

# Ricorsione

---

Angelo Di Iorio

dal materiale di Stefano Ferretti  
[s.ferretti@unibo.it](mailto:s.ferretti@unibo.it)

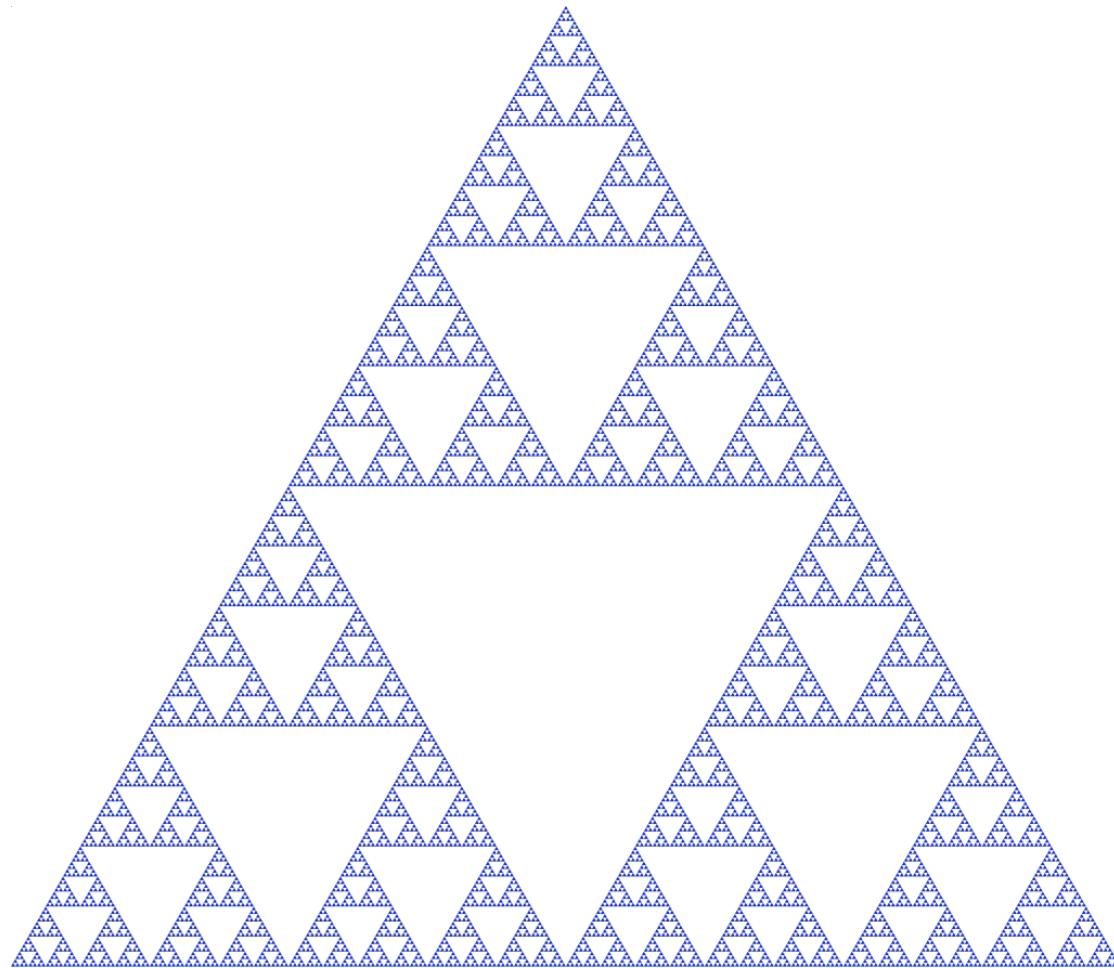
# Ricorsione

- Definizione ricorsiva in matematica:
  - Definizione di un concetto usando il concetto stesso
- Definizione ricorsiva in programmazione (Java):
  - Definizione di un metodo usando il metodo stesso

# Ricorsione in natura: broccolo romanesco



# Triangolo di Sierpinski



By Bejan Stanislaus, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=8862246>

# Rircorsione in arte

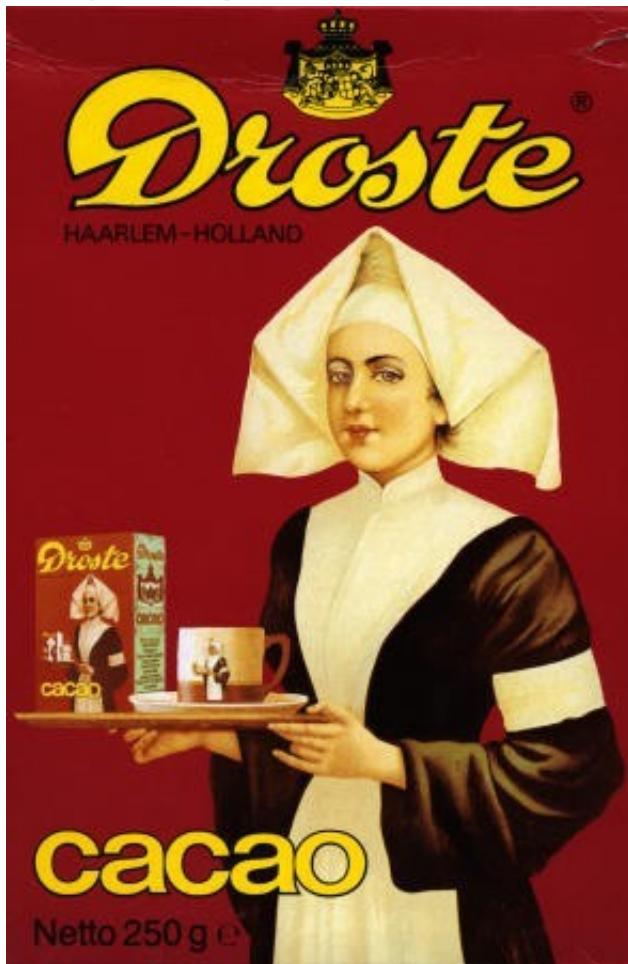


Immagine sulla confezione del cacao Droste (anno 1904); fonte Wikipedia



Copertina dell'album *Ummagumma* dei Pink Floyd (anno 1969); fonte Wikipedia

# Ricorsione in matematica

- Somma (+)
  - $x + 0 = x$
  - $x + \text{succ}(y) = \text{succ}(x + y)$   
(succ è il successore)
- Fattoriale
  - $0! = 1$
  - $n! = n * (n - 1)!$
- Sono **definizioni operative**
  - $4 + 3 = 4 + \text{succ}(2) = \text{succ}(4 + 2) \dots$
  - $5! = 5 * 4! = 5 * \dots$
- Caso base e passo

# Funzione di Fibonacci

- $F(0) = 1$
- $F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$
- $F(5) = F(4) + F(3) = \dots$

# Fattoriale

- $\text{fatt}(0) = 1$
- $\text{fatt}(n) = n * \text{fatt}(n - 1)$
  
- $\text{fatt}(5) = 5 * \text{fatt}(4) = 5 * 24 = 120$
- $\text{fatt}(4) = 4 * \text{fatt}(3) = 4 * 6 = 24$
- $\text{fatt}(3) = 3 * \text{fatt}(2) = 3 * 2 = 6$
- $\text{fatt}(2) = 2 * \text{fatt}(1) = 2 * 1 = 2$
- $\text{fatt}(1) = 1 * \text{fatt}(0) = 1$
- $\text{fatt}(0) = 1$

# Fattoriale in Java

- Funzione ricorsiva per il calcolo del fattoriale
  - $0! = 1$
  - $n! = n * (n - 1)!$

```
int fatt(int n) {  
    if (n == 0)          // caso base  
        return 1;  
    else                // caso ricorsivo  
        return n * fatt(n - 1);  
}
```

# Fibonacci in Java

- Funzione ricorsiva per il calcolo della funzione di Fibonacci

- $F(0) = 1$
- $F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$

```
int F(int n) {  
    if (n == 0 || n == 1)          // caso base  
        return 1;  
    else                          // caso ricorsivo  
        return (F(n - 1) + F(n - 2));  
}
```

# La ricorsione e la pila dei record di attivazione

## ■ Perché la ricorsione funziona

- ❑ Ogni **invocazione/esecuzione di** metodo ha un'allocazione di un record di attivazione sulla pila della JVM
- ❑ I record di attivazione allocati mantengono i risultati intermedi.

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
.  
// poi nel main, o in un altro metodo ...  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	2
res	0

fatt(2) chiama al suo interno fatt(1)

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	1
res	0

fatt(1) chiama al suo interno fatt(0)

n	2
res	0

fatt(2) chiama al suo interno fatt(1)

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	0
res	1

fatt(0) assegna a res il valore 1 e ritorna il valore

n	1
res	0

fatt(1) chiama al suo interno fatt(0)

n	2
res	0

fatt(2) chiama al suo interno fatt(1)

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	1
res	1

res= 1 \* fatt(0), il risultato di fatt(0) era 1  
→ questa funzione ritorna il valore  $1 * 1 = 1$

n	2
res	0

fatt(2) chiama al suo interno fatt(1)

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale – chiamo fatt(3)

n	2
res	2

res = 2 \* fatt(1), il risultato di fatt(1) era 1  
→ questa funzione ritorna il valore  $2 * 1 = 2$

n	3
res	0

fatt(3) chiama al suo interno fatt(2)

```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

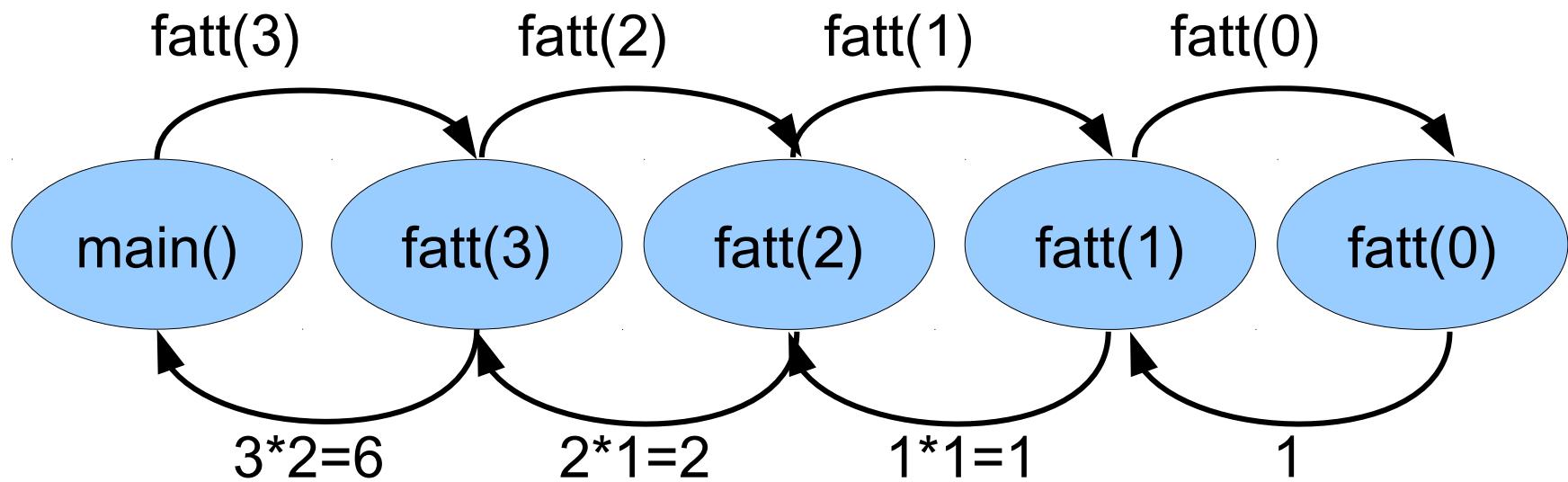
# Fattoriale – chiamo fatt(3)

n	3
res	6

res= 3 \* fatt(2), il risultato di fatt(3) era 2  
→ questa funzione ritorna il valore  $3 * 2 = 6$

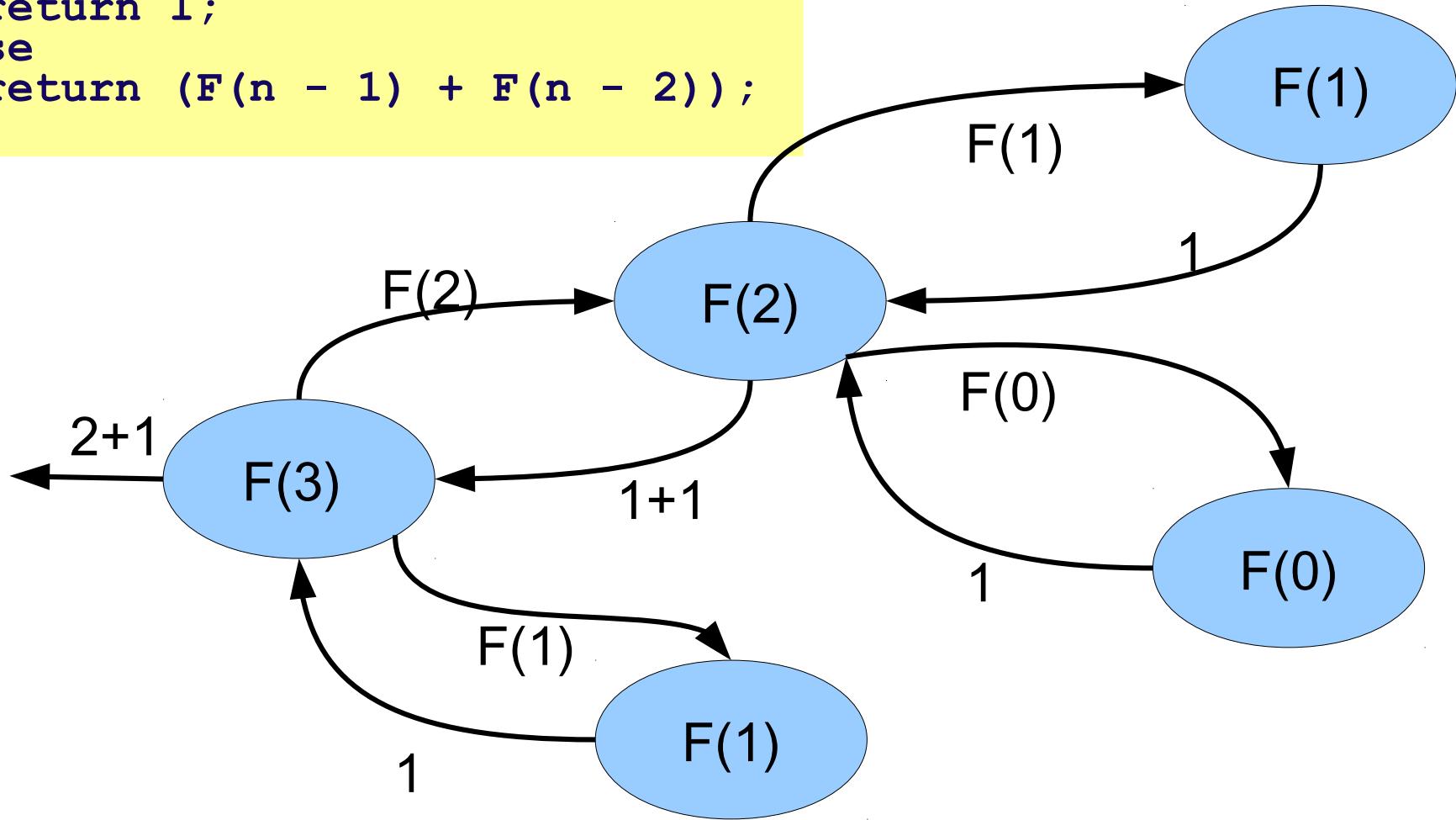
```
int fatt(int n) {  
    int res = 0;  
    if (n == 0)  
        res = 1;  
    else  
        res = n * fatt(n - 1);  
    return res;  
}  
  
int x = 3;  
System.out.println(fatt(x));
```

# Fattoriale: figura alternativa



# Fibonacci: le chiamate

```
int F(int n) {  
    if (n == 0 || n == 1)  
        return 1;  
    else  
        return (F(n - 1) + F(n - 2));  
}
```



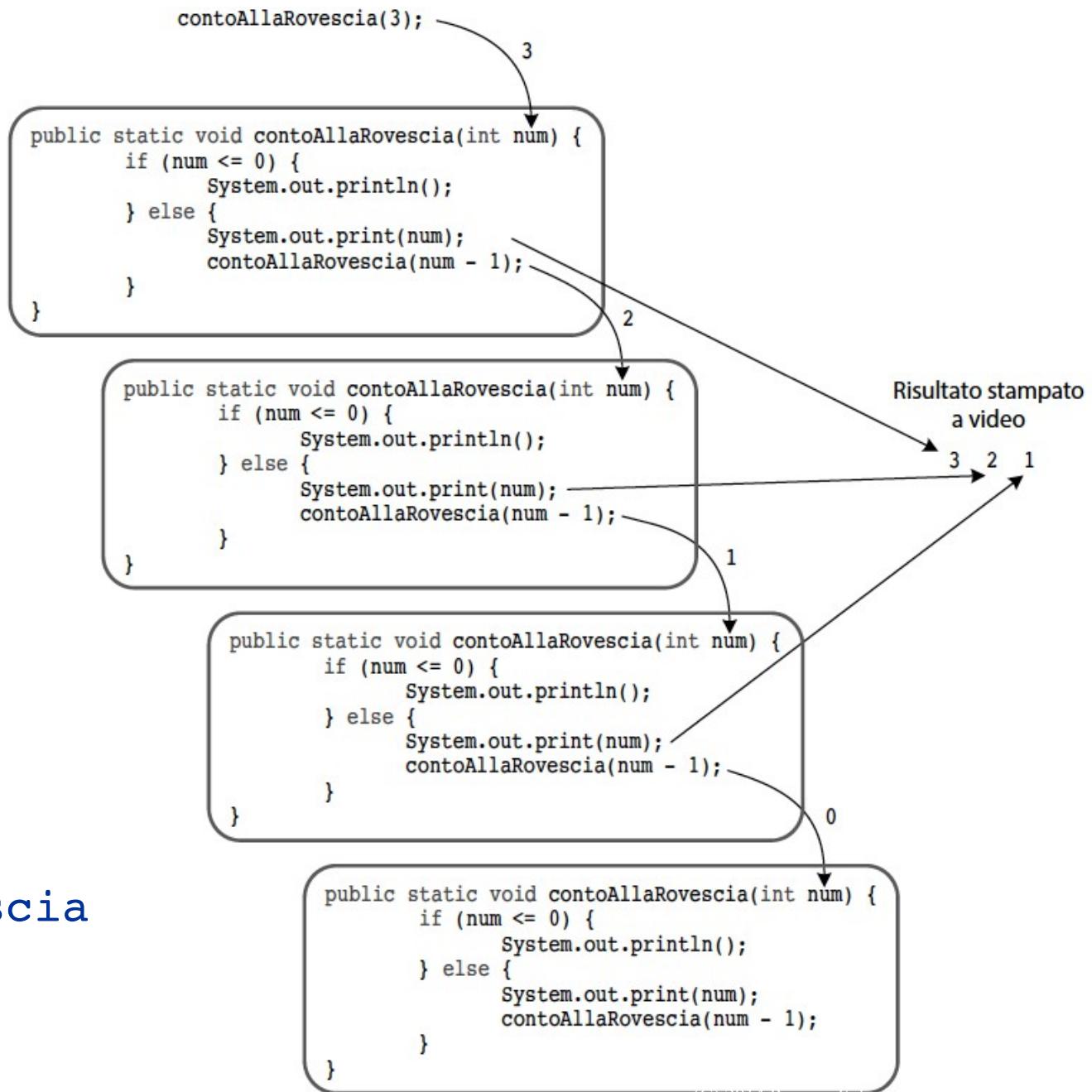
# Conto alla rovescia

```
public class ContoAllaRovesciaRicorsivo {  
    public static void main(String[] args) {  
        contoAllaRovescia(3);  
    }  
  
    public static void contoAllaRovescia(int num) {  
        if (num <= 0) {  
            System.out.println();  
        } else {  
            System.out.print(num);  
            contoAllaRovescia(num - 1);  
        }  
    }  
}
```

## Esempio di output

321

# Chiamate ricorsive per il metodo conto alla rovescia



# Conto alla Rovescia

- Cap 7 Savitch
  - ContoAllaRovescia.java
  - ContoAllaRovesciaRicorsivo.java

# Quando utilizzarla

## ■ PRO

- Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice
- Espressività: A volte la definizione più semplice e naturale di un algoritmo è ricorsiva
  - Es.: algoritmi di visita su alberi... ASD...

## ■ 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, etc)
  - consuma molta memoria (alloca una nuova attivazione nello stack 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

# Somma dei primi N interi

## ■ Algoritmo ricorsivo:

- Se  $N$  vale 1 allora la somma vale 1
- altrimenti la somma vale  $N +$  il risultato della somma dei primi  $N-1$  interi

# Somma dei primi N interi

## ■ Algoritmo ricorsivo:

- Se N vale 1 allora la somma vale 1
- altrimenti la somma vale N + il risultato della somma dei primi N-1 interi

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

# Somma degli elementi di un array

- Progettiamo la soluzione ricorsiva (risolto precedentemente con una semplice iterazione)

```
/* a: array di cui calcolare la somma
   n: contatore per memorizzare dove siamo
       arrivati a calcolare la somma
*/
int somma(int[] a, int n) {
    if (n == 0)
        return 0;
    else
        return a[n-1] + somma(a, n-1);
}
```

## Listato 7.2

(Savitch)

Un esempio  
di ricorsione

# 1/4

```
import java.util.Scanner;
public class RicorsioneDemo1 {
    public static void main(String[] args) {
        System.out.println("Inserire un intero:");
        Scanner tastiera = new Scanner(System.in);
        int numero = tastiera.nextInt();
        System.out.println("Le cifre in questo numero sono:");
        scriviAParole(numero);
        System.out.println();
        System.out.println("Aaggiungendo 10 a questo numero, ");
        System.out.println("le cifre nel nuovo numero sono:");
        numero = numero + 10;
        scriviAParole(numero);
        System.out.println();
    }
}
```

## Listato 7.2 (Savitch)

### Un esempio di ricorsione # 2/4

```
/**  
Precondizione: numero >= 0  
Scrive a parole le cifre del numero.  
*/  
public static void scriviAParole(int numero) {  
    if (numero < 10)  
        System.out.print(daCifraAParola(numero) + " ");  
    else { //numero ha due o più cifre  
        scriviAParole(numero / 10); ← Chiamata ricorsiva  
        System.out.print(daCifraAParola(numero % 10) + " ");  
    }  
}
```

```
// Precondizione: 0 <= cifra <= 9
// Restituisce la parola corrispondente alla cifra passata come parametro.
public static String daCifraAParola(int cifra) {
    String risultato = null;
    switch (cifra) {
        case 0: risultato = "zero"; break;
        case 1: risultato = "uno"; break;
        case 2: risultato = "due"; break;
        case 3: risultato = "tre"; break;
        case 4: risultato = "quattro"; break;
        case 5: risultato = "cinque"; break;
        case 6: risultato = "sei"; break;
        case 7: risultato = "sette"; break;
        case 8: risultato = "otto"; break;
        case 9: risultato = "nove"; break;
        default:
            System.out.println("Errore grave.");
            System.exit(0);
            break;
    }
    return risultato;
}
```

## Listato 7.2 (Savitch) Un esempio di ricorsione # 3/4

## Listato 7.2 (Savitch)

### Un esempio di ricorsione # 4/4

#### Esempio di output

Inserire un intero:

987

Le cifre in questo numero sono:

nove otto sette

Aggiungendo 10 a questo numero,

le cifre nel nuovo numero sono:

nove nove sette

## Figura 7.2 (Savitch) Esecuzione di una chiamata ricorsiva

`scriviAParole(987)` è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(987)
    if (987 < 10)
        System.out.print(daCifraAParola(987) + " ");
    else { //987 ha due o più cifre
        scriviAParole (987 / 10); ← L'esecuzione si ferma qui in attesa del
        System.out.print(daCifraAParola(987 % 10) + " ");
    }
}
```

`scriviAParole(987/10)` è equivalente a `scriviAParole(98)`, che a sua volta è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(98)
    if (98 < 10)
        System.out.print(daCifraAParola(98) + " ");
    else { //98 ha due o più cifre
        scriviAParole(98 / 10); ← L'esecuzione si ferma qui in attesa del
        System.out.print(daCifraAParola(98 % 10) + " ");
    }
}
```

`scriviAParole(98/10)` è equivalente a `scriviAParole(9)`, che a sua volta è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(9)
    if (9 < 10)
        System.out.print(daCifraAParola(9) + " ");
    else { //9 ha due o più cifre
        scriviAParole(9 / 10);
        System.out.print(daCifraAParola(9 % 10) + " ");
    }
}
```

Non si verifica nessun'altra chiamata ricorsiva.

```
import java.util.Scanner;  
  
public class IterativoDemo1 {  
    public static void main(String[] args)  
    <Il resto di main è lo stesso del Listato 7.2.>
```

```
/**  
 * Precondizione: numero >= 0  
 * Scrive a parole le cifre del numero.  
  
public static void scriviAParole(int numero) {  
    int divisore = calcolaPotenzaDiDieci(numero);  
    int prossimo = numero;  
    while (divisore >= 10) {  
        System.out.print(daCifraAParola(prossimo / divisore) + " ");  
        prossimo = prossimo % divisore;  
        divisore = divisore / 10;  
    }  
    System.out.print(daCifraAParola(prossimo / divisore) + " ");  
}
```

## Listato 7.3 Una versione iterativa di scriviAParole # 1/2

## Listato 7.4 (Savitch)

### Un metodo ricorsivo

#### che restituisce un valore # 1/2

```
import java.util.Scanner;

public class RicorsioneDemo2 {
    public static void main(String[] args) {
        System.out.println("Inserire un numero non negativo:");
        Scanner tastiera = new Scanner(System.in);
        int numero = tastiera.nextInt();
        System.out.println(numero + " contiene " +
                           contaNumeroDiZeri(numero) + " zeri.");
    }
}
```

## Listato 7.4 (Savitch)

### Un metodo ricorsivo

#### che restituisce un valore # 1/2

```
/*
Precondizione: n >= 0
Restituisce il numero di zeri in n.
*/
public static int contaNumeroDiZeri(int n) {
    int risultato;
    if (n == 0)
        risultato = 1;
    else if (n < 10)
        risultato = 0; //n ha una sola cifra e questa non è 0
    else if (n % 10 == 0)
        risultato = contaNumeroDiZeri(n / 10) + 1;
    else //n % 10 != 0
        risultato = contaNumeroDiZeri(n / 10);
    return risultato;
```

# Listato 7.4 Un metodo ricorsivo che restituisce un valore # 2/2

## Esempio di output

Inserire un numero non negativo:

2008

2008 contiene 2 zeri.

# Esercizio

- Scrivere un metodo **ricorsivo** che prende in input due interi  $x$  ed  $n$  e calcola la potenza  $x^n$
- Definiamo la classe EsercizioRicorsione con un `main()` che invoca il metodo precedente

```
public class RicorsioneDemo3 {  
    public static void main(String args[]) {  
        for (int n = 0; n < 4; n++)  
            System.out.println("3 alla " + n  
                + " e' uguale a " + potenza(3, n));  
    }  
  
    public static int potenza(int x, int n) {  
        if (n < 0) {  
            System.out.println("Argomento non ammesso per potenza.");  
            System.exit(0);  
        }  
  
        if (n > 0)  
            return (potenza(x, n - 1) * x);  
        else // n == 0  
            return (1);  
    }  
}
```

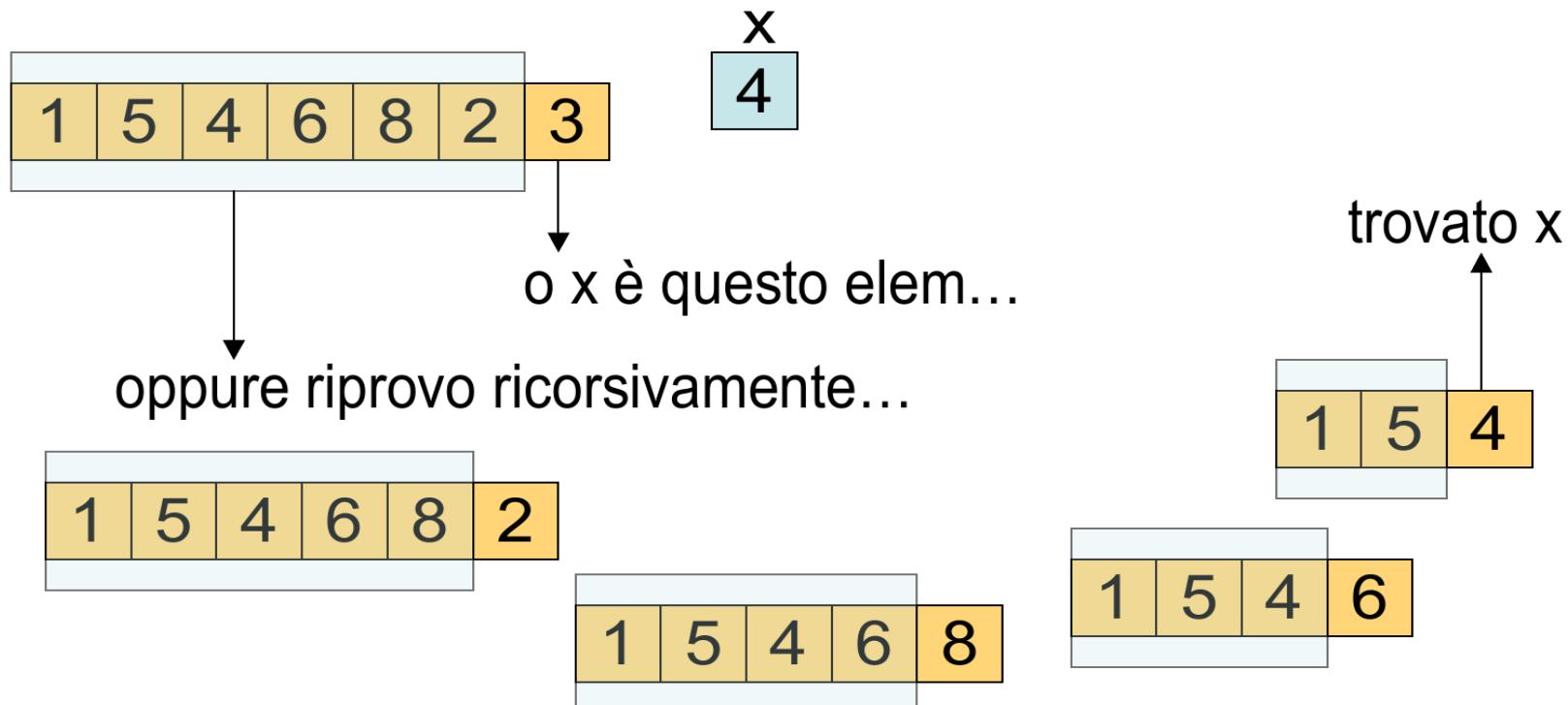
## Esempio di output

```
3 alla 0 e' uguale a 1  
3 alla 1 e' uguale a 3  
3 alla 2 e' uguale a 9  
3 alla 3 e' uguale a 27
```

## Listato 7.5 Il metodo ricorsivo potenza

# Ricerca lineare

- Scrivere un metodo ricorsivo che, dato un array di interi ed un intero  $x$  restituisca true se  $x$  è tra gli elementi dell'array, false altrimenti

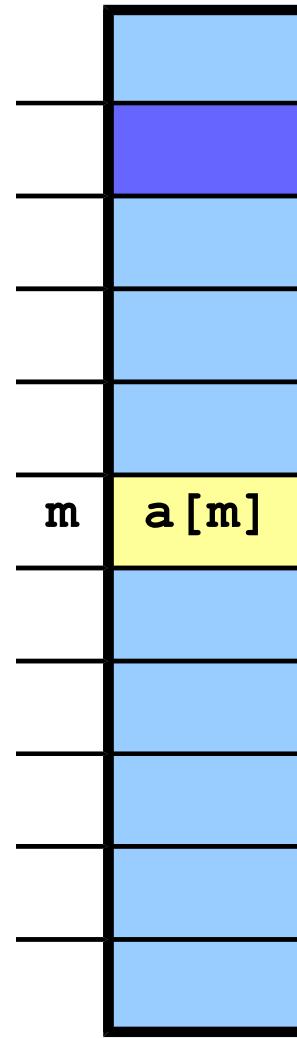


# Ricerca lineare

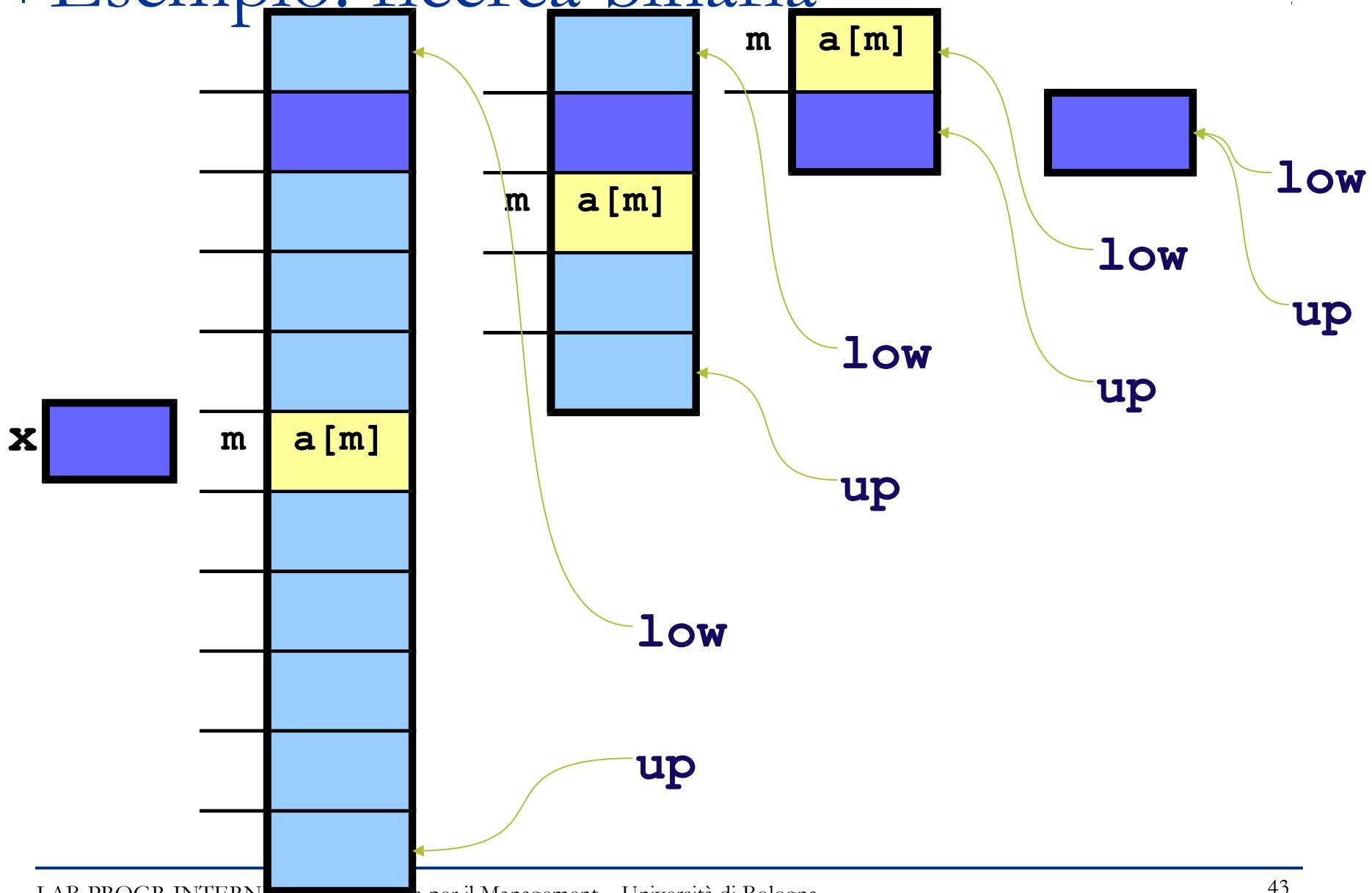
```
public boolean ricercaLineare(int[] vett, int n,  
                             int x)  
{  
    if (n==1)  
        return (vett[0]==x);  
    else  
        return ( (vett[n-1]==x) ||  
                 (ricercaLineare(vett, n-1, x)) );  
}
```

```
// la chiamata sarà così  
int[] a = ...;  
int valoreDaCercare = ...;  
boolean trovato = ricercaLineare(a, a.length,  
                                   valoreDaCercare);
```

# Come cercare un elemento in un array ordinato?



# Esempio: ricerca binaria



# Figura 7.4 Un esempio di ricerca binaria

obiettivo è uguale a 33

Elimina metà degli elementi dell'array:

0	1	2	3	4	5	6	7	8	9
5	7	9	13	32	33	42	54	56	88

1.  $\text{med} = (0 + 9)/2$  (uguale a 4)
2.  $33 > a[\text{med}]$  (cioè  $33 > a[4]$ )
3. Quindi se 33 è nell'array, 33 è uno tra  $a[5], a[6], a[7], a[8], a[9]$ .

Elimina metà degli elementi rimanenti dell'array:

5	6	7	8	9
33	42	54	56	88

1.  $\text{med} = (5 + 9)/2$  (uguale a 7)
2.  $33 < a[\text{med}]$  (cioè  $33 < a[7]$ )
3. Quindi se 33 è nell'array, 33 è uno tra  $a[5], a[6]$ .

Elimina metà degli elementi rimanenti dell'array:

5	6
33	42

1.  $\text{med} = (5 + 6)/2$  (uguale a 5)
2. 33 è uguale a  $a[\text{med}]$ .  
Così 33 si trova all'indice 5.

33 trovato in  $a[5]$

# Il codice

```
/*a: array dove sono gli elementi
 x: valore da cercare
 low, up: indici minore e maggiore dell'array
 Restituisce -1 o la posizione in cui si trova x
*/
int ricerca (char[] a, char x,
              int low, int up) {
    int m;
    m = (up + low) / 2;

    if (up < low)
        return -1;
    else if (a[m] == x)
        return m;
    else if (a[m] < x)
        return ricerca(a, x, m + 1, up);
    else //a[m] > x
        return ricerca(a, x, low, m - 1);

}
```

# Tutti maggiori di 10

- Scrivere un metodo ricorsivo che, avendo in input un array di n interi, dia in output TRUE se tutti gli elementi sono maggiori di 10, FALSE altrimenti.
- Algoritmo ricorsivo
  - (Caso Base) Se l'array è vuoto allora tutti gli elementi della lista sono maggiori di 10
  - (Caso Base) Se l'array non è vuoto e se il primo elemento  $a_1$  dell'array è minore o uguale a 10, allora **non** tutti gli elementi della lista sono maggiori di 10
  - (Passo generico) Se l'array  $[a_1, \dots, a_n]$  non è vuoto e se  $a_n > 10$ , tutti gli elementi della lista sono maggiori di 10 se e solo se tutti gli elementi della lista  $[a_1, \dots, a_{n-1}]$  sono maggiori di 10.

# Tutti maggiori di 10

```
boolean tuttiMaggiori(int a[], int n) {  
    if (n==0)  
        return true;  
    else  
        if (a[n] <= 10 )  
            return false;  
        else  
            return tuttiMaggiori(a, n-1);  
}
```

# Esercizio: tutti maggiori di NUM

```
boolean tuttiMaggiori(int a[], int n, int NUM)
{
    if (n==0)
        return true;
    else
        if (a[n] <= NUM )
            return false;
        else
            return tuttiMaggiori(a, n-1, NUM);
}
```

# Palindrome

# Pensare ricorsivamente

- Problema: verificare se una frase è una *palindroma*
- *Palindroma*: una stringa uguale a se stessa quando ne invertite l'ordine dei caratteri
  - A man, a plan, a canal – Panama!
  - Go hang a salami, I'm a lasagna hog
  - Madam, I'm Adam

# Realizzare il metodo isPalindrome

```
public class Sentence {  
  
    // la stringa da usare è una var istanza  
    private String text;  
  
    // metodo che verifica se è palindrome  
    public boolean isPalindrome(int start, int end) {  
        . . .  
    }  
  
    ... //da qualche parte  
    boolean verifica = isPalindrome(0, text.length() -  
        1);  
  
}
```

# Pensare ricorsivamente

- Combinate le soluzioni dei casi più semplici per fornire una soluzione al problema originario
  - Semplifichiamo eliminando il primo e l'ultimo carattere.
  - Se elimino i due caratteri da
    - “Madam, I’m Adam”ottengo
    - “adam, I’m Ada”: anche questo è una palindrome
- Una parola è una palindrome se
  - la prima e l’ultima lettera sono uguali (tralasciando le differenze tra maiuscole e minuscole)
  - e
  - la parola che si ottiene eliminando la prima e l’ultima lettera è una palindrome

# Metodo ricorsivo (semplificato): isPalindrome

```
public boolean isPalindrome(int start, int end)
{
    // considera separatamente i casi delle stringhe più brevi
    if (start >= end) return true;

    // prendi primo e ultimo carattere
    char first = text.charAt(start));
    char last = text.charAt(end);

    if (first == last) {
        // verifica la sottostringa che non contiene
        // le due lettere uguali
        return isPalindrome(start + 1, end - 1);
    }
    else
        return false;
}
```

# Metodo ricorsivo: isPalindrome

- La soluzione precedente non considera la possibilità che nella stringa ci siano
  - Spazi (che sarebbero da non considerare)
  - Caratteri di interpunzione
  - Lettere minuscole e maiuscole
- Provare a svolgerlo come esercizio

# Creazione ricorsiva di oggetti

- Negli esempi precedenti abbiamo richiamato in maniera ricorsiva uno stesso metodo
  - Il metodo (ex: `fatt()`) richiamava sé stesso passando come parametro un valore più piccolo
- Possiamo anche in realtà far sì che un oggetto crei una nuova istanza di un oggetto atto a risolvere un sotto-problema
  - Vediamo un esempio ...

# Esempio: Permutazioni

# Permutazioni

- Progettiamo una classe che elenchi tutte le permutazioni di una stringa
- Una permutazione è una qualsiasi disposizione delle lettere
- La stringa “eat” ha sei permutazioni (compresa la stringa stessa)

“eat”

“eta”

“aet”

“ate”

“tea”

“tae”

# Stringhe permutate

```
public class PermutationGenerator {  
  
    public PermutationGenerator(String aWord)  
    { . . . }  
    public ArrayList<String> getPermutations()  
    { . . . }  
}
```

Sequenza di dimensioni variabili, ci torneremo.

Per aggiungere elementi alla sequenza:

```
ArrayList<String> words = new  
ArrayList<String>();  
  
words.add("eat");  
words.add("eta");
```

# File

## PermutationGeneratorDemo.java

```
01: import java.util.ArrayList;
02:
03: /**
04:  * Questo programma utilizza il generatore di permutazioni.
05: */
06: public class PermutationGeneratorDemo
07: {
08:     public static void main(String[] args)
09:     {
10:         PermutationGenerator generator
11:             = new PermutationGenerator("eat");
12:         ArrayList<String> permutations
13:             = generator.getPermutations();
14:         for (String s : permutations)
15:         {
16:             System.out.println(s);
17:         }
18:     }
19: }
```

Altro modo per dire: fai un  
ciclo considerando tutti gli  
elementi **s** (stringa)  
dell'ArrayList  
**permutations**

File

PermutationGeneratorDemo.java

Visualizza

```
eat  
eta  
aet  
ate  
tea  
tae
```

# Generare le permutazioni ricorsive

- Generare tutte le permutazioni che iniziano con la lettera ‘e’ , poi quelle che iniziano con ‘a’ , infine quelle che iniziano con ‘t’
- Per generare le permutazioni che iniziano con ‘e’ abbiamo bisogno di conoscere le permutazioni della sottostringa “at”
- Questo non è altro che il problema precedente con un dato di ingresso più semplice
- Usiamo quindi la ricorsione

# Generare le permutazioni

- **getPermutations**: scriviamo un ciclo che prenda in esame tutte le posizioni all'interno della parola che deve essere permutata
- Per ciascuna posizione  $i$ , calcoliamo la parola più breve che si ottiene eliminando il carattere  $i$ -esimo

```
String shorterWord = word.substring(0, i)  
+ word.substring(i + 1);
```

# Generare le permutazioni

- Costruiamo poi un generatore di permutazioni che fornisca le permutazioni di tale parola più breve:

```
PermutationGenerator shorterPermutationGenerator  
= new PermutationGenerator(shorterWord);  
  
ArrayList<String> shorterWordPermutations  
= shorterPermutationGenerator.getPermutations();
```

# Generare le permutazioni

- Infine, aggiungiamo il carattere precedentemente escluso a tutte le permutazioni della parola più

```
for (String s : shorterWordPermutations)
{
    result.add(word.charAt(i) + s);
}
```

Altro modo per dire: fai un ciclo considerando tutti gli elementi **s** (stringa) dell'ArrayList **shortWordPermutations**

- Caso speciale: la più semplice stringa possibile è la stringa vuota, che ha un'unica permutazione: se stessa.

# File PermutationGenerator.java

```
01: import java.util.ArrayList;
02:
03: /**
04: Questa classe genera le permutazioni di una parola.
05: */
06: public class PermutationGenerator
07: {
08:     /**
09:         Costruisce un generatore di permutazioni.
10:         @param aWord la parola da permutare
11:     */
12:     public PermutationGenerator(String aWord)
13:     {
14:         word = aWord;
15:     }
16:
```

Continua...

# File PermutationGenerator.java

```
17:     /**
18:      * Fornisce tutte le permutazioni della parola.
19:      * @return un vettore contenente tutte le permutazioni
20:     public ArrayList<String> getPermutations()
21:     {
22:         ArrayList<String> result = new ArrayList<String>();
23:
24:         // La stringa vuota ha un'unica permutazione: se stessa
25:         if (word.length() == 0)
26:         {
27:             result.add(word);
28:             return result;
29:         }
30:
31:         // effettua un ciclo su tutti i caratteri della stringa
32:         for (int i = 0; i < word.length(); i++)
33:         {
```

Continua...

# File PermutationGenerator.java

```
34: // Componi una parola più breve eliminando il car. i-esimo
35:     String shorterWord = word.substring(0, i)
36:         + word.substring(i + 1);
37:
38:     // Genera tutte le permutazioni della parola più breve
39:     PermutationGenerator shorterPermutationGenerator
40:         = new PermutationGenerator(shorterWord);
41:     ArrayList<String> shorterWordPermutations
42:         = shorterPermutationGenerator.getPermutations();
43:
44:     // Aggiungi il carattere escluso all'inizio di ciascuna
45:     // permutazione della parola più breve
46:     for (String s : shorterWordPermutations)
47:     {
48:         result.add(word.charAt(i) + s);
49:     }
50: }
```

Continua...

# File PermutationGenerator.java

```
51:         // Restituisce tutte le permutazioni
52:         return result;
53:     }
54:
55:     private String word;
56: }
```

# Riferimenti

- Lucidi del Libro di Riferimento
- <http://java.sun.com/>