TPFI 2021/22
Hw 3: Liste Infinite

assegnato: 13 aprile 2022, consegna 29 aprile 2022

Esercizio 1 (INSOMNIA) Scrivere una funzione Haskell che genera la lista infinita di caratteri `insonnia = "1 sheep 2 sheep 3 sheep 4 sheep ..."`. Provare a scrivere un "one-liner", cioè un programma che semplicemente compone opportunamente funzioni. Può essere utile la funzione `show :: Show a => a => String` che trasforma un elemento di qualsiasi tipo che implementa la classe `Show` in una stringa, cioè una lista di caratteri.

Esercizio 2 (TRIANGOLO INFINITO DI TARTAGLIA) Definite in Haskell la lista infinita di liste finite `tartaglia`, tale che `tartaglia!!n` sia l'n-esima riga del triangolo di Tartaglia, e quindi `tartaglia!!n!!k` sia il coefficiente binomiale $\begin{pmatrix} n \\ k \end{pmatrix}$.

Esercizio 3 (NUMERI FORTUNATI) I *numeri fortunati*, introdotti anch'essi da Stanislaw Ulam, sono definiti come segue.

Dalla sequenza dei numeri naturali tolgo tutti i *secondi numeri*, cioè i pari.

A quel punto, il *secondo numero rimasto* è il 3 e quindi si tolgono tutti i terzi numeri tra i sopravvissuti (5, 11, 17, ...).

Ora considero il *terzo numero rimasto* cioè il 7 e rimuovo tutti i settimi numeri (il primo è il 19) e così via, fino a ottenere tutti i numeri sopravvissuti a tutte le operazioni di "filtraggio".

Scrivere una funzione Haskell che genera lo stream dei numeri fortunati.

Esercizio 4* (ULAM NUMBERS RELOADED) Dare una definizione circolare in Haskell della lista infinita dei numeri di Ulam (lezione **13**), selezionandoli tra le possibili somme tra numeri di Ulam, cioè una definizione nella forma:

```
allSums (x:xs) = map (x+) xs : allSums xs
ulams = 1:2: f (allSums ulams)
```

per una certa funzione `f`. Ovviamente, `f` può usare altre funzioni ausiliarie. Una soluzione più che accettabile può essere usare `diags` e lavorare sulle liste finite di candidati Ulam generate da un nuovo Ulam number (vedi Lezione **13**) ottenendo un programma dal comportamento analogo a quello generativo.

Se qualcuno riuscisse a trovare un genuino generatore circolare che non considera liste finite... beh, chapeaux: io non ci sono mai riuscito!

Esercizio 5★★ (Partizionare l'Infinito o anche Le Partizioni di Tutti i Numeri) Trovare una regola per generare le partizioni di un numero (Homework **1**) seguendo l'ordine lessicografico ad esempio definendo una funzione `nextPart` che data una partizione trova la successiva. Ad esempio, `nextPart [1,1,1,2] = [1,1,3]` e `nextPart [1,1,3] = [1,2,2]` (esempio ambientato nelle partizioni di 5).

Io ho trovato impegnativo capire come riconoscere e trattare correttamente il caso `nextPart [1,5] = [2,2,2]`, fenomeno che si origina la prima volta nelle partizioni del 6. Ma forse voi seguite ragionamenti diversi e schemi generativi diversi e questo problema non vi tocca (o ne riscontrate altri).

Immaginate ora di scrivere le partizioni rovesciate (quindi quelle di 5 saranno [1,1,1,1,1], [2,1,1,1], [3,1,1], [2,2,1], [4,1], [3,2], [5] nell'ordine lessicografico, ma rovesciate) e completatele con una sequenza infinita di 1. La prima è *ones*, lo stream infinito definito da $ones = 1 : ones$, la seconda è 2:*ones*, la terza è 3:*ones*, la quarta 2:2:*ones* e così via: dovrebbe essere chiaro che applicando iterativamente la regola di `nextPart` a partire dallo stream *ones*, si genera uno stream di streams, di cui i prefissi opportuni dei primi $part(n)$ streams sono le partizioni di n (dove $part(n)$ qui è il numero delle partizioni di n). Vedere la figura allegata nel Classroom.

Ma ciò è vero per ogni n! D'altra parte, ogni stream può essere visto come una distinta partizione del numero naturale infinito ω. E direi che ci sono tutte quelle che (in un certo senso) possono essere considerate le partizioni di ω, almeno quelle che hanno un numero finito di numeri maggiori di 1.

Voi dovete:

1. Generare lo stream sopra descritto (chiamiamolo `allPartitions`)

2. Dare anche una funzione `partFin: Int -> [[Int]] -> [[Int]]` che ottiene le partizioni di un numero n, prendendo gli opportuni prefissi dei primi `part(n)` streams in `allPartitions`.

Osservo che il punto 2. si pu fare senza fare il punto 1. provando la funzione `partFin` sullo stream: `map (++ones) (parts n) ++ [m:ones | m<-[n+1..]]` dove `parts` è la funzione che genera le partizioni che avete definito nell'Homework **1** ed `n` è lo stesso input di `partFin`.