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. 7383 _____ Лосев М.Л.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2018

СОДЕРЖАНИЕ

Постановка задачи.

Цель работы: получение знаний и опыта в использовании алгоритмов поиска с возвратом.

Формулировка 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). 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 N ($2 \leq N \leq 20$).

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 ($1 \leq x, y \leq N$ $1 \leq x, y \leq N$) и длину стороны соответствующего обрезка(квадрата).

Реализация

Был использован следующий класс:

```
class field
{
public:
	field(int size);
	void run();
	void print_result();
	~field();

private:
	int O = 0;
	int size;
	int **pieces;
	int ans_size = 0;
```

```
    int *ans_x;
    int *ans_y;
    int *ans_w;
    int array_size = 20;
    bool final = false;

    void extend_array();
    bool find_emty_pice(int &x, int &y);
    int find_max_size(int x, int y);
    void put_sqare(int x, int y, int s, int n);
    void clear_field(int x, int y, int s);
    void put_1st_square();
    void put_2_3st_square();
    int back_tracking(int deep);
};
```

bool find_emty_pice(int &x, int &y) находит самую верхнюю и левую пустую клетку поля.

`void extend_array()` увеличивает в два раза место, выделенное для массива квадратов.

`int find_max_size(int x, int y)` находит максимальное значение стороны квадрата, который можно поместить верхним левым углом в клетку (x; y).

`void put_sqare(int x, int y, int s, int n)` помещает верхним левым углом в клетку (x; y) квадрат со стороной s и номером n.

`void clear_field(int x, int y, int s)` удаляет квадрат с поля.

`void put_1st_square()` ставит первый квадрат разумным образом: если сторона кратна 2, то ставит квадрат со стороной в 1/2 стороны поля; если она кратна 3, то 2/3; если она кратна 5, то 3/5; если она кратна 7, то 4/7; если же она не кратна этим числам, то сторона первого квадрата равна половине стороны поля + 1. Это надо для того, чтобы уменьшить количество вариантов, которые будет перебирать процедура поиска с возвратом.

`void put_2_3st_square()` ставит второй и третий квадрат справа и снизу от первого так, чтобы их стороны были максимально возможны.

`int back_tracking(int deep);` перебирает варианты квадрирования неразбитого участка поля.

Исследование

Было решено считать сложность алгоритма по количеству вызовов функции `put_sqare,` потому что она работает за квадратичное время, то есть довольно долго. Можно было считать по по количеству вызовов `clear_field.`

В таблице один представлены результаты исследования для некоторых простых значений стороны квадрата.

Таблица 1. Результаты исследования.

Сторона квадрата	Количество вызовов функции `put_sqare`
7	55
11	708
13	1606
17	9964
19	28266
23	105690
29	733269
31	1746940
37	8463490
41	28047085

Был построен график сложности алгоритма, который представлен на рисунке 1. Из этого графика видно, что сложность не превышает 1.6^x.

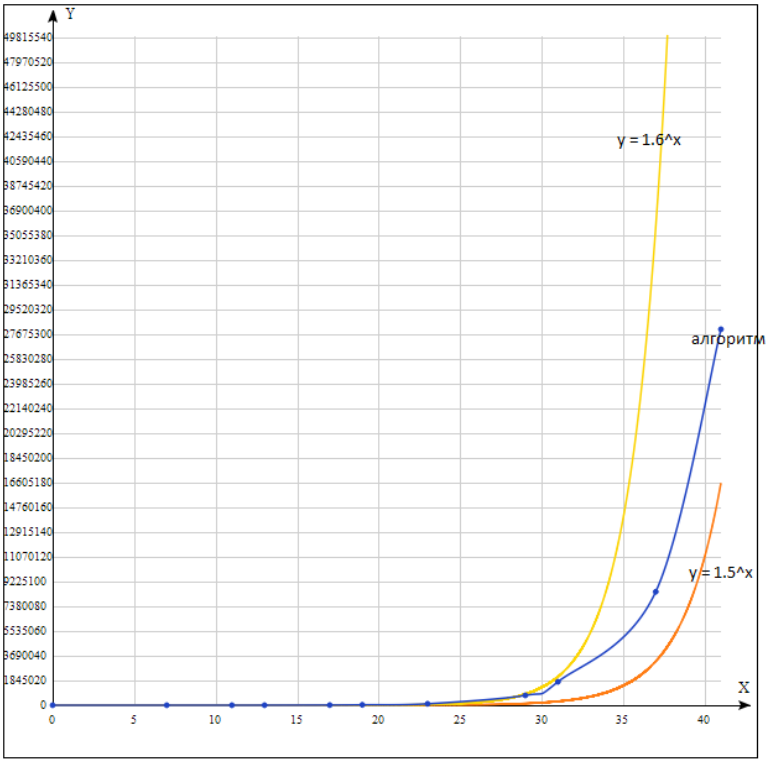

Рисунок 1. Сложность алгоритма

Алгоритм не выделяет дополнительную память, то есть затраты памяти определяются стороной поля, от которой зависят квадратично.

Тестирование

Тестирование проводилось в ОС Ubuntu 18.04 компилятором GCC. Результаты тестирования показали, что программа работает корректно. Результаты тестирования представлены в приложении А.

Вывод

При выполнении работы был изучен метод поиска с возвратом. Была реализована программа, применяющая этот метод для квадрирования квадрата. Эта программа была исследована на предмет сложности ее алгоритма: по времени она не превышает 1,6^x, а по памяти не превышает x^2.

ПРИЛОЖЕИЕ А.

Тестовые случаи

Размер стороны квадрата	Результат
7	9 1 1 4 5 1 3 1 5 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2
17	12 1 1 9 10 1 8 1 10 8 9 10 2 9 12 4 9 16 2 10 9 1 11 9 3 11 16 2 13 12 1 13 13 5 14 9 4
23	13 1 1 12 13 1 11 1 13 11 12 13 2 12 15 5 12 20 4 13 12 1 14 12 3 16 20 1 16 21 3 17 12 7 17 19 2 19 19 5
31	15 1 1 16 17 1 15 1 17 15 16 17 3 16 20 6 16 26 6 17 16 1 18 16 1

	19 16 4 22 20 1 22 21 1 22 22 10 23 16 6 29 16 3 29 19 3
37	15 1 1 19 20 1 18 1 20 18 19 20 2 19 22 5 19 27 11 20 19 1 21 19 3 24 19 8 30 27 3 30 30 8 32 19 6 32 25 1 32 26 1 33 25 5

ПРИЛОЖЕИЕ Б.

Исходный код программы

```
#include <iostream>

class field
{
public:
	field(int size);
	void run();
	void print_result();
	~field();

private:
	int O = 0;
	int size;
	int **pieces;
	int ans_size = 0;
	int *ans_x;
	int *ans_y;
	int *ans_w;
	int array_size = 20;
	bool final = false;

	void extend_array();
	int find_max_size(int x, int y);
	void put_sqare(int x, int y, int s, int n);
	void del_sqare(int x, int y, int s);
	void put_1st_square();
	void put_2_3st_square();
	bool find_emty_pice(int &x, int &y);
	int back_tracking(int deep);
};
void field::extend_array()
{
	array_size *= 2;
	int *tmp_x = new int[array_size];
	int *tmp_y = new int[array_size];
	int *tmp_w = new int[array_size];

	for (int i = 0; i < ans_size; i++){
		tmp_x[i] = ans_x[i];
		tmp_y[i] = ans_y[i];
		tmp_w[i] = ans_w[i];
}

	delete[] ans_x;
	delete[] ans_y;
```

```
	delete[] ans_w;

	ans_x = tmp_x;
	ans_y = tmp_y;
	ans_w = tmp_w;
}

int field::find_max_size(int x, int y)
{
	for (int s = 1; s <= size - x && s <= size - y; s++)
		for (int i = 0; i < s; i++)
			for (int j = 0; j < s; j++)
				if (pieces[x + i][y + j] != 0)
					return --s;
}

bool field::find_emty_pice(int &x, int &y)
{
	for (int i = 0; i < size; i++)
		for (int j = 0; j < size; j++)
			if (pieces[i][j] == 0) {
				x = i;
				y = j;
				return true; // нашли
			}
	return false;
}

void field::put_sqare(int x, int y, int s, int n)
{
	for (int i = 0; i < s; i++)
		for (int j = 0; j < s; j++)
			pieces[x + i][y + j] = n;
	O++;
}

void field::del_sqare(int x, int y, int s)
{
	for (int i = 0; i < s; i++)
		for (int j = 0; j < s; j++)
			pieces[x + i][y + j] = 0;
}

void field::put_1st_square()
{
	ans_x[ans_size] = 0;
	ans_y[ans_size] = 0;
	if (size % 2 == 0){
		ans_w[ans_size] = size / 2;
```

```
	} // в опт. разб. кв. 2*2 всего 4 квадрата, один 1*1
	else if (size % 3 == 0){
		ans_w[ans_size] = size * 2 / 3;
	} // в опт. разб. кв. 3*3 всего 6 квадратов, один 2*2
	else if (size % 5 == 0){
		ans_w[ans_size] = size * 3 / 5;
	} // в опт. разб. кв. 5*5 всего 8 квадратов, один 3*3
	else if (size % 7 == 0){
		ans_w[ans_size] = size * 4 / 7;
	} // в опт. разб. кв. 7*7 всего 9 квадратов, один 4*4
	// 4 < 6 < 8 < 9
	else{
		ans_w[ans_size] = size / 2 + 1;
	}

	put_sqare(0, 0, ans_w[ans_size], ans_size + 1);
	ans_size ++;
}

void field::put_2_3st_square()
{
	ans_x[ans_size] = ans_w[0];
	ans_y[ans_size] = 0;
	ans_w[ans_size] = find_max_size(ans_x[ans_size], ans_y[ans_size]);
	put_sqare(ans_x[ans_size], ans_y[ans_size], ans_w[ans_size],
ans_size + 1);
	ans_size++;

	ans_x[ans_size] = 0;
	ans_y[ans_size] = ans_w[0];
	ans_w[ans_size] = find_max_size(ans_x[ans_size], ans_y[ans_size]);
	put_sqare(ans_x[ans_size], ans_y[ans_size] , ans_w[ans_size],
ans_size + 1);
	ans_size++;
}

int field::back_tracking(int deep)
{
	if (final && (deep > ans_size))
		return deep;

	int cur_x;
	int cur_y;

	if (!find_emty_pice(cur_x, cur_y)){ // если не нашли свободное
место
		if (!final || (final && deep - 1 < ans_size))
			ans_size = deep - 1;
		final = true;
```

```
            return ans_size;
        }

        if (deep >= array_size)
            extend_array();

        int min_ans = size * size;
        for (int cur_w = find_max_size(cur_x, cur_y); cur_w > 0; cur_w--) {
            put_sqare(cur_x, cur_y, cur_w, deep);

            int cur_ans = back_tracking(deep + 1);
            min_ans = min_ans < cur_ans ? min_ans : cur_ans;
            if (final && cur_ans <= ans_size){
                ans_x[deep - 1] = cur_x;
                ans_y[deep - 1] = cur_y;
                ans_w[deep - 1] = cur_w;
            }

            del_sqare(cur_x, cur_y, cur_w);
        }

        return min_ans;
    }

    field::field(int size) : size(size)
    {
        pieces = new int*[size];
        for (int i = 0; i < size; i++){
            pieces[i] = new int[size];
            for (int j = 0; j < size; j++)
                pieces[i][j] = 0;
        }
        ans_x = new int[array_size];
        ans_y = new int[array_size];
        ans_w = new int[array_size];
    }

    void field::run()
    {
        put_1st_square();
        put_2_3st_square();
        back_tracking(4);
    }

    void field::print_result()
    {
        std::cout << "O = " << O << std::endl;
        std::cout << ans_size << std::endl;
```

```
	for (int i = 0; i < ans_size; i++)
		std::cout << ans_x[i] + 1 << ' ' << ans_y[i] + 1 << ' ' <<
ans_w[i] << std::endl;
}

field::~field()
{
	delete[] ans_x;
	delete[] ans_y;
	delete[] ans_w;
	for (int i = 0; i < size; i++)
		delete[] pieces[i];
	delete[] pieces;
}

int main()
{
	int n;
	std::cin >> n;

	field table(n);

	table.run();
	table.print_result();

	return 0;
}
```