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 **9**, 29 marzo 2021

Lezione 8a

Minimo intero libero

Il problema

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

Qual è il primo valore mancante?

Ovviamente, 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**.

Vediamone un paio come allenamento al **pensiero funzionale**.

Facili soluzioni quadratiche

Definiamo una funzione: `belongsTo :: (Eq a)=>a->[a]->Bool` tale che `belongsTo x xs` restituisce `True` se `x` occorre nella lista `xs` e `False` altrimenti.

Ora è sufficiente trovare il primo elemento che di xs che non sta nella lista `[0..(length xs)]`: Osservate che, 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 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<-[1..(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..])))
```

Un simpatico mergesort "iterativo"

Colgo l'occasione per mostrarvi un simpatico mergesort "moralmente iterativo" in Haskell.

Esercizio: scrivete mergesort iterativo in Python/C/Java.

```
 -- applica una funzione finché non si verifica p
myUntil p f x = if p x then x else myUntil f (f x) p

-- crea una lista di singoletti e poi le fonde 2 a 2
-- finché non ne resta una sola
mergesort xs =
extract (until singleton msort (map (\x->[x]) xs))
        where msort xs = map2by2 merge xs

-- map che applica una funzione binaria a due a due…
map2by2 f (x:y:xs) = f x y:map2by2 f xs
map2by2 f    xs    = xs
-- la seconda clausola si attiva su liste di 0 e 1 el.

-- estrae un elemento da una lista
extract   [x] = x

 -- verifica se una lista ha un solo elemento
singleton [x] = True
singleton _ False
```

Soluzione Divide et Impera

Il problema è definire: `minfree (xs ++ ys)` in termini di `minfree xs` e `minfree ys`.

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 la *m* ha meno di *p* elementi, cerco il minore in *m*, altrimenti cerco in *M*.

Osservare che in questo ragionamento gioca un ruolo cruciale il fatto che la lista **contenga interi distinti**!

A differenza di Quicksort, attivo solo una chiamata ricorsiva, e per giunta sulla lista più corta! 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 `us` anche disgiunti abbiamo inoltre:

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

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]` e `bs=[b..]` e

`us=filter (<b) xs` e `vs=filter (>=b) xs`.

Programma "specifica"

Da quanto detto, otteniamo che:

```
[0..b] \\ 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)
```

con 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
       | length us == b – a = minFrom b vs
       | otherwise            = minFrom a us
   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 il 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 almeno 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** ad ogni chiamata (senza modificare la complessità asintotica) calcolandola all'inizio e poi ricalcolanda 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
           m = length us
```

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

Lezione 8c

Sottosequenza di somma massima

Il problema e la sua storia

Si tratta di identificare il sottovettore (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 $\mathcal{O}(n^3)$;
2. una naturale e facile soluzione quadratica $\mathcal{O}(n^2)$;
3. una ingegnosa (ma complicata) soluzione $\mathcal{O}(n \log n)$, raccontata nel **Cormen**, cap. **4** come limpido esempio di divide et impera;
4. e infine una facilissima (ma difficile da scoprire) soluzione lineare $\mathcal{O}(n)$.

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:

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

```
 -- idea: per ottenere tutte le sottoliste, faccio
tutti i prefissi di tutti i suffissi
segments = concat . map inits . tails
 -- ha l'unico difetto di replicare più volte la
lista vuota, ma potremmo 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 il < sulle coppie guarda prima il
primo elemento e poi il secondo, oppure:
mss''=snd.maximum.\xs->zip (map sum xs) xs.segments
```

Complessità brute force

Ci sono ovviamente **$\mathcal{O}(n^2)$ segmenti**:

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

A questo punto, `sum` è lineare e quindi costa $\mathcal{O}(n)$ per ciascun segmento.

Risultato $\mathcal{O}(n^3)$.

Le altre operazioni (selezionare il massimo) sono proporzionali alla lunghezza della lista dei segmenti, che ha lunghezza $\mathcal{O}(n^2)$ e sono quindi dominate dalla componente di complessità $\mathcal{O}(n^3)$

Cosa si può migliorare?

Ovviamente molte operazioni vengono ripetute (**esempio**: somme 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 $\mathcal{O}(n^2)$

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`.

Vediamo 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 prec.) 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, 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 8b

Minimo intero libero

Lezione 9

That's all Folks…

Grazie per l'attenzione…

…Domande?