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. 8303 _____ Сенюшкин Е.В

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

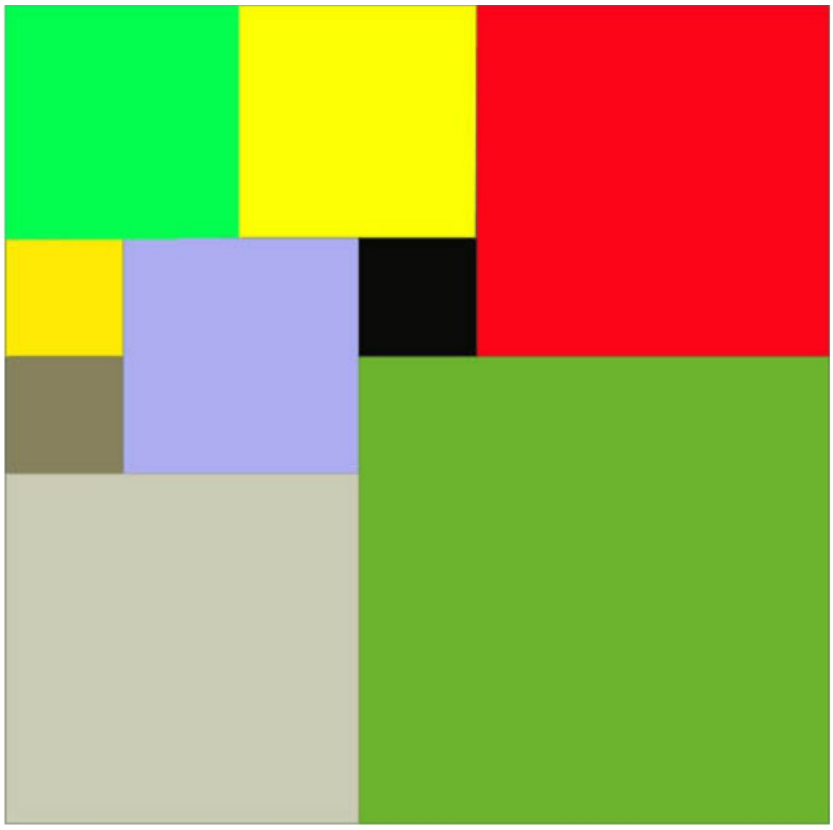
Цель работы.

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

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≤x, y≤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

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 комбинаций квадратов можно представить в виде поиска в глубину. Для дерева, вершинами которого являются все возможные размеры квадратов, которые можно вставить в первую пустую точку (с лева на права, сверху вниз). Корень дерева пустая вершина, с нее запускается алгоритм для первой пустой точки ищутся все квадраты, которые в нее могут вставиться и кладутся на стек, это новые вершины. Переходим в вершину, в которой хранится самый большой еще не посещенный 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 разложение, то количество квадратов запоминается, если количество квадратов равно текущему минимальному квадрированию, то счетчик количества минимального покрытия увеличивается. После начинаем подниматься обратно по дереву и посещать еще не посещенный вершины.

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

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

```
struct Figure {
    int** rectangle; // прямоугольник в памяти
    int delivered; // площадь, которая закрашена
    int N, M; // размеры фигуры
    stack <Trio> coordinates; // 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, левый верхний угол которого находится в точке a.x, a.y и размером a.size.

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

Тестирование.

```
5 20
Minimum number of squares: 19
Number of minimum constellations: 168
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

Process finished with exit code 0
```

```
13 13
Minimum number of squares: 11
Number of minimum constellations: 8
8 11 2
6 11 2
10 10 3
9 10 1
6 8 3
6 7 1
0 7 6
9 6 4
7 6 2
7 0 6
0 0 7

Process finished with exit code 0
```

```
14 12
Minimum number of squares: 5
Number of minimum constellations: 4
4 8 4
0 8 4
8 6 6
8 0 6
0 0 8

Process finished with exit code 0
```

```
18 2
Minimum number of squares: 36
Number of minimum constellations: 1
17 1 1
16 1 1
15 1 1
14 1 1
13 1 1
12 1 1
11 1 1
10 1 1
9 1 1
8 1 1
7 1 1
6 1 1
5 1 1
4 1 1
3 1 1
2 1 1
1 1 1
0 1 1
17 0 1
16 0 1
15 0 1
14 0 1
13 0 1
12 0 1
11 0 1
10 0 1
9 0 1
8 0 1
7 0 1
6 0 1
5 0 1
4 0 1
3 0 1
2 0 1
1 0 1
0 0 1
```

Вывод.

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

Приложение А.

```
#include <iostream>
#include <algorithm>
#include <stack>

using namespace std;

struct Trio{ // структура, которая хранит координаты поставленного квадрата и его размер
    int x;
    int y;
    int size;
};

struct Figure { // структура, которая хранит все данные о фигуре, которую нужно замостить
    int** rectangle;
    int delivered;
    int N, M;
    stack <Trio> coordinates;
} figure;

stack <pair<int, bool>> sizeSquare; // 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 <Trio> ansCoordinates; // стек, в котором хранится текущие 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();
}


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;
            }
        sizeSquare.push(make_pair(size, false));
    }
}


void clear(Trio a) // функия, которая удаляется квадрат из фигуры
{
    figure.delivered -= 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.delivered += 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 (sizeSquare.top().second)
```

```
                maxInsert(x, y);

            insert(x, y, sizeSquare.top().first, color);
            figure.coordinates.push(Trio{x, y, sizeSquare.top().first});
            sizeSquare.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.delivered != figure.M * figure.N)
    {
        cout << "More squares than the minimum known partition.Delete squares\n";
        while ( !sizeSquare.empty() && sizeSquare.top().second)
        {
            sizeSquare.pop();
            clear(figure.coordinates.top());
            figure.coordinates.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 ( !sizeSquare.empty() && figure.delivered == figure.M * figure.N)
    {
        cout << "The figure is tiled with the number of squares less than or equal to the current split. Remember the number of squares and delete.\n";
        if (numberSquares == color - 1)
        {
            numberColorings++;
        } else{
            ansCoordinates = figure.coordinates;
            numberSquares = color - 1;
            numberColorings = 1;
        }


        while ( !sizeSquare.empty() && sizeSquare.top().second )
        {
            sizeSquare.pop();
```

```
                clear(figure.coordinates.top());
                figure.coordinates.pop();
                color--;
            }
            print();

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

    return make_pair(numberSquares, numberColorings);
}


int main() {
    pair <int, int> ans;
    cout << "Enter the size of the rectangle:\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 constellations: " << ans.second << endl;
    cout << "The coordinates of the inserted squares and their size:\n";
    for (int i = 0; i < ans.first; i++)
    {
        cout << ansCoordinates.top().x << ' ' << ansCoordinates.top().y << ' ' << ansCoordinates.top().size << endl;
        ansCoordinates.pop();
    }

    return 0;
}
```