

Ancora su sorting

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014



Questa lezione

- **Recap su code di priorità e sorting**
 - ~ Insertion sort
 - ~ Selection sort
- **In place sorting**
- **Merge sort iterativo**
- **Radix sort**
 - ~ Complessità $O(n)$
 - ~ Non basato sul confronto



Ordinamento e code di priorità - recap



Ordinamento con code di priorità

- Si inseriscono gli elementi nella coda di priorità con una serie di operazioni *insert*
- Si rimuovono gli elementi dalla coda di priorità con una sequenza di *removeMin*

```
// Usiamo una coda di priorità coda
Output: <array a ordinato rispetto alle chiavi degli elementi in S>
while (!S.isEmpty):
    coda.insert(S.remove())
    // Assumiamo che gli elementi di S siano coppie (k, v)
    // S.remove() rimuove un elemento da S secondo un criterio qualsiasi
while (!coda.isEmpty):
    a.add(coda.removeMin()) //add aggiunge alla fine dell'array
return a
```



Esempio: Selection Sort

	<i>Sequenza S</i>	<i>Coda di priorità P</i>	
Input:	(7,4,8,2,5,3,9)	()	
Fase 1			
(a)	(4,8,2,5,3,9)	(7)	O(n)
(b)	(8,2,5,3,9)	(7,4)	
..	..		
(g)	()	(7,4,8,2,5,3,9)	
Fase 2			
(a)	(2)	(7,4,8,5,3,9)	O(n ²)
(b)	(2,3)	(7,4,8,5,9)	
(c)	(2,3,4)	(7,8,5,9)	
(d)	(2,3,4,5)	(7,8,9)	
(e)	(2,3,4,5,7)	(8,9)	
(f)	(2,3,4,5,7,8)	(9)	
(g)	(2,3,4,5,7,8,9)	()	



Esempio: Insertion Sort

	Sequenza S	Coda di priorità P	
Input:	(7,4,8,2,5,3,9)	()	
Phase 1			
(a)	(4,8,2,5,3,9)	(7)	O(n ²)
(b)	(8,2,5,3,9)	(4,7)	
(c)	(2,5,3,9)	(4,7,8)	
(d)	(5,3,9)	(2,4,7,8)	
(e)	(3,9)	(2,4,5,7,8)	
(f)	(9)	(2,3,4,5,7,8)	
(g)	()	(2,3,4,5,7,8,9)	
Phase 2			
(a)	(2)	(3,4,5,7,8,9)	O(n)
(b)	(2,3)	(4,5,7,8,9)	
..	
(g)	(2,3,4,5,7,8,9)	()	



Ordinamento sul posto (in-place)

Obiettivo:

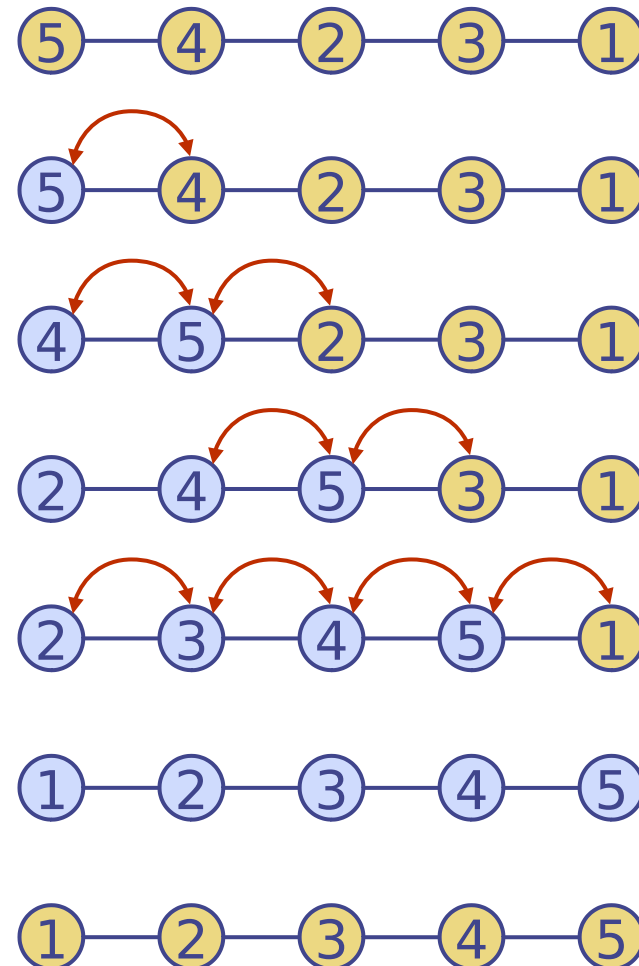
Non usare strutture dati aggiuntive
Ridurre la memoria richiesta



In-place Insertion sort

- **Invece di usare una struttura dati ausiliaria ...**
 - ~ Array di destinazione
 - ~ Heap
 - ~ ...
- **Una porzione della sequenza di input è mantenuta ordinata**
- **Per insertion sort**
 - ~ Prima parte della sequenza è mantenuta ordinata
 - ~ Si usano scambi tra elementi
- **Analogamente per Selection sort (v. libro di testo)**

- **Insertion sort**



In-place Heap sort

- Usiamo un max-Heap

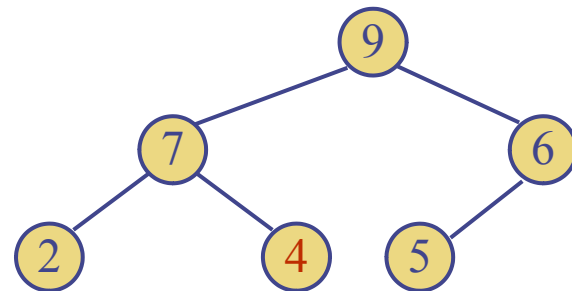
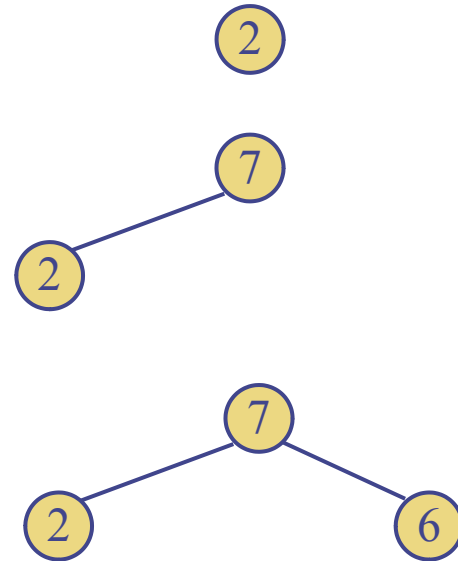
2	7	6	4	9	5
---	---	---	---	---	---

7	2	6	4	9	5
---	---	---	---	---	---

7	2	6	4	9	5
---	---	---	---	---	---

9	7	6	2	4	5
---	---	---	---	---	---

Fine della prima fase



In-place Heap sort/Fase 2

9	7	6	2	4	5
---	---	---	---	---	---

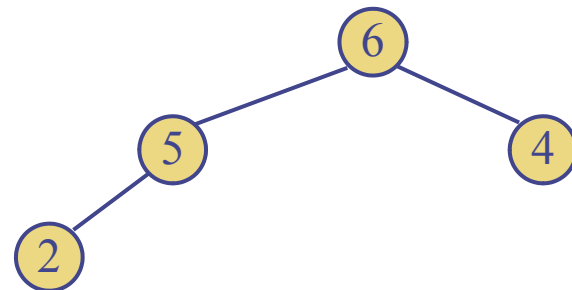
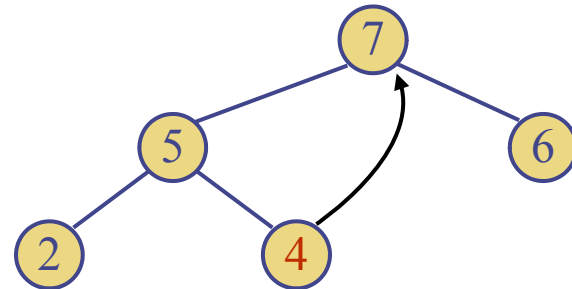
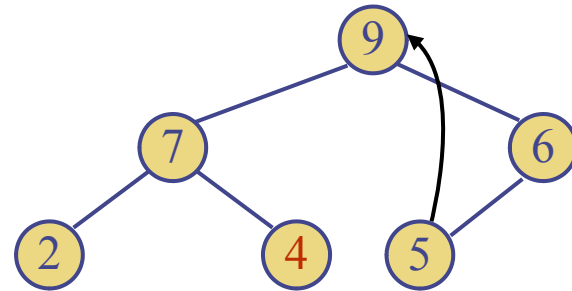
Scambio tra max e ultimo elemento

- 9 è il più grande numero e deve essere in ultima posizione nell'array

7	5	6	2	4	9
---	---	---	---	---	---

- L'elemento con chiave 9 è ancora nell'array
- `heap.size` → `heap.size()-1`
- In questo modo, l'ultima posizione non è più parte dell'heap

6	5	4	2	7	9
---	---	---	---	---	---

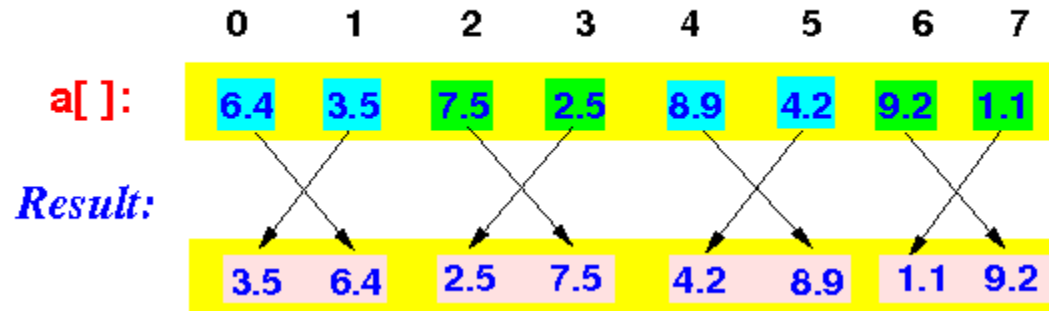


Merge sort iterativo

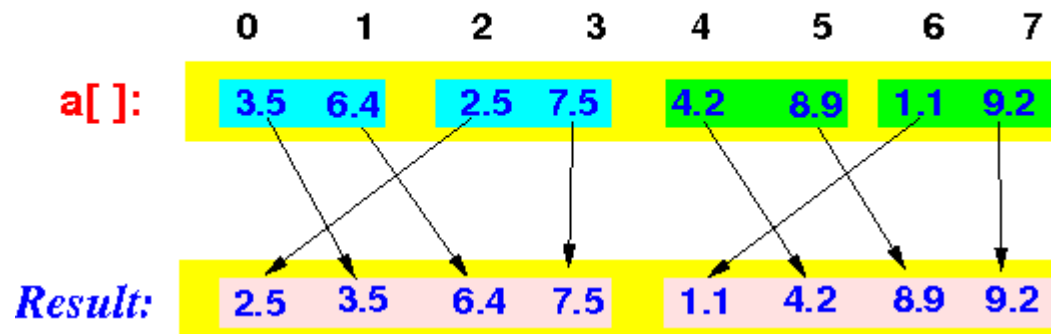


Merge sort iterativo

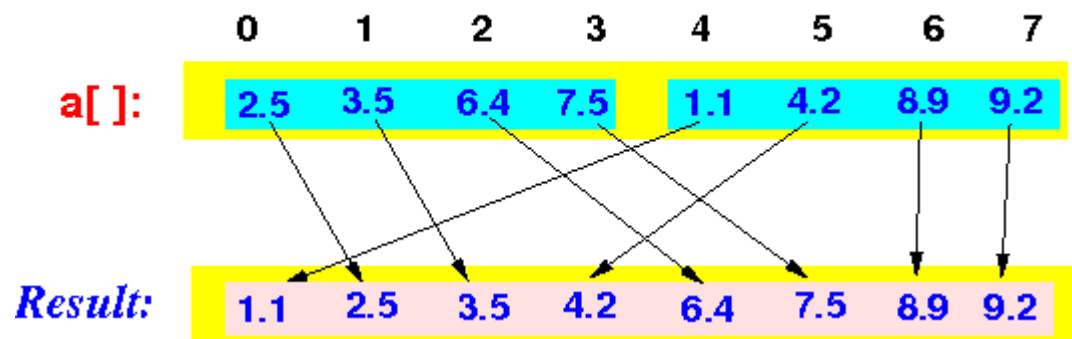
Merge pairs of arrays of size 1



Merge pairs of arrays of size 2




Merge pairs of arrays of size 4



Merge sort iterativo (bottom up)

```
// Credits: Prof. Shun Yan Cheung - Emory College
public static void sort(double[] a)
{
    int width;
    for ( width = 1; width < a.length; width = 2*width )
    {
        // Combine pairs of array a of width "width"; values of
variable width: 1,2,4,8,...
        int i;
        for ( i = 0; i < a.length; i = i + 2*width )
        {
            int left, middle, right;
            left = i;
            middle = i + width;
            right = i + 2*width;
            merge( a, left, middle, right );
            // Merge è quello solito
        }
    }
}
```



Complessità

- **Ovviamente, la complessità è la stessa dell'implementazione ricorsiva**
- **Prova**
 - ~ Ricorda che il costo di ciascun merge è lineare nella somma degli array parziali
 - ~ Numero di iterazioni del ciclo "for" esterno: $\log N$
 - ~ Il costo di esecuzione del ciclo "for" interno: $O(N)$ – la prova è analoga al caso ricorsivo

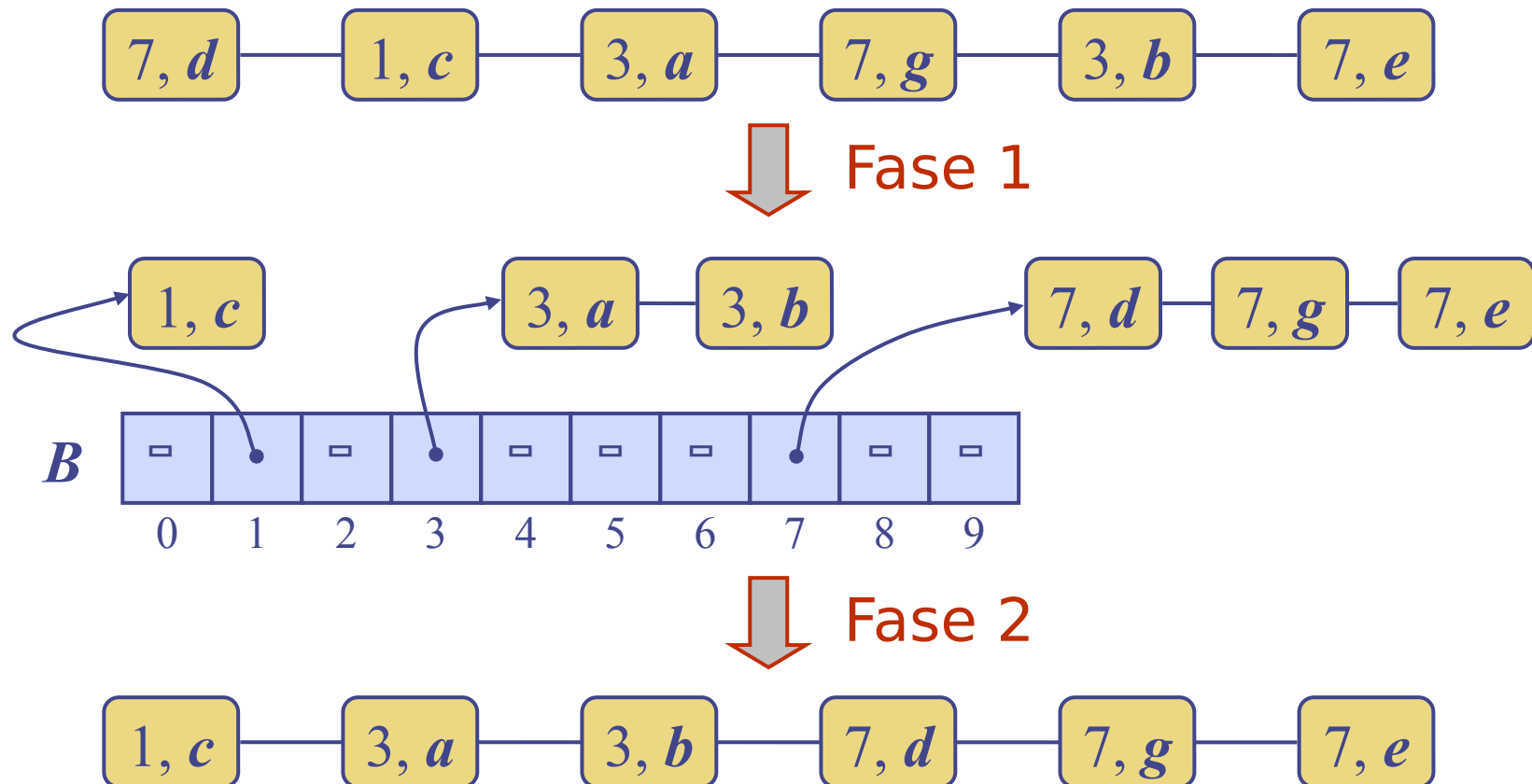


Radix sort



Bucket sort

- Chiavi nell'intervallo $[0, 9]$
- Implementazione *stabile*



Recap su Bucket sort

Algorithm bucketSort(S):

Input: Sequence S of entries with integer keys in the range $[0, N - 1]$

Output: Sequence S sorted in nondecreasing order of the keys

B = <array di N sequenze> // Inizialmente vuoto

// Fase 1

```
while (!S.isEmpty()) {  
    e = S.removeFirst()  
    k = e.key()  
    B[k].addLast(e)  
}
```

// Fase 2

```
for (k = 0; k < N; k++)  
    while (!B[k].isEmpty())  
        S.addLast(B[k].removeFirst())
```

Ingredienti principali

- Chiavi intere
- Sia $n = |S|$. Complessità Bucket sort $\rightarrow O(n + N)$
- Si noti che non si effettuano confronti



Ordinamento lessicografico

- ***d*-tupla: sequenza di *d* chiavi (k_1, \dots, k_d) appartenenti ciascuna a un insieme ordinato**
- **Ordinamento lessicografico tra *d*-tuple**
 - ~ $(x_1, \dots, x_d) < (y_1, \dots, y_d)$ se e solo se:
 - ~ $x_1 < y_1$ OR $(x_1 = y_1)$ AND $(x_2, \dots, x_d) < (y_2, \dots, y_d)$
 - ~ Si noti che 1 è la componente più “importante”



Radix sort

- Applicabile quando la chiave può a sua volta essere vista come una tupla di chiavi
- Esempio: chiavi intere ≥ 0 rappresentate con b bit
- Si usa l'algoritmo Bucket sort come routine
 - ~ Si ordina per d volte
 - ~ La i -esima volta si ordina rispetto alla componente $d-i+1$ delle chiavi \rightarrow da destra a sinistra
 - ~ Si usa Bucket sort

Algorithm *radixSort*(S, N)

Input sequenza S di d -tuple tali che

$(0, \dots, 0) \leq (x_1, \dots, x_d)$ e

$(x_1, \dots, x_d) \leq (N-1, \dots, N-1)$

per ogni tupla (x_1, \dots, x_d) in S

Output sequenza S in ordine lessicografico

for $i = d$ downto 1

bucketSort(S, N, i) // i -esima cifra



Esempio: sequenza di n interi rappresentati con b bit

- Ogni intero è rappresentato come

$$X = X_{b-1} \dots X_0$$

- Nella i -esima iterazione
si ordina rispetto a x_i
- Si usano $N = 2$ bucket

Algorithm *binaryRadixSort*(S)

Input sequenza S di interi a b -bit

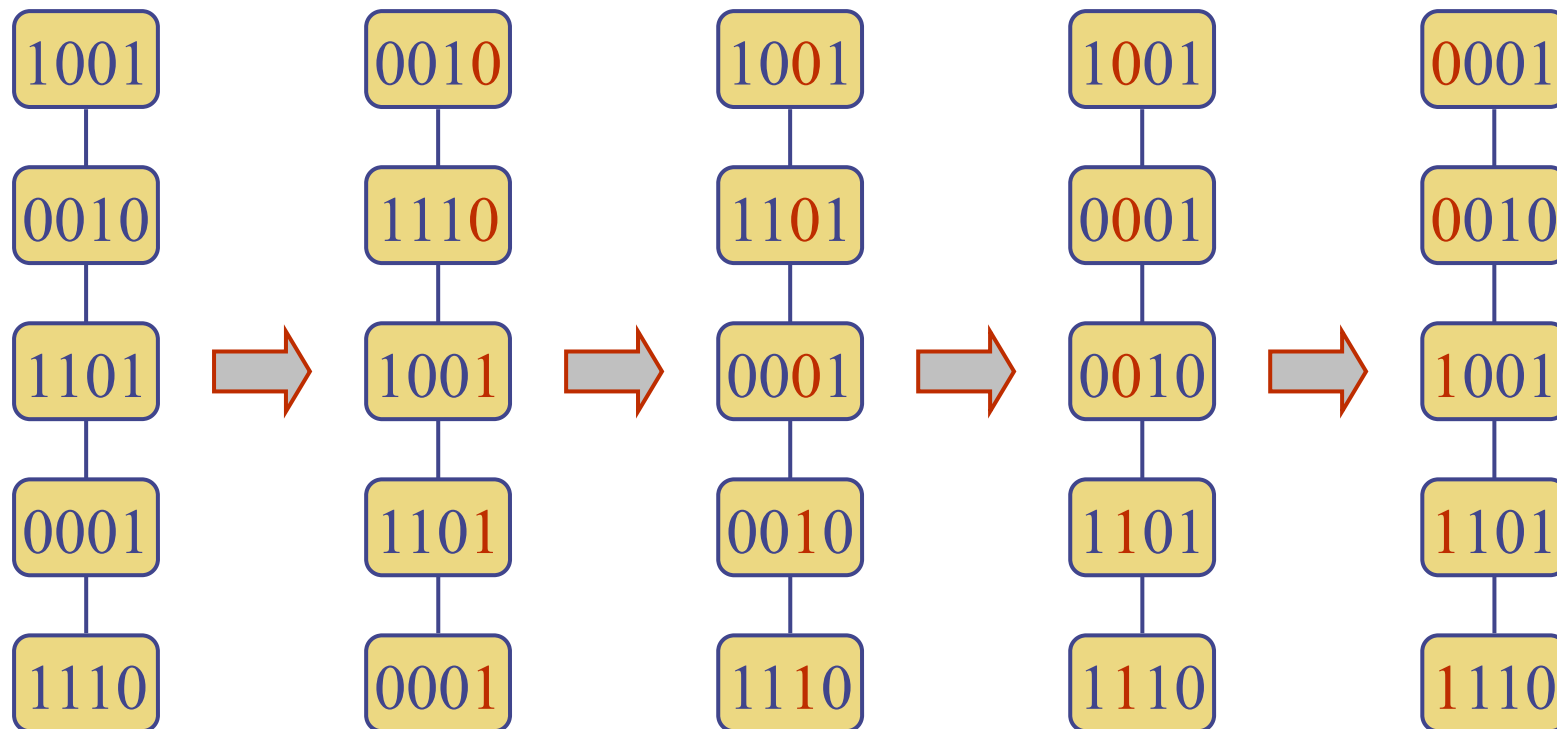
Output sequenza S ordinata

for $i = 0$ to $b - 1$

bucketSort($S, 2, i$)



Esempio: sequenza di interi a 4 bit



Stabilità: se due elementi hanno l' i -esimo bit uguale, il loro ordinamento alla fine della $(i-1)$ -esima fase non cambia nella i -esima fase



Conclusioni: cosa scegliere in pratica?

- **Quicksort** è una buona scelta predefinita. Tende ad essere veloce in pratica e con alcune piccole modifiche il costo $O(N^2)$ della complessità temporale nel caso peggiore diventa molto improbabile.
- **Heapsort** va scelto se non puoi tollerare una complessità temporale $O(N^2)$ nel caso peggiore o necessitano di bassi costi di spazio. Il kernel Linux usa heapsort invece di quicksort per entrambi questi motivi.
- **Mergesort** è una buona scelta se vuoi un algoritmo stabile. Inoltre, merge sort può essere esteso per gestire dati che non possono essere inseriti in RAM; in questi casi il costo è dato dalla lettura e scrittura dell'input su disco, non il confronto e lo scambio di singoli dati.
- **Radix -sort** sembra veloce ($O(N)$ nel caso peggiore). Ma se si usa per ordinare numeri binari, allora c'è un fattore costante nascosto (di solito 32 o 64 - numero di bit dei dati). Nota: se $N < 4$ miliardi allora $32 > \log N$; invece per dati a 64 bit abbiamo $\log N > 64$ se $N < 16$ miliardi di miliardi; questo significa che tende ad essere lento nella pratica.



Conclusioni: cosa scegliere in pratica?

- **Insertion sort, Bubble sort:** non sono di solito buone scelte solo in alcuni casi particolari; esempio sappiamo che i dati sono quasi ordinati e con pochi scambi li ordiniamo (bubblesort), oppure che il numero dei dati è molto piccolo (insertion sort)

Spesso si utilizzano **algoritmi ibridi**:

- si combina un algoritmo efficiente per l'ordinamento complessivo (es. Quicksort, heapsort) con Insertion sort per piccoli elenchi in fondo a una ricorsione
- le implementazioni altamente ottimizzate utilizzano varianti più sofisticate, come Timsort (merge sort, insert sort e logica aggiuntiva), utilizzato in Android, Java e Python, e introsort (quicksort e heapsort), utilizzato (in diverse varianti) in alcune implementazioni C++.

