(*) (1) Fib(0)=0 Fib(1)=1 Fib(n)=Fib(n-1)+Fib(n-2)

Definire la funzione che ritorna l'ennesimo numero di Fibonacci in versione ricorsiva di coda in modo che sia il corretto risultato se applicata a 1000000.

(2.1) Dare la versione ricorsiva di coda della seguente definizione

```
val lunghezza: (List[Int] => Int) = (l) =>
    l match {
        case Nil => 0
        case _::tl => 1+ lunghezza(tl)
    }
```

(2.2) Scrivere la funzione mioMap:List[Int]=>(Int=>Int)=>List[Int] ricorsiva di coda

(2.3) Scrivere la funzione flatMap: List[Int] =>(Int=>List[Int])=>List[Int]

(*) (3) Definire una funzione caratteri ricorsiva di coda che ha come input una lista di caratteri e ritorna una tupla di 5 elementi che elenca il numero di 'a', 'e', 'i', 'o', 'u' che occorrono nella stringa. Ad esempio

caratteri (List('a','b','e','i','a' 'c'))

deve ritornare

(2,1,1,0,0)

(4) Si definisca una funzione che presa una lista di liste la appiattisce cioe' data:

List(List(1,2,3),List(4,5),List(6))

restituisce

List(1,2,3,4,5,6)

Versione ricorsiva e ricorsiva di coda.

(*) (5) Scrivere una funzione che inserisce un elemento in una lista di interi ordinata usando takeWhile e dropWhile

(6) Scrivere flatMap: List[Int] =>(Int=>List[Int])=>Lis[Int] usando i metodi map e reduce di Scala

(*) (7) Il metodo to degli interi es 1.to (10) ha un parametro intero e produce un Range di numeri dal receiver del metodo al parametro. Con questo metodo e il metodo reduce delle sequenze (Range e' una sequenza) scrivere

la funzione fattoriale (senza chiamate ricorsive e/o test).

(8) Scrivere una funzione somma che ha come parametro una funzione f:Int=>Int e restituisce una funzione che prende come parametri 2 interi a e b e restituisce la somma fi f(x) per ogni x compreso fra a e b (estremi inclusi)
Es: somma(x=>2*x) restituisce (a,b)=>f(a)+.....+f(b) che poi applicata a 5 e 7 ritorna 36

(*) (9) Data una lista di interi che rappresenta una stringa binaria, cioe'
val listBin=List(0,1,1,1) rappresenta il nunero binario 1110
usare gli operatori foldLeft e foldRight per calcolare il valore
della stringa binaria.

(*) (10) Data una lista di interi scrivere una funzione che ritorna la sottostringa più lunga di elementi che soddisfano tutti una proprieta'. Quindi i tipi dei parametri sono List[Int] e Int => Boolean e risultato List[Int]. Usare i metodi di ordine superiore delle liste (pag. 17 lucidi Pattern di programmazione funzionale: ricorsione di coda, ricorsione su liste).
sottostringa (List(1,2,3,6,4,9,12,15),(x:Int)=>x%3==0) ⇒ List(9,12,15)

(11) Assumendo che una tripla
(String, Boolean, List[(String,Boolean, List[(String,Boolean,Nil.type)])])
rappresenti una persona (nome,sesso, lista dei figli) (decidete se Uomo/Donna e' rappresentato da true/false)
ad esempio:

```
val paola =("Paola", true,Nil)
val andrea =("Andrea", false,List(paola))
val peter =("Peter", false,Nil)
val giulia =("Giulia", true, List(paola, peter))
val persone = List(paola, peter, giulia)
```

usando il for espressione (pag. 29 lucidi Pattern di programmazione funzionale: ricorsione di coda, ricorsione su liste) e i pattern per estrarre i campi delle tuple generare la lista
1) delle persone il cui nome inizia con “P”
2) delle coppie (nome madre, nome figlia/o)

(*) (12) Il metodo zip in una collezione iterabile di tipo A, list
def zip[B](that: GenIterable[B]): Iterable[(A, B)]
ritorna la collezione delle coppie di elementi con lo stesso indice in list e that. Esempio
List(1,2,3) zip List('a','b','c','d') ritorna List((1,a), (2,b), (3,c))
List(1,2,3) zip List('a','b') ritorna List((1,a), (2,b))

Usare il metodo zip e map per generare; data una lista, la lista delle differenze degli elementi adiacenti della lista. Esempio
List(3,2,7,4,4) => List(1,-5,3,0)

(*) (13) Scrivere una funzione che ha come input il nome di un file.
Dopo aver diviso il file in parole usando come separatori ' ', ';', ','
(usate il metodo split della classe delle stringhe e la classe Scanner per la lettura delle parole da file)
producete una lista con la frequenza delle parole (cioe' quante volte compaiono nel file) ordinata dalla più frequente alla meno frequente.
Cercate di fare questo esercizio nel modo + funzionale possibile (cioe' come una sequenza di trasformazioni di strutture dati!)

(14) Scrivere la funzione take:LazyList[Int]=>Int=>LazyList[Int] tale che
take (s) (n)
e' uno Stream contenente i primi n elementi di s (o tutti gli elementi se ce ne sono meno).
DOMANDA: Come fate ad accertarvi che la LazyList risultante abbia la lunghezza giusta?

(*) (15) Scrivere una funzione
val sequenza: List[Option[Int]] => Option[List[Int]]
tale che se la lista di input contiene tutti Some(n) ritorna Some della lista degli elementi. Se c'e un None ritorna None. Cioe'
sequenza (List(Some(1),Some(2),Some(77)) ==> Some(List(1,2,77))
sequenza (List(Some(1),None,Some(77)) ==> None
Definiamo la funzione con un parametro per nome
val tentaVal:(=> Int)=>Option[Int] =(a)=>
try Some(a)
catch { case e: Exception => None }
Poi usando le funzioni sequenza e tentaVal scrivere una funzione
val parseInts:List[String]=>Option[List[Int]]
che ritorna una lista di interi se tutte le stringhe nella lista di input rappresentano interi altrimenti ritorna None.
Cioe'
parseInts (List("1","2","77")) ==> Some(List(1,2,77))
parseInts (List("1","due","77")) ==> None

(*) (16) Prendete la seguente definizione parziale della classe Rational

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val m = mcd(n.abs, d.abs)
  val numer = n / m
  val denom = d / m
  def this(n: Int) = this(n, 1)
```

```
    def + (that: Rational): Rational =
      new Rational(
        numer * that.denom + that.numer * denom,
        denom * that.denom
      )
    override def toString = numer.toString +"/"+ denom.toString
    private def  mcd(a: Int, b: Int): Int = 1
      if (b==0) a else mcd(b,a%b)
  }
```

a) definire un overloading del + che prende come input in Int
d) definire i metodi analoghi per l'operatore *
e) definire i metodi apply e unapply in modo tale che sia possibile applicare il pattern matching a oggetti del tipo Rational.

(*) (17) Date le seguenti definizioni di classi case

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
          left: Expr, right: Expr) extends Expr
```

definite una funzione

```
semplifica:Expr => Expr
```

che:

a) semplifica la doppia negazione
b) semplifica la somma di 0, la moltiplicazione per 1 e per 0.

produce un'espressione completamente semplificata

c) ridefinite i metodi toString delle classi in modo che producano una versione stringa delle espressioni simile alla scrittura usuale.

(*) (18) Definire una classe Stack generica con le operazioni push, top, pop e isEmpty che abbia un parametro di tipo (per gli elementi contenuti nello stack) covariante. Quindi dovra' necessariamente essere immutabile. Definite prima una classe astratta e poi due sottoclassi (case) una per lo stack vuoto e una per uno stack che contiene un elemento al top e uno stack come tail.

(*) (19) Definire altri metodi per la classe Conto (che la rendano utilizzabile)
Definire il trait Logger astratto con ConsoleLogger ShortLogger e TimedLogger come definiti nelle slide
Modificare FileLogger in modo da specificare il nome del file in cui mettere i log
Definire CeasarLogger che produce una versione cifrata della stringa

(usare la cifratura Cesare: trasformare i caratteri shiftando i caratteri di n posizioni
3 se non diversamente specificato)
Definire oggetti con diverse combinazioni dei precedenti