Ricorsione in C

slides credit Prof. Paolo Romano

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`)

Ricorsione

Una funzione che contiene al suo interno una attivazione di se stessa è detta *ricorsiva*.

Esempio:

```
void f(int a){
  if (a==0)
    printf("f(%d): ho finito \n“,a);
  else {
    printf("sono f(%d) ",a);
    printf("chiamata di f(%d) \n",a-1);
    f(a-1);
    printf ("f(%d): ho finito \n",a);
  }
}
int main(){
  f(3);
}
```

Output prodotto:

```
Sono f(3) chiamata di f(2)
Sono f(2) chiamata di f(1)
Sono f(1) chiamata di f(0)
f(0):ho finito
f(1):ho finito
f(2):ho finito
f(3):ho finito
```

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)

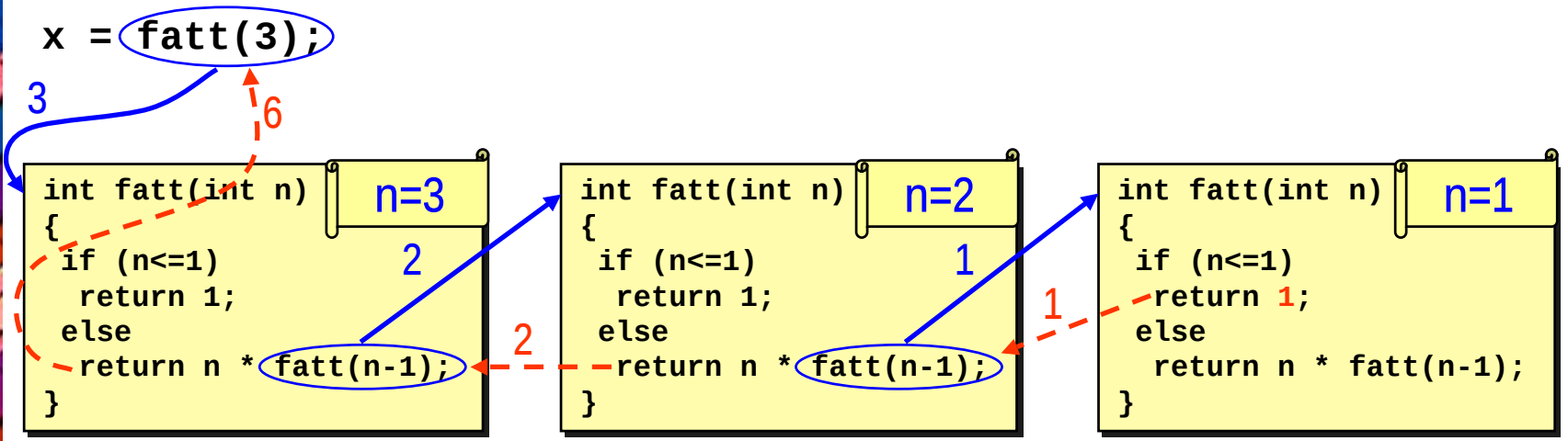
Esempio

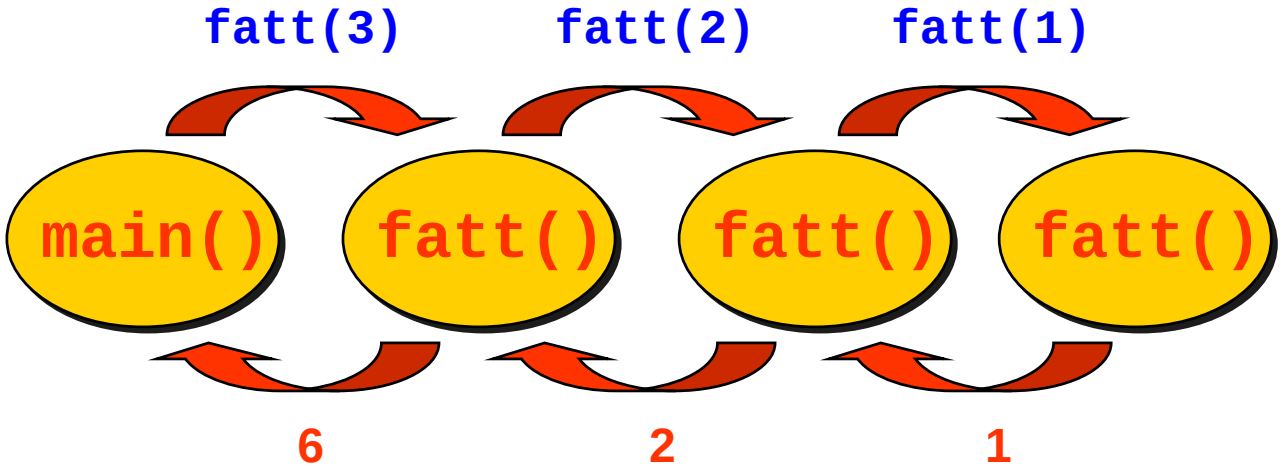
- Supponendo che nel `main` ci sia: `x=fatt(4);`
  - **1ª chiamata:** in `fatt` ora `n=4`, non è il caso di base e quindi richiede il calcolo `4*fatt(3)`, la funzione viene sospesa in questo punto per calcolare `fatt(3)`
  - **2ª chiamata:** in `fatt` ora `n=3`, non è il caso di base e quindi richiede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

Ricorsione

Altre definizioni induttive

Somma non negativi

$$somma(x,y) = \begin{cases} x & \text{se } y=0 \quad \text{(base)} \\ 1 + somma(x,y-1) & \text{se } y>0 \quad \text{(passo)} \end{cases}$$

Prodotto non negativi

$$prod(x,y) = \begin{cases} 0 & \text{se } y=0 \quad \text{(base)} \\ somma(x, prod(x,y-1)) & \text{se } y>0 \quad \text{(passo)} \end{cases}$$

Potenza non negativi

$$espon(x,y) = \begin{cases} 1 & \text{se } y=0 \quad \text{(base)} \\ prod(x, espon(x,y-1)) & \text{se } y>0 \quad \text{(passo)} \end{cases}$$

Esercizio: implementare queste operazioni come funzioni C.

Ricorsione

Esempio: leggere sequenza di caratteri interrotte da "\n" e stampare invertite. Es. legge: "arco" e stampa "ocra"

Problema: per invertire occorre prima memorizzare tutta le lettere, occorre una struttura per la memorizzazione:
1. Vettore (vediamo più avanti)
2. Sfruttare lo stack di RDA

Algoritmo:

leggi carattere i-esimo
if carattere letto "\n" fine
else {
  stampa il carattere i+1
  stampa il carattere corrente
}

```
void stampaInv(){
   char a;

   scanf("%c",&a);
   if(a != '\n'){
      stampaInv();
      printf("%c",a);
   }
}
```

Esempio

Leggere una sequenza di caratteri con un punto centrale e decidere se è palindroma (ignorando gli spazi bianchi)

Una sequenza è detta palindroma se la seq. letta da destra a sinistra è identica a quella letta da destra (es. anna).

Caratterizzazione ricorsiva di una plaindroma:
1. La sequenza costituita solo da ' . ' è palindroma
2. Una sequenza x s y è palindorma se lo è s ed x = y

Come si può fare senza il ' . ' centrale?

Soluzione

```
void palind() {
        char x,y;

        scanf(“%c”,&x);
        if (x==‘.’)  return 1;

        if ( palind() ) {
                scanf(“%c”,&y);
                if (x!=y) return 0;
                return 1;
        }
return 0;
}
```

```
int main() {
while ( palind() )
  printf(“Stringa palindroma, prova ancora!\n”);
printf(“Stringa non palindroma”);
}
```

Ricorsione Multipla

Si ha ricorsione multipla quando un'attivazione di funzione causa **più di una attivazione ricorsiva** della stessa funzione

Esempio: n-esimo numero di Fibonacci

Sequenza Fibonacci: 0, 1, 1, 2, 3, 5, 8, …

Def. induttiva

$$F(0) = 0 \qquad n=0$$
$$F(1) = 1 \qquad n=1$$
$$F(n) = F(n-2) + F(n-1) \qquad n > 1$$

Algoritmo

if n = 0 return 0
if n = 1 return 1
if n > 1
  calcola F(n-2)
  calcola F(n-1)
  return F(n-2)+F(n-1)

```
int fib(int n){
   if (n==0) return 0;
   if (n==1) return 1;
   if (n>1)  return fib(n-2) + fib(n-1);
}
```

Esempi

MCD con funzione ricorsiva:

$$MCD(x,y) \begin{cases} x & \text{se } y = x \\ MCD(x,y-x) & \text{se } y > x \\ MCD(x-y,y) & \text{se } x > y \end{cases}$$

```
int mcd(int x, int y) {
   if (y==x) return x;
   else if (y > 0) return mcd(x,y-x);
        else return mcd(x-y,x); }
```

Verifica interi primi tra loro:

$$primi(x,y) \begin{cases} vero & \text{se } x=1 \text{oppure } y=1 \\ falso & \text{se } x \neq 1,\ y \neq 1 \text{e } x = y \\ primi(x,y-x) & \text{se } x \neq 1,\ y \neq 1 \text{e } x < y \\ primi(x-y,y) & \text{se } x \neq 1,\ y \neq 1 \text{e } x > y \end{cases}$$

```
int primi(int x,int y) {
   if(x==1 || y==1) return 1;
   if(x!=1 && y!=1 && x ==y) return 0;
   if(x!=1 && y!=1 && x > y) return primi(x,y-x);
   if(x!=1 && y!=1 && x < y) return primi(x-y,y);
}
```

Esempi

Resto:

$$\text{resto}(x,y) \begin{cases} x & \text{se } 0 \leq x < y \\ \text{resto}(x-y,y) & \text{se } x > y \\ \text{resto}(x+y,y) & \text{se } x < 0 \end{cases}$$

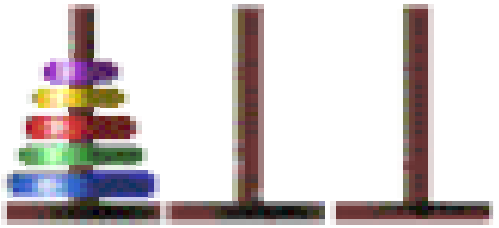
```
int resto(int x){
   if (x < y && x ≥ y) return x;
   if (x > y) return resto(x-y,y);
   if (x <0) return resto(x+y,y);
}
```

Ackerman:

$$A(m,n) \begin{cases} n+1 & \text{se } m=0 \text{ (caso base)} \\ A(m-1,1) & \text{se } n=0 \text{ (passo ricorsivo)} \\ A(m-1,A(m,n-1)) & \text{altrimenti} \end{cases}$$

Esempio Torre di Hanoi

- Tre perni 1, 2, 3
- Pila di dischi dimensione crescente su perno 1
- vincoli:
  - uno solo disco alla volta può essere spostato
  - disco più grande mai sopra uno più piccolo



Problema: come spostare m dischi dal perno 1 al perno 2?

Algoritmo:
1. Spostare n-1 dischi da 1 a 3
2. Spostare m-esimo da 1 a 2
3. Spostare n-1 dischi da 3 a 2

Esempio Torre di Hanoi

Algoritmo:
1. Spostare n-1 dischi da 1 a 3
2. Spostare m-esimo da 1 a 2
3. Spostare n-1 dischi da 3 a 2

Si utilizza una funzione muovi:

```
void muovi(int n, int s, int d, int a)
{
 if (n == 1)
     muoviUnDisco(s, d);
 else {
     muovi(n-1, s, a, d);
     muoviUnDisco(s, d);
     muovi(n - 1, a, d, s);
 }
}  /* muovi */
```

Esempio Torre di Hanoi

```
int main(void)
{
  int dischi;    /* numero di dischi */
  int s, d;      /* pali sorgente e destinazione */

  printf("Numero di dischi? ");
  scanf("%d", &dischi);
  printf("Palo sorgente?     [1, 2 o 3] ");
  scanf("%d", &s);
  printf("Palo destinazione? [1, 2 o 3] ");
  scanf("%d", &d);

  muovi(dischi, s, d, 6 - d - s);
  return 0;

}  /* main */
```

Esempio Torre di Hanoi

Quando si utilizza la ricorsione multipla, il numero di attivazioni può essere esponenziale nella profondità della chiamata ricorsiva (cioè nell'altezza della pila di attivazione)

Esempio Torre Hanoi:

att(n) = numero attivazioni di `muovi(…)` per n dischi = numero spostamento dischi

$$att(n) = \begin{cases} 1, & n=1 \\ 1+2\ att(n-1), & n>1 \end{cases}$$

$$att(n) > 2^{n-1}$$