МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Отчет по выполнению практического задания №1

Тема: Оценка сложности и определение эффективности алгоритма

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент Антонов А.Д.

группа ИКБО-01-20

Москва 2021

Содержание

Задание 1 4

1. Постановка задачи 4

2. Модель решения поставленной задачи в первом алгоритме 4

2.1 Описание алгоритма 4

2.2 Инвариант цикла 4

2.3 Определение вычислительной сложности алгоритма 5

3. Алгоритм в виде функции 7

4. Функции заполнения массива случайными числами и вывода на экран 7

5. Результаты тестирования 8

6. Результаты тестирования крайних случаев 8

7. Модель решения поставленной задачи в первом алгоритме 10

7.1 Описание алгоритма 10

7.2 Инвариант цикла 10

7.3 Определение вычислительной сложности алгоритма 10

8. Алгоритм в виде функции 11

9. Функции заполнения массива случайными числами и вывода на экран 1210.
Результаты тестирования 12

11. Результаты тестирования крайних случаев 12

ВЫВОД 14

Задание 2. Выполнение индивидуального задания в соответствии с вариантом 15

1. Постановка задачи 15

2. Модель решения 15

3. Разработка эффективного алгоритма 15

3.1 Алгоритм 15

3.2 Инварианты 15

3.3 Определение вычислительной сложности алгоритма 16

3.4. Реализация алгоритма в виде функции 18

3.5. Тестирование алгоритма 18

3.6. Практическая оценка сложности алгоритма 19

ВЫВОДЫ 21

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 21

Задание 1

1. Постановка задачи

Определить эффективный алгоритм из двух предложенных, используя оценку теоретической сложности каждого из алгоритмов и емкостную сложность, решения следующей задачи: дан массив из n элементов целого типа, удалить из массива все значения равные заданному.

Таблица 1: предложенные алгоритмы на псевдокоде

<table>
<tr><td colspan="2">x-массив, n – количество элементов в массиве, key – удаляемое значение</td></tr>
<tr><td>Алгоритм 1.
delFirstMetod(x,n,key){
i←1
while (i<=n) do
    if x[i]=key then
        //удаление
        for j←i to n-1 do
          x[j] ←x[j+1]
        od
        n←n-1
    else
        i←i+1
    endif
od
}</td><td>Алгоритм 2
delOtherMetod(x,n,key){
j←1
for i←1 to n do
  x[j]=x[i];
  if x[i]!=key then
    j++
  endif
od
n←j
}</td></tr>
</table>

2. Модель решения поставленной задачи в первом алгоритме

2.1 Описание алгоритма:

С помощью цикла while находим элементы массива равные key, с помощью вложенного цикла от i до n-1 смещаем все элементы влево и уменьшаем значение n на единицу, если элемент массива не равен key, то просто переходим к следующему, увеличивая i.

2.2 Инвариант цикла:

1. i находится в промежутке от [0, n-1],
2. j находится в промежутке от [i, n-2]

2.3 Определение вычислительной сложности алгоритма

Таблица 2. Подсчет количества операторов алгоритма

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
Номер столбца	1	2	3
1	int i = 0;	C1	1 раз
2	while (i < n){	C2	n+1 раз
3	if (x[i] == key){	C3	n раз
4	for (int j = i; j < n - 1; j++){	C4	n*(n+1) раз
5	x[j] = x[j + 1];	C5	n*(n+1)-1 раз
6	n = n - 1;}	C6	n раз
7	else{i++;}}	C7	n-n_{C3} раз

Оператор 1. Выполняется один раз.

Оператор 2. В худшем случае – в массиве нет заданного значения, а значит n не уменьшается – выполнится n+1 раз, n раз с заходом внутрь цикла и 1 раз с проверкой, когда I = n.

Оператор 3 (if). Это оператор тела цикла, т.е. он выполняется n раз, сколько количество входов в тело цикла.

Оператор 4. Вложенный цикл. В худшем случае выполнится n*(n+1) раз

Оператор 5. Выполнится соответственно вложенному циклу, без одной операции: n*(n+1)-1 раз

Оператор 7 (else). Выполнится $(n-n_{C3})$.

Определим функцию роста для времени выполнения алгоритма в худшем случае:
T(n)= C1 + C2*(n + 1) + C3*n + C4*n*(n + 1) + C5*(n*(n - 1) - 1)+ C6*n+C7*n =(C4+C5)*n^2 +(C2+C3+C4+C5+C6+C7)*n + (C1+C2+*C4-*C5) = An^2 + Bn+C

Пренебрегаем константой C. Получаем T(n) = An^2+Bn. Функция n^2 имеет порядок роста выше, чем функция n. T(n)= An2+Bn, доминирующей функцией является n^2, и она определяет порядок роста для алгоритма в худшем случае. Т.е. $T(n)=\Theta(n^2)$.

Выведем функцию роста для времени выполнения алгоритма в лучшем случае (введенного значения в массиве нет):

$T(n) = C1*1+C2(n+1)+C3*n+C7*n=An+B$

Порядок роста времени в зависимости от n в наилучшем случае линейный, т.е. $T(n)=\Theta(n)$.

3. Алгоритм в виде функции:

```
#include <iostream>

int i, j;

void delFirstMethod (int *x, int n, int key)
{
    int comp = 0, del = 0;
    i = 0;

    while (i < n) {
        comp++;
        if (x[i] == key)
        {
            for (j = i; j < n - 1; j++)
            {
                comp++;
                x[j] = x[j + 1];
            }
            n--; del++;
        }
        else i++;
    }

    for (i = 0; i < n; i++)
        std::cout << x[i] << ' ';
    delete[] x;
    std::cout << std::endl << std::endl;
    std::cout << "Comparisons: " << comp << " Deletions: " << del << std::endl;
}
```

4. Функции заполнения массива случайными числами и вывода на экран

```cpp
void random (int *x, int n) {
    for (i = 0; i < n; i++)
        x[i] = rand() % 10;
}

void print (int *x, int n) {
    std::cout << "Array:" << std::endl;
    for (i = 0; i < n; i++)
        std::cout << x[i] << ' ';
    std::cout << std::endl;
}
```

5. Результаты тестирования

Для n = 10:

```
Enter array length:
10
Array:
1 7 4 0 9 4 8 8 2 4
Enter key:
8

New array:
1 7 4 0 9 4 2 4

Comparisons: 15 Deletions: 2
```

Для n = 100:

```
Enter array length:
100
Array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9
2 7 9 5 4 3 1 2 3 3 4 1 1 3 8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8
6 5 0 9 0 0 6 1 3 8 9 3 4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2 9 1
Enter key:
7

New array:
1 4 0 9 4 8 8 2 4 5 5 1 1 1 5 2 6 1 4 2 3 2 2 1 6 8 5 6 1 8 9 2 9 5 4
3 1 2 3 3 4 1 1 3 8 4 2 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1
3 8 9 3 4 4 6 0 6 6 1 8 4 9 6 3 8 8 2 9 1

Comparisons: 642 Deletions: 9
```

Как видно, при увеличении n на 1 порядок, кол-во удалений возрастает на два порядка, что подтверждает полученный порядок роста $T(n)=O(n^2)$

6. Результаты тестирования крайних случаев:

Лучший случай (нет чисел равных заданному)

Для n = 10:

```
Enter array length:
10
Array:
1 7 4 0 9 4 8 8 2 4
Enter key:
-1

New array:
1 7 4 0 9 4 8 8 2 4

Comparisons: 10 Deletions: 0
```

Для n =100:

```
Enter array length:
100
Array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9 2 7 9 5 4 3
1 2 3 3 4 1 1 3 8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1 3 8 9 3
4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2 9 1
Enter key:
-1

New array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9 2 7 9 5 4 3
1 2 3 3 4 1 1 3 8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1 3 8 9 3
4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2 9 1

Comparisons: 100 Deletions: 0
```

Как видно, в лучшем случае кол-во операций линейно зависит от размера массива, что подтверждает полученный порядок роста для лучшего случая $T(n) = \Theta(n)$.

Худший случай (все числа равных заданному)

Для n = 10:

```
Enter array length:
10
Enter key:
5
Array:
5 5 5 5 5 5 5 5 5 5

New array:


Comparisons: 55 Deletions: 10
```

Для n=100:

```
Enter array length:
100
Enter key:
5
Array:
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5

New array:


Comparisons: 5050 Deletions: 100
```

Как видно, в худшем случае кол-во операций квадратично зависит от размера массива, что подтверждает полученный порядок роста для худшего случая $T(n) = O(n^2)$.

7. Модель решения поставленной задачи во втором алгоритме

7.1 Описание алгоритма

С помощью цикла for проходимся по всем элементам массива, при входе в цикл мы записываем выбранный элемент в самого себя же, если до этого нам не встретились числа равные заданному, в противном случае мы перезаписываем выбранный элемент внутрь элемента равного заданному, т.е. производим удаление.

7.2 Инвариант цикла

1. i находится в промежутке от [0, n-1],

7.3 Определение вычислительной сложности алгоритма

Таблица 3. Подсчет количества операторов алгоритма 2

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
Номер столбца	1	2	3
1	int j = 1;	C1	1 раз
2	for (int i = 0; i < n; i++){	C2	n+1 раз
3	x[j] = x[i];	C3	n раз

4	if (x[i] != key){	C4	n раз
5	j++;}}	C5	n раз
6	n = j;}	C6	n раз

Оператор 1 выполняется один раз.

Оператор 2. Согласно циклу с предусловием: первый вход в цикл при i=0; последний вход в цикл при i=n-1; после последнего входа i=n, т.е. ещё одна проверка и завершение цикла. Считаем сколько раз выполнялся оператор i <n: n раз обеспечивался вход в цикл и один раз при выходе из цикла, таким образом, всего n+1 раз за время работы.

Оператор 3. Выполнится n раз.

Оператор 4 (if). Это оператор тела цикла, т.е. он выполняется n раз – количество входов в тело цикла.

Оператор 5. Это оператор – блок оператора if. Выполняется столько раз, сколько и if (в худшем случае) – n раз.

Оператор 6. Выполнится n раз.

Определим время выполнения алгоритма - как сумму времени выполнения каждого оператора:

$T(n)=C1*1+C2*(n+1)+C3*n+C4*n+C5*n+C6*n = (C2+C3+C4+C5+C6)n + (C1 + C2) = An + B$.

В результате T(n) в худшем случае линейно зависит от n.

В лучшем и среднем случае порядок роста все равно будет линейно зависит от n, так как цикл будет все равно выполняться n+1 раз.

Вывод. Порядок роста $T(n)=\Theta(n)$.

8. Алгоритм в виде функции

```cpp
void delOtherMethod (int *x, int n, int key)
{
    int comp = 0, del = 0;
    j = 0;

    for (i = 0; i < n; i++) {
        x[j] = x[i];
        comp += 2;
        if (x[i] != key) j++;
        else del++;
    }
    n = j;

    for (i = 0; i < n; i++)
        std::cout << x[i] << ' ';
    delete[] x;
    std::cout << std::endl << std::endl;
    std::cout << "Comparisons: " << comp << " Deletions: " << del << std::endl;
}
```

9. Функции заполнения массива случайными числами и вывода на экран

```cpp
void random (int *x, int n) {
    for (i = 0; i < n; i++)
        x[i] = rand() % 10;
}

void print (int *x, int n) {
    std::cout << "Array:" << std::endl;
    for (i = 0; i < n; i++)
        std::cout << x[i] << ' ';
    std::cout << std::endl;
}
```

10. Результаты тестирования

Для n = 10:

```
Enter array length:
10
Array:
1 7 4 0 9 4 8 8 2 4
Enter key:
8

New array:
1 7 4 0 9 4 2 4

Comparisons: 20 Deletions: 2
```

Для n = 100:

```
Enter array length:
100
Array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9 2 7 9 5 4 3
1 2 3 3 4 1 1 3 8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1 3 8 9 3
4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2 9 1
Enter key:
6

New array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 1 4 2 3 2 2 1 8 5 7 1 8 9 2 7 9 5 4 3 1 2 3
3 4 1 1 3 8 7 4 2 7 7 9 3 1 9 8 5 0 2 8 0 2 4 8 5 0 9 0 0 1 3 8 9 3 4 4 0 1 8 4 9
3 7 8 8 2 9 1

Comparisons: 200 Deletions: 11
```

Как видно при увеличении n на 1 порядок, количество операций увеличивается также на 1 порядок, что подтверждает порядок роста T(n)=Θ(n).

11. Результаты тестирования крайних случаев

Лучший случай (нет чисел равных заданному)

Для n = 10:

```
Enter array length:
10
Array:
1 7 4 0 9 4 8 8 2 4
Enter key:
-1

New array:
1 7 4 0 9 4 8 8 2 4

Comparisons: 20 Deletions: 0
```

Для n =100:

```
Enter array length:
100
Array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9 2 7 9 5 4 3 1 2 3 3 4 1 1 3
8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1 3 8 9 3 4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2
9 1
Enter key:
-1

New array:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1 8 9 2 7 9 5 4 3 1 2 3 3 4 1 1 3
8 7 4 2 7 7 9 3 1 9 8 6 5 0 2 8 6 0 2 4 8 6 5 0 9 0 0 6 1 3 8 9 3 4 4 6 0 6 6 1 8 4 9 6 3 7 8 8 2
9 1

Comparisons: 200 Deletions: 0
```

Как видно, в лучшем случае кол-во операций линейно зависит от размера массива, что подтверждает полученный порядок роста для лучшего случая T(n) = Θ(n).

Худший случай (все числа равны заданному)

Для n = 10:

```
Enter array length:
10
Enter key:
3
Array:
3 3 3 3 3 3 3 3 3 3

New array:


Comparisons: 20 Deletions: 10
```

Для n =100:

```
Enter array length:
100
Enter key:
4
Array:
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4

New array:


Comparisons: 200 Deletions: 100
```

Как видно, в худшем случае кол-во операций также линейно зависит от размера массива, что подтверждает полученный порядок роста для худшего случая $T(n) = \Theta(n)$.

ВЫВОД

В ходе выполнения практического задания для данных алгоритмов были определены инварианты внешнего и внутренних циклов. Также была определена вычислительная сложность обоих алгоритмов теоретическим подходом. На основании оценки вычислительной сложности было выявлено, что второй алгоритм является более эффективным. Обе версии алгоритма были реализованы на C++ и протестированы. Теоретические оценкам вычислительной сложности совпадают с результатом практических тестировании представленного алгоритма.

Задание 2. Выполнение индивидуального задания в соответствии с вариантом. Вариант 1.

1. Постановка задачи

Умножение квадратных матриц.

2. Модель решения

Считаем значение каждого элемента результирующей матрицы - оно равно произведению строки первой матрицы на столбец второй матрицы. С помощью вложенного цикла проходим по каждой паре строка-столбец, и с помощью еще одного цикла считаем значение элемента.

3. Разработка эффективного алгоритма

3.1 Алгоритм

Проходимся по парам строк первого двумерного массива и столбцов второго двумерного массива, суммируем произведения элементов в них.

3.2 Инварианты

Цикла i: i принадлежит от [0..n-1]
Цикла j: j принадлежит от [0..n-1]
Цикла k: k принадлежит от [0..n-1]

Циклы корректны, т.к. в любой момент времени рассматриваем элементы массивов с индексом от [0][0] до [n-1][n-1], т.е. все элементы массивов.

3.3 Определение вычислительной сложности алгоритма

Таблица 4. Подсчет количества операторов в алгоритме

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
Номер столбца	1	2	3
1	int temp;	C1	1
2	for (i=0; i<n; i++)	C2	n+1
3	for (j=0; j<n; j++)	C3	n*(n+1)
4	temp = 0;	C4	n^2
5	for (k=0; k<n; k++)	C5	$(n+1)*n^2$
6	temp += a[i][k] * b[k][j]	C6	n^3
7	std::cout << temp;	C7	n^2
8	std::cout << std::endl;	C8	n

Оператор 1. Выполняется 1 раз.

Оператор 2. Согласно циклу с предусловием: первый вход в цикл при $i=0$; последний вход в цикл при $i=n-1$; после последнего входа $i=n$, т.е. ещё одна проверка и завершение цикла. Считаем сколько раз выполнялся оператор $i<n$: n раз обеспечивался вход в цикл и один раз при выходе из цикла, таким образом, всего $n+1$ за время работы.

Оператор 3. Согласно циклу с предусловием: первый вход в цикл при $j=0$; последний вход в цикл при $j=n-1$; после последнего входа $j=n$, т.е. ещё одна проверка и завершение цикла. Считаем сколько раз выполнялся оператор $j<n$: n раз обеспечивался вход в цикл и один раз при выходе из цикла, таким образом, всего $n+1$ за время работы, и, так как цикл внутри другого цикла, общее выполнение будет равно $n*(n+1)$

Оператор 4. Выполняется внутри двух циклов, соответственно n^2 раз

Оператор 5. Согласно циклу с предусловием: первый вход в цикл при $k=0$; последний вход в цикл при $k=n-1$; после последнего входа $k=n$, т.е. ещё одна проверка и завершение цикла. Считаем сколько раз выполнялся оператор $k<n$: n раз обеспечивался вход в цикл и один раз при выходе из цикла, таким образом, всего $n+1$ за время работы, и, так как цикл внутри другого цикла, общее выполнение будет равно $(n+1)*n^2$

Оператор 6. Выполняется внутри трех циклов, соответственно n^3 раз

Оператор 7. Выполняется внутри двух циклов, соответственно n^2 раз

Оператор 8. Выполняется внутри цикла, соответственно n раз

Определим время выполнения алгоритма - как сумму времени выполнения каждого оператора:

$$T(n) = C1 + C2*(n+1) + C3*(n^2+n) + C4*n^2 + C5*(n^3+n^2) + C6*n^3 + C7*n^2 + C8*n =$$
$$(C5+C6)*n^3 + (C3+C4+C5+C7)*n^2 + (C2+C3+C8)*n + (C1+C2) = An3 + Bn2 + Cn + D$$

Пренебрегаем константой D. Получаем $T(n) = An^3+Bn^2+Cn$. Функция n^3 имеет порядок роста выше, чем функции n^2 и n. $T(n) = An^3+Bn^2+Cn$, доминирующей функцией является $n^{3,}$ и она определяет порядок роста для алгоритма в худшем случае. Т.е. $T(n)=\Theta(n^3)$.

3.4. Реализация алгоритма в виде функции

```cpp
#include <iostream>
#include <iomanip>

int i, j, k;

void matrixProduct (int** a, int** b, int n)
{
    int temp;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            temp = 0;
            for (k = 0; k < n; k++)
                temp += a[i][k] * b[k][j];
            std::cout << std::setw(4) << temp;
        }
        std::cout << std::endl;
    }
}
```

3.5. Тестирование алгоритма.

Входные значения	Ожидаемый вывод	Вывод
2 1 2 3 4 1 0 0 1	1 2 3 4	Enter size of matrices: 2 Enter elements of 1st martix: 1 2 3 4 Enter elements of 2nd martix: 1 0 0 1 Matrix product: 1 2 3 4
3 1 2 3 4 5 4 3 2 1 5 4 3 2 1 2 3 4 5	18 18 22 42 37 42 22 18 18	Enter size of matrices: 3 Enter elements of 1st martix: 1 2 3 4 5 4 3 2 1 Enter elements of 2nd martix: 5 4 3 2 1 2 3 4 5 Matrix product: 18 18 22 42 37 42 22 18 18

Из тестов видно, что алгоритм работает согласно описанию и модели.

3.6. Практическая оценка сложности алгоритма

Для оценки практической сложности будем считать количество операций внутри каждого цикла. Изменим код следующим образом:

```
void matrixProduct (int** a, int** b, int n)
{
    int temp, iter = 0;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            temp = 0;
            for (k = 0; k < n; k++)
            {
                temp += a[i][k] * b[k][j];
                iter++;
            }
            std::cout << std::setw(4) << temp;
        }
        std::cout << std::endl;
    }
    std::cout << std::endl << "Iterations: " << iter;
}
```

Добавим переменную iter, которая будут считать количество операций, совершенных внутренними циклами для оценки сложности.

Для n = 1 (лучший случай):

```
Enter size of matrices: 1
Enter elements of 1st martix:
1
Enter elements of 2nd martix:
1
Matrix product:
   1

Iterations: 1
```

Для n = 10:

```
Matrix product:
 147 267 194 159 237 256 261 243 179 182
  92 166 179 203 225 148 174 186 196 185
  91 143 140 140 191 162 137 201 181 165
 146 289 246 239 310 232 340 302 266 281
 119 116 150 117 161 118  99 150 105 146
 178 251 270 251 300 275 222 274 226 303
 161 199 220 150 194 207 179 229 155 222
  91 151 184 187 212  96 178 155 181 166
 167 220 234 211 258 149 264 255 177 207
 141 285 225 259 278 349 261 247 223 246

Iterations: 1000
```

Можно заметить, что при увеличении значения n на порядок, количество операций увеличивается на три порядка, что подтверждается тем, что порядок роста для алгоритма $T(n)=\Theta(n^3)$.

Прогон на n = 100 (не вмещается на экран):

```
  2032  2055  1981  1830  1888  2132  1886  1986  1933  1869  18
  2228  1872  2131  1942  2047  2243  1980  1876  1910  2115  20
  1919  2062  2201  2016  2005  1973  1994  1965  2208  2050  22
  2179  2144  1780  2002  2001  2057  1944  1864  1804  2049  17
  2046  1815  2035  1863  2213  2092  1937  2146  1974  1720  19

  1884  1935  1780  1796  1975  2011  1757  1776  1931  1787  18
  2143  1828  2109  1714  1885  1851  2160  1956  1946  1922  20
  1774  2062  1926  2095  2060  1961  1941  1884  1999  2064  19
  1930  1988  1768  1906  1817  1975  1848  1802  1848  1965  16
  2104  1828  1789  1784  2066  2104  1880  2055  2003  2092  19

  2059  2087  2033  1935  2081  2140  2090  2102  1920  1845  21
  2264  2180  2225  1911  2085  2159  2213  2140  1991  2278  20
  2013  2062  2402  2113  2242  2116  2057  2163  2355  2100  21
  2344  2032  2069  2124  2099  2010  1987  2078  2016  2194  19
  2328  1809  1985  2070  2157  2160  1949  2170  2101  1974  18

  2050  2182  1979  1954  1996  2058  1868  2048  2031  2012  20
  2274  1807  2125  1784  2068  2078  2225  2149  1869  2183  20
  1910  2130  2194  2028  2177  2040  2130  2016  2133  2149  22
  2175  1978  2058  2050  2041  2012  1949  2065  1861  2289  18
  2213  1845  2040  2026  2141  2225  1914  2409  2029  1984  19


Iterations: 1000000
```

ВЫВОДЫ

В ходе данной практической работы мы изучили определение сложности алгоритма, научились теоретически определять порядок роста, разработали алгоритм в соответствии с требованием индивидуального задания, определили его порядок роста и на практике подтвердили полученный результат.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Лекционный материал по структуре и алгоритмам обработки данных Гданского Н.И.

2. Теоретический материал по структурам и алгоритмам обработки данных.

3. Кораблин Ю.П., Сыромятников В.П., Скворцова Л.А. Учебно-методическое пособие Структуры и алгоритмы обработки данных, М.:МИРЭА, 2020

4. Методичка с примерами определения функции роста.