Programmazione ricorsiva

Funzioni ricorsive

Definizione di ricorsione

Una funzione matematica è definita per ricorsione (o per induzione) quando è espressa in termini di se stessa.

ESEMPIO

il fattoriale

$$f(n) = \begin{cases} 1 & \text{se } n=0 \\ n*f(n-1) & \text{se } n>0 \end{cases}$$

La base teorica è il *principio di induzione:*

Una funzione ricorsiva si specifica definendo:

- quanto vale in uno o più **casi base**
- come si può ricondurre il **generico caso** "di grado n" a uno o più casi più semplici (di grado < n)

RICORSIONE E PROGRAMMAZIONE

- Una **funzione** di un linguaggio di programmazione è un **servitore** che realizza l'astrazione di funzione matematica.
- Il servitore, a sua volta, può essere cliente di altri servitori.
- Come caso particolare, un servitore può essere **cliente di se stesso** (**funzione ricorsiva***)*
- Una **funzione ricorsiva**
  - sa come risolvere i *casi* ***base***
  - per risolvere il caso complesso
    - ***suddivide*** il problema in termini dello stesso problema applicato a casi più semplici
    - ***combina*** le soluzioni di ogni sottoproblema
  - per ogni sottoproblema
    - conosce la soluzione oppure
    - ***richiama*** se stessa

Esempio 1

Il fattoriale

$$f(n) = \begin{cases} 1 \text{se } n=0 \\ n*f(n-1) \text{se } n>0 \end{cases}$$

CODIFICA

```
unsigned long fattoriale(unsigned long n) {
      if (n==0)
            return 1;
      else
            return  n * fattoriale(n-1);}
```

ESEMPIO D'USO

```
main(){
  int x = 2, y;
  y = fattoriale(x);}
```

Esempio 2

Calcolare la somma dei primi N numeri positivi

SPECIFICA ITERATIVA

- Inizializza *sum=0*
- Ripeti per N volte l'operazione elementare *sum* = *sum* + *i*

SPECIFICA RICORSIVA

- considera la *somma dei primi N numeri positivi*

  (1+2+3+...+(N-1)+N)
- come la somma di due termini:

  $\underbrace{(1+2+3+...+(N-1))}_{\text{somma dei primi N-1 numeri positivi}}$ **+** N
- è facile identificare un caso base:

  la *somma del primo numero positivo (N=1)* vale 1.

Esempio 2 - CODIFICA

```
int sommaFinoA(int n){
 if (n==1)
     return 1; /* caso base */
 else/* ricorsione */
     return sommaFinoA(n-1)+n;
}
```

I NUMERI DI FIBONACCI

Nel 1223 a Pisa, Leonardo Pisano, meglio noto come **Fibonacci**, vinse un singolare torneo tra abachisti e algoritmisti, armati soltanto di carta, penna e pallottoliere.

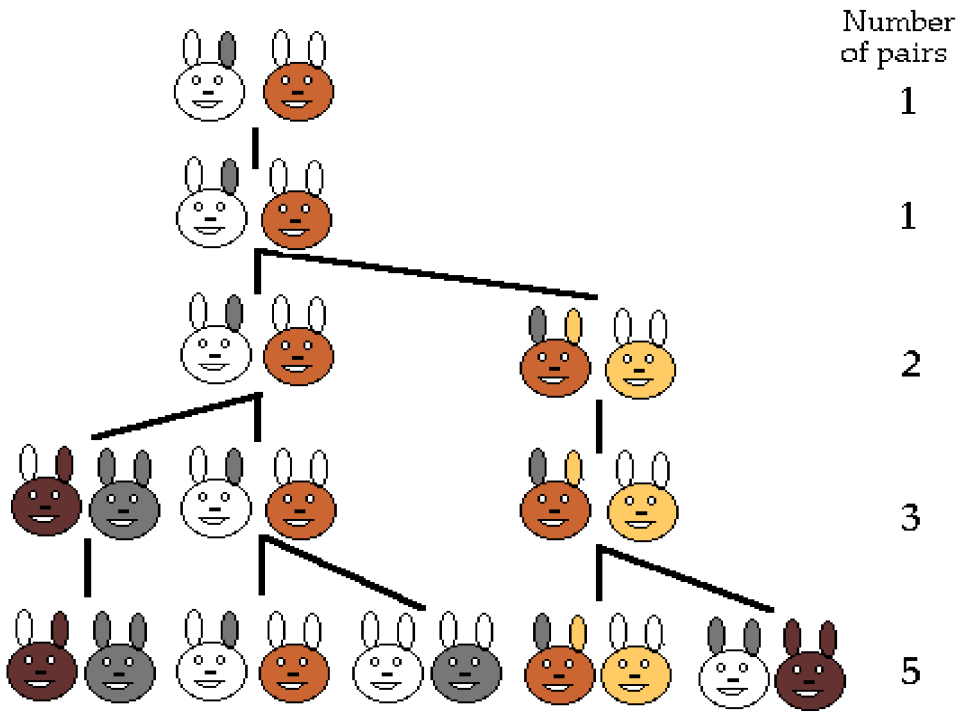
In quella gara si dimostrò che col metodo posizionale indiano appreso dagli arabi si poteva calcolare più velocemente di qualsiasi abaco.

Il quesito era il seguente:

"Quante coppie di conigli si ottengono in un anno a partire da una singola coppia supponendo che

- le coppie non muoiano mai
- ogni coppia dia alla luce un'altra coppia ogni mese
- le coppie più giovani siano in grado di riprodursi dal secondo mese di vita?"

I NUMERI DI FIBONACCI

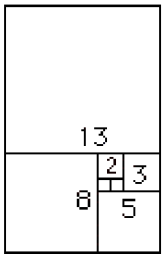


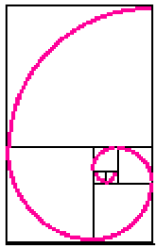
$$\text{fib}(n) = \begin{cases} n & \text{se } n=0 \text{ o } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{altrimenti} \end{cases}$$

La serie di Fibonacci

- ha riscontro in vari fenomeni naturali.
  Ad esempio il numero dei petali di molte piante (3 per iris e lilium, 21 per cicoria, ecc.) o il numero dei semi delle piante (girasoli, ecc.).

- descrive una forma di spira





- Il rapporto tra i numeri consecutivi di Fibonacci converge verso
  la sezione aurea *o* ***costante di Fidia***

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi$$

$$\phi = \frac{1+\sqrt{5}}{2} = 1,6180339887...$$

La sezione aurea:

- corrisponde al rapporto tra la lunghezza del braccio e l'avambraccio
- … e altre proporzioni sono verificabili per il corpo umano
- è usata in architettura per progettare finestre, stanze, palazzi, ecc.

La funzione di Fibonacci

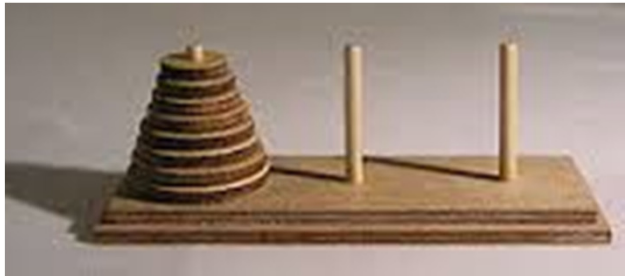
$$\text{fib}(n) = \begin{cases} n & \text{se } n=0 \text{ o } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{altrimenti} \end{cases}$$

CODIFICA

```
unsigned fibonacci(unsigned n) {
  if (n<2)
  return n; /* riassume 2 casi */
  return fibonacci(n-1) + fibonacci(n-2);}
```

- Questo è un esempio di **ricorsione non lineare** (la definizione della funzione si basa su *più* chiamate ricorsive)
- I due casi precedenti erano invece esempi di **ricorsione lineare** (una sola chiamata ricorsiva)

La torre di Hanoi



- La Torre di Hanoi è un rompicapo matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti.
- Il gioco inizia con tutti i dischi incolonnati su un paletto in ordine decrescente, in modo da formare un cono.
- Lo scopo del gioco è portare tutti i dischi sull'ultimo paletto.
- E' possibile spostare solo un disco alla volta e potendo mettere un disco solo su un altro disco più grande, mai su uno più piccolo.

La soluzione del gioco

La soluzione base del gioco della torre di Hanoi si formula in modo ricorsivo:

ALGORITIMO PER LA TORRE DI HANNOI

Siano i paletti etichettati con A, B e C, e i dischi numerati da 1 (il più piccolo) a n (il più grande):

① Sposta i primi n-1 dischi da A a B. (Questo lascia il disco n da solo sul paletto A)

② Sposta il disco n da A a C

③ Sposta n-1 dischi da B a C

La **soluzione del problema** per n dischi è stata ricondotta alla **soluzione del problema** per n-1 dischi (cfr. passi 1 e 3)!

Programma Torre di Hannoi

```
#include <stdio.h>
/* Dichiarazione della procedura ricorsiva */
/*pioloS: sorgente pioloD: destinazione pioloA: ausiliario */
void muovi_dischi(int n, char pioloS, char pioloD, char pioloA);


void main()
{
  int n;

  do {
    printf("Numero n dei dischi che vuoi considerare? (n>0)\n");
    scanf("%d", &n);
  } while(n <= 0);
  muovi_dischi(n, 'A', 'B', 'C');
}
```

Programma Torre di Hannoi (cont.)

```
void muovi_dischi(int n, char pioloS, char pioloD, char pioloA)
{
if(n == 1) {
    printf("Muovi un disco dal piolo %c al piolo %c\n”,pioloS,pioloD);
  }
  else {
    muovi_dischi(n-1, pioloS, pioloA, pioloD);
    printf("Muovi un disco dal piolo %c al piolo %c\n”,pioloS,pioloD);
    muovi_dischi(n-1, pioloA, pioloD, pioloS);
  }
}
```

Esempio 3

Calcolare il minimo di una sequenza di elementi [a1, a2, a3, ... aN]

SPECIFICA RICORSIVA

- Il *minimo della sequenza* $[a_1, a_2, a_3, ..., a_N]$ è il minimo tra la coppia
  - a_1
  - il minimo della sequenza $[a_2, a_3, ..., a_N]$
- il *minimo della sequenza [a]* è *a* (caso base)

L'algoritmo per trovare il minimo si esprime allora così:

min_seq($[a_1, a_2, a_3, ..., a_N]$)

= min(a_1, min_seq($[a_2,a_3,... a_N]$))

= min(a_1, min(a_2, min_seq($[a_3,...]$)))

= …

NOTA Si inizia a sintetizzare il risultato solo quando si sono aperte tutte le chiamate ovvero quando si applica min_seq($[a_N]$)=a_N.

Esempio 3 (cont.)

- Scrivere una funzione C per applicare la funzione ricorsiva di ricerca del minimo a vettori

```
int recursiveMin(int V[], int i, int n){
      if(i == n-1)
            return V[i];
      else
            return min(V[i], recursiveMin(V,i+1,n));}

int min(int a, int b){
      if(a<b)
            return a;
      else
            return b;}
```

Esempio 3 (cont.)

```
#include <stdio.h>
int min(int a, int b);
int recursiveMin(int* V, int i, int n);

main(){
  int vett[100];
  int num, min, i;

  printf("Quanti elementi: ");
  scanf("%d",&num);
  for(i=0;i<num;i++)
     scanf("%d",&vett[i]);
  min=recursiveMin(vett,0,num);
  printf("Elemento minimo: %d", min);}
```

Esempio 4

Calcolare la lunghezza di una sequenza di elementi [a_1, a_2, a_3, ... a_N]

SPECIFICA RICORSIVA

Data una sequenza S: $\text{lung}(S) = \begin{cases} 0 & \text{se } S = [\,] \\ 1 + \text{lung}(S1) & \text{se } S = [\,a, S1\,] \end{cases}$

Ad esempio

lung([a, 3, 56, h, #])
= (1 + lung([3, 56, h, #]))
= (1 + (1 + lung([56, h, #])))
= (1 + (1 + (1 + lung([h, #]))))
= (1 + (1 + (1 + (1 + lung([#])))))
= (1 + (1 + (1 + (1 + (1 + lung([ ]))))))
= (1 + (1 + (1 + (1 + (1 + 0)))))
= (1 + 4) = 5

Esempio 4 (cont.)

- Scrivere una funzione C per applicare la funzione ricorsiva per il calcolo della lunghezza ad una stringa

```
int recursiveL(char S[], int i){
      if(S[i] == '\0')
            return 0;
      else
            return 1 + recursiveL(S, i+1);}
```

Esercizio

Scrivere un adeguato main di prova.

Esempio 5

Verificare l'appartenenza di un elemento b ad una sequenza di elementi $[a_1, a_2, a_3, \ldots a_N]$

SPECIFICA RICORSIVA

Data la sequenza S e l'elemento b:

$$\text{in}(S, b) = \begin{cases} \text{falso se } S = [\,] \\ \text{vero se } S = [b, S1] \\ \text{in}(S1, b) \text{se } S = [a, S1] \end{cases}$$

Ad esempio

- in([a, b, 3, $], 3) = in([b, 3, $], 3)
  = in([b, 3, $], 3) = in([3, $], 3) = true
- in([a, b, 3], c) = in([b, 3], c)
  = in([3], c) = in([ ], c) = false

Esempio 5 (cont.)

NOTA

Il risultato viene sintetizzato *via via* che le chiamate ricorsive si succedono.

Esercizio

Scrivere una funzione ricorsiva in C per la ricerca di un elemento in un vettore.

Esempio 6

Calcolare il Massimo Comune Divisore fra due numeri.

SPECIFICA RICORSIVA

$$MCD(m, n) = \begin{cases} m & \text{se } m=n \\ MCD(m-n, n) & \text{se } m>n \\ MCD(m, n-m) & \text{se } m<n \end{cases}$$

CODIFICA

```
int mcd(int m, int n){
      if (m==n) return m;
      else
      if (m>n)  return mcd(m-n, n);
      else return mcd(m, n-m);}
```

Esercizi

- Dato un intero x, scrivere una funzione ricorsiva che calcola x^n con n intero positivo
- Un palindromo è una stringa che si legge allo stesso modo da sinistra a destra e da destra a sinistra (sono ad esempio palindromi “radar” e “a man a plan a canal panama”). Supponendo che la stringa sia priva di spazi e punteggiatura, scrivere una funzione booleana ricorsiva `testPalindrome` che restituisca true, qualora la stringa sia un palindromo, e false in caso contrario. Scrivere poi un possibile `main` di prova.
- Supponendo che la stringa possa contenere spazi e punteggiatura, modificare la funzione `testPalindrome` in modo tale che ignori gli spazi e la punteggiatura eventualmente presenti.

CONSIDERAZIONI FINALI: ITERAZIONE E RICORSIONE

- **L'approccio iterativo** richiede di *vedere la soluzione del problema "tutta insieme"* in termini di mosse elementari (vedi ESEMPIO 2).
  Le soluzioni iterative sono generalmente *più efficienti* di quelle ricorsive, in termini sia di memoria occupata sia di tempo di esecuzione.
- **L'approccio ricorsivo** richiede invece solo di *esprimere il problema in termini dello stesso problema in casi più semplici*, più qualche elaborazione elementare.
  Le soluzioni ricorsive sono quindi *più espressive* e *molto più compatte* di soluzioni iterative.