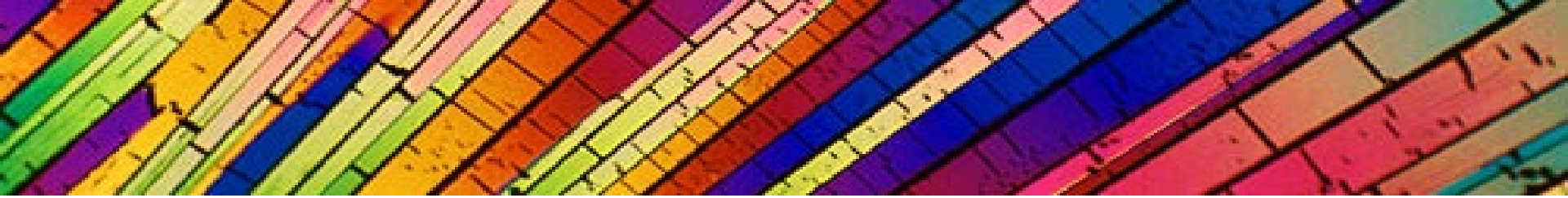


La ricorsione
(introduzione)

Ver. 3

Divide et impera

- “Dìvide et ìmpera” – “Divide and conquer” – “Separa e conquista” è un metodo di approccio ai problemi che consiste nel dividere il problema dato in problemi simili ma più semplici e affrontare la soluzione di questi eventualmente ripetendo il processo
- I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme secondo lo schema opposto alla suddivisione per costituire la soluzione del problema originale

Divide et impera

- 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 a 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 a un certo punto si arriva a uno dei casi di base
- Quando la funzione chiama se stessa, sospende la sua esecuzione attuale per eseguire la nuova chiamata
- L'esecuzione sospesa riprende quando la chiamata che ha effettuato 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 ($n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$)

  Premessa (definizione ricorsiva):

  se $n \leq 1 \rightarrow n! = 1$

  se $n > 1 \rightarrow n! = n \cdot (n-1)!$

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

→ *Caso di base*

Riduzione dei dati del problema

Esempio

- Nella prima invocazione di `fatt`, l'esecuzione si sospende alla moltiplicazione `n * fatt(n-1)` per invocare `fatt(n-1)` L'invocazione `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)`, a `fatt` viene passato come *argomento* il valore **`n-1`**, questo diventa il *parametro formale* **`n`** della nuova esecuzione: a ogni chiamata la funzione ha un *suo* parametro `n` dal valore sempre più piccolo
- I parametri `n` delle varie chiamate sono variabili automatiche e quindi sono riallocate ogni volta, quindi sono tra 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)`, quindi esegue la 2ª chiamata
  - **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)`, quindi esegue la 3ª chiamata
  - **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)`, quindi esegue la 4ª chiamata
  - **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

```
int main()
x = fatt(3);
```

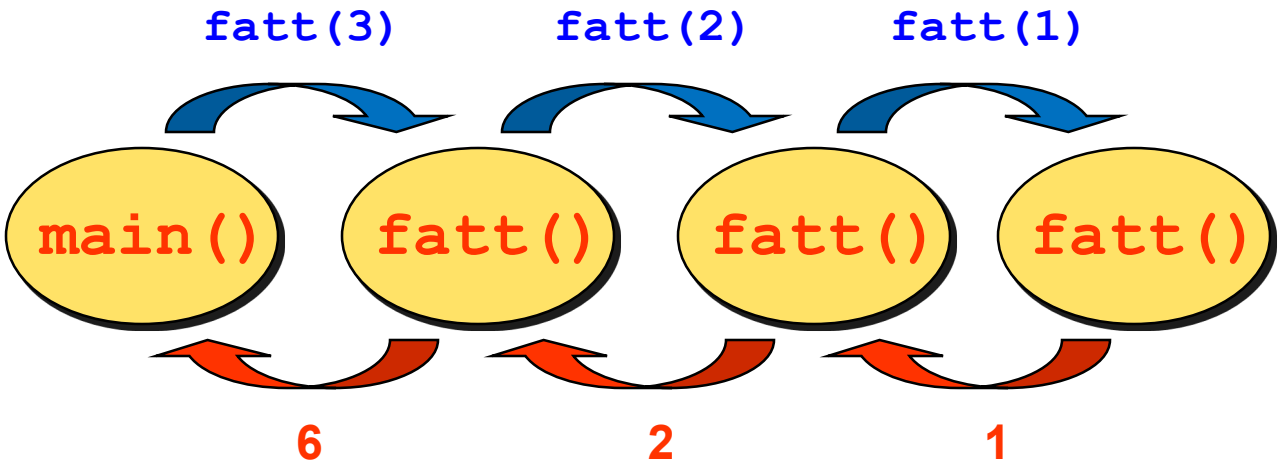
→ *qui l'esempio è con* **n=3** *e non* **n=4**

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

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

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

3 6 2 2 1 1



Altro esempio

- Funzione ricorsiva che calcola x^n

  Premessa (definizione ricorsiva):

  $$\begin{cases} \text{se } n = 0 \rightarrow x^n = 1 \\ \text{se } n > 1 \rightarrow x^n = x \cdot x^{n-1} \end{cases}$$

```
int power(int x, int n)
{
   if (n==0)
      return 1;
   else
      return x * power(x, n-1);
}
```

→ *Caso di base*

Riduzione dei dati del problema

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 *man mano* che si *chiudono* le chiamate ricorsive: ogni chiamata produce valori intermedi diversi *a partire dall'ultima*
- Nella ricorsione "pura" le funzioni elaborano il valore ricevuto dalla chiamata ricorsiva prima di passarlo al chiamante (nella ricorsione "in coda" si vedrà che sarà diverso)

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
  - consuma molta memoria (alloca un nuovo stack frame a 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
  L'approccio ricorsivo è in genere da preferire se:
  - non ci sono particolari problemi di efficienza e/o di memoria
  - è più intuitivo di quello iterativo
  - la soluzione iterativa non è evidente o agevole

Quando utilizzarla

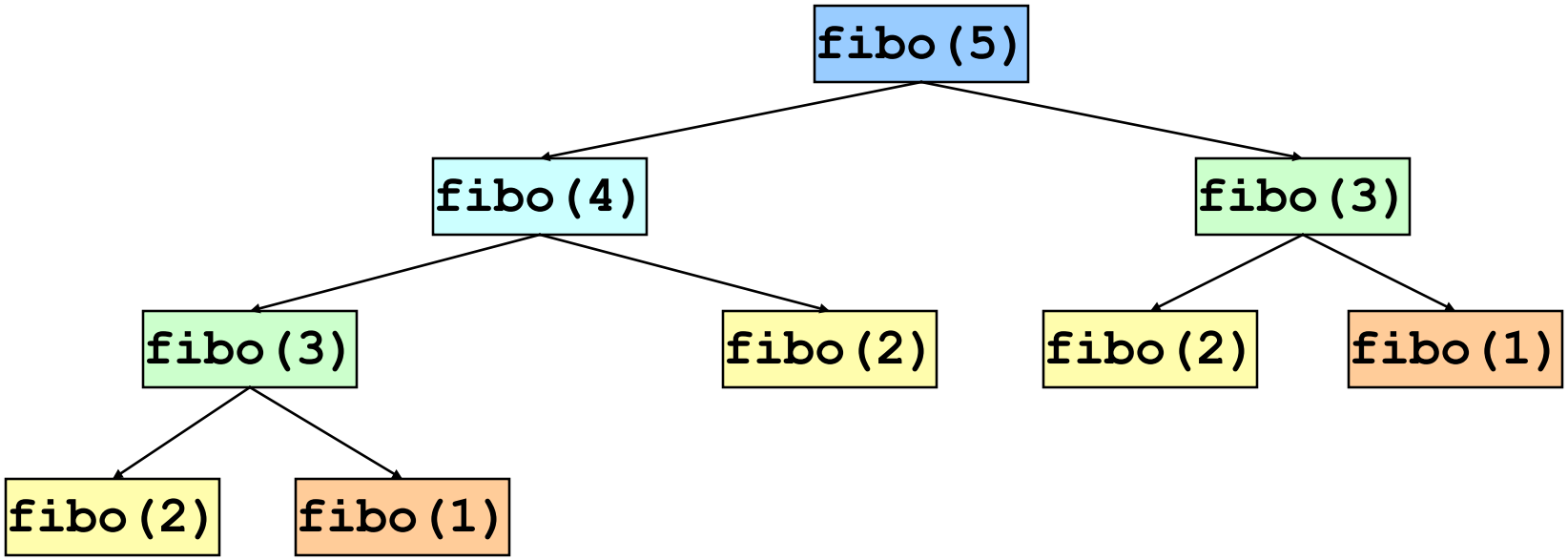
- Una funzione ricorsiva non dovrebbe effettuare a sua volta più di una chiamata ricorsiva
- Esempio di utilizzo da evitare:

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

- Ogni chiamata genera altre 2 chiamate (per `n`=10 ci sono 109 chiamate: `fibo(1)` chiamata 21 volte, `fibo(2)` 34, `fibo(3)` 21, `fibo(4)` 13, per `n`=20 sono 13529 chiamate (`fibo(2)` 4181) → 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!*

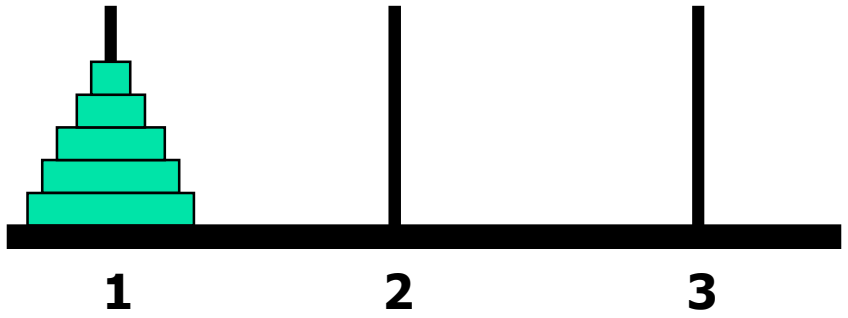


Esercizi

1. Scrivere una funzione ricorsiva per determinare se una stringa è palindroma. Suggerimento: una stringa è palindroma se i suoi caratteri estremi sono uguali e i restanti centrali costituiscono una stringa 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.



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 questo puntatore deve essere passato per poterla visualizzare a partire dal suo primo carattere (`s` punta a solo una parte di `stringa`)
- Nota (dopo aver visto la ricorsione in coda): 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

Quicksort ricorsivo

- È tra i più efficienti algoritmi di ordinamento di vettori
- Stabilisce un valore (detto *pivot*) e divide il vettore in 2 parti in base ad esso:
  - nella parte sinistra mette (senza ordinarli) tutti i valori minori del pivot
  - nella parte destra tutti gli altri
- Dopo la partizione il pivot è nella pos. giusta
- Ripete la procedura su ciascuna delle due parti finché queste non hanno lunghezza ≤ 1
- Ogni suddivisione richiede che venga determinato un nuovo pivot

Quicksort ricorsivo

- quick(vettore)
  - se lunghezza <= 1
    - ordinamento finito, termina
  - seleziona pivot (ad es. il valore centrale)
  - i=indice primo valore del vettore
  - j=indice ultimo valore del vettore
  - ripeti finché i<=j
    - incrementa i finché vettore[i] < pivot
    - decrementa j finché vettore[j] > pivot
    - se i<=j
      - scambia vettore[i] e vettore[j]
      - i++
      - j--
  - quick(vettore a sinistra dell'elemento i)
  - quick(vettore a destra dell'elemento j)

Ricorsione in coda

- Si ha una *ricorsione in coda* (*tail recursion*) quando ogni chiamata ricorsiva esegue il calcolo di valori intermedi e li passa all'invocazione successiva per un'ulteriore elaborazione.
- In questo modo è l'ultima chiamata quella che calcola il risultato finale.
- Questo deve poi solo essere passato indietro al chiamante con una semplice `return` alla terminazione di ciascuna delle chiamate.

Ricorsione in coda

- La struttura tipica ha la forma seguente:

```
tipo_rest funzione(tipo x)
{
   y = espressione con x;
   return funzione(y);
}
```

ma sia *tipo_rest* sia *tipo* potrebbero essere `void` se si usano variabili esterne o `static`

Ricorsione in coda

- Nella "ricorsione pura" istanze locali (automatiche) diverse della stessa variabile vengono utilizzate per calcolare il risultato man mano che si chiudono le chiamate dall'ultima alla prima, il risultato finale si ottiene solo alla chiusura della prima
- Nella ricorsione in coda il risultato viene calcolato man mano che si chiamano le funzioni ricorsive e si ottiene all'esecuzione dell'ultima chiamata, per poi solo trasferirlo indietro fino alla prima invocazione; le variabili possono essere non solo locali ma anche esterne/`static`

Ricorsione in coda
Esempio

- Funzione `mcd` che calcola il MCD di due valori `x` e `y` con la 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

```
int fatt(int n, int acc)
{ if (n>1)
   { accum = accum * n;
     return fatt(n-1, acc);
   }
   return acc;
}
```

→ *caso di base*

La prima chiamata deve essere `fatt(`*val*`,1)`
Il valore finale passa attraverso la 1ª `return` di tutte le chiamate *senza subire elaborazioni*.
Qui è necessario usare una variabile di appoggio (poteva anche essere extern o `static` per non essere un argomento) per contenere i risultati delle elaborazioni parziali

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
Considerazioni

- Nella ricorsione in coda, i dati automatici della chiamata a funzione, salvati sullo stack nei vari *stack frame*, non servono più alla funzione quando questa torna in esecuzione e alla chiusura vengono solo scartati: di ciascun stack frame serve solo l'*indirizzo di ritorno* (e il risultato finale)
- Si potrebbe quindi pensare di eliminare gli stack frame man mano che hanno esaurito il loro compito, mantenendo solo le informazioni che servono, ma il Linguaggio C non prevede questa ottimizzazione (*tail optimization*)

Ricorsione in coda
Considerazioni

- 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 tail:

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

- Le 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.