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

ОТЧЕТ

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

Студентка гр. 1304 _____ Ярусова Т. В.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить алгоритм поиска с возвратом. Написать программу, которая решает задачу разбиения квадратной площади на минимальное количество квадратов разного размера. Оптимизировать решение.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 11 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).

Выполнение работы.

Структура Square.

Данная структура отвечает за хранение оптимального разбиения квадратной доски. Поле *int min_count_square* создано для хранения количества квадратов в разбиении, поле *std::vector<std::vector<int>> square_matrix* создано для хранения самого разбиения, поле *int size* создано для хранения размера стороны доски.

Структура Answer.

Данная структура отвечает за хранение ответа в необходимом формате. Поле *int count_square* создано для хранения минимального количества квадратов в разбиении, *поле std::vector<std::vector<int>> coordinates_and_size_squares* создано для хранения координат левого верхнего угла одного из квадратов разбиения и его размер.

Структура Point.

Данная структура отвечает за хранение координат первой пустой (нулевой) ячейки в текущем разбиении. Поля *int x* и *int y* созданы для хранения координаты, а поле *bool is_empty* является флагом, который показывает, нашлась ли пустая ячейка.

Головная функция main().

Головная функция, в которой происходит только вызов функции *solution()*.

Функция solution().

В данной функции происходит считывание входных данных, т.е. размер стороны квадратной доски. Далее происходит выбор необходимого алгоритма для разбиения. Если сторона квадрата четная, то вызывается функция *even_size(),* если сторона кратна трем, то вызывается функция *multiple_of_three(),* во всех остальных случаях вызывается функция

find_splitting(). Для оптимизации алгоритма были написаны функции *even_size()* и *multiple_of_three()*, которые ищут частные решения.

Функция even_size().

Данная функция находит частное решение для квадрата с четной стороной. На вход принимает размер стороны. Было замечено, что все квадраты с четной стороной разбиваются на 4 равных квадрата, сторона которых равна *size / 2*. В связи с этими наблюдениями, в данной функции происходит заполнение поля *coordinates_and_size_square* объекта *answer* типа *Answer* по формуле *{y * size/2 + 1, x * size/2 + 1, size / 2}*. Данная функция возвращает переменную answer типа *Answer*.

Функция multiple_of_three().

Данная функция находит частное решение для квадрата со стороной кратной 3. На вход принимает размер стороны. Было замечено, что все квадраты со стороной кратной 3 разбиваются на 6 квадратов: на один большой квадрат, сторона которого равна (size /3) * 2, и на 5 поменьше, сторона которых равна size/3. В связи с этим наблюдением, в данной функции происходит заполнение поля *coordinates_and_size_square* объекта *answer* типа *Answer*, что является частным решением. Данная функция возвращает объект *answer* типа *Answer*.

Функция find_splitting().

Данная функция находит решения для доски с нечетной стороной. На вход получает размер стороны исходного квадрата. В функции создаются 2 объекта типа *Square*: *square*, в котором будет храниться конечное оптимальное разбиение, и *tmp_square*, в котором будет храниться текущее разбиение.

Вызывается функция *start_splitting()*, которая внесет в *square* начальное разбиение. Далее вызывается основная функция *backtracking()*, которая вернет конечное разбиение.

Данная функция возвращает объект типа *Answer*, который получается путем конвертации типа *Square* в тип *Answer* с помощью функции *covert-ing_answer()*.

Функция start_splitting().

Было замечено, что в любом оптимальном разбиении для стороны, являющейся нечетным простым числом, есть один квадрат размером *size / 2 + 1* и два квадрата размером *size / 2 – 1* смежных с первым квадратом. Исходя из наблюдений, данная функция делает начальное разбиение, «закрашивая» разными числами участок квадрата. На вход функция получает указатель на объект *square* типа *Square*. С помощью условных операторов происходит “закрашивание”. Схему начального разбиения для квадрата со стороной 7 (см. рис. 1).

```
7
1 1 1 1 2 2 2
1 1 1 1 2 2 2
1 1 1 1 2 2 2
1 1 1 1 0 0 0
3 3 3 0 0 0 0
3 3 3 0 0 0 0
3 3 3 0 0 0 0
```

Рисунок 1 – схема начального разбиения для квадрата со стороной 7

Функция backtracking().

Данная функция реализует рекурсивный алгоритм поиска с возвратом. На вход функции подаются переменные *square*,в которой хранится оптимальное разбиение, *tmp_sq*, в которой хранится текущее разбиение и *ordinal_num square*, в которой хранится порядковый номер вписываемого квадрата.

Создается объект *point_corner* типа *Point*. Вызывается функция *find_empty_coord()*, которая вернет координаты первой нулевой (не закрашенной) клетки. Далее происходит ряд проверок на заполненность схемы разбиения и на ее оптимальность. Если не нашлась пустая ячейка и в *tmp_sq* хранится минимальное разбиение, тогда вернется *tmp_sq*. В случае, если пустая

ячейка все-таки нашлась, то происходит проверка на оптимальность решения. Если порядковый номер равен минимальному количеству квадратов в *square*, то вернется *square*, т.к. дальше нет смысла смотреть.

После нахождения первой незаполненной ячейки необходимо установить максимальный размер квадрата, который можно вписать в незаполненную зону. Вызывается функция *find_min_side()*, которая вернет максимальный размер возможного квадрата, который можно вписать.

Далее с помощью тройного цикла *for* и вызова функции *backtracking()* совершается полный перебор всех возможных вариантов.

Функция find_empty_coord().

Данная функция ищет первую незаполненную ячейку в разбиении. На вход получает объект *tmp_sq* типа *Square*, в котором хранится текущее разбиение. В функции совершается проход по матрице, в случае, если нулевая точка найдена, то запоминаются ее координаты в переменную *point_corner* типа *Point*. Функция возвращает *point_corner.*

Функция find_min_side().

Данная функция ищет максимальный размер квадрата, который можно вписать в разбиение. На вход принимает объект *point_corner* типа *Point*, в котором хранятся координаты нулевой точки, и объект *tmp_sq* типа *Square*, в котором хранится текущее разбиение.

В функции ищется количество свободных клеток по x и по y от нулевой точки до правой и нижней границы соответственно. Из этих двух параметров выбирается минимальный и данное значение возвращается из функции.

Функция converting_answer().

Данная функция преобразует разбиение в необходимый формат ответа.

На вход функция получает объект *square* типа *Square*. Возвращает объект answer типа Answer.

Функция print_answer().

Данная функция осуществляет печать ответа в необходимом формате, хранящегося в объекте *answer* типа *Answer*.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1	10	4 1 1 5 1 6 5 6 1 5 6 6 5	Разбиение квадрата с четной стороной
2	15	6 1 1 10 11 1 5 1 11 5 11 6 5 6 11 5 11 11 5	Разбиение квадрата со стороной кратной 3
3	7	9 1 1 4 1 5 3 4 5 1 4 6 2 5 1 3 5 4 1 5 5 1 6 4 2 6 6 2	Разбиение квадрата с нечетной стороной
4	19	13 1 1 10 1 11 9 10 11 1 10 12 1 10 13 2 10 15 5 11 1 9 11 10 2 11 12 1 12 12 3	Разбиение квадрата нечетной стороной

		13 10 2 15 10 5 15 15 5	

Выводы.

Разработана программа на основе алгоритма поиска с возвратом, которая решает задачу оптимального разбиения квадратной площади.

Пройдено тестирование на платформе Stepik для квадратов размером 2 <= N <= 20. Для успешного прохождения тестов на данной платформе, реализован ряд оптимизаций:

1. Функция, являющаяся частным решением для квадрата с четным размером стороны.

2. Функция, являющаяся частным решением для квадрата со стороной кратной 3.

3. Создание начального разбиения квадратной площади для сокращения времени рекурсивного перебора.

4. Дополнительные условия выхода в рекурсивной функции осуществляющей поиск с возвратом.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: 1304_Yarusova_TV_lb1.cpp

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

#define EVEN_COUNT 4
#define FIRST_ORDINAL_NUM_SQUARE 4
#define THREE_COUNT 6

//Структура, отвечающая за хранение разбиения
typedef struct {
    int min_count_square;
    std::vector<std::vector<int>> square_matrix;
    int size;
} Square;

//Структура, отвечающая за хранение требуемого формата ответа
typedef struct {
    int count_square;
    std::vector<std::vector<int>> coordinates_and_size_squares;
} Answer;

//  Структура, отвечающая за хранение координат первой встретившийся пустой клетки
typedef struct {
    int x = 0;
    int y = 0;
     bool is_empty = false; //Флаг, отвечающий за то, нашлась ли пустая клетка
}Point;

//Функция, печатающая ответ в необходимом формате
//На вход принимает объект типа Answer
void print_answer(Answer answer) {
    std::cout << answer.count_square << std::endl;
                for (int i = 0; i < answer.coordinates_and_size_squares.size(); i++)
            std::cout << answer.coordinates_and_size_squares[i][0] << "  " << answer.coordinates_and_size_squares[i][1] << "  " << answer.coordinates_and_size_squares[i][2] << std::endl;
}

//Функция, конвертирующая данные из объекта типа Square
//в данные типа Answer
//На вход принимает объект типа Square, возвращает объект типа Answer
Answer converting_answer(Square square){
    Answer answer;
    answer.count_square = square.min_count_square;

    for(int y = 0; y < square.size; y++){
        for(int x = 0; x < square.size; x++){
            if(square.square_matrix[y][x] != 0){
```

```
                    int num = square.square_matrix[y][x];
                    int count = 0;
                    for(int i = x; i < square.size; i++){
                        if(square.square_matrix[y][i] == num)
                            count++;
                    }
                     answer.coordinates_and_size_squares.push_back({y
+ 1, x + 1, count});
                    for(int y1 = 0; y1 < square.size; y1++){
                        for(int x1 = 0; x1 < square.size; x1++){
                            if(square.square_matrix[y1][x1] == num)
                                square.square_matrix[y1][x1] = 0;
                        }
                    }
                }
            }
        }
        return answer;
    }

    //Функция, которая находит решение для доски с четной стороной
    //На вход получает размер доски, возвращает объект типа Answer
    Answer even_size(int size) {
        Answer answer;
        answer.count_square = EVEN_COUNT;
        for (int y = 0; y < answer.count_square / 2; y++) {
            for (int x = 0; x < answer.count_square / 2; x++) {
                 answer.coordinates_and_size_squares.push_back({ y *
size / 2 + 1, x * size / 2 + 1, size / 2 });
            }
        }
        return answer;
    }

    //Функция, которая ищет координаты первой встретившейся пустой
ячейки
    //На вход получает объект типа Square
    //Возвращает объект типа Point, в котором хранится координата
первой встретившейся пустой ячеейки
    Point find_empty_coord(Square tmp_sq){
        Point point_corner;
        for (int y = tmp_sq.size / 2; y < tmp_sq.size; y++) {
            for (int x = tmp_sq.size / 2; x < tmp_sq.size; x++) {
                if (tmp_sq.square_matrix[y][x] == 0) {
                    point_corner.x = x;
                    point_corner.y = y;
                    point_corner.is_empty = true;
                    break;
                }
            }
            if (point_corner.is_empty)break;
        }
        return point_corner;
    }

    //Функция, которая ищет минимальную сторону по x и по y.
    //Данная сторона будет максимальной стороной квадрата, который
можно вписать в разбиение
```

```
    //Принимает на вход  объект типа Point и объект типа Square
    //Возвраащает целочисленное значение - максимально возможную сторону квадрата, который можно вписать
    int find_max_side(Point point_corner, Square tmp_sq){
        int x_side = 0;
        int y_side = 0;
        for (int y = point_corner.y; y < tmp_sq.size; y++) {
            if (tmp_sq.square_matrix[y][point_corner.x] == 0)
                y_side++;
            else
                break;
        }
        for (int x = point_corner.x; x < tmp_sq.size; x++) {
            if (tmp_sq.square_matrix[point_corner.y][x] == 0)
                x_side++;
            else
                break;
        }
        return std::min(x_side, y_side);
    }

    //Рекурсивная функция, которая ищет оптимальное разбиение доски
    //На вход принимает объект типа Square, который отвечает за текущее оптимальное разбиение,
    //объект типа Square, который отвечает за текущее разбиение,
    //и целочисленное значение - порядковый номер вставляемого квадрата
    //Возвращает объект типа Square
    Square backtracking(Square square, Square tmp_sq, int ordinal_num_square) {
        Point point_corner = find_empty_coord(tmp_sq);

        if (!point_corner.is_empty) {
            tmp_sq.min_count_square = ordinal_num_square - 1;
             return square.min_count_square < tmp_sq.min_count_square ? square : tmp_sq;
        }

          if (ordinal_num_square == square.min_count_square) return square;


        int min_side = find_max_side(point_corner, tmp_sq);
           for (int size_square = 1; size_square <= min_side; size_square++) {
            for (int y = 0; y < size_square; y++) {
                for (int x = 0; x < size_square; x++) {
                                tmp_sq.square_matrix[point_corner.y + y][point_corner.x + x] = ordinal_num_square;
                }
            }
             square = backtracking(square, tmp_sq, ordinal_num_square + 1);
        }
        return square;
    }

    //Функция, создающее начальное разбиение "раскраску"
```

```
    //На вход принимает указатель на объект типа Square
    void start_splitting(Square* square){
        square->min_count_square = square->size * square->size;
        for (int y = 0; y < square->size; y++) {
            square->square_matrix.push_back(std::vector<int>());
            for (int x = 0; x < square->size; x++) {
                 if ((y < (square->size + 1) / 2) && (x < (square-
>size + 1) / 2))
                   square->square_matrix[y].push_back(1);
                 else if ((y < square->size / 2) && (x >= (square-
>size + 1) / 2))
                   square->square_matrix[y].push_back(2);
                  else if ((y >= (square->size + 1) / 2) && (x <
square->size / 2))
                   square->square_matrix[y].push_back(3);
                else
                    square->square_matrix[y].push_back(0);
            }
        }

    }

    //Функция, которая находит решение для доски с нечетной стороной
    //На вход получает размер доски, возвращает объект типа Answer
    Answer find_splitting(int size) {

        Square square;
        square.size = size;
        start_splitting(&square);

        Square tmp_square;
        tmp_square = square;
                square     =     backtracking(square,     tmp_square,
FIRST_ORDINAL_NUM_SQUARE);

        return converting_answer(square);
    }

    //Функция частного решения для доски со стороной кратной 3
    //На вход принимает целочисленное значение - сторону доски
    //Возвращает объект типа Answer
    Answer multiple_of_three(int size){
        Answer answer;
        answer.count_square = THREE_COUNT;
        int formul = (size / 3) * 2;

answer.coordinates_and_size_squares.push_back(std::vector<int>({1,   1,
formul}));

answer.coordinates_and_size_squares.push_back(std::vector<int>({formul
+ 1, 1, formul / 2}));

answer.coordinates_and_size_squares.push_back(std::vector<int>({1,
formul + 1, formul / 2}));

answer.coordinates_and_size_squares.push_back(std::vector<int>({formul
+ 1, formul / 2 +1 , formul / 2}));
```

```
answer.coordinates_and_size_squares.push_back(std::vector<int>({formul
/ 2 + 1, formul + 1, formul / 2}));

answer.coordinates_and_size_squares.push_back(std::vector<int>({formul
+ 1,formul + 1, formul / 2}));
        return answer;
    }

    //Функция, в который определяется случай решения и вызывается
необходимая функция
    void solution() {
        int size_square;
        std::cin >> size_square;
        if (size_square % 2 == 0)
            print_answer(even_size(size_square));
        else if((size_square % 3 == 0) && (size_square > 3))
            print_answer(multiple_of_three(size_square));
        else
            print_answer(find_splitting(size_square));

    }

    //Головная функция
    int main() {
        solution();
    }
```