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

Вариант 4и

Студент гр. 8382 Торосян Т.А.

Преподаватель Фирсов М.А.

Санкт-Петербург

2020

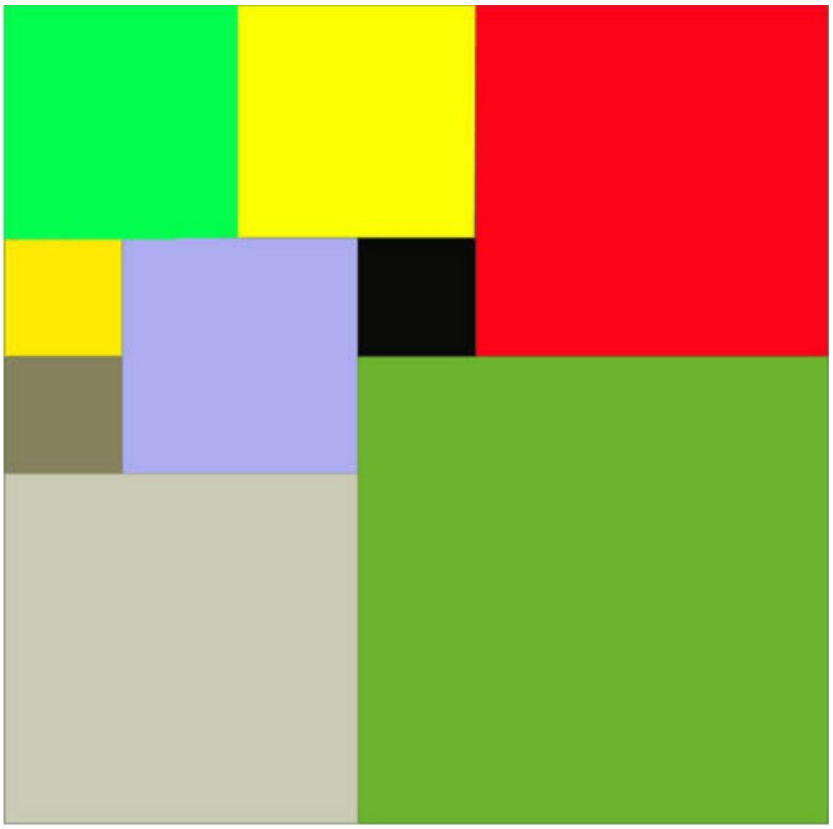
Цель работы.

Изучение алгоритма поиска с возвратом.

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≤N≤20).

Выходные данные

Число K, задающее минимальное количество обрезков-квадратов, из которых можно построить столешницу (квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x, y и w, задающие координаты левого верхнего угла (1≤x, y≤N) и сторону соответствующего обрезка.

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

Индивидуализация.

Итеративный бэктрекинг. Расширение задачи на прямоугольные поля,ребра квадратов меньше ребер поля. Подсчет количества вариантов покрытия минимальным числом квадратов.

Описание алгоритма.

Работа программа основана на алгоритме поиска с возвратом. Он заключается в полном переборе всех возможных вариантов заполнения прямоугольника квадратами.

Перебор всех расстановок квадратов представлен в виде поиска в глубину с использованием дерева, вершинами которого являются все возможные квадраты, которые можно вставить в точку (0, 0) (слева направо, сверху вниз). Корень дерева - пустая вершина, с нее запускается алгоритм для первой пустой точки: ищутся все квадраты, которые можно в ней разместить – они помещаются в стек и становятся новыми вершинами дерева. Затем выбирается вершина, в которой хранится самый большой еще не использованный квадрат, он вставляется в прямоугольник, ищется следующая пустая точка и для нее все повторяется.

Когда прямоугольник полностью заполнен, запоминается количество вставленных квадратов, если найдено решение лучше минимального на данный момент, то количество квадратов запоминается, если количество квадратов

равно текущему минимальному квадрированию, то счетчик количества вариантов увеличивается. После этого программа поднимается обратно по дереву и приступает к еще не посещенным вершинам.

Сложность алгоритма.

На каждом шаге мы пытаемся вставить квадраты размеров от n – 1 до 1, тогда максимальный размер стека будет составлять n * m, где n и m высота и ширина соответственно, тогда сложность по кол-ву операций будет составлять O($(m \times n)^{m \times n}$), а по памяти O($m \times n$).

Использованные оптимизации.

При обходе дерева, глубина ветви является количеством уже вставленных квадратов, если глубина ветки становится равной уже найденному мощению, а прямоугольник полностью не замощен, то дальше вставлять квадраты нет смысла и нужно подниматься выше и искать другие варианты.

Описание структур данных.

Структура, которая хранит координаты поставленного квадрата и его размер.

```
struct Square{
    int x; // координаты верхней левой точки
    int y; //  вставленного квадрата
    int size; // размер квадрата
};
```

Структура, которая хранит все данные о фигуре, которую нужно замостить

```
struct Figure {
    int** rectangle; // массив ячеек
    прямоугольника, хранящих цвет текущего
    квадрата
    int summary; // площадь, которая закрашена
    int N, M; // размеры фигуры
    stack <Square> squareStack; // координаты и размеры квадратов уже
} figure;                       //  вставленных в прямоугольник
```

Описание функций.

Функция `void maxInsert(int x, int y)` ищет все возможные квадраты от 1 до n - 1, которые можно вставить в точку с координатами x, y и кладет их на стек `sizeSquare`.

Функция `void clear(Trio a)` удаляется из прямоугольника квадрат с размером a.size, левый верхний угол которого находится в точке a.x, a.y.

Функция `void insert(int x, int y, int size, int color)` вставляет в прямоугольник квадрат размером size в точку (x, y), и использует для этого цифру-цвет color.

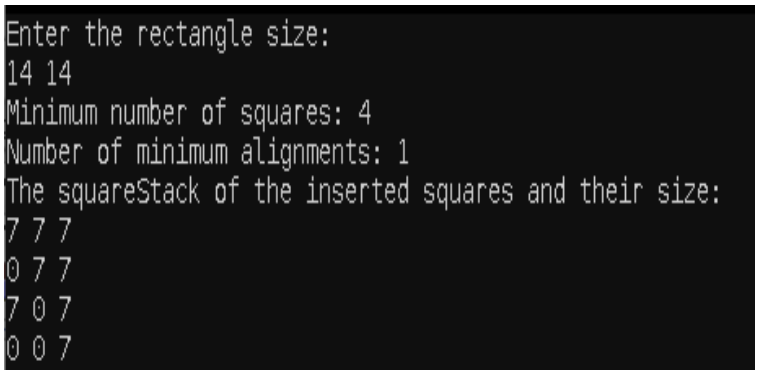
Функция `pair<int, int> tiling()` ищет минимальное разбиение и возвращает пару значений — минимальное разбиение и количество вариантов покрытия минимальным числом квадратов.

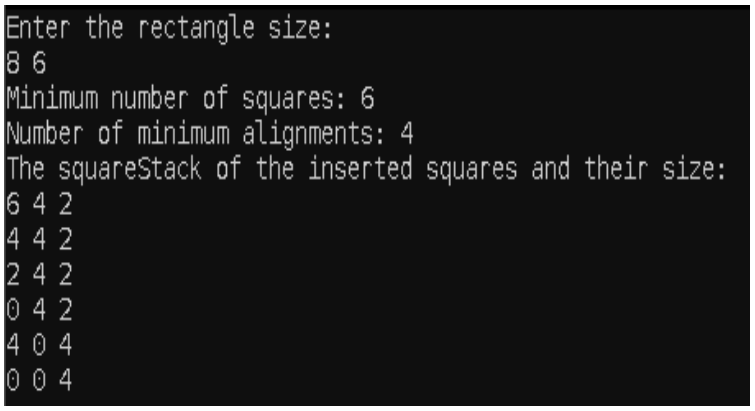
Частичные решения.

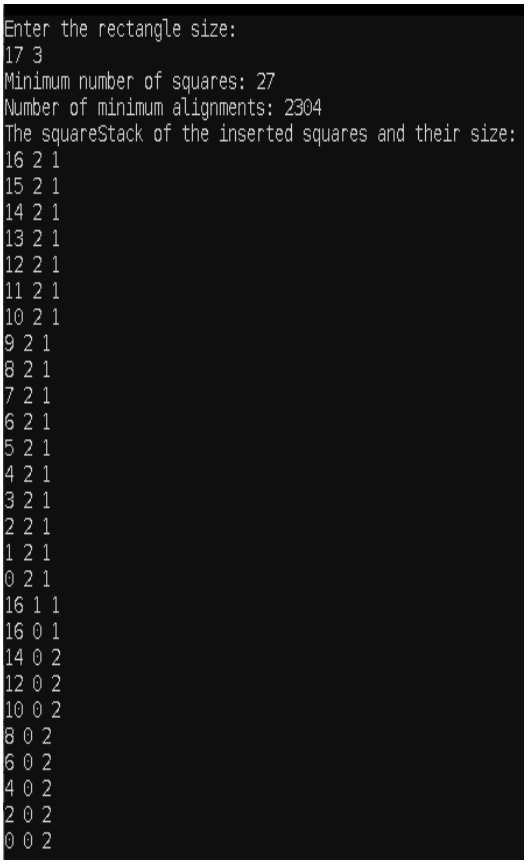
Частичные решения хранятся в стеке stack <Trio> currentSolutions

Тестирование.

```
Enter the rectangle size:
5 20
Minimum number of squares: 19
Number of minimum alignments: 168
The squareStack of the inserted squares and their size:
19 4 1
18 4 1
16 3 2
14 3 2
12 3 2
10 3 2
8 3 2
6 3 2
4 3 2
2 3 2
0 3 2
18 2 2
18 0 2
15 0 3
12 0 3
9 0 3
6 0 3
3 0 3
0 0 3
```







Вывод.

В ходе выполнения лабораторной работы, в процессе написания программы, решающей задачу квадрирования прямоугольников, был изучен алгоритм поиска с возвратом.

Приложение А.

```
#include <iostream>
#include <algorithm>
#include <stack>
#ifdef _WIN32
#include <windows.h>
#define cls() system("cls")
#else
#define cls() ;
#endif // _WIN32
using namespace std;

struct Square{
    int x;
    int y;
    int size;
};

struct Figure {
    int** rectangle;
    int summary;
    int N, M;
    stack <Square> squareStack;
} figure;

stack <pair<int, bool>> availableSquaresStack; // стек на котором хранятся размеры квадратов,которые могут быть вставлены в точку(координаты точки получаются из структуры Figure) и был ли он уже поставлены в фигуру.

stack <Square> currentSolutions; // стек, в котором хранится текущее минимальное разбиение фигуры

void print()
{
    for (int i = 0; i < figure.N; i++) {
        for (int j = 0; j < figure.M; j++) {
            cout << figure.rectangle[i][j] << ' ';
        }
        cout << endl;
    }
    //getchar();
    //cls();
}


void maxInsert(int x, int y)// функции передается точка
{                           // функия ищет размеры всех квадратов, которые
    int size;                                   // можно вставить в точку и при этом не
                            // этом не перекрыть другие квадраты, которые
                            //уже стоят в фигуре
    for (size = 1; size <= figure.N - 1; size++)
    {
        if (y + size > figure.N)
            return;

        if (x + size > figure.M)
            return;

        for (int i = y; i < y + size; i++)
            for (int j = x; j < x + size; j++)
            {
                if (figure.rectangle[i][j] != 0)
                    return;
            }
        availableSquaresStack.push(make_pair(size, false));
    }
}


void clear(Square a) // функия, которая удаляет квадрат из фигуры
{
    figure.summary -= a.size * a.size;

    for (int i = a.y; i < a.y + a.size; i++)
        for (int j = a.x; j < a.x + a.size; j++)
            figure.rectangle[i][j] = 0;
}


void insert(int x, int y, int size, int color) // функция, которая вставляет квадрат в фигуру
{
```

```
    figure.summary += size * size;

    for (int i = y; i < y + size; i++)
        for (int j = x; j < x + size; j++)
            figure.rectangle[i][j] = color;
}


pair<int, int> tiling() // функция, которая перебирает все возможные варианты мощения квадрата и ищет среди них минимальное
{
    int x = 0, y = 0;
    int color = 1;
    int numberColorings = 0;
    int numberSquares = figure.N * figure.M + 1;
    bool flag;

    maxInsert(x, y);

    do{ // цикл, который работает пока не будут проверенны все комбинации квадратов
        flag = false;

        for (y = 0; y < figure.N; y++) { // циклы, которые ищут первую, пустую клетку и вставляют в нее квадраты
            for (x = 0; x < figure.M; x++)
            {
                if (figure.rectangle[y][x] == 0)
                {
                    flag = true;

                    if (availableSquaresStack.top().second)
                        maxInsert(x, y);

                    insert(x, y, availableSquaresStack.top().first, color);
                    figure.squareStack.push(Square{x, y, availableSquaresStack.top().first});
                    availableSquaresStack.top().second = true;

                    color++;

                    //cout << "Insert a new square into the Figure"<< endl;
                    //print();

                    break;
                }
            }
            if (flag)
                break;
        }
        // если в фигуру вставленно квадратов больше, чем уже в каком-то из известных разбиений, то происходится откат к другим вариантам разбияния
        if (  color - 1 == numberSquares && figure.summary != figure.M * figure.N)
        {
            //cout << "This can be done with less amount of squares. Backtracking...\n";
            while ( !availableSquaresStack.empty() && availableSquaresStack.top().second)
            {
                availableSquaresStack.pop();
                clear(figure.squareStack.top());
                figure.squareStack.pop();
                color--;

            }
            //print();
        }
        // если фигура была полность покрыта, тогда проверяется минимальное ли это разбиение, если да, то оно запоминается, если разбиение на столько квадратов уже сущетсвует, то счетчик вариантон разбиение увеличивается
        if ( !availableSquaresStack.empty() && figure.summary == figure.M * figure.N)
        {
            //cout << "The figure is tiled with the number of squares less than or equal to the current split. Saving the alignment of squares and wiping out.\n";
            if (numberSquares == color - 1)
            {
                numberColorings++;
            } else{
                currentSolutions = figure.squareStack;
                numberSquares = color - 1;
                numberColorings = 1;
            }


            while ( !availableSquaresStack.empty() && availableSquaresStack.top().second )
            {
                availableSquaresStack.pop();
                clear(figure.squareStack.top());
```

```
                figure.squareStack.pop();
                color--;
            }
            //print();

        }
    }while (!availableSquaresStack.empty());

    return make_pair(numberSquares, numberColorings);
}


int main() {
    pair <int, int> ans;
    cout << "Enter the rectangle size:\n";
    cin >> figure.N >> figure.M;

    if (figure.N > figure.M)
    {
        swap(figure.N, figure.M);
    }

    figure.rectangle = new int * [figure.N];

    for (int i = 0; i < figure.N; i++)
    {
        figure.rectangle[i] = new int[figure.M];
    }

    for (int i = 0; i < figure.N; i++)
        for (int j = 0; j < figure.M; j++)
        {
            figure.rectangle[i][j] = 0;
        }

    ans = tiling();

    cout << "Minimum number of squares: " << ans.first << endl
         << "Number of minimum alignments: " << ans.second << endl;
    cout << "The squareStack of the inserted squares and their size:\n";
    for (int i = 0; i < ans.first; i++)
    {
        cout << currentSolutions.top().x << ' ' << currentSolutions.top().y << ' ' << currentSolutions.top().size << endl;
        currentSolutions.pop();
    }

    return 0;
}
```