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. 9383 Поплавский И.

Преподаватель Фирсов М.А.

Санкт-Петербург

2021

Цель работы

Изучить принцип работы алгоритма поиска с возвратом. Решить с его помощью задачу на языке программирования C++.

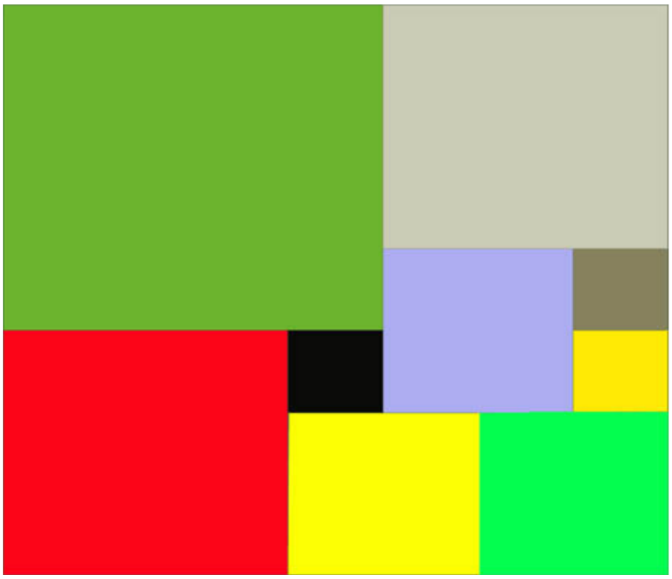
Основные теоретические положения.

Поиск с возвратом, бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения. Однако время нахождения решения может быть очень велико даже при небольших размерностях задачи (количестве исходных данных), причём настолько велико, что о практическом применении не может быть и речи. Поэтому при проектировании таких алгоритмов, обязательно нужно теоретически оценивать время их работы на конкретных данных.

Постановка 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).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



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≤N≤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 ≤ x, y ≤ N) и длину стороны соответствующего обрезка (квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 4

5 1 3

1 5 3

4 5 2

4 7 1

5 4 1

5 7 1

6 4 2

6 6 2

Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

Реализация задачи

Описание алгоритма

Для решения задачи был использован рекурсивный поиск с возвратом. Поиск осуществляется перебором вариантов для расстановки очередной фигуры. Для оптимизации алгоритма рассмотрены частные случаи для квадратов со сторонами кратными двум, трем и пяти. Для уменьшения количества расстановок в остальных случаях строятся первые три квадрата, таких что сторона первого равна (N+1)/2, а второго и третьего – N/2.

Размеры квадратов в бэктрекинге берутся от большего к меньшему, координаты от левого верхнего угла, к правому нижнему. Когда поле оказывается заполнено, результат сравнивается с предыдущим лучшим и при необходимости запоминается, после чего происходит возврат к предыдущему заполнению и рассматривается другая переборная комбинация.

Сложность по времени витка рекурсии является многочленом большой степени, с учетом вложенности — четвертую. Учитывая рекурсивный бэктрекинг сложность близка к экспоненте. Затраты по памяти пропорциональны глубине рекурсии, которая не превышает N^2.

Частичные решения хранятся в виде троек чисел с координатами и размером, а также в виде матрицы с нулями в свободных клетках и иными значениями в местах, соответствующих квадратам.

Описание функций и структур данных

Был использован следующий класс: class Square

Переменные класса Square:

int **coloring - раскраска по номеру квадрата

int *abscissa - массив координат x,

int *ordinate - массив координат y,

int *length - массив длин сторон квадратов,

int count - возможное кол-во квадратов,

int size - размер текущего квадрата,

int num - порядковый номер квадрата,

bool f - последний возможный квадрат.

Методы класса Square:

- Square(int size) – конструктор класса. Принимает размер квадрата и создает двумерный массив coloring, заполненный нулями. Также создает массивы координат abscissa и ordinate и массив длин length. Устанавливает флаг f = false.
- ~Square() – деструктор класса. Удаляет двумерный массив coloring, массивы координат abscissa и ordinate и массив длин length.
- insert_square(int x, int y, int n, int side) - помещает по заданным координатам x и y левого верхнего угла квадрат со стороной side и номером n. Т.е. заполняет его в массиве «цветом» порядкового номера квадрата.
- remove_square(int x, int y, int side) - удаляет квадрат по заданным координатам x и y левого верхнего угла квадрат со стороной side. Т.е. обнуляет его в массиве.
- place_to_insert(int &x, int &y) - поиск места для вставки нового квадрата, функция ищет самую верхнюю и левую пустую клетку поля. Устанавливает x и y как координаты такой пустой клетки и возвращает true, если пустая клетка имеется, и false – нет пустых клеток.

- multiple_of_three(int side) – частный случай разбиение квадрата, сторона которого кратна трём. Принимает сторону исходного квадрата и заполняет массивы координат и массив длин согласно частному случаю.
- find_max_size(int x, int y) - находит максимальное значение стороны квадрата, который возможно поместить на поле от переданных координат левого верхнего угла. Проверяет на пересечение с границами квадрата и уже существующими квадратами, возвращает максимальную возможную длину стороны.
- multiple_of_five(int side) - частный случай разбиение квадрата, сторона которого кратна пяти. Принимает сторону исходного квадрата и заполняет массивы координат и массив длин согласно частному случаю.
- insert_the_second_square() - вставляет второй квадрат снизу от первого так, чтобы его сторона была максимально возможной.
- insert_the_third_square() - вставляет третий квадрат слева от первого так, чтобы его сторона была максимально возможной.
- even_square(int side) - частный случай разбиение квадрата, сторона которого кратна двум. Принимает сторону исходного квадрата и заполняет массивы координат и массив длин согласно частному случаю.
- print_square(int deep) - вспомогательная функция, которая выводит получившийся квадрат с рекурсивным отступом deep.
- output_of_the_result(int amount) - передается минимальное количество квадратов для разбиения, выводит результат работы программы на консоль.
- insert_the_first_square() - функция вставки самого первого квадрата, учитывая размер всего поля: если сторона кратна двум, то вызывает функцию even_square(); если она кратна трём, то multiple_of_three(); если она кратна пяти, то multiple_of_five(); если же сторона не кратна приведенным выше числам, то сторона первого квадрата будет равна половине стороны поля n + 1, далее вызываются функции вставки второго и третьего квадрата, запускается бэктрекинг. Это нужно для

оптимизации работы кода, чтобы уменьшить количество вариантов, в использовании backtracking.

- int backtracking(int deep) – рекурсивная функция, перебирает всевозможные варианты кадрирования неразбитого участка поля. Принимает текущее минимальное количество квадратов для разбиения. Находит свободное место для вставки квадрата и его максимальную длину стороны. Когда поле оказывается заполнено результат сравнивается с предыдущим лучшим и при необходимости запоминается, после чего происходит возврат к предыдущему заполнению и рассматривается другая переборная комбинация.

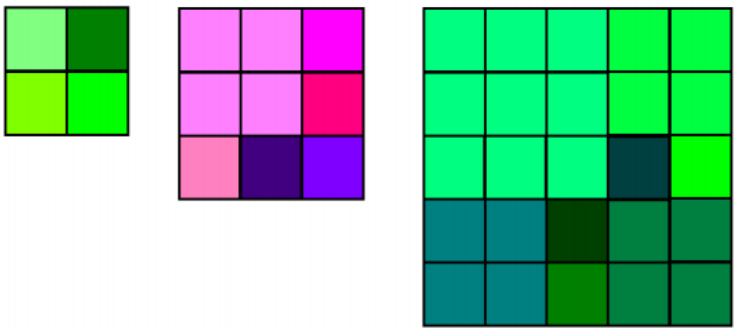


Рисунок 1 – разбиение квадратов, сторона которых кратна 2, 3 и 5

Тестирование

Входные данные		Выходные данные
Кратные 2	2	Квадрат с четной стороной. Частный случай. Делим квадрат на 4 части. Минимальное число квадратов для разбиения 4 1 1 1 2 1 1 1 2 1 2 2 1 Время работы программы в секундах 0
	6	Квадрат с четной стороной. Частный случай. Делим квадрат на 4 части. Минимальное число квадратов для разбиения 4 1 1 3 4 1 3 1 4 3 4 4 3 Время работы программы в секундах 0

	20	Квадрат с четной стороной. Частный случай. Делим квадрат на 4 части. Минимальное число квадратов для разбиения 4 1 1 10 11 1 10 1 11 10 11 11 10 Время работы программы в секундах 0
	222	Квадрат с четной стороной. Частный случай. Делим квадрат на 4 части. Минимальное число квадратов для разбиения 4 1 1 111 112 1 111 1 112 111 112 112 111 Время работы программы в секундах 0
Кратные 3	9	Квадрат со стороной кратной 3. Частный случай. Делим квадрат на 6 частей. Минимальное число квадратов для разбиения 6 1 1 6 7 1 3 7 4 3 7 7 3 1 7 3 4 7 3 Время работы программы в секундах 0
	33	Квадрат со стороной кратной 3. Частный случай. Делим квадрат на 6 частей. Минимальное число квадратов для разбиения 6 1 1 22 23 1 11 23 12 11 23 23 11 1 23 11 12 23 11 Время работы программы в секундах 0
	333	Квадрат со стороной кратной 3. Частный случай. Делим квадрат на 6 частей. Минимальное число квадратов для разбиения 6 1 1 222 223 1 111 223 112 111 223 223 111 1 223 111 112 223 111

		Время работы программы в секундах 0
Кратные 5	5	Квадрат со стороной кратной 5. Частный случай. Делим квадрат на 8 частей. Минимальное число квадратов для разбиения 8 1 1 3 4 1 2 4 3 1 5 3 1 1 4 2 3 4 1 3 5 1 4 4 2 Время работы программы в секундах 0
	25	Квадрат со стороной кратной 5. Частный случай. Делим квадрат на 8 частей. Минимальное число квадратов для разбиения 8 1 1 15 16 1 10 16 11 5 21 11 5 1 16 10 11 16 5 11 21 5 16 16 10 Время работы программы в секундах 0
	125	Квадрат со стороной кратной 5. Частный случай. Делим квадрат на 8 частей. Минимальное число квадратов для разбиения 8 1 1 75 76 1 50 76 51 25 101 51 25 1 76 50 51 76 25 51 101 25 76 76 50 Время работы программы в секундах 0
Простые числа	7	Общий случай. Используем бэктрекинг. Минимальное число квадратов для разбиения 9 1 1 4 5 1 3 1 5 3 4 5 2 4 7 1 5 4 1

		5 7 1 6 4 2 6 6 2 Время работы программы в секундах 0
	11	Общий случай. Используем бэктрекинг. Минимальное число квадратов для разбиения 11 1 1 6 7 1 5 1 7 5 6 7 3 6 10 2 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3 Время работы программы в секундах 0
	13	Минимальное число квадратов для разбиения 11 1 1 7 8 1 6 1 8 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 Время работы программы в секундах 0
	29	Общий случай. Используем бэктрекинг. Минимальное число квадратов для разбиения 14 1 1 15 16 1 14 1 16 14 15 16 2 15 18 5 15 23 7 16 15 1 17 15 3 20 15 3 20 18 3 20 21 2 22 21 1

		22 22 8 23 15 7 Время работы программы в секундах 0
	37	Общий случай. Используем бэктрекинг. Минимальное число квадратов для разбиения 15 1 1 19 20 1 18 1 20 18 19 20 2 19 22 5 19 27 11 20 19 1 21 19 3 24 19 8 30 27 3 30 30 8 32 19 6 32 25 1 32 26 1 33 25 5 Время работы программы в секундах 9
	41	Общий случай. Используем бэктрекинг. Минимальное число квадратов для разбиения 16 1 1 21 22 1 20 1 22 20 21 22 3 21 25 7 21 32 10 22 21 1 23 21 1 24 21 4 28 21 4 28 25 4 28 29 3 31 29 1 31 30 1 31 31 11 32 21 10 Время работы программы в секундах 35
Некорректный ввод	0	Неверный ввод
	1	Неверный ввод
	-6	Неверный ввод

Пример вывода промежуточных данных

```
Введите размер стола:
7
Общий случай. Используем бэктрекинг.
Построили первые три квадрата и запустили рекурсивную функцию.
1 1 1 1 3 3 3
1 1 1 1 3 3 3
1 1 1 1 3 3 3
1 1 1 1 0 0 0
2 2 2 0 0 0 0
2 2 2 0 0 0 0
2 2 2 0 0 0 0

       Вставляем новый квадрат по координатам 4 5 со стороной 3 под номером 4
       1 1 1 1 3 3 3
       1 1 1 1 3 3 3
       1 1 1 1 3 3 3
       1 1 1 1 4 4 4
       2 2 2 0 4 4 4
       2 2 2 0 4 4 4
       2 2 2 0 0 0 0

              Вставляем новый квадрат по координатам 5 4 со стороной 1 под номером 5
              1 1 1 1 3 3 3
              1 1 1 1 3 3 3
              1 1 1 1 3 3 3
              1 1 1 1 4 4 4
              2 2 2 5 4 4 4
              2 2 2 0 4 4 4
              2 2 2 0 0 0 0

                     Вставляем новый квадрат по координатам 6 4 со стороной 1 под номером 6
                     1 1 1 1 3 3 3
                     1 1 1 1 3 3 3
                     1 1 1 1 3 3 3
                     1 1 1 1 4 4 4
                     2 2 2 5 4 4 4
                     2 2 2 6 4 4 4
                     2 2 2 0 0 0 0

                            Вставляем новый квадрат по координатам 7 4 со стороной 1 под номером 7
                            1 1 1 1 3 3 3
                            1 1 1 1 3 3 3
                            1 1 1 1 3 3 3
                            1 1 1 1 4 4 4
                            2 2 2 5 4 4 4
                            2 2 2 6 4 4 4
                            2 2 2 7 0 0 0

                                   Вставляем новый квадрат по координатам 7 5 со стороной 1 под номером 8
                                   1 1 1 1 3 3 3
                                   1 1 1 1 3 3 3
                                   1 1 1 1 3 3 3
                                   1 1 1 1 4 4 4
                                   2 2 2 5 4 4 4
                                   2 2 2 6 4 4 4
                                   2 2 2 7 8 0 0

                                          Вставляем новый квадрат по координатам 7 6 со стороной 1 под номером 9
                                          1 1 1 1 3 3 3
                                          1 1 1 1 3 3 3
                                          1 1 1 1 3 3 3
                                          1 1 1 1 4 4 4
                                          2 2 2 5 4 4 4
                                          2 2 2 6 4 4 4
                                          2 2 2 7 8 9 0

                                                 Вставляем новый квадрат по координатам 7 7 со стороной 1 под номером 10
                                                 1 1 1 1 3 3 3
                                                 1 1 1 1 3 3 3
                                                 1 1 1 1 3 3 3
                                                 1 1 1 1 4 4 4
                                                 2 2 2 5 4 4 4
                                                 2 2 2 6 4 4 4
                                                 2 2 2 7 8 9 10

                                                        Больше нет места для вставки. Текущее разбиение 10
                                                        Получили новый меньший результат 10
                                                 Сохраним координаты и размер текущего квадрата
                                                 Вернемся назад. Удалим последний квадрат.
                                          Сохраним координаты и размер текущего квадрата
                                          Вернемся назад. Удалим последний квадрат.
                                   Сохраним координаты и размер текущего квадрата
```

Зависимость времени выполнения от размера входных данных

N	С промежуточными выводами (сек)	Без промежуточных выводов (сек)
:2	0	0
:3	0	0
:5	0	0
7	1	0
11	14	0
13	32	0
17	342	0

19	1216	0
23	5940	0
29	-	0
31	-	5
37	-	9
39	-	9
41	-	35

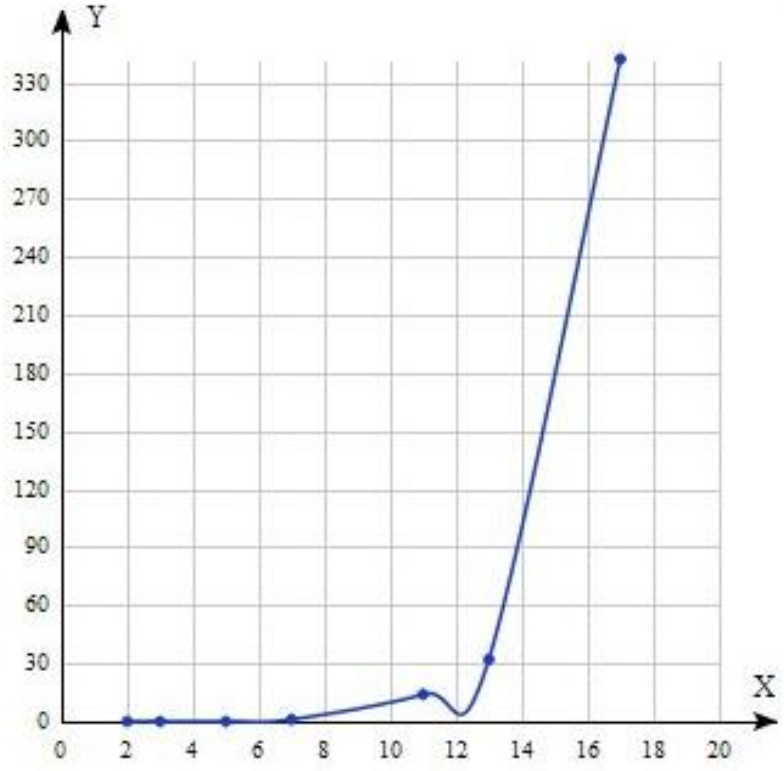


Рисунок 2 – график зависимости времени от входных данных

Вывод

В результате работы была написана полностью рабочая программа, решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны. По результатам исследования можем заключить, что зависимости времени вычислений от размера поля экспоненциальна.

ПРИЛОЖЕНИЕ

ИСХОДНЫЙ КОД ПОРОГРАММЫ

Файл Square.h

```
#pragma once
#include <iostream>
#include <iostream>
#include <ctime>

class Square {
private:
	int** coloring; //раскраска по номеру квадрата
	int* abscissa;  //массив координат x
	int* ordinate;  //массив координат y
	int* length;    //массив длин сторон квадратов
	int  count;      //возможное кол-во квадратов
	int  size;       //размер текущего квадрата
	int  num;        //порядковый номер квадрата
	bool f;          //последний возможный квадарт

	void insert_square(int x, int y, int n, int side);
	void remove_square(int x, int y, int side);
	bool place_to_insert(int& x, int& y);
	bool multiple_of_three(int side);
	int  find_max_size(int x, int y);
	bool multiple_of_five(int side);
	bool insert_the_second_square();
	bool insert_the_third_square();
	bool even_square(int side);
	void print_square(int deep);
	friend class test;

public:
	void output_of_the_result(int amount);
	int  insert_the_first_square();
	int  backtracking(int deep);
	Square(int size);
	~Square();
};
```

Файл source.cpp

```
#include "Square.h"
#define N 40


void Square::insert_square(int x, int y, int n, int side) { //заполнение квадрата
(координаты, цвет и сторона)
	for (int i = x; i < side + x; i++)
		for (int j = y; j < side + y; j++)
			coloring[i][j] = n;
}

void Square::remove_square(int x, int y, int side) { //удаление квадрата (координаты и
сторона)
	for (int i = x; i < side + x; i++)
		for (int j = y; j < side + y; j++)
			coloring[i][j] = 0;
}
```

```
bool Square::place_to_insert(int &x, int &y) { //поиск места для вставки нового квадрата, функция ищет самую верхнюю и левую пустую клетку поля
	for (int i = 0; i < size; i++)
		for (int j = 0; j < size; j++)
			if (!coloring[i][j]) {
				x = i;
				y = j;
				return true;
			}
	return false;
}

bool Square::multiple_of_three(int side) { //разбиение квадрата, сторона которого кратна трём
	try
	{
		abscissa[1] = abscissa[2] = abscissa[3] = ordinate[3] = ordinate[5] = ordinate[4] = side;
		abscissa[5] = ordinate[2] = side * 1 / 2;
		abscissa[4] = ordinate[1] = 0;
		for (int i = 1; i < 6; i++) {
			length[i] = size * 1 / 3;
			insert_square(abscissa[i], ordinate[i], i + 1, length[i]);
		}
		return true;
	}
	catch (...)
	{
		return false;
	}

}

int Square::find_max_size(int x, int y) { //находит максимальное значение стороны квадрата, который возможно поместить на поле
	int  max_size;
	bool allowed = true;
	for (max_size = 1; allowed && max_size <= size - x && max_size <= size - y; max_size++) //проверка на пересечение границ квадрата
		for (int i = 0; i < max_size; i++)
			for (int j = 0; j < max_size; j++)
				if (coloring[x + i][y + j]) { //проверка на пересечение с уже существующим квадратом
					allowed = false;
					max_size--;
				}
	max_size--;
	return max_size;
}

bool Square::multiple_of_five(int side) {  //разбиение квадрата, сторона которого кратна пяти
	try
	{
		abscissa[1] = abscissa[2] = abscissa[7] = ordinate[4] = ordinate[5] = ordinate[7] = side;
		abscissa[5] = abscissa[6] = ordinate[2] = ordinate[3] = size * 2 / 5;
		length[2] = length[3] = length[5] = length[6] = side * 1 / 3;
		length[1] = length[4] = length[7] = side * 2 / 3;
		abscissa[3] = ordinate[6] = size * 4 / 5;
		abscissa[4] = ordinate[1] = 0;
		for (int i = 1; i < 8; i++) {
			insert_square(abscissa[i], ordinate[i], i + 1, length[i]);
		}
		return true;
```

```
	}
	catch (...)
	{
		return false;
	}
}

bool Square::insert_the_second_square() {  //вставляет второй квадрат слева от 1 на поле так, чтобы его сторона была максимально возможной
	try
	{
		abscissa[num] = length[0];
		ordinate[num] = 0;
		length[num] = find_max_size(abscissa[num], ordinate[num]);
		insert_square(abscissa[num], ordinate[num], num + 1, length[num]);
		num++;
		return true;
	}
	catch (...)
	{
		return false;
	}
}

bool Square::insert_the_third_square() {  //вставляет третий квадрат снизу от 1 на поле так, чтобы его сторона была максимально возможной
	try
	{
		abscissa[num] = 0;
		ordinate[num] = length[0];
		length[num] = find_max_size(abscissa[num], ordinate[num]);
		insert_square(abscissa[num], ordinate[num], num + 1, length[num]);
		num++;
		return true;
	}
	catch (...)
	{
		return false;
	}
}

bool Square::even_square(int side) {  //разбиение квадрата, сторона которого кратна двум
	try
	{
		abscissa[1] = abscissa[3] = ordinate[2] = ordinate[3] = size * 1 / 2;
		abscissa[2] = ordinate[1] = 0;
		for (int i = 0; i < 4; i++) {
			length[i] = size * 1 / 2;
			insert_square(abscissa[i], ordinate[i], i + 1, length[i]);
		}
		return true;
	}
	catch (...)
	{
		return false;
	}

}

void Square::print_square(int deep) { //вспомогательная функция, которая выводит получившийся квадрат
	for (int i = 0; i < size; i++) {
		for (int f = 0; f < deep - 3; f++) std::cout << "\t";
		for (int j = 0; j < size; j++)
			std::cout << coloring[i][j] << ' ';
```

```
			std::cout << std::endl;
		}
		std::cout << std::endl;
}

void Square::output_of_the_result(int amount) {  //выводит результат работы программы на консоль
		std::cout <<"Минимальное число квадратов для разбиения "<< amount << std::endl;//количество квадратов
		std::cout << amount << std::endl;
		for (int i = 0; i < amount; i++)
			std::cout << abscissa[i] + 1 << " " << ordinate[i] + 1 << " " << length[i] << std::endl;
		for (int j = 3; j < amount; j++) insert_square(abscissa[j], ordinate[j], j + 1, length[j]);
		std::cout << "Матричное представление результата" << std::endl;
		print_square(3);
}

int Square::insert_the_first_square() {  //функция вставки самого первого квадрата
		abscissa[num] = ordinate[num] = 0;
		if (size % 2 == 0) {
			std::cout << "Квадрат с четной стороной. Частный случай. Делим квадрат на 4 части." << std::endl;
			insert_square(0,0,1, size * 1 / 2);
			even_square(size * 1 / 2);  //квадраты со стороной 1/2 от длины
			return 4;
		}
		if (size % 3 == 0) {                  //если сторона квадрата кратна 3-м
			std::cout << "Квадрат со стороной кратной 3. Частный случай. Делим квадрат на 6 частей." << std::endl;
			length[num] = size * 2 / 3;
			insert_square(0, 0, 1, length[num]);
			multiple_of_three(length[num]); //рисуем 6 квадратов со сторонами 2/3 и 5 по 1/3 от длины
			return 6;
		}
		if (size % 5 == 0) {                  //если сторона квадрата кратна 5-ти
			std::cout << "Квадрат со стороной кратной 5. Частный случай. Делим квадрат на 8 частей." << std::endl;
			length[num] = size * 3 / 5;
			insert_square(0, 0, 1, length[num]);
			multiple_of_five(length[num]); //рисуем 8 квадратов со сторонами 3/5 3 по 2/5 и 4 по 1/5 от длины
			return 8;
		}
		else {                                //все остальные случаи
			std::cout << "Общий случай. Используем бэктрекинг." << std::endl;
			length[num] = size * 1 / 2 + 1;
			insert_square(abscissa[num], ordinate[num], num + 1, length[num]);//вставляем квадрат со стороной больше половины
			num++;
			insert_the_second_square();//второй квадрат с максимальной длиной стороны
			insert_the_third_square();//и третий
			std::cout << "Построили первые три квадрата и запустили рекурсивную функцию." << std::endl;
			print_square(3);
			return backtracking(4);
		}
}

int Square::backtracking(int deep) {//перебирает всевозможные варианты кадрирования неразбитого участка поля
		if (f && deep > num) //bool f-последний квадрат              //если текущее разбиение больше предыдущего
```

```
		return deep;
	int min_result = size * size;
	int temp_length;
	int temp_result;
	int temp_x;
	int temp_y;
	if (!place_to_insert(temp_x, temp_y)) { //если нет места для вставки
		for (int i = 0; i < deep - 3; i++) std::cout << "\t";
		std::cout << "Больше нет места для вставки. Текущее разбиение " << deep-1
<< std::endl;

		if (deep < min_result) {
			for (int i = 0; i < deep - 3; i++) std::cout << "\t";
			std::cout << "Получили новый меньший результат " << deep-1 <<
std::endl;
		}

		if (!f || (f && deep - 1 < num))
			num = deep - 1;//номер последнего квадрата
		f = true;//последний квадрат
		return num;//количество квадратов
	}
	for (temp_length = find_max_size(temp_x, temp_y); temp_length > 0; temp_length--)
{//длина-максимальная для вставки
		for (int i = 0; i < deep - 3; i++) std::cout << "\t";
		std::cout << "Вставляем новый квадрат по координатам " << temp_x + 1 << ' '
<< temp_y + 1 <<" со стороной " << temp_length << " под номером " << deep << std::endl;
		insert_square(temp_x, temp_y, deep, temp_length);//вставляем новый квадрат
		print_square(deep);
		temp_result = backtracking(deep + 1);//рекурсия-вставляем квадраты дальше
		min_result = min_result < temp_result ? min_result : temp_result;//если
промежуточный результат меньше, то заменяем

		if (temp_result <= num) { //если текущий результат не превосходит номер
текущего квадрата (получили меньший результат)
			for (int i = 0; i < deep - 3; i++) std::cout << "\t";
			std::cout << "Сохраним координаты и размер текущего квадрата" <<
std::endl;
			length[deep - 1] = temp_length;//сохраняем в массивы данные
			abscissa[deep - 1] = temp_x;
			ordinate[deep - 1] = temp_y;
		}
		for (int i = 0; i < deep - 3; i++) std::cout << "\t";
		std::cout << "Вернемся назад. Удалим последний квадрат." << std::endl;
		remove_square(temp_x, temp_y, temp_length);//удаляем квадрат
	}
	return min_result;
}

Square::Square(int size) : size(size) {//конструктор
	coloring = new int*[size];//массив квадрата заполнен 0
	for (int i = 0; i < size; i++) {
		coloring[i] = new int[size];
		for (int j = 0; j < size; j++)
			coloring[i][j] = 0;
	}
	abscissa = new int[N];
	ordinate = new int[N];
	length = new int[N];
	count = N;
	f = false;
	num = 0;
}

Square::~Square() {
```

```
	delete[] abscissa;
	delete[] ordinate;
	delete[] length;
	for (int i = 0; i < size; i++)
		delete[] coloring[i];
	delete[] coloring;
}

int main() {
	setlocale(LC_ALL, "Russian");
	int size, count;
	clock_t time;
	std::cout << "Введите размер стола:" << std::endl;
	while (std::cin >> size && size > 1) {
		time = clock();
		Square table(size);
		count = table.insert_the_first_square();
		table.output_of_the_result(count);
		time = clock() - time;
		std::cout << "Время работы программы в секундах  " << time / CLOCKS_PER_SEC
<< std::endl;
		std::cout << std::endl << "Введите размер стола:" << std::endl;
	}
	std::cout << "Неверный ввод" << std::endl;
}
```

Файл tests.cpp

```
#include "Square.h"
#include <boost/test/minimal.hpp>
#include <memory>

using namespace std;

class unitTest {
public:
    virtual void check() = 0;
};

class test : public unitTest {
public:
    void check() override {
        bool check = true;
        int size = 11;
        Square table(size);
        try {
            BOOST_REQUIRE(table.even_square(18) == true);
        }
        catch (...) {
            std::cout << "Test 1 not OK" << std::endl;
            check = false;
        }
        if (check) {
            std::cout << "Test 1 OK" << std::endl;
        }
        else {
            check = true;
        }

        try {
            BOOST_REQUIRE(table.insert_the_third_square() == true);
        }
        catch (...) {
```

```
            std::cout << "Test 2 not OK" << std::endl;
            check = false;
        }
        if (check) {
            std::cout << "Test 2 OK" << std::endl;
        }
        else {
            check = true;
        }

        try {
            BOOST_REQUIRE(table.insert_the_second_square() == true);
        }
        catch (...) {
            std::cout << "Test 3 not OK" << std::endl;
            check = false;
        }
        if (check) {
            std::cout << "Test 3 OK" << std::endl;
        }
        else {
            check = true;
        }

        try {
            BOOST_REQUIRE(table.multiple_of_five() == true);
        }
        catch (...) {
            std::cout << "Test 4 not OK" << std::endl;
            check = false;
        }
        if (check) {
            std::cout << "Test 4 OK" << std::endl;
        }
        else {
            check = true;
        }

        try {
            BOOST_REQUIRE(table.multiple_of_five(15) == true);
        }
        catch (...) {
            std::cout << "Test 5 not OK" << std::endl;
            check = false;
        }
        if (check) {
            std::cout << "Test 5 OK" << std::endl;
        }
    }

};
```