

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МОЭВМ**

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

Студент гр. 0382

\_\_\_\_\_

Корсунов А.А.

Преподаватель

\_\_\_\_\_

Шевская Н.В.

Санкт-Петербург

2022

### **Цель работы.**

Применить на практике знания о построение алгоритмов поиска с возвратом, реализовать заполнение поля наименьшим числом квадратов, применив алгоритм поиска с возвратом. Исследовать количество операций в зависимости от размеров поля.

### **Основные теоретические положения.**

Поиск с возвратом или бэктрекинг — метод нахождения решений некоторой задачи, основанный на полном переборе всех возможных вариантов решения. Поиск с возвратом есть последовательное расширение частичного решения, где на очередном шаге убирается расширение, в случае если оно невозможно, и производится переход к более короткому решению.

### **Задание.**

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).

Например, столешница размера  $7 \times 7$  может быть построена из 9 обрезков.

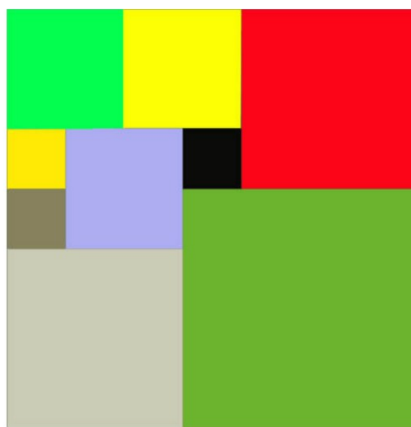


Рисунок 1 - Графическое представление столешницы

Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

### **Входные данные**

Размер столешницы - одно целое число  $N$  ( $2 \leq N \leq 20$ ).

### **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).

**Вариант 3и** — Итеративный бэктрекинг. Исследование количества операций от размера квадрата.

### Ход работы.

1. Был произведен анализ задания.
2. Был реализован итеративный алгоритм поиска с возвратом:

Алгоритм:

1) Столешница — матрица квадратов размера 1 на 1, элементы этой матрицы — значения типа `int`, которые принимают «0», если этот элемент не входит хоть в какой-то квадрат и «1», если входит в хоть какой-то квадрат.

2) Происходит проверка матрицы на пустые ячейки, если есть — в нее ставится квадрат с наиболее возможной стороной, фиксируется текущее минимальное число квадратов в столешнице, то же самое производится для следующих пустых ячеек. Когда матрица заполнена — фиксируется текущее установленное минимальное число квадратов в столешнице, происходит возврат до того квадрата, который имеет сторону большую 1-цы. Для такого квадрата сторона уменьшается на единицу и рассматривается проверка данной комбинации квадратов по уже описанному выше алгоритму с проверкой на пустые ячейки в матрице. Также производится проверка текущего минимального числа квадратов с уже установленным текущим числом квадратов, если первое число превышает и равно второму — происходит возврат по описанному выше алгоритму. Алгоритм будет работать до тех пор, пока в столешнице есть квадраты.

3) Были определены частные случаи алгоритма, а именно, когда размер столешницы кратен 2, и когда размер столешницы кратен 3, но не кратен 2:

а)

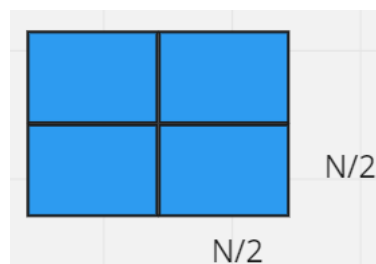


Рисунок 2 - Иллюстрация столешницы, сторона кратна 2,  $K = 4$

Очевидно, что такой квадрат можно разбить на 4 подкварата. Такое разбиение будет наименее возможным, потому как выборка квадратов с большей или меньшей стороной  $N$  приведет к вынужденному дроблению области на большее число квадратов.

б)

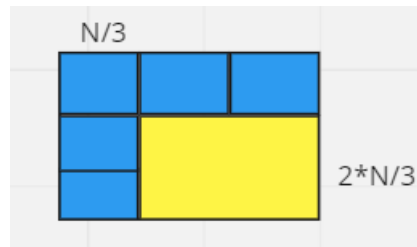


Рисунок 3 - Иллюстрация столешницы, сторона кратна 2,  $K = 6$

Очевидно, что квадрат, сторона которого не кратна 2-ум, но кратна 3-ем можно разбить на 9-ть подкваратов со сторонами  $N/3$ , 4-ре таких подкварата, имеющие по 2 общие стороны с 1-им из 4-ех квадратов, составляют подквadrat, которым можно будет их заменить. Таким образом получится 6 подкваратов, один из которых имеет сторону  $2 \cdot N/3$ , а все другие -  $N/3$ . Такое разбиение будет наименьше-возможным, потому как:

а) если изначально рассматривать не равное разбиение подкваратов (т.е. не  $N/3$ ), то получится ситуация, где в разбиение будут подквараты со сторонами  $a=1$ . Другими словами, всегда будет оставаться пустая прямоугольная область, которую придется вынужденно заполнять такими подкваратами.

б) другое объединение нескольких подкваратов в 1-ин подквadrat невозможно, потому как в других случаях будет получаться не квадратная фигура.

4) После этого было замечено, что во всех оставшихся ситуациях заполненная столешница имеет в одном из углов три квадрата со сторонами  $(N/2) + 1$ ,  $N/2$  и  $N/2$ , которые должны располагаться в одном из углов.

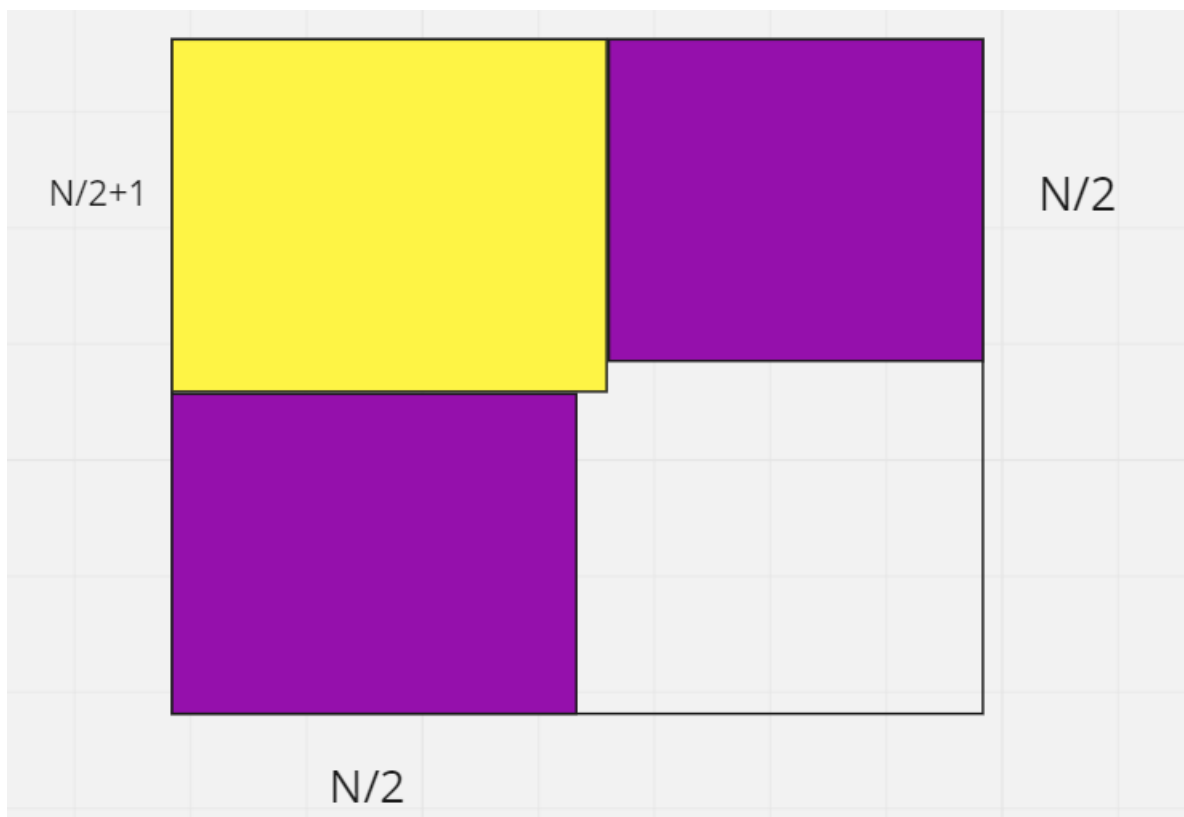


Рисунок 4 - Иллюстрация столешницы, сторона столешницы не кратна 2 и 3

5) Было проведено исследование количества операций от размера столешницы:

Операция — запись в ячейке матрицы столешницы (запись «0» или «1»). В силу оптимизаций алгоритма, измерялось лишь количество операций на столешницах, длина сторон которых была простым числом.

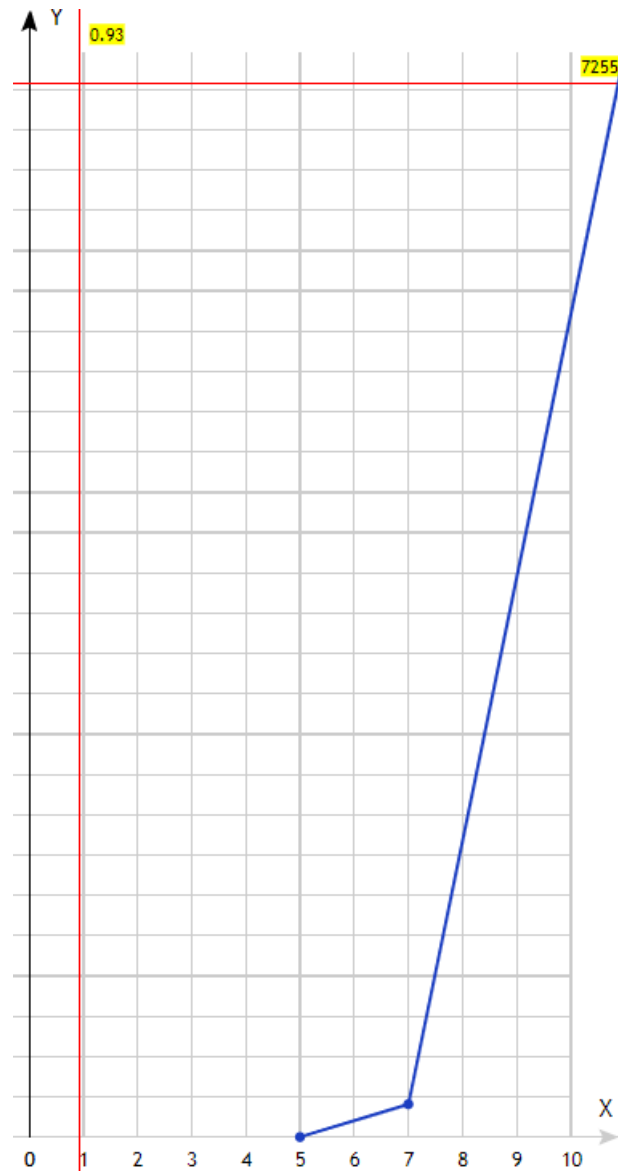


Рисунок 5 - Иллюстрация графика, где «x» – размер столешницы, «y» – число операций.

Из графика видно, что зависимость - экспоненциальная.

Сложность алгоритма по времени —  $O((N/2)^{(N/2)})$ ;

Сложность алгоритма по памяти —  $O(N^2)$ ;

Функции и структуры данных:

-class Square:

\*side, x, y – сторона квадрата, его координаты.

\*Square(int, int, int) – конструктор квадратов;

\*~Square(int, int, int) – деструктор квадратов;

\*...\_get() – получить поле;

\*...set(int) – поставить поле;

-class Field:

\*std::vector <Square> squares — вектор квадратов;

\*std::vector <Square> local\_squares — локальный вектор квадратов;

\*int main\_size = 0 — сторона столешницы;

\*int square\_number = 0 — количество квадратов;

\*int operation\_amount = 0 — количество операций;

\*void backtracking(Field\*, int) — основной метод в классе Field бэктрекинга, который принимает на вход объект класса Field и размер столешницы;

\*void add\_vector\_local\_squares(int, int, int) — метод добавления квадратов в локальный вектор квадратов, на вход принимает сторону и координаты;

\*void set\_default\_local\_squares(int, int\*\*) - установка начальных параметров для первых трех квадратов, на вход принимает сторону и матрицу;

\*void set\_main\_size(int) — установка начального минимального количества квадратов и стороны столешницы, принимает сторону столешницы;

\*void fill\_empty\_square(int\*\*, int, int, int, int); - заполнение текущего квадрата в матрице «0» или «1», принимает на вход матрицу, координаты, сторону квадрата, «0» или «1»);

\*bool check\_matrix(int\*\*, int, int&, int&) - проверка, пуста ли матрица, принимает на вход матрицу, сторону столешницы;

\*int find\_side(int\*\*, int, int, int) — ищет максимально возможный квадрат для данной координаты сверху-вниз, слева-направо, принимает матрицу, сторону столешницы, координаты);

\*void back\_vector(int\*\*, int&, int&) - удаляет из матрицы и из вектора квадратов «0» или «1» и квадрат соответственно, проверяет общее число квадратов, передает флаг о том, что алгоритм закончен, принимает на вход матрицу и координаты;

\*void print\_squares() - выводит минимальное число квадратов и вектор



квадратов.

6 ) Тестирование:

а) Входные данные: 2

Выходные данные: 4

1 1 1

2 1 1

1 2 1

2 2 1

б) Входные данные:3

Выходные данные: 6

1 1 2

3 1 1

3 2 1

1 3 1

2 3 1

3 3 1

в) Входные данные:5

Выходные данные: 8

1 1 3

1 4 2

4 1 2

3 4 2

4 3 1

5 3 1

5 4 1

5 5 1

г) Входные данные: 7

Выходные данные: 9

1 1 4

1 5 3

5 1 3

4 5 2

4 7 1

5 4 1

5 7 1

6 4 2

6 6 2

д) Входные данные: 13

Выходные данные: 11

1 1 7

1 8 6

8 1 6

7 8 2

7 10 4

8 7 1

9 7 3

11 10 1

11 11 3

12 7 2

12 9 2

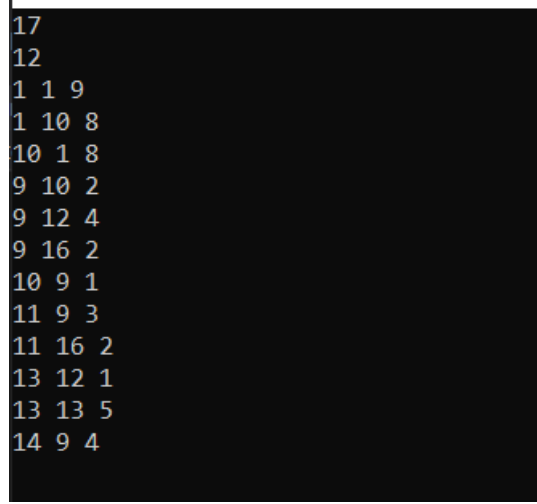
е) Входные данные: 17

Выходные данные: 12

1 1 9

1 10 8

10 1 8  
9 10 2  
9 12 4  
9 16 2  
10 9 1  
11 9 3  
11 16 2  
13 12 1  
13 13 5  
14 9 4



```
17  
12  
1 1 9  
1 10 8  
10 1 8  
9 10 2  
9 12 4  
9 16 2  
10 9 1  
11 9 3  
11 16 2  
13 12 1  
13 13 5  
14 9 4
```

Рисунок 6 - Пример работы программы

**Вывод.**

Были применены на практике знания о построение алгоритмов поиска с возвратом, были реализовано заполнение поля наименьшим числом квадратов, применив алгоритм поиска с возвратом. Было произведено исследование количества операций в зависимости от размеров поля.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <vector>
#include <algorithm>
#include <iostream>
```

```
class Square
```

```
{
```

```
private:
```

```
    int side = 0;
```

```
    int x = 0;
```

```
    int y = 0;
```

```
public:
```

```
    Square() = default;
```

```
    Square(int s, int x, int y) : side(s), x(x), y(y) {}
```

```
    ~Square() = default;
```

```
    void set_side(int side)
```

```
    {
```

```
        this->side = side;
```

```
    }
```

```
    void set_x(int x)
```

```
    {
```

```
        this->x = x;
```

```
    }
```

```
    void set_y(int y)
```

```
    {
```

```
        this->y = y;
```

```

    }
    int get_x()
    {
        return x;
    }
    int get_y()
    {
        return y;
    }
    int get_side()
    {
        return side;
    }
};

```

```

class Field
{
private:
    std::vector <Square> squares;
    std::vector <Square> local_squares;
    int main_size = 0;
    int square_number = 0;
    int operation_amount = 0;
public:
    void backtracking(Field*, int);
    void add_vector_local_squares(int, int, int);
    void set_deafult_local_squares(int, int**);
    void set_main_size(int);
    void fill_empty_square(int**, int, int, int, int);

```

```

    bool check_matrix(int**, int, int&, int&);
    int find_side(int**, int, int, int);
    void back_vector(int**, int&, int&);
    void print_squares();
};

```

```

void Field::fill_empty_square(int** matrix, int x, int y, int square_side, int flag)
{
    for (int i = x; i < x + square_side; i++)
    {
        for (int j = y; j < y + square_side; j++)
        {
            matrix[i][j] = flag;
            operation_amount++;
        }
    }
}

```

```

void Field::backtracking(Field* square_vector, int size)
{
    set_main_size(size);

    int** matrix = new int* [main_size];
    for (int i = 0; i < main_size; i++)
    {
        matrix[i] = new int[main_size];
    }

    for (int i = 0; i < main_size; i++)
    {

```

```

    for (int j = 0; j < main_size; j++)
    {
        matrix[i][j] = 0;
    }
}

int local_x = 0;
int local_y = 0;
int local_max_side = 3;
int local_next_side = 0;
int flag = 0;

set_deafult_local_squares(main_size, matrix);

while (flag != 1)
{
    if (check_matrix(matrix, main_size, local_x, local_y))// если пуста = true
    {
        int next_side = find_side(matrix, main_size, local_x, local_y); // сторона
        следующего квадрата
        fill_empty_square(matrix, local_x, local_y, next_side, 1); // заполняется
        матрица этого квадрата
        local_squares.push_back(Square(next_side, local_x, local_y)); //в локальный
        вектор квадратов помещается новый квадрат
        local_max_side++; //локальное количество квадратов увеличивается на
        1
    }
    else // если не пуста - запоминаем в square_vector
    {
        squares = local_squares;
    }
}

```



```

        square_number = local_max_side;
        back_vector(matrix, local_max_side, flag);
    }

    if (local_max_side >= square_number) //если количество локальных
квадратов больше/равно текущего минимума
    {
        back_vector(matrix, local_max_side, flag);
    }

}

for (int i = 0; i < main_size; i++)
{
    delete[] matrix[i];
}

delete[] matrix;

std::cout << "operation amount: " << operation_amount << "\n";
std::cout << "square number: " << square_number << "\n";
print_squares();
}

void Field::back_vector(int** matrix, int& local_max_side, int& flag)
{
    while (true)
    {
        if (local_max_side > 3) //локальное число квадратов > 3
        {
            Square square = local_squares[local_max_side - 1]; //квадрат равный
последнему квадрату

```

```

        if (local_squares[local_max_side - 1].get_side() >= 2) //сторона
последнего квадрата >= 2
        {
            local_squares.pop_back(); //удаление последнего квадрата
            fill_empty_square(matrix, square.get_x(), square.get_y(),
square.get_side(), 0); //обнуление ячеек последнего квадрата в матрице
            fill_empty_square(matrix, square.get_x(), square.get_y(), square.get_side()
- 1, 1); //заполнение квадрата side - 1
            local_squares.push_back(Square(square.get_side() - 1, square.get_x(),
square.get_y())); //добавление такого квадрата в вектор квадратов
            break;
        }
        else //сторона последнего квадрата < 2
        {
            local_squares.pop_back(); //удаление последнего квадрата
            local_max_side--; //уменьшение локального числа квадратов на 1
            fill_empty_square(matrix, square.get_x(), square.get_y(),
square.get_side(), 0); //обнуление ячеек последнего квадрата в матрице
            continue;
        }
    }
    else //локальное число квадратов <= 3
    {
        flag = 1;
        break;
    }
}
}

```

```

int Field::find_side(int** matrix, int matrix_size, int x, int y)

```

```

{
    int right_side = 1;
    int down_side = 1;

    while (y + right_side < matrix_size && matrix[x][y + right_side] != 1)
    {
        right_side++;
    }

    while (x + down_side < matrix_size && matrix[x + down_side][y] != 1)
    {
        down_side++;
    }
    return std::min(right_side, down_side);
}

```

```

bool Field::check_matrix(int** array, int array_size, int& x, int& y)
{
    for (int i = 0; i < array_size; i++)
    {
        for (int j = 0; j < array_size; j++)
        {
            if (array[i][j] == 0)
            {
                x = i;
                y = j;
                return true;
            }
        }
    }
}

```

```

    return false;
}

void Field::set_deafult_local_squares(int size, int** matrix)
{
    set_main_size(size);
    add_vector_local_squares(main_size / 2 + 1, 0, 0);
    add_vector_local_squares(main_size / 2, 0, main_size / 2 + 1);
    add_vector_local_squares(main_size / 2, main_size / 2 + 1, 0);
    fill_empty_square(matrix, 0, 0, main_size / 2 + 1, 1);
    fill_empty_square(matrix, 0, main_size / 2 + 1, main_size / 2, 1);
    fill_empty_square(matrix, main_size / 2 + 1, 0, main_size / 2, 1);
}

void Field::add_vector_local_squares(int side, int x, int y)
{
    local_squares.push_back(Square(side, x, y));
}

void Field::set_main_size(int size)
{
    main_size = size;
    square_number = size * size;
}

void Field::print_squares()
{
    for (int i = 0; i < squares.size(); i++)
    {
        squares[i].set_x(squares[i].get_x() + 1);
    }
}

```

```

    squares[i].set_y(squares[i].get_y() + 1);
    std::cout << squares[i].get_x() << ' ' << squares[i].get_y() << ' ' <<
squares[i].get_side() << "\n";
}
}

```

```

int main()

```

```

{

```

```

    while (true)

```

```

    {

```

```

        Field matrix;

```

```

        int N;

```

```

        std::cin >> N;

```

```

        if (N < 2 || N > 20)

```

```

        {

```

```

            break;

```

```

        }

```

```

        if (N % 2 == 0)

```

```

        {

```

```

            std::cout << "4\n"; //для четного

```

```

            std::cout << "1 1 " << N / 2 << "\n"; //верхний левый

```

```

            std::cout << 1 + N / 2 << " " << "1 " << N / 2 << "\n"; //верхний правый

```

```

            std::cout << "1 " << 1 + N / 2 << " " << N / 2 << "\n"; //нижний левый

```

```

            std::cout << 1 + N / 2 << " " << 1 + N / 2 << " " << N / 2 << "\n";

```

```

//нижний правый

```

```

        }

```

```

        else if (N % 3 == 0)

```

```

{
    std::cout << "6\n"; // для mod 3 = 0
    std::cout << "1 1 " << 2 * N / 3 << "\n"; //верхний левый
    std::cout << 1 + 2 * N / 3 << " " << "1 " << N / 3 << "\n"; //верхний
    правый 1
    std::cout << 1 + 2 * N / 3 << " " << 1 + N / 3 << " " << N / 3 << "\n";
    //верхний правый 2
    std::cout << "1 " << 1 + 2 * N / 3 << " " << N / 3 << "\n"; //нижний
    левый 1
    std::cout << 1 + N / 3 << " " << 1 + 2 * N / 3 << " " << N / 3 << "\n";
    //нижний левый 2
    std::cout << 1 + 2 * N / 3 << " " << 1 + 2 * N / 3 << " " << N / 3 << "\n"; //нижний правый
}

else
{
    matrix.backtracking(&matrix, N);
}

std::cout << " \n";
}

return 0;
}

```