Linguaggi di Programmazione

Ivano Salvo

FP7: Perle di Laziness

Corso di Laurea in **Matematica**

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 8, 17 novembre 2020

Lezione 8a:
Perle di Laziness 1:
Hamming numbers

Hamming Numbers (1)

Problema: generare tutti i composti di 2, 3 e 5 in ordine crescente.

Formalmente: $H = \{ 2^i \cdot 3^j \cdot 5^k \mid i, j, k \geq 0 \}$

Cominciamo con una soluzione che filtra tutti i numeri che non sono composti di 2, 3 e 5 (in verità **generalizziamo a un qualsiasi insieme di generatori**).

Questa soluzione è estremamente inefficiente, perché gli Hamming numbers hanno **densità 0**! ($\lim_{n\to\infty} H_n/n = 0$, dove H_n è il numero di elementi di una successione minori di n).

```
 -- funzione che controlla se un numero n è composto
 -- solo dei numeri in una lista gs
compositeOf    _    1   = true
compositeOf   []    _   = false
compositeOf gs@(g:tgs) n
   | n `mod` g == 0   = compositeOf gs (n `div` g)
   | otherwise        = compositeOf tgs n

hamming gs = filter (compositOf gs) [1..]
```

Hamming Numbers (2)

Pensiamo a una definizione induttiva dell'insieme H degli Hamming numbers. Vediamo il caso di `soli' 2 numeri p ed q:

$$H = \{1\} \cup \{p \cdot n \mid n \in H\} \cup \{q \cdot n \mid n \in H\}$$

che è una **proprietà di chiusura** e praticamente si può scrivere in Haskell, ricordando che lavoriamo con liste ordinate, e vogliamo **evitare duplicati** e **produrre una lista** ordinata.

Osservate che si consuma al più un elemento per ogni lista per produrre un nuovo elemento della lista risultato!

```
 -- definiamo prima union (merge senza duplicati)
union xs@(x:txs) ys@(y:tys)
      | x < y       = x:union txs ys
      | x > y       = y:union xs tys
      | otherwise   = x:union txs tys

hamming p q = hs where
   hs = 1:union (map (p*) hs)(map (q*) hs)
```

Hamming Numbers (3)

Problema: Proviamo a generare tutti i composti di un insieme G di generatori. Come prima avremo:

$$H = \{1\} \cup \bigcup_{g \in G} \{g \cdot n \mid n \in H\}$$

che non è diverso da prima, ricordando che possiamo `foldare' la funzione `union` su una lista di liste e `mappare' l'operazione `(g*)` dentro la stessa lista di liste…

Nota: `foldr1` evita di dover dare un valore iniziale.

Attenzione invece che `foldl` non termina mai su stream!

```
union xs@(x:txs) ys@(y:tys)
      | x < y       = x:union txs ys
      | x > y       = y:union xs tys
      | otherwise   = x:union txs tys

hamming gs = hs where
   hs = 1:foldr union [] (map (\g->map (g*) hs) gs)
-- hs = 1:foldr1 union (map (\x->map (x*) hs) gs)
```

inutile perché non si arriva mai al fondo dello stream

foldr1

`foldr1` evita di dover scegliere un valore iniziale (quando non necessario).

Ovviamente, quando foldiamo una funzione dentro uno stream infinito non c'è mai bisogno di un valore `iniziale' su lista vuota.

È comoda in funzioni come min o max su una lista, che hanno solo valori come +∞ o -∞ come valori iniziali `naturali'.

```
 -- foldr1 # [x1,...,xn]=[x1#x2#...#xn]
myFoldr1 f [x] = x
myFoldr1 f (x:xs) = f x (myFoldr1 f xs)

 -- attenzione:
sum'= foldr1 (+)
 -- ma questa dà errore e non 0!
sum' []
 -- min x y = if x<y then x else y
minList = foldr1 min
maxList = foldr1 max
```

Hamming Numbers (4)

Per un matematico dovrebbe essere intuitivo ragionare in termini di equazioni ricorsive, ma se **c'è un procedimento più artigianale.**

Immaginando di avere la lista risultato, il *prossimo numero sarà il minimo delle teste delle liste ottenute mappando (p*) nella lista risultato.*

Immaginiamo che 6 sia l'ultimo elemento inserito in `hs`:

```
hs           = 1   2   3   4   5   6 |  8   9  10  12  15
map (2*) hs  = 2   4   6   8  10  12 | 16  18  20  24  30
map (3*) hs  = 3   6   9  12  15  18 | 24  27  30  36  45
map (5*) hs  = 5  10  15  20  25  30 | 40  45  50  60  75
                                     |
                 input pronto        | input non disponibile
```

```
hamming gs = hs where
   hs = 1:nextH map (\x->map (x*) hs) gs

   nextH xs = m:nextH ys where
      m  = foldr1 min (map head xs)
      ys = map (\zs@(z:tzs)->
                if z==m then tzs else zs) xs
```

generiamo le |gs| liste dalla lista risultato hs

calcoliamo il minimo delle teste

rimuoviamo il minimo

Hamming Numbers (5)

Gli eleganti programmi precedenti **generano più volte gli stessi numeri** (rimossi dalla `union`). È possibile fare meglio?

Ripensiamo le equazioni ricorsive degli Hamming numbers, **assumendo che i generatori *G* siano primi tra loro.**

Sia $G=\{g\} \cup G'$, l'insieme $H(G)$ contiene tutte le potenze di g moltiplicate per tutti gli $H(G')$. Questo **genera tutti gli elementi una sola volta**, perché gli $H(G')$ non contengono g come fattore.

Attenzione! `allProduct xs ys = [x*y | x<-xs, y<-ys]` qui non funzionerebbe! Non li genera in ordine!

L'ordine è fondamentale, vedi trucco per `unionP`.

```
hamming gs = 1:hamming' gs where
   hamming' (g:gs) = ps `unionP` hs `unionP` allProducts ps hs
   where   ps = g : map (g*) ps -- potenze di g
           hs = hamming' gs -- chiamata ricorsiva
allProducts (x:xs) zs = map (x*) zs `unionP` allProducts xs zs
 -- occorre un trucco per rendere produttiva union
 -- so che il minore è il primo della prima lista… gs ordinata
unionP (x:xs) ys = x:union xs ys
```

Hamming Numbers (6)

Vediamo un'altra versione simile, ma migliore. Da:

$$H(G) = g \cdot H(G) \cup H(G \setminus \{g\})$$

da cui, considerando $H'(G) = H(G) \setminus \{1\}$ ho:

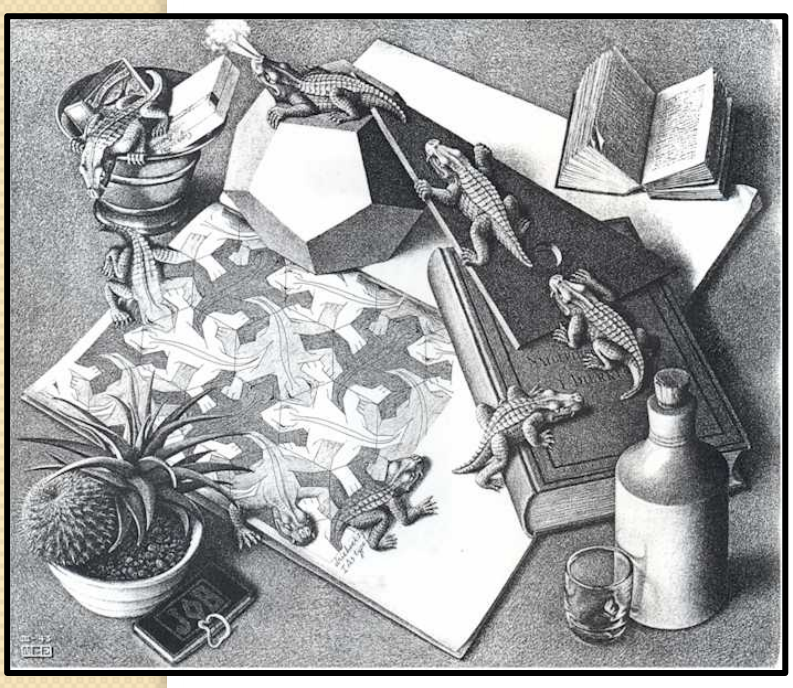
$$H'(G) = \{g\} \cup g \cdot H'(G) \cup H'(G \setminus \{g\})$$

che **sono tutti disgiunti**… (dimostrare!).

Anche questa equazione si traduce immediatamente in Haskell:

```
-- possiamo semplificare union visto che gli stream
-- sono disgiunti: migliora un po' l'efficienza
dUnion xs@(x:txs) ys@(y:tys) =
   if x<y then x:dUnion txs ys
          else y:dUnion xs tys
--
hamming gs = 1:hamming' gs where
   hamming' [] = []
   hamming' (g:gs) = hs where
      hs = g : map (g*) hs `dUnion` hamming gs
```

Figura Sfondo: primi e composti



Gli Hammings funzionano **anche per una lista infinita di generatori**.

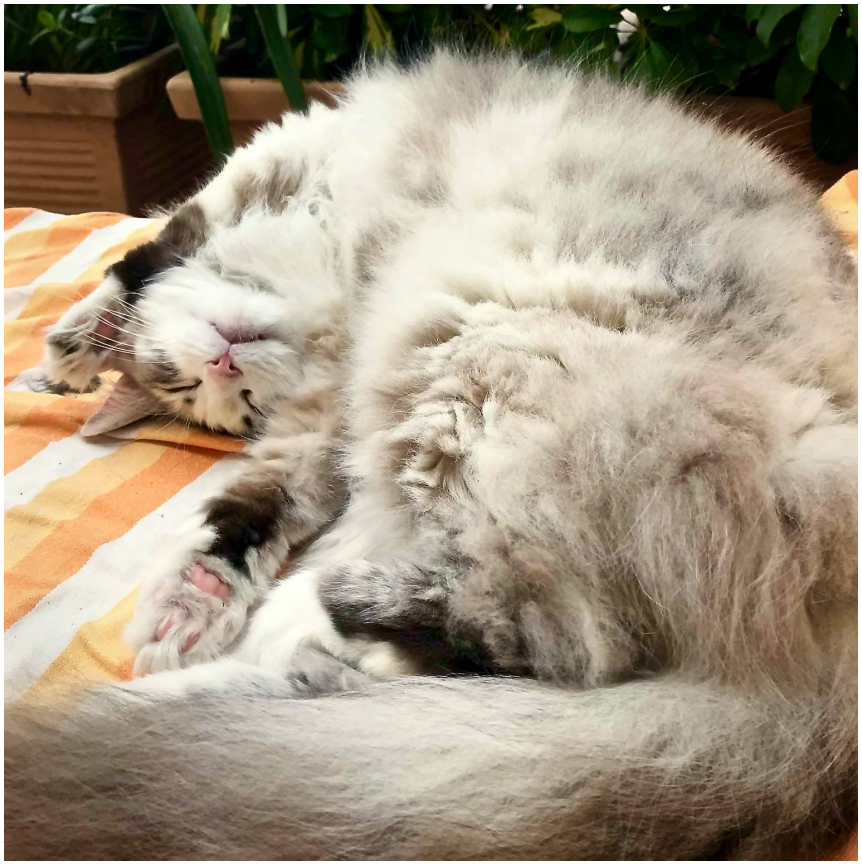
Idea: Calcolare i **primi** come i Naturali meno gli **hamming dei primi** (che sono ovviamente i composti).

Devo **evitare di produrre tra gli Hamming i generatori stessi**: comincio da `(p*p)`: tuttavia questi mi servono per generare altri composti e li **devo reintrodurre con** `ps `dUnion cmpts`

È una sorta di **Crivello di Eulero**.

```
primes = 2:([3..] `minus` composites primes) where
   composites (p:ps) = cmpts where
      cmpts = (p*p):map (p*) (ps `dUnion` cmpts)
                  `union` (composites ps)
```

Lezione 8b:
Perle di Laziness 2
Ulam numbers



Ulam Numbers: Defin. e CodeGolf

Problema: I **numeri di Ulam** $\{u_n\}_{n \in \mathbb{N}}$ sono definiti come segue:

u_1=1, u_2=2, mentre u_{n+1} è il minimo numero che si scrive in **modo unico** come somma di due elementi in $\{u_1, \ldots, u_n\}$.

La sequenza generata è: 1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, 28, 36, …

Ad esempio, 5 **non è un Ulam number** perché 5 = 1+4 = 2+3 e i numeri 1, 2, 3, 4 sono tutti Ulam numbers.

Cominciamo con un virtuosismo: esiste un programma Haskell di soli 67 caratteri, dalla **codeGolf community**:

Haskell, ~~70~~ 67 characters

```
u n=take n$1:2:[x|x<-[1..],[_]<-[[y|y<-u$n-1,z<-u$n-1,y<z,y+z==x]]]
```

Questo programma seleziona gli Ulam numbers tra tutti i naturali, scegliendo quegli x tali che la lista delle somme x=y+z ha lunghezza 1, con y e z presi dagli Ulam già calcolati.

Ulam Numbers: from OEIS

Andando sulla pagina dell'OEIS (On-line Encyclopedia of Integer Sequences) troviamo un altro oscuro programma per generare gli Ulam numbers:

```
(Haskell)
a002858 n = a002858_list !! (n-1)
a002858_list = 1 : 2 : ulam 2 2 a002858_list
ulam :: Int -> Integer -> [Integer] -> [Integer]
ulam n u us = u' : ulam (n + 1) u' us where
   u' = f 0 (u+1) us'
   f 2 z _                          = f 0 (z + 1) us'
   f e z (v:vs) | z - v <= v        = if e == 1 then z else f 0 (z + 1) us'
                | z - v `elem` us'  = f (e + 1) z vs
                | otherwise         = f e z vs
   us' = take n us
-- Reinhard Zumkeller, Nov 03 2011
```

che filtra i naturali, contando per ogni *n* il numero di tutte le somme di Ulam number precedenti che danno *n* come risultato, e eleggendo n come prossimo Ulam number se e solo se questo numero è 1.

Cerchiamo soluzioni **più eleganti** e **più efficienti**.

Amarcord: Informatica Generale

A Informatica Generale (vedi slides Lezione **13**) avevamo visto come avendo i primi n Ulam numbers in ordine crescente (come è naturale avere) e un candidato $m > u_n$, potevamo contare in **tempo lineare** in n quante coppie $u_i + u_j = m$.

Infatti, sapendo che $u_i + u_j < m$ **possiamo escludere** tutte le somme del tipo $u_i + u_k$ con $k < j$, perché essendo gli elementi ordinati in ordine crescente, $u_i + u_k < u_i + u_j < m$.

Analogamente, sapendo che $u_i + u_j > m$ **possiamo escludere** tutte le somme del tipo $u_k + u_j$ con $k > i$, perché essendo gli elementi ordinati in ordine crescente, $u_k + u_i > u_i + u_j > m$.

Ma come implementare questa idea in Haskell dove **non abbiamo né vettori**, **né liste doppiamente concatenate** che si possono scorrere sia da sinistra a destra che da destra a sinistra?

Idea: rovesciare la lista degli Ulam

Avendo lo **stream `us` degli Ulam in costruzione** e la **lista rovesciata degli Ulam generati finora `rus`**, possiamo simulare il precedete procedimento: **avanzare su `us` significa andare avanti**, mentre **avanzare su `rus` significa andare indietro**.

Definiamo una funzione `isUlam` che segue questo principio avalendosi di una funzione ausiliaria `countSums m us rs`, sotto la precondizione che `rs` sia il reverse di `us`.

```
isUlam m us = countSums m us (reverse us)==1 where
   countSums m us@(u:tus) rs@(r:trs)
  -- us contiene i primi n<m ulam numbers
  -- rs = reverse us
      | u >= r    = 0 -- poi genero somme simmetriche
      | m == s    = 1 + countSums m tus trs
      | m  < s    = countSums m us trs
                -- avanzo su rs
      | otherwise = countSums m tus rs where
                -- avanzo su us
   s = u + r
```

Tentazioni & soluzioni

A questo punto, uno sarebbe tentato di scrivere:

```
ulams = 1:2:[x | x<-[3..], isUlam x ulams]
```

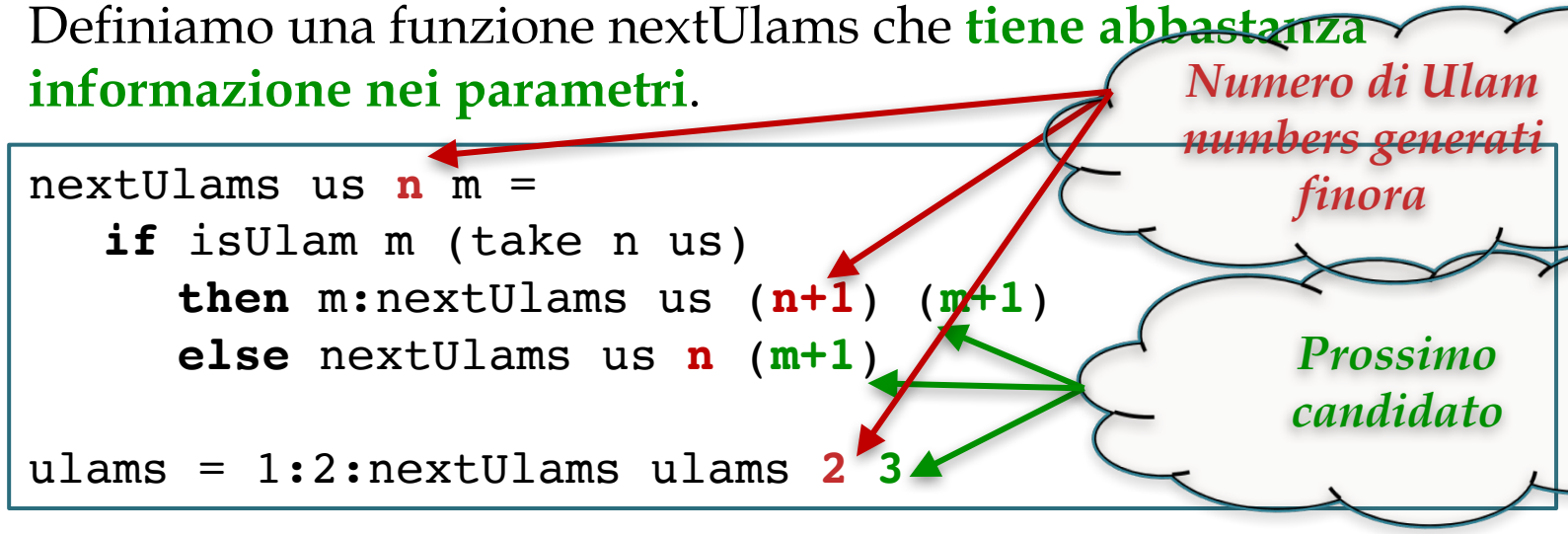
Purtroppo, `isUlam` **vuole una lista finita** e si potrebbe provare:

```
ulams = 1:2:[x | x<-[3..],
          isUlam x (takeWhile (<x) ulams)]
```

ma purtroppo `takeWhile` **si arresta quando trova un numero maggiore o uguale a x** e questo non succede perché x è maggiore di tutti gli Ulam generati finora.

Occorre conoscere il numero degli ulam numbers già generati.

Definiamo una funzione nextUlams che **tiene abbastanza informazione nei parametri**.



Raffinamenti

La soluzione precedente ha numerose inefficienze:

- deve sempre **estrarre i primi *n* elementi** dallo stream `ulams`,
- ad ogni iterazione **deve rovesciarli,**
- la funzione `countSums` conta il numero di somme, ma **arrivati a 2 potrebbe arrestarsi** decretando che il numero in questione non è un numero di Ulam.

Possiamo evitare tutto questo **aggiungendo informazioni sui parametri**!

- Trasmettiamo **`in avanti' il numero di somme già trovate**,
- **manteniamo una lista rovesciata** degli Ulam già trovati,
- Non occorre estrarre i primi n elementi dallo stream degli ulam perché la condizione `r<=u` **verrà soddisfatta prima di andare oltre l'ultimo Ulam già calcolato**.

Raffinamenti

```
nextUlams n us rs
  | isUlam n 0 us rs = n: nextUlams (n+1) us (n:rs)
  | otherwise        = nextUlams (n+1) us rs
 where
   isUlam n 2 _ _ = False
   isUlam n k us@(u:tus) rs@(r:trs) =
     | r<=u      = k == 1
     | s==n      = isUlam n (k+1) tus trs
     | s <n      = isUlam n   k   tus rs
     | otherwise = isUlam n   k   us trs
    where s = u + r

ulams = 1:2:nextUlams 3 ulams [2,1]
```

Generatori di Ulam Numbers

Abbiamo seguito l'idea tipica dei crivelli per generare i numeri primi: filtriamo i numeri di Ulam partendo dai naturali.

Ma i numeri di Ulam sono necessariamente somme di numeri di Ulam precedenti, mentre **molti numeri non sono somma di alcuna coppia di Ulam numbers**, essi formano la successione:

$$v = 23, 25, 33, 35, 43, 45, 67, 92, 94, 96, \ldots$$

(asintoticamente sono **molti di più degli Ulam numbers** $u_n \approx 13.5n$ mentre $v_n \approx 2.5n$).

Avendo uno stream infinito, possiamo facilmente generare lo stream di streams contenente le somma del primo elemento con i successivi, le somme del secondo con tutti i successivi etc.

Ma come usarlo?

```
-- allSums :: Num a => [a]->[[a]]
allSums (x:xs) = map (x+) xs:allSums xs
```

Generatori circolari?

ulam	**1**	**2**	**3**	**4**	**6**	**8**	11	13	16	18	...
(1+)	**3**	**4**	**5**	**7**	**9**	12	14	17	19	27	...
(2+)	**5**	**6**	**8**	**10**	13	15	18	20	28	30	...
(3+)	**7**	**9**	**11**	14	16	19	21	29	31	39	...
(4+)	**10**	**12**	15	17	20	22	30	32	40	42	...
(6+)	**14**	17	19	22	24	32	34	42	44	53	...
(8+)	19	21	24	26	34	36	44	46	55	56	...
(11+)	24	27	29	37	39	47	49	58	59	64	...
(13+)	29	31	39	41	49	51	60	61	66	70	...
(16+)	34	42	44	52	54	63	64	69	73	78	...
	...										

Se avessimo gli Ulam numbers già pronti, questo sarebbe lo stream di stream. Un'idea potrebbe essere quella di **fondere questi stream** (ad es. qualcosa del tipo `foldr1 merge`) e scegliere solo i numeri che appaiono 1 volta solta.

Purtroppo, se ho appena generato l'**8**, scarto facilmente il 9 e il 10 (ho già due occorrenze), ma non sono mai nella condizione di dire che l'**11** sia il prossimo perché la merge vorrebbe andare a `vedere' cosa c'è dopo il **9** e il **10**… **ma non è ancora disponibile.**

Soluzione Generativa Artigianale

Noi sappiamo che quell'**informazione è irrilevante**, ma la funzione merge no. In realtà sappiamo che per generare u_{n+1} è sufficiente conoscere tutte le somme u_i+u_j con $1 \leq i < j \leq n$.

Possiamo ragionare al finito, ovviamente, sommando il nuovo numero coi precedenti.

Manteniamo un **multiinsieme** (ordinato) di candidati, e scriviamo un generatore che:

1. sceglie il prossimo Ulam number come il primo con molteciplicità 1,
2. riaggiorna il multiinsieme di candidati facendo l'unione tra i candidati rimasti con le nuove somme generate dal nuovo Ulam number.

Rappresentiamo il multiinsieme semplicemente come un insieme di coppie (x, n) dove n è la molteciplicità di x.

Generativo finitario

```
-- trasforma una lista in un multiinsieme
toMS = map (\x->(x,1))

-- unione tra multiinsiemi
unionMS xs@((x,m):txs) ys@((y,n):tys)
   | x < y     = (x,m):unionMS txs ys
   | y < x     = (y,n):unionMS xs tys
   | otherwise = (x,m+n):unionMS txs tys
unionMS [] xs  = xs
unionMS xs []  = xs

-- generatore, tiene multiinsieme candidati cs
nextUlams cs us = u:nextUlams cs' us where
   (u,_)@tcs = dropWhile (\(x,n)->n!=1) cs
   newSums = toMS (map (u+) (takeWhile (<u) us))
   cs' = unionMS tcs newSums

ulams = 1:2:nextUlams [(3,1)] ulams
```

Soluzione Generativa Circolare

Ci sono vari modi per `aggirare' i problemi di produttività lavorando sullo stream di stream di somme, ma ovviamente sarebbe bello una soluzione elegante in cui i flussi di dati si `sincronizzino' senza trucchi.

Una possibilità è consumare lo stream di stream a diagonali…

Ovviamente, le **diagonali**, altri non sono che la **somma tra il nuovo ulam number** generato e i suoi **precedenti** ☺

```
-- genera le diagonali di uno stream di streams
diags ((x:xs):xss) = [x]:zipWith (:) xs (diags xss)

ulams = 1:2: map (\(x,_)->x) (filter (\(_,y)->y==1))
   (foldr1 unionMS (map toMS (diags allSums ulams)))
```

Lezione 8c:
Perle di Laziness 3
Primi, primo amore

The `unfaithful' sieve of Eratosthenes

Trattasi di un programma molto famoso. Tutt'oggi sulla Home page ufficiale di Haskell (`https://www.haskell.org`).

È dovuto a **Turner** (**1975**) ed ha influenzato la progettazione stessa di Haskell (laziness, list comprehension etc.).

Personalmente, fatico a considerare `crivello di Eratostene' qualcosa che usa il `mod`. Cmq, lo **stream risultato passa successivi filtri per ogni nuovo primo trovato**, come nell'originale greco.

```
primes = filterP [2..] where
  filterP (p:ps) = p:filterP [x | x<-xs, x `mod` p /= 0]
```

Trial Division

È sicuramente l'algoritmo funzionale **più efficiente** tra quelli elementari basati sul **filtraggio**.

L'idea consiste nel **testare la divisibilità di ogni naturale per i primi già trovati**. In Haskell, tutto ciò è estremamente semplice e non richiede particolari progettazioni di strutture dati, oppure uso di indici per scorrere array etc.

Osservate come lo stream primes viene usato per andare a prendere i divisori da testare…

```
-- definizione del funzionale predefinito all
myAll f (x:xs) = if f x then myAll f xs else False
myAll f []     = True
--
primes = 2:[x | x<-[3..], isPrime x] where
   isPrime x = all (\p-> x `mod` p /= 0) (factorsToTry x)
   factorsToTry x = takeWhile (\p->p*p <= x) primes
```

Crivello di Eulero naïve

È un programma **estremamente inefficiente** (a causa di un esagerato nesting di chiamate ricorsive). Tuttavia esprime in modo estremamente naturale l'idea di eliminare **una sola volta** ciascun composto, a causa del **suo minimo divisore primo**.

La funzione `eulerSieve` si applica all'insieme dei naturali **ancora potenziali primi** a cui vengono `tolti' tutti i multipli del nuovo primo trovato (non multipli di primi precedenti).

```
-- differenza di insiemi come liste ordinate
-- si assume ys 'contenuta' in xs
minus xs@(x:txs) ys@(y:tys)
  | x==y      = minus txs tys
  | otherwise = x:minus txs ys
-- ss è la lista dei sopravvissuti...
primes = eulerSieve [2..] where
  eulerSieve ss@(p:tss) =
     p : eulerSieve tss `minus` (map (p*) ss)
```

Crivello di Eratostene di Richard Bird

Ritroviamo l'idea del generatore di primi basato sul vedere i **primi come sfondo dei composti**: qui però i composti, in accordo con l'algoritmo di Eratostene, sono visti come:

$$composites = \bigcup_{p \in primes} p \cdot \mathbb{N}_{\geq p}$$

In questo caso, ovviamente, faccio **l'unione di insiemi con intersezione non vuota** in accordo col crivello di Eratostene, dove molti numeri vengono cancellati più volte (ad esempio il 30, come multiplo di 2, 3 e 5).

È un programma estremamente efficiente rispetto a quelli visti finora, anche se, lavorando con streams, memoria e tempo dedicato alla fusione di stream sono molto onerosi rispetto alle versioni imperative.

```
primes = 2:([3..] `minus` cmps where
  cmps = foldr1 unionP [multiples p | p<-primes]
  multiples p = map (p*) [p..]
  unionP (x:xs) ys = x : union xs ys
```

Crivello di Eulero

Questo programma ricalca la struttura del crivello di Eratostene di Richard Bird, ma **caratterizza i composti come l'unione di insiemi disgiunti**: i multipli di 2 (e di primi maggiori di 2), i multipli di 3 (e primi maggiori di 3) etc.

Si basa sulla definizione induttiva dei numeri S_k **sopravvissuti** e dei numeri E_k **cancellati** alla k-esima passata del crivello di Eulero.

$$S_0 = \mathbb{N}_{\geq 2} \qquad E_0 = \varnothing \qquad S_{k+1} = S_k \setminus E_{k+1} \qquad E_{k+1} = p_{k+1} \cdot S_k \,|^{k+1}$$

Ad esempio, E_1 sono i pari maggiori di 2 e S_1 i dispari maggiori uguali di 3, E_2 sono i dispari moltiplicati per 3 e quindi S_2 non contiene numeri divisibili per 2 e 3. Alla fine:

$$primes = \mathbb{N}_{\geq 2} \setminus \bigcup_{k \in \mathbb{N}} E_k$$

Osserviamo che `eulerSieve` **calcolava**: $primes = \bigcap_{k \in \mathbb{N}} S_k$

```
primes = 2:([3..] `sMinus` (cmps primes [2..])) where
 cmps (p:ps) ss@(s:tss) = es `unionP` cmps ps ss' where
       es = map (p*) ss     -- i nuovi cancellati
       ss'= tss `minus` es -- i nuovi sopavvissuti
```

Lezione 8

That's all Folks…

Grazie per l'attenzione…

…Domande?