Algoritmi e Strutture di Dati

Quick-sort

m.patrignani

Nota di copyright

Quick-sort

- Algoritmo di ordinamento in loco ma non stabile
- Tempo di esecuzione
  - nel caso peggiore $\Theta(n^2)$
  - nel caso migliore e medio $\Theta(n \log n)$
    - i fattori costanti nascosti nella notazione Θ sono abbastanza piccoli
- Introdotto da Hoare nel 1962
  - la versione che vedremo è una variante dovuta a Lomuto
- Basato sul paradigma divide et impera

Divide et impera nel quick-sort

Per ordinare un sottoarray A[p...r]

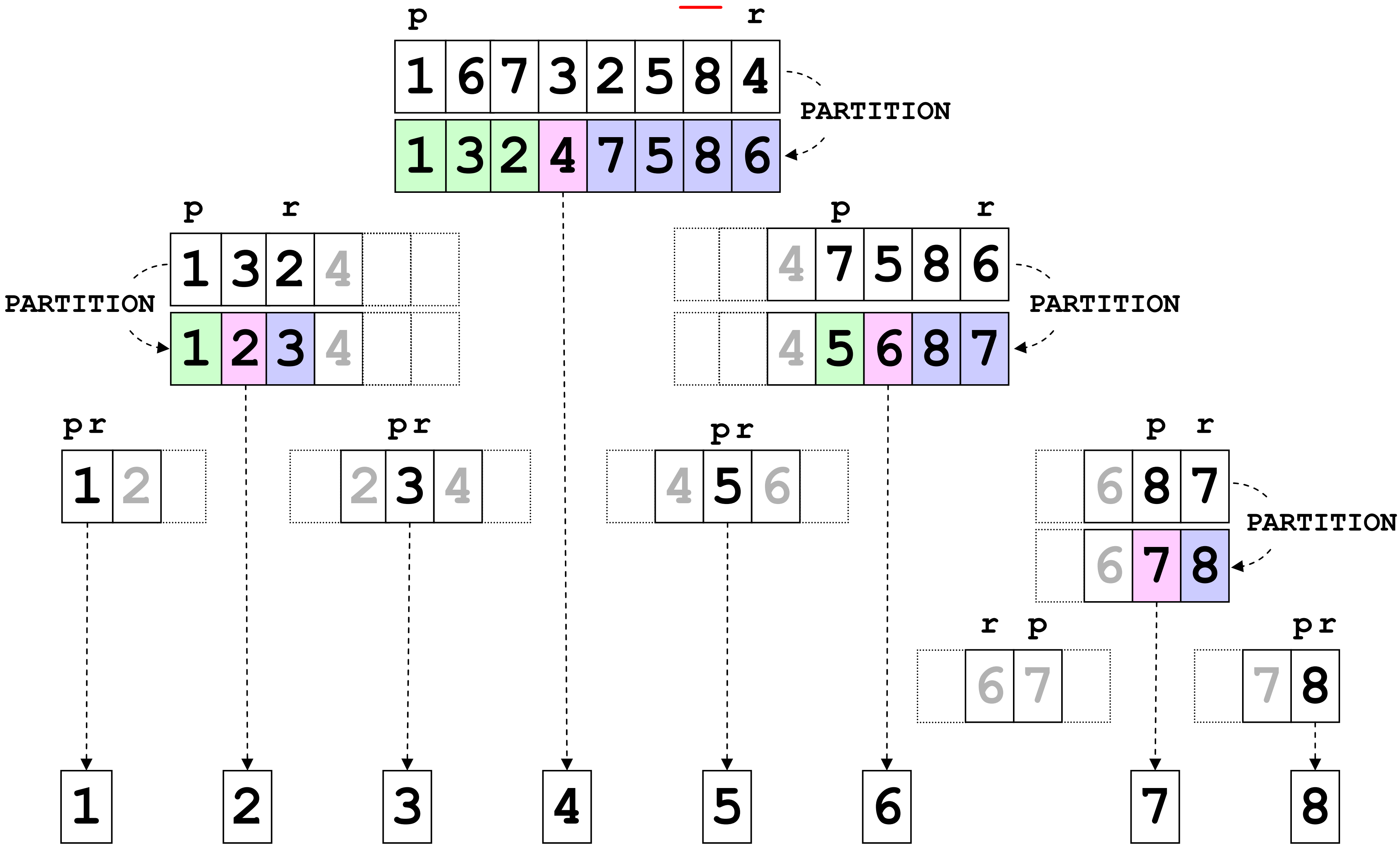
- Divide
  - A[p...r] viene ripartito (e risistemato) in due sottoarray non vuoti A[p...q–1] e A[q+1...r], in modo che ogni elemento del primo sia minore o uguale ad A[q] e ogni elemento del secondo sia maggiore ad A[q]
  - l’indice q viene calcolato dalla procedura di partizionamento
- Impera
  - i due sottoarray A[p...q–1] e A[q+1...r] sono ordinati, ricorsivamente
- Combina
  - non c’è niente da fare: A[p...r] è ordinato

Procedura QUICK_SORT

```
QUICK_SORT(A,p,r)
if p < r
      q = PARTITION(A,p,r)
      QUICK_SORT(A,p,q-1)
      QUICK_SORT(A,q+1,r)
```

- La procedura QUICK_SORT ordina in loco l'intervallo A[$p..r$]
  - se $p = r$, allora l'intervallo contiene una sola casella ed è già ordinato: l'invocazione di QUICK_SORT non ha effetto
  - se $p > r$, allora l'intervallo è un intervallo degenere e l'invocazione di QUICK_SORT non ha effetto
- Il valore q ritornato da PARTITION è tale che $p \leq q \leq r$
- Per ordinare l'intero array viene invocata la procedura:
  QUICK_SORT(A,0,A.length-1)

Esecuzione di QUICK_SORT su 1 6 7 3 2 5 8 4

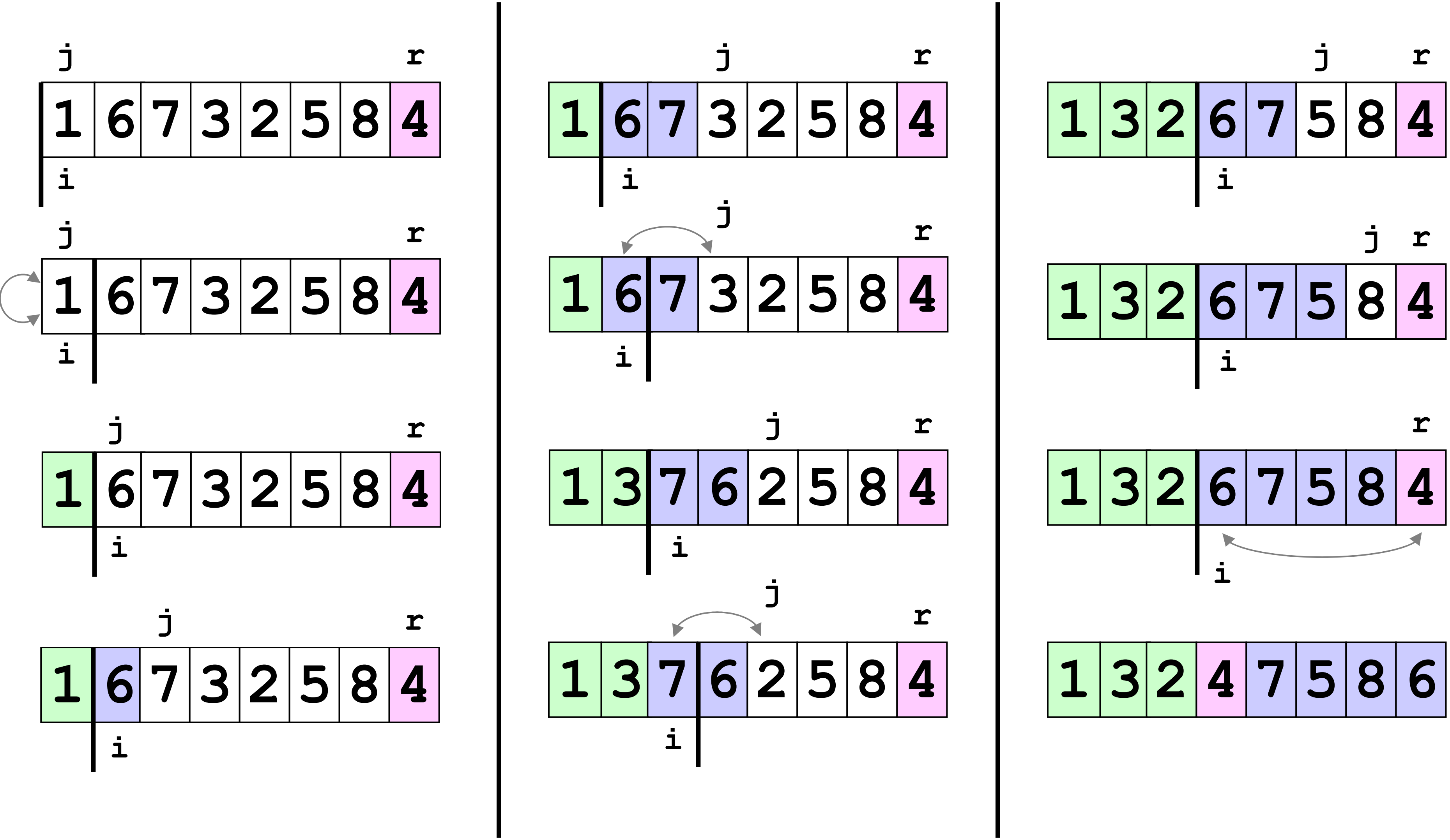


Procedura PARTITION

```
PARTITION(A,p,r)      /* si assume p < r */
  i = p        /* i è il primo elemento > A[r] = pivot */
     for j = p to r – 1   /* scorro l’array (non il pivot)*/
          if A[j] ≤ A[r]    /* A[r] è il pivot */
              SCAMBIA(A,i,j)
              i = i + 1
  SCAMBIA(A,i,r)        /* metto il pivot al centro */
  return i        /* ritorno la posizione del pivot */
```

- La procedura PARTITION viene invocata su un intervallo di almeno due elementi ($p < r$)
  - due casi base
    - i due elementi sono ordinati
    - i due elementi non sono ordinati

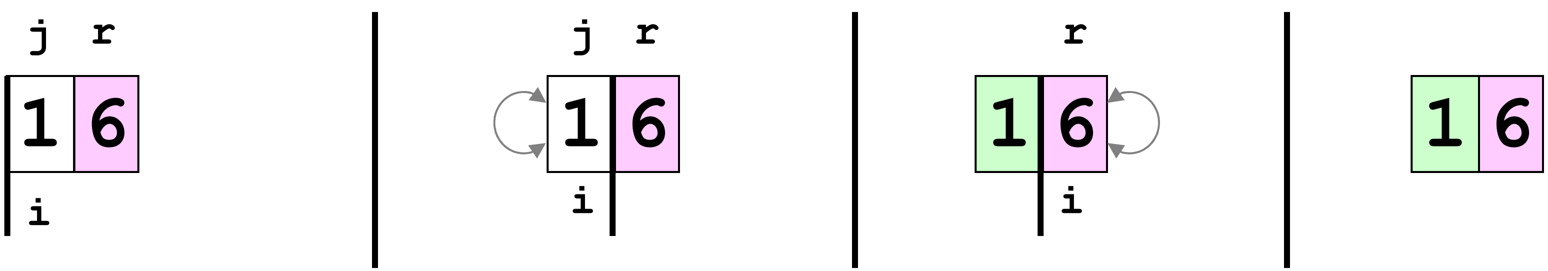
Esecuzione di `PARTITION` su | 1 | 6 | 7 | 3 | 2 | 5 | 8 | 4 |



Esecuzione di `PARTITION` su 1 6

- Caso base 1: `PARTITION` su una coppia ordinata

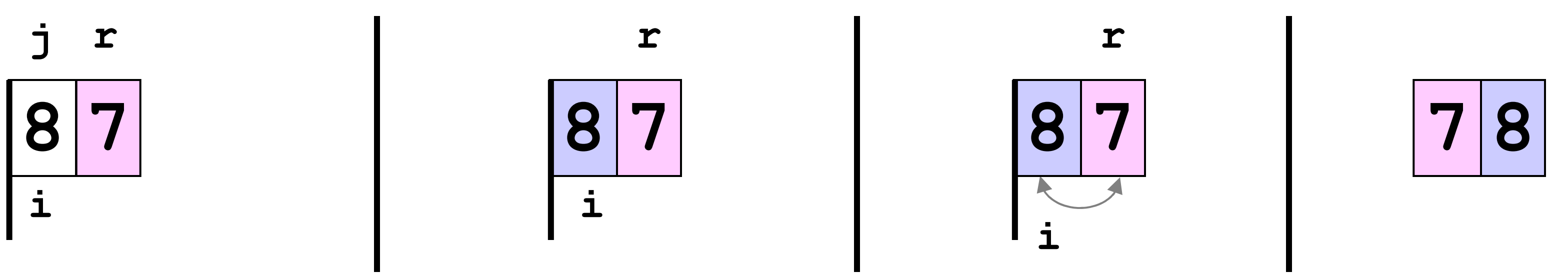
```
PARTITION(A,p,r)     /* si assume p < r */
   i = p        /* i è il primo elemento > A[r] = pivot */
      for j = p to r – 1  /* scorro l’array (non il pivot)*/
            if A[j] ≤ A[r]    /* A[r] è il pivot */
                  SCAMBIA(A,i,j)
                  i = i + 1
   SCAMBIA(A,i,r)        /* metto il pivot al centro */
   return i        /* ritorno la posizione del pivot */
```



Esecuzione di PARTITION su 8 7

- Caso base 2: PARTITION su una coppia non ordinata

```
PARTITION(A,p,r)     /* si assume p < r */
i = p        /* i è il primo elemento > A[r] = pivot */
   for j = p to r – 1   /* scorro l’array (non il pivot)*/
        if A[j] ≤ A[r]    /* A[r] è il pivot */
            SCAMBIA(A,i,j)
            i = i + 1
SCAMBIA(A,i,r)       /* metto il pivot al centro */
return i        /* ritorno la posizione del pivot */
```



Tempo di esecuzione di PARTITION

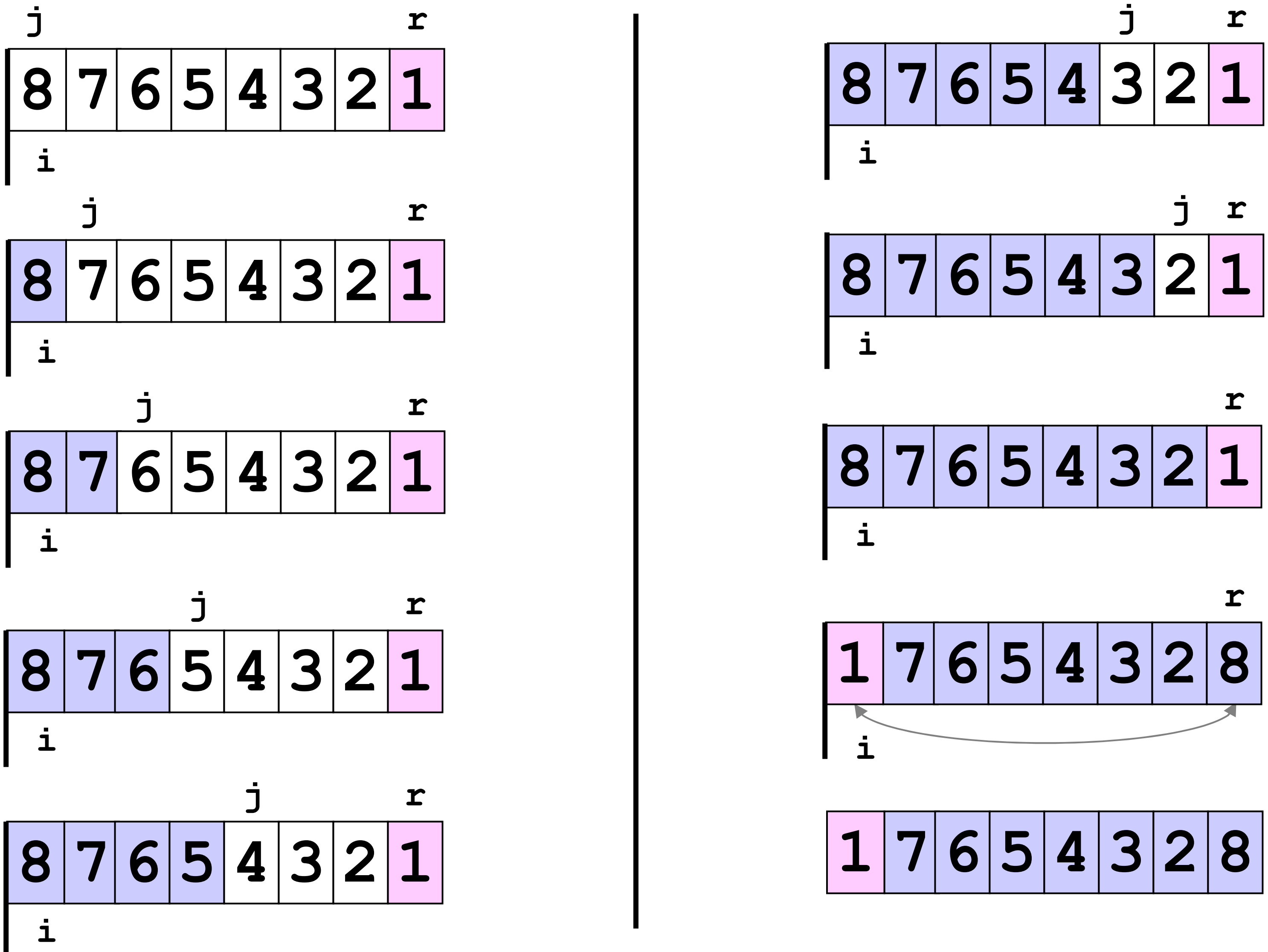
```
PARTITION(A,p,r)      /* si assume p < r */
1.  i = p        /* i è il primo elemento > A[r] = pivot */
2.     for j = p to r – 1  /* scorro l'array (non il pivot)*/
3.          if A[j] ≤ A[r]    /* A[r] è il pivot */
4.              SCAMBIA(A,i,j)
5.              i = i + 1
6.  SCAMBIA(A,i,r)        /* metto il pivot al centro */
7.  return i        /* ritorno la posizione del pivot */
```

- Le assegnazioni iniziali e finali richiedono tempo costante
- Nel caso peggiore, come nel caso migliore, il sottoarray A[p...r] viene scorso per intero da sinistra verso destra
- Il tempo di esecuzione $T_{\mathtt{PARTITION}}(n) \in \Theta(n)$

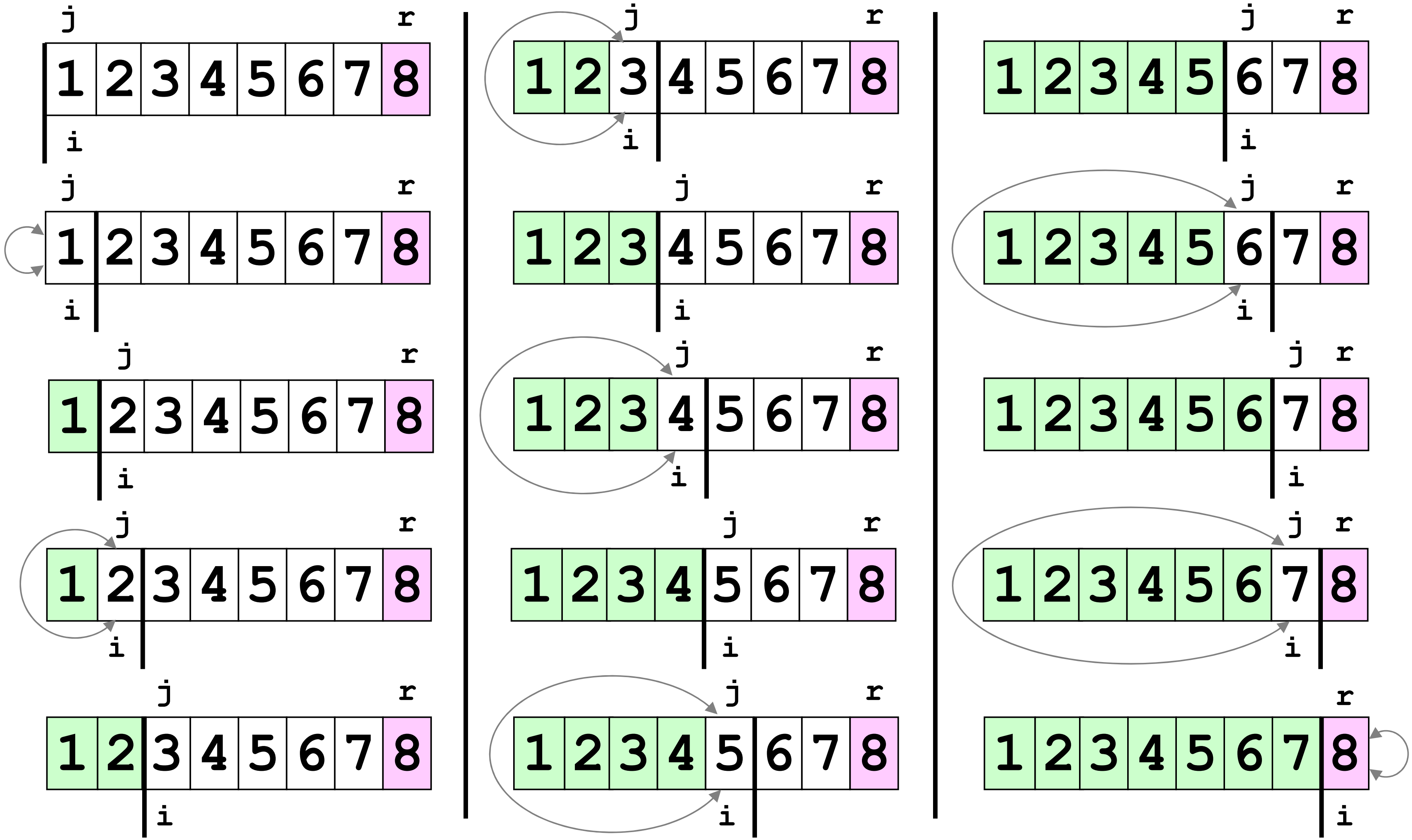
Esercizi

1. Che cosa succederebbe nel `QUICK_SORT` se `PARTITION`(A,p,r) restituisse un valore q uguale a r?
2. Illustrare le operazioni di `PARTITION` sull'array A = <13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21>
3. Illustrare le operazioni di `PARTITION` su un array
   - già ordinato in senso decrescente
   - già ordinato in senso crescente
4. Quale valore restituisce `PARTITION` se tutti gli elementi dell'array A[p...r] hanno lo stesso valore?

Esecuzione di `PARTITION` su 8 7 6 5 4 3 2 1



Esecuzione di `PARTITION` su | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |



Caso peggiore e migliore per `QUICK_SORT`

- Il caso peggiore per `QUICK_SORT` è quando `PARTITION` elegge a *pivot* il valore massimo o minimo dell'array
  - in questo caso `QUICK_SORT` non ricorre su due sottoarray bilanciati, ma ricorre su un sottoarray più corto di una casella ed un sottoarray degenere
- Il caso migliore per `QUICK_SORT` è invece quando `PARTITION` elegge a *pivot* il valore mediano dell'array
  - in questo caso `QUICK_SORT` ricorre su due sottoarray bilanciati

Analisi del caso migliore per `QUICK_SORT`

- Nel caso migliore il tempo di calcolo di `QUICK_SORT` su un array con n posizioni è

$$\mathrm{T}(n) = 2 \cdot \mathrm{T}(n/2) + \Theta(n)$$

- Questa equazione di ricorrenza può essere risolta con il teorema dell'esperto

$$\mathrm{T}(n) = a \cdot \mathrm{T}(n/b) + \mathrm{p}(n^k)$$

- Nello speciale caso in cui

$$a=2 \qquad b=2 \qquad k=1$$

- Che per $a = b^k$ si risolve in

$$\mathrm{T}(n) = \Theta(n^k \log n) = \Theta(n \log n)$$

Analisi del caso peggiore per QUICK_SORT

- Si ha

$$T(0) = a$$
$$T(n) = T(n\text{-}1) + \Theta(n)$$

- Sappiamo che la soluzione di questa equazione di riccorrenza è

$$T(n) = a + \sum_{k=1}^{n} g(k)$$

- E dunque

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

Analisi del caso medio per `QUICK_SORT`

- Si può dimostrare formalmente che nel caso medio `QUICK_SORT` ha una complessità $\Theta(n \log n)$
  - l'analisi, però, è molto più complessa del caso migliore e del caso peggiore
- Nel seguito vedremo solamente due considerazioni intuitive che ci aiutano a giustificare questo risultato
  1. qual è la complessità nel caso in cui lo sbilanciamento della ricorsione non supera mai una determinata soglia
  2. qual è la complessità nel caso in cui ricorsioni sbilanciate si alternano a ricorsioni più bilanciate

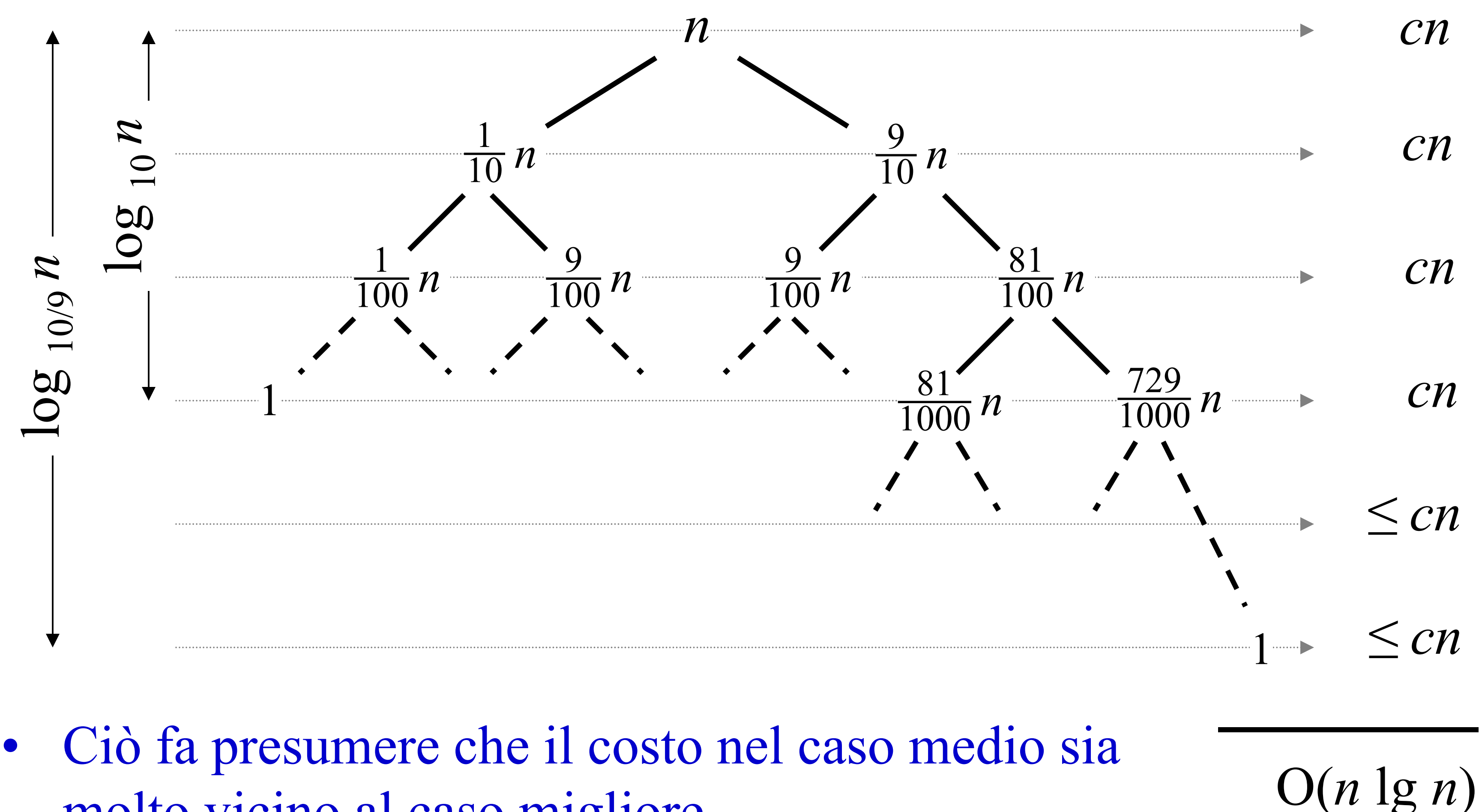
Caso bilanciato 9-a-1

- Supponiamo che `PARTITION` divida il sottoarray in due parti che hanno una proporzione fissa
  - supponiamo che la proporzione sia 9-a-1
- Abbiamo

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

dove cn esplicita $\Theta(n)$

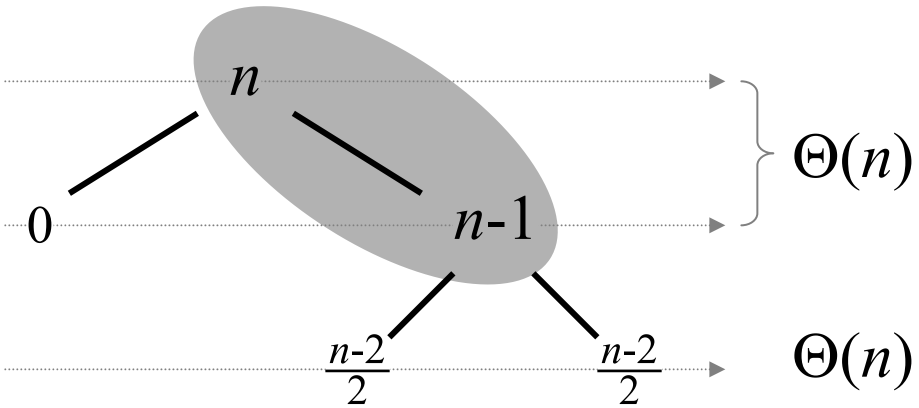
Ricorsione con proporzione 9-a-1

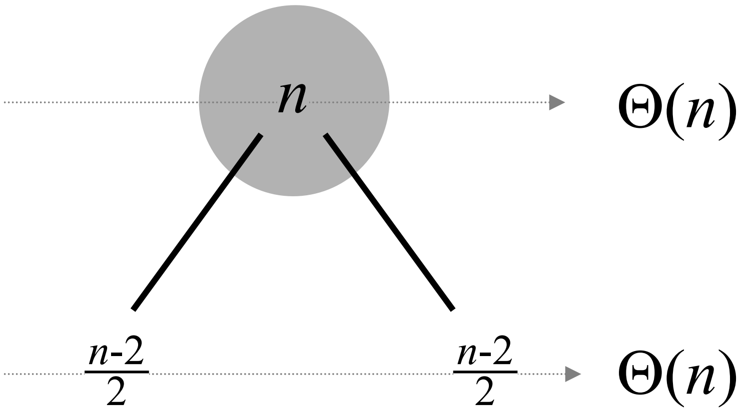


- Ciò fa presumere che il costo nel caso medio sia molto vicino al caso migliore

Alternanza di ricorsioni bilanciate e sbilanciate

- Supponiamo che nel 20% dei casi `PARTITION` produca una partizione meno bilanciata di 9-a-1
- Supponiamo che nell'albero delle chiamate ricorsive una ripartizione sbilanciata sia sempre seguita da una bilanciata
- Il costo di una ripartizione sbilanciata può essere assorbito dal costo della ripartizione bilanciata





Versione randomizzata di QUICK_SORT

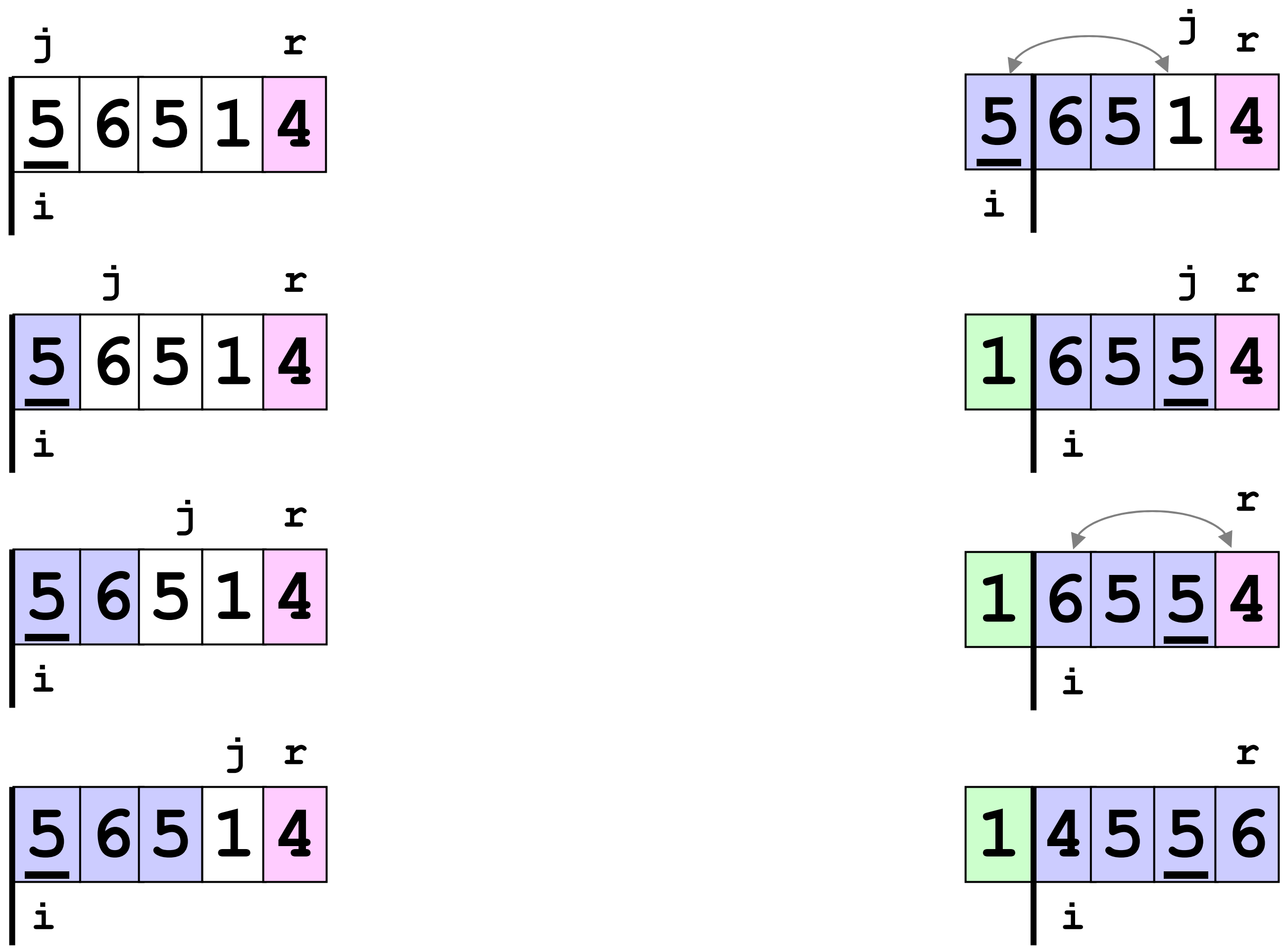
- E' possibile modificare QUICK_SORT in maniera che i casi peggiori non coincidano con disposizioni notevoli degli elementi

```
RANDOMIZED_PARTITION(A,p,r)
i = RANDOM(p,r)
SCAMBIA(A,r,i)
return PARTITION(A,p,r)
```

```
RANDOMIZED_QUICK_SORT(A,p,r)
if p < r then
    q = RANDOMIZED_PARTITION(A,p,r)
    RANDOMIZED_QUICK_SORT(A,p,q-1)
    RANDOMIZED_QUICK_SORT(A,q+1,r)
```


Stabilità di QUICK_SORT

- QUICK_SORT non è stabile:



Algoritmi di ordinamento per confronto

	caso migliore	caso medio	caso peggiore	in loco	stabile
SELECTION-SORT	$\Theta(n^2)$			*si*	*si*
INSERTION-SORT	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	*si*	*si*
MERGE-SORT	$\Theta(n \log n)$			*no*	*si*
HEAP-SORT	$\Theta(n \log n)$			*si*	*no*
QUICK-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	*si*	*no*

