

Algoritmi e Strutture Dati

Algoritmi di Ordinamento

Maria Rita Di Berardini, Emanuela Merelli¹

¹Dipartimento 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 sé 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 $\{a_1, a_2, \dots, a_n\}$, trovare un'opportuna permutazione $\{a'_1, a'_2, \dots, a'_n\}$ tale che $a'_1 \leq a'_2 \leq \dots \leq a'_n$

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

Output: $\{a'_1, a'_2, \dots, a'_n\}$ oppure

$\{a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}\}$

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

Il problema dell'ordinamento

Più in generale, sia $A = \{a_1, a_2, \dots, a_n\}$ un insieme di n elementi qualsiasi su cui è definita una **relazione d'ordine totale** (ossia una relazione **riflessiva, antisimmetrica e transitiva** che chiamiamo \leq), trovare una opportuna permutazione $\{a'_1, a'_2, \dots, a'_n\}$ tale che

$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

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

I primi tre – **SelectionSort**, **InsertionSort** e **BubbleSort** – sono estremamente elementari e consentono un'implementazione assai semplice; il prezzo da pagare alla semplicità di questi algoritmi è la complessità computazionale che è $O(n^2)$ (complessità espressa in termini del numero di confronti)

L'algoritmo **QuickSort** ci consente di raggiungere una complessità di $O(n \log_2 n)$ solo nel caso medio, mentre nel caso più sfavorevole ritorna ad una complessità di $O(n^2)$

Il problema dell'ordinamento

Gli algoritmi **HeapSort** e **MergeSort** hanno una complessità di $O(n \log_2 n)$ anche nel caso peggiore

È possibile dimostrare che il limite inferiore alla complessità del **problema dell'ordinamento** mediante confronti (e quindi senza sfruttare informazioni sull'insieme da ordinare) è proprio $n \log_2 n$

Possiamo concludere che gli algoritmi **HeapSort** e **MergeSort** sono **ottimali**

Soluzioni più efficienti si basano su assunzioni aggiuntive come ad es. la presenza di elementi duplicati, il valore massimo e minimo all'interno dell'insieme o altre informazioni che consentono di introdurre delle ottimizzazioni (CountingSort, BucketSort)

Selection sort

È un algoritmo molto intuitivo ed estremamente 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 sofisticata la codifica del programma che lo implementa

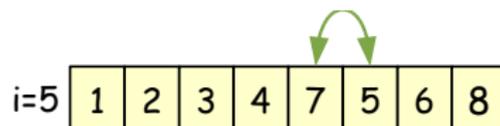
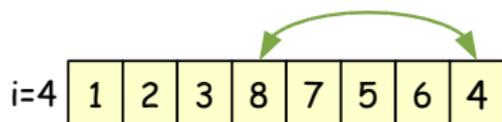
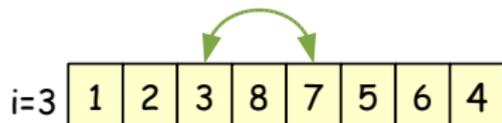
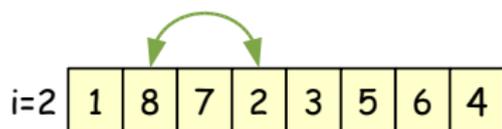
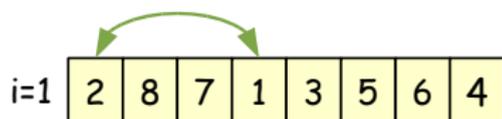
Idea: ripetere $n - 1$ volte una procedura in grado di selezionare alla i -esima iterazione i -esimo elemento più piccolo dell'insieme e scambiarlo con l'elemento che di trova in posizione i

SelectionSort(A)

SelectionSort(A)

1. **for** $i \leftarrow 1$ to $length[A] - 1$
2. **do** $min \leftarrow i$
3. **for** $j \leftarrow i + 1$ to $length[A]$
4. **do if** $A[j] < A[min]$ **then** $min \leftarrow j$
5. scambia $A[min] \leftrightarrow A[i]$

SelectionSort(A) – un esempio



SelectionSort(A)

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 4) esegue esattamente $n - i$ confronti. Il numero totale di confronti è :

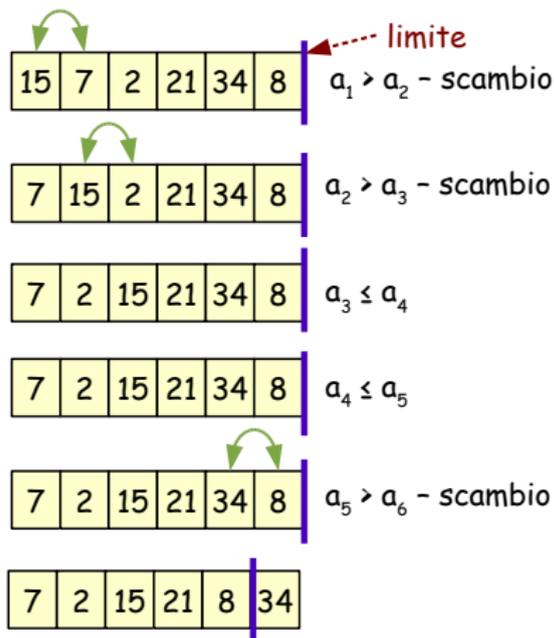
$$\sum_{i=1}^{n-1} n - i = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

Bubble sort

Idea: far risalire gli elementi più grandi nelle posizioni di indice più alto e – nel contempo – far ridiscendere gli elementi più piccoli verso le posizioni di indice più basso

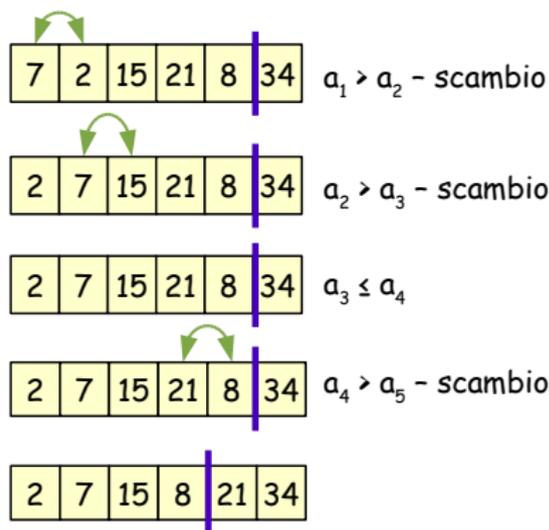
Strategia: scorrere più volte la sequenza in input confrontando, ad ogni passo, l'ordinamento reciproco di elementi contigui e scambiando le posizioni di eventuali coppie non ordinate

Bubble sort



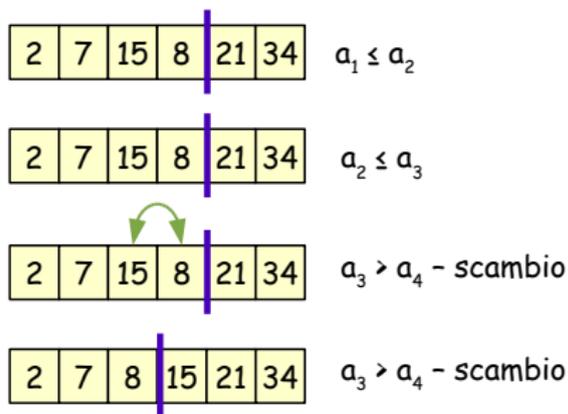
Dopo la prima scansione l'elemento massimo (34) è nella posizione corretta (l'ultima); le scansioni successive potranno non considerare l'ultima posizione

Bubble sort



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

Bubble sort



A questo punto il vettore è ordinato; una (eventuale) scansione successiva non effettuerebbe alcun scambio

BubbleSort(A) – prima versione

BubbleSort(A)

1. **for** $j \leftarrow \text{length}[A]$ 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 di **BubbleSort** è corretta ma, il numero di confronti eseguiti dall'algoritmo è lo stesso per ogni possibile configurazione dell'input

Analisi di BubbleSort(A) – prima versione

Caso migliore – vettore ordinato: fissato $j = n \dots, 2$, il ciclo più interno (riga 4) esegue $j - 1$ confronti e 0 scambi. Il numero totale di scambi è 0; il numero totale di confronti è:

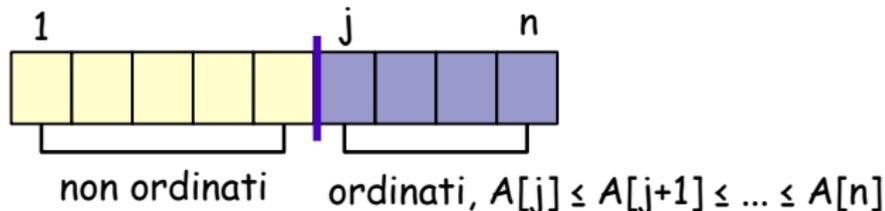
$$\sum_{j=2}^n j - 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

Caso peggiore – vettore ordinato in maniera decrescente: fissato $j = n, \dots, 2$, il ciclo più interno (riga 4) esegue $j - 1$ confronti e $j - 1$ scambi. Il numero totale di confronti (e di scambi) è:

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

Bubble sort

Esaminiamo lo stato del vettore durante un generica iterazione del ciclo più esterno (riga 2). Fissiamo j abbiamo che:



Assumiamo ora che durante l'esecuzione del ciclo più interno (riga 4) non viene effettuato alcuno scambio, e quindi che $A[1] \leq A[2] \leq \dots \leq A[j-1] \leq A[j]$. Allora il vettore è già ordinato

Possiamo usare questa informazione per (provare a) ridurre il numero di confronti necessari per ordinare il vettore

BubbleSort(A)

BubbleSort(A)

1. **for** $j \leftarrow \text{length}[A]$ downto 2 ▷ j è il limite
2. $\text{scambi} \leftarrow 0$
3. **for** $i \leftarrow 1$ to $j - 1$
4. **do if** $A[i] > A[i + 1]$
5. **then** scambia $A[i] \leftrightarrow A[i + 1]$
6. $\text{scambi} \leftrightarrow \text{scambi} + 1$
7. **if** $\text{scambi} = 0$ **then return**

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

Analisi di BubbleSort(A)

Caso migliore: durante la prima 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 $j = n, \dots, 2$, il ciclo più interno (riga 4) esegue $j - 1$ confronti e $j - 1$ scambi. Ogni iterazione del ciclo più interno esegue almeno uno scambio e, quindi, il numero totale di confronti (e di scambi) è (esattamente come nell'altro caso):

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

Analisi di BubbleSort(A)

L'analisi nel **caso medio** del bubblesort è abbastanza complessa; si basa sul numero di **inversioni** presenti nel vettore; il numero di inversioni del vettore A è dato da:

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

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 alquanto più complessa (per la quale rinviamo a Knuth) dà un valore

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

per il numero di confronti nel caso medio

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

Poi, 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 si 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 in due sottosequenze di $n/2$ 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

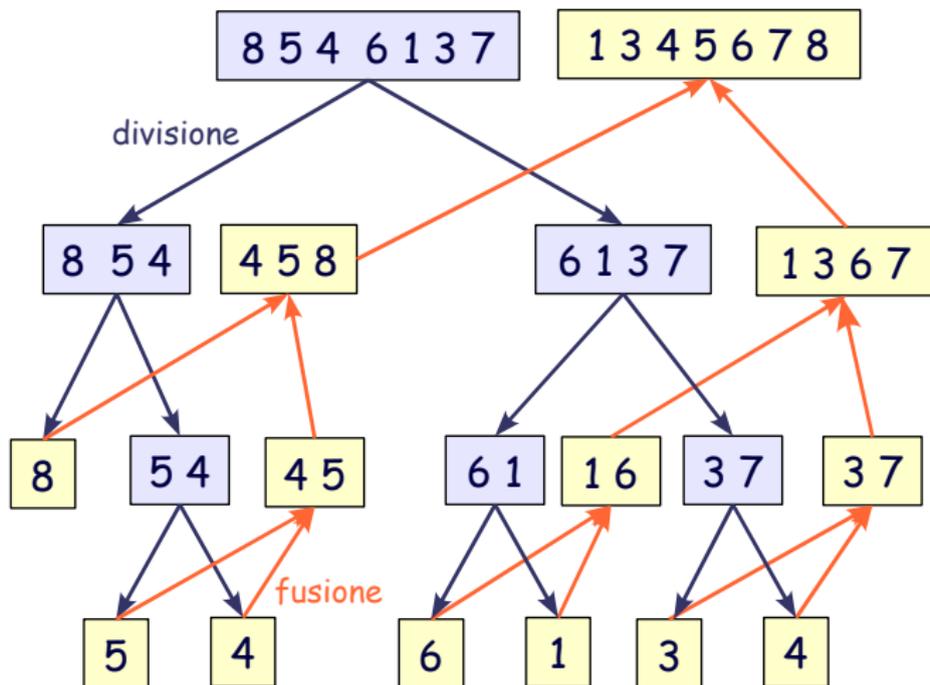
Il passo combina fonde le due sequenze utilizzando una procedura ausiliaria **merge**(A, p, q, r) (fusione), dove: A è un array e p, q, r sono indici di elementi dell'array tali che $p < q < r$

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

```
MergeSort(A, left, right)  
  if left < right  
    then mid =  $\lfloor (left + right) / 2 \rfloor$   
          MergeSort(A, left, mid)  
          MergeSort(A, mid + 1, right)  
          Merge(A, left, mid, right)
```

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



Fusione di due sottosequenze ordinate

```
Merge(A, left, mid, right)  
   $m_1 \leftarrow mid - left + 1$  ▷ dim di  $A[left..mid]$   
   $m_2 \leftarrow right - mid$  ▷ dim di  $A[mid + 1..right]$   
   $B[1, \dots, m_1] \leftarrow A[left..mid]$   
   $C[1, \dots, m_2] \leftarrow A[mid + 1..right]$   
   $i, j \leftarrow 1, \quad k \leftarrow left$   
  while  $i \leq m_1$  and  $j \leq m_2$   
    do if  $B[i] \leq C[j]$   
      then  $A[k] \leftarrow B[i]$   
         $i \leftarrow i + 1$   
      else  $A[k] \leftarrow C[j]$   
         $j \leftarrow j + 1$   
     $k \leftarrow k + 1$   
  if  $i \leq m_1$   
    then  $A[k, \dots, right] \leftarrow B[i..m_1]$   
  else  $A[k, \dots, right] \leftarrow C[j..m_2]$ 
```

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 $f(n) = O(n)$, per il teorema del Master abbiamo che

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

QuickSort

Altro algoritmo di ordinamento basato sul divide-et-impera; fasi per ordinare una sequenza $A[p..r]$

- **divide**: partizionare la sequenza $A[p..r]$ in due sottoarray $A[p..q - 1]$ e $A[q + 1..r]$ (eventualmente vuoti) tali che ogni elemento di $A[p..q - 1]$ sia minore o uguale a $A[q]$ che a sua volta è minore o uguale di ogni elemento in $A[q + 1..r]$. Calcolare l'indice q è parte di questa procedura di partizionamento
- **impera**: ordina $A[p..q - 1]$ e $A[q + 1..r]$ chiamando ricorsivamente il quickSort
- **combina**: i sottoarray sono ordinati sul posto non occorre alcun lavoro per combinarli; $A[p..r]$ è già ordinato

Algoritmo di ordinamento QuickSort

```
QuickSort( $A, p, r$ )  
  if  $p < r$   
    then  $q \leftarrow$  Partition( $A, p, r$ )  
         QuickSort( $A, p, q - 1$ )  
         QuickSort( $A, q + 1, r$ )
```

Per ordinare un array A , la chiamata iniziale è **QuickSort**($A, 1, \text{length}[A]$)

La procedura Partition

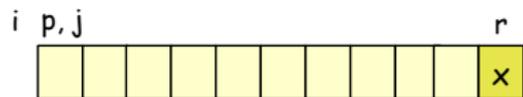


$$p \leq k \leq i \Rightarrow A[k] \leq x,$$

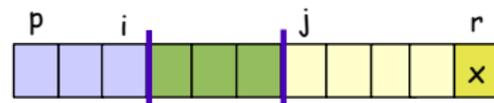
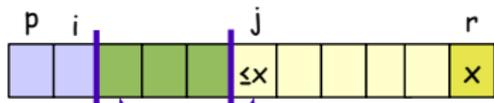
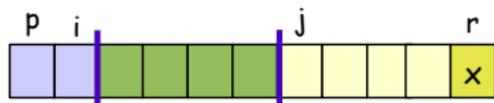
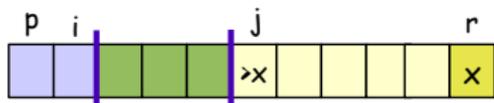
$$i+1 \leq k \leq j-1 \Rightarrow A[k] > x,$$

$$j \leq k \leq r-1 \Rightarrow A[k] \text{ senza limitazioni}$$

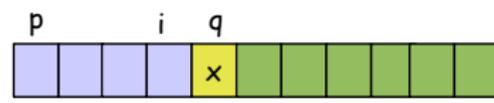
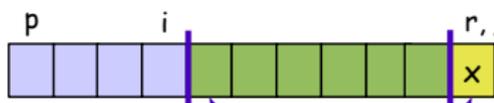
$$A[r] = x \text{ (pivot)}$$



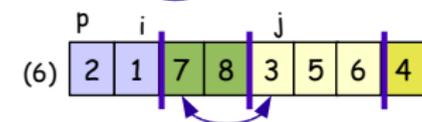
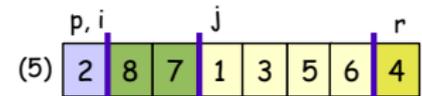
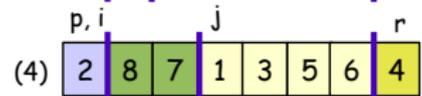
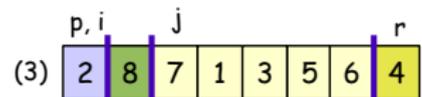
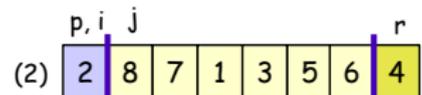
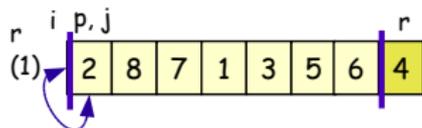
$$j=p, i=j-1=p-1$$



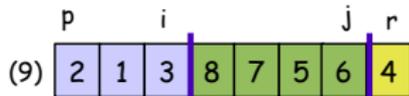
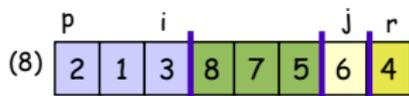
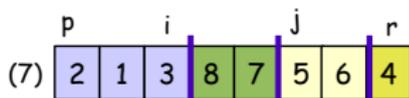
$A[j]$ viene scambiato con $A[i+1]$



La procedura Partition



$p \leq k \leq i \Rightarrow A[k] \leq x$,
 $i+1 \leq k \leq j-1 \Rightarrow A[k] > x$,
 $j \leq k \leq r-1 \Rightarrow A[k]$ senza limitazioni
 $A[r] = x$ (pivot)



La procedura **Partition**

```
Partition( $A, p, r$ )  
   $x \leftarrow A[r]$  ▷ elemento pivot  
   $i \leftarrow p - 1$  ▷ dim di  $A[mid + 1..right]$   
  for  $j \leftarrow p$  to  $r - 1$   
    do if  $A[j] \leq x$   
      then  $i \leftarrow i + 1$   
           scambia  $A[i] \leftrightarrow A[j]$   
  scambia  $A[i + 1] \leftarrow A[r]$   
  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 dal fatto che il partizionamento sia bilanciato o sbilanciato

Partizionamento sbilanciato: si verifica quando **Partition** produce due sottoproblemi con $n - 1$ ed zero elementi, 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 $\Theta(n^2)$

n
|
n - 1
|
n - 2
|
...
|
1

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 non maggiore di $n/2$ (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

Il tempo del **QuickSort** nel caso medio è molto più vicino al caso migliore che non al caso peggiore

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

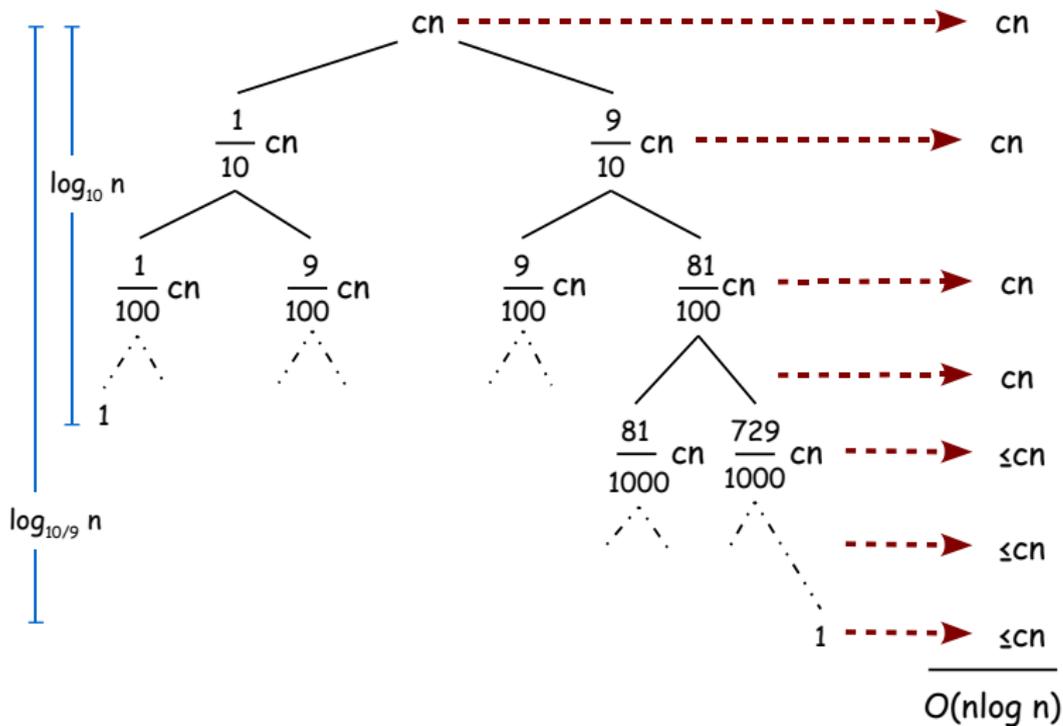
Supponiamo che **Partition** produca *sempre* una ripartizione proporzionale 9-a-1 (in apparenza molto sbilanciata)

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



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

$$T(n) = O(n \log_{10/9} n) = O(n \log_2 n)$$

In effetti 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

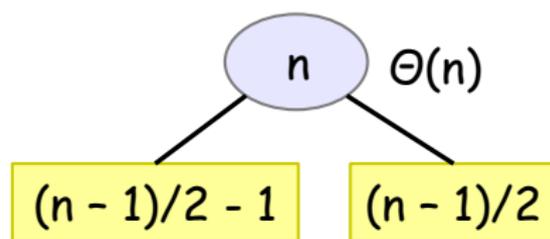
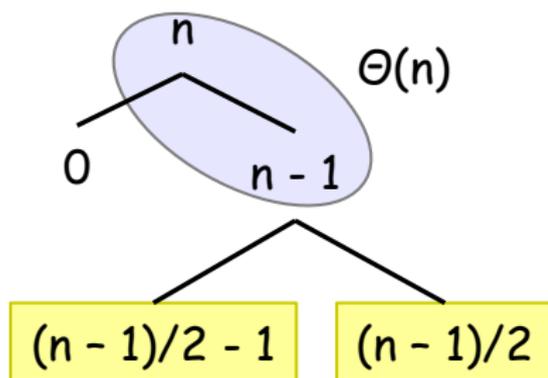
Se eseguiamo **QuickSort** su un input casuale, è poco probabile che il partizionamento avvenga sempre nello stesso modo ad ogni livello

È logico supporre, invece, che qualche ripartizione sarà 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

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

Alcune intuizioni sul caso medio



Parte I

Ordinamento in tempo lineare

Limite inferiore al PROBLEMA dell'ordinamento

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

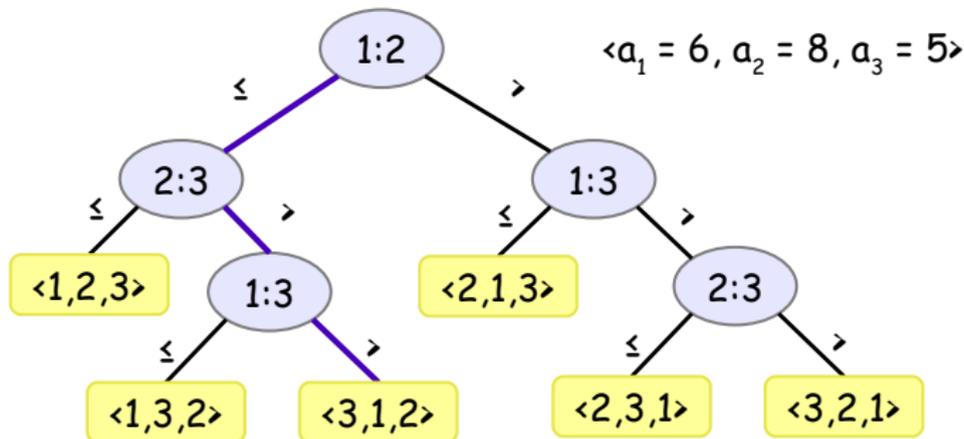
Gli algoritmi migliori ci consentono di ordinare una data sequenza in input in un tempo $O(n \log_2 n)$ (heap-sort e merge-sort anche nel caso peggiore, il quick-sort solo nel caso medio)

In realtà, un qualsiasi ordinamento per confronti deve effettuare $\Omega(n \log_2 n)$ 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 completo che rappresenta i confronti che vengono effettuati da un ordinamento per confronti

Ogni nodo interno:

- è rappresentato da $i:j$ per qualche i e j nell'intervallo $1 \leq i, j \leq n$, dove n è il numero di elementi da ordinare a_i e a_j
- corrisponde ad eseguire il confronto tra a_i ed a_j ; il sottoalbero sinistro (destra) detta i successivi confronti nel caso in cui $a_i \leq a_j$ ($a_i > a_j$, rispettivamente)

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

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

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

Consideriamo un albero di decisione di altezza h e con $n!$ foglie

Poichè l'albero di decisione è un albero binario completo, abbiamo che $n! = 2^h$, ossia numero di foglie di un albero binari completo di altezza h , e quindi

$$h = \log_2 n! \geq \Omega(n \log_2 n)$$

Counting sort

È un esempio di 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 Counting sort assume che **ciascuno degli n elementi da ordinare sia un intero compreso tra 0 e k**

Quando $k = O(n)$, 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 nella posizione 14)

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

Counting Sort

(a) A

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

C

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

$C[i]$ = numero di
elementi uguali a i

(b)

C

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

$C[i]$ = numero di
elementi $\leq i$

(c)

B

1	2	3	4	5	6

C

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

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

(d)

B

1	2	3	4	5	6
	1				

C

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

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

(e)

B

1	2	3	4	5	6
	1		2		

C

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

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

(f)

B

1	2	3	4	5	6
0	1		2		

C

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

Counting Sort

(a) A

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

C

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

$C[i]$ = numero di
elementi uguali a i

(b)

C

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

$C[i]$ = numero di
elementi $\leq a_i$

(f) B

1	2	3	4	5	6
0	1		2		

C

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

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

(g) B

1	2	3	4	5	6
0	1		2	3	

C

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

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

(h) B

1	2	3	4	5	6
0	1		2	3	5

C

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

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

(i) B

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

C

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

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 $\Theta(n + k) = \Theta(n)$