La ricorsione

Ver. 2.4

© 2010 - Claudio Fornaro - Corso di programmazione in C

Divide et impera

- Metodo di approccio ai problemi che consiste nel dividere il problema dato in problemi più semplici
- I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme per costituire la soluzione del problema originale
- Generalmente, quando la semplificazione del problema consiste essenzialmente nella *semplificazione dei DATI* da elaborare (ad es. la riduzione della dimensione del vettore da elaborare), si può pensare ad una soluzione *ricorsiva*

La ricorsione

- Una funzione è detta ***ricorsiva*** se chiama se stessa
- Se due funzioni si chiamano l'un l'altra, sono dette ***mutuamente ricorsive***
- La funzione ricorsiva sa risolvere *direttamente* solo casi particolari di un problema detti ***casi di base****:* se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora restituisce un risultato
- Se invece viene chiamata passandole dei dati che NON costituiscono uno dei casi di base, allora chiama se stessa (***passo ricorsivo***) passando dei DATI semplificati/ridotti

La ricorsione

- Ad ogni chiamata si semplificano/riducono i dati, così ad un certo punto si arriva ad uno dei casi di base
- Quando la funzione chiama se stessa, sospende la sua esecuzione per eseguire la nuova chiamata
- L'esecuzione riprende quando la chiamata interna a se stessa termina
- La sequenza di chiamate ricorsive termina quando quella *più interna* (annidata) incontra uno dei casi di base
- Ogni chiamata alloca sullo stack (in *stack frame* diversi) nuove istanze dei parametri e delle variabili locali (non `static`)

Esempio

- Funzione ricorsiva che calcola il fattoriale di un numero n

  Premessa (definizione ricorsiva):

  se n ≤ 1 ➔ n! = 1
  se n > 1 ➔ n! = n * (n-1)!

```
int fatt(int n)
{
  if (n<=1)
    return 1;  → Caso di base
  else
    return n * fatt(n-1);
}
```

Semplificazione dei dati del problema

Esempio

- La chiamata a `fatt(n-1)` chiede a `fatt` di risolvere un problema più semplice di quello iniziale (il valore è più basso), ma è sempre lo stesso problema
- La funzione continua a chiamare se stessa fino a raggiungere il caso di base che sa risolvere immediatamente

Esempio

- Quando viene chiamata `fatt(n-1)`, le viene passato come *argomento* il valore `n-1`, questo diventa il *parametro formale* n della nuova esecuzione: ad ogni chiamata la funzione ha un *suo* parametro n dal valore sempre più piccolo
- I parametri n delle varie chiamate sono tra di loro indipendenti (sono allocati nello stack ogni volta in stack frame successivi)

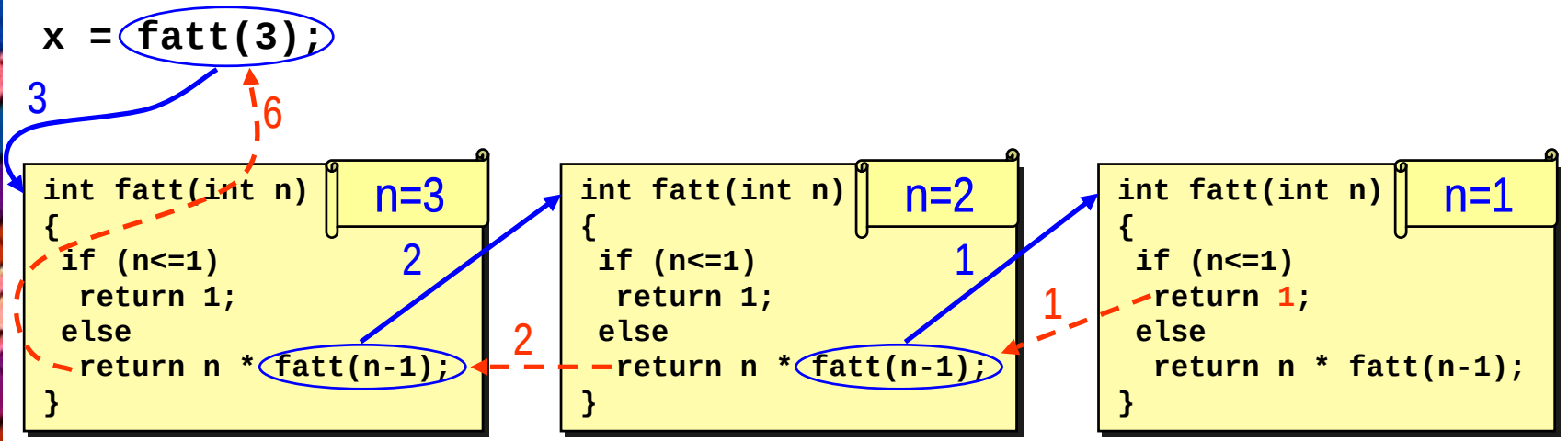
Esempio

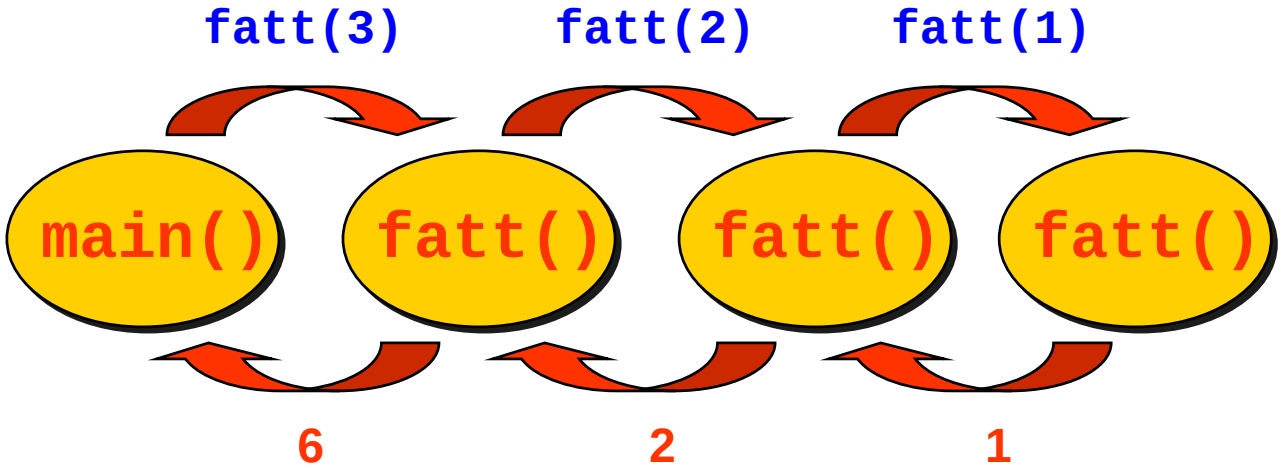
- Supponendo che nel `main` ci sia: `x=fatt(4);`
  - **1ª chiamata:** in `fatt` ora `n=4`, non è il caso di base e quindi richiede il calcolo `4*fatt(3)`, la funzione viene sospesa in questo punto per calcolare `fatt(3)`
  - **2ª chiamata:** in `fatt` ora `n=3`, non è il caso di base e quindi richiede il calcolo `3*fatt(2)`, la funzione viene sospesa in questo punto per calcolare `fatt(2)`
  - **3ª chiamata:** in `fatt` ora `n=2`, non è il caso di base e quindi richiede il calcolo `2*fatt(1)`, la funzione viene sospesa in questo punto per calcolare `fatt(1)`
  - **4ª chiamata:** in `fatt` ora `n=1`, è il caso di base e quindi essa termina restituendo il valore `1` alla 3ª chiamata, lasciata sospesa nel calcolo `2*fatt(1)`

Esempio

- **3ª chiamata:** ottiene il valore di `fatt(1)` che vale **1** e lo usa per il calcolo lasciato in sospeso `2*fatt(1)`, il risultato 2 viene restituito dalla `return` alla 2ª chiamata, lasciata sospesa
- **2ª chiamata:** ottiene il valore di `fatt(2)` che vale **2** e lo usa per il calcolo lasciato in sospeso `3*fatt(2)`, il risultato 6 viene restituito dalla `return` alla 1ª chiamata, lasciata sospesa
- **1ª chiamata:** ottiene il valore di `fatt(3)` che vale **6** e lo usa per il calcolo lasciato in sospeso `4*fatt(3)`, il risultato 24 viene restituito dalla `return` al `main`

Esempio


x = fatt(3);
3
6
int fatt(int n)
{
if (n<=1)
return 1;
else
return n * fatt(n-1);
}
n=3
2
n=2
1
n=1
2
1


fatt(3)
fatt(2)
fatt(1)
main()
fatt()
fatt()
fatt()
6
2
1

Analisi

- L'apertura delle chiamate ricorsive semplifica il problema, ma non calcola ancora nulla
- Il valore restituito dalle funzioni viene utilizzato per calcolare il valore finale man mano che si *chiudono* le chiamate ricorsive: ogni chiamata genera valori intermedi *a partire dalla fine*
- Nella ricorsione vera e propria non c'è un mero passaggio di un risultato calcolato nella chiamata più interna a quelle più esterne, ossia le `return` non si limitano a passare indietro invariato un valore, ma c'è un'elaborazione intermedia

Quando utilizzarla

- PRO
  Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice
- CONTRO
  La *ricorsione è poco efficiente* perché richiama molte volte una funzione e questo:
  - richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno, e i valori di alcuni registri della CPU)
  - consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non `static` e dei parametri ogni volta)

Quando utilizzarla

- CONSIDERAZIONE
  Qualsiasi problema ricorsivo può essere risolto in modo non ricorsivo (ossia iterativo), ma la soluzione iterativa potrebbe non essere facile da individuare oppure essere molto più complessa
- CONCLUSIONE
  Quando non ci sono particolari problemi di efficienza e/o memoria, l'approccio ricorsivo è in genere da preferire se:
  - è più intuitivo di quello iterativo
  - la soluzione iterativa non è evidente o agevole

Quando utilizzarla

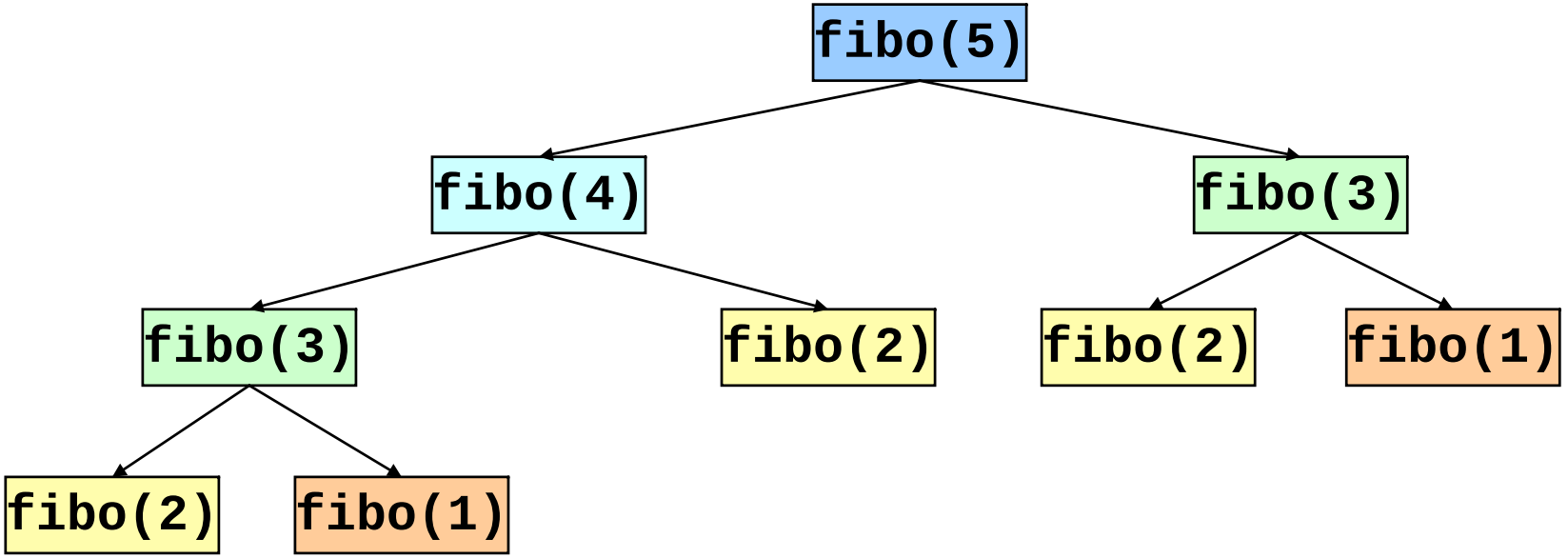
- Una funzione ricorsiva non dovrebbe, in generale, eseguire a sua volta più di una chiamata ricorsiva
- Esempio di utilizzo da evitare:

```
long fibo(long n)
{
  if (n<=2)
    return 1;
  else
    return fibo(n-1) * fibo(n-2);
}
```

- Ogni chiamata genera altre 2 chiamate, in totale vengono effettuate 2^n chiamate (complessità esponenziale)

Quando utilizzarla

- Inoltre `fibo(n)` chiama `fibo(n-1)` e `fibo(n-2)`, anche `fibo(n-1)` chiama `fibo(n-2)`, ecc.:
  si hanno *calcoli ripetuti, inefficiente!*

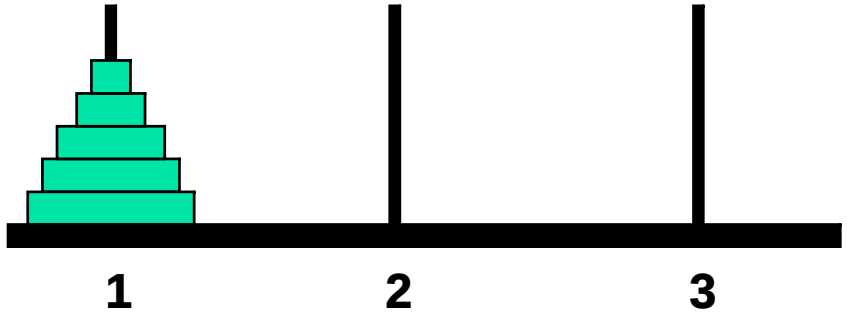


Esercizi

1. Scrivere una funzione ricorsiva per determinare se una stringa è palindroma.
2. Scrivere una funzione ricorsiva per stampare una stringa a rovescio (non la deve invertire, ma solo stampare).
3. Scrivere una funzione ricorsiva che determini il minimo di un vettore di interi.

Esercizi

4. Scrivere un programma per risolvere il problema delle torri di Hanoi: spostare tutti i dischi da 1 a 2 usando 3 come temporaneo. Si può spostare un solo pezzo per volta. Un pezzo grande non può stare sopra uno più piccolo. Suggerimento: usare la funzione `muovi(`*quanti*`, `*from*`, `*to*`, `*temp*`)` che muove un disco direttamente da *from* a *to* solo se *quanti* vale 1.



Ricorsione in coda (tail)

- Si ha quando una funzione ricorsiva chiama se stessa come ultima istruzione prima di terminare e quando termina (se non è `void`) passa al chiamante il risultato senza alcuna successiva elaborazione

```
int funzione(int x) {
   ...
   return funzione(y);
}
```

- Il risultato viene calcolato man mano che si chiamano le funzioni ricorsive (al contrario della ricorsione pura) e si ottiene solo all'ultima chiamata

Ricorsione in coda
Esempio

- Funzione `mcd` che calcola il MCD di due valori `x` e `y`.
  Algoritmo (formula di Euclide):
  - se `y` vale `0`, allora `gcd(x,y)` è pari a `x`
  - altrimenti `gcd(x,y)` è pari a `gcd(y,x%y)`

```
int gcd(int x, int y)
{
  if (y==0)
    return x;
  return gcd(y,x%y);
}
```

Il valore finale passa attraverso la 2ª **`return`** di tutte le chiamate *senza subire elaborazioni*

Ricorsione in coda

- Quando vi è il calcolo di valori intermedi che costituiscono successive approssimazioni del risultato finale, le variabili contenenti questi valori (dette ***accumulatori***) vengono passate ad ogni chiamata della funzione (o sono variabili esterne o locali `static`)

```
int fatt(int n, int accum)
{ if (n>1)
    { accum = accum * n;
      return fatt(n-1, accum);
    }
    return accum;      →  caso di base
}
```

La prima chiamata deve essere `fatt(`*val*`,1)`

Esercizi

5. Scrivere una funzione ricorsiva per cercare un valore all'interno di un vettore non ordinato (ricerca lineare). Risultato: –1 se non lo trova, altrimenti l'indice della posizione dove è stato trovato.
6. Scrivere una funzione ricorsiva per cercare un valore all'interno di un vettore ordinato (ricerca dicotomica). Risultato: – 1 o l'indice.
7. Scrivere una funzione che realizzi un *selection sort* (cerca il valore più piccolo e lo mette in testa, ecc.) in modo ricorsivo su un vettore di interi.

Ricorsione in coda (tail)
Considerazioni

- Escluso il risultato, tutti i dati della chiamata a funzione (salvati sullo stack di sistema) non servono più e alla chiusura vengono solo scartati (serve solo l'*indirizzo di ritorno*)
- Si potrebbe quindi pensare di poterli eliminare dallo stack, ma il Linguaggio C non prevede questa ottimizzazione (***tail optimization***)
- Una funzione che realizza la ricorsione in coda è facilmente riscrivibile come funzione iterativa con il vantaggio di consumare meno memoria e di essere più veloce (non ci sono chiamate a funzione)

Eliminazione di ricorsione tail

- Funzione ricorsiva:

```
tipo Fr(tipo x)
{
  if (casobase(x))
  {
    istruzioni_casobase;
    return risultato;
  }
  else
  {
    istruzioni_nonbase;
    return Fr(riduciComplessita(x);
  }
}
```

- *L*e cancellazioni si applicano se FR è `void`

Eliminazione di ricorsione tail

- Funzione iterativa equivalente:

```
tipo Fi(tipo x)
{
  while (!casobase(x))
  {
    istruzioni_nonbase;
    x=riduciComplessita(x);
  }
  istruzioni_casobase;
  return risultato;
}
```

- `risultato` potrebbe essere `x` o altro

Esercizi

8. Eliminare la ricorsione nell'esercizio 4 secondo la modalità illustrata.
9. Eliminare la ricorsione nell'esercizio 5 secondo la modalità illustrata.
10. Eliminare la ricorsione nell'esercizio 6 secondo la modalità illustrata.

Esempio di soluzione ricorsiva

- Scrivere un programma che stampi tutti gli anagrammi di una *stringa* data permutandone tutti i caratteri
- Il `main` chiama semplicemente la funzione ricorsiva `permuta` passandole la *stringa* da anagrammare (ed eventualmente altro)
- Algoritmo ricorsivo
  la funzione `permuta`:
  - prende (scambia) uno per volta i caratteri della *stringa* passata e li mette all'inizio della *stringa*
  - permuta tutti gli altri caratteri chiamando `permuta` sulla *stringa* privata del primo carattere
  - rimette a posto il car. che aveva messo all'inizio
- Caso di base: lunghezza = 1, stampa *stringa*

Esempio di soluzione ricorsiva

```
void permuta(char *s, char *stringa)
{
  int i, len = strlen(s);
  if (len == 1)                    caso di base
    printf("%s\n", stringa);
  else
  {
    for (i=0; i<len; i++)
    {
      swap(s[0], s[i]);    scambia il 1° chr
      permuta(s+1, stringa);
      swap(s[i], s[0]);    ripristina il 1° chr
    }
}
```

Esempio di soluzione ricorsiva

- La stringa privata di volta in volta del primo carattere è puntata da `s`
- La *stringa* intera è puntata da `stringa` e viene passata per poter essere visualizzata a partire dal suo primo carattere (`s` punta a solo una parte di `stringa`)
- La funzione esegue elaborazioni anche dopo la chiamata (ripristina il primo carattere), quindi l'eliminazione dello stack frame non sarebbe possibile; per questo non è una ricorsione di tipo tail