МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по практической работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 7383 _____ Кирсанов А.Я.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Постановка задачи.

Цель работы.

Решение задачи квадрирования квадрата с помощью поиска с возвратом, исследование построенного алгоритма.

Формулировка задачи: У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от ***1*** до ***N−1***, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера ***N***. Он может получить ее, собрав из уже имеющихся обрезков (квадратов).

Входные данные: размер столешницы – одно целое число $N(2 \leq N \leq 40)$.

Выходные данные: одно число ***K***, задающее минимальное количество обрезков (квадратов), из которых можно построить столешницу (квадрат) заданного размера ***N***. Далее должны идти ***K*** строк, каждая из которых должна содержать три целых числа ***x***, ***y*** и ***w***, задающие координаты левого верхнего угла $(1 \leq x, y \leq N)$ и длину стороны соответствующего обрезка (квадрата).

Реализация задачи.

Был создан класс **Matrix** со следующими полями:

unsigned int size – размер матрицы.

unsigned int savecount – минимальное число квадратов.

vector<vector<unsigned> > m – матрица NxN, заполненная нулями.

vector<unsigned> printvec – вектор, хранящий координаты и размер квадратов в матрице m.

vector<unsigned> printvecsave – вектор, хранящий координаты и размер минимального количества квадратов в матрице m.

Для класса были созданы следующие методы:

Matrix (unsigned int N) – конструктор объекта класса. Инициализирует вектор **m** нулями.

~Matrix () – деструктор класса.

void push_3_squares() – метод, вызываемый из main(). В зависимости от размера матрицы ставит первые 3 квадрата в левый верхний угол матрицы таким образом, чтобы сократить поле для выполнения поиска с возвратом. Выяснено, что если матрица четной длины, то в наилучшем случае она заполняется четырьмя квадратами со сторонами size/2. Если размер матрицы кратен трем – матрица заполняется тремя квадратами со сторонами 2/3*size, если кратна пяти – со сторонами 3/5*size, иначе – size/2 + 1. Данная стратегия позволяет увеличить производительность поиска с возвратом. После вставки квадратов данный метод вызывает метод поиска с возвратом.

void search(unsigned int x, unsigned int y, unsigned int num) – метод поиска с возвратом, находит разбиение квадрата с координатами левого верхнего угла ***x***, ***y*** на минимальное число квадратов.

void findzero(unsigned int & x, unsigned int & y) – устанавливает координаты на место, в котором еще нет квадрата, если такого места нет – возвращает **false**.

void border_check(unsigned int x, unsigned int y, unsigned int & size) – изменяет размер вставляемого квадрата таким образом, чтобы он не вышел за границы матрицы и не перекрыл уже существующие квадраты.

void push_square(unsigned int x, unsigned int y, unsigned int size, unsigned int num) – метод вставляет по координатам квадрат размера **size** цвета **num**.

void remove_square(unsigned int x, unsigned int y, unsigned int size) – по координатам ***x***, ***y*** удаляет квадрат размера **size**.

void print() – выводит количество квадратов при минимальном разбиении, из координаты и размер.

В функции **main()** считывается размер ***N*** матрицы, затем создается экземпляр класса **Matrix** размера ***N*x*N*** и вызывается метод **push_3_squares()**.

Исследование сложности алгоритма.

Сложность алгоритма рассчитывалась по количеству вызовов метода search для простых чисел, чисел кратных двум, трем и пяти, так как количество

итераций для поиска наименьшего квадрирования определяются этими параметрами.

Количество итераций для четных чисел представлено в табл. 1, для чисел кратных трем – в табл. 2, кратных пяти – в табл. 3 и для простых – в табл. 4.

Таблица 1 – Количество итераций для четных чисел.

Размер матрицы	Количество итераций
2	5
4	6
6	7
8	8
10	9
12	10
14	11
16	12
18	13
20	14
22	15
24	16
26	17
28	18
30	19
32	20
34	21
36	22
38	23
40	24

Таблица 2 – Количество итераций для чисел кратных трем.

Размер матрицы	Количество итераций
3	7
9	25

15	65
21	135
27	243
33	397
39	605

Таблица 3 – Количество итераций для чисел кратных пяти.

Размер матрицы	Количество итераций
5	19
25	1937
35	7102

Таблица 4 – Количество итераций для простых чисел.

Размер матрицы	Количество итераций
7	56
11	709
13	1607
17	9965
19	28267
23	105691
29	733270
31	1746941
37	8463491
41	28047086

Сложность алгоритма зависит от размера матрицы. В худшем случае в N^2 клеток вставляется $N-1$ квадратов, поэтому сложность при переборе всех вариантов составляет $O(N^{N^2})$. Однако метод **push_3_squares** значительно оптимизирует алгоритм, поэтому его сложность не превышает экспоненциальную.

Для четных чисел алгоритм имеет линейную сложность, как видно из табл. 1.

Построены графики для оценки сложности алгоритма для чисел, кратных трем – рис. 1, кратных пяти – рис. 2 и простых чисел – рис. 3.

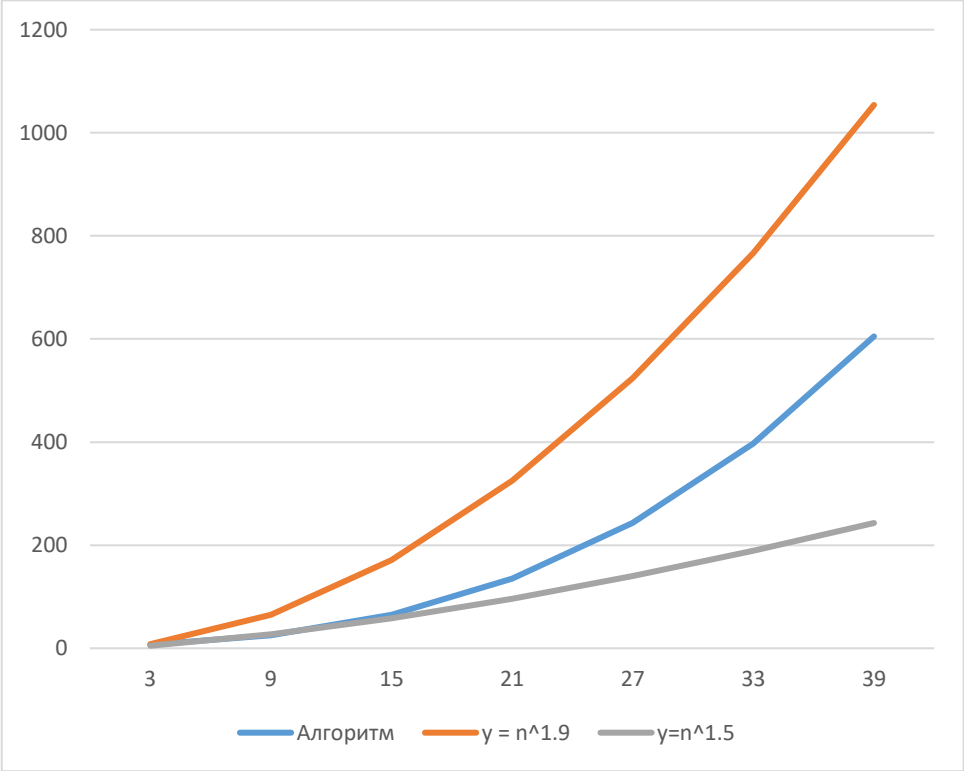

Рисунок 1 – Оценка сложности для чисел кратных трем.

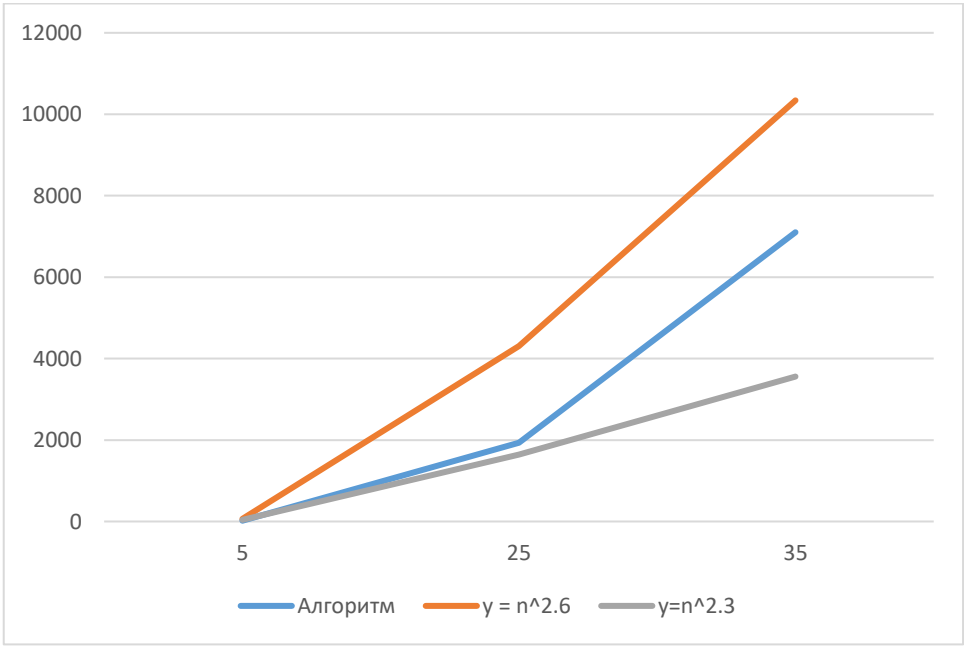

Рисунок 2 – Оценка сложности для чисел кратных пяти.

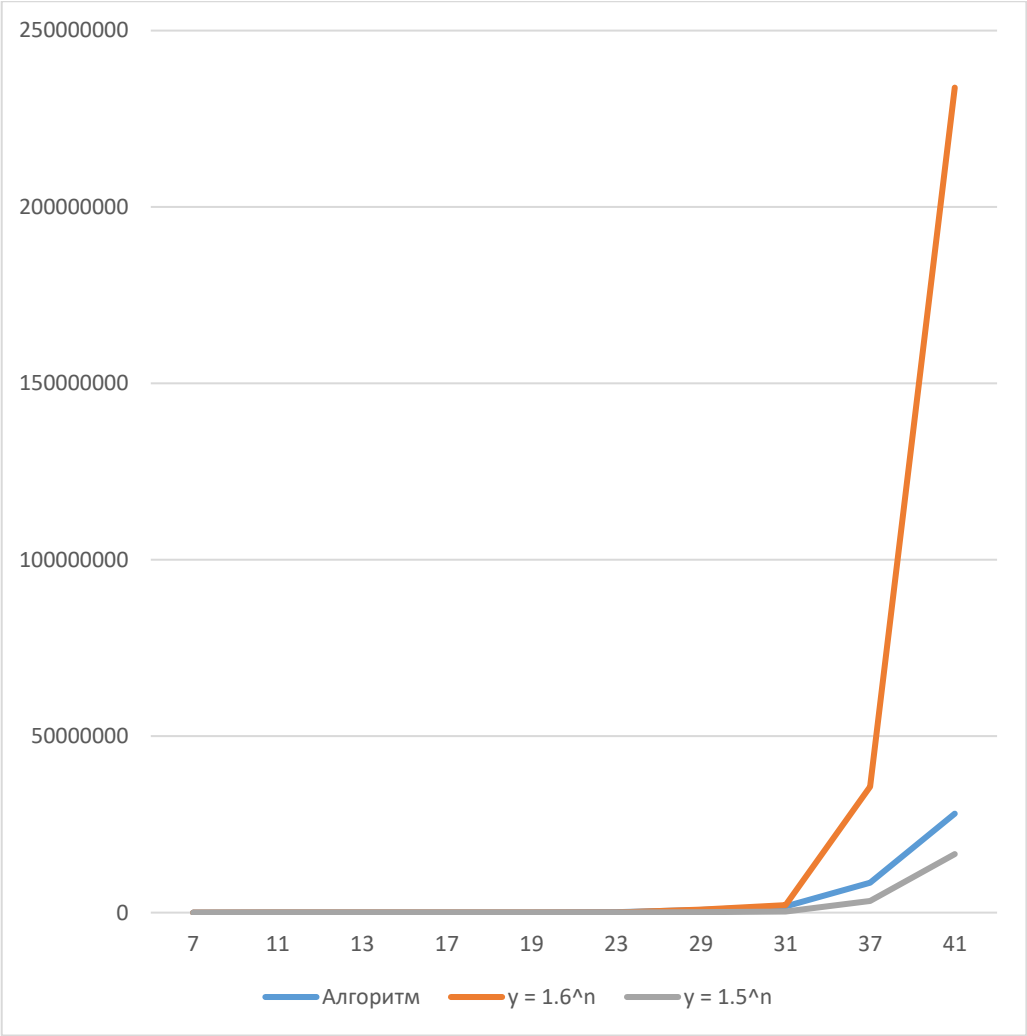

Рисунок 3 – Оценка сложности для простых чисел.

Как видно из рисунка 1, сложность алгоритма для чисел, кратных трем не превышает $N^{1.9}$.

Из рисунка 2 видно, что сложность алгоритма для чисел, кратных пяти не превышает $N^{2.6}$.

Из рисунка 3 видно, что для простых чисел сложность алгоритма не превышает 1.6^N.

Тестирование.

Программа тестировалась в среде разработки Qt с помощью компилятора MinGW 5.3.0 в операционной системе Windows 10.

Тестовые случаи представлены в приложении А.

Вывод.

В ходе выполнения задания был реализован алгоритм квадрирования наименьшим числом квадратов. В алгоритме реализован метод поиска с возвратом. В ходе работы были выяснены сложности алгоритма при работе с числами разной кратности, в том числе простые. Сложность алгоритма не превышает 1.6^N.

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Выходные данные
7	9 0 0 4 4 0 3 0 4 3 3 4 2 3 6 1 4 3 1 4 6 1 5 3 2 5 5 2
23	13 0 0 12 12 0 11 0 12 11 11 12 2 11 14 5 11 19 4 12 11 1 13 11 3 15 19 1 15 20 3 16 11 7 16 18 2 18 18 5
40	4 0 0 20 0 20 20 20 0 20 20 20 20
39	6 0 0 26 0 26 13 26 0 13 13 26 13 26 13 13 26 26 13
37	15 0 0 19 19 0 18 0 19 18

	18 19 2 18 21 5 18 26 11 19 18 1 20 18 3 23 18 8 29 26 3 29 29 8 31 18 6 31 24 1 31 25 1 32 24 5

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

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

class Matrix{
private:
    unsigned int size;
    unsigned int savecount;

    vector<vector<unsigned> > m;
    vector<unsigned> printvec;
    vector<unsigned> printvecsave;

public:
    Matrix(unsigned int N) : size(N){
        for(unsigned int i = 0; i < size; i++){
            savecount = size*size;
            vector<unsigned> temp;
            printvec.reserve(3*size*size);
            printvecsave.reserve(3*size*size);

            for(unsigned int j = 0; j < size; j++)
                temp.push_back(0);
            m.push_back(temp);
        }
    }
    ~Matrix(){
        for(unsigned int i = 0; i < size; i++){
            m[i].clear();
        }
        m.clear();
           printvec.clear();
```

```
            printvecsave.clear();
        size = 0;
            savecount = 0;
    }

    void push_3_squares();
    void push_square(unsigned int x, unsigned int y, unsigned int size,
unsigned int num);
    bool findzero(unsigned int & x, unsigned int & y);
    void search(unsigned int x, unsigned int y, unsigned int num);
    void border_check(unsigned int x, unsigned int y, unsigned int &
size);
    void remove_square(unsigned int x, unsigned int y, unsigned int
size);
    void print();
};

void Matrix :: push_3_squares(){
    if(size % 2 == 0){
        push_square(0, 0, size/2, 1);
        push_square(0, size/2, size/2, 2);
        push_square(size/2, 0, size/2, 3);
        search(size/2, size/2, 3);
    }
    else if(size % 3 == 0){
        push_square(0, 0, 2*size/3, 1);
        push_square(0, 2*size/3, size/3, 2);
        push_square(2*size/3, 0, size/3, 3);
        search(2*size/3, 2*size/3, 3);
    }
    else if(size % 5 == 0){
        push_square(0, 0, 3*size/5, 1);
        push_square(0, 3*size/5, 2*size/5, 2);
        push_square(3*size/5, 0, 2*size/5, 3);
```

```
        search(3*size/5, 2*size/5, 3);
    }
    else{
        push_square(0, 0, size/2 + 1, 1);
        push_square(size/2 + 1, 0, size/2, 2);
        push_square(0, size/2 + 1, size/2, 3);
        search(size/2 + 1, size/2, 3);
    }

}

bool Matrix :: findzero(unsigned int & x, unsigned int & y){
    for(unsigned int i = 0; i < size; i++)
        for (unsigned int j = 0; j < size; j++)
            if(m[i][j] == 0){
                x = i;
                y = j;
                return true;
            }
    return false;
}

void Matrix :: remove_square(unsigned int x, unsigned int y, unsigned int
size){
    printvec.pop_back();
    printvec.pop_back();
    printvec.pop_back();
    for (unsigned int i = x; i < x + size; i++) {
        for (unsigned int j = y; j < y + size; j++){
                m[i][j] = 0;
        }

    }
}
```

```
void Matrix :: push_square( unsigned int x, unsigned int y, unsigned int
size, unsigned int num){
    printvec.push_back(x);
    printvec.push_back(y);
    printvec.push_back(size);
    for(unsigned int i = x; i < x + size; i++){
        for (unsigned int j = y; j < y + size; j++){
                m[i][j] = num;
        }
    }
}

void Matrix :: search(unsigned int x, unsigned int y, unsigned int num){

        if(findzero(x, y)){
            if(num == savecount){
                return;
            }
            for (unsigned int tempsize = size - 1; tempsize > 0 ;
tempsize--) {
                border_check(x, y, tempsize);
                push_square(x, y, tempsize, num + 1);
                search(x, y, num + 1);
                remove_square(x, y, tempsize);
            }
        }
        else {
            if(savecount > num){
                savecount = num;
                printvecsave = printvec;
            }
            return;
        }
```

```
}

void Matrix :: border_check(unsigned int x, unsigned int y, unsigned int
& size){
    unsigned int i = x, j = y;
    if(x + size > this->size){
         size = this->size - x;
     }

     if(y + size > this->size){
         size = this->size - y;
     }

    for(; i < x + size; i++){
        if(m[i][y] != 0){
            break;
        }
    }
    for (; j < y + size; j++) {
        if(m[x][j] != 0){
            break;
        }
    }
    if(i - x < j - y)
        size = i - x;
    else if(j - y < i - x){
        size = j - y;
    }
    return;
}

void Matrix :: print(){
    cout << savecount << endl;
```

```
    for (unsigned i = 0; i < printvecsave.size(); i += 3) {
        cout << printvecsave[i] << ' ' << printvecsave[i + 1] << ' ' <<
printvecsave[i + 2] << endl;
    }
}


int main()
{
    unsigned int N;
    cin >> N;
    Matrix table(N);
    table.push_3_squares();
    table.print();
    return 0;
}
```