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*

Сложность алгоритма.

Поиск пустого квадрата в поле в худшем случае будет происходить за O(mn), а в среднем за O(n), т. к. перемещение происходит по одномерному массиву с учетом граней квадратов.

Удаление элемента из стека, до нахождения квадрата для квадрирования, будет происходить в среднем за O(n).

Вставка элемент в стек за O(1).

Вставка и удаление квадрата за O(n), где n – грань квадрата.

Общая сложность алгоритма рекурсивного бектрекинга, при подсчете всех вариаций покрытия, будет происходить экспоненциально от длины и ширины прямоугольника.

Тестирование.

Тестирование программы произведено с помощью сторонней библиотеки catch2. Тесты описаны в файле test.cpp.

TEST_CASE("find square width in matrix") – осуществляет проверку функции поиска длинны для квадрата в прямоугольнике. Проверка осуществляется на 2 тестах: когда возможна вставка квадрата длинной 1 и при пустом прямоугольнике.

TEST_CASE("add new element to stack") – осуществляет проверку функции вставки элемента в стек. Проверяется вставка в пустой стек, вставка последнего элемента и вставка в полный стек.

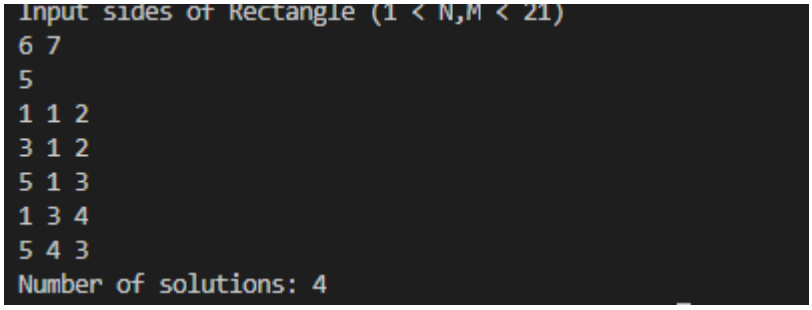
TEST_CASE("remove last elemnt in stack, until found side > 1") – осуществляет проверку функции удаления из стека. Удаление из стека происходит пока не найдена сторона квадрата> 1. Проверяется отсутствие таковых квадратов в стеке, а также присутствие квадрата со стороной 2.

TEST_CASE("check if there's empty space for square") – осуществляет проверку функции поиска свободного места. Тестирование происходит на 3 тестах: когда прямоугольник пустой, когда доступно свободное место начиная с координат x = 7 и y = 7, а также заполненный прямоугольник.

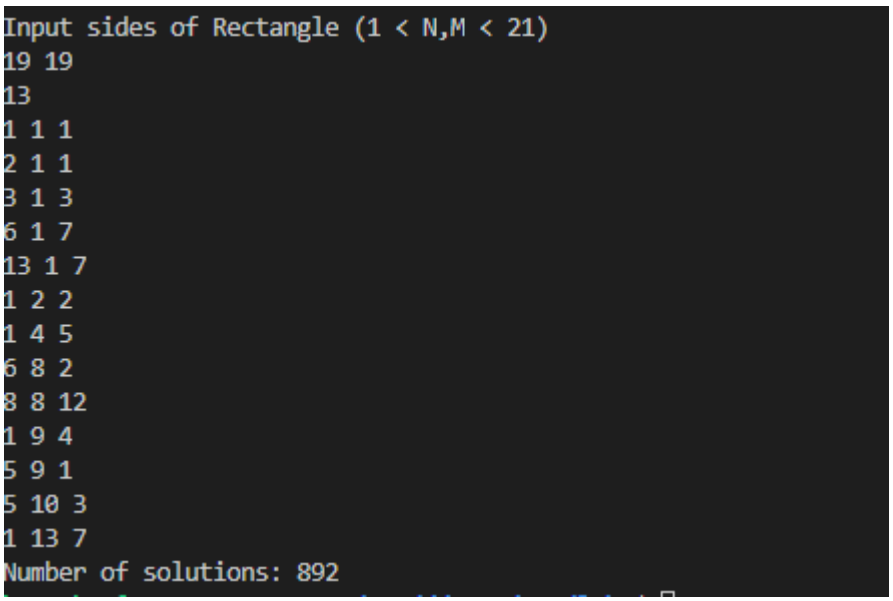
Результаты работы программы представлены в Таблица 1.

Таблица 1

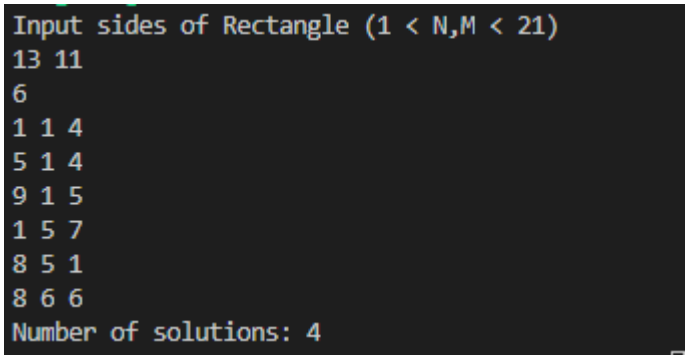
Входные данные	*Минимальное кол-во квадратов*	*Число покрытий*	*Изображение*
6 7	*5*	*4*	*Изображение 1*
19 19	*13*	*892*	*Изображение 2*
13 11	*6*	*4*	*Изображение 3*
15 15	*6*	*4*	*Изображение 4*



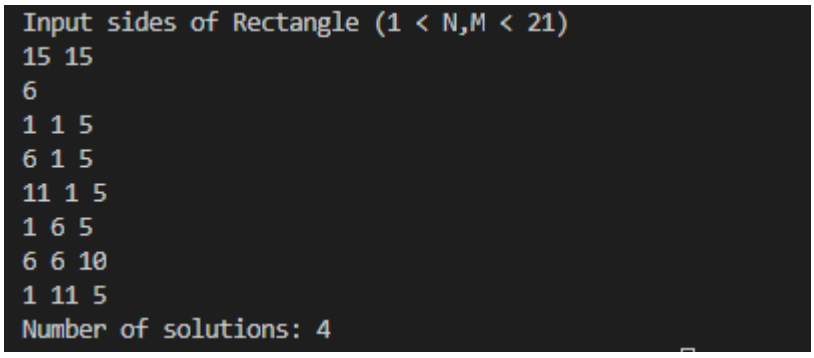
Изображение 1 – прямоугольник 6 на 7



Изображение 2 – квадрат 19 на 19



Изображение 3 – прямоугольник 13 на 11



Изображение 4 – квадрат 15 на 15

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 "square.h"


Square::Square()
{
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < 3; j++) {
            stack[i][j] = 0;
            answer[i][j] = 0;
        }
    }
    for (int i = 0; i < 400; i++) {
        rectMap[i] = 0;
    }
    width = 0;
    height = 0;
    type = false;
    solve = 0;
}


void Square::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);
}

void Square::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);
return;
}
else {
    w--;
    return;
}
}

void Square::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;
        }
    }
```

```
}

void Square::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];
        }
    }
}

void Square::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];
    }
}

void Square::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;
    }
}
```

```
bool Square::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;
}

bool Square::push_back(int* square, int& count)
{
    if (count < 20) {
        stack[count][0] = square[0];
        stack[count][1] = square[1];
        stack[count][2] = square[2];
        count++;
```

```
        return true;
    }
    else {
        return false;
    }
}

bool Square::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;
}

void Square::squareset(int n, int m, bool t)
{
    width = n;
    height = m;
```

```
    type = t;
}
```

Название файла: square.cpp

```
#include "square.h"

Square::Square()
{
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < 3; j++) {
            stack[i][j] = 0;
            answer[i][j] = 0;
        }
    }
    for (int i = 0; i < 400; i++) {
        rectMap[i] = 0;
    }
    width = 0;
    height = 0;
    type = false;
    solve = 0;
}

void Square::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);
}

void Square::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);
    return;
  }
  else {
    w--;
    return;
  }
}

void Square::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;
```

```
            }
        }
    }

    void Square::placeSquare(const int square[3])
    {
        for (int* i = &rectMap[square[0] + square[1] * width]; i != &rectMap[square[0] + squa
re[1] * width + square[2]]; ++i) {
            for (int j = 0; j < square[2]; ++j) {
                *(i + j * width) = square[2];
            }
        }
    }

    void Square::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];
        }
    }

    void Square::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;
```

```
    }
}

bool Square::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;
}

bool Square::push_back(int* square, int& count)
{
    if (count < 20) {
        stack[count][0] = square[0];
        stack[count][1] = square[1];
```

```
        stack[count][2] = square[2];
        count++;
        return true;
    }
    else {
        return false;
    }
}

bool Square::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;
}

void Square::squareset(int n, int m, bool t)
{
```

```
    width = n;
    height = m;
    type = t;
}
```

Название файла: square.h

```
#pragma once
#include <iostream>
#include <algorithm>

class Square
{
public:
    Square();
    ~Square() { };
    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);
    bool push_back(int* square, int& count);
    bool isEmptySquares(int* square);
    void squareset(int n, int m, bool t);
```

```
    int width, height, solve;
    bool type; // false - Square, true - Rectangle
private:
    int stack[20][3];
    int answer[20][3];
    int rectMap[400];
};
```

Название файла: test.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include <vector>
#include "../source/square.h"

TEST_CASE("find square width in matrix")
{
    Square test1;
    test1.squareset(7, 7, false);
    SECTION("width of a square == 1"){
        int square[3] = {0, 0, 6};
        int w = 1;
        test1.placeSquare(square);
        test1.findSquare(6, 6, w);
        REQUIRE(w == 1);
        test1.removeSquare(square);
    }
```

```
    SECTION("empty map, widht of a square == 6"){
        int w = 1;
        test1.findSquare(0, 0, w);
        REQUIRE(w == 6);
    }
}

TEST_CASE("add new element to stack")
{
    Square test2;
    test2.squareset(20, 20, false);
    int count = 0;
    int square[3] = {0, 0, 1};
    SECTION("add first element to stack"){
        REQUIRE(test2.push_back(square, count) == true);
    }
    SECTION("add last element to stack"){
    for (int i = 0; i < 19; i++){
        test2.push_back(square, count);
    }
        REQUIRE(test2.push_back(square, count) == true);
    }
    SECTION("reach limit"){
    for (int i = 0; i < 20; i++){
        test2.push_back(square, count);
    }
        REQUIRE(test2.push_back(square, count) == false);
```

```
    }
}

TEST_CASE("remove last elemnt in stack, until found side > 1")
{
    Square test3;
    test3.squareset(5, 5, false);
    int square[3] = {0, 0, 0};
    int count = 0;
    SECTION("No element in stack"){
        square[2] = 1;
    for (int i = 0; i < 5; i++){
        square[0] = i;
        test3.placeSquare(square);
        test3.push_back(square, count);
    }
    REQUIRE(test3.pop_back(square, count) == false);
    }
    SECTION("Found element > 1"){
        square[2] = 2;
        test3.placeSquare(square);
        test3.push_back(square, count);
        REQUIRE(test3.pop_back(square, count) == true);
    }
}

TEST_CASE("check if there's empty space for square")
```

```
{
    Square test4;
    test4.squareset(7, 7, false);
    int square[3] = {0, 0, 0};

    SECTION("Empty matrix"){
        REQUIRE(test4.isEmptySquares(square) == true);
    }
    SECTION("Empty square on x = 7, y = 7"){
        square[2] = 6;
        test4.placeSquare(square);
        REQUIRE(test4.isEmptySquares(square) == true);
    }
    SECTION("Matrix is filled"){
        square[2] = 7;
        test4.placeSquare(square);
        REQUIRE(test4.isEmptySquares(square) == false);
    }
}
```