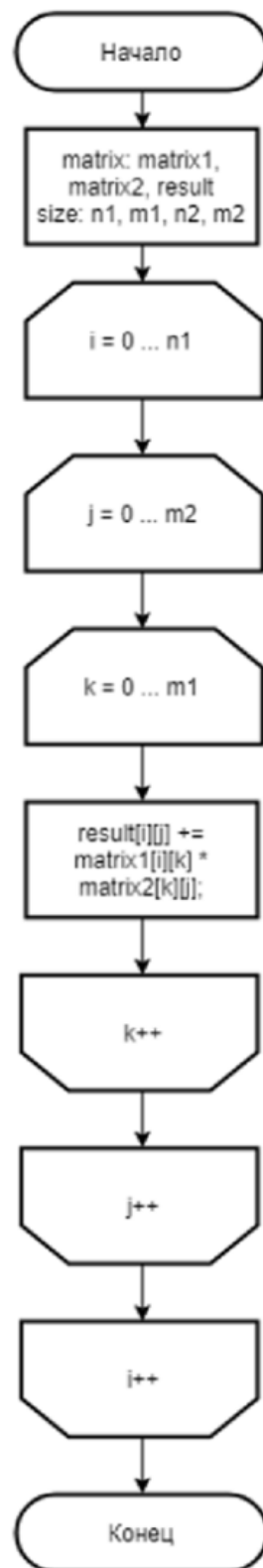


Рис. 2.1: Схема классического алгоритма умножения матриц

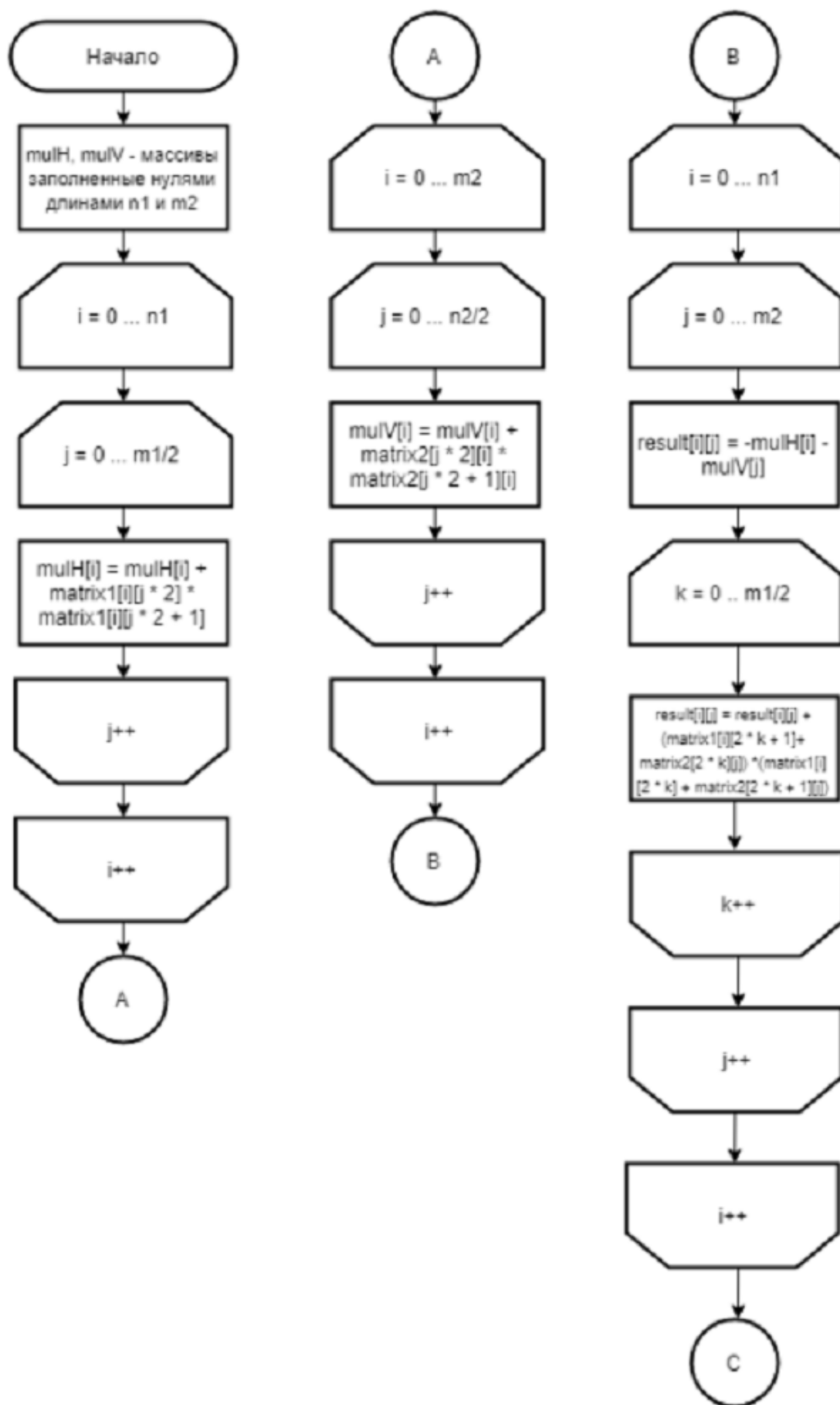


Рис. 2.2: Схема алгоритма Винограда

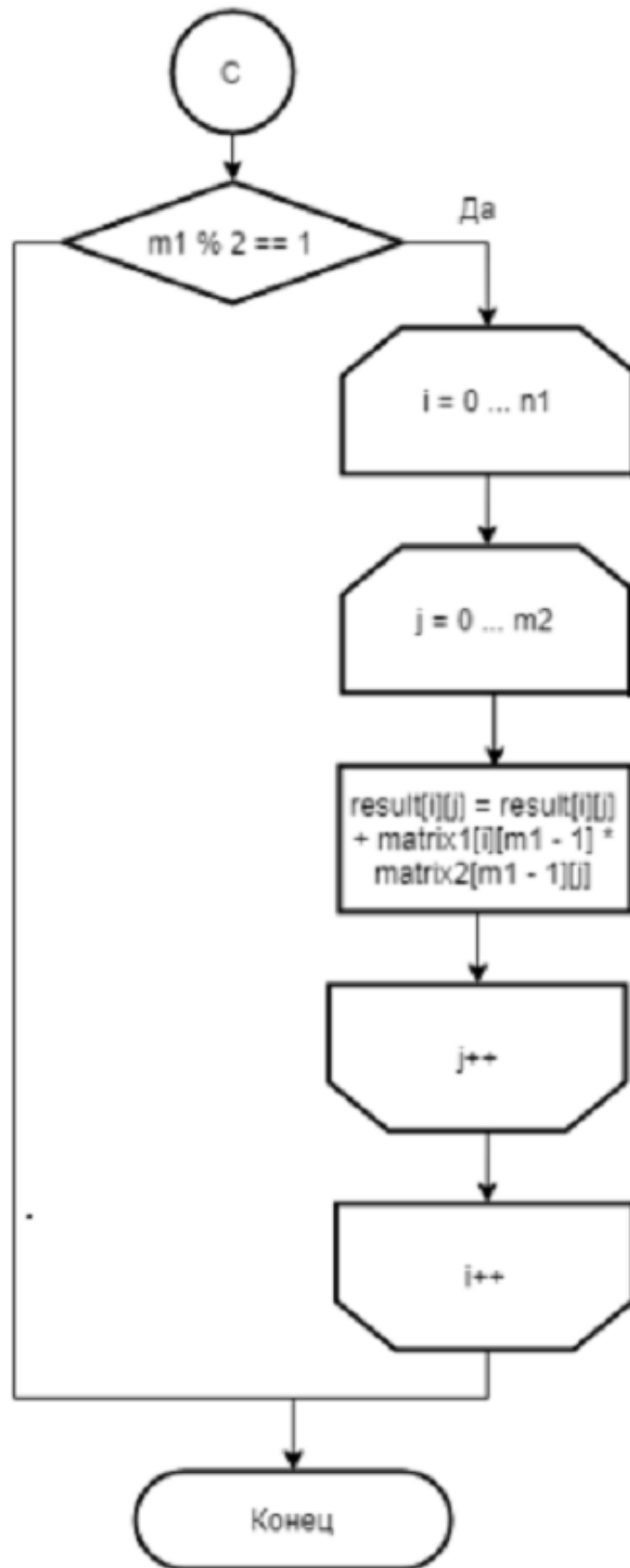


Рис. 2.3: Схема алгоритма Винограда (продолжение)

2.3 Вывод

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

3 Технологический раздел

В данном разделе 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 реализации алгоритмов был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответственно для компиляции кода будет использоваться компилятор `G++`.

Чтобы оценить время выполнения программы будет замеряться 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

```
1 int standartMult(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrx2, int n2, int m2)
2 {
3     if (m1 != n2)
4         return SIZE_ERROR;
5
6     for (int i = 0; i < n1; i++)
7         for (int j = 0; j < m2; j++)
8         {
9             rez[i][j] = 0;
10            for (int k = 0; k < n2; k++)
11                rez[i][j] = rez[i][j] + mtrx1[i][k] * mtrx2[k][j];
12        }
13    return 0;
14 }
```

Листинг 3.2: Алгоритм Винограда

```
1 int vinograd(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrx2, int n2, int m2)
2 {
3     if (m1 != n2)
4         return SIZE_ERROR;
5     vector<int> mulH(n1, 0);
6     for (int i = 0; i < n1; i++)
7         for (int j = 0; j < m1 / 2; j++)
8             mulH[i] = mulH[i] + mtrx1[i][j * 2] * mtrx1[i][j * 2 + 1];
9
10    vector<int> mulV(n1, 0);
11    for (int i = 0; i < m2; i++)
12        for (int j = 0; j < n2 / 2; j++)
13            mulV[i] = mulV[i] + mtrx2[j * 2][i] * mtrx2[j * 2 + 1][i];
14
15    for (int i = 0; i < n1; i++)
16        for (int j = 0; j < m2; j++)
17        {
18            rez[i][j] = -mulH[i] - mulV[j];
19            for (int k = 0; k < n2 / 2; k++)
20                rez[i][j] = rez[i][j] + (mtrx1[i][2 * k + 1] + mtrx2[2 * k][j]) *
21                    (mtrx1[i][2 * k] + mtrx2[2 * k + 1][j]);
22        }
23
24    if (n2 % 2 == 1)
25        for (int i = 0; i < n1; i++)
26            for (int j = 0; j < m2; j++)
27                rez[i][j] = rez[i][j] + mtrx1[i][n2 - 1] * mtrx2[n2 - 1][j];
28    return 0;
29 }
```

Листинг 3.3: Оптимизированный алгоритм Винограда

```

1 int optimizedVinograd(mtrx &rez, mtrx mtrx1, int n1, int m1, mtrx mtrx2, int n2, int
  m2)
2 {
3     if (m1 != n2)
4         return SIZE_ERROR;
5
6     vector<int> mulH(n1, 0);
7     for (int i = 0; i < n1; i++)
8         for (int j = 0; j < m1 - 1; j += 2)
9             mulH[i] -= mtrx1[i][j] * mtrx1[i][j + 1];
10
11     vector<int> mulV(n1, 0);
12     for (int i = 0; i < m2; i++)
13         for (int j = 0; j < n2 - 1; j += 2)
14             mulV[i] -= mtrx2[j][i] * mtrx2[j + 1][i];
15
16     bool flag = false;
17     if (n2 % 2 == 1)
18         flag = true;
19
20     for (int i = 0; i < n1; i++)
21         for (int j = 0; j < m2; j++)
22         {
23             rez[i][j] = mulH[i] + mulV[j];
24             for (int k = 0; k < n2 - 1; k += 2)
25                 rez[i][j] += (mtrx1[i][k + 1] + mtrx2[k][j]) * (mtrx1[i][k] + mtrx2[k
26                     + 1][j]);
27
28             if (flag)
29                 rez[i][j] += mtrx1[i][n2 - 1] * mtrx2[n2 - 1][j];
30         }
31     return 0;
32 }

```

STOPED HERE !!!

3.3 Тестирование

Для проверки написанных алгоритмов были подготовлены функциональные тесты с двумя массивами:

- mas0 = пустой массив
- mas1 = {1};

- $\text{mas10} = \{10, 7, 9, 5, 3, 1, 8, 2, 4, 6\}$.

Правильными результатами тестов для них будут следующие массивы:

- `sorted0` = пустой массив
- `sorted1` = $\{1\}$;
- `sorted10` = $\{1,2,3,4,5,6,7,8,9,10\}$.

На рисунке 3.1 приведены результаты тестирования.

```
mas0 =
sorted0 by BubbleSort =
BbubbleSort() with mas0 is correct!
sorted0 by BubbleSort =
SelectionSort() with mas0 is correct!
sorted0 by BubbleSort =
InsertionSort() with mas0 is correct!
mas1 = 10
sorted1 by BubbleSort = 10
BbubbleSort() with mas1 is correct!
sorted1 by SelectionSort = 10
SelectionSort() with mas1 is correct!
sorted1 by InsertionSort = 10
SelectionSort() with mas1 is correct!
mas10 = 10 7 9 5 3 1 8 2 4 6
sorted10 by BubbleSort = 1 2 3 4 5 6 7 8 9 10
BbubbleSort() with mas10 is correct!
sorted10 by SelectionSort = 1 2 3 4 5 6 7 8 9 10
SelectionSort() with mas10 is correct!
sorted10 by InsertionSort = 1 2 3 4 5 6 7 8 9 10
InsertionSort() with mas10 is correct!
```

Рис. 3.1: Результаты функционального тестирования

Как видно по рисунку, функциональные тесты пройдены.

3.4 Вывод

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

4 Исследовательский раздел

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

4.1 Примеры работы программы

На рисунках 4.1-4.3 представлены результаты работы программы для массивов разных длин, заполненных случайными значениями.

```
Array:  
46 22  
BubbleSort:  
22 46  
SelectionSort:  
22 46  
InsertionSort:  
22 46
```

Рис. 4.1: Результаты сортировки массива с размером = 2

```
Array:  
59 34 92 33 4 22 49 38 60 99  
BubbleSort:  
4 22 33 34 38 49 59 60 92 99  
SelectionSort:  
4 22 33 34 38 49 59 60 92 99  
InsertionSort:  
4 22 33 34 38 49 59 60 92 99
```

Рис. 4.2: Результаты сортировки массива с размером = 10

```

Array:
55 31 91 52 25 47 63 3 37 0 41 75 69 71 27 93 96 6 52 85 32 51 5 79 95 13 60 83 98 97
BubbleSort:
0 3 5 6 13 25 27 31 32 37 41 47 51 52 52 55 60 63 69 71 75 79 83 85 91 93 95 96 97 98
SelectionSort:
0 3 5 6 13 25 27 31 32 37 41 47 51 52 52 55 60 63 69 71 75 79 83 85 91 93 95 96 97 98
InsertionSort:
0 3 5 6 13 25 27 31 32 37 41 47 51 52 52 55 60 63 69 71 75 79 83 85 91 93 95 96 97 98

```

Рис. 4.3: Результаты сортировки массива с размером = 30

4.2 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 числами 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]

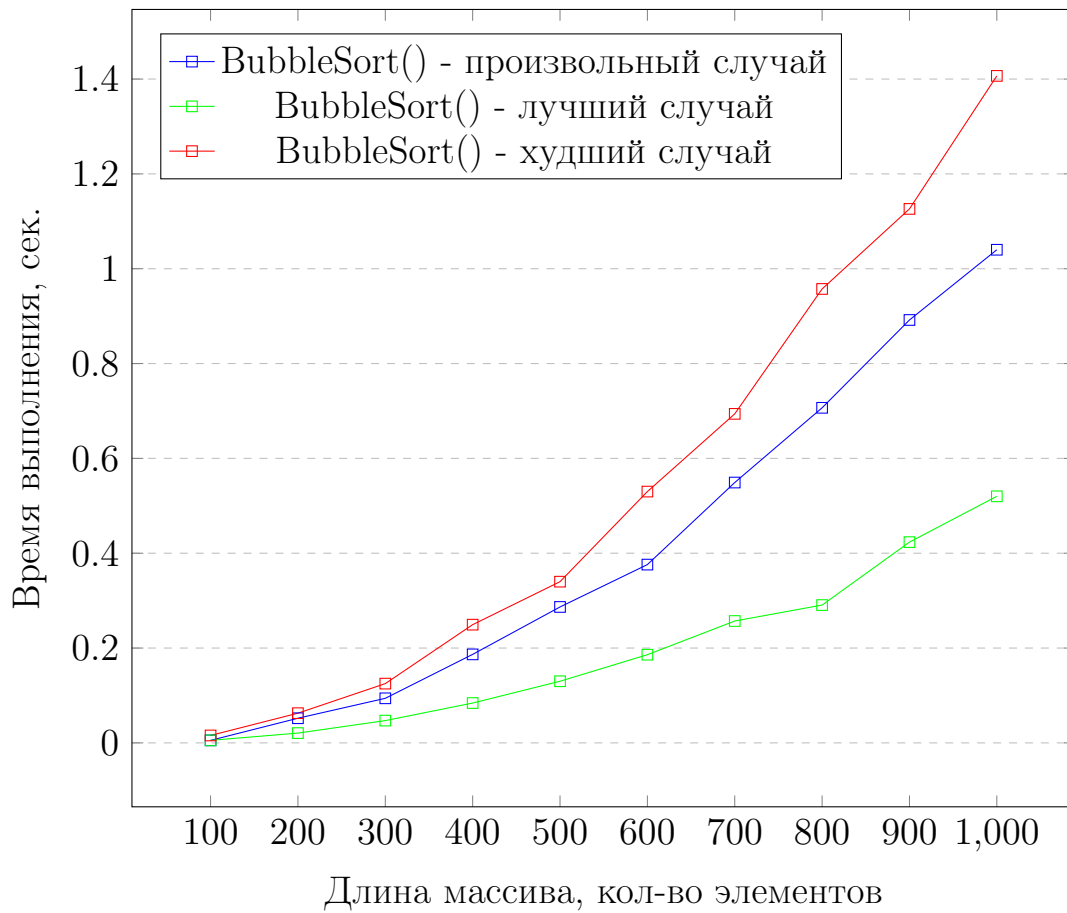


Рис. 4.4: Зависимость времени выполнения сортировки пузырьком от длины массива в разных случаях

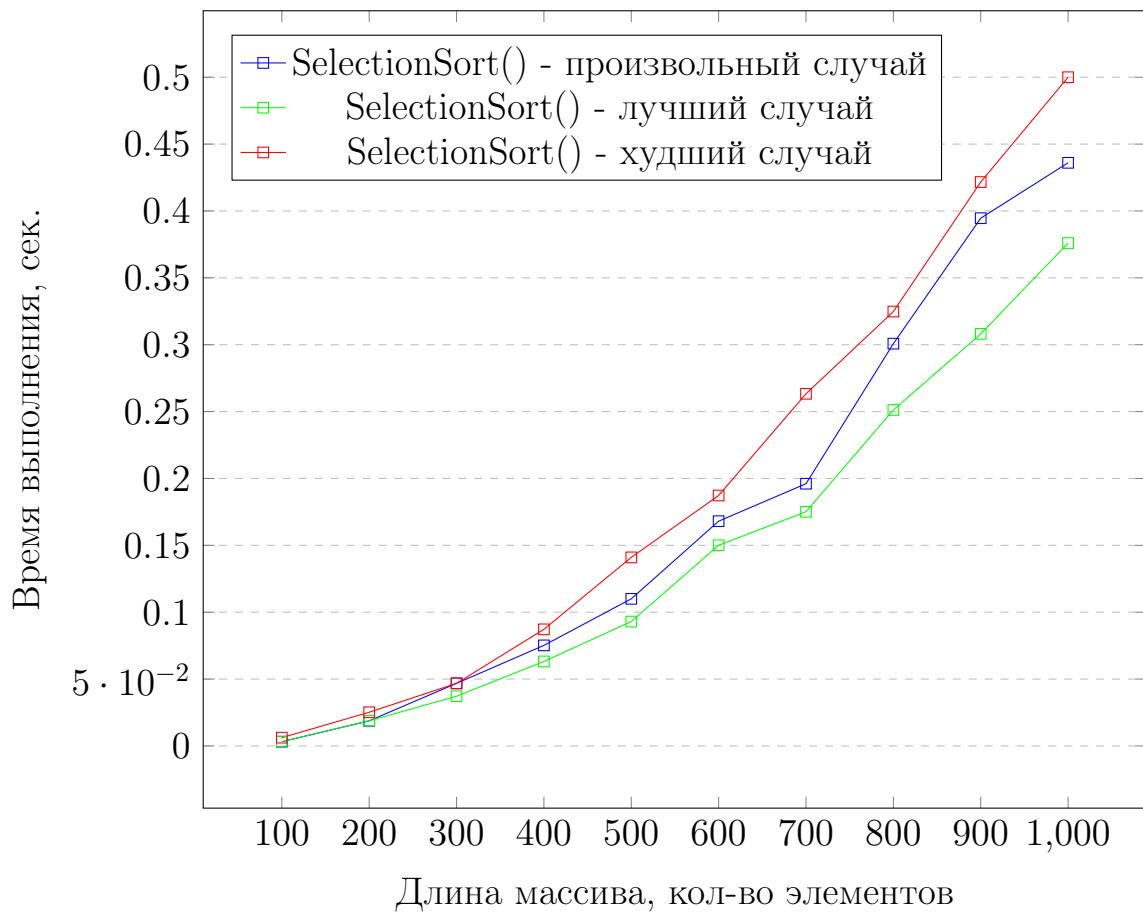


Рис. 4.5: Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

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]

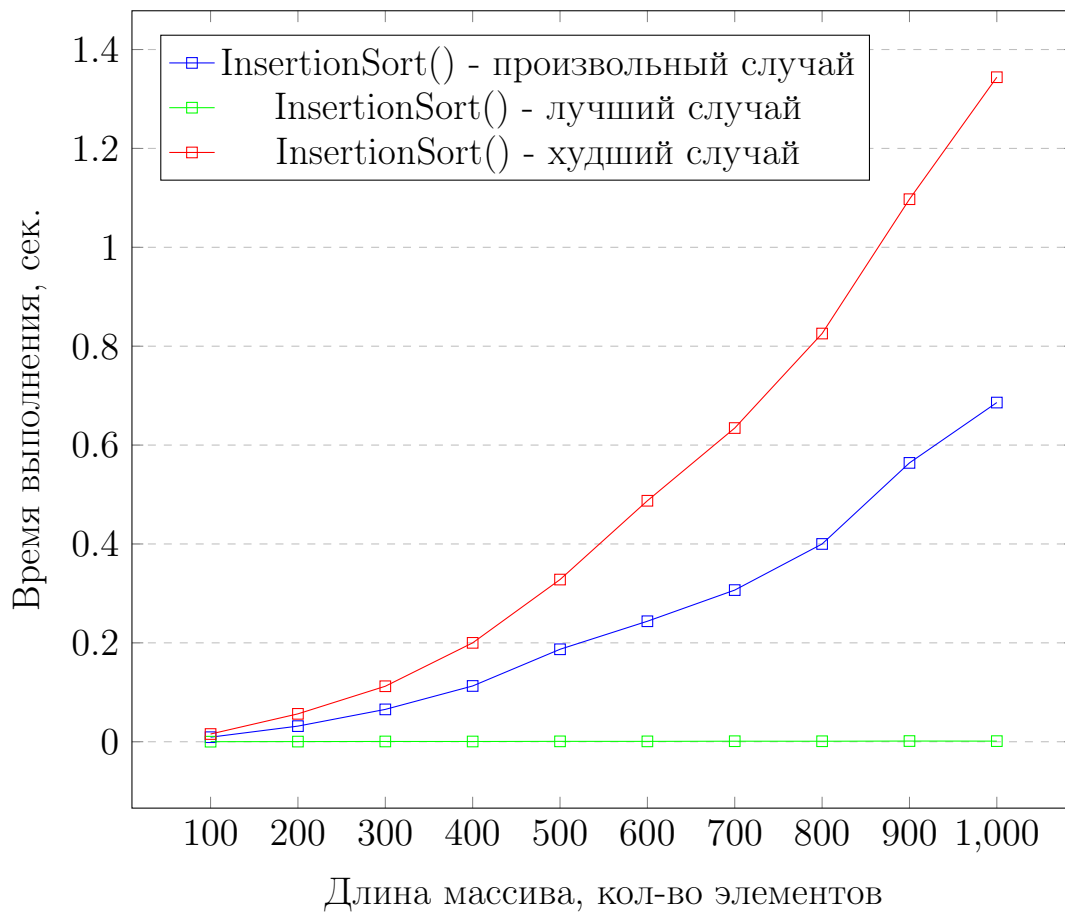


Рис. 4.6: Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

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]

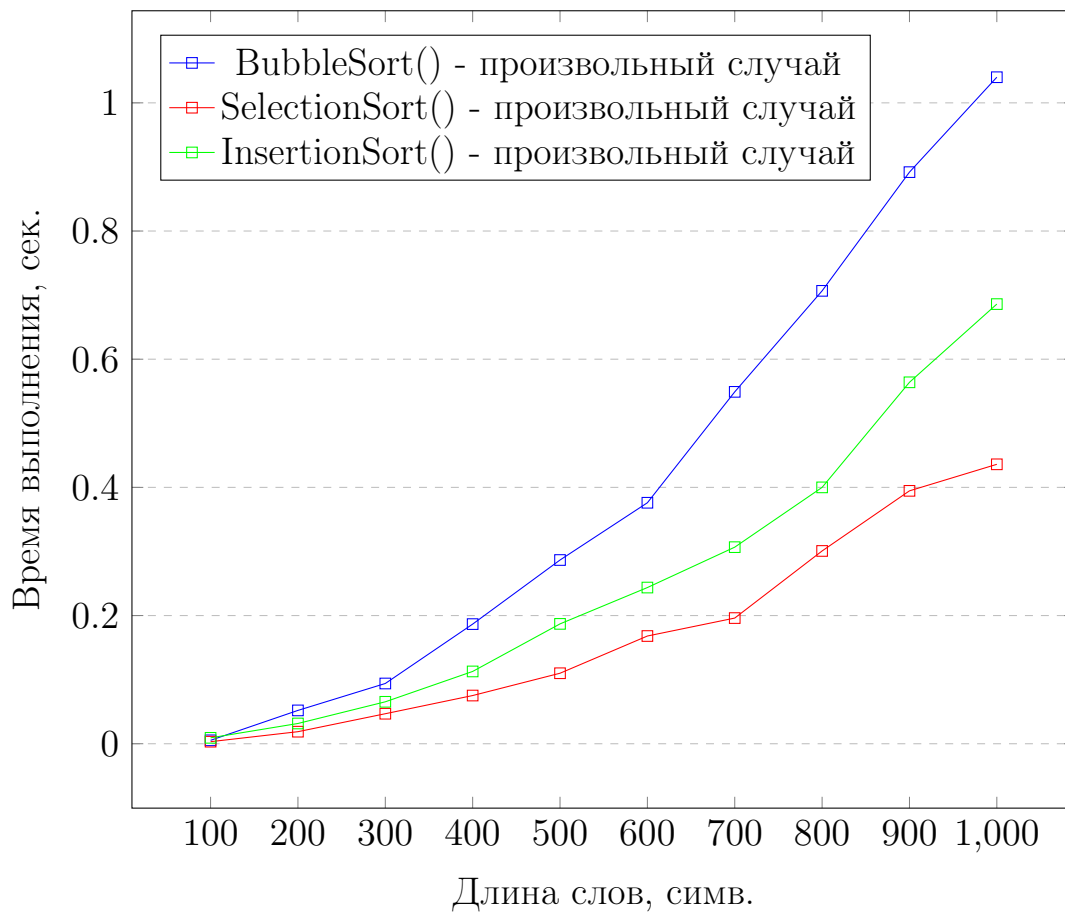


Рис. 4.7: Зависимость времени выполнения алгоритмов сортировок от длины массива в произвольном случае

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

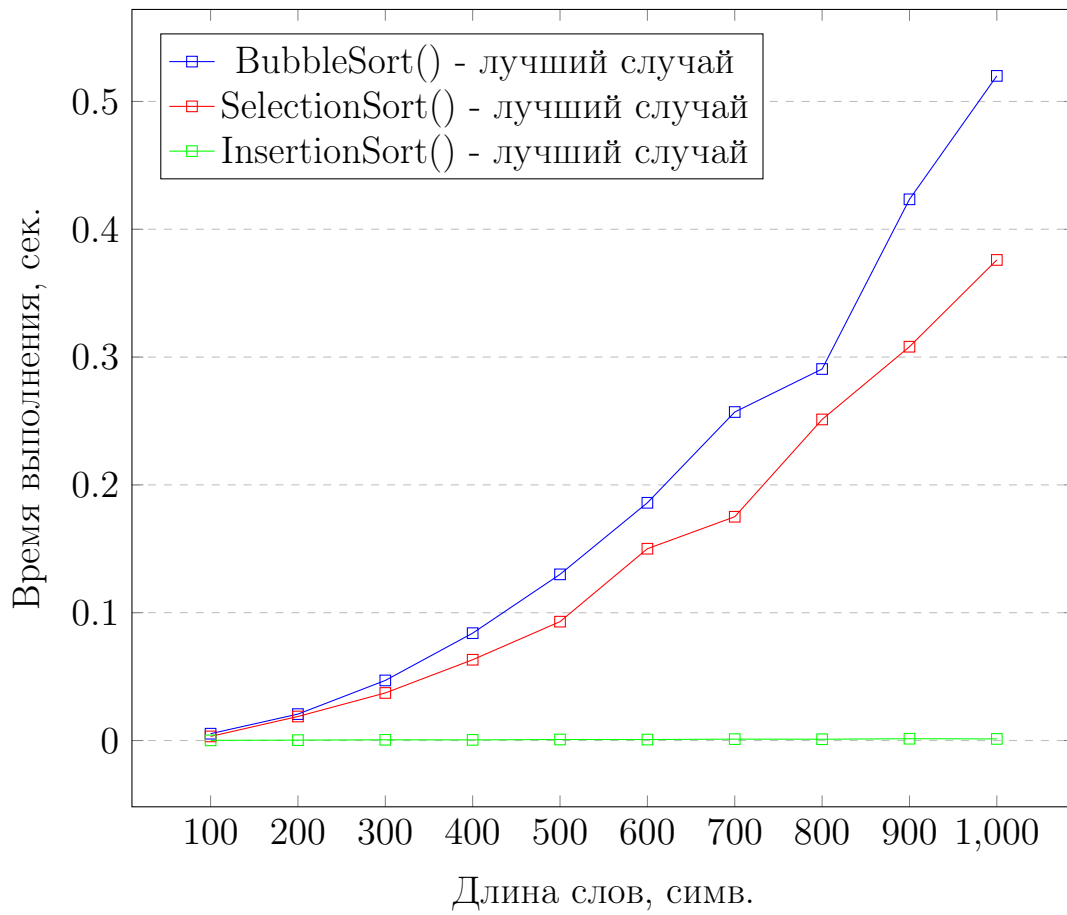


Рис. 4.8: Зависимость времени выполнения алгоритмов сортировок от длины массива в лучшем случае

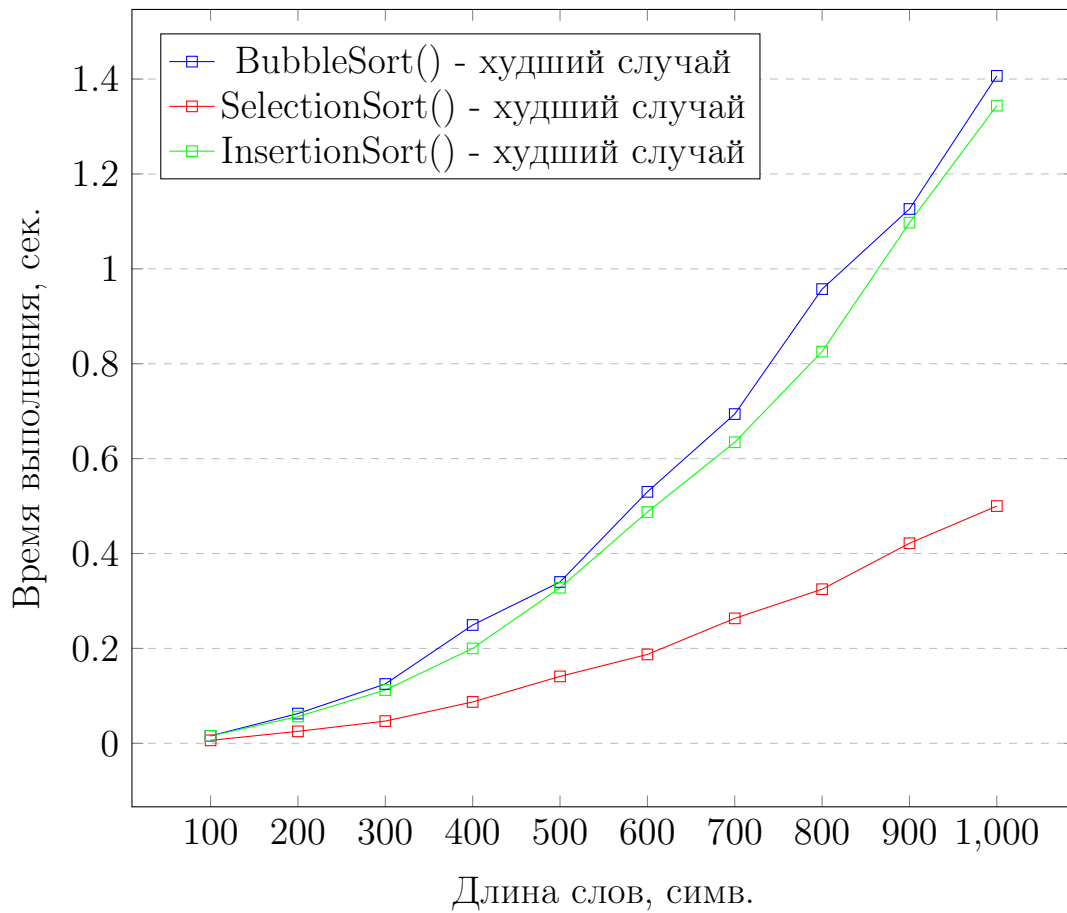


Рис. 4.9: Зависимость времени выполнения алгоритмов сортировок от длины массива в худшем случае

4.3 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 4.1: Вычисление трудоёмкости алгоритма сортировки пузырьком

```
1 int getBubbleSort(int *l, int *r)
2 {
3     int rez = 3; //init+srav
4     for (int i = 0; i < r-l; i++)
5     {
6         rez += 3; //init+srav
7         for (int *j = l; j < r-i; j++)
8         {
9             if (*j > *(j+1))
10                swap(j, (j+1));
11            rez+=5; //telo j
12            rez++; //increment
13            rez+=2; //srav
14        }
15        rez++; //increment
16        rez+=2; //srav
17    }
18    return rez;
19 }
```

Соответственно получается следующая формула трудоёмкости:

$$F_{bubblesort} = 3 + (3 * (N - 1) + ((N - 1) * N / 2) * 8) + 3 * (N - 1),$$

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:

$$F_{bubble_sort} = 3 + (3 * (N - 1) + ((N - 1) * N / 2) * 5) + 3 * (N - 1)$$

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

Листинг 4.2: Вычисление трудоёмкости алгоритма сортировки выбором

```
1 int getSelectionSort(int *l, int *r)
2 {
3     int rez = 2; //init+srav
4     for (int *i = l; i <= r; i++)
5     {
6         rez += 2; //double = (assignment)
7         int minz = *i, *ind = i;
8         rez += 3; //init+srav
9         for (int *j = i + 1; j <= r; j++)
10        {
11            if (*j < minz)
12            {
13                minz = *j;
14                ind = j;
15            }
16            rez += 3; //telo j
17            rez++; //increment
18            rez++; //srav
19        }
20        rez+=3; //swap
21        swap(i, ind);
22        rez++; //increment
23        rez++; //srav
24    }
25    return rez;
26 }
```

Следовательно формула трудоёмкости будет следующей:

$$F_{selection_sort} = 2 + 10 * N + ((N - 1) * N/2) * 5$$

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

$$F_{selection_sort} = 2 + 10 * N + ((N - 1) * N/2) * 3$$

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

Листинг 4.3: Вычисление трудоёмкости алгоритма сортировки вставками

```
1 int getInsertionSort(int* l, int* r)
2 {
3     int rez = 3; //init+srav
4     for (int *i = l + 1; i <= r; i++)
5     {
6         rez++; //assigment
7         int* j = i;
8         rez+=3; //srav
9         while (j > l && *(j - 1) > *j)
10        {
11            rez+=4; //swap
12            swap((j - 1), j);
13            j--;
14            rez++; //deccrement
15            rez+=3; //srav
16        }
17    }
18    return rez;
19 }
```

Формула трудоёмкости:

$$F_{insertion_sort} = 3 + 4 * (N - 1) + 8 * ((N - 1) * N / 2)$$

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

$$F_{insertion_sort} = 3 + 4 * (N - 1)$$

4.4 Вывод

По итогу исследования 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.onlimeschool.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.