# 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. 8304 | Самакаев Д.И. |
| Преподаватель | Размочаева Н.В. |

Санкт-Петербург 2020

**Вариант 4р**

**Цель работы**

Ознакомиться и закрепить знания, связанные с алгоритмами поиска с возвратом.

**Постановка задачи**

  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

**Описание алгоритма**

Рекурсивный алгоритм, для каждой клетки квадрата (прямоугольника), для каждого возможного размещенного в этой клетке обрезка, рассматриваются все варианты расположения отрезков в следующей клетке обрезков всех возможных размеров.

**Описание функций**

1. bool add\_sq(size\_t x, size\_t y, size\_t side\_size, size\_t sq\_number, std::shared\_ptr<size\_t[]> sq\_arr, size\_t sq\_width, size\_t sq\_height)

добавляет квадрат в точку с координатами x, y.

1. void delete\_sq(size\_t x, size\_t y, size\_t side\_size, std::shared\_ptr<size\_t[]> sq\_arr, size\_t sq\_width, size\_t sq\_height)

удаляет квадрат из точки x, y.

1. void req\_fill(std::shared\_ptr<size\_t[]> &sq\_arr, size\_t sq\_width, size\_t sq\_height, size\_t i, size\_t sq\_number, size\_t &min\_cnt, std::shared\_ptr<size\_t[]> &min\_sq\_arr, size\_t &covers\_cnt)

Рекурсивная функция, основная, осуществляющая поиск решения.

1. void run\_algorithm(size\_t sq\_width, size\_t sq\_height)

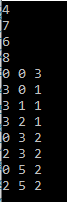
функция передает в основную параметры и обрабатывает результат.

**Вывод**

Был получен опыт работы с алгоритмами поиска с возвратом, реализована программа, рассчитывающая минимальное разбиение квадрата (прямоугольника) на квадраты.

**ПРИЛОЖЕНИЕ А.**

**ТЕСТИРОВАНИЕ.**



Здесь 4 – ширина прямоугольника , 7 – высота. Результат – 6 разбиений на 8 квадратов. Квадраты имеют параметры, которые идут впоследствии.

**ПРИЛОЖЕНИЕ B.**

**Исходный код.**

#include <iostream>

#include <vector>

#include <algorithm>

void delete\_sq(size\_t x, size\_t y, size\_t side\_size, std::shared\_ptr<size\_t[]> sq\_arr, size\_t sq\_width, size\_t sq\_height) {

for (size\_t j = y; j < y + side\_size; j++)

for (size\_t i = x; i < x + side\_size; i++)

sq\_arr[i + sq\_width \* j] = 0;

}

bool add\_sq(size\_t x, size\_t y, size\_t side\_size, size\_t sq\_number, std::shared\_ptr<size\_t[]> sq\_arr, size\_t sq\_width, size\_t sq\_height) {

if (x + side\_size >= sq\_width + 1 || y + side\_size >= sq\_height + 1)

return false;

for (size\_t j = y; j < y + side\_size; j++)

for (size\_t i = x; i < x + side\_size; i++)

if (sq\_arr[i + sq\_width \* j] != 0)

return false;

for (size\_t j = y; j < y + side\_size; j++)

for (size\_t i = x; i < x + side\_size; i++)

sq\_arr[i + sq\_width \* j] = sq\_number;

return true;

}

void req\_fill(std::shared\_ptr<size\_t[]> &sq\_arr, size\_t sq\_width, size\_t sq\_height, size\_t i, size\_t sq\_number, size\_t &min\_cnt,

std::shared\_ptr<size\_t[]> &min\_sq\_arr, size\_t &covers\_cnt) {

if (i == sq\_width \* sq\_height) {

if (sq\_number == min\_cnt + 1)

covers\_cnt++;

if (sq\_number < min\_cnt) {

covers\_cnt = 1;

min\_cnt = sq\_number - 1;

for (size\_t i = 0; i < (sq\_width \* sq\_height); i++) {

min\_sq\_arr[i] = sq\_arr[i];

}

}

return;

}

if (sq\_number > min\_cnt + 1)

return;

for (size\_t j = std::min(sq\_width, sq\_height) - 1; j > 0; j--) {

if (add\_sq(i % sq\_width, i / sq\_width, j, sq\_number, sq\_arr, sq\_width, sq\_height)) {

req\_fill(sq\_arr, sq\_width, sq\_height, i + 1, sq\_number + 1, min\_cnt, min\_sq\_arr, covers\_cnt);

delete\_sq(i % sq\_width, i / sq\_width, j, sq\_arr, sq\_width, sq\_height);

}

else {

if(j == 1)

req\_fill(sq\_arr, sq\_width, sq\_height, i + 1, sq\_number, min\_cnt, min\_sq\_arr, covers\_cnt);

continue;

}

}

return;

}

void run\_algorithm(size\_t sq\_width, size\_t sq\_height) {

//sq\_height = sq\_height / piece\_size;

//sq\_width = sq\_width / piece\_size;

std::shared\_ptr<size\_t[]> sq\_arr(new size\_t[sq\_height \* sq\_width]);

std::shared\_ptr<size\_t[]> min\_sq\_arr(new size\_t[sq\_height \* sq\_width]);

for (size\_t i = 0; i < (sq\_height \* sq\_width); i++) {

sq\_arr[i] = 0;

}

size\_t min\_cnt = sq\_height \* sq\_width + 1;

size\_t covers\_cnt = 0;

req\_fill(sq\_arr, sq\_width, sq\_height, 0, 1, min\_cnt, min\_sq\_arr, covers\_cnt);

std::cout << covers\_cnt << std::endl;

std::vector<size\_t> sqs\_sizes(min\_cnt);

for (size\_t j = 1; j <= min\_cnt; j++) {

for (size\_t i = 0; i < (sq\_height \* sq\_width); i++) {

if (min\_sq\_arr[i] == j) {

sqs\_sizes.at(j - 1) += 1;

}

else {

continue;

}

}

}

//for (size\_t i = 0; i < min\_cnt; i++)

// std::cout << sqs\_sizes.at(i) << '\n';

std::cout << min\_cnt << std::endl;

for (size\_t j = 1; j <= min\_cnt; j++) {

sqs\_sizes.at(j - 1) = sqrt(sqs\_sizes.at(j - 1));

for (size\_t i = 0; i < (sq\_height \* sq\_width); i++) {

if (min\_sq\_arr[i] == j) {

std::cout << (i % sq\_width) << " " << (i / sq\_width) << " " << sqs\_sizes.at(j - 1) << std::endl;

break;

}

}

}

}

int main() {

size\_t piece\_size = 1;

size\_t sq\_width;

std::cin >> sq\_width;

size\_t sq\_height;

std::cin >> sq\_height;

//for (size\_t i = std::min(sq\_height, sq\_width) - 1; i > 1; i--) {

// if (sq\_width % i == 0 && sq\_height % i == 0) {

// piece\_size = i;

// break;

// }

//}

run\_algorithm(sq\_width, sq\_height);

return 0;

}