Tecniche di Programmazione Funzionale e Imperativa

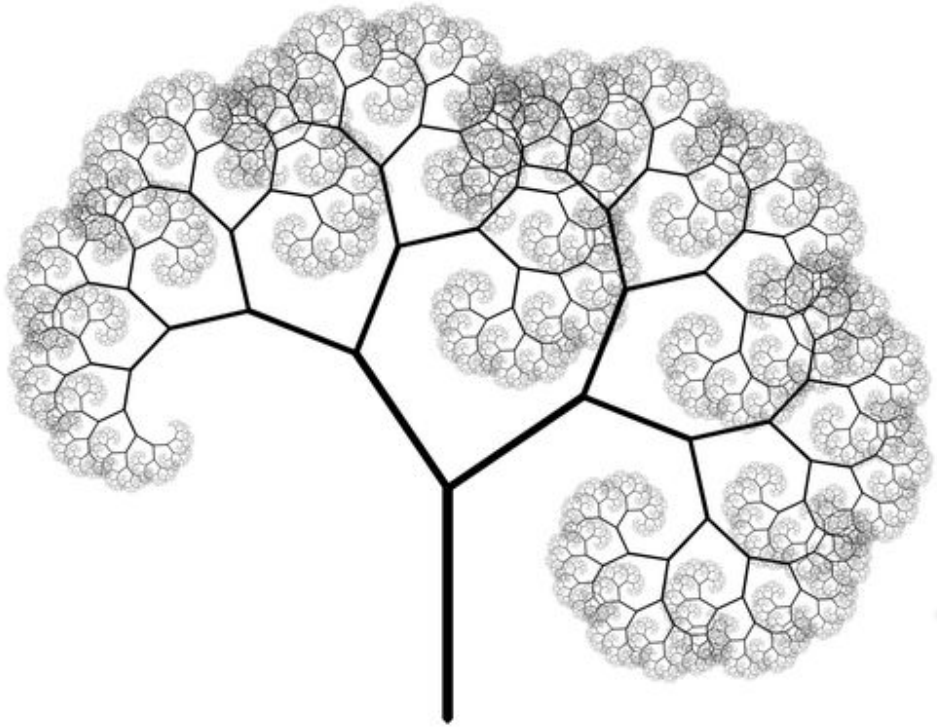
Ivano Salvo

Prime Perle (di Programmazione Funzionale)

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **10**, 25 marzo 2022



*Lezione 10**

Antipasto con alberi ennari

Esempio: per fare un albero ennario...

Facciamo conoscenza con gli alberi a branching arbitrario.

Ciascun nodo ha una label e una **lista di sottoalberi**.

Poniamoci il problema di generare la **lista** di **tutte le labels**: si può scrivere questo programma in modo molto naturale e semplice, ma la sua complessità è ancora una volta più che lineare.

⏵ **Esercizio**: scrivere la complessità nel caso semplice che si tratti un albero k-perfetto (ogni nodo non foglia ha esattamente k figli) di altezza h. In questo caso l'albero ha $n = \Theta(k^h)$ nodi.

Si può fare meglio accumulando nei parametri? Ragioniamo in analogia a **`reverse`**.

La funzione generalizza in due modi: un secondo parametro lista, e il primo una lista di alberi: `labelsAP [Tree a] -> [a] -> [a]`

e dovrà soddisfare: `labelsAP fs xs = concat (map labels fs) ++ xs`

```
Tree = N a [Tree a]

labels :: Tree a -> [a]
labels (N x fs) = x : concat (map labels xs)
```

Esempio: per fare un albero ennario...

Il caso base della funzione è evidente (vedi riquadro).

Vediamo di **derivare** il caso induttivo:

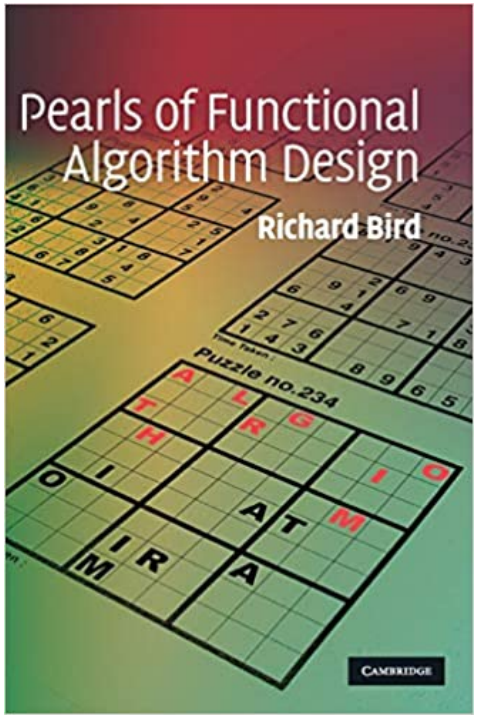
```
labelsAP (N x us:vs) xs
   = { def di labelAP }
concat (map labels (N x us:vs)) ++ xs
   = { def varie (labels, concat, etc. }
labels (N x us) ++ concat (map labels vs) ++ xs
   = { def. di labelsAP }
x:concat (map labels us) ++ labelsAP vs xs
   = { def. di labelsAP }
x : labelsAP us (labelsAP vs xs)
```

Esercizio: Dimostrare che questa funzione è $\theta(k^h)$ nelle ipotesi dell'esercizio precedente.

```
labelsAP [] xs = xs
labelsAP (N x us:vs) xs = x:labelsAP us (labelsAP vs xs)

labels t = labelsAP t []
```

tratta da:



Lezione 10a

Minimo intero libero

Il problema

Sia data una struttura **dati lineare** (ad esempio un array, ma al solito in Haskell si lavora con **liste**) che contiene un insieme di valori di un tipo **discreto ordinato**.

Qual è il **primo valore mancante**?

Assumeremo che gli elementi della lista siano semplicemente degli **interi positivi**.

Per una soluzione "divide et impera" sarà anche necessario assumere che gli interi siano **distinti**.

In Haskell (ma anche in un linguaggio imperativo), possiamo dare **immediate soluzioni quadratiche**.

Applicazioni: immaginate di avere una tabella con le celle (o pagine) di memoria occupate. Volete sapere la prima libera. Le celle occupate appaiono in ordine sparso (esempio: l'ordine con cui sono state allocate)

Facili soluzioni quadratiche (2021)

Definendo una funzione: `belongsTo` che verifica se un valore è presente in una lista…

… è sufficiente trovare il primo elemento che di *xs* che non appartiene alla lista `[0..length xs]`: per il **principio dei buchi di piccionaia** necessariamente dev'essere che il primo intero libero in questa lista.

Ma **cosa calcolano** queste funzioni **se fosse 0** il primo mancante?

```
 -- verifica se un elemento sta in una lista
belongsTo :: (Eq a) => a -> [a] -> Bool
belongsTo x [] = False
belongsTo x (y:ys) =
          if x==y then True else belongsTo x ys

-- uso il funzionale filter predefinito
minFree xs = head (filter
      (not . flip belongsTo xs) [0..(length xs)])

-- ma anche per list comprehension
minFree xs = head
   [x | x<-[0..(length xs)], not (belongsTo x xs)]
```

Facili soluzioni quadratiche (2022)

Ridefinisco **`belongsTo`** usando **`filter`**… ma a ben vedere mi è più utile il suo negato, **`notIn`**.

… è sufficiente trovare il primo elemento che di *xs* che non appartiene alla lista `[0..length xs]`: per il **principio dei buchi di piccionaia** necessariamente dev'essere che il primo intero libero in questa lista.

Ma **cosa calcolano** queste funzioni **se fosse 0** il primo mancante?

```
 -- verifica se un elemento sta in una lista
belongsTo :: (Eq a) => a -> [a] -> Bool
belongsTo = \x -> not . null . filter (==x)

-- uso il funzionale filter predefinito
minFree xs = head (filter
      (not . flip belongsTo xs) [0..(length xs)])

-- ma anche per list comprehension
minFree xs = head
   [x | x<-[0..(length xs)], not (belongsTo x xs)]
```

Soluzioni n log n

Un'ovvia soluzione per migliorare la complessità è quella di **ordinare** la lista in tempo ottimo (cioè $\mathcal{O}(n \log n)$) e poi con una scansione linera trovare il primo "buco".

[0..] è la lista infinita di naturali (la rivedremo presto…)

Devo **aggiungere un elemento in più** per trattare il caso che la lista non contenga buchi in $[0, n)$ (il minfree è n in tal caso)

```
 -- elimina il segmento iniziale di elementi
 -- che soddisfano una condizione p
myDropWhile p [] = []
myDropWhile p (x:xs) =
   if p x then myDropWhile p xs else x:xs

 -- ordino la lista, accoppio con [0..]
 -- elimino le coppie iniziali (n,n)
 -- prendo il secondo elemento della coppia in testa
minfreeOrd xs = snd
          (head (dropWhile (uncurry (==))
           (zip (mergesort ((1+length xs):xs)) [0..])))
```

Soluzione Divide et Impera

Il problema è definire: `minfree (us ++ vs)` in termini di `minfree us` e `minfree vs`.

Idea: scimmiottare la funzione **partiziona di quickSort**: scegliere un perno `p` e dividere in due la lista: lista `us` dei minori e lista `vs` dei maggiori del perno.

Se `us` ha meno di `p` elementi, cerco il minimo intero libero in `us`, altrimenti cerco in `vs`.

Osservare che in questo ragionamento gioca un ruolo cruciale il fatto che la lista **contenga interi distinti**: altrimenti il fatto che `length us ≥ p` non implica che non ci siano ``buchi''

A differenza di quickSort, attivo **solo una chiamata ricorsiva**, e per giunta **sulla lista più corta**! Infine, il perno, **non deve necessariamente** appartenere alla lista.

Giustifichiamo più rigorosamente quest'idea.

*Algebra di ++ e *

Indicando con \\ la **differenza tra liste**, abbiamo le seguenti proprietà, **analoghe a quelle di** $\cup$ **e** $\setminus$ **tra insiemi**:

```
(as ++ bs) \\ cs = as \\ cs ++ bs \\ cs
as \\ (bs ++ cs) = as \\ bs \\ cs
(as \\ bs) \\ cs = (as \\ cs) \\ bs
```

Supponendo che `as` e `us` siano **disgiunti** (il che implica che `as \\ us = as`) e `bs` e `vs` anche siano disgiunti abbiamo inoltre:

```
(as++bs) \\ (us ++ vs) = (as \\ vs) ++ (bs \\ us)
```

Fissata una lista di interi `xs` (quella di cui vogliamo conoscere il minimo intero mancante) e un qualsiasi numero naturale `b`, possiamo considerare:

```
as = [0..b-1]
us = filter (<b) xs
```

```
bs = [b..]
vs = filter (>=b) xs.
```

Programma "specifica"

Da quanto detto, otteniamo che:

```
[0..] \\ xs = [0..b-1] \\ us ++ [b..] \\ vs
      where (us, vs) = partition b xs
```

Questo programmino Haskell **genera la lista infinita di tutti gli interi mancanti** in `xs`. A noi, interessa solo il primo e possiamo semplicemente applicare la funzione **`head`**, che gode della seguente proprietà:

```
head (xs++ys) = if (null xs)
                   then head ys
                   else head xs
```

E quindi fissato un **qualsiasi intero** **`b`**, abbiamo le equazioni nel riquadro che **sono già un programma Haskell** (o quasi):

```
minFree xs = if null ([0..b-1] \\ us)
                  then head ([b..] \\ vs)
                  else head [0..b-1] \\ us where
                            (us, vs) = partition b xs
```

verso il Divide et Impera

Ovviamente, siamo ben lontani da un programma lineare, in quanto solo la **differenza tra liste** è, in generale, **quadratica**! (si può fare lineare se le liste sono ordinate sulla falsariga di `merge` [⏵ **Esercizio**]), ma questo non è il caso di `xs`.

Prima osservazione: il test **`null [0..b-1]\\ us`** **essendo gli interi distinti**, equivale a stabilire se **`length us == b`**.

Seconda osservazione: per applicare il principio divide et impera dobbiamo risolvere ricorsivamente i sotto-casi, ma allora **ci serve una funzione `minFree` generalizzata**, con un altro parametro, in quanto ci serve sapere quale sia il **minimo intero a partire da un certo intero `a`**, non necessariamente 0.

```
minFrom :: Int -> [Int] -> Int
minFrom a xs = head ([a..] \\ xs)
```

a patto che valga la precondizione che **tutti gli interi in `xs` siano maggiori o uguali ad `a`**.

Prima approssimazione

Siamo giunti quasi alla fine come nel riquadro:

```
minFree xs = minFrom 0 xs
minFrom a xs
   | null xs             = a             -- caso base
   | length us == b - a = minFrom b vs -- cerco a dx
   | otherwise           = minFrom a us -- cerco a sx
 where (us, vs) = partition b xs
```

Ci manca solo da scegliere **b**: idealmente **vorremmo dividere a metà xs** ma **non è importante se b sia nella lista** (come invece nel caso del perno di quickSort).

Una buona scelta è **`b = a + length xs `div` 2`**: così facendo, la partizione a sinistra avrà al più **`b - a`** elementi: se ne ha **esattamente `b - a`** so che posso andare a cercare il minimo intero mancante nella **parte destra**, altrimenti ho **più che dimezzato** e la lunghezza e cerco nella **parte sinistra**.

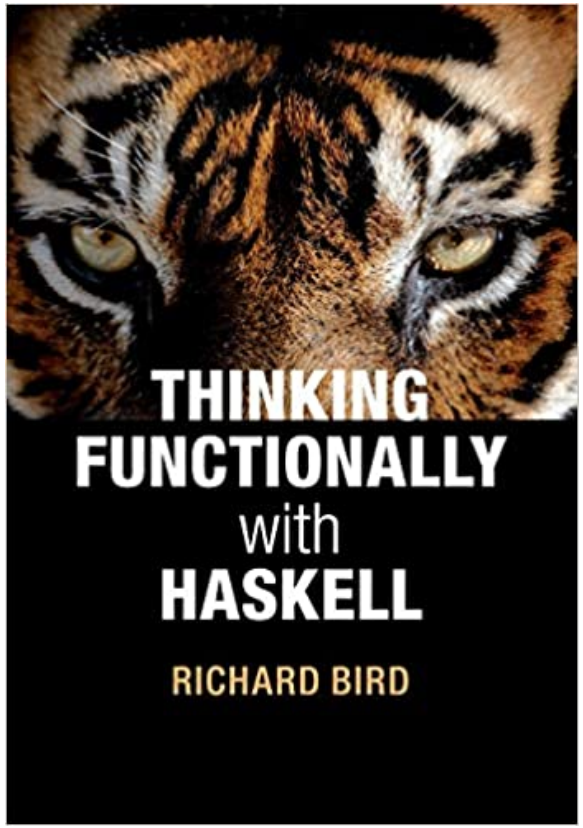
Gran Finale

Ecco il programma finale: un'ultima astuzia **evita di ricalcolare la lunghezza della lista** (vista anche su mergeSort): la calcolo all'inizio e poi ad ogni chiamata viene ricalcolanta con una semplice operazione aritmetica, passandola come parametro.

```
minFree xs = minFrom 0 xs (length xs) where
  minFrom a xs n
        | n == 0          = a
        | m == b - a      = minFrom b vs (n-m)
        | otherwise       = minFrom a us m
     where (us, vs) = partition b xs
           b = a + 1 + n div 2
        -- partition potrebbe calcolare m
           m = length us
```

La complessità $T(n) = T(n/2) + \theta(n)$ che si risolve facilmente e dà come risultato $\boldsymbol{\theta(n)}$.

soluzione tratta da:



Lezione 10b

Sotto-sequenza di somma massima

Il problema e la sua storia

Si tratta di identificare il segmento di vettore (di **elementi consecutivi**) di **somma massima**. Ovviamente il problema è interessante se il vettore contiene anche numeri negativi.

Si tratta di un problema molto noto, che fu oggetto di una mirabile *Programming Pearl* di **John Bentley** nel Journal of the ACM, raccontata come fosse una barzelletta, del tipo: ”Ci sono un ingegnere, un informatico e uno statistico…”

La storia narra numerose soluzioni:

1. naturale (e maldestra) soluzione cubica $\theta(n^3)$;
2. una naturale e facile soluzione quadratica $\theta(n^2)$;
3. una ingegnosa (ma complicata) soluzione $\theta(n \log n)$, raccontata nel **Cormen**, cap. **4** come limpido esempio di *divide et impera*;
4. e infine una facilissima (ma difficile da scoprire) soluzione lineare $\theta(n)$.

Versione Funzionale

Noi vedremo (per ora) la versione “funzionale”.

Il problema quindi riguarderà le **liste** invece che gli array.

Partiremo da una soluzione **brute-force** che ancora una volta brilla per la facilità, costruita **componendo funzioni** di “chiara semantica”.

Faremo **manipolazioni algebriche** per cercare di eliminare le sorgenti di inefficienza.

La soluzione finale sarà **corretta per costruzione** rispetto alla prima versione facile, intuitiva ma inefficiente.

brute force

Ovviamente c'è sempre una soluzione **brute-force** (molto naturale in Haskell, tra l'altro)

1. generare tutte le sotto-sequenze come una lista di liste;
2. mappare la funzione `sum` in questa lista;
3. selezionare quella di somma massima.

```
 -- idea: per ottenere tutte i segmenti,
 -- si fanno i prefissi di tutti i suffissi
segments = concat . map inits . tails
 -- ha il difetto di replicare più volte la lista vuota,
 -- ma è possibile dare definizioni alternative
 -- di inits e tails per evitarlo

 -- se voglio conoscere il valore...
mss = maximum . map sum . segments
 -- se voglio il segmento...
mss' = snd . maximum . map \x->(sum x, x) . segments
 -- sfruttando che < sulle coppie
 -- usa l'ordine lessicografico:
mss'' = snd . maximum . \xs->zip (map sum xs) xs . segments
```

Complessità brute force

Ci sono ovviamente **$\Theta(n^2)$ segmenti**:

- possiamo ragionare pensando alle **$\Theta(n^2)$ coppie inizio/fine**,
- oppure osservare che `tails` produce **$\Theta(n)$ code** e per ciascuna di esse `inits` produce **$\Theta(n)$ prefissi**.

A questo punto, `sum` è lineare e quindi costa $\Theta(n)$ per ciascun segmento. **Risultato $\Theta(n^3)$**.

Le altre operazioni (selezionare il **massimo**) sono proporzionali alla lunghezza della lista dei segmenti, cioè $\Theta(n^2)$ e sono quindi dominate dalla componente di complessità $\Theta(n^3)$.

Cosa si può migliorare?

Ovviamente molte operazioni vengono ripetute (**esempio lampante**: le sommatorie dei valori di segmenti contenuti uno dentro l'altro).

In generale, fare **`map` annidate** porta a **moltiplicare le complessità** delle funzioni mappate.

Posso percorrere due strade:

1. vedere se "entrando" nelle liste riesco a fare le stesse operazioni evitando scansioni inefficienti ripetute o, peggio, annidate;
2. cercare delle **manipolazioni** puramente **algebriche** che portino a trasformazioni del programma favorevoli.

Ovviamente ☺ **percorriamo la seconda strada**…

Massaggiamo il programma

1. Partiamo dalla definizione globale:

```
maximum . map sum . concat . map inits . tails
```

Sapendo che `map f . concat = concat . map (map f)` otteniamo:

```
maximum . concat . map (map sum). map inits . tails
```

2. Ora possiamo applicare la proprietà fondamentale di **`map`**, e cioè `map f . map g = map (f . g)`, da cui:

```
maximum . concat . map (map sum . inits) . tails
```

3. Abbiamo che **`maximum . concat`** è equivalente a **`maximum . map maximum`**, a patto che tutte le liste sia non vuote (**`maximum`** non è definita su lista vuota! – **verificare sempre precondizioni!**)

```
maximum . map (maximum . map sum . inits) . tails
```

4. Infine occorre ricordarsi che c'è una funzione che calcola i **valori di una funzione sui prefissi, `scanl`**.

```
maximum . map (maximum . scanl (+) 0) . tails
```

e finalmente otteniamo una soluzione $\Theta(n^2)$

Possiamo andare oltre?

Rassegnamoci ad "aprire le scatole": **`maximum . scanl (+) 0`** è **`foldr1 max . scanl (+) 0`**: potremmo applicare una **fusion law** per **`foldr1`** se potessimo scrivere **`scanl`** in funzione di **`foldr`**.

Ricordiamo un esempio di computazione di **`scanl`**:

```
scanl (+) 0 [x, y, z] =
       = [0, 0+x, (0+x)+y, ((0+x)+y)+z)]
       = [0, 0+x, 0+x+y, 0+x+y+z] -- + assoc
       = 0 : map (x+) [0, y, y+z]
       = 0 : map (x+) (scanl (+) [y,z])
```

Generalizzando, se **`#`** è un'operazione associativa con elemento neutro **`e`**, abbiamo l'equazione:

```
scanl (#) e = foldr f [e] where
                      f x xs = e : map (x#) xs
```

Occorre vedere **se c'è una fusion law** per **`foldr1`** e **`foldr`** nella forma (per opportune scelte di **`h`**):

```
foldr1 (@) (e:map (x#) xs) = h x (foldr1 (@) xs)
```

Verifichiamo la fusion law

```
(*) foldr1 (@) (e:map (x#) xs) = h x (foldr1 (@) xs)
```

Cominciamo con il verificare sotto quali condizioni vale se **`xs = [y]`**, applicando le facili equazioni:

```
foldr1 (@) [y] = y
map (x#) [y] = x#y
foldr1 (@) (e:xs) = e @ foldr1 xs
```

otteniamo che **`h x y = e@(x#y).`**

Risostituendo `h` in `(*)` ci rimane di far vedere sotto quali condizioni:

```
foldr1(@)(e:map (x#) xs) = e@(x#foldr1 (@) xs)
```

Semplificando, questo riduce a dimostrare:

```
foldr1(@). map (x#) = (x#).foldr1 (@)
```

che vale facilmente a patto che valga la proprietà distributiva di # rispetto a @, cioè: `x@(y#z)=(x#y)@(x#z)` [**Esercizio**].

A noi serve che (+) distribuisca su min e max, e chiaramente:

```
x + (y `min` z)) = x+y `min` x+z
x + (y `max` z)) = x+y `max` x+z
```

Gran Finale: programma lineare

Siamo quindi arrivati al seguente programma:

```
mss = maximum . map foldr (@) 0 . tails
        where x @ y = 0 `max` (x + y)
```

che sembra la specifica di **`scanl`** con **`foldl`** (vedi Lezione **6**) a meno del fatto che abbiamo **`foldr`** al posto di **`foldl`**.

Ragionando in modo analogo a quanto fatto per **`scanl`**, si trova una funzione lineare che fa questo lavoro (vedi Homework **2**), predefinita in `Prelude` che si chiama **`scanr`** che calcola tutti i valori di una funzione partendo dalle code. Quindi:

```
mss = maximum . scanr (@) 0
        where x @ y = 0 `max` (x+y)
```

che è lineare in quanto composizione di funzioni lineari!

```
scanr f e [] = [e]
scanr f e (x:xs) = f x (head ys) ys
       where ys = scanr f e xs
```

Lezione 10

That's all Folks…

Grazie per l'attenzione…

…Domande?