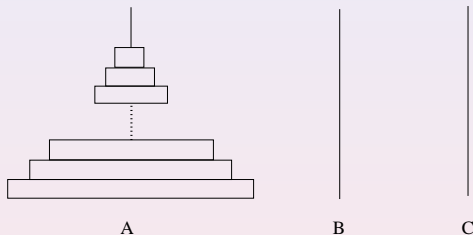


Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure **ricorsive**: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- ▶ Ciò può avvenire
 - ▶ **direttamente**: il corpo di F contiene una chiamata a F stessa.
 - ▶ **indirettamente**: F contiene una chiamata a G che a sua volta contiene una chiamata a F .
- ▶ Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema \mathcal{P} , una definizione ricorsiva sembra indicare che per risolvere \mathcal{P} dobbiamo . . . saper risolvere \mathcal{P} !

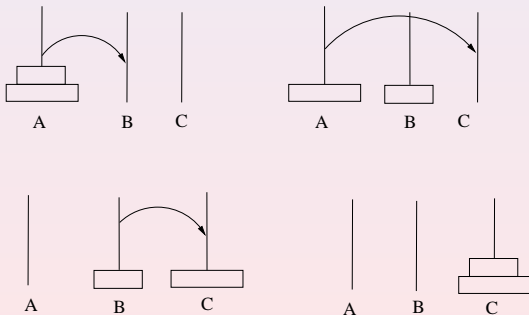
- ▶ In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi **la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.**
- ▶ La programmazione ricorsiva trova radici teoriche nel **principio di induzione ben fondata** che può essere visto come una generalizzazione del **principio di induzione** sui naturali
- ▶ La soluzione di un problema viene individuata **supponendo** di saperlo risolvere su casi più semplici.
- ▶ Bisogna poi essere in grado di risolvere **direttamente** il problema sui casi più semplici di qualunque altro.

Esempio: Torre di Hanoi (leggenda Vietnamita).



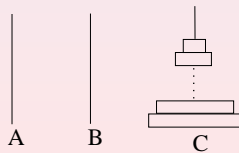
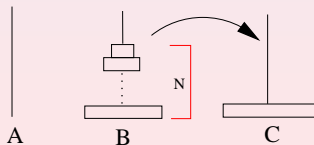
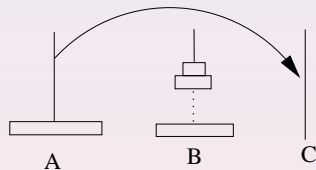
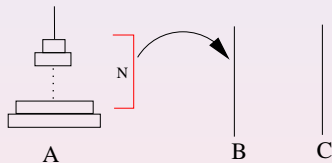
- ▶ pila di dischi di dimensione decrescente su un perno A
- ▶ vogliamo spostarla sul perno C, usando un perno di appoggio B
- ▶ vincoli:
 - ▶ possiamo spostare un solo disco alla volta
 - ▶ un disco più grande non può mai stare su un disco più piccolo
- ▶ secondo la leggenda: i monaci stanno spostando 64 dischi: quando avranno finito, ci sarà la fine del mondo

- ▶ Come individuare una soluzione per un numero N di dischi arbitrario?
 - ▶ per $N=1$ la soluzione è immediata: spostiamo l'unico disco da A a C
 - ▶ se sappiamo risolvere il problema per $N=1$ lo sappiamo risolvere anche per $N=2$: come?



- ▶ Notiamo l'utilizzo del perno ausiliario B

- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con $N+1$ dischi?



- ▶ Formalizziamo il ragionamento
- ▶ Indichiamo con `hanoi(N, P1, P2, P3)` il problema: “spostare `N` dischi dal perno `P1` al perno `P2` utilizzando `P3` come perno d'appoggio”.

```
hanoi(N, P1, P2, P3)
  if (N=1)
    sposta da P1 a P2;
  else
    {
      hanoi(N-1, P1, P3, P2);
      sposta da P1 a P2;
      hanoi(N-1, P3, P2, P1);
    }
```

Esempio: Soluzione di `hanoi(3,A,C,B)`

| | | | |
|-------------------------------|-------------------------------|-------------------------------|--------------------------|
| | | <code>hanoi(1,A,C,B) =</code> | <code>sposta(A,C)</code> |
| | <code>hanoi(2,A,B,C) =</code> | <code>sposta(A,B)</code> | |
| | | <code>hanoi(1,C,B,A) =</code> | <code>sposta(C,B)</code> |
| <code>hanoi(3,A,C,B) =</code> | <code>sposta(A, C)</code> | | |
| | | <code>hanoi(1,B,A,C) =</code> | <code>sposta(B,A)</code> |
| | <code>hanoi(2,B,C,A) =</code> | <code>sposta(B,C)</code> | |
| | | <code>hanoi(1,A,C,B) =</code> | <code>sposta(A,C)</code> |

- ▶ Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo **induttivo**.

Esempio: Definizione induttiva di somma tra due interi non negativi:

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

- ▶ La somma di x con 0 viene definita in modo immediato;
 - ▶ la somma di x con il successore di y viene definita come il successore della somma tra x e y .
- ▶ **Esempio:** somma di 3 e 2 :

$$\begin{aligned} somma(3, 2) &= 1 + (somma(3, 1)) = \\ &= 1 + (1 + (somma(3, 0))) = \\ &= 1 + (1 + (3)) = \\ &= 1 + 4 = \\ &= 5 \end{aligned}$$

Esempio: Funzione fattoriale.

- ▶ definizione iterativa: $fatt(n) = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1$
- ▶ definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 & \text{(caso base)} \\ n \cdot fatt(n - 1) & \text{se } n > 0 & \text{(caso induttivo)} \end{cases}$$

- ▶ È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) &= 3 \cdot \underline{fatt(2)} = \\ &= 3 \cdot \underline{(2 \cdot \underline{fatt(1)})} = \\ &= 3 \cdot \underline{(2 \cdot \underline{(1 \cdot \underline{fatt(0)})})} = \\ &= 3 \cdot \underline{(2 \cdot \underline{(1 \cdot 1)})} = \\ &= 3 \cdot \underline{(2 \cdot 1)} = \\ &= 3 \cdot 2 = \\ &= 6 \end{aligned}$$

Il codice delle due diverse versioni

► definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
        ris=ris*i;
    return ris;
}
```

► definizione ricorsiva:

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

Esempio: Programma che usa una funzione ricorsiva.

```
#include <stdio.h>

int fattric (int);

main()
{
  int x, f;
  scanf("%d", &x);
  f = fattric(x);
  printf("Fattoriale di %d: %d\n", x, f);
}

int fattric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattric(n-1);
  return ris;
}
```

Evoluzione della pila (supponendo $x=3$).

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 2 |
| ris | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 1 |
| ris | ? |

| | |
|-----|---|
| n | 2 |
| ris | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 0 |
| ris | ? |

| | |
|-----|---|
| n | 1 |
| ris | ? |

| | |
|-----|---|
| n | 2 |
| ris | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 0 |
| ris | 1 |

| | |
|-----|---|
| n | 1 |
| ris | ? |

| | |
|-----|---|
| n | 2 |
| ris | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 1 |
| ris | 1 |

| | |
|-----|---|
| n | 2 |
| ris | ? |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 2 |
| ris | 2 |

| | |
|-----|---|
| n | 3 |
| ris | ? |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|-----|---|
| n | 3 |
| ris | 6 |

| | |
|---|---|
| x | 3 |
| f | ? |

| | |
|---|---|
| x | 3 |
| f | 6 |

Esempio: Leggere una sequenza di caratteri terminata da '`\n`' e stamparla invertita. Ad esempio: `casa` \implies `asac`

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
 1. usando una struttura dati opportuna ma **dinamica** (liste, le vedremo più avanti)
 2. usando un procedimento ricorsivo.
 - ▶ leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
 - ▶ il caso base è rappresentato dalla lettura del carattere di fine sequenza.

```
void invertInputRic()
{ char ch;

  ch = getchar();
  if (ch != '\n')
  {
    invertInputRic();
    putchar(ch);
  }
  else
    printf("Sequenza invertita: ");
}
```

```

main()
{
    printf("Immetti una sequenza di caratteri\n");
    invertInputRic();
    printf("\n");
}

```

Vediamo come evolve la pila per l'input `ABC\n`

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | B |
|----|---|

| | |
|----|---|
| ch | C |
|----|---|

| | |
|----|----|
| ch | \n |
|----|----|

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | B |
|----|---|

| | |
|----|---|
| ch | C |
|----|---|

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | B |
|----|---|

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | C |
|----|---|

| | |
|----|---|
| ch | B |
|----|---|

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | B |
|----|---|

| | |
|----|---|
| ch | A |
|----|---|

| | |
|----|---|
| ch | A |
|----|---|

L'output prodotto è il seguente

Sequenza invertita: `CBA`

Ricorsione multipla

- ▶ Si ha ricorsione multipla quando un'attivazione di una funzione può causare **più di una attivazione ricorsiva** della stessa funzione (es. torre di Hanoi)

Esempio: Definizione induttiva dei numeri di Fibonacci.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-2) + F(n-1) \quad \text{se } n > 1$$

- ▶ $F(0), F(1), F(2), \dots$ è detta sequenza dei numeri di Fibonacci:
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
#include <stdio.h>

int fibonacci (int);

main() {
    int n;

    printf("Inserire un intero >= 0: ");
    scanf("%d", &n);
    printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

int fibonacci(int i)
{
    int ris;
    if (i == 0)
        ris = 0;
    else if (i == 1)
        ris = 1;
    else
        ris = fibonacci(i-1) + fibonacci(i-2);
    return ris;
}
```


Esempi di funzioni ricorsive

- ▶ Tradurre in C la definizione induttiva già vista:

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

```
int somma (int x, int y)
{
    int ris;
    if (y==0)
        ris = x;
    else
        ris = 1 + somma(x, y-1);
    return ris;
}
```

- Calcolo ricorsivo di x^y (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \text{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
    int ris;
    if (y==0)
        ris = 1;
    else
        ris = x * exp(x, y-1);
    return ris;
}
```

- ▶ Calcolare ricorsivamente la somma degli elementi di un array v compreso tra gli indici j e $dim-1$.
- ▶ Esprimiamo formalmente quanto richiesto:

$$sumVet(v, j, dim - 1) = \sum_{i=j}^{d-1} v[i]$$

- ▶ È evidente che:

$$\sum_{i=j}^{d-1} v[i] = \begin{cases} 0 & \text{se } dim = 0 \\ v[j] + \sum_{i=j+1}^{d-1} v[i] & \text{se } dim > 0 \end{cases}$$

- ▶ La traduzione in C è immediata.

```
int sumVet(int *v, int dim)
{
    if (dim==0)
        return 0;
    else
        return v[0] + sumvet(v+1,dim-1);
}
```

```
int sumVet(int *v, int dim)
{
    int somma;
    if (dim==0)
        somma = 0;
    else
        somma = v[0] + sumvet(v+1,dim-1);
    return somma;
}
```

- ▶ Calcolare ricorsivamente il numero di occorrenze dell'elemento x nell'array v compreso tra gli indici j e $dim-1$.

$$f(v, x, j, dim) = \#\{i \in [j, dim - 1] \mid v[i] = x\}$$

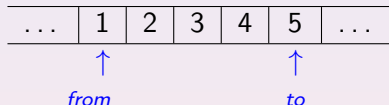
- ▶ Anche in questo caso ragioniamo induttivamente:

$$f(v, x, j, dim) = \begin{cases} 0 & \text{se } dim = 0 \\ f(v, x, j + 1, dim) & \text{se } dim > 0 \wedge v[j] \neq x \\ 1 + f(v, x, j + 1, dim) & \text{se } dim > 0 \wedge v[j] = x \end{cases}$$

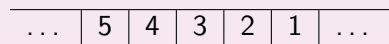
```
int occorrenze (int *v, int dim)
{
    int occ;

    if (dim==0)
        occ= 0;
    else
        if (v[0]!=x)
            occ = occorrenze(v+1,x,dim-1);
        else
            occ = 1+occorrenze(v+1,x,dim-1);
}
```

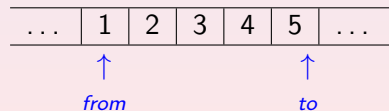
- ▶ Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



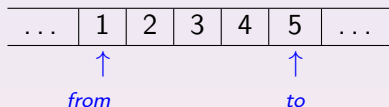
- ▶ Vogliamo ottenere:



- ▶ Induttivamente:



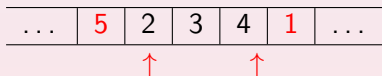
- ▶ Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici *from* e *to*.



- ▶ Vogliamo ottenere:



- ▶ Induttivamente:



- ▶ Questa situazione corrisponde alla chiamata ricorsiva su una porzione più piccola del vettore


```
void swap(int *v, int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void invertiric (int *v, int from, int to)
{
    if (from < to)
    {
        swap(v, from, to);
        invertiric(v, from+1, to-1);
    }
}
```

- ▶ Si noti che la procedura non fa niente se la porzione individuata dal secondo e terzo parametro è vuota ($from > to$) o contiene un solo elemento ($from = to$)