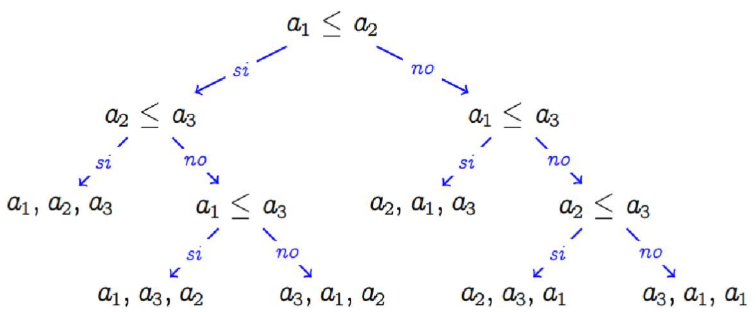
Comparison Sort

Definizione

Un algoritmo di ordinamento rientra nella famiglia dei comparison sort quando la sua logica **ignora il valore specifico degli elementi** e si basa solo sull'**esito dei confronti**

Di conseguenza, ogni esecuzione di un algoritmo comparison sort è rappresentabile da un **albero dei confronti**



Confronti necessari per ordinare un vettore di 3 elementi

Proprietà: Albero dei Confronti

In un algoritmo comparison sort

- Il **costo del caso pessimo** sarà equivalente all'altezza dell'albero
- Dati n valori, esistono $n!$ **possibili ordinamenti**, di conseguenza l'albero dei confronti avrà **almeno** $n!$ **foglie**

Lower Bound

Idea

Trovando l'altezza dell'albero dei confronti di un algoritmo comparison sort con $n!$ foglie, troveremo il **lower bound degli algoritmi comparison sort**

In un albero binario con n nodi, sappiamo che

- $h = \Theta(\log(n))$
- $LIV_h = \Theta(n)$

Sapendo che il valore minimo di foglie è $n!$ e che il numero di foglie è $LIV_h = \Theta(n)$, l'altezza dell'albero e il costo del caso pessimo saranno

$$h = \Theta(\log(n!))$$

Scomponiamo

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \ \ldots \ \cdot n) \\ &= \underbrace{\log(1) + \log(2) + \cdots + \log(n)}_{n}\end{aligned}$$

Confrontando con solo parte della sommatoria

$$\begin{aligned}\log(n!) &\geq \underbrace{\log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}+1\right) + \cdots + \log(n)}_{\frac{n}{2}} \\ &\geq \frac{n}{2}\log\left(\frac{n}{2}\right) \\ &= \Theta(n\log(n))\end{aligned}$$

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

Lower Bound Comparison Sort

Sia A un algoritmo di comparison sort, il **costo computazionale del caso pessimo** di A è

$$\Omega(n\log(n))$$

Counting Sort

> 💡 **Idea**
>
> Contando quanti elementi sono minori o uguali ad un determinato valore, possiamo capire la sua posizione nel vettore

Consideriamo i seguenti vettori

- **A**: vettore di partenza
- **C**: occorrenze di ogni valore di A

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

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

C[2]: occorrenze di 2 in A

Trasformiamo ora C per contenere le **occorrenze dei valori minori uguali** di ogni elemento

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

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

	0	1	2	3	4	5
C	2	2	4	7	7	8

C[2]: Occorrenze totali dei valori di A ≤ 2

Da qui possiamo prendere ogni valore di A e metterlo in un nuovo vettore B in base agli **spazi necessari per i valori minori e uguali**.

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							3	

Infine **decrementiamo le occorrenze del valore** in C e ripetiamo per ogni elemento di A

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0					3	

Il vettore B sarà il nostro vettore ordinato risultante

Osservazione: Ordinamento Stabile

Se si scorrono gli elementi di A dal fondo, il counting sort **mantiene l'ordinamento originale** degli elementi di A.

Algoritmi di ordinamento con questo comportamento vengono definiti **stabili**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0				3	3	

Definizione: Algoritmo di Ordinamento Stabile

Supponiamo di dover orinare n elementi

$$el_i = < k_i, d_i >$$

Un algoritmo di ordinamento è stabile se, per ogni coppia di indici $i, j \mid i < j \wedge k_i = k_j$, $< k_i, d_i >$ precede $< k_j, d_j >$ nell'ordinamento

COUNTING-SORT(A,B,k)

```
1  let C[0..k] be a new array
2  for i = 0 to k
3          C[i] = 0
4  for j = 1 to A.length
5          C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements = i
7  for i = 1 to k
8          C[i] = C[i] + C[i - 1]
9  // C[i] now contains the number of elements <= i
10 for j = A.length downto 1
11         B[C[A[j]]] = A[j]
12         C[A[j]] = C[A[j]] - 1
```

Parametri: *A=vettore, B=vettore ordinato risultante, k=valore massimo in A*

⚠ Non Comparison Sort

Basandosi sul valore degli elementi di A, il counting sort **non è un [algoritmo comparison sort]**

Costo Computazionale

Supponiamo di ordinare n numeri tra $0..k$ e consideriamo il costo delle varie operazioni

- **2-3**: $\Theta(k)$
- **4-5**: $\Theta(n)$
- **7-8**: $\Theta(k)$
- **10-12**: $\Theta(k)$

ⓘ Costo Computazionale

Il costo computazionale varia a seconda del valore massimo contenuto in A

$$\Theta(n + k)$$

Quindi, per avere un **costo lineare**, bisognerà avere $k = O(n)$

Radix Sort

Idea

Consideriamo un vettore di numeri con lo **stesso numero di cifre**.
Possiamo riordinare gli elementi una cifra alla volta partendo dalle unità

$$
\begin{array}{ccccccccccccccc}
 & & \downarrow & & & \downarrow & & & \downarrow & & & & & & \\
3 & 2 & 9 & & 7 & 2 & 0 & & 7 & 2 & 0 & & 3 & 2 & 9 \\
4 & 5 & 7 & & 3 & 5 & 5 & & 3 & 2 & 9 & & 3 & 5 & 5 \\
6 & 5 & 7 & & 4 & 3 & 6 & & 4 & 3 & 6 & & 4 & 3 & 6 \\
8 & 3 & 9 & \to & 4 & 5 & 7 & \to & 8 & 3 & 9 & \to & 4 & 5 & 7 \\
4 & 3 & 6 & & 6 & 5 & 7 & & 3 & 5 & 5 & & 6 & 5 & 7 \\
7 & 2 & 0 & & 3 & 2 & 9 & & 4 & 5 & 7 & & 7 & 2 & 0 \\
3 & 5 & 5 & & 8 & 3 & 9 & & 6 & 5 & 7 & & 8 & 3 & 9
\end{array}
$$

Osservazione: Necessità di Algoritmo Stabile

Per funzionare, sarà necessario che ogni iterazione su una cifra mantenga l'ordinamento originale della cifra precedente. Per questo necessitiamo di un algoritmo di ordinamento stabile, come il counting sort

RADIX-SORT(A, d)

```
for i = 1 to d
        use a stable sort algorith to sort A on digit i
```

Parametri*: A=vettore, d=numero di cifre degli elementi di A*

Costo Computazionale

Supponiamo di usare il counting sort come algoritmo di ordinamento stabile.

$$d \cdot \Theta(n+k)$$

Considerando una cifra alla volta, i valori per ogni vettore di cifre andranno da 0 a 9, portando quindi il costo del counting sort a $\Theta(n)$

Costo Computazionale

Il costo computazionale varia a seconda del numero di cifre dei valori di A

$$\Theta(d \cdot n)$$

Quindi, per avere un **costo lineare**, bisognerà avere $d = O(n)$