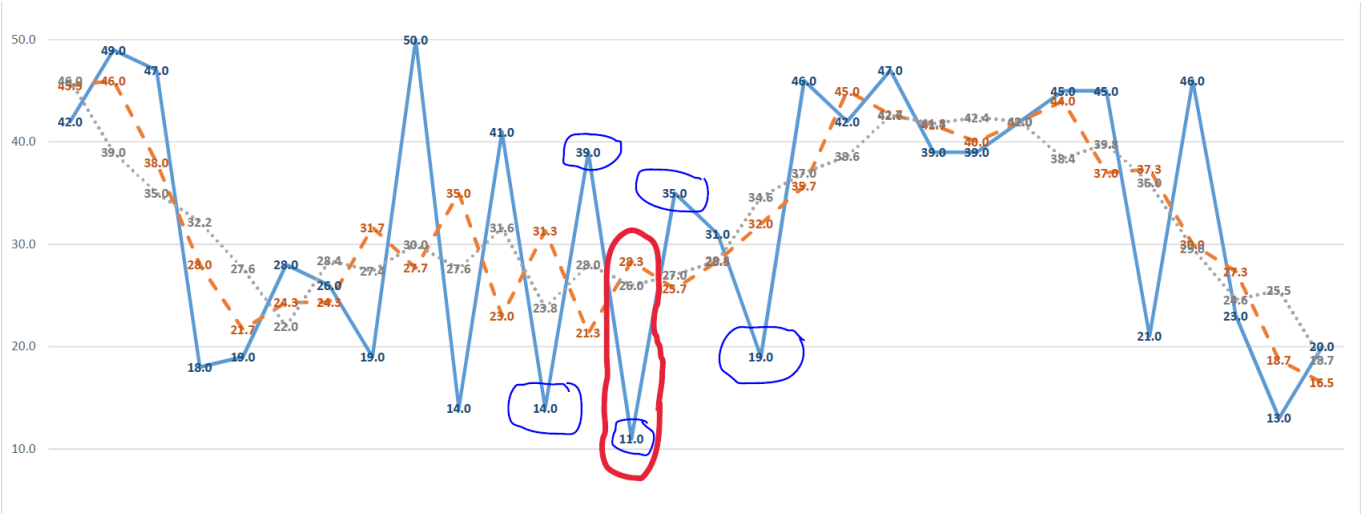
Appello TAP del 7/7/2021

Scrivere nome, cognome e matricola sul foglio protocollo. Avete a disposizione due ore e mezza.

Esercizio 1 (9 punti)

Si parla di *smoothing* dei dati quando si ha una sorgente di valori e se ne produce una che li approssima cercando di diminuire i dislivelli fra valori adiacenti. Uno dei modi semplici di fare smoothing è mediare ciascun valore con quelli precedenti e successivi. Ad esempio in figura potete vedere una serie di valori (linea continua azzurra) e i relativi smoothing usando la media del valore con il precedente e il successivo (linea tratteggiata arancione), o con i due precedenti e i due successivi (linea grigia a puntini). Il punto evidenziato in rosso, ad esempio, vale 11.0 nella serie originaria, 28,3 nello smoothing più grezzo (media di 39, 11 e 35) e 26.0 in quello più raffinato (media di 14, 39, 11, 35 e 31).



Scrivere l'extension-method `Smooth` che, data una sequenza **infinita** `s` di elementi di tipo double e un parametro intero `N`, produce una sequenza **infinita** di elementi di tipo double ottenuti come media dell'elemento nella stessa posizione, gli `N` precedenti e gli `N` successivi.

All'inizio della sequenza la media sarà calcolata sui soli punti disponibili, ad esempio se `N` vale 3, il risultato sarà la sequenza (dove *avg* indica la media aritmetica)

$$avg(s_0, ..., s_3),\ avg(s_0, ..., s_4),\ avg(s_0, ..., s_5),\ avg(s_0, ..., s_6),\ avg(s_1, ..., s_7),\ avg(s_2, ..., s_8)...$$

Il metodo dovrà sollevare:

- `ArgumentNullException` se `s` è null
- `ArgumentOutOfRangeException` se `N` è strettamente negativo
- `FiniteSourceException` se la sorgente è finita.

Esercizio 2 ([1+2+5] = 8 punti)

Implementare, usando NUnit, i seguenti test relativi a `Smooth`, dell'esercizio 1.

1. Input della chiamata sotto test: `s` è la sequenza $42.0, 49.0, 47.0, 18.0, 19.0, 28.0, 26.0$, `N` vale 2

   Output atteso: una `FiniteSourceException`.

2. Input della chiamata sotto test: `s` è una qualsiasi sequenza **infinita**, `N` vale -1

   Output atteso: una `ArgumentOutOfRangeException` sollevata **senza enumerare la sorgente** neppure parzialmente

3. Test parametrico con 4 parametri:

   - un intero `N` da usare per la chiamata sotto test
   - un array `sourceSample` di double contenente `K` elementi da usare per costruire una sequenza infinita, la sorgente per la chiamata sotto test
   - un array `expectedSample` di double contenente `K + N` elementi da usare per costruire il risultato atteso della chiamata sotto test
   - parametro intero `howMany`, strettamente positivo che rappresenta quanti elementi del risultato devono essere verificati dal test.

   Input della chiamata sotto test: una sequenza **infinita** in cui si ripetono ciclicamente gli elementi di `sourceSample` Ad esempio se `sourceSample` è la sequenza i_0, i_1, i_2, i_3, i_4, l'input per la chiamata sotto test sarà la sequenza

   $$i_0, i_1, i_2, i_3, i_4, i_0, i_1, i_2, i_3, i_4, i_0, i_1, i_2, i_3, i_4, i_0, i_1, i_2, i_3, i_4 ...$$

   Output atteso: una sequenza **infinita** i cui primi `N` coincidono con i primi `N` di `expectedSample` e i successivi sono ottenuti ripetendo ciclicamente gli altri elementi di `expectedSample`.

   Ad esempio se `N` vale 3 e `expectedSample` è la sequenza $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$, l'output atteso sarà la sequenza $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_4, x_5, x_6, x_7, x_3, x_4, x_5, x_6, x_7 ...$

   N.B. Il test deve verificare **solo** che i primi `howMany` elementi del risultato coincidano con quanto atteso.

Esercizio 3 (4 punti)

Dato il seguente frammento di codice, quali test avranno successo?

```
public static IEnumerable<double> M(this IEnumerable<int> s) {
    if (null==s) throw new ArgumentNullException();
    return PrivateM();

    IEnumerable<double> PrivateM() {
        foreach (var n in s) {
            if (0 == n) throw new ArgumentException();
            yield return 1.0 / n;
        }
    }
}

[TestFixture]
public class Test {
IEnumerable<int> S() {
    var i = -123;
    while (true) yield return i++;
}
IEnumerable<int> Null() { return null; }
[Test]
public void Test1()
  {Assert.That(S().M(),Throws.TypeOf<ArgumentException>());}
[Test]
public void Test2()
  {Assert.That(()=>S().M(), Throws.TypeOf<ArgumentException>());}
[Test]
public void Test3()
  {Assert.That(()=>S().M().ToArray(), Throws.TypeOf<ArgumentException>());}
[Test]
public void Test4()
  {Assert.That(() => S().M().Take(200), Throws.TypeOf<ArgumentException>());}
[Test]
public void Test1Null()
  {Assert.That(Null().M(), Throws.TypeOf<ArgumentNullException>());}
[Test]
public void Test2Null()
  {Assert.That(() => Null().M(), Throws.TypeOf<ArgumentNullException>());}
[Test]
public void Test3Null()
  {Assert.That(()=>Null().M().ToArray(),Throws.TypeOf<ArgumentNullException>());}
[Test]
public void Test4Null()
  {Assert.That(()=> Null().M().Take(200),Throws.TypeOf<ArgumentNullException>());}
}
```

	Success	Fail
Test1	☐	☒
Test2	☐	☒
Test3	☒	☐
Test4	☐	☒

manca la lambda

non c'è il .ToList()

.Take() non enumera

However, the Take(200) method is used to limit the number of elements taken from the sequence to 200. Since there is no condition to stop the infinite sequence in the S() method, the Take(200) will keep trying to take elements indefinitely, resulting in the test case not completing and eventually timing out.

	Success	Fail
Test1Null	☐	☒
Test2Null	☒	☐
Test3Null	☒	☐
Test4Null	☒	☐

effettivamente ritornano tutti null

Esercizio 4 (4 punti)

Supponendo che le seguenti classi siano gestite usando l'Entity Framework, indicare se le affermazioni seguenti sono vere o false.

```
public class Student {
  public int StudentId { get; set; }
  [Required]
  [Index("IeIndex",Order=1,IsUnique = true)]
  public string StudentName { get; set; }
  [Index("IeIndex", Order = 2, IsUnique = true)]
  public DateTime? DateOfBirth { get; set; }
  public byte[] Photo { get; set; }
  public decimal Height { get; set; }
  [Required]
  public float Weight { get; set; }

  public Grade Grade { get; set; }
  public int TutorId { get; set; }
}
public class Grade {public int GradeId { get; set; }/*...*/ }
public class Tutor {public int TutorId { get; set; }/*...*/ }
```

Vero	Falso	
☒	☐	l'attribute Required per la property StudentName è superfluo, perché è una chiave quindi non nullabile
☒	☐	l'attribute Required per la property StudentName rende non nullabile la colonna corrispondente nel DB
☒	☐	l'attribute Required per la property StudentName genera la verifica che il valore non sia nullo durante la chiamata di SaveChanges prima di effettuare la connessione al DB
☐	☒	l'attribute Required per la property Weight è superfluo, perché il tipo float non è nullabile → i tipi valore non sono nullabili di default, hanno bisogno del "?"
☒	☐	l'attribute Required per la property Weight è indispensabile, perché il tipo float non è nullabile, se no si avrebbe un conflitto con il default del DB in cui le colonne sono nullabili. Una valida alternativa per evitare il conflitto sarebbe dichiarare Weight con tipo float?. Ma se si lascia l'inconsistenza si ottiene un errore di compilazione.
☒	☐	la property Grade rappresenta una proprietà di navigazione verso la classe Grade
☐	☒	affinché la property Grade rappresenti una proprietà di navigazione verso la classe Grade è indispensabile aggiungere anche la property public int GradeId → non è un requisito assoluto
☐	☒	la property Grade rappresenta il lato uno di una associazione uno a molti verso la classe Grade qualunque sia il codice di Grade → credo sia un'associazione (1,1)
☐	☒	se la classe Grade non contiene nessuna property di navigazione verso la classe Student, allora la property Grade rappresenta il lato uno di una associazione uno a molti verso la classe Grade → Per stabilire un'associazione uno-a-molti tra due classi, entrambe le classi devono avere una relazione tra di loro, di solito attraverso una property di navigazione
☒	☐	la property TutorId rappresenta una chiave esterna che riferisce a Tutor
☐	☒	la property TutorId è un intero che nulla ha a che vedere con la classe Tutor