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

ОТЧЕТ

по практической работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 7383 _____ Зуев Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Цель работы.

Цель работы: ознакомиться с алгоритмом поиска с возвратом на примере построения алгоритма для выполнения задачи.

Формулировка задачи: Заполнить квадратное поле наименьшим количеством меньших квадратов без перекрывания друг друга и без оставления пустых областей. Исследовать сложность написанного алгоритма.

Реализация задачи.

Для реализации задачи был создан класс square.

Ниже представлены поля класса:

unsigned short int size — длина стороны заполняемого квадрата.

bool** is_empty — двумерный массив в котором хранится информация о свободных позициях основного квадрата.

int* min — массив координат и длин сторон квадратов, заполняющих основной квадрат, при этом их количество наименьшее.

unsigned short int minnum — наименьшее количество квадратов, используемых для заполнения основного квадрата.

unsigned short int num — количество квадратов, используемых для заполнения основного квадрата во время поиска.

int* current — массив координат и длин сторон квадратов, заполняющих основной квадрат во время поиска.

Далее представлены методы класса:

point search_empty() — находит пустую верхнюю левую клетку (полагается, что основной квадрат разбит на клетки единичной длины).

void add(point p, unsigned short int s) — заполняет is_empty с клетки p квадратом со стороной s.

void backtrack(point p, int s) — удаляет квадрат размером s с клетки p.

void search() — поиск разбиения основного квадрата минимальным числом меньших квадратов. При полном заполнении квадрата или при

неполном заполнении квадрата числом меньших квадратов меньшим минимального на 1 переходит к предыдущему квадрату и выполняет метод backtrack. На каждом ходу уменьшает размер вставляемого квадрата так, чтобы тот поместился на поле не перекрывая остяльные.

void set_first() — размещает на квадрате первые три квадрата. Написана для оптимизации. Квадраты, длины сторон которых — это большие простые числа, разбиваются большим числом квадратов (сильно большим чем если бы длины сторон были составные числа). Из-за этого программа совершает гораздо большее число итераций, что сильно повышает время выполнения программы. Проверка на кратность двум, трем и пяти перекрывает все числа кроме простых, больших пяти. Поэтому добавлено дополнение для больших простых чисел. При добавлении одного или двух квадратов существенного повышения производительности добиться не удается, поэтому данной функцией добавляется три квадрата.

void print() — выводит минимальное число квадратов, их координаты и размеры.

В главной функции main считывается вводимое с консоли число — длина стороны главного квадрата. Создается класс для этого квадрата. Вставляются первые три квадрата, выполняется поиск наименьшего числа квадратов и после этого результат выводится в консоль.

Код программы представлен в приложении Б.

Исследование сложности алгоритма.

Исследовать алгоритм было решено по количеству вызовов метода search, отдельно для простых чисел, для чисел кратным двум, трем и пяти, так как вид числа изначально определяет количество итераций необходимое для поиска наименьшего количества квадратов.

Результат исследования для простых чисел представлен в табл. 1, для чисел кратных двум в табл. 2, для чисел кратных трем в табл. 3, для чисел кратных пяти в табл. 4.

Таблица 1 — Количество итераций для простых чисел

Длина стороны квадрата	Количество итераций
2	2
3	4
5	16
7	50
11	554
13	1240
17	7056
19	19040
23	70068
29	475994
31	1094646
37	5405951
41	18093344

Таблица 2 — Количество итераций для четных чисел

Длина стороны квадрата	Количество итераций
2	2
4	3
6	3
8	3
10	3
12	3
14	3
16	3
18	3
20	3
22	3
24	3
26	3
28	3
30	3
32	3
34	3
36	3

38	3
40	3

Таблица 3 — Количество итераций для чисел кратных трем

Длина стороны квадрата	Количество итераций
3	4
9	17
15	37
21	65
27	101
33	145
39	197

Таблица 4 — Количество итераций для чисел кратных пяти

Длина стороны квадрата	Количество итераций
5	16
25	1123
35	1094646

В основном квадрате остается примерно N^2 свободных клеток. В каждую из них в худшем случае вставляется N-1 квадратов. Если бы алгоритм перебирал все варианты, то его сложность была бы $O(N^{N*N})$. Однако алгоритм существенно уменьшает количество рассматриваемых вариантов.

Алгоритм для каждой из групп чисел сравнивается с экспоненциальными и степенными функциями.

По рисунку 1 видно, что для простых чисел алгоритм имеет сложность $O(1.6^n)$.

По табл. 2 видно, что для четных чисел алгоритм имеет константную сложность.

По рисунку 2 видно, что для чисел кратных трем алгоритм имеет сложность $O(n^{1.5})$.

По рисунку 3 видно, что для чисел кратных пяти алгоритм имеет сложность $O(1,5^n)$.

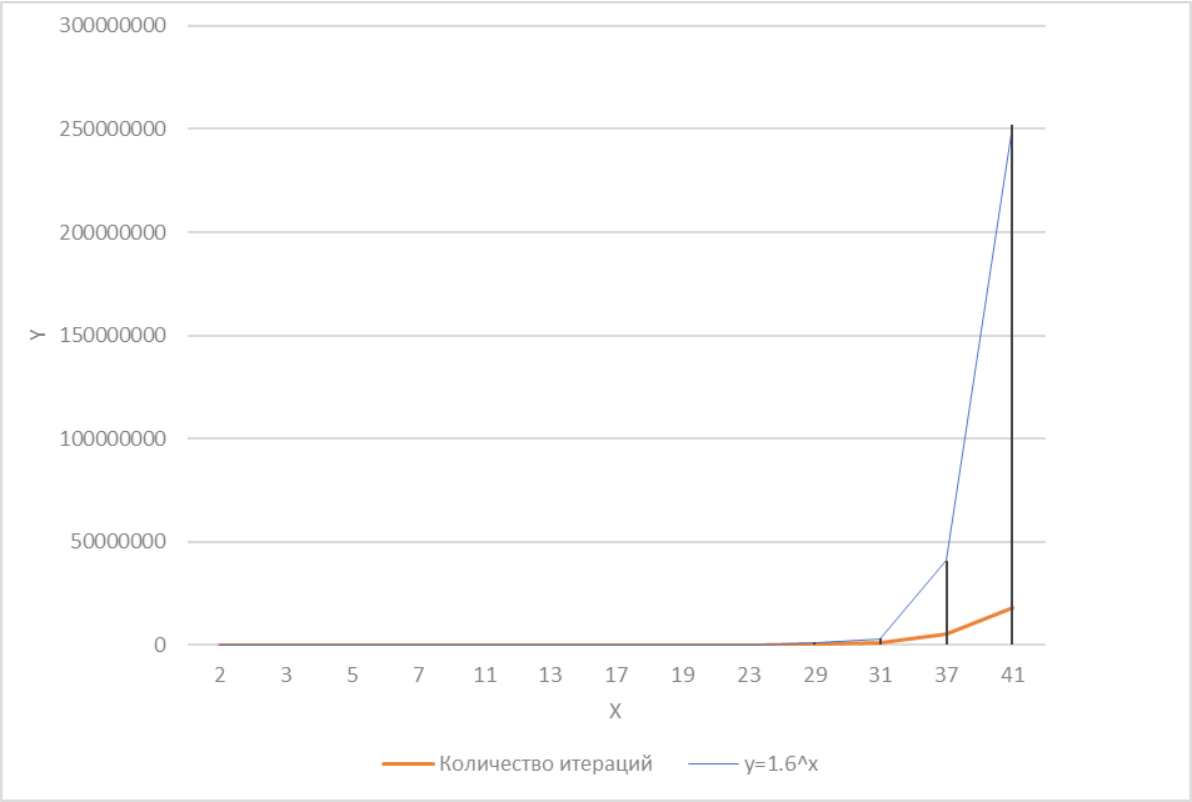

Рисунок 1 - Сложность алгоритма для простых чисел

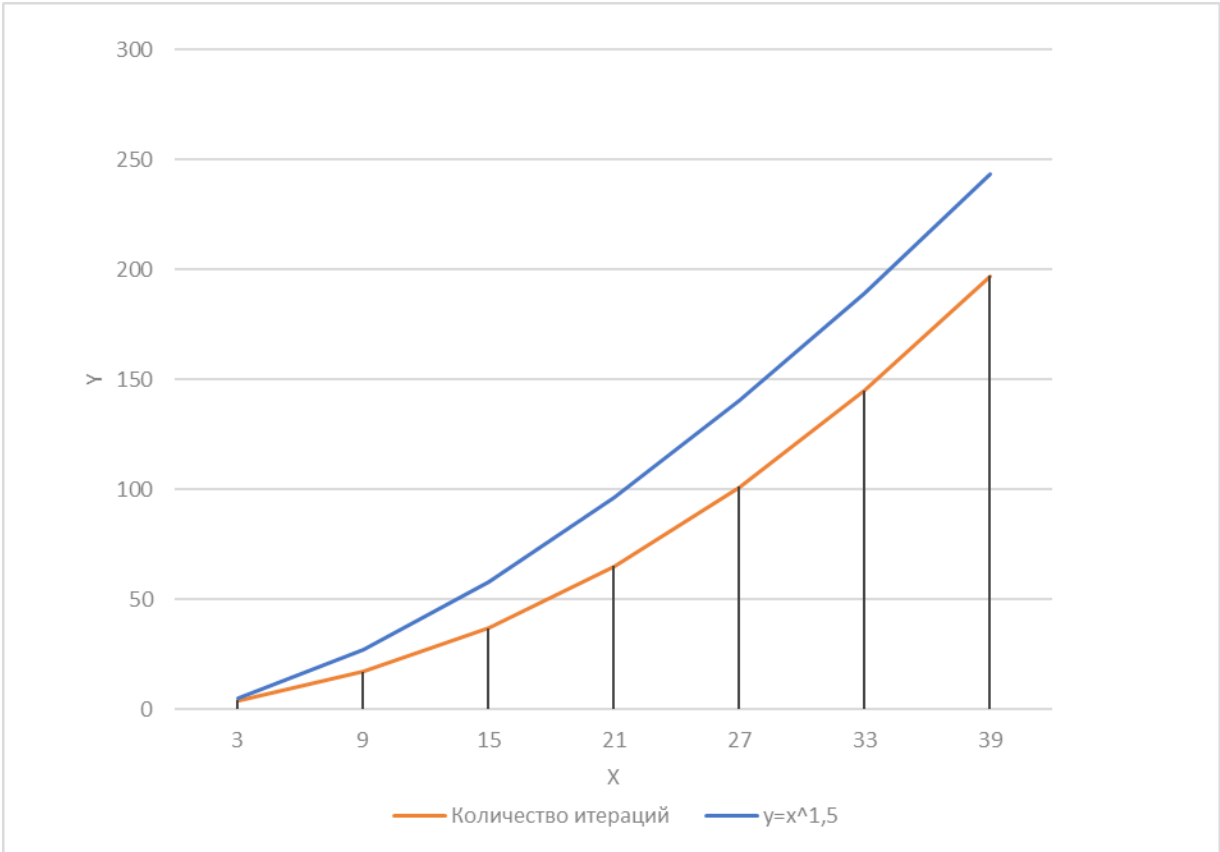

Рисунок 2 - Сложность алгоритма для чисел кратных трем

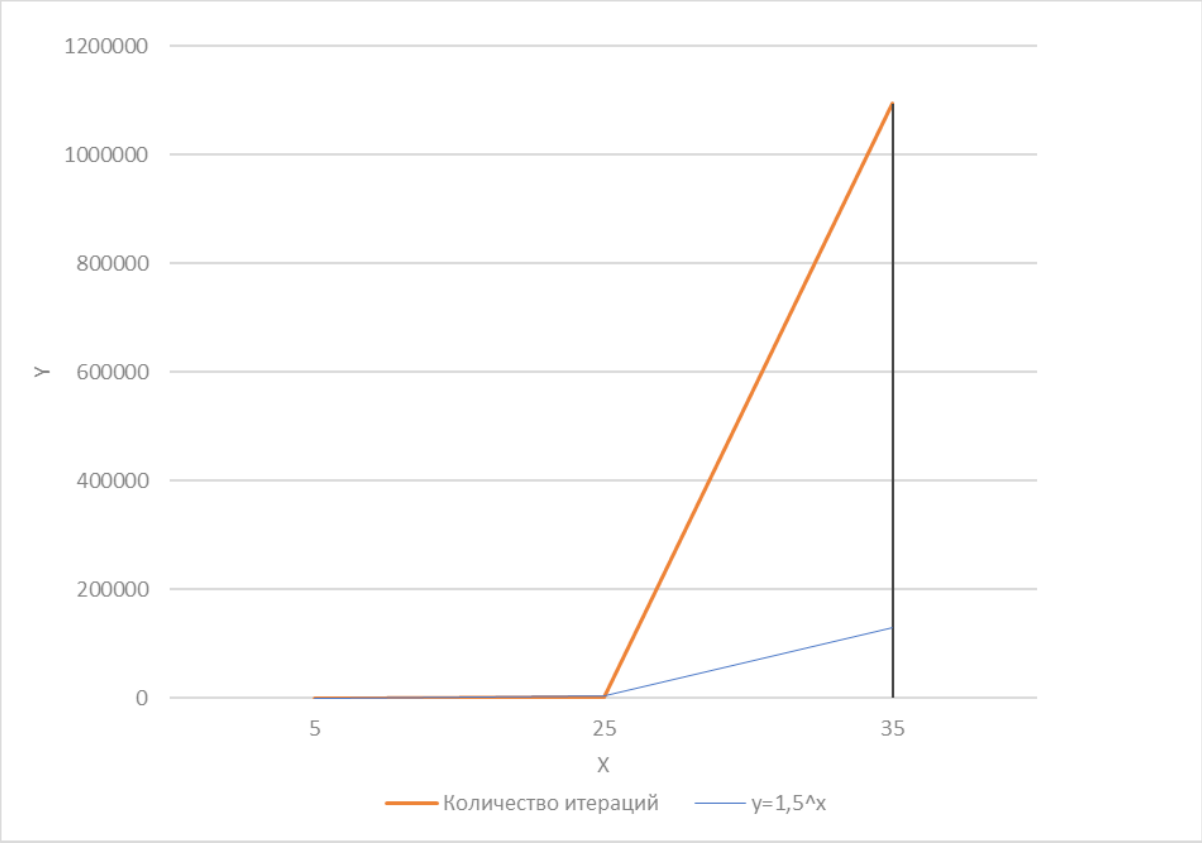

Рисунок 3 - Сложность алгоритма для чисел кратных пяти

Тестирование.

Программа была в компиляторе G++ в среде разработки Qt в операционной системе Linux Ubuntu 17.10.

В ходе тестирования ошибок выявлено не было.

Корректные тестовые случаи представлены в приложении А.

Выводы.

В ходе выполнения данной работы был изучен метод поиска с возвратом. В данной работе был реализован алгоритм, находящий такое разбиение квадрата меньшими квадратами, что их количество является наименьшим. Сложность разработанного алгоритма по количеству итераций функции поиска не превышает $1{,}6^n$.

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Выходные данные
10	4 1 1 5 6 1 5 1 6 5 6 6 5
15	6 1 1 10 11 1 5 1 11 5 6 11 5 11 6 5 11 11 5
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
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
29	14 1 1 15 16 1 14 1 16 14 15 16 2 15 18 5 15 23 7 16 15 1 17 15 3 20 15 3

	20 18 3 20 21 2 22 21 1 22 22 8 23 15 7

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>

using namespace std;

struct point{
   int x;
   int y;
};

class square{
private:
   unsigned short int size;
   bool** is_empty;
   int* min;
   unsigned short int minnum;
   unsigned short int num;
   int* current;
public:
   point search_empty(){
      point empty;
      for(int x = 0; x < size; x++)
      {
         for(int y = 0; y < size; y++)
            if(is_empty[x][y] == 0)
            {
               empty.x = x;
               empty.y = y;
               return empty;
            }
      }
      empty.x = size+1;
      empty.y = size+1;
      return empty;
   }
   square(unsigned short int n):size(n),is_empty(size ? new bool*[size] :
nullptr), min(size ? new int[3*size*size] :
nullptr),minnum(size*size+1),num(0),current(size ? new int[3*size*size] :
nullptr),number_op(0)
   {
      for(int i = 0; i<size;i++)
      {
         is_empty[i] = new bool[size];
         for(int j = 0; j<size; j++)
            is_empty[i][j] = false;
      }
   }
   ~square(){
      delete [] is_empty;
```

```
    delete [] min;
    delete [] current;
}
void add(point p, unsigned short int s)
{
    current[num*3] = p.x+1;
    current[num*3+1] = p.y+1;
    current[num*3+2] = s;
    num++;
    for(int i = p.x; i < p.x+s ; i++)
        for(int j = p.y ; j < p.y+s ; j++)
            is_empty[i][j] = true;
}
void backtrack(point p, int s)
{
    current[num*3] = 0;
    current[num*3+1] = 0;
    current[num*3+2] = 0;
    num--;
    for(int i = p.x; i < p.x+s ; i++)
        for(int j = p.y ; j < p.y+s ; j++)
            is_empty[i][j] = false;
}
void search()
{
    for(int i = size-1;i>0;i--)
    {
        point p = search_empty();
        if(p.x > size || p.y > size)
        {
            if(num < minnum)
            {
                for(int j = 0; j < num*3;j++)
                {
                    min[j] = current[j];
                }
                minnum = num;
            }
            break;
        }
        if(num+1 == minnum)
        {
            break;
        }
        int a;
        int b;
        for(a = p.x; a<size;a++)
            if(is_empty[a][p.y] != false)
                break;
        for(b = p.y; b<size;b++)
            if(is_empty[p.x][b] != false)
```

```
                break;
        if(a - p.x < i || b - p.y  <i)
        {
           i = a - p.x  > b - p.y ? b - p.y : a - p.x;
        }
        add(this->search_empty(),i);
        this->search();
        this->backtrack(p,i);
    }
}
void set_first()
{
    point first_sq{0,0};
    point second_sq{0,0};
    point third_sq{0,0};
    if(!(size%2))
    {
        add(first_sq, size/2);
        second_sq.x = size/2;
        third_sq.y = size/2;
        add(second_sq,size/2);
        add(third_sq,size/2);
        return;
    }
    if(!(size%3))
    {
        add(first_sq, 2*size/3);
        second_sq.x = 2*size/3;
        third_sq.y = 2*size/3;
        add(second_sq,size/3);
        add(third_sq,size/3);
        return;
    }
    if(!(size%5))
    {
        add(first_sq, 3*size/5);
        second_sq.x = 3*size/5;
        third_sq.y = 3*size/5;
        add(second_sq, 2*size/5);
        add(third_sq,2*size/5);
        return;
    }
    add(first_sq, size/2+1);
    second_sq.x = size/2+1;
    third_sq.y = size/2+1;
    add(second_sq,size - (size/2+1));
    add(third_sq,size - (size/2+1));
    return;
}

void print()
```

```
    {
        cout<<minnum<<endl;
        for(int j = 0; j<minnum;j++)
        {
            for(int k = 0; k<3; k++)
                cout<<min[j*3 + k]<<' ';
            cout<<endl;
        }
    }
};

int main()
{
    int n;
    cin>>n;
    square s(n);
    s.set_first();
    s.search();
    s.print();
}
```