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. 0382 _____ Бочаров Г.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Изучить принцип работы алгоритма поиска с возвратом. Решить с его помощью задачу

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.

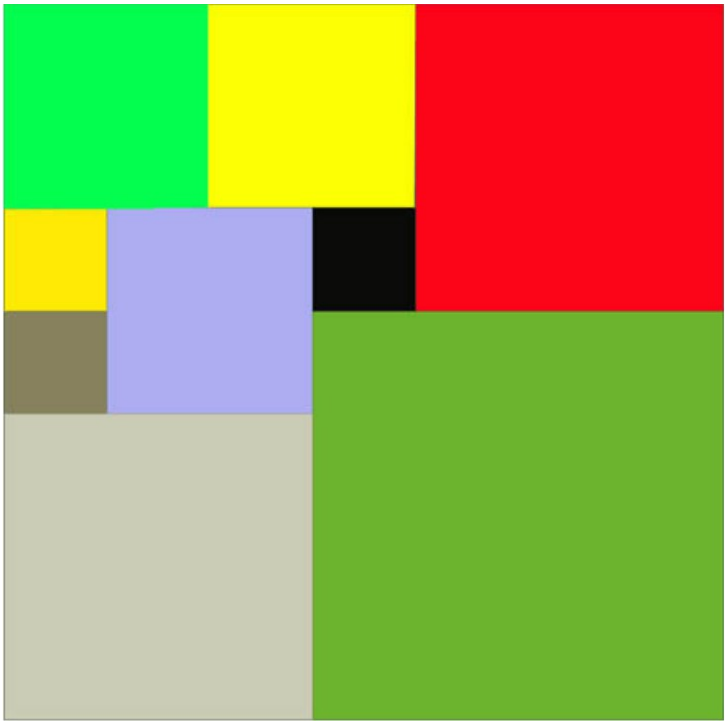

Рисунок 1 — разбиение квадрата 7x7

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.

Вар. 4И:

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

Выполнение работы.

Исходный код программы представлен в приложении А.

Описание алгоритма:

Для решения задачи перебираются всевозможные варианты покрытия квадратного или прямоугольного поля квадратами меньшего размера.

Поле заполняется справа налево сверху вниз. В клетке, которая еще не покрыта квадратной областью, размещается квадрат с максимальной длиной стороны.

Повторяем предыдущий шаг до полного покрытия поля квадратами. Таким образом, мы получим полное покрытие поля квадратами, упорядоченными в порядке их добавления в покрытие.

Далее ищем квадрат сторона которого больше 1 при этом с максимальным порядковым номером добавления в покрытие. Пытаемся разместить на его месте квадрат меньшего размера. Возвращаемся к предыдущему пункту , получая уже другое покрытие и сравниваем с исходным.

В результате сравнения покрытий в конечном итоге найходится минимальное.

Частичные решения (покрытия) хранятся в виде массива квадратов (координаты левого верхнего угла и размер).

Описание функций и структур данных

1. Структура square - описывает квадрат. Поля : x_, y_, - координаты верхнего левого угла квадрата , size_ - размер квадрата.

2. fill_arr(int** arr, int size_x, int size_y) — функция заполнения двумерной матрицы нулями

3. fill_square(int** arr, int a, int b, int size, int c) - — функция заполнения области двумерной матрицы, соответствующей квадрату числом, соответствующим номеру квадрата в покрытии.

4. void find_max_square(int** arr, std::vector<square>& current, int a, int b, int field_size) — Функция размещает квадрат максимального размера по переданным координатам и добавляет квадрат в покрытие.

5. void erase_squre(int** arr, square s) — стирает квадрат из матрицы

6. void fill_1(int** arr, std::vector<square>& current, int field_size, int min_size) — функция заполняет двумерную матрицу (возможно уже частично заполненную) квадратами максимального размера. Если частичое решение получается больше минимального на данный момент, процесс построения решения прекращается.

7. unsigned int greatest_common_divisor(unsigned int a, unsigned int b) — функция поиска НОД

8. int max_del(int a) — поиск паксимального делителя числа

9. void go_(int size_x, int size_y) — главная функция программы. По заданным размером поля строит первичное покрытие. Далее в цикле происходит поиск более оптимальных вариантов покрытия. В массив res - заносится минимальное на данный момент покрытие, в массив current – записывается текущее покрытие. Перебор вариантов прекращается в

момент когда функция доходит до квадратов, размеры которых менять не нужно или когда их размеры уже не могут стать меньше.

Примененные оптимизации

1. Масштабирование поля.

а) Квадратное поле. Если размер поля не является простым числом, ищется наименьший делитель (min_del) числа соответствующего размеру поля. Далее, вычисляется масштаб scale = (размер поля) / (минимальный делитель), а наименьший делитель теперь ассоциируется с размером поля.

Пример: Поле 15 x 15 можно заполнить как поле 3 x 3, только размеры квадратов в покрытии будут в 5 раз больше .

б) Прямоугольное поле. Если размеры поля не являются простыми числми, ищется наибольший общий делитель чисел, соответствующих размерам поля и поле масштабируется. Масштаб scale = NOD(a, b), где a, b – размеры поля.

2. При добавлении очердного квадрата в покрытие, проверяется не превышает ли кол-во квадратов в покрытии размеров наименьшего на данный момент покрытия. Если такое происходит, то дальнейшее построение данного частичного решения не имеет смысла. Кол-во перебираемых решений сильно сокращается.

3. В случае квадратного поля в левом верхнем углу размещается квадрат размера (2*a + 1) / 2, а также смежные с ним квадраты (справа и снизу) размера (2*a + 1) / 2 – 1.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	3 3	6	
2.	4 4	4	
3.	5 5	8	
4.	7 7	9	
5.	15 15	6	Разбивается как 3x3
6.	35 35	8	Разбивается как 5x5
7.	4 5	6	
8.	5 8	9	
9.	6 5	5	
10.	4 3	6	
11.	2 8	16	
12.	5 9	9	
13.	2 7	14	

Выводы.

В результате работы была написана программа поиска минимального покрытия прямоугольной области квадратами. Были проведены необходимый оптимизации, для уменьшения времени работы алгоритма.

Приложение А

Исходный код программы

Название файла: Source.cpp

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <math.h>

class square
{
public:
    int x_;
    int y_;
    int size_;

    square(int x, int y, int size) : x_(x), y_(y), size_(size)
    {

    }
    void print()
    {

            std::cout << x_ + 1 << " " << y_ + 1 << " " << size_ <<
std::endl;
        //std::cout << std::endl;
    }
};

square operator * (square s, int a)
{
    return square(s.x_ * a, s.y_ * a, s.size_ * a);
}

void fill_arr(int** arr, int size)
{
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
            arr[i][j] = 0;
    }
}
void fill_arr(int** arr, int size_x, int size_y)
{
    for (int i = 0; i < size_x; i++) {
        for (int j = 0; j < size_y; j++)
            arr[i][j] = 0;
    }
}

void print_arr(int** arr, int size)
{
    std::cout << std::endl;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
             std::cout << std::setw(3) << std::setprecision(1) << arr[i]
[j];
```

```
        std::cout << std::endl;
    }
}
void print_arr(int** arr, int size_x, int size_y)
{
    std::cout << std::endl;
    for (int i = 0; i < size_x; i++) {
        for (int j = 0; j < size_y; j++)
             std::cout << std::setw(3) << std::setprecision(1) << arr[i]
[j];
        std::cout << std::endl;
    }
}




void fill_square(int** arr, int a, int b, int size, int c)
{
    for (int i = a; i < a + size; i++)
        for (int j = b; j < b + size; j++)
            arr[i][j] = c;
}
void fill_square(int** arr, square s, int c)
{
    for (int i = s.x_; i < s.x_ + s.size_; i++)
        for (int j = s.y_; j < s.y_ + s.size_; j++)
            arr[i][j] = c;
}

void find_max_square(int** arr, std::vector<square>& current, int a, int
b, int field_size)
{
    bool is_size = true;
    int step = 0;
    while (is_size && step < field_size)
    {
        for (int i = a; i < a + step; i++)
             if (b + step >= field_size || i >= field_size || arr[i][b +
step] != 0)
                 is_size = false;
        for (int i = b; i < b + step; i++)
              if (a + step >= field_size || i >= field_size || arr[a +
step][i] != 0)
                 is_size = false;
        step++;

        //std::cout << step << std::endl;
    }
    current.push_back(square(a, b, step - 1));

    fill_square(arr, a, b, step - 1, current.size());

}
void find_max_square(int** arr, std::vector<square>& current, int a, int
b, int field_size_x, int field_size_y)
{
    bool is_size = true;
    int step = 0;
```

```
    while (is_size)
    {
        if (step >= std::min(field_size_x, field_size_y) - 1)
            is_size = false;
        for (int i = a; i < a + step; i++)
            if (b + step >= field_size_y || i >= field_size_x || arr[i][b
+ step] != 0)
                is_size = false;
        for (int i = b; i < b + step; i++)
            if (a + step >= field_size_x || i >= field_size_y || arr[a +
step][i] != 0)
                is_size = false;
        step++;

        //std::cout << step << std::endl;
    }
    current.push_back(square(a, b, step - 1));

    fill_square(arr, a, b, step - 1, current.size());

}

void erase_squre(int** arr, square s)
{

    for (int i = s.x_; i < s.x_ + s.size_; i++)
        for (int j = s.y_; j < s.y_ + s.size_; j++)
            arr[i][j] = 0;
}


void fill(int** arr, std::vector<square>& current, int field_size)
{
    for (int i = 0; i < field_size; i++)
        for (int j = 0; j < field_size; j++)
            if (arr[i][j] == 0)
                find_max_square(arr, current, i, j, field_size);
}

void fill(int** arr, std::vector<square>& current, int field_size_x, int
field_size_y)
{
    for (int i = 0; i < field_size_x; i++)
        for (int j = 0; j < field_size_y; j++)
            if (arr[i][j] == 0)
                    find_max_square(arr, current, i, j, field_size_x,
field_size_y);
}

void fill_1(int** arr, std::vector<square>& current, int field_size, int
min_size)
{
    for (int i = 0; i < field_size; i++)
        for (int j = 0; j < field_size; j++)
            if (arr[i][j] == 0)
            {
                find_max_square(arr, current, i, j, field_size);
                if (current.size() >= min_size)
                    return;
```

```
            }
}

void fill_1(int** arr, std::vector<square>& current, int field_size_x,
int field_size_y, int min_size)
{
    for (int i = 0; i < field_size_x; i++)
        for (int j = 0; j < field_size_y; j++)
            if (arr[i][j] == 0)
            {
                    find_max_square(arr, current, i, j, field_size_x,
field_size_y);
                if (current.size() >= min_size)
                    return;
            }
}



void print_squares(std::vector<square>& current, int scale = 1)
{
    for (auto i : current)
        (i * scale).print();
}

int min_del(int a)
{
    for (int i = 2; i <= sqrt(a) + 1; i++)
    {
        if (a % i == 0)
            return i;
    }
    return 1;
}
int max_del(int a)
{
    for (int i = sqrt(a); i > 1; i--)
    {
        if (a % i == 0)
            return i;
    }
    return 1;
}


unsigned int greatest_common_divisor(unsigned int a, unsigned int b) {
    if (a == b)
        return max_del(a);
    if (a > b)
        return greatest_common_divisor(a - b, b);
    return greatest_common_divisor(a, b - a);
}

void init_start_cover(std::vector<square>& current, int** arr, int size)
{
    if (size == 2)
    {
        square s1(0, 0, 1);
        square s2(0, 1, 1);
```

```
        square s3(1, 0, 1);
        square s4(1, 1, 1);

        current.push_back(s1);
        current.push_back(s2);
        current.push_back(s3);
        current.push_back(s4);
    }
    else {
        square s1(0, 0, ceil((float)size / 2));
        square s2(0, ceil((float)size / 2), ceil((float)size / 2) - 1);
        square s3(ceil((float)size / 2), 0, ceil((float)size / 2) - 1);

        current.push_back(s1);
        current.push_back(s2);
        current.push_back(s3);
    }

    // #fil matrix with start squares
    for (int i = 0; i < current.size(); i++)
    {
        fill_square(arr, current[i], i + 1);
    }
    // #fil matrix
    fill(arr, current, size);
}

void go_(int size_x, int size_y)
{
    int** res_arr = new int* [size_x];
    for (int i = 0; i < size_x; i++)
        res_arr[i] = new int[size_y];
    fill_arr(res_arr, size_x, size_y);

    //print_arr(res_arr, size_x, size_y);
    std::vector<square> result;
    std::vector<square> current;

    int scale = greatest_common_divisor(size_x, size_y);

    size_x = size_x / scale;
    size_y = size_y / scale;


    int** arr = new int* [size_x];
    for (int i = 0; i < size_x; i++)
        arr[i] = new int[size_y];
    fill_arr(arr, size_x, size_y);


    if (size_x == size_y)
        init_start_cover(current, arr, size_x);


    fill(arr, current, size_x, size_y);
    result = current;
    int min_size = current.size();

    int last = min_size — 1;
```

```
    while (current[0].size_ >= std::min(size_x, size_y) / 2)
    {

        last = 0;
        for (int i = current.size() - 1; i >= 0; i--)
            if (current[i].size_ > 1)
            {
                last = i;
                break;
            }

        if ((size_x == size_y) && (last <= 2)) break;

        if (last + 1 > min_size)
        {
            for (int i = current.size() - 1; i >= last; i--)
            {
                erase_squre(arr, current[i]);
                current.erase(current.end() - 1);
            }
            continue;
        }



        for (int i = current.size() - 1; i >= last; i--)
        {
            erase_squre(arr, current[i]);
            if (i != last)
                current.erase(current.end() - 1);
        }

        current[last].size_--;
                  fill_square(arr,  current[last].x_,  current[last].y_,
current[last].size_, last + 1);

        fill_1(arr, current, size_x, size_y, min_size);
        if (current.size() < min_size)
        {
            min_size = current.size();
            result = current;
        }
    }

    std::cout << min_size << std::endl;
    for (int i = 0; i < result.size(); i++)
    {
        fill_square(res_arr, result[i] * scale, i + 1);
    }
   // print_arr(res_arr, size_x * scale, size_y * scale);

    //print_squares(result, scale);
}

int main()
{
    int x, y;
    std::cin >> x;
    std::cin >> y;
```

```
    go_(x, y);


    /*for (int i = 2; i < 10; i++)
        for (int j = 2; j < 10; j++)
        {
             std::cout << "a == " << i << " b == " << j << " min_cover =
";
            go_(i, j);
        }

    while (n != 1) {
        std::cin >> n;
        go_(n);
    }*/
    return 0;
}
```