```
namespace _11set2023;
public enum Cards {Ace, Two, Three, Four, Five, Six, Seven, Jack, Queen, King} public enum Suits {Spades, Clubs, Diamonds, Hearts}
public interface IPlayingCard {
    Cards Value { get ; } Suits Suit { get ; }
    static bool operator <=(IPlayingCard first, IPlayingCard second) {
        if (first.Value < second.Value) return true;
        if (first.Value > second.Value) return false;
        if (first.Suit <= second.Suit) return true;
        return false;}
    static bool operator >=(IPlayingCard first, IPlayingCard second) { return second <= first; }
}
public class PlayingCard : IPlayingCard { public Cards Value { get; } public Suits Suit { get; } public PlayingCard(Cards value, Suits suit) { Value = value; Suit = suit; } }
public static class Extensions {
    public static IEnumerable<bool> FirstWins(this IEnumerable<IPlayingCard> deck) { // : IPlayingCard NON SERVE !!!
        // var count = 1; // IPlayingCard player1; // IPlayingCard player2; // IPlayingCard winnerCard; // int winneerIndex;
        // using (var enumerator = deck.GetEnumerator()) {
        //     if(enumerator.MoveNext()) player1 = enumerator.Current;
        //     else throw new ArgumentException();
        //     while (enumerator.MoveNext()) {
        //         player2 = enumerator.Current;
        //         if(player1 >= player2){
        //             winnerCard = player1;
        //             winneerIndex = count;
        //         }
        //         else{
        //             winnerCard = player2;
        //             winneerIndex = count;
        //         }
        //         if(count%6 == 0 && winneerIndex%2 == 0) yield return true;
        //         else yield return false;
        //         player1 = enumerator.Current;
        //         count++;
        //     }
        // }
        var currentHand = new List<IPlayingCard>();
        using (var enumerator = deck.GetEnumerator()) {
            while (enumerator.MoveNext()) {
                currentHand.Add(enumerator.Current);
                if (currentHand.Count%6 == 0) {
                    var maxFirst = currentHand[0];
                    var maxSecond = currentHand[1]; // inside so that it can be index 1 (that's effectevely some cards)
                    for(int i=0; i<6; i+=2)
                        if (currentHand[i] >= maxFirst)
                            maxFirst = currentHand[i];
                    for(int i=1; i<6; i+=2)
                        if (currentHand[i] >= maxSecond)
                            maxSecond = currentHand[i];
                    yield return maxFirst >= maxSecond;
                    currentHand.Clear();
                }
            }
        }
        if (currentHand.Count%6 != 0)
            throw new ArgumentException();

        // if (count%6 != 0)
        //     throw new ArgumentException();
    }
```

```
public class Tests
{
    [TestCase(new Cards[] { Cards.Ace, Cards.Two }, new Suits[] { Suits.Spades, Suits.Clubs })]
    [TestCase(new Cards[] { Cards.Ace, Cards.Two, Cards.Three }, new Suits[] { Suits.Spades, Suits.Clubs, Suits.Diamonds })]
    public void Test1(Cards[] cards, Suits[] suits)
    {
        if(cards.Length != suits.Length || cards.Length % 3 != 0)
            Assert.Inconclusive();

        var deck = GenerateSequence(cards, suits);
        var actual = deck.FirstWins();

        var expected = new Queue<bool>();
        for (int i = 0; i < cards.Length; i += 3)
            expected.Enqueue(true);

        Assert.That(actual, Is.EqualTo(expected)); // non c'è bisogno di fare ToList() perché l'EqualTo "usa un foreach"
    }
    private IEnumerable<IPlayingCard> GenerateSequence(Cards[] cards, Suits[] suits)
    {
        for (int i = 0; i < cards.Length; i++){
            yield return new PlayingCard(cards[i], suits[i]);
            yield return new PlayingCard(cards[i], suits[i]);
        }
    }

    [Test]
    public void Test2()
    {
        var source = InfiniteSequence();
        var actual = source.FirstWins().Take(1000)./*ToArray().Length;*/Count();
        var expected = 1000;

        // Assert.That(actual, Is.GreaterThanOrEqualTo(expected));
        Assert.That(actual, Is.EqualTo(expected));

    }
    IEnumerable<IPlayingCard> InfiniteSequence()
    {
        var random = new Random();
        while (true)
            yield return new PlayingCard((Cards)random.Next(0, 10), (Suits)random.Next(0, 4));
    }
    [Test]
    public void Test3()
    {
        var cards = new Cards[] { Cards.Ace, Cards.Two, Cards.Three, Cards.Four, Cards.Five, Cards.Six, Cards.Seven };
        var suits = new Suits[] { Suits.Spades, Suits.Clubs, Suits.Diamonds, Suits.Hearts, Suits.Spades, Suits.Clubs, Suits.Diamonds };
        var source = GenerateSequence(cards, suits); // OCCHIO!!! -> 14 invece che 7 :/
        // Assert.That(() => source.FirstWins().ToList(), Throws.ArgumentException);
        Assert.That(() => source.FirstWins().ToList(), Throws.TypeOf<ArgumentException>());
    }
}
```

```
public void Test1(){
    var s = new[] { 42.0, 49.0, 47.0, 18.0, 19.0, 28.0, 26.0 };
    var N = 2;
    Assert.That(() => s.Smooth(N).ToArray(), Throws.TypeOf<FiniteSourceException>());
}
[Test]
0 references
public void Test2(){
    var s = GenerateDoubles(0.0);
    var N = -1;
    // sollevata senza enumerare la sorgente neppure parzialmente perchè prima di enumerare faccio controlli su N etc
    Assert.That(() => s.Smooth(N).ToList(), Throws.TypeOf<ArgumentOutOfRangeException>());
}
1 reference
private static IEnumerable<double> GenerateDoubles(double d) { while (true) yield return d; }
[Test]
[TestCase(4, new[] { 0.0, 0.0, 0.0, 0.0 }, new[] { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 }, 8)]
// [TestCase(3, new[] { 1.0, 2.0, 3.0 }, new[] { 1.75, 1.8, 2, 1.8571428571428572, 1.875 }, 3)]
[TestCase(2, new[] { 1.0, 2.0, 3.0 }, new[] { 2, 1.75, 1.8, 2, 2 }, 5)]
// expected: 2 - 1,75 - 1,8 - 2 - 2 - 1,8 - 2 - 2 - 1,8 - 2 - 2 - 1,8 - 2 - 2 - 1,8 - 2 - 2
// actual:   2 - 1,75 - 1,8 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
0 references
public void Test3(int N, double[] sourceSample, double[] expectedSample, int howMany)
{ // la cerioli è malata
    var s = Repeat(sourceSample);
    var expected = NElementsPlusRepeat(expectedSample, N);
    var actual = s.Smooth(N);

    Console.WriteLine("Controllo i primi " + howMany + " elementi");
    // print first 20 elements of s
    Console.WriteLine(("source:   ") + string.Join(" - ", s.Take(20)));
    // print first 20 elements of expected
    Console.WriteLine(("\nexpected: ") + string.Join(" - ", expected.Take(20)));

    Console.WriteLine(("actual:   ") + string.Join(" - ", actual.Take(20)));

    Assert.That(actual.Take(howMany), Is.EqualTo(expected.Take(howMany)));
}
1 reference
private static IEnumerable<double> NElementsPlusRepeat(double[] d, int N){
    for (int i = 0; i < N; i++) yield return d[i];
    while (true) for(int i = N; i < d.Length; i++) yield return d[i];
}
1 reference
private static IEnumerable<double> Repeat(double[] d){
    while (true) foreach (var item in d) yield return item;
}
```

```
namespace _7lug2021;
0 references
public static class Extensions
{
    3 references
    public static IEnumerable<double> Smooth(this IEnumerable<double> s, int N) {
        if (s == null) throw new ArgumentNullException();
        if (N < 0) throw new ArgumentOutOfRangeException();
        // if (s is ICollection<double> c && c.Count == 0) throw new FiniteSourceException();
        var buffer = new Queue<double>();

        using (var enumerator = s.GetEnumerator()){
            var current_index = 0;

            while (enumerator.MoveNext()){
                buffer.Enqueue(enumerator.Current);

                if (current_index >= N && current_index < N*2+1)
                    yield return buffer.Average();

                if (current_index >= N*2+1){
                    yield return buffer.Average();
                    buffer.Dequeue();
                }
                current_index++;
            }
        }
        throw new FiniteSourceException();
    }
}
2 references
public class FiniteSourceException : Exception{}
```

```
[Test]
0 references
public void Test1() {
    IEnumerable<int>[] s = { new[] { 1 }, new[] { 1, 2 }, new[] { 1, 2, 3 } };
    // E' importante il take, perche' altrimenti l'IEnumerable non viene "caricato"
    Assert.That(() => s.Zip().Take(2).ToArray(), Throws.InstanceOf<ArgumentException>());
}
[Test]
0 references
public void Test2() {
    IEnumerable<int>[] s = { new[] { 1, 2, 3, 4 }, new[] { 10, 20, 30, 40 }, new[] { 100, 200, 300, 400 } };
    var actual = s.Zip();
    IEnumerable<int[]> expected = new[] { new[] { 1, 10, 100 }, new[] { 2, 20, 200 }, new[] { 3, 30, 300 }, new[] { 4, 40, 400 } };
    // Console.WriteLine("\nActual:"); foreach (var i in actual){
                                    //     foreach (var j in i)
                                    //         Console.Write(j + " ");
                                    //     Console.WriteLine();
                                    // }
    Assert.That(actual, Is.EqualTo(expected));
}
[Test]
[TestCase(45)]
0 references
public void Test3(int approx) {
    IEnumerable<T> Repeat<T>(T[] source) {
        while (true)
            foreach (var i in source)
                yield return i;
    }
    const int a = 1;
    const int b = 2;
    const int c = 3;
    IEnumerable<int>[] s = {
        Repeat(new [] {a, b, c}),
        Repeat(new [] {b, c, a}),
        Repeat(new [] {c, a, b})
    };
    var res = s.Zip().Take(approx);
    var expected = Repeat(new[] {
        new[] { a, b, c },
        new[] { b, c, a },
        new[] { c, a, b }
    }).Take(approx);
    Assert.That(res, Is.EqualTo(expected));
}
```

```
namespace _7feb2020;
public static class Extensions{
    public static IEnumerable<T[]> Zip<T>(this IEnumerable<T>[] s){
        if (s == null) throw new ArgumentNullException(nameof(s));
        var result = new List<T>();
        var enumerators = new IEnumerator<T>[s.Length];
        for(int i = 0; i < s.Length; i++){
            if (s[i] == null) throw new ArgumentNullException(nameof(s));
            enumerators[i] = s[i].GetEnumerator();}
        var rows = s.Length;
        while(enumerators.MoveNextAll()){
            for (var r = 0; r < rows; r++)
                result.Add(enumerators[r].Current); // result[c, r] = matrix[r, c];
            yield return result.ToArray();
            result.Clear();
        }
        for(int i = 0; i < s.Length; i++) //check if not equal lengths
            if (enumerators[i].MoveNext())
                throw new ArgumentException("Some sequences have different lengths", nameof(s));
    }
    public static bool MoveNextAll<T>(this IEnumerator<T>[] enumerators){ // method to move next all enumerators
        foreach (var enumerator in enumerators)
            if (!enumerator.MoveNext()) return false;
        return true;
    }
}
/*
if (s == null) throw new ArgumentNullException(nameof(s));
var enumerators = new IEnumerator<T>[s.Length];
for (int i = 0; i < s.Length; i++){
    if (s[i] == null) throw new ArgumentNullException(nameof(s));
    enumerators[i] = s[i].GetEnumerator();
}
for(int i = 0; i < s.Length; i++){
    using (var current_enumerator = enumerators[i]){
        while(current_enumerator.MoveNext()){
            var res = new List<T>();
            res.Add(current_enumerator.Current);
            for(int j = 0; j < s.Length; j++){
                if (j == i) continue;
                if (enumerators[j].MoveNext())
                    res.Add(enumerators[j].Current);
                else
                    throw new ArgumentException("Sequences have different lengths", nameof(s));
            }
            yield return res.ToArray();
```

```
namespace _8set2022;
public class ExtraMath
{ /**/
    public static IEnumerable<IEnumerable<T>> GeneralizedTartaglia<T>(T seed, Func<T,T,T> generator)
    {
        var dict = new Dictionary<int, List<T>>();
        var exceptions = new List<Exception>();
        var line = 0;
        while (true)
        {
            dict.Add(line, new List<T>());
            for (var i = 0; i <= line; i++)
            {
                if (i == line || i == 0)
                    dict[line].Add(seed);
                else
                {
                    var aux = dict[line - 1].ToArray();
                    try
                    {
                        dict[line].Add(generator(aux[i - 1], aux[i]));
                    }
                    catch (Exception ex)
                    {
                        exceptions.Add(ex);
                    }
                }
            }
            if (exceptions.Count > 0)
                throw new AggregateException("Multiple errors during row creation", exceptions);
            line++;
            yield return dict[line-1];
        }
    }
}
```

```
namespace _8set2022;
public class Tests
{
    public class MyException : Exception{
        private static int _count;
        public int Index { get; } = ++_count;
        public MyException() { }
        public MyException(string message) : base(message) { }
        public MyException(string message, Exception innerException) : base(message, innerException) { }
    }
    [Test]
    public void Test1(){
        int Generator(int a, int b) => a + b;
        var actual = new Dictionary<int, IEnumerable<int>>();
        var i = 0; const int seed = 1;
        foreach(var line in ExtraMath.GeneralizedTartaglia(seed, Generator).Take(5))
            actual.Add(i++, line);
        var expected = new Dictionary<int, IEnumerable<int>>() {
            { 0, new[] { 1 } },
            { 1, new[] { 1,1 } },
            { 2, new[] { 1,2,1 } },
            { 3, new[] { 1,3,3,1 } },
            { 4, new[] { 1,4,6,4,1 } },
        };
        Assert.That(actual, Is.EqualTo(expected));
    }
    [Test]
    public void Test2(){
        string Concat(string a, string b) => a.Length != 4 && b.Length != 4 ? a + b : throw new MyException("concat");
        try{
            var generator = Concat; var seed = "x";
            ExtraMath.GeneralizedTartaglia(seed, generator); // non metto var, tanto deve lanciare un'eccezione
        }
        catch (AggregateException aex){
            var indexList = new List<int>();
            foreach(MyException exception in aex.InnerExceptions)
                indexList.Add(exception.Index);
            Assert.That(indexList, Is.EqualTo(new[] {1,2,3,4}));
        }
    }
    [TestCase(5)]
    [TestCase(0)]
    public void Test3(int lineNumber){
        if (lineNumber <= 0) Assert.Inconclusive();
        int isCalled = 0;
        int Generator(int a, int b) => a + b; const int seed = 1;
        var triangle = ExtraMath.GeneralizedTartaglia(seed, Generator).Take(lineNumber);
        foreach(var line in triangle){
            var lenght = line.ToArray().Length; //nella prima e nella seconda riga non ci sono somme ma la seconda si annulla con il -2
            if (lenght != 1) isCalled += lenght - 2;
        }
        Assert.That(isCalled, Is.EqualTo(((lineNumber-1)*(lineNumber-2))/2)); }
}
```

```
namespace _16feb2023;
public static class Extensions
{
    public static bool EnoughSmaller<T>(this IEnumerable<T>? s, T threshold, int howMany) where T : IComparable
    {
        if (threshold == null || s == null)
            throw new ArgumentNullException();
        if (!(howMany > 0))
            throw new ArgumentOutOfRangeException();

        using (var it = s.GetEnumerator())
        {
            while (it.MoveNext())
            {
                if (it.Current.CompareTo(threshold) < 0)
                    howMany--;
                if (0 == howMany)
                    return true;
            }
            return false;
        }
    }
}
```

```
namespace _16feb2023;
public class Tests
{
    [Test]
    public void Test1()
    {
        var s = new[] { '0', '1', '2', '3', '4' };
        var threshold = '7';
        var howMany = 0;
        Assert.That(() => s.EnoughSmaller(threshold, howMany), Throws.TypeOf<ArgumentOutOfRangeException>());
    }
    [Test]
    public void Test2()
    {
        var s = new[] { "100", "200", "300", "400", "500" };
        var threshold = "700";
        var howMany = 42;
        Assert.That(s.EnoughSmaller(threshold, howMany), Is.False);
    }
    [TestCase(45)]
    public void Test3(int n)
    {
        if (n <= 0)
            Assert.Inconclusive("n is <= 0");

        var threshold = 7.42;
        int howMany = n;
        IEnumerable<double> GenS()
        {
            double x = -0.5;
            while (true)
            {
                yield return x - 1;
            }
        }
        var s = GenS();
        Assert.That(s.EnoughSmaller(threshold, howMany), Is.True);
    }
    [Test]
    public void Test4()
    {
        const int howMany = 7;
        var threshold = new MyClass(20);
        var s = new List<MyClass>();
        for (var i = 19; i >= 0; i--)
            s.Add(new MyClass(i));
        s.EnoughSmaller(threshold, howMany);
        Assert.That(MyClass.CallsNumber, Is.EqualTo(7));
```

```
public class MyClass : IComparable
{
    public static int CallsNumber { get; private set; }
    private readonly int _value;
    public MyClass(int value)
    {
        _value = value;
    }
    public override bool Equals(object obj)
    {
        if (obj == null || GetType() != obj.GetType())
            return false;
        MyClass other = (MyClass)obj;
        return _value == other._value;
    }
    public override int GetHashCode()
    {
        return _value.GetHashCode();
    }
    public int CompareTo(object? obj){
        CallsNumber += 1;
        var myObj = obj as MyClass;
        if (myObj == null)
            throw new ArgumentException();
        return _value.CompareTo(myObj._value);
    }
}
```

```
namespace _6lug2022;
public class Tests
{
    [Test]
    public void Test1()
    {
        var s = new[] {8, 11, 35};
        var multiplicity = 2;
        var forbidden = 112;
        var expected = new[] {8, 8, 11, 11, 35, 35};
        var actual = s.Repeat(multiplicity, forbidden);
        Assert.That(actual, Is.EqualTo(expected));
    }

    [Test]
    public void Test2()
    {
        var s = new[] {3.8, 24.31, 3.675};
        var multiplicity = 0;
        var forbidden = -4.67;
        Assert.That(s.Repeat(multiplicity, forbidden).Any, Throws.TypeOf<ArgumentOutOfRangeException>());
    }

    [TestCase(0, 0)]
    [TestCase(1, 0)]
    [TestCase(2, 0)]
    public void Test3(int position, int badGuy)
    {

        IEnumerable<int>GenerateSequence(int position, int badGuy)
        {
            if(position <= 0) Assert.Inconclusive();
            var r = new Random();
            var i = 0;
            while (true){
                var x = r.Next();
                if (i == position) yield return badGuy;
                else yield return x;
                i++;
            }
        }
        var s = GenerateSequence(position, badGuy);
        var multiplicity = 3;
        var forbidden = badGuy;
        if(position != 0)
            Assert.Throws<ArgumentException>(() => s.Repeat(multiplicity, forbidden).Take(3 * position).ToList());
    }
```

```
namespace _6lug2022;
public static class Extensions
{
    public static IEnumerable<T> Repeat<T>(this IEnumerable<T> s, int multiplicity, T forbidden) /*where T : struct*/{
        if(multiplicity <= 0) throw new ArgumentOutOfRangeException();
        if(s.Contains(forbidden)) throw new ArgumentException();

        foreach(var item in s){
            for(int i = 0; i < multiplicity; i++){
                yield return item;
            }
        }
    }
}
```

```
namespace _9giu2022;
public class Tests
{
    [Test]
    public void Test1()
    {
        var s1 = new[] {8, 11, 35};
        var s2 = new[] {100, 34, 23};
        var f = new Func<int, int>(x => x / 7);
        var expected = new[] {1, 14, 1, 4, 5, 3};
        var actual = s1.InterleavingApply(s2, f);
        Assert.That(actual, Is.EqualTo(expected));
    }

    [TestCase(new[] {'a', 'b', 'c'}, new[] {'1', '2', '3'})]
    [TestCase(new[] {'a', 'b', 'c'}, new[] {'1', '2', '3', '4'})]
    public void Test2(char[] a1, char[] a2)
    {
        if(a1.Length == a2.Length) Assert.Inconclusive("...");

        var f = new Func<char, bool>(x => char.IsDigit(x));
        Assert.That(() => a1.InterleavingApply(a2, f).Count(), Throws.TypeOf<DifferentLengthException>().With.Property("FirstIsLonger").EqualTo(a1.Length > a2.Length));
    }

    [Test]
    public void Test3()
    {
        var s1 = CycleSequence(new[] {7, 5, 17});
        var s2 = CycleSequence(new[] {2, 18, 42, 128, 512});
        var f = new Func<int, bool>(x => x % 2 == 0);
        var expected = CycleSequence(new[] {false, true}).Take(100).ToList();
        var actual = s1.InterleavingApply(s2, f).Take(100).ToList();
        Assert.That(actual, Is.EqualTo(expected));
    }
    IEnumerable<T> CycleSequence<T>(IEnumerable<T> source)
    {
        while (true)
        {
            foreach (var item in source)
            {
                yield return item;
            }
        }
    }
}
```

```
namespace _9giu2022;
[Serializable]
public class DifferentLengthException : Exception
{
    public bool FirstIsLonger { get; }
    public DifferentLengthException(bool firstIsLonger = true) { FirstIsLonger = firstIsLonger; }
    public DifferentLengthException(string message, bool firstIsLonger = true) : base(message) { FirstIsLonger = firstIsLonger; }
    public DifferentLengthException(string message, Exception inner, bool firstIsLonger = true) : base(message, inner) { FirstIsLonger = firstIsLonger; }
}
[Serializable]
public class FunctionApplicationException : Exception
{
    public FunctionApplicationException() { }
    public FunctionApplicationException(string message) : base(message) { }
    public FunctionApplicationException(string message, Exception inner) : base(message, inner) { }
}
public static class Extensions
{
    public static IEnumerable<TRes> InterleavingApply<T, TRes>(this IEnumerable<T> s1, IEnumerable<T> s2, Func<T, TRes> f)
    {
        using (var it1 = s1.GetEnumerator())
        using (var it2 = s2.GetEnumerator())
        {
            var firstHasNext = it1.MoveNext();
            var secondHasNext = it2.MoveNext();
            while (firstHasNext && secondHasNext) {
                TRes r1;
                TRes r2;

                try{
                    r1 = f(it1.Current);
                    r2 = f(it2.Current);
                }
                catch (Exception ex) {
                    throw new FunctionApplicationException(nameof(f), ex);
                }

                yield return r1;
                yield return r2;
                firstHasNext = it1.MoveNext();
                secondHasNext = it2.MoveNext();
            }

            if (firstHasNext && !secondHasNext)
                throw new DifferentLengthException(true);
            if (!firstHasNext && secondHasNext)
                throw new DifferentLengthException(false);
        }
```

```
namespace _4feb2022;
public class Tests
{
    [Test]
    public void Test1()
    {
        var source = new[] {"45", "-8gg", "lll lo sembra ma non lo e'"};
        Assert.That(() => source.GetContainedNumbers().Count(), Throws.TypeOf<ArgumentException>());
        // .Count() on a collection iterates over the elements of the it and counts them. The method returns the total count as an integer value.
    }

    [Test]
    public void Test2()
    {
        var source = new[] {"f55h7", "90", "-45", "H 6YY5"};
        var expected = new[] {557, 90, 45, 65};
        var actual = source.GetContainedNumbers();
        Assert.That(actual, Is.EqualTo(expected));
    }

    [Test]
    public void Test3()
    {
        var source = GenerateSequence();
        var expected = Enumerable.Range(0, 100);
        var actual = source.GetContainedNumbers().Take(100);
        Assert.That(actual, Is.EqualTo(expected));
    }
    private static IEnumerable<string> GenerateSequence()
    {
        var i = 0;
        while (true)
        {
            yield return $"a{i++}z";
        }
    }
}
```

```
namespace _4feb2022;
public static class Extensions
{
    public static IEnumerable<int> GetContainedNumbers(this IEnumerable<string> source)
    {
        if(source == null) throw new ArgumentNullException();
        foreach(var s in source){
            if(s == null) throw new ArgumentException();
            var number = "";
            foreach(var c in s){
                if(char.IsDigit(c)) number += c;
            }
            if(number == "") throw new ArgumentException();
            yield return int.Parse(number);
        }
    }
}
```

.

```
using Moq;
namespace _18feb2022;
public class Tests
{
    [Test]
    public void Test1() {
        var m = new Mock<IAppointment>();
        m.Setup(x => x.PatientName).Returns("giorgio");
        m.Setup(x => x.ProposedTimes).Returns(new[] { new DateTime(2022, 5, 3).AddHours(11) });
        var apps = new List<IAppointment> { m.Object };
        var ms = new MedicalScheduler();
        Assert.That(ms.Schedule(apps), Is.EqualTo(new[] { new Tuple<string, bool>("giorgio", false) }));
    }
    [Test]
    public void Test2()
    {
        var m1 = new Mock<IAppointment>();
        m1.Setup(x => x.PatientName).Returns("ada");
        m1.Setup(x => x.ProposedTimes).Returns(new[] { new DateTime(2022, 7, 2).AddHours(18), new DateTime(2022, 4, 16).AddHours(10) });
        var ms = new MedicalScheduler(); // Create an instance of MedicalScheduler
        ms.FreeSlots = new List<DateTime> { new DateTime(2022, 8, 2).AddHours(8), new DateTime(2022, 4, 16).AddHours(11),
        new DateTime(2022, 4, 16).AddHours(10), new DateTime(2022, 5, 3).AddHours(10) };
        var apps = new List<IAppointment> { m1.Object };
        var expected = new DateTime(2022, 4, 16).AddHours(10);
        ms.Schedule(apps);
        m1.VerifySet(x => x.SelectedAppointmentTime = expected);
    }
    [Test]
    public void Test3() {
        var m1 = new Mock<IAppointment>();
        m1.Setup(x => x.PatientName).Returns("ugo");
        m1.Setup(x => x.ProposedTimes).Returns(new[] { new DateTime(2022, 3, 12).AddHours(15), new DateTime(2022, 3, 13).AddHours(15) });
        var m2 = new Mock<MedicalScheduler>();
        m2.Setup(x => x.FreeSlots).Returns(new List<DateTime> { new DateTime(2022, 8, 20).AddHours(18), new DateTime(2022, 6, 6).AddHours(16),
            new DateTime(2022, 3, 13).AddHours(15), new DateTime(2022, 3, 3).AddHours(15) });
        m2.Setup(x => x.Appointments).Returns(new Dictionary<DateTime, string>
        { { new DateTime(2022, 8, 13).AddHours(10), "franco" }, { new DateTime(2022, 6, 6).AddHours(15), "luca" }, { new DateTime(2022, 6, 6).AddHours(17), "paola" }, });
        var apps = new List<IAppointment> { m1.Object };
        var actual = m2.Object.Schedule(apps);
        m1.VerifySet(x => x.SelectedAppointmentTime = new DateTime(2022, 3, 13).AddHours(15));
        Assert.Multiple(() => {
            Assert.That(actual, Is.EqualTo(new[] { new Tuple<string, bool>("ugo", true) }));
            Assert.That(m2.Object.FreeSlots, Is.EqualTo(new List<DateTime>
            { new DateTime(2022, 8, 20).AddHours(18), new DateTime(2022, 6, 6).AddHours(16), new DateTime(2022, 3, 3).AddHours(15) }));
            Assert.That(m2.Object.Appointments, Is.EqualTo(new Dictionary<DateTime, string>
            { {new DateTime(2022, 8, 13).AddHours(10), "franco"}, {new DateTime(2022, 6, 6).AddHours(15), "luca"}, {new DateTime(2022, 6, 6).AddHours(17), "paola"},
            {new DateTime(2022, 3, 13).AddHours(15), "ugo"}, }));
        });
```

```
namespace _18feb2022;
public interface IAppointment
{
    public string PatientName { get; }
    public IEnumerable<DateTime> ProposedTimes { get; }
    public DateTime? SelectedAppointmentTime { get; set; }
}
public class MedicalScheduler
{
    public virtual Dictionary<DateTime, string> Appointments { get; } = new();
    public virtual List<DateTime> FreeSlots { get; set; } = new();
    public IEnumerable<Tuple<string, bool>> Schedule(IEnumerable<IAppointment> requests)
    {
        var res = new List<Tuple<string, bool>>();
        foreach (var r in requests) {
            foreach (var t in r.ProposedTimes) {
                if (FreeSlots.Contains(t)) {
                    FreeSlots.Remove(t);
                    r.SelectedAppointmentTime = t;
                    Appointments.Add(t, r.PatientName);
                    res.Add(new Tuple<string, bool>(r.PatientName, true));
                }
            }
            if (!res.Contains(new Tuple<string, bool>(r.PatientName, true)))
                res.Add(new Tuple<string, bool>(r.PatientName, false));
        }
        return res;
    }
}
```

```
using ArgumentException = System.ArgumentException;

namespace _7feb2020;
public static class Extension {
    public static IEnumerable<T[]> Zip<T>(this IEnumerable<T>[]? s) {

        if (s == null) throw new ArgumentNullException();
        for (int i = 0; i < s.Length; i++)
            if (s[i] == null) throw new ArgumentNullException();

        var size = s.Length;
        var sEnumerator = new IEnumerator<T>[size];
        for (int i = 0; i < size; i++) sEnumerator[i] = s[i].GetEnumerator();

// elemOfSequence itera per ogni seq di s (perchè potrebbe essere inf); s[0] perchè siccome righecolonne colonnerighe lo posso fare solo se s quadrata, allora qualsiasi seq di s + ok
        foreach (var elemOfSequence in s[0]) {
            var item = new T[size];
            for (int i = 0; i < size; i++) {
                if (!sEnumerator[i].MoveNext())
                    throw new ArgumentException($"element in position {i + 1} is shorter than element in first position");
                item[i] = sEnumerator[i].Current;
            }
            yield return item;
        }

        for (int i = 0; i < size; i++)
            if (sEnumerator[i].MoveNext())
                throw new ArgumentException($"element in position {i + 1} is longer than element in first position");
    }
}
```

```
namespace _7feb2020;
public class Tests {
    [Test]
    public void Test1() {
        IEnumerable<int>[] s = { new[] { 1 }, new[] { 2, 3 }, new[] { 6, 78, 8 } };
        Assert.That(() => s.Zip().Take(2).ToArray(), Throws.TypeOf<ArgumentException>());
    }

    [Test]
    public void Test2() {

        IEnumerable<int>[] s = { new[] { 1, 2, 3, 4 }, new[] { 10, 20, 30, 40 }, new[] { 100, 200, 300, 400 } };
        var expected = new[] { (IEnumerable<int>)new[] { 1, 10, 100 }, new[] { 2, 20, 200 }, new[] { 3, 30, 300 }, new[] { 4, 40, 400 } };

        Assert.That(s.Zip(), Is.EqualTo(expected));
    }

    [TestCase(43)]
    public void Test3(int approx) {

        IEnumerable<T> Repeat<T>(T[] source) {
            while (true)
                foreach (var e in source)
                    yield return e;
        }

        var a = "pippo"; var b = "qui"; var c = "quo";
        var abc = new[] { a, b, c };
        var bca = new[] { b, c, a };
        var cab = new[] { c, a, b };
        var s = new[] { Repeat(abc), Repeat(bca), Repeat(cab) };
        var actual = s.Zip().Take(approx);
        // var expected = new List<string[]>();
        //
        // for (int i = 0; i < approx / 3; i++) {
        //     expected.Add(abc);
        //     expected.Add(bca);
        //     expected.Add(cab);
        // }
        //
        // if (approx % 3 > 0) expected.Add(abc);
        // if (approx % 3 > 1) expected.Add(bca);

        /*alternative:*/ var  expected = Repeat(new[] { abc, bca, cab }).Take(approx);

        Assert.That(actual, Is.EqualTo(expected));
    }
}
```

```
namespace _7lug2021;
public static class Extensions
{
    public static IEnumerable<double> Smooth(this IEnumerable<double> s, int N) {
            if (s == null)
                throw new ArgumentNullException("s");
            if (N < 0)
                throw new ArgumentOutOfRangeException("N");
            // if (s is ICollection<double> c) throw new FiniteSourceException();
            var buffer = new List<double>();
            using (var enumerator = s.GetEnumerator()) {
                var index = 0;
                while (enumerator.MoveNext()) {
                    buffer.Add(enumerator.Current);
                    if (buffer.Count >= (N * 2) + 1) {  // We can start smoothing
                        var avg = 0.0;
                        for (var i = index - N; i < index + N; i++)
                            avg += buffer[i];
                        avg /= (N * 2 + 1);
                        buffer.Add(avg);
                    }
                    yield return buffer[index];
                    index++;
                }
            }
            throw new FiniteSourceException();
    }
    // var buffer = new Queue<double>();
    // buffer.Enqueue(enumerator.Current);
    // if (buffer.Count < N + 1)
    //     continue; // non ho ancora abbastanza elementi per calcolare la media
    // yield return buffer.Average();
    // buffer.Dequeue();
}
public class FiniteSourceException : Exception{}
```

```
namespace _7lug2021;
public class Tests
{
    [Test]
    public void Test1()
    {
        var s = new[] { 42.0, 49.0, 47.0, 18.0, 19.0, 28.0, 26.0 };
        var N = 2;
        Assert.Throws<FiniteSourceException>(() => s.Smooth(N).ToList());
    }
    [Test]
    public void Test2()
    {
        var s = GenerateDoubles(0.0);
        var N = -1;
        // sollevata senza enumerare la sorgente neppure parzialmente perchè prima di enumerare faccio controlli su N etc
        Assert.That(() => s.Smooth(N).ToList(), Throws.TypeOf<ArgumentOutOfRangeException>());
    }
    private static IEnumerable<double> GenerateDoubles(double d) { while (true) yield return d; }

    [TestCase(3, new[] { 1.0, 2.0, 3.0 }, new[] { 1.0, 2.0, 3.0, 2.0, 3.0, 4.0 }, 3)]
    [TestCase(2, new[] { 1.0, 2.0, 3.0 }, new[] { 1.0, 2.0, 3.0, 2.0, 3.0 }, 2)]
    [TestCase(1, new[] { 1.0, 2.0, 3.0 }, new[] { 1.0, 2.0, 3.0, 2.0 }, 1)]
    public void Test3(int N, double[] sourceSample, double[] expectedSample, int howMany)
    {
        var s = GenerateDoublesArray(sourceSample).Smooth(N);
        var e = expectedSample.Smooth(N);
        Assert.That(s.Take(howMany), Is.EqualTo(e.Take(howMany)));
    }
    private static IEnumerable<double> GenerateDoublesArray(double[] d)
    {
        while (true)
            foreach (var item in d)
                yield return item;
    }
}
```

```
using Moq; using Range = System.Range; namespace _13gen2020;
public class Tests{
    public enum Day { Mo, Tu, We, Th, Fr } public enum Colors { Bianco, Grigio, Nero }
    public class C<T> : I<T> {
        private I<T> _fieldImplementation;
        public T Value { get; set; }
        public C(T value) { Value = value; }
        public override bool Equals(object? obj) { return obj is C<T> elem && Value.Equals(elem.Value); }
        public override int GetHashCode() { return Value.GetHashCode(); }
        public T P => _fieldImplementation.P; }
    [Test]
    public void Test1() {
        var e0 = new Mock<I<Day>>(); e0.Setup(x => x.P).Returns(Day.Mo);
        var e1 = new Mock<I<Day>>(); e1.Setup(x => x.P).Returns(Day.Mo);
        var e3 = new Mock<I<Day>>(); e3.Setup(x => x.P).Returns(Day.Mo);
        var e4 = new Mock<I<Day>>(); e4.Setup(x => x.P).Returns(Day.Mo);
        var arr = new[] { e0.Object, e1.Object, null, e3.Object, e4.Object };
        Assert.That(() => arr.Indexing(), Throws.TypeOf<ArgumentNullException>()); }
    [Test]
    public void Test2() {
        var e0 = new Mock<I<Day>>();
        e0.Setup(x => x.P).Returns(Day.Mo);
        var e1 = new Mock<I<Day>>();
        e1.Setup(x => x.P).Returns(Day.Mo);
        var e2 = new Mock<I<Day>>();
        e2.Setup(x => x.P).Returns(Day.We);
        var e3 = new Mock<I<Day>>();
        e3.Setup(x => x.P).Returns(Day.Mo);
        var e4 = new Mock<I<Day>>();
        e4.Setup(x => x.P).Returns(Day.Fr);
        var e5 = new Mock<I<Day>>();
        e5.Setup(x => x.P).Returns(Day.We);
        var arr = new[] { e0.Object, e1.Object, e2.Object, e3.Object, e4.Object, e5.Object };
        var result = arr.Indexing();
        Assert.Multiple(() =>{ Assert.That(result[Day.Mo], Is.EqualTo(new[] { e0.Object, e1.Object, e3.Object }));
            Assert.That(result[Day.Tu], Is.Empty);
            Assert.That(result[Day.We], Is.EqualTo(new[] { e2.Object, e5.Object }));
            Assert.That(result[Day.Th], Is.Empty);
            Assert.That(result[Day.Fr], Is.EqualTo(new[] { e4.Object })); }); }
    [TestCase(9)]
    public void Test3(int howMany) {
        var arr = new I<Colors>[3 * howMany];
        var e0 = new Mock<I<Colors>>();
        e0.Setup(x => x.P).Returns(Colors.Bianco);
        var e1 = new Mock<I<Colors>>();
        e1.Setup(x => x.P).Returns(Colors.Grigio);
        var e2 = new Mock<I<Colors>>();
        e2.Setup(x => x.P).Returns(Colors.Nero);
        for (int i = 0; i < arr.Length; i += 3){
            arr[i] = e0.Object;
            arr[i + 1] = e1.Object;
            arr[i + 2] = e2.Object;
        }
        var result = arr.Indexing();
        Assert.Multiple(() => { Assert.That(result[Colors.Bianco].Take(Range.All).Count(x => x.P == Colors.Bianco), Is.EqualTo(howMany));
            Assert.That(result[Colors.Grigio].Take(Range.All).Count(x => x.P == Colors.Grigio), Is.EqualTo(howMany));
            Assert.That(result[Colors.Nero].Take(Range.All).Count(x => x.P == Colors.Nero), Is.EqualTo(howMany)); });
    }
```

```
namespace _13gen2020;
public interface I<T> { T P { get; } }
public static class Extensions
{
    public static Dictionary<T, I<T>[]> Indexing<T>(this I<T>?[]? s) where T : Enum {
        if (s == null)
            throw new ArgumentNullException();

        var elements = Enum.GetValues(typeof(T));
        var myDict = new Dictionary<T, I<T>[]>();
        foreach (var element in elements)
            myDict.Add((T)element, new I<T>[] { });

        foreach (var ss in s) {
            if (ss == null)
                throw new ArgumentNullException();
            myDict[ss.P] = myDict[ss.P].Append(ss).ToArray();
        }
        return myDict;
    }
}
```

```
namespace _9set2021;
public class Tests
{
    [Test]
    public void Test1()
    {
        var leftSeq = new[] { "bianco", "rosso", "verde" };
        var rightSeq = GenerateStrings("pippo");
        Assert.That(() => leftSeq.MinUpToNow(rightSeq).Take(5).ToList(), Throws.TypeOf<ArgumentException>());
    }

    [Test]
    public void Test2()
    {
        var leftSeq = new[] { "qui", "quo", "qua", "paperino", "paperone" };
        var rightSeq = new[] { "topolino", "pippo", "pluto", "tip", "tap" };
        // var expected = new[] { "qui", "pippo", "pippo", "paperino", "paperino" }; >:(
        var expected = new[] { "qui", "pippo", "pluto", "paperino", "paperone" };
        var actual = leftSeq.MinUpToNow(rightSeq);
        Assert.That(actual, Is.EquivalentTo(expected));
    }

    [Test]
    public void Test3([Range(0, 1000, 100)] int errorIndex)
    {
        var leftSeq = GenerateStrings("rosa");
        var rightSeq = GenerateStrings("viola", errorIndex);
        // rightSeq[errorIndex] = null;
        Assert.Throws<ArgumentNullException>(() => leftSeq.MinUpToNow(rightSeq).ToList()); // se mi aspetto un'eccezione, devo usare lambda e ToList() SE MI ASPETTO UNA LISTA
    }
    // generate infinite sequences of strings
    private static IEnumerable<string?> GenerateStrings(string? s, int? errorIndex = null) // se errorIndex non c'è, viene messo a null
    {
        var i = 0;
        while (true)
        {
            if (i++ == errorIndex && errorIndex != null)
                yield return null;
            yield return s;
        }
    }
}
```

```
namespace _9set2021;
public static class Extensions
{
    public static IEnumerable<T> MinUpToNow<T>(this IEnumerable<T> leftSeq, IEnumerable<T> rightSeq) where T : IComparable<T>
    {
        if (leftSeq == null || rightSeq == null)
            throw new ArgumentNullException();

        var leftEnumerator = leftSeq.GetEnumerator();
        var rightEnumerator = rightSeq.GetEnumerator();
        var leftHasNext = leftEnumerator.MoveNext();
        var rightHasNext = rightEnumerator.MoveNext();

        while (leftHasNext && rightHasNext)
        {
            if (leftEnumerator.Current == null || rightEnumerator.Current == null)
                throw new ArgumentNullException();

            if (leftEnumerator.Current.CompareTo(rightEnumerator.Current) <= 0)
                yield return leftEnumerator.Current;
            else
                yield return rightEnumerator.Current;

            leftHasNext = leftEnumerator.MoveNext();
            rightHasNext = rightEnumerator.MoveNext();
        }

        if (leftHasNext || rightHasNext)
            throw new ArgumentException("Sequences have unequal lengths.");
    }
}
```

```
namespace _30gen2023
{
    public interface IIdentified { public int Key { get; } }

    public class Impl : IIdentified
    {

        private int _key;
        public int Key
        {
            get
            {
                Count++;
                return _key;
            }
            set { _key = value; } // value=valore di sist. che gli do mentre lo sto settando :)
        }

        public int Count { get; set; } = 0;

        public Impl(int Key)
        {
            this.Key = Key;
            this.Count = 0;
        }
    }

    public static class Extension
    {
        public static int? Lookup<T>(this IEnumerable<T?>? db, int what) where T : IIdentified
        {
            if (db == null) throw new ArgumentNullException(nameof(db));
            var pos = 0;
            foreach(var elem in db)
            {
                if(elem == null) throw new ArgumentNullException(nameof(elem));
                if (elem.Key == what)
                    return pos;
                pos++;
            }
            return null;
        }
    }
}
```

```
using Moq;
namespace _30gen2023;
public class Tests{
    [Test]
    public void Test1(){
        var l = new List<IIdentified>();
        foreach(var elem in new[] { 8, -70, 5, 7, 5 }){
            var m = new Mock<IIdentified>();
            m.Setup(x => x.Key).Returns(elem);
            l.Add(m.Object);
        }
        Assert.Multiple(() => { Assert.That(l.Lookup(8), Is.EqualTo(0)); Assert.That(l.Lookup(5), Is.EqualTo(2)); Assert.That(l.Lookup(11), Is.Null); });
    }
    [Test]
    public void Test2(){
        var r = new Random();
        int next;
        IEnumerable<int?> Infinite(){
            for (var i = 0; i < 42; i++){
                do{ next = r.Next();
                } while (next == 42);
                yield return next;
            }
            yield return null;
            while (true) { yield return r.Next();}
        }
        var l = new List<IIdentified?>();
        foreach(var item in Infinite().Take(50).ToArray()){
            if (item == null) l.Add(null);
            else{
                var m = new Mock<IIdentified>();
                m.Setup(x => x.Key).Returns((int)item);
                l.Add(m.Object);
            }
        }
        Assert.That(() => l.Lookup(42), Throws.TypeOf<ArgumentNullException>());
    }
    [TestCase(20)]
    public void Test3(int size){
        if (size < 20) Assert.Inconclusive(nameof(size));
        var db = new Impl[size];
        for (var i = 0; i < size; i++)
            db[i] = new Impl(i);

        var what = 10;
        var actual = db.Lookup(what);
        Assert.Multiple(() =>{ for (int i = 0; i <= what; i++) Assert.That(db[i].Count, Is.EqualTo(1));
                               for (int i = what+1; i < size; i++) Assert.That(db[i].Count, Is.EqualTo(0)); });
```

```
namespace _5giu2023;
public class InconsistentSourceException : Exception{
    public InconsistentSourceException() {}
    public InconsistentSourceException(string message) : base (message) {}
    public InconsistentSourceException(string message, Exception inner) : base (message, inner) {}
}
public static class Extentions
{
    public static IEnumerable<int[]> MultipleApply<T>(this IEnumerable<Func<T, int>> s, T v, int n) // <T> needs to be specified in order to use the extension method syntax
    {
        if (s == null) throw new ArgumentNullException("s");
        if (n <= 0) throw new ArgumentOutOfRangeException("n");

        return  MultipleApplyHelper(s, v, n);
    }

    private static IEnumerable<int[]> MultipleApplyHelper<T>(IEnumerable<Func<T, int>> s, T v, int n)
    {
        using (var it = s.GetEnumerator())
        {
            var array = new int[n];
            var i = 0;
            while (it.MoveNext())
            {
                array[i++] = it.Current(v);
                if (i == n){
                    yield return array;
                    array = new int[n]; // reset
                    i = 0;
                }
            } // end while = finite source

            if (s.Count() % n != 0)
                throw new InconsistentSourceException();
        }
    }
}
```

```
namespace _5giu2023;
public class Tests
{
    [Test]
    public void Test1()
    {
        var v = 2;
        var n = 3;
        var s = new Func<int, int>[] { x => 2 * x, x => 3 * x, x => 4 * x, x => 5 * x, x => 6 * x, x => 7 * x };
        var expected = new[] { new[] {4, 6, 8}, new[] {10, 12, 14}};
        var actual = s.MultipleApply(v, n);
        Assert.That(actual, Is.EqualTo(expected));
    }

    [Test]
    public void Test2()
    {
        var v = "boom";
        var n = 2;
        var s = new Func<string, int>[] { str => 2 * str.Length, str => 3 * str.Length, str => 4 * str.Length,
                                          str => 5 * str.Length, str => 6 * str.Length, str => 7 * str.Length,
                                          str => 8 * str.Length, str => 9 * str.Length, str => 10 * str.Length
                                        };
        Assert.Throws<InconsistentSourceException>(() => s.MultipleApply(v, n).ToArray());
        // str => string.IsNullOrEmpty(str) ? 1 : 0;
    }

    [Test]
    public void Test3()
    {
        var v = "boom";
        var n = 0;
        var s = NonFinisconoMai();
        Assert.Throws<ArgumentOutOfRangeException>(() => s.MultipleApply(v, n).ToArray());
    }
    public IEnumerable<Func<string, int>> NonFinisconoMai()
    {
        while (true)
        {
            yield return str => 2 * "RotoloniRegina".Length;
        }
    }
}
```

```
[Test]
0 references
public void Test1(){
    Func<int, int, int> coFunc1 = (x, y) => x + y;
    IEnumerable<int[]> exp = new[]{
        new int[] { 1 }, new int[] { 1, 1 }, new int[] { 1, 2, 1 }, new int[] { 1, 3, 3, 1 },
        new int[] { 1, 4, 6, 4, 1 }
    };
    Assert.That(ExtraMath.GeneralizedTartaglia(1, coFunc1).Take(5), Is.EqualTo(exp));
}
[Test]
0 references
public void Test2()
{
    try{
        Func<string, string, string> coFunc = delegate(string s, string s1){
            return (s.Length != 4 && s1.Length != 4) ? s + s1 : throw new MyException("Concat");
        };
        var seed = "X";
        var a = ExtraMath.GeneralizedTartaglia(seed, coFunc).ToArray().Take(6);
    }
    catch (AggregateException aex){
        int i = 0;
        foreach (MyException innerException in aex.InnerExceptions){
            Console.WriteLine(innerException.Index);
            Assert.That(innerException.Index, Is.EqualTo(++i));
        }
    }
}
[TestCase(0)] // [TestCase(1)] // [TestCase(5)]
0 references
public void Test3(int lineNumber){
    if (lineNumber < 1) Assert.Inconclusive();
    int isCalled = 0;
    Func<int, int, int> coFunc = (x, y) => x + y;
    var a = ExtraMath.GeneralizedTartaglia(1, coFunc).Take(lineNumber);
    foreach (var line in a){
        for (int i = 0; i < line.Length; i++)
            Console.Write(line[i]);
        var lenght = line.ToArray().Length;
        if (lenght != 1) isCalled += lenght - 2;
        Console.Write(isCalled);
        Console.WriteLine();
    }
    Assert.That(isCalled, Is.EqualTo(((lineNumber - 1) * (lineNumber - 2)) / 2));
}
```

```
namespace Esame_8_9_22
{
    3 references
    public class MyException : Exception{
        1 reference
        private static int _count;
        0 references
        public int Index { get; } = ++_count;
        0 references
        public MyException(){}
        0 references
        public MyException(string? message) : base(message){}
        0 references
        public MyException(string? message, Exception? innerException) : base(message, innerException){}
    }
}
```

```
namespace Esame_8_9_22
{
    0 references
    public static class ExtraMath{
        0 references
        public static IEnumerable<T[]> GeneralizedTartaglia<T> (T seed,Func<T,T,T>generator){
            var exception = new List<Exception>();
            int i = 0;
            T[] pre = new T[1];
            pre[0]= seed;
            while (true){
                yield return pre;
                int size = 2 + i;
                T[] next = new T[size];
                next[0] = seed;
                next[size - 1] = seed;
                    if (i != 0){
                        for (int j = 1; j < size - 1; j++){
                            try{
                                var contr = generator(pre[j - 1], pre[j]);
                                next[j] = contr;
                            }
                            catch (MyException e){
                                exception.Add(e);
                            }
                        }
                    }
                    if (exception.Any())
                        throw new AggregateException("Aggregation", exception);
                    pre = next;
                i++;
            }
        }
    }
}
```

```
public class MultipleEnumerable<T> : IEnumerable<T[]>
{
    IEnumerable<T>[] Source { get;}
    public MultipleEnumerable(IEnumerable<T>[] source)
    {
        Source = source;
    }
    public IEnumerator<T[]> GetEnumerator()
    {
        return new MultipleEnumerator<T>(Source.Select(e=> e.GetEnumerator()).ToArray());
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
public class MultipleEnumerator<T> : IEnumerator<T[]>
{
    IEnumerator<T>[] Enumerators { get; }
    public MultipleEnumerator(IEnumerator<T>[] enumerators){
        Enumerators = enumerators;
    }
    public bool MoveNext(){
        foreach (var e in Enumerators)
            if (!(e.MoveNext())) return false;
        return true;
    }
    public void Reset(){
        foreach (var e in Enumerators)
            e.Reset();
    }
    public T[] Current{
        get { return Enumerators.Select(e => e.Current).ToArray(); }
    }
    object IEnumerator.Current {
        get{ return /*this.*/Current; }
    }
    public void Dispose(){
        foreach (var e in Enumerators)
            e.Dispose();
    }
}
```

```
[Test]
0 references | Run Test | Debug Test
public void TestDisposeCallsForEachInnerEnumerator()
{
    // Mock the IEnumerator<int>
    var mockEnumerator = new Mock<IEnumerator<int>>();
    mockEnumerator.Setup(e => e.MoveNext()).Returns(false); // Enumerator is done

    // Mock the IEnumerable<int>
    var mockEnumerable = new Mock<IEnumerable<int>>();
    mockEnumerable.Setup(e => e.GetEnumerator()).Returns(mockEnumerator.Object);

    // Create the MultipleEnumerable<int> instance with 5 mock IEnumerable<int> instances
    var source = new List<IEnumerable<int>>
    {
        mockEnumerable.Object,
        mockEnumerable.Object,
        mockEnumerable.Object,
        mockEnumerable.Object,
        mockEnumerable.Object
    };

    var receiver = new MultipleEnumerable<int>(source.ToArray()).GetEnumerator();
    receiver.Dispose();

    // Verify that Dispose was called once for each inner IEnumerator<int>
    mockEnumerator.Verify(e => e.Dispose(), Times.Exactly(5));
}
```

```
[Test]
public void Test4()
{
    var source = new IEnumerable<int>[] {new[]{1, 2}, new[]{3, 4, 5}};
    var enumerators = source.Select(e => new Mock<IEnumerator<int>>().Object).ToArray();
    var receiver = new MultipleEnumerator<int>(enumerators);
    receiver.Dispose();
    foreach (var e in enumerators)
        Mock.Get(e).Verify(x => x.Dispose(), Times.Once);
}
```

```
public class Tests{
    [Test]
    public void Test1(){
        var source = new IEnumerable<int>[] {new[]{1, 2}, new[]{3, 4, 5}};
        var receiver = new MultipleEnumerable<int>(source).GetEnumerator();
        Assert.Multiple(() => {
            Assert.That(receiver.MoveNext(), Is.True);
            Assert.That(receiver.MoveNext(), Is.True);
            Assert.That(receiver.MoveNext(), Is.False);
        });
    }
    [Test]
    public void Test2([Range(1, 10)] int howMany){
        if (!(howMany < 0)) Assert.Inconclusive();
        static IEnumerable M(int i){var x = 0; while(true) yield return x += i;}
        var source = new IEnumerable<int>[10];
        for (int i = 0; i < 10; i++)
            source[i] = (IEnumerable<int>)M(i);
        var receiver = new MultipleEnumerable<int>(source).GetEnumerator();
        Assert.Multiple(() => {
            for (int i = 0; i < 10; i++){
                var expected = new int[10];
                for (int j = 0; j < 10; j++) expected[j] = j * i;
                receiver.MoveNext();
                Assert.That(receiver.Current, Is.EqualTo(expected));
            }
        });
    }
    public class MyMock : IEnumerable<int>{
        public int Calls { get; /*private*/ set; }
        public IEnumerator<int> GetEnumerator(){ return new MyEnum(this); }
        IEnumerator IEnumerable.GetEnumerator(){ return /*this.*/GetEnumerator(); }
    }
    public class MyEnum : IEnumerator<int>{
        public MyEnum(MyMock k) { _k = k; }
        MyMock _k;
        public bool MoveNext => true;
        public int Current => 42;
        object IEnumerator.Current => 42;
        public void Dispose() { _k.Calls++; }
        public void Reset() { }
        bool IEnumerator.MoveNext(){ return MoveNext; }
    }
    [Test]
    public void Test3(){
        var source = new IEnumerable<int>[] {new MyMock(), new MyMock(), new MyMock(), new MyMock(), new MyMock()};
        var receiver = new MultipleEnumerable<int>(source).GetEnumerator();
        receiver.Dispose();
        var calls = source.Select(s => ((MyMock)s).Calls);
        Assert.Multiple(() => {
            Assert.That(calls, Is.All.EqualTo(1));
            for (int i = 0; i < 5; i++)
                Assert.That(((MyMock)source[i]).Calls,Is.EqualTo(1));
        });
    }
```

```
namespace _11set2023;
public class Tests
{
    [TestCase(new Cards[] { Cards.Ace, Cards.Two }, new Suits[] { Suits.Spades, Suits.Clubs })]
    [TestCase(new Cards[] { Cards.Ace, Cards.Two, Cards.Three }, new Suits[] { Suits.Spades, Suits.Clubs, Suits.Diamonds })]
    public void Test1(Cards[] cards, Suits[] suits)
    {
        if(cards.Length != suits.Length || cards.Length % 3 != 0 || suits.Length % 3 != 0)
            Assert.Inconclusive();

        var deck = GenerateSequence(cards, suits);
        var actual = deck.FirstWins();
        var expected = Enumerable.Repeat(true, cards.Length / 3); // in questo caso abbiamo deck=6, quindi solo una mano -> 3/3 valori di ritorno

        Assert.That(actual.ToList(), Is.EqualTo(expected.ToList()));
    }
    private IEnumerable<IPlayingCard> GenerateSequence(Cards[] cards, Suits[] suits)
    {
        for (int i = 0; i < cards.Length; i++){
            yield return new PlayingCard(cards[i], suits[i]);
            yield return new PlayingCard(cards[i], suits[i]);
        }
    }

    [Test]
    public void Test2()
    {
        var source = InfiniteSequence();
        var actual = source.FirstWins().Take(1000).ToArray().Length;
        var expected = 1000;

        Assert.That(actual, Is.GreaterThanOrEqualTo(expected));
    }
    IEnumerable<IPlayingCard> InfiniteSequence()
    {
        var random = new Random();
        while (true)
            yield return new PlayingCard((Cards)random.Next(0, 10), (Suits)random.Next(0, 4));
    }
    [Test]
    public void Test3()
    {
        var cards = new Cards[] { Cards.Ace, Cards.Two, Cards.Three, Cards.Four, Cards.Five, Cards.Six, Cards.Seven };
        var suits = new Suits[] { Suits.Spades, Suits.Clubs, Suits.Diamonds, Suits.Hearts, Suits.Spades, Suits.Clubs, Suits.Diamonds };
        var source = GenerateSequence(cards, suits);
        Assert.That(() => source.FirstWins().ToList(), Throws.ArgumentException);
    }
```

```
namespace _11set2023;
21 references | 3 references | 3 references | 2 references | 1 reference | 1 reference | 1 reference | 1 reference | 0 references | 0 references | 0 references | 21 references | 4 references | 4 references | 3 ref
public enum Cards {Ace, Two, Three, Four, Five, Six, Seven, Jack, Queen, King} public enum Suits {Spades, Clubs, Diamonds, Hearts}
9 references
public interface IPlayingCard {
    10 references | 6 references
    Cards Value { get ; } Suits Suit { get ; }
    static bool operator <=(IPlayingCard first, IPlayingCard second) {
        if (first.Value <= second.Value) return true;
        if (first.Value > second.Value) return false;
        if (first.Suit <= second.Suit) return true;
        return false;}
    static bool operator >=(IPlayingCard first, IPlayingCard second) {
        if (first.Value <= second.Value) return true;
        if (first.Value > second.Value) return false;
        if (first.Suit <= second.Suit) return true;
        return false;}
}
4 references
public class PlayingCard : IPlayingCard {
    10 references | 6 references | 3 references
    public Cards Value { get; } public Suits Suit { get; } public PlayingCard(Cards value, Suits suit) { Value = value; Suit = suit; }
}
0 references
public static class Extensions {
    3 references
    public static IEnumerable<bool> FirstWins(this IEnumerable<IPlayingCard> deck) { // : IPlayingCard NON SERVE !!!
        var currentHand = new List<IPlayingCard>();
        using (var enumerator = deck.GetEnumerator()) {
            while (enumerator.MoveNext()) {
                currentHand.Add(enumerator.Current);
                var maxFirst = currentHand[0];
                var maxSecond = currentHand[0]; // can't be index 1 because it'll go out of range
                if (currentHand.Count%6 == 0) {
                    for(int i=0; i<6; i+=2)
                        if (currentHand[i] >= maxFirst)
                            maxFirst = currentHand[i];
                    for(int i=1; i<6; i+=2)
                        if (currentHand[i] >= maxSecond)
                            maxSecond = currentHand[i];
                    yield return maxFirst >= maxSecond;
                    currentHand.Clear();
                }
            }
        }
        if (currentHand.Count%6 != 0)
            throw new ArgumentException();
```