**FINAL PROJECT**

CS012 – Discrete Structures 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | <insert ID>  <1.6”x1.6”> |  | <insert ID>  <1.6”x1.6”> |  | <insert ID>  <1.6”x1.6”> |
| **Florese,**  **August Bryan**  201710437  CSSE |  | **<LastName>,**  **<FirstName>**  <Student No>  <Program> |  | **<LastName>,**  **<FirstName>**  <Student No>  <Program> |  | **<LastName>,**  **<FirstName>**  <Student No>  <Program> |

Submitted to:

**Mr. Kim Howel D. delos Reyes**

Course Adviser

Table of Contents

[Project Specifications 2](#_Toc1852813)

[Team Members 3](#_Toc1852814)

[Introduction 4](#_Toc1852815)

[Objectives 4](#_Toc1852816)

[Discussion 4](#_Toc1852817)

[Source Code 5](#_Toc1852818)

[Conclusion 5](#_Toc1852819)

# Project Specifications

The class Natural represents the set of natural numbers. As defined, the set of natural numbers are non-negative whole numbers meaning, these are counting numbers starting from 1 up until to positive infinity including 0.

The class contains two constructors and a list of different methods. Below are the specifications on how the implementation for these parts of the class will be done.

**Constructors**

- Natural() : default constructor which sets the value of the Natural number to 0.

- Natural(BigInteger value) : constructor which sets the value of the Natural Number with the given BigInteger value, throwing RuntimeException if the value is less than 0.

**Methods/Functions**

- add(Natural value) : add the value of this instance by another instance

- subtract(Natural value) : subtract the value of this instance by another instance

- multiply(Natural value) : multiply the value of this instance by another instance

- divide(Natural value) : divide the value of this instance by another instance

- modulo(Natural value) : divide the value of this instance by another instance then return the remainder

- divideAndRemainder(Natural value) : divide the value of this instance by another instance then return the quotient and remainder as an array

- max(Natural value) : returns the maximum value between this instance and another instance

- min(Natural value) : returns the minimum value between this instance and another instance

- compareTo(Natural value) : return 1 if this instance value is greater than another instance value, 0 if this instance value is equal to another instance value, or -1 if this instance value is less than another instance value

- pow(BigInteger power) : returns the value of the expression thisvalue

- modPow(Natural exponent, Natural mod) : return the value of the expression thisvalue % mod

- nextProbablePrime() : returns the next prime number greater than the value of this instance

- gcd(Natural value) : returns the greatest common denominator between the value of this instance and the value of another instance

- lcm(Natural value) : returns the least common multiple between the value of this instance and the value of another instance

- isPrime() : checks if the value of this instance is a prime number, return true if yes, otherwise false

- toString() : returns a String representation of the value of this instance

- toString(Integer base) : returns a String representation of the value of this instance given a specific base value, where 2 ≤ base ≤ 62 and symbols are {0,...,9} U {A,...,Z} U {a,...,z}

- countRelativelyPrimes() : returns the number of relatively prime numbers with respect to the value of this instance, implement Euler's totient function

- isRelativelyPrimeTo(Natural value) : checks whether the value of this instance is relatively prime to the value of another instance

- divisionAlgorithm(Natural value) : returns a String value with the format "this.value = another.value \* quotient + remainder"

- primeFactorize() : returns a list of prime factors of the value for this instance

- distinctPrimeFactors() : returns a set of prime factors of the value for this instance

- getIntValue() : returns the integer representation of the value of this instance

- getLongValue() : returns the long representation of the value of this instance

# Team Members

FLORESE, AUGUST BRYAN, N.

A Computer Science student, favors the C# language above others due to experience, creating useful programs both for work and for fun.

Built an awful Unity-Engine game ridden with bugs, an M.U.D. framework, and more.

I’m interested in re-visiting these ideas, reconstructing them to their full potential once I have mastered the discipline.

LAST NAME, FIRST NAME, MI.

Describe yourself in exactly three sentences including, but not limited to, your hobbies and interests.

LAST NAME, FIRST NAME, MI.

Describe yourself in exactly three sentences including, but not limited to, your hobbies and interests.

# Introduction

Discrete mathematics is essential in studying the core concepts of programming especially for Computer Science students. Why? It will help you with the algorithms, complexity and computability. The understanding of set theory, probability, and combinations will allow you to analyze algorithms. You will be able to successfully identify parameters and limitations of your algorithms and can realize how complex a problem or a solution is.

As far as the programming language, discrete math doesn’t touch on how to actually program; but rather it can be used for software system design specification. In fact, applications like “ZED” can be used to designing a system using set theory.

The last important concept to grab out of discrete math is Boolean algebra. This is very useful not only in creating logical solution but is very useful in programming too. Software can be made or broken simply on the Boolean logic in it.

# Objectives

The objectives of this projects are the following:

* To be able to implement the behavior of set of natural numbers into a class containing different methods;
* To be able to provide the optimized solution for each method/function depending on the specifications provided; and
* To be able to understand the use of some of the concepts learned from Number Theories especially the natural numbers to real world situations.

# Discussion

The following sections will be observable throughout the document:

1. The Natural class
2. The Unit Test project
3. The test output (if any)

Using the Visual Studio Test Suite, Unit Testing each method is made simpler.

The project utilized the following import directives:

using System;

using System.Linq;

using System.Text;

using System.Numerics;

using Discrete\_Solution;

using System.Threading.Tasks;

using System.Collections.Generic;

using Microsoft.VisualStudio.TestTools.UnitTesting;

**Numerics**: Imports the BigInteger class. **TestTools**.**UnitTestng**: References the test suite.

**Linq**: Allows access to basic lambda query operations, a C# advantage.

φ **The Unit Test Class:**

[TestClass]

public class Discrete\_Test

{

[TestMethod, Description("Addition test must meet expected output")]

public void Arithmetic\_Test()

{

//method actions

}

}

We opted for the unit test method of testing due to the east in handling multiple methods and scenarios simultaneously. A screenshot will be included whether calculations meet expected values through the unit test. There is no console interface, however, we can still retrieve the output of the program. Throughout, you may notice the frequent usage of var to declare types, this is common practice if you come from a scripting language as var can practically mean any type. Java has no concept of this as it is a strongly typed language.

The type of var is dependent on the type of the object assigned to it, visible under intellisense.

For the sake of readability, Natural class methods are bulleted with the symbol **±**, while Discrete\_Test methods will start with **φ (phi).**

///<summary>

///Returns a String representation of the...

/// </summary>

/// <remarks>

/// Your base must be greater than 1 or less than 63...

/// The table of conversion are the numbers 0-9, A-Z and a-z (62 varia...

/// Could be replaced with a dictionary key value loo…

/// </remarks>

In-line documentation was omitted for the discussion section. Refer to the source code. We used XML in-line markup to auto-generate a documentation file in the backend. (Found in the debug folder)

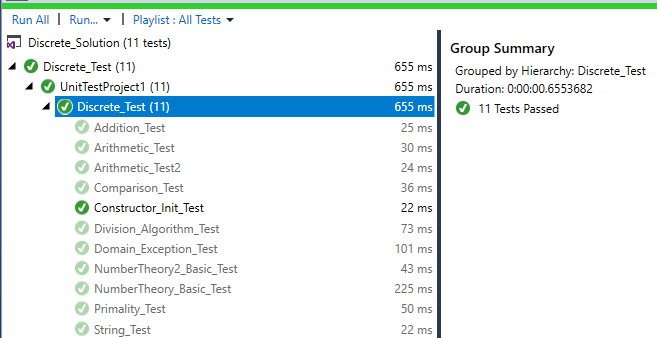
Unit Testing was formatted under the AAA guidelines as follows:

* The **Arrange** section of a unit test method initializes objects and sets the value of the data that is passed to the method under test.
* The **Act** section invokes the method under test with the arranged parameters.
* The **Assert** section verifies that the action of the method under test behaves as expected.

In order to save print material and design space, multiple act and assertion sections can be found per test method.

**Relevant Tests:**

Regarding these tests, many of these are checked against an initial Assertion that the calculated value is equal to a certain expected value. In the case of strings or list iteration, the compiler pass is sufficient.



± **The Natural Class:**

(Comments are extracted from this discussion, refer to the source code for in-line documentation)

Method Signatures are outlined in **Bold** to distinguish them.

**public class Natural**

{

private BigInteger value;

**public Natural()**

{

this.value = 0;

}

**public Natural(BigInteger value)**

{

try

{

if (value < 0)

{

throw new ArgumentException("Natural numbers cannot be negative");

}

this.value = value;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

throw;

}

}

}

The Natural class has a default constructor and a constructor taking one parameter of type BigInteger. Due to the nature of natural numbers, its private member value must contain numbers falling in the domain n_small.gif = {x | x >= 0}; throwing an exception otherwise.

± **public Natural Add(Natural value)**

As you might notice, in this method and mostly throughout the Natural class we return this, rather than a new instance of Natural. This was entirely out of personal preference and is quite common in the C# environment under the paradigm of Fluent Interfaces; This allows method chaining for the unit test later.

{

this.value += value.value;

return this;

}

± **public Natural Subtract(Natural value)**

Natural numbers cannot be negative, therefore we throw an ArgumentException. (I chose an ArgumentException over RuntimeException because it better describes the cause.)

{

try

{

if ((this.value - value.value) < 0)

throw new ArgumentException("Natural numbers cannot be negative");

else

this.value -= value.value;

return this;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

throw;

}

}

± **public Natural Multiply(Natural value)**

These methods are for the most part self-explanatory; for a more detailed discussion, see the more complex methods such as those involving number theory.

{

this.value \*= value.value;

return this;

}

± **public Natural Divide(Natural value)**

We throw an exception if at all any of the arguments passed are zero or less than zero.

{

try

{

if (this.value <= 0 || value.value <= 0)

throw new ArgumentException("You cannot divide by zero!");

else

{

this.value /= value.value;

return this;

}

}

catch (Exception e)

{

Console.WriteLine(e);

throw;

}

}

[TestMethod, Description("Expect exception thrown when initializing with negatives")]

[ExpectedException(typeof(ArgumentException))]

φ **public void Domain\_Exception\_Test()**

We test for negative number initialization. The method properly throws an exception, passing the test.

{

var negative\_natural = new Natural(-5);

}

[TestMethod, Description("Test for default initialization")]

φ **public void Constructor\_Init\_Test()**

Here we test for number initialization. The objects properly equate to zero, passing the test.

{

var zero\_natural = new Natural(0);

var default\_natural = new Natural();

var valid\_natural = new Natural(Int32.MaxValue);

Assert.AreEqual(zero\_natural.GetIntValue(), default\_natural.GetIntValue());

Assert.IsNotNull(default\_natural.GetIntValue());

Assert.AreNotSame(zero\_natural.GetIntValue(), valid\_natural.GetIntValue());

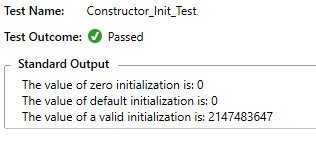
Assert.AreEqual(zero\_natural.GetIntValue(), valid\_natural.GetIntValue() - valid\_natural.GetIntValue());

Console.WriteLine(zero\_natural.GetIntValue());

Console.WriteLine(default\_natural.GetIntValue());

Console.WriteLine(valid\_natural.GetIntValue());

}



* **How to read the Unit Test output:**

Each line is based on the Console.WriteLine, look out for these statements.

Statements in each corresponding code block, these will usually grab the integer conversion of the object.

[TestMethod, Description("Addition test must meet expected output")]

φ **public void Addition\_Test()**

{

var natural1 = new Natural(50);

var natural2 = new Natural(50);

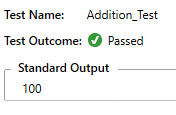
int expected = 100;

var actual = natural1.Add(natural2);

Assert.AreEqual(expected, actual.GetIntValue());

Console.WriteLine(actual.GetIntValue());

}



[TestMethod, Description("Test all arithmetic through complex method chaining!")]

φ **public void Arithmetic\_Test()**

Here we try stressing the consistency and limits of our function chain, such that despite its length, it would still return our expected values.

{

var natural1 = new Natural(10);

var natural2 = new Natural(2);

var natural3 = new Natural(7);

int expected = 50;

var actual = natural1.Add(natural2).Subtract(natural2).Multiply(natural1).Divide(natural2);

BigInteger expected2 = 20248916;

var actual2 = natural3.Pow(actual.GetBigValue()).Divide(new Natural(new Natural(5).Pow(50).GetBigValue()));

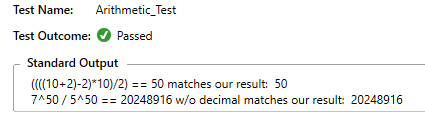
Assert.AreEqual(expected, actual.GetIntValue());

Assert.AreEqual(expected2, actual2.GetBigValue());

Console.WriteLine(actual.GetBigValue());

Console.WriteLine(actual2.GetBigValue());

}



Notable from our tests so far is that these operations are effectively confined; **natural1.Add(natural2)** is equivalent to writing **(10 + 2)**, completing the operations before moving to the next method in the queue. Our next test involves an array of Natural objects.

**More Natural Methods:**

± **public Natural Modulo(Natural value)**

{

try

{

if (value.value <= 0 || this.value <= 0)

throw new ArgumentException("You cannot divide by zero!");

this.value %= value.value;

return this;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

this.value = 0;

return this;

}

}

± **public Natural[] DivideAndRemainder(Natural value)**

Whenever division is involved, we must always check the values of the involved objects.

Here, we return an array of the Naturals, first index has the quotient, the second index has the remainder.

{

Natural[] divided = new Natural[2];

try

{

if (value.value <= 0 || this.value <= 0)

throw new ArgumentException("You cannot divide by zero!");

BigInteger quo = this.value / value.value;

BigInteger mod = this.value % value.value;

divided[0] = new Natural(quo);

divided[1] = new Natural(mod);

return divided;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

throw;

}

}

± **public Natural Pow(BigInteger power)**

{

BigInteger var = 1;

if (power > 0)

{

for (int i = 1; i <= power; ++i)

{

var \*= this.value;

}

}

else if (power < 0)

{

for (int i = -1; i >= power; --i)

{

var /= this.value;

}

}

this.value = var;

return this;

}

± **public Natural ModPow(Natural exponent, Natural mod)**

Since we already created a Power and Modulo method, we can use this in the composite method.

{

BigInteger exp = exponent.value, m = mod.value;

this.value = this.Pow(exp).Modulo(mod).value;

return this;

}

[TestMethod, Description("Test all arithmetic through complex method chaining!")]

φ **public void Arithmetic\_Test2()**

{

var natural1 = new Natural(10);

var natural2 = new Natural(6);

var natural3 = new Natural(3);

var natural4 = new Natural(10);

int expected = 2;

int expected2 = 12;

var natural\_array = natural1.DivideAndRemainder(natural2);

natural\_array[0].Add(natural3);

natural\_array[0].Add(natural2.Divide(natural3));

var actual = natural\_array[1].Gcd(natural\_array[0]);

Assert.AreEqual(expected, actual.GetIntValue());

Assert.AreEqual(2, natural\_array[1].GetIntValue());

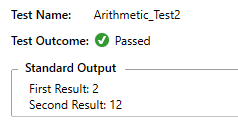
var actual2 = actual.ModPow(natural\_array[0], natural3.Add(natural4));

Assert.AreEqual(expected2, actual2.GetIntValue());

Console.WriteLine(actual.GetBigValue());

Console.WriteLine(actual2.GetBigValue());

}



**More Natural Methods:**

± **public Natural Max(Natural value)**

{

return this.value > value.value ? this : value;

}

± **public Natural Min(Natural value)**

{

return this.value < value.value ? this : value;

}

± **public int CompareTo(Natural value)**

{

return this.value == value.value ? 0 : this.value < value.value ? -1 : 1;

}

[TestMethod, Description("Various methods of comparison")]

φ **public void Comparison\_Test()**

{

var natural1 = new Natural(50);

var natural2 = new Natural(25);

int larger = 50, smaller = 25;

int expected = -1;

Natural actual;

actual = natural1.Max(natural2);

Assert.AreEqual(larger, actual.GetIntValue());

Console.WriteLine(actual.GetIntValue());

actual = natural1.Min(natural2);

Assert.AreEqual(smaller, actual.GetIntValue());

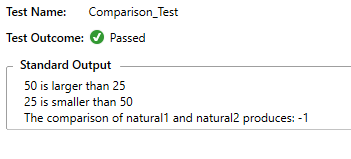
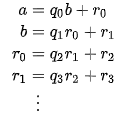
Console.WriteLine(actual.GetIntValue());

int output = actual.CompareTo(natural1);

Assert.AreEqual(expected, output);

Console.WriteLine(output);

}



± **public Natural Gcd(Natural value)**

Figure

First, we assign the value members of the objects to local variables.

To get the GCD, we apply a non-recursive implementation of Euclid’s algorithm that states that every common divisor of a and b is a common divisor of b and r; thus, computing the larger number first and the remainder, then exchanging it with the larger number until you reach zero, will result in the GCD. See Figure 1.

{

BigInteger x = this.value, y = value.value, temp = 0;

if (x <= 0 || y <= 0)

throw new ArgumentException("You cannot divide by zero!");

else if (y < x)

{

temp = x;

x = y;

y = temp;

}

while (y != 0)

{

temp = x % y;

x = y;

y = temp;

}

this.value = x;

return this;

}

± **public Natural Lcm(Natural value)**

The LCM two integers is the smallest integer that is a multiple of both numbers, we can get this and the GCD simultaneously as the GCD is the largest integer that divides both numbers. If we follow the division algorithm, we can get the missing multiple.

{

Natural temp = new Natural(this.value);

BigInteger z = temp.Gcd(value).value;

this.value = (this.value / z) \* value.value;

return this;

}

± **public Boolean IsPrime()**

First, we dispose of numbers less than 0, as these are not natural. Next, since the case for 1 and 2 are constant, we can return these immediately, otherwise, we check the most obvious divisibilities, if they equate to zero, then they are not prime. Finally, we run the numbers into our program, constantly checking increasing levels of divisibility. Normally you would make the square root of the number the limit of checking, however in this case, it is difficult to approximate BigInteger values through this method. Though it takes much longer, checking up to half the total should satisfy the requirement.

{

BigInteger num = this.value;

try

{

if (this.value < 0)

throw new ArgumentException("Number should be within the domain of natural numbers.");

if (this.value == 1)

return false;

if (this.value == 2 || this.value == 3)

return true;

if (this.value % 2 == 0 || this.value % 3 == 0)

return false;

int i;

for (i = 2; i <= num / 2; i++)

{

if (num % i == 0)

{

return false;

}

}

return true;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

return false;

}

}

± **public Boolean IsRelativelyPrimeTo(Natural value)**

All numbers are said to be relatively prime if their gcd is 1.

{

return (this.Gcd(value).value == 1) ? true : false;

}

[TestMethod, Description("Basic observations on Number Theory")]

φ **public void NumberTheory\_Basic\_Test()**

{

var natural = new Natural(9);

var relative\_natural = new Natural(38);

var valid\_natural = new Natural(36);

int relativelyprime = 1, gcd = 9, lcm = 342;

Natural actual;

actual = natural.Gcd(relative\_natural);

actual = relative\_natural.Gcd(natural);

Assert.AreEqual(relativelyprime, actual.GetIntValue());

Console.WriteLine("Relatively Prime (38,9): " + actual.GetIntValue());

natural = new Natural(9);

var IsRelativelyPrime = natural.IsRelativelyPrimeTo(relative\_natural);

Assert.AreEqual(true, IsRelativelyPrime);

Console.WriteLine("Relatively Prime (38,9): " + IsRelativelyPrime);

natural = new Natural(9);

var IsPrime = natural.IsPrime();

Console.WriteLine("Is 38 Prime?: " + IsPrime);

natural = new Natural(9);

actual = natural.Lcm(relative\_natural);

Assert.AreEqual(lcm, actual.GetIntValue());

Console.WriteLine("Valid LCM (38,9): " + actual.GetIntValue());

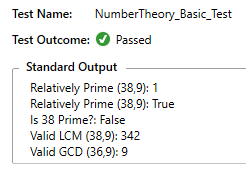
natural = new Natural(9);

actual = natural.Gcd(valid\_natural);

Assert.AreEqual(gcd, actual.GetIntValue());

Console.WriteLine("Valid GCD (36,9): " + actual.GetIntValue());

}



± **public override String ToString()**

In C#, everything is derived from Object, therefore, when creating similar methods it is important to either write new or override into the method signature to indicate that you want to use *this* implementation. The work of this method is pretty barebones but the override carries over to the overloaded method below that takes one parameter.

{

return Convert.ToString(this.value);

}

± **public String ToString(Integer num)**

We could use a dictionary map tinstead of iterating through a character array, but I believe that results in writing overall more lines than this solution. In C#, the maximum value of int64 is 9223372036854775807, which may be larger than the long of some other language (Int64 is equivalent to long). 2,147,483,647 is the maximum value of Int32. Note the Integer type, C# has no concept of boxed data types as the functions of these are already inherently usable on C# primitive types, I could simply write Int32 as the formal parameter, but to preserve the java similarities, an Integer class was included to mimic its usage.

{

try

{

if (num.Value <= 1 || num.Value >= 63)

throw new ArgumentException("Number should be greater than or equal to 2, and less than or equal to 62");

char[] map = new char[] { '0','1','2','3','4','5','6','7','8','9',

'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z',

'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x'};

string builder = string.Empty;

do

{

builder = map[(Int64)this.value % num.Value] + builder;

value = value / num.Value;

}

while (value > 0);

return builder;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

return null;

}

}

[TestMethod, Description("String map, number conversion to specified base")]

φ **public void String\_Test()**

{

var natural1 = new Natural(Int32.MaxValue);

var natural2 = new Natural(Int64.MaxValue);

string intmax = "2jg3E7";

string longmax = "Ca06U74ZPEa7";

Console.WriteLine(natural1.ToString());

Console.WriteLine(natural2.ToString());

var actual = natural1.ToString(60);

Assert.AreEqual(intmax, actual);

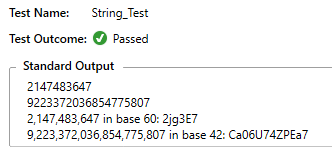
Console.WriteLine("2,147,483,647 in base 60: " + actual);

var actual2 = natural2.ToString(42);

Assert.AreEqual(longmax, actual2);

Console.WriteLine("9,223,372,036,854,775,807 in base 42: " + actual2);

}



± **public Int32 countRelativelyPrimes()**

Euler’s Totient noted with the greek letter phi (φ) is the value representing the number of integers less than n that are coprime with n. φ(n)= n ∏i=1(1−1/pi) where p is a distinct prime factor of n. Casting to double is required for precision, therefore some precision may be lost if this.value is strictly within the domain of BigIntegers.

{

Double n = (Double)this.value;

Double p = 1;

var distinct\_factors = this.DistinctPrimeFactors();

foreach(var element in distinct\_factors)

{

p \*= 1-(1/(Double)element.value);

}

n \*= p;

return (Int32)n;

}

± **public List<Natural> PrimeFactorize()**

The prime factorization method used is called the direct search method. It consists of searching for factors of a number by systematically performing trial divisions. In this case, we first check the divisibility by 2, and decompose these to their lowest forms. Whether or not they were initially divisible by 3 or odd numbers, they are divisible after the first loop. The same is performed with increasing prime divisibility checks; we then add these to a list.

{

var factors = new List<Natural>();

BigInteger value = this.value;

Double squared = Math.Sqrt((double)value);

while (value % 2 == 0)

{

factors.Add(new Natural(2));

value = value / 2;

}

for (int i = 3; i <= squared; i = i + 2)

{

while (value % i == 0)

{

factors.Add(new Natural(i));

value = value / i;

}

}

if (value > 2)

factors.Add(new Natural(value));

return factors;

}

± **public List<Natural> DistinctPrimeFactors()**

In this block we make use of a C# linq iteration; this allows us to perform query capable instructions, in this case, we’ve grouped each element such that it corresponds to a value and select the first ocurrence of these, creating a unique list.

{

var list = this.PrimeFactorize();

List<Natural> unique = new List<Natural>();

unique = list.GroupBy(elem => elem.value).Select(group => group.First()).ToList();

return unique;

}

[TestMethod, Description("Tests various cases of primality")]

φ **public void Primality\_Test()**

{

var natural1 = new Natural(Int32.MaxValue - 1);

var natural2 = new Natural(678);

var hardphi = new Natural(667);

var actual = natural1.PrimeFactorize();

Console.WriteLine("\nFactors of {0} are: ", natural1);

foreach (var element in actual)

{

Console.Write(element + " ");

}

var actual2 = natural2.PrimeFactorize();

Console.WriteLine("\nFactors of {0} are: ", natural2);

foreach (var element in actual2)

{

Console.Write(element + " ");

}

var distinctlist = natural1.DistinctPrimeFactors();

Console.WriteLine("\nDistinct Factors of {0} are: ", natural1);

foreach (var element in distinctlist)

{

Console.Write(element + " ");

}

var distinctlist2 = hardphi.DistinctPrimeFactors();

Console.WriteLine("\nDistinct Factors of {0} are: ", hardphi);

foreach (var element in distinctlist2)

{

Console.Write(element + " ");

}

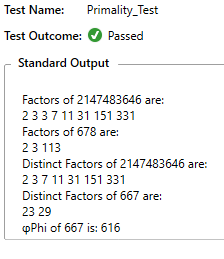
int expected = 616;

var phi = hardphi.countRelativelyPrimes();

Console.Write("\nφPhi of {0} is: " + phi, hardphi);

Assert.AreEqual(expected, phi);

}



± **public Natural NextProbablePrime()**

{

Natural n = new Natural(++this.value);

if (n.value < 0)

throw new ArgumentException("Number should be within the domain of natural numbers.");

bool PrimeCheck = n.IsPrime();

try

{

while (!PrimeCheck)

{

n.value++;

PrimeCheck = n.IsPrime();

if (PrimeCheck)

{

PrimeCheck = true;

return n;

}

}

return n;

}

catch (ArgumentException e)

{

Console.WriteLine(e);

return this;

}

}

[TestMethod, Description("Basic primality")]

φ **public void NumberTheory2\_Basic\_Test()**

For this unit test, we increment the value of n by 12 every iteration, then we select the next prime number strictly *after* the number given, we then perform an assertion check to see if this passes the test of primality.

{

int n = 19;

var natural = new Natural(n);

bool expected = true;

var actual = natural.NextProbablePrime();

for (int i = 1; i < 50; i+=6)

{

Console.WriteLine(n + i);

natural = new Natural(n + i);

actual = natural.NextProbablePrime();

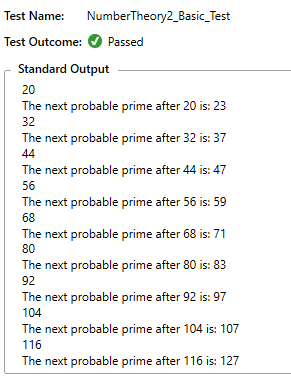
Assert.AreEqual(expected, actual.IsPrime());

Console.WriteLine("The next probable prime after {0} is: " + actual.GetIntValue(), n + i);

n += 6;

}

}



± **public String DivisionAlgorithm(Natural value)**

Due to the nature of division, some precision may be lost in calculating BigInteger values. A BigDecimal type may have been more optimal. As I coded this solution the nature of method chaining seems to lose its value when I have to implicitly create new instances this way to avoid overriding values, something notable.

{

Natural X = new Natural(this.value); Natural X2 = new Natural(this.value);

Natural Y = new Natural(value.value);

if(X.value <= 0 || Y.value <=0)

throw new ArgumentException("Invalid arguments detected!");

BigInteger q = X.Divide(value).value, r = X2.Modulo(Y).value;

return string.Format("{0} = {1} \* {2} + {3}", X.value, Y.value, q, r); }

[TestMethod, Description("Division Algorithm")]

φ **public void Division\_Algorithm\_Test()**

{

var natural = new Natural(158);

var natural2 = new Natural(17);

var actual = natural.DivisionAlgorithm(natural2);

Console.WriteLine("The Division Algorithm of {0} / {1} is: " + actual, natural, natural2);

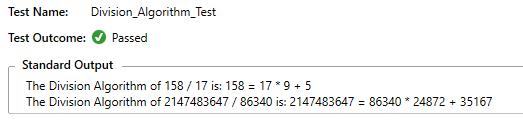
natural = new Natural(int.MaxValue);

natural2 = new Natural(86340);

actual = natural.DivisionAlgorithm(natural2);

Console.WriteLine("The Division Algorithm of {0} / {1} is: " + actual, natural, natural2);

}



# Source Code

# Conclusion

<Simply state whatever you have learned from doing the project. DELETE THIS PART AFTER EDIT.>