

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

Власов Р.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург
2019

Содержание

Цель работы	3
Реализация задачи	4
Исследование алгоритма	5
Тестирование	7
1. Процесс тестирования.....	7
2. Результаты тестирования.....	7
Вывод	8
Приложение А. Тестовые случаи	9
Приложение Б. Исходный код	10

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

Программу было решено писать на языке программирования C++.

Для реализации поставленной задачи был создан класс field.

```
class field
{
    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 clear_field(int x, int y, int s);
    void step1();
    void step2();
    int step3(int deep = 4);

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

Конструктор класса инициализирует дерево строкой, содержащейся в начале декодируемого файла.

Методы void step1() и void step2() добавляют 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 void step3() рекурсивно проверяет все возможные варианты расположения последующих квадратов на поле. Метод void run() поочередно вызывает методы void step1(), void step2() и void step3(). Метод void print_result() выводит на экран результат работы программы. Также были реализованы вспомогательные методы: int find_max_size(int x, int y) возвращает максимальный размер квадрата, который может поместиться в указанной точке, void extend_array() увеличивает размер массивов, в которых хранятся результаты, void clear_field() очищает с поля квадрат, добавленный на предыдущей итерации.

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 1.6^n . На рисунке 1 приведен график зависимости длины стороны поля от количества итераций, а также графики функций 1.6^n и 1.7^n .

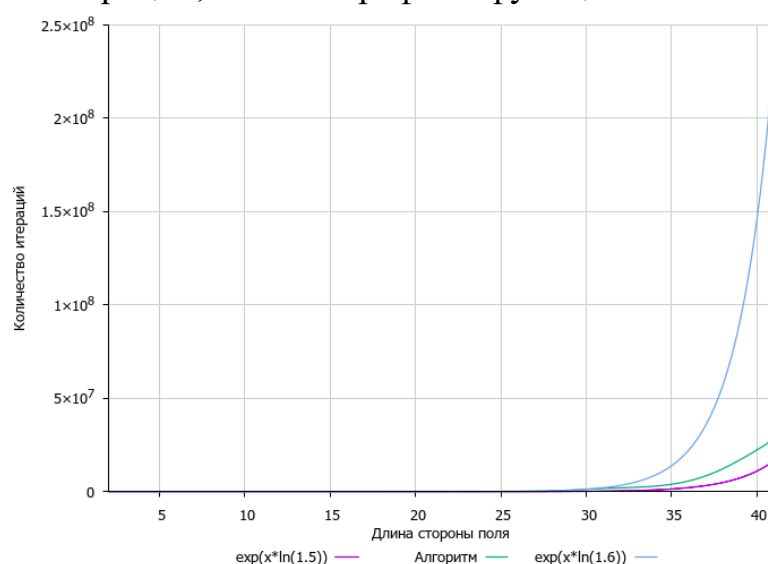


Рисунок 1 – График зависимости длины стороны поля от количества итераций

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

Тестирование

1. Процесс тестирования

Программа собрана в операционной системе Ubuntu 18.04.2 LTS bionic компилятором g++ version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04). 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 разбиения 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 19 1 9 1 19 9 10 19 9 19 10 9 19 19 9
35	8 1 1 21 22 1 14 1 22 14 15 22 14 22 15 7 29 15 7 29 22 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>

class field
{
    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 clear_field(int x, int y, int s);
    void step1();
    void step2();
    int step3(int deep = 4);

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

int main()
{
    int n;
    std::cin >> n;
    field f(n);
    f.run();
    f.print_result();
    return 0;
}

void field::extend_array()
{
    array_size += 20;
```

```

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)
{
    int s = 1;
    bool flag = true;
    while(flag && s <= size - x && s <= size - y)
    {
        for (int i = 0; i < s; i++)
            for (int j = 0; j < s; j++)
                if (pieces[x + i][y + j] != 0)
                {
                    flag = false;
                    s--;
                }
        s++;
    }
    s--;
    if (s == size)
        s--;
    return s;
}

void field::clear_field(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::step1()
{
    ans_x[ans_size] = 0;
    ans_y[ans_size] = 0;
    if (size % 2 == 0)
    {
        ans_w[ans_size] = size / 2;
    }
    else if (size % 3 == 0)
    {
        ans_w[ans_size] = size * 2 / 3;
    }
    else if (size % 5 == 0)
    {
        ans_w[ans_size] = size * 3 / 5;
    }
    else
    {
        ans_w[ans_size] = size / 2 + 1;
    }
    for (int i = 0; i < ans_w[ans_size]; i++)
        for (int j = 0; j < ans_w[ans_size]; j++)
            pieces[i][j] = ans_size + 1;
    ans_size++;
}

void field::step2()
{
    ans_y[ans_size] = 0;
    for (int i = 0; i < size; i++)
    {
        if (pieces[i][0] == 0)
        {
            ans_x[ans_size] = i;
            break;
        }
    }
}

```

```

        }
        ans_w[ans_size] = find_max_size(ans_x[ans_size],
ans_y[ans_size]);
        for (int i = 0; i < ans_w[ans_size]; i++)
            for (int j = 0; j < ans_w[ans_size]; j++)
                pieces[ans_x[ans_size] + i][ans_y[ans_size] + j] =
ans_size + 1;
        ans_size++;

        ans_x[ans_size] = 0;
        for (int i = 0; i < size; i++)
        {
            if (pieces[0][i] == 0)
            {
                ans_y[ans_size] = i;
                break;
            }
        }
        ans_w[ans_size] = find_max_size(ans_x[ans_size],
ans_y[ans_size]);
        for (int i = 0; i < ans_w[ans_size]; i++)
            for (int j = 0; j < ans_w[ans_size]; j++)
                pieces[ans_x[ans_size] + i][ans_y[ans_size] + j] =
ans_size + 1;
        ans_size++;
    }

    int field::step3(int deep)
    {
        if (final && deep > ans_size)
            return deep;

        int cur_x = -1;
        int cur_y = -1;
        int cur_w = -1;
        int max_square_size;

        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                if (cur_x == -1 && pieces[i][j] == 0)
                {
                    cur_x = i;

```

```

        cur_y = j;
    }
    if (cur_x == -1)
    {
        if (!final || (final && deep - 1 < ans_size))
            ans_size = deep - 1;
        final = true;
        return ans_size;
    }
    if (deep >= array_size)
        extend_array();
    max_square_size = find_max_size(cur_x, cur_y);
    int min_ans = size * size;
    for (cur_w = max_square_size; cur_w > 0; cur_w--)
    {
        for (int x = 0; x < cur_w; x++)
            for (int y = 0; y < cur_w; y++)
                pieces[cur_x + x][cur_y + y] = deep;
        int cur_ans = step3(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;
        }
        clear_field(cur_x, cur_y, cur_w);
    }
    //clear_field(deep);
    return min_ans;
}

void field::show_field()
{
    for (int i = 0 ; i < size; i++)
    {
        for (int j = 0; j < size; j++)
            std::cout << pieces[i][j];
        std::cout << std::endl;
    }
}

```

```

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()
{
    step1();
    step2();
    step3(4);
}

void field::print_result()
{
    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;
}

```