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. 9383 _____ Орлов Д.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения исходного квадрата минимальным количеством 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. (рисунок 1)

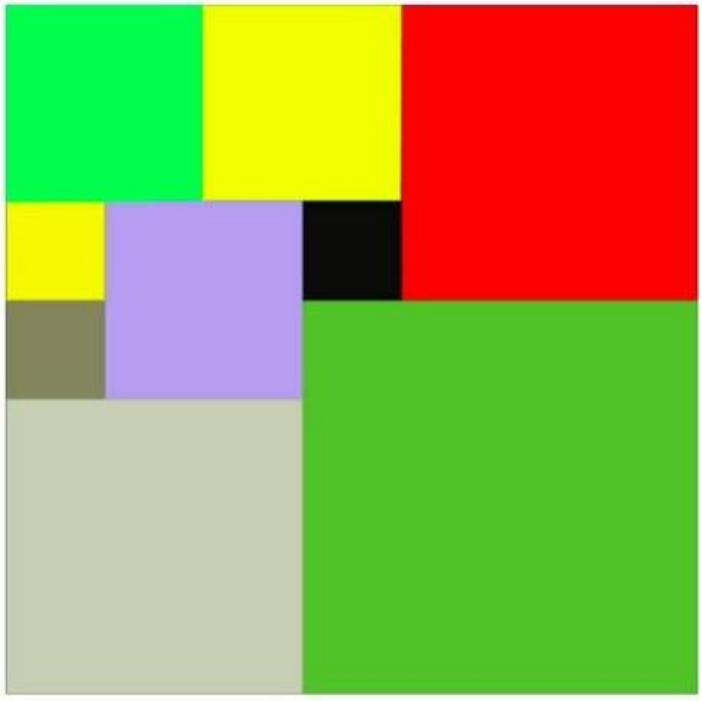

Рисунок 1- пример построения столешницы из 9 обрезков

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 $N(2 \le N \le 20)$.

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

Вариант 4и.

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

Основные теоретические положения.

Бэктрекинг (поиск с возвратом) – это общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве.

Описание метода:

Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют.

Недостатки:

Метод поиска с возвратом является универсальным. Достаточно легко проектировать и программировать алгоритмы решения задач с использованием этого метода. Однако время нахождения решения может быть очень велико даже при небольших размерностях задачи (количестве исходных данных), причём настолько велико (может составлять годы или даже века), что о практическом применении не может быть и речи. Поэтому при проектировании таких алгоритмов, обязательно нужно теоретически оценивать время их работы на конкретных данных. Существуют также задачи выбора, для решения которых можно построить уникальные, «быстрые» алгоритмы, позволяющие быстро получить решение даже при больших размерностях задачи. Метод поиска с возвратом в таких задачах применять неэффективно.

Выполнение работы:

В программе использованы следующие структуры:

struct Square — для хранения информации о квадратах (длина ребра, координаты еговерхнего левого угла по x и y)

struct Field — для хранения данных поля (длина, ширина, количество способов замощения минимальным количеством квадратов) и реализации итеративного бэктрекинга с помощью методов класса. Также класс хранит вектор поставленных квадратов, вектор лучшей расстановки квадратов и вектор векторов для реализации поля. Реализован конструктор класса, создающий матрицу типа bool размера n на m для хранения информации о заполнении поля.

В программе были реализованы следующие методы:

void printField () — выводит на экран поле;

bool isFree (int x, int y) – проверяет, свободна ли клетка с координатами x,y;

void setFree(int x, int y) — делает клетку с координатами x,y свободной;

void setBusy (int x, int y) — делает клетку с координатами x,y занятой;

void buildSquare (int x, int y, int sizeS) — с помощью метода setBusy строит квадрат: x и y – координаты левого верхнего угла, sizeS – длина ребра квадрата;

void destroyLastSquare () - удаляет последний поставленный квадрат, очищая поле с помощью метода setFree.

Square findNextFreeCell () — возвращает переменную типа Square, которая хранит в себе координаты следующей пустой клетки, куда можно поставить квадрат, либо {-1, -1, -1}, что означает полностью заполненное поле.

int findMaxSizeOfSquare (int x, int y, int maxSize) - метод возвращает максимальный размер квадрата, который возможно поставить в клетку с координатами x и y.

int backtracking () - метод, реализующий итеративный бэктрекинг с помощью всех вышеописанных функций.

Описание алгоритма работы программы:

Программа на вход получает числа n и m — размеры исходного поля. Заводим переменную maxSize, в которой будем хранить размер максимально возможного квадрата. Также переменную full, которая равна 1, когда поле полностью заполнено и 0, когда в поле есть пустые клетки. Объявляем переменную newSquare типа Square, которая будет хранить информацию о последнем удаленном квадрате.

Затем начинается цикл, заканчивающийся, когда массив квадратов, поставленных на поле, опустеет. В этом цикле объявляется другой цикл, который ставит на поле максимально возможные для определенной клетки квадраты. Этот цикл заканчивается, когда поле заполняется полностью, или мы уже поставили слишком много квадратов (размер массива с наилучшим замощением квадратов становится меньше массива с текущим замощением).

Затем происходит обработка полученной информации. Если нашлось замощение меньшим количеством квадратов, то запоминаем его. Если нашлось замощение таким же количеством квадратов, то увеличиваем счетчик counter на единицу.

После этого удаляются все квадраты, у которых длина ребра равна 1. После чего удаляется и следующий квадрат, на место которого ставится квадрат с размером на 1 меньше. Таким образом происходит итеративный бэетрекинг: произведен полный перебор вариантов замощения поля квадратами.

В программе были использованы следующие оптимизации:

1) Чтобы не перебирать все возможные варианты, было решено останавливать цикл, когда количество квадратов становится больше минимального.

Тестирование

1) Входные данные: 2 5
   Выходные данные:
   10
   1 1 1
   2 1 1
   1 2 1
   2 2 1
   1 3 1
   2 3 1
   1 4 1
   2 4 1
   1 5 1
   2 5 1

   Количество вариантов покрытия минимальным числом квадратов = 1

2) Входные данные: 5 7

   Выходные данные:

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

   Количество вариантов покрытия минимальным числом квадратов = 2

3) Входные данные: 3 3

   Выходные данные:
   6
   1 1 2
   3 1 1
   3 2 1
   1 3 1
   2 3 1
   3 3 1

   Количество вариантов покрытия минимальным числом квадратов = 4

4) Входные данные: 9 10
   Выходные данные:
   6
   1 1 5
   6 1 4
   6 5 4
   1 6 5
   6 9 2
   8 9 2

   Количество вариантов покрытия минимальным числом квадратов = 6

Вывод.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным количеством квадратов. Произведено расширение задачи до полей прямоугольного размера. Реализован подсчет количества вариантов покрытия минимальным числом квадратов.

Приложение А
Исходный код программы

Название файла: main.cpp

```
#include "lab1.h"

using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    int n,m;
    cin>>n>>m;
    int div = 1;

    if(n % 2 == 1 || m % 2 == 1)
```

```
    {
        for (int i = 2; i <= sqrt(min(n,m)); i++)
        {
            if (min(n,m) % i == 0 && max(n,m) % i == 0)
            {
                if(n==m)
                {
                    div *= max(i,n/i);
                    n /= max(i,n/i);
                    m /= max(i,m/i);
                }
                else
                {
                    div *= i;
                    n /= i;
                    m /= i;
                }
            }
        }
    }

    Field F(n,m);
    F.backtracking();

    cout<<F.bestSquares.size()<<"\n";
    for(int i = 0; i < F.bestSquares.size(); i++)
    {
        cout<<F.bestSquares[i].x*div + 1<<" "<<F.bestSquares[i].y*div +
1<<" "<<F.bestSquares[i].sizeSquare*div<<"\n";
    }

    cout<<"--------------------\n";
    cout<<"Количество вариантов покрытия минимальным числом квадратов = "<<
F.counter<<"\n";

    return 0;
}
```

Название файла: lab1.h

```
#include <iostream>
#include <vector>
#include <cmath>


using namespace std;


struct Square
{
    int sizeSquare;
    int x;
    int y;
};
```

```
struct Field
{
public:

    int Width, Height;
    int counter = 0;

    vector <vector <bool>> table;
    vector <Square> Squares;
    vector <Square> bestSquares;

    Field(int w, int h);

    void printField();

    bool isFree(int x, int y);
    bool isFull();

    void setFree(int x, int y);
    void setBusy(int x, int y);

    void buildSquare(int x, int y, int sizeS);
    void destroyLastSquare();

    int findMaxSizeOfSquare(int x, int y, int maxSize);

    Square findNextFreeCell();


    void backtracking();

};
```

Название файла: lab1.cpp

```
#include "lab1.h"


Field::Field(int w, int h)
{
    this->Width = w;
    this->Height = h;

    for(int i = 0; i < this->Height; i++)
    {
        this->table.push_back(vector<bool>());
        for(int j = 0; j < this->Width; j++)
        {
            this->table[this->table.size()-1].push_back(0);
        }
    }
}

void Field::printField()
{
    for (int i = 0; i < this->Height; i++)
    {
        for (int j = 0; j < this->Width; j++)
        {
            std::cout << this->table[i][j] << " ";
```

```
        }
        std::cout <<"\n";
    }
}

bool Field::isFree(int x, int y)
{
    if(this->table[y][x])
    {
        return false;
    }
    else
    {
        return true;
    }
}

void Field::setFree(int x, int y)
{
    this->table[x][y] = false;
}

void Field::setBusy(int x, int y)
{
    this->table[x][y] = true;
}


void Field::buildSquare(int x, int y, int sizeS)
{
    if(sizeS > 0 && sizeS < min(this->Width, this->Height))
    {
        this->Squares.push_back({sizeS, x, y});
        for(int i = y; i < y+sizeS; i++)
        {
            for(int j = x; j < x+sizeS; j++)
            {
                this->setBusy(i,j);
            }
        }
    }
}

void Field::destroyLastSquare()
{
    Square S;
    S = this->Squares.back();
    this->Squares.pop_back();
    for(int i = S.y; i < S.y+S.sizeSquare; i++)
    {
        for(int j = S.x; j < S.x+S.sizeSquare; j++)
        {
            this->setFree(i,j);
        }
    }
}

Square Field::findNextFreeCell()
{
```

```
    for(int i = 0; i < this->Height; i++)
    {
        for(int j = 0; j < this->Width; j++)
        {
            if(this->isFree(j,i))
            {
                return {0, j, i};
            }
        }
    }

    return {-1, -1, -1};
}

int Field::findMaxSizeOfSquare(int x, int y, int maxSize)
{
    int newSize = 1;
    while((x + newSize < this->Width) && (y + newSize < this->Height) &&
(this->isFree(x + newSize, y)) && (newSize < maxSize))
    {
        newSize++;
    }

    return newSize;
}


void Field::backtracking()
{
    int maxSize = min(this->Height, this->Width) - 1;
    bool full = false;
    Square S;
    Square newSquare;
    S.sizeSquare = maxSize;
    S.x = 0;
    S.y = 0;


    do
    {
        while(!full)
        {
            newSquare = findNextFreeCell();

            if(newSquare.x == -1)
            {
                full = true;
                continue;
            }

            this->buildSquare(newSquare.x, newSquare.y,
findMaxSizeOfSquare(newSquare.x,newSquare.y, maxSize));

            if(this->Squares.size()>0 && this->bestSquares.size()>0 &&
this->bestSquares.size() < this->Squares.size())
            {
                break;
            }
        }
```

```
        if((this->bestSquares.size() > this->Squares.size() || this-
>bestSquares.size() == 0) && full)
        {
            this->bestSquares = this->Squares;
            this->counter = 0;
        }

        if(this->bestSquares.size() == this->Squares.size() && full)
        {
            this->counter++;
        }

        while(this->Squares.back().sizeSquare == 1 && this-
>Squares.size()>1)
        {
            this->destroyLastSquare();
        }

        if(this->Squares.empty())
        {
            break;
        }

        S = this->Squares.back();
        this->destroyLastSquare();
        full = false;

        this->buildSquare(S.x, S.y, S.sizeSquare - 1);

    }
    while(!this->Squares.empty());
}
```