Appello TAP del 16/02/2015

Scrivere nome, cognome e matricola sul foglio protocollo, indicando anche se avete nel piano di studi TAP da 8 CFU (quello attuale) o da 6 CFU (quello "vecchio") e se avete consegnato i test durante l'anno e intendete usufruire del bonus così conseguito.

Chi deve sostenere TAP da 6 CFU non deve svolgere l'esercizio 3 e chi ha consegnato i test durante l'anno (e intende usufruire del bonus conseguito) non deve svolgere l'esercizio 4; per loro il punteggio indicato nel testo sarà scalato, di conseguenza, in modo che il massimo conseguibile sia sempre 30. Avete a disposizione mezzora per esercizio. In sintesi:

Tipo TAP	Bonus Test	Esercizi da svolgere	Tempo a disposizione	Fattore di normalizzazione
6CFU	sì	1 e 2	1h	$\frac{30}{10+6}$
6CFU	no	1, 2 e 4	1h 30'	$\frac{30}{10+6+9}$
8CFU	sì	3, 1 e 2	1h 30'	$\frac{30}{5+10+6}$
8CFU	no	tutti (3, 1, 2 e 4)	2h	1

Esercizio 1 (punti 10) Utilizzando la seguente struct

```
public struct Range    {
        public double Min;
        public double Max;
        public Range(double min, double max){Min = min; Max = max;}
    }
```

per rappresentare un intervallo su $\mathbb{R}$, scrivere l'extension-method `ConvergingToZero` che, presa una funzione `f` (che si assume continua senza fare verifiche), un intervallo di partenza ed un double `epsilon` che rappresenta il livello di tolleranza accettato, restituisce una sequenza (potenzialmente infinita) di intervalli sempre più piccoli che contengono uno degli zeri di `f`, secondo il metodo *double false position*, descritto in seguito.

Il metodo dovrà prendere come parametri:

1. (come parametro "this") `f`, la funzione di cui calcolare uno zero, di tipo `Fun`<double, double>;
2. `range`, l'intervallo di partenza da cui cominciare ad applicare il metodo; il segno di `f` su `range.Min` e su `range.Max` dovrà essere diverso, altrimenti `ConvergingToZero` dovrà sollevare un'eccezione di tipo `ArgumentException`;
3. `epsilon`, l'approssimazione scelta per lavorare sui double.

Il metodo deve sollevare l'eccezione...

- `ArgumentNullException` se `f` è null;
- `ArgumentException` se si verifica (almeno) una fra le seguenti condizioni:
  - `range.Min` > `range.Max` (cioè l'intervallo è vuoto);
  - `f` ha lo stesso segno su `range.Min` e `range.Max`;
  - `f` su `range.Min` o su `range.Max` vale 0 a meno di `epsilon`;
  - `epsilon` ≤ 0.

Metodo double false position. Si parte con un intervallo $[a_0, b_0]$ tale che f abbia segno diverso sui suoi estremi, ovvero che $f(a_0) > 0$ e $f(b_0) < 0$ o, viceversa, $f(a_0) < 0$ e $f(b_0) > 0$.

Al passo $k+1$ si calcola il punto

$$c_{k+1} = b_k - f(b_k)\frac{b_k - a_k}{f(b_k) - f(a_k)}$$

e si restituisce $[a_{k+1}, b_{k+1}]$, dove

- se $f(c_{k+1}) = 0$ (a meno di `epsilon`), allora è stata trovata la soluzione esatta, quindi **si interrompe l'iterazione**, e si pone $a_{k+1} = b_{k+1} = c_{k+1}$;
- se $f(c_{k+1})$ ha lo stesso segno di $f(a_k)$, allora si pone $a_{k+1} = c_{k+1}$ e $b_{k+1} = b_k$;
- se $f(c_{k+1})$ ha lo stesso segno di $f(b_k)$, allora si pone $a_{k+1} = a_k$ e $b_{k+1} = c_{k+1}$.

Esercizio 2 (punti 6)

Si implementi una struct `Temperature` per rappresentare la temperatura, sia in gradi centigradi (Celsius) che in gradi fahrenheit.

La struct `Temperature` dovrà fornire:

- una proprietà booleana vera se l'elemento è espresso in gradi Celsius (attenzione: a seconda di come implementate il tipo di dato, quando si modifica questa proprietà potrebbe essere necessario modificare anche campi/altre proprietà per continuare a rappresentare la stessa temperatura);
- un costruttore che preso un double `degree` e un booleano (opzionale) `isCelsius` produce la temperatura corrispondente a `degree` gradi in centigradi, se `isCelsius` è vero od omesso, altrimenti in fahrenheit; se il parametro `degree` è tale per cui la temperatura risulterebbe inferiore allo zero assoluto ($-273.15\ C^\circ$) il costruttore solleverà `ArgumentException`;
- l'override del metodo `ToString`() che su un elemento che rappresenta, ad esempio, 42 gradi centigradi restituisca la stringa "42 C" e su un elemento che rappresenta 42 gradi fahrenheit restituisca la stringa "42 F";
- la ridefinizione degli operatori $<$ e $>$;
- la conversione implicita di un double alla temperatura corrispondente, assumendo che il valore sia espresso in centigradi
- la definizione dell'operatore $+$ che permetta di sommare un double `x` ad una temperatura `t` per produrre una nuova temperatura ottenuta sommando `x` ai gradi di `t`.

Per chi non si ricordasse, la formula di conversione fra centigradi e fahrenheit è (con ragionevole approssimazione) $x\ C^\circ == x * \frac{9}{5} + 32\ F^\circ$

Esercizio 3 (punti 5) Si consideri il seguente metodo

```
public double AverageError<T>(IEnumerable<Func<T, int>> measures, T sample,
                                                        Func<int,int> correction )
        {
            if (measures == null)
                throw new ArgumentNullException("measures");
            if (correction == null)
                throw new ArgumentNullException("correction");
            //Some refactoring needed to avoid double enumeration of measures
            var b = measures.Count();
            if (b == 0)
                throw new ArgumentException("No measures to be analized");
            var a = measures.Select(f => f(sample)).
                                Select(x => Math.Abs(x - correction(x))).Sum();
            return a / b;
        }
```

Dare l'implementazione della sua versione asincrona `AverageErrorAsync<T>`, evitando doppie enumerazioni di `measures` ed usando `PLinq`

Esercizio 4 (punti 3+3+3 = 9 punti)

- Elencare, descrivendoli a parole, una lista di test significativi per il metodo `ConvergingToZero`, dell'esercizio 1.
- Implementare, usando NUnit, due test della lista precedente; uno che vada a testare un caso "buono" (ovvero, dove ci si aspetta che l'invocazione di `ConvergingToZero` vada a buon fine) e uno che vada a testare un caso "cattivo" (ovvero, dove ci si aspetta che l'invocazione di `ConvergingToZero` sollevi un'eccezione).
- Implementare, usando NUnit ed eventualmente Moq, un test in cui si verifica che `f` venga effettivamente invocata.