Algoritmi e Strutture Dati

Algoritmi di Ordinamento

Maria Rita Di Berardini, Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

Il problema dell'ordinamento

Il problema dell'ordinamento di un insieme è un problema classico dell'informatica

Ha una indiscutibile valenza in ambito applicativo: spesso si ritrova all'interno di problemi ben più complicati

È anche un utile strumento didattico: il problema in se è molto semplice e chiunque è in grado di comprenderne i termini essenziali

Per la sua risoluzione sono stati proposti numerosi algoritmi molto eleganti che consentono di evidenziare gli aspetti fondamentali della progettazione e della costruzione di un algoritmo efficiente

Il problema dell'ordinamento

Dato un insieme di n numeri $\{a_1, a_2, \ldots, a_n\}$, trovare un'opportuna permutazione π degli indici tale che $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

Input: $\{a_1, a_2, \ldots, a_n\}$

Output: $\{a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}\}$

dove π è un'opportuna permutazione degli indici $1, \ldots, n$

Il problema dell'ordinamento

Più in generale, se $A = \{a_1, a_2, \ldots, a_n\}$ è un insieme di n elementi **qualsiasi** su cui è definita una relazione d'ordine totale (i.e. riflessiva, antisimmetrica e transitiva) $\preceq$, trovare un'opportuna permutazione π degli indici tale che $a_{\pi(1)} \preceq a_{\pi(2)} \preceq \ldots \preceq a_{\pi(n)}$

Il problema dell'ordinamento ammette una soluzione unica a meno di elementi uguali; se $A = \{a_1 = 7, a_2 = 3, a_3 = 13, a_4 = 7\}$, allora abbiamo due soluzioni distinte ma equivalenti $\{a_2, a_1, a_4, a_3\}$ e $\{a_2, a_4, a_1, a_3\}$

Complessità dei principali algoritmi di ordinamento

Algoritmo	Caso migliore	Caso medio	Caso peggiore
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
InsertionSort √	$O(n)$	$O(n^2)$	$O(n^2)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
MergeSort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
QuickSort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
HeapSort √	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

Il problema dell'ordinamento

Esamineremo i principali algoritmi che operano esclusivamente mediante confronti:

- l'InsertionSort,
- il SelectionSort,
- il BubbleSort,
- il QuickSort,
- il MergeSort e
- lo HeapSort

Il problema dell'ordinamento

I primi tre, SelectionSort, InsertionSort e BubbleSort, sono estremamente semplici; il prezzo da pagare alla semplicità di questi algoritmi è la complessità computazionale (data in termini del numero di confronti): $O(n^2)$ anche nel caso medio

Il QuickSort ci consente di raggiungere una complessità di $O(n \log_2 n)$ solo nel caso medio, mentre nel caso peggiore ha una complessità di $O(n^2)$

Lo HeapSort e il MergeSort hanno una complessità, anche nel caso peggiore, pari a $O(n \log_2 n)$

Il problema dell'ordinamento

Il limite inferiore alla complessità del **problema dell'ordinamento** se risolto esclusivamente mediante confronti (senza informazioni aggiuntive sull'insieme da ordinare) è proprio $n \log_2 n$

Possiamo quindi concludere lo HeapSort e il MergeSort sono algoritmi di ordinamento **ottimali**

Esistono delle soluzioni, meno generali, ma più efficienti che si basano su assunzioni aggiuntive come ad esempio la presenza di elementi duplicati, il valore massimo e minimo all'interno dell'insieme o altre informazioni che consento di introdurre delle ottimizzazioni (CountingSort, BucketSort)

SelectionSort

È un algoritmo estremamente intuitivo e semplice

Nella pratica è utile quando l'insieme da ordinare è abbastanza piccolo e dunque può essere utilizzato anche un algoritmo non molto efficiente con il vantaggio di non rendere troppo sofistica la codifica del programma che lo implementa

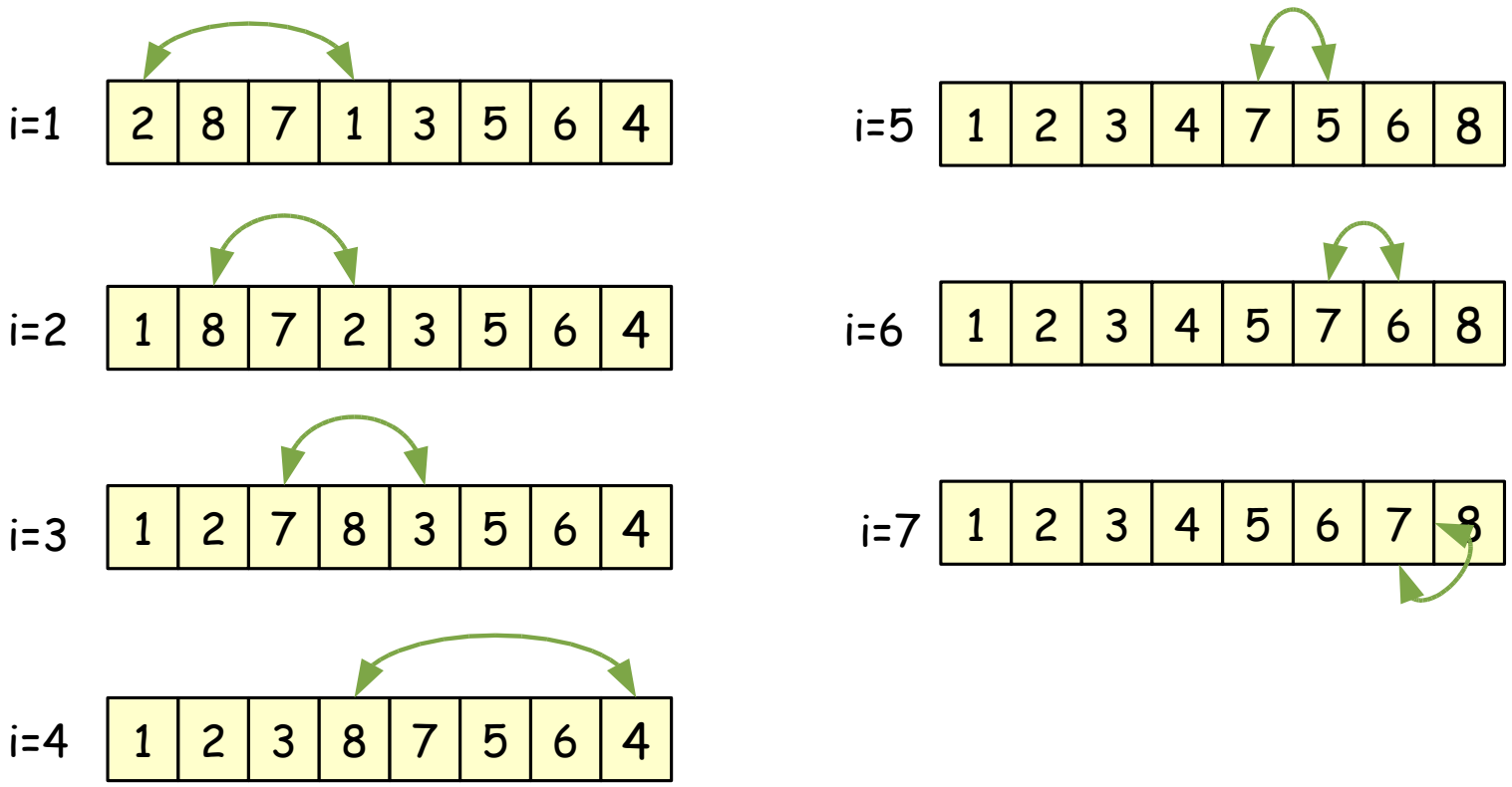
Idea: ripetere $n-1$ volte una procedura che, durante la i-esima iterazione, seleziona l'i-esimo elemento più piccolo dell'insieme e lo scambia con quello che di trova in posizione i

SelectionSort: l'algoritmo

SelectionSort(A)

```
1.  for i ← 1 to length[A] - 1
2.      do min ← i
3.          for j ← i + 1 to length[A]
4.              do if A[j] < A[min] then min ← j
5.          scambia A[min] ↔ A[i]
```

SelectionSort: un esempio



SelectionSort: analisi della complessità

Dal punto di vista delle operazioni svolte, non esiste un caso particolarmente favorevole, o al contrario, particolarmente sfavorevole: l'algoritmo esegue lo stesso numero di operazioni qualunque sia la configurazione iniziale dell'array A

Ad ogni iterazione del ciclo più esterno, il ciclo più interno (riga 3) esegue esattamente $n - i$ confronti. Il numero totale di confronti è :

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

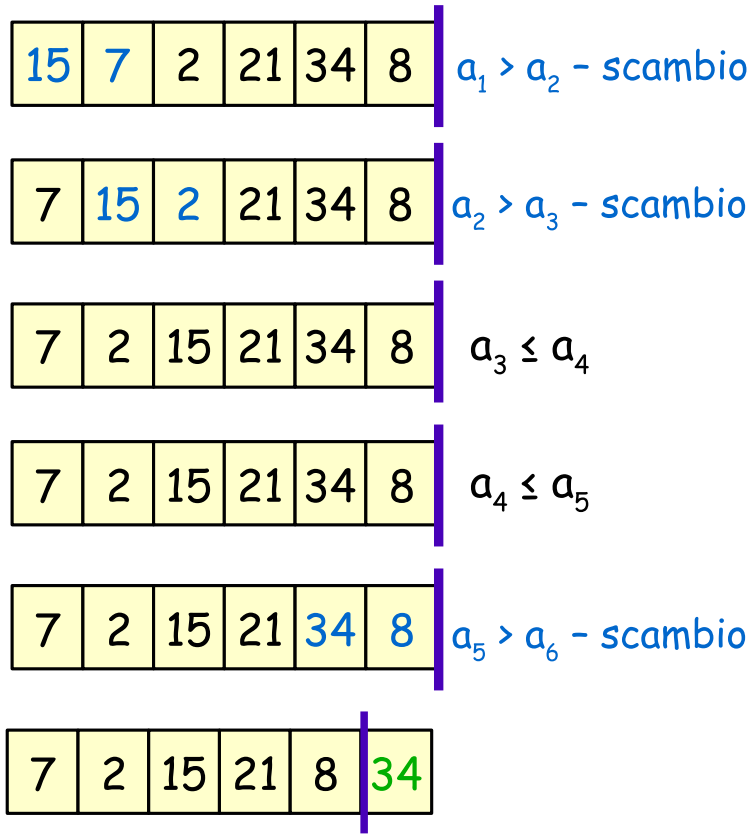
BubbleSort

Idea: far risalire gli elementi più grandi verso l'alto (i.e. nelle posizioni di indice più alto) e, nel contempo, far ridiscendere gli elementi più piccoli verso verso il basso (i.e. posizioni di indice più basso)

Strategia: scorre più volte la sequenza in input confrontando, ad ogni passo, l'ordinamento reciproco di elementi contigui e scambiando le posizioni di eventuali coppie non ordinate

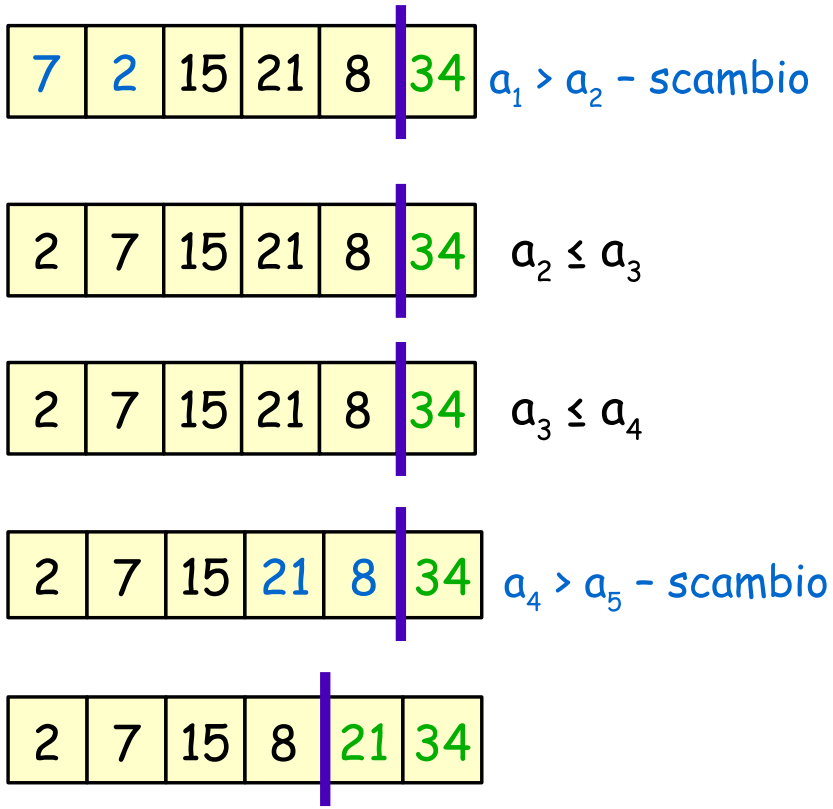
BubbleSort: un esempio

Dopo la prima scansione l'elemento maggiore (**34**) viene collocato nella posizione corretta (l'ultima); le scansioni successive potranno non considerare l'ultima posizione



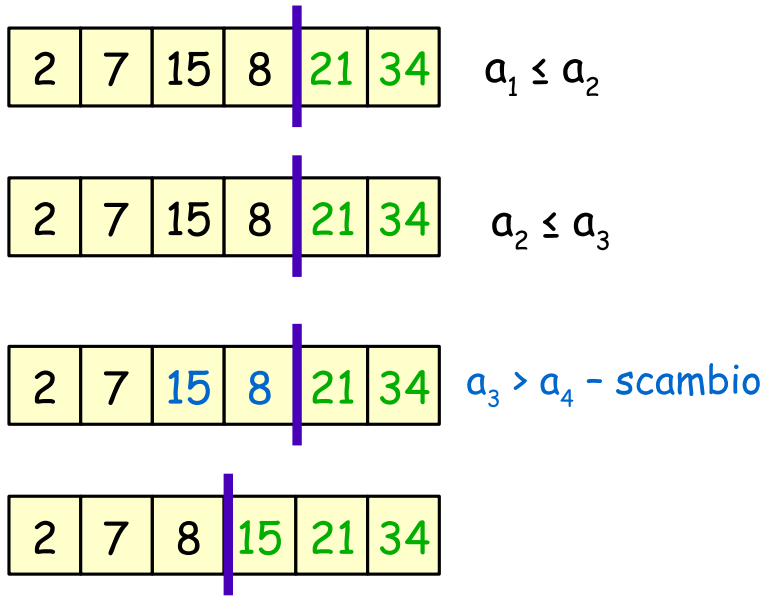
BubbleSort: un esempio

Al termine della seconda iterazione il secondo elemento più grande (**21**) si trova nella posizione corretta (la penultima); le scansioni successive potranno non considerare la penultima posizione, di nuovo aggiorniamo il limite



BubbleSort: un esempio

Al termine della terza iterazione **15** si trova in terzultima posizione.
A questo punto il vettore è ordinato; una (eventuale) scansione successiva non effettuerebbe alcun scambio



BubbleSort: l'algoritmo (prima versione)

BubbleSort(A)

1. **for** $j \leftarrow length[A]$ downto 2 ▷ j è il limite
2. **for** $i \leftarrow 1$ to $j - 1$
3. **do if** $A[i] > A[i+1]$
4. **then** scambia $A[i] \leftrightarrow A[i+1]$

Questa prima versione del BubbleSort è corretta ma, il numero di confronti eseguiti dall'algoritmo è lo stesso per ogni possibile configurazione dell'input

BubbleSort: analisi della complessità

Caso migliore (vettore ordinato in maniera crescente): per $j = n, .., 2$, il ciclo più interno (riga 2) esegue $j - 1$ confronti e 0 scambi. Il numero totale di scambi è 0; il numero totale di confronti è:

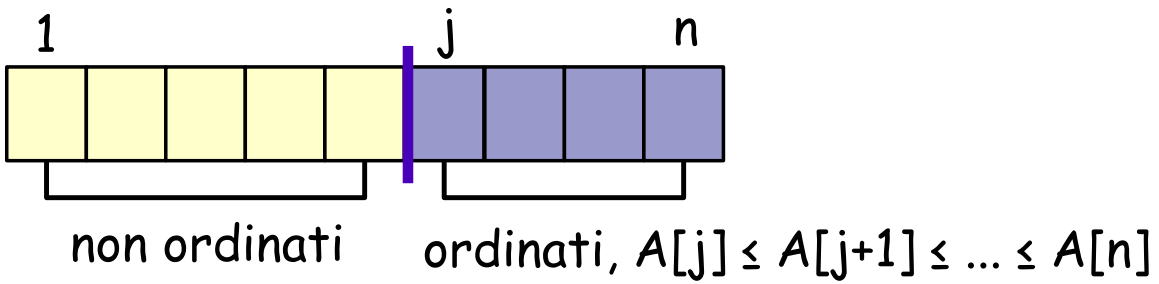
$$\sum_{j=2}^{n} j - 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

Caso peggiore (vettore ordinato in maniera decrescente): per $j = n, .., 2$, il ciclo pi'u interno (riga 2) esegue $j - 1$ confronti e $j - 1$ scambi. Il numero totale di confronti (e di scambi) è:

$$\sum_{j=2}^{n} j - 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

BubbleSort: l'algoritmo (seconda versione)

Esaminiamo lo stato del vettore durante una generica iterazione del ciclo più esterno (riga 1). Fissiamo j, abbiamo che:



Assumiamo ora che durante l'esecuzione del ciclo più interno (riga 2) non venga effettuato alcuno scambio. Allora $A[1] \leq A[2] \leq ... \leq A[j-1] \leq A[j]$ e il vettore è già ordinato

Possiamo usare questa informazione per ridurre il numero di confronti necessari per ordinare il vettore

BubbleSort: l'algoritmo (seconda versione)

BubbleSort(A)

```
1.  for j ← length[A] downto 2   ▷ j è il limite
2.      scambi ← 0
3.      for i ← 1 to j − 1
4.          do if A[i] > A[i + 1]
5.                then scambia A[i] ↔ A[i + 1]
6.                     scambi ↔ scambi + 1
7.      if scambi = 0 then return
```

Questa nuova versione si comporta meglio della precedente **solo** nel caso migliore

BubbleSort: analisi della complessità

Caso migliore: durante l'unica iterazione del ciclo più interno, vengono eseguiti $n-1$ confronti e 0 scambi (questo causa la terminazione dell'algoritmo). Il numero totale di scambi è di nuovo 0; il numero totale di confronti è:

$$n-1 = O(n)$$

Caso peggiore: fissato $j = n, \ldots, 2$, il ciclo più interno (riga 2) esegue $j-1$ confronti e $j-1$ scambi. Esattamente come nell'altro caso:

$$\sum_{j=2}^{n} j-1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

BubbleSort: analisi della complessità

L'analisi nel **caso medio** del bubblesort è abbastanza complessa; si basa sul numero di inversioni presenti nel vettore, dove:

$$\text{inv}(A) = |\{i = 1, \ldots, n-1 \mid a_i > a_{i+1}\}|$$

e definisce il numero di elementi del vettore "fuori posto". Applicando un analisi del tutto simile a quella vista per l'InsertionSort dà un numero di scambi nel caso medio pari a

$$\frac{n(n-1)}{4} = O(n^2)$$

Un analisi più complessa (per la quale rinviamo a Knuth) da un numero di confronti nel caso medio pari a

$$\frac{1}{2}n^2 - n\log_2 n + o(n) = O(n^2)$$

Progettazione di Algoritmi

Approccio incrementale: assumo la porzione $A[1..j-1]$ del vettore ordinata ed inserisco il successivo elemento $A[j]$ nella giusta posizione

Approccio divide-et-impera: divido il problema in input in k sottoproblemi e opero **ricorsivamente** sui k sottoproblemi (la cui dimensione è approssimativamente n/k)

Concetto di ricorsione: un algoritmo si dice ricorsivo se al suo interno (tra i suoi comandi) sono presenti chiamate a se stesso per gestire sottoproblemi analoghi a quello dato

Divide-et-Impera

L'approccio seguito da questa tipologia di algoritmi consiste nel suddividere il problema in input in un certo numero di sottoproblemi simili a quello di partenza ma di dimensione inferiore

Una volta risolti ricorsivamente i sottoproblemi, combinano le soluzioni trovate per creare la soluzione al problema dato

Distinguiamo tre fasi principali: Divide, Impera e Combina

Un esempio classico di algoritmo che su basa su un approccio divide-et-impera è un algoritmo di ordinamento noto come merge-sort

Algoritmo di ordinamento MergeSort

Le tre fasi possono essere così descritte

- **Divide**: gli n elementi della sequenza da ordinare vengono in due sottosequenze di (approssimativamente) $n/2$ elementi ciascuna
- **Impera**: ordina, usando ricorsivamente il merge sort, le due sottosequenze
- **Combina**: fonde le due sottosequenze per produrre come risposta la sequenza ordinata

Algoritmo di ordinamento MergeSort

Il processo di suddivisione si ferma quando la sequenza da ordinare ha lunghezza 1

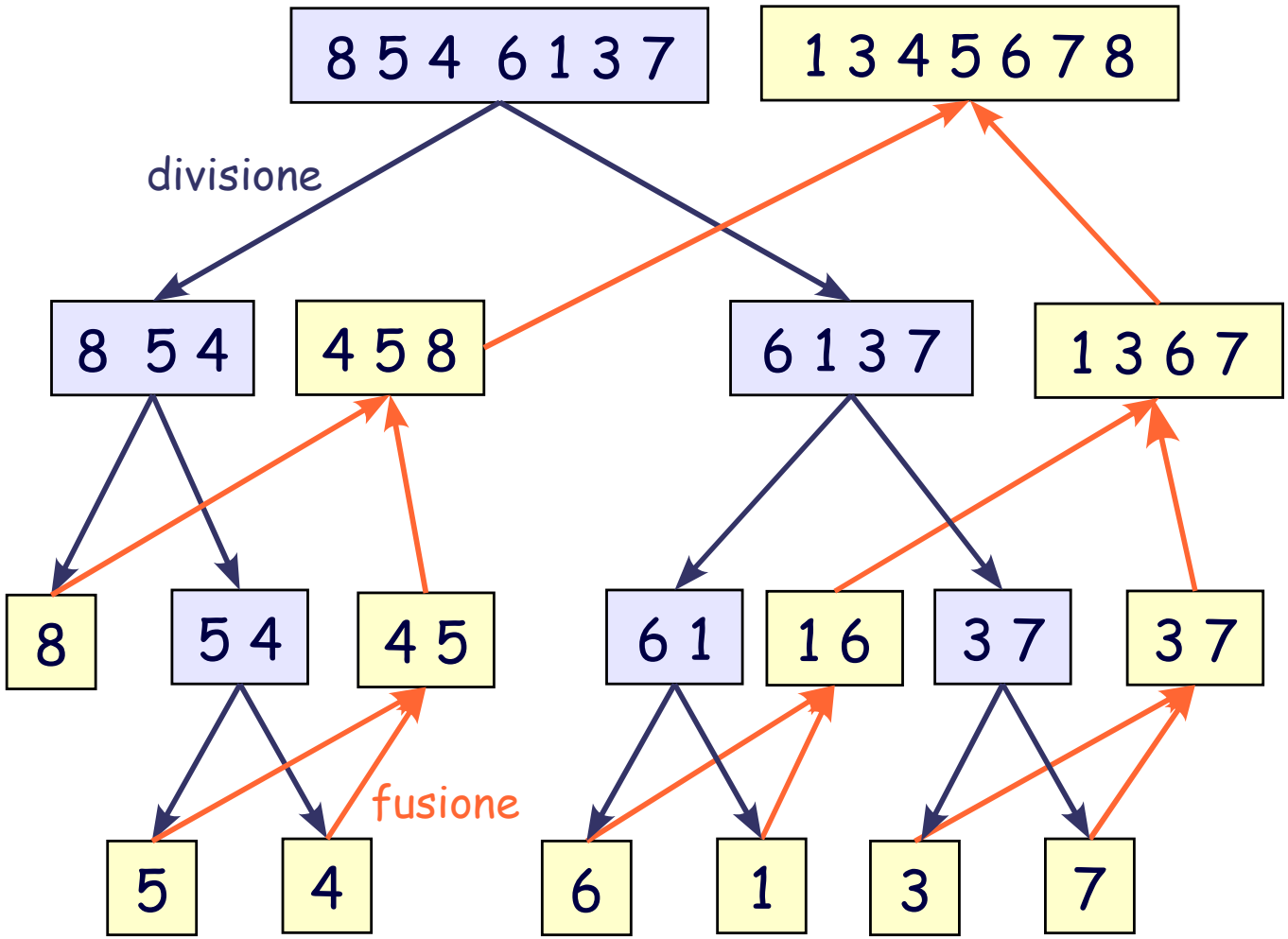
Il passo **Combina** fonde le due sequenze utilizzando una procedura ausiliaria di fusione $\text{MERGE}(A, p, q, r)$, dove: A è un array e p, q, r sono indici di elementi dell'array tali che $p < q < r$

$\text{MERGE}(A, p, q, r)$ assume che $A[p, ..., q]$ e $A[q + 1, ..., r]$ siano ordinati e genera $A[p, ..., r]$ ordinato

Algoritmo di ordinamento MergeSort

$\textsc{MergeSort}(A, \mathit{left}, \mathit{right})$
  if $\mathit{left} < \mathit{right}$
    then $\mathit{mid} \leftarrow \lfloor (\mathit{left} + \mathit{right})/2 \rfloor$
      $\textsc{MergeSort}(A, \mathit{left}, \mathit{mid})$
      $\textsc{MergeSort}(A, \mathit{mid} + 1, \mathit{right})$
      $\textsc{Merge}(A, \mathit{left}, \mathit{mid}, \mathit{right})$

$A = \{8, 5, 4, 6, 1, 3, 7\}$


8 5 4 6 1 3 7
1 3 4 5 6 7 8
divisione
8 5 4
4 5 8
6 1 3 7
1 3 6 7
8
5 4
4 5
6 1
1 6
3 7
3 7
fusione
5
4
6
1
3
7

Fusione di due sottosequenze ordinate

```
MERGE(A, p, q, r)
    m1 ← q − p + 1  ▷ dim di A[p, .., q]
    m2 ← r − q      ▷ dim di A[q + 1, ..., r]
    B[1, ..., m1] ← A[p, ..., q]
    C[1, ..., m2] ← A[q + 1, ..., r]
    i, j ← 1,        k ← p
    while i ≤ m1 and j ≤ m2
        do if B[i] ≤ C[j]
            then A[k] ← B[i]
                 i ← i + 1
            else A[k] ← C[j]
                 j ← j + 1
           k ← k + 1
    if i ≤ m1
        then A[k, ..., r] ← B[i, ..., m1]
        else A[k, ..., r] ← C[j, ..., m2]
```

Analisi della complessità del MergeSort

Il costo dell'algoritmo mergeSort è espresso dalla seguente equazione di ricorrenza:

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

dove con $f(n)$ abbiamo indicato il costo della procedura di fusione

Poichè $f(n) = n = \Theta(n)$, per il teorema del Master abbiamo che

$$T(n) = \Theta(n \log_2 n)$$

QuickSort

Le fasi per ordinare una sequenza $A[p, ..., r]$ sono le seguenti:

- **Divide**: partiziona $A[p, ..., r]$ in due sottoarray $A[p, ..., q-1]$ e $A[q+1, ..., r]$ (anche vuoti) tali che

$$A[p, ..., q-1] \leq A[q] \leq A[q+1, ..., r]$$

  Determinare l'indice q è parte integrante di questa procedura di partizionamento (Partition)

- **Impera**: ordina $A[p, ..., q-1]$ e $A[q+1, ..., r]$ chiamando ricorsivamente la procedura QuickSort

- **Combina**: i sottoarry sono ordinati sul posto non occorre alcun lavoro per combinarli; $A[p, ..., r]$ è già ordinato

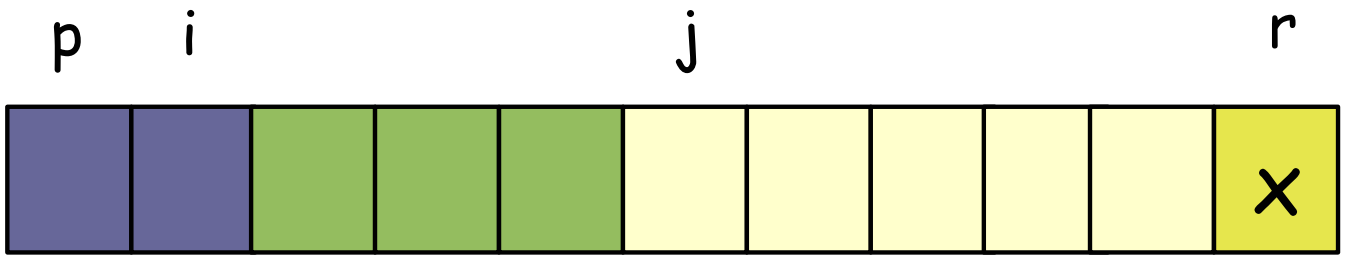
Algoritmo di ordinamento **QuickSort**

$$
\begin{aligned}
&\textsc{QuickSort}(A, p, r)\\
&\quad \textbf{if } p < r\\
&\quad\quad \textbf{then } q \leftarrow \textsc{Partition}(A, p, r)\\
&\quad\quad\quad \textsc{QuickSort}(A, p, q-1)\\
&\quad\quad\quad \textsc{QuickSort}(A, q+1, r)
\end{aligned}
$$

Per ordinare un array A, la chiamata iniziale è **Quicksort**$(A, 1, length[A])$

La procedura Partition(A, p, r)

Una generica iterazione di PARTITION(A, p, r) gestisce un'array così partizionato:



$p \le k \le i$ implica $A[k] \le x$

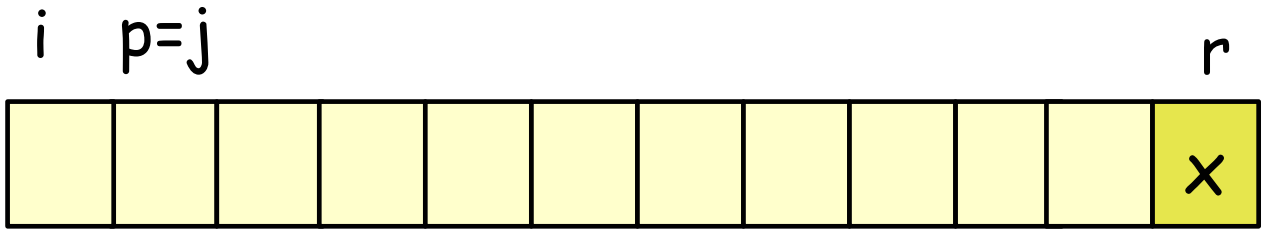
$i+1 \le k \le j-1$ implica $A[k] > x$

la porzione di vettore compresa tra **j** ed **r-1** non è stata ancora esaminata

x è l'elemento **pivot**

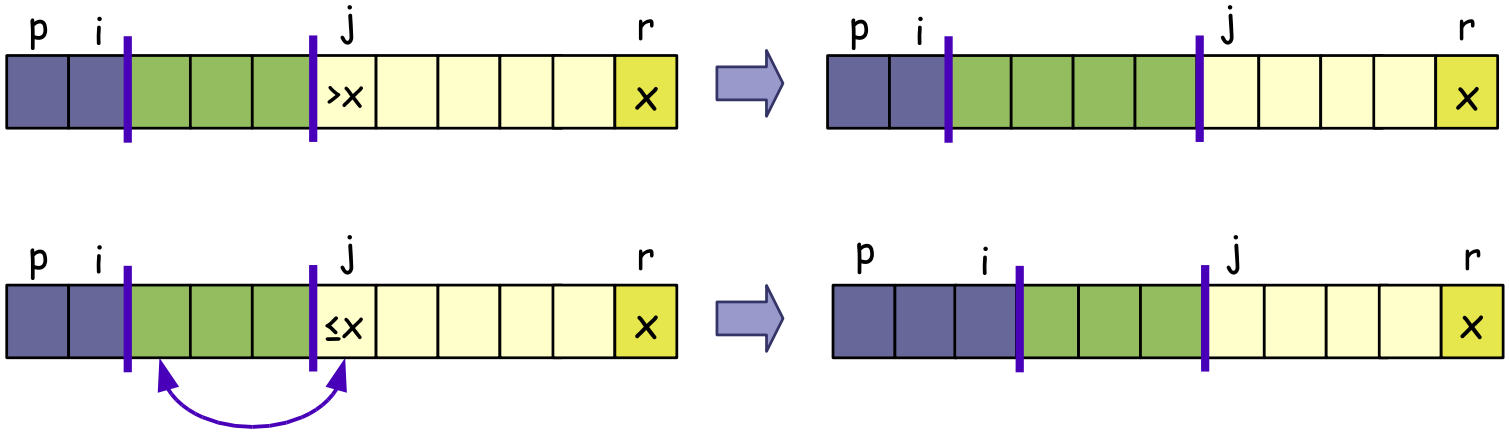
La procedura **Partition**(A, p, r)

Stato iniziale: nessun elemento della sequenza $A[p, ..., r-1]$ è stato esaminato. Quindi $j = p$ e $i = j - 1$



La procedura **Partition**(A, p, r)

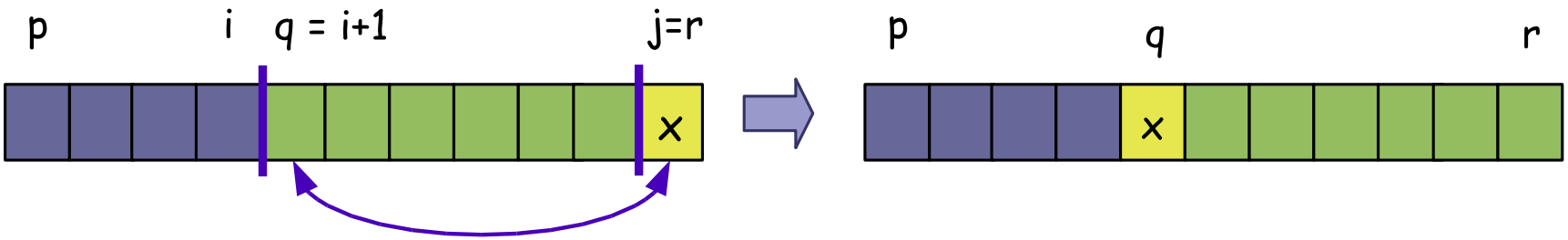
Generica iterazione: confronta $A[j]$ con x e modifica lo stato come segue



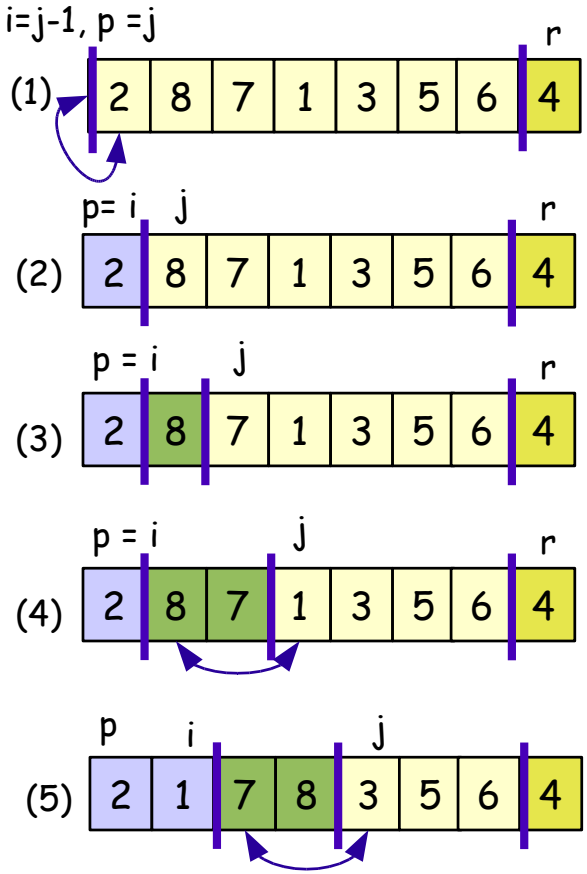
- $A[j] > x$: $A[j]$ è già nella porzione di vettore che gli compete
- $A[j] \leq x$: scambiamo $A[j]$ con $A[i+1]$; $A[i+1]$ è il primo elemento maggiore di x, $A[j]$ viene quindi spostato nella zona "blu"

La procedura **Partition**(A, p, r)

Stato finale:



La procedura **Partition**(A, i, $length[A]$)



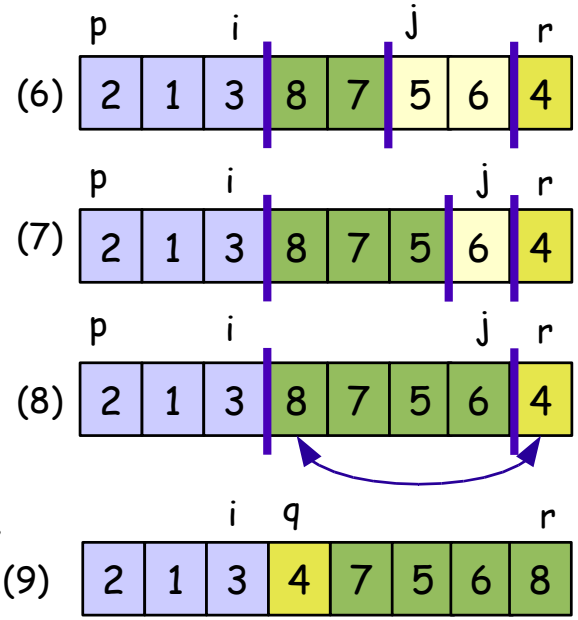
j=p, i=j-1 e A[j] < x
viene scambiato con $A[i+1] = A[j]$ ossia con se stesso

j=p+1, i=p e A[j] =8 > x
$j \leftarrow j+1$

j=p+2, i=p, A[j] =7 > x
$j \leftarrow j+1$

j=p+3, i=p, A[j] =1 ≤ x
$A[j] = 1 \leftrightarrow A[i+1] = 8$
$j \leftarrow j+1$, $i \leftarrow i+1$

j=p+4, i=p+1, A[j] =3 ≤ x
$A[j] = 3 \leftrightarrow A[i+1] = 7$
$j \leftarrow j+1$, $i \leftarrow i+1$



La procedura **Partition**

```
PARTITION(A, p, r)
    x ← A[r] ▷ elemento pivot
    i ← p − 1
    for j ← p to r − 1
        do if A[j] ≤ x
            then i ← i + 1
                 scambia A[i] ↔ A[j]
    scambia A[i + 1] ← A[r]
    return i + 1
```

Il tempo di esecuzione di PARTITION per un array di dimensione n è $\Theta(n)$

Prestazioni del **QuickSort**

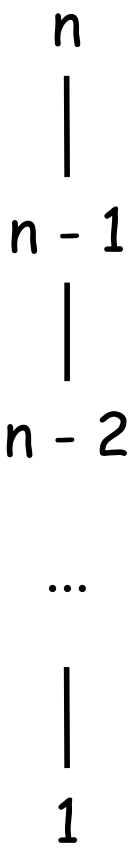
Il tempo di esecuzione del QuickSort dipende da come lavora la Partition ed, in particolare, da quanto è "sbilanciato" il partizionamento

Partizionamento sbilanciato: la procedura Partition produce due sottoproblemi di dimensione $n-1$ ed zero, rispettivamente

Caso peggiore: questo partizionamento sbilanciato si verifica ad ogni chiamata ricorsiva; la ricorrenza che definisce il tempo di esecuzione del QuickSort è:

$$\begin{aligned} T(n) &= \overbrace{T(n-1)+T(0)}^{\text{costo delle chiamate di QuickSort}} + \overbrace{\Theta(n)}^{\text{costo di Partition}} \\ &= T(n-1)+\Theta(n) \end{aligned}$$

Prestazioni del QuickSort nel caso peggiore



Sommando i costi ad ogni livello di ricorsione, otteniamo la serie aritmetica il cui valore è $\Theta(n^2)$

Basta applicare il metodo della sostituzione per dimostrare che la soluzione della ricorrenza $T(n) = T(n-1) + \Theta(n)$ è $T(n) = \Theta(n^2)$

Il tempo di esecuzione del QuickSort nel caso peggiore non è migliore di quello dell'InsertionSort

Inoltre, il tempo di esecuzione $\Theta(n^2)$ si ha quando l'array è già ordinato, nella stessa situazione l'InsertionSort richiede un tempo $O(n)$

Prestazioni del QuickSort nel caso migliore

Nel caso di bilanciamento massimo, la procedura Partition genera due sottoproblemi, ciascuno di dimensione $\sim n/2$ (uno con dimensione $\lfloor n/2 \rfloor$ e l'altro di dimensione $\lceil n/2 \rceil - 1$)

In questo caso QuickSort viene eseguito molto più velocemente; la ricorrenza per il tempo di esecuzione è

$$T(n) \leq 2T(n/2) + \Theta(n)$$

la cui soluzione, per il teorema del Master, è

$$T(n) = O(n \log_2 n)$$

Prestazioni del **QuickSort** nel caso medio

Nel caso medio il QUICKSORT si comporta come nel caso migliore $O(n \log_2 n)$

Per spiegare perchè, dobbiamo capire come il partizionamento influisce sul comportamento dell'algoritmo

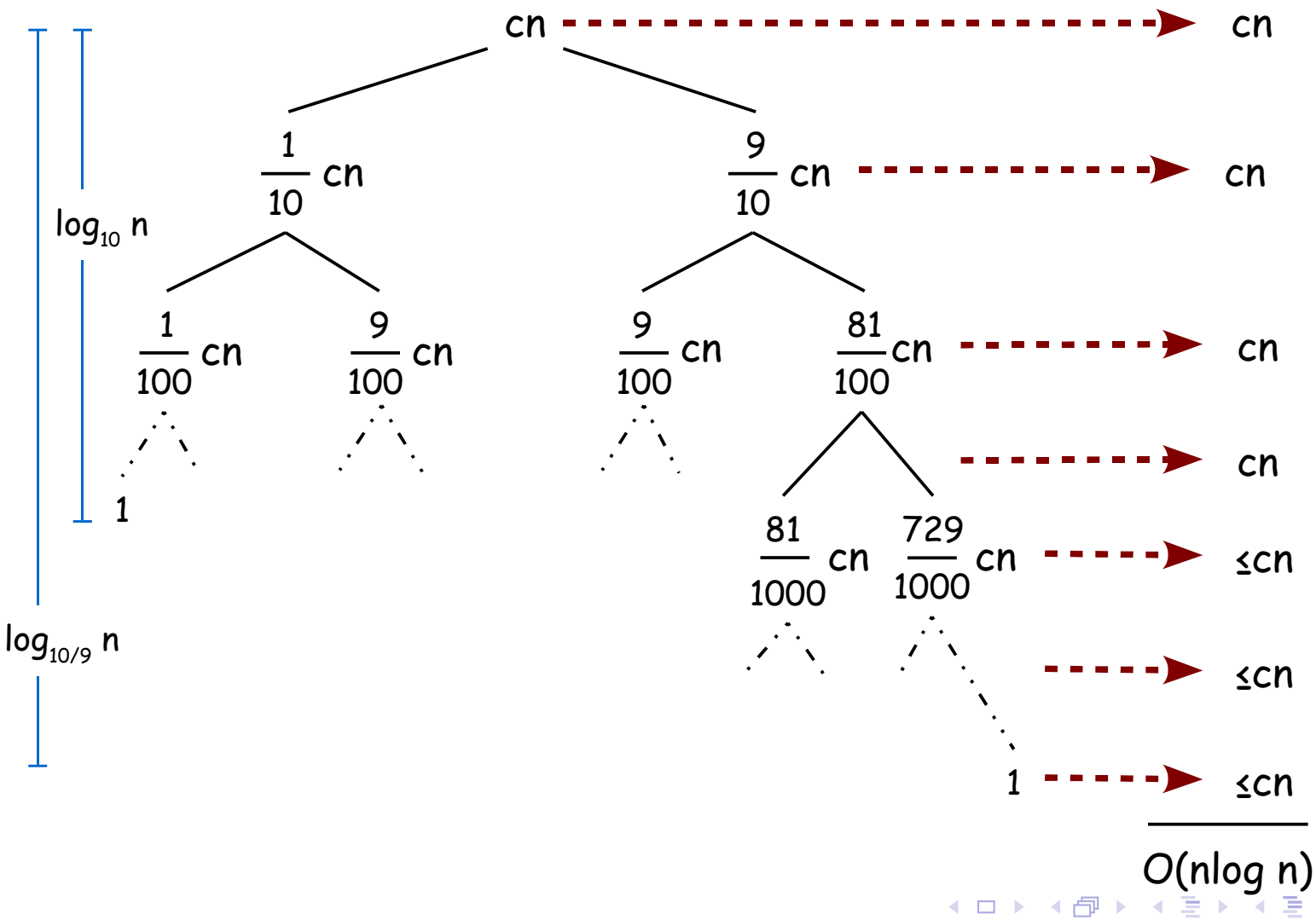
Supponiamo che PARTITION produca *sempre* una ripartizione, in apparenza molto sbilanciata, proporzionale 9-a-1

In questo caso otteniamo una ricorrenza

$$T(n) \leq T(9/10n) + T(1/10n) + cn$$

dove abbiamo incluso la costante c nascosta nel termine $\Theta(n)$

Un esempio


cn
cn
1/10 cn
9/10 cn
cn
log10 n
1/100 cn
9/100 cn
9/100 cn
81/100 cn
cn
cn
1
81/1000 cn
729/1000 cn
≤cn
≤cn
log10/9 n
1
≤cn
O(nlog n)

Un esempio

Sommando i costi di ciascun livello abbiamo che $T(n) \leq cn(h+1)$ dove $h = \log_{10/9} n$ è l'altezza dell'albero. Allora

$$T(n) = O(n \log_{10/9} n)$$

Inoltre, $\log_{10/9} n = \log_{10/9} 2 \log_2 n$ (regola del cambiamento di base dei logaritmi) e quindi

$$T(n) = O(n \log_2 n)$$

Anche una ripartizione 99-a-1 determina un tempo di esecuzione pari a $O(n \log_2 n)$

La ragione: una qualsiasi ripartizione con proporzionalità *costante* produce un albero di ricorsione di profondità $\Theta(\log_2 n)$; il costo di ciascun livello è $O(n)$. Quindi, il tempo di esecuzione è $O(n \log_2 n)$

Alcune intuizioni sul caso medio

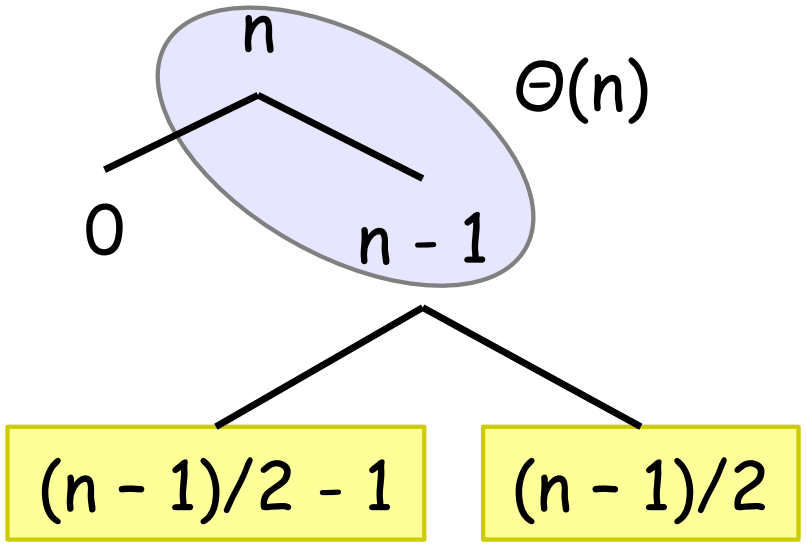
Se eseguiamo QUICKSORT su un input casuale, è poco probabile che il partizionamento avvenga sempre nello stesso modo ad ogni livello

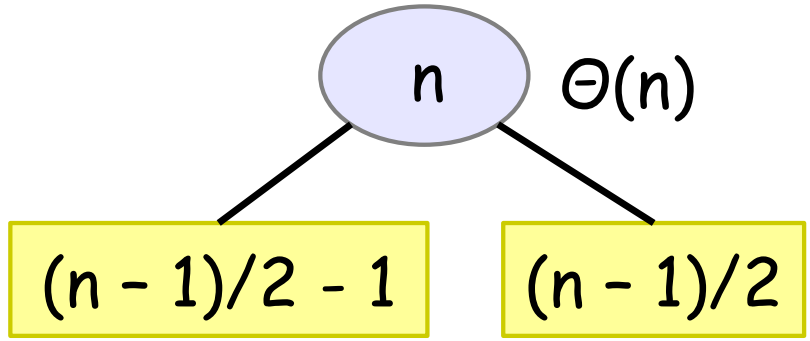
È logico supporre che qualche ripartizione sarè ben bilanciata e qualche altra sarà molto sbilanciata

Nel caso medio PARTITION produce una combinazione di ripartizioni "buone" e di ripartizioni "cattive" distribuite a caso nell'albero di ricorsione

Assumiamo che le ripartizioni buone e cattive si alternino nei vari livelli dell'albero

Alcune intuizioni sul caso medio





Limite inferiore al PROBLEMA dell'ordinamento

Gli algoritmi di ordinamento visti finora condividono una proprietà: **effettuano l'ordinamento basandosi solo su confronti tra gli elementi in input**

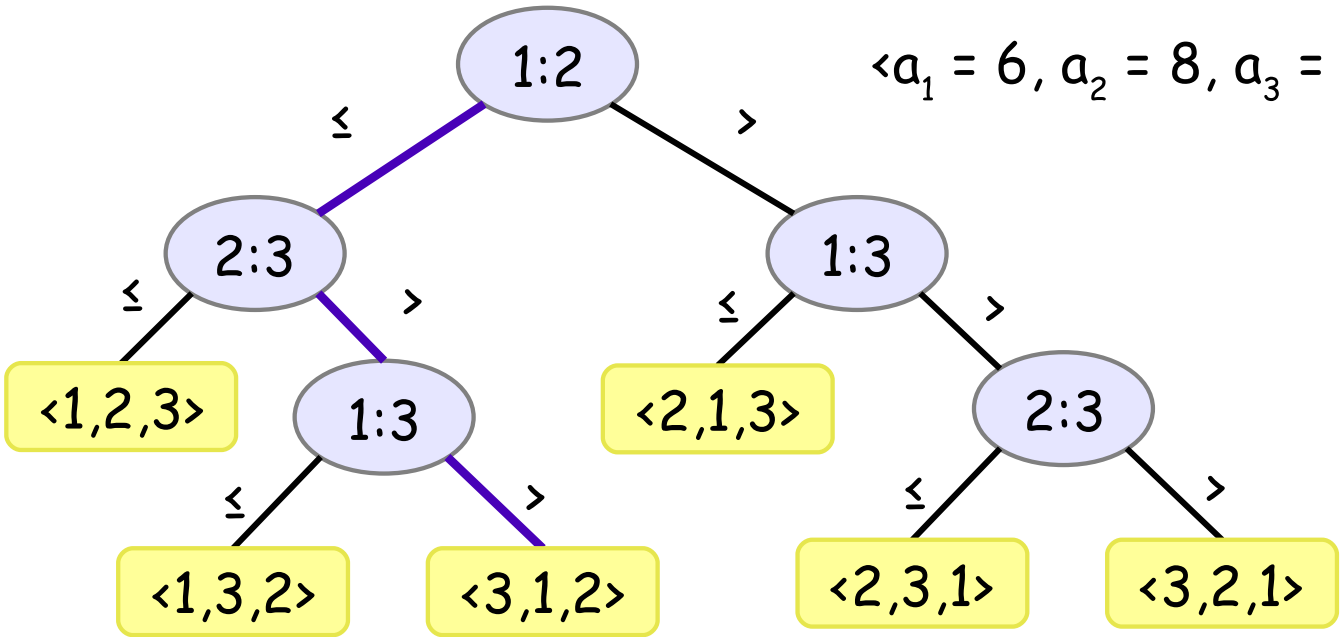
Gli algoritmi migliori ci consentono di ordinare una data sequenza in input in un tempo $O(n \log_2 n)$ (heap-sort e merge-sort anche nel caso peggiore, il quick-sort solo nel caso medio)

In realtà, un qualsiasi ordinamento per confronti deve effettuare almeno $\Omega(n \log_2 n)$ confronti nel caso peggiore per ordinare n elementi

Heap-sort e merge-sort sono asintoticamente ottimali

Un modello astratto per gli ordinamenti per confronti

Gli ordinamenti per confronti possono essere visti astrattamente in termini di **alberi di decisione**



albero di decisione per un insieme di tre elementi

Alberi di decisione

Un albero di decisione è un albero binario usato per rappresentare la sequenza di confronti che vengono effettuati da un ordinamento per confronti

Ogni nodo interno:

- è etichettato con $i : j$ dove $1 \leq i, j \leq n$
- corrisponde ad eseguire il confronto tra a_i ed a_j;
- il sottoalbero sinistro (destro) definisce i successivi confronti nel caso in cui $a_i \leq a_j$ ($a_i > a_j$, rispettivamente)

Ogni foglia corrisponde ad una permutazione $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$

Quando si raggiunge una foglia, l'algoritmo di ordinamento ha stabilito l'ordinamento $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

Alberi di decisione

Qualsiasi algoritmo di ordinamento corretto deve essere in grado di produrre ogni permutazione del suo input

Quindi, ciascuna delle $n!$ permutazioni di n elementi deve essere raggiungibile dalla radice lungo un dato percorso

Un qualsiasi percorso dalla radice fino ad una foglia corrisponde all'effettiva esecuzione di un algoritmo per confronti

Alberi di decisione

Teorema: un qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \log_2 n)$ confronti nel caso peggiore

Dimostrazione: è sufficiente determinare l'altezza di un albero di decisione dove ogni permutazione appare come foglia raggiungibile

Sia T un albero di decisione di altezza h con $n!$ foglie; $n! \leq 2^h$ dove 2^h è il numero di foglie di un albero binario completo di altezza h

Approssimazione di Stirling: $(n/e)^n \leq n! \leq n(n/e)^n$. Quindi:

$$h \geq \log_2 n! \geq \log_2 (n/e)^n = n \log_2 (n/e) = \Theta(n \log_2 n)$$

Possiamo quindi concludere che $h = \Omega(n \log_2 n)$

Parte I

Ordinamento in tempo lineare

Counting Sort

È un algoritmo in grado di ordinare un vettore di n elementi in tempo **lineare**

Come è possibile? Non opera **esclusivamente** sul confronto di elementi, ma fa delle ipotesi aggiuntive

Il CountingSort assume che gli n elementi da ordinare siano interi compresi tra 0 e k

Se $k = O(n)$, l'ordinamento viene effettuato in un tempo $\Theta(n)$

Counting sort

Idea di base: determinare, per ogni elemento x in input, il numero di elementi minori di x (se ci sono 13 elementi minori di x allora x deve andare in posizione 14)

Questo schema va leggermente modificato per gestire la presenza di più elementi con lo stesso valore

Counting Sort

(a)

	1	2	3	4	5	6
A	2	5	3	0	2	1

	0	1	2	3	4	5
C	1	1	2	1	0	1

C[i] = # di occorrenze di i

(b)

	0	1	2	3	4	5
C	1	2	4	5	5	6

C[i] = # di elementi ≤ i

(c)

	1	2	3	4	5	6
B						

	0	1	2	3	4	5
C	1	2	4	5	5	6

A[6] = 1, C[1] = 2

(d)

	1	2	3	4	5	6
B		1				

	0	1	2	3	4	5
C	1	1	4	5	5	6

A[5] = 2, C[2] = 4

(e)

	1	2	3	4	5	6
B		1		2		

	0	1	2	3	4	5
C	1	1	3	5	5	6

A[4] =0, C[0] = 1

(f)

	1	2	3	4	5	6
B	0	1		2		

	0	1	2	3	4	5
C	0	1	3	5	5	6

Counting Sort

(a)

	1	2	3	4	5	6
A	2	5	3	0	2	1

	0	1	2	3	4	5
C	1	1	2	1	0	1

C[i] = numero di elementi uguali a i

(b)

	0	1	2	3	4	5
C	1	2	4	5	5	6

C[i] = numero di elementi ≤ a i

(f)

	1	2	3	4	5	6
B	0	1		2		

	0	1	2	3	4	5
C	**0**	1	3	5	5	6

A[3] =3, C[3] = 5

(g)

	1	2	3	4	5	6
B	0	1		2	3	

	0	1	2	3	4	5
C	0	1	3	**4**	5	6

A[2] =5, C[5] = 6

(h)

	1	2	3	4	5	6
B	0	1		2	3	5

	0	1	2	3	4	5
C	0	1	3	4	5	**5**

A[1] =2, C[2] = 3

(i)

	1	2	3	4	5	6
B	0	1	2	2	3	5

	0	1	2	3	4	5
C	0	1	**2**	4	5	5

Counting Sort

CountingSort(A, B, k)

1. **for** $i \leftarrow 0$ **to** k **do** $C[i] \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** $length[A]$ **do** $C[A[j]] \leftarrow C[A[j]] + 1$
   ▷ $C[i]$ contiene il numero di elementi $= i$
3. **for** $i \leftarrow 1$ **to** k **do** $C[i] \leftarrow C[i] + C[i-1]$
   ▷ $C[i]$ contiene il numero di elementi $\leq i$
4. **for** $j \leftarrow length[A]$ **downto** 1
5. **do** $B[C[A[j]]] \leftarrow A[j]$ ▷ $A[j]$ è il valore, $C[A[j]]$ la posizione
6. $C[A[j]] \leftarrow C[A[j]] - 1$

I cicli for di riga 1 e 4 costano $\Theta(k)$ e quelli di riga 2 e 6 costano $\Theta(n)$. Il costo complessivo è $\Theta(n+k)$. Se, $k = O(n)$ allora $\Theta(n+k) = \Theta(n)$