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. 9382 _____ Русинов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

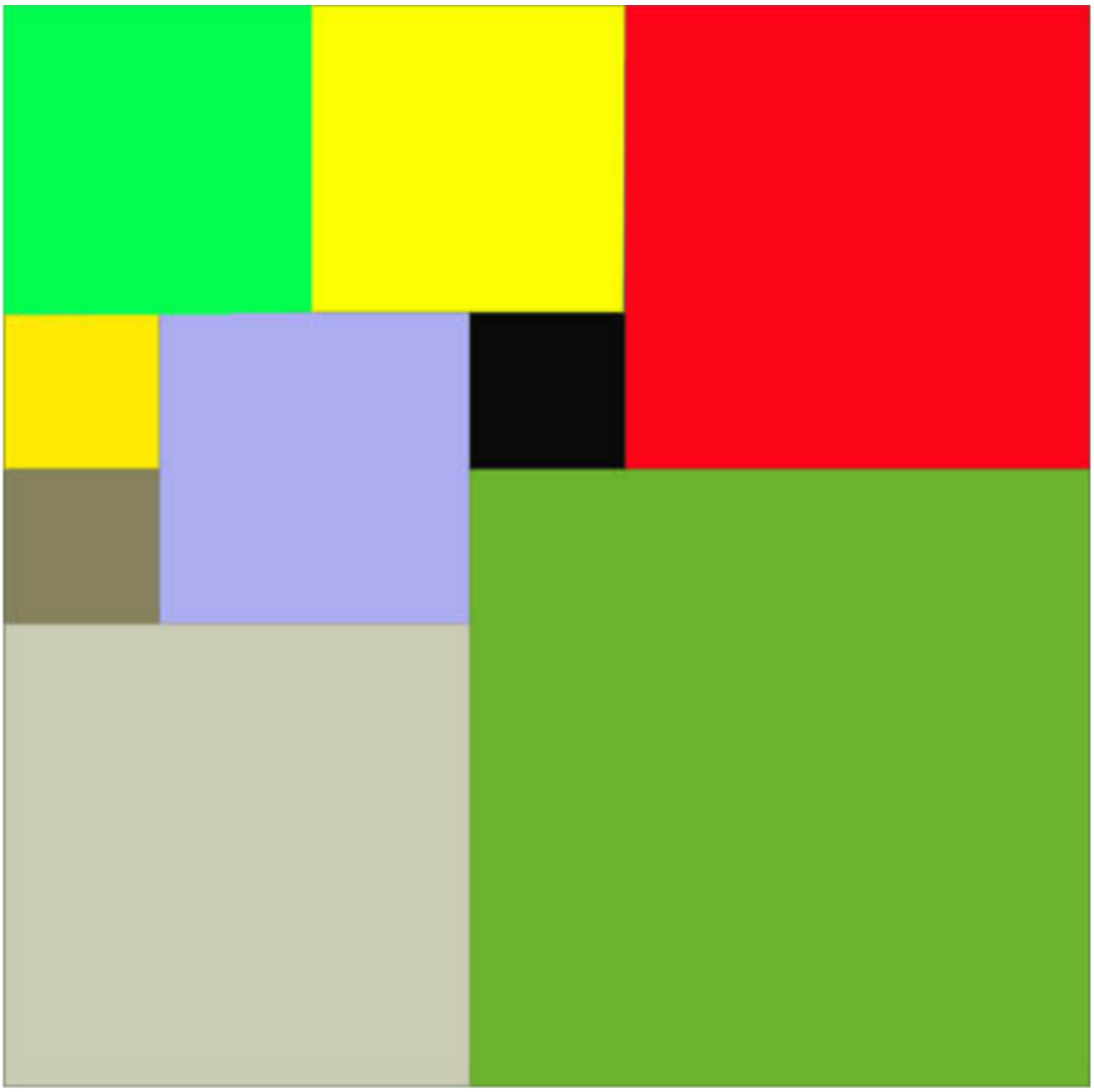
Цель работы.

Получить представление о решении NP — полных задач, изучить такой метод решения, как поиск с возвратом, проследить зависимость количества операций для решения поставленной задачи от входных данных.

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 7×7 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 ($2 \le N \le 20$).

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 ($1 \le x, y \le N$) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вариант 1р. Рекурсивный бэктрекинг. Выполнение на Stepik всех трёх заданий в разделе 2.

Описание алгоритма.

Первоначально необходимо завести две матрицы размера N*N, где N – размер данного квадрата, обе матрицы заполняются нулями. В первой матрице будет происходить перебор решений для квадрата, а во второй будет содержаться оптимальное решение для данного квадрата. При этом в ходе работы алгоритма, матрица, в которой происходит перебор решений, может стать оптимальной.

После создания двух матриц, необходимо инициализировать первоначальные значения в них:

1) В оптимальную матрицу вставляется квадрат размера N-1, затем в матрице остается пространство только для единичных квадратов, поэтому они вставляются до тех пор, пока матрица не будет полностью заполнена.
2) В матрицу, в которой происходит перебор решений, вставляется 3 квадрата, которые занимают 75% площади квадрата. Поэтому перебор квадратов будет происходить в оставшихся 25% площади квадрата.

Далее начинается перебор размера квадрата, который можно поставить по координатам с пустой ячейкой. Перебор происходит от большего размера к меньшему. Если квадрат возможно поставить, то происходит его вставка. Затем выполняется поиск свободной клетки в матрице перебора. Если же свободная клетка есть, то необходимо перебрать и для этой клетки квадрат, который можно поставить. Таким образом, формируется рекурсивная функция. Если же свободных ячеек в матрице более нет, то квадрат считается заполненным. Необходимо сравнить количество квадратов, которое находится в текущей матрице и в оптимальной матрице. Если количество квадратов в оптимальной матрице больше, чем в текущей, то текущая матрица становится оптимальной. Так как квадрат является заполненным, то более не будет вызова рекурсивной функции. В таком случае будет произведено удаление последнего поставленного квадрата в матрице перебора и выход из функции. Также при входе в рекурсивную функцию стоит проверить текущее количество квадратов в матрице перебора и сравнить с количеством квадратов в оптимальной матрице. Если

квадратов в оптимальной матрице уже меньше, чем в матрице перебора, то далее перебирать квадраты не имеет смысла, а стоит прервать эту ветку перебора. После завершения перебора выводится на экран результат оптимальной матрицы.

Использованные оптимизации.

1) Матрицу перебора изначально можно заполнить на 75% тремя квадратами.
2) Поскольку матрица перебора заполнена на 75%, то поиск свободной клетки, куда можно поставить квадрат, можно осуществлять только в оставшихся 25% квадрата.
3) Квадрат с четной стороной имеет постоянное решение – 4 квадрата. Поэтому можно не осуществлять перебор для таких квадратов, а сразу выводить ответ.
4) Сжатие квадрата. Квадрат с размером N, можно сжать до размера значения наименьшего простого делителя числа N. Например, квадрат размера 15 можно сжать до размера 3 и осуществлять перебор для квадрата размера 3. Результат количества квадратов будет одинаков.
5) Проверка количества квадратов в текущей матрице перебора и в оптимальной матрице. Если в текущей матрице перебора квадратов уже больше, чем в оптимальной матрице, то продолжать перебор нет смысла для текущей расстановки квадратов, лучшего решения уже не добиться.
6) Поскольку 75% квадрата заполнены, то максимальный размер квадрата, который можно поставить в матрицу перебора – N // 2.

Описание рекурсивной функции.

void _solve(int x, int y);

Данный метод находится в классе SquareSolver. Метод принимает координаты x и y, где расположена пустая ячейка в матрице перебора, и начинает перебор квадратов от большего размера к меньшему. Если квадрат удалось поставить, и есть еще свободное пространство в матрице перебора, то

происходит вызов рекурсивного метода для найденной пустой ячейки. Для перебора квадратов, которые можно поставить, используется цикл от N // 2 до 0. Если квадрат удалось поставить, то после выполнения последующей необходимой логики, этот квадрат необходимо удалить для дальнейшего перебора квадратов. Данный метод не имеет возвращаемого значения, поскольку экземпляр класса SquareSolver имеет поля _currentMap и _optimalMap. Результат работы метода будет записан в поле _optimalMap.

Описание способа хранения частичных решений.

Для хранения частичных решений и удобной работы был реализован класс SquareMap.

Он содержит следующие поля:

1) int _size – размер квадрата.
2) int _compression – сжатие квадрата.
3) int _countSquares – количество квадратов в карте.
4) std::vector<std::vector<int>> _array – двумерный массив, который содержит в себе расстановку квадратов. Массив размера _size * _size.

Таким образом, при переборе решений используется в соответствии с алгоритмом два экземпляра SquareMap – карта перебора и оптимальная карта. В карте перебора происходит вставка/удаление квадратов, а если в ней становится меньше квадратов, чем в оптимальной карте, то оптимальной картой становится текущая карта перебора. Оптимальная карта и карта перебора хранятся в экземпляре класса SquareSolver.

Описание функций и структур данных.

Данный класс предназначен для вывода промежуточных результатов.

class MessagePrinter:

1) static void _printMsgWithRecursionLevel(const std::string& message) – статический метод для печати сообщения с отступом размера текущего уровня рекурсии.
2) static void optimalLessCurrent(int x, int y, int countInOptimal, int countInCurrent) – данный метод печатает сообщение, меньше ли квадратов в оптимальной карте, чем в текущей. Принимает координаты, из которой был вызван метод, количество квадратов в картах.
3) static void enterInRecursion(int x, int y) – данный метод печатает сообщение о вызове рекурсивного метода из координат x и y.
4) static void tryToSetSquare(int x, int y, int size, bool canSetSq) – данный метод печатает сообщение, удалось ли поставить квадрат по координатам x, y размера size. Параметр canSetSq отвечает за то, удалось поставить или нет.
5) static void isThereEmpty(int x, int y, bool isThereEmpty, int newX = 0, int newY = 0) – данный метод печатает сообщение, удалось ли найти свободную клетку из координат x, y. Если удалось, то в аргументе isThereEmpty будет передан true, а координаты свободной клетки будут находиться в аргументах newX и newY.
6) static void currentLessOptimal(int x, int y, bool isLess, int countInCurrent, int countInOptimal) – данный метод вызывается в случае, когда текущий квадрат уже заполнен. Он печатает сообщение, меньше ли квадратов в оптимальной карте, чем в текущей. Метод принимает координаты, если в текущей карте перебора меньше квадратов, чем в оптимальной, то в аргумент isLess будет передан true, также передаются значения, сколько квадратов в картах перебора и текущей.

7) static void removeSquare(int x, int y, int size) – данный метод печатает сообщение об удалении квадрата по координатам x и y размера size.

Данный класс предназначен для удобства при переборе.

class SquareMap:

1) int _size – размер карты.
2) int _compression – сжатие карты.
3) int _countSquares – количество квадратов в карте.
4) std::vector<std::vector<int>> _array - двумерный массив, который содержит расстановку квадратов.
5) SquareMap(int size, int compression) – конструктор данного класса. Принимает размер и сжатие квадрата, который нужно будет перебрать. Создает массив _array.
6) int countSquares() const – геттер для поля _countSquares.
7) void insertSquare(int x, int y, int size) – данный метод предназначен для вставки квадрата по координатам x и y размера size в _array.
8) void removeSquare(int x, int y, int size) – данный метод предназначен для удаления квадрата по координатам x и y размера size в _array.
9) bool canSetSquare(int x, int y, int size) – данный метод предназначен для проверки, можно ли вставить квадрат по координатам x и y размера size. Возвращает булево значение.
10) bool isThereEmpty(int& x, int& y) – данный метод предназначен для поиска, есть ли свободная клетка в карте. Принимает ссылки на координаты, с которых нужно начать проверку. Возвращает булево значение.
11) Explicit operator std::string() – данный метод предназначен для преобразования карты в строку. Строка генерируется в формате, которое представлено в задании.

12) void print(const std::string& innerMessage) – данный метод предназначен для визуализации карты. Принимает сообщение, которое нужно распечатать перед выводом карты.

Класс для решения поставленной задачи.

class SquareSolver:

1) int _size – размер квадрата.
2) int _compression – сжатие квадрата.
3) SquareMap* _currentMap – текущая карта перебора
4) SquareMap* _optimalMap – оптимальная карта.
5) static std::pair<int, int> _doCompression(int size) – метод для сжатия квадрата размера size. Возвращает новый размер квадрата и его сжатие.
6) void _initOptimalMap() – метод для инициализации оптимальной карты.
7) void _initCurrentMap() – метод для инициализации текущей карты перебора.
8) void _solveEvenSquare() – метод для создания решения квадрата с четной стороной.
9) void _solve(int x, int y) – метод для решения поставленной задачи, данный метод рекурсивный, принимает координаты, с которых начнется перебор. Более подробное описание представлено в разделе “Описание рекурсивной функции”.
10) explicit SquareSolver(int size) – конструктор. Делает сжатие и инициализирует поля.
11) SquareMap* solve() – публичный метод решения задачи. Возвращает оптимальную карту.

Оценка сложности по времени.

Поскольку используется довольно большое количество оптимизаций, то дать точную оценку сложности алгоритма – трудоемкая задача. Было принято решение дать алгоритму верхнюю границу того, сколько квадратов он переберет.

N – размер квадрата. Есть N^2 свободных клеток и N размеров квадрата, которые будут перебираться.

Ставим первый квадрат, его можно поставить N^2 * N способами.

Ставим второй квадрат, его можно поставить (N^2-1) * N способами.

И так дойдем до последней клетки. Получаем O((N^2)! * N^N).

Оценка сложности алгоритма по памяти.

Всего используется два матрица, которые содержат текущее решение и оптимальное решение, поэтому сложность по памяти – O(2 * N^2), где N – размер квадрата.

Тестирование.

Тестирование проведено с помощью системы Stepik. Также результаты представлены в таблице ниже.

Таблица 1. Результаты работы программы

Входные данные	Выходные данные без промежуточного вывода
3	6 1 1 2 3 1 1 3 2 1 1 3 1 2 3 1 3 3 1
5	8

	1 1 3 4 1 2 4 3 2 1 4 2 3 4 1 3 5 1 4 5 1 5 5 1
7	9 1 1 4 5 1 3 5 4 2 7 4 1 1 5 3 4 5 1 7 5 1 4 6 2 6 6 2
9	6 1 1 6 7 1 3 7 4 3 1 7 3 4 7 3 7 7 3
11	11 1 1 6 7 1 5 7 6 3 10 6 2 1 7 5 6 7 1 6 8 1

	10 8 1 11 8 1 6 9 3 9 9 3
12	4 1 1 6 7 1 6 1 7 6 7 7 6
37	15 1 1 19 20 1 18 20 19 2 22 19 5 27 19 11 1 20 18 19 20 1 19 21 3 19 24 8 27 30 3 30 30 8 19 32 6 25 32 1 26 32 1 25 33 5

Выводы.

Был реализован поиск с возвратом для поиска минимального количества непересекающихся квадратов, заполняющих исходный квадрат. Была написана рекурсивная функция, которая выполняет поставленную задачу.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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


// Данная переменная отвечает за уровень рекурсии в данный момент времени
int RECURSION_LEVEL = 0;


class MessagePrinter {

    // Класс предназначен для вывода промежуточных данных
    // Чтобы избавиться от некрасивого кода в алгоритме решения задачи

    static void _printMsgWithRecursionLevel(const std::string& message) {
        std::string recursionLevelString = std::string();
        for (int i = 0; i < RECURSION_LEVEL; ++i) recursionLevelString +=
"_";
        std::cout << recursionLevelString << message;
    }

public:

    static void optimalLessCurrent(int x, int y, int countInOptimal, int
countInCurrent) {
        std::string msg = "Кол-во квадратов оптимальной карты [" +
std::to_string(countInOptimal) + "] ";
        msg += "<=, чем кол-во квадратов карты перебора [" +
std::to_string(countInCurrent) + "]";
        msg += ", далее смысла перебирать нет.\n";
        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }

    static void enterInRecursion(int x, int y) {
        std::string msg = "Вход в рекурсию\n";
        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }

    static void tryToSetSquare(int x, int y, int size, bool canSetSq) {
        std::string msg = "Попытка поставить квадрат размера " +
std::to_string(size);
        if (canSetSq) msg += " была удачной\n";
        else msg += " была неудачной\n";
        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }
```

```
    static void isThereEmpty(int x, int y, bool isThereEmpty, int newX =
0, int newY = 0) {
        std::string msg = "В квадрате";
        if (isThereEmpty)
            msg += " есть пустая клетка по координатам (" +
std::to_string(newX) + "; " + std::to_string(newY) + ")\n";
        else
            msg += " нет пустых клеток. Он полностью заполнен\n";

        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }

    static void currentLessOptimal(int x, int y, bool isLess, int
countInCurrent, int countInOptimal) {
        std::string msg = "Кол-во квадратов в карте перебора [" +
std::to_string(countInCurrent) + "]";
        if (isLess) {
            msg += " <, чем в оптимальной карте [" + std::to_string(coun-
tInOptimal) + "].";
            msg += " Поэтому текущая карта перебора становится оптималь-
ной.\n";
        }
        else msg += " >=, чем в оптимальной карте [" +
std::to_string(countInOptimal) + "].\n";
        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }

    static void removeSquare(int x, int y, int size) {
        std::string msg = "Был удален квадрат размера [" +
std::to_string(size) + "] ";
        msg += "для дальнейшего перебора.\n";
        msg = "[" + std::to_string(x) + "; " + std::to_string(y) + "] " +
msg;
        _printMsgWithRecursionLevel(msg);
    }
};


class SquareMap {
    int _size;
    int _compression;
    int _countSquares = 0;
    std::vector<std::vector<int>> _array;

public:
    SquareMap(int size, int compression) : _size(size), _compression(com-
pression) {
        _array.resize(size);
        for (int i = 0; i < size; ++i) _array[i].resize(size, 0);
    }

    int countSquares() const {
        return _countSquares;
```

```
    }

    void insertSquare(int x, int y, int size) {

        // Метод вставки квадрата в карту
        // по координатам x и y размера size

        _countSquares++;
        for (int i = y; i < y + size; ++i) {
            for (int j = x; j < x + size; ++j)
                _array[i][j] = size;
        }
    }

    void removeSquare(int x, int y, int size) {

        // Метод удаления квадрата из карты
        // по координатам x и y размера size

        _countSquares -= 1;
        for (int i = y; i < y + size; ++i) {
            for (int j = x; j < x + size; ++j)
                _array[i][j] = 0;
        }
    }

    bool canSetSquare(int x, int y, int size) {

        // Метод проверки, можно ли вставить квадрат размера size в карту
        // по координатам x и y

        if (x + size > _size || y + size > _size)
            return false;

        for (int i = y; i < y + size; ++i) {
            for (int j = x; j < x + size; ++j)
                if (_array[i][j]) return false;
        }

        return true;
    }

    bool isThereEmpty(int& x, int& y) {

        // Метод проверки, есть ли пустое пространство в квадрате
        // Поскольку 75 % площади квадрата уже заняты изначально
        // то есть смысл искать пустое пространство только в 25% площади
всего квадрата

        while (_array[y][x]) {
            if (x == _size - 1) {
                if (y == _size - 1) return false;
                x = _size / 2;
                y++;
                continue;
            }
            x++;
        }
```

```
        return true;
    }

    explicit operator std::string() {

        // Метод преобразования карты в тип std::string

        auto text = std::string();
        text.append(std::to_string(_countSquares) + "\n");
        for (int y = 0; y < _size; ++y) {
            for (int x = 0; x < _size; ++x) {
                if (_array[y][x]) {
                    // координаты и размер домножаются на коэф. сжатия,
чтобы получить
                    // их верное значение
                    auto size = std::to_string(_array[y][x] * _compres-
sion);
                    auto transformedX = std::to_string(x * _compression +
1);
                    auto transformedY = std::to_string(y * _compression +
1);
                    auto row = std::string();
                    row.append(transformedX + " ");
                    row.append(transformedY + " ");
                    row.append(size + "\n");
                    text.append(row);
                    removeSquare(x, y, _array[y][x]);
                }
            }
        }
        return text;
    }

    void print(const std::string& innerMessage) {

        std::cout << "*******" << innerMessage <<  "*******" <<
std::endl;

        // вывод элементов матрицы

        for (int i = 0; i < _size; ++i) {
            for (int j = 0; j < _size; ++j)
                std::cout << _array[i][j] << " ";
            std::cout << std::endl;
        }

        auto closeMsg = std::string();
        // формирование закрывающего сообщения
        for (int i = 0; i < 14 + innerMessage.length(); ++i) closeMsg +=
"*";

        std::cout << closeMsg << std::endl;

    }
};


class SquareSolver {
```

```
    int _size;
    int _compression;
    SquareMap* _currentMap;
    SquareMap* _optimalMap;

    static std::pair<int, int> _doCompression(int size) {

        // Метод поиска наименьшего общего делителя в размере квадрата
        // Это одна из оптимизаций

        int compression = 1;
        int compressedSize = size;

        for (int delimiter = size / 2; delimiter > 1; --delimiter) {
            // перебор делителей от большего к меньшему
            if (!(size % delimiter)) {
                compression = delimiter;
                compressedSize = size / delimiter;
                // если нашли делитель, то он будет максимальным
                break;
            }
        }

        std::cout << "Было выполненино сжатие размера квадрата" <<
std::endl;
        std::cout << "Текущий размер квадрата - " << std::to_string(com-
pressedSize) << std::endl;
        std::cout << "Коэффициент сжатия - " << std::to_string(compres-
sion) << std::endl;

        return {compressedSize, compression};
    }

    void _initOptimalMap() {

        // Метод инициализации первоначальной оптимальной карты
        // Ставим квадрат N-1 размера и окружаем его квадратами единич-
ного размера

        std::cout << "Инициализация оптимальной карты" << std::endl;

        _optimalMap->insertSquare(0, 0, _size - 1);
        // окружаем квадрат N-1 единичными квадратами по правой части
        for (int y = 0; y < _size; ++y)
            _optimalMap->insertSquare(_size - 1, y, 1);
        // окружаем квадрат N-1 единичными квадратами по нижней части
        for (int x = 0; x < _size - 1; ++x)
            _optimalMap->insertSquare(x, _size - 1, 1);

        std::cout << "Начальная оптимальная карта имеет " << _optimalMap-
>countSquares() << " квадратов\n";

    }

    void _initCurrentMap() {

        // Метод инициализации карты перебора
        // Используется оптимизация, что 75% площади квадрата можно сразу
```

```
покрыть 3-мя квадратами

        std::cout << "Инициализация карты для перебора" << std::endl;

        _currentMap->insertSquare(0, 0, _size / 2 + 1);
        _currentMap->insertSquare(_size / 2 + 1, 0, _size / 2);
        _currentMap->insertSquare(0, _size / 2 + 1, _size / 2);

        std::cout << "Карта для перебора заполнена на 75%" << std::endl;

    }

    void _solveEvenSquare() {

        // Квадрат с четной стороной имеет заранее определенное значение
        // Его сразу можно покрыть 4-мя квадратами

        std::cout << "Квадрат имеет четную сторону, поэтому его оптималь-
ное решение - 4 квадрата" << std::endl;

        _optimalMap->insertSquare(0, 0, 1);
        _optimalMap->insertSquare(0, 1, 1);
        _optimalMap->insertSquare(1, 0, 1);
        _optimalMap->insertSquare(1, 1, 1);
    }

    void _solve(int x, int y) {

        // Рекурсивная функция перебора квадратов в карте перебора

        // Если кол-во квадратов в оптимальной карте уже меньше, чем в
карте перебора
        // То дальше перебор квадратов не имеет смысла и можно откинуть
эту ветку

        if (_optimalMap->countSquares() <= _currentMap->countSquares()) {
            MessagePrinter::optimalLessCurrent(x, y, _optimalMap-
>countSquares(), _currentMap->countSquares());
            return;
        }

        MessagePrinter::enterInRecursion(x, y);

        // Перебор размера квадрата, который будет поставлен по координа-
там X и Y от большего к меньшему

        for (int size = _size / 2; size > 0; --size) {
            if (_currentMap->canSetSquare(x, y, size)) {
                // если можем поставить квадрат, ставим его
                _currentMap->insertSquare(x, y, size);

                MessagePrinter::tryToSetSquare(x, y, size, true);

                int copyX = x;
                int copyY = y;

                bool isThereEmpty = _currentMap->isThereEmpty(copyX,
copyY);
```

```
                // ищем свободную клетку
                if (!isThereEmpty) {
                    // если нет свободной клетки, квадрат заполнен
                    MessagePrinter::isThereEmpty(x, y, false);

                    if (_currentMap->countSquares() < _optimalMap-
>countSquares()) {
                        // смотрим, больше ли квадратов в оптимальной
карте
                        // если больше, то ставим текущую карту как опти-
мальную
                        MessagePrinter::currentLessOptimal(x, y, true,
_currentMap->countSquares(), _optimalMap->countSquares());

                        _currentMap->print("Current filled map");

                        *_optimalMap = *_currentMap;
                    } else MessagePrinter::currentLessOptimal(x, y,
false, _currentMap->countSquares(), _optimalMap->countSquares());
                } else {

                    MessagePrinter::isThereEmpty(x, y, true, copyX,
copyY);

                    RECURSION_LEVEL ++ ;
                    _solve(copyX, copyY);
                    RECURSION_LEVEL -- ;
                }

                // удаляем квадрат, который поставили, чтобы перебирать
дальше
                _currentMap->removeSquare(x, y, size);
                MessagePrinter::removeSquare(x, y, size);
            } else MessagePrinter::tryToSetSquare(x, y, size, false);
        }
    }

public:
    explicit SquareSolver(int size) {
        auto resultOfCompression = _doCompression(size);
        _size = resultOfCompression.first;
        _compression = resultOfCompression.second;
        _currentMap = new SquareMap(_size, _compression);
        _optimalMap = new SquareMap(_size, _compression);
    }

    SquareMap* solve() {
        if (!(_size % 2)) _solveEvenSquare(); // если квадрат с четной
стороной, то для него подготовлено
                                              // отдельное решение
        else {

            // инициализируем карты перебора и оптимальную
            _initOptimalMap();
            _initCurrentMap();

            std::cout << "Визуализация карты перебора и оптимальной карты
после инициализации:" << std::endl;
```

```
            _optimalMap->print("Optimal Map");
            _currentMap->print("Current Map");

            _solve(_size / 2 + 1, _size / 2);

            std::cout << "Визуализация оптимальной карты после решения"
<< std::endl;
            _optimalMap->print("Optimal Map");

        }
        return _optimalMap;
    }
};


int main() {
    int size;
    std::cin >> size;
    std::cout << (std::string) *SquareSolver(size).solve() << std::endl;
    return 0;
}
```