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 _____ Ханова Ю.А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2019

Цель работы

Ознакомиться с алгоритмом поиска с возвратом, написать программу для квадрирования квадрата с заданной стороной с использованием поиска с возвратом.

Реализация задачи

В ходе работы был разработан алгоритм, позволяющий разбить заданный квадрат на более мелкие квадраты. Были применены некоторые модификации для улучшения работы алгоритма (в случае если сторона исходного квадрата четная, кратна 5 или 3). Для остальных случаев сначала вставляется наибольший квадрат в левый верхний угол (со стороной больше половины исходного квадрата), после – два одинаковых квадрата справа и снизу от него и вызывается алгоритм поиска с возвратом.

В данной работе для решения поставленной цели был написан класс `Square` и несколько методов, содержащихся в данном классе.

Конструктор класса создает массив целых чисел, заполняет его нулями. Так же был написан конструктор копирования.

Метод `void Split()` определяет тип модификации и в соответствии с этим вызывает метод `Insert(int x, int y, int k)` или `Insert(int x, int y, int k, Point &len)`, который сначала вставляет большой квадрат, а затем два маленьких, после этого для стороны кратной 5 и общего случая вызываются функции бэктрекинга.

Перебор с вохвратом реализован двумя функциями:

Метод `void Backtracking_of_square(int x, int y, int k)` в цикле создает массив точек и передает в функцию `Backtracking`, затем удаляет созданный массив.

Метод `bool Backtracking(const int &x0, const int &y0, const int &k, const int &num, int r, int x, int y)` основная рекурсивная функция, которая проверяет какое минимальное

количество квадратов поместится в данный квадрат при помощи поиска с возвратом.

Метод `bool Check_this_position(int x, int y, int k)` проверяет возможно ли поместить в данную точку квадрат с даной стороной.

Метод `void Remove_elements(int x, int y, int k)` удаляет квадрат с заданной длинной.

Метод void print() выводит ответ на экран.

Исходный код программы представлен в Приложении А.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04 с использованием компилятора g++. В других ОС и компиляторах тестирование не проводилось. Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении Б.

Так же было проведено исследование алгоритма. Было изучено необходимое количество итераций при некоторых длинах квадрата. Результаты исследования представлены в табл. 1. В исследовании были использованы простые числа.

Таблица 1 – Исследование алгоритма

Длина стороны квадрата	Кол-во итераций
3	3
5	14
7	60
11	950
13	2370
17	16578
19	65461
23	250838
29	2046067

Из результатов исследования алгоритмов видно, что сложность алгоритма не превышает 2^n, где n – сторона квадрата. Память алгоритм выделяет только для двух двумерных массивов, поэтому по памяти сложность константная.

Выводы

В ходе выполнения лабораторной работы был изучен метод поиска с возвратом, была написана программа для квадрирования квадрата заданной длины и исследован алгоритм. Была определена сложность алгоритма по количеству вызовов функции, осуществляющей поиск с возвратом: сложность алгоритма не превышает 2n.

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ

lr1.cpp

```
#include <iostream>

using namespace std;


struct Point{
    int x;
    int y;
    int len; //сторона квадрата
};


class Square{
public:

  Square(int k)
  {
     arr = new int[(k + 1)*(k + 1)]();
     this->num = 0;
     this->k = k;
  }

  ~Square()
  {
     delete arr;
  }

  void Insert(int x, int y, int k)
    {
      for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
          arr[(i + y)*this->k + (j + x)] = k;
      num++;
    }


  void Insert(int x, int y, int k, Point &len)
    {
      for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
          arr[(i + y)*this->k + (j + x)] = k;

      len.x = x;
      len.y = y;
      len.len = k;
      num++;
    }
```

```
  bool Check_this_position(int x, int y, int k)
    {
      if ((k > this->k - x)
        ||
        (k > this->k - y)) return false;

      for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
          if (arr[(i + y)*this->k + (j + x)]) return 0;

      return true;
    }


  void Remove_elements(int x, int y, int k)
    {
      for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
          arr[(i + y)*this->k + (j + x)] = 0;

      num--;
    }


  void Backtracking_of_square(int x, int y, int k)
    {

      for (int i = 2; i < k*k; i++)
      {
        point = new Point[i + 3];
        if (Backtracking(x, y, k, i, 0, 0, 0)) break;
        delete point;
      }

    }


  bool Backtracking(const int &x0, const int &y0, const int &k, const
int &num, int r, int x, int y)
    {
      if (num == r)
      {
        for (int i = 0; i < k; i++)
        {
          if (!(arr[(y0 + i)*this->k + (x0 + k - 1)]
            &&            arr[(y0 + k - 1)*this->k + (x0 + i)])) return
false;
        }

        return true;
      }
```

```
      while ((arr[(y0 + y)*this->k + (x0 + x)]) || (x == k))
      {
        if (x >= k)
        {
          if (y == k) return true;
          y++;
          x = 0;
        }
        else x++;
      }


      for (int i = 1; i<k; i++)
      {
        if (Check_this_position(x0 + x, y0 + y, i))
        {
          Insert(x0 + x, y0 + y, i, point[r]); r++;

          if (Backtracking(x0, y0, k, num, r, x + i, y)) return true;

          Remove_elements(x0 + x, y0 + y, i); r--;
        }
      }

      return false;
    }

  void Print()
  {
    cout << num;
      for (int i = 0; i < num; i++)
           cout << endl << point[i].x + 1 << " " << point[i].y + 1 << "
" << point[i].len;
  }

  void Split()
  {
     if (!(k % 2))
     {
          point = new Point[4];
          Insert(0, 0, k / 2, point[0]);
          Insert(0, k / 2, k / 2, point[1]);
          Insert(k / 2, 0, k / 2, point[2]);
          Insert(k / 2, k / 2, k / 2, point[3]);

          return;
     }
     if (!(k % 3))
     {
          point = new Point[6];
          Insert(0, 0, k * 2 / 3, point[0]);
          Insert(0, k * 2 / 3, k / 3, point[1]);
```

```
            Insert(k * 2 / 3, 0, k / 3, point[2]);
            Insert(k / 3, k * 2 / 3, k / 3, point[3]);
            Insert(k * 2 / 3, k / 3, k / 3, point[4]);
            Insert(k * 2 / 3, k * 2 / 3, k / 3, point[5]);

            return;
      }
      if (!(k % 5))
      {
            point = new Point[8];
            Insert(0, 0, k * 3 / 5, point[0]);
            Insert(k * 3 / 5, k / 5, k * 2 / 5, point[1]);
            Insert(k / 5, k * 3 / 5, k * 2 / 5, point[2]);
            Insert(k * 3 / 5, k * 3 / 5, k * 2 / 5, point[3]);

            Backtracking(0, 0, k, 8, 4, k * 3 / 5, 0);
            return;
      }


      int m = k / 2 + 1;
      Insert(0, 0, m);
      Insert(0, m, m - 1);
      Insert(m, 0, m - 1);

      Backtracking_of_square(m - 1, m - 1, m);

      point[num - 3].x = 0; point[num - 3].y = 0; point[num - 3].len = m;
      point[num - 2].x = 0; point[num - 2].y = m; point[num - 2].len = m
- 1;
      point[num - 1].x = m; point[num - 1].y = 0; point[num - 1].len = m
- 1;
  }

private:
  int* arr; //заполнение квадрата
  int k; //размер квадрата
  Point* point; //верхний левый угол и сторона квадрата
  int num; //колличество обрезков


};



int main()
{
      int n;
      cin >> n;
      Square Big_square(n);
  Big_square.Split();
  Big_square.Print();
```

```
	return 0;
}
```

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ

Тест 1:
6
4
1 1 3
1 4 3
4 1 3
4 4 3
Тест 2:
9
6
1 1 6
1 7 3
7 1 3
4 7 3
7 4 3
7 7 3
Тест 3:
25
8
1 1 15
16 6 10
6 16 10
16 16 10
16 1 5
21 1 5
1 16 5
1 21 5
Тест 4:
37
15
20 19 1
21 19 3
24 19 7
31 19 7
19 20 2
19 22 5
24 26 2
26 26 12
19 27 4
23 27 1
23 28 3
19 31 7
1 1 19

1 20 18
20 1 18