|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования |
| **«МИРЭА – Российский технологический университет»** |
| **РТУ МИРЭА** |
|  |

|  |  |
| --- | --- |
| **Отчет по выполнению практического задания № 1** | |
| **Тема:** | |
| **«Оценка вычислительной сложности алгоритма»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Жаворонкова А.А. |
|  |  |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[**1 ЦЕЛЬ 2**](#_Toc95918683)

[**2 ЗАДАНИЕ 1 4**](#_Toc95918684)

[**1.1 Первый алгоритм 4**](#_Toc95918685)

[**1.1.1 Описание математической модели 4**](#_Toc95918686)

[**1.1.2 Блок-схема алгоритма, доказательство корректности циклов 4**](#_Toc95918687)

[**1.1.3 Определение вычислительной сложности алгоритма 5**](#_Toc95918688)

[**1.1.5 Тестирование 7**](#_Toc95918689)

[**1.2 Второй алгоритм 10**](#_Toc95918690)

[**1.2.1 Описание математической модели 10**](#_Toc95918691)

[**1.2.2 Блок-схема алгоритма, доказательство корректности циклов 10**](#_Toc95918692)

[**1.2.3 Определение вычислительной сложности алгоритма 11**](#_Toc95918693)

[**1.2.4 Реализация алгоритма на языке С++ 12**](#_Toc95918694)

[**1.2.5 Тестирование 12**](#_Toc95918695)

[**1.3 Выводы по заданию 1 15**](#_Toc95918696)

[**3 ЗАДАНИЕ 2 16**](#_Toc95918697)

[**2.1 Описание математической модели 16**](#_Toc95918698)

[**2.2 Блок-схема алгоритма, доказательство корректности циклов 16**](#_Toc95918699)

[**2.3 Определение вычислительной сложности алгоритма 17**](#_Toc95918700)

[**2.4 Реализация алгоритма на языке С++ 18**](#_Toc95918701)

[**2.5 Тестирование 19**](#_Toc95918702)

[**2.6 Выводы по заданию 2 21**](#_Toc95918703)

[**4 ВЫВОДЫ 22**](#_Toc95918704)

# 1 ЦЕЛЬ

Приобретение практических навыков:

* Эмпирическому определению вычислительной сложности алгоритмов на теоретическом и практическом уровнях;
* Выбору эффективного алгоритма решения вычислительной задачи из нескольких.
* Разработка собственного алгоритма в соответствии с задачей.

# 2 ЗАДАНИЕ 1

**Формулировка задания:** выбрать эффективный алгоритм вычислительной задачи из двух предложенных, используя теоретическую и практическую оценку вычислительной сложности каждого из алгоритмов, а также его емкостную сложность. Пусть имеется вычислительная задача:

– дан массив х из n элементов целого типа; удалить из этого массива все значения равные заданному (ключевому) key.

Удаление состоит в уменьшении размера массива с сохранением порядка следования всех элементов, как до, так и следующих после удаляемого.

## 

## **1.1 Первый алгоритм**

### **1.1.1 Описание математической модели**

С помощью цикла идем по массиву с первого элемента до n-ного, где n – размер массива. Если текущий элемент равен заданному значению, то смещаем все следующие значения в массиве на 1 позицию влево, тем самым заменяя и удаляя требуемый элемент. Переменную n, отвечающую за размер массива, уменьшаем на 1.

### **1.1.2 Блок-схема алгоритма, доказательство корректности циклов**

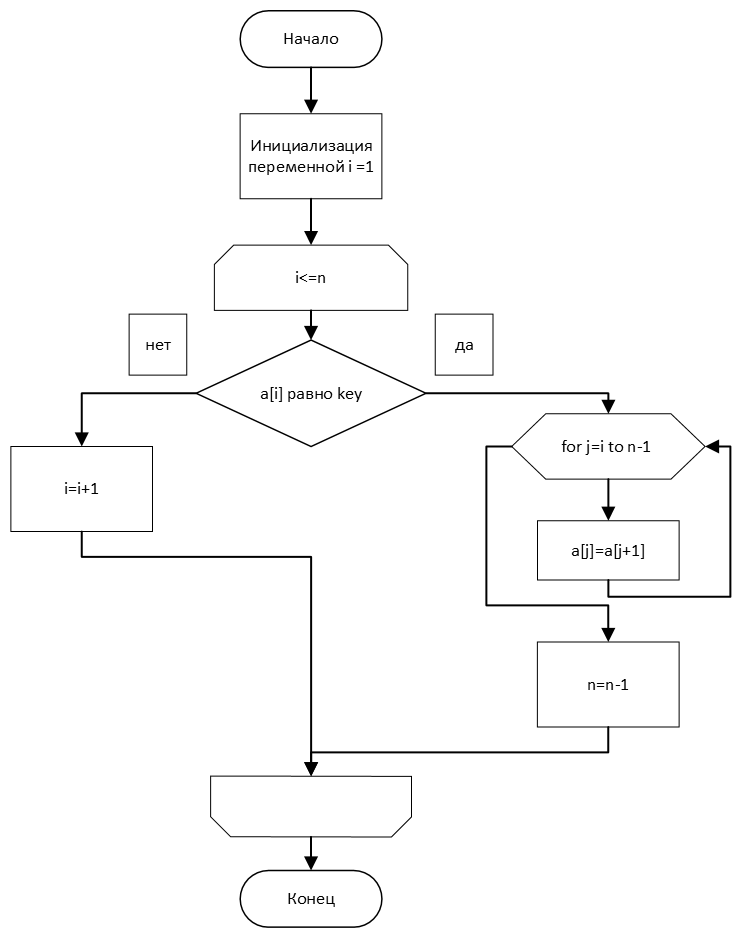


Рисунок 1 – Блок-схема первого алгоритма

Определим инвариант для внешнего цикла: ни одно значение, индекс которого меньше i, не равно удаляемому значению key

Определим инвариант для внутреннего цикла: j всегда не больше n

Доказательство конечности цикла: при каждой итерации по переменной i область неопределённости сужается на 1 элемент. До начала цикла не просмотрено n элементов, после первой итерации n-1, после второй n-2 и так далее. После n-ной итерации будет не просмотрено n-n=0 элементов, следовательно цикл завершится.

Таким образом, все циклы алгоритма корректны, а значит и сам алгоритм, корректен.

### **1.1.3 Определение вычислительной сложности алгоритма**

|  |  |  |
| --- | --- | --- |
| Номер строки | Алгоритм, записанный на псевдокоде | Количество повторений действия в  зависимости от объема входных  данных n |
| 1 | delFirstMetod(x,n,key){ |  |
| 2 | i←1 | 1 |
| 3 | while (i<=n) **do** | n+1 |
| 4 | if x[i]=key then | n |
| 5 | for j←i to n-1 do |  |
| 6 | x[j] ←x[j+1] |  |
| 7 | оd |  |
| 8 | n←n-1 | n |
| 9 | else |  |
| 10 | i←i+1 |  |
| 11 | endif |  |
| 12 | **od** |  |
| 13 | } |  |

Количество повторений действия в строке 6 представляет собой арифметическую прогрессию. Найдем ее сумму  *.*

Тогда общая вычислительная сложность алгоритма в худшем случае определяется функцией. То есть алгоритм имеет квадратичный порядок роста времени вычисления.

В лучшем случае, когда ни один элемент удалять не нужно, сложность определяется функцией T(n)=2n+2.

**1.1.4 Реализация алгоритма на языке С++**

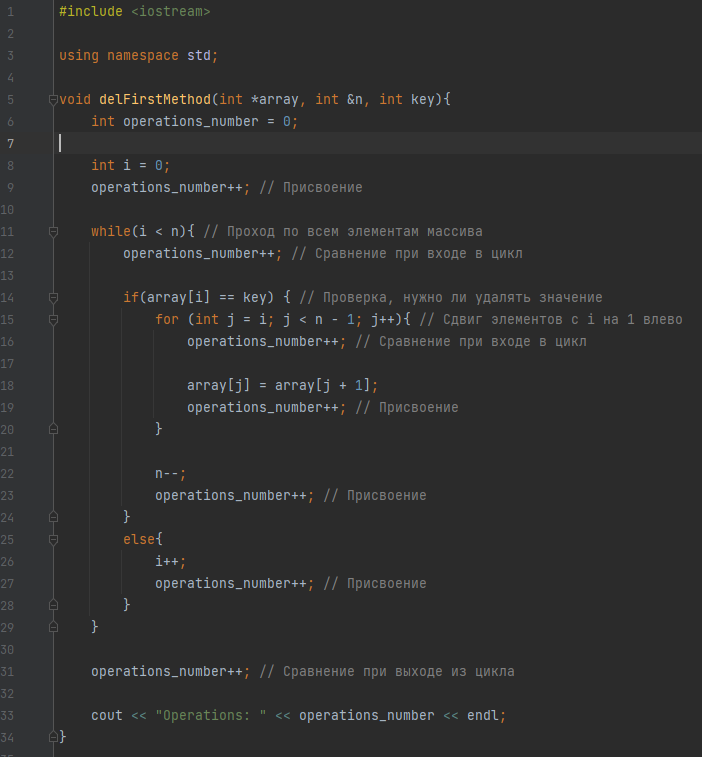
****

Рисунок 2 – Первый алгоритм удаления

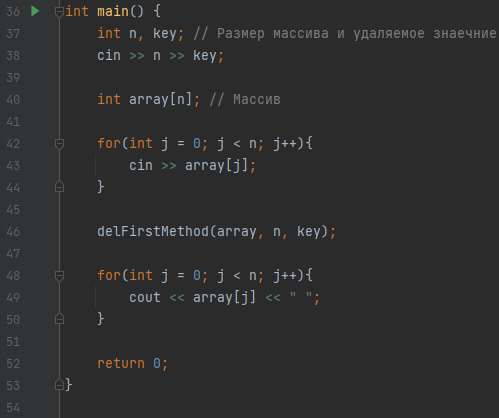
****

Рисунок 3 – Функция main, обеспечивающая работу программы

### **1.1.5 Тестирование**

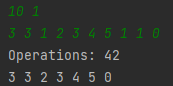


Рисунок 5 – Тестирование при 10 элементах. Случайная ситуация.

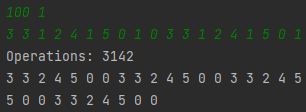


Рисунок 6 – Тестирование при 100 элементах. Случайная ситуация.

Теоретическая сложность вычисления при 10 элементах, когда ничего не нужно удалять, T(10)=2\*10+2=22

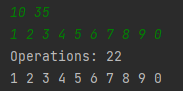


Рисунок 7 – Тестирование при 10 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 100 элементах, когда ничего не нужно удалять, T(100)=2\*100+2=202

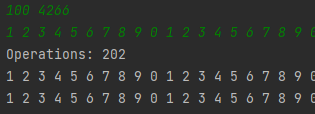


Рисунок 8 – Тестирование при 100 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 10 элементах, когда все нужно удалять, T(10)=10\*10+2\*10+2= 122

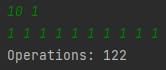


Рисунок 9 – Тестирование при 10 элементах. Все нужно удалить

Теоретическая сложность вычисления при 100 элементах, когда все нужно удалять, T(10)= 100\*100+2\*100+2= 10202

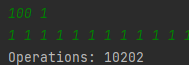


Рисунок 10 – Тестирование при 100 элементах. Все нужно удалить

Как мы видим, реальные результаты близки к теоретическим, квадратичная зависимость прослеживается, значит теоретический расчет верный.

## **1.2 Второй алгоритм**

### **1.2.1 Описание математической модели**

С помощью цикла идем по массиву с первого элемента до n-ного, где n – размер массива. В переменной i хранится номер рассматриваемого элемента исходного массива, в переменной j хранится номер размещаемого в данный момент элемента в конечном массиве. В j тый элемент постоянно помещаем i тый, но увеличиваем j на 1 ( то есть размещаем теперь в следующий элемент конечного массива) только если текущий не равен искомому значению.

### **1.2.2 Блок-схема алгоритма, доказательство корректности циклов**

Рисунок 11 – Блок-схема второго алгоритма

Определим инвариант для цикла: j всегда не больше i и элементы с номерами меньшими j не содержат значения key.

Доказательство конечности цикла: при каждой итерации область неопределённости сужается на 1 элемент. До начала цикла не просмотрено n элементов, после первой итерации n-1, после второй n-2 и так далее. После n-ной итерации будет не просмотрено n-n=0 элементов, следовательно цикл завершится.

Таким образом, цикл алгоритма корректен, а значит и сам алгоритм, корректен.

### **1.2.3 Определение вычислительной сложности алгоритма**

|  |  |  |
| --- | --- | --- |
| Номер строки | Алгоритм, записанный на псевдокоде | Количество повторений действия в  зависимости от объема входных  данных n |
| 1 | delOtherMetod(x,n,key){ |  |
| 2 | j←1 | 1 |
| 3 | for i←1 to n **do** | n+1 |
| 4 | x[j]=x[i]; | n |
| 5 | if x[i]!=key then | n |
| 6 | j++ | n |
| 7 | endif |  |
| 8 | **od** |  |
| 9 | **n**←j | 1 |
| 10 | } |  |

Общая вычислительная сложность алгоритма в худшем случае определяется функцией. То есть алгоритм имеет линейны1 порядок роста времени вычисления.

В лучшем случае, когда все нужно удалять, сложность определяется функцией T(n)=2n+3.

### **1.2.4 Реализация алгоритма на языке С++**

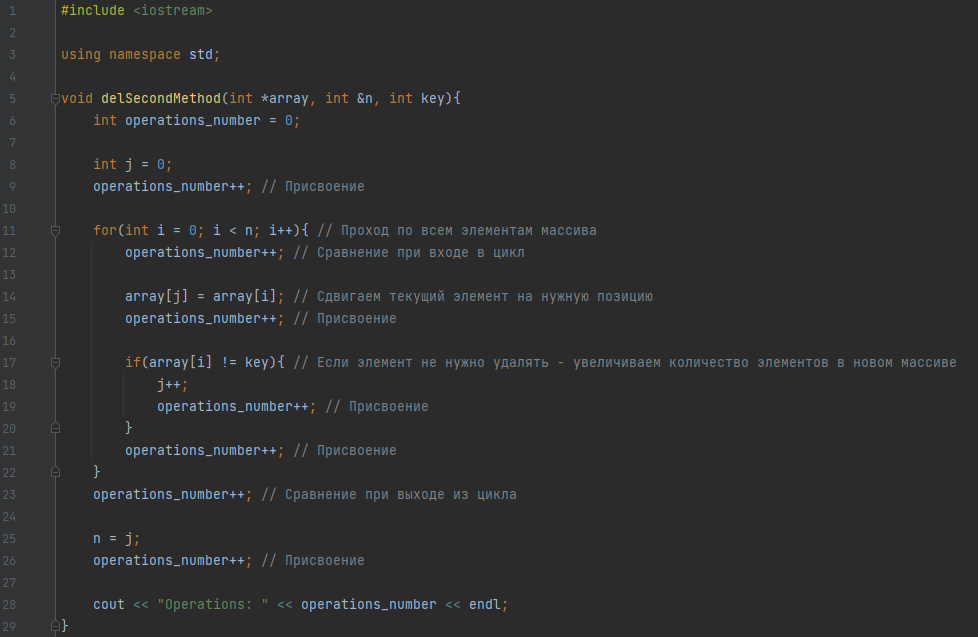


Рисунок 12 – Второй алгоритм удаления

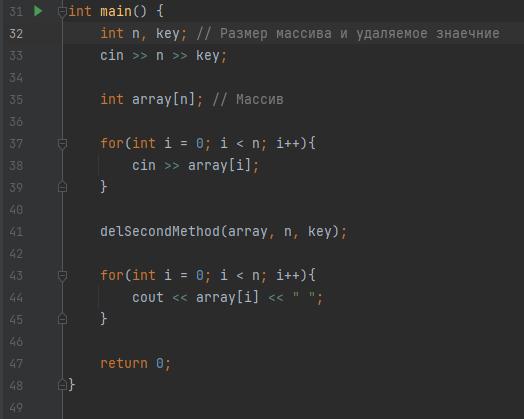


Рисунок 13 – Функция main

Остальные функции (заполнение и вывод) остались без изменения.

### **1.2.5 Тестирование**

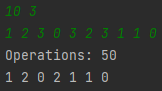


Рисунок 14 – Тестирование при 10 элементах. Случайная ситуация.

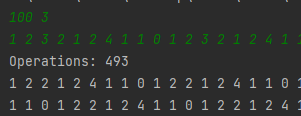


Рисунок 15 – Тестирование при 100 элементах. Случайная ситуация.

Теоретическая сложность вычисления при 10 элементах, когда ничего не нужно удалять, T(10)=4\*10+3=43

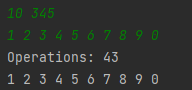


Рисунок 16 – Тестирование при 10 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 100 элементах, когда ничего не нужно удалять, T(100)=4\*100+3=403

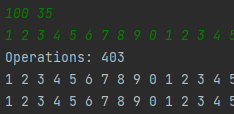


Рисунок 17 – Тестирование при 100 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 10 элементах, когда все нужно удалять, T(10)=2\*10+3= 23

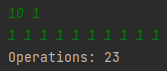


Рисунок 18 – Тестирование при 10 элементах. Все нужно удалить

Теоретическая сложность вычисления при 100 элементах, когда все нужно удалять, T(10)= 2\*100+3 = 203

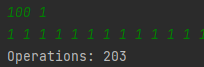


Рисунок 19 – Тестирование при 100 элементах. Все нужно удалить

Как мы видим, реальные результаты близки к теоретическим, линейная зависимость прослеживается, значит теоретический расчет верный.

## **1.3 Выводы по заданию 1**

Основываясь на полученных результатах можно сделать вывод, что второй алгоритм ( с линейной зависимостью сложности, формула худшего случая ) более эффективен, чем первый ( с квадратичной зависимостью сложности, формула худшего случая .) и в среднем и худшем случае требует намного меньше действий для выполнения.

Поскольку результаты теоретического расчета сложности практически совпадают с экспериментально полученными, можно заявить, что расчеты выполнены верно.

# 3 ЗАДАНИЕ 2

**Формулировка задания (Вариант 10)**: Выполнить транспонирование матрицы

## **2.1 Описание математической модели**

Создадим матрицу A размера n x m. Затем создаётся новая матрица В, которая будет являться транспонированной матрицей, размером m и n. Затем с помощью циклов создаются элементы матрицы. Затем с помощью циклов для каждого элемента в матрице A с индексами i и j присваиваем элементы матрице B с индексами j и i.

## **2.2 Блок-схема алгоритма, доказательство корректности циклов**

Рисунок 20 – Блок-схема алгоритма

Определим инвариант для внешнего цикла: i всегда не больше n

Определим инвариант для внутреннего цикла: j всегда не больше n

Доказательство конечности цикла: при каждой итерации по переменной i область неопределённости сужается на 1 строку. До начала цикла не просмотрено n элементов, после первой итерации n-1, после второй n-2 и так далее. После n-ной итерации будет не просмотрено n-n=0 срок, следовательно цикл завершится. Аналогично с циклом j, идущим по элементам столбцов.

Таким образом, все циклы алгоритма корректны, а значит и сам алгоритм, корректен.

## **2.3 Определение вычислительной сложности алгоритма**

|  |  |  |
| --- | --- | --- |
| Номер строки | Алгоритм, записанный на псевдокоде | Количество повторений действия в  зависимости от объема входных  данных n |
| 1 | find\_max\_diagonal\_sum(n, m, matrix, max\_diagonal, result){ |  |
| 2 | diagonal\_sum ← int[n + m - 1] | 1 |
| 3 | for i←0 to n do | n + 1 |
| 4 | for j←0 to m do | m + 1 |
| 5 | diagonal\_sum[i + j] ← diagonal\_sum[i + j] + matrix [i][j] | 1 |
| 6 | od |  |
| 7 | od |  |
| 8 | max\_diagonal ← diagonal\_sum[0] | 1 |
| 9 | result ← 0 | 1 |
| 10 | for i←0 to n do | n |
| 12 | if max\_diagonal > diagonal\_sum[i] then | 1 |
| 13 | result ← i | 1 |
| 14 | max\_diagonal ← diagonal\_sum[i] | 1 |
| 15 | endif |  |
| 16 | od |  |
| 17 | } |  |

Общая вычислительная сложность алгоритма в худшем случае определяется функцией. То есть алгоритм имеет квадратичный порядок роста времени вычисления относительно размерности матрицы. Но так как количество элементов в матрице N=n\*m, то относительно количества элементов , то есть линейно зависит.

В лучшем случае сложность станет: , так как не придётся присваивать новое значение переменным result и max\_diagonal

## **2.4 Реализация алгоритма на языке С++**

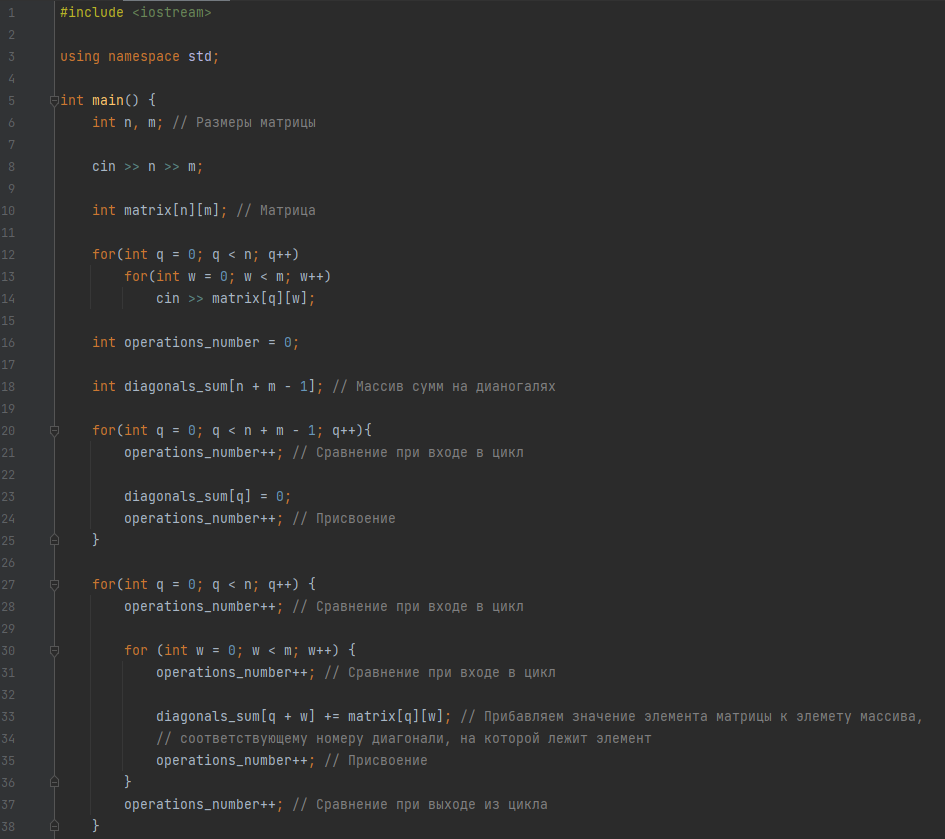


Рисунок 21- Программа на C++

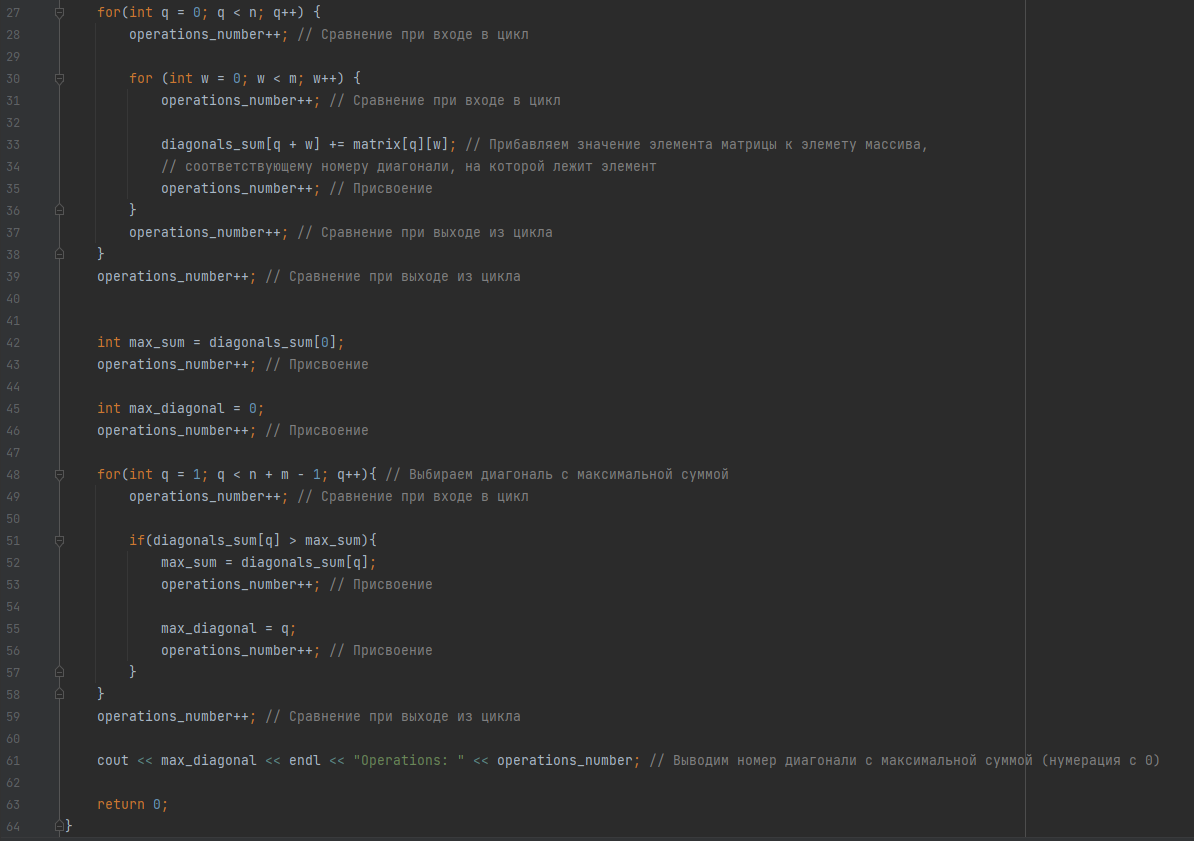


Рисунок 22- Программа на C++

## **2.5 Тестирование**

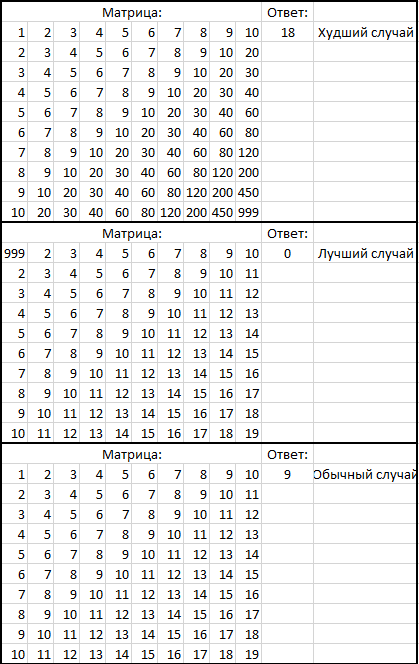


Рисунок 23 – Тестовые данные

Для худшего случая количество операций должно быть 2 \* 10 \* 10 + 10 \* 10 – 4 = 296

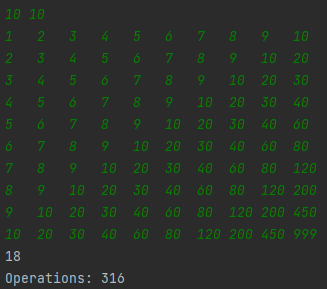


Рисунок 24 – Тест 1 (худший)

Для худшего случая количество операций должно быть 2 \* 10 \* 10 + 8 \* 10 – 4 = 276

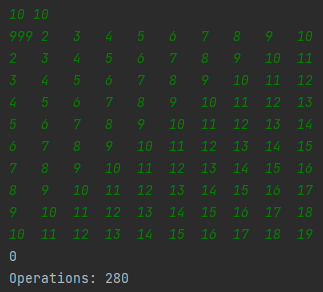


Рисунок 25 – Тест 2 (лучший)

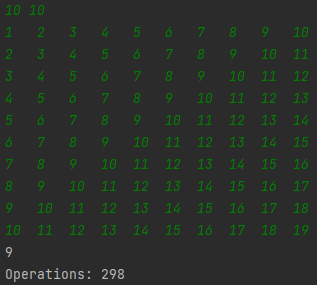


Рисунок 26 – Тест 3 (среднестатистический)

## **2.6 Выводы по заданию 2**

В ходе работы был разработан алгоритм в соответствии с индивидуальным вариантом, оценена его сложность теоретическим и практическим методами. Основываясь на полученных данных, можно утверждать, что алгоритм поиска восходящей диагонали с максимальной суммой в матрице имеет линейную зависимость от количества элементов в матрице.

# 4 ВЫВОДЫ

В ходе работы отработаны навыки определению:

* сложности алгоритмов на теоретическом и практическом уровнях;
* эффективного алгоритма решения задачи из нескольких.

Разработан собственный алгоритм решения задачи и оценена его эффективность. Тестирование подтвердило правильность решения задачи алгоритмом, а также правильность теоретического расчета вычислительной сложности алгоритмов.