

Lezione XII

Ricorsione

Scopo della lezione

- Definire e implementare metodi ricorsivi

Ricorsione

- Dal dizionario Garzanti (<http://www.garzantilinguistica.it>)

Lemma

ricorsivo

Etimologia

Deriv. di *ricorrere*, sul modello del fr. *récuratif*

Definizione

agg.

1 (*mat.*) si dice di una successione di funzioni ognuna delle quali si ricavi dalla precedente

Una definizione più operativa

:)

Lemma

ricorsivo

Etimologia

Deriv. di *ricorrere*, sul modello del fr. *récuratif*

Definizione

agg.

vedere *ricorsivo*

E per gli informatici?

- Un metodo si dice **ricorsivo** quando la sua esecuzione può prevedere una chiamata a esso stesso
 - in modo **diretto**, i.e. il corpo del metodo prevede una chiamata al metodo stesso
 - in modo **indiretto**, i.e. il corpo del metodo prevede chiamate a metodi che a loro volta chiamano il metodo originario

Esempio

- Cosa succede durante l'esecuzione del seguente metodo `main()` ?

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

viene inizializzata
la variabile statica
n al valore 0 ...

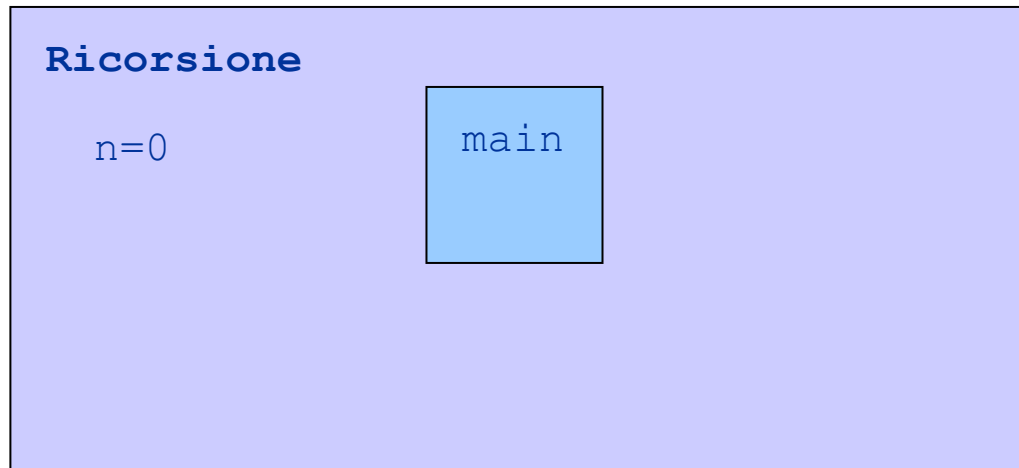
Ricorsione

n=0

Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

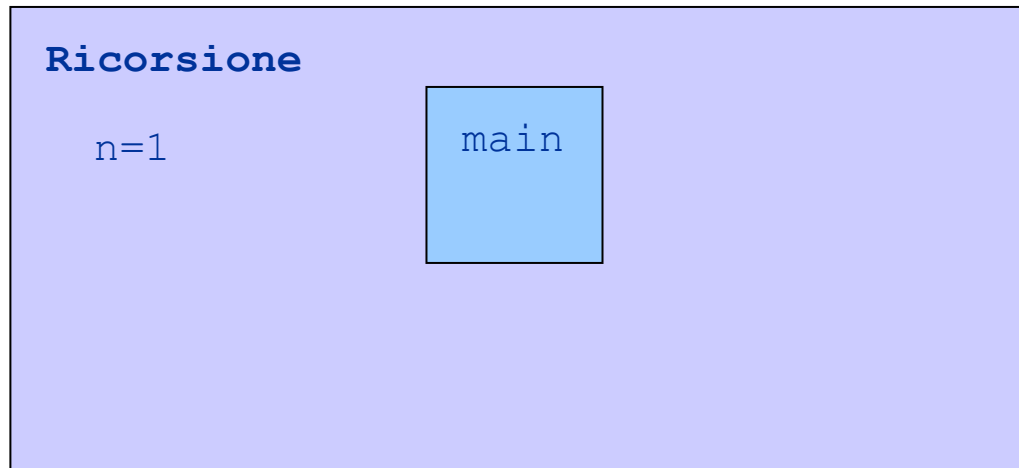
... viene eseguito il
metodo `main` ...



Esecuzione

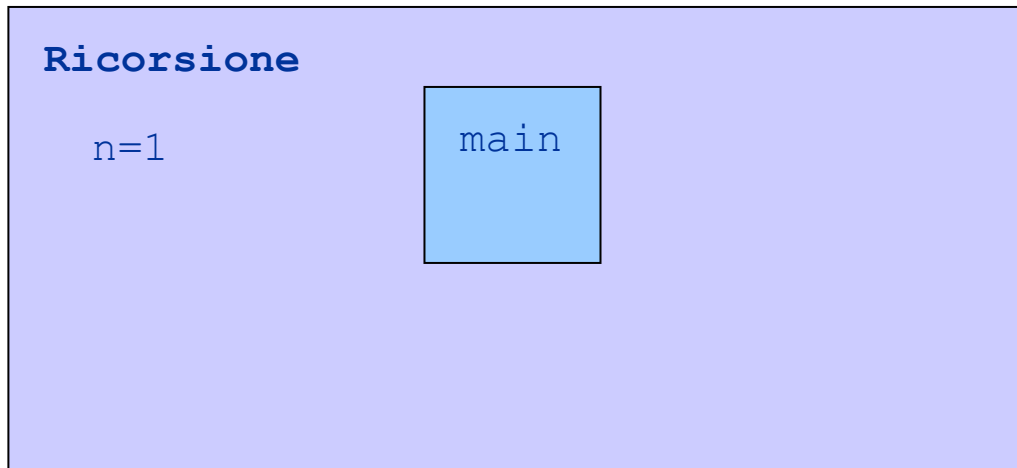
```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

... che incrementa
il valore di n ...



Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

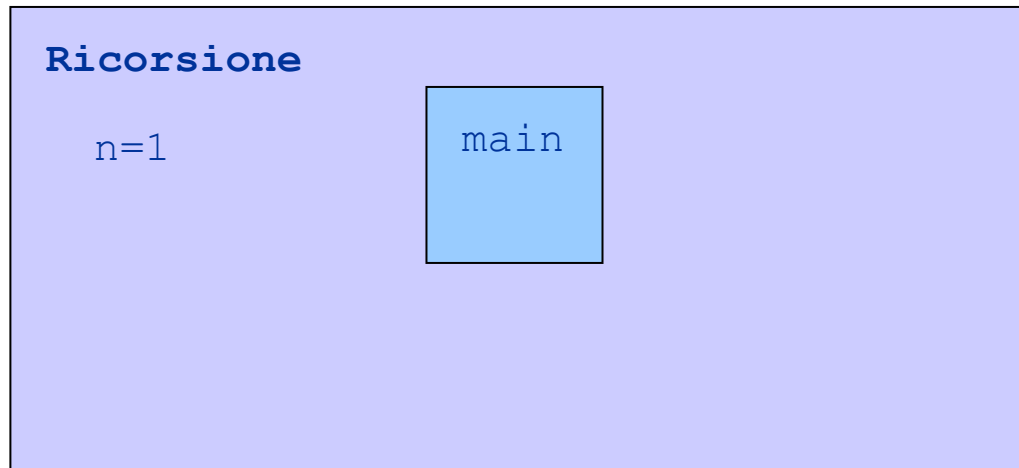


... viene salvato lo “stato” corrente dell’esecuzione (ultima istruzione eseguita, valori delle variabili locali, etc.) ...

Esecuzione

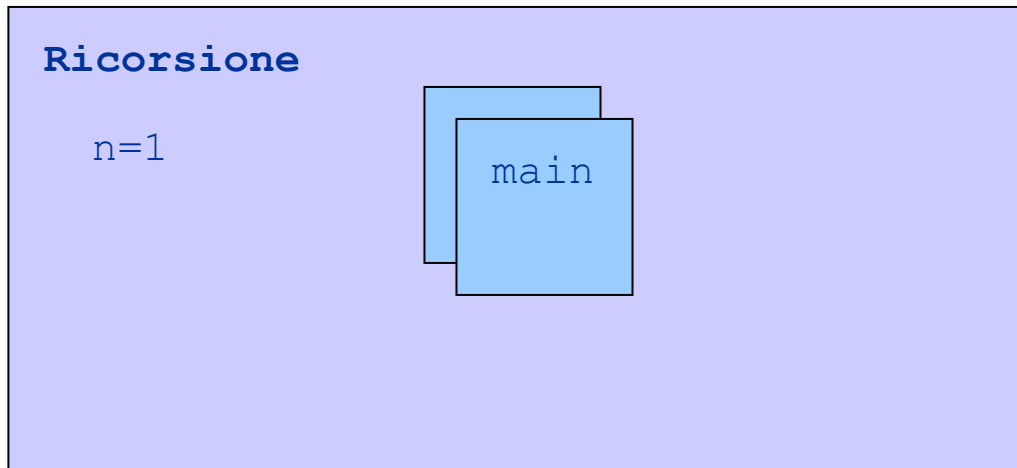
```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

ed eseguita la
chiamata ricorsiva



Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```

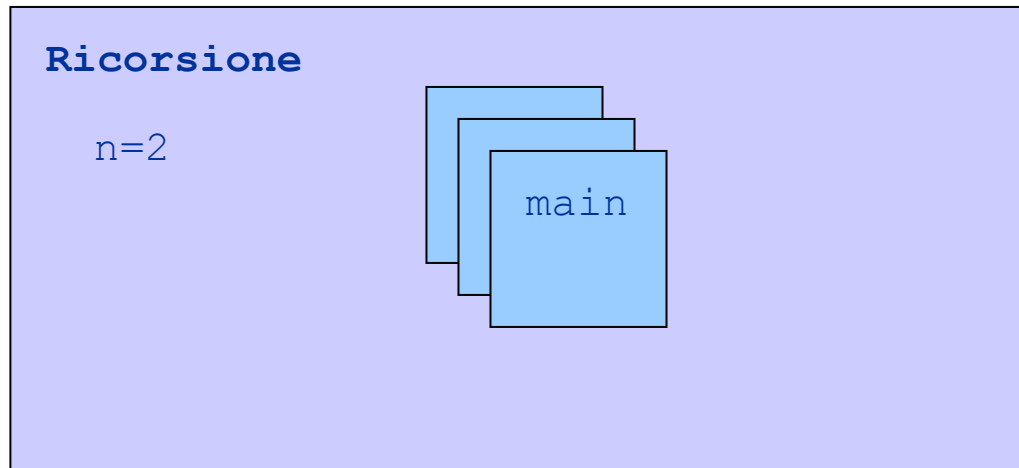


... ricominciando da capo l'esecuzione di main.

Va notato che il metodo abbandonato rimane in uno stato "sospeso", pronto a continuare la sua esecuzione

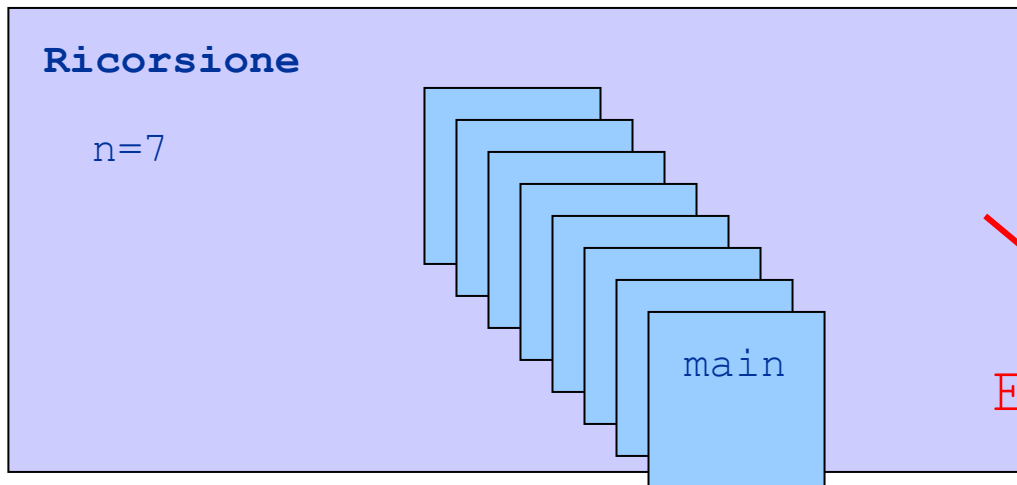
Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```



Esecuzione

```
class Ricorsione {  
    static int n=0;  
    public static void main(String[] args) {  
        n++;  
        main(args);  
    }  
}
```



Fino a che nella macchina virtuale non c'è più spazio per allocare nuove copie del metodo `main`. In quel momento viene generata un'eccezione.

Eccezione

Prova di esecuzione

```
class Ricorsione {
    static int n=0;
    public static void main(String[] args) {
        n++;
        System.out.println(n);
        main(args);
    }
}
malchiod% java Ricorsione
1
2
...
Exception in thread "main" java.lang.StackOverflowError
```

Chiamate ricorsive

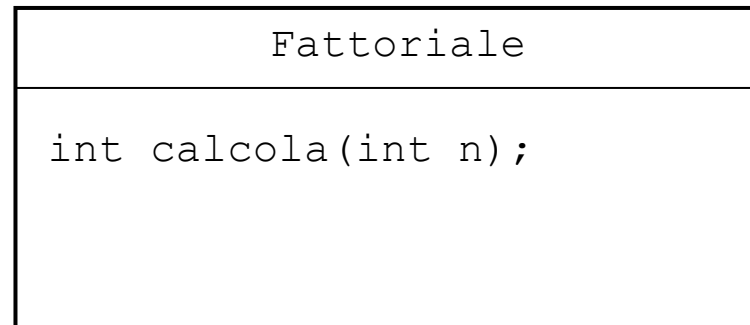
- Il fatto che lo stato locale del metodo venga salvato prima di ogni chiamata ricorsiva evita che ci siano interferenze tra diverse istanze dello stesso metodo

Attenzione!

- Ma l'esecuzione non controllata di metodi ricorsivi è potenzialmente pericolosa
- La progettazione di metodi ricorsivi deve quindi prevedere che la catena di chiamate vada a termine

Esercizio 1-1

- Implementare una classe `Fattoriale` che calcola ricorsivamente il fattoriale di un numero



difficoltà

Idea

- Sfruttiamo la definizione ricorsiva di fattoriale

$$\text{fattoriale}(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \text{ fattoriale}(n - 1) & \text{altrimenti} \end{cases}$$

La classe Fattoriale

```
class Fattoriale {
    public static int calcola(int n) {
        int ritorno;
        if(n<=1)
            ritorno = 1;
        else
            ritorno = n * calcola(n-1);
        return ritorno;
    }
    public static void main(String args[]) {
        int n = Integer.parseInt(args[0]);
        System.out.println("fatt("+n+")="
            +calcola(n));
    }
}
```

Esecuzione

```
% javac Fattoriale.java
% java Fattoriale 4
fatt(4)=24
% java Fattoriale 10
fatt(10)=3628800
% java Fattoriale 20
fatt(20)=-2102132736
%
```

Sperimentiamo

- Perché non si riesce a calcolare il fattoriale di 20?
- Modificate il programma in modo da riuscire a evitare questo problema.

Esercizio 1-2

- Modificare il programma in modo che visualizzi quanti e quali chiamate vengono effettuate al metodo ricorsivo

Fattoriale
<pre>int calcola(int n);</pre>

difficoltà

Idea

- Aggiungiamo la stampa di opportuni messaggi all'inizio e alla fine della procedura

La classe Fattoriale

```
public static int calcola(int n) {
    int ritorno;
    System.out.println(
        "Inizio chiamata per n=" + n);
    if(n<=1)
        ritorno = 1;
    else ritorno = n * calcola(n-1);
    System.out.println("Fine chiamata per
        n=" + n + " (valore ritornato: " +
        ritorno + ")");
    return ritorno;
}
```

Esecuzione

```
% java Fattoriale 5
Inizio chiamata per n=5
Inizio chiamata per n=4
Inizio chiamata per n=3
Inizio chiamata per n=2
Inizio chiamata per n=1
Fine chiamata per n=1 (valore ritornato: 1)
Fine chiamata per n=2 (valore ritornato: 2)
Fine chiamata per n=3 (valore ritornato: 6)
Fine chiamata per n=4 (valore ritornato: 24)
Fine chiamata per n=5 (valore ritornato: 120)
fatt(5)=120
%
```

Sperimentiamo

- La leggibilità dell'output migliora se si riesce a indentare i messaggi in apertura e chiusura di ogni chiamata;
- Riuscite a modificare il codice in modo che i messaggi relativi all'argomento n vengano preceduti da n caratteri separatori (spazio, trattino, ...)?

Esercizio 1-3

- Indentare i messaggi stampati nell'esercizio 1-2

Fattoriale
<pre>int calcola(int n);</pre>

difficoltà

La classe Fattoriale

```
public static int calcola(int n) {  
    for(int i=0;i<n;i++)  
        System.out.print("-");  
    System.out.println("Inizio chiamata per n=" + n);  
    int ritorno;  
    if(n<=1)  
        ritorno = 1;  
    else  
        ritorno = n * calcola(n-1);  
    for(int i=0;i<n;i++)  
        System.out.print("-");  
    System.out.println("Fine chiamata per n=" +  
        +n+" (valore ritornato: "+ritorno+"");  
    return ritorno;  
}
```

Esecuzione

```
% java Fattoriale 5
----Inizio chiamata per n=5
----Inizio chiamata per n=4
---Inizio chiamata per n=3
--Inizio chiamata per n=2
-Inizio chiamata per n=1
-Fine chiamata per n=1 (valore ritornato: 1)
--Fine chiamata per n=2 (valore ritornato: 2)
---Fine chiamata per n=3 (valore ritornato: 6)
----Fine chiamata per n=4 (valore ritornato: 24)
-----Fine chiamata per n=5 (valore ritornato: 120)
fatt(5)=120
%
```

I numeri di Fibonacci

- Nel 1223 a Pisa si svolse una gara tra abachisti e algoritmisti. Il quesito era
Quante coppie di conigli si ottengono in un anno - salvo i casi di morte- supponendo che ogni coppia dia alla luce un'altra coppia ogni mese e che la riproduzione inizi al secondo mese di vita?
- Leonardo Pisano, detto Fibonacci, vinse la gara indicando come soluzione una sequenza il cui generico elemento era pari alla somma dei due precedenti.

I numeri di Fibonacci (2)

$$f(1) = 1$$

$$f(2) = 1$$

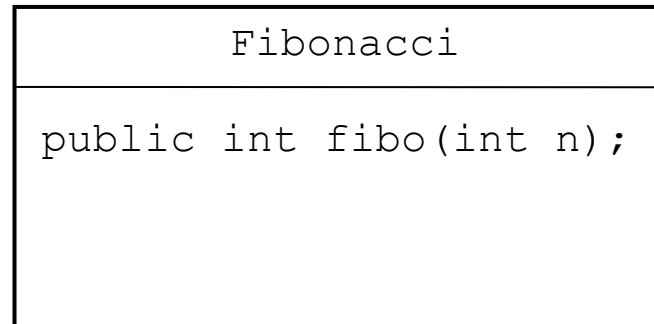
$$f(n) = f(n-1) + f(n-2) \quad \forall n > 2$$

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817
39088169 63245986 102334155 165580141 267914296 433494437
701408733 1134903170 1836311903 2971215073 4807526976 7778742049
12586269025 20365011074 32951280099 53316291173 86267571272
139583862445 225851433717 365435296162 591286729879 956722026041
1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853
72723460248141 117669030460994 190392490709135 308061521170129
498454011879264 806515533049393 1304969544928657
2111485077978050 3416454622906707 5527939700884757

Esercizio 2-1

- Implementare una classe che stampi un generico numero di Fibonacci

La classe Fibonacci



difficoltà

La classe Fibonacci

```
class Fibonacci {
    public static int fibo(int n) {
        int ritorno;
        if(n<=1)
            ritorno = n;
        else
            ritorno = fibo(n-1) + fibo(n-2);
        return ritorno;
    }
    public static void main(String args[]) {
        int n = Integer.parseInt(args[0]);
        System.out.println("fib(" + n + ")=" + fibo(n));
    }
}
```

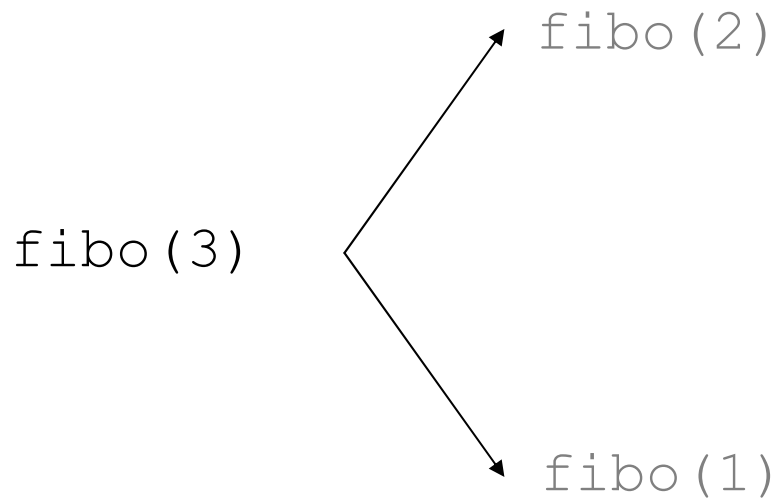
Esecuzione

```
% javac Fibonacci.java  
% java Fibonacci 5  
fib(5)=5  
%
```

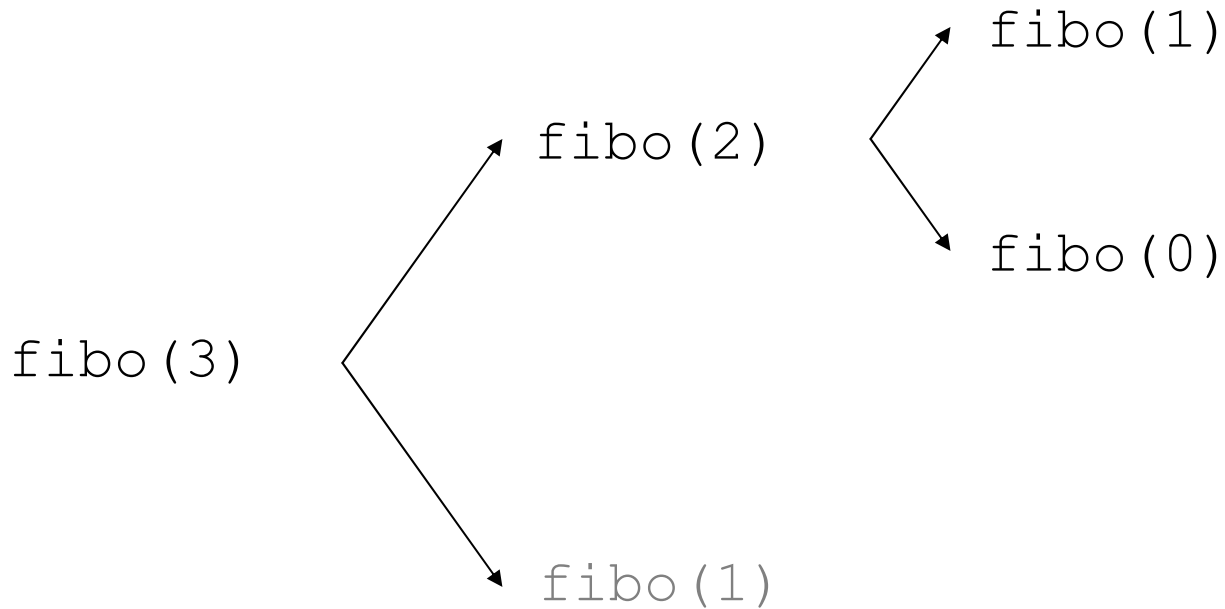
Albero di esecuzione

`fibonacci(3)`

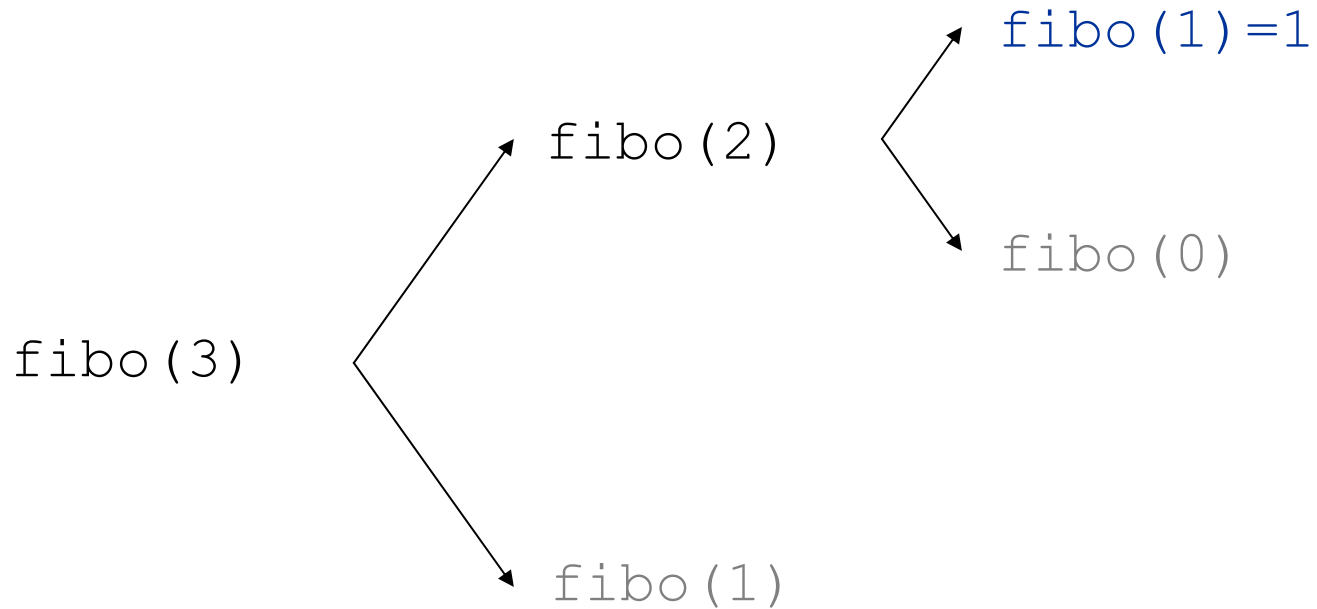
Albero di esecuzione



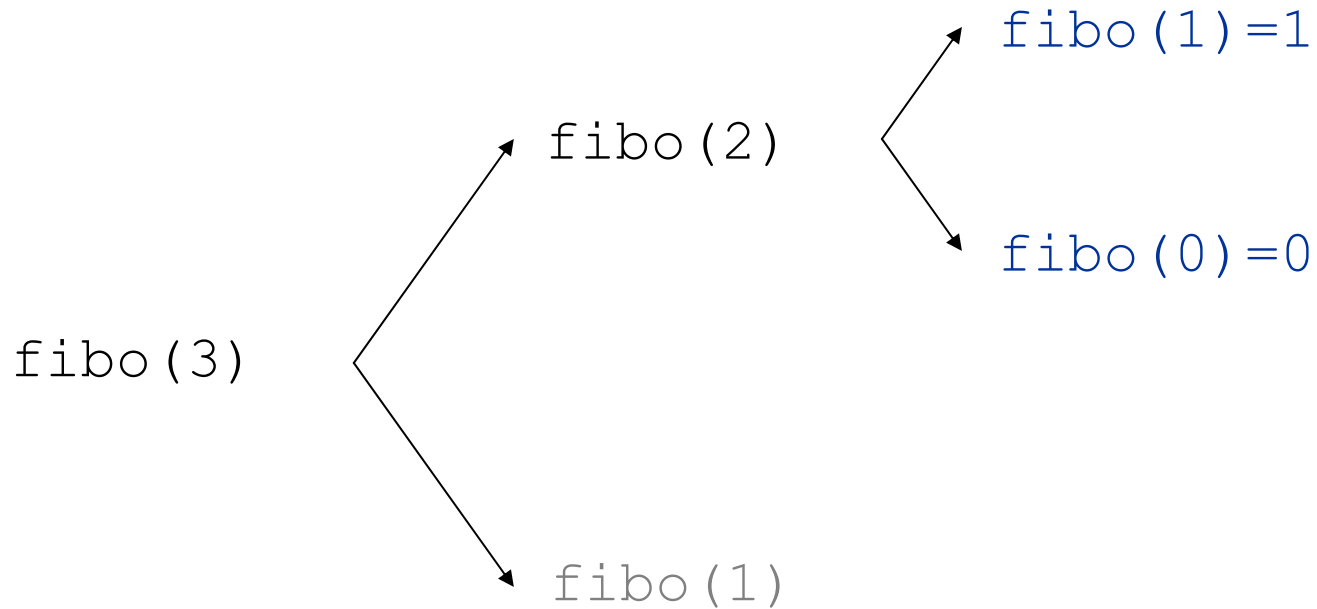
Albero di esecuzione



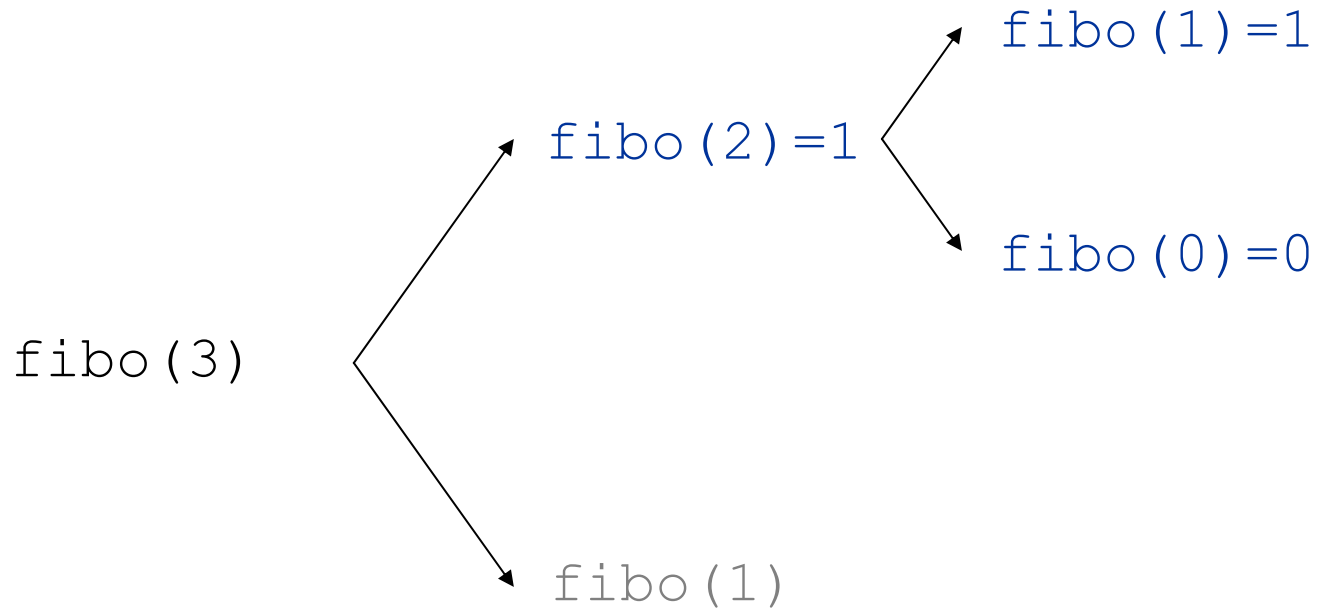
Albero di esecuzione



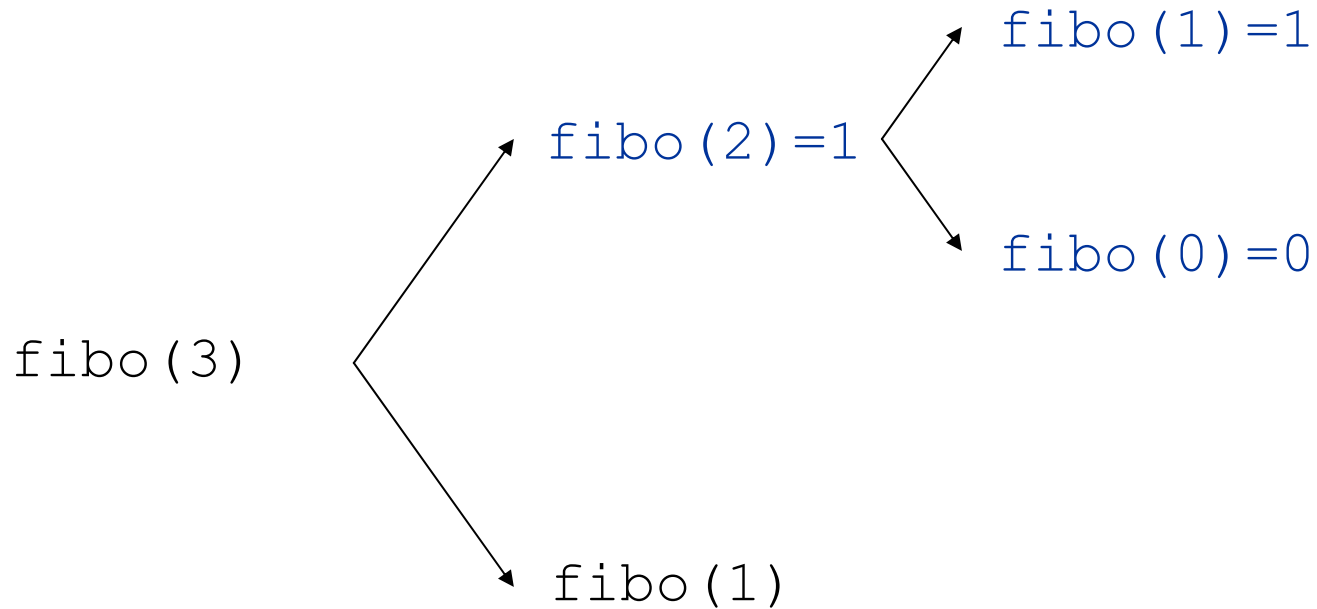
Albero di esecuzione



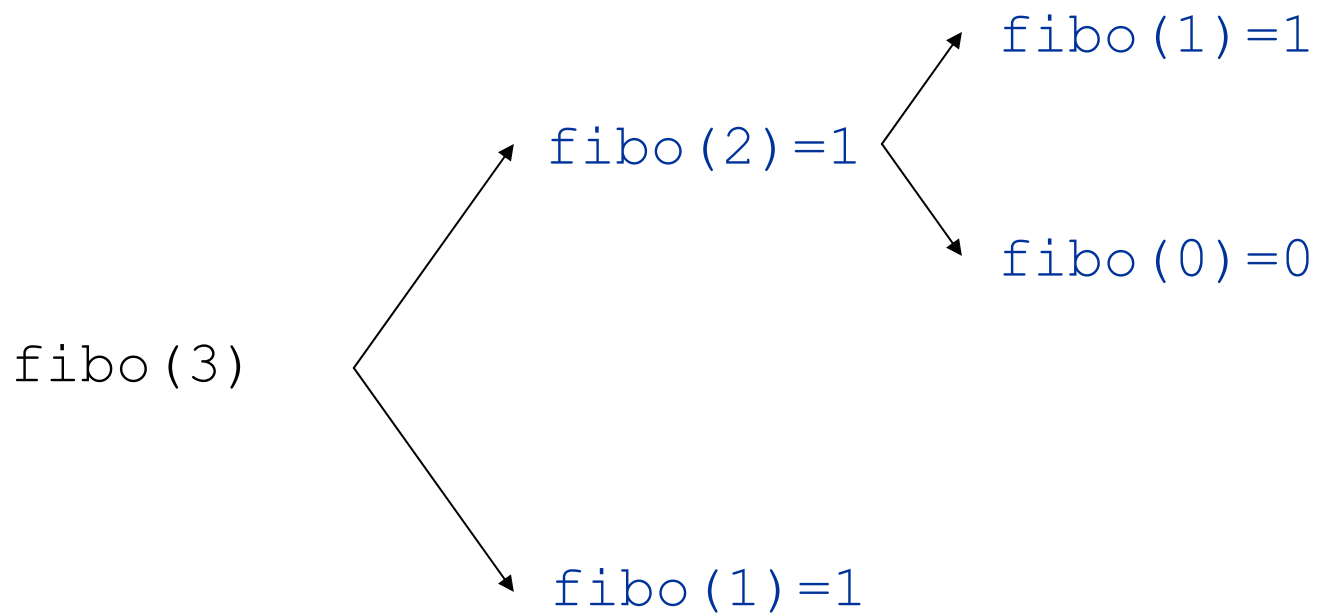
Albero di esecuzione



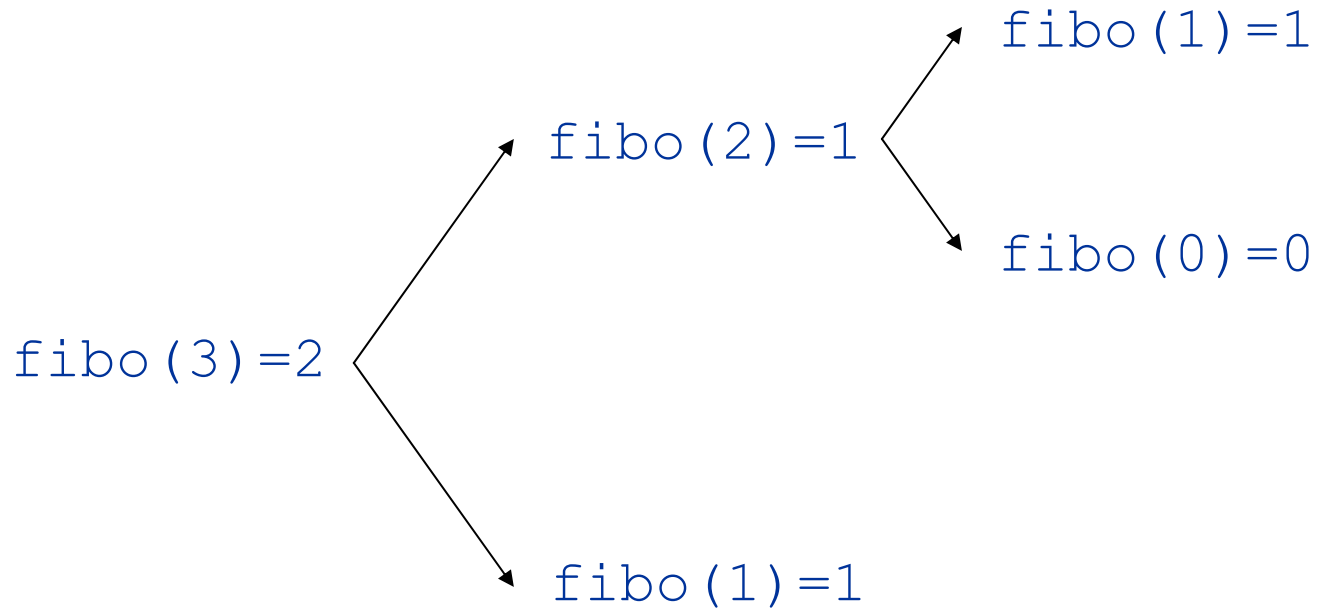
Albero di esecuzione



Albero di esecuzione



Albero di esecuzione



Esercizio 2-2

- Modificare il programma in modo che visualizzi quanti e quali chiamate vengono effettuate al metodo ricorsivo

```
Fibonacci  
  
public int fibo(int n);
```

difficoltà

La classe Fibonacci

```
public static int fibo(int n) {
    for(int i=0;i<n;i++)
        System.out.print("-");
    System.out.println("Inizio chiamata n="+n);
    int ritorno;
    if(n<=1) ritorno = n;
    else ritorno = fibo(n-1) + fibo(n-2);
    for(int i=0;i<n;i++)
        System.out.print("-");
    System.out.println("Fine chiamata n="+n);
    return ritorno;
}
```

Esecuzione

```

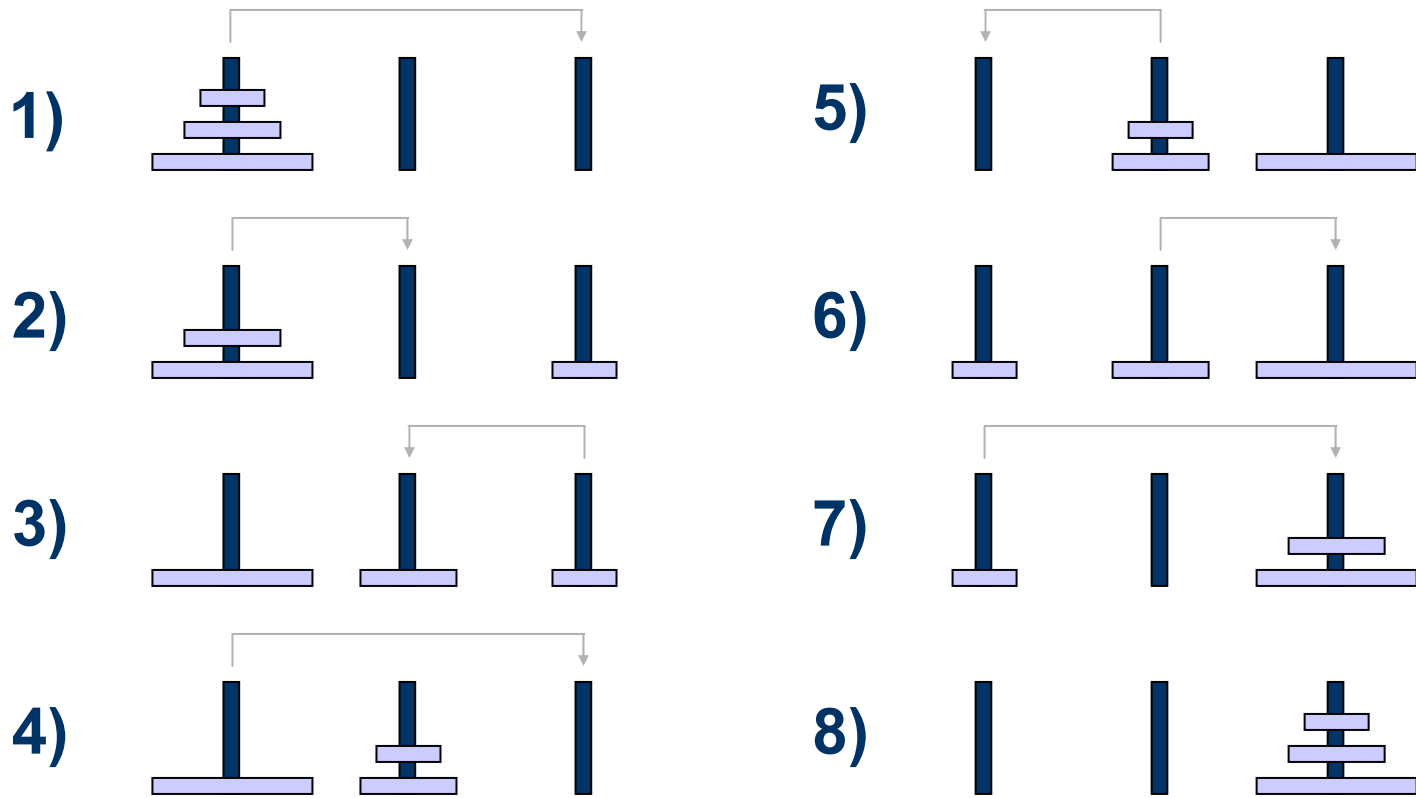
% java Fibonacci 5
-----Inizio chiamata n=5
----Inizio chiamata n=4
---Inizio chiamata n=3
--Inizio chiamata n=2
-Inizio chiamata n=1
-Fine chiamata n=1
Inizio chiamata n=0
Fine chiamata n=0
--Fine chiamata n=2
-Inizio chiamata n=1
-Fine chiamata n=1
---Fine chiamata n=3
--Inizio chiamata n=2
-Inizio chiamata n=1
-Fine chiamata n=1
Inizio chiamata n=0
Fine chiamata n=0
--Fine chiamata n=2
-Inizio chiamata n=1
-Fine chiamata n=1
---Fine chiamata n=3
--Inizio chiamata n=2
-Inizio chiamata n=1
-Fine chiamata n=1
-----Fine chiamata n=5
fib(5)=5

```

Le torri di Hanoi (Edouard Lucas, 1883)

- Una serie di dischi, ognuno con un differente diametro, è impilata su un palo dal disco più grande a quello più piccolo; accanto ad essa vi sono altri due pali.
- Bisogna spostare l'intera pila su un altro palo, muovendo un disco per volta e senza mai posizionare un disco sopra un altro disco di diametro minore.

Esempio



L'algoritmo

- Immaginiamo di poter spostare un'arbitraria sotto-pila di dischi, sempre rispettando il vincolo di non appoggiare un disco sopra uno più piccolo
- La soluzione sarebbe banale: basterebbe spostare l'intera pila di tre dischi da un palo all'altro

L'algoritmo ricorsivo

- Una generica pila di n elementi si può spostare ricorsivamente da un palo sorgente (S) a un palo destinazione (D):
 - Se $n=1$, basta spostare il disco da S a D;
 - Negli altri casi, basta spostare
 - $n-1$ dischi da S al palo libero (L)
 - Un disco da S a D
 - $n-1$ dischi da L a D

Esercizio 3-1

- Implementare una classe che risolva il problema delle torri di Hanoi, per un generico numero di dischi
- Provate a visualizzare le torri man mano che i dischi vengono spostati

La classe Hanoi

Hanoi

```
int numDischi;  
int torri[][];  
public Hanoi(int numDischi);  
public int calcola(int n);  
public void muovi(int numDischi, int da, int a, int libero);  
public void sposta(int da, int a);  
public void visualizza();
```

difficoltà 

La classe Hanoi

```
public class Hanoi {
    int numDischi;
    int torri[][];
    public Hanoi(int numDischi) {
        this.numDischi = numDischi;
        torri = new int[3][numDischi];
        for(int i=0;i<numDischi;i++) {
            torri[0][i] = i+1;
            torri[1][i] = 0;
            torri[2][i] = 0;
        }
        visualizza();
        muoviDischi(numDischi, 0, 2, 1);
    }
}
```

La classe Hanoi

```
public void muoviDischi(int numDischi,
    int da, int a, int libero) {
    if(numDischi <= 1) {
        System.out.println("muovo da"+da+"a"+ a);
        sposta(da, a);
    } else {
        muoviDischi(numDischi-1, da, libero, a);
        System.out.println("muovo da"+da+"a"+ a);
        sposta(da, a);
        muoviDischi(numDischi-1, libero, a, da);
    }
}
```

La classe Hanoi

```
public void sposta(int da, int a) {
    int posDa, posA, i=0;
    while((torri[da][i] == 0) && (i<numDischi))
        i++;
    posDa = (i==numDischi) ? numDischi-1:i;
    i=numDischi-1;
    while((torri[a][i] != 0) && (i>0))
        i--;
    posA = i;
    torri[a][posA] = torri[da][posDa];
    torri[da][posDa] = 0;
    visualizza();    }
```

La classe Hanoi

```
public void visualizza() {
    for(int j=0;j<numDischi;j++) {
        for(int i=0;i<3;i++)
            if(torri[i][j]==0)
                System.out.print("\t" + 0 + "\t");
            else
                System.out.print("\t"+torri[i][j]
                    +"\t");
            System.out.println("");
        }
    System.out.println("\n\n");
}
```

La classe Hanoi

```
public static void main(String args[]) {  
    Hanoi h = new Hanoi(3);  
}
```

Esecuzione

```

1      0      0      muovo da 0 a 2
2      0      0      0      0      0
3      0      0      0      1      0
                                0      2      3

muovo da 0 a 2
0      0      0      muovo da 1 a 0
2      0      0      0      0      0
3      0      1      0      0      0
                                1      2      3

muovo da 0 a 1
0      0      0      muovo da 1 a 2
0      0      0      0      0      0
3      2      1      0      0      2
                                1      0      3

muovo da 2 a 1
0      0      0      muovo da 0 a 2
0      1      0      0      0      1
3      2      0      0      0      2
                                0      0      3

```

Sperimentiamo

- La leggenda vuole che i monaci di un tempio induista dovessero affrontare questo problema utilizzando 64 dischi in oro, e che a problema risolto il mondo sarebbe finito.
- Secondo voi, possiamo stare tranquilli?

Sperimentiamo

- Quante operazioni sono necessarie per terminare il nostro algoritmo? A ogni chiamata di `muoviDischi` vengono effettuate
 - operazioni in un numero fisso
 - due chiamate ricorsive alla stessa funzione
- Si può mostrare che questo implica che il numero totale di istruzioni è circa pari a due elevato al numero di dischi

Quindi...

- Riuscendo a spostare un disco al secondo, in quanto tempo si riesce a spostare la pila di 64 dischi?
- Tenete conto dei seguenti fatti
 - $2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616$
 - L'età stimata dell'universo è dell'ordine di dieci miliardi di anni

Ulteriori informazioni

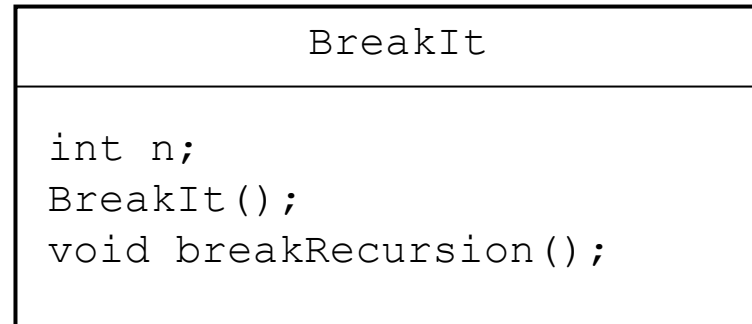
- <http://mathworld.wolfram.com/TowersofHanoi.html>
- <http://www.lhs.berkeley.edu/Java/Tower/towerhistory.html>

Esercizio 4

- Scrivere un programma che verifichi dopo quante chiamate a un metodo ricorsivo si verifica un'eccezione di Stack Overflow (`StackOverflowError` è l'eccezione che JAVA lancia, tra le altre cose, quando non può più allocare spazio nella macchina virtuale per ulteriori chiamate a un metodo ricorsivo)

La classe `BreakIt`

difficoltà



Idea: `n` viene inizializzato nel costruttore, `breakRecursion` lo incrementa e poi richiama se stessa ricorsivamente

La classe BreakIt

```
class BreakIt {
    public int n;
    public BreakIt() {
        n = 0;
    }
    public void breakRecursion() {
        n++;
        breakRecursion();
    }
    public static void main(String args[]) {
        BreakIt b = new BreakIt();
        b.breakRecursion();
    }
}
```

Esecuzione

```
% javac BreakIt.java
% java BreakIt
Exception in thread "main"
    java.lang.StackOverflowError
%
```

Miglioriamo BreakIt

- Se lo stack è pieno e non è più possibile effettuare un'ulteriore chiamata ricorsiva a un metodo, viene generata l'eccezione `StackOverflowError`
- E' possibile intercettare questa eccezione per poter leggere in `n` il numero totale di chiamate effettuate

Miglioriamo BreakIt

```
public static void main(String args[]) {  
    BreakIt b = new BreakIt();  
    try {  
        b.breakRecursion();  
    } catch (StackOverflowError e) {  
        System.out.println("Overflow dopo " +  
            b.n + " chiamate");  
    }  
}
```

Esecuzione

```
% javac BreakIt.java  
% java BreakIt  
Overflow dopo 7129 chiamate  
%
```

Sperimentiamo

Se dichiariamo la variabile di istanza di BreakIt di tipo long...

```
% javac BreakIt.java
% java BreakIt
Overflow dopo 6225 chiamate
%
```

Il numero di iterazioni dipende dalla quantità di memoria a disposizione.

Rimozione della ricorsione

- Sebbene gli algoritmi ricorsivi siano tipicamente compatti ed eleganti, la loro esecuzione richiede un tempo maggiore dei corrispettivi algoritmi non ricorsivi.

Esercizio 5-1

- Modificare la classe `Fibonacci` in modo che stampi la sequenza dei primi `n` numeri di Fibonacci

Fibonacci
<pre>public int fibo(int n); public void sequenza(int n);</pre>

difficoltà

Idea

- Avendo a disposizione un metodo che stampa l'ennesimo numero della sequenza di Fibonacci, basta chiamare questo metodo successivamente, all'interno di un opportuno ciclo.

La classe Fibonacci

```
public static void sequenza(int n) {
    for(int i=1;i<=n;i++)
        System.out.println(fibo(i));
}

public static void main(String args[]) {
    int n = Integer.parseInt(args[0]);
    sequenza(n);
}
```

Esecuzione

```
% java Fibonacci 10  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

Sperimentiamo

- Cosa succede se si prova ad eseguire questo programma stampando sequenze via via più lunghe?

Esercizio 5-2

- L'implementazione dell'esercizio 5-1 è decisamente inadeguata, in quanto per ogni output ricomincia a calcolare la sequenza dal suo inizio, mentre basterebbe ogni volta mantenere in memoria solo gli ultimi due elementi prodotti.
- Risolvere l'esercizio 5-1 in modo non ricorsivo

La classe Fibonacci

```
public static void sequenza(int n) {  
    int prev=1;  
    int prevprev=1;  
    int fibo=2;  
    for(int i=3;i<=n;i++) {  
        fibo=prev+prevprev;  
        prevprev=prev;  
        prev=fibo;  
        System.out.println(fibo);  
    }  
    // Attenzione: per semplicita' presupponiamo  
    // n maggiore di 2.
```

Esecuzione

```
% java Fibonacci 10  
2  
3  
5  
8  
13  
21  
34  
55
```