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 _____ Круглова В.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

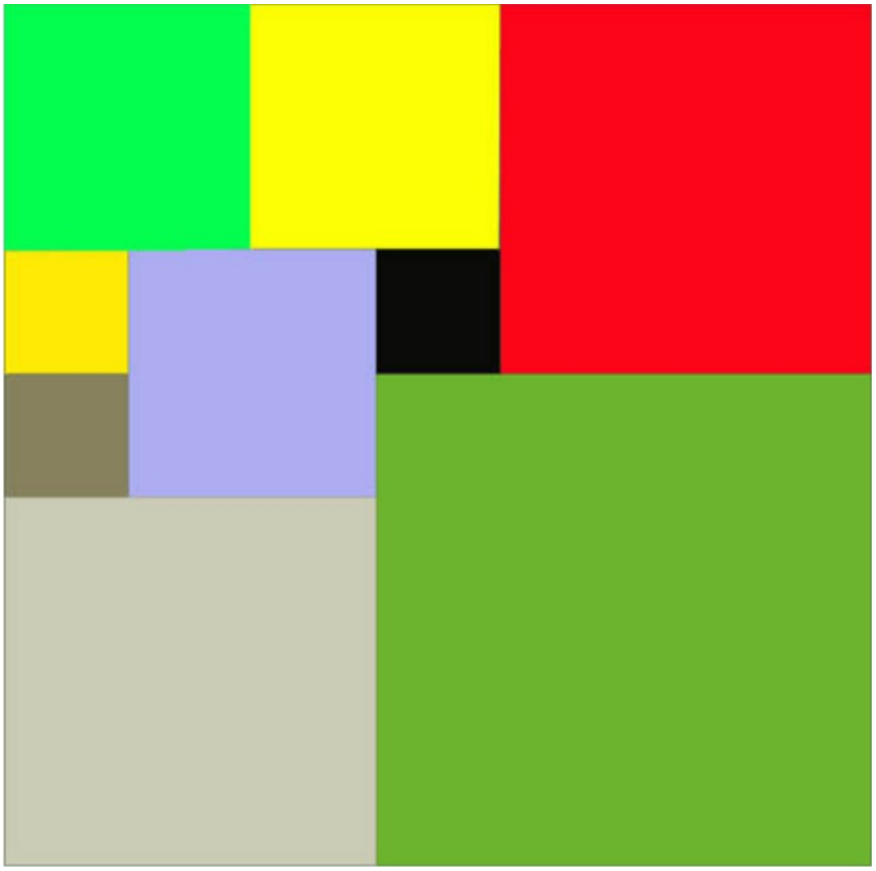
Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным количеством меньших квадратов.

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 x 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.

Вариант 3р.

Рекурсивный бэктрекинг. Исследование количества операций от размера квадрата.

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

Теоретические сведения.

Бэктрекинг – это 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.

Описание алгоритма.

Рекурсивный алгоритм поиска с возвратом реализован методом doTaskRecursive.

При нахождении свободной клетки с помощью перебора всех длин сторон в порядке уменьшения на поле добавляется квадрат максимально возможного размера, верхний левый угол которого расположен в этой клетке. Для хранения промежуточных решений создан currArrayOfSmallSqrs, который содержит координаты и размеры квадратов.

Если в ходе алгоритма число квадратов в текущем разбиении при непустом поле становится больше или равным числу квадратов в минимальном разбиении или если поле оказывается заполнено, происходит удаление с поля «хвоста» разбиения: поле освобождается от размещенных последними единичных квадратов, а квадрат, добавленный на поле перед ними или последним, если в конце разбиения нет единичных квадратов, заменяется квадратом, начинающимся в той же точке поля, со стороной на 1 меньше. При этом дальнейший обход матрицы начинается с правого верхнего угла последнего квадрата в новом разбиении, т. е. оттуда, откуда он продолжился бы, если бы квадрат был добавлен при обычном обходе поля.

Удаление последних квадратов разбиения также служит критерием выхода из рекурсии. Если при удалении «хвоста» был удален последний квадрат разбиения, т. е. больше нет нерассмотренных вариантов разбиения, работа алгоритма завершается.

В случае, когда после добавления нового квадрата поле оказывается заполнено, проверяется число квадратов в текущем разбиении. Если оно меньше числа квадратов в минимальном разбиении, минимальное разбиение заменяется

текущим, а массив результата bestArrayOfSmallSqrs перезаписывается.

Описание рекурсивной функции.

void doTaskRecursive(std::vector<std::vector<int>>& mainSquare, int freeAreaOfMainSquare, int sizeOfSmallSqr, int currCountOfSmallSqrs, std::vector<SmallSquare>& currArrayOfSmallSqrs, int recursionDepth, int sizeOfMainSquare, int& bestCountOfSquares, std::vector<SmallSquare>& bestArrayOfSmallSqrs) -- это рекурсивная функция, принимающая следующие аргументы:

mainSquare — ссылка на заполняемый квадрат; freeAreaOfMainSquare — свободная площадь начального квадрата; sizeOfSmallSqr — размер вставляемого обрезка; currCountOfSmallSqrs — текущее количество уже вставленных обрезков; currArrayOfSmallSqrs — ссылка на массив вставленных обрезков с текущей попытки; recursionDepth — количество отступов для вывода промежуточной информации; sizeOfMainSquare — длина стороны начального квадрата; bestCountOfSquares — пока что лучшее количество обрезков, которыми можно покрыть начальный квадрат; bestArrayOfSmallSqrs — ссылка на массив обрезков пока что лучшего разложения.

Принцип работы функции:

В начале идет проверка-оптимизация, которая сравнивает текущее количество обрезков с лучшим, и если разница между ними равна единице при ненулевой свободной площади начального квадрата, то функция завершается.

После этого функция начинает рекурсивно себя вызывать для перебора всех возможных вариантов заполнения квадрата, после чего пробует вставить в пустое место начального квадрата обрезок размера currSizeOfAddedSquare. При невозможности такой вставки функция завершается.

Если удалось вставить обрезок такого размера, то происходит проверка, возможно ли после вставки добиться наилучшего результата. Если нельзя —

добавленный обрезок удаляется и функция завершается.

Далее идет проверка на то является ли текущее заполнение уже минимальным. Если является, то происходит сохранение результата, как наилучшего, последний квадрат удаляется и функция завершается.

После этого функция циклически вызывает себя же, чтобы попробовать вставить в начальный квадрат другие обрезки для достижения наилучшего результата.

После всего, в самом конце, функция удаляет добавленный обрезок и завершается.

Эта функция ничего не возвращает.

Иные оптимизации:

- Если N кратно 2, то минимальное разбиение всегда будет состоять из 4 равных частей.
- Если N кратно 3 или 5, то будет произведено сжатие квадрата для уменьшения количества вычислений.

Оценка сложности.

В алгоритме используется начальный квадрат размера N*N и другие переменные, зависимые от N, но они не дают весомого вклада в увеличение сложности по памяти. Поэтому сложность алгоритма по памяти = O(N^2), где N — размер исходного квадрата.

В исходном квадрате N*N свободных клеток, количество размеров квадратов которые будут перебираться N. Место для первого квадрата можно выбрать N^2 * N способами. Для второго (N^2-1) * N способами. Таким образом, сложность алгоритма по времени = O((N^2)! * N^N), где N - размер исходного квадрата.

Исследование.

Исследование количества операций от размера квадрата.

Одной операцией считается вызов любой из функций, использующихся логикой алгоритма: isPossibleToAddSquare, addToVisibleSquare, fillVisibleSquareWithZeros, delLastAddedSquare, makeTaskPreparations, findTaskAnswerWithRecursion.

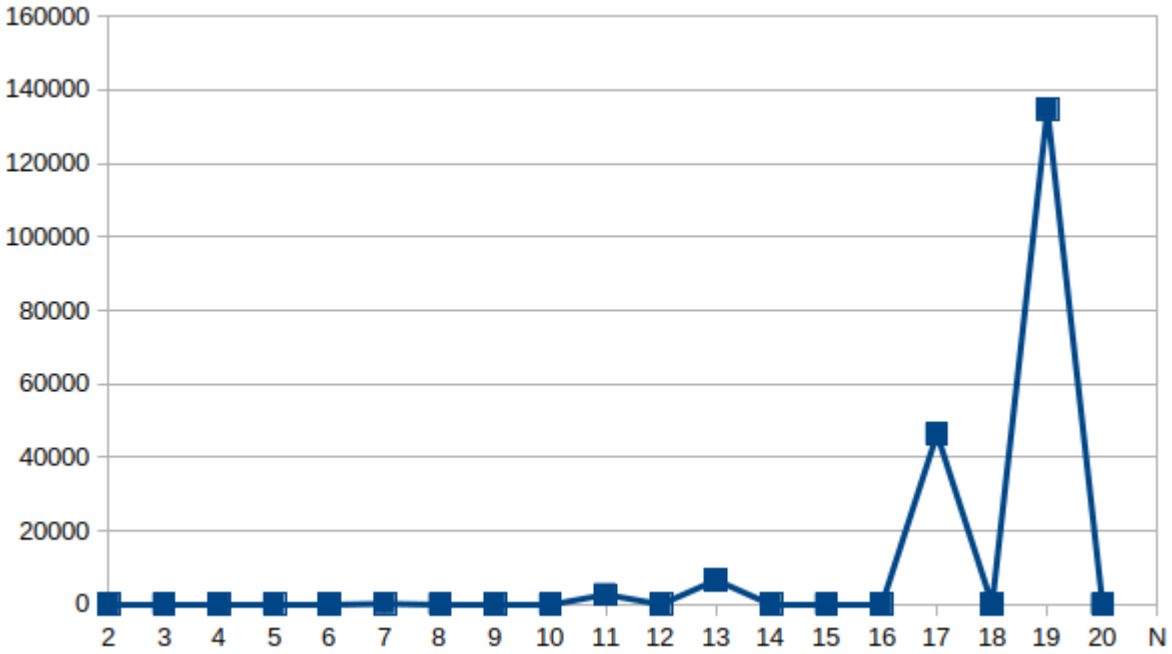
Результаты измерения количества операций в зависимости от размера квадрата представлены в таблице 1.

Таблица 1.

Сторона квадрата, N	Общее количество операций
2	9
3	17
4	9
5	61
6	9
7	202
8	9
9	17
10	9
11	2685
12	9
13	6620
14	9
15	17
16	9
17	46342
18	9
19	134705

20	9

График 1.



По графику 1 можно увидеть, что за исключением случаев, когда используются оптимизации, количество итераций растет экспоненциально.

Тестирование.

Ввод	Вывод

9	6 1 1 6 1 7 3 7 1 3 7 4 3 4 7 3 7 7 3
7	9 1 1 4 1 5 3 5 1 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2

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

20	4 1 1 10 1 11 10 11 1 10 11 11 10

Выводы.

В ходе выполнения лабораторной работы была написана программа, реализующая алгоритм поиска с возвратом рекурсивно.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <ctime>
// промежуточный вывод
#define OUTPUT
// подсчет операций
#define COUNTOPERATIONS
#ifdef COUNTOPERATIONS
int countOfOperations = 0;
#endif
class SmallSquare // класс для содержания информации о конкретных обрезках
{
public:
    int x;
    int y;
    int size;
};
bool tryAbilityToAddSquare(std::vector<std::vector<int>>& mainSquare, int x, int y,
int sizeOfSmallSqr)
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    // базовая проверка на адекватность
    if ((x + sizeOfSmallSqr) > mainSquare.size() || (y + sizeOfSmallSqr) >
mainSquare.size())
    {
        return false;
    }
    for (int i = y; i < y + sizeOfSmallSqr; i++)
    {
        for (int j = x; j < x + sizeOfSmallSqr; j++)
        {
            if (mainSquare[i][j])
            {
                return false;
            }
        }
    }
    return true;
}
void addSmallSqrToMainSqr(std::vector<std::vector<int>>& mainSquare, int x, int y,
int sizeOfSmallSqr)
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    // просто закрашиваем указанные клетки без проверки
    for (int i = y; i < y + sizeOfSmallSqr; i++)
    {
        for (int j = x; j < x + sizeOfSmallSqr; j++)
        {
            mainSquare[i][j] = sizeOfSmallSqr;
        }
    }
}
void printMainSquare(std::vector <std::vector <int>>& mainSquare, int compr, int
sizeOfMainSquare)
{
    for (int i = 0; i < sizeOfMainSquare * compr; i++)
    {
        for (int j = 0; j < sizeOfMainSquare * compr; j++) // просто печатаем все
клетки по порядку
        {
```

```
            std::cout.width(3); // чтобы получился почти квадратик, а не
прямоугольник
            std::cout << mainSquare[i][j];
        }
        std::cout << std::endl;
    }
}
void initMainSqrWithZeros(std::vector<std::vector<int>>& mainSquare, int
sizeOfMainSquare) // заполнен нулями == не поставлено ни одного квадрата
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    // просто заполняем все нулями
    mainSquare.resize(sizeOfMainSquare);
    for (int i = 0; i < sizeOfMainSquare; i++)
    {
        mainSquare[i].resize(sizeOfMainSquare);
        for (int j = 0; j < sizeOfMainSquare; j++)
        {
            mainSquare[i][j] = 0;
        }
    }
}
void deleteLastSmallSqr(std::vector<std::vector<int>>& mainSquare,
std::vector<SmallSquare>& currArrayOfSmallSqrs, int recursionDepth)
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    SmallSquare lastSqr = currArrayOfSmallSqrs.back();
    // очистка из массива
    currArrayOfSmallSqrs.pop_back();
    // очистка из основного квадрата
    for (int i = lastSqr.y; i < lastSqr.y + lastSqr.size; i++)
    {
        for (int j = lastSqr.x; j < lastSqr.x + lastSqr.size; j++)
        {
            mainSquare[i][j] = 0;
        }
    }
#ifdef OUTPUT
    for(int l = 0; l < recursionDepth; l++)
    {
        std::cout << " ";
    }
    std::cout << "Удаляем квадрат (l = " << lastSqr.size << ", x = " <<  lastSqr.x
+ 1 << ", y = " << lastSqr.y + 1 << ')' << std::endl;
#endif
}
void preworkWithMainSqr(int &compr, std::vector<std::vector<int>>& mainSquare,
std::vector<SmallSquare>& currArrayOfSmallSqrs, int& freeAreaOfMainSquare, int&
sizeOfMainSquare, int& bestCountOfSquares)
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    initMainSqrWithZeros(mainSquare, sizeOfMainSquare); // инициализация пустого
"видимого" квадрата
    // оптимизации для простейших кратностей
    if (sizeOfMainSquare % 2 == 0)
    {
        compr = sizeOfMainSquare / 2;
        sizeOfMainSquare = 2;
    }
    else if (sizeOfMainSquare % 3 == 0)
    {
        compr = sizeOfMainSquare / 3;
        sizeOfMainSquare = 3;
    }
```

```
    else if (sizeOfMainSquare % 5 == 0)
    {
        compr = sizeOfMainSquare / 5;
        sizeOfMainSquare = 5;
    }
    // для 7 и дальше нет смысла, т.к. следующее место, где оно поможет слишком
далеко
    bestCountOfSquares = 2 * sizeOfMainSquare + 1;
    // первые 3 квадрата
    currArrayOfSmallSqrs.push_back({ 0, 0, (sizeOfMainSquare + 1) / 2 });
    currArrayOfSmallSqrs.push_back({ 0, (sizeOfMainSquare + 1) / 2,
sizeOfMainSquare / 2 });
    currArrayOfSmallSqrs.push_back({ (sizeOfMainSquare + 1) / 2, 0,
sizeOfMainSquare / 2 });
    addSmallSqrToMainSqr(mainSquare, 0, 0, (sizeOfMainSquare + 1) / 2);
    addSmallSqrToMainSqr(mainSquare, 0, (sizeOfMainSquare + 1) / 2,
sizeOfMainSquare / 2);
    addSmallSqrToMainSqr(mainSquare, (sizeOfMainSquare + 1) / 2, 0,
sizeOfMainSquare / 2);
#ifdef OUTPUT
    std::cout << "Вставляем квадрат (l = " << (sizeOfMainSquare + 1) / 2 << ", x =
" <<  1 << ", y = " << 1 << ")" << std::endl;
    std::cout << "Вставляем квадрат (l = " << sizeOfMainSquare / 2 << ", x = " <<
1 << ", y = " << (sizeOfMainSquare + 1) / 2 + 1 << ")" << std::endl;
    std::cout << "Вставляем квадрат (l = " << sizeOfMainSquare / 2 << ", x = " <<
(sizeOfMainSquare + 1) / 2 + 1 << ", y = " << 1 << ")" << std::endl;
#endif
    // обновление к-ва пустого пространства
    freeAreaOfMainSquare = sizeOfMainSquare * sizeOfMainSquare - ((sizeOfMainSquare
+ 1) / 2) * ((sizeOfMainSquare + 1) / 2) - 2 * (sizeOfMainSquare / 2) *
(sizeOfMainSquare / 2);
}
void doTaskRecursive(std::vector<std::vector<int>>& mainSquare, int
freeAreaOfMainSquare, int sizeOfSmallSqr, int currCountOfSmallSqrs,
std::vector<SmallSquare>& currArrayOfSmallSqrs, int recursionDepth, int
sizeOfMainSquare, int& bestCountOfSquares, std::vector<SmallSquare>&
bestArrayOfSmallSqrs)
{
#ifdef COUNTOPERATIONS
    ++countOfOperations;
#endif
    // даже если квадрат заполнит все пустоты - результат будет не лучше нынешнего
    if ( (currCountOfSmallSqrs == (bestCountOfSquares - 1)) &&
(freeAreaOfMainSquare) )
    {
#ifdef OUTPUT
        for(int l = 0; l < recursionDepth; l++)
        {
            std::cout << " ";
        }
        std::cout << "Количество квадратов не меньше лучшего, выход из рекурсии" <<
std::endl;
#endif
        return;
    }
    // первый добавленный квадрат (после начальных 3)
    if ( ((sizeOfSmallSqr + 1) <= (sizeOfMainSquare / 2)) && (currCountOfSmallSqrs
== 3) )
    {
        // рекурсивный вызов этой же функции
        doTaskRecursive(mainSquare, freeAreaOfMainSquare, (sizeOfSmallSqr+1),
currArrayOfSmallSqrs.size(), currArrayOfSmallSqrs, 0, sizeOfMainSquare,
bestCountOfSquares, bestArrayOfSmallSqrs);
    }
    bool ableToAddSmallSqr = false;
    for (int y = 0; y < sizeOfMainSquare; y++)
    {
        for (int x = 0; x < sizeOfMainSquare; x++)
        {
            // для каждой пустой клетки происходит попытка вставить квадрат
```

```
текущего размера
            if (mainSquare[y][x] == 0)
            {
                if (tryAbilityToAddSquare(mainSquare, x, y, sizeOfSmallSqr))
                {
                    ableToAddSmallSqr = true;
                    addSmallSqrToMainSqr(mainSquare, x, y, sizeOfSmallSqr);
                    freeAreaOfMainSquare -= sizeOfSmallSqr * sizeOfSmallSqr;
                    currArrayOfSmallSqrs.push_back({ x, y, sizeOfSmallSqr });
#ifdef OUTPUT
                    for(int l=0; l < recursionDepth; l++)
                    {
                        std::cout << " ";
                    }
                    std::cout << "Вставляем квадрат (l = " << sizeOfSmallSqr<< ", x
= " <<  x + 1 << ", y = " << y + 1 << ")" << std::endl;
#endif
                    break;
                }
                else
                {
#ifdef OUTPUT
                    for(int l=0; l < recursionDepth; l++)
                    {
                        std::cout << " ";
                    }
                    std::cout << "Не можем поставить квадрат (l = " <<
sizeOfSmallSqr<< ", x = " <<  x + 1 << ", y = " << y + 1 << ")" << std::endl;
#endif
                    return;
                }
            }
            else
            {
                x += (mainSquare[y][x] - 1);
            }
        }
        // выход из цикла
        if(ableToAddSmallSqr)
        {
            break;
        }
    }
    // нет смысла обновлять показатели лучшего результата при таком же количестве
обрезков
    if ( (currCountOfSmallSqrs + 1) == bestCountOfSquares)
    {
#ifdef OUTPUT
        for(int l = 0; l < recursionDepth; l++)
        {
            std::cout << " ";
        }
        std::cout << "Количество квадратов не меньше лучшего, выход из рекурсии" <<
'\n';
#endif
        deleteLastSmallSqr(mainSquare, currArrayOfSmallSqrs, recursionDepth);
        return;
    }
    // минимальное заполнение
    if ( ((currCountOfSmallSqrs + 1) < bestCountOfSquares) && (freeAreaOfMainSquare
== 0))
    {
        bestCountOfSquares = currCountOfSmallSqrs + 1;
        bestArrayOfSmallSqrs.assign(currArrayOfSmallSqrs.begin(),
currArrayOfSmallSqrs.end());
#ifdef OUTPUT
        std::cout << "!!!Нашли новое лучшее количество квадратов: " <<
bestCountOfSquares << std::endl;
#endif
        deleteLastSmallSqr(mainSquare, currArrayOfSmallSqrs, recursionDepth);
```

```
        return;
    }
    // рекурсивный вызов этой же функции
    for (int i = sizeOfMainSquare / 2; i > 0; i--)
    {
        if (i * i <= freeAreaOfMainSquare)
        {
#ifdef OUTPUT
            for (int l=0; l < recursionDepth+2; l++)
            {
                std::cout << " ";
            }
            std::cout << "Вызываем рекурсию для квадрата со стороной " << i << '\
n';
#endif
            doTaskRecursive(mainSquare, freeAreaOfMainSquare, i,
currCountOfSmallSqrs + 1, currArrayOfSmallSqrs, recursionDepth+2, sizeOfMainSquare,
bestCountOfSquares, bestArrayOfSmallSqrs);
        }
    }
    deleteLastSmallSqr(mainSquare, currArrayOfSmallSqrs, recursionDepth);
}
int main()
{
    setlocale(LC_ALL, "Russian");
    std::vector <std::vector<int>> mainSquare; // главный квадрат (0 = не занято)
    std::vector <SmallSquare> currArrayOfSmallSqrs;
    std::vector <SmallSquare> bestArrayOfSmallSqrs;
    int compr = 1; // во сколько раз сжали квадрат
    int freeAreaOfMainSquare;
    int sizeOfMainSquare;
    int bestCountOfSquares; // лучшее количество маленьких квадратов, покрывающих
основной
#ifdef OUTPUT
    std::cout << "Введите размер главного квадрата:" << std::endl;
#endif
    std::cin >> sizeOfMainSquare;
    preworkWithMainSqr(compr, mainSquare, currArrayOfSmallSqrs,
freeAreaOfMainSquare, sizeOfMainSquare, bestCountOfSquares);
#ifdef OUTPUT
    clock_t start = clock();
#endif
    doTaskRecursive(mainSquare, freeAreaOfMainSquare, 1,
currArrayOfSmallSqrs.size(), currArrayOfSmallSqrs, 0, sizeOfMainSquare,
bestCountOfSquares, bestArrayOfSmallSqrs);
#ifdef OUTPUT
    clock_t end = clock();
    std::cout << "\nВремя выполнения: " << (double) (end - start) / CLOCKS_PER_SEC
<< "\n\n";
#endif
#ifdef COUNTOPERATIONS
    std::cout << "Количество операций = " << ::countOfOperations << '\n' <<
std::endl;
#endif
    std::cout << bestCountOfSquares << std::endl;
    for (int i = 0; i < bestArrayOfSmallSqrs.size(); i++)
    {
        std::cout << bestArrayOfSmallSqrs[i].x * compr + 1 << " " <<
bestArrayOfSmallSqrs[i].y * compr + 1 << " " << bestArrayOfSmallSqrs[i].size *
compr << std::endl;
#ifdef OUTPUT
        addSmallSqrToMainSqr(mainSquare, bestArrayOfSmallSqrs[i].x * compr,
bestArrayOfSmallSqrs[i].y * compr, bestArrayOfSmallSqrs[i].size * compr);
#endif
    }
#ifdef OUTPUT
    std::cout << std::endl;
    printMainSquare(mainSquare, compr, sizeOfMainSquare);
#endif
    return 0;
```

```
}
```