

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

Студентка гр. 7383

Иолшина В.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2019

Содержание

Цель работы	3
Реализация задачи	4
Тестирование	7
Исследование	8
Выводы	9
ПРИЛОЖЕНИЕ А	10
ПРИЛОЖЕНИЕ Б.....	11

Цель работы

Исследовать и реализовать квадрирования квадрата, используя алгоритм 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 её, 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 минимальное возможное число обрезков.

Входные данные: размер столешницы N ($2 \leq N \leq 40$).

Выходные данные: 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 и длину стороны соответствующего обрезка.

Реализация задачи

В данной работе используются главная функция `main()` и дополнительные функции.

Был реализован следующий класс:

```
class Square
{
private:
    int **s_array;
    int *x;
    int *y;
    int *w; //длины сторон
    int size;
```

```
int s_amount; //количество квадратов
int num; //порядковый номер квадрата
bool if_last; //последний ли квадрат
```

```
void set_square(int x, int y, int n, int side);
void set_second();
void set_third();
bool is_empty(int &x, int &y);
int get_max_size(int x, int y);
void delete_square(int x, int y, int side);
```

```
public:
```

```
Square(int size);
int set_first();
void result(int amount);
int backtracking(int deep);
~Square();
```

```
};
```

- void set_square(int x, int y, int n, int side) по заданным координатам x и y левого верхнего угла помещает квадрат со стороной side и номером n.
- int set_first() помещает первый квадрат на поле, в зависимости от стороны квадрата. Так, если сторона квадрата кратна 2, то помещается квадрат со стороной равной половине стороны изначального квадрата, если сторона кратна 3, то помещается квадрат со стороной $\frac{2}{3}$ от изначального, а если 5, то со стороной $\frac{3}{5}$. Но если сторона не кратна 2, 3 или 5, то помещается квадрат со стороной $N/2 + 1$.
- void set_second() помещает второй квадрат с максимально возможной стороной
- void set_third() помещает третий квадрат с максимально возможной стороной

- `bool is_empty(int &x, int &y)` ищет левое верхнее свободное поле для вставки нового квадрата
- `int get_max_size(int x, int y)` находит максимальную сторону квадрата, который можно поместить на поле
- `void result(int amount)` выводит результат работы программы
- `int backtracking(int deep)` с помощью алгоритма поиска с возвратом находит наилучший вариант квадрирования оставшегося участка поля.

Тестирование

Программа собрана в операционной системе Ubuntu 16.04.2 LTS", с использованием компилятора g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5). В других ОС и компиляторах тестирование не проводилось.

Тестовые случаи представлены в Приложении А.

Исследование

Была исследована частота вставки новых квадратов, для квадрата изначально имеющего сторону соответствующую простому числу, для чего в функцию `void set_square(int x, int y, int n, int side)` был добавлен счетчик.

Результаты, полученные при исследовании сложности алгоритма, приведены в табл. 1.

Таблица 1 – Результаты исследования

Сторона поля	Количество вызовов функции
7	59
11	709
13	1611
17	9970
19	28260
23	105693
29	733262
31	1746944

По полученным данным построен график, представленный на рис.2.

Можно сделать вывод, что сложность алгоритма экспоненциальная.



Рисунок 2 – зависимость числа операций от вставки

квадрата.

Выводы

В ходе лабораторной работы был изучен алгоритм поиска с возвратом, а также способы квадрирования квадрата. Поставленная задача была реализована на языке C++. Так же была исследована сложность алгоритма.

ПРИЛОЖЕНИЕ А **ТЕСТОВЫЕ СЛУЧАИ**

Ввод	Вывод	Верно?
4	4 1 1 2 3 1 2 1 3 2 3 3 2	Да
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	Да
27	6 1 1 18 19 1 9 19 10 9 19 19 9 1 19 9 10 19 9	Да

ПРИЛОЖЕНИЕ Б

Исходный код программы

```
#include <iostream>
#define N 40

class Square
{
private:
    int **s_array;
    int *x;
    int *y;
    int *w; //длины сторон
    int size;
    int s_amount; //количество квадратов
    int num; //порядковый номер квадрата
    bool if_last; //последний ли квадрат

    void set_square(int x, int y, int n, int side);
    void set_second();
    void set_third();
    bool is_empty(int &x, int &y);
    int get_max_size(int x, int y);
    void delete_square(int x, int y, int side);

public:
    Square(int size);
    int set_first();
    void result(int amount);
    int backtracking(int deep);
    ~Square();
};

Square::Square(int size) : size(size)
{
```



```

s_array= new int*[size];
for(int i=0; i<size; i++)
{
    s_array[i]= new int[size];
    for(int j=0; j<size; j++)
        s_array[i][j]=0;
}
s_amount= N;
x= new int[N];
y= new int[N];
w= new int[N];
if_last=false;
num=0;
}

void Square::set_square(int x, int y, int n, int side)
{
    for(int i=x; i < x+side; i++)
        for(int j=y; j < y+side; j++)
            s_array[i][j] = n;
}

int Square::set_first()
{
    x[num]=y[num]=0;
    if(size%2 == 0)
    {
        x[2]=y[1]=0;
        x[1]=x[3]=y[2]=y[3]=size*1/2;
        for(int i=0; i<4; i++)
            w[i]=size*1/2;
        return 4;
    }
    if(size%3 == 0)
    {
        w[num]=size*2/3;
        x[4]=y[1]=0;
    }
}

```

```

        x[5]=y[2]=w[num]*1/2;
        x[1]=x[2]=x[3]=y[3]=y[5]=y[4]=w[num];
        for(int i=1; i<6; i++)
            w[i]=size*1/3;
        return 6;
    }
    if(size%5 == 0)
    {
        w[num]=size*3/5;
        x[4]=y[1]=0;
        w[2]=w[3]=w[5]=w[6]=w[num]*1/3;
        w[1]=w[4]=w[7]=w[num]*2/3;
        x[5]=x[6]=y[2]=y[3]=size*2/5;
        x[3]=y[6]=size*4/5;
        x[1]=x[2]=x[7]=y[4]=y[5]=y[7]=w[num];
        return 8;
    }
    else
    {
        w[num]=size*1/2 + 1;
        set_square(x[num], y[num], num+1, w[num]);
        num++;
        set_second();
        set_third();
        return backtracking(4);
    }
}

void Square::set_second()
{
    x[num]=w[0];
    y[num]=0;
    w[num]=get_max_size(x[num], y[num]);
    set_square(x[num], y[num], num+1, w[num]);
    num++;
}

```

```

void Square::set_third()
{
    x[num]=0;
    y[num]=w[0];
    w[num]=get_max_size(x[num], y[num]);
    set_square(x[num], y[num], num+1, w[num]);
    num++;
}

```

```

void Square::delete_square(int x, int y, int side)
{
    for(int i=x; i < x+side; i++)
        for(int j=y; j < y+side; j++)
            s_array[i][j] = 0;
}

```

```

bool Square::is_empty(int &x, int &y)
{
    for(int i=0; i < size; i++)
        for(int j=0; j < size; j++)
            if(!s_array[i][j])
            {
                x=i;
                y=j;
                return true;
            }
    return false;
}

```

```

int Square::get_max_size(int x, int y)
{
    int max_size=1, size1=0, size2=0, i=0, j=0;
    while(i < size-x && s_array[x+i][y] == 0)
    {
        size1++;
        i++;
    }
}

```

```

while(j < size-y && s_array[x][y+j] == 0)
{
    size2++;
    j++;
}
(size1 < size2) ? (max_size = size1) : (max_size = size2);
return max_size;
}

int Square::backtracking(int deep)
{
    if(if_last && deep > num)
        return deep;
    int x_tmp;
    int y_tmp;
    int w_tmp;
    int result_tmp;
    int min_result=size*size;
    if(!is_empty(x_tmp, y_tmp))
    {
        if((if_last && deep-1 < num) || !if_last)
            num=deep-1;
        if_last=true;
        return num;
    }
    for(w_tmp=get_max_size(x_tmp, y_tmp); w_tmp>0; w_tmp--)
    {
        set_square(x_tmp, y_tmp, deep, w_tmp);
        result_tmp = backtracking(deep+1);
        min_result = min_result < result_tmp ? min_result : result_tmp;
        if(result_tmp<=num)
        {
            x[deep-1] = x_tmp;
            y[deep-1] = y_tmp;
            w[deep-1] = w_tmp;
        }
        delete_square(x_tmp, y_tmp, w_tmp);
    }
}

```

```

    }
    return min_result;
}

void Square::result(int amount)
{
    std::cout<<amount<<"\n";
    for(int i=0; i<amount; i++)
        std::cout<<x[i]+1<<" "<<y[i]+1<<" "<<w[i]<<"\n";
}

Square::~~Square()
{
    delete[] x;
    delete[] y;
    delete[] w;
    for(int i=0; i<size; i++)
        delete[] s_array[i];
    delete[] s_array;
}

int main()
{
    int size, s_amount;
    std::cin>>size;
    Square table(size);
    s_amount=table.set_first();
    table.result(s_amount);
    return 0;
}

```