



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



La Ricorsione

Didattica Integrativa - Fondamenti di Informatica

Anno Accademico 2023/2024

Giulio Martini

giulio.martini@igi.cnr.it

La Ricorsione nelle Immagini

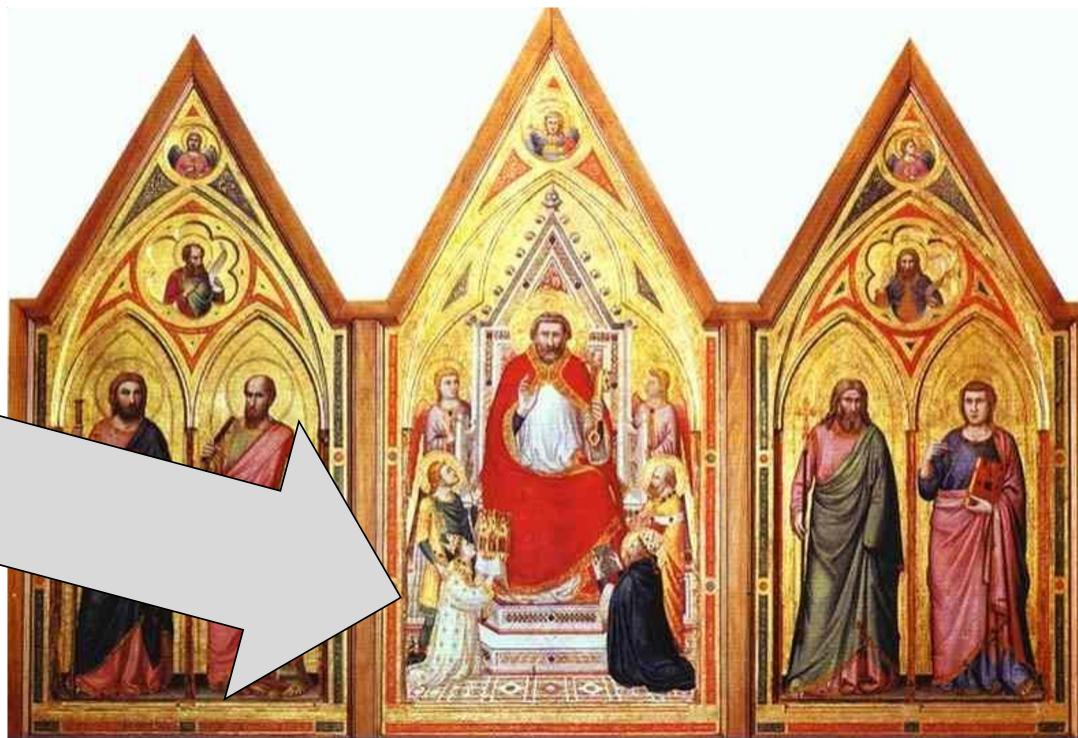
Graficamente la ricorsione è spesso associata al cosiddetto **effetto Droste**: un'immagine in cui è presente una piccola immagine di se stessa.

Questa piccola imagine (derivante dalla marca olandese di cacao Droste) contiene a sua volta una versione ancora più ridotta di se stessa, e così via.



The Droste-effect
1904 Droste Cacao by Jan Misset

La Ricorsione nelle Immagini



Giotto

Polittico Stefaneschi 1320, Pinacoteca
Vaticana



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

La Ricorsione in Natura



Broccolo romanesco



Felce



La Ricorsione Secondo Google

Google recursion

All Images Videos Books News More Settings Tools

About 40,400,000 results (0.56 seconds)

Did you mean: *recursion*

Recursion is the process a procedure goes through when one of the steps of the procedure involves invoking the procedure itself. A procedure that goes through **recursion** is said to be '**recursive**'. To understand **recursion**, one must recognize the distinction between a procedure and the running of a procedure.

Recursion - Wikipedia
<https://en.wikipedia.org/wiki/Recursion>

blog.angularindepth.com

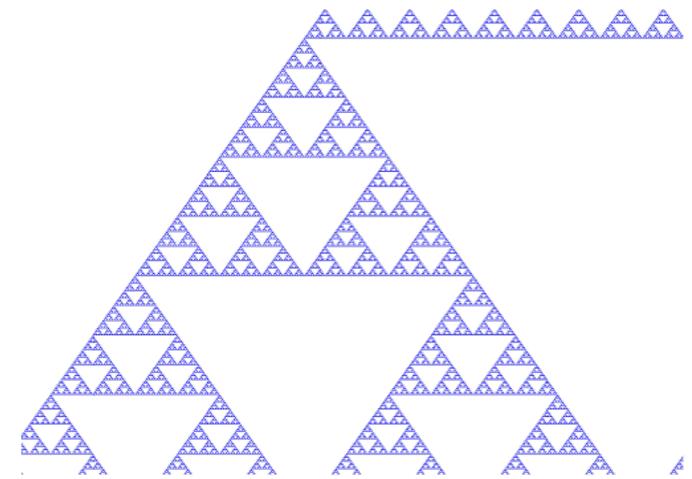
About Featured Snippets Feedback

**NOTA: non funziona
in italiano**

La Ricorsione in Matematica

In matematica la “**definizione ricorsiva**” si basa sul **principio di induzione**:

- se P è una proprietà che vale per il numero n_0 (**passo zero**) e se $P(n) \Rightarrow P(n+1) \forall n$ (**passo induttivo**), allora P vale $\forall n \geq n_0$ (n numero *naturale*)
- Esempio: operatore esponenziale
 - $a^0 = 1$
 - $a^{n+1} = aa^n$
- Esempio: triangolo di Sierpiński





La Ricorsione in Informatica

Un **algoritmo ricorsivo** è un algoritmo espresso in termini **di se stesso**, ovvero in cui l'esecuzione dell'algoritmo su un insieme di dati comporta la **semplificazione o suddivisione** dell'insieme di dati e l'applicazione dello **stesso algoritmo** agli insiemi di **dati semplificati**

La **ricorsione** è una tecnica di programmazione molto potente, che sfrutta l'idea di suddividere un problema da risolvere in **sottoproblemi simili** a quello originale, ma **più semplici**

Può però generare **errori difficili da diagnosticare**



La Ricorsione: Numeri Triangolari

Calcolare l'area di una forma triangolare di dimensione $n > 0$ (riportata sotto), dove **ciascun quadrato [] ha area unitaria**. Il valore dell'area corrispondente viene chiamato **numero triangolare n-esimo**

Esempio $n = 4$:

[]
[][]
[][][]
[][][][]

[]
[][]
[][][]
[][][][]

[]
[][]
[][][]

[]
[][]

[]

Generalizzando:
 $R(n) = n + R(n-1)$,
 $n > 1$

$R(4)$

$R(4)=4+R(3)$ $R(3)=3+R(2)$ $R(2)=2+R(1)$ $R(1)=1$

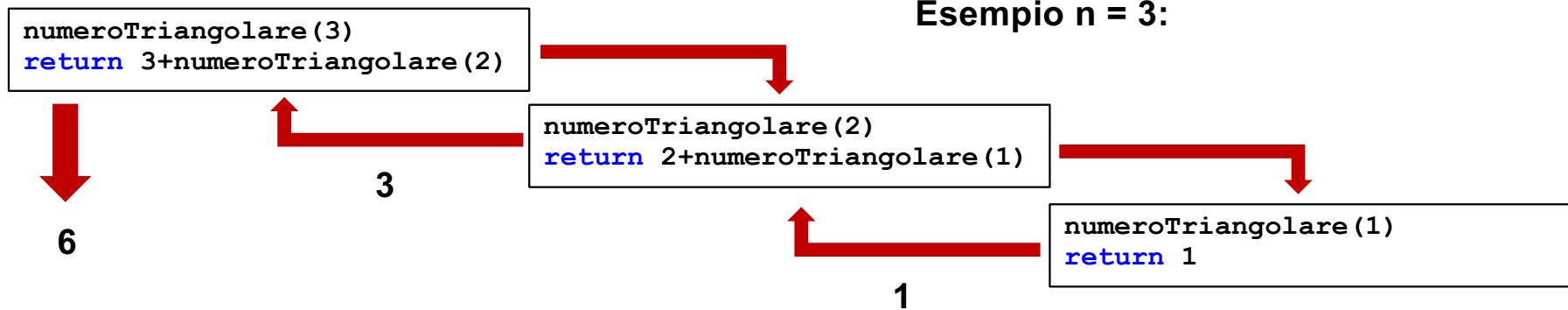
Sfruttiamo la struttura di tali numeri: possono essere calcolati considerando i numeri che li precedono: $R(4) = 4 + 3 + 2 + 1 = 10$

La Ricorsione: Numeri Triangolari

```
public static int numeroTriangolare(int n) {  
  
    if (n < 1) throw new IllegalArgumentException();  
  
    if (n == 1) return 1; //caso base  
  
    return n + numeroTriangolare(n - 1); //ricorsione  
  
}
```

$$[] = 1$$

Esempio $n = 3$:





La Ricorsione: Numeri Triangolari

La ricorsione **non** è un **sistema efficiente** per calcolare l' n -esimo numero triangolare

$R(4) = 4 + 3 + 2 + 1 = 10$ somma dei primi n numeri positivi

$$R(n) = \sum_{i=1}^n i = \frac{n * (n+1)}{2} \quad \rightarrow \quad \text{formula di Gauss}$$

[] [] [] []
[] [] [] [] []
[] [] [] [] [] []
[] [] [] [] [] [] []

```
public static int numeroTriangolareSmart(int n)  {  
    if (n < 1) throw new IllegalArgumentException();  
    return n*(n+1)/2;  
}
```

$$R(4) = (4*5)/2$$



La JVM nella Ricorsione

Quando un metodo ricorsivo **invoca se stesso**, la Java Virtual Machine (JVM) esegue le **stesse azioni** che vengono eseguite quando viene chiamato un **altro metodo**:

- sospende l'esecuzione del metodo invocante
- esegue il metodo invocato fino alla sua terminazione
- riprende l'esecuzione del metodo invocante dal punto in cui era stata sospesa



Lo Stack della Ricorsione

La JVM gestisce lo **stack of activation records (AR)**

- Stack: LIFO (Last In, First Out)
- Ad ogni **attivazione** di un metodo, un nuovo AR viene creato sulla parte superiore dello stack
- Ad ogni **terminazione** di un metodo, il corrispondente AR viene eliminato

AR contiene tutte le informazioni necessarie per l'esecuzione di un metodo (valore delle variabili locali, riferimento degli oggetti, ecc..)



Esempio di Ricorsione

```
public class MostraRicorsione{
    public static void ricorsione(int i){
        System.out.print("In ricorsione("+i+ ")");
        if(i==0) System.out.println(" - Terminato");
        else{
            System.out.println(" - Invocazione ricorsione("+(i-1)+ ")");
            ricorsione(i-1);
            System.out.print("Tornato in ricorsione("+i+ ")");
            System.out.println(" - Terminato");
        }
    }
    public static void main(String[] args){
        int j = 2;
        System.out.println("In main");
        System.out.println("Invocazione ricorsione("+j+ ")");
        ricorsione(j);
        System.out.print("Tornato in main");
        System.out.println(" - Terminato");
    }
}
```

```
In main
Invocazione ricorsione(2)
In ricorsione(2) - Invocazione ricorsione(1)
In ricorsione(1) - Invocazione ricorsione(0)
In ricorsione(0) - Terminato
Tornato in ricorsione(1) - Terminato
Tornato in ricorsione(2) - Terminato
Tornato in main - Terminato
```

Esempio di stack
durante l'esecuzione
di **ricorsione(0)**

i assume valori diversi
in ciascun
“esemplare” del
metodo ricorsivo

4) ricorsione	
i	0
3) ricorsione	
i	1
2) ricorsione	
i	2
1) main	
j	2



Progettare Metodi Ricorsivi

IDEA: scomporre il problema in una **porzione semplice** (caso base, che si riesce a risolvere) e in un'altra **porzione complessa, simile a quella iniziale** ma di **taglia inferiore**

Nel metodo si eseguono le seguenti fasi (l'ordine può variare):

- risolvere la porzione semplice (caso base)
- invocazione ricorsiva per risolvere la porzione complessa
- calcolare la soluzione combinando i risultati delle fasi precedenti

Progettare Metodi Ricorsivi

$$R(n) = \begin{cases} 1 & \text{se } n = 1 \\ n + R(n - 1) & \text{se } n > 1 \end{cases}$$

```
public static int numeroTriangolare(int n)  {

    if (n == 1) return 1;      → caso base: porzione semplice n=1

    return n + numeroTriangolare(n - 1);
}

→ chiamata ricorsiva: porzione complessa n-1

}
→ combinazione dei risultati precedenti
```



Progettare Metodi Ricorsivi

Alcune regole fondamentali:

- Il metodo ricorsivo deve fornire la soluzione di almeno un **caso base**, che non ricorra ad un'invocazione ricorsiva
- L'invocazione ricorsiva deve avvenire su un **problema di taglia inferiore** rispetto al problema originale

L'idea è quella di **semplificare mano a mano il problema da risolvere, convergendo al caso base** che non richiede ricorsione

Progettare Metodi Ricorsivi

Se almeno una delle due condizioni **non è rispettata** il metodo ricorsivo **invoca se stesso “all’infinito”** (finché lo stack dei metodi in attesa non esaurisce la memoria)



Exception in thread "main" java.lang.StackOverflowError

**il problema
non viene
semplificato**

```
public static void ricorsione(int i){  
    System.out.print("In ricorsione("+i+");");  
    if(i==0) System.out.println(" - Terminato");  
    else{  
        System.out.println(" - Invocazione ricorsione("+(i+"));");  
        ricorsione(i);  
        System.out.print("Tornato in ricorsione("+i+");");  
        System.out.println(" - Terminato");  
    }  
}
```



Progettare Metodi Ricorsivi

Se almeno una delle due condizioni **non è rispettata** il metodo ricorsivo **invoca se stesso “all’infinito”** (finché lo stack dei metodi in attesa non esaurisce la memoria)



Exception in thread "main" java.lang.StackOverflowError

**manca il caso
base**

```
public static void ricorsione(int i){  
    System.out.print("In ricorsione("+i+")");  
    System.out.println(" - Invocazione ricorsione("+i-1+");");  
    ricorsione(i-1);  
    System.out.print("Tornato in ricorsione("+i+");");  
    System.out.println(" - Terminato");  
}
```



- Numeri Triangolari: $R(n) = \sum_{i=1}^n i = \frac{n * (n+1)}{2} \dots$

...esiste una formula analitica!

- Fattoriale: $n! = \prod_{k=1}^n k$ (con $0! = 1$)

Il fattoriale **non è calcolabile direttamente** con un'unica formula, come avviene per i numeri triangolari



Definizione Ricorsiva:

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

```
public static int fattorialeRi(int n){  
  
    if (n < 0) throw new  
        IllegalArgumentException();  
  
    if (n == 0) return 1; //caso base  
  
    return n * fattorialeRi(n - 1);  
  
}
```

Definizione Iterativa:

$$n! = \prod_{k=1}^n k$$

```
public static int fattorialeIt(int n){  
  
    if (n < 0) throw new  
        IllegalArgumentException();  
  
    int result = 1;  
  
    for (int k = 2; k <= n; k++)  
        result *= k;  
  
    return result;  
}
```



Ricorsione in Coda

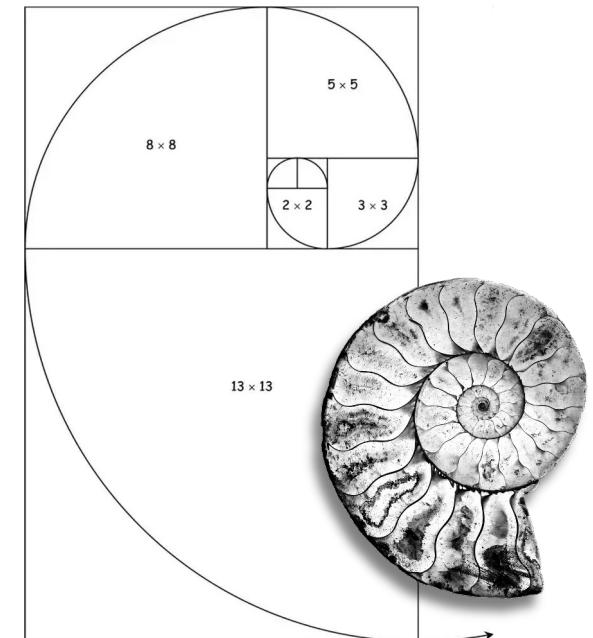
- Gli algoritmi ricorsivi per il calcolo dei numeri triangolari e del fattoriale sono esempi di **ricorsione in coda**: effettuano **un'unica chiamata ricorsiva** che è **l'ultima istruzione del metodo**
- Facilmente **trasformabili in algoritmi iterativi**, è sufficiente eliminare la ricorsione con un ciclo
- **Nota:** un metodo con ricorsione in coda è sempre meno efficiente del rispettivo metodo iterativo

La Ricorsione Multipla: Fibonacci

- **Ricorsione multipla:** un metodo invoca se stesso più volte durante la sua esecuzione
- Più difficile da eliminare
- Esempio: successione di Fibonacci

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n - 2) + F(n - 1) & \text{se } n > 1 \end{cases}$$

0 1 1 2 3 5 8 13 ...

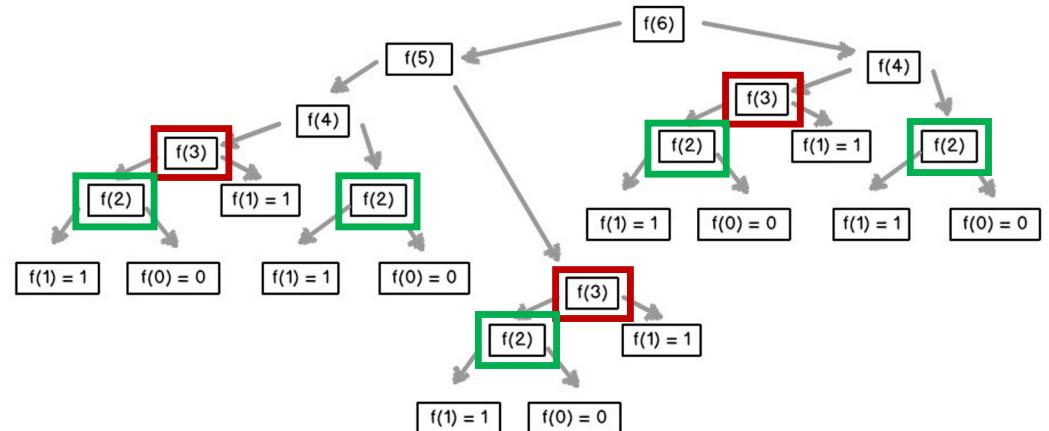


La Ricorsione Multipla: Fibonacci

```
public static long fibBad(int n) {  
  
    if (n < 0) throw new IllegalArgumentException();  
  
    if (n < 2) return n;  
  
    return fibBad(n-2) + fibBad(n-1);  
}
```

Non è un sistema efficiente: **invocazioni multiple con gli stessi parametri**

Albero delle chiamate ricorsive per $F(6)$



Fibonacci Good

Restituisce un array contenente $F(n)$ e $F(n-1)$

```
public static long[] fibGood(int n) {  
    if (n < 0) throw new IllegalArgumentException();  
    if (n < 2){  
        long[] res = {n,0};  
        return res;  
    }  
    long[] temp = fibGood(n-1); //ritorna [F(n-1), F(n-2)]  
    long[] res = {temp[0] + temp[1], temp[0]}; // [F(n), F(n-1)]  
    return res;  
}
```

Esempio Fibonacci $n=5$

fibGood(5)	{5,3}
fibGood(4)	{3,2}
fibGood(3)	{2,1}
fibGood(2)	{1,1}
fibGood(1)	{1,0}



È sufficiente un ciclo di $n-1$ iterazioni:

```
public static long fibIt(int n) {  
    if (n < 0) throw new IllegalArgumentException();  
    if (n < 2) return n;  
    long fib0 = 0;  
    long fib1 = 1;  
    for (int i = 2; i <= n; i++) {  
        long newFib = fib0 + fib1;  
        fib0 = fib1;  
        fib1 = newFib;  
    }  
    return fib1;  
}
```



Errori nella Ricorsione

Essenziale capire bene il funzionamento del paradigma della ricorsione (sospensione del metodo invocante, esecuzione del metodo invocato, riesecuzione del metodo sospeso)

- Manca il caso base
- Non viene ridotta la taglia del problema ad ogni chiamata ricorsiva
- Algoritmi inefficienti (ricomputazione)

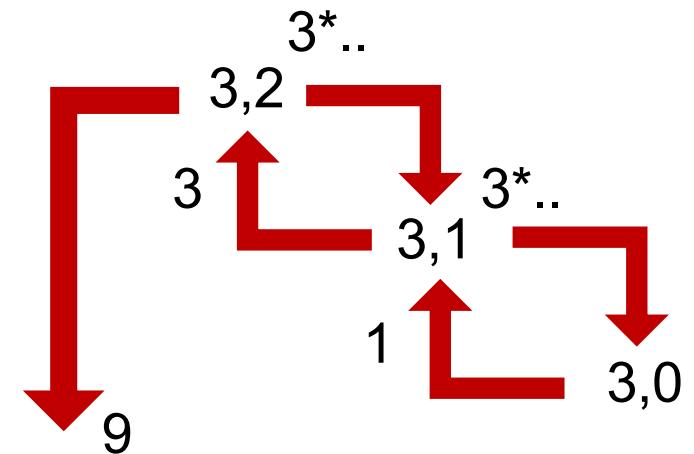
Esercizi: Potenza

Dati due interi n e k , calcolare n^k con un metodo ricorsivo

$$n^k = \begin{cases} 1 & \text{se } k = 0 \\ n * n^{k-1} & \text{se } k > 0 \end{cases}$$

Esempio $n=3, k=2$:

```
public static int pot(int n, int k){  
    if (k < 0) throw new  
        IllegalArgumentException();  
    if (k == 0) return 1;  
    return n*pot(n,k-1);  
}
```

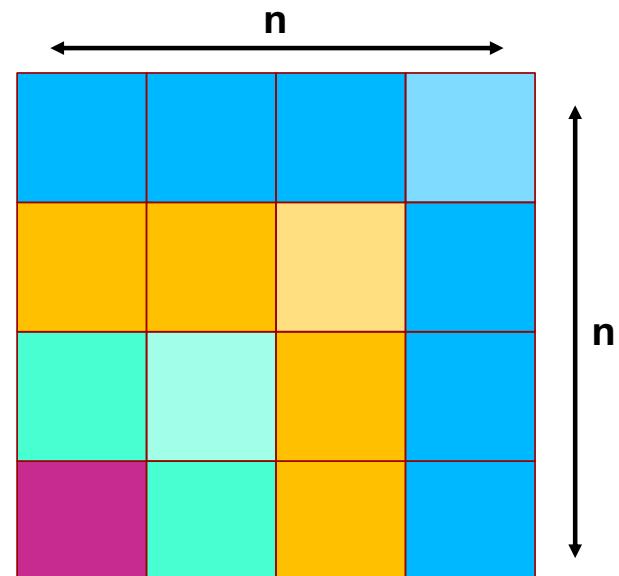


Esercizi: Area di un Quadrato

Dato un quadrato di lato n , calcolarne l'area $A(n)$ ricorsivamente

$$A(n) = \begin{cases} 0 & se n = 0 \\ A(n - 1) + 2n - 1 & se n > 0 \end{cases}$$

```
public static int square(int n){  
    if (n < 0)  
        throw new IllegalArgumentException();  
    else if (n == 0)  
        return 0;  
    else  
        return ( square(n-1) + 2*n - 1 );  
}
```





Esercizi: Numero Primo

Dato un intero $n > 1$, dire se è un numero primo con un metodo ricorsivo

IDEA:

- Testare in ordine crescente i divisori $K = \{k \in \mathbb{N} : k \geq 2 \wedge k^2 \leq n\}$
- Utilizzare un metodo ausiliario
- Casi base: $n \% k = 0 \Rightarrow (n \text{ non è primo})$ e $k^2 > n \Rightarrow (n \text{ è primo})$
- Chiamata ricorsiva: $k+1$



Esercizi: Numero Primo

Dato un intero $n > 1$, dire se è un numero primo con un metodo ricorsivo

```
public static boolean isPrimo(int n) {  
    if (n < 2) throw new  
        IllegalArgumentException();  
    return primoRi(n, 2);  
}  
  
public static boolean primoRi(int n, int k) {  
    if (k*k > n) return true;  
    if (n%k == 0) return false;  
    return primoRi(n, k+1);  
}
```



Esercizi: Invertire una Stringa

Data una stringa s , invertire l'ordine dei suoi caratteri con un metodo ricorsivo (e.g., “REVERSE” -> “ESREVER”)

IDEA: $\text{rev}(\text{"REVERSE"}) = \text{rev}(\text{"EVERSE"}) + \text{"R"}$

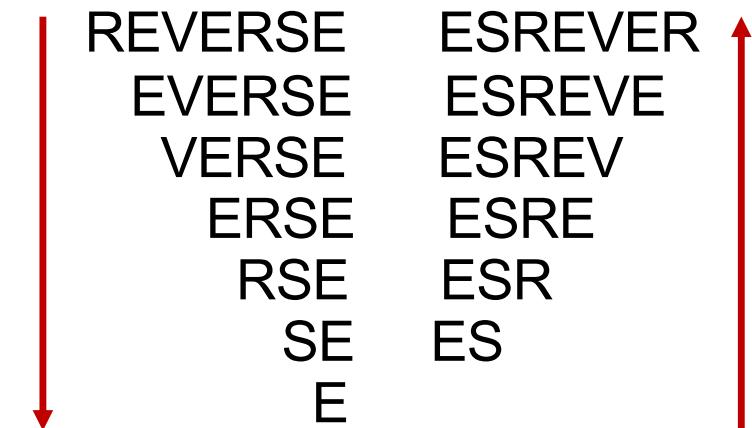
- Caso base: $|s| < 2$ (non c’è nulla da invertire)
- Passo ricorsivo: inversione della sottostringa senza il primo carattere
- Restituire la sottostringa invertita concatenata al primo carattere

Esercizi: Invertire una Stringa (2/2)

Data una stringa s , invertire l'ordine dei suoi caratteri con un metodo ricorsivo (e.g., “REVERSE” -> “ESREVER”)

Esempio $s = \text{“REVERSE”}$

```
public static String rev(String s){  
    if (s == null) return null;  
    if (s.length() < 2) return s;  
    return rev(s.substring(1)) + s.charAt(0);  
}
```





Esercizi: Riconoscere una Stringa Palindroma

Riconoscere se una stringa è palindroma con un metodo ricorsivo

Stringa palindroma: uguale alla stringa con i caratteri in ordine invertito

Es: “anna ama otto e otto ama anna”, “amor a roma”

IDEA: **a - nna ama otto e otto ama ann - a**

- Casi base:
 - $|s| < 2$: la stringa è palindroma
 - primo e ultimo caratteri sono diversi: la stringa non è palindroma
- Passo ricorsivo: se primo e ultimo carattere sono uguali allora la stringa è palindroma se e solo se la stringa senza il primo e l'ultimo carattere è palindroma



Esercizi: Riconoscere una Stringa Palindroma

Riconoscere se una stringa è palindroma con un metodo ricorsivo

```
public static boolean isPalindroma(String s){  
    if (s == null) throw new IllegalArgumentException();  
    if (s.length() < 2) return true;  
    if (s.charAt(0) != s.charAt(s.length() - 1))  
        return false;  
    return isPalindroma(s.substring(1, s.length() - 1));  
}
```

Esempio s = “RADAR” :

RADAR
ADA
D true

Esempio s = “RADER”:

RADER
ADE false



Esercizi: Stampare tutte le Sottostringhe

Data una stringa s , stampare tutte le sue sottostringhe

Esempio $s = \text{"RUM"} : \{\text{"R"}, \text{"RU"}, \text{"RUM"}, \text{"U"}, \text{"UM"}, \text{"M"}, \text{""}\}$

```
public static void printSubstring(String s){  
    if (s == null) throw new IllegalArgumentException();  
    if (s.length() == 0) System.out.println(s);  
    else{  
        for(int i = 1; i <= s.length();i++){  
            System.out.println(s.substring(0,i));  
        }  
        printSubstring(s.substring(1));  
    }  
}
```

Esempio $s = \text{"RUM"}$

$s = \text{"RUM"}$
println:
 “R”
 “RU”
 “RUM”

$s = \text{"UM"}$
println:
 “U”
 “UM”

$s = \text{"M"}$
println:
 “M”

$s = \text{""}$
println:
 “”

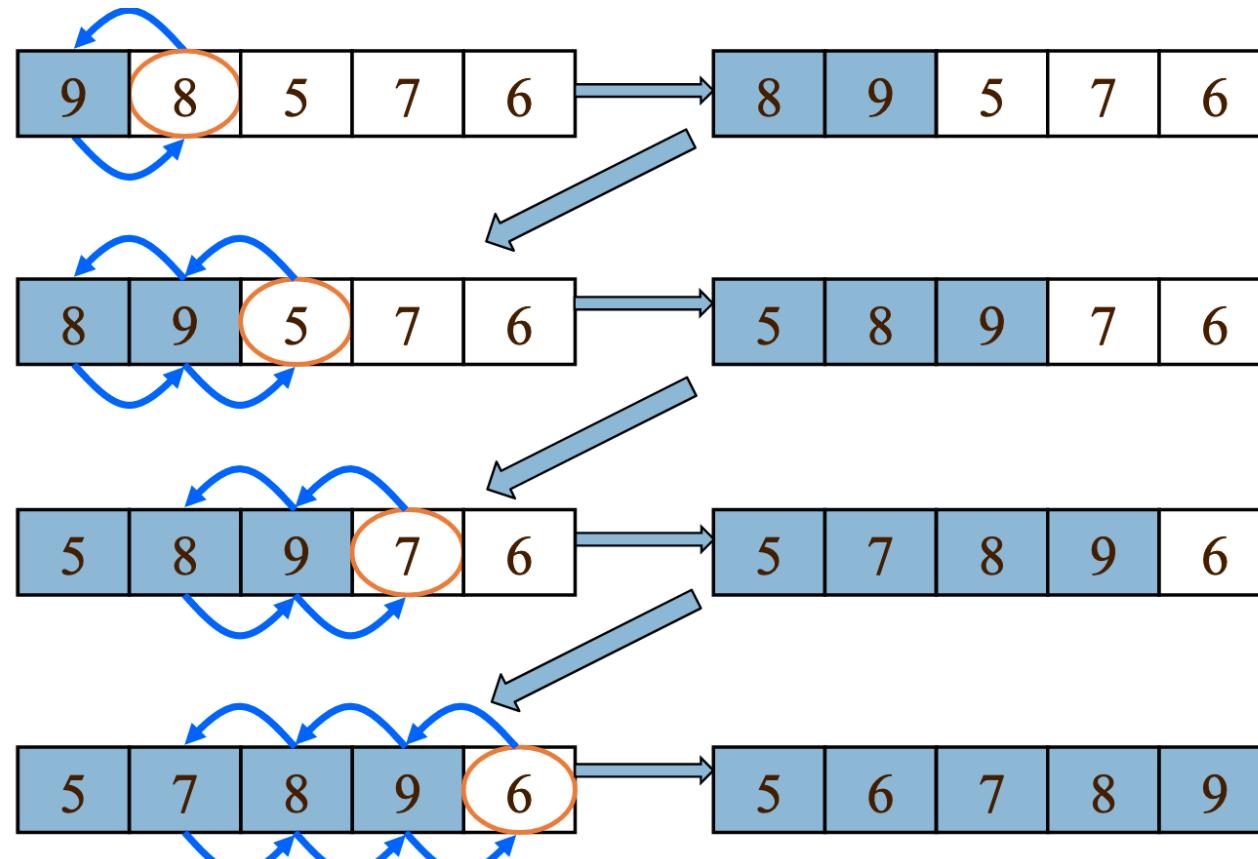


Esercizi: Insertion Sort Ricorsivo

Insertion Sort:

- Il sottoarray di lunghezza unitaria (prima cella dell'array) è ordinato
- Estende mano a mano la parte ordinata includendo ad ogni iterazione il primo elemento alla destra della parte ordinata
 - Per farlo, sposta verso sinistra tale elemento fino a trovare la sua posizione corretta, spostando verso destra gli elementi intermedi

Esercizi: Insertion Sort Ricorsivo





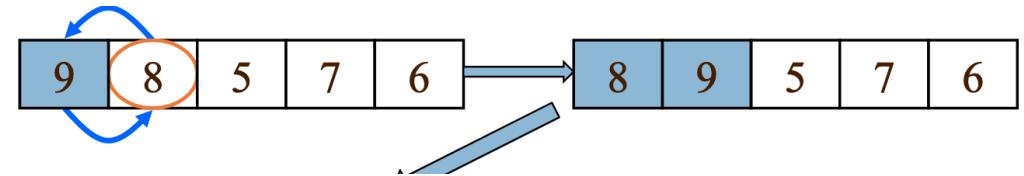
Esercizi: Insertion Sort Ricorsivo

IDEA:

- Caso base: l'array composto da un solo elemento è ordinato
- Passo ricorsivo: ordino l'array contenente i primi $n-1$ elementi in modo ricorsivo
- Sposto l'ultimo elemento verso sinistra nella sua posizione corretta spostando verso destra gli elementi intermedi

Esercizi: Insertion Sort Ricorsivo

```
//a è l'array da ordinare
//n è il numero di componenti di a da ordinare
public static void insertionSortRi(int[] a, int n) {
    if ( n > 1){
        insertionSortRi(a,n-1);
        int value = a[n-1];
        int i;
        for (i = n-2; i >= 0 && a[i] > value; i--){
            a[i+1] = a[i];
        }
        a[i+1] = value; //N.B. i è -1 usciti dal ciclo!
    }
}
```





Esercizi: Insertion Sort Ricorsivo

```
public static void insertionSortRi(int[] a, int n) {  
    if (n > 1) {  
        insertionSortRi(a, n-1);  
        int value = a[n-1];  
        int i;  
        for (i = n-2; i >= 0 && a[i] > value; i--) {  
            a[i+1] = a[i];  
        }  
        a[i+1] = value;  
    }  
}
```

Esempio $a = \{7,6,5\}$

$\text{insertionSortRi}(a,3)$

$a = \{7,6,5\}$

$\text{insertionSortRi}(a,2)$

$a = \{7,6,5\}$

$\text{insertionSortRi}(a,1)$

$a = \{7,6,5\}$

value = 6

$a[1] = 7$ //for $i = 0$

$a[0] = 6$

$a = \{7,7,5\}$

$a = \{6,7,5\}$

value = 5

$a[2] = 7$ //for $i = 1$

$a[1] = 6$ //for $i = 0$

$a[0] = 5$

$a = \{6,7,7\}$

$a = \{6,6,7\}$

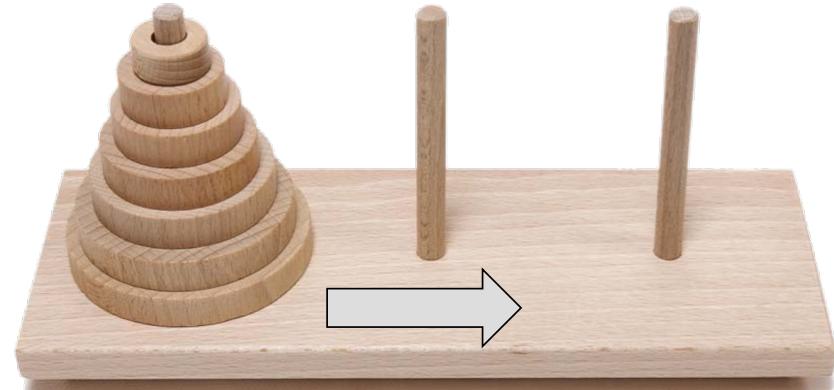
$a = \{5,6,7\}$



Rompicapo costruito da 3 “torri” allineate e da n dischi di dimensione diversa

Inizio: tutti i dischi si trovano sulla torre di sinistra

Fine: tutti i dischi devono trovarsi sulla torre di destra

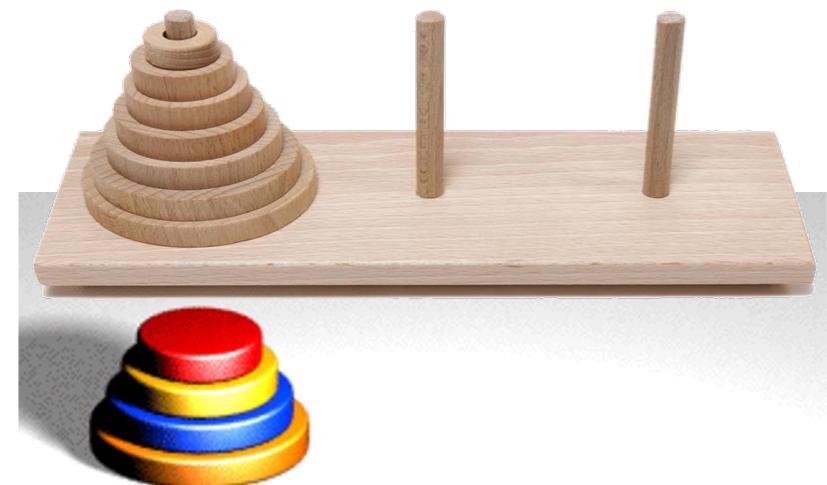


Nessun disco può avere sopra di se dischi più grandi di lui

Torre di Hanoi

REGOLE:

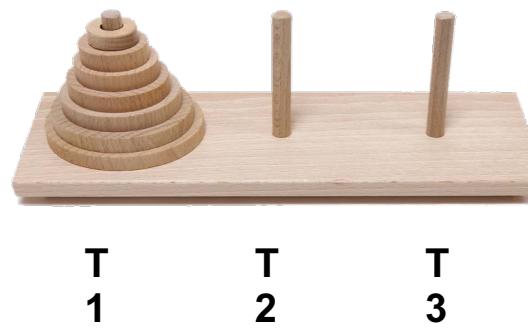
- Si può spostare un disco alla volta
- Un disco quando viene rimosso da una torre deve essere inserito in cima ad un'altra torre
- **In nessun momento un disco può avere sopra di se dischi più grandi di lui**



È noto un **algoritmo ricorsivo** per risolverlo:

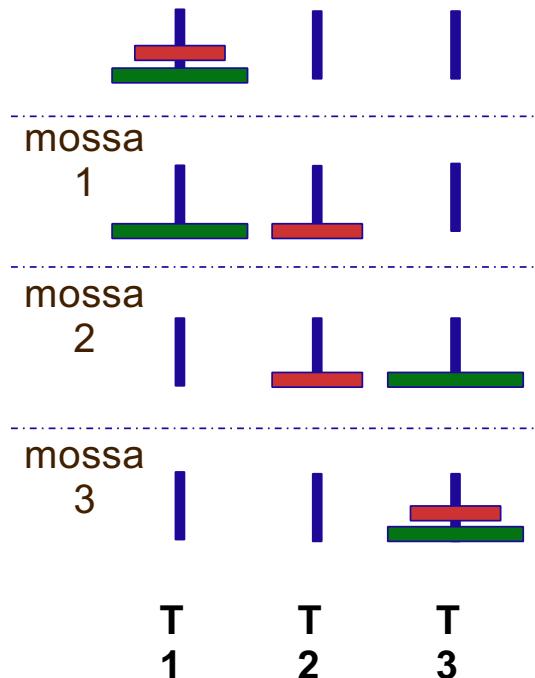
- Vogliamo spostare n dischi da una torre ad un'altra, utilizzando una terza torre come **deposito temporaneo**
- Per spostare n dischi da una torre ad un'altra si suppone di saper spostare $n-1$ dischi, come si fa nella ricorsione

Per descrivere l'algoritmo identifichiamo le 3 torri con 1, 2 e 3



Torre di Hanoi ($n=2$)

Situazione Iniziale

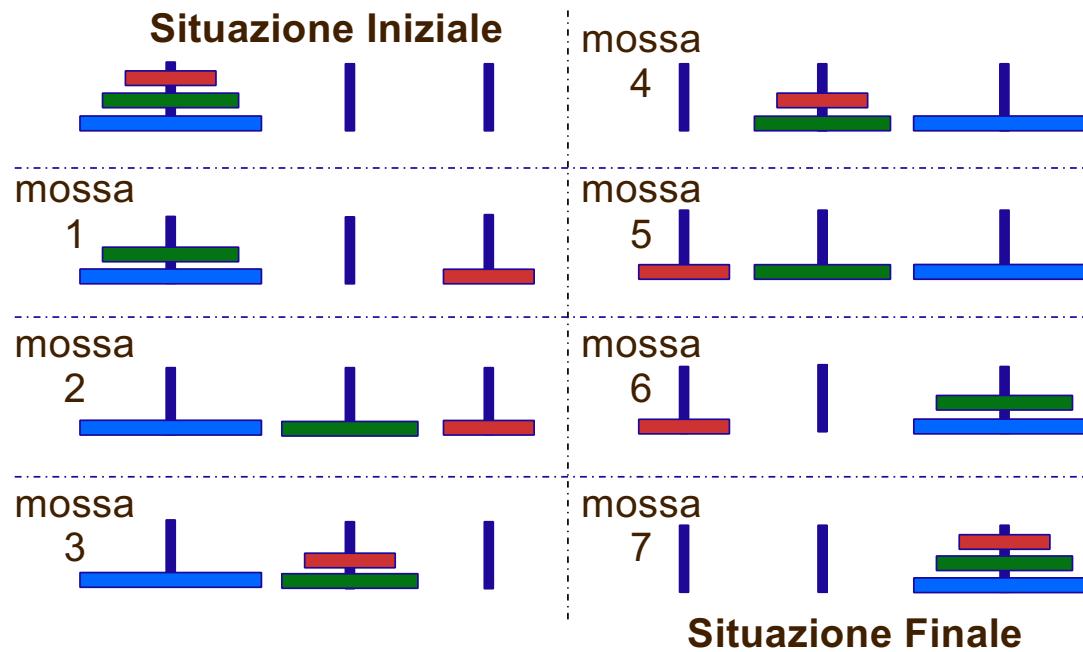


Per $n=2$ la soluzione è banale

Si usa la torre 2 come posizione temporanea per il disco rosso, in modo da poter spostare il disco verde nella torre finale

Torre di Hanoi ($n=3$)

Per 3 dischi la soluzione è meno ovvia ma diventa semplice se si ragiona ricorsivamente applicando il caso precedente ($n=2$) per spostare i dischi rosso e verde nella torre 2





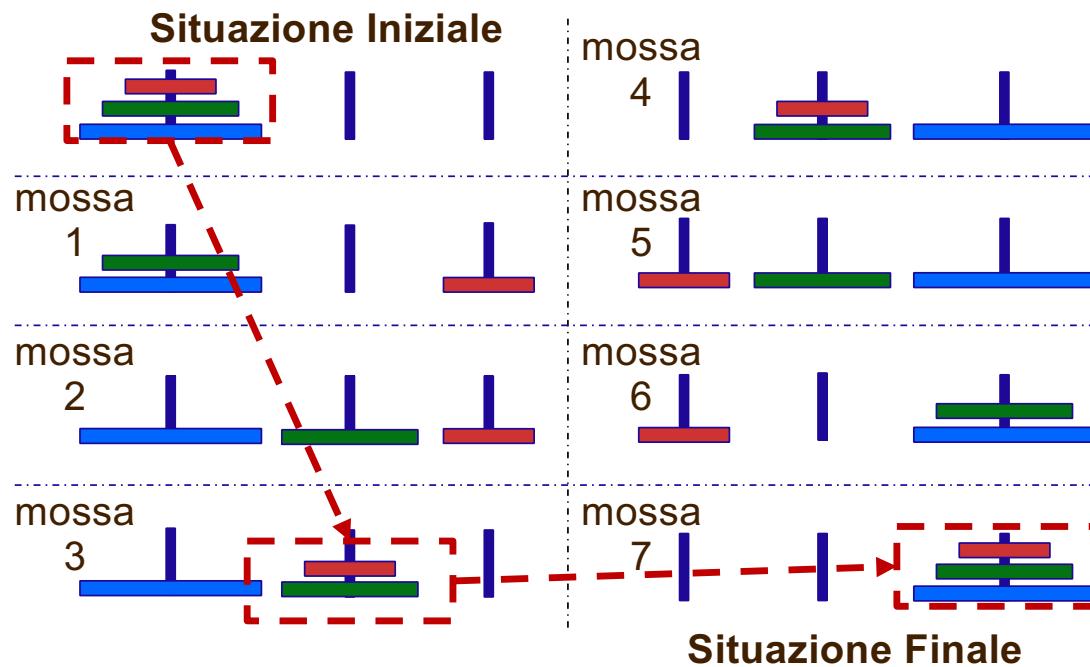
IDEA (per spostare n dischi da T1 a T3):

- Sposto $n-1$ dischi da T1 a T2 usando T3 come deposito temporaneo con una chiamata ricorsiva (1 disco in T1, $n-1$ dischi in T2, T3 vuota)
- Sposto il disco rimasto su T1 in T3
- Sposto gli $n-1$ dischi da T2 a T3 usando T1 come deposito temporaneo con una chiamata ricorsiva (T1 e T2 vuote, n dischi in T3)

Caso base: quando $n = 1$ l'algoritmo usa solo il passo 2 che non ha chiamate ricorsive

Torre di Hanoi ($n=3$)

Per 3 dischi la soluzione è meno ovvia ma diventa semplice se si ragiona ricorsivamente applicando il caso precedente ($n=2$) per spostare i dischi rosso e verde nella torre 2

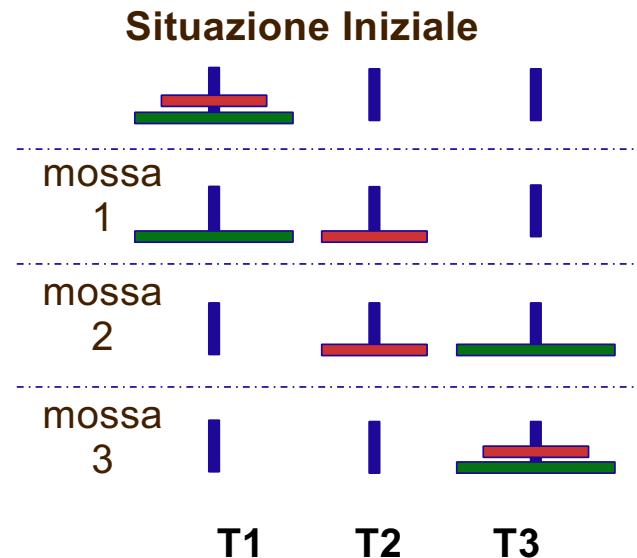




```
public class Hanoi{  
  
    public static void main(String [] args){  
  
        System.out.println("Soluzione Torre di Hanoi (n = " + args[0] + ")");  
  
        solveHanoi(1,3,2,Integer.parseInt(args[0]));  
  
    }  
  
    public static void solveHanoi(int from, int to, int temp, int dim){  
  
        if (dim > 0){  
  
            solveHanoi(from, temp, to, dim-1);  
  
            System.out.println("Sposta il disco " + dim + " da T" + from + " a T" + to);  
  
            solveHanoi(temp, to, from, dim-1);  
  
        }  
  
    }  
}
```

Torre di Hanoi ($n=2$)

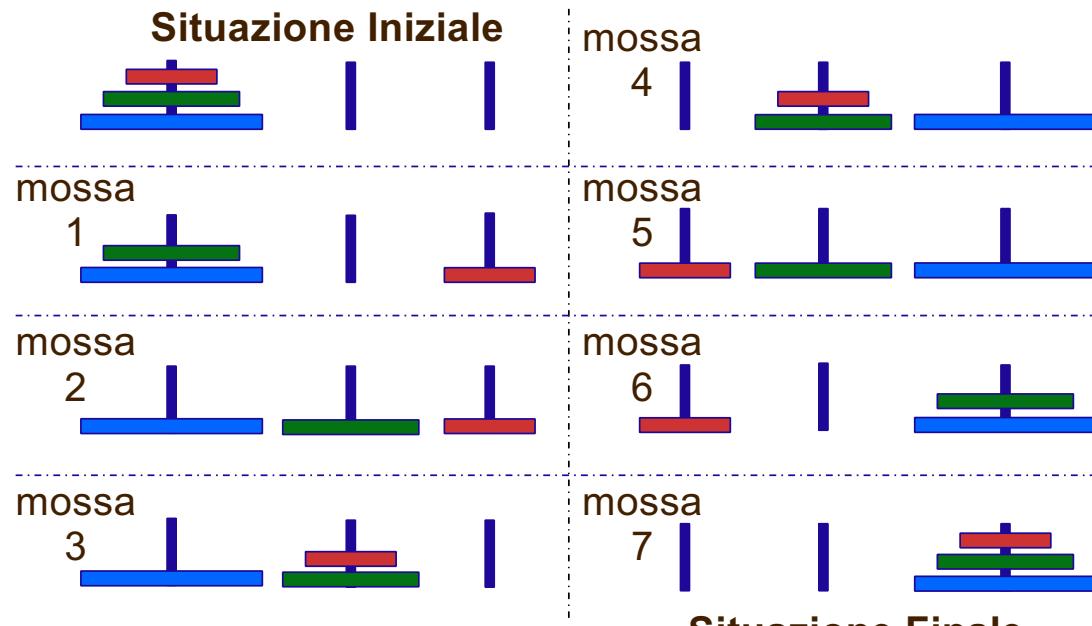
```
solveHanoi(1,3,2,2)
    solveHanoi(1,2,3,1)
        solveHanoi(1,3,2,0)
            Sposta il disco 1 da T1 a T2
            solveHanoi(3,2,1,0)
            Sposta il disco 2 da T1 a T3
            solveHanoi(2,3,1,1)
            solveHanoi(2,1,3,0)
            Sposta il disco 1 da T2 a T3
            solveHanoi(1,3,2,0)
```



Soluzione Torre di Hanoi ($n=2$)

```
Sposta il disco 1 da T1 a T2
Sposta il disco 2 da T1 a T3
Sposta il disco 1 da T2 a T3
```

Torre di Hanoi ($n=3$)



Soluzione Torre di Hanoi ($n=3$)

- Sposta il disco 1 da T1 a T3
- Sposta il disco 2 da T1 a T2
- Sposta il disco 1 da T3 a T2
- Sposta il disco 3 da T1 a T3
- Sposta il disco 1 da T2 a T1
- Sposta il disco 2 da T2 a T3
- Sposta il disco 1 da T1 a T3



Si può dimostrare che il numero di mosse per risolvere il rompicapo con l'algoritmo proposto è: $2^n - 1$

$$T(n) = 2 * T(n - 1) + 1$$

Iterando la formula si ottiene: $T(n) = 2^k * T(n - k) + \sum_{i=0}^{k-1} 2^i$

e considerando $T(1) = 1$, si ottiene $T(n) = 2^n - 1$



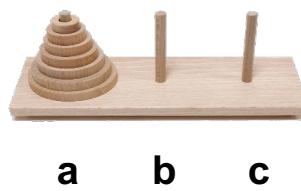
Torre di Hanoi: la Leggenda

Una leggenda (inventata dalla ditta che per prima ha messo in commercio il rompicapo) narra che alcuni monaci buddisti in un tempio dell'Estremo Oriente siano costantemente impegnati nella soluzione del rompicapo, spostando fisicamente i loro 64 dischi da una torre all'altra, consapevoli che **quando avranno terminato il mondo finirà**.

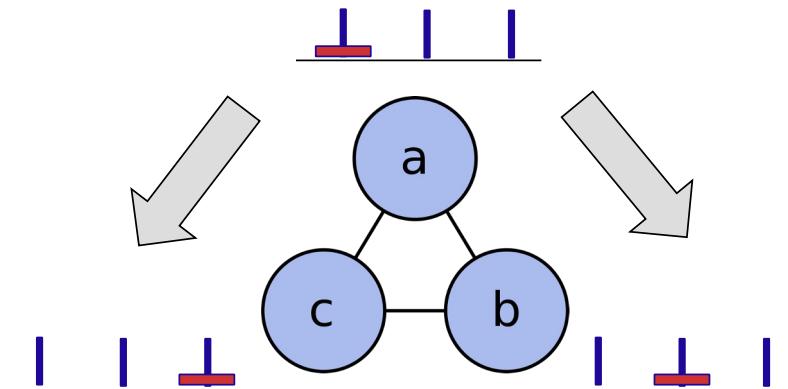
Ricordando che il numero di mosse è **$2^n - 1$** , secondo la leggenda, i monaci di Hanoi dovrebbero effettuare almeno **18.446.744.073.709.551.615** mosse prima che il mondo finisca ($2^{64} - 1$). In altre parole, anche supponendo che i monaci facciano una mossa al secondo, impiegheranno circa 6 miliardi di secoli per terminarlo, un tempo così lungo che quando il sole diverrà una gigante rossa e brucerà la Terra, il gioco non sarà stato ancora completato.

Torre di Hanoi: Graficamente

Se rappresentiamo graficamente tutte le possibili mosse della Torre di Hanoi

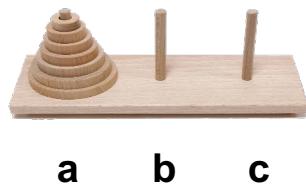


Con 1 disco:

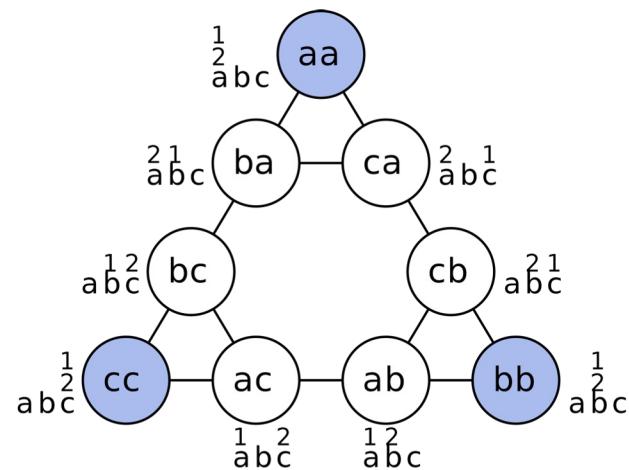


Torre di Hanoi: Graficamente

Se rappresentiamo graficamente tutte le possibili mosse della Torre di Hanoi

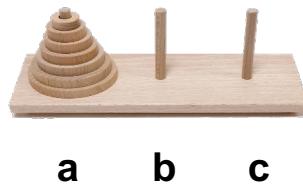


Con 2 dischi:

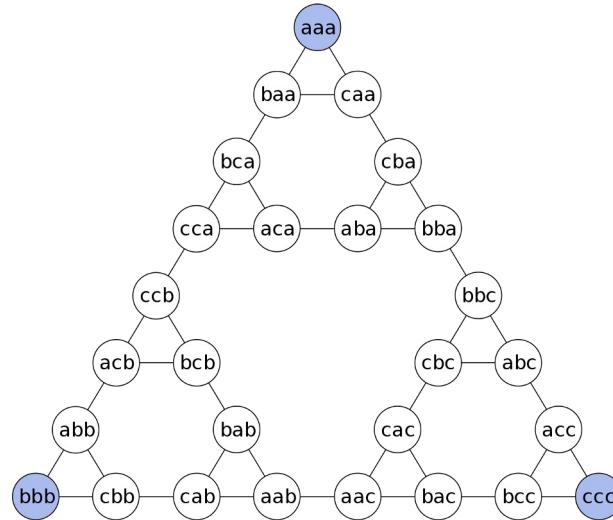


Torre di Hanoi: Graficamente

Se rappresentiamo graficamente tutte le possibili mosse della Torre di Hanoi

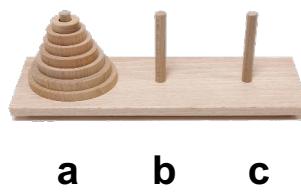


Con 3 dischi:

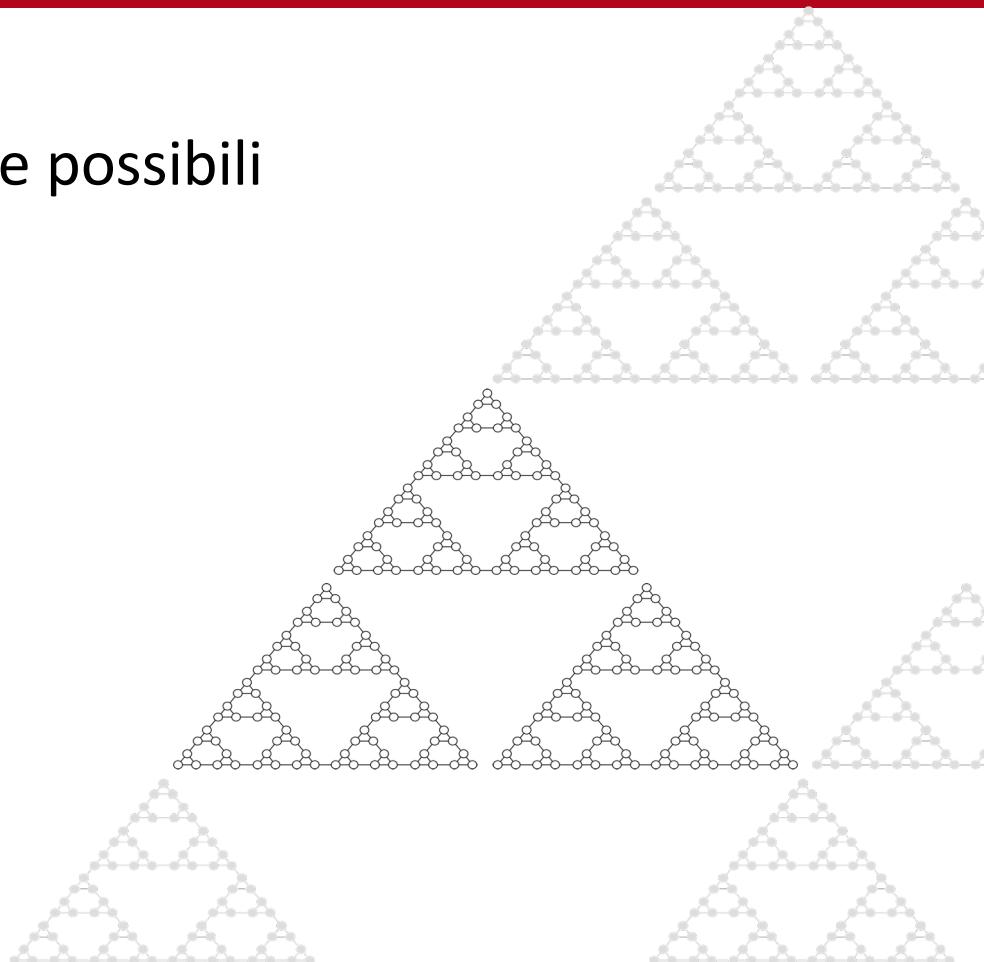


Torre di Hanoi: Graficamente

Se rappresentiamo graficamente tutte le possibili mosse della Torre di Hanoi



Con n dischi il grafo delle mosse ricorda la struttura frattale del **Triangolo di Sierpiński**



Acknowledgements

Grazie per l'attenzione!

giulio.martini@igi.cnr.it



- Andrea Rigoni e Niccolò Pretto: slide iniziali (AA. 2019/20)
- Andrea Tonon: evoluzione delle slide con integrazione di esercizi (AA. 2020/2021)
- Luca Trevisan: slide ricorsione natura (AA. 2021/2022)