▼ **4.0 - Algoritmi di ordinamento**

> Il **problema dell'ordinamento** consiste nei seguenti input e output.
>
> - **Input**: una sequenza di n numeri $[a_1, \dots, a_n]$
> - **Output**: una permutazione $p : \{1, \dots, n\} \to \{1, \dots, n\}$ degli indici degli elementi della sequenza in input al fine di ottenere un'altra sequenza $[a_{p(1)}, \dots, a_{p(n)}]$ tale che $a_{p(1)} \leq \dots, \leq a_{p(n)}$.

Discuteremo dei seguenti algoritmi di ordinamenti e dei loro andamenti asintotici:

- **Incrementali**: SelectionSort, InsertionSort
- **Divide et impera**: MergeSort, QuickSort
- **Non-comparativi**: CountingSort, RadixSort

Nozioni preliminari

Assumiamo di utilizzare algoritmi di ordinamenti per array in cui ogni elemento è composto da:

- Una **chiave**, confrontabili tra loro rispetto a $\leq, =, \geq$.
- Un **valore**, il quale rappresenta il contenuto associato alla chiave.

Tali algoritmi ordineranno quindi gli array rispetto alla chiave e non al valore, e in molti casi l'array in input conterrà solamente la chiave.

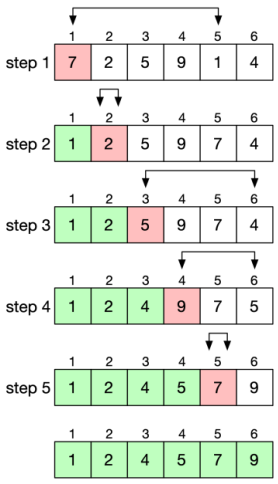
Inoltre gli algoritmi di ordinamento possono presentare due **proprietà**:

- **In place**: l'algoritmo riordina gli array utilizzando solamente l'array in input, senza bisogno di array di sostegno.
- **Stabile**: valori che hanno la stessa chiave appaiono nell'array in output nello stesso ordine in cui si trovavano nell'array in input.

▼ **4.1 - Algoritmi incrementali**

SelectionSort

Dato un array A di dimensione n, ad ogni passo i da 1 a $n-1$, viene cercata la posizione j corrispondente alla minima chiave nel sottoarray $A[i, \dots, n]$ e viene scambiato l'elemento $A[j]$ con $A[i]$.



Esempio grafico di SelectionSort.

Lo pseudocodice dell'algoritmo di SelectionSort è il seguente:

```
void selectionsort(Array A[1, ..., n]) {
  for (int i = 0; i < n - 1; i++) {
    m = i

    for (int j = i + 1; j < n; j++) {
      if (A[j] < A[m]) m = j
    }

    if (m != i) swap(A, i, m)
  }
}


void swap(Array A[1, ... , n], int i, int j) {
  tmp = A[i]
  A[i] = A[j]
  A[j] = tmp
}
```

Complessità computazionale di SelectionSort

Notiamo che l'implementazione del SelectionSort utilizza due cicli, dei quali il primo viene eseguito esattamente $n-1$ volte, mentre il secondo viene eseguito $n-i$ volte per ogni passo i. La funzione swap che viene richiamata all'interno dei due cicli invece ha costo costante $O(1)$.

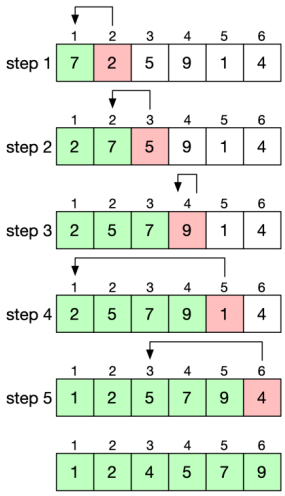
Siccome sappiamo che i due cicli vengono eseguiti per intero ogni volta che l'algoritmo viene eseguito, allora il caso ottimo, medio e pessimo possono essere condensati in un unico:

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

Possiamo dunque concludere che l'algoritmo di ordinamento **SelectionSort** ha **complessità quadratica** sulla lunghezza dell'array.

InsertionSort

Dato un array A di dimensione n, ad ogni passo i da 2 a n, il sottoarray $A[1,\dots,i-1]$ risulta ordinato e viene inserito l'elemento $A[i]$ nella corretta posizione in $A[1,\dots,i]$.



Esempio grafico di InsertionSort.

Lo pseudocodice dell'agoritmo InsertionSort è il seguente:

```
void insertionsort(Array A[1, ... , n]) {
  for (int i = 1; i < n; i++) {
    j = i
    while (j > 0 && A[j] < A[j − 1]) {
      swap(A, j, j − 1)
      j = j − 1
    }
  }
}
```

Complessità computazionale di InsertionSort

Notiamo che anche in questo caso l'algoritmo è formato da due cicli annidati, dei quali il primo viene sempre eseguito interamente $n-1$ volte, mentre il secondo potrebbe anche non venire eseguito. Dividiamo dunque l'analisi nel caso ottimo, pessimo e medio.

Nel **caso ottimo** le chiavi nell'array sono già ordinate dalla più piccola alla più grande, dunque il secondo ciclo non viene mai eseguito. Abbiamo dunque che la complessità è $\Theta(n)$, costo **lineare**.

Il caso ottimo si verifica anche per array in input **quasi ordinati**, in quanto se A è ordinato per tutti tranne k elementi, allora, visto che il secondo ciclo viene eseguito solo su questi k elementi, la complessità totale sarà $\Theta(n)\ (\text{ciclo for}) + O(nk)\ (\text{ciclo while}) = O(nk)$. Quindi abbiamo che se il numero k di elementi non ordinati è costante rispetto ad n, allora il costo di InsertionSort rimane **lineare**: $\Theta(n)$.

Nel **caso pessimo** le chiavi nell'array sono ordinate dalla più grande alla più piccola, dunque il secondo ciclo viene eseguito ogni volta $i-1$ volte. Calcoliamo dunque la complessità tramite la sommatoria $\sum_{i=2}^{n} i-1 = \sum_{i=2}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$, costo **quadratico**.

Per il **caso medio** occorre fare delle assunzioni probabilistiche. Sappiamo infatti che il ciclo while viene eseguito al minimo 0 volte e al massimo $i-1$, dunque possiamo assumere che questo viene eseguito in media $\frac{i-1}{2}$ volte. Il costo totale nel caso medio è dunque calcolabile tramite la sommatoria $\sum_{i=2}^{n} \frac{i-1}{2} = \frac{1}{2}\sum_{i=2}^{n-1} i = \frac{n(n-1)}{4} = \Theta(n^2)$, costo **quadratico**.

▼ 4.2 - Algoritmi divide et impera

Quella del **divide et impera** è una strategia per ottenere controllo militare e politico: fare in modo che gli avversari siano divisi e si combattano tra di loro.

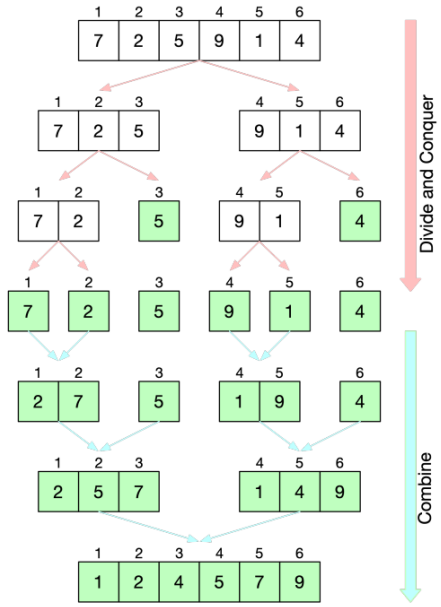
Negli algoritmi la strategia del divide et impera consiste nei seguenti step:

- **Divide**: dividere il problema in sotto-problemi più piccoli.
- **Conquer**: risolvere i sotto-problemi in maniera ricorsiva.
- **Combine**: fondere le soluzioni dei sotto-problemi per ottenere quella del problema originario.

MergeSort

Gli step divide et impera del **MergeSort** sono i seguenti:

1. **Divide**: dividere l'array in input $A[1, \dots, n]$ in due metà $A_1 = A[1, \dots, \frac{1+n}{2}]$ e $A_2 = A[\frac{1+n}{2} + 1, \dots, n]$.
2. **Conquer**: richiamare ricorsivamente l'algoritmo su A_1 e A_2 se hanno lunghezza > 1, in quanto array con lunghezza $= 1$ sono già ordinati.
3. **Combine**: combinare i due array ordinati A_1, A_2 in un unico array ordinato.



Esempio grafico di MergeSort.

Lo pseudocodice dell'algoritmo di MergeSort è il seguente:

```
void mergesort(Array A[1, ... , n], int p, int r) {
  if (p < r) {
    q = (p + r) / 2
    mergesort(A, p, q)
    mergesort(A, q + 1, r)
    merge(A, p, q, r)
  }
}

void merge(Array A[1, ... , n], int p, int q, int r) {
  B = A[1, ... , r − p + 1]
  i = p
  j = q + 1
  k = 1
```

```
  while (i <= q && j <= r) {
    if (A[i] <= A[j]) {
      B[k] = A[i]
      i = i + 1
    } else {
      B[k] = A[j]
      j = j + 1
    }
    k = k + 1
  }

  while (i ≤ q) {
    B[k] = A[i]
    k = k + 1
    i = i + 1
  }

  while (j ≤ r) {
    B[k] = A[j]
    k = k + 1
    j = j + 1
  }

  for (int k = 0; k++; k < r − p + 1) {
    A[p + k − 1] = B[k]
  }
}
```

Complessità computazionale di MergeSort

Non è necessario comprendere alla perfezione l'algoritmo di **merge**, ci basta sapere che esso ha un **costo lineare** sulla lunghezza $r - p + 1$. Questo costo viene dal fatto che in ognuno dei tre cicli while viene incrementato il valore di i oppure di j, mai insieme, e i viene incrementato da 1 a q, mentre j va da $q + 1$ a r, dunque vengono effettuate $r - p + 1$ iterazioni. Notiamo che anche il ciclo for viene eseguito $r - p + 1$ volte.

Dopo aver analizzato il costo dell'algoritmo merge ($\Theta(n)$) possiamo ricavare l'**equazione di ricorrenza di mergesort**:

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

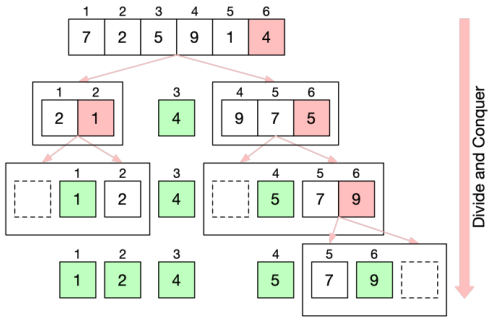
Possiamo dunque ricavare il costo tramite l'utilizzo del Master Theorem: $\Theta(n \log n)$, ovvero **pseudologaritmico**.

Inoltre notiamo che il costo di mergesort non dipende da come i numeri sono inizialmente organizzati all'interno dell'array, dunque il caso ottimo, medio e pessimo coincidono.

QuickSort

Gli step divide et impera del **QuickSort** sono i seguenti:

1. **Divide**: partizionare l'array in input $A[1, \ldots, n]$ in due sottoarray $A_1 = A[1, \ldots, q - 1]$ e $A_1 = A[q + 1, \ldots, n]$ tali che tutte le chiavi in A_1 e A_2 siano rispettivamente $\leq$ e $>$ della chiave $A[q]$ scelta durante il partizionamento.
2. **Conquer**: richiamare in modo ricorsivo l'algoritmo su A_1 e A_2.
3. **Combine**: non è necessario ricombinare A_1 e A_2 in quanto già ordinati.



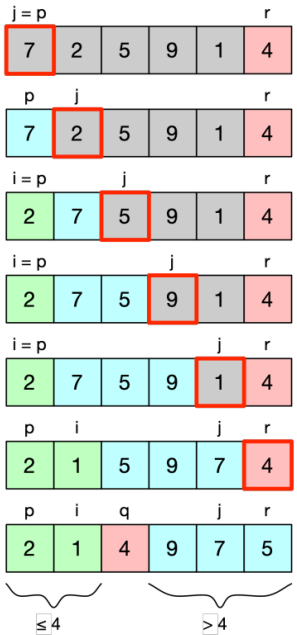
Esempio grafico di QuickSort.

Lo pseudocodice dell'algoritmo QuickSort è il seguente:

```
void quicksort(Array A[1, ... , n], int p, int r) {
  if (p < r) {
    q = partition(A, p, r)
    quicksort(A, p, q − 1)
    quicksort(A, q + 1, r)
  }
}

int partition(Array A[1, ... , n], int p, int r) {
  x = A[r] // scelta deterministica del pivot
  i = p − 1
  for (j = p, ... , r − 1) {
    if (A[j] ≤ x) {
      swap(A, i + 1, j)
      i = i + 1
    }
  }
  swap(A, i + 1, r) // move the pivot in A[i + 1]
  return i + 1
}
```

Osserviamo che la funzione partition contiene un unico ciclo che viene sempre eseguito da p a r-1, dunque essa ha un costo nel caso ottimo, medio e pessimo equivalente a $\Theta(r - p + 1)$.



Esempio grafico di partition.

Il costo computazionale della funzione QuickSort dipende invece dalla scelta del pivot, la quale causa il bilanciamento/sbilanciamento dei sottoarray, basati sulla loro lunghezza. Distinguiamo dunque il costo di QuickSort in caso ottimo, pessimo e medio:

- **Caso ottimo**: i due sottoarray sono entrambi lunghi $n/2$.

  L'equazione di ricorrenza è dunque la seguente:

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

  Possiamo utilizzare il Master Theorem per calcolare che il costo nel caso ottimo è **pseudologaritmico** $\Theta(n \log n)$.

- **Caso pessimo**: un array di lunghezza 0 e l'altro $n - 1$.

  L'equazione di ricorrenza è dunque la seguente:

  $$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + T(0) + n & n > 1 \end{cases}$$

  Possiamo risolverla tramite il metodo iterativo:

  $T(n) = T(n-1) + T(0) + n$
  $= T(n-1) + 1 + n$
  $= T(n-2) + 2 + (n-1) + n$
  $= T(n-3) + 3 + (n-2) + (n-1) + n$
  $\dots$
  $= T(n-i) + i + \sum_{k=0}^{i-1} n - k$

  La ricorsione termina per $n - i = 0$, ovvero $i = n$. Sostituiamo dunque n a i per trovare il costo del caso pessimo: $1 + n + \sum_{k=0}^{n-1} n - k = 1 + n + \sum_{k=1}^{n} k = 1 + n + \frac{n(n+1)}{2} = \Theta(n^2)$.

- **Caso medio**: l'equazione di ricorrenza è la seguente:

  $$T(n) = \begin{cases} 1 & n \leq 1 \\ T(i) + T(n-i-1) + n & n > 1 \end{cases}$$

  Osservando che i e $n - i - 1$ possono cambiare ad ogni chiamata ricorsiva allora dobbiamo fare un'assunzione probabilistica pensando al fatto che tutte le partizioni sono equiprobabili e costruendo dunque la seguente equazione di ricorrenza:

  $$T(n) = \begin{cases} 1 & n \leq 1 \\ \sum_{i=0}^{n-1} \frac{T(i) + T(n-i-1)}{n} + n & n > 1 \end{cases}$$

  Notando che $\sum_{i=0}^{n-1} T(n-i-1) = \sum_{i=0}^{n-1} T(i)$, in quanto una sommatoria è equivalente all'altra al contrario, possiamo semplificare l'equazione di ricorrenza ottenendo:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n & n > 1 \end{cases}$$

Infine utilizziamo il metodo di sostituzione per dimostrare (slide 34 del pacco di slide “Algoritmi di ordinamento”) che $T(n) \leq cn \ln n$, e potendo dunque concludere che QuickSort nel caso medio ha un costo **pseudologaritmico**: $O(n \log n)$.

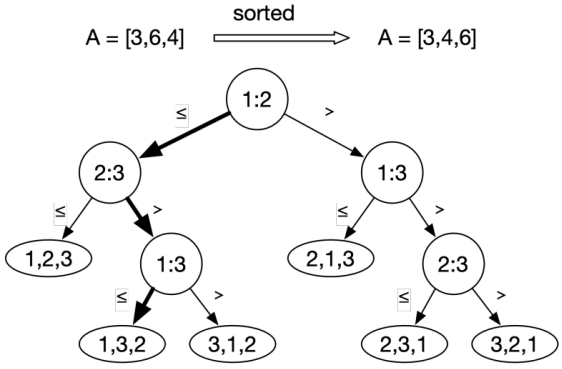
Nell’algoritmo di QuickSort che abbiamo scritto e analizzato viene scelto come pivot sempre l’ultimo elemento dell’array. Questo, nel caso in cui l’array di partenza è quasi ordinato, comporta spesso il caso pessimo, e nella realtà array quasi ordinati sono più probabili di array totalmente disordinati. Lasciando questa scelta del pivot dunque si ricadrebbe più spesso nel caso pessimo. Per risolvere questo problema è possibile scegliere il pivot in maniera **randomica**, causando più spesso il caso medio.

▼ **4.3 - Algoritmi non comparativi**

Lower bound per ordinamento comparativo

Tutti gli algoritmi che abbiamo descritto finora sono considerabili **algoritmi comparativi**, ovvero che si basano sul confronto tra valori.

Ciascun algoritmo comparativo può essere descritto tramite un **albero di decisione**, ossia un albero binario in cui ogni nodo corrisponde al confronto tra due argomenti dell’array.



Esempio di albero di decisione.

Osservazioni sull’albero di decisione:

- L’esecuzione dell’algoritmo dato un determinato array in input corrisponde dunque ad un percorso radice-foglia.
- L’altezza dell’albero corrisponde numero di confronti nel caso pessimo, mentre l’altezza media corrisponde al numero di confronti nel caso medio.
- Il numero di foglie è almeno uguale a $n!$, ossia il numero di permutazioni dell’array di partenza.

Teorema altezza albero di decisione

> Sia T_k un albero binario con k foglie in cui ogni nodo ha esattamente due figli e sia $h(T_k)$ l’altezza di T_k. Allora $h(T_k) \geq \log k$.

Dimostrazione slide 41 del pacco di slide “Algoritmi di ordinamento”.

Teorema lower bound per ordinamento comparativo

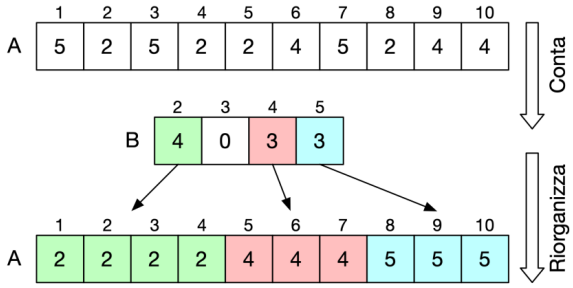
> Ogni algoritmo di ordinamento comparativo richiede $\Omega(n \log n)$ confronti nel caso pessimo.

Dimostrazione: siccome abbiamo visto che ogni albero di decisione ha almeno $n!$ foglie, utilizzando il teorema precedente otteniamo che ogni albero di decisione ha un altezza $\Omega(\log n!) = \Omega(n \log n)$, e tale altezza corrisponde ai confronti da effettuare nel caso pessimo.

Tale lower bound appena dimostrato vale però solo per algoritmi di tipo comparativo. Vediamo dunque due algoritmi che utilizzano una struttura di tipo non comparativa.

CountingSort

L'algoritmo di **CountingSort** deriva dall'idea che nell'array A dato in input sono contenuti interi contenuto in un certo intervallo $[a, b]$. È possibile dunque creare un secondo array B di lunghezza $b - a + 1$ in cui per ogni intero nell'intervallo $[a, b]$ viene inserito il numero di volte che compare all'interno dell'array A. Fatto ciò vengono riorganizzati tali valori all'interno dell'arrray A.



Esempio grafico di CountingSort.

Lo pseudocodice dell'algoritmo di CountingSort è il seguente:

```
void countingsort(Array A[1, ... , n]) {
  a = min(A)
  b = max(A)
  k = b − a + 1
  B[1, ... , k]
  // initialize B
  for (i = 1, ... , k)
    B[i] = 0
  // insert values into B
  for (i = 1, ... , n)
    B[A[i] − a + 1] = B[A[i] − a + 1] + 1
  // insert ordered values into A
  j = 1
  for (i = 1, ... , k) {
    while (B[i] > 0) {
      A[j] = i + a − 1
      B[i] = B[i] − 1
      j = j + 1
    }
  }
}
```

Complessità computazionale di CountingSort

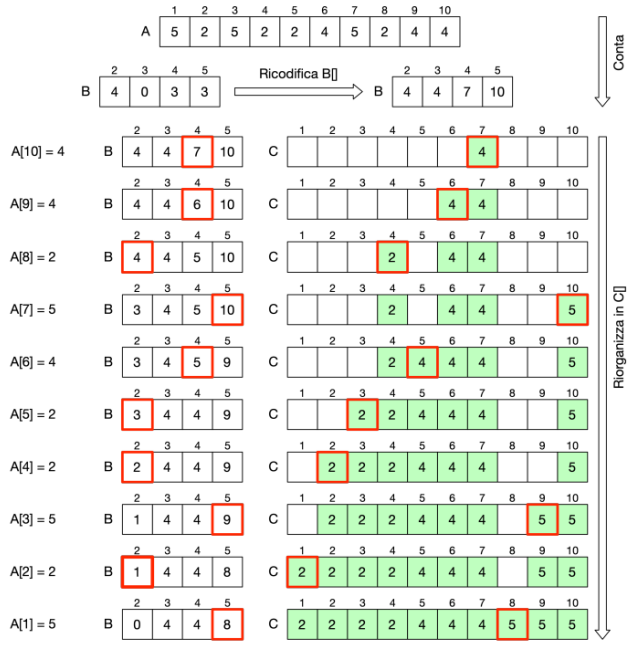
Osserviamo che le funzioni min e max costano entrambe $\Theta(n)$, il primo ciclo for viene eseguito k volte, il secondo ciclo for viene eseguito n volte, infine gli ultimi cicli annidati for e while vengono

eseguiti n volte in quanto devono inserire in A tutti gli n numeri. Concludiamo dunque che il caso ottimo, medio e pessimo coincidono e hanno un costo equivalente a $\Theta(n+k)$.

Notiamo dunque che il costo dell'algoritmo dipende sia dal numero di elementi nell'array in input ma anche da k, ovvero la dimensione del range $[a, b]$.

CountingSort stabile

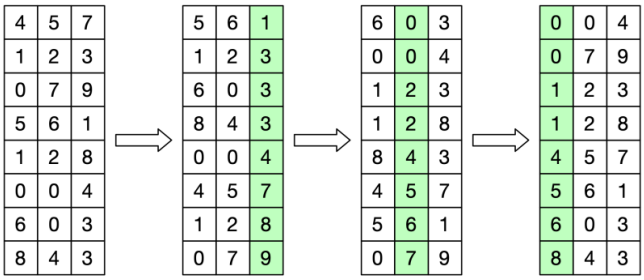
L'algoritmo di CountingSort che abbiamo appena analizzato non è stabile, ma è comunque possibile effettuare delle piccole modifiche all'algoritmo per renderlo stabile e mantenere i costi asintotici.



Esempio grafico di CountingSort stabile.

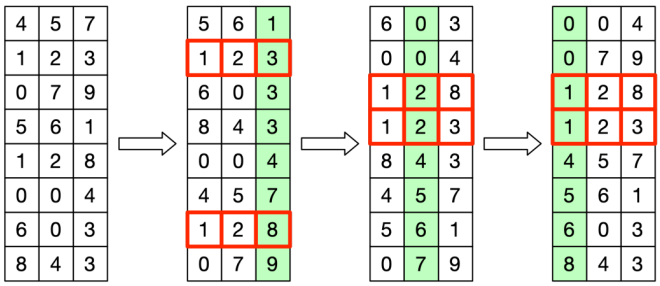
RadixSort

L'algoritmo di **RadixSort** consiste nell'ordinare i valori dell'array dato in input prima rispetto alla cifra meno significativa, poi rispetto alla penultima cifra meno significativa e così via.



Esempio grafico di RadixSort.

È importante notare che l'algoritmo di ordinamento utilizzato per ordinare ogni cifra dei numeri dati input deve essere **stabile**, altrimenti al termine dell'esecuzione i numeri potrebbero non risultare ordinati:



Esempio di errore causato dall'utilizzo di un algoritmo non stabile.

Lo pseudocodice dell'algoritmo di RadixSort è il seguente:

```
void radixsort(Array A[1, ... , n])
  d = max key length
  for (i = 1, ... , d)
    // ordinamento stabile della i-esima cifra delle chiavi
```

Complessità computazionale di RadixSort

La complessità computazionale di RadixSort dipende dall'algoritmo di ordinamento utilizzato per ordinare le cifre delle chiavi.

Possiamo ad esempio pensare di utilizzare l'algoritmo di CountingSort stabile in quanto la sua complessità dipende dal range delle chiavi, dunque se le chiavi dell'array in input sono interi allora il range massimo sarà 10, mentre se le chiavi sono delle stringhe il range massimo sarà equivalente al numero di caratteri ammessi nella stringa. Il costo dell'algoritmo di RadixSort con l'utilizzo del CountingSort equivale dunque a $\Theta(d(n+k))$, dove d rappresenta il numero di cifre massime delle chiavi in input.

Osserviamo inoltre che se $k = O(n)$ e d è un valore costante, allora il costo è $\Theta(n)$, quindi lineare sul numero degli elementi dati in input.

Riassunto

Il riassunto dei costi e delle proprietà degli algoritmi di ordinamento che abbiamo analizzato è dunque rappresentato nella seguente tabella:

	Caso ottimo	Caso medio	Caso pessimo
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n^2)$
CountingSort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$
RadixSort	$\Theta(d(n+k))$	$\Theta(d(n+k))$	$\Theta(d(n+k))$

	In place	Stabile
SelectionSort	Si	Si
InsertionSort	Si	Si
MergeSort	No	Si
QuickSort	Si	No
CountingSort	No	Si
RadixSort	No	Si

Riassunto dei costi e delle proprietà degli algoritmi di ordinamento.

- n: numero di elementi da ordinare.
- k: ampiezza del range di valori chiave.
- d: numero massimo di cifre in una chiave.