

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

Студент гр. 8303

_____ Гришин К. И.

Преподаватель

_____ Фирсов М. А.

Санкт-Петербург

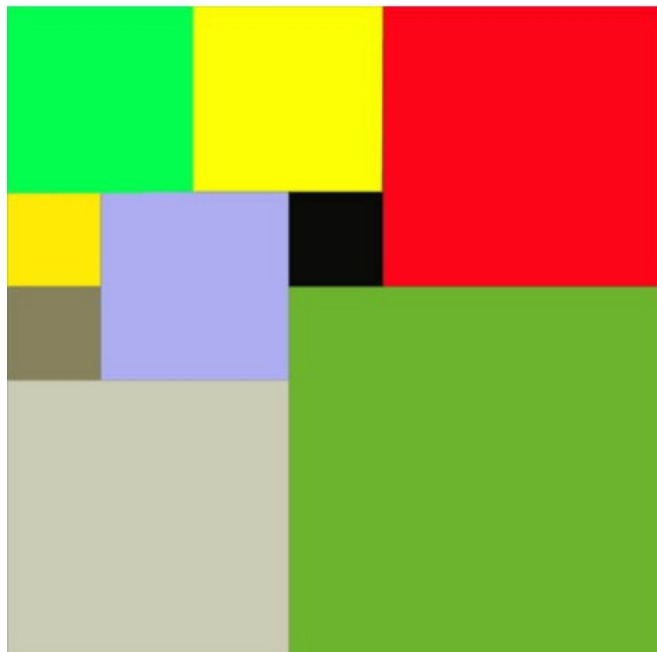
2020

Цель работы

Изучить алгоритм «Backtracking» (он же «Поиск с возвратом») на примере замещения квадратной столешницы заданного размера наименьшим числом 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 7×7 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 ($2 \leq N \leq 20$).

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).

Пример входных данных	Соответствующие выходные данные
7	9 1 1 2 1 3 2 3 1 1 4 1 1 3 2 2 5 3 1 4 4 4 1 5 3 3 4 1

Индивидуальное задание: вариант 5р. Рекурсивный бэктрекинг. Возможность задать список квадратов (от 0 до N^2 квадратов в списке), которые обязательно должны быть использованы в покрытии квадрата со стороной N .

Код (основной код программы, заголовок упрощающий графику в терминале, упрощенная версия программы для stepik) приведен в приложениях А, В и С к лабораторной работе.

Описание работы программы

При запуске требуется указать тип ввода: «из файла» - *<file>* или «вручную» - *<manual>*. Затем нужно ввести либо имя файла, либо уже непосредственно данные. Оформление ввода из файла ничем не отличается от ручного ввода. Если введенный квадрат выходит за пределы столешницы, программа прекращает работу. Если какой-то квадрат на место в котором он задевает другой квадрат, программа завершает работу.

Ввод	Соответствующий выход
Размер квадрата	Изображение замощенного квадрата
Кол-во обязательных частей	Количество полученных частей
Квадрат 1	Квадрат 1
...	...
Квадрат n	Квадрат $k \geq n$

Описание алгоритма

Алгоритм имеет постоянный контроль уже вписанных квадратов и обновляет лучшую конфигурацию, если вся область квадрата занята и количество вписанных частей меньше, чем предыдущее (изначально это количество инициализируется числом $size * size$ — по количеству максимально возможных частей квадрата).

Если в момент неполного заполнения квадрата частей уже больше, чем в лучшем результате, то дальше не имеет смысла продолжать построение, производится выход из функции

Алгоритм:

1. Вводится количество обязательных квадратов.
2. Построчно вводятся определяющие квадратов « x y w », где x и y положение на поле, w — размер.
3. Введенные квадраты размещаются на пустом поле функцией *setSquare(int x, int y, int w)*.

Основной ход функции *backtracking*:

4. Найти ближайший к левому верхнему углу свободный квадрат.
5. Если такого квадрата нет, столешница замощена, запоминается результат.
6. Уменьшить размер полученного квадрата на 1, если он занимает исходный квадрат целиком.
6. Отобразить квадрат на исходном поле, если его размер больше 0.
7. Рекурсивно войти в функцию бэктрекинга, сообщив ей об увеличении квадратов на поле.

8. Удалить квадрат с поля

9. Уменьшить размер квадрата на единицу вернуться к п. 6.

Каждый добавленный и удаленный квадрат логируются. Также сохраняются в файл все лучшие конфигурации.

Описание функций и классов

«Point» - класс точки, содержит два целых числа *int* *x*, *y* для хранения координат.

«SquareData» - класс для хранения частей квадрата отдельно от самого квадрата, содержит поле *Point* для хранения координат и целое число *w* — размер. Класс имеет перегруженный оператор *<<*, выводит через пробелы значения полей классов, после чего ставит перенос строки.

«Square» - основной класс хранения заданного квадрата. Содержит в себе матрицу целых чисел *iArray*, лучшую конфигурацию *bestConfiguration*, результирующее количество квадратов *iColors* и вспомогательную переменную *res*, которая контролирует выход из функции бэктрекинга. Содержит поле файлового дескриптора для логирования данных.

Методы класса:

Конструктор(int iSize) — на вход получает размер квадрата для обработки, выделяет память под матрицу, создает файл логирования, .

Деструктор — закрывает файл лога, освобождает память матрицы.

SquareData findPotentialSquare() — находит ближайшую к верхнему левому углу свободную точку, от которой начинает отсчет для поиска размера свободного квадрата. Если квадрат найден не был, возвращает обнуленную структуру.

SquareData findSquare(int Color) — на вход получает цвет (номер части квадрата), выводит структуру определяющую его положение. Если квадрата с таким номером не найдено, возвращает пустую структуру.

float proceed(bool logger = 1) — на вход получает флаг логирования, определяющий нужна ли запись в логов, настраивает и инициализирует метод бэктрекинга. Запоминает текущее количество тиков перед вызовом метода, а затем считает прошедшее количество тиков, после чего возвращает время, затраченное на функцию бэктрекинга.

void setSquare(int x, int y, int w) — получает на вход координаты квадрата и размер. Устанавливает в заданную позицию квадрат с заданным размером, устанавливается цвет следующий за текущим. Количество квадратов увеличивается на 1.

void delSquare(int x, int y) — получает на вход координаты, по которым находит квадрат, обнуляет его, уменьшает количество квадратов, а все квадраты с номерами больше удаленного уменьшают их на 1.

const vector<SquareData>& configuration() — выдает доступ к чтению лучшей конфигурации.

void setConfiguration() — очищает текущую лучшую конфигурацию, после чего находит все квадраты и помещает их в лучшую конфигурацию. Квадраты пронумерованы от 1 до iColors, следовательно легко находятся.

void updateMatrix() — восстанавливает значения матрицы по вектору лучшей конфигурации.

long long int getBacktrackCallsCount() — возвращает количество вызовов функции бэктрекинга.

Хранение решения

Временное заполнение квадрата хранится в матрице $N \times N$ *int **iArray*. Лучшие конфигурации хранятся в виде вектора *vector<SquareData>* определяющих частей квадрата.


Использованные оптимизации


1. Каждый раз, когда при входе в функцию бэктрекинга обнаруживается, что текущее количество квадратов больше, чем при лучшей конфигурации, производится выход из функции.

2. Проверяется делимость стороны квадрата на простые числа 2, 3, 5, если сторона делится, то квадрат заполняется как квадрат размером с соответствующий делитель, без вызова функции бэктрекинга.


3. В упрощенной версии программы используется апостериорная оптимизация. При первом вызове функции бэктрекинга, в квадрат сразу помещается три части: в левый верхний угол размером $iSize/2+1$, верхний правый и левый нижний размером $iSize/2$.

Тестирование

```
Enter input type:
manual
11
0

11
1 1 7
8 1 4
8 5 4
1 8 4
5 8 2
7 8 1
7 9 3
10 9 2
5 10 2
10 11 1
11 11 1
Calculated in 2.3417 seconds
Backtracking function calls: 262229
```

```
Enter input type:
manual
5
1
3 3 2

13
3 3 2
1 1 2
3 1 2
5 1 1
5 2 1
1 3 2
5 3 1
5 4 1
1 5 1
2 5 1
3 5 1
4 5 1
5 5 1
Calculated in 0.005496 seconds
Backtracking function calls: 190
```

```

Enter input type:
manual
6
2
2 2 4
6 6 1

21
2 2 4
6 6 1
1 1 1
2 1 1
3 1 1
4 1 1
5 1 1
6 1 1
1 2 1
6 2 1
1 3 1
6 3 1
1 4 1
6 4 1
1 5 1
6 5 1
1 6 1
2 6 1
3 6 1
4 6 1
5 6 1
Calculated in 0.000318 seconds
Backtracking function calls: 20

```

Упрощенная версия


```

19

13
1 1 10
11 1 9
1 11 9
11 10 3
14 10 6
10 11 1
10 12 1
10 13 4
14 16 1
15 16 1
16 16 4
10 17 3
13 17 3
time: 0.091522

```

```

Enter input type:
manual
3
0

6
1 1 2
3 1 1
3 2 1
1 3 1
2 3 1
3 3 1
Calculated in 0.000764 seconds
Backtracking function calls: 23

```


Вывод

В ходе выполнения работы была изучена идея бэктрекинга. Описан алгоритм замощения квадрата при помощи поиска с возвратом. Алгоритм имплементирован на языке C++.

ПРИЛОЖЕНИЕ А

Код основной программы (файл main.cpp)

```
#include <iostream> // to use standart stream of input and output
#include <algorithm> // to use std::min function
#include <vector> // to use vector container
#include <ctime> // for calculating time of backtracking work
#include <fstream> // to use file streams (log and file reading)
#include <string> // to use std::string

// lib for terminal graphics
#include "atr.h"

//storage of coordinate
struct Point
{
    int x;
    int y;
};

//defining square parameters. Its position and size
struct SquareData
{
    Point pos; // coordinates (begin from (0;0))
    int w; // square edge size

    friend std::ostream& operator<<(std::ostream& os, const SquareData& sq){
        os << sq.pos.x << " " << sq.pos.y << " " << sq.w << std::endl;
    }
};

// transfer an integer to one of the terminal colors (using deduction ring)
void setColor(int matrixColor){ // change tarminal parameters
    switch(matrixColor % 9){ // sets background color
        case 0:
            set_display_atr(B_BLACK);
            break;

        case 1:
            set_display_atr(B_RED);
            break;

        case 2:
            set_display_atr(B_GREEN);
            break;

        case 3:
            set_display_atr(B_YELLOW);
            break;

        case 4:
            set_display_atr(B_BLUE);
            break;

        case 5:
            set_display_atr(B_MAGENTA);
            break;

        case 6:
            set_display_atr(B_CYAN);
            break;

        case 7:
            set_display_atr(B_WHITE);
            break;

        case 8:
            resetcolor();
    }
}

class Square
{
private:
    // vector which contains SquareData's of
    // best configuration of splitting
    std::vector<SquareData> bestConfiguration;

    const int iSize; // size of square side

    int **iArray; // Matrix for storing colored squares
    int iColors = 0; // How many color used to split matrix (also this is a quantity of squares on field)
    int res; // Flag for backtracking out
    long long int ops = 0; // backtracking calls

    std::ofstream log; // file stream for log
    bool logger = 1; // is logging on?

    // find first free cell from left to right and from top to bot
    // calculate maximum size of square, which can be placed from that cell
    SquareData findPotentialSquare() const{
        int x = -1;
        int y = -1;
        for(int i = 0; i < iSize; i++){ // find left upper free cell
            bool b = 0;

```

```

        for(int j = 0; j < iSize; j++){
            if(iArray[i][j] == 0){
                y = i;
                x = j;
                b = 1;
                break; // remember it and break
            }
        }
        if(b) break; // breaker of included cycle
    }

    if(x == -1 && y == -1){ // if free cell didn't found return empty structure
        return SquareData{Point{0, 0}, 0};
    }

    // find size of maximal free square from free cell
    int s;
    for(s = 0; s <= iSize - std::max(x, y); s++){ // extend possible border
        bool b = 0;
        //check borders, if they are contain non-zero elements break
        for(int i = y; i < y+s; i++){
            if(iArray[i][x+s-1] != 0){
                b = 1;
                break;
            }
        }
        for(int i = x; i < x+s; i++){
            if(iArray[y+s-1][i] != 0){
                b = 1;
                break;
            }
        }
        if(b) break;
    }
    s--;
    return SquareData{Point{x, y}, s};
}

// find first cell from left to right and top to bot
// with required cell, and calculate size of square from found cell
SquareData findSquare(int color){
    int x = -1;
    int y = -1;

    // find first cell
    for(int i = 0; i < iSize; i++){
        bool b = 0;
        for(int j = 0; j < iSize; j++){
            if(iArray[i][j] == color){
                y = i;
                x = j;
                b = 1;
                break;
            }
        }
        if(b) break;
    }

    // if cell wasn't found? return
    if(x == -1 && y == -1)
        return SquareData{Point{0, 0}, 0};

    // instead of extending borders, just go along one side
    int s;
    for(s = x; s < iSize; s++){
        if(iArray[y][s] != color)
            break;
    }
    s -= x;
    return SquareData{Point{x, y}, s};
}

// backtracking funbction itself
// iCurSize keeps quantity of placed squares
void backtrack(int iCurSize){
    if(logger) log << std::endl << "Enter backtracking function: " << std::endl;
    ops++;

    // res - quantity of squares in best configuration
    // if current quantity of squares more then res,
    // no point continuing placing squares, return
    if(iCurSize >= res){ // if current quantity of squares more than best config, return
        if(logger) log << "Current squares quantity cause function out" << std::endl << std::endl;
        return;
    }

    // find free square
    auto emptySquare = findPotentialSquare();

    // if square if whole field size found, decrease it size by 1
    if(emptySquare.w == iSize)
        emptySquare.w--;

    // if structure of founded square is empty, check for best result and save it
    if(emptySquare.w == 0){
        // get minimum between current square quantity on field
        // and quantity of squares in best configuration;
        res = std::min(res, iCurSize);

        // remember all squares on field as SquareInfo structures;
        setConfiguration();
    }
}

```

```

        if(logger){
            log << "\nNew best conf set: " << std::endl;
            for(auto sq: bestConfiguration)
                log << sq;
            log << std::endl;
        }
    }

else{
    int w = emptySquare.w; // save edge size of square
    if(logger) log << "Position for square found: " << std::endl;

    // cycle decreasing edge size by one every lapse
    while(w > 0){
        // set found square with size w int matrix
        setSquare(emptySquare.pos.x, emptySquare.pos.y, w);

        if(logger){
            log << "square added: ";
            log << emptySquare.pos.x << " " << emptySquare.pos.y << " " << w << std::endl;
        }

        // call backtracking function to other part of field
        backtrack(iCurSize+1);

        if(logger){
            log << "Square deleted: ";
            log << emptySquare.pos.x << " " << emptySquare.pos.y << " " << w << std::endl;
        }

        // remove square from field to set in the same pos square with less edge
        delSquare(emptySquare.pos.x, emptySquare.pos.y);
        // decrement size of square edge to put
        w--;
        if (logger && w > 0) log << "Decrease square in pos: "
                                << emptySquare.pos.x << " "
                                << emptySquare.pos.y << std::endl;
    }
    if(logger) log << "function out\n" << std::endl;
}

// remember field as dynamic array of
// squares definers (called SquareData)
void setConfiguration(){
    bestConfiguration.clear();
    // find all squares in matrix and save it to best config
    // is square has number, which he gets, when added on table
    // using this numbers we can find all squares on field
    for(int i = 1; i <= iColors; i++){
        auto square = findSquare(i);
        if(square.w != 0)
            // bestConfiguration is a part of main working class
            bestConfiguration.emplace_back(square);
    }
}

// put all squares from bestConfiguration on field
void updateMatrix(){
    int c = 1;
    // colorize matrix using best config
    for(auto data: bestConfiguration){
        for(int i = data.pos.y; i < data.pos.y + data.w; i++){
            for(int j = data.pos.x; j < data.pos.x + data.w; j++){
                iArray[i][j] = c;
            }
        }
        c++;
    }
}

// checks for prime divisibility
int checkOptimalSolution(){
    // check dividing of iSize. And set best config if it dividing
    if (iSize % 2 == 0)
    {
        div2();
        setConfiguration();
        return 1;
    }
    else if (iSize % 3 == 0)
    {
        div3();
        setConfiguration();
        return 1;
    }
    else if (iSize % 5 == 0)
    {
        div5();
        setConfiguration();
        return 1;
    }
    return 0;
}

// function of the same nature
// if square size can be divided by 2, 3 or 5
// we just can fill it manually and keeping proportions
void div2(){

```

```

        setSquare(0, 0, iSize/2);
        setSquare(iSize/2, 0, iSize/2);
        setSquare(0, iSize/2, iSize/2);
        setSquare(iSize/2, iSize/2, iSize/2);
    }

    void div3(){
        setSquare(0, 0, iSize/3*2);
        setSquare(iSize/3*2, 0, iSize/3);
        setSquare(iSize/3*2, iSize/3, iSize/3);
        setSquare(0, iSize/3*2, iSize/3);
        setSquare(iSize/3, iSize/3*2, iSize/3);
        setSquare(iSize/3*2, iSize/3*2, iSize/3);
    }

    void div5(){
        setSquare(0, 0, iSize/5*3);
        setSquare(iSize/5*3, 0, iSize/5*2);
        setSquare(iSize/5*3, iSize/5*2, iSize/5*2);
        setSquare(0, iSize/5*3, iSize/5*2);
        setSquare(iSize/5*2, iSize/5*3, iSize/5);
        setSquare(iSize/5*2, iSize/5*4, iSize/5);
        setSquare(iSize/5*3, iSize/5*4, iSize/5);
        setSquare(iSize/5*4, iSize/5*4, iSize/5);
    }

public:
    // Constructor
    Square(int iSize):
        iSize(iSize)
    {
        // create or clear log file
        log.open("log", std::ios::out | std::ios::trunc);

        // create matrix
        iArray = new int*[iSize];
        for(int i = 0; i < iSize; i++){
            iArray[i] = new int[iSize];
            for(int j = 0; j < iSize; j++){
                iArray[i][j] = 0;
            }
        }
    }

    // Destructor
    ~Square(){
        // close log file
        log.close();

        // free memory from matrix
        for(int i = 0; i < iSize; i++){
            delete [] iArray[i];
        }
        delete iArray;
    }

    int getSize() const { // returns size of field
        return iSize;
    }

    // initialize and run backtracking function
    // returns time of working
    float proceed(bool logger = 1){
        // set logger flag
        this->logger = logger;

        // initialize fictional best config size
        // maximal quantity of squares is quantity of cells
        // therefore, the fictious best result is equal
        // to the area
        res = iSize*iSize;

        // save time of program before function
        auto t = clock();

        // run backtracking function
        backtrack(0);

        // update time after function
        t = clock() - t;

        //set best config into matrix
        updateMatrix();

        // turn logger off
        logger = 0;
        log << std::endl;

        return (float)t/CLOCKS_PER_SEC;
    }

    // fills square defined as top-left point is (x;y) and width w
    // if in this area already was any square (there is cell, which not zero)
    // finish the programm and says about conflict
    void setSquare(int x, int y, int w){
        for(int i = y; i < y + w && i < iSize; i++){

```

```

        for(int j = x; j < x + w && j < iSize; j++){
            if(iArray[i][j] != 0){ // if there is painted cell, can't place square
                if(logger){
                    auto occupying = findSquare(iArray[i][j]);
                    auto to_insert = SquareData{x, y, w};
                    log << "Conflict of squares " << std::endl;
                    log << "Occupying square: " << occupying.pos.x+1 << " "
                        << occupying.pos.y+1 << " "
                        << occupying.w << std::endl;
                    log << "Square to set: " << x+1 << " " << y+1
                        << " " << w << std::endl;
                }

                std::cout << "Conflicting of squares, hard shutdown" << std::endl;

                // close log file
                log.close();
                // free memory from matrix
                for(int i = 0; i < iSize; i++){
                    delete [] iArray[i];
                }
                delete iArray;

                std::exit(0);
            }

            // Покрасить клетку в цвет выше максимального
            iArray[i][j] = iColors+1;
        }
        // увеличить максимальный цвет на 1
        iColors++;
    }

    // gets vector of SquareData, and then
    // foreach data in vector, set appropriate square on field
    void setSquareVector(const std::vector<SquareData> &vec){
        for(auto square: vec){
            setSquare(square.pos.x, square.pos.y, square.w);
        }
    }

    // set all cells with color of cell(x, y) as zero, keep color of that cell
    // for each square which has color bigger then removed, decrease colors on
    // its cells. Decrease colors quantity
    void delSquare(int x, int y){
        // save color of input cell
        int color = iArray[y][x];

        // make all cells with saved color as zero
        for(int i = y; i < iSize; i++){
            for(int j = x; j < iSize; j++){
                if(iArray[i][j] == color)
                    iArray[i][j] = 0;
                else break;
            }
        }

        // decrease all colors more than saved
        for(int i = 0; i < iSize; i++){
            for(int j = 0; j < iSize; j++){
                if(iArray[i][j] > color)
                    iArray[i][j]--;
            }
        }

        //Decrease colors quantity
        iColors--;
    }

    // Returns refer to best configuration
    // so that it can be printed
    const std::vector<SquareData>&
    configuration(){
        return bestConfiguration;
    }

    // at begining of backtrack function ops counter increases
    // so we can get quantity of backtracking calls
    long long int getBacktrackCallsCount() const {
        return ops;
    }

    // operator of matrix displaying.
    // gets color of cell, then converts it into
    // some color of terminal, then prints two spaces
    // (two characters are square, like pixel)
    friend std::ostream& operator<<(std::ostream& os, const Square& sq){
        for(int y = 0; y < sq.iSize; y++){
            for(int x = 0; x < sq.iSize; x++){
                // dye terminal
                setColor(sq.iArray[y][x]);
                // print pixel
                os << "  ";
            }
            // set standart terminal colors
            resetcolor();
            // get next line

```

```

        os << '\n';
    }
    // set standart terminal colors
    resetcolor();
}
};

int main(){
    // get type of input. Manual: put all data using terminal
    // File: print name of input file, which lies near programm executor
    std::string type;
    std::cout << "Enter input type:" << std::endl;

    // Ask user to enter a function until the correct one is entered
    // if instead of type, user enters "quit", programm closes
    std::cin >> type;
    while(type != "file" && type != "manual" && type != "quit"){
        std::cout << "Unknown command. Enter:\n"
            << "<manual> to read directly\n"
            << "<file> to specify file with data\n"
            << "<quit> to quit programm" << std::endl;
        std::cin >> type;
    }

    // initialize vector of predisposed squares
    int size;
    std::vector<SquareData> enterData;
    int squaresCount;

    // if "quit" entered, end the programm
    if(type == "quit")
        return 0;

    // if "manual" entered.
    // Program gets size of field and
    // quantity of predisposed squares.
    // Then user should enter squares itself
    if(type == "manual"){
        std::cin >> size;
        std::cin >> squaresCount;
        for(int i = 1; i <= squaresCount; i++){
            int x, y, w;
            std::cin >> x >> y >> w;
            x--;
            y--;

            // if read square goes beyond the field,
            // end programm with appropriate message
            if(x+w-1 >= size || y+w-1 >= size || x < 0 || y < 0){
                std::cout << "Square " << i << " out of bounds" << std::endl;
                return 0;
            }

            // save SquareData to input vector
            enterData.emplace_back(SquareData{Point{x, y}, w});
        }

    // if "file" entered
    // programm asks name of file with input.
    // Then it tries to open this. If it can't, then end programm.
    // Else read size of square from first string,
    // Afterwards read quantity of predisposed squares (call it squaresCount)
    // And then 'squaresCount' times read next line, and save SquareData of predisposed squares
    if(type == "file"){
        // get file name
        std::cout << "File name: ";
        std::string fname;
        std::cin >> fname;

        // create file descriptor and open file
        std::ifstream fs(fname.c_str());

        // if file is not openned
        // end programm with appropriate message in terminal
        if(!fs.is_open()){
            std::cout << "Unable to open" << std::endl;
            return 0;
        }

        // read size of field
        fs >> size;
        // read quantity of predisposed squares
        fs >> squaresCount;
        // read SquareData of predisposed squares in other lines
        for(int i = 1; i <= squaresCount; i++){
            int x, y, w;
            fs >> x >> y >> w;
            x--;
            y--;

            // if read square goes beyond the field,
            // end programm with appropriate message
            if(x+w-1 >= size || y+w-1 >= size || x < 0 || y < 0){
                std::cout << "Square " << i << " out of bounds" << std::endl;
                return 0;
            }
        }
    }
}

```

```

        // insert read Square into input vector
        enterData.emplace_back(SquareData{Point{x, y}, w});
    }

    // initialize field
    Square square(size);

    // set read squares to field
    if(!enterData.empty()){
        square.setSquareVector(enterData);
    }

    // proceed field. Split into smaller squares
    float t = square.proceed();

    // Display the resulting matrix
    std::cout << square;

    // Print all squares in field
    std::cout << square.configuration().size() << std::endl;
    for(auto piece: square.configuration()){
        std::cout << piece.pos.x + 1 << " "
                  << piece.pos.y + 1 << " "
                  << piece.w << std::endl;
    }

    // Print data about working time, and
    // quantity of calls backtracking function
    std::cout << "Calculated in " << t << " seconds" << std::endl
              << "Backtracking function calls: " << square.getBacktrackCallsCount() << std::endl;

    return 0;
}

```


ПРИЛОЖЕНИЕ В

Код графического консольного дополнения atr.h

```
#ifndef __ATR__
#define __ATR__

#define ESC "\033"

#define RESET          0
#define BRIGHT         1
#define DIM            2
#define UNDERSCORE    3
#define BLINK          4
#define REVERSE        5
#define HIDDEN         6

//foreground

#define F_BLACK        30
#define F_RED          31
#define F_GREEN        32
#define F_YELLOW      33
#define F_BLUE         34
#define F_MAGENTA     35
#define F_CYAN         36
#define F_WHITE        37

//background

#define B_BLACK        40
#define B_RED          41
#define B_GREEN        42
#define B_YELLOW      43
#define B_BLUE         44
#define B_MAGENTA     45
#define B_CYAN         46
#define B_WHITE        47

#define home()          printf(ESC "[H")
#define clrscr()        printf(ESC "[2J")
#define gotoxy(x, y)    printf(ESC "[%d;%dH", y, x)
#define visible_cursor() printf(ESC "[?251")
#define resetcolor()    printf(ESC "[0m")
#define set_display_atr(color) printf(ESC "[%dm", color)

#endif
```

ПРИЛОЖЕНИЕ С

Код упрощенной версии программы (файл stpk.cpp), которая использовалась на *stepik.org*

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

struct Point
{
    int x;
    int y;
};

struct SquareData
{
    Point pos;
    int w;

    friend std::ostream& operator<<(std::ostream& os, const SquareData& sq){
        os << sq.pos.x << " " << sq.pos.y << " " << sq.w << std::endl;
    }
};

class Square
{
private:
    int **iBestConfiguration;

    const int iSize;
    int **iArray;
    int iColors = 0;
    int res;

    SquareData findPotentialSquare() const{
        int x = -1;
        int y = -1;
        for(int i = 0; i < iSize; i++){
            bool b = 0;
            for(int j = 0; j < iSize; j++){
                if(iArray[i][j] == 0){
                    y = i;
                    x = j;
                    b = 1;
                    break;
                }
            }
            if(b) break;
        }

        if(x == -1 && y == -1){
            return SquareData{Point{0, 0}, 0};
        }

        int s;
        for(s = 0; s <= iSize - std::max(x, y); s++){
            bool b = 0;
            for(int i = y; i < y+s; i++){
                if(iArray[i][x+s-1] != 0){
                    b = 1;
                    break;
                }
            }
            for(int i = x; i < x+s; i++){
                if(iArray[y+s-1][i] != 0){
                    b = 1;
                    break;
                }
            }
            if(b) break;
        }
        s--;
        return SquareData{Point{x, y}, s};
    }

    SquareData findSquare(int color){
        int x = -1;
        int y = -1;
        for(int i = 0; i < iSize; i++){
            bool b = 0;
            for(int j = 0; j < iSize; j++){
                if(iArray[i][j] == color){
                    y = i;
                    x = j;
                    b = 1;
                    break;
                }
            }
            if(b) break;
        }

        if(x == -1 && y == -1)
            return SquareData{Point{0, 0}, 0};

        int s;
    }
```

```

        for(s = x; s < iSize; s++)
            if(iArray[y][s] != color)
                break;
        s -= x;
        return SquareData{Point{x, y}, s};
    }

void backtrack(int iCurSize){
    if(iCurSize >= res)
        return;
    auto emptySquare = findPotentialSquare();
    if(emptySquare.w == iSize){
        emptySquare.w /= 3;
        emptySquare.w *= 2;
    }
    if(emptySquare.w == 0){
        res = std::min(res, iCurSize);
        setConfiguration();
    }
    else{
        int w = emptySquare.w;

        while(w > 0){
            setSquare(emptySquare.pos.x, emptySquare.pos.y, w);
            if(emptySquare.pos.x == 0 && emptySquare.pos.y == 0){
                if(w < iSize/2+1) return;
                setSquare(w, 0, iSize - w);
                setSquare(0, w, iSize - w);
                backtrack(3);
            }
            else
                backtrack(iCurSize+1);

            delSquare(emptySquare.pos.x, emptySquare.pos.y);
            if(emptySquare.pos.x == 0 && emptySquare.pos.y == 0 && w >= iSize/2+1){
                delSquare(w, 0);
                delSquare(0, w);
            }
            w--;
        }
    }
}

void setConfiguration(){
    for(int i = 0; i < iSize; i++){
        for(int j = 0; j < iSize; j++){
            iBestConfiguration[i][j] = iArray[i][j];
        }
    }
}

void updateMatrix(){
    for(int i = 0; i < iSize; i++){
        for(int j = 0; j < iSize; j++){
            iArray[i][j] = iBestConfiguration[i][j];
        }
    }
}

int checkOptimalSolution(){
    if (iSize % 2 == 0)
    {
        div2();
        return 1;
    }
    else if (iSize % 3 == 0)
    {
        div3();
        return 1;
    }
    else if (iSize % 5 == 0)
    {
        div5();
        return 1;
    }
    return 0;
}

void div2(){
    setSquare(0, 0, iSize/2);
    setSquare(iSize/2, 0, iSize/2);
    setSquare(0, iSize/2, iSize/2);
    setSquare(iSize/2, iSize/2, iSize/2);
}

void div3(){
    setSquare(0, 0, iSize/3*2);
    setSquare(iSize/3*2, 0, iSize/3);
    setSquare(iSize/3*2, iSize/3, iSize/3);
    setSquare(0, iSize/3*2, iSize/3);
    setSquare(iSize/3, iSize/3*2, iSize/3);
    setSquare(iSize/3*2, iSize/3*2, iSize/3);
}

void div5(){
    setSquare(0, 0, iSize/5*3);
    setSquare(iSize/5*3, 0, iSize/5*2);
    setSquare(iSize/5*3, iSize/5*2, iSize/5*2);
    setSquare(0, iSize/5*3, iSize/5*2);
    setSquare(iSize/5*2, iSize/5*3, iSize/5);
}

```

```

        setSquare(iSize/5*2, iSize/5*4, iSize/5);
        setSquare(iSize/5*3, iSize/5*4, iSize/5);
        setSquare(iSize/5*4, iSize/5*4, iSize/5);
    }

public:
    Square(int iSize): iSize(iSize)
    {
        iBestConfiguration = new int*[iSize];
        iArray = new int*[iSize];
        for(int i = 0; i < iSize; i++){
            iBestConfiguration[i] = new int[iSize];
            iArray[i] = new int[iSize];
            for(int j = 0; j < iSize; j++){
                iBestConfiguration[i][j] = 0;
                iArray[i][j] = 0;
            }
        }

        ~Square(){
            for(int i = 0; i < iSize; i++){
                delete [] iBestConfiguration[i];
                delete [] iArray[i];
            }
            delete [] iBestConfiguration;
            delete [] iArray;
        }

        int getSize() const {
            return iSize;
        }

        float proceed(){
            if(!checkOptimalSolution()){
                res = iSize*iSize;
                backtrack(0);
                iColors = res;
                updateMatrix();
            }
        }

        void setSquare(int x, int y, int w){
            for(int i = y; i < y + w && i < iSize; i++){
                for(int j = x; j < x + w && j < iSize; j++){
                    iArray[i][j] = iColors+1;
                }
            }
            iColors++;
        }

        void delSquare(int x, int y){
            int color = iArray[y][x];
            for(int i = y; i < iSize; i++){
                for(int j = x; j < iSize; j++){
                    if(iArray[i][j] == color)
                        iArray[i][j] = 0;
                    else break;
                }
            }
            for(int i = 0; i < iSize; i++){
                for(int j = 0; j < iSize; j++){
                    if(iArray[i][j] > color)
                        iArray[i][j]--;
                }
            }
            iColors--;
        }

        void printConfiguration(std::ostream &os){
            os << iColors << std::endl;
            for(int i = 1; i <= iColors; i++){
                auto sq = findSquare(i);
                os << sq.pos.x+1 << " " << sq.pos.y+1 << " " << sq.w << std::endl;
            }
        }
};

int main(){
    int size;
    std::cin >> size;
    Square square(size);

    square.proceed();
    square.printConfiguration(std::cout);
    return 0;
}

```