4) La ricorsione

Sommario: Partendo dal caso specifico della ricerca binaria e da altri problemi esemplificativi, in questo capitolo si affronta la tematica degli algoritmi ricorsivi.

Consideriamo una nuova formulazione dell'algoritmo di ricerca binaria (nella quale sono volutamente tralasciati i dettagli per catturarne l'essenza):

```
Ricerca_binaria (A, v)
       se il vettore è vuoto restituisci null
       ispeziona l'elemento A[centrale]
       se esso è uguale a v restituisci il suo indice
       se v < A[centrale]
              esegui Ricerca_binaria (metà sinistra di A, v)
       se v > A[centrale]
              esegui Ricerca_binaria (metà destra di A, v)
```

L'aspetto cruciale di questa formulazione risiede nel fatto che l'algoritmo risolve il problema ***"riapplicando" se stesso su un sottoproblema*** (una delle due metà del vettore).

Questa tecnica si chiama ***ricorsione***, ed è un argomento importantissimo.

4.1 Funzioni matematiche ricorsive

Partiamo da un concetto ben noto, quello delle ***funzioni matematiche ricorsive***. Una funzione matematica è detta ricorsiva quando la sua definizione è espressa in termini di se stessa.

Un tipico esempio è la definizione del fattoriale *n!* di un numero *n*, la cui definizione è la seguente:

- *n! = n*(n – 1)! se n > 0*
- *0! = 1*

In questa definizione la prima riga determina il meccanismo di calcolo ricorsivo vero e proprio, ossia stabilisce come, conoscendo il valore della funzione per un certo numero intero, si calcola il valore della funzione per l'intero successivo.

La seconda riga della definizione invece determina un altro aspetto fondamentale, il ***caso base***: in questo caso esso indica che, per il numero 0, il valore della funzione fattoriale è 1.

Una funzione matematica ricorsiva ***deve sempre avere un caso base***, altrimenti non è possibile calcolarla poiché si riapplicherebbe all'infinito la definizione ricorsiva, senza mai iniziare a effettuare il calcolo vero e proprio.

4.2 Algoritmi ricorsivi

Nel campo degli algoritmi vi è un concetto del tutto analogo, quello degli algoritmi ricorsivi: ***un algoritmo è detto ricorsivo quando è espresso in termini di se stesso***. Analogamente, ***una funzione (o una procedura) viene detta ricorsiva quando all'interno del corpo della funzione vi è una chiamata alla funzione (o procedura) stessa***.

Un algoritmo ricorsivo ha sempre queste proprietà:

- la soluzione del problema complessivo è costruita risolvendo (ricorsivamente) uno o più sottoproblemi di dimensione minore e successivamente producendo la soluzione complessiva mediante combinazione delle soluzioni dei sottoproblemi;
- la successione dei sottoproblemi, che sono sempre più piccoli, deve sempre convergere ad un sottoproblema che costituisca un caso base (detto anche ***condizione di terminazione***), incontrato il quale la ricorsione termina.

4.2.1 Calcolo del fattoriale

Una possibile funzione (non ricorsiva) che calcoli il fattoriale è la seguente:

```
Funzione Fattoriale_Iter (n: intero non negativo)
```

```
if (n > 0)
      fatt ← 1
      for i=1 to n do
            fatt ← fatt*i
      return fatt
else return 1
```

Vediamo ora una funzione ricorsiva per il calcolo del fattoriale che riflette molto fedelmente la definizione matematica sopra vista:

```
Funzione Fattoriale_Infinita (n: intero)
      if (n = 0)
            return 1
      else return n*Fattoriale_Infinita (n - 1)
```

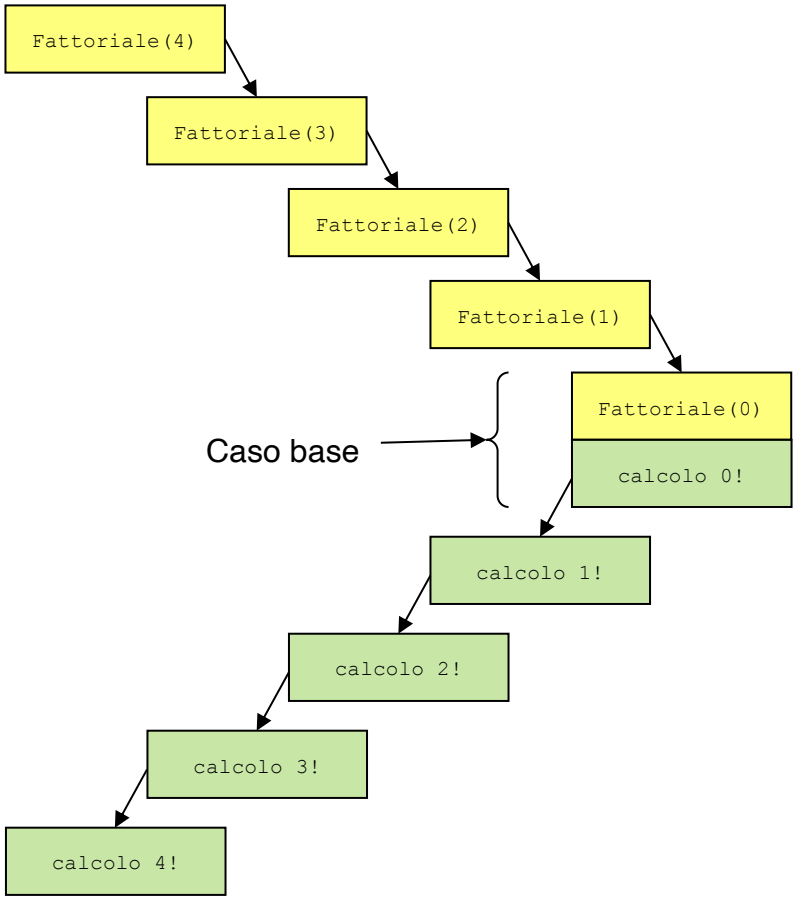
Purtroppo, la precedente formulazione della funzione non terminerebbe mai se le venisse passato come valore iniziale un intero negativo. Pertanto, dobbiamo modificare leggermente lo pseudocodice:

```
Funzione Fattoriale_Ric (n: intero non negativo)
      if (n > 0)
            return n*Fattoriale_Ric (n - 1)
      else return 1
```

In generale, è fondamentale assicurarsi che le funzioni ricorsive abbiano almeno un caso base, altrimenti la loro esecuzione genererebbe una catena illimitata di chiamate ricorsive che non terminerebbe mai (provocando ben presto la terminazione forzata dell'elaborazione a causa dell'esaurimento delle risorse di calcolo, per ragioni che vedremo fra breve).

Bisogna quindi essere assolutamente certi che il caso base non possa essere mai "saltato" durante l'esecuzione. Nulla vieta di prevedere più di un caso base, basterà assicurarsi che uno tra i casi base sia sempre e comunque incontrato.

Ma come si sviluppa il procedimento di calcolo effettivo? La figura seguente illustra sinteticamente il flusso dell'esecuzione che ha luogo a seguito della chiamata `Fattoriale_Ric(4)`:



Si noti che, nel momento in cui facciamo la prima operazione di calcolo vera e propria, il che avviene solo quando abbiamo raggiunto il caso base $n = 0$, vi è una catena di chiamate della funzione ricorsiva (caselle gialle) ancora "aperte", cioè chiamate della funzione che sono già iniziate ma non ancora terminate.

Dopo aver effettuato il calcolo del caso base, la chiamata della funzione `Fattoriale(0)` temina e innesca la catena (caselle verdi) dei successivi "ritorni" alle funzioni chiamanti, che via via effettuano il loro calcolo e terminano.

Ora, ogni funzione, sia essa ricorsiva o no, richiede per la sua esecuzione una certa quantità di memoria RAM, per:

- caricare in memoria il codice della funzione;
- passare i parametri alla funzione;
- memorizzare i valori delle variabili locali della funzione.

Quindi le funzioni ricorsive in generale hanno maggiori esigenze, in termini di memoria, delle funzioni non ricorsive (dette anche funzioni iterative). Per inciso, questa è la ragione per la quale una funzione ricorsiva mal progettata, nella quale la condizione di terminazione non si incontra mai, esaurisce con estrema rapidità la memoria disponibile con la conseguente inevitabile terminazione forzata dell'esecuzione.

Inoltre, va detto che qualsiasi problema risolvibile con un algoritmo ricorsivo può essere risolto anche con un algoritmo iterativo.

Ma allora, in quali situazioni è consigliabile utilizzare un algoritmo ricorsivo anziché uno iterativo? Queste considerazioni possono essere d'aiuto nella decisione:

- è conveniente utilizzare la ricorsione quando essa permette di formulare la soluzione del problema in un modo che è chiaro ed aderente alla natura del problema stesso, mentre la soluzione iterativa è molto più complicata o addirittura non evidente;
- non è conveniente utilizzare la ricorsione quando esiste una soluzione iterativa altrettanto semplice e chiara;
- non è conveniente utilizzare la ricorsione quando l'efficienza è un requisito primario.

La ricorsione può essere ***diretta o indiretta*** (detta anche ***mutua ricorsione***):

- si ha ricorsione diretta nel caso in cui nel corpo di una funzione vi è la chiamata alla funzione stessa (come nel caso del calcolo del fattoriale sopra visto);
- si ha ricorsione indiretta quando nel corpo di una funzione A vi è la chiamata a una funzione B ed in quello di B vi è la chiamata alla funzione A; nella ricorsione indiretta possono in generale essere coinvolte più di due funzioni: ad esempio A chiama B, B chiama C, C chiama A.

4.2.2 Ricerca sequenziale ricorsiva

Progettiamo ora un algoritmo ricorsivo per la soluzione del problema della ricerca sequenziale che abbiamo definito nel par. 3.1.

Un'idea può essere questa. Per risolvere il problema su *n* elementi:

- ispezioniamo l'*n*-esimo elemento;
- se non è l'elemento cercato risolviamo il problema - ricorsivamente - sui primi *(n – 1)* elementi.

La formulazione di questo algoritmo ricorsivo è la seguente.

```
Funzione Ricerca_sequenziale_ricorsiva(A: vettore; v: intero; n: intero)
     if (A[n] = v)
          return n
     else
          if (n = 1)
               return NULL
          else
               return Ricerca_sequenziale_ricorsiva(A, v, n - 1)
```

Notiamo che nel corpo della funzione si attiva una sola chiamata ricorsiva, quindi la successione delle chiamate segue uno schema simile a quello del par. 4.2.1.

Nel prossimo capitolo valuteremo il costo computazionale di questo algoritmo.

4.2.3 Ricerca binaria ricorsiva

Formuliamo ora più precisamente la versione ricorsiva della ricerca binaria:

```
Funzione Ricerca_binaria_ricorsiva(A: vettore; v:intero; i_min, i_max: intero)
     if (i_min > i_max) return null
     m ← ⌈(i_min+i_max)/2⌉
     if (A[m] = v)
          return m
     if (A[m] > v)
               return Ricerca_binaria_ricorsiva(A, v, i_min, m - 1)
          else
               return Ricerca_binaria_ricorsiva(A, v, m + 1, i_max)
```

La formulazione ricorsiva della ricerca binaria mantiene praticamente tutte le buone proprietà della versione iterativa. Infatti:

- anche in questo caso nel corpo della funzione si attiva una sola chiamata ricorsiva, dato che le due chiamate sono alternative l'una all'altra; quindi la successione delle chiamate segue uno schema simile a quello del par. 4.2.1;
- ogni nuova chiamata ricorsiva riceve un sottoproblema da risolvere la cui dimensione è circa la metà di quello originario quindi, come nella versione iterativa, si giunge molto rapidamente al caso base.

Da ciò deriva un costo computazionale che, come vedremo, è *O(log n)* nel caso peggiore come per la ricerca binaria iterativa.

L'unica differenza è una maggiore richiesta di memoria, dovuta al problema già descritto delle catene di chiamate ricorsive "aperte", compensata però dalla notevole chiarezza e leggibilità della soluzione.

4.2.4 Calcolo dei numeri di Fibonacci

Vediamo ora un altro esempio di algoritmo ricorsivo, il calcolo dei numeri di Fibonacci, per evidenziare un altro aspetto importante.

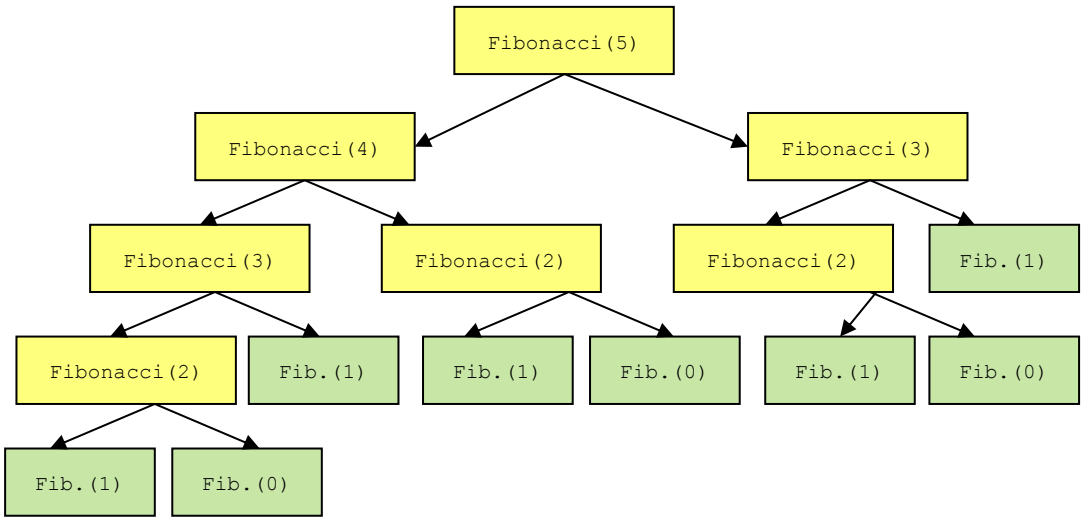
Ogni numero di Fibonacci è la somma dei due numeri di Fibonacci precedenti:

- *F(0) = 0*
- *F(1) = 1*
- *F(i) = F(i – 1) + F(i – 2) se i > 1*

Una funzione ricorsiva che li calcola è la seguente:

```
Funzione Fibonacci_ric (n: intero non negativo)
     if ((n = 0) oppure (n = 1))
          return n
     else
          return (Fibonacci_ric(n - 1) + (Fibonacci_ric(n - 2))
```

Notiamo che, nel corpo della funzione, ci sono due chiamate alla funzione stessa, non una sola come nel caso precedente. Questo si riflette in una più complessa articolazione delle chiamate di funzione, che è illustrata, per *n = 5,* nella figura seguente.



Questa figura ci consente di focalizzare due aspetti importanti.

Il primo è relativo al costo computazionale. Come vedremo più precisamente quando avremo gli strumenti (equazioni di ricorrenza) necessari, il costo computazionale è elevato: cresce esponenzialmente in funzione del valore dato in input. La ragione per cui questo accada si può intuire già ora osservando che uno stesso calcolo viene ripetuto molte volte: ad esempio, nella figura, `Fibonacci(2)` viene richiamata 3 volte e `Fibonacci(1)` 5 volte. Questo deriva da una proprietà non molto desiderabile della soluzione ricorsiva proposta: in sostanza, per risolvere il problema su *n* dobbiamo risolvere altri due problemi di costo computazionale ben poco inferiore: quello su *(n – 1)* e quello su *(n – 2)*.

Il secondo aspetto che vogliamo sottolineare è relativo all'occupazione dello spazio di memoria, legata alle chiamate di funzione. Il numero totale delle chiamate effettuate dall'inizio alla fine dell'elaborazione cresce molto velocemente con *n*. Inoltre, quando si arriva a ciascun caso base vi è una catena di chiamate "aperte" lungo il percorso che va dalla chiamata iniziale `Fibonacci(n)` al caso base in questione.

La seguente tabella dà un'idea del numero totale di chiamate in funzione di *n*.

n	*F(n)*	Numero di chiamate di funzione
0	0	1
1	1	1
2	1	3
3	2	5
…	…	…
23	28657	92735
24	46368	150049
25	75025	242785
…	…	…
41	165580141	535828591
42	267914296	866988873

Considerando i costi già citati legati all'attivazione di una nuova chiamata di funzione, tutto questo fiorire di chiamate e relativi ritorni non è consigliabile, soprattutto perché può facilmente essere evitato mediante il ricorso a un algoritmo iterativo, come il seguente:

```
Funzione Fibonacci_iterativa (n: intero non negativo)
     if ((n = 0)oppure (n = 1))
          return n
     else
          fib_prec_prec ← 0
          fib_prec ← 1
          for (i = 2 to n)
               fib ← fib_prec + fib_prec_prec
               fib_prec_prec ← fib_prec
               fib_prec ← fib;
          return fib
```

che, banalmente, ha un costo computazionale di $\Theta(n)$ e si risolve con una sola chiamata di funzione.