Algoritmi di ordinamento

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2022/2023

Introduzione

- Problema dell'ordinamento
  - **Input**: una sequenza di n numeri $[a_1, a_2, \ldots, a_n]$
  - **Output**: una permutazione $p : \{1, .., n\} \to \{1, .., n\}$ degli indici

    $$[a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)}]$$

    tale che

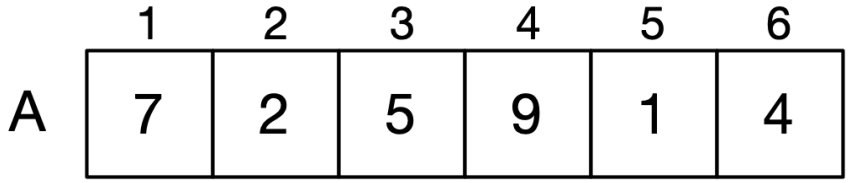
    $$a_{p(1)} \leq a_{p(2)} \leq \cdots \leq a_{p(n)}$$
- Esempio:
  - Input: $[7, 32, 88, 21, 92, -4]$, Output: $[-4, 7, 21, 32, 88, 92]$
- In questa lezione discutiamo alcuni algoritmi di ordinamento
  - Incrementali: SelectionSort, InsertionSort
  - Divide et Impera: MergeSort, QuickSort
  - Non-comparativi: CountingSort, RadixSort
- Discuteremo, inoltre, anche il limite inferiore all complessità del problema dell'ordinamento

Nozioni preliminari

- Assumiamo di avere in input un array di cui ogni elemento ha
  - una chiave, tutte le chiavi sono confrontabili rispetto a $<, =, >$
  - un valore, contenuto associato alla chiave
- Ordiniamo l'array rispetto alla chiave, non rispetto al valore
  - In molti casi il nostro array conterrà solo la chiave
- Definiamo due properità degli algoritmi di ordinamento
  - in place: l'algoritmo riordina gli elementi nell'array in input (non è richiesto nessun array aggiuntivo per il calcolo)
  - stabile: valori con la stessa chiave appaiono nell'array ordinato nello stesso ordine in cui appaiono nell'array in input
- Nota: algoritmi non stabili possono sempre trasformati in stabili
  - Usiamo come chiave la coppia $(key, index)$
  - *index* è la posizione del valore nell'array in input
  - *key* è la chiave primaria di ordinamento, *index* la secondaria
  - Esempio: $(10, 2) < (10, 6)$

La struttura dati array

- Array: struttura dati costituita da una sequenza di elementi omogenei accessibili tramite un indice (primo indice=1 nello pseudocodice)

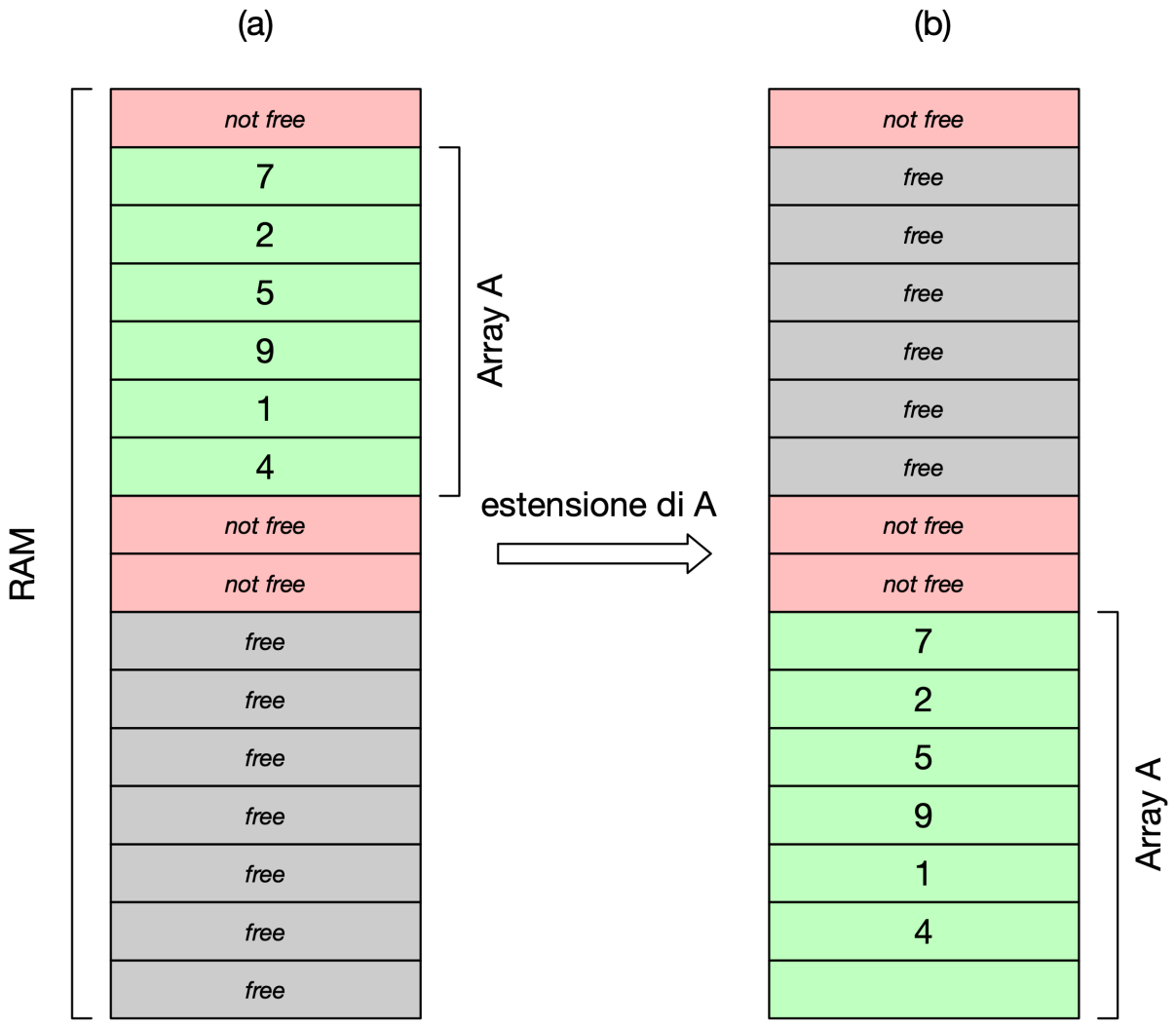


- E' possibile accedere agli elementi dell'array tramite il loro indice
  - Esempio: $A[1] = 7$, $A[5] = 1$
  - Il costo di accesso è constante per tutti gli elementi
- Usiamo la notazione $A[i, \cdots, j]$ per indicare il sotto-array di elementi

$$A[i], A[i+1], \cdots, A[j]$$

- Gli array sono una struttura dati a dimensione fissa
  - La dimensione è fissata quando l'array viene creato
  - Un array può essere ridimensionato (ampliato) solo creando un nuovo array della dimensione desiderata e copiando tutti gli elementi dal vecchio al nuovo array

Ridimensionamento di un array



(a) Non c'è spazio libero contiguo in RAM per estendere A

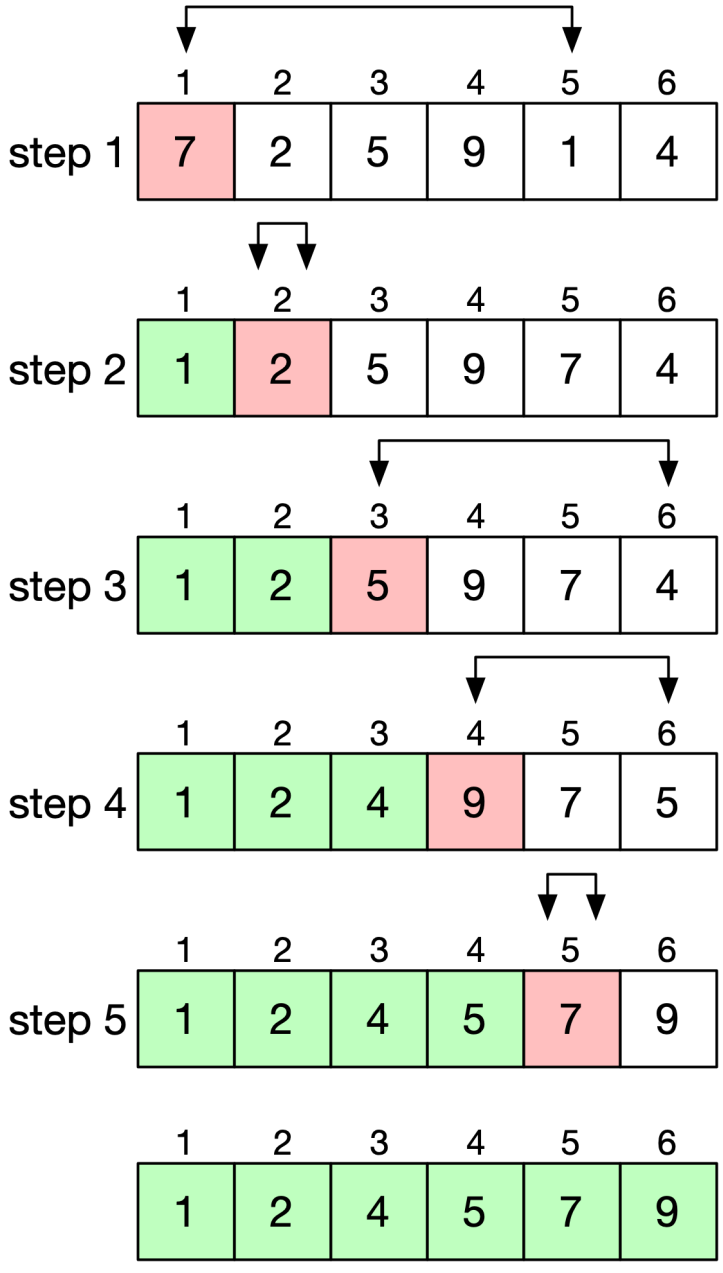
(b) Estendiamo A spostandolo in una nuova posizione in RAM

Algoritmi di ordinamento incrementali

- Algoritmi semplici che costruiscono l'ordinamento un elemento alla volta
- Idea: partendo da un prefisso ordinato $A[1, \cdots, k]$ dell'intero array estendiamo l'ordinamento ad un prefisso più ampio $A[1, \cdots, k+1]$
- SelectionSort
  - Cerca il minimo in $A[k+1, \cdots, n]$ e lo sposta in posizione $k+1$
- InsertionSort
  - Inserisce l'elemento $A[k+1]$ nella corretta posizione in $A[1, \cdots, k+1]$
- Entrambi gli algoritmi sono in place e stabili

SelectionSort

- Ad ogni passo $i = 1, \cdots, n-1$
  1. cerca la posizione j della minima chiave in $A[i, \cdots, n]$
  2. scambia (swap) $A[j]$ con $A[i]$



SelectionSort: pseudocodice

```
 1: function SELECTIONSORT(ARRAY A[1, ··· , n])
 2:     for i = 1, ··· , n − 1 do
 3:         ▷ Search min in A[i, ··· , n]
 4:         m = i
 5:         for j = i + 1, ··· , n do
 6:             if A[j] < A[m] then
 7:                 m = j
 8:         ▷ Swap A[m] with A[i]
 9:         if m ≠ i then
10:             SWAP(A, i, m)
11:
12: function SWAP(ARRAY A[1, ··· , n], INT i, INT j)
13:     tmp = A[i]
14:     A[i] = A[j]
15:     A[j] = tmp
```

- La funzione SWAP scambia due elementi nell'array

SelectionSort: implementazione in Java

```
1  public static void selectionSort(int A[]) {
2      for (int i = 0; i < A.length - 1; i++) {
3          int m = i;
4          for (int j = i + 1; j < A.length; j++)
5              if (A[j] < A[m])
6                  m = j;
7          if (m != i)
8              swap(A,i,m)
9      }
10 }
11
12 private static void swap(int A[],int i, int j) {
13     int tmp = A[i];
14     A[i] = A[j];
15     A[j] = tmp;
16 }
```

Da notare che a riga 2 facciamo partire l'indice da 0, non da 1 come nello pseudocodice

Complessità computazionale di SelectionSort

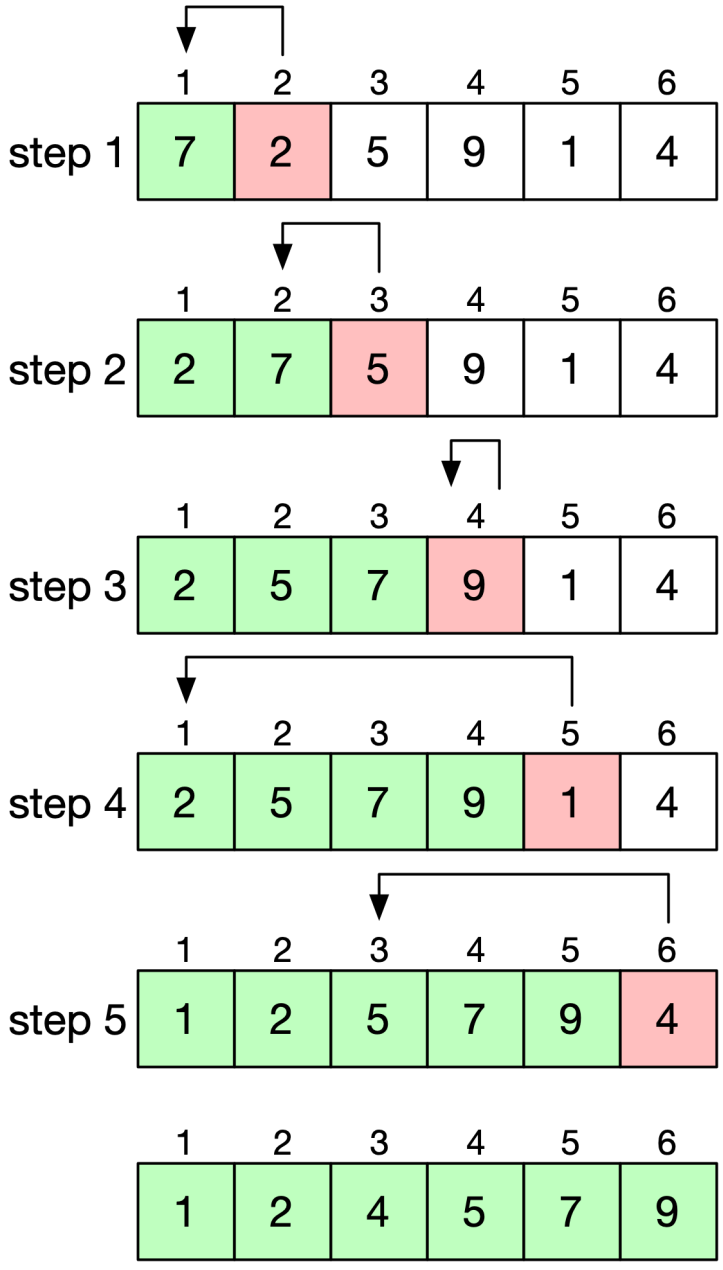
- Il ciclo a riga 2 viene eseguito $n-1$ volte. Al passo i
  - Costo ricerca del minimo (righe 4-7): $\Theta(n-i)$ ($n-i$ iterazioni)
  - Costo swap (righe 12-15): $O(1)$ (operazioni costanti)
- I due cicli sono eseguiti interamente ogni volta
  - Il caso ottimo, medio e pessimo coincidono
- Costo pessimo, medio e ottimo:

$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} n-i=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=\Theta(n^2)$$

- Complessità quadratica sulla lunghezza dell'array

InsertionSort (John Mauchly, 1946)

- Ad ogni passo $i = 2, \cdots, n$
  1. $A[1, \cdots, i-1]$ è ordinato
  2. inserisci $A[i]$ nella posizione corretta in $A[1, \cdots, i]$
- Intuitivamente, approccio utilizzato per ordinare carte da gioco



InsertionSort: pseudocodice

```
1: function INSERTIONSORT(ARRAY A[1, ··· , n])
2:     for i = 2, ··· , n do
3:         j = i
4:         while j > 1 and A[j] < A[j − 1] do
5:             SWAP(A, j, j − 1)
6:             j = j − 1
```

- Notiamo che:
  - il ciclo for a riga 2 viene eseguito sempre interamente
  - il ciclo while a riga 4 potrebbe non essere eseguito

InsertionSort: implementazione in Java

```
1 public static void insertionSort(int A[]) {
2     for (int i = 1; i <= A.length - 1; i++) {
3         int j = i;
4         while(j > 0 && A[j] < A[j-1]) {
5             swap(A,j,j-1);
6             j--;
7         }
8 }
```

- Anche in questo caso notiamo lo *shift* del primo indice di A da 1 a 0
- La funzione swap è la stessa utilizzata per selectionsort

Complessità computazionale di InsertionSort

- Costo nel caso pessimo
  - Le chiavi nell'array sono ordinate dalla più grande alla più piccola
  - Il ciclo for a riga 2 viene eseguito $n-1$ volte
  - Per ogni iterazione i del ciclo for, il ciclo while a riga 4 viene eseguito $i-1$ volte

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

- Costo nel caso ottimo
  - Le chiavi nell'array sono ordinate dalla più piccola alla più grande
  - Il ciclo for a riga 2 viene eseguito $n-1$ volte
  - Il ciclo while a riga 4 non viene mai eseguito $(\forall j, A[j] \geq A[j-1])$

$$\sum_{i=2}^{n} i = n - 1 = \Theta(n)$$

- Complessità quadratica nel caso pessimo e lineare nel caso ottimo

Complessità di InsertionSort nel caso ottimo

- Il caso ottimo si verifica anche per array quasi ordinati
- Esempio: A già ordinato tranne che per le k chiavi più piccole
  - Il ciclo while (riga 5) sarà eseguito solo su tali k elementi
  - Al peggio, ogni volta il ciclo while verrà eseguito per n iterazioni
  - Il costo per ordinare tale array quasi ordinato è quindi

$$\Theta(n) \text{ (ciclo for)} + O(nk) \text{ (ciclo while)} = O(nk)$$

- Quando il numero k di elementi non ordinati è costante rispetto ad n, cioè $k = O(1)$, il ciclo while ha costo $O(n)$ e quindi InsertionSort ha un costo complessivo pari a $\Theta(n)$ (costo lineare)
- Questa proprietà vale anche per altri casi di array quasi ordinati
  - Ad esempio, vale per array ordinati tranne che per $k = O(1)$ chiavi
  - Più complesso da analizzare rispetto al caso "k chiavi più piccole"
- Array quasi ordinati sono abbastanza comuni in alcune applicazioni reali

Complessità di InsertionSort nel caso medio

- Dobbiamo formulare qualche assunzione probabilistica per il caso medio
- Ricordiamo che:
  - Il ciclo for (riga 2) viene sempre eseguito interamente
  - Ad ogni iterazione i del ciclo for, il ciclo while viene eseguito al massimo per $i-1$ iterazioni (da 0 a $i-1$ iterazioni)
  - Possiamo solo fare assunzioni sulla lunghezza media del ciclo while
- Assunzione ragionevole: ciclo while eseguito in media per $(i-1)/2$ volte
- Sotto tale assunzione probabilistica abbiamo il costo totale

$$\sum_{i=2}^{n} \frac{i-1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{1}{2} \frac{n(n-1)}{2} = \Theta(n^2)$$

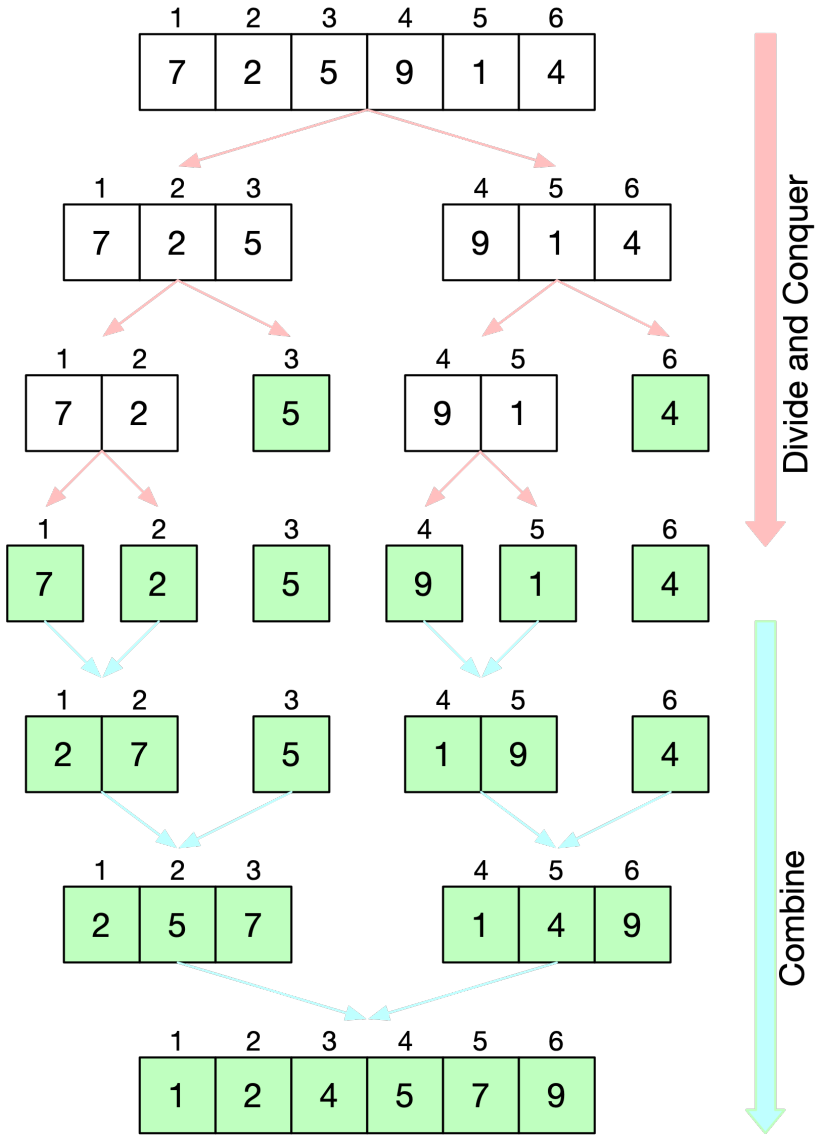
- Il costo medio di InsertionSort è quadratico, come nel caso pessimo

Algoritmi di ordinamento Divide et impera

- Divide et impera. Strategia per ottenere controllo militare e politico: fare in modo che gli avversari siano divisi e si combattano tra di loro
  - Denominata anche Divide and Conquer in inglese
- Divide et impera negli algoritmi:
  - Divide: dividere il problema in sotto-problemi più piccoli
  - Conquer: risolvere ("conquistare") i sotto-problemi ricorsivamente
  - Combine: fondere le soluzioni ai sotto-problemi in una più generale
- Vediamo due algoritmi di ordinamento nella classe divide et impera
  - MergeSort
  - QuickSort
- Tipicamente più efficienti degli algoritmi incrementali

MergeSort (John von Neumann, 1945)

1. **Divide**: dividiamo $A[1,\cdots,n]$ in due metà $A_1 = A[1,\cdots,q]$ e $A_2 = A[q+1,\cdots,n]$ dove $q = \lfloor (1+n)/2 \rfloor$
2. **Conquer**: richiamiamo ricorsivamente l'algoritmo su A_1 and A_2 se hanno lunghezza > 1 (un array di lunghezza 1 è già ordinato)
3. **Combine**: combiniamo i due array ordinati A_1, A_2 in un unico array ordinato



MergeSort: pseudocodice

```
1: function MERGESORT(ARRAY A[1, ··· , n], INT p, INT r)
2:     if p < r then
3:         q = ⌊(p + r)/2⌋
4:         MERGESORT(A, p, q)
5:         MERGESORT(A, q + 1, r)
6:         ▷ Here A[p, ··· , q] and A[q + 1, ··· , r] are sorted
7:         MERGE(A, p, q, r)
```

- Prima chiamata: $\text{MERGESORT}(A, 1, n)$
- E' semplice dividere, più complesso ricombinare
- La funzione MERGE combina gli array ordinati $A[p, \cdots, q]$, $A[q+1, \cdots, r]$
- Dopo MERGE il sotto-array $A[p, \cdots , r]$ è ordinato
- MERGE può essere implementata in modo che MERGESORT sia stabile
- Complesso implementare MERGE in modo che MERGESORT sia in-place

MergeSort: implementazione in Java

```
1 public static void mergesort(int A[], int p, int r) {
2     if (p < r) {
3           int q = p + (r-p) / 2;
4           mergesort(A,p,q);
5           mergesort(A,q+1,r);
6           merge(A,p,q,r);
7     }
8 }
```

- Nota: preferiamo usare $p + (r - p)/2$ invece di $(p + r)/2$ (linea 3)
  - Espressioni equivalenti: evita overflow di $p + r$ per p e r grandi
  - L'operatore / esegue una divisione tra interi

- Chiamata per eseguire mergesort su un array A:

  `mergesort(A,0,A.length-1)`

MergeSort: pseudocodice di merge

```
1:  function MERGE(ARRAY A[1, ··· , n], INT p, INT q, INT r)
2:      LET B[1, ··· , r − p + 1] BE A NEW ARRAY
3:      i = p                                   ▷ Index over A[p, ··· , q]
4:      j = q + 1                               ▷ Index over A[q + 1, ··· , r]
5:      k = 1                                   ▷ Index over B[1, ··· , r − p + 1]
6:      while i ≤ q and j ≤ r do
7:          if A[i] ≤ A[j] then ▷ Condition A[i] ≤ A[j] makes the algorithm stable
8:              B[k] = A[i]
9:              i = i + 1
10:         else
11:             B[k] = A[j]
12:             j = j + 1
13:         k = k + 1
14:     ▷ If i ≤ q append A[i, ··· , q] to B[1, ··· , k − 1]
15:     while i ≤ q do
16:         B[k] = A[i]
17:         k = k + 1, i = i + 1
18:     ▷ If j ≤ r append A[j, ··· , r] to B[1, ··· , k − 1]
19:     while j ≤ r do
20:         B[k] = A[j]
21:         k = k + 1, j = j + 1
22:     ▷ Copy the sorted numbers from B[1, ··· , r − p + 1] to A[p, ··· , r]
23:     for k = 1, ··· , r − p + 1 do
24:         A[p + k − 1] = B[k]
```

MergeSort: implementazione di merge in Java

```
1  private static void merge(int A[], int p, int q, int r) {
2    int[] B = new int[r - p + 1];
3    int   i = p;
4    int   j = q + 1;
5    int   k = 0;
6
7    while (i <= q && j <= r)
8      if (A[i] <= A[j])
9        B[k++] = A[i++];
10     else
11       B[k++] = A[j++];
12
13   while (i <= q)
14     B[k++] = A[i++];
15
16   while (j <= r)
17     B[k++] = A[j++];
18
19   for (k = 0; k < r-p+1; k++)
20     A[p+k] = B[k];
21 }
```

Notiamo lo shift degli indici su B a riga 5 e 19 (partiamo da 0, non da 1)

Complessità computazionale di MergeSort

- La procedura MERGE ha un costo lineare sulla lunghezza $r - p + 1$
  - In ognuno dei tre cicli while incrementiamo i oppure j, mai insieme
  - i viene incrementato da 1 a q, j da $q + 1$ a $r \Rightarrow r - p + 1$ iterazioni
  - Il ciclo for viene eseguito per $r - p + 1$ volte

- L'equazione di ricorrenza di MERGESORT è quindi

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + f(n) & n > 1 \end{cases}$$

  dove $f(n) = \Theta(n)$ è il costo della procedura MERGE

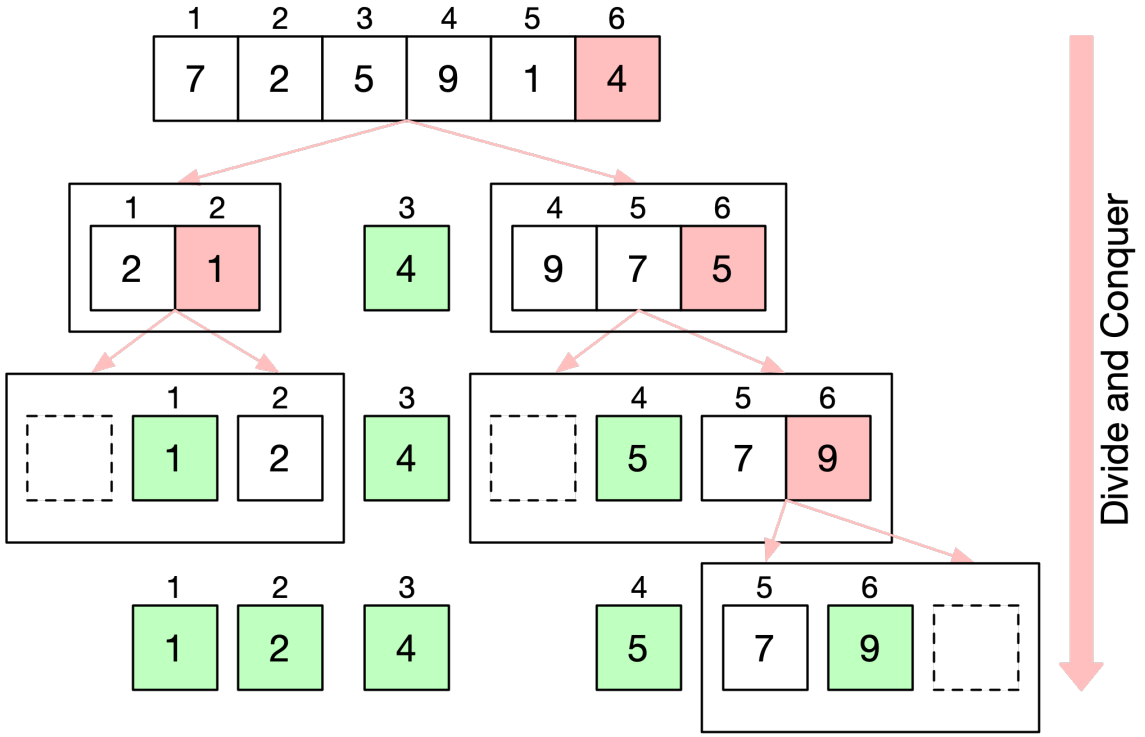
- Risolvendo la ricorrenza con il Master Theorem otteniamo il costo

$$T(n) = \Theta(n \log n)$$

- Il costo di MERGESORT non dipende da come i numeri sono inizialmente organizzati nell'array (caso ottimo, medio e pessimo coincidono)

QuickSort (Tony Hoare, 1959)

1. **Divide**: partizioniamo $A[1, \cdots, n]$ in due sotto-array $A_1 = A[1, \cdots, q-1]$ e $A_2 = A[q+1, \cdots, n]$ tali che tutte le chiavi in A_1 e A_2 siano rispettivamente $\leq$ e $>$ della chiave di $A[q]$ ($A[q]$ scelto durante il partizionamento)
2. **Conquer**: richiamiamo ricorsivamente l'algoritmo su A_1 e A_2
3. **Combine**: non necessario ricombinare A_1 e A_2 (sono già ordinati)



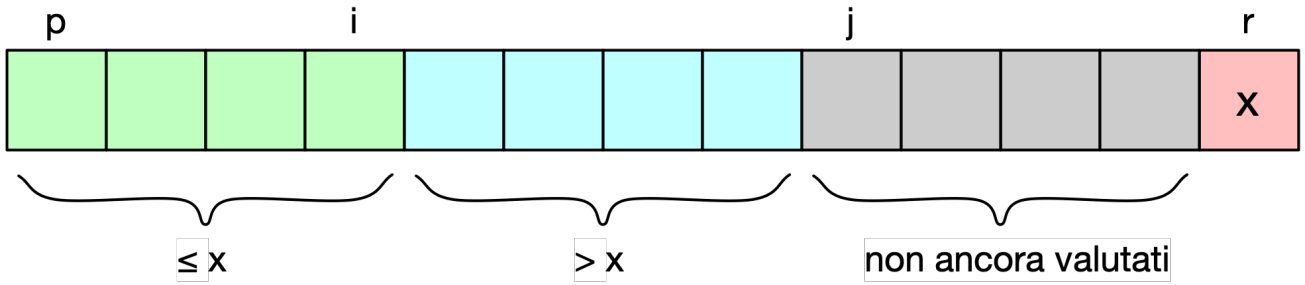
QuickSort: pseudocodice

```
1: function QUICKSORT(ARRAY A[1, ··· , n], INT p, INT r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         ▷ Here ∀i ≤ q − 1, A[i] ≤ A[q] and ∀i ≥ q + 1, A[i] > A[q]
5:         QUICKSORT(A, p, q − 1)
6:         QUICKSORT(A, q + 1, r)
```

- Prima chiamata: $\text{QUICKSORT}(A, 1, n)$
- Partizionamento complesso ma non necessario ricombinare i sotto-array
- La funzione PARTITION sceglie un elemento pivot e riorganizza il sotto-array $A[p, \cdots, r]$ in modo che
  - Tutti le chiavi più piccole della chiave pivot sono nella parte sinistra
  - Tutti le chiavi più grandi della chiave pivot sono nella parte destra
  - Ritorna l'indice q della posizione del pivot in $A[p, \cdots, r]$

QuickSort: pseudocodice di partition

```
1: function PARTITION(ARRAY A[1, ··· , n], INT p, INT r) → INT
2:     x = A[r]                          ▷ Deterministic choice of the pivot
3:     i = p − 1
4:     for j = p, ··· , r − 1 do
5:         if A[j] ≤ x then
6:             SWAP(A, i + 1, j)
7:             i = i + 1
8:     SWAP(A, i + 1, r)                 ▷ Move the pivot value in A[i + 1]
9:     return i + 1                      ▷ Return the pivot index in A
```



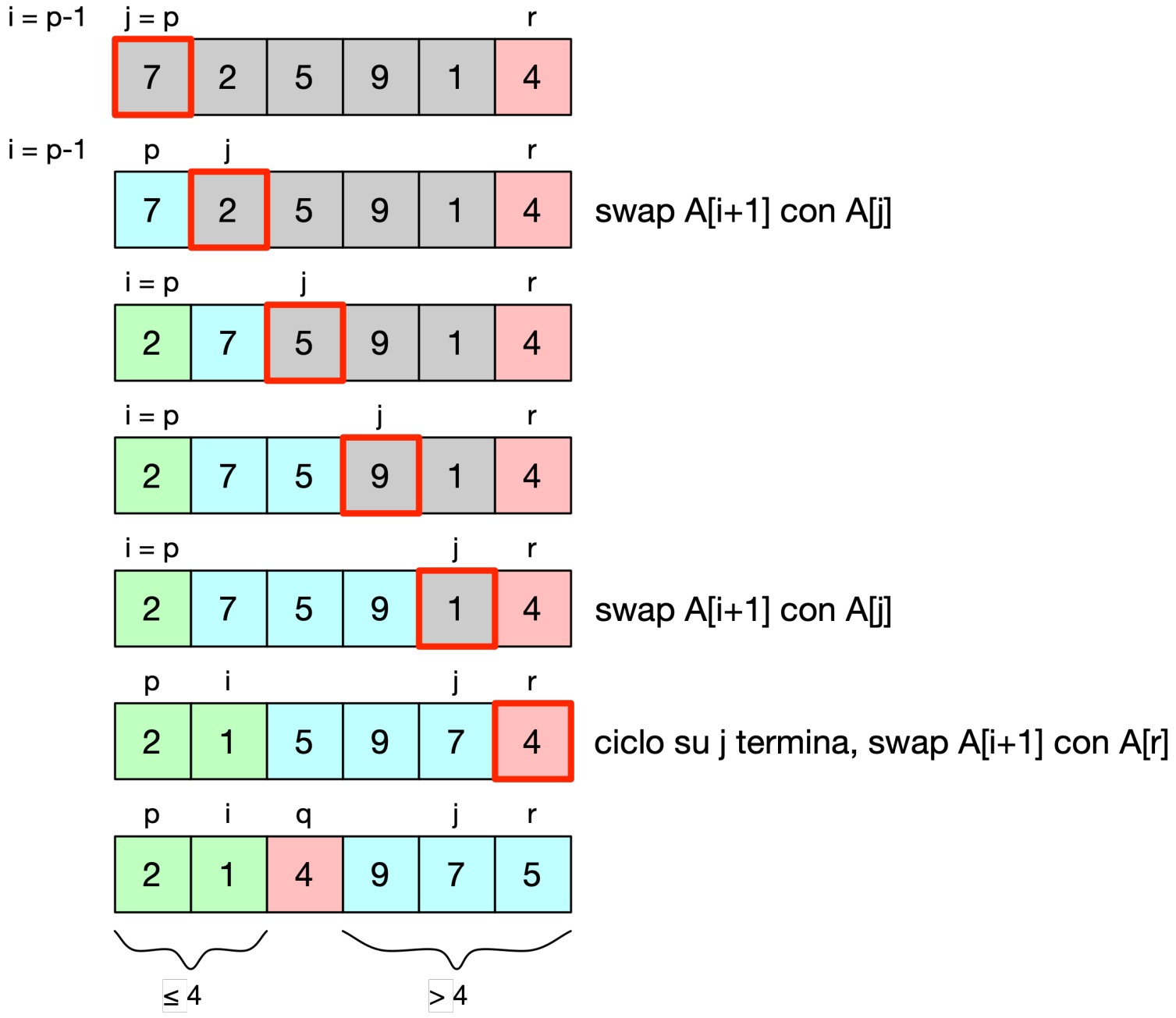
- Costo nel caso ottimo, medio e pessimo: $\Theta(r - p + 1)$
  - PARTITON *valuta* ogni cella in $A[p, \cdots, r]$
- PARTITION agisce in-place ma non è stabile

QuickSort: implementazione in Java

```
1  public static void quicksort(int A[], int p, int r) {
2     if (p < r) {
3       int q = partition(A,p,r);
4       quicksort(A,p,q-1);
5       quicksort(A,q+1,r);
6     }
7  }
8
9  private static int partition(int A[], int p, int r) {
10    int x = A[r];
11    int i = p-1;
12    for(int j = p; j < r; j++)
13      if (A[j] <= x)
14        swap(A,++i,j);
15    swap(A,i+1,r);
16    return i+1;
17 }
```

Molto simile allo pseudocodice: non è nemmeno necessario spostare gli indici

QuickSort: esecuzione di partition



Complessità computazionale di QuickSort

- Il costo computazionale di QUICKSORT dipende da quanto bene PARTITION riesce a partizionare l'array A nei due sotto-array

$$A[p, \cdots, q-1] \text{ and } A[q+1, \cdots, r]$$

- La scelta del pivot comporta sotto-array bilanciati o sbilanciati
  - Sono bilanciati se hanno approssimativamente la stessa lunghezza
  - Sono sbilanciati se le loro lunghezze differiscono di molto

- Il partizionamento determina i costi nel casi ottimo, medio e pessimo

Analisi del caso pessimo di QuickSort

- Partizionamento pessimo: un array ha lunghezza 0 e l'altro $n-1$

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + T(0) + n & n > 1 \end{cases}$$

- Possiamo risolvere l'equazione di ricorrenza col metodo iterativo

$$\begin{aligned} T(n) &= T(n-1) + T(0) + n \\ &= T(n-1) + 1 + n \\ &= T(n-2) + 2 + (n-1) + n \\ &= T(n-3) + 3 + (n-2) + (n-1) + n \\ &\dots \\ &= T(n-i) + i + \sum_{k=0}^{i-1} n - k \end{aligned}$$

La ricorsione termina per $n - i = 0 \Rightarrow i = n$. Il costo nel caso pessimo è

$$T(n) = 1 + n + \sum_{k=0}^{n-1}(n-k) = 1 + n + \sum_{k=1}^{n} k = 1 + n + \frac{n(n+1)}{2} = \Theta(n^2)$$

Analisi del caso ottimo di QuickSort

- Partizionamento ottimo: i due sotto-array sono entrambi lunghi $n/2$

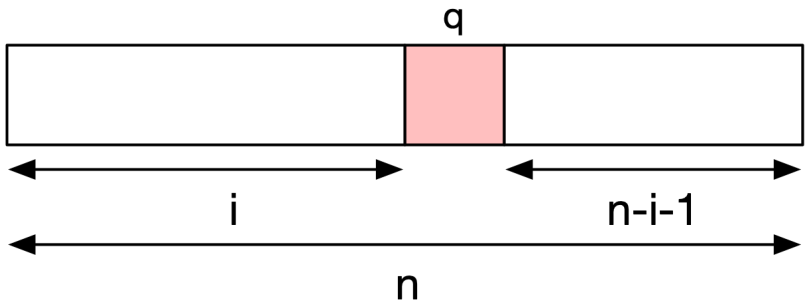
$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

- Possiamo usare il Master Theorem ($\alpha = \beta = 1$)
- Il costo nel caso ottimo è $T(n) = \Theta(n \log n)$
- Nel caso ottimo stesso costo di MERGESORT
- Nel caso pessimo, stesso costo di INSERTIONSORT and SELECTIONSORT

Analisi del caso medio di QuickSort

- Nel caso generale, l'equazione di ricorrenza di QUICKSORT è

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(i) + T(n-i-1) + n & n > 1 \end{cases}$$



- Problema: i (e $n - i - 1$) possono cambiare in ogni chiamata ricorsiva

- Sotto l'assunzione probabilistica che tutte le partizioni siano equiprobabili, possiamo considerare il valore medio e valutare l'equazione di ricorrenza

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-i-1)] + n & n > 1 \end{cases}$$

Analisi del caso medio di QuickSort

- Semplifichiamo l'equazione di ricorrenza del caso medio notando che

$$\sum_{i=0}^{n-1} T(n-i-1) = \sum_{i=0}^{n-1} T(i)$$

ed otteniamo

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n & n > 1 \end{cases}$$

- Con il metodo di sostituzione possiamo dimostrare (slide successiva) che

$$T(n) \leq cn \ln n$$

- Il costo nel caso medio di QUICKSORT è quindi $O(n \log n)$

Analisi del caso medio di QuickSort

- Caso base: $T(2) = T(0) + T(1) + 2 = 4 \leq c \cdot 2 \ln 2 \Rightarrow$ vera $\forall c \geq \frac{2}{\ln 2}$
- Induzione: assumiamo che l'ipotesi $T(i) \leq ci \ln i$ sia vera $\forall 2 \leq i < n$

$$
\begin{aligned}
T(n) &= n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) = n + 2\frac{T(0)+T(1)}{n} + \frac{2}{n}\sum_{i=2}^{n-1} T(i) \\
&\leq n + \frac{4}{n} + \frac{2c}{n}\sum_{i=0}^{n} i \ln i \\
&\leq n + \frac{4}{n} + \frac{2c}{n}\int_0^n x \ln x \, dx \qquad \text{(integrazione per parti: } \frac{x^2}{2}\ln x - \frac{x^2}{4}\text{)} \\
&= n + \frac{4}{n} + \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) \\
&= n + \frac{4}{n} + cn \ln n - c\frac{n}{2} \\
&\leq cn \ln n
\end{aligned}
$$

Vera per $c = 4$ e $n_0 = 2$

QuickSort: scelta del pivot

- Nella nostra implementazione di PARTITION il pivot è sempre l'ultimo elemento $A[q]$ nel sotto-array $A[p, \cdots, q]$
  - Quali sono istanze in input che comportano il caso pessimo?
- La nostra analisi del caso medio si basa sull'assunzione probabilistica che tutte le partizioni siano equiprobabili, mentre con la nostra strategia per la scelta del pivot il bilanciamento delle partizioni dipende dall'ordinamento iniziale del vettore in input
  - Array quasi ordinati comportano il caso pessimo
  - In situazioni reali, array quasi ordinati sono più probabili di array totalmente *disordinati*
- Possiamo utilizzare la randomizzazione per correggere tale problema

```
1: function RPARTITION(ARRAY A[1, ··· , n], INT p, INT r) → INT
2:     i = RANDOM(p,r)                    ▷ Random choice of the pivot
3:     SWAP(A, i, r)
4:     return PARTITION(A,P,R)
```

- Una scelta randomizzata del pivot aumenta la probabilità che le partizioni siano equiprobabili, cosi come assunto nell'analisi del caso medio

Algoritmo di ordinamento in Java

- La classe *java.util.Arrays* in Java contiene metodi per manipolare array
- Il metodo *sort* della classe fornisce un algoritmo di ordinamento
- L'algoritmo implementato è una combinazione di InsertionSort, MergeSort e QuickSort
- Il QuickSort implementato è noto con il nome di *DualPivot Quicksort*
  - Utilizza due pivot invece che uno
  - Stesso costo computazionale di QuickSort
  - Costo $O(n \log n)$ su molte istanze su cui QuickSort ha costo pessimo
- InsertionSort è utilizzato su istanze piccole (≤ 47):
  - Più veloce (in sec) di QuickSort e MergeSort
- MergeSort viene preferito a QuickSort su istanze *grandi* (≥ 286) che presentano poche sotto-sequenze decrescenti
  - QuickSort rischia di avere caso pessimo su istanze *quasi ordinate*

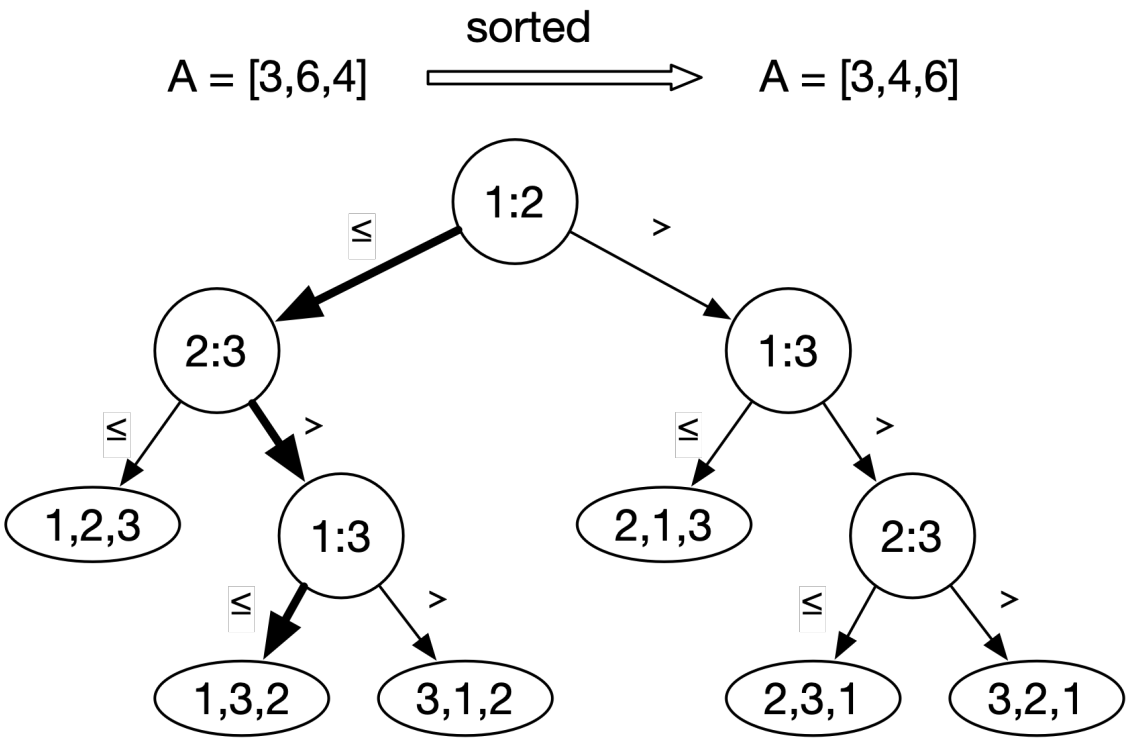
Algoritmi di ordinamento: riassunto

	Caso ottimo	Caso medio	Caso pessimo
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n^2)$

	In place	Stabile
SelectionSort	Si	Si
InsertionSort	Si	Si
MergeSort	No	Si
QuickSort	Si	No

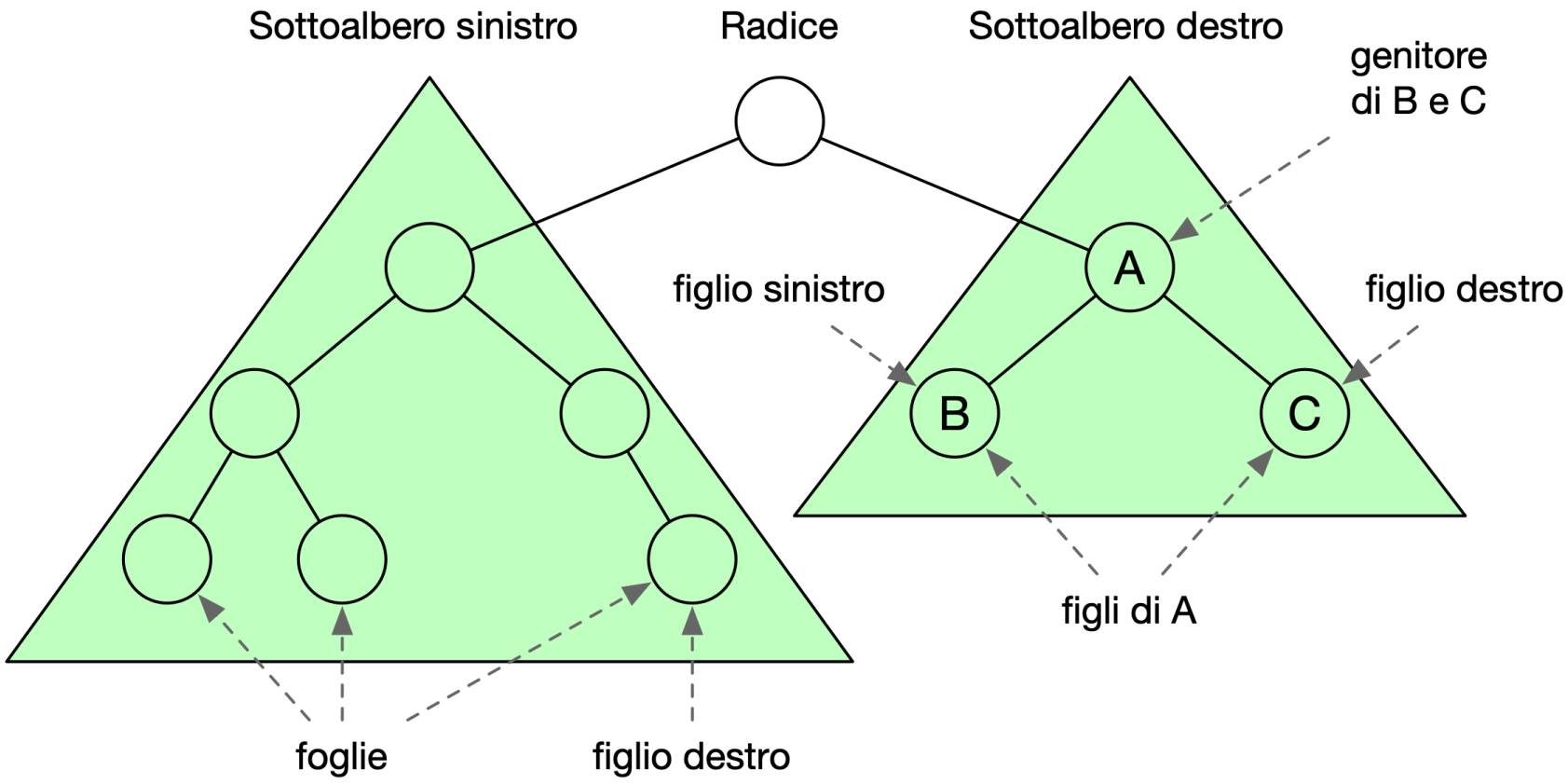
Lower Bound per ordinamento comparativo

- Tutti gli algoritmi considerati finora si basano sul confronto
  - L'ordinamento è basato sul confronto $(<, =, >)$ tra valori
- Esistono algoritmi di ordinamento comparativi con prestazioni migliori di $O(n \log n)$ nel caso pessimo?
- Possiamo descrivere un algoritmo comparativo con un albero di decisione
- Esempio: albero di decisione di INSERTIONSORT



Alberi Binari: terminologia

- Un Albero Binario è un Albero in cui ogni nodo ha al massimo due figli



- La profondità di un nodo u è la lunghezza del percorso (unico) che va dalla radice al nodo u (numero di archi)
- L'altezza di un Albero è la sua massima profondità

Lower Bound per ordinamento comparativo

- Idea
  - Ogni algoritmo di ordinamento comparativo può essere descritto da un albero di decisione
  - Ogni nodo corrisponde al confronto di un paio di elementi nell'array
  - L'esecuzione dell'algoritmo può essere tracciata sull'albero seguendo un percorso radice-foglia
- Properietà dell'albero di decisione
  - Albero binario in cui ogni nodo interno ha esattamente due figli
  - La sua altezza corrisponde al numero di confronti nel caso pessimo
  - L'altezza media corrisponde al numero di confronti nel caso medio
  - Il numero di foglie dell'albero è almeno uguale a $n!$, i.e. il numero possibile di permutazioni degli indici di un array di lunghezza n

Lower Bound per ordinamento comparativo

Teorema

Sia T_k un albero binario con k foglie in cui ogni nodo interno ha esattamente due figli e sia $h(T_k)$ l'altezza di T_k. Allora $h(T_k) \geq \log_2 k$

- Dimostrazione (per induzione su k)
  - Caso base: $h(T_1) = 0 \geq \log_2 1 = 0$
  - Induzione: siano T_{k_1} e T_{k_2} rispettivamente il sotto-albero sinistro e destro di T_k, dove $k = k_1 + k_2$ e assumiamo che $k_1 \geq k_2$. La nostra ipotesi induttiva è che $h(T_{k'}) \geq \log_2 k'$ per ogni $k' < k$. Allora

$$\begin{aligned} h(T_k) &= 1 + \max(h(T_{k_1}), h(T_{k_2})) \\ &\geq 1 + h(T_{k_1}) \\ &\geq 1 + \log_2 k_1 \\ &= \log_2 2 + \log_2 k_1 = \log_2 2k_1 \\ &\geq \log_2 (k_1 + k_2) = \log_2 k \end{aligned}$$

Lower Bound per ordinamento comparativo

Teorema

Ogni algoritmo di ordinamento comparativo richiede $\Omega(n \log n)$ confronti nel caso pessimo

- Dimostrazione
  - Ogni algoritmo di ordinamento comparativo richiede nel caso pessimo un tempo di calcolo proporzionale all'altezza dell'albero di decisione (numero di confronti nel caso pessimo)
  - L'albero di decisione ha almeno $n!$ foglie, dove n è il numero di elementi da ordinare
  - Per il teorema precedente, l'altezza di un albero binario in cui ogni nodo interno ha esattamente due figli e con almeno $n!$ foglie è

$$\Omega(\log n!) = \Omega(n \log n)$$

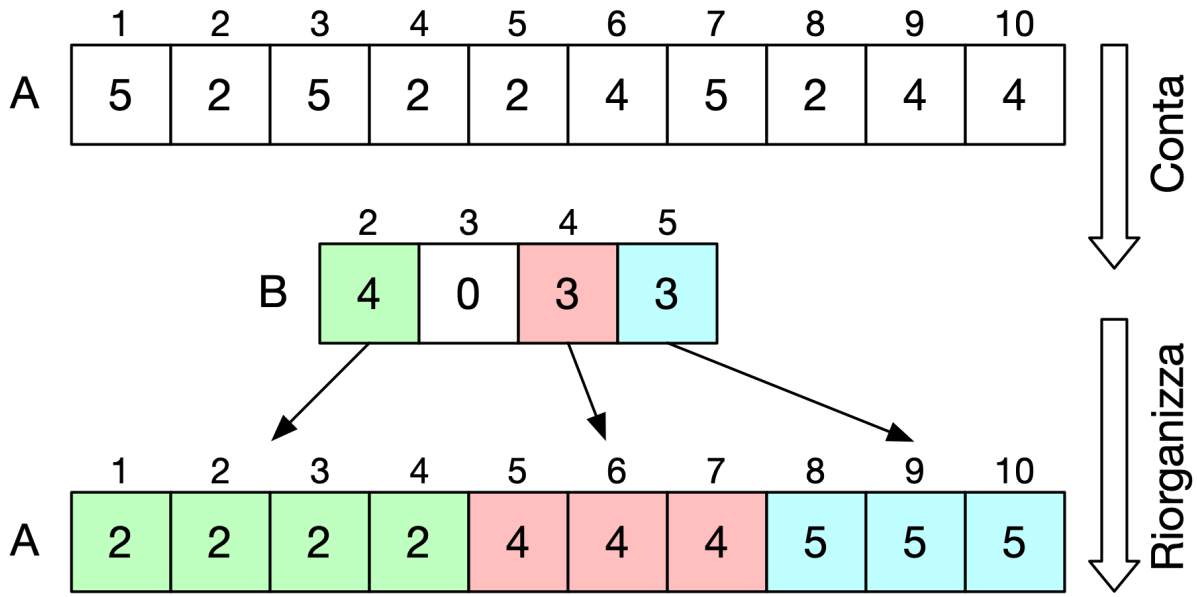
- Conseguenza: MERGESORT è un algoritmo di ordinamento comparativo asintoticamente ottimo

Ordinamento in tempo lineare

- Il lower bound per il caso pessimo vale solo per algoritmi comparativi
- Possiamo utilizzare altre operazioni per determinare un ordinamento
  - Il lower-bound per il caso pessimo non si applica a tali algoritmi
  - Questo tipicamente implica vincoli sul tipo di elementi da ordinare
- Vediamo due algoritmi di ordinamento che, sotto determinate condizioni, possono ordinare in tempo lineare nel caso pessimo
  - CountingSort
  - RadixSort

CountingSort

- Assunzioni: le chiavi da ordinare sono interi nell'intervallo $[a, b]$
  - a e b possono essere interi negativi
- Idea del CountingSort:
  1. Usa un array $B[]$ di lunghezza $b - a + 1$ per contare quante volte ogni valore chiave $x \in [a, b]$ appare nell'array in input $A[]$
  2. Riorganizza tali valori in $A[]$



Nota: nell'esempio usiamo un array $B[a, .., b]$ per semplicità

CountingSort: pseudocodice

```
1:  function CountingSort(Array A[1, ··· , n])
2:      a =min(A), b =max(A), k = b − a + 1
3:      Let B[1, ··· , k] be a new Array
4:      for i = 1, ··· , k do
5:          B[i] = 0
6:      ▷ Count
7:      for i = 1, ··· , n do
8:          B[A[i] − a + 1] = B[A[i] − a + 1] + 1
9:      ▷ Re-arrange
10:     j = 1
11:     for i = 1, ··· , k do
12:         while B[i] > 0 do
13:             A[j] = i + a − 1
14:             B[i] = B[i] − 1
15:             j = j + 1
```

- Usiamo un array $B[1,\cdots,b-a+1]$ e riscaliamo da $[a, b]$ in $[1,b-a+1]$
  - per ogni $x \in [a, b]$, $x - a + 1 \in [1, b - a + 1]$
  - per ogni $x \in [1, b - a + 1]$, $x + a - 1 \in [a, b]$

CountingSort: implementazione in Java

```
1  public static void countingsort(int A[]) {
2    int a = Arrays.stream(A).min().getAsInt();
3    int b = Arrays.stream(A).max().getAsInt();
4    int k = b-a+1;
5    int[] B = new int[b - a + 1];
6
7    for(int i = 0; i < A.length; i++)
8      B[A[i]-a]++;
9
10   for(int i = 0, j = 0; i < k; i++)
11     while(B[i]-- > 0)
12       A[j++] = i+a;
13 }
```

- Notiamo lo shift degli indici a riga 8 e 12
- Non è necessario inizializzare a zero l'array B

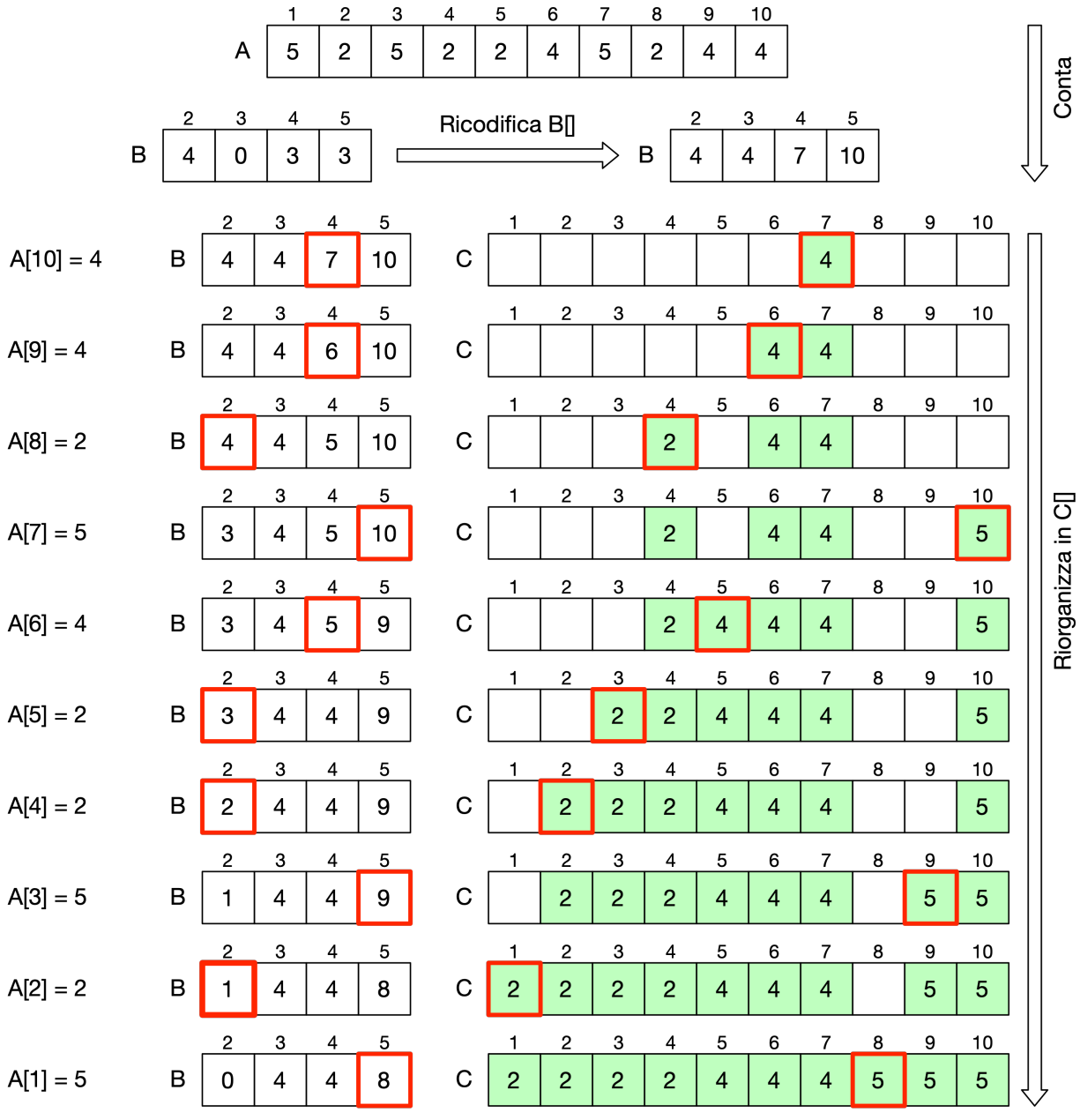
Complessità computazionale di CountingSort

- Costo ottimo, medio e ottimo: $\Theta(n + k)$, dove $k = b - a + 1$
  - Le funzioni MIN e MAX costano $\Theta(n)$
  - Il ciclo for a riga 4 viene eseguito k volte
  - Il ciclo for a riga 7 viene eseguito n volte
  - I cicli for e while a righe 11-12 sono eseguiti n volte
    - La somma totale su B è uguale ad n
- Se $k = O(n)$ allora il costo è $\Theta(n)$
  - Il costo dipende anche dalla dimensione del range $[a, b]$

Ulteriori considerazioni su CountingSort

- La nostra implementazione di CountingSort ordina solo array di interi
- Non possiamo usare tale implementazione per ordinare array di dati associati a chiavi intere
- Comunque, possiamo modificare di poco la nostra implementazione per ottenere un algoritmo di ordinamento lineare in tale caso più generale
- La nostra nuova implementazione può essere anche stabile
- Benchè aggiungiamo un po' di overhead in termini di memoria e tempo di calcolo il costo asintotico in termini di memoria e tempo rimane invariato

CountingSort (chiavi e dati): esempio



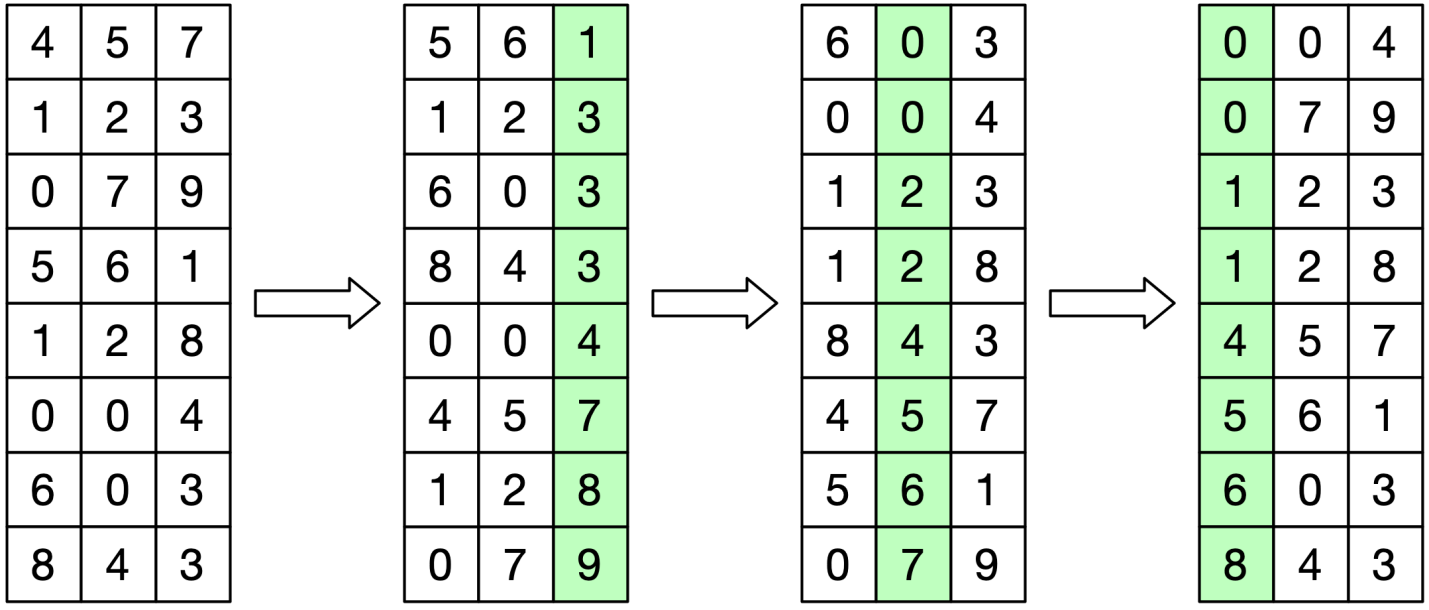
CountingSort (chiavi e dati): pseudocodice

```
 1: function CountingSort(Array A[1, ··· , n])
 2:     a = MinKey(A), b = MaxKey(A), k = b − a + 1
 3:     Let B[1, ··· , k] be a new Array
 4:     Let C[1, ··· , n] be a new Array
 5:     for i = 1, ··· , k do B[i] = 0
 6:     ▷ Count
 7:     for i = 1, ··· , n do B[A[i].key − a + 1] = B[A[i].key − a + 1] + 1
 8:     ▷ Recode B[]
 9:     for i = 2, ··· , k do
10:         B[i] = B[i] + B[i − 1] ▷ B[i] now contains the number of keys ≤ i + a − 1
11:     ▷ Re-arrange in C[]
12:     for i = n, ··· , 1 do
13:         C[B[A[i].key − a + 1]] = A[i]        ▷ Copy keys and data from A[] to C[]
14:         B[A[i].key − a + 1] = B[A[i].key − a + 1] − 1
15:     for i = 1, ··· , n do
16:         A[i] = C[i]                          ▷ Copy sorted data from C[] to A[]
```

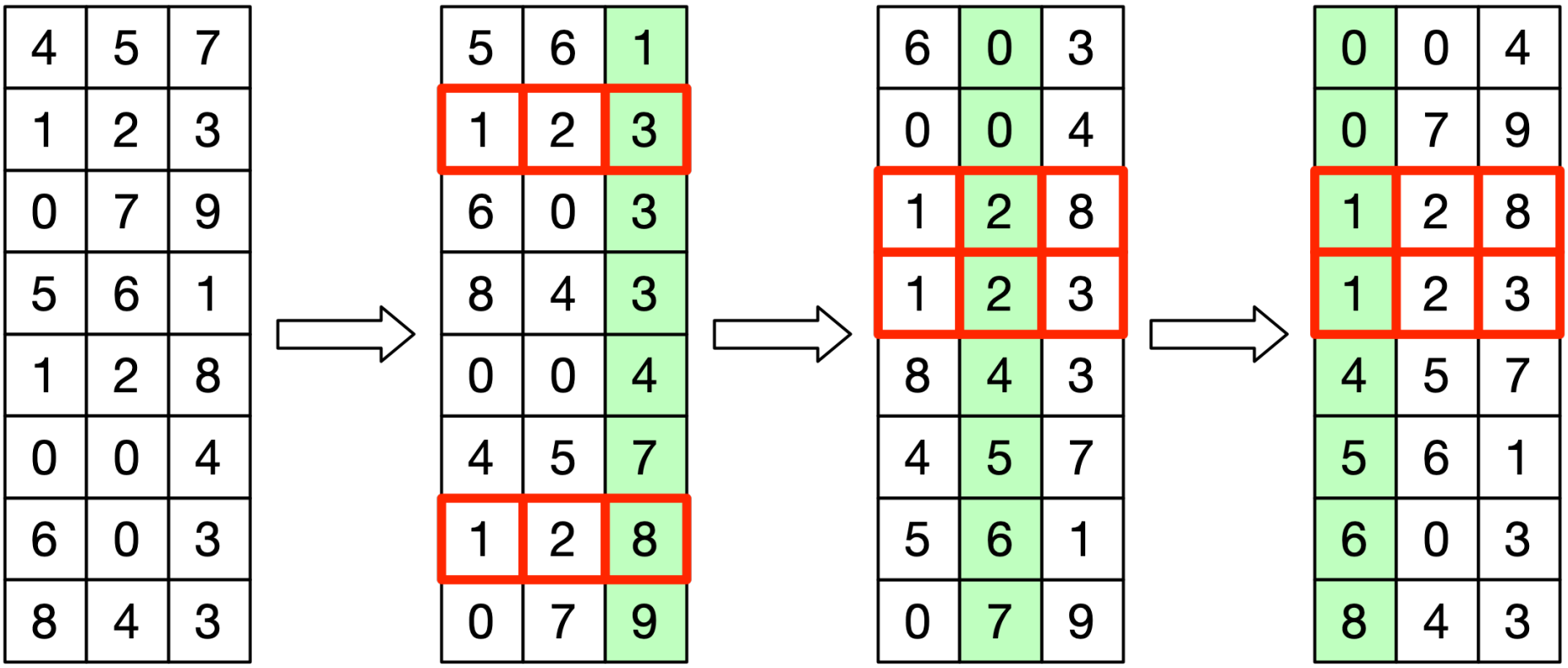
- Il ciclo da n ad 1 a riga 12 rende la nuova implementazione stabile

RadixSort

- Assunzioni: i valori chiave sono composti da cifre o caratteri
  - La radice è il numero di cifre univoche che rappresentano un numero
- Idea del RadixSort:
  1. Ordina prima rispetto alla cifra meno significativa, poi rispetto alla penultima cifra meno significativa e cosi via
  2. L'ordinamento delle cifre deve essere stabile



RadixSort con ordinamento non stabile



RadixSort: pseudocodice e complessità

```
1: function RADIXSORT(ARRAY A[1, ··· , n])
2:     d = max key length
3:     for i = 1, ··· , d do
4:         stable sort of A on the i-th digit of the keys
```

- Il massimo numero di cifre in una chiave è d
- Le cifre sono numerate da 1 a d da destra verso sinistra
- Se usiamo COUNTINGSORT come algoritmo stabile il costo ottimo, medio e pessimo di RADIXSORT è $\Theta(d(n+k))$
  - d è il numero massimo di cifre in una chiave
  - k è il numero di possibili valori per una cifa
  - Se le chiavi sono interi $k = 10$, mentre se le chiavi sono stringhe $k =$ *numero di caratteri ammessi nella stringa*
- Se $k = O(n)$ e d è un valore costante (lunghezza massima limitata) allora il costo è $\Theta(n)$ (lineare sul numero di elementi in $A[]$)

Riassunto

	Caso ottimo	Caso medio	Caso pessimo
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n^2)$
CountingSort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$
RadixSort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$

	In place	Stabile
SelectionSort	Si	Si
InsertionSort	Si	Si
MergeSort	No	Si
QuickSort	Si	No
CountingSort	No	Si
RadixSort	No	Si

- $n =$ numero di elementi da ordinare
- $d =$ numero massimo di cifre in una chiave
- $k =$ ampiezza del range di valori chiave