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. 9383 _____ Нистратов Д.Г.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 11 до N - 1N−1, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера NN. Он может получить ее, собрав из уже имеющихся обрезков(квадратов). Например, столешница размера 7\times77×7 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.

Вариант 4р.

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

Постановка задачи.

На вход подается два числа N и M, находящиеся в промежутке от 2 до 20 включительно. Число N задает длину прямоугольника, а M – ширину. После вызывается рекурсивный поиск с возвратом, выполняющий нахождение минимального количества квадратов, из которых можно построить столешницу. Результатом работы программы должен быть массив состоящий из x, y, w, где x,y – координаты квадрата в столешнице, а w – его длинна, а также количество вариантов покрытия минимальным числом квадратов.

Выполнение работы.

Для работы алгоритмы были заданы глобальные переменные:

Answer – массив содержащий координаты квадратов для наилучшего покрытия.

rectMap – прямоугольник, необходимый для поиска свободного пространства.

Stack – стек значений координат квадратов.

Width, height – ширина и длинна прямоугольника

Solve – кол-во вариантов покрытия

Для ввода знаний ширины и длинны прямоугольника была реализована функция input(), осуществляющая считывание двух значений с терминала и проверяющая вхождение каждого значения в рамки 1 < N, M < 21. Если значения длины и ширины одинаковые, то устанавливается булевое значение определяющее формат фигуры. Для ускорения алгоритма, если фигура является квадратом, то стартовое значение кол-ва фигур, которыми можно собрать столешницу, устанавливается равной 17.

Основной рекурсивный алгоритм поиска с возвратом реализован в функции backtracking. При каждом вызове осуществляется постановка квадрата и занесение его координат в стек, если после постановки квадрата небыли найден свободного места или кол-во квадратов превышает наилучший результат, то вызывается функция pop_back удаляющая предыдущие элементы из стека и возвращая алгоритм на прошлые шаги. Если поле не является пустым и кол-во фигур меньше, чем в наилучшем результате, то данный результат заносится в ответы и обнуляется счетчик кол-ва вариантов покрытия. Если кол-во фигур оказывается равным кол-ву фигур в наилучшем результате, то увеличивается кол-во вариантов покрытия.

Список функций:

void backtracking(int square[3], int& count, int& min); - основная функция рекурсивного бектрекинга

void findSquare(const int x, const int y, int& w); - поиск квадрата в области

void removeSquare(const int square[3]); - удаление квадрата по координатам

void placeSquare(const int square[3]); - установка квадрата по координатам

void updateAnswer(const int square_size); - изменение матрицы с наилучшим результатом

void printAnswer(const int size); - вывод результатов в терминал

bool pop_back(int square[3], int& count); - удаление последних элементов из стека

void push_back(int square, int& count); - добавление элемента в стек*

bool isEmptySquares(int square); - поиск пустого места в прямоугольнике*

Тестирование.

Результаты тестирования представлены в Таблица 1.

Таблица 1

Входные данные	*Минимальное кол-во квадратов*	*Число покрытий*
6 7	*5*	*4*
19 19	*13*	*892*
13 11	*6*	*4*
15 15	*6*	*4*

Вывод.

В ходе лабораторный работы был исследован алгоритм квадрирования квадратов, а также расширение данного алгоритма до квадрирования прямоугольников. Был разработан алгоритм, выполняющий поиск минимального кол-ва квадратов для заполнения столешницы. Так же была осуществлена оптимизация для работы алгоритма, таким образом что, прямоугольники, с шириной и высотой в пределе от 2 до 20 включительно, выполняются меньше чем за 1 секунду.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include "square.h"

extern int stack[20][3];
extern int answer[20][3];
extern int rectMap[400];
extern bool type; // 0 - Square, 1 - Rectangle
extern int width, height, solve;

void input();

int main()
{
    input();
    int square[3] = { 0, 0, std::min(width, height) - 1 };
    int min = width * height;
    int count = 0;

    if (!type)
        min = 17;

    backtracking(square, count, min);
    std::cout << min << std::endl;
    printAnswer(min);
```

```
    std::cout << "Number of solutions: " << solve << std::endl;
    return 0;
}

void input()
{
    std::cout << "Input sides of Rectangle (1 < N,M < 21)" << std::endl;
    std::cin >> width >> height;
    if (width < 2 || width > 20 || height < 2 || height > 20){
        std::cout << "Out of sides border (1 < N,M < 21)" << std::endl;
        std::exit(1);
    }
    if (width < height) {
        std::swap(width, height);
    }

    if (width == height) {
        type = false;
    }
    else {
        type = true;
    }
}
```

Название файла: square.cpp

```
#include "square.h"
```

```
int stack[20][3];
int answer[20][3];
int rectMap[400];
bool type; // 0 - Square, 1 - Rectangle
int width, height, solve;

// Recursive backtracking
void backtracking(int square[3], int& count, int& min)
{
    if (count >= min && !pop_back(square, count)) {
        return;
    }
    placeSquare(square);
    push_back(square, count);
    if (!isEmptySquares(square) && count <= min) {
        if (count < min) {
            min = count;
            solve = 1;
        }
        else {
            solve++;
        }
        updateAnswer(min);
    }
    backtracking(square, count, min);
}
```

```
// Find max sided square to fit map
void findSquare(const int x, const int y, int& w)
{
    if (w <= std::min(width, height) - 1) {
        if (x + w > width || y + w > height) {
            w--;
            return;
        }
        if (type) {
            for (int* i = &rectMap[x + y * width]; i != &rectMap[x + y * width + w]; ++i) {
                for (int j = 0; j < w; ++j) {
                    if (*(i + j * width)) {
                        w--;
                        return;
                    }
                }
            }
        }
        else if (rectMap[x + y * width + w - 1] || rectMap[x + y * width + (w -
1) * width] || rectMap[x + y * width + w - 1 + (w - 1) * width]) {
            w--;
            return;
        }
        w++;
        findSquare(x, y, w);
    }
    else{
```

```
            w--;
            return;
        }
    }

    // Remove square from map
    void removeSquare(const int square[3])
    {
        for (int* i = &rectMap[square[0] + square[1] * width]; i != &rectMap[square[0] + square[1] * width + square[2]]; ++i) {
            for (int j = 0; j < square[2]; ++j) {
                *(i + j * width) = 0;
            }
        }
    }

    // Place square on map
    void placeSquare(const int square[3])
    {
        for (int* i = &rectMap[square[0] + square[1] * width]; i != &rectMap[square[0] + square[1] * width + square[2]]; ++i) {
            for (int j = 0; j < square[2]; ++j) {
                *(i + j * width) = square[2];
            }
        }
    }

    // Write answer for new best solution
```

```
void updateAnswer(const int square_size)
{
    for (size_t i = 0; i < square_size; ++i) {
        answer[i][0] = stack[i][0];
        answer[i][1] = stack[i][1];
        answer[i][2] = stack[i][2];
    }
}

// Print best solution
void printAnswer(const int size)
{
    for (int i = 0; i < size; ++i) {
        std::cout << answer[i][0] + 1 << " " << answer[i][1] + 1 << " " << answer[i][2] <
< std::endl;
    }
}

// Remove last element from stack
bool pop_back(int square[3], int& count)
{
    while (count) {
        count--;
        square[0] = stack[count][0];
        square[1] = stack[count][1];
        square[2] = stack[count][2];
        removeSquare(square);
```

```
        if (!count && square[2] < 2) {
            return false;
        }
        if (square[2] > 1) {
            square[2]--;
            return true;
        }
    }
    return false;
}

// Add new element to stack
void push_back(int* square, int& count)
{
    stack[count][0] = square[0];
    stack[count][1] = square[1];
    stack[count][2] = square[2];
    count++;
}

// Jump on map by square sides to find empty spot
bool isEmptySquares(int* square)
{
    int index = square[0] + square[1] * width;
    while (index < height * width) {
        if (!rectMap[index]) {
            square[0] = index % width;
```

```
            square[1] = index / width;
            square[2] = 1;
            findSquare(square[0], square[1], square[2]);
            return true;
        }
        index += rectMap[index];
    }
    return false;
}

// set map #FOR TEST ONLY!
void setMap(std::vector<std::vector<int>> square, int n, int m)
{
    for (int i = 0; i < n; i++){
        for (int j = 0; j < m; j++){
            rectMap[j + i * width] = square[i][j];
        }
    }
}
```

Название файла: square.h

```
#pragma once

#include <iostream>
#include <vector>

void backtracking(int square[3], int& count, int& min);
```

```
void findSquare(const int x, const int y, int& w);

void removeSquare(const int square[3]);

void placeSquare(const int square[3]);

void updateAnswer(const int square_size);

void printAnswer(const int size);

bool pop_back(int square[3], int& count);

void push_back(int* square, int& count);

bool isEmptySquares(int* square);

//FOR TEST ONLY

void setMap(std::vector<std::vector<int>> square, int n, int m);
```