ORDINAMENTO
QUICKSORT

INFORMATICA

QUICKSORT

- Abbiamo visto due algoritmi di ordinamento per comparazione che sono ottimali in termini di tempo: $\Theta(n \log n)$
- Ora vedremo un algoritmo che nel caso peggiore richiede tempo quadratico…
- …ma nel caso medio richiede tempo $O(n \log n)$
- Domanda: perché potrebbe avere senso studiare questo algoritmo?

QUICKSORT: STORIA

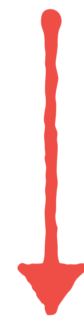
- Ideato da Tony Hoare (vincitore del premio Turing nel 1980) nel 1959-60
- Quando ben implementato il Quicksort è, nella pratica, più veloce di mergesort e heapsort
- Questo nonostante abbia un caso peggiore quadratico…
- … perché il caso **medio** è $O(n \log n)$

QUICKSORT: IDEA DI BASE

- Il quick sort è un algoritmo "divide et impera"
- L'idea di base è quella di scegliere in un array di n elementi un **pivot**, spostare gli elementi più piccoli prima del pivot e quelli più grandi dopo il pivot.
- Se applichiamo ricorsivamente lo stesso algoritmo ai due sotto-array risultanti (elementi minori e maggiori del pivot) otteniamo un array ordinato

PROCEDURA DI PARTIZIONAMENTO

Pivot



Dobbiamo partizionare l'array in base al pivot

Pivot

Il Pivot adesso è già nella posizione corretta!

Tutti gli elementi minori lo precedono e quelli maggiori lo seguono

Ora dobbiamo fare la stessa operazione sui due sottoarray

PROCEDURA DI PARTIZIONAMENTO

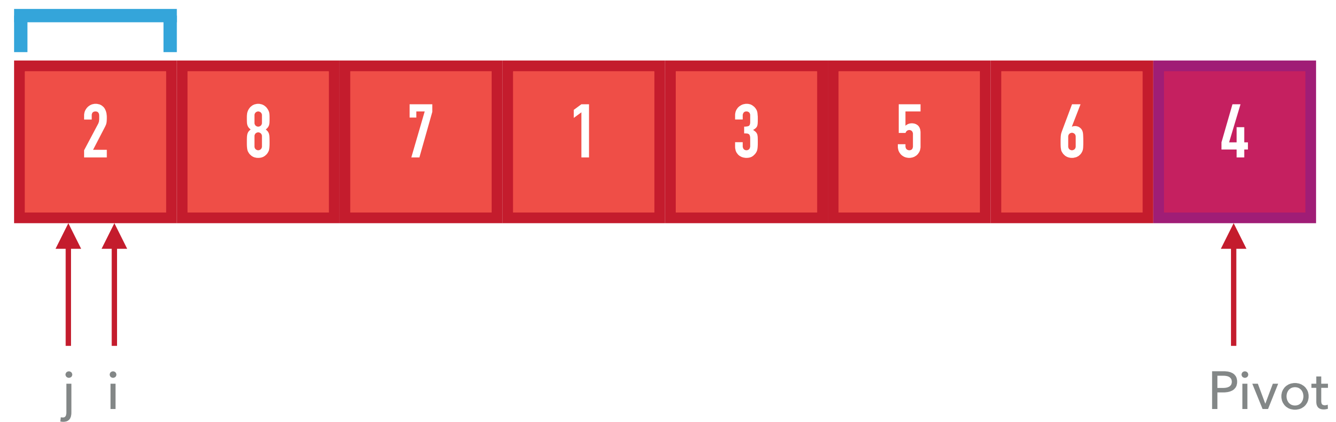
- Dobbiamo definire la procedura di partizionamento in modo efficiente
- Ne esistono diverse, noi vediamo lo schema di partizionamento di Hoare
- Idea di base: teniamo due indici:
  - i indica l'ultimo degli elementi minori del pivot
  - j viene utilizzato per scorrere l'array

PROCEDURA DI PARTIZIONAMENTO

PROCEDURA DI PARTIZIONAMENTO

Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

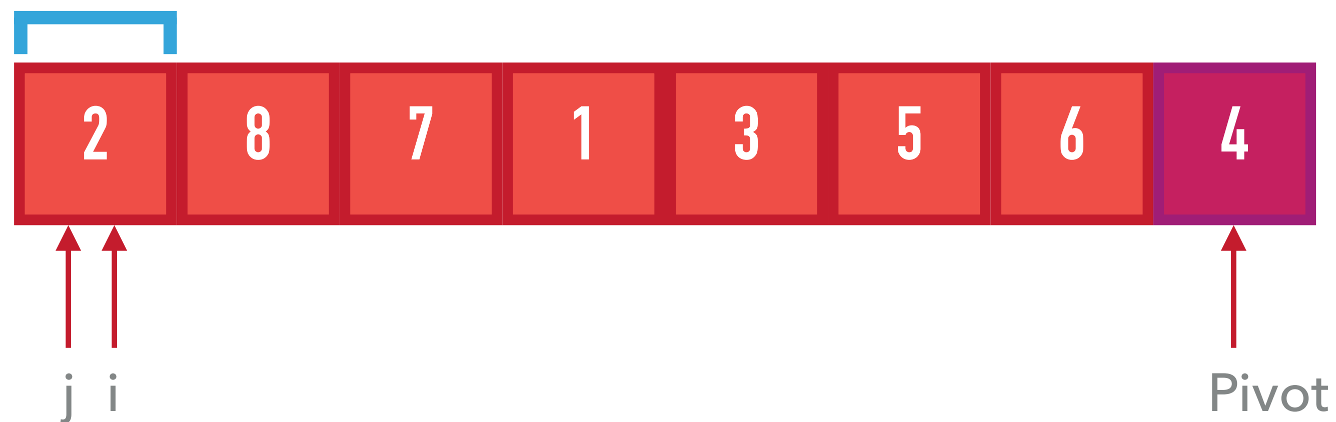
PROCEDURA DI PARTIZIONAMENTO



Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

In questo caso non cambia nulla (i è uguale a j)

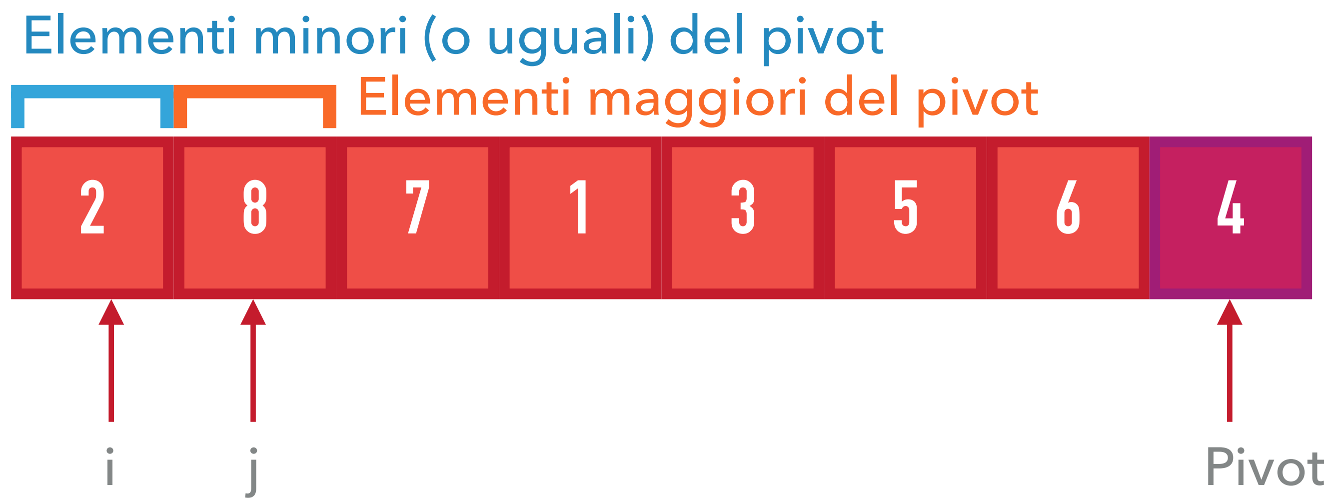
PROCEDURA DI PARTIZIONAMENTO



Se A[j] è minore (o uguale) del pivot incrementiamo i e scambiamo A[i] e A[j]

In questo caso non cambia nulla (i è uguale a j)

PROCEDURA DI PARTIZIONAMENTO

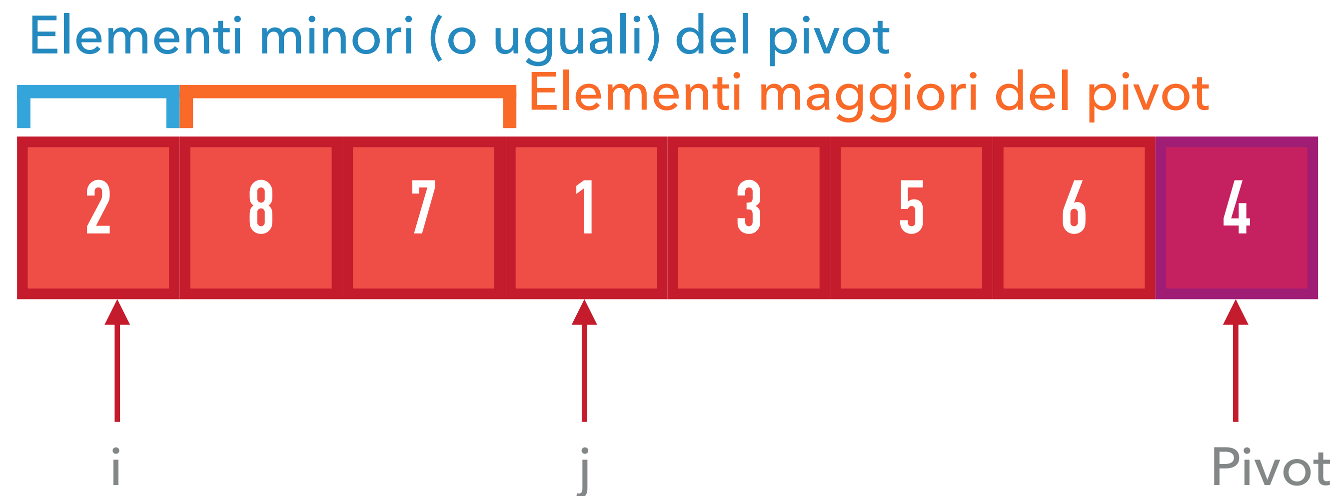


Se A[j] è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO

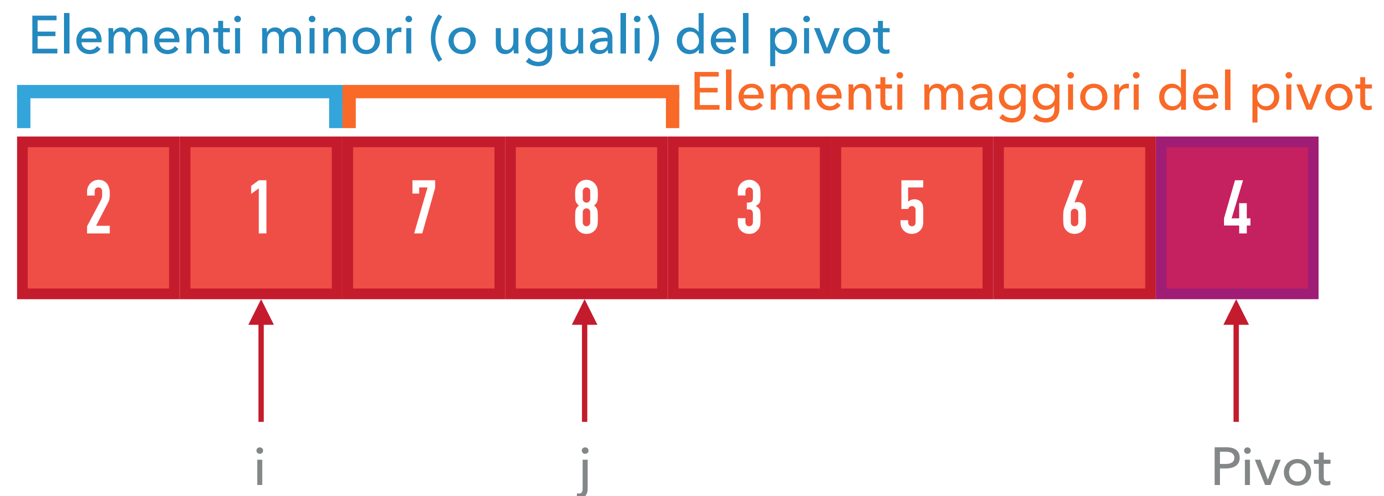
Se A[j] è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



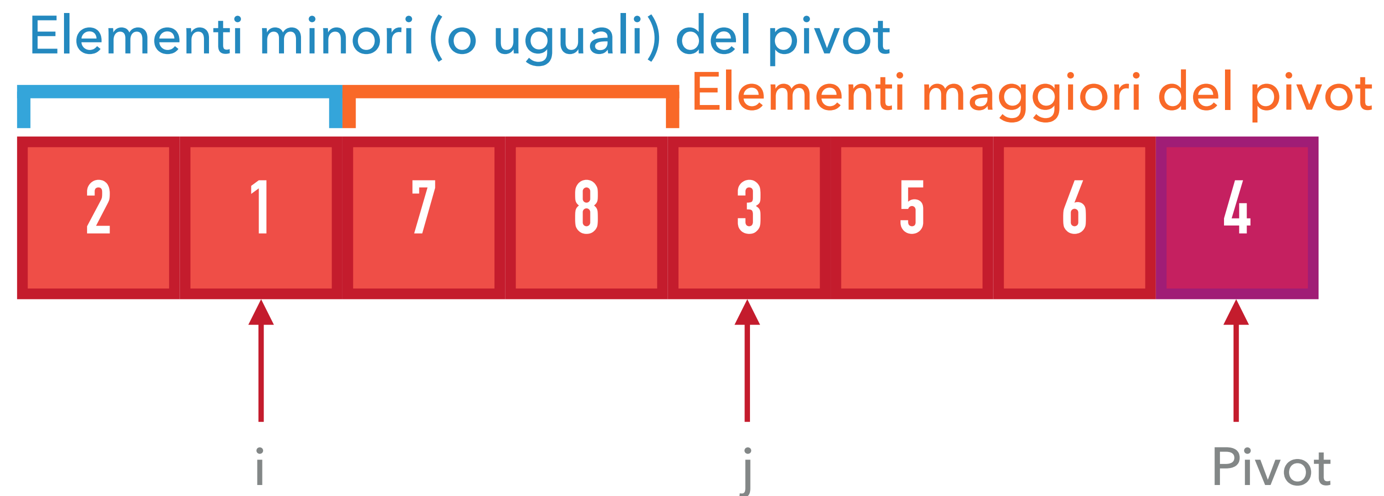
Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

PROCEDURA DI PARTIZIONAMENTO



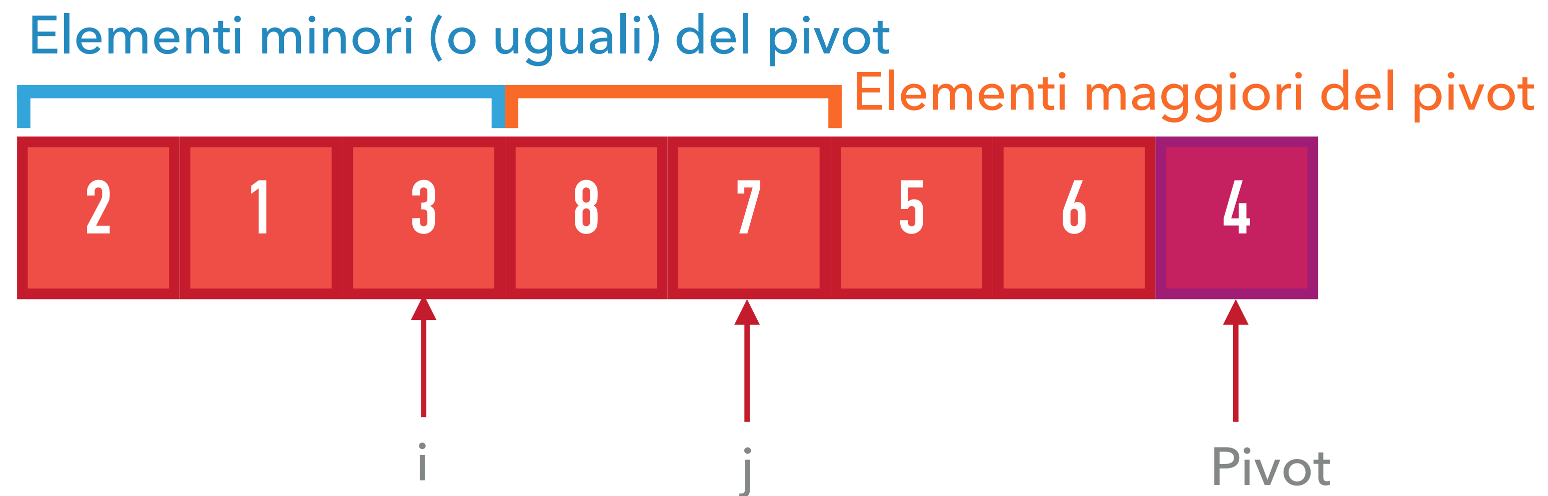
Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

PROCEDURA DI PARTIZIONAMENTO



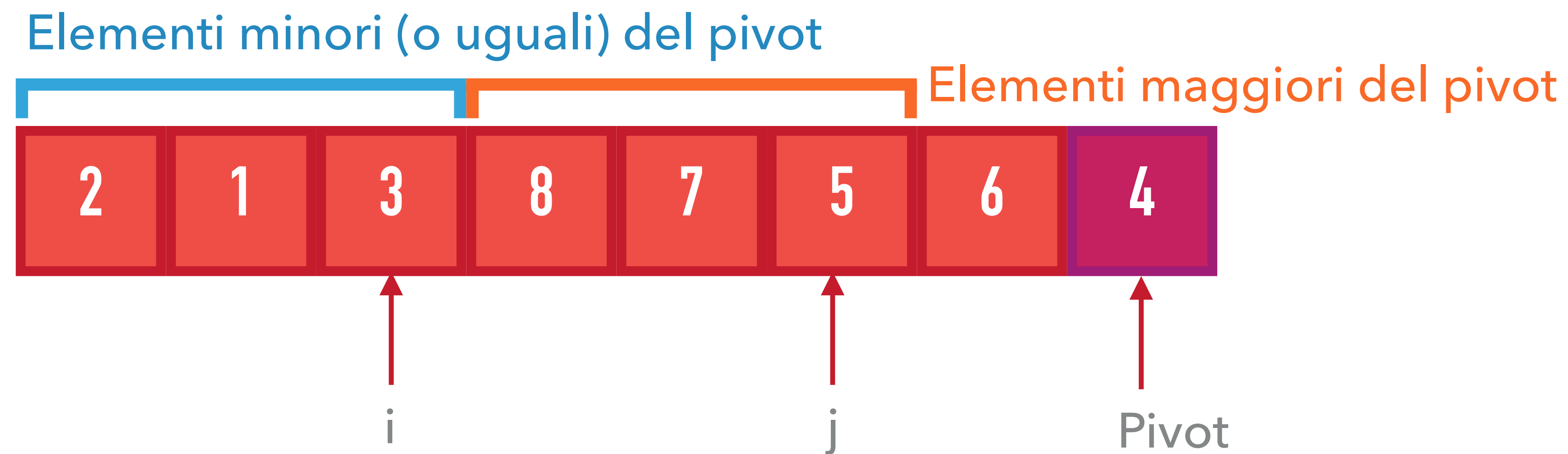
Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

PROCEDURA DI PARTIZIONAMENTO



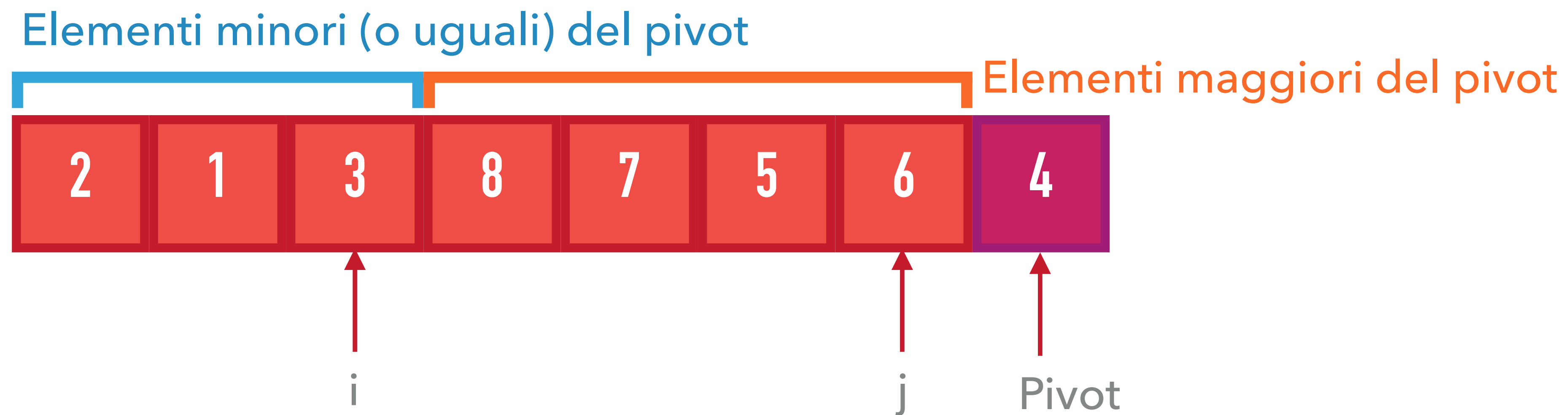
Se A[j] è minore del pivot
incrementiamo i e scambiamo A[i] e A[j]

PROCEDURA DI PARTIZIONAMENTO



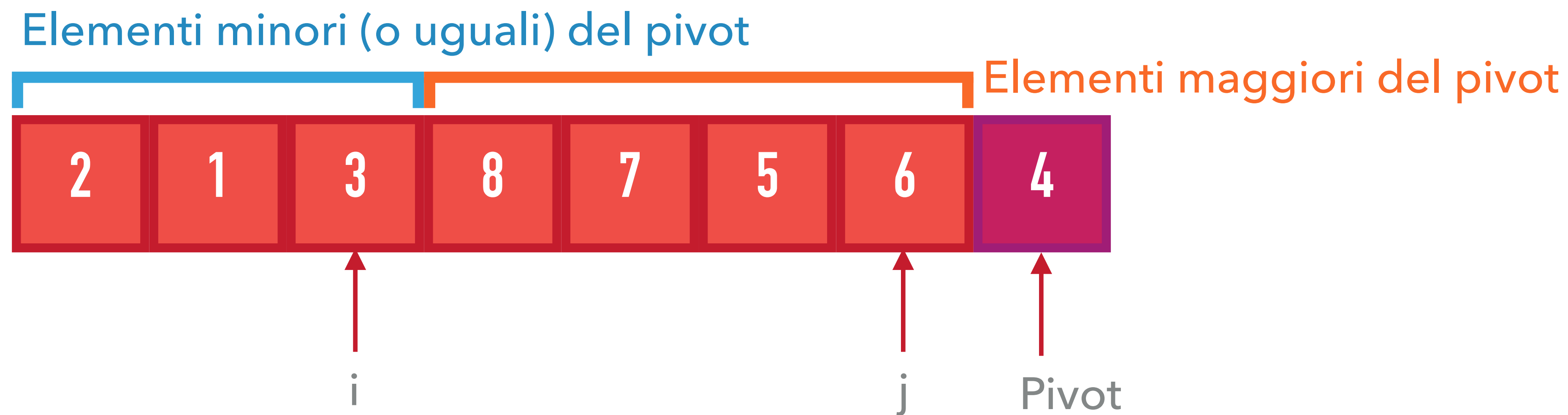
Se A[j] è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



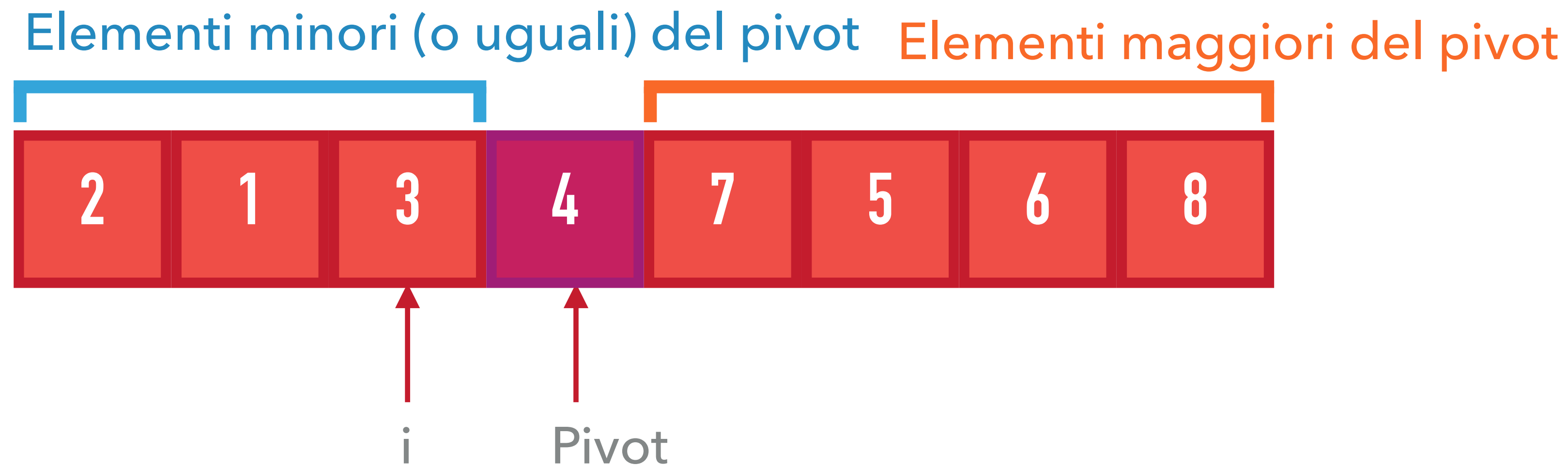
Se A[j] è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



Abbiamo effettuato una scansione di tutto l'array e abbiamo raccolto tutti gli elementi minori o uguali del pivot negli indici $[0,i]$.
Dobbiamo solo posizionale il pivot tra le due partizioni

PROCEDURA DI PARTIZIONAMENTO



Ci basta scambiare l'elemento in posizione $i + 1$ (che è necessariamente maggiore del pivot o il pivot stesso) con il pivot

PARTIZIONAMENTO: PSEUDOCODICE

```
Parametri: A (un array)
x = A[n−1] # il pivot è l’ultimo elemento dell’array
i = −1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n−1) # per tutti gli elementi dell’array tranne l’ultimo
  if A[j] ≤ x # se troviamo un elemento minore del pivot…
     i = i + 1
     scambia A[i] e A[j] # …lo spostiamo nella prima parte dell’array
Scambia A[i+1] con A[n−1] # mettiamo il pivot nella sua posizione finale
return i+1
```

Ma a noi servirà partizionare segmenti arbitrari di un array, quindi possiamo usare un’altra versione della procedura di partizionamento che lo applica solo tra due indici

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

```
Parametri: A (un array), p, r (indice di inizio e fine)
x = A[r] # il pivot è l’ultimo elemento dell’array (tra gli indici p e r)
i = p−1 # posizione iniziale degli elementi minori del pivot
for j in range(p,r) # per tutti gli elementi dell’array tranne il pivot
if A[j] ≤ x # se troviamo un elemento minore del pivot…
    i = i + 1
    scambia A[i] e A[j] # …lo spostiamo nella prima parte dell’array
Scambia A[i+1] con A[r] # mettiamo il pivot nella sua posizione finale
return i+1
```

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- Invariante (condizione che rimane vera ad ogni ciclo):
  - Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
  - Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot
- Ogni iterazione continua a far rispettare questa condizione:
  - Se il nuovo elemento è maggiore del pivot viene mantenuto nella sua posizione, rispettando quindi la condizione
  - Se il nuovo elemento è minore o uguale del pivot, i viene incrementato l'elemento in posizione j viene scambiato con quello in posizione i (che, dato che i è stato incrementato, è maggiore del pivot)

PARTIZIONAMENTO: COMPLESSITÀ

- Il partizionamento viene fatto con una singola “passata” dell’array (il ciclo for esterno)
- Tutte le operazioni all’interno e all’esterno del ciclo for hanno un costo costante
- Ne segue che il partizionamento viene fatto in tempo $\Theta(n)$

QUICKSORT: PSEUDOCODICE

```
Parametri: A (un array), p, r (indice di inizio e fine)
if p ≥ r:
   return
q = partiziona(A, p, r) # indice del pivot dopo il partizionamento
quicksort(A, p, q−1) # chiamata ricorsiva sugli elementi minori o uguali
quicksort(A, q+1, r) # chiamata ricorsiva sugli elementi maggiori
```

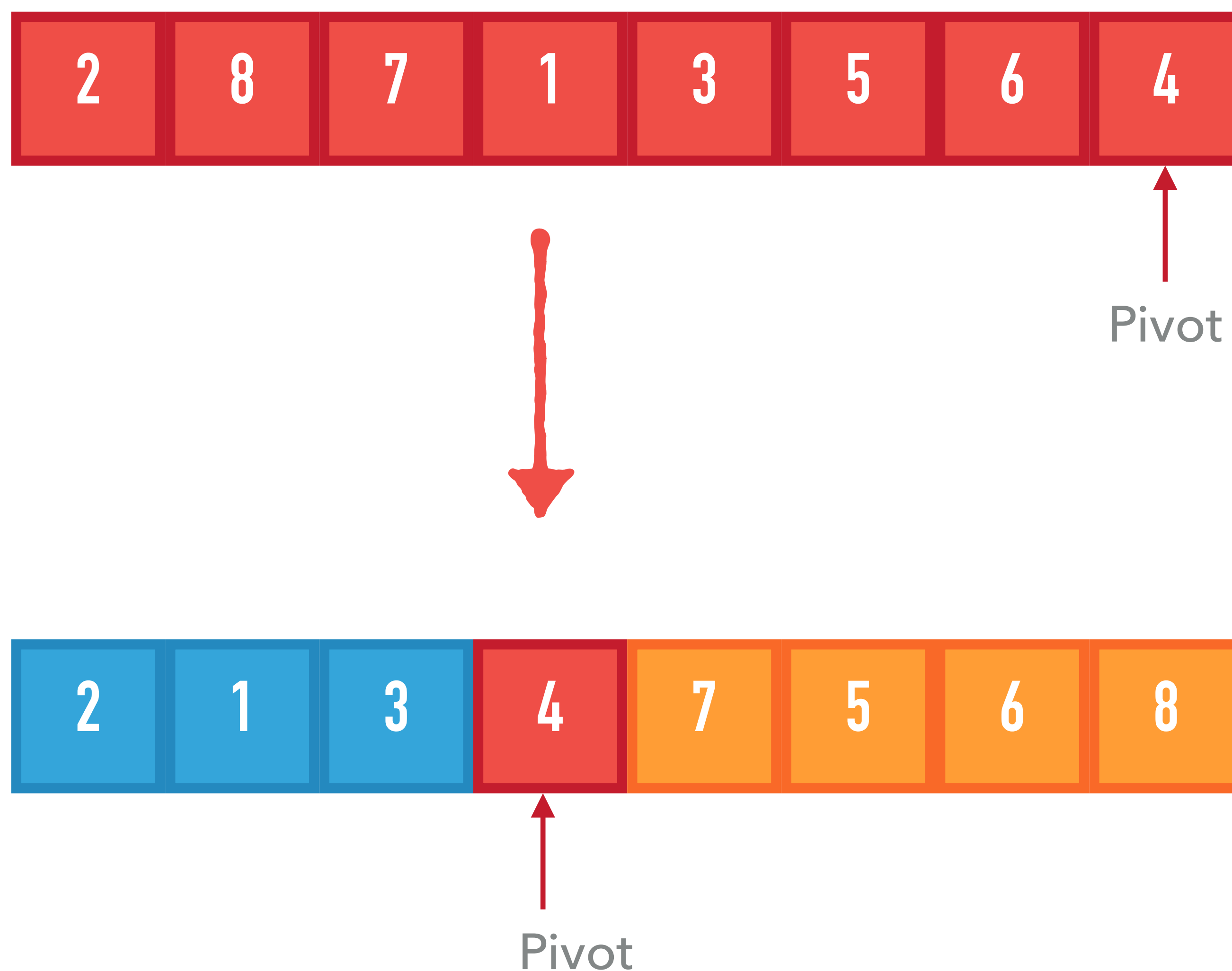
QUICKSORT: PERCHÉ FUNZIONA?

- Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali
- Supponiamo di avere provato che il quicksort funziona per ogni dimensione minore di n, vediamo che funziona per la dimensione n
- Dopo la procedura di partizionamento il pivot è nella posizione corretta e otteniamo due sotto-array uno che precedete il pivot con tutti gli elementi minori o uguali al pivot e uno con tutti gli elementi maggiori
- Richiamiamo quicksort sui due sotto-array di dimensione minore di n, che per ipotesi ci restituiranno gli array ordinati
- Otteniamo tutti gli elementi minori o uguali al pivot ordinati, seguiti dal pivot e da tutti gli elementi maggiori del pivot ordinati. Quindi l'array di n elementi è ordinato.

ANALISI DELLA COMPLESSITÀ

- L’analisi della complessità del quicksort è più delicata di mergesort e heapsort
- La dimensione degli array nelle chiamate ricorsive dipende dalla procedura di partizionamento
- La procedura di partizionamento dipende, a sua volta dai dati che abbiamo

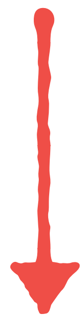
POSSIBILI PARTIZIONAMENTI

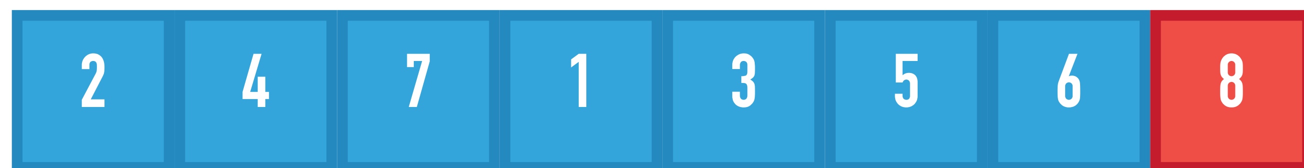


Un "buon" partizionamento:
i due sottoarray risultanti sono ognuno circa la metà dell'array di partenza

POSSIBILI PARTIZIONAMENTI

Pivot





Pivot

Un “cattivo” partizionamento: uno dei sotto-array risultanti contiene tutti gli elementi tranne uno e l’altro è vuoto!

QUIZ

In **quali** dei seguenti casi il partizionamento è maggiormente sbilanciato?

1) [1, 2, 3, 4, 5]

2) [2, 6, 5, 3, 4]

3) [1, 2, 5, 4, 3]

4) [5, 4, 3, 2, 1]

“BUON PARTIZIONAMENTO”: ANALISI

- Assumiamo che ogni partizionamento crei due sottoarray di dimensioni approssimativamente $n/2$
- L’equazione di ricorrenza diventa quindi:
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
- Per il teorema dell’esperto il tempo di calcolo del quicksort è $\Theta(n \log n)$
- Ma questo risultato vale solo per un “buon” partizionamento

"CATTIVO PARTIZIONAMENTO": ANALISI

- Assumiamo che ogni partizionamento crei un sottoarray vuoto ed uno di dimensione $n - 1$
- L'equazione di ricorrenza diventa quindi: $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$
- Se espandiamo $T(n)$ vediamo che abbiamo n "passi ricorsivi", ognuno dei quali esegue un lavoro lineare rispetto alla dimensione dell'array da ordinare
- Come risultato otteniamo $\Theta(n^2)$

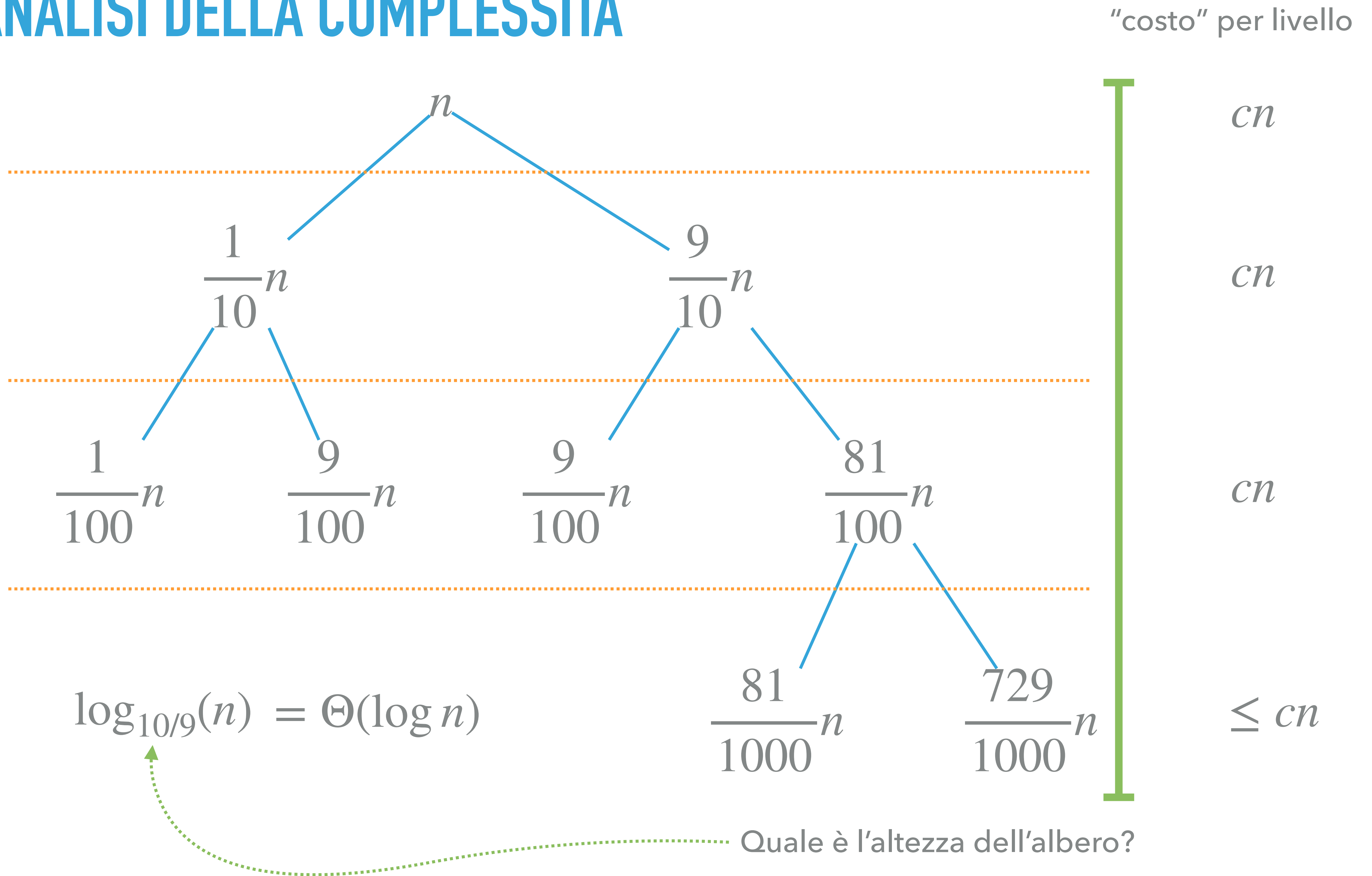
ANALISI DELLA COMPLESSITÀ

- Nel caso peggiore quindi otteniamo $\Theta(n^2)$
- Ma quanto è frequente il caso peggiore?
- Supponiamo che le nostre partizioni siano molto sbilanciate: la prima metà include $1/10$ dell'array e la seconda ne include $9/10$
- Lo stesso ragionamento vale anche per $1/100$ e $99/100$, etc.

ANALISI DELLA COMPLESSITÀ

- $$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$
- Esplicitiamo la costante nascosta c nell'$\Theta(n)$:
  $$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + cn$$
- Costruiamo l'albero delle chiamate ricorsive

ANALISI DELLA COMPLESSITÀ



ANALISI DELLA COMPLESSITÀ

- Stiamo eseguendo al più cn passi per ognuno degli $\Theta(\log n)$ livelli dell'albero
- Di conseguenza il tempo di esecuzione è ancora $\Theta(n \log n)$
- Cosa succede nel "caso medio"?
- È possibile provare che, se la scelta del pivot viene effettuata in modo casuale, il **tempo atteso** di esecuzione è $\Theta(n \log n)$

QUICKSORT: POSSIBILI VARIANTI

- Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore
- Invece di scegliere l'ultimo elemento come pivot, viene scelto un elemento a caso che viene spostato in ultima posizione
- Vengono presi tre elementi e la mediana dei tre viene usata come pivot
- I libri "algorithms in C", "algorithms in Java" di Robert Sedgewick trattano molti di questi miglioramenti

MODIFICARE IL QUICKSORT

- Esiste una semplice modifica del quicksort che ci permette di ottenere la **mediana** di un array
- Se il pivot si trova nella posizione centrale dell'array allora è la mediana
- Ci basta fare ricorsione su un lato solo dell'array (quello che contiene la posizione centrale) finché non troviamo il pivot al centro
- Il costo computazionale medio in questo caso è $\Theta(n)$