МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №7

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

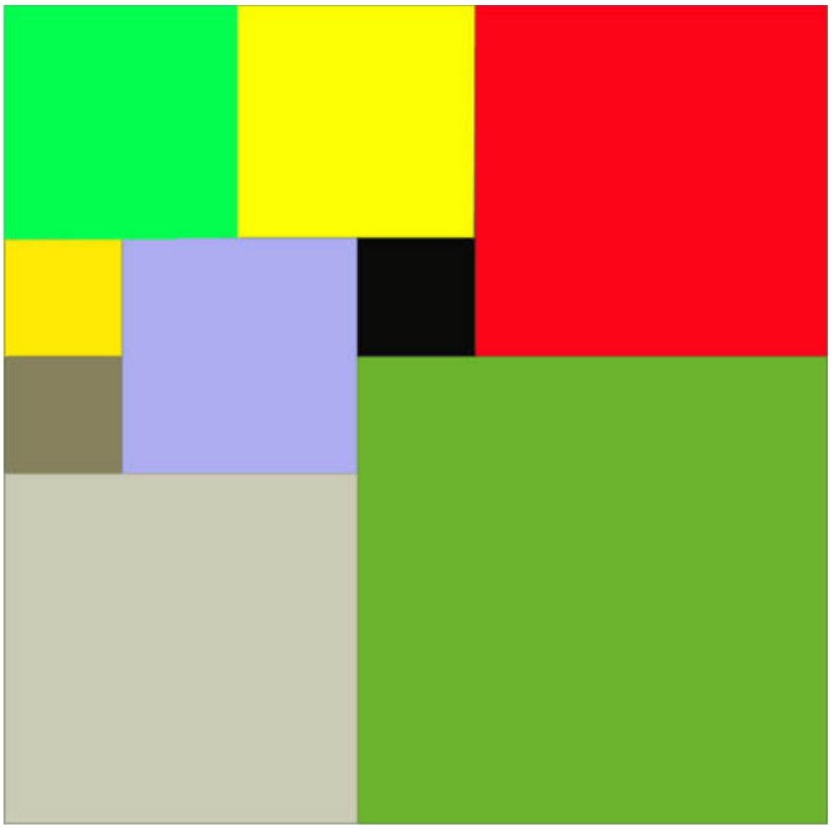
Цель работы

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

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≤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.
2. 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.
```

3. `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.

4. `void run_algorithm(size_t sq_width, size_t sq_height)`

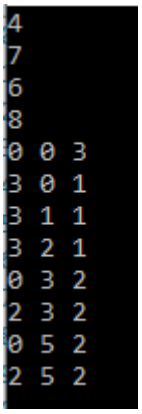
функция передает в основную параметры и обрабатывает результат.

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 В.

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;
	 }
```