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 _____ Левкович Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Содержание

Цель работы

Цель работы: познакомиться с алгоритмом поиска с возвратом, создать программу, использующую метод бэктрекинга.

Формулировка задачи: Разбить квадрат со стороной N на минимально возможное число квадратов со сторонами от 1 до N-1. Внутри квадрата не должно быть пустот, квадраты не должны перекрывать друг друга и выходить за пределы основного квадрата. Программа должна вывести количество квадратов, а также координаты левого верхнего угла и размер стороны каждого квадрата.

Реализация задачи

Программа была написана на языке программирования C++.

Для реализации поставленной задачи был создан класс Square.

Метод `PutSquare` ставит на поле первые 3 квадрата. Если длина стороны четная, первый квадрат имеет сторону в 1/2 от исходной, если она кратна 3, то в 2/3, если кратна 5, то в 3/5 от исходной длины. В остальных случаях квадрат имеет сторону в 1/2 от исходной длины плюс 1. Второй и третий квадраты добавляются в соседние с первым квадратом углы и имеют максимальный возможный размер. Метод `BackTracking` рекурсивно проверяет все возможные варианты расположение последующих квадратов на поле. Метод `solve` поочередно вызывает методы `void PutSquare` и `BackTracking`. Метод `print` вызывается в случае, если поле заполнено. Метод хранит координаты расставленных квадратов и их ширину. Так были написаны дополнительные методы: `findSquare` – находит квадрат заданного цвета, `findEmptySquare` – находит пустую область на поле, `isOutofBounds` – проверка на случай, если вставляемый квадрат выходит за границы поля, `RemoveSqr` – удаляет квадрат заданного цвета.

Исходный код программы представлен в приложении Б.

Исследование алгоритма

Было принято исследовать сложность алгоритма по количеству вызовов функции, добавляющей квадрат на поле, когда длина стороны поля – простое число. Количество итераций для некоторых простых чисел приведено в таблице ниже.

Размер стороны квадрата	Количество итераций
2	5
3	7
5	19
7	56
11	709
13	1607
17	9965
19	28267
23	105691
29	733270
31	1746941
37	8463491
41	28047086

Из результатов исследования видно, что сложность алгоритма не превышает 1.8^n. На рисунке 1 приведет график зависимости длины стороны поля от количества итераций, а также графики функций 1.5^n и 1.6^n.

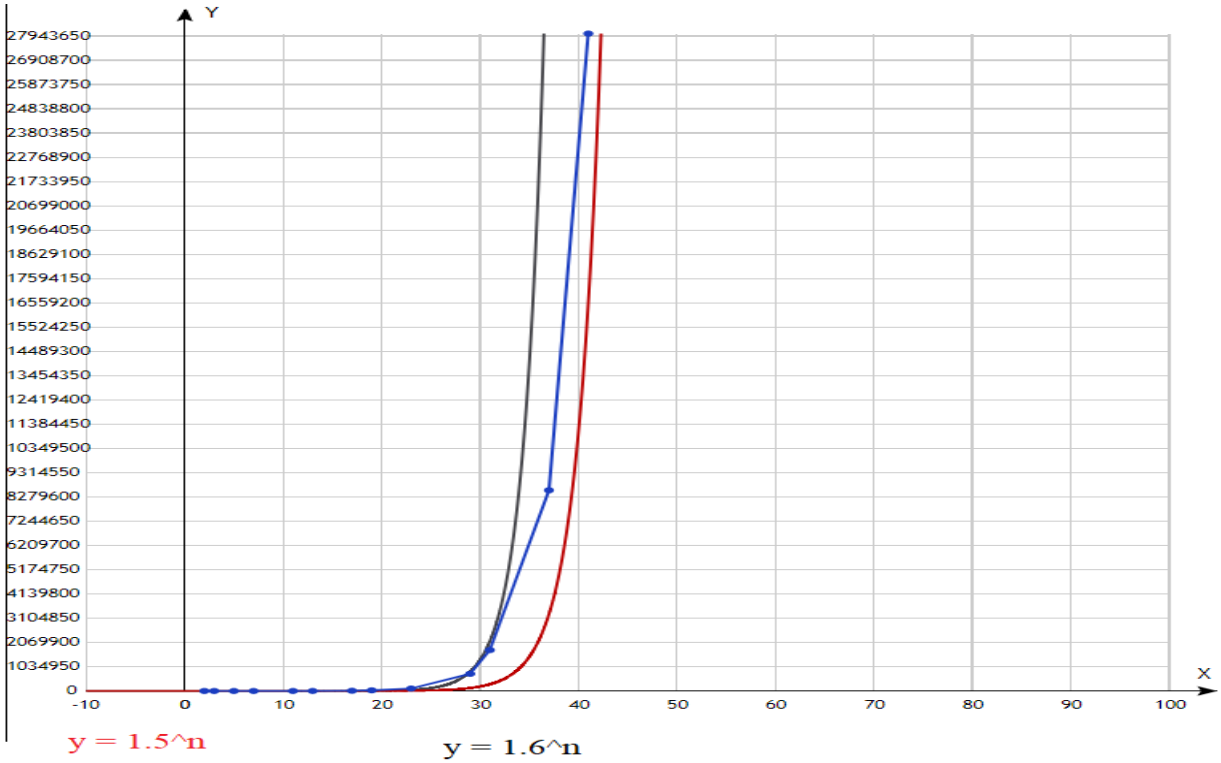

Рисунок 1 – График зависимости длины стороны поля от количества итераций

Во время работы алгоритм не выделяет дополнительной памяти, поэтому все затраты определяются количеством памяти, необходимой для хранения поля и координат.

Тестирование

1. Процесс тестирования

Программа собрана в операционной системе Windows компилятором g++. В других ОС и компиляторах тестирование не проводилось.

2. Результаты тестирования

В результате тестирования были обнаружены и исправлены ошибки, приводящие к некорректным результатам на некоторых исходных данных. Тестовые случаи представлены в приложении А.

Вывод

В ходе выполнения данной работы был изучен метод поиска с возвратом. Была написана программа, применяющая метод бэктрекинга для поиска разбиение квадрата на минимально возможное число меньших квадратов. Также сложность алгоритма была исследована по количеству вызовов функции, осуществляющей поиск с возвратом: сложность алгоритма не превышает 2^n.

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

Размер стороны квадрата	Результат
8	4 1 1 4 5 1 4 1 5 4 5 5 4
27	6 1 1 18 1 19 9 19 1 9 19 10 9 10 19 9 19 19 9
35	8 1 1 21 1 22 14 22 1 14 22 15 14 15 22 7 15 29 7 22 29 7 29 29 7
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

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД

```
#include <iostream>
#include <vector>
#include <cmath>
#include <exception>
#include <fstream>
using namespace std;

class Square{
public:
    Square(int N):arr(N, vector<int>(N, 0)), best(15, vector<int>(3,
0))
    {
        size = N;
        bestcolor = 16;
        color = 4;
        count = 3;
    }
    ~Square(){

    }

    void solve(){
        if(size%2 == 0){
            PutSquare(0, 0, size/2, 1, size/2);
            PutSquare(0, size/2,size/2, 2,size/2);
            PutSquare(size/2, 0,size/2, 3,size/2);
            BackTracking( size/2, size/2, size-size/2);
            ans();
        }
        else if(size%3 == 0){
            PutSquare(0, 0, 2*size/3, 1,size/3);
            PutSquare(0, 2*size/3, size/3, 2,size/3);
            PutSquare(2*size/3, 0, size/3, 3,size/3);
            BackTracking( 2*size/3, size/3, size/3);
            ans();
        }
        else if(size%5 == 0){
            PutSquare(0, 0, 3*size/5, 1,2*size/5);
            PutSquare(0, 3*size/5, 2*size/5, 2,2*size/5);
            PutSquare(3*size/5, 0, 2*size/5, 3,2*size/5);
            BackTracking(3*size/5, 2*size/5, 2*size/5);
```

```
            ans();
        }
        else{
            PutSquare(0,0,size/2+1,1,size-(size/2+1));
            PutSquare(size/2+1,0,size/2,2,size-(size/2+1));
            PutSquare(0,size/2+1,size/2,3,size-(size/2+1));
            BackTracking(size/2+1, size/2, size-(size/2+1));
            ans();
        }
    }

private:
    vector<vector<int>> best;
    vector<vector<int>> arr;
    int size;
    int bestcolor;
    int color;
    int count;


    void RemoveSqr(int N, int color){
        for(auto i = N; i<size;i++)
            for(auto j = N; j < size; j++){
                if(arr[i][j]==color)
                    arr[i][j] = 0;
            }

    }

    bool findEmptySquare(int & x, int & y, int N){
        for (y = N;y<size;y++) {
            for (x = N;x<size;x++) {
                if(arr[y][x]==0)
                    return true;
            }
        }
        return false;
    }

    bool isOutOfBounds(int x, int y, int width)
    {
        if(width +x > size || width +y > size)
            return true;
        else
```

```
        return  false;
}

bool PutSquare(int x, int y, int w, int color, int N){
    if(isOutOfBounds(x, y, w))
        return false;
    for(auto i = 0; i<w;i++){
        for(auto j = 0;j<w;j++){
            if(arr[y+i][x+j] == 0){
                arr[y+i][x+j] = color;
            }
            else{
                RemoveSqr(N, color);
                return false;
            }
        }
    }

    return true;
}

void BackTracking(int x, int y, unsigned N){
    if(color-1>bestcolor){
        return;
    }
    if(findEmptySquare(x, y, N)){
        for(int i = N; i > 0;i--){
            if(PutSquare(x, y, i, color, N)){
                count++;
                color++;
                BackTracking(x, y, N);
                color--;
            }
         RemoveSqr(N, color);
        }
    }
    else if((color-1) < bestcolor){
        bestcolor = color-1;
        print(bestcolor);
    }
}

int findSquare(int & x, int & y,int K){
    while (arr[y][x] != K){
```

```
            if (x == size-1){
                x = 0;
                ++y;
            }
            else
                ++x;
            if (x>=size || y>=size)
                return 0;
        }
        int i = 0;
        while (arr[y][x+i] == K){
            if ((x+i) < size)
            {
                ++i;
            }
            else return --i;
        }
        return i;
    }

    void print(int color){
        int x, y, size;
        for(int k = 1; k <= color; ++k)
        {
            x = 0;
            y = 0;
            size = findSquare(x, y, k);
            best[k-1][0] = x + 1;
            best[k-1][1] = y + 1;
            best[k-1][2] = size;
        }
    }
    void ans(){
        cout << bestcolor << endl;
        for(int i = 0;i<bestcolor;i++){
           cout<<best[i][0]<<" "<<best[i][1]<<" "<<best[i][2]<<endl;
        }
        cout<<count<<endl;
    }
};

int main()
{
    int N = 0;
```

```
    cin>>N;
    Square a(N);
    a.solve();
    return 0;
}
```