Università di Roma
Tor Vergata

Il problema dell'ordinamento

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Il problema dell'ordinamento

Problema dell'ordinamento

- **Input**: una sequenza $A=a_1, a_2, a_3, \ldots, a_n$ di n valori
- **Output**: una sequenza $B=b_1, b_2, b_3, \ldots, b_n$ che sia una permutazione di A e tale per cui $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$

- Esiste una grandissima quantità di algoritmi che permettono di risolvere questo problema
- Spesso, questi algoritmi si comportano diversamente tra loro, in funzione dell'input
- Capire la complessità computazionale di questi algoritmi consente di scegliere l'algoritmo migliore

Quali analisi svolgere per gli algoritmi

- Analisi del caso pessimo
  - È tipicamente la più importante, perché fornisce un limite superiore al tempo di esecuzione per qualsisi input

- Analisi del caso medio
  - In molti casi non è semplice: che cos'è il caso medio?
  - Si utilizza tipicamente una distribuzione uniforme dei dati

- Analisi del caso ottimo
  - Se si conoscono informazioni particolari sull'input, può dare delle indicazioni aggiuntive

Alcuni algoritmi di ordinamento

- Stupid sort
- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Bucket Sort
- Quick sort
- Radix sort
- …

Stupid sort

- Utilizza un approccio totalmente privo di senno

- Prende la sequenza A
- Verifica se la sequenza è ordinata
  - In caso contrario genera una permutazione casuale della sequenza A
- Ripete il controllo

```
STUPIDSORT(A):
  while not sorted(A):
    A ← random_permutation(A)
```

Stupid sort

- Utilizza un approccio totalmente privo di senno

- Prende la sequenza A
- Verifica se la sequenza è ordinata
  - In caso contrario genera una permutazione casuale della sequenza A
- Ripete il controllo

```
STUPIDSORT(A):
  while not sorted(A):
    A ← random_permutation(A)
```

```
BOZOSORT(A):
  while not sorted(A):
    A ← invert_two_elements(A)
```

Stupid sort: analisi

- Analisi del caso pessimo
  - $T(n) = \infty$

- Analisi del caso medio
  - $T(n) = O(n \cdot n!)$

- Analisi del caso ottimo
  - $T(n) = O(n)$

Approccio naïf: Selection Sort

- Ispirato a come un umano metterebbe in ordine una sequenza di numeri

- Cerco il valore minimo nella sequenza
- Lo metto al posto giusto
- Rimangono da ordinare i restanti $n - 1$ elementi
- Posso applicare la stessa strategia

Selection Sort

```
SELECTIONSORT(A, n):
  for i ← 0 to n - 1
    min ← i
    for j ← (i + 1) to n - 1
      if A[j] < A[min] then
        min ← j
    if min ≠ i then
      tmp ← A[i]
      A[i] ← A[min]
      A[min] ← tmp
```

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=1$	7	4	2	1	8	3	5
$i=2$	1	4	2	7	8	3	5
$i=3$	1	2	4	7	8	3	5
$i=4$	1	2	3	7	8	4	5
$i=5$	1	2	3	4	8	7	5
$i=6$	1	2	3	4	5	7	8
$i=7$	1	2	3	4	5	7	8

Selection Sort: analisi

```
SELECTIONSORT(A, n):
  for i ← 0 to n - 1
    min ← i
    for j ← (i + 1) to n - 1
      if A[j] < A[min] then
        min ← j
    if min ≠ i then
      tmp ← A[i]
      A[i] ← A[min]
      A[min] ← tmp
```

- Per calcolare la complessità nel caso medio, pessimo ed ottimo, ragiono sull'operazione dominante.

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

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

Bubble Sort

- Ordina gli elementi facendo “salire” come bolle gli elementi più piccoli, mentre quelli “più pesanti” scendono verso il basso
- Compara ogni elemento adiacente e li inverte di posizione se sono nell'ordine sbagliato
- Alcuni elementi si spostano verso la posizione corretta più in fretta di altri

Bubble Sort

```
BUBBLESORT(A, n):
  scambio ← true
  while scambio do
    scambio ← false
    for i ← 0 to n-1  do
      if A[i] > A[i+1] then
        swap(A[i], A[i+1])
        scambio ← true
```

- L'operazione dominante è il confronto nel ciclo più interno.
- Vengono effettuati $\frac{n^2}{2}$ confronti e scambi sia in media che nel caso pessimo: $\Theta(n^2)$

6 5 3 1 8 7 2 4

Insertion Sort

- Ordina in modo non decrescente
- Inserisce l'elemento A[i] nella posizione corretta nel vettore ordinato A[0, ..., i-1]

```
INSERTIONSORT(A, n):
    for i ← 1 to n:
        key ← A[i]
        j ← i - 1
        while j ≥ 1 and A[j] > key:
            A[j + 1] ← A[j]
            j ← j - 1
        A[j + 1] ← key
```

Complessità:

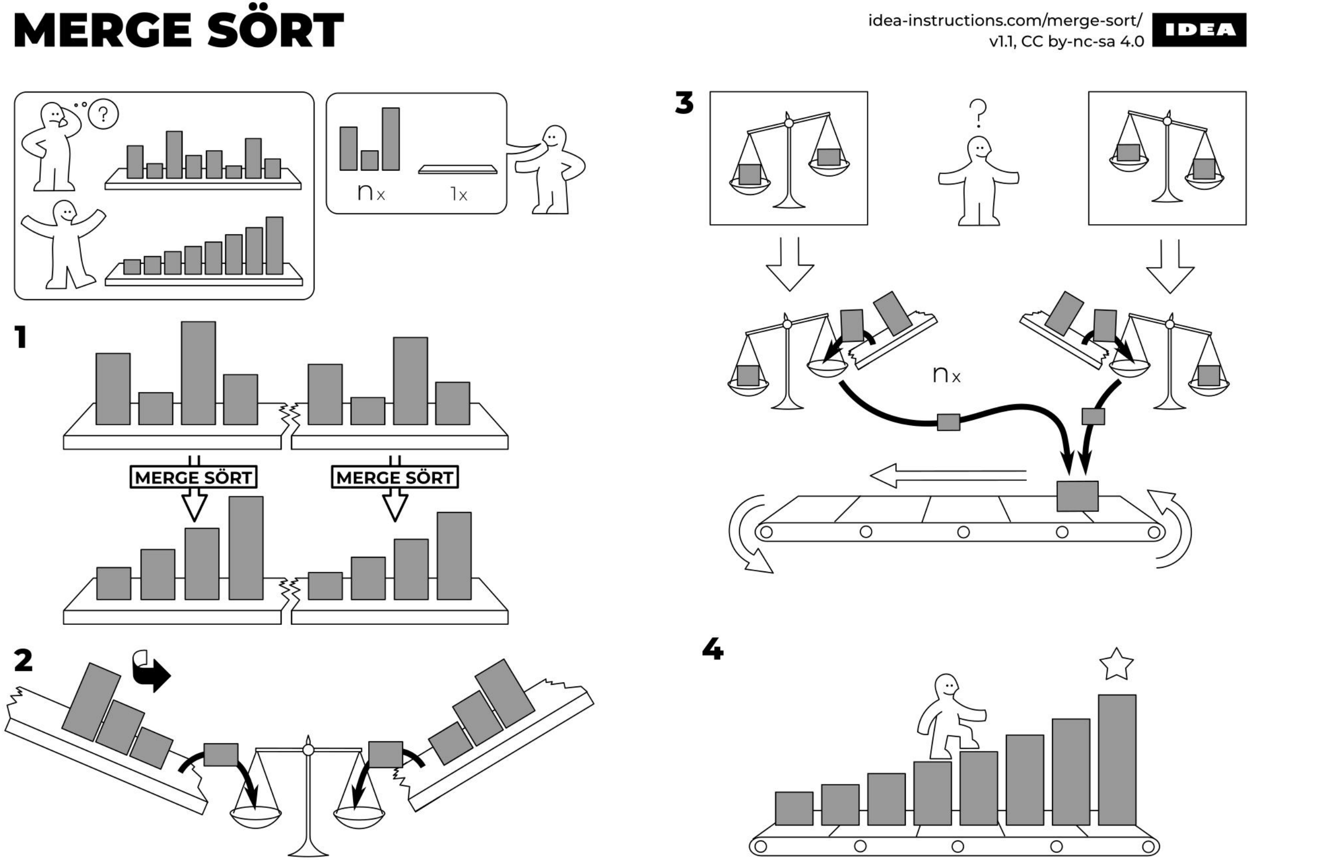
- Caso migliore (vettore ordinato): $\Theta(n)$
- Caso pessimo (vettore ordinato al contrario): $O(n^2)$
- Caso medio: $\Theta(n^2)$

Merge Sort

- Utilizza la tecnica del Divide et Impera ed opera in maniera ricorsiva
- Proposto da John von Neumann nel 1945

- Idea di base
  - Se la sequenza ha lunghezza 1 è già ordinata, altrimenti:
  1. Si divide la sequenza in due metà
  2. Ciascuna delle due sottosequenze viene ordinata ricorsivamente
  3. Le due sottosequenze vengono “fuse” estraendo ripetutamente il minimo delle due sottosequenze

Merge Sort


MERGE SÖRT
idea-instructions.com/merge-sort/
v1.1, CC by-nc-sa 4.0
IDEA
n x
1x
1
MERGE SÖRT
MERGE SÖRT
2
3
?
n x
4

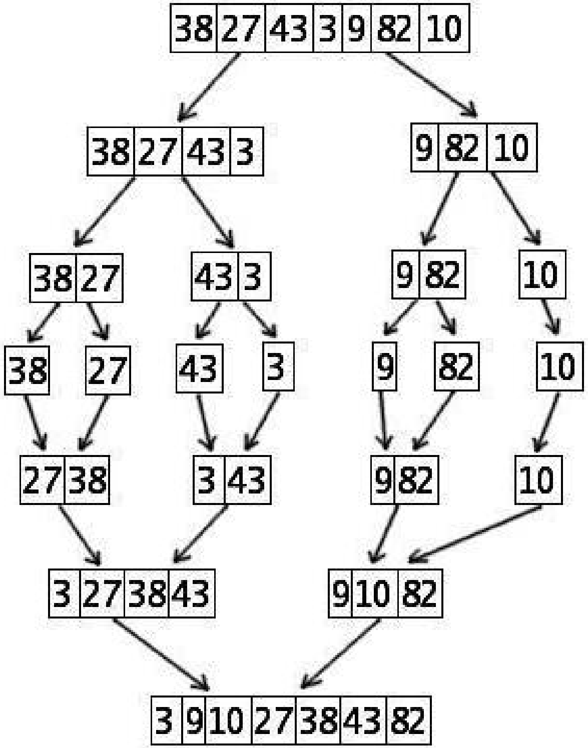
Merge Sort

```
MERGESORT (A, left, right)
    if left < right then
        center ← (left + right) / 2
        MERGESORT(A, left, center)
        MERGESORT(A, center+1, right)
        MERGE(A, left, center, right)
```

Merge Sort

```
MERGE(A, left, center, right):
  i ← left
  j ← center + 1
  k ← left
  while i ≤ center and j ≤ right:
    if A[i] ≤ A[j] then
      B[k] ← A[i]
      i ← i + 1
    else
      B[k] ← A[j]
      j ← j +1
    k ← k + 1
  j ← right
  for h ← center downto i:
    A[j] ← A[h]
    j ← j - 1
  for j ← left to k - 1:
    A[j] ← B[j]
```

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

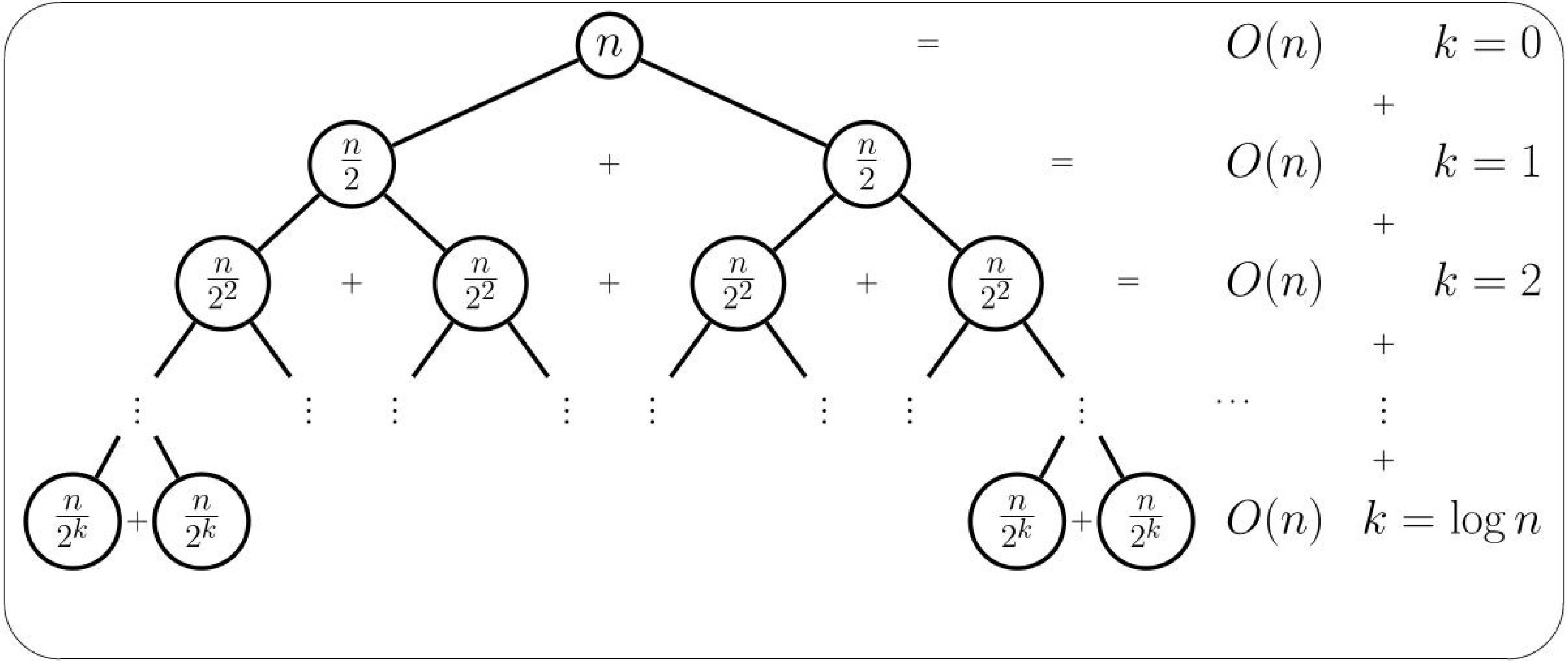
Merge Sort: analisi

- Semplificazioni:
  - $n = 2^k$, ossia il numero di suddivisioni sarà esattamente $k = \log n$
  - Tutti i sottovettori hanno un numero di elementi che sono potenze esatte di 2

- La relazione di ricorrenza è:

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

Merge Sort: analisi



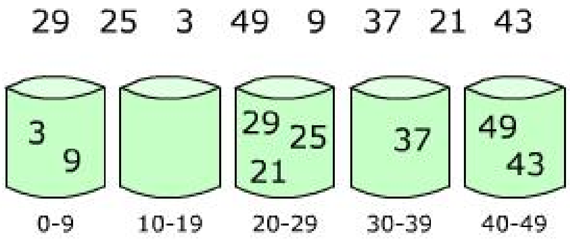
$$O\left(\sum_{i=1}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=1}^{k} n\right) = O(k \cdot n) = O(n \log n)$$

Merge Sort: analisi

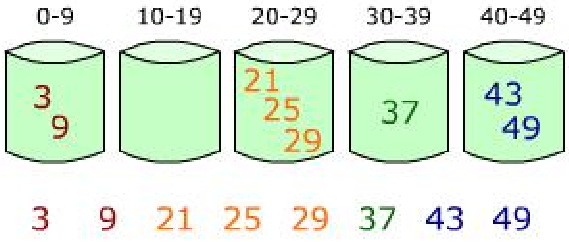
- Un altro approccio
- La relazione di ricorrenza è: $T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$
- Si può applicare il Teorema Principale:
  - $T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$
  - $T(n) = \begin{cases} \Theta(n^{\alpha}) & \text{se } \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \text{se } \alpha = \beta \\ \Theta(n^{\beta}) & \text{se } \alpha < \beta \end{cases}$, con $\alpha = \log_b a$
- La complessità è quindi $\Theta(n \log n)$

Bucket Sort

- Assume che gli elementi siano uniformemente distribuiti in un certo intervallo
- Algoritmo suddiviso in fasi
    1. Genera un certo numero di bucket
    2. *Scatter*: scandisci il vettore ed inserisci gli elementi in un bucket
    3. Ordina ciascun bucket non vuoto (tipicamente insertion sort)
    4. *Gather*: estrai da ciascun bucket gli elementi ed inseriscili nel vettore originale



Fase di scatter



Fase di gather

Bucket Sort

```
BUCKETSORT(A, k):
    buckets ← vettore di k bucket
    M ← elemento più grande nel vettore
    for i ← 1 to len(A):
        inserisci A[i] in buckets[⌊k * A[i] / M⌋]
    for i ← 1 to k:
        SORT(buckets[i])
    return concatenazione di buckets[1], …., buckets[k]
```

Bucket Sort: analisi

- Caso peggiore:
  - Cosa succede se tutti gli elementi cadono in un solo bucket?
  - Il costo è dominato dal costo dell'ordinamento del vettore
  - Nella pratica si tende ad utilizzare l'insertion sort: $O(n^2)$
- Caso medio (nel caso di input distribuito uniformemente):
  - Per determinare M si può utilizzare ARRAYMAX(): $O(n)$
  - L'operazione di scatter può anch'essa essere svolta in $O(n)$
  - L'ordinamento di ciascun bucket dipende dal numero di elementi che vengono inseriti nel bucket i-esimo:

$$O\left(\sum_{i=1}^{k} n_i^2\right)$$

Bucket Sort: analisi

- Per calcolare il costo nel caso medio siamo interessati a $E(n_i^2)$
- Sia X_{ij} una variabile aleatoria che vale 1 se l'elemento j-esimo viene inserito all'interno del bucket i-esimo
- Abbiamo che $n_i = \sum_{j=1}^{n} X_{ij}$
- Da cui: $E(n_i^2) = E\left(\sum_{j=1}^{n} X_{ij} \sum_{k=1}^{n} X_{ik}\right) = E\left(\sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij} X_{ik}\right) =$ $E\left(\sum_{j=1}^{n} X_{ij}^2\right) + E\left(\sum_{i \leq j, k \leq n} \sum_{j \neq k} X_{ij} X_{ik}\right)$
- Per l'assunzione di uniformità dell'input, $X_{ij} = 1$ con probabilità $1/k$:
  - $E\left(X_{ij}^2\right) = 1^2 \cdot \left(\frac{1}{k}\right) + 0^2 \cdot \left(1 - \frac{1}{k}\right)$
  - $E(X_{ij} X_{ik}) = \left(\frac{1}{k}\right)^2$

Bucket Sort: analisi

- Pertanto:

$$E\left(\sum_{j=1}^{n} X_{ij}^2\right) + E\left(\sum_{i\le j,k\le n}\sum_{j\ne k} X_{ij}X_{ik}\right) = n\cdot\frac{1}{k} + n(n-1)\frac{1}{k^2} = \frac{n^2+nk-n}{k^2}$$

- Da cui si ottiene che la complessità è:

$$O\left(\sum_{i=1}^{k} E(n_i^2)\right) = O\left(\sum_{i=1}^{k}\frac{n^2+nk-n}{k^2}\right) = O\left(\frac{n^2}{k}+n\right)$$

- L'ultimo passo è la concatenazione dei bucket che può essere svolta in $O(k)$. La complessità totale è quindi $O\left(\frac{n^2}{k}+n+k\right)$
- Se k è scelto in maniera tale da essere $k = \Theta(n)$, allora la complessità si riduce a $O(n)$

Quick Sort

- Si tratta di un algoritmo basato sul Divide et Impera
- È organizzato in tre fasi:
    1. Seleziona un elemento (*pivot*) dal vettore
    2. Partiziona il vettore: tutti gli elementi più grandi del pivot vengono spostati a destra, tutti quelli più piccoli a sinistra
    3. Applica questa strategia ricorsivamente alla parte destra e sinistra
- La scelta del pivot influisce fortemente sulle performance dell'algoritmo

Quick Sort

```
PARTITION(A, low, high):
    pivot ← A[low + (high - low) / 2]
    i ← low
    j ← high
    loop forever:
        while A[i] < pivot:
            i ← i + 1
        while A[j] > pivot:
            j ← j - 1
        if i >= j then
            return j
        swap A[i] with A[j]
```

```
QUICKSORT(A, low, high):
    if low < high then
        p ← PARTITION(A, low, high)
        QUICKSORT(A, low, p)
        QUICKSORT(A, p + 1, high)
```

```
QUICKSORT(A, 0, len(A)-1)
```

Quick Sort: analisi

- Caso pessimo:
  - Il caso peggiore si ha quando una delle due partizioni ha dimensione n-1
  - Questo fenomeno si verifica quando viene scelto come pivot l'elemento più piccolo o più grande del vettore
  - Le chiamate ricorsive si riducono quindi *ad n* chiamate su un vettore ciascuna volta di taglia *n-1*
  - L'*i*-esima chiamata ha costo $O(n-i)$
  - Il costo è quindi dato da $T(n) = \sum_{i=1}^{n}(n-i) = O(n^2)$
- Caso migliore:
  - Il caso migliore si ottiene quando il vettore viene partizionato esattamente a metà
  - $\log n$ chiamate, ciascun livello $O(n)$: $O(n \log n)$

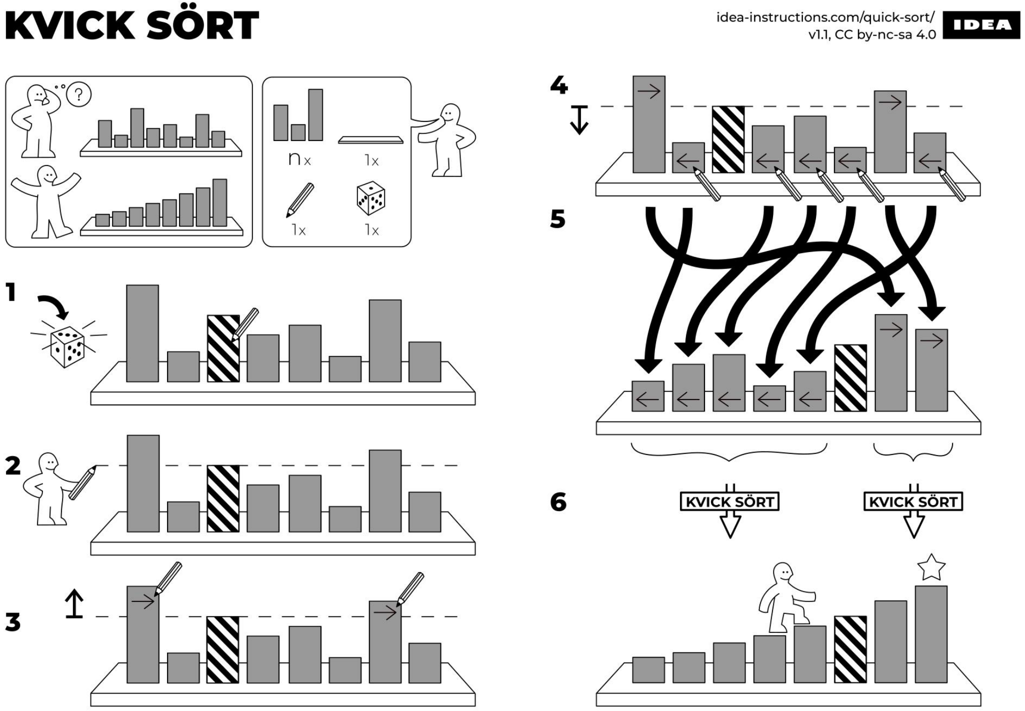
Quick Sort: analisi

- Caso medio:
  - $T(n) = O(n) + 2T\left(\frac{n}{2}\right)$
  - Dal Master Theorem: $T(n) = \ O(n\log n\ )$

- Se si riesce ad evitare il caso peggiore, quindi, si tratta di un algoritmo buono in pratica

Quick Sort

- Scegliere la chiave in maniera randomica può essere una buona strategia per evitare il caso peggiore



Radix Sort

- Utilizza un approccio controintuitivo per l'uomo
- Utilizza un ordinamento (tipo bucket sort) per ciascuna cifra dei numeri
- Complessità $O(nk)$

1ª iterazione	2ª iterazione	3ª iterazione	4ª iterazione	Risultato
253	10	5	5	5
346	253	10	10	10
1034	1034	127	1034	127
10	5	1034	127	253
5	346	346	253	346
127	127	253	346	1034

Altre proprietà degli algoritmi di ordinamento

- *Stabilità*: un algoritmo di ordinamento è stabile se preserva l'ordine iniziale tra due elementi con la stessa chiave
  - Ad esempio: ordinamento per nome *e* per cognome
- *Ordinamento sul posto (in place)*: si tratta di un algoritmo che non crea copie dell'input per generare la sequenza ordinata
- *Adattatività*: un algoritmo di ordinamento è adattativo se trae vantaggio dagli elementi già ordinati

Algoritmi di ordinamento: un riassunto

Algoritmo	T(n) - Caso ottimo	T(n) - Caso medio	T(n) - caso pessimo	S(n) (escluso input)	Stabile	In-Place	Adattativo
Stupid sort	$O(n)$	$O(n \cdot n!)$	∞	$O(n)$	No	No	No
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	No
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Insertion sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Merge sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n)$	Sì	No	No
Bucket sort	$O(n)$	$O(n)$	$O(n^2)$	$\Theta(n)$	Sì	No	No
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$\Theta(1)$	No	Sì	No
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$\Theta(n)$	No	No	No

Complessità computazionale del problema

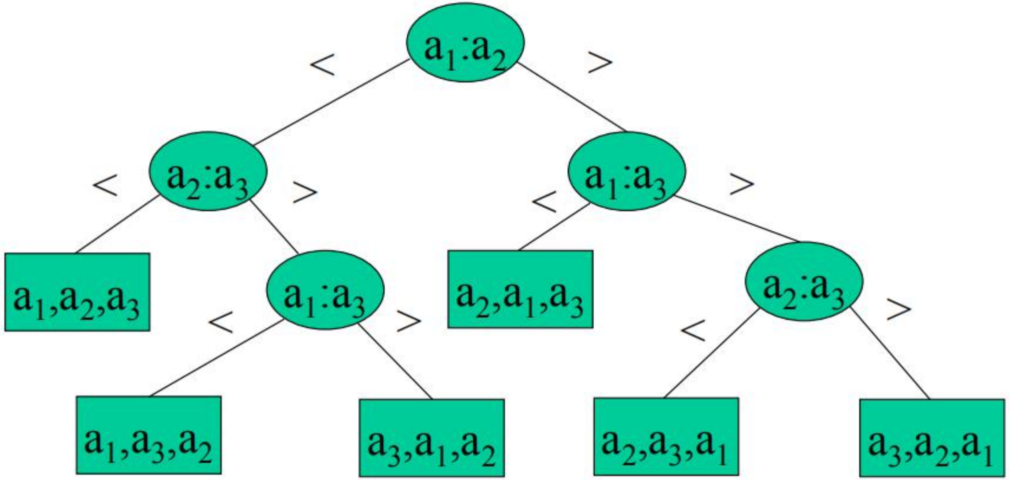
Teorema

- La complessità temporale di un qualsiasi algoritmo di ordinamento per confronto è pari a $\Omega(NlogN)$, dove N è il numero di elementi da ordinare.

- Si tratta di un importantissimo risultato
- Questo teorema fissa il limite inferiore di complessità per gli algoritmi che si basano sul confronto

Complessità computazionale del problema

- Per dimostrare questo risultato si può costruire un *albero di decisione*



Complessità computazionale del problema

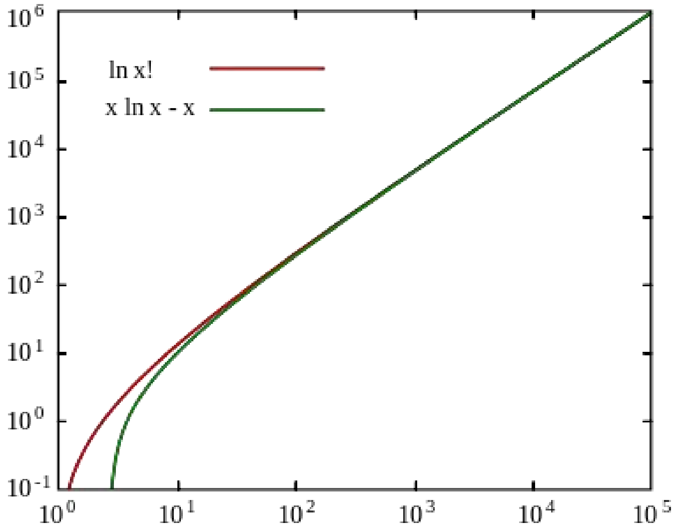
- Data una sequenza in input $e_1, e_2, \ldots, e_n$ per ogni sequenza ci sarà un cammino all'interno dell'albero
- Il numero possibile di permutazioni sull'input $e_1, e_2, \ldots, e_n$ è pari ad $n!$
- L'albero avrà quindi $n!$ foglie
- L'altezza dell'albero sarà $h(T) \leq \lceil \log n! \rceil$, che corrisponde al numero di confronti che vengono eseguiti
- Il valore di $n!$ può essere approssimato con la formula di Stirling

Complessità computazionale del problema

- La formula di Stirling fornisce un'approssimazione per fattoriali molto grandi:

$$\lim_{n \to +\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!}, \text{ ovverosia } n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- Quindi $n! > \left(\frac{n}{e}\right)^n$
- Da cui: $h(T) \geq \log\left(\frac{n}{e}\right)^n \geq n \log \frac{n}{e} =$
- $= n \log n - n \log e = \Omega(n \log n)$



Algoritmi di ordinamento: un riassunto

Algoritmo	T(n) - Caso ottimo	T(n) - Caso medio	T(n) - caso pessimo	S(n) (escluso input)	Stabile	In-Place	Adattativo
Stupid sort	$O(n)$	$O(n \cdot \mathrm{n!})$	∞	$O(n)$	No	No	No
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	No
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Insertion sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Merge sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n)$	Sì	No	No
Bucket sort	$O(n)$	$O(n)$	? $O(n^2)$	$\Theta(n)$	Sì	No	No
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$\Theta(1)$	No	Sì	No
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$\Theta(n)$	No	No	No