

# Tecniche Algoritmiche / 1

## Divide et Impera

Jocelyne Elias

<https://www.unibo.it/sitoweb/jocelyne.elias>

**Moreno Marzolla**

<https://www.moreno.marzolla.name/>

Dipartimento di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

Copyright © Alberto Montresor, Università di Trento, Italy

<http://cricca.disi.unitn.it/montresor/teaching/asd/>

Copyright © 2009—2016, 2021 Moreno Marzolla, Università di Bologna, Italy

<https://www.moreno.marzolla.name/teaching/ASD/>



*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit*

*<http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

# Tecniche algoritmiche

- **Divide-et-impera**
  - Un problema viene suddiviso in sotto-problemi *indipendenti*, che vengono risolti *ricorsivamente* (top-down)
  - Ambito: problemi di decisione, ricerca
- **Programmazione dinamica**
  - La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi *potenzialmente ripetuti*
  - Ambito: problemi di ottimizzazione
- **Algoritmi greedy**
  - Ad ogni passo si fa sempre la scelta che in quel momento appare ottima; le scelte fatte non vengono mai disfatte
  - Ambito: problemi di ottimizzazione

# Divide-et-impera

- Tre fasi:
  - *Divide*: Dividi il problema in sotto-problemi indipendenti, di dimensioni “minori”
  - *Impera*: Risolvi i sotto-problemi ricorsivamente
  - *Combina*: Unisci le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza
- Non esiste una “ricetta” unica per implementare un algoritmo divide-et-impera:
  - Ricerca binaria: “divide” banale, niente fase di “combina”
  - Quick Sort: “divide” complesso, niente fase di “combina”
  - Merge Sort: “divide” banale, “combina” complesso

# Ricerca del minimo

# Ricerca del minimo

- Dato un array  $A[1..n]$  di  $n > 0$  valori reali arbitrari, determinare il valore minimo
  - Nota che in questo esempio l'approccio divide-et-impera non è vantaggioso rispetto alla soluzione diretta
- **Idea:**
  - Divido l'array in due sottovettori di lunghezza (circa) uguale
    - Determino ricorsivamente i valori minimi dei due sottovettori
    - Il minimo dell'array di partenza è il più piccolo dei due minimi
  - Caso base: array di un singolo elemento
    - Il minimo in questo caso è il singolo elemento

# Implementazione

```
double MinDivideEtImpera(double A[1..n], integer i, integer j)
  if ( i > j ) then
    return +∞;
  elseif ( i = j ) then
    return A[i];
  else
    integer m ← Floor((i + j) / 2);
    double min1 ← MinDivideEtImpera( A, i, m );
    double min2 ← MinDivideEtImpera( A, m+1, j );
    if ( min1 < min2 ) then
      return min1;
    else
      return min2;
    endif
  endif
```

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 2T(n/2) + c_2 & \text{se } n > 1 \end{cases}$$

# Soluzione della ricorrenza

- **Master Theorem** (formulazione leggermente diversa rispetto a quella vista nel modulo 1):  
L'equazione di ricorrenza

$$T(n) = \begin{cases} d & \text{se } n=1 \\ aT(n/b) + cn^\beta & \text{se } n>1 \end{cases}$$

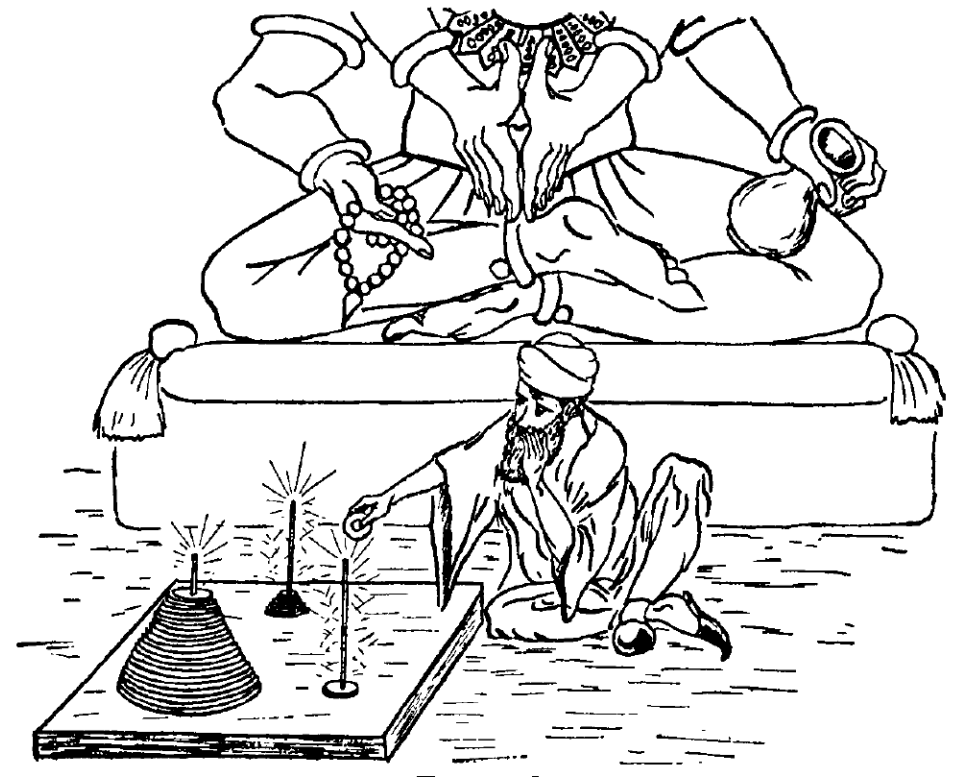
- posto  $\alpha = (\log a) / (\log b)$  ha soluzione:
  - $T(n) = O(n^\alpha)$  se  $\alpha > \beta$
  - $T(n) = O(n^\alpha \log n)$  se  $\alpha = \beta$
  - $T(n) = O(n^\beta)$  se  $\alpha < \beta$



# Le torri di Hanoi

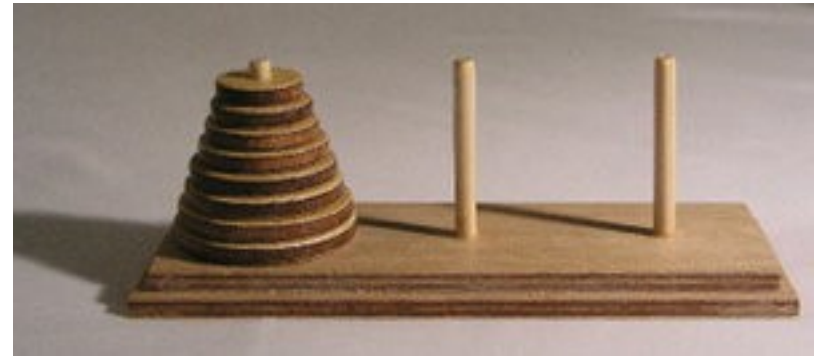
# Le torri di Hanoi

*Narra la leggenda che in un tempio indiano si trovi una stanza contenente una lastra di bronzo con sopra tre pioli di diamante. Dopo la creazione dell'universo, 64 dischi d'oro sono stati infilati in uno dei pioli in ordine decrescente di diametro, con il disco più largo appoggiato al piano di bronzo. Da allora, i monaci del tempio si alternano per spostare i dischi dal piolo originario in un altro piolo, seguendo le rigide regole di Brahma: i dischi vanno maneggiati uno alla volta, e non si deve mai verificare che un disco più largo sovrasti uno più piccolo sullo stesso piolo. Quando tutti i dischi saranno spostati su un altro piolo, l'universo avrà fine.*



Fonte: George Gamow, *One, two, tree... infinity!*, Dover Publications, 1947

# Le torri di Hanoi



[http://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](http://en.wikipedia.org/wiki/Tower_of_Hanoi)

- Gioco matematico
  - tre pioli
  - $n$  dischi di dimensioni diverse
  - Inizialmente tutti i dischi sono impilati in ordine decrescente (più piccolo in alto) nel piolo di sinistra
- Scopo del gioco
  - Impilare in ordine decrescente i dischi sul piolo di destra
  - Senza mai impilare un disco più grande su uno più piccolo
  - Muovendo un disco alla volta
  - Utilizzando se serve anche il piolo centrale

# Le torri di Hanoi

## Soluzione divide-et-impera

[http://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](http://en.wikipedia.org/wiki/Tower_of_Hanoi)

```
Hanoi(Stack p1, Stack p2, Stack p3, integer n)
  if (n == 1) then
    p3.push(p1.pop())
  else
    Hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    Hanoi(p2, p1, p3, n-1)
  endif
```



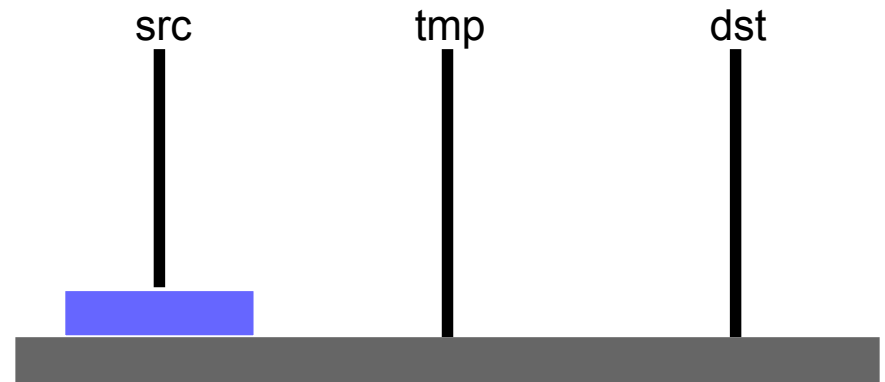
Sposta  $n$  dischi da p1 a p3  
usando p2 come appoggio

- **Divide:**
  - $n - 1$  dischi da p1 a p2
  - 1 disco da p1 a p3
  - $n - 1$  dischi da p2 a p3
- **Impera**
  - Esegui ricorsivamente gli spostamenti

# Idea

Per spostare  $n$  dischi dal piolo **src** al piolo **dst**, usando **tmp** come appoggio

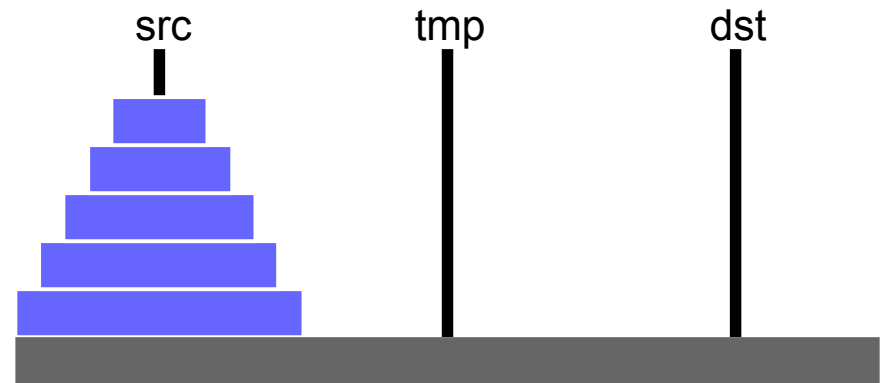
- Se c'è un solo disco
  - Spostalo da **src** a **dst**
- Se ci sono  $n > 1$  dischi
  - Sposta gli  $n - 1$  dischi in cima a **src** verso **tmp**, usando **dst** come appoggio
  - Sposta un disco da **src** a **dst**
  - Sposta  $n - 1$  dischi in cima a **tmp** verso **dst**, usando **src** come appoggio



# Idea

Per spostare  $n$  dischi dal piolo **src** al piolo **dst**, usando **tmp** come appoggio

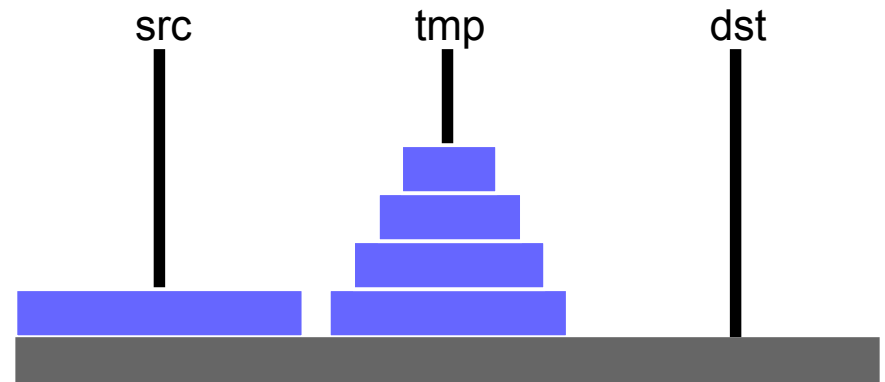
- Se c'è un solo disco
  - Spostalo da **src** a **dst**
- Se ci sono  $n > 1$  dischi
  - Sposta gli  $n - 1$  dischi in cima a **src** verso **tmp**, usando **dst** come appoggio
  - Sposta un disco da **src** a **dst**
  - Sposta  $n - 1$  dischi in cima a **tmp** verso **dst**, usando **src** come appoggio



# Idea

Per spostare  $n$  dischi dal piolo **src** al piolo **dst**, usando **tmp** come appoggio

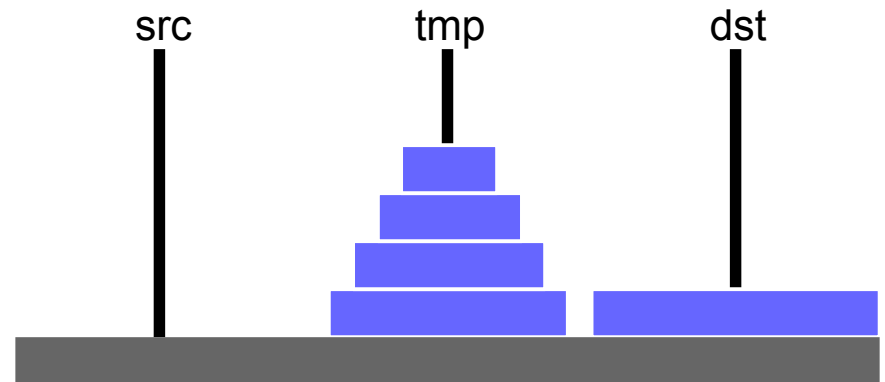
- Se c'è un solo disco
  - Spostalo da **src** a **dst**
- Se ci sono  $n > 1$  dischi
  - Sposta gli  $n - 1$  dischi in cima a **src** verso **tmp**, usando **dst** come appoggio
  - Sposta un disco da **src** a **dst**
  - Sposta  $n - 1$  dischi in cima a **tmp** verso **dst**, usando **src** come appoggio



# Idea

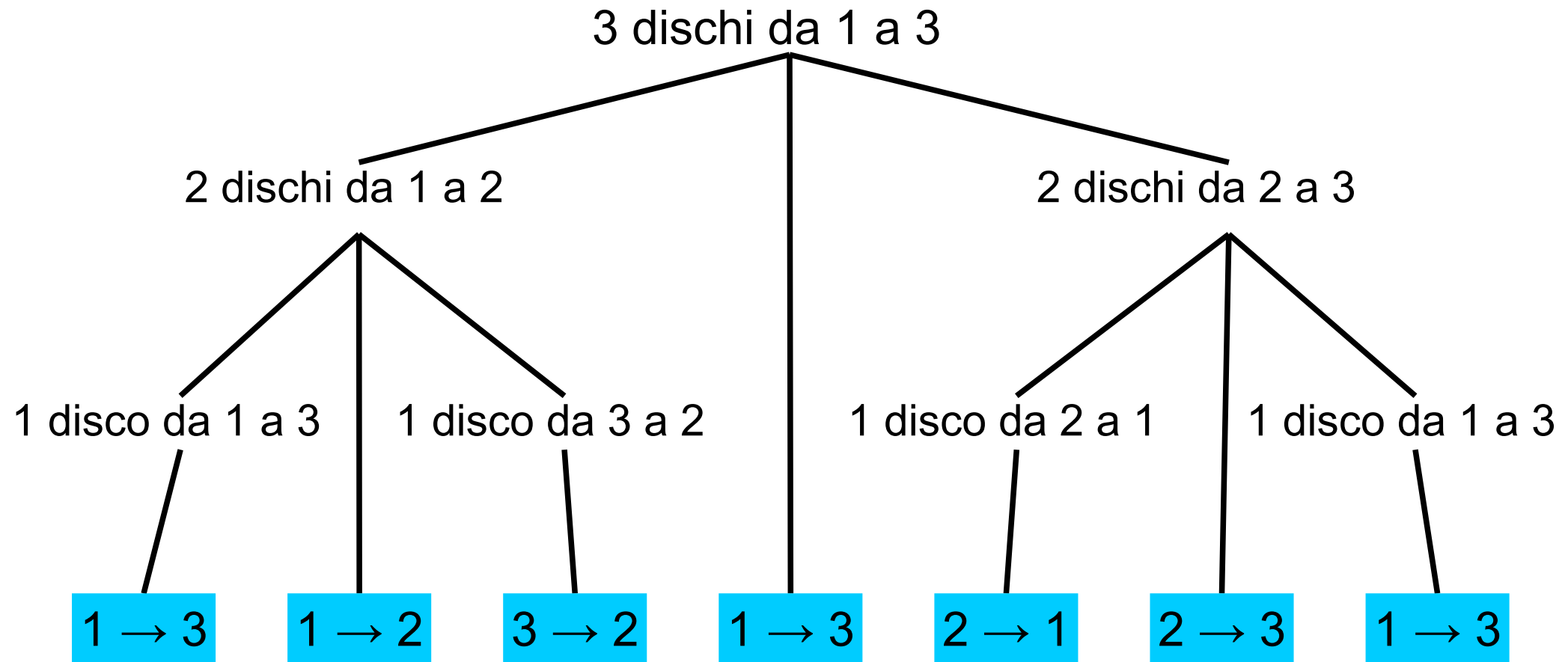
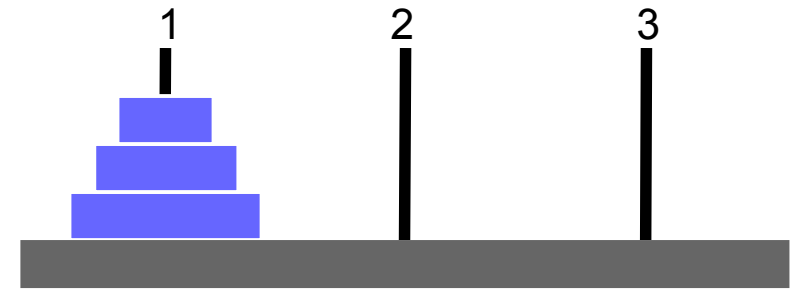
Per spostare  $n$  dischi dal piolo **src** al piolo **dst**, usando **tmp** come appoggio

- Se c'è un solo disco
  - Spostalo da **src** a **dst**
- Se ci sono  $n > 1$  dischi
  - Sposta gli  $n - 1$  dischi in cima a **src** verso **tmp**, usando **dst** come appoggio
  - Sposta un disco da **src** a **dst**
  - Sposta  $n - 1$  dischi in cima a **tmp** verso **dst**, usando **src** come appoggio

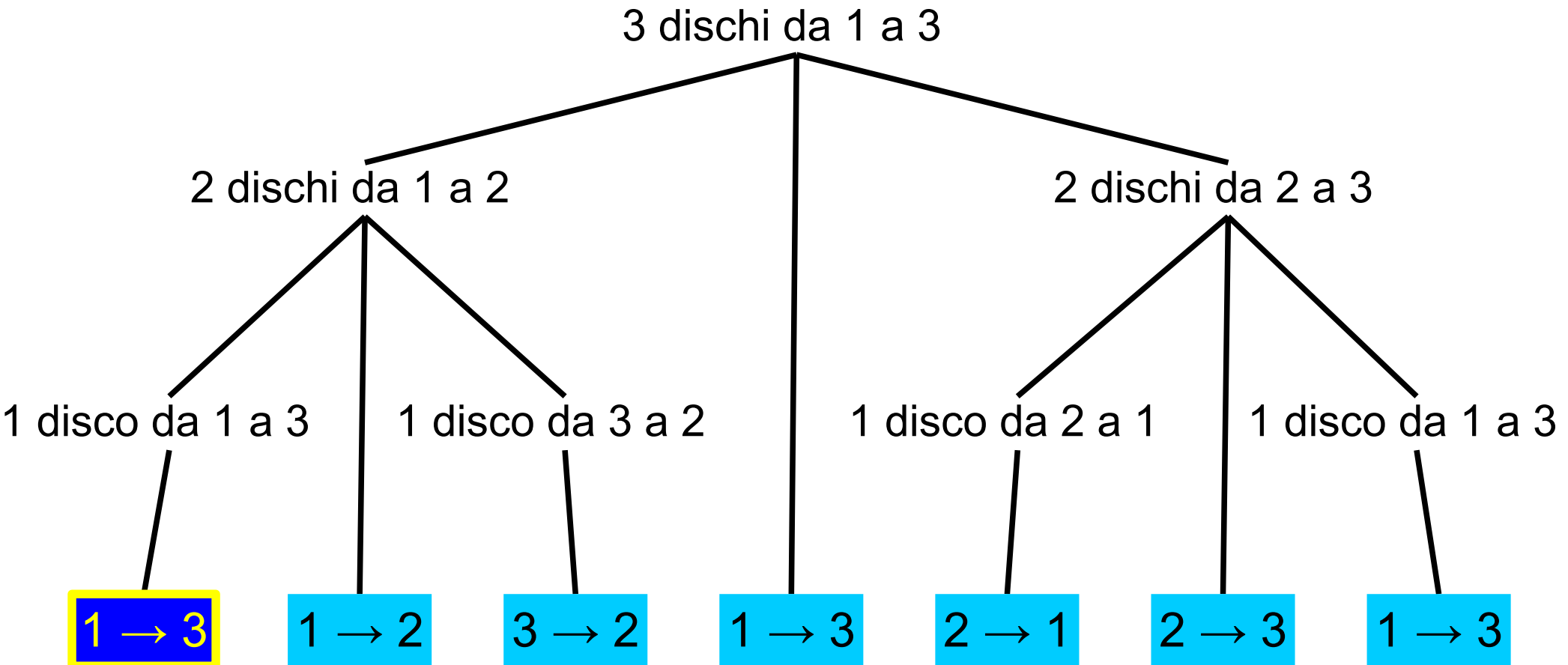
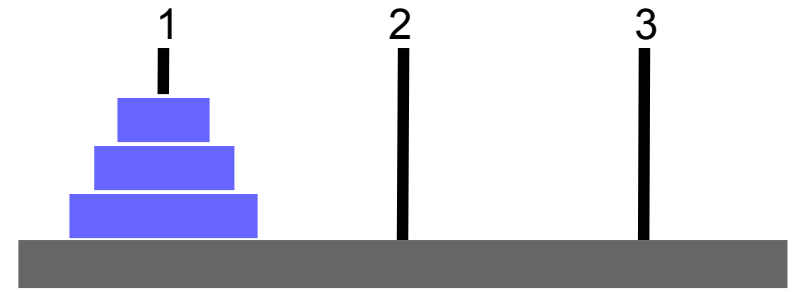




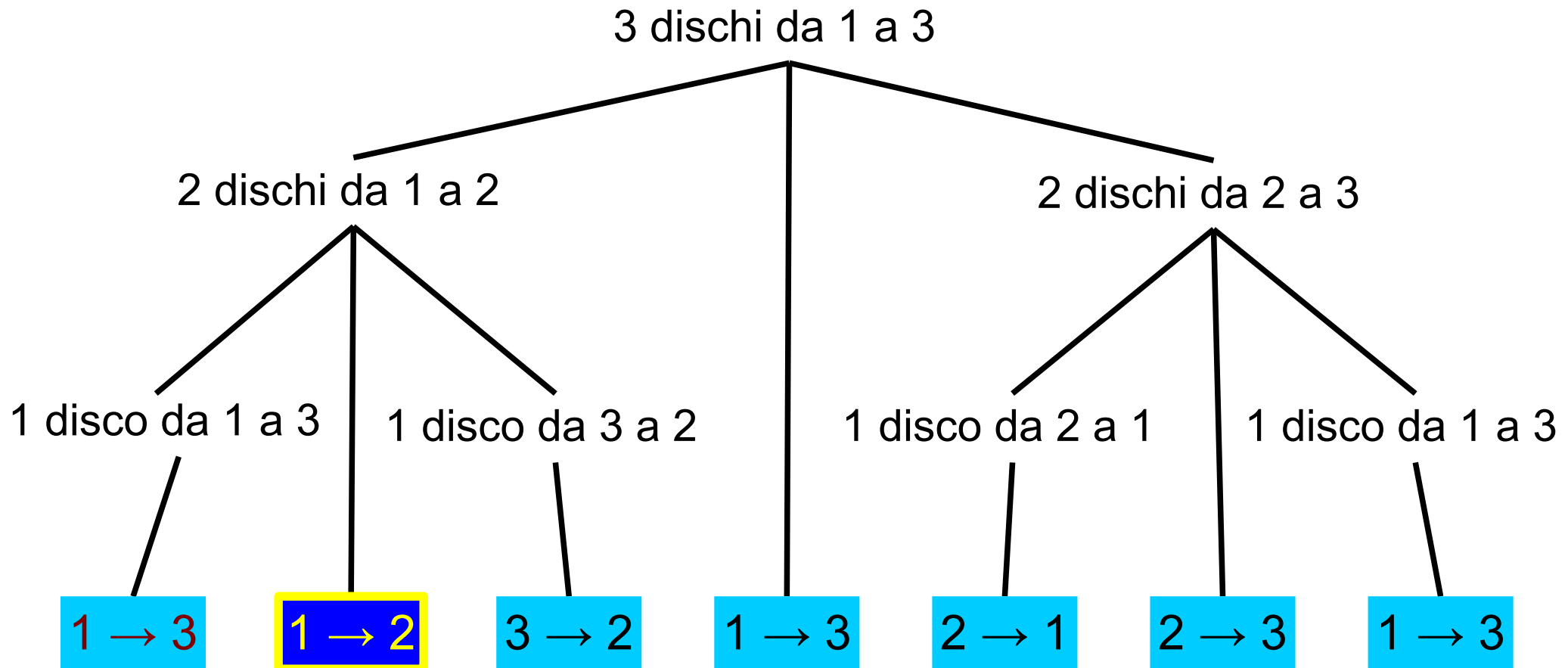
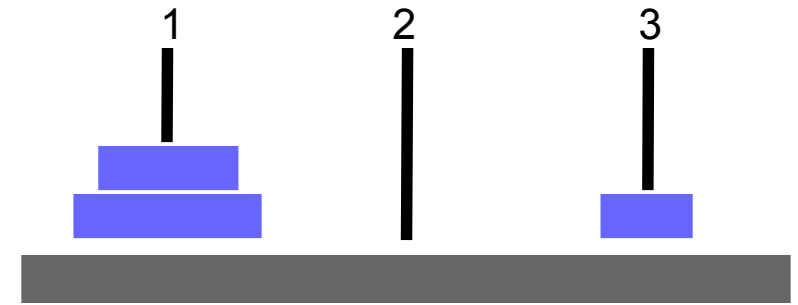
# Esempio con 3 dischi



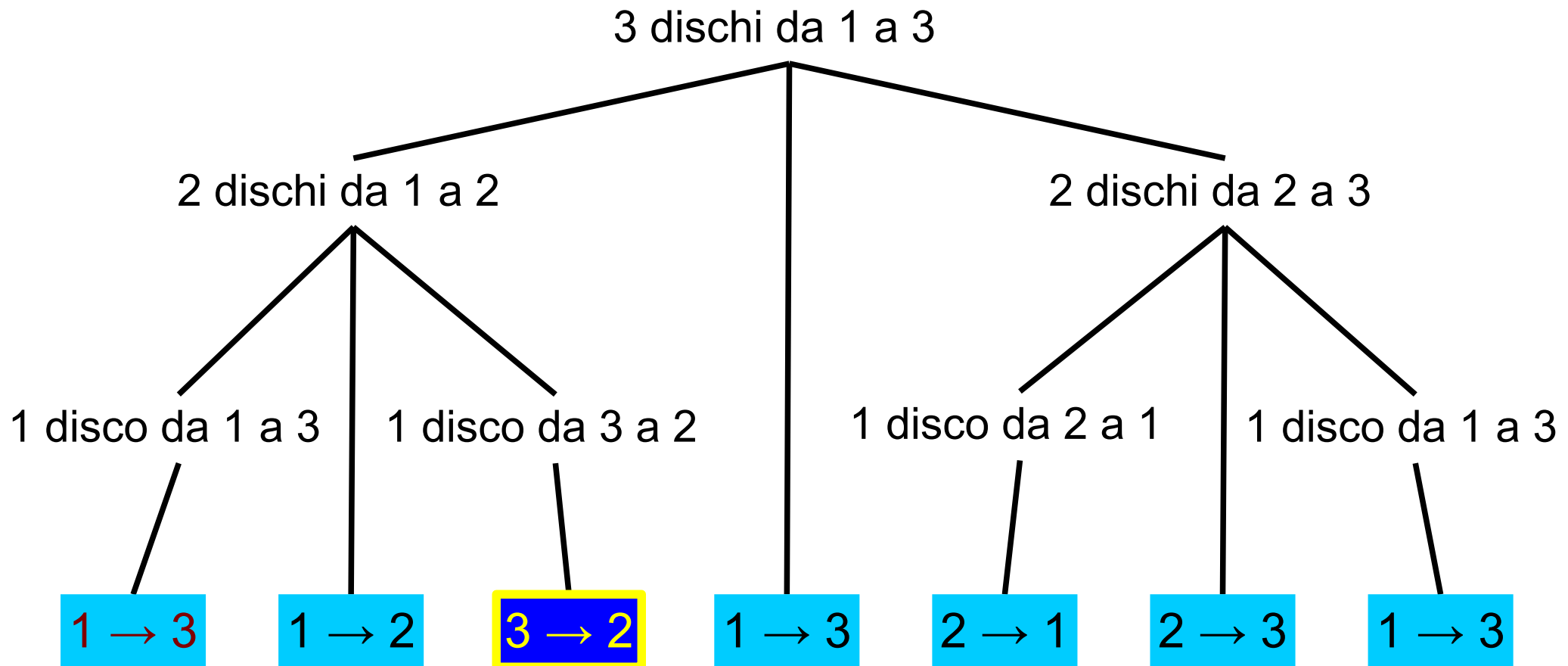
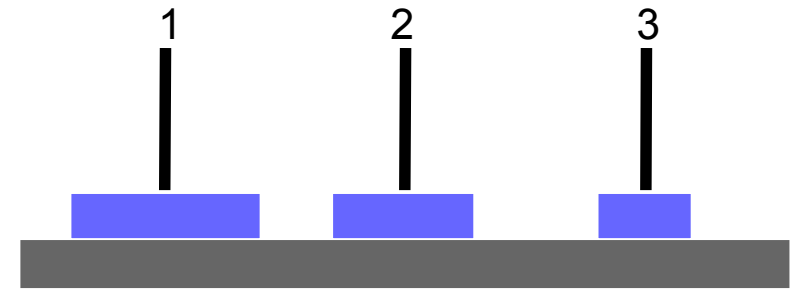
# Esempio con 3 dischi



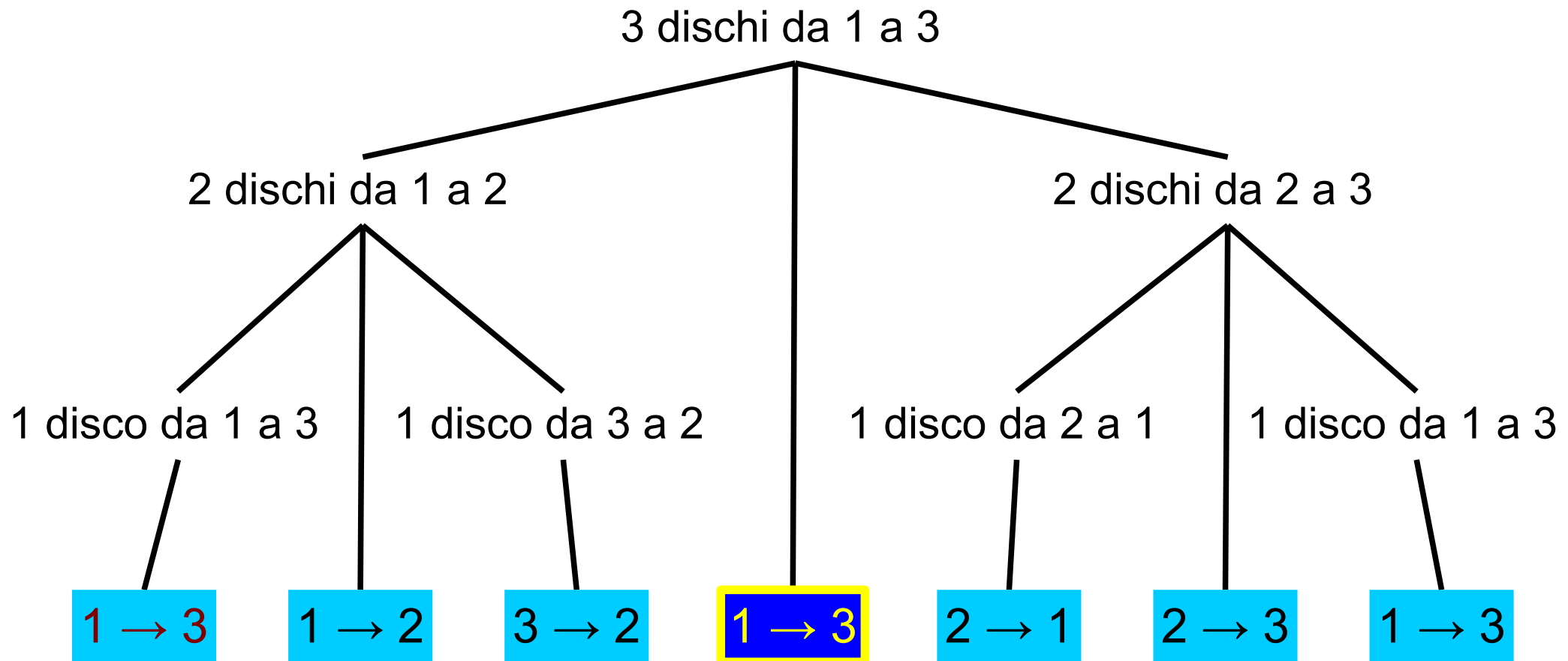
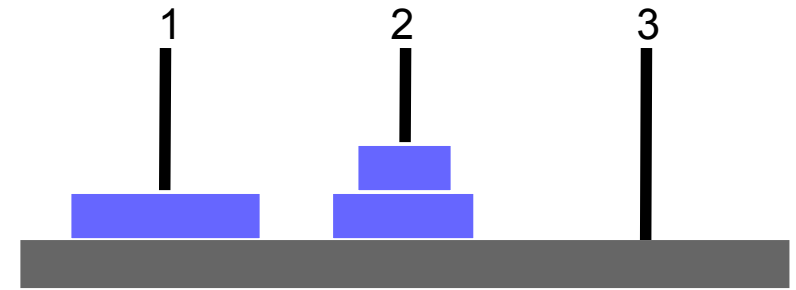
# Esempio con 3 dischi



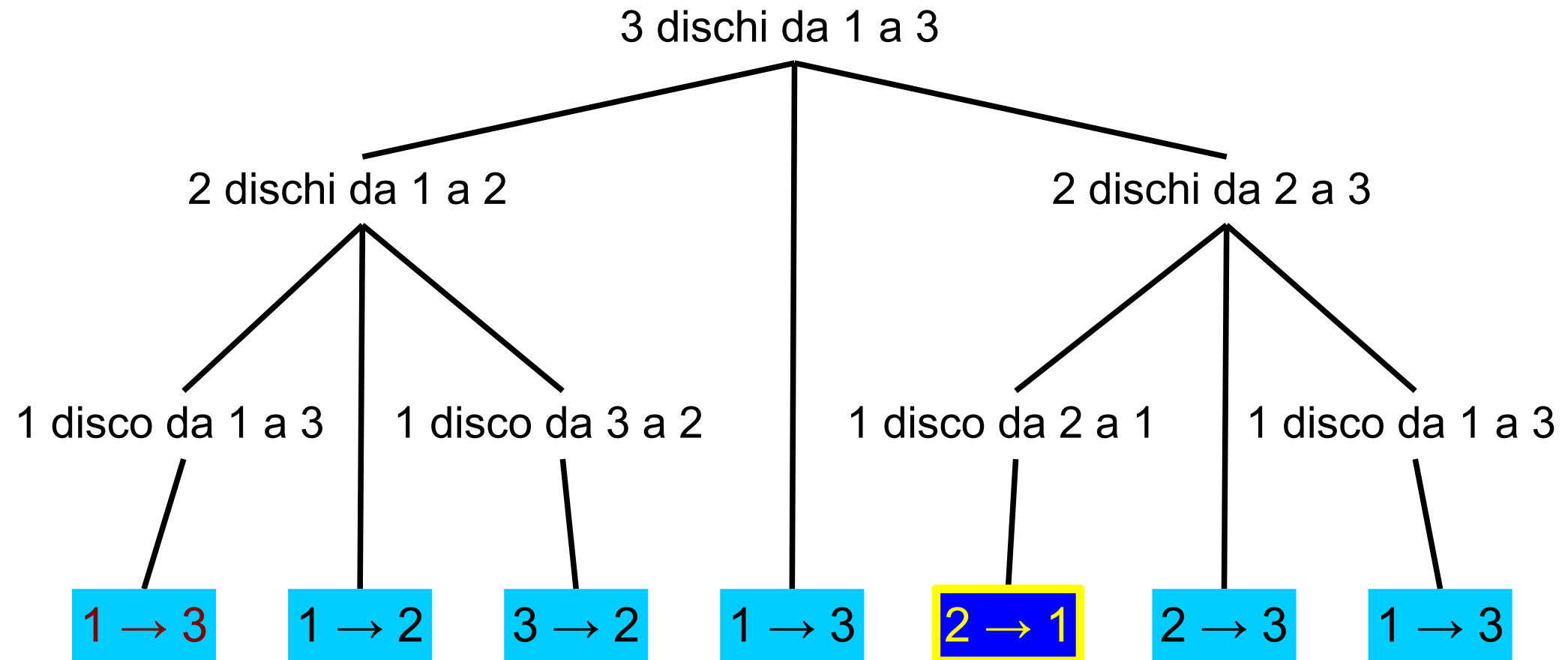
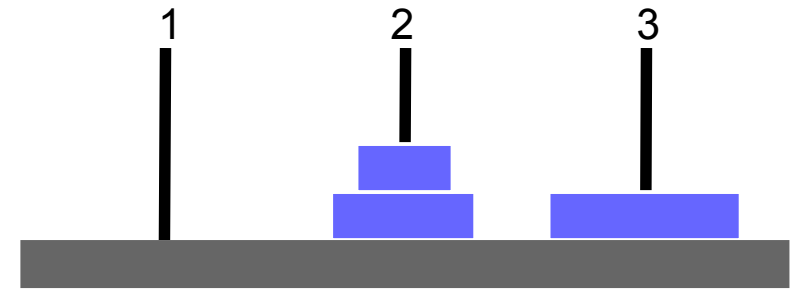
# Esempio con 3 dischi



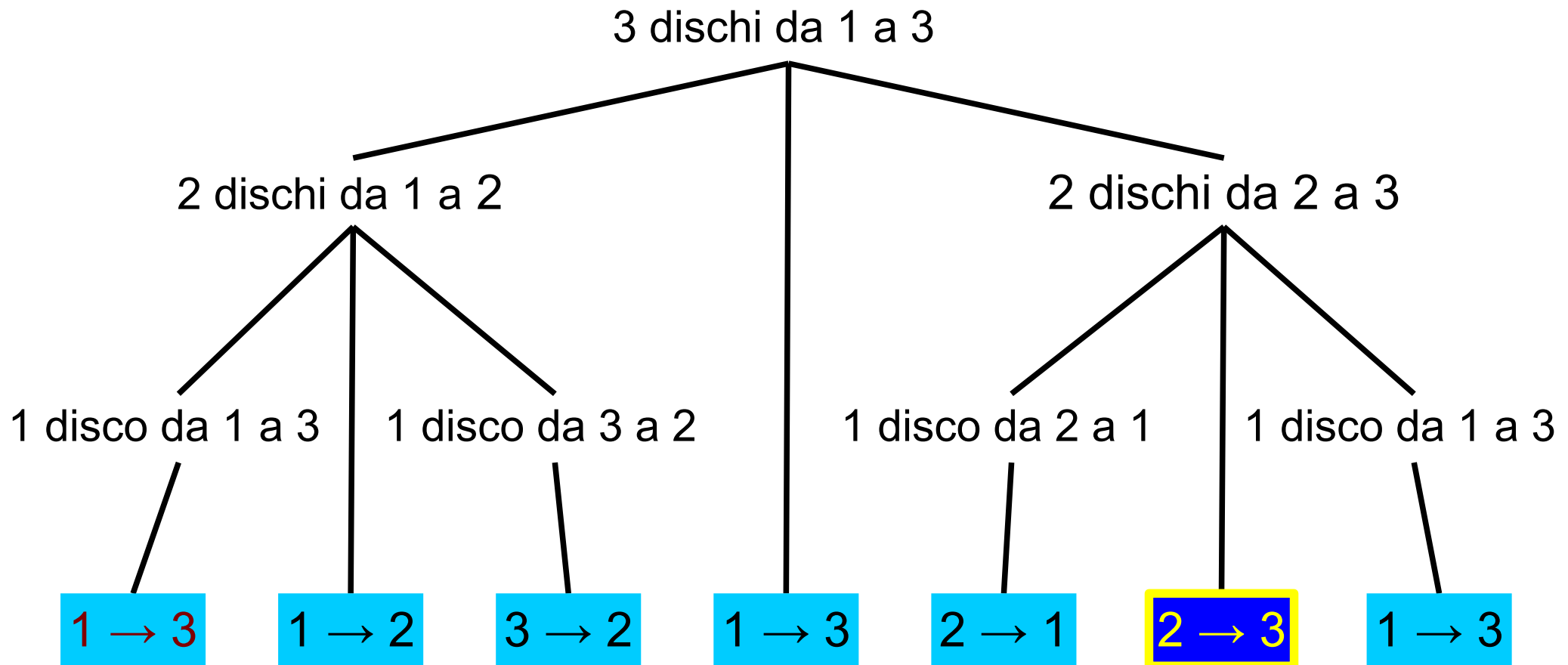
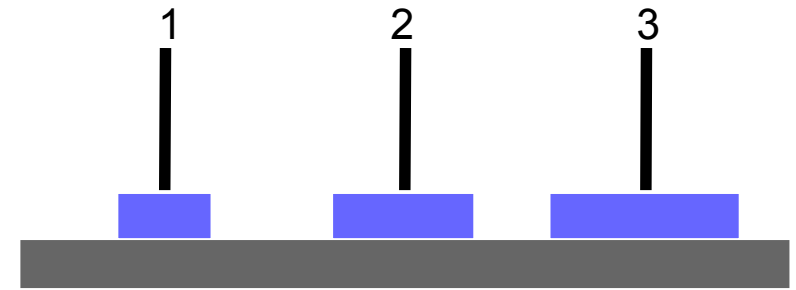
# Esempio con 3 dischi



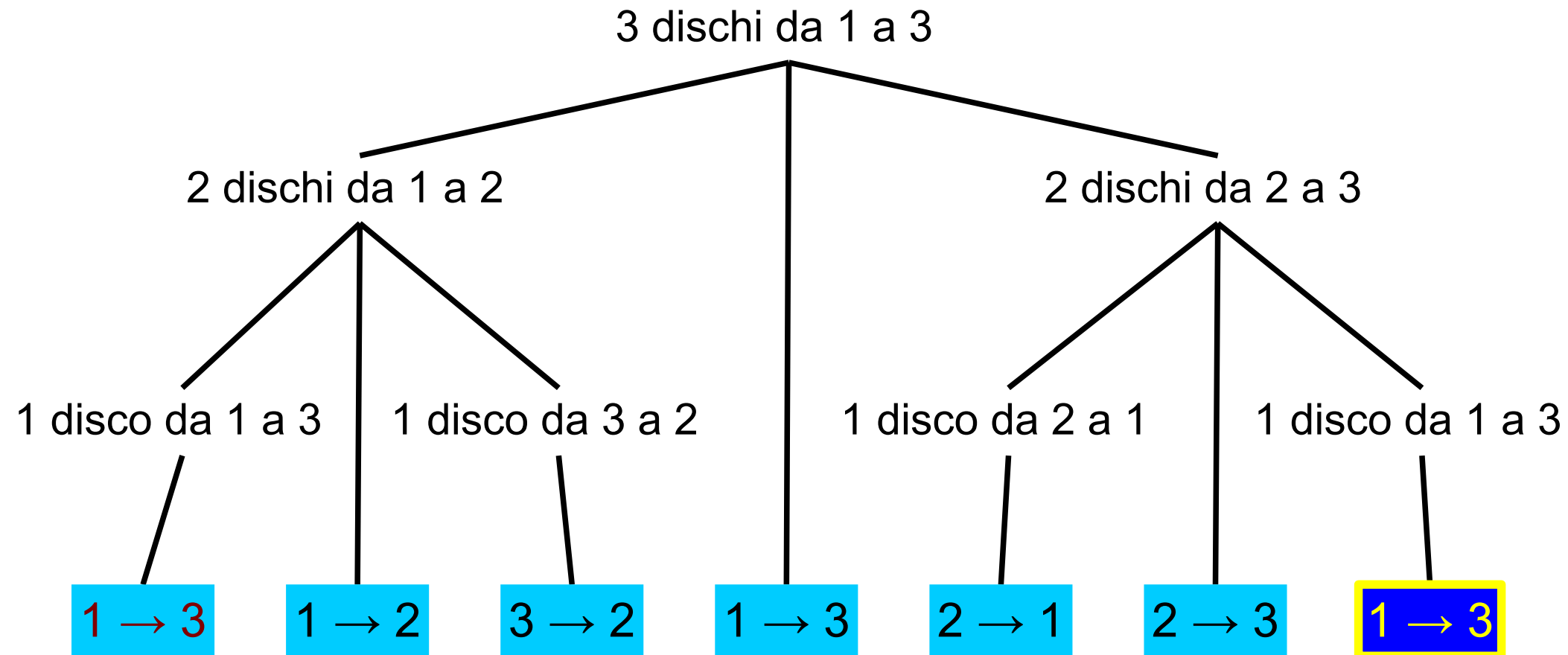
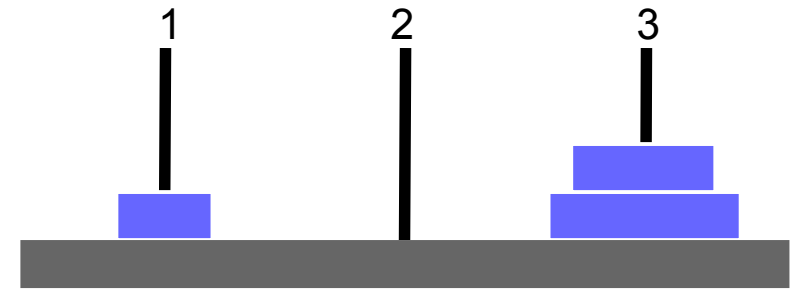
# Esempio con 3 dischi



# Esempio con 3 dischi



# Esempio con 3 dischi





# Le torri di Hanoi

## Soluzione divide-et-impera

```
Hanoi(Stack p1, Stack p2, Stack p3, integer n)
  if (n = 1) then
    p3.push(p1.pop())
  else
    Hanoi(p1, p3, p2, n-1)
    p3.push(p1.pop())
    Hanoi(p2, p1, p3, n-1)
  endif
```

- Costo computazionale:
  - $T(1) = 1$
  - $T(n) = 2 T(n - 1) + 1$  per  $n > 1$
- **Domanda:** Quale è la soluzione della ricorrenza?
  - Se i monaci effettuano una mossa al secondo, per trasferire tutti i 64 dischi servirebbe un tempo pari a circa 127 volte l'età del nostro sole

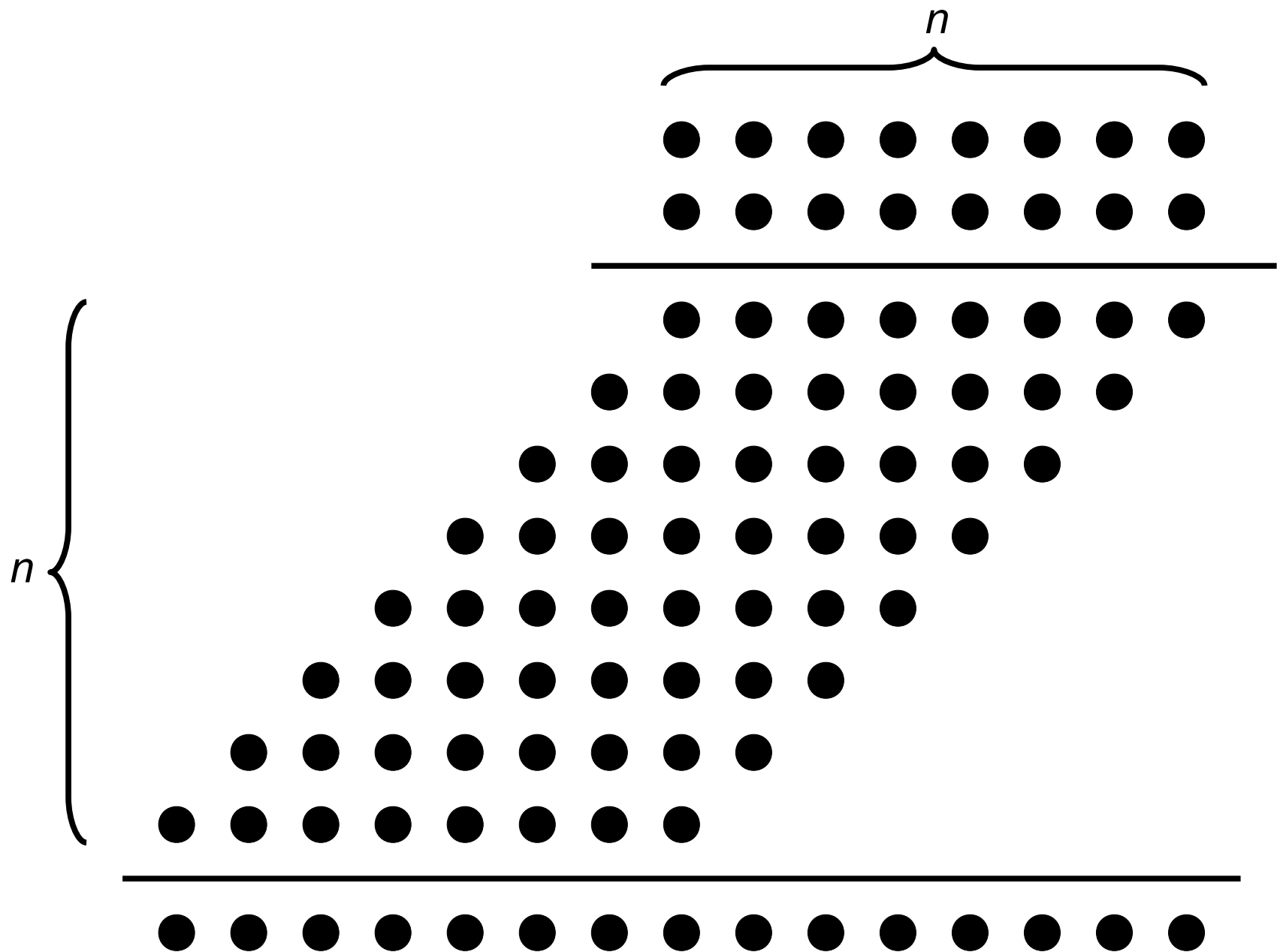
# Moltiplicazione di interi

# Moltiplicazione di interi di grandezza arbitraria

- Consideriamo due interi di  $n$  cifre decimali,  $X$  e  $Y$

$$X = x_{n-1}x_{n-2}\dots x_1x_0 = \sum_{i=0}^{n-1} x_i \times 10^i$$
$$Y = y_{n-1}y_{n-2}\dots y_1y_0 = \sum_{i=0}^{n-1} y_i \times 10^i$$

- Vogliamo calcolare il prodotto  $XY$ 
  - L'algoritmo che abbiamo imparato a scuola ha costo  $O(n^2)$
  - Proviamo a fare di meglio con un algoritmo di tipo divide et impera



# Idea

- Supponiamo che sia  $X$  che  $Y$  abbiano lo stesso numero di cifre (possiamo aggiungere zeri all'inizio)
- Dividiamo le sequenze di cifre in due parti uguali

$$X = X_1 \times 10^{n/2} + X_0$$

$$Y = Y_1 \times 10^{n/2} + Y_0$$

$X_1$	$X_0$
$Y_1$	$Y_0$

- Calcoliamo il prodotto come:

$$\begin{aligned} X \times Y &= (X_1 10^{n/2} + X_0) \times (Y_1 10^{n/2} + Y_0) \\ &= (X_1 Y_1) \times 10^n + (X_1 Y_0 + X_0 Y_1) \times 10^{n/2} + X_0 Y_0 \end{aligned}$$

# Osservazione

$$X \times Y = (\underline{X_1 Y_1}) \times 10^n + (\underline{X_1 Y_0} + \underline{X_0 Y_1}) \times 10^{n/2} + \underline{X_0 Y_0}$$

- La moltiplicazione per  $10^n$  richiede tempo  $O(n)$ 
  - Equivale ad uno shift a sinistra di  $n$  posizioni
- Ci sono 4 prodotti di numeri di  $n/2$  cifre
- Possiamo scrivere la relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 4T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

- Soluzione (Master Theorem, caso 1):  $\Theta(n^2)$ 
  - Non abbiamo migliorato nulla rispetto all'algoritmo banale :-)

# Miglioramento

- Se poniamo

$$P_1 = (X_1 + X_0) \times (Y_1 + Y_0)$$

$$P_2 = (X_1 Y_1)$$

$$P_3 = (X_0 Y_0)$$

possiamo scrivere

$$X \times Y = P_2 10^n + (P_1 - P_2 - P_3) \times 10^{n/2} + P_3$$

- Il calcolo di P1, P2 e P3 richiede in tutto solo 3 prodotti tra numeri di  $n/2$  cifre

# Analisi del miglioramento

- Otteniamo la nuova relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

- In base al Master Theorem (caso 1) la soluzione è

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$$



# Moltiplicazione di matrici

# Moltiplicazione di matrici

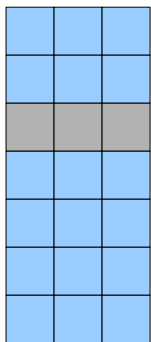
Dimensione  
 $p \times q$

Dimensione  
 $p \times c$

Dimensione  
 $c \times q$

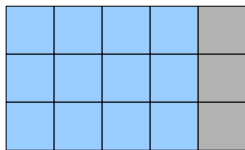
$$P[i, j] = \sum_{k=1}^c A[i, k] \times B[k, j]$$

```
double[1..p,1..q] mat_prod(double
A[1..p,1..c], double B[1..c,1..q])
double P[1..p,1..q];
for integer i ← 1 to p do
  for integer j ← 1 to q do
    double sum ← 0;
    for integer k ← 1 to c do
      sum ← sum + A[i,k]*B[k,j];
    endfor
    P[i,j] ← sum;
  endfor
endfor
return P;
```



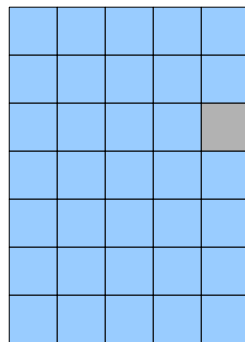
A: 7×3

×



B: 3×5

=



P: 7×5

Complessità

- $T(p, c, q) = p \cdot c \cdot q$
- $T(n) = \Theta(n^3)$

# Proviamo a migliorare l'algoritmo

- Suddividiamo le matrici  $n \times n$  in quattro matrici  $n/2 \times n/2$

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

- La matrice prodotto  $P$  risulta definita come

$$P = \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix}$$

- Equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 8T(n/2) + c_2n^2 & \text{altrimenti} \end{cases}$$

La soluzione della ricorrenza è  
 **$T(n) = \Theta(n^3)$**   
cioè non meglio dell'algoritmo "banale"

# Come migliorare il prodotto fra matrici

7 moltiplicazioni  
di matrici  $n/2 \times n/2$

- Calcoliamo alcuni termini intermedi

$$M_1 = (A_{2,1} + A_{2,2} - A_{1,1}) \times (B_{2,2} - B_{1,2} + B_{1,1})$$

$$M_2 = A_{1,1} \times B_{1,1}$$

$$M_3 = A_{1,2} \times B_{2,1}$$

$$M_4 = (A_{1,1} - A_{2,1}) \times (B_{2,2} - B_{1,1})$$

$$M_5 = (A_{2,1} + A_{2,2}) \times (B_{1,2} - B_{1,1})$$

$$M_6 = (A_{1,2} - A_{2,1} + A_{1,1} - A_{2,2}) \times B_{2,2}$$

$$M_7 = A_{2,2} \times (B_{1,1} + B_{2,2} - B_{1,2} - B_{2,1})$$

- Matrice finale: 
$$P = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

# Analisi del miglioramento

- Si ottiene la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 7T(n/2) + c_2n^2 & \text{altrimenti} \end{cases}$$

- Dal Master Theorem (caso 1) si ricava

$$T(n) = \Theta(n^{\log 7 / \log 2}) \approx \Theta(n^{2.81})$$

Sottovettore non vuoto di valore massimo

# Sottovettore di valore massimo

- Vettore  $V[1..n]$  di  $n$  valori reali arbitrari
- Vogliamo individuare un sottovettore **non vuoto** di  $V$  la somma dei cui elementi sia massima

Elementi contigui!

3	-5	10	2	-3	1	4	-8	7	-6	-1
---	----	----	---	----	---	---	----	---	----	----

- Domanda: quanti sono i sottovettori di  $V$ ?

- 1 sottovettore di lunghezza  $n$
- 2 sottovettori di lunghezza  $n - 1$
- 3 sottovettori di lunghezza  $n - 2$
- ...
- $k$  sottovettori di lunghezza  $n - k + 1$
- ...
- $n$  sottovettori di lunghezza 1

Risposta:

$n(n+1)/2 = \Theta(n^2)$  sottovettori

# Prima versione

```
double VecSum1( double v[1..n] )
  double smax ← v[1];
  for integer i ← 1 to n do
    for integer j ← i to n do
      double s ← 0;
      for integer k ← i to j do
        s ← s + v[k];
      endfor
      if (s > smax) then
        smax ← s;
      endif
    endfor
  endfor
  return smax;
```

Costo?



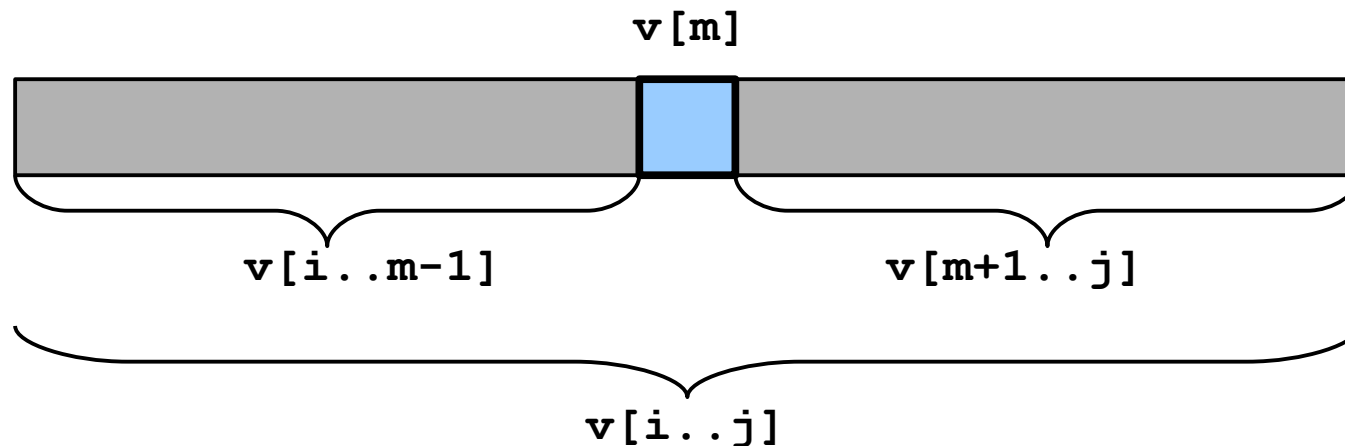
# Seconda versione

```
double VecSum2( double v[1..n] )  
  double smax ← v[1];  
  for integer i ← 1 to n do  
    double s ← 0;  
    for integer j ← i to n do  
      s ← s + v[j];  
      if (s > smax) then  
        smax ← s;  
      endif  
    endfor  
  endfor  
  return smax;
```

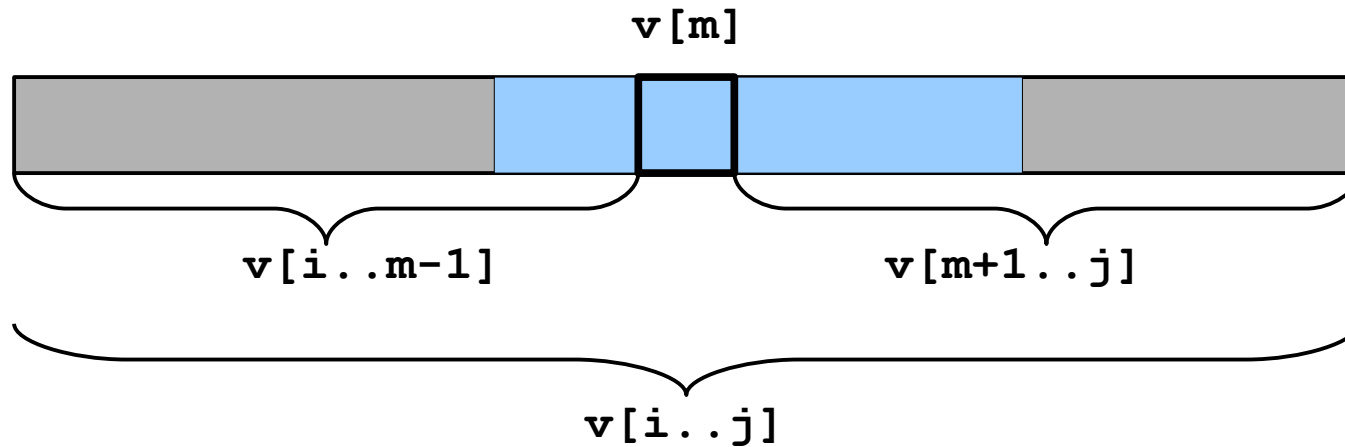
Costo?

# Strategia divide-et-impera

- Dividiamo il vettore in due parti, separate dall'elemento in posizione centrale  $v[m]$
- Il sottovettore di somma massima potrebbe trovarsi:
  - Interamente nella prima metà  $v[i..m-1]$
  - Interamente nella seconda metà  $v[m+1..j]$
  - “A cavallo” tra la prima e la seconda metà

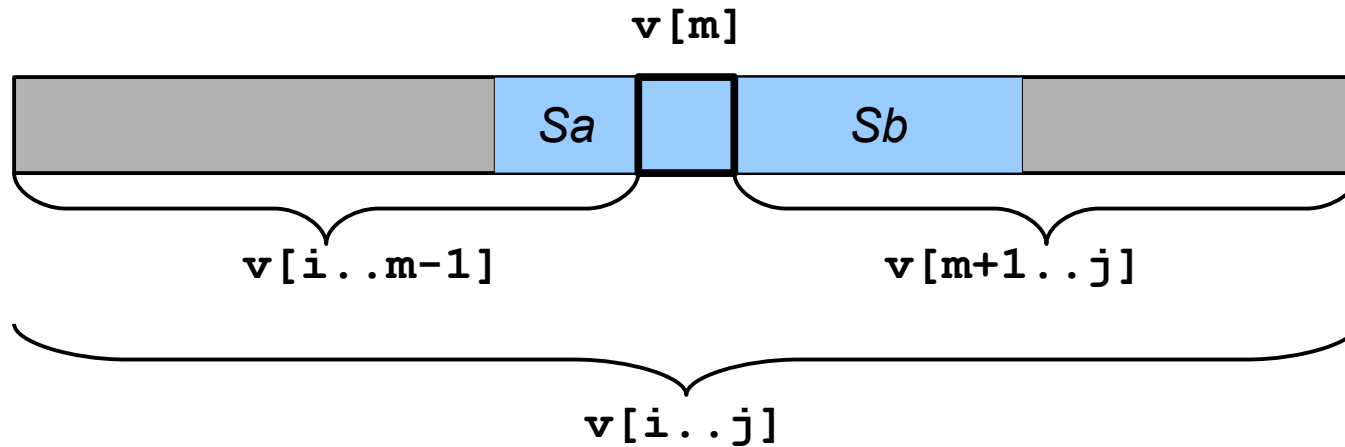


# Strategia divide-et-impera



- La parte più complicata è la ricerca del sottovettore di somma massima che contiene  $v[m]$
- Tale vettore
  - Può includere una parte prima di  $v[m]$
  - Può includere una parte dopo di  $v[m]$

# Strategia divide-et-impera



- Calcolo la max somma  $Sa$  tra tutti i sottovettori (incluso quello vuoto) il cui ultimo elemento è  $v[m-1]$
- Calcolo la max somma  $Sb$  tra tutti i sottovettori (incluso quello vuoto) il cui primo elemento è  $v[m+1]$
- Il sottovettore di somma max che include  $v[m]$  avrà somma  $(v[m] + Sa + Sb)$

# Strategia divide-et-impera

```
double VecSumDI( double v[1..n], integer i, integer j)
  if (i = j) then
    return v[i];
  else
    integer k, m  $\leftarrow$  (i+j)/2;
    double sleft  $\leftarrow$  VecSumDI(v,i,m-1);
    double sright  $\leftarrow$  VecSumDI(v,m+1,j);
    double sa, sb, s;
    sa  $\leftarrow$  0; s  $\leftarrow$  0;
    for k  $\leftarrow$  m-1 downto i do
      s  $\leftarrow$  s + v[k]
      if (s > sa) sa  $\leftarrow$  s;
    endfor
    sb  $\leftarrow$  0; s  $\leftarrow$  0;
    for k  $\leftarrow$  m+1 to j do
      s  $\leftarrow$  s + v[k];
      if (s > sb) sb  $\leftarrow$  s;
    endfor
    return max(sleft, sright, v[m] + sa + sb);
  endif
```

# Strategia divide-et-impera

- Vedi implementazione Java
- Costo asintotico?

# Metodo divide-et-impera

- Quando applicare divide-et-impera
  - I passi “divide” e “combina” devono essere “semplici”
  - Idealmente, il costo complessivo deve essere migliore del corrispondente algoritmo iterativo
  - Esempio **buono**: ordinamento
  - Esempio **non buono**: ricerca del minimo

# Esercizio

- Si consideri un array  $A[1..n]$  di  $n$  valori reali, non necessariamente distinti, **ordinato in senso non decrescente**. Scrivere un algoritmo **divide-et-impera** che restituisca *true* se e solo se in  $A$  sono presenti dei valori duplicati
  - In altre parole l'algoritmo ritorna *true* se esiste almeno un valore di  $A$  che compare almeno due volte, *false* altrimenti