Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

Отчёт

по расчётно-графической работе

«Протокол доказательства с нулевым знанием для задачи “Раскраска графа”»

Выполнил:

студент группы ИП-111

Кузьменок Д. В.

ФИО студента

Работу проверила: Дьячкова И. С.

ФИО преподавателя

Новосибирск 2024 г.

# **Содержание**

[**Содержание** 2](#_Toc181897381)

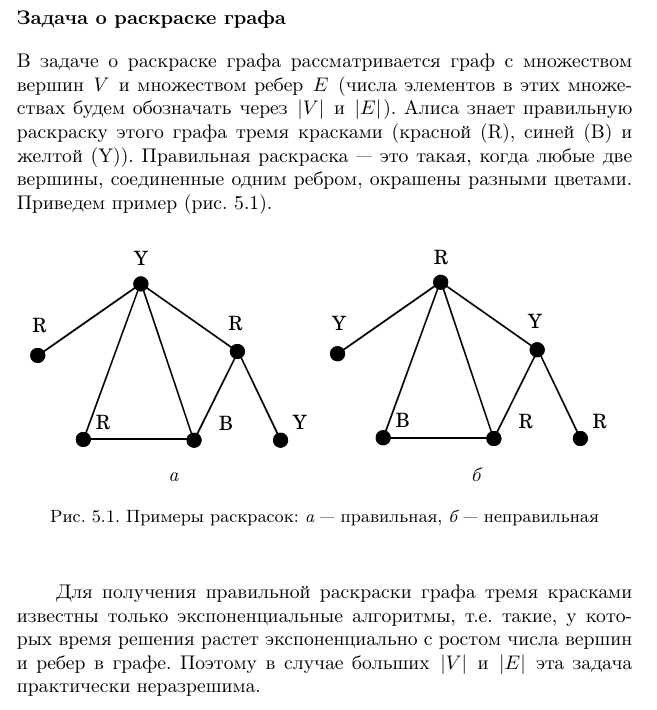
[**Задание** 3](#_Toc181897382)

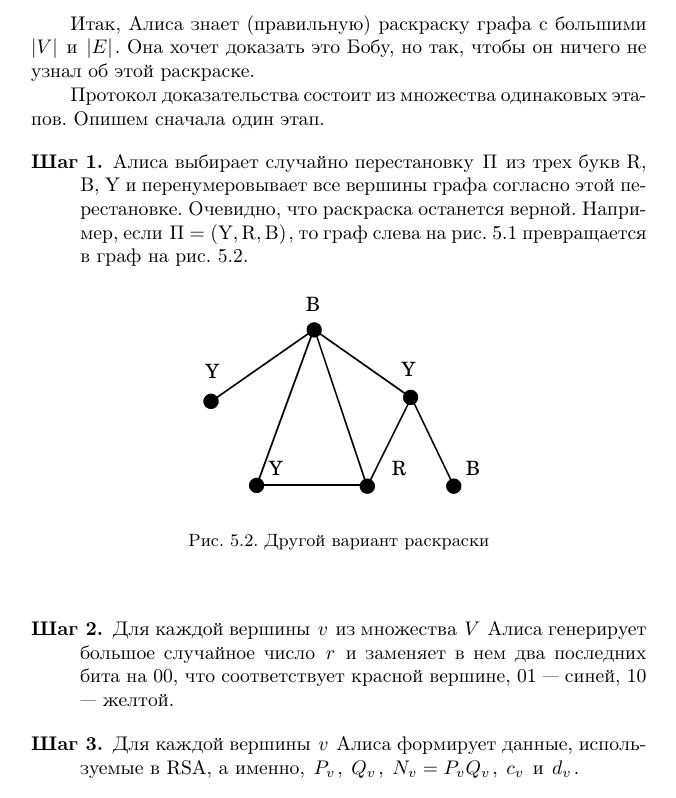
[**Реализация программы** 6](#_Toc181897383)

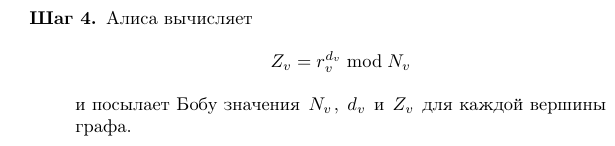
[**Результаты работы** 7](#_Toc181897384)

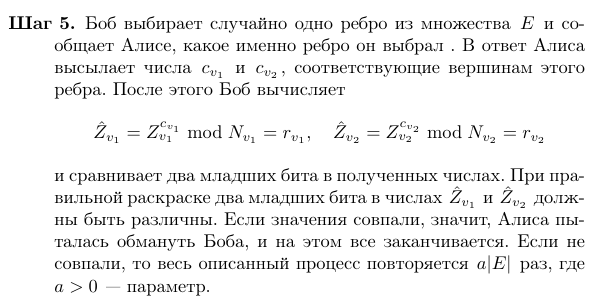
[**Листинг программы** 11](#_Toc181897385)

# **Задание**









Информацию о графах необходимо считывать из файла. В файле описание графа будет определяться следующим образом:

1) в первой строке файла содержатся два числа n <1001 и m ≤, количество вершин графа и количество рёбер соответственно;

2) в последующих m строках содержится информация о рёбрах графа, каждое из которых описывается с помощью двух чисел (номера вершин, соединяемых этим ребром);

3) в зависимости от варианта указывается необходимая дополнительная информация: в первом варианте перечисляются цвета вершин графа (этот пункт можно вынести в отдельный файл).

# **Реализация программы**

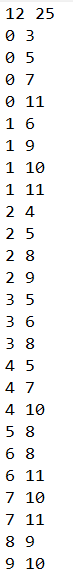
Для реализации задания я создал 4 класса, каждый из которых выполняет свою функцию, необходимую для корректной работы программы.

* Класс MainOperations: предоставляет необходимые функции, используемые для генерации значений в протоколе RSA (быстрое возведение числа в степень по модулю; генерация простого числа с проверкой, основанной на теореме Ферма; генерация случайного большого числа; обобщённый алгоритм Евклида; поиск взаимно простого числа).
* Класс RSAMethods: генерирует необходимые для RSA значение (большие простые значение P и Q; N = P \* Q; φ = (P - 1) \* (Q - 1); D и C – открытый и секретный ключи соответственно).
* Классы Edge и Graph: первый позволяет создать структуру графа, путем сохранения значений Source и Destination ребра (отправная точка и точка прибытия); второй же – реализует алгоритм раскраски графа, используя алгоритм Уэлша-Пауэлла (метод GetSortedVerticesByDegree), где вершины сначала сортируются по убыванию инцидентных ребер, а затем идет присваивание цветов в зависимости от цвета соседа (метод ColorGraph).
* Класс Program: реализует чтение файла, сохранение рёбер, сохраняет цвета вершин и присваивание каждой вершине необходимых параметров (r, N, D, C, Z). Затем в методе BobChecks идет проверка корректности раскраски графа, где сравниваются последние биты у каждой вершины, путем возведения числа Z в степень C по модулю N.

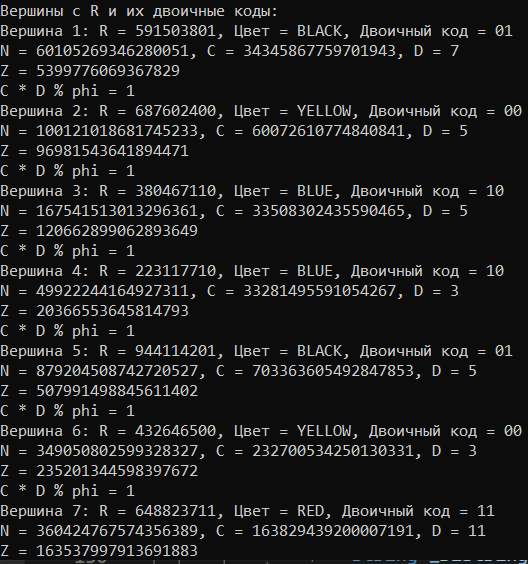
# **Результаты работы**

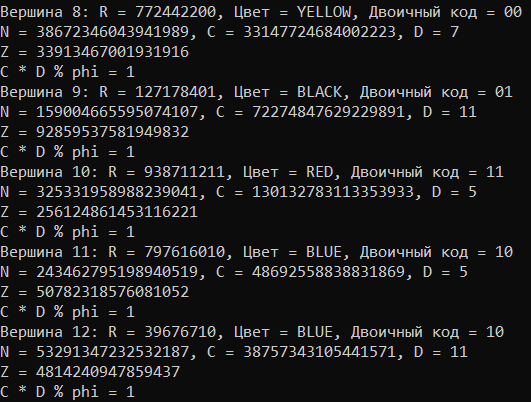
Результат работы программы:

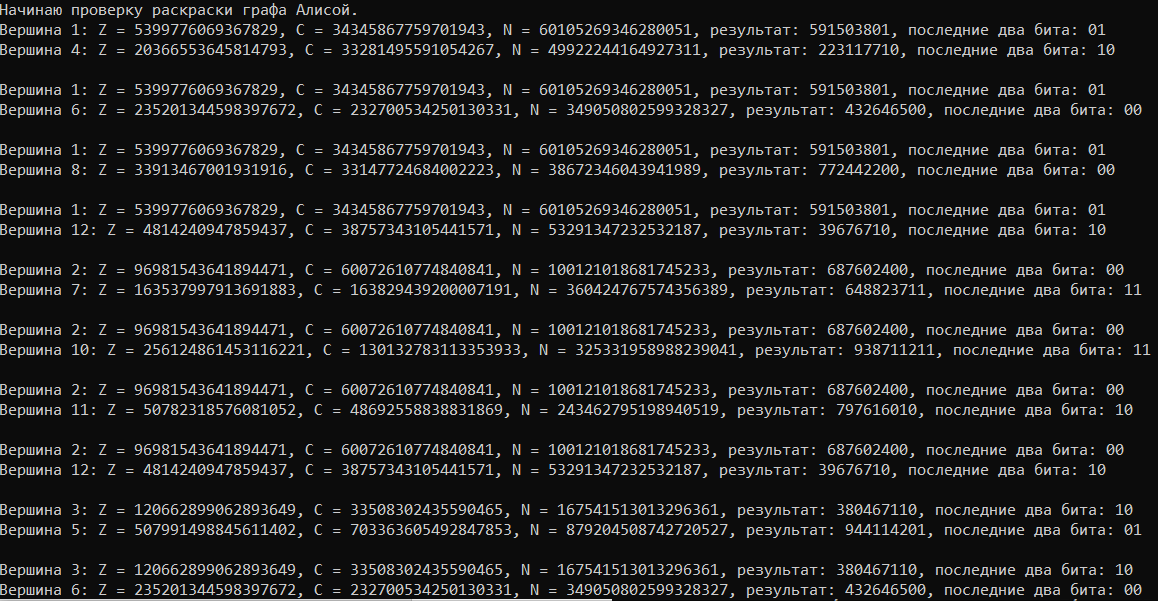
Входной файл выглядит следующим образом:

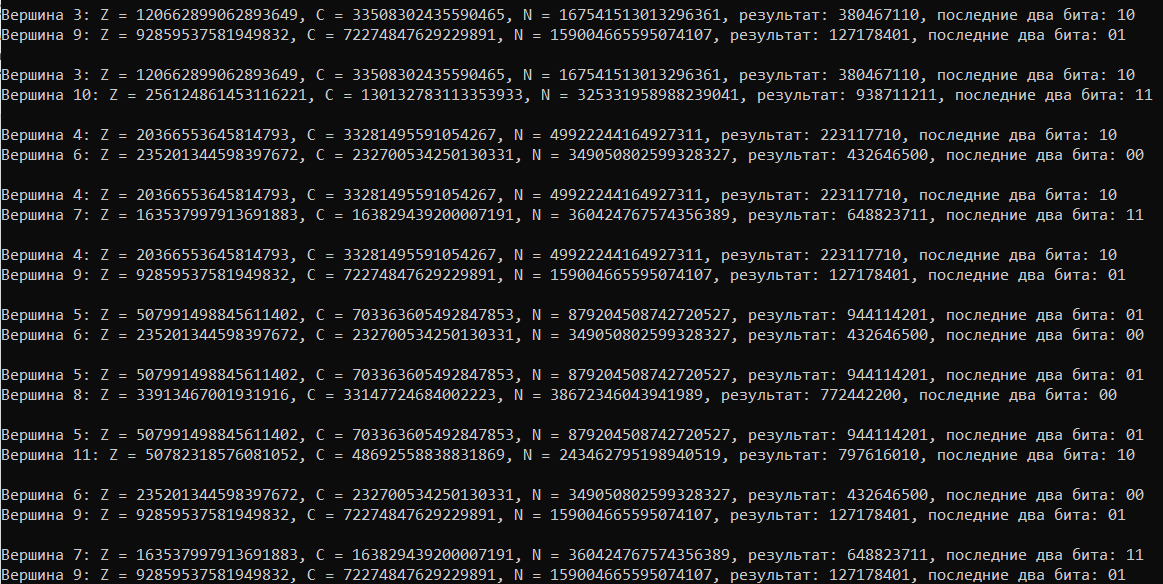


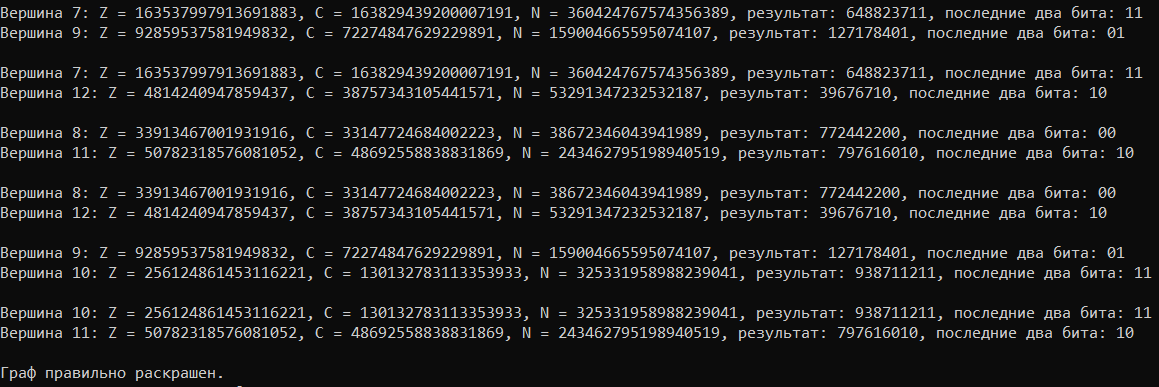
При правильной раскраске графа:



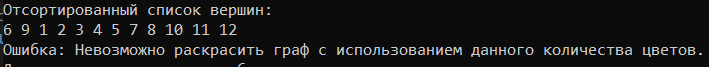




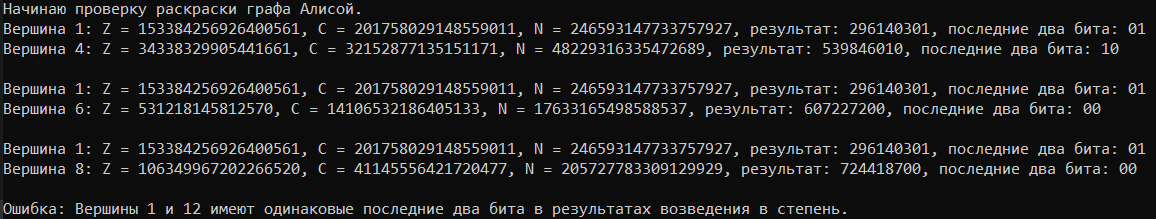




Также алгоритм может самостоятельно определить, достаточно ли будет цветов для раскраски текущего графа. В случае, если цветов мало, будет выведено следующее сообщение:



Если же мы специально захотим испортить раскраску графа, сделав её неправильной, то Боб при проверке выведет информацию, какие две вершины были неправильно раскрашены:



# **Листинг программы**

**MainOperations.cs:**

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Numerics**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **RGR**

{

**class** **MainOperations**

{

**public** **static** **ulong** **FastPow**(**ulong** a, **ulong** x, **ulong** p)

{

**ulong** result = **1**;

List<**ulong**> temp = **new** List<**ulong**> { a % p };

**ulong** t = (**ulong**)Math.Floor(Math.Log(x, **2**));

List<**ulong**> binaryExponent = ToBinary(x);

**for** (**int** i = **1**; i <= (**int**)t; i++)

{

temp.Add((temp[i - **1**] \* temp[i - **1**]) % p);

}

**for** (**int** i = **0**; i <= (**int**)t; i++)

{

**if** (binaryExponent[i] != **0**)

{

result = (result \* temp[i]) % p;

}

}

**return** result;

}

**public** **static** BigInteger **FastPow1**(BigInteger a, BigInteger x, BigInteger p)

{

BigInteger result = **1**;

List<BigInteger> temp = **new** List<BigInteger> { a % p };

**long** t = (**long**)Math.Floor(Math.Log((**double**)x, **2**));

List<BigInteger> binaryExponent = ToBinary1(x);

**for** (**int** i = **1**; i <= t; i++)

{

temp.Add((temp[i - **1**] \* temp[i - **1**]) % p);

}

**for** (**int** i = **0**; i <= t; i++)

{

**if** (binaryExponent[i] != **0**)

{

result = (result \* temp[i]) % p;

}

}

**return** result;

}

**private** **static** List<**ulong**> ToBinary(**ulong** x)

{

List<**ulong**> result = **new** List<**ulong**>();

**while** (x != **0**)

{

result.Add(x & **1**);

x = x >> **1**;

}

**return** result;

}

**public** **static** List<BigInteger> ToBinary1(BigInteger x)

{

List<BigInteger> result = **new** List<BigInteger>();

**while** (x != **0**)

{

result.Add(x & **1**);

x = x >> **1**;

}

**return** result;

}

**public** **static** **ulong** **GenerateModule**(**ulong** left, **ulong** right, Random rnd)

{

//Random rnd = new Random();

**ulong** p = **0**;

**ulong** range = right - left + **1**;

**while** (**true**)

{

//p = rnd.Next((int)left, (int)right);

p = (**ulong**)(rnd.NextDouble() \* range) + left;

**if** (IsPrime(p))

{

**return** p;

}

}

}

//Ферма

**public** **static** **bool** **IsPrime**(**ulong** number)

{

Random rnd = **new** Random();

**if** (number <= **1**) **return** **false**;

**else** **if** (number == **2**) **return** **true**;

**for** (**ulong** i = **0**; i < **100**; i++)

{

**ulong** a = (**ulong**)rnd.Next(**2**, (**int**)number - **1**);

**if** (FastPow(a, number - **1**, number) != **1** || Gcd(number, a) != **1**) **return** **false**;

}

**return** **true**;

}

**public** **static** **ulong** **Gcd**(**ulong** a, **ulong** b)

{

**while** (b != **0**)

{

**ulong** r = a % b;

a = b;

b = r;

}

**return** a;

}

**public** **static** BigInteger **Gcd1**(BigInteger a, BigInteger b)

{

**while** (b != **0**)

{

BigInteger r = a % b;

a = b;

b = r;

}

**return** a;

}

**public** **static** **ulong** **GenerateExponent**(**ulong** left, **ulong** right, Random rnd)

{

**return** (**ulong**)rnd.Next((**int**)left, (**int**)right);

}

**public** **static** (BigInteger, BigInteger, BigInteger) EvklidSolve1(BigInteger a, BigInteger b)

{

(BigInteger, BigInteger, BigInteger) U = (a, **1**, **0**);

(BigInteger, BigInteger, BigInteger) V = (b, **0**, **1**);

**while** (V.Item1 != **0**)

{

BigInteger q = U.Item1 / V.Item1;

(BigInteger, BigInteger, BigInteger) T = (U.Item1 % V.Item1, U.Item2 - q \* V.Item2, U.Item3 - q \* V.Item3);

U = V;

V = T;

}

**return** U;

}

**public** **static** **ulong** **GenerateCoprime**(BigInteger p)

{

**ulong** result = **0**;

**for** (**ulong** i = **2**; i < p; i++)

{

**if** (MainOperations.Gcd1(p, i) == **1**)

{

result = i;

**break**;

}

}

**return** result;

}

}

}

**Graph.cs:**

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **RGR**

{

//класс ребра

**public** **class** **Edge**

{

**public** **int** Source { **get**; **set**; }

**public** **int** Destination { **get**; **set**; }

**public** **Edge**(**int** source, **int** destination)

{

Source = source;

Destination = destination;

}

}

**public** **class** **Graph**

{

**public** **static** **string**[] colors = { "YELLOW", "RED", "BLUE", "BLACK" };

**public** **static** Dictionary<**int**, **string**> ColorGraph(List<Edge> edges, **int** numVertices)

{

// Перемешиваем цвета случайным образом

Random rnd = **new** Random();

colors = colors.OrderBy(x => rnd.Next()).ToArray();

// Словарь для хранения цвета каждой вершины

Dictionary<**int**, **string**> vertexColors = **new** Dictionary<**int**, **string**>();

// Инициализация всех вершин как не раскрашенных

**for** (**int** i = **0**; i < numVertices; i++)

{

vertexColors[i] = **null**;

}

// Сортируем вершины по степени

List<**int**> sortedVertices = GetSortedVerticesByDegree(edges, numVertices);

Console.WriteLine("Отсортированный список вершин:");

**foreach** (**var** vertex **in** sortedVertices)

{

Console.Write($"{vertex + 1} ");

}

Console.WriteLine();

// Раскрашиваем вершины по порядку, начиная с вершины с максимальной степенью

**foreach** (**int** vertex **in** sortedVertices)

{

// Список цветов, которые уже использованы смежными вершинами

List<**string**> usedColors = **new** List<**string**>();

// Проверяем цвета соседей

**foreach** (**var** edge **in** edges.Where(e => e.Source == vertex || e.Destination == vertex))

{

**int** neighbor = (edge.Source == vertex) ? edge.Destination : edge.Source;

**if** (vertexColors.ContainsKey(neighbor) && vertexColors[neighbor] != **null**)

{

usedColors.Add(vertexColors[neighbor]);

}

}

// Проверяем, есть ли доступный цвет

**if** (!colors.Except(usedColors).Any())

{

// Если нет доступного цвета, то не удается раскрасить граф

**throw** **new** **Exception**("Невозможно раскрасить граф с использованием данного количества цветов.");

}

// Выбираем первый доступный цвет

**foreach** (**var** color **in** colors)

{

**if** (!usedColors.Contains(color))

{

vertexColors[vertex] = color;

**break**;

}

}

}

**return** vertexColors;

}

// Метод для получения отсортированного списка вершин по степени

**private** **static** List<**int**> GetSortedVerticesByDegree(List<Edge> edges, **int** numVertices)

{

// Создаем словарь для хранения степеней вершин

Dictionary<**int**, **int**> degrees = **new** Dictionary<**int**, **int**>();

**for** (**int** i = **0**; i < numVertices; i++)

{

degrees[i] = **0**;

}

// Подсчитываем степени вершин

**foreach** (**var** edge **in** edges)

{

degrees[edge.Source]++;

degrees[edge.Destination]++;

}

// Сортируем вершины по степени

**return** degrees.OrderByDescending(x => x.Value).Select(x => x.Key).ToList();

}

}

}

**RSAMethods.cs:**

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Numerics**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **RGR**

{

**class** **RSAMethods**

{

**internal** BigInteger N;

**internal** BigInteger C;

**internal** **ulong** D;

**internal** BigInteger phi;

**private** **static** Random rnd = **new** Random();

**public** **RSAMethods**()

{

**ulong** P = MainOperations.GenerateModule(**1000000**, **1000000000**, rnd);

**ulong** Q = MainOperations.GenerateModule(**1000000**, **1000000000**, rnd);

N = P \* Q;

phi = (P - **1**) \* (Q - **1**);

D = MainOperations.GenerateCoprime(phi);

C = MainOperations.EvklidSolve1(D, phi).Item2;

**if**(C < **0**)

{

C += phi;

}

}

}

}

**Program.cs:**

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.IO**;

**using** **System.Linq**;

**using** **System.Numerics**;

**namespace** **RGR**

{

**public** **class** **Program**

{

**static** **void** **Main**(**string**[] args)

{

List<Edge> edges = **new** List<Edge>();

**int** numVertices = **0**;

List<Dictionary<**string**, BigInteger>> informationVertexes = **new** List<Dictionary<**string**, BigInteger>>();

List<BigInteger> zVertex = **new** List<BigInteger>();

**try**

{

**string** filePath = "graph.txt";

**string**[] lines = File.ReadAllLines(filePath);

**string**[] firstLine = lines[**0**].Split(' ');

numVertices = **int**.Parse(firstLine[**0**]);

**int** numEdges = **int**.Parse(firstLine[**1**]);

**for** (**int** i = **1**; i <= numEdges; i++)

{

**string**[] edgeData = lines[i].Split(' ');

**int** source = **int**.Parse(edgeData[**0**]);

**int** destination = **int**.Parse(edgeData[**1**]);

edges.Add(**new** Edge(source, destination));

}

}

**catch** (Exception ex)

{

Console.WriteLine($"Ошибка при чтении файла: {ex.Message}");

**return**;

}

**try**

{

Dictionary<**int**, **string**> vertexColors = Graph.ColorGraph(edges, numVertices);

//vertexColors[11] = "BLUE";

Console.WriteLine("Раскраска графа:");

**for** (**int** i = **0**; i < numVertices; i++)

{

Console.WriteLine($"Вершина {i + 1}: {vertexColors[i]}");

}

List<**ulong**> vertexesWithR = **new** List<**ulong**>(numVertices);

Random random = **new** Random();

**for** (**int** i = **0**; i < numVertices; i++)

{

**ulong** r = MainOperations.GenerateExponent(**1000000**, **1000000000**, random);

Console.WriteLine($"r = {r}");

vertexesWithR.Add(r);

}

Dictionary<**string**, **string**> colorToBinary = **new** Dictionary<**string**, **string**>();

**for** (**int** i = **0**; i < Graph.colors.Length; i++)

{

**string** binaryCode = Convert.ToString(i, **2**).PadLeft(**2**, '0');

colorToBinary.Add(Graph.colors[i], binaryCode);

}

**for**(**int** i = **0**; i < numVertices; i++)

{

**string** binaryColor = colorToBinary[vertexColors[i]];

**string** tempNumber = vertexesWithR[i].ToString();

tempNumber = tempNumber.Substring(**0**, tempNumber.Length - **2**) + binaryColor;

vertexesWithR[i] = **ulong**.Parse(tempNumber);

}

**for**(**int** i = **0**; i < numVertices; i++)

{

RSAMethods rsa = **new** RSAMethods();

Dictionary<**string**, BigInteger> vertexInfo = **new** Dictionary<**string**, BigInteger>()

{

{"N", rsa.N },

{"D", rsa.D },

{"C", rsa.C },

{"phi", rsa.phi }

};

informationVertexes.Add(vertexInfo);

Console.WriteLine($"R = {vertexesWithR[i]}, D = {informationVertexes[i]["D"]}, N = {informationVertexes[i]["N"]}, C = {informationVertexes[i]["C"]}, phi = {informationVertexes[i]["phi"]}");

zVertex.Add(MainOperations.FastPow1(vertexesWithR[i], informationVertexes[i]["D"], informationVertexes[i]["N"]));

}

Console.WriteLine("\nВершины с R и их двоичные коды:");

**for** (**int** i = **0**; i < numVertices; i++)

{

Console.WriteLine($"Вершина {i + 1}: R = {vertexesWithR[i]}, Цвет = {vertexColors[i]}, Двоичный код = {colorToBinary[vertexColors[i]]}");

Console.WriteLine($"N = {informationVertexes[i]["N"]}, C = {informationVertexes[i]["C"]}, D = {informationVertexes[i]["D"]}");

Console.WriteLine($"Z = {zVertex[i]}");

Console.WriteLine($"C \* D % phi = {MainOperations.FastPow1(informationVertexes[i]["C"] \* informationVertexes[i]["D"], 1, informationVertexes[i]["phi"])}");

}

BobChecks(edges, informationVertexes, zVertex);

}

**catch** (Exception ex)

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

}

**static** **void** **BobChecks**(List<Edge> edges, List<Dictionary<**string**, BigInteger>> informationVertexes, List<BigInteger> zVertex)

{

**string** lastTwoBits\_r1 = "", lastTwoBits\_r2 = "";

Console.WriteLine("Начинаю проверку раскраски графа Алисой.");

**for** (**int** i = **0**; i < edges.Count; i++)

{

**int** source = edges[i].Source;

**int** destination = edges[i].Destination;

BigInteger nSource = informationVertexes[source]["N"];

BigInteger cSource = informationVertexes[source]["C"];

BigInteger nDestination = informationVertexes[destination]["N"];

BigInteger cDestination = informationVertexes[destination]["C"];

BigInteger \_r1 = MainOperations.FastPow1(zVertex[source], cSource, nSource);

BigInteger \_r2 = MainOperations.FastPow1(zVertex[destination], cDestination, nDestination);

**string** \_r1String = \_r1.ToString();

**string** \_r2String = \_r2.ToString();

// Проверка на равенство последних двух битов

**if** (\_r1String.Length >= **2** && \_r2String.Length >= **2**)

{

lastTwoBits\_r1 = \_r1String.Substring(\_r1String.Length - **2**);

lastTwoBits\_r2 = \_r2String.Substring(\_r2String.Length - **2**);

**if** (lastTwoBits\_r1 == lastTwoBits\_r2)

{

Console.WriteLine($"Ошибка: Вершины {source + 1} и {destination + 1} имеют одинаковые последние два бита в результатах возведения в степень.");

**return**;

}

}

Console.WriteLine($"Вершина {source + 1}: Z = {zVertex[source]}, C = {cSource}, N = {nSource}, результат: {\_r1}, последние два бита: {lastTwoBits\_r1}");

Console.WriteLine($"Вершина {destination + 1}: Z = {zVertex[destination]}, C = {cDestination}, N = {nDestination}, результат: {\_r2}, последние два бита: {lastTwoBits\_r2}");

Console.WriteLine();

}

Console.WriteLine("Граф правильно раскрашен.");

}

}

}