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. 8383 _____ Федоров И.А

Преподаватель _____ Фирсов М.А

Санкт-Петербург
2020

Цель работы.

Ознакомиться с алгоритмом поиска с возвратом (backtracking). Реализовать программу с итеративным бэктрекингом, находящим решение задачи за разумное время. Составить способы оптимизации работы алгоритма для поставленной задачи.

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 (см. рис. 1).

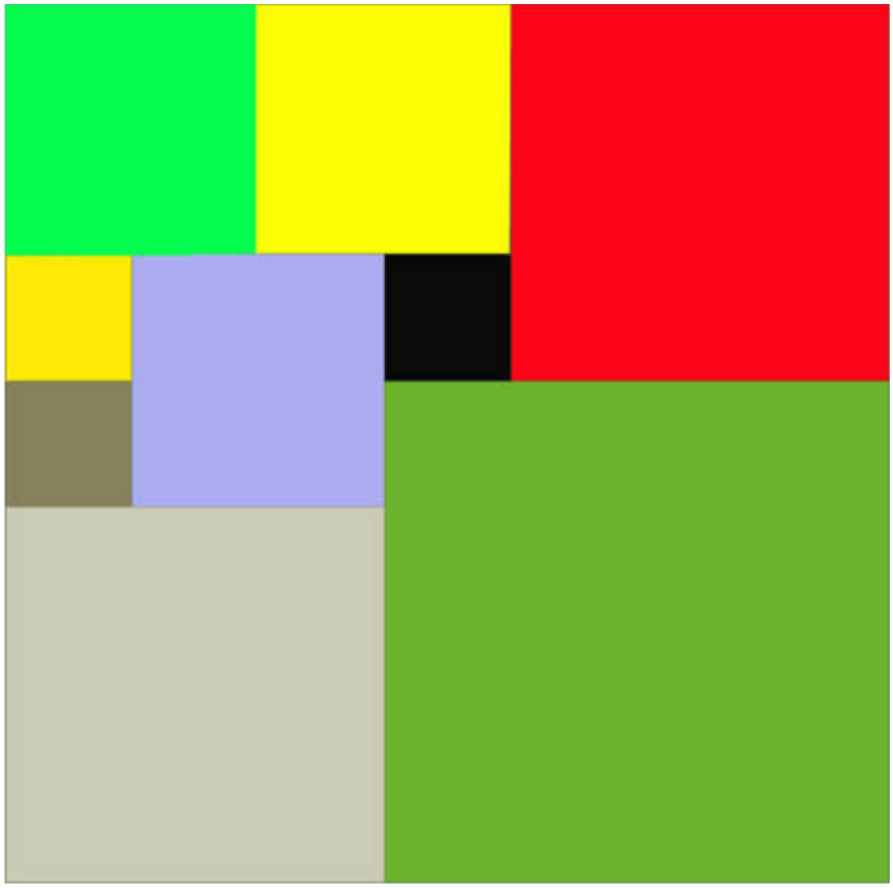


Рисунок 1 - Разбиение квадрата 7×7

Внутри квадрата не должно быть пустот, обрезки не должны выходить за пределы квадрата и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные:

Размер столешницы - одно целое число N ($2 \leq N \leq 30$).

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 и длину соответствующего обрезка (квадрата).

Решение должно находиться за разумное время (меньше минуты) для квадрата размером $2 \leq N \leq 30$.

Вар. 1и. Итеративный бэктрекинг. Поиск решения за разумное время (меньше минуты) для 2≤N≤30. Выполнение на Stepik двух заданий в разделе 2.

Описание функций и структур данных.

```
class Map;

class Piece{              //класс обрезка
friend Map;
private:
    int cellX;            //координаты вверхнего
    int cellY;            //левого угла
    int width;            //размер обрезка
    int index;            //индекс (для вывода)
    void setIndex();
public:
    int getX();           //геттеры
    int getY();
    int getWidth();
    Piece(int x, int y,  int width);
    virtual ~Piece() { }
    Piece* copy();
};
```

1) Класс **Piece** - класс обрезка(квадратика), хранит в себе координаты левого верхнего угла и размер. Имеет функцию **copy()**, которая возвращает полную копию данного квадратика. Класс необходим для заполнения поля.

```
class Map{
private:
    int** matrix;                // матрица заполненности столешницы
    int nullCell;                // счетчик "разряженности" матрицы
    int countPiece;
    std::vector<Piece*> buffer;  // буфер обрезков
    void fillNull();             // заполнение матрицы нулями.
    int width;                   // размер столешницы
    int resultOut;               // результат: список обрезков
public:
    std::vector<Piece*> resultArray; // текущее наилучшее решение
    Map(int width);
    virtual ~Map();
    bool addPiece(Piece*);           // добавление в буфер обрезка
    Piece* pullPiece();              // удаление из буфера обрезка
    bool findEmptyCell(int&, int&);  // поиск свободной клетки
    Piece* fillMax(int, int);        // "жадное" размещение обрезка
(максимально возможного размера)
    int findMinPartition();          // поиск минимального разбиения
    void optimiseEven();             // оптимизации
    void optimiseTrip();
    void optimisePrime();
    void optimiseMultipFive();
    friend std::ostream& operator<<(std::ostream &stream, const Map
&obj);
    void show();                     // вывод состояния столешницы
};
```

2) Класс **Map** - класс столешницы (квадрата). Класс хранит в себе размер столешницы, матрицу из целочисленных элементов (характеризует заполненность столешницы, если в ячейке число > 0, то она занята, если "0", то свободная), счетчик свободных ячеек, результирующий список обрезков, а так же "буферный" список. Буферный список хранит текущее найденное решение, а `resultArray` хранит наилучшее из найденных решений, если в буфере решение окажется лучше, то оно перезапишется в `resultArray`. Методы класса описаны ниже.

Описание методов класса Map:

1) `bool addPiece(Piece* );`

Функция **`addPiece()`** добавляет переданный обрезок в буферный список обрезков.

2) `Piece* pullPiece();`

Функция **`pullPiece()`** вытягивает из буферного списка последний обрезок и возвращает указатель на него.

3) `bool findEmptyCell(int&, int&);`

Функция **`findEmptyCell()`** проходит по всей матрице "заполненности" и ищет свободную клетку (в которой записан 0). Функция принимает два аргумента по ссылке, в которые при нахождении свободной клетки будут записаны соответственно ее координаты, если же свободная клетка не найдена, то функция возвращает false.

4) `Piece* fillMax(int, int);`

Функция **`fillMax()`** "жадным" образом определяет максимальный размер квадрата, который можно вставить в клетку, координаты которой принимает в качестве параметров. Функция начинает "разрастание" квадрата от переданной ячейки, как только происходит наложение на другой обрезок или выход за границы столешницы, функция "откатывает" назад и сохраняет последний возможный размер. После этого функция создает объект класса-обрезка (передавая в конструктор координаты и полученный размер) и возвращает в качестве результата ссылку на созданный объект.

5) `friend std::ostream& operator<<(std::ostream &stream, const Map &obj);`

Был перегружен оператор вывода в поток для более удобного вывода результирующего списка обрезков.

6) `int findMinPartition();`

Функция `findMinPartition()` находит минимальное разбиение столешницы на обрезки, и возвращает их количество. В начале функция вызывает соответствующий метод оптимизации, в зависимости от размер столешницы. После этого начинается поиск наилучшего разбиения. Текущее

найденное разбиение сохраняется в векторе `buffer`, лучшее из встреченных решений хранится в векторе `resultArray`, если очередное решение оказалось лучше, то оно перезаписывает содержимое `resultArray`.

Описание алгоритма.

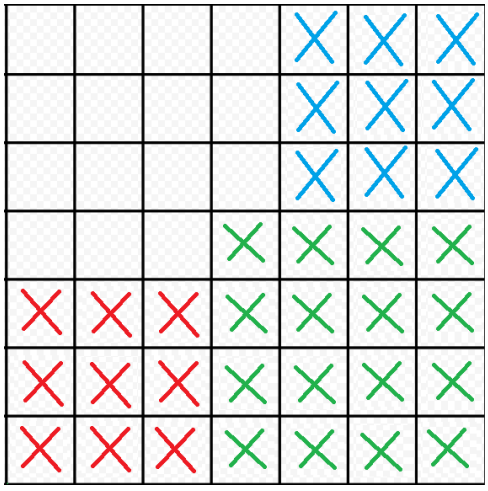
Алгоритм перебора с возвратом, рассматривающий все возможные решения для поиска оптимального реализован в функии `findMinPartition()`, которая находит минимальное разбиение столешницы на обрезки, и возвращает их количество. В начале функция вызывает соответствующий метод оптимизации, в зависимости от входных данных (размер столешницы):

1) Если сторона квадрата *N* является четной, то такой квадрат всегда можно разбить на 4 равных обрезка со сторонами *N/2*, это и будет минимальное разбиение (любой квадрат размером *>2* можно разбить на минимальное число обрезков $k \geq 4$).

2) Если сторона квадрата кратна 3, то минимальное разбиение при этом будет состоять из одного обрезка размером *2N/3* и 5 обрезков размером *N/3*.

3) Если сторона квадрата кратна 5, то "оптимизатор" частично заполнит квадрат обрезком размером *3N/5* и 2 обрезками размером *N/5*.

4) Если N - нечетное, и не подходит под все предыдущие критерии, то "оптимизатор" заполнит его тремя первыми обрезками: один в углу размером *(N-1)/2 + 1*, и двумя квадратами размером *(N-1)/2*, показано на рисунке.



После начальной оптимизации начинает работать поиск с возвратом. На каждой итерации происходит ряд действий: С помощью функции **`findEmptyCell()`** алгоритм находит свободные клетки. Свободное пространство столешницы заполняется "жадным" образом, то есть в каждую свободную клетку ставятся обрезки с максимально возможным размером с помощью метода **`fillMax()`**. Как только при расширении соответствующего обрезка он накладывается на другие или выходит за пределы столешницы, то происходит откат на последний допустимый размер. Полученное разбиение (то есть список обрезков) записывается в буфер и сравнивается по количеству элементов с вектором `resultArray`, в котором хранится последнее наименьшее решение. Если количество обрезков полученного решения меньше чем предыдущее (то есть текущее решение лучше предыдущего), то запоминаем в результирующем списке это решение.

После нахождения текущего решения происходит возврат к более короткому частичному решению: из списка вытягиваются все "единичные" обрезки и последний не единичный обрезок уменьшается на **1,** после чего все действия описанные выше повторяются. Для ускорения метода применяется примитивное выявление заведомо не подходящих решений: жадное заполнение свободного пространства столешницы прекращается, если размер (количество обрезков) текущего решение становится больше предыдущего наилучшего решения, т.к. нет смысла заполнять оставшиеся клетки.

В итоге в результирующем списке будут хранится обрезки минимального разбиения столешницы и метод вернет минимальное их число.

Сложность алгоритма по памяти будет $O(N^2 + N^2 + N^2) = O(N^2)$, т.к. алгоритм хранит матрицу "заполненности" размером $N*N$, размеры буфера текущего решения `buffer` и вектор наилучшего решения `resultArray` так же имеют в худшем случае размер $N*N$ и тем самым для них сложность по памяти $O(N^2)$, т.к. в худшем случае все обрезки могут быть единичного размера. В итоге сложность по памяти $O(N^2)$.

Сложность алгоритма по времени будет экспоненциальной. Внутри главного (внешнего) цикла с экспоненциальной сложность. выполняются следующие по сложности действия:

Цикл удаления единичных обрезков имеет линейную сложность.

Идет цикл заполнения оставшегося свободного места, внутри которого вызываются функция `findEmptyCell()`, которая имеет сложность $O(N^2)$, и функция `fillMax()`, которая имеет туже сложность равную $O(N^2)$. Так как применена простейшая оптимизация (при превышении числа обрезков по сравнению с предыдущем лучшим решением цикл прекращается), то общая сложность данного цикла будет $O(N)*(O(N^2)+O(N^2))=O(N^3)$.

Так как оба этих цикла идут последовательно внутри внешнего, то общая сложность внутренней части будет суммироваться: $(O(N^3)+O(N))=O(N^3)$.

Выводы.

Ознакомился с алгоритмом поиска с возвратом. Реализовал программу с итеративным бэктрекингом, которая находит решение задачи за разумное время.

Тестирование.

Таблица 1 - тестирование

№ теста	Входные данные	Выходные данные	Время работы сек.
1	7	9 4 4 4 5 1 3 1 5 3 1 1 2 1 3 2 3 1 2 3 3 1 3 4 1 4 3 1	0.000692
2	18	4 10 10 9 10 1 9 1 10 9 1 1 9	1.6e-05
3	2	4 2 2 1 2 1 1 1 2 1 1 1 1	3e-06
4	19	13 10 10 10 11 1 9 1 11 9 1 1 6 1 7 4 5 7 4 7 1 4 7 5 2 9 5 2 9 7 2 9 9 1 9 10 1 10 9 1	0.026139

5	29	14 15 15 15 16 1 14 1 16 14 1 1 8 1 9 7 8 9 2 8 11 5 9 1 7 9 8 1 10 8 3 13 8 3 13 11 3 13 14 2 15 14 1	1.22193
6	11	11 6 6 6 7 1 5 1 7 5 1 1 4 1 5 2 3 5 2 5 1 2 5 3 2 5 5 1 5 6 1 6 5 1	0.000468
7	23	13 12 12 12 13 1 11 1 13 11 1 1 5 1 6 7 6 1 3 6 4 2 8 4 1 8 5 5 8 10 3 9 1 4 11 10 2 11 12 1	0.128968

8	37	15 19 19 19 20 1 18 1 20 18 1 1 12 1 13 7 8 13 7 13 1 7 13 8 3 13 11 2 15 11 1 15 12 5 15 17 3 16 8 4 18 17 2 18 19 1	18.3133

Действия программа при некорректном вводе (см. рис 2):

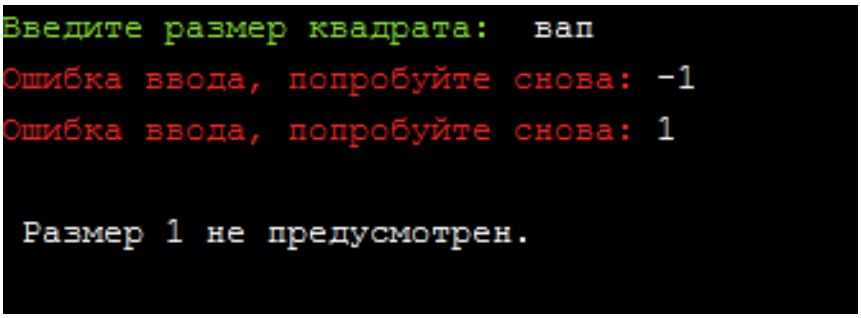


Рисунок 2 - Некорректный ввод

Пример работы программы без вывода промежуточных данных (рис. 5):

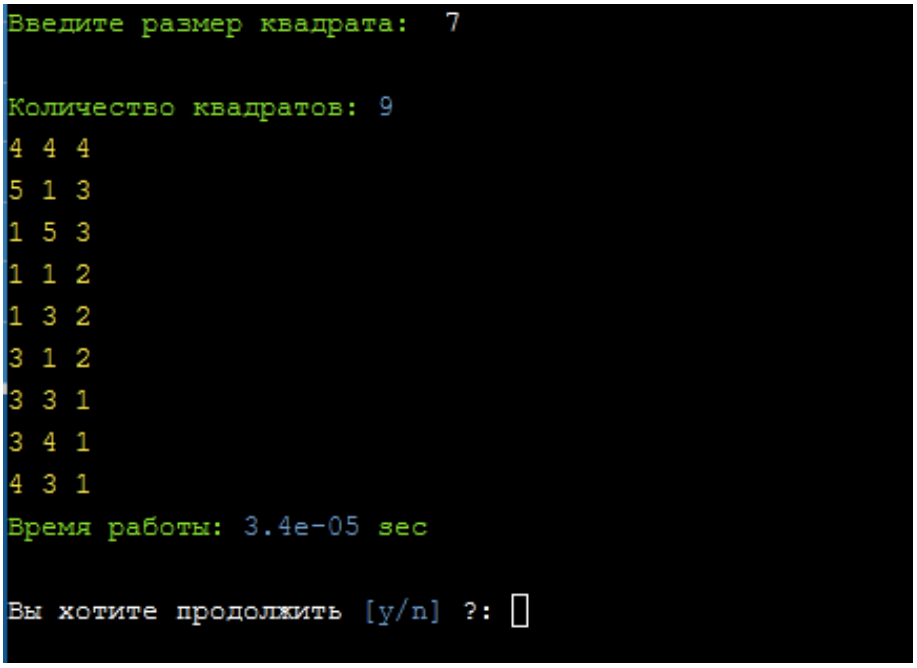


Рисунок 5 - Вывод результата (без промежуточных)

Пример работы программы с выводом промежуточных данных (рис. 6):

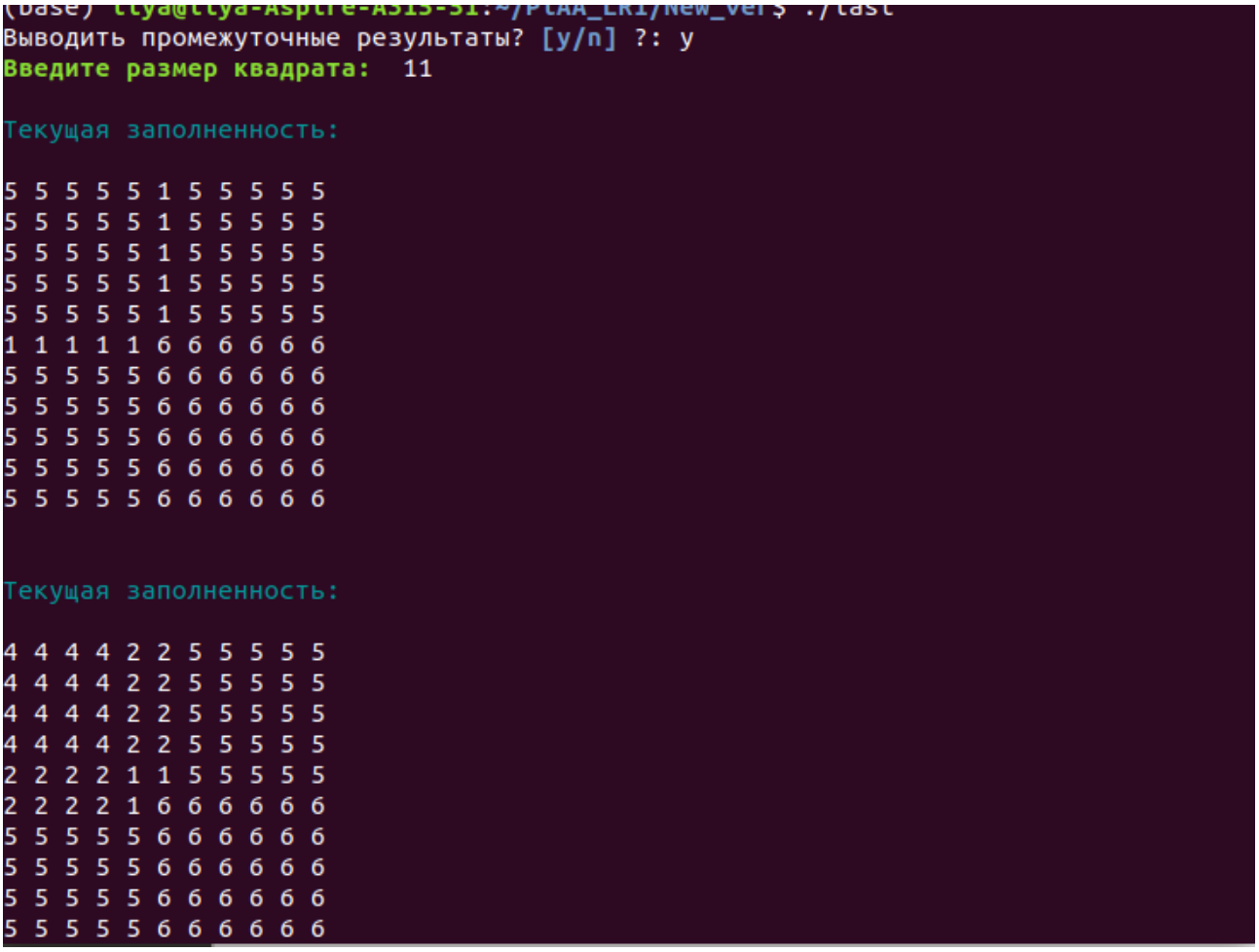
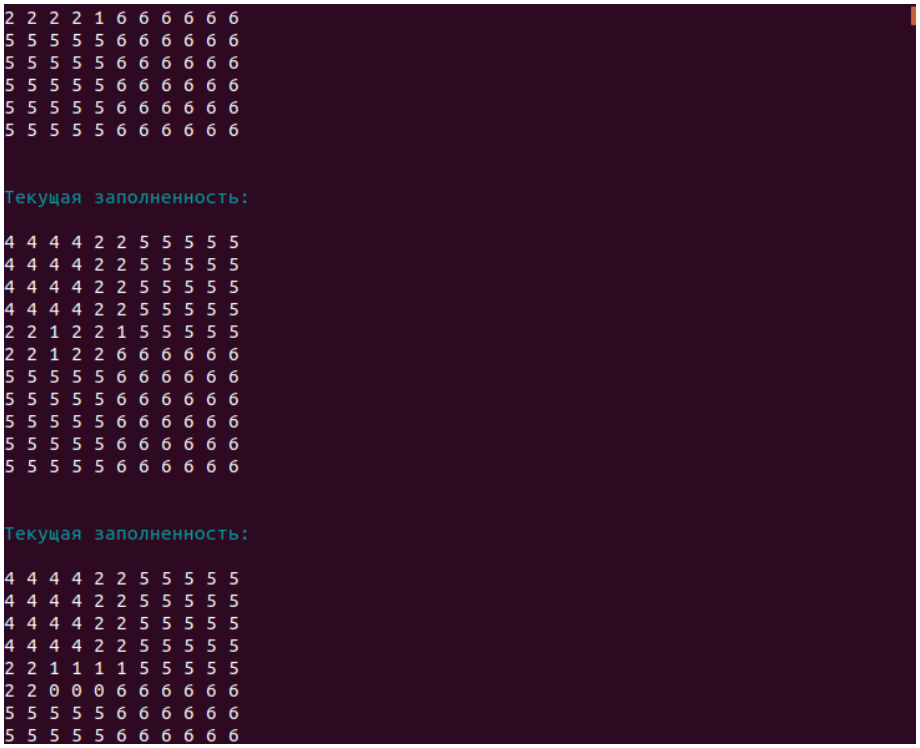
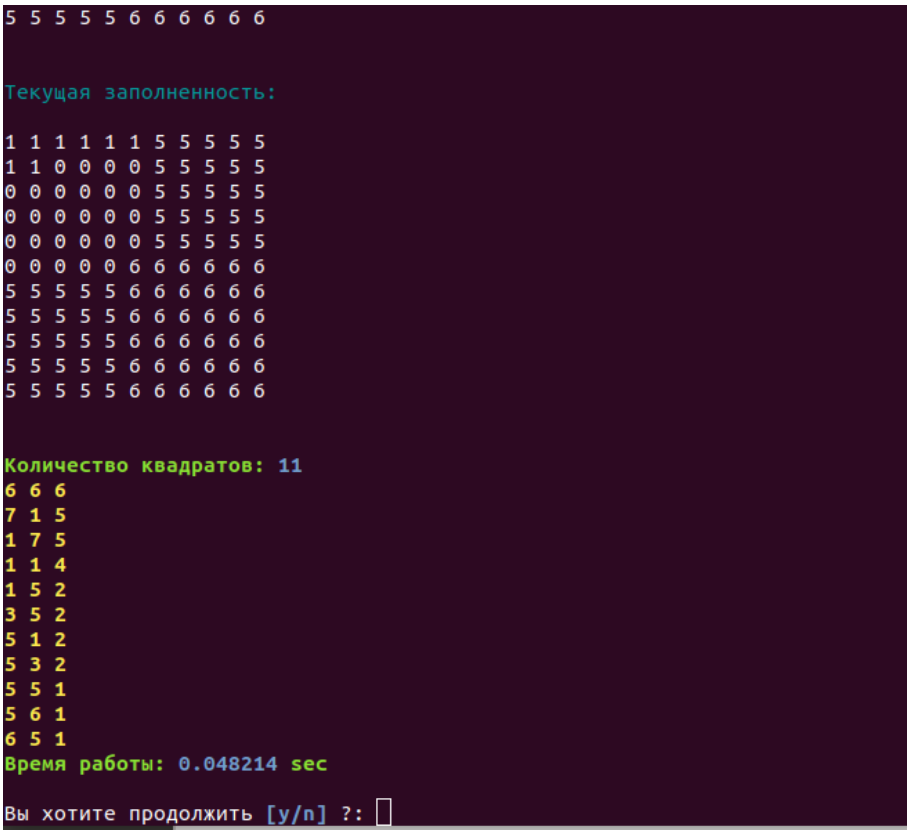
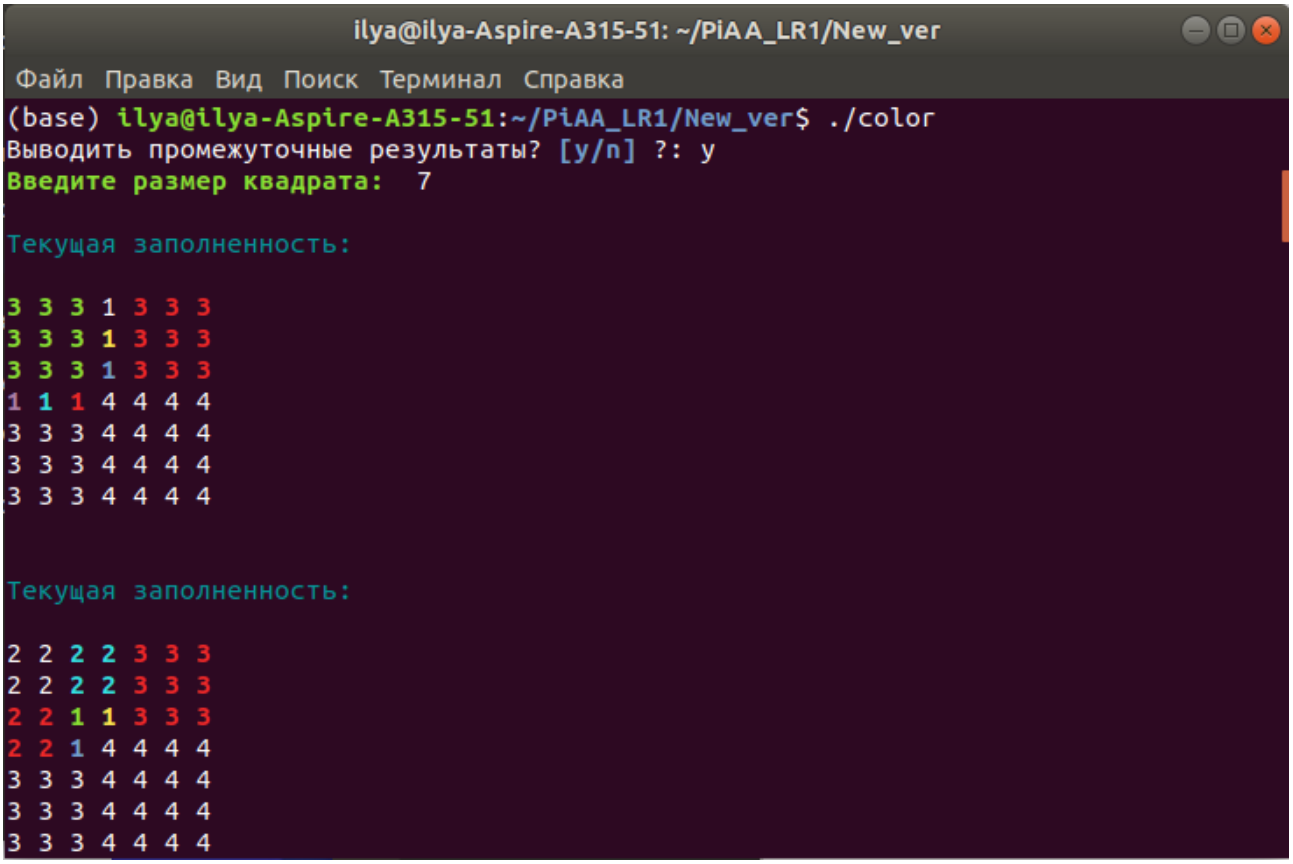


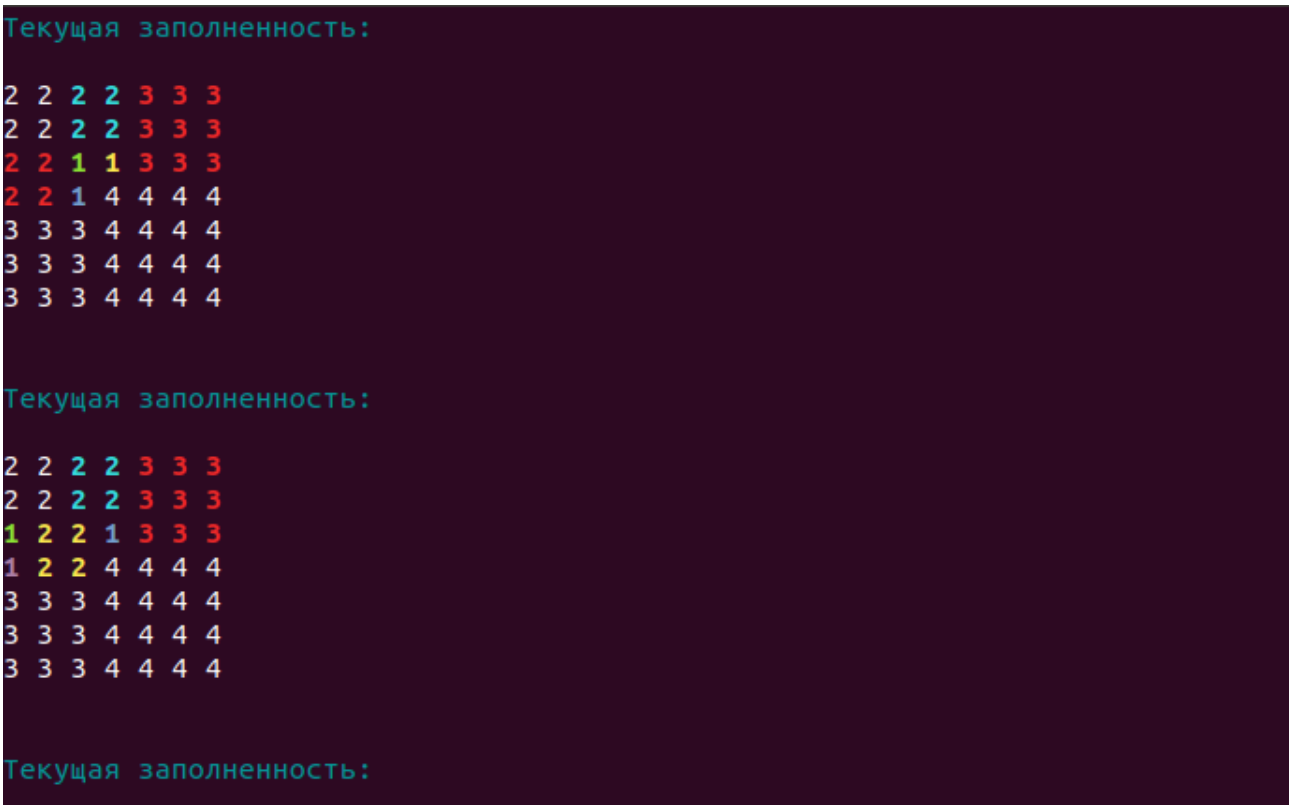
Рисунок 6 - Работа с выводом промежуточных данных





Пример с цветным выводом (файл main_color.cpp). Цвета добавлены, чтобы различать разные квадраты одного размера.


ilya@ilya-Aspire-A315-51: ~/PiAA_LR1/New_ver
Файл Правка Вид Поиск Терминал Справка
(base) ilya@ilya-Aspire-A315-51:~/PiAA_LR1/New_ver$ ./color
Выводить промежуточные результаты? [y/n] ?: y
Введите размер квадрата: 7
Текущая заполненность:
3 3 3 1 3 3 3
3 3 3 1 3 3 3
3 3 3 1 3 3 3
1 1 1 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
Текущая заполненность:
2 2 2 2 3 3 3
2 2 2 2 3 3 3
2 2 1 1 3 3 3
2 2 1 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4


Текущая заполненность:
2 2 2 2 3 3 3
2 2 2 2 3 3 3
2 2 1 1 3 3 3
2 2 1 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
Текущая заполненность:
2 2 2 2 3 3 3
2 2 2 2 3 3 3
1 2 2 1 3 3 3
1 2 2 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
Текущая заполненность:

```
3 3 3 4 4 4 4


Текущая заполненность:

1 1 1 1 3 3 3
1 2 2 0 3 3 3
0 2 2 0 3 3 3
0 0 0 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4


Текущая заполненность:

1 1 1 1 3 3 3
1 1 0 0 3 3 3
0 0 0 0 3 3 3
0 0 0 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4


Количество квадратов: 9
4 4 4
5 1 3
1 5 3
1 1 2
1 3 2
3 1 2
3 3 1
3 4 1
4 3 1
Время работы: 0.002595 sec

Вы хотите продолжить [y/n] ?:
```

Приложение (код программы).

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstring>
#include <stdlib.h>
#include <vector>

using namespace std;

class Map;

class Piece{              //класс обрезка
friend Map;
private:
    int cellX;            //координаты вверхнего
    int cellY;            //левого угла
    int width;            //размер обрезка
    int index;            //индекс (для вывода)
    void setIndex();
public:
    int getX();           //геттеры
    int getY();
    int getWidth();
    Piece(int x, int y,  int width);
    virtual ~Piece() { }
    Piece* copy();
};

// установка индекса (для вывода)
void Piece::setIndex()
{
    if (width == 1) index = 1;
    else if (width > 9) index = 9;
    else index = width;
}


Piece::Piece(int x, int y, int width)
: cellX(x), cellY(y), width(width)
{
    this->setIndex();
}

//метод копирования обрезка
Piece* Piece::copy()
{
    return new Piece(cellX, cellY, width);
}

//геттеры
int Piece::getX()
{
    return cellX;
}

int Piece::getY()
{
    return cellY;
}

int Piece::getWidth()
{
```

```
    return width;
}

bool PRINT = true;


class Map{
private:
    int** matrix;                // матрица заполненности столешницы
    int nullCell;                // счетчик "разряженности" матрицы
    int countPiece;
    std::vector<Piece*> buffer;  // буфер обрезков
    void fillNull();             // заполнение матрицы нулями.
    int width;                   // размер столешницы
    int resultOut;               // результат: список обрезков
public:
    std::vector<Piece*> resultArray; // текущее наилучшее решение
    Map(int width);
    virtual ~Map();
    bool addPiece(Piece*);           // добавление в буфер обрезка
    Piece* pullPiece();              // удаление из буфера обрезка
    bool findEmptyCell(int&, int&);  // поиск свободной клетки
    Piece* fillMax(int, int);        // "жадное" размещение обрезка (максимально
возможного размера)
    int findMinPartition();          // поиск минимального разбиения
    void optimiseEven();             // оптимизации
    void optimiseTrip();
    void optimisePrime();
    void optimiseMultipFive();
    friend std::ostream& operator<<(std::ostream &stream, const Map &obj);
    void show();                     // вывод состояния столешницы
};


void Map::show()                     // промежуточный вывод
{                                    // состояния столешницы
    for (int i = 0; i < width; i++)
    {
        cout << "\n";
        for (int j = 0; j < width; j++)
        {
            cout << matrix[i][j] << " ";
        }
    }
    cout << "\n\n";
}


void Map::fillNull()                  //заполнение нулями матрицы
{
    for (int x = 0; x < this->width; x++)
    {
        for (int y = 0; y < this->width; y++)
        {
            matrix[x][y] = 0;
        }
    }
    countPiece = 0;
    nullCell = width*width;
}


Map::Map (int width)                 //конструктор
{
```

```
    this->width = width;
    matrix = new int* [width];
    for (int i = 0; i < width; i++)
    {
        matrix[i] = new int[width];
    }
    fillNull();                     //обнуление
    nullCell = width*width;
    buffer.resize(nullCell, nullptr);
    resultArray.resize(nullCell, nullptr);
    countPiece = 0;
}


Map::~Map()
{
    for (int i = 0; i < width; i++)
        delete(matrix[i]);
    delete(matrix);
    for (int i = 0; i < width*width; i++){
        if (buffer[i]) delete(buffer[i]);
        if (resultArray[i]) delete(resultArray[i]);
    }
}



bool Map::addPiece(Piece* piece){          // метод добавления обрезка
    if (nullCell == 0) {
        return false;                     //матрица полностью заполнена
    }

    for (int x = piece->cellX; x < piece->cellX + piece->width; x++)
    {
        for (int y = piece->cellY; y < piece->cellY + piece->width; y++)
        {
            matrix[x][y] = piece->index;     //заполняем клетки текущего обрезка
        }
    }
    int num = countPiece;
    if (buffer[countPiece] != nullptr)
    {
        delete(buffer[countPiece]);
    }
    buffer[countPiece++] = piece;            //добавляем в буфферный список
    nullCell -= piece->width * piece->width; //уменьшение счетчика
"разряженности"
    return true;                             //т.к. заполнили
}



Piece* Map::pullPiece(){                    //метод удаления обрезка
    if (countPiece == 0) {                  //буфер пустой, удалять
        return NULL;                        //нечего
    }
    Piece* tmp = buffer[countPiece-1];
    for (int i = tmp->cellX; i < tmp->cellX + tmp->width; i++)
    {
        for (int j = tmp->cellY; j < tmp->cellY + tmp->width; j++)
        {
            matrix[i][j] = 0;            //обнуление клеток, которые
        }                                //занимал обрезок
```

```
    }
    countPiece--;
    nullCell += tmp->width * tmp->width;   //увеличение счетчика "разряженности"
    return tmp;
}


bool Map::findEmptyCell(int& cellX, int& cellY){ // метод поиска свободной
клетки
    for (int i = 0; i < this->width; i++)
    {
        for (int j = 0; j < this->width; j++)
        {
            if (matrix[i][j] == 0){
                cellX = i;              //присвоение переданным переменным
                cellY = j;              //координаты найденной клетки
                return true;
            }
        }
    }
  return false;                 //свободная клетка не была найдена
}


Piece* Map::fillMax(int coordX, int coordY){  //заполнение максимально возможным
    int size = 1;                             //обрезком с переданной координаты
    bool check = true;
    while(check && (size < width)){
        size++;
        check = (coordX + size - 1 < width) &&  (coordY + size - 1 < width);
//контроль выхода за границы
        if (check){
            for (int i = 0; i < size; i++)
            {
                if ((matrix[coordX+size-1][coordY+i] != 0) ||
(matrix[coordX+i][coordY+size-1] != 0))
                {
                    check = false;                      //контроль что не
накладывается на
                    break;                              //соседние обрезки (по
периметру)
                }
            }
       }
    }
    size--;
    return new Piece(coordX, coordY, size);
}



int Map::findMinPartition(){        // поиск минимального разбиения
    int resultCount = nullCell;
    if ((width % 2) == 0){          // оптимизации
        optimiseEven();
    }
    else if ((width % 3) == 0){
        optimiseTrip();
    }
    else if ((width % 5) == 0){
        optimiseMultipFive();
    }
    else{
        optimisePrime();
```

```
    }
    int tmp_x, tmp_y;
    while (nullCell){
        findEmptyCell(tmp_x, tmp_y);
        addPiece(fillMax(tmp_x, tmp_y));
    }
    resultCount = countPiece;

    for (int i = 0; i < countPiece; i++) {
        resultArray[i] = buffer[i]->copy();
    }

    if (width == 2) {
        resultOut = resultCount;
        return resultCount;
    }
    int k_While;
    while (true) {
        if(PRINT)  show();
        k_While = countPiece - 1;
        while (buffer[k_While--]->width == 1)           //убираем единичные
            pullPiece();
        k_While++;
        if (k_While < 3) break;
                                                          /* Откат назад на
размер меньше */
        Piece* poppedSquare = pullPiece();                 //ставит на единицу
меньше предыдущего
        addPiece(new Piece(poppedSquare->cellX, poppedSquare->cellY,
poppedSquare->width - 1));
        while (nullCell && (countPiece < resultCount)){
            findEmptyCell(tmp_x, tmp_y);                    //заполняем снова,
если текущее число
            addPiece(fillMax(tmp_x, tmp_y));                //обрезков
превысило предыдущее, нет смысла
        }                                                   //продолжать,
прошлое лучше -> выходим

        if (countPiece < resultCount) {
            resultCount = countPiece;
            for (int j = 0; j < countPiece; j++) {
                if (resultArray[j]) delete(resultArray[j]);
                resultArray[j] = buffer[j]->copy();          //сохраняем лучший
на данный
            }                                                //момент рзультат
        }
    }
    resultOut = resultCount;
    return resultCount;
}


void Map::optimiseEven()
{
    addPiece(new Piece(width/2, width/2, width/2));
    addPiece(new Piece(width/2, 0, width / 2));
    addPiece(new Piece(0, width/2, width/2));
}


void Map::optimiseTrip()
{
    addPiece(new Piece(0, 0,width*2/3));
    addPiece(new Piece(width*2/3, 0, width/3));
```

```
    addPiece(new Piece(0, width*2/3, width/3));
}


void Map::optimiseMultipFive()
{
    addPiece(new Piece(0, 0, width*3/5));
    addPiece(new Piece(width*3/5, 0, width/5));
    addPiece(new Piece(0, width*3/5, width/5));
}



void Map::optimisePrime()
{
    addPiece(new Piece(width/2, width/2, ceil(width/2.0)));
    addPiece(new Piece(ceil(width/2.0), 0, width/2));
    addPiece(new Piece(0, ceil(width / 2.0), width/2));
}


//перегруженный оператор вывода в поток
ostream& operator<<(ostream &stream, const Map &obj)
{
    for (int i = 0; i < obj.resultOut; i++)
    {
        stream << "\033[1;33m";
        stream << obj.resultArray[i]->getX()+1 << " ";
        stream << obj.resultArray[i]->getY()+1 << " ";
        stream << obj.resultArray[i]->getWidth();
        stream << endl;
    }
    stream << "\033[0m";
}

// класс для диалога с пользователем, не относится к алгоритму
class Dialog{
  private:
    int fieldSize;
    Map* field;
  public:
    Dialog();
    virtual ~Dialog();
    void dialog();
    bool isDigit(const char* str);
    void update();
    void toLower(string &str);

};


Dialog::Dialog() : fieldSize(0), field(nullptr)
{
}

Dialog::~Dialog()
{
    if (field != nullptr){
        delete field;
    }
}

void Dialog::update()
{
```

```
    if (field != nullptr){
        delete field;
        field = nullptr;
    }
}


void Dialog::toLower(string &str){
    int i = 0;
    while (str[i]){
        str[i] = tolower(str[i]);
        i++;
    }
}


bool Dialog::isDigit(const char* str){
    if (((str[0] == '+') && strlen(str)>1) || isdigit(str[0])){
       for (int i = 1; i < strlen(str) ; i++){
            if (isdigit(str[i]) == 0) return false;
    }
    return true;
    }
    else
        return false;
}



void Dialog::dialog()
{
   string str;
   do{
      Map* field_;
      cout  << "Выводить промежуточные результаты? \033[1;34m[y/n]\033[0m ?: ";
      cin >> str;
      toLower(str);
      while (str!="y" && str!="n"){
          cout << "\nОшибка, введите [y/n] :";
          cin >> str; toLower(str);
      }
      if (str == "n")  PRINT = false;
      cout << "\033[1;32mВведите размер квадрата:\033[0m  ";
      cin >> str;
      while (!isDigit(str.c_str()))
        {
          cout << "\033[1;31mОшибка ввода, попробуйте снова: \033[0m" ;
          cin >> str;
        }

      fieldSize = atoi(str.c_str());
      if (fieldSize == 1){
        cout << "\n Размер 1 не предусмотрен.";
        return;
      }
      field_ = new Map(fieldSize);
      size_t start_time = clock();
      int result = field_->findMinPartition();
      size_t end_time = clock();
      size_t time = end_time-start_time;
      cout << "\n\033[1;32m" << "Количество квадратов: \033[1;34m" << result <<
"\033[0m" << endl;
      cout << (*field_);
```

```
      cout << "\033[1;32mВремя работы: \033[1;34m" << ((float)time)/
CLOCKS_PER_SEC << "\033[1;32m sec\033[0m";
      cout << "\n\n" << "Вы хотите продолжить \033[1;34m[y/n]\033[0m ?: ";
      cin >> str;
      toLower(str);
      while (str!="y" && str!="n"){
          cout << "\nОшибка, введите [y/n] :";
          cin >> str; toLower(str);
      }
      cout << "\n";
      delete(field_);
      field_ = nullptr;
      PRINT = true;
    }while(str != "n");
}




int main(){
    Dialog work;
    work.dialog();
    return 0;
}
```