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: Зернов Н.А.

Группа: ИКБО-74-23

Москва – 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ

Приобретение практических навыков:

- Эмпирическому определению вычислительной сложности алгоритмов на теоретическом и практическом уровнях;
- Выбору эффективного алгоритма решения вычислительной задачи из нескольких.

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 x 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

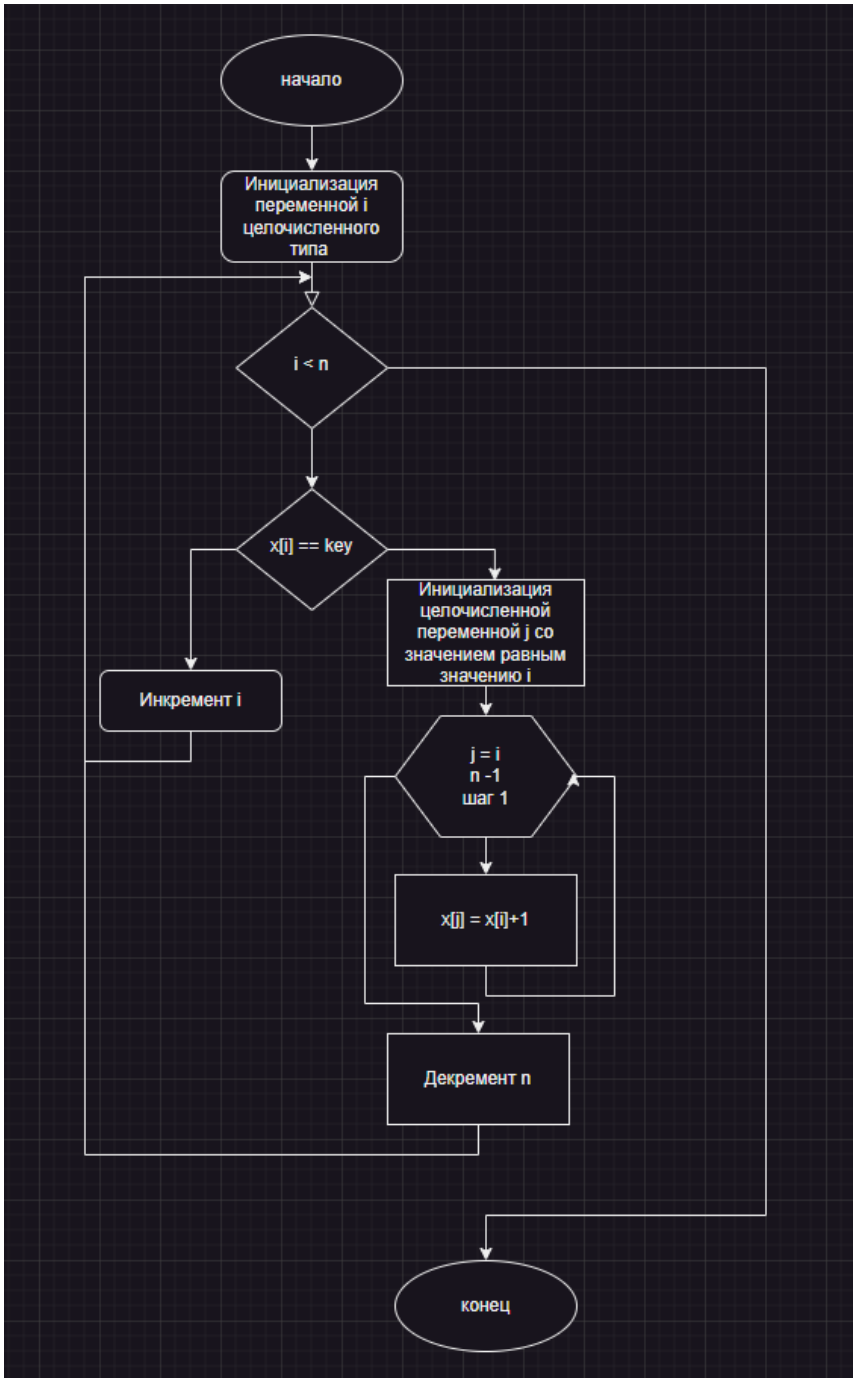

Рисунок 1 – Блок-схема первого алгоритма

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 выполнений оператора в худшем случае	Количество выполнений оператора в лучшем случае
1	delFirstMetod(x,n,key){		
2	i←1	1	1
3	while (i<=n) **do**	n+1	n
4	if x[i]=key then	n	
5	for j←i to n-1 do	$\sum(n-1)\,(n-2)\ldots 2\,1$	
6	x[j] ←x[j+1]	$\sum(n-1)\,(n-2)\ldots 2\,1$	
7	od		
8	n←n-1	n	
9	else		
10	i←i+1		n
11	endif		
12	**od**		
13	}		

Количество повторений действия в строке 6 представляет собой арифметическую прогрессию. Найдем ее сумму $\sum(n-1)\,(n-2)\ldots 2\,1 = \frac{n-1+1}{2} * (n-1) = 0.5n^2 - 0.5n.$

Тогда общая вычислительная сложность алгоритма в худшем случае определяется функцией $\mathbf{T(n) = n^2 + n + 2.}$ 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+1.**

1.1.4 Реализация алгоритма на языке C++

```cpp
#include <iostream>
using namespace std;


void delFirstMethod(int* x, int& n, int key) {
    int operation_counter = 0; // Счетчик кол-ва операций
    int i = 0;
    ++operation_counter; // Присвоение i = 0
    while (i < n) { // Итерирование массива x
        ++operation_counter; // i < n
        if (x[i] == key) { // Если элемент массива i-ый равен значению key, то удаляем элемент
            for (int j = i; j < n - 1; ++j) { // Сдвиг элементов x с i-ого влево
                ++operation_counter; // j < n - 1
                x[j] = x[j + 1];
                ++operation_counter; // Присвоение x[j] = x[j + 1]
            }
            --n;
            ++operation_counter; // Дискримент
        }
        else {
            ++i;
            ++operation_counter; // Икримент
        }
    }
    cout << "Количество операций: " << operation_counter << endl;
}
```

Рисунок 2 – Первый алгоритм удаления

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    int n, key;
    cin >> n >> key;
    int* array = new int[n]; // Массив
    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a':
    {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(1, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array[i] = dist(gen); // Заполнение массива случайными значениями от 1 до 10
        for (int i = 0; i < n; ++i) // Вывод сгенерированного массива
            cout << array[i] << " ";
        cout << endl;
    }
    break;
    case 'b':
    {
        cout << "Введите значение" << endl;
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array[i] = x; // Заполнение массива значением x
        for (int i = 0; i < n; ++i) // Вывод сгенерированного массива
            cout << array[i] << " ";
        cout << endl;
    }
    break;
    default:
    {
        cout << "Допущенны ошибки" << endl;
        return -1; // Завершение программы при некоректных данных
    }
    }
    delFirstMethod(array, n, key);

    for (int i = 0; i < n; ++i)
        cout << array[i] << " ";
    delete[] array;
    return 0;
}
```

Рисунок 3 – Функция main, обеспечивающая работу программы

1.1.5 Тестирование

```
10 1
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
7 5 9 9 9 1 8 1 8 2
Количество операций: 33
7 5 9 9 9 8 8 2
```

Рисунок 5 – Тестирование при 10 элементах. Случайная ситуация.

```
100 1
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
1 8 4 6 4 10 10 2 5 8 5 9 10 8 2 5 8 7 3 5 6 8 4 4 9 1 6 9 9 9 8 6 6 5 1 10 10 3 6 6 2 2 7 5 6 7 1 5 3 6 1 5 1 5 8 1 7 5
 1 2 3 10 6 10 6 2 8 1 2 6 7 9 10 5 3 1 8 6 8 8 9 9 2 3 7 9 2 3 5 10 4 8 3 10 7 7 5 7 4 10
Количество операций: 1257
8 4 6 4 10 10 2 5 8 5 9 10 8 2 5 8 7 3 5 6 8 4 4 9 6 9 9 9 8 6 6 5 10 10 3 6 6 2 2 7 5 6 7 5 3 6 5 5 8 7 5 2 3 10 6 10 6
 2 8 2 6 7 9 10 5 3 8 6 8 8 9 9 2 3 7 9 2 3 5 10 4 8 3 10 7 7 5 7 4 10
```

Рисунок 6 – Тестирование при 100 элементах. Случайная ситуация.

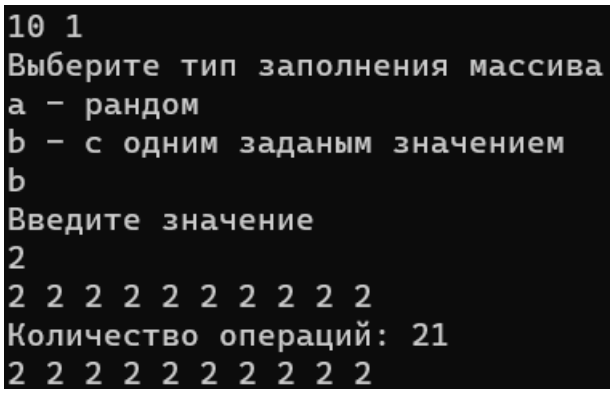

Рисунок 7 – Тестирование при 10 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 10 элементах, когда ничего не нужно удалять, T(10)=2*10+1=21

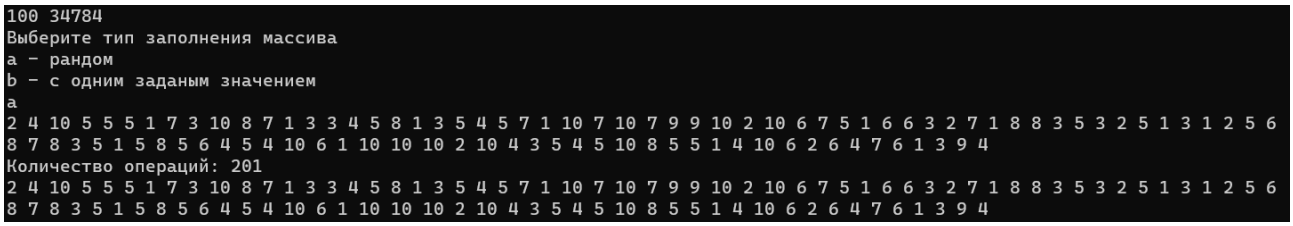

Рисунок 8 – Тестирование при 100 элементах. Ни одного элемента не нужно удалять.

Теоретическая сложность вычисления при 100 элементах, когда ничего не нужно удалять, T(100)=2*100+1=201

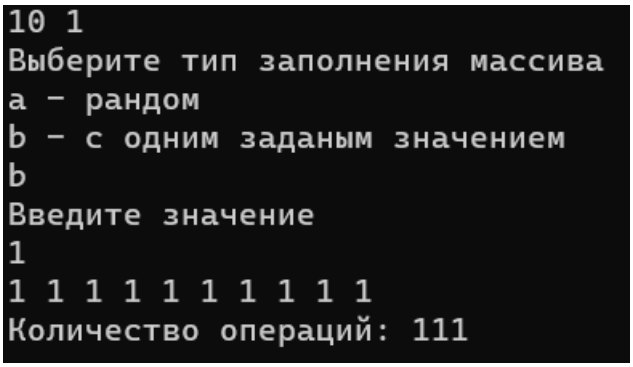

Рисунок 9 – Тестирование при 10 элементах. Все нужно удалить.

Теоретическая сложность вычисления при 10 элементах, когда все нужно удалять, T(10)=10*10+10+1= 111

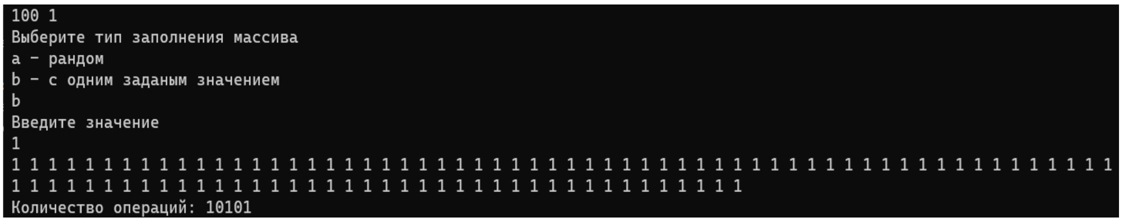

Рисунок 10 – Тестирование при 100 элементах. Все нужно удалить.

Теоретическая сложность вычисления при 100 элементах, когда все нужно удалять, T(10)= 100*100+100+1= 10101

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

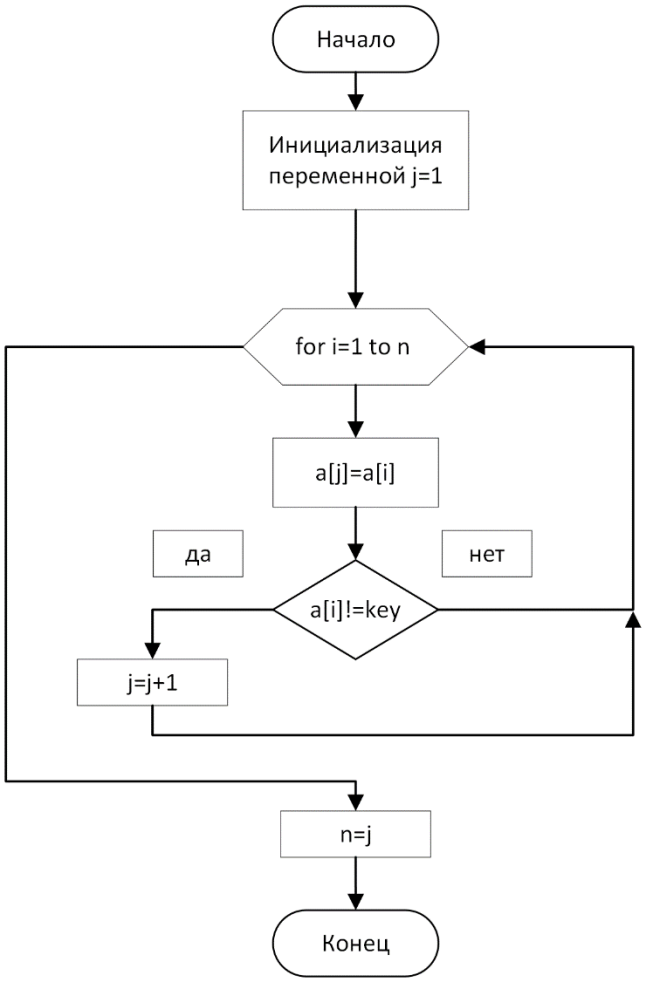

Рисунок 11 – Блок-схема второго алгоритма

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 выполнений оператора в худшем случае	Количество выполнений оператора в лучшем случае
1	delOtherMetod(x,n,key){		
2	j←1	1	1
3	for i←1 to n **do**	n+1	n+1
4	x[j]=x[i];	n	n
5	if x[i]!=key then	n	n
6	j++	n	
7	endif		
8	**od**		
9	**n**←j	1	1
10	}		

Общая вычислительная сложность алгоритма в худшем случае определяется функцией $\mathbf{T(n)} = \boldsymbol{4n} + \mathbf{3}$. То есть алгоритм имеет линейный порядок роста времени вычисления.

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

1.2.4 Реализация алгоритма на языке C++

```cpp
void delSecondMethod(int* array, int& n, int key) {
    int operation_counter = 0;
    int j = 0;
    operation_counter++; // Присвоение
    for (int i = 0; i < n; ++i) {
        operation_counter++; // i < n
        array[j] = array[i]; // Сдвиг элемента на j-ый
        operation_counter++; //Присвоение
        if (array[i] != key) { // Если не нужно удалять
            ++j;
            operation_counter++; // Инкремент
        }
        operation_counter++; // Присвоение
    }
    operation_counter++; // Сравнение при выходе из цикла
    n = j;
    operation_counter++; // Присвоение
    cout << "Количество операций " << operation_counter << endl;
}
```

Рисунок 12 – Второй алгоритм удаления

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    int n, key;
    cin >> n >> key;
    int* array = new int[n]; // Массив
    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a':
    {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(1, 10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            array[i] = dist(gen); // Заполнение массива случайными значениями от 1 до 10
        for (int i = 0; i < n; ++i) // Вывод сгенерированного массива
            cout << array[i] << " ";
        cout << endl;
    }
    break;
    case 'b':
    {
        cout << "Введите значение" << endl;
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            array[i] = x; // Заполнение массива значением x
        for (int i = 0; i < n; ++i) // Вывод сгенерированного массива
            cout << array[i] << " ";
        cout << endl;
    }
    break;
    default:
    {
        cout << "Допущенны ошибки" << endl;
        return -1; // Завершение программы при некоректных данных
    }
    }
    delSecondMethod(array, n, key);

    for (int i = 0; i < n; ++i)
        cout << array[i] << " ";
    delete[] array;
    return 0;
}
```

Рисунок 13 – Функция main

Остальные функции (заполнение и вывод) остались без изменения.

1.2.5 Тестирование

```
10 1
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
10 8 7 4 6 6 1 8 4 3
Количество операций 42
10 8 7 4 6 6 8 4 3
```

Рисунок 14 – Тестирование при 10 элементах. Случайная ситуация.

```
100 1
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
7 1 2 8 7 8 1 1 5 9 7 8 6 4 3 8 6 8 5 10 5 1 4 9 6 2 6 8 9 1 5 7 4 1 10 4 5 1 1 6 6 3 4 7 2 2 2 8 10 6 9 1 3 10 9 2 9 1
4 4 9 5 2 1 6 9 5 6 10 1 2 5 2 1 5 10 8 2 7 4 1 9 2 1 3 7 3 10 1 6 7 8 8 9 9 5 9 10 7 10
Количество операций 387
7 2 8 7 8 5 9 7 8 6 4 3 8 6 8 5 10 5 4 9 6 2 6 8 9 5 7 4 10 4 5 6 6 3 4 7 2 2 2 8 10 6 9 3 10 9 2 9 4 4 9 5 2 6 9 5 6 10
 2 5 2 5 10 8 2 7 4 9 2 3 7 3 10 6 7 8 8 9 9 5 9 10 7 10
```

Рисунок 15 – Тестирование при 100 элементах. Случайная ситуация.

```
10 3244324
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
4 1 8 10 5 1 4 5 4 7
Количество операций 43
4 1 8 10 5 1 4 5 4 7
```

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

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

```
100 2143254325
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
a
9 7 5 4 2 5 6 7 7 5 5 2 3 7 7 7 3 8 10 5 7 6 4 9 1 5 8 3 5 8 3 4 1 2 1 6 1 9 10 9 8 1 7 5 6 10 1 3 3 7 7 5 5 7 9 7 7 3 2
 4 8 5 3 10 8 4 10 8 6 9 9 3 10 5 2 10 8 7 5 4 7 2 5 8 7 9 1 5 2 10 1 2 4 8 1 9 10 6 7 10
Количество операций 403
9 7 5 4 2 5 6 7 7 5 5 2 3 7 7 7 3 8 10 5 7 6 4 9 1 5 8 3 5 8 3 4 1 2 1 6 1 9 10 9 8 1 7 5 6 10 1 3 3 7 7 5 5 7 9 7 7 3 2
 4 8 5 3 10 8 4 10 8 6 9 9 3 10 5 2 10 8 7 5 4 7 2 5 8 7 9 1 5 2 10 1 2 4 8 1 9 10 6 7 10
```

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

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

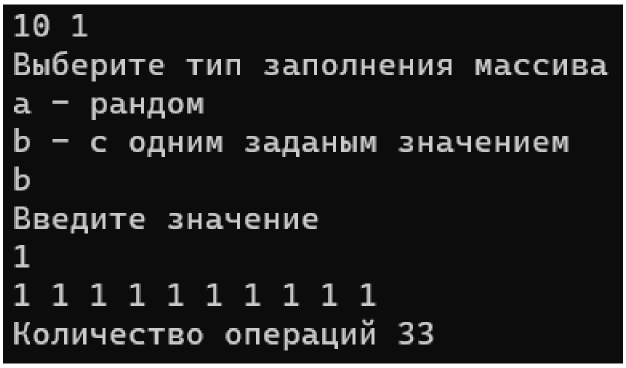

Рисунок 18 – Тестирование при 10 элементах. Все нужно удалить

Теоретическая сложность вычисления при 10 элементах, когда все нужно удалять, T(10)=3*10+3= 33

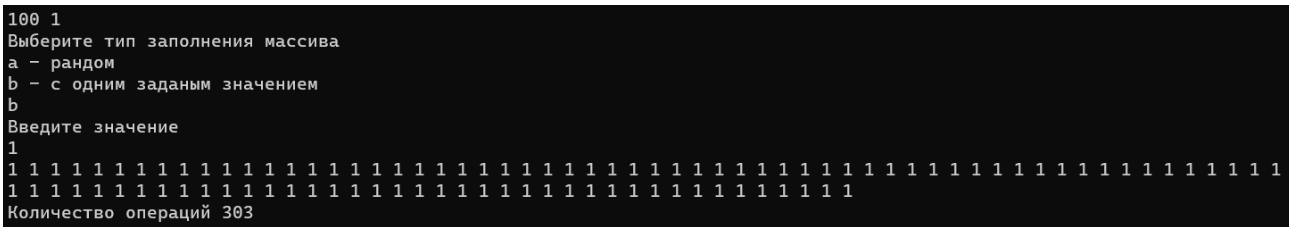

Рисунок 19 – Тестирование при 100 элементах. Все нужно удалить

Теоретическая сложность вычисления при 100 элементах, когда все нужно удалять, T(10)= 3*100+3 = 303

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 $\mathrm{T(n)} = 4n + 3$) более эффективен, чем первый (с квадратичной зависимостью сложности, формула худшего случая $\mathrm{T(n)} = \mathrm{n}^2 + n + 2$.) 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 13): Дана прямоугольная матрица размером n*m. Определить максимальное из чисел, встретившихся в матрице более одного раза.

2.1 Описание математической модели:

Функция принимает на вход матрицу n на m. Функция находит максимальное значение, которое встречается более одного раза в данной матрице. Она создает динамический массив countArray размером 1000000 элементов, который используется для подсчета количества встречающихся чисел в матрице. Этот массив инициализируется нулями, что автоматически обнуляет выделенную память. Затем функция проходит по всем элементам матрицы и увеличивает соответствующий счетчик в массиве countArray. Если какое-то число встречается более одного раза и оно больше текущего значения max_duplicate, то max_duplicate обновляется. После обработки всей матрицы, память, выделенная для countArray, и возвращается найденное максимальное повторяющееся число, а если не найдено, то возвращается -1.

2.2 Блок-схема алгоритма, доказательство корректности циклов

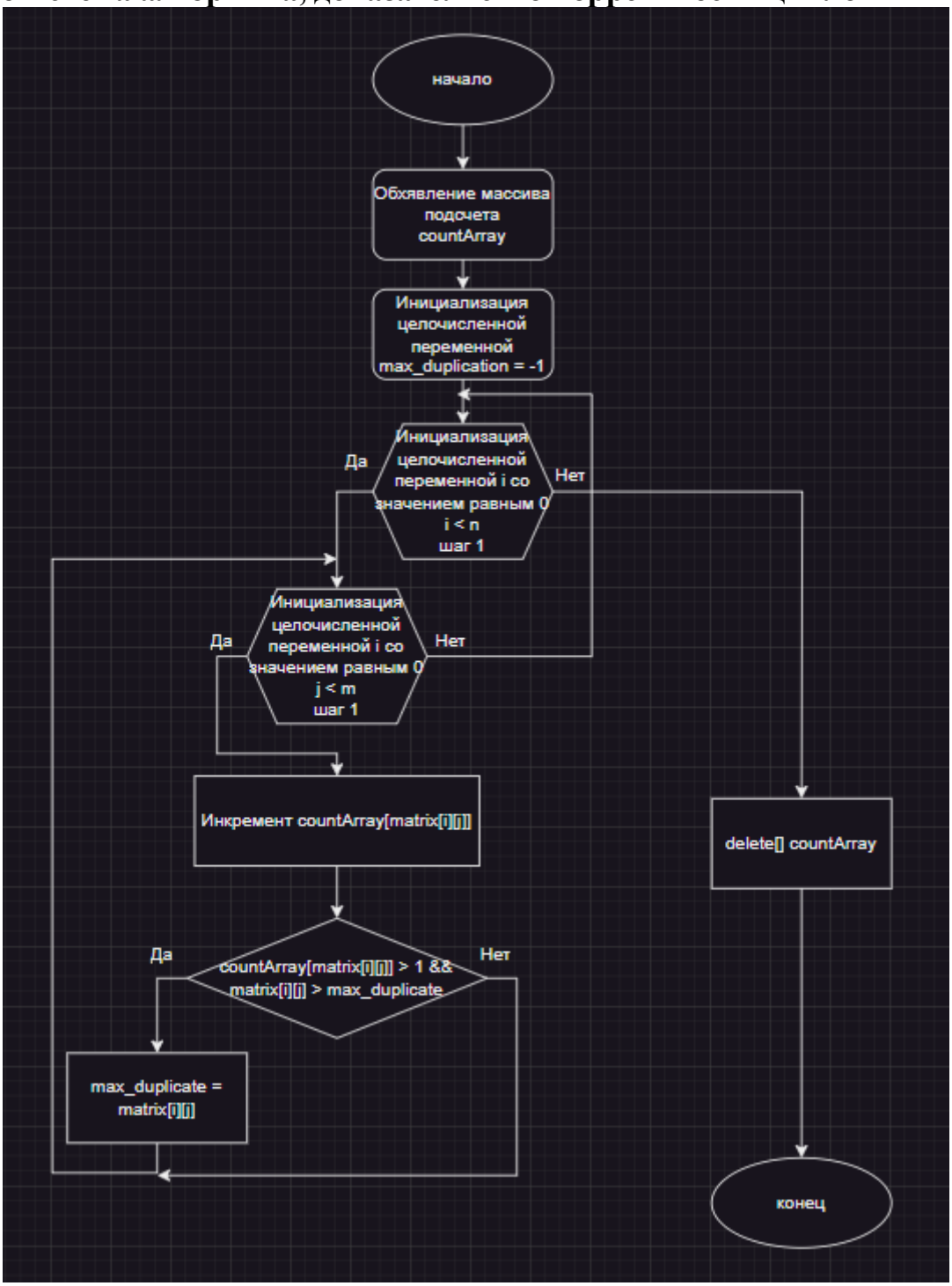

рисунок 20 – Блок-схема алгоритма

Инвариант цикла - это утверждение, которое остается истинным на каждой итерации цикла. В данной функции findmax_duplicate, можно определить следующий инвариант цикла: на каждой итерации внешнего цикла for по строкам (переменная i) и внутреннего цикла for по столбцам (переменная j), массив countArray содержит корректные счетчики для чисел, встречающихся в соответствующих ячейках матрицы matrix. Доказательство корректности

цикла: Инициализация: Перед началом цикла массив countArray инициализируется нулями. Это гарантирует корректное начальное состояние перед началом подсчета чисел. Сохранение: На каждой итерации цикла мы увеличиваем соответствующий элемент массива countArray для числа, находящегося в текущей ячейке матрицы matrix[i][j]. Это не изменяет инвариант, так как мы корректно обновляем счетчики для чисел. Завершение: Цикл завершается после того, как все элементы матрицы пройдены. На этом этапе мы завершаем подсчет, и countArray содержит корректные счетчики для всех чисел в матрице.

Таким образом, инвариант цикла поддерживается на каждой итерации, что обеспечивает корректное выполнение алгоритма. Конечность цикла следует из того, что количество итераций определяется размерами матрицы n и m, которые являются конечными.

2.3 Определение вычислительной сложности алгоритма

Номер строки	Операторы	Количество повторений действия в зависимости от объема входных данных n
1	int findmax_duplicate(int** matrix, int n, int m) {	
2	int* countArray = new int[1000000]();	1
3	int max_duplicate = -1;	1
4	for (int i = 0; i < n; ++i) {	n
5	for (int j = 0; j < m; ++j) {	m
6	countArray[matrix[i][j]]++;	n*m
7	if (countArray[matrix[i][j]] > 1 && matrix[i][j] > max_duplicate) {	n*m
8	max_duplicate = matrix[i][j];	1
9	}	
10	}	
12	}	
13	delete[] countArray;	1
14	return max_duplicate;	1

Общая вычислительная сложность алгоритма в худшем случае определяется функцией $\mathbf{T(n)} = \boldsymbol{n} * \boldsymbol{m} + +\boldsymbol{m} + \boldsymbol{n} + \mathbf{5}$. То есть алгоритм имеет кубический порядок роста времени вычисления относительно размерности матрицы. В лучшем случае сложность станет: $\mathbf{T}(\boldsymbol{n}) = \boldsymbol{n} * \boldsymbol{m}$

При худшем вычислительная сложность равна кубическому порядку роста времени вычисления. Для лучшего, так как матрица уже имеет нужный порядок по диагонали, то вычислительная сложность равна квадратному порядку роста времени вычисления.

2.4 Реализация алгоритма на языке C++

```cpp
#include <iostream>
#include <random>
using namespace std;

int findmax_duplicate(int** matrix, int n, int m) {
    int operation_counter = 0;
    int* countArray = new int[1000000](); // Используем массив для подсчета количества чисел, встречающихся в матрице
    operation_counter++; // Присвоение
    int max_duplicate = -1;
    operation_counter++; // Присвоение
    // Подсчитываем количество встречающихся чисел
    for (int i = 0; i < n; ++i) {
        operation_counter++; //  Сравнение при входе в цикл
        for (int j = 0; j < m; ++j) {
            operation_counter++; //  Сравнение при входе в цикл
            countArray[matrix[i][j]]++;
            operation_counter++; // Инкремент
            if (countArray[matrix[i][j]] > 1 && matrix[i][j] > max_duplicate) {
                max_duplicate = matrix[i][j];
                operation_counter++; // Присвоение
            }
        }
    }

    delete[] countArray; // Освобождаем выделенную память
    cout << "Количество операций " << operation_counter << endl;
    return max_duplicate;
}
```

Рисунок 21- Программа на C++

```cpp
int main() {
    setlocale(LC_ALL, "Rus");
    int n, m;
    cout << "Введите размеры матрицы (n m): ";
    cin >> n >> m;

    int** matrix = new int* [n]; // Выделяем память для двумерного динамического массива
    for (int i = 0; i < n; ++i)
        matrix[i] = new int[m];

    cout << "Выберите тип заполнения массива" << endl;
    cout << "a - рандом" << endl;
    cout << "b - с одним заданым значением" << endl;
    cout << "c - заполняем матрицу элементами, где следующий больше предыдущего на 1" << endl;
    cout << "d - ручной ввод матрицы" << endl;
    char ty;
    cin >> ty;
    switch (ty) { //Конструкция switch case default для выбор типа заполения массива
    case 'a': {
        mt19937 gen(random_device{}()); // mt19937 gen - генератор случайных чисел
        uniform_int_distribution<int> dist(1,10); // Инструмент, позволяющий генерировать случайное целое число в заданном диапазоне

        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; ++j)
                matrix[i][j] = dist(gen); // Заполнение матрицы случайными значениями от 0 до 10
        for (int i = 0; i < n; i++) { // Вывод сгенерированной матрицы
            for (int j = 0; j < m; ++j)
                cout << matrix[i][j] << " ";
            cout << endl;
        }
    }
            break;
    case 'b': {
        cout << "Введите значение" << endl;
        int x;
        cin >> x;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; ++j)
                matrix[i][j] = x;
        for (int i = 0; i < n; i++) { // Вывод сгенерированной матрицы
            for (int j = 0; j < m; ++j)
```

Рисунок 22- Программа на C++

```
71                  cout << matrix[i][j] << " ";
72              cout << endl;
73              }
74          }
75              break;
76      case 'c': {
77          int c = 0; // Заполняем матрицу элементами, где следующий больше предыдущего на 1
78          for (int i = 0; i < n; i++) {
79              for (int j = 0; j < m; ++j) {
80                  matrix[i][j] = c;
81                  c += 1;
82              }
83          }
84          for (int i = 0; i < n; i++) { // Вывод сгенерированной матрицы
85              for (int j = 0; j < m; ++j)
86                  cout << matrix[i][j] << " ";
87              cout << endl;
88          }
89      }
90              break;
91      case 'd': {
92          // Ручной ввдо матрицы
93          for (int i = 0; i < n; i++) {
94              for (int j = 0; j < m; ++j) {
95                  cin >> matrix[i][j];
96              }
97          }
98      }
99              break;
100     default: {
101         cout << "Допущенны ошибки" << endl;
102         return -1; // Завершение программы при некоректных данных
103     }
104     }
105     int max_duplicate = findmax_duplicate(matrix, n, m);
106     if (max_duplicate != -1)
107         cout << "Максимальное число, встретившееся более одного раза: " << max_duplicate << endl;
108     else
109         cout << "В матрице нет чисел, встречающихся более одного раза." << endl;
110
111     // Освобождаем память, выделенную для двумерного динамического массива
112     for (int i = 0; i < n; ++i)
```

Рисунок 23- Программа на C++

```
113         delete[] matrix[i];
114     delete[] matrix;
115     return 0;
116 }
```

Рисунок 24- Программа на C++

2.5 Тестирование

```
Введите размеры матрицы (n m): 10 17
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - заполняем матрицу элементами, где следующий больше предыдущего на 1
a
2 9 6 5 5 2 1 10 4 10 6 9 5 5 1 6 3
5 9 8 2 10 8 2 7 4 7 7 7 10 9 3 4 9
6 1 10 1 7 2 8 5 7 10 6 8 5 5 2 8 4
1 4 10 8 1 1 6 6 5 2 3 5 9 5 9 1 2
6 1 5 7 5 9 5 2 4 3 2 1 5 9 4 9 4
8 8 9 3 7 5 4 7 10 4 9 5 2 9 6 7 6
2 7 4 8 5 3 2 8 2 7 6 9 1 7 4 5 8
6 6 10 8 10 7 9 1 2 9 8 6 5 2 4 4 6
7 4 3 2 10 7 8 1 9 5 10 6 6 3 7 8 2
1 9 3 2 6 10 6 5 1 9 8 6 9 7 10 1 1
Количество операций 354
Максимальное число, встретившееся более одного раза: 10
```

Рисунок 25 – Тестовые данные

На этом тесте мы рассматриваем обычный (случайный) случай.

```
Введите размеры матрицы (n m): 3 4
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - заполняем матрицу элементами, где следующий больше предыдущего на 1
c
0 1 2 3
4 5 6 7
8 9 10 11
Количество операций 29
В матрице нет чисел, встречающихся более одного раза.
```

Рисунок 26 – Тест 2 (лучший случай)

```
Введите размеры матрицы (n m): 3 4
Выберите тип заполнения массива
a - рандом
b - с одним заданым значением
c - заполняем матрицу элементами, где следующий больше предыдущего на 1
b
Введите значение
1
1 1 1 1
1 1 1 1
1 1 1 1
Количество операций 30
Максимальное число, встретившееся более одного раза: 1
```

Рисунок 27 – Тест 3 (худший случай)

2.6 Выводы по заданию 2

Мы научились создавать алгоритм для вычисления треугольной матрицы методом Гаусса, а также исследовали его. При худшем вычислительная сложность равна кубическому порядку роста времени вычисления. Для лучшего, так как матрица уже имеет нужный порядок по диагонали, то вычислительная сложность равна квадратному порядку роста времени вычисления. Эти данные схожи с практическими результатами, а это значит что расчеты выполнены правильно.

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.

5. Литература

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона, 2010.
2. Кнут Д. Искусство программирования. Тома 1-4, 1976-2013.
3. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих, 2017.
4. Кормен Т.Х. и др. Алгоритмы. Построение и анализ, 2013.
5. Лафоре Р. Структуры данных и алгоритмы в Java. 2-е изд., 2013.
6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., 2018.
7. Скиена С. Алгоритмы. Руководство по разработке, 2011.
8. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017.

9. Гасфилд Д. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология, 2003.

По языку C++:

10. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

11. Павловская Т.А. C/C++. Программирование на языке высокого уровня, 2003.

12. Прата С. Язык программирования C++. Лекции и упражнения. - 6-е изд., 2012.

13. Седжвик Р. Фундаментальные алгоритмы на C++, 2001-2002

14. Хортон А. Visual C++ 2010. Полный курс, 2011.

15. Шилдт Г. Полный справочник по C++. 4-е изд., 2006.