Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет имени первого Президента России Б.Н.Ельцина»
Институт радиоэлектроники и информационных технологий – РТФ

Анализ сложности алгоритмов сортировки строк

Отчёт по лабораторной работе

по дисциплине «Алгоритмы, структуры данных и анализ сложности»

Вариант 2

Выполнил: Тарасенко А. Р.
студент гр. РИ-230913
Преподаватель:

Доцент, к.ф. -м.н.
Трофимов С.П.

Екатеринбург, 2025

Оглавление

Задание

Написать класс, который содержит целое число со знаком в виде массива однобайтовых элементов. Реализовать конструкторы, деструктор, перегрузить операции: аддитивные (+, –), мультипликативные (*, /, %), сравнения (==, !=, <, >), взятие обратного по заданному модулю.

Написать функцию шифрования строки с помощью алгоритм RSA.

Зашифровать/расшифровать текстовый файл с помощью открытого RSA-ключа.

Теоретическая часть

RSA ключи и шифрование данных

В отличии от симметричных алгоритмов шифрования, имеющих всего один ключ для шифрования и расшифровки информации, в алгоритме RSA используется 2 ключа – открытый (публичный) и закрытый (приватный). Публичный ключ шифрования передаётся по открытым каналам связи, а приватный всегда держится в секрете. Но зачем нужно целых два ключа и как они работают?

В ассиметричной криптографии и алгоритме RSA, в частности, публичный и приватный ключи являются двумя частями одного целого и неразрывны друг с другом. Для шифрования информации используется открытый ключ, а для её расшифровки приватный.

Например, если мы перемножим числа 592939 и 592967 мы получим число 351593260013. Но как имея только число 351593260013 узнать числа 592939 и 592967? А если каждое из этих двух чисел будут длиной более 1000 знаков? Это называется «сложность задачи факторизации произведения двух больших простых чисел», т.е. в одну сторону просто, а в обратную невероятно сложно.

Теперь рассмотрим процедуру создания публичного и приватного ключей:

1. Выбираем два случайных простых числа p и q
2. Вычисляем их произведение: N = p * q
3. Вычисляем функцию Эйлера: (N) = (p-1) * (q-1)
4. Выбираем число e, которое меньше (N) и является взаимно простым с (N) (не имеющих общих делителей друг с другом, кроме 1).

Ищем число d, обратное числу e по модулю (N) .Т.е. остаток от деления (d*e) и (N) должен быть равен 1. Найти его можно через расширенный алгоритм Евклида.

После произведённый вычислений, у нас будут:

e и n – открытый ключ

d и n – закрытый ключ

Для работы просто необходимо понимание быстрого возведения в степень

Алгоритм быстрого возведения в степень (возведение в квадрат и умножение) работает по следующему принципу:

1. **Если степень чётная**, число возводится в квадрат, а степень делится на два.
   Пример: a^8 = (a^4)^2
2. **Если степень нечётная**, результат умножается на основание, а степень уменьшается на один, превращаясь в чётную.
   Пример: a^9=a·a^8

Алгоритм повторяет эти шаги, пока степень не станет равной нулю.

Почему это эффективно:

Обычное умножение требует n-1 операций для a^n. Быстрое возведение в степень сокращает количество операций до log2(n), так как степень уменьшается вдвое при каждом шаге. Это существенно экономит время и ресурсы, особенно при больших степенях.[3]

Инструкция пользователя

В первую очередь пользователю нужно заполнить файл message.txt нужным ему число. Затем достаточно лишь запустить программу и результат работы будет доступен в выводе программы.

```
PS D:\Learn\Алгосы\LR2> dotnet run .
D:\Learn\Алгосы\LR2\LongNumerics.cs(8,14): warning CS0660: "BigInt" определяет оператор "==" или оператор "!=", но не переопределяет Object.Equals(object o).
D:\Learn\Алгосы\LR2\LongNumerics.cs(8,14): warning CS0661: "BigInt" определяет оператор "==" или оператор "!=", но не переопределяет Object.GetHashCode().
D:\Learn\Алгосы\LR2\Program.cs(13,23): warning CS8600: Преобразование литерала, допускающего значение NULL или возможного значения NULL в тип, не допускающий значение NULL.
D:\Learn\Алгосы\LR2\Program.cs(14,37): warning CS8604: Возможно, аргумент-ссылка, допускающий значение NULL, для параметра "str" в "BigInt.BigInt(string str)".
Encrypted: 7044973
Decrypted: 9213123
PS D:\Learn\Алгосы\LR2>
```

Рисунок 1 – Работа программы.

Инструкция программиста

Основу всей программы содержит в себе класс BigInt.

Он поддерживает базовые арифметические операции (+, -, *, /, %), операции сравнения, а также модульное возведение в степень и нахождение обратного элемента по модулю. Предназначен для случаев, когда стандартные типы (int, long) недостаточны по диапазону значений.

Поля

int[] digits — массив цифр числа, где цифры хранятся в порядке от старшего разряда к младшему (аналогично обычной записи числа).

bool isNegative — флаг, указывающий, является ли число отрицательным.

Конструкторы

BigInt(string str) — конструктор, принимающий строку и преобразующий её в массив цифр. Поддерживает знак «минус» в начале строки.

private BigInt(int[] digits, bool isNegative) — вспомогательный внутренний конструктор, используемый для создания новых объектов на основе массива цифр и знака.

Основные методы и операторы

print() — выводит число в консоль, включая знак.

Операторы ==, !=, >, < — сравнивают два числа с учётом знака и длины массивов цифр.

Арифметические операторы:

+, - — реализуют корректное сложение и вычитание чисел с одинаковыми или разными знаками.

* — реализует умножение в столбик с учётом переноса разрядов.

/ — деление по цифрам, возвращает частное. Используется наивный алгоритм.

% — вычисляет остаток от деления. При отрицательном остатке корректирует его в положительное значение.

Вспомогательные методы

AddDigits, SubtractDigits — выполняют поразрядное сложение и вычитание двух

массивов цифр.

CompareAbs — сравнивает два массива цифр по модулю.

TrimLeadingZeros — удаляет незначащие нули в начале числа.

Математические методы

ModPow(BigInt base, BigInt exponent, BigInt modulus) — реализует алгоритм быстрого возведения в степень по модулю (бинарный метод).

ExtendedGCD(BigInt a, BigInt b, out BigInt x, out BigInt y) — расширенный алгоритм Евклида для нахождения НОД и коэффициентов Безу.

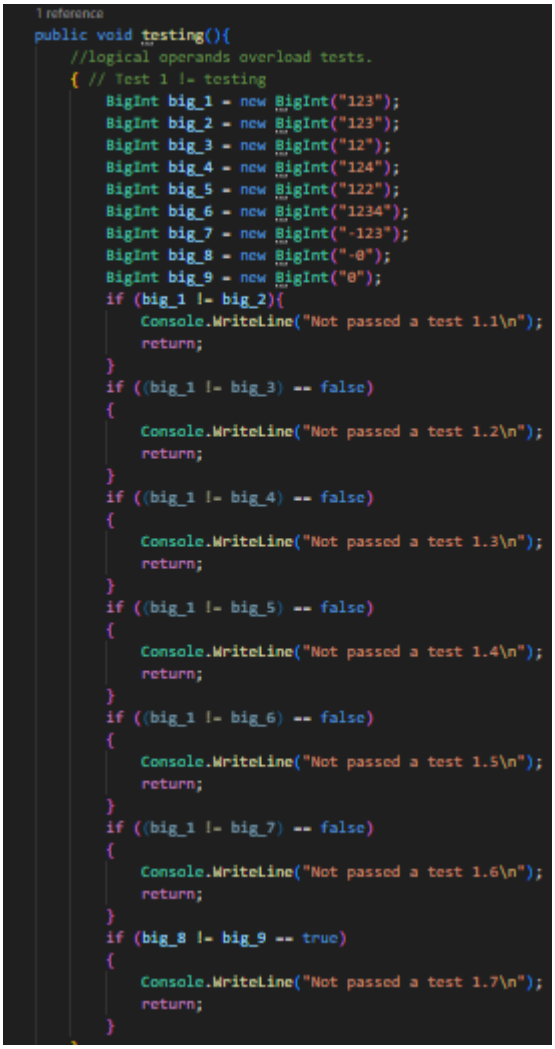
ModInverse(BigInt a, BigInt m) — вычисляет обратный элемент по модулю (если он существует), используя расширенный алгоритм Евклида.

Принцип работы

Все вычисления выполняются над массивами цифр, что позволяет обходить ограничения стандартных числовых типов. Массив цифр формируется таким образом, чтобы старшие разряды находились в начале массива. Отдельно хранится знак числа. При выполнении арифметических операций и сравнений знаки учитываются отдельно.

Тестирование

Кроме основных и вспомогательных, программа также содержит функцию файл Tests.cs Она содержит тесты, проверяющие корректность работы алгоритма сортировки и выполняется в момент запуска программы. Если один или несколько тестов завершились неудачно, на консоль будут выведены соответствующие сообщения, и программа завершит работу. Часть кода тестов представлена ниже, с полным кодом можно ознакомиться в Приложении. Весь код тестов занимает 460 строк и включает в себя тесты 12 функций в различных ситуациях:



```
1 reference
public void testing(){
    //logical operands overload tests.
    { // Test 1 != testing
        BigInt big_1 = new BigInt("123");
        BigInt big_2 = new BigInt("123");
        BigInt big_3 = new BigInt("12");
        BigInt big_4 = new BigInt("124");
        BigInt big_5 = new BigInt("122");
        BigInt big_6 = new BigInt("1234");
        BigInt big_7 = new BigInt("-123");
        BigInt big_8 = new BigInt("-0");
        BigInt big_9 = new BigInt("0");
        if (big_1 != big_2){
            Console.WriteLine("Not passed a test 1.1\n");
            return;
        }
        if ((big_1 != big_3) == false)
        {
            Console.WriteLine("Not passed a test 1.2\n");
            return;
        }
        if ((big_1 != big_4) == false)
        {
            Console.WriteLine("Not passed a test 1.3\n");
            return;
        }
        if ((big_1 != big_5) == false)
        {
            Console.WriteLine("Not passed a test 1.4\n");
            return;
        }
        if ((big_1 != big_6) == false)
        {
            Console.WriteLine("Not passed a test 1.5\n");
            return;
        }
        if ((big_1 != big_7) == false)
        {
            Console.WriteLine("Not passed a test 1.6\n");
            return;
        }
        if (big_8 != big_9 == true)
        {
            Console.WriteLine("Not passed a test 1.7\n");
            return;
        }
```

Рисунок 2 - Часть кода тестирующей функции Test

Выводы

В данной работе мы познакомились с одним из алгоритмов шифрования-RSA, написали программу шифрования длинных чисел с его использованием, Полученная программа работает исправно и позволяет достаточно быстро шифровать числа.

Литература

1. Книги:

1. Таненбаум Э. С., Уэзеролл Х. Современные операционные системы. — М.: Питер, 2020. — 1120 с. — Гл. 12. Криптография. — С. 935–942.
2. Стахов А. П., Шень А. М. Введение в криптографию. — М.: МЦНМО, 2019. — 264 с. — Гл. 4. Шифр RSA. — С. 112–125.

2. Электронные ресурсы:

3) Алгоритмы быстрого возведения в степень// Википедия : [сайт]. — URL: https://habr.com/ru/companies/otus/articles/779396/ (дата обращения: 26.05.2025).
4) Как работает RSA // Habr : [сайт]. — URL: https://habr.com/ru/articles/112953/ (дата обращения: 26.05.2025).

Приложение А

Ссылка на исходный код

Ссылка на репозиторий с реализацией алгоритма пузырьковой сортировки:
GitHub: https://github.com/delilit/LongNumerics_RSA

Приложение Б

Программный код

Основной код:

```
using System;
    using System.Collections.Generic;
    using System.IO.Pipelines;
    using System.Text;
    using System.Threading.Channels;

    namespace Longnumerics{
    public class BigInt
    {
       private int[] digits;
       private bool isNegative;

       private BigInt(int[] digits, bool isNegative)
       {
          this.digits = digits;
          this.isNegative = isNegative;
       }
       public BigInt(string str){
          if (str.StartsWith("-"))
          {
             str = str.Substring(1);
             isNegative = true;
          }

          int[] result = new int[str.Length];
```

```
            for (int i =0 ; i<str.Length; i++){
                if (!char.IsDigit(str[i]))
                    throw new FormatException("Input string contains non-numeric
characters.");
                result[i] = str[i]-'0';
            }
            digits = result;
        }
        public void print(){
            if (this.isNegative){
                Console.Write("-");
            }
            foreach (var value in this.digits){
                Console.Write($"{value}");
            }
            Console.Write("\n");
        }
        public static bool operator == (BigInt a, BigInt b){
            if (a.digits.Length == 1 && a.digits[0] == 0 && b.digits.Length == 1
&& b.digits[0] == 0) return true;
            if (a.isNegative != b.isNegative)
                return false;
            if (a.digits.Length != b.digits.Length){
                return false;
            }
            for (int i = 0; i<a.digits.Length; i++){
                if (a.digits[i]!=b.digits[i]){
                    return false;
                }
            }
```

```
            return true;
        }

        public static bool operator != (BigInt a, BigInt b){
            if (a.digits.Length == 1 && a.digits[0] == 0 && b.digits.Length == 1
&& b.digits[0] == 0) return false;
            if (a.isNegative != b.isNegative)
                return true;
            if (a.digits.Length != b.digits.Length){
                return true;
            }
            for (int i = 0; i<a.digits.Length; i++){
                if (a.digits[i]!=b.digits[i]){
                    return true;
                }
            }
            return false;
        }

        public static bool operator > (BigInt a, BigInt b){
            if (a.digits.Length == 1 && a.digits[0] == 0 && b.digits.Length == 1
&& b.digits[0] == 0) return false;
                bool notequal = a.isNegative != b.isNegative;
                if (notequal){
                if (a.isNegative == false & b.isNegative == true){
                    return true;
                }
                else return false;
                }
                if (a.isNegative == false){
```

```
                if (a.digits.Length > b.digits.Length) return true;
                if (a.digits.Length < b.digits.Length) return false;
                for (int i = 0; i< a.digits.Length; i++){
                    if (a.digits[i] > b.digits[i]) return true;
                    if (a.digits[i] < b.digits[i]) return false;
                }
            }
            else{
                if (a.digits.Length > b.digits.Length) return false;
                if (a.digits.Length < b.digits.Length) return true;
                for (int i = 0; i< a.digits.Length; i++){
                    if (a.digits[i] > b.digits[i]) return false;
                    if (a.digits[i] < b.digits[i]) return true;
                }
        }
        return false;
    }

    public static bool operator < (BigInt a, BigInt b){
            if (a.digits.Length == 1 && a.digits[0] == 0 && b.digits.Length == 1
&& b.digits[0] == 0) return false;
            bool notequal = a.isNegative != b.isNegative;
            if (notequal){
            if (a.isNegative == false & b.isNegative == true){
                return false;
            }
            else return true;
            }
            if (a.isNegative == false){
                if (a.digits.Length > b.digits.Length) return false;
```

```
            if (a.digits.Length < b.digits.Length) return true;
            for (int i = 0; i< a.digits.Length; i++){
                if (a.digits[i] > b.digits[i]) return false;
                if (a.digits[i] < b.digits[i]) return true;
            }
        }
        else{
            if (a.digits.Length > b.digits.Length) return true;
            if (a.digits.Length < b.digits.Length) return false;
            for (int i = 0; i< a.digits.Length; i++){
                if (a.digits[i] > b.digits[i]) return true;
                if (a.digits[i] < b.digits[i]) return false;
            }
        }
        return false;
    }
    public static BigInt operator + (BigInt a, BigInt b)
{
    if (a.isNegative == b.isNegative)
    {
        // Сложение по модулю
        List<int> result = AddDigits(a.digits, b.digits);
        return new BigInt(result.ToArray(), a.isNegative);
    }
    else
    {
        // Разные знаки — вычитание по модулю
        int cmp = CompareAbs(a.digits, b.digits);
        if (cmp == 0)
        {
```

```
            return new BigInt(new int[] { 0 }, false); // 123 + (-123) = 0
        }
        else if (cmp > 0)
        {
            List<int> result = SubtractDigits(a.digits, b.digits);
            return new BigInt(result.ToArray(), a.isNegative);
        }
        else
        {
            List<int> result = SubtractDigits(b.digits, a.digits);
            return new BigInt(result.ToArray(), b.isNegative);
        }
    }
}
public static BigInt operator - (BigInt a, BigInt b)
{
    if (a.isNegative != b.isNegative)
    {
        // Превращаем в сложение: a - (-b) = a + b
        List<int> result = AddDigits(a.digits, b.digits);
        return new BigInt(result.ToArray(), a.isNegative);
    }
    else
    {
        // Оба одного знака — вычитание по модулю
        int cmp = CompareAbs(a.digits, b.digits);
        if (cmp == 0)
        {
            return new BigInt(new int[] { 0 }, false);
        }
```

```
            else if (cmp > 0)
            {
                List<int> result = SubtractDigits(a.digits, b.digits);
                return new BigInt(result.ToArray(), a.isNegative);
            }
            else
            {
                List<int> result = SubtractDigits(b.digits, a.digits);
                return new BigInt(result.ToArray(), !a.isNegative); // знак
противоположен a
            }
        }
    }
    public static BigInt operator * (BigInt a, BigInt b)
    {
        int[] result = new int[a.digits.Length + b.digits.Length];

        for (int i = a.digits.Length - 1; i >= 0; i--)
        {
            for (int j = b.digits.Length - 1; j >= 0; j--)
            {
                int mul = a.digits[i] * b.digits[j];
                int p1 = i + j;
                int p2 = i + j + 1;

                int sum = mul + result[p2];
                result[p2] = sum % 10;
                result[p1] += sum / 10;
            }
        }
```

```
    int startIndex = 0;
    while (startIndex < result.Length - 1 && result[startIndex] == 0)
        startIndex++;

    int[] trimmedResult = new int[result.Length - startIndex];
    Array.Copy(result, startIndex, trimmedResult, 0, trimmedResult.Length);

    bool resultIsNegative = a.isNegative != b.isNegative;
    return new BigInt(trimmedResult, resultIsNegative);
}
public static BigInt operator / (BigInt dividend, BigInt divisor)
{
    if (divisor.digits.Length == 1 && divisor.digits[0] == 0)
        throw new DivideByZeroException("Cannot divide by zero.");

    bool resultNegative = dividend.isNegative != divisor.isNegative;

    int[] quotient = new int[dividend.digits.Length];
    List<int> remainder = new List<int>();

    for (int i = 0; i < dividend.digits.Length; i++)
    {
        remainder.Add(dividend.digits[i]);
        remainder = TrimLeadingZeros(remainder);

        int x = 0;
        while (CompareAbs(remainder.ToArray(), divisor.digits) >= 0)
        {
            remainder = SubtractDigits(remainder.ToArray(), divisor.digits);
```

```
            x++;
        }

        quotient[i] = x;
    }

    quotient = TrimLeadingZeros(quotient.ToList()).ToArray();

    if (quotient.Length == 0)
        return new BigInt(new int[] { 0 }, false);

    return new BigInt(quotient, resultNegative);
}

public static BigInt operator % (BigInt a, BigInt b)
{
    if (b == new BigInt("0"))
        throw new DivideByZeroException();

    BigInt quotient = a / b;
    BigInt product = quotient * b;
    BigInt remainder = a - product;

    if (remainder.isNegative)
        remainder = remainder + b;

    return remainder;
}
```

```
private static List<int> TrimLeadingZeros(List<int> digits)
{
   while (digits.Count > 1 && digits[0] == 0)
      digits.RemoveAt(0);
   return digits;
}



private static List<int> AddDigits(int[] a, int[] b)
{
   List<int> result = new List<int>();
   int carry = 0;
   int i = a.Length - 1;
   int j = b.Length - 1;

   while (i >= 0 || j >= 0 || carry > 0)
   {
      int sum = carry;
      if (i >= 0)
      {
      sum += a[i];
      i--;
      }
      if (j >= 0){
         sum += b[j];
         j--;
      }
      result.Insert(0, sum % 10);
      carry = sum / 10;
```

```
    }
    return result;
}

private static List<int> SubtractDigits(int[] a, int[] b)
{
    List<int> result = new List<int>();
    int borrow = 0;
    int i = a.Length - 1;
    int j = b.Length - 1;

    while (i >= 0)
    {
        int diff = a[i] - borrow - (j >= 0 ? b[j] : 0);
        if (diff < 0)
        {
            diff += 10;
            borrow = 1;
        }
        else
        {
            borrow = 0;
        }
        result.Insert(0, diff);
        i--; j--;
    }

    while (result.Count > 1 && result[0] == 0)
    {
        result.RemoveAt(0);
```

```
        }

        return result;
    }

    private static int CompareAbs(int[] a, int[] b)
    {
        if (a.Length != b.Length)
            return a.Length.CompareTo(b.Length);
        for (int i = 0; i < a.Length; i++)
        {
            if (a[i] != b[i])
                return a[i].CompareTo(b[i]);
        }
        return 0;
    }
    public static BigInt ModPow(BigInt baseValue, BigInt exponent, BigInt
modulus) //Алгоритм быстрого возведения в степень является
необходимостью.
    {
        BigInt result = new BigInt("1");
        baseValue = baseValue % modulus; // по свойствам остатков мы имеем
право работать чисто с ними, ибо умножение чисел и остатков в нашем
случае есть одно и то же.

        while (exponent != new BigInt("0"))
        {
            if ((exponent.digits[^1] % 2) == 1)
                result = (result * baseValue) % modulus;
                if (exponent == new BigInt("1")) return result; //ещё одна
```

оптимизация, созданная для того, чтобы не делать лишние операции умножения.

```
            exponent = exponent / new BigInt("2");
            baseValue = (baseValue * baseValue) % modulus;
        }

        return result;
    }
    // Слишком медленно

    // public static BigInt ModPow(BigInt baseValue, BigInt exponent, BigInt modulus)
    // {
    //     baseValue = baseValue % modulus;

    //     while (exponent != new BigInt("0"))
    //     {
    //         exponent = exponent - new BigInt("1");
    //         baseValue = (baseValue * baseValue);
    //     }

    //     return baseValue % modulus;
    // }
    public static BigInt ExtendedGCD(BigInt a, BigInt b, out BigInt x, out BigInt y)
    {
        if (b == new BigInt("0"))
        {
            x = new BigInt("1");
```

```
            y = new BigInt("0");
            return a;
        }

        BigInt x1, y1;
        BigInt gcd = ExtendedGCD(b, a % b, out x1, out y1);
        x = y1;
        y = x1 - (a / b) * y1;
        return gcd;
    }

    public static BigInt ModInverse(BigInt a, BigInt m)
    {
        BigInt x, y;
        BigInt g = ExtendedGCD(a, m, out x, out y);

        if (g != new BigInt("1"))
            throw new ArgumentException("Inverse does not exist (GCD != 1).");

        if (x.isNegative)
            x = (x + m);

        return x % m;
    }
    }
}
```

Код программы:

```
    using System;
    using System.Text;
    using System.Globalization;
```

```
using System.Numerics;
using System.Reflection.Metadata.Ecma335;
using Longnumerics;
using Testes;
class Program{
   static void drain(){
      // Запустить тесты
      Tester test = new Tester();
      test.testing();

      StreamReader sr = new
StreamReader("D:\\Learn\\Алгосы\\LR2\\message.txt");
      string line = sr.ReadLine();
      BigInt message = new BigInt(line);


      int number = 3;
      int[] numbers = new int[] {0, 0};

      while (true){

         if (is_prime(number)){
            numbers[0] = numbers[1];
            numbers[1] = number;
         }
         number += 1;
         BigInt result = new BigInt((numbers[0] * numbers[1]).ToString());
         if (result > message) {break;}
         }
```

```
    BigInt p = new BigInt(numbers[0].ToString());
    BigInt q = new BigInt(numbers[1].ToString());
    BigInt n = p * q;
    BigInt phi = (p - new BigInt("1")) * (q - new BigInt("1"));
    BigInt e = new BigInt("3");


    BigInt y = new BigInt("0");
    BigInt x = new BigInt("0");
    while (BigInt.ExtendedGCD(e, phi, out x, out y) != new BigInt("1"))
    {
       e += new BigInt("2");
    }
    BigInt d = BigInt.ModInverse(e, phi);


    BigInt encrypted = BigInt.ModPow(message, e, n);
    Console.Write("Encrypted: "); encrypted.print();

    BigInt decrypted = BigInt.ModPow(encrypted, d, n);
    Console.Write("Decrypted: "); decrypted.print();
    }

public static bool is_prime(int number){
    for (int i = 2; i<Math.Pow(number, 0.5); i++){
       if (number % i == 0){
       return false;
       }
    }
```

```
    return true;
}
static void Main(){
    Encoding ascii = Encoding.ASCII;
    String unicodeString =
      "This unicode string contains two characters " +
      "with codes outside the ASCII code range, " +
      "Pi (\u03a0) and Sigma (\u03a3).";
  Console.WriteLine("Original string:");
  Console.WriteLine(unicodeString);

  // Save the positions of the special characters for later reference.
  int indexOfPi = unicodeString.IndexOf('\u03a0');
  int indexOfSigma = unicodeString.IndexOf('\u03a3');

  // Encode the string.
  Byte[] encodedBytes = ascii.GetBytes(unicodeString);
  Console.WriteLine();
  Console.WriteLine("Encoded bytes:");
  foreach (Byte b in encodedBytes)
  {
     Console.Write($"[{b}]");
  }
  Console.WriteLine();

  String decodedString = ascii.GetString(encodedBytes);
  Console.WriteLine();
  Console.WriteLine("Decoded bytes:");
  Console.WriteLine(decodedString);
}
```

```
    }
```

Код тестов:

```
using System;
        using System.Collections.Generic;
        using System.IO.Pipelines;
        using System.Text;
        using System.Threading.Channels;using Longnumerics;
        namespace Testes{
            public class Tester{
                public Tester(){

                }
                public void testing(){
                    //logical operands overload tests.
                    { // Test 1 != testing
                        BigInt big_1 = new BigInt("123");
                        BigInt big_2 = new BigInt("123");
                        BigInt big_3 = new BigInt("12");
                        BigInt big_4 = new BigInt("124");
                        BigInt big_5 = new BigInt("122");
                        BigInt big_6 = new BigInt("1234");
                        BigInt big_7 = new BigInt("-123");
                        BigInt big_8 = new BigInt("-0");
                        BigInt big_9 = new BigInt("0");
                        if (big_1 != big_2){
                            Console.WriteLine("Not passed a test 1.1\n");
                            return;
                        }
                        if ((big_1 != big_3) == false)
                        {
```

```
            Console.WriteLine("Not passed a test 1.2\n");
            return;
        }
        if ((big_1 != big_4) == false)
        {
            Console.WriteLine("Not passed a test 1.3\n");
            return;
        }
        if ((big_1 != big_5) == false)
        {
            Console.WriteLine("Not passed a test 1.4\n");
            return;
        }
        if ((big_1 != big_6) == false)
        {
            Console.WriteLine("Not passed a test 1.5\n");
            return;
        }
        if ((big_1 != big_7) == false)
        {
            Console.WriteLine("Not passed a test 1.6\n");
            return;
        }
        if (big_8 != big_9 == true)
        {
            Console.WriteLine("Not passed a test 1.7\n");
            return;
        }
    }
```

```
{ // Test 2 == testing
   BigInt big_1 = new BigInt("123");
   BigInt big_2 = new BigInt("123");
   BigInt big_3 = new BigInt("12");
   BigInt big_4 = new BigInt("124");
   BigInt big_5 = new BigInt("122");
   BigInt big_6 = new BigInt("1234");
   BigInt big_7 = new BigInt("-123");
   BigInt big_8 = new BigInt("-0");
   BigInt big_9 = new BigInt("0");
   if ((big_1 == big_2) == false){
      Console.WriteLine("Not passed a test 2.1\n");
      return;
   }
   if ((big_1 == big_3))
   {
      Console.WriteLine("Not passed a test 2.2\n");
      return;
   }
   if ((big_1 == big_4))
   {
      Console.WriteLine("Not passed a test 2.3\n");
      return;
   }
   if ((big_1 == big_5))
   {
      Console.WriteLine("Not passed a test 2.4\n");
      return;
   }
   if ((big_1 == big_6))
```

```
    {
        Console.WriteLine("Not passed a test 2.5\n");
        return;
    }
    if ((big_1 == big_7))
    {
        Console.WriteLine("Not passed a test 2.6\n");
        return;
    }
    if (big_8 == big_9 == false)
    {
        Console.WriteLine("Not passed a test 2.7\n");
        return;
    }
}
{
    // Test 3 > testing
    BigInt big_1 = new BigInt("123");
    BigInt big_2 = new BigInt("123");
    BigInt big_3 = new BigInt("12");
    BigInt big_4 = new BigInt("124");
    BigInt big_5 = new BigInt("122");
    BigInt big_6 = new BigInt("1234");
    BigInt big_7 = new BigInt("-123");
    BigInt big_8 = new BigInt("-0");
    BigInt big_9 = new BigInt("0");
    if ((big_1 > big_2) == true){
        Console.WriteLine("Not passed a test 3.1\n");
        return;
    }
```

```
if ((big_1 > big_3) == false)
{
   Console.WriteLine("Not passed a test 3.2\n");
   return;
}
if ((big_1 > big_4) == true)
{
   Console.WriteLine("Not passed a test 3.3\n");
   return;
}
if ((big_1 > big_5) == false)
{
   Console.WriteLine("Not passed a test 3.4\n");
   return;
}
if ((big_1 > big_6) == true)
{
   Console.WriteLine("Not passed a test 3.5\n");
   return;
}
if ((big_1 > big_7) == false)
{
   Console.WriteLine("Not passed a test 3.6\n");
   return;
}
if ((big_8 > big_9) == true)
{
   Console.WriteLine("Not passed a test 3.7\n");
   return;
}
```

```
}

{ // Test 4 < testing
   BigInt big_1 = new BigInt("123");
   BigInt big_2 = new BigInt("123");
   BigInt big_3 = new BigInt("12");
   BigInt big_4 = new BigInt("124");
   BigInt big_5 = new BigInt("126");
   BigInt big_6 = new BigInt("1234");
   BigInt big_7 = new BigInt("-123");
   BigInt big_8 = new BigInt("-0");
   BigInt big_9 = new BigInt("0");
   if (big_1 < big_2){
      Console.WriteLine("Not passed a test 4.1\n");
      return;
   }
   if ((big_1 < big_3) == true)
   {
      Console.WriteLine("Not passed a test 4.2\n");
      return;
   }
   if ((big_1 < big_4) == false)
   {
      Console.WriteLine("Not passed a test 4.3\n");
      return;
   }
   if ((big_1 < big_5) == false)
   {
      Console.WriteLine("Not passed a test 4.4\n");
      return;
```

```
            }
            if ((big_1 < big_6) == false)
            {
                Console.WriteLine("Not passed a test 4.5\n");
                return;
            }
            if ((big_1 < big_7) == true)
            {
                Console.WriteLine("Not passed a test 4.6\n");
                return;
            }
            if ((big_8 < big_9) == true)
            {
                Console.WriteLine("Not passed a test 4.7\n");
                return;
            }
        }
        {
            // Test 5 + testing.
            BigInt big_1 = new BigInt("100");
            BigInt big_2 = new BigInt("10");
            BigInt big_3 = new BigInt("-10");
            BigInt big_4 = new BigInt("-100");
            BigInt big_5 = new
BigInt("12345678912345678901234567890123456789012345678901234567890");
            BigInt big_6 = new BigInt("-300");

            if (big_1 + big_1 != new BigInt("200")){
                Console.WriteLine("Not passed a test 5.1\n");
                return;
```

```
            }
            if (big_1 + big_2 != new BigInt("110")){
                Console.WriteLine("Not passed a test 5.2\n");
                return;
            }
            if (big_1 + big_3 != new BigInt("90")){
                Console.WriteLine("Not passed a test 5.3\n");
                return;
            }
            if (big_1 + big_4 != new BigInt("0")){
                Console.WriteLine("Not passed a test 5.4\n");
                return;
            }
            if (big_2 + big_5 != new
BigInt("123456789123456789012345678901234567890123456790012345678901234567900")){
                Console.WriteLine("Not passed a test 5.5\n");
                return;
            }
            if (big_1 + big_6 != new BigInt("-200")){
                Console.WriteLine("Not passed a test 5.6\n");
                return;
            }
            if (big_6 + big_1 != new BigInt("-200")){
                Console.WriteLine("Not passed a test 5.7\n");
                return;
            }
        }
        // Test 5 - testing.
        {
            BigInt big_1 = new BigInt("100");
```

```
                BigInt big_2 = new BigInt("10");
                BigInt big_3 = new BigInt("-10");
                BigInt big_4 = new BigInt("-100");
                BigInt big_5 = new
BigInt("12345678912345678901234567890123456789012345678901234567890");
                BigInt big_6 = new BigInt("-300");

                if (big_1 - big_1 != new BigInt("0")){
                   Console.WriteLine("Not passed a test 6.1\n");
                   return;
                }
                if (big_1 - big_2 != new BigInt("90")){
                   Console.WriteLine("Not passed a test 6.2\n");
                   return;
                }
                if (big_1 - big_3 != new BigInt("110")){
                   Console.WriteLine("Not passed a test 6.3\n");
                   return;
                }
                if (big_1 - big_4 != new BigInt("200")){
                   Console.WriteLine("Not passed a test 6.4\n");
                   return;
                }
                if (big_2 - big_5 != new BigInt("-
12345678912345678901234567890123456789012345678901234567880")){
                   Console.WriteLine("Not passed a test 6.5\n");
                   return;
                }
                if (big_1 - big_6 != new BigInt("400")){
                   Console.WriteLine("Not passed a test 6.6\n");
```

```
                return;
            }
            if (big_6 - big_1 != new BigInt("-400")){
                Console.WriteLine("Not passed a test 6.7\n");
                return;
            }
        }
        // Test 7 * testing.
        {
            BigInt big_1 = new BigInt("100");
            BigInt big_2 = new BigInt("10");
            BigInt big_3 = new BigInt("-10");
            BigInt big_4 = new BigInt("-100");
            BigInt big_5 = new
BigInt("12345678912345678901234567890123456789012345678901234567890");
            BigInt big_6 = new BigInt("-300");

            if (big_1 * big_1 != new BigInt("10000")){
                Console.WriteLine("Not passed a test 7.1\n");
                return;
            }
            if (big_1 * big_2 != new BigInt("1000")){
                Console.WriteLine("Not passed a test 7.2\n");
                return;
            }
            if (big_1 * big_3 != new BigInt("-1000")){
                Console.WriteLine("Not passed a test 7.3\n");
                return;
            }
            if (big_1 * big_4 != new BigInt("-10000")){
```

```
                Console.WriteLine("Not passed a test 7.4\n");
                return;
            }
            if (big_2 * big_5 != new
BigInt("12345678912345678901234567890123456789012345678900")){
                Console.WriteLine("Not passed a test 7.5\n");
                return;
            }
            if (big_1 * big_6 != new BigInt("-30000")){
                Console.WriteLine("Not passed a test 7.6\n");
                return;
            }
            if (big_6 * big_1 != new BigInt("-30000")){
                Console.WriteLine("Not passed a test 7.7\n");
                return;
            }
        }
        // Test 8 / testing.
        {
            BigInt big_1 = new BigInt("100");
            BigInt big_2 = new BigInt("10");
            BigInt big_3 = new BigInt("-10");
            BigInt big_4 = new BigInt("-100");
            BigInt big_5 = new
BigInt("123456789123456789012345678901234567890123456789012345678901234567890");
            BigInt big_6 = new BigInt("-300");

            if (big_1 / big_1 != new BigInt("1")){
                Console.WriteLine("Not passed a test 8.1\n");
                return;
```

```
            }
            if (big_1 / big_2 != new BigInt("10")){
                Console.WriteLine("Not passed a test 8.2\n");
                return;
            }
            if (big_1 / big_3 != new BigInt("-10")){
                Console.WriteLine("Not passed a test 8.3\n");
                return;
            }
            if (big_1 / big_4 != new BigInt("-1")){
                Console.WriteLine("Not passed a test 8.4\n");
                return;
            }
            if (big_1 / big_5 != new BigInt("0")){
                Console.WriteLine("Not passed a test 8.5\n");
                return;
            }
            if (big_1 / big_6 != new BigInt("-0")){
                Console.WriteLine("Not passed a test 8.6\n");
                return;
            }
        }
        // Test 9 % testing.
        {
            BigInt big_1 = new BigInt("100");
            BigInt big_2 = new BigInt("10");
            BigInt big_3 = new BigInt("-10");
            BigInt big_4 = new BigInt("-100");
            BigInt big_5 = new
BigInt("12345678912345678901234567890123456789012345678901234567890");
```

```
    BigInt big_6 = new BigInt("-300");

    if (big_1 % big_1 != new BigInt("0")){
        Console.WriteLine("Not passed a test 9.1\n");
        return;
    }
    if (big_1 % big_2 != new BigInt("0")){
        Console.WriteLine("Not passed a test 9.2\n");
        return;
    }
    if (big_1 % big_3 != new BigInt("0")){
        Console.WriteLine("Not passed a test 9.3\n");
        return;
    }
    if (big_1 % big_4 != new BigInt("0")){
        Console.WriteLine("Not passed a test 9.4\n");
        return;
    }
    if (big_1 % big_5 != new BigInt("100")){
        Console.WriteLine("Not passed a test 9.5\n");
        return;
    }
    if (big_1 % big_6 != new BigInt("100")){
        Console.WriteLine("Not passed a test 9.6\n");
        return;
    }
}
{
    // Test 10 ModInverse testing.
    BigInt big_1 = new BigInt("200");
```

```
BigInt big_2 = new BigInt("51");
BigInt big_3 = new BigInt("9");
BigInt big_4 = new BigInt("11");
BigInt big_5 = new BigInt("88");
BigInt big_6 = new BigInt("15");

if (BigInt.ModInverse(big_1, big_2) != new BigInt("38")){
   Console.WriteLine("Not passed a test 10.1\n");
   return;
}
if (BigInt.ModInverse(big_1, big_3) != new BigInt("5")){
   Console.WriteLine("Not passed a test 10.2\n");
   return;
}
if (BigInt.ModInverse(big_1, big_4) != new BigInt("6")){
   Console.WriteLine("Not passed a test 10.3\n");
   return;
}
try{
if (BigInt.ModInverse(big_1, big_5) != new BigInt("2")){
   Console.WriteLine("Not passed a test 10.4\n");
   return;
}
}
catch (Exception){}
try{
if (BigInt.ModInverse(big_1, big_6) != new BigInt("2")){
   Console.WriteLine("Not passed a test 10.5\n");
   return;
}
```

```
            }
            catch (Exception){}
        }
        // Test 11 ExtendedGCD testing.
        {
            BigInt x = new BigInt("0");
            BigInt y = new BigInt("0");
            BigInt big_1 = new BigInt("200");
            BigInt big_2 = new BigInt("51");
            BigInt big_3 = new BigInt("90");

            if (BigInt.ExtendedGCD(big_1, big_2, out x, out y) != new
BigInt("1")){
                Console.WriteLine("Not passed a test 11.1\n");
                return;
            }
            if (x != new BigInt("-13") || y != new BigInt("51")){
                Console.WriteLine("Not passed a test 11.2\n");
                return;
            }
            if (BigInt.ExtendedGCD(big_1, big_3, out x, out y) != new
BigInt("10")){
                Console.WriteLine("Not passed a test 11.3\n");
                return;
            }
        }
        // Test 12 ModPow testing.
        {
            BigInt big_1 = new BigInt("2");
            BigInt big_2 = new BigInt("3");
```

```
                BigInt big_3 = new BigInt("4");
                BigInt big_4 = new BigInt("5");
                BigInt big_5 = new BigInt("6");
                if (BigInt.ModPow(big_1, big_2, big_3) != new BigInt("0")){ //
2^3 mod 4 = 0
                    Console.WriteLine("Not passed a test 12.1\n");
                    return;
                }
                if (BigInt.ModPow(big_1, big_4, big_2) != new BigInt("2")){ //
2^5 mod 3 = 0
                    Console.WriteLine("Not passed a test 12.2\n");
                    return;
                }
                if (BigInt.ModPow(big_1, big_5, big_5) != new BigInt("4")){ //
2^6 mod 6 = 4
                    Console.WriteLine("Not passed a test 12.3\n");
                    return;
                }
            }
        }
        }
}
```