

Dati e Algoritmi: A. Pietracaprina

Ordinamento

L'ordinamento è una **primitiva fondamentale**. La sua importanza ha motivato un'intensa attività di ricerca per più di mezzo secolo.

Noi vedremo:

- ① **Algoritmi basati su confronti:** MergeSort, QuickSort (design pattern *divide-and-conquer*), InsertionSort, e un lower bound che si applica a qualsiasi algoritmo basato su confronti.
- ② **Algoritmi non-basati su confronti:** BucketSort, RadixSort.

Ordinamento basato su confronti

Specifica Input-Output

Input: sequenza $S = S[0]S[1] \dots S[n-1]$ di $n \geq 0$ chiavi da un universo ordinato

Output: sequenza S ordinata in senso non decrescente

Definizione

Un **algoritmo di ordinamento basato su confronti (comparison-based)** stabilisce l'ordine delle chiavi di una sequenza S basandosi **ESCLUSIVAMENTE** su confronti tra coppie di chiavi di S . In altre parole, date due sequenze di input S_1 e S_2 , se l'algoritmo esegue gli stessi confronti, nello stesso ordine, e con lo stesso esito, esso determina la stessa permutazione che ordina le chiavi.

N.B.: Rappresentiamo S come array $S[0], S[1], \dots$, assumendo di avere accesso diretto a ciascuna chiave $S[i]$. Gli algoritmi che presenteremo possono anche essere implementati su liste, ma risulterebbero meno efficienti in pratica.

Design Pattern: Divide-and-Conquer

Una strategia *Divide-and-Conquer* per un problema computazionale Π risolve ogni istanza i di Π di taglia n in 3 fasi:

1 Divide:

- Se $n \leq n_b$, risolve i direttamente (CASO BASE)
- Se $n > n_b$, divide i in 2 o più istanze di Π , $i_1, i_2, \dots$, ciascuna taglia $< n$

2 Conquer: Risolve ciascuna istanza i_j , con $j \geq 1$

3 Combine: Determina la soluzione di i combinando opportunamente le soluzioni delle istanze i_j , con $j \geq 1$.

Osservazione. L'implementazione più naturale del *Divide-and-Conquer* è tramite la **ricorsione**. Tuttavia, se la ricorsione porta a risolvere tante volte la stessa istanza (è il caso, ad esempio, del calcolo dell' n -esimo numero di Fibonacci), allora altre implementazioni risultano più efficienti: ad esempio, il **dynamic programming**, studiato nel corso di Algoritmi per l'Ingegneria.

MergeSort

Algoritmo MergeSort(S)

Input: come da specifica generale

Output: come da specifica generale

```
/* Divide */  
if  $n \leq 1$  then return;  
 $S_1, S_2 \leftarrow$  sequenze vuote;  
for  $i \leftarrow 0$  to  $\lceil n/2 \rceil - 1$  do  $S_1[i] \leftarrow S[i]$ ;  
for  $i \leftarrow \lceil n/2 \rceil$  to  $n - 1$  do  $S_2[i - \lceil n/2 \rceil] \leftarrow S[i]$ ;  
/* Conquer */  
MergeSort( $S_1$ );  
MergeSort( $S_2$ );  
/* Combine */  
Merge( $S_1, S_2, S$ );
```

Merge

Algoritmo Merge(S_1, S_2, S)

Input: sequenze S_1, S_2 ordinate in senso crescente, sequenza vuota S

Output: sequenza $S = S_1 \cup S_2$ ordinata in senso crescente

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0;$

while ($(i < S_1.size())$ and $(j < S_2.size())$) **do**

if $S_1[i] \leq S_2[j]$ **then** $S[k++] \leftarrow S_1[i++]$;
 else $S[k++] \leftarrow S_2[j++]$;

while ($i < S_1.size()$) **do** $S[k++] \leftarrow S_1[i++]$;

while ($j < S_2.size()$) **do** $S[k++] \leftarrow S_2[j++]$;

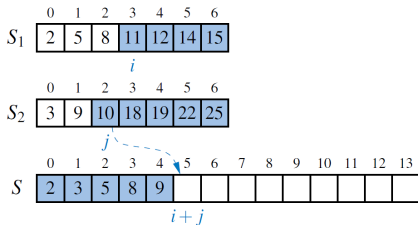
Osservazione: L'algoritmo funziona correttamente anche se S_1 e S_2 hanno lunghezze diverse.

Le prove di correttezza per MergeSort e Merge sono lasciate come semplici esercizi.

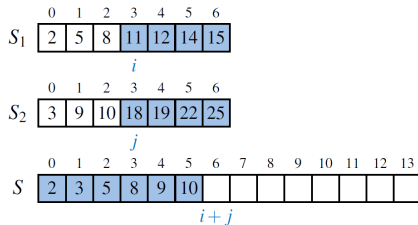
Esempio: generica iterazione del Merge Sort

(a): inizio iterazione

(b): fine iterazione



(a)



(b)

Proposizione 1

La complessità di $\text{Merge}(S_1, S_2, S)$ è $\Theta(n)$, dove n è il numero di chiavi in S_1 più il numero di chiavi in S_2 .

Dimostrazione

Ogni iterazione di ciascuno dei 3 cicli while

- * Esegue $\Theta(1)$ operazioni

- * Si inserisce definitivamente una chiave in S

$\Rightarrow n$ iterazioni (sui 3 cicli) di costo $\Theta(1)$

Ciascuna

$\Rightarrow$ Complessità: $\Theta(n)$



Proposizione 2

La complessità di MergeSort(S) è $\Theta(n \log n)$, dove $n \geq 1$ è il numero di chiavi in S .

Dimostrazione

Consideriamo l'albero delle ricorsioni relativo alla esecuzione di MergeSort per una sequenza S arbitraria di n chiavi. Distinguiamo 2 casi:

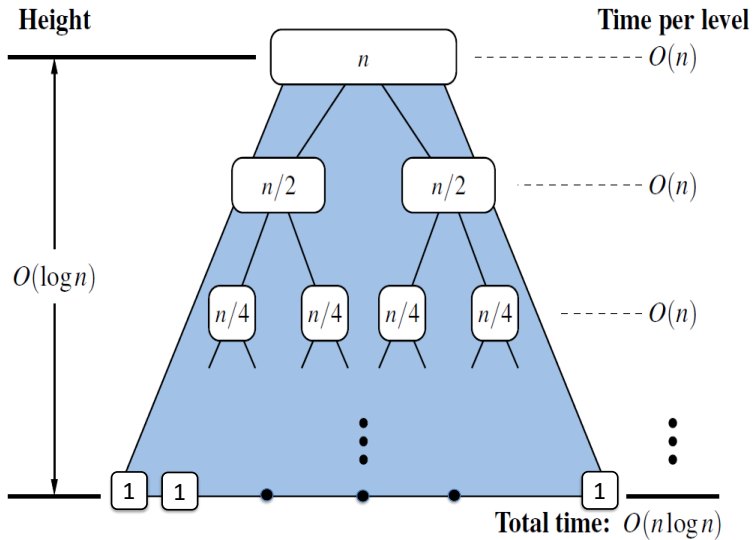
CASO 1 : $n = 2^d$ per un qualche intero $d \geq 0$

CASO 2 : $2^d < n < 2^{d+1}$ per un qualche intero $d \geq 0$

Caso 1: $n = 2^d$. L'albero delle ricorsive ha
* 2^i nodi al livello i corrispondenti a invocazioni
su istanze di taglia $n/2^i$, per $i = 0, 1, \dots, d$
 $\Rightarrow d+1$ livelli

* Il costo associato a un nodo del livello i è $\Theta(n/2^i)$
(dovuto alle fasi di divide e combine ($\equiv$ merge))
 $\Rightarrow$ Ogni livello i contribuisce un costo approssimato
 $\Theta(2^i \cdot n/2^i) = \Theta(n)$

$\Rightarrow$ Complessità: $\Theta(d \cdot n) = \Theta(n \log n)$ ($d = \log_2 n$)



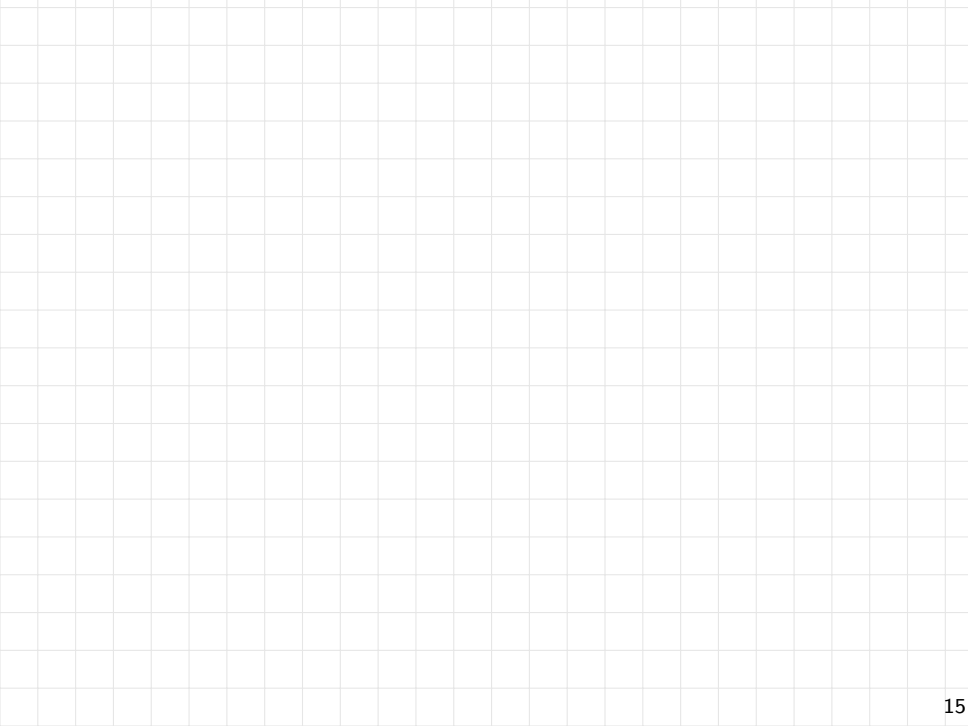
Caso 2: $2^d < n < 2^{d+1}$ $d \geq 0$ ESERCIZIO

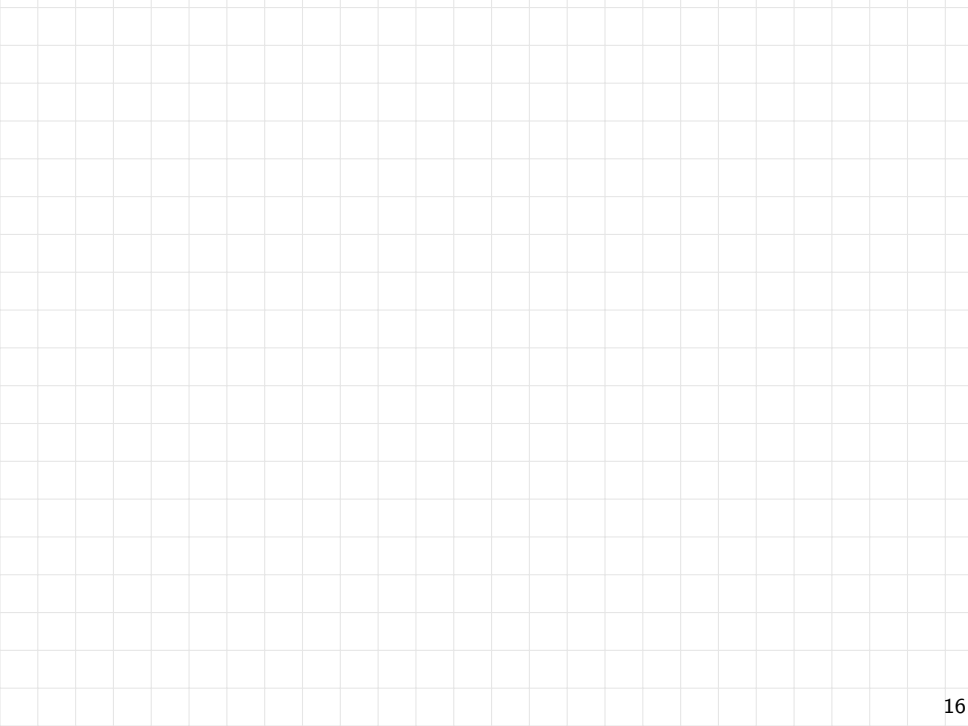
Suggerimento: partire dalla osservazione (facile da provare)
che ogni livello i dell'albero della ricorrenza
ha $\leq 2^i$ nodi associati a invocazioni su
istanze di taglia $\leq 2^{d+1}/2^i < 2n/2^i$

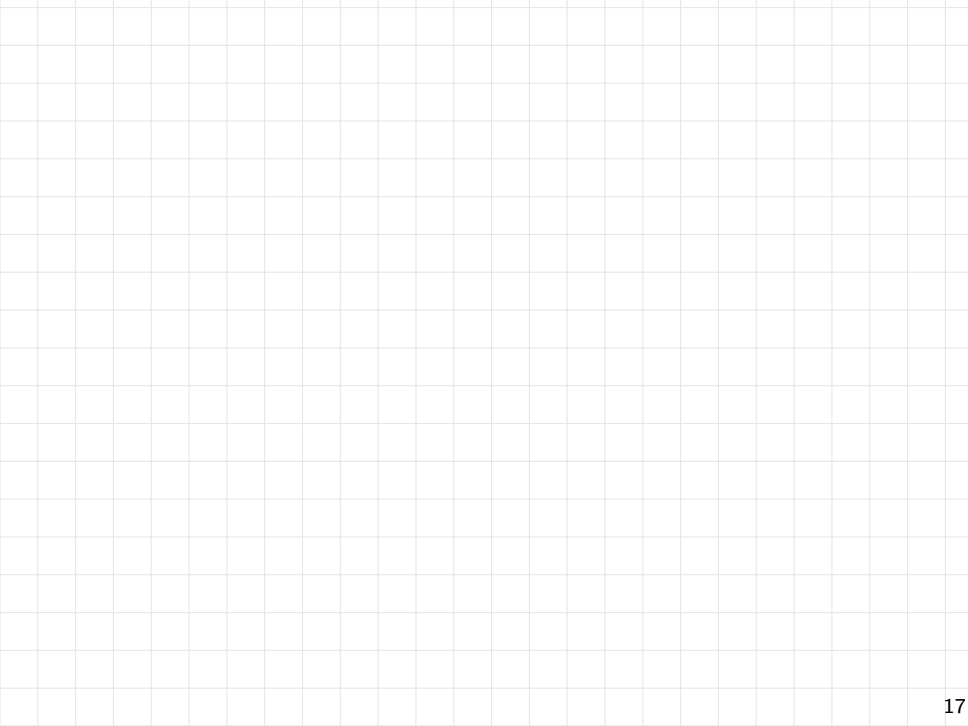
Esercizio

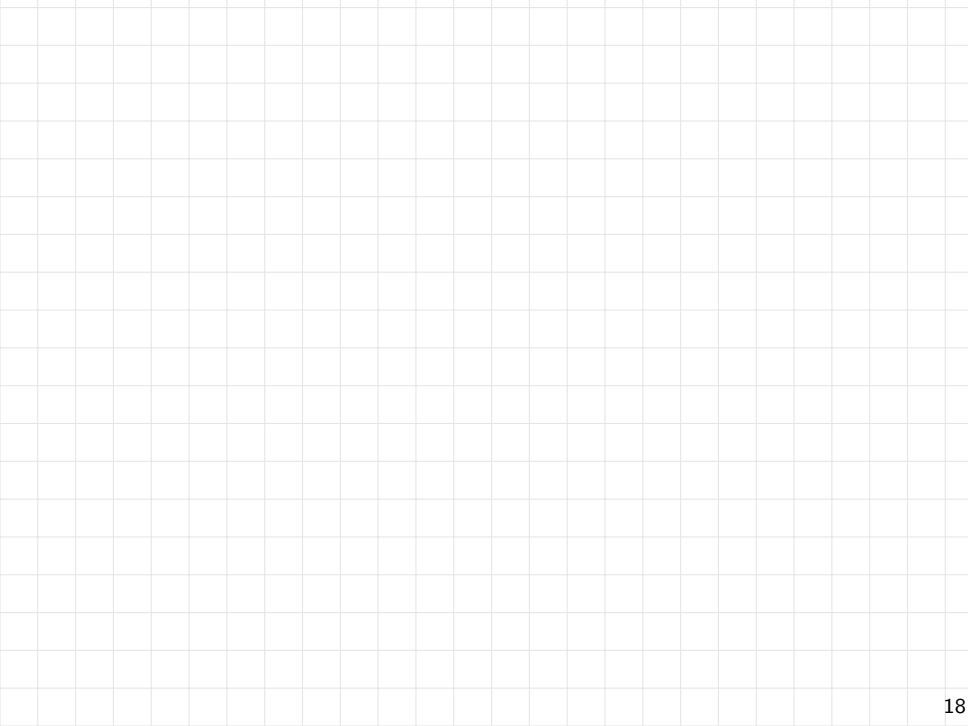
Sia $k = 2^d$, per un qualche intero $d > 0$, e siano $S_1, S_2, \dots, S_k$ sequenze ordinate di taglia m ciascuna. Si vuole progettare un algoritmo efficiente per fondere le sequenze in un'unica sequenza ordinata di taglia $n = k \cdot m$.

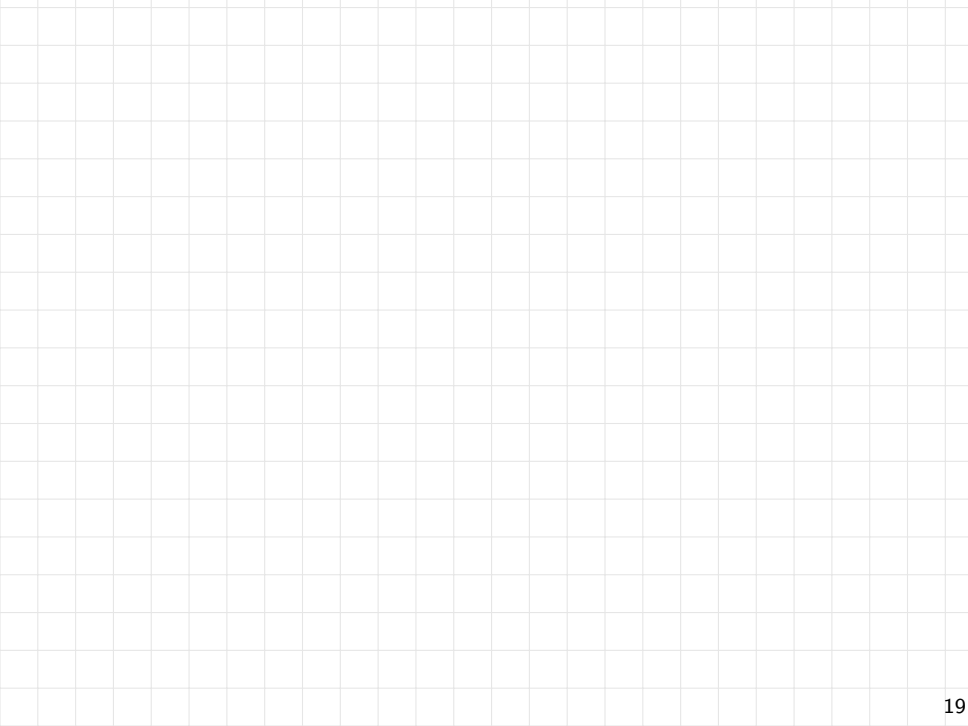
- 1 Analizzare la complessità dell'algoritmo banale che fonde S_1 con S_2 , poi $S_1 \cup S_2$ con S_3 , poi $S_1 \cup S_2 \cup S_3$ con S_4 ecc.
- 2 Progettare e analizzare un algoritmo più efficiente di quello banale.

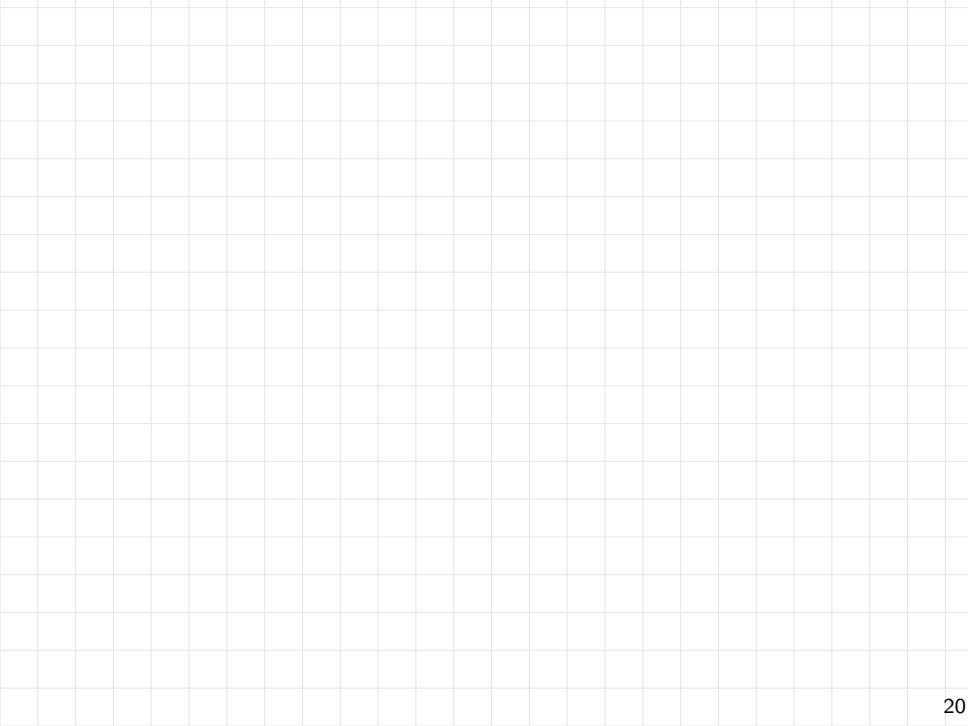












Esercizio

Siano $S_1 = S_1[0]S_1[1] \dots S_1[n_1 - 1]$ e $S_2 = S_2[0]S_2[1] \dots S_2[n_2 - 1]$ due sequenze ordinate di chiavi intere. In ogni sequenza le chiavi sono distinte, ma una stessa chiave può comparire sia in S_1 che in S_2 . Progettare una modifica dell'algoritmo Merge che restituisca il numero di chiavi in $S_1 \cap S_2$. L'algoritmo deve usare spazio aggiuntivo costante oltre alle due sequenze di input.

Esercizio C-13.42 [GTG14]

distinti

Sia S una sequenza di n elementi sui quali è definito un ordine totale. Si ricordi che una *inversione* in S è una coppia di elementi $S[i], S[j]$ tali che $j < i$ ma $S[j] > S[i]$. Descrivere un algoritmo che determina il numero di inversioni in S in tempo $O(n \log n)$.
(**Suggerimento:** adattare MergeSort.)

QuickSort

Strategia Divide-and-Conquer:

① **Divide:** se $n > 1$ (caso base $n \leq 1$):

$p \leftarrow S[n-1]$ (pivot)

$L \leftarrow \{\text{chiavi} < p \text{ in } S\}$

$E \leftarrow \{\text{chiavi} = p \text{ in } S\}$

$G \leftarrow \{\text{chiavi} > p \text{ in } S\}$

② **Conquer:** ordinamento ricorsivo delle due sottosequenze

- QuickSort(L)
- QuickSort(G)

③ **Combine:** $S \leftarrow L \circ E \circ G$, dove $\circ =$ “concatenazione”.

Implementazione in-place di QuickSort

Algoritmo QuickSortInPlace(S, a, b)

Input: sequenza S di n chiavi, indici $0 \leq a, b < n$

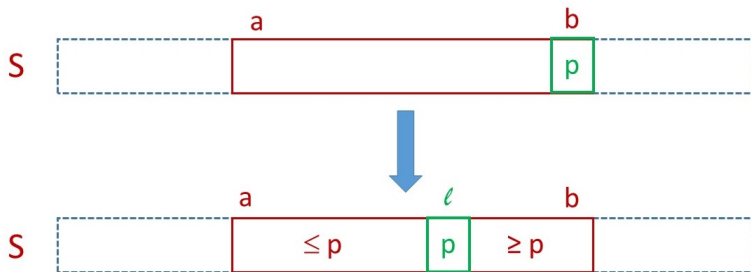
Output: sequenza S ordinata in senso non decrescente tra $S[a]$ e $S[b]$

```
/* Divide */
if  $a \geq b$  then return;
 $\ell \leftarrow \text{Partition}(S, a, b)$ ;
/* Conquer */
QuickSortInPlace( $S, a, \ell - 1$ );
QuickSortInPlace( $S, \ell + 1, b$ );
/* Combine */
```

Osservazioni:

- Per ordinare tutta la sequenza si invoca QuickSortInPlace($S, 0, n - 1$).
- Partition(S, a, b) riorganizza le chiavi in $S[a \div b]$ intorno a un pivot p , separando quelle $\leq p$ da quelle $\geq p$, e restituendo l'indice ℓ del pivot.

Immagine che rappresenta la relazione input/output di Partition



Partition

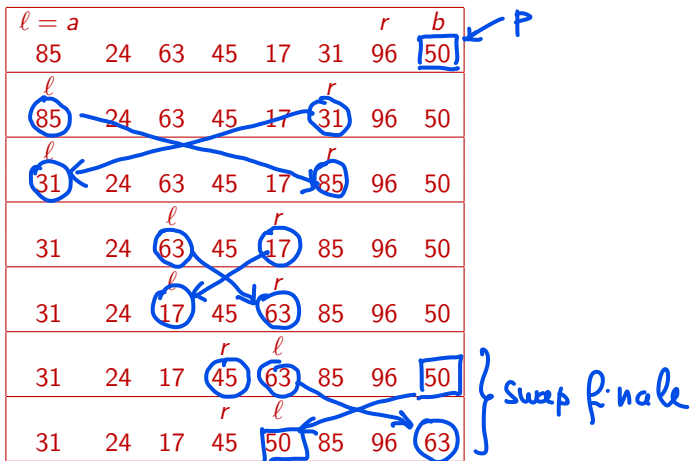
Algoritmo Partition(S, a, b)

Input: sequenza S di n chiavi, indici $0 \leq a < b < n$

Output: sequenza S riorganizzata tra a e b , e indice $\ell \in [a, b]$ tale
che $S[i] \leq S[\ell] \leq S[j]$ per ogni i, j con $a \leq i \leq \ell \leq j \leq b$

```
 $p \leftarrow S[b]$  ;                               /* Scelta del pivot */  
 $\ell \leftarrow a$ ;  $r \leftarrow b - 1$ ;  
while ( $\ell \leq r$ ) do  
    while ( $(\ell \leq r)$  AND  $(S[\ell] \leq p)$ ) do  $\ell \leftarrow \ell + 1$ ;  
    while ( $(\ell \leq r)$  AND  $(S[r] \geq p)$ ) do  $r \leftarrow r - 1$ ;  
    if ( $\ell < r$ ) then swap( $S[\ell], S[r]$ );  
swap( $S[\ell], S[b]$ );  
return  $\ell$ 
```

Esempio (Partition)



Correttezza di QuickSortInPlace

È immediata conseguenza della correttezza di Partition

CORRETTEZZA di PARTITION: basato sul seguente invariante che vale alla fine di ogni iterazione del while esterno



$$* \quad p = S[b]$$

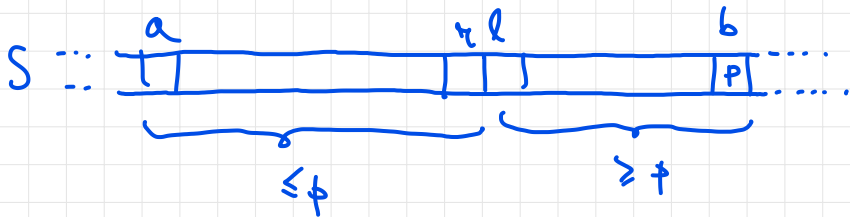
$$* \quad S[j] \leq p \quad \forall a \leq j < l$$

$$* \quad S[j] \geq p \quad \forall r < j \leq b$$

$$* \quad l \leq r+1$$

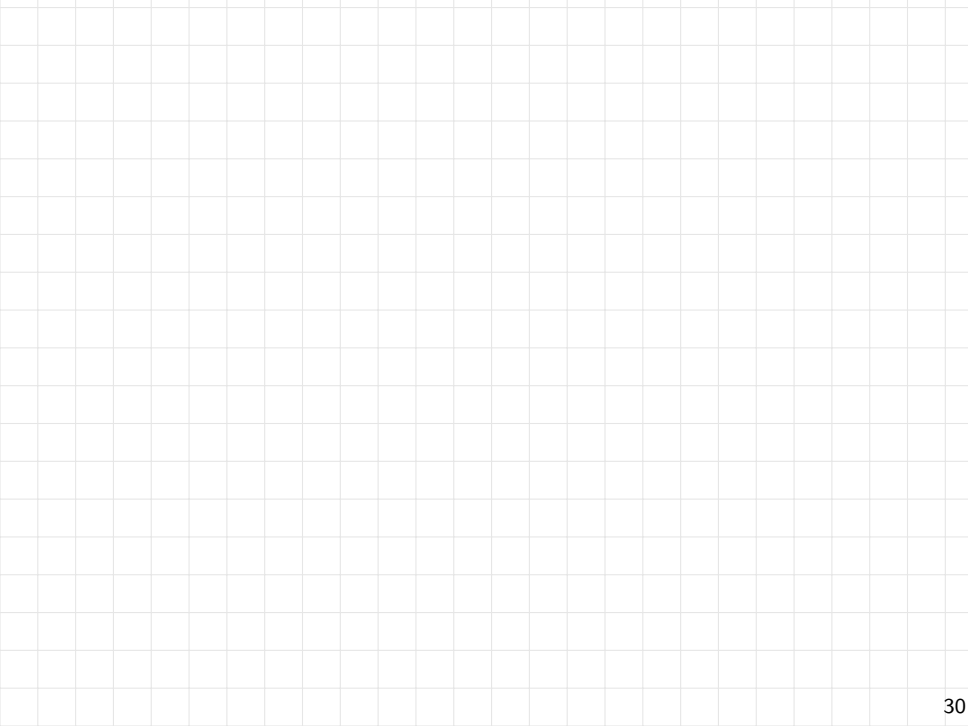
È facile dimostrare che l'invariante è mantenuto alla fine di ogni iterazione del while esterno.

Quando il while esterno termina la sua esecuzione, avviene che $l = r+1$, e a quel punto S sarà nel seguente stato



Tutto questo grazie all'invariante.

Finito il while esterno, l'istruzione $\text{Swap}(S[b], S[l])$ assicura che l'output di Partition (cioè l'indice l) è corretto, in base alla specifica input/output $\square$



Complessità di Partition

Proposizione

La complessità di `Partition`(S, a, b) è $\Theta(b - a + 1)$, ovvero lineare nel numero di chiavi presenti tra gli indici a e b .

La proposizione discende dalle seguenti osservazioni:

- Ogni due iterazioni consecutive del while esterno:
 - l'indice ℓ cresce e l'indice r decresce;
 - il numero di operazioni eseguito è proporzionale al numero di incrementi/decrementi dei due indici ℓ e r .
- Il numero totale di incrementi/decrementi dei due indici ℓ e r è proporzionale alla lunghezza della sottosequenza: $b - a + 1$.

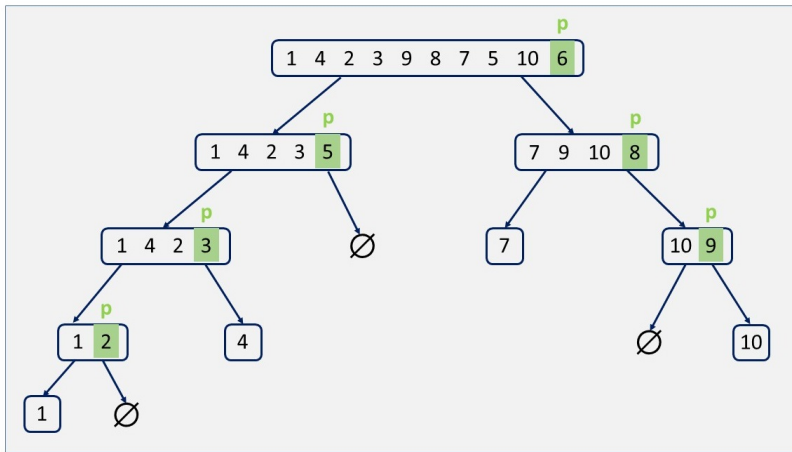
Proposizione

La complessità di `QuickSortInPlace`($S, 0, n - 1$) è $\Theta(n^2)$.

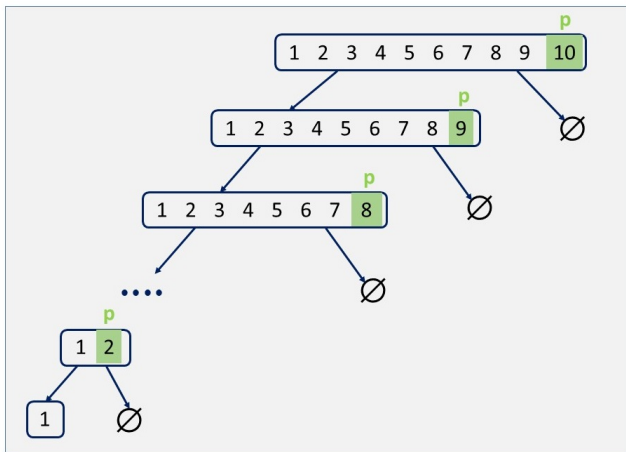
Per l'analisi di complessità dobbiamo considerare l'albero della ricorsione.

Prima di procedere con l'analisi facciamo due esempi.

Albero della ricorsione per $S = 1\ 4\ 2\ 3\ 9\ 8\ 7\ 5\ 10\ 6$



Albero della ricorsione per $S = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$



Consideriamo l'albero delle ricorsione per una istanza arbitraria di taglia n

- * Il numero di livelli dell'albero delle ricorsione è $\leq n$, perché da un livello al successivo almeno 1 pivot è sistemato definitivamente
- * Per questo motivo le tabelle aggregate delle istanze del livello i è $\leq n-i$
- * Il costo contribuito da un nodo dell'albero delle ricorsione è lineare nelle tabelle delle

istanza a cui è associato, in quanto è dominato
dal costo di Partition per tale istanza

$\Rightarrow$ Nell'albero delle ricorrenze abbiamo

* $\leq n$ livelli.

* il livello i contribuisce un costo $O(n-i)$

$\Rightarrow$ Complessità: $O\left(\sum_{i=0}^{n-1} (n-i)\right) = O(n^2)$

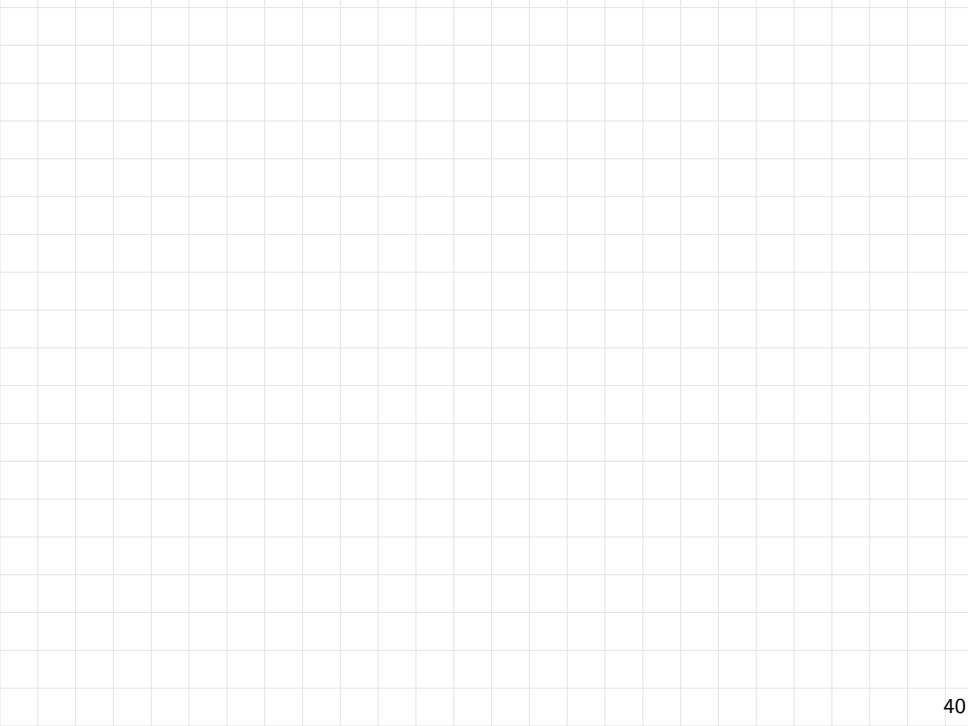
Dimostriamo ora che la complessità è $\Omega(n^2)$
che quindi implicherà che essa è $\Theta(n^2)$

Consideriamo come istanza S le sequenze di n chiacchi ordinate. Generalizzando l'Esempio 2 si vede facilmente che per questa istanza l'albero delle ricorrenze ha ESATTAMENTE n livelli, e il costo contribuito dal livello i è $\Theta(n-i)$ e quindi questa istanza richiederà $\Theta\left(\sum_{i=0}^{n-1} (n-i)\right) = \Theta(n^2)$ operazioni, dimostrando che la complessità dell'algoritmo è $\Omega(n^2)$

OSSERVAZIONE: QuickSort in Place

- * è in place rispetto alla memoria Heap che mantiene le chiavi sempre nelle seguenti posizioni di input
- * non è in place rispetto alla memoria Stack in quanto delle variabili a, b, l essa mantiene una copia per ogni chiamata ricorsiva attiva, e il numero di chiamate attive in uno stesso istante può essere pari all'altezza dell'albero

delle ricorrenze che, come abbiamo visto, per
certi istanti può essere n



Randomized QuickSort

Randomized QuickSort: variante di QuickSortInPlace che

Evita il caso pessimo probabilisticamente

Dentro Partition, sostituisce l'istruzione

$p \leftarrow S[b]$

con le seguenti tre istruzioni

$i \leftarrow \text{intero random in } [a, b];$

$\text{swap}(S[i], S[b]);$

$p \leftarrow S[b]$

Osservazione: Per ogni istanza S esistono molte esecuzioni possibili e quindi molti diversi tempi di esecuzione, funzione delle scelte probabilistiche (scelta del pivot) fatte dall'algoritmo.

Analisi di Randomized QuickSort

Randomized QuickSort è un algoritmo probabilistico in quanto esistono diverse possibili esecuzioni per ogni istanza, che sono determinate dalle scelte dei pivot.

Sia $t_{i,\text{RQS}}$ la variabile aleatoria che rappresenta il numero di operazioni eseguite da Randomized QuickSort per risolvere l'istanza i .

Proposizione (senza prova)

Per ogni istanza i di taglia n si ha che

$$E[t_{i,\text{RQS}}] \in \Theta(n \log n) \quad (\text{complessità in media})$$

$$\Pr(t_{i,\text{RQS}} \in O(n \log n)) \geq 1 - \frac{1}{n} \quad (\text{complessità in alta probabilità})$$

La proposizione rende quantitativamente evidente il fatto che, in pratica, la performance di Randomized QuickSort *per una qualsiasi istanza di input* è paragonabile a quella di MergeSort e dei miglior algoritmi basati su confronti.

Esercizio

Progettare un algoritmo che ordini una sequenza di n bit $S = S[0]S[1] \dots S[n-1]$ in tempo $\Theta(n)$. L'algoritmo deve essere in-place e utilizzare un solo ciclo.

Esercizio

Svolgere i seguenti punti:

- 1 Dire per quali istanze QuickSortInPlace esegue fedelmente la strategia L-E-G descritta all'inizio.
- 2 Dire come si comporta QuickSortInPlace su un'istanza con tutte chiavi uguali.
- 3 Modificare QuickSortInPlace in modo che implementi fedelmente la strategia L-E-G *per tutte le possibili istanze*, e dire come si comporta la modifica su un'istanza con tutte chiavi uguali.

Esercizio C-13.48 [GTG14]

Sia dato un insieme A di n viti e un insieme B di n dadi. Ogni vite va bene per un solo dado. Progettare un algoritmo per trovare tutte le coppie (vite,dado) giuste avendo a disposizione una primitiva che data una vite a e un dado b decide in tempo costante se a è *piccola*, *giusta*, o *grande* per b . Inoltre, dare una stima in ordine di grandezza del numero di volte in cui la primitiva viene invocata.

InsertionSort

Algoritmo InsertionSort(S)

Input: sequenza S di n chiavi

Output: sequenza S ordinata in senso crescente

```
for  $i \leftarrow 1$  to  $n - 1$  do
    curr  $\leftarrow S[i]$ ;
     $j \leftarrow i - 1$ ;
    while (( $j \geq 0$ ) AND ( $S[j] > \text{curr}$ )) do
         $S[j + 1] \leftarrow S[j]$ ;
         $j \leftarrow j - 1$ ;
     $S[j + 1] \leftarrow \text{curr}$ ;
```

Osservazione: InsertionSort è un algoritmo **in-place**

Complessità di InsertionSort

Analisi standard:

- **Complessità** $O(n^2)$. Nell'iterazione i del **for** il ciclo **while** esegue **al più** i iterazioni di costo $\Theta(1)$ ciascuna. La complessità totale risulta quindi essere

$$O\left(\sum_{i=1}^{n-1} i\right) = O(n^2).$$

- **Complessità** $\Omega(n^2)$. Per l'istanza costituita da chiavi distinte inizialmente ordinate in senso decrescente si ha che nell'iterazione i del **for** il ciclo **while** esegue **esattamente** i iterazioni di costo $\Theta(1)$ ciascuna.

Complessità di InsertionSort: analisi più fine

Definizione

Data una sequenza di n chiavi S , una **inversione**, rispetto all'ordinamento crescente, è una coppia di indici (i, j) con $0 \leq i \neq j < n$ tale che $j < i$ e $S[j] > S[i]$.

Osservazione: Sia K il numero di inversioni in una sequenza S di n chiavi. È immediato vedere che

$$0 \leq K \leq \binom{n}{2} = n(n-1)/2.$$

Goal: analisi della complessità di InsertionSort in funzione della taglia e del numero di inversioni della sequenza S .

* S ordinate in senso crescente $\Rightarrow K=0$

* S ordinate in senso decrescente $\Rightarrow K = \binom{n}{2}$

Complessità di InsertionSort

Proposizione

La complessità di $\text{InsertionSort}(S)$ è $\Theta(n + K)$, dove n è il numero di chiavi e K il numero di inversioni in S .

Osservazioni:

- L'analisi non contraddice quella standard, dato che si può avere $K = \Theta(n^2)$, ma evidenzia il fatto che InsertionSort è molto efficiente quando la sequenza di input è quasi ordinata (ovvero, ha poche inversioni)
- L'introduzione di K permette una partizione più fine delle istanze e quindi una più precisa analisi della complessità.

Dimostrazione della Proposizione

Sia S una sequenza di n chiavi con K inversioni.
Le operazioni eseguite da Insertion Sort(S) sono
raggruppabili in 2 termini.

A: Operazioni eseguite nel cd while

B: Tutte le operazioni eseguite fuori
del while.

È immediato vedere che $B \in \Theta(n)$

Per ogni $1 \leq i \leq n-1$ definiamo

$k_i = \# \text{indici } j < i \text{ tali che } S[j] > S[i]$ (cioè tali che (i, j) è una inversione). È facile vedere che il costo del while eseguito nella iterazione i del for è $\Theta(k_i)$

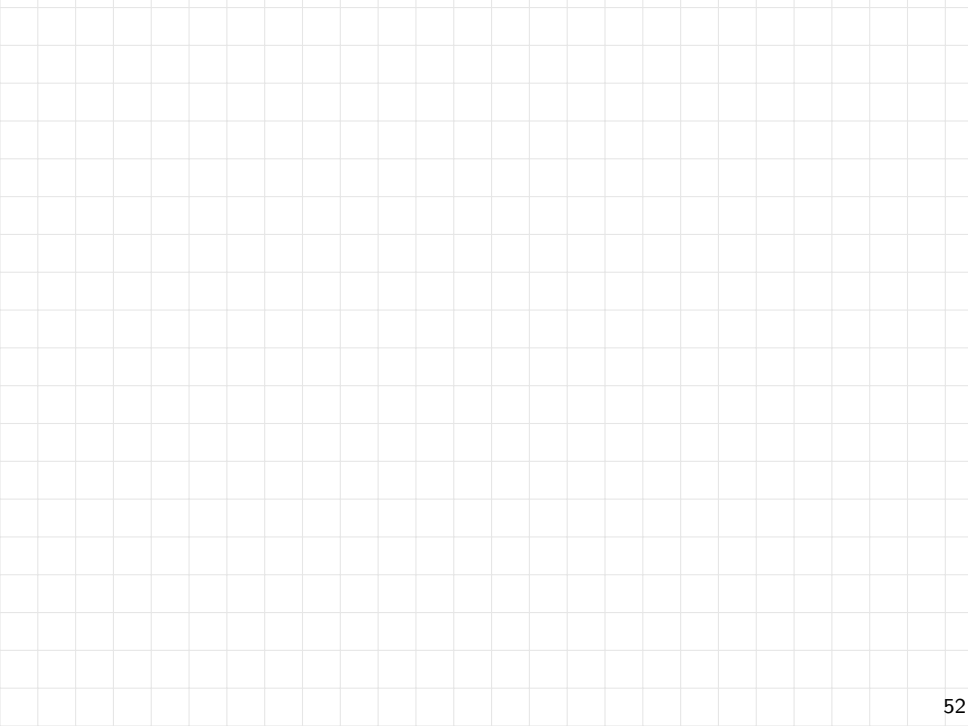
$$\Rightarrow A \in O\left(\sum_{i=1}^{n-1} k_i\right) = O(K)$$

dato che ogni inversione è contata in un solo k_i , ed esattamente una volta

$\Rightarrow$ la complessità di InsertionSort è

$$\Theta(A+B) = \Theta(n+K)$$





Lower Bound per Algoritmi Basati su Confronti

Teorema

Per istanze di taglia n , un qualsiasi algoritmo di ordinamento A basato su confronti richiede $\Omega(n \log n)$ confronti al caso pessimo.

Dato che MergeSort, HeapSort, (Randomized) QuickSort e InsertionSort sono basati su confronti, dal lower bound consegue che

- MergeSort, HeapSort e Randomized QuickSort hanno complessità asintoticamente ottima.
- QuickSort (deterministico) e InsertionSort non hanno complessità asintoticamente ottima ma per InsertionSort si può definire una famiglia abbastanza ampia di istanze per cui esso risulta molto efficiente.

Dimostrazione

Restringiamo il dominio delle istanze, per ogni taglia n , all'insieme $\mathcal{I}(n)$ delle sequenze S costituite dagli n interi $0, 1, \dots, n-1$, in qualche ordine

$\Rightarrow n!$ **istanze**

Oss.: se dimostriamo il lower bound per le sequenze di $\mathcal{I}(n)$, esso varrà, a maggior ragione, per tutte le possibili istanze.

Dato che in queste istanze le chiavi sono tutte distinte, ogni confronto $S[i]$ vs $S[j]$, con $i \neq j$, ha 2 soli esiti possibili:

- $S[i] > S[j]$, oppure
- $S[i] < S[j]$

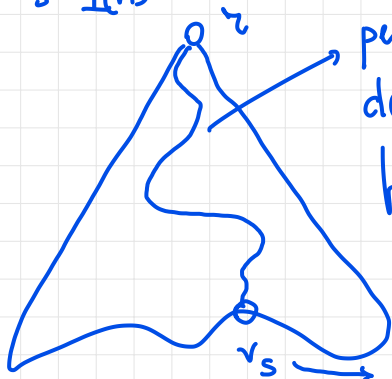
L'esecuzione di A su una qualsiasi sequenza $S \in \mathcal{I}(n)$ può essere rappresentata da un **DECISION TREE** $T_A(n)$, che è un albero binario proprio definito come segue:

- Ogni **nodo interno** è associato a un possibile confronto $S[i]$ vs $S[j]$ eseguito da A. Dopo tale confronto, A procede l'esecuzione
 - sul figlio sx, se $S[i] < S[j]$
 - sul figlio dx, se $S[i] > S[j]$
- La **radice** r è associata al **primo confronto** fatto dall'algoritmo.
- Ogni **foglia** v rappresenta una terminazione dell'algoritmo ed è associata a una **permutazione** che serve per ordinare le chiavi in input, e che è **determinata univocamente dai confronti eseguiti lungo il cammino da r a v e dai loro esiti.**

Osservazione cruciale 1

Per ogni sequenza $S \in \mathcal{I}(n)$ esiste un unico cammino in $T_A(n)$ dalla radice r a una foglia v_S che è associata alla permutazione che ordina S .

$S \in \mathcal{I}(n)$



cammino da r a v_S
determinato dai confronti
fatti da A per l'istanza S ,
e dei br es.t.

Foglia associata alla
permutazione che ordina S

Osservazione cruciale 2

Date due sequenze $S_1, S_2 \in \mathcal{I}(n)$, con $S_1 \neq S_2$ i cammini in $T_A(n)$ associati a esse devono portare per forza a due foglie v_{S_1}, v_{S_2} distinte, altrimenti si avrebbe una stessa permutazione che ordina sia S_1 che S_2 , che è impossibile.

E' questa osservazione che fa uso delle ipotesi che A è basato su confronti.

Dalle due osservazioni ricaviamo immediatamente che $T_A(n)$ deve avere $n!$ foglie distinte, una per ciascuna $S \in \mathcal{I}(n)$.

Ricordiamo che in un albero binario proprio con m foglie e altezza h deve valere che $m \leq 2^h$ (Slide 6 sugli Alberi Binari), e quindi

$$h \geq \log_2 m$$

Osserviamo inoltre che

$$n! \geq (\lfloor n/2 \rfloor)^{\lfloor n/2 \rfloor}$$

Quindi $T_A(n)$ deve avere altezza

$$h \geq \log_2 n! \geq \log_2 (\lfloor n/2 \rfloor)^{\lfloor n/2 \rfloor} \in \Omega(n \log n).$$

$= \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor$

Possiamo allora concludere che deve esistere una sequenza $S \in \mathcal{I}(n)$ per ordinare la quale A esegue $\Omega(n \log n)$ confronti. $\square$

Ordinamento
non basato su confronti

Assumiamo ora che la sequenza S da ordinare sia costituita da $n \geq 0$ entry e che l'ordinamento debba produrre in output la sequenza di entry ordinate per chiave (in senso crescente).

Gli algoritmi non basati su confronti che vedremo sono:

- **BucketSort** (chiavi intere in $[0, N - 1]$): crea N code (**bucket**) associate alle chiavi ed esegue **2 scansioni** della sequenza S
 - *I scansione*: memorizza le entry nei bucket in base al valore della chiave
 - *II scansione*: estrae le entry dai bucket ordinate.
- **RadixSort** (chiavi composte da d cifre intere in $[0, N - 1]$): esegue d volte BucketSort. È una generalizzazione di BucketSort.

BucketSort

Algoritmo BucketSort(S)

Input: sequenza S di n entry con chiavi intere in $[0, N - 1]$

Output: sequenza S con le entry ordinate per chiave crescente

Crea un array B di N code $B[0], B[1], \dots, B[N - 1]$;

for $i \leftarrow 0$ **to** $n - 1$ **do**

$k \leftarrow S[i].\text{getKey}();$
 $B[k].\text{enqueue}(S[i]);$

$i \leftarrow 0;$

for $k \leftarrow 0$ **to** $N - 1$ **do**

while $!(B[k].\text{isEmpty}())$ **do** $S[i++] \leftarrow B[k].\text{dequeue}();$

Correttezza: banale.

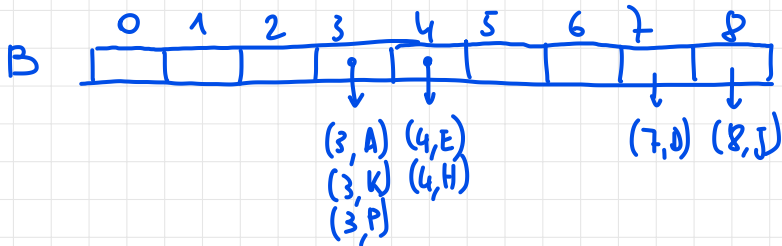
Oss.: i bucket sono implementati con politica FIFO (non serve per la correttezza di BucketSort ma servirà per la correttezza di RadixSort)

Esempio

$N=9$, $n=7$

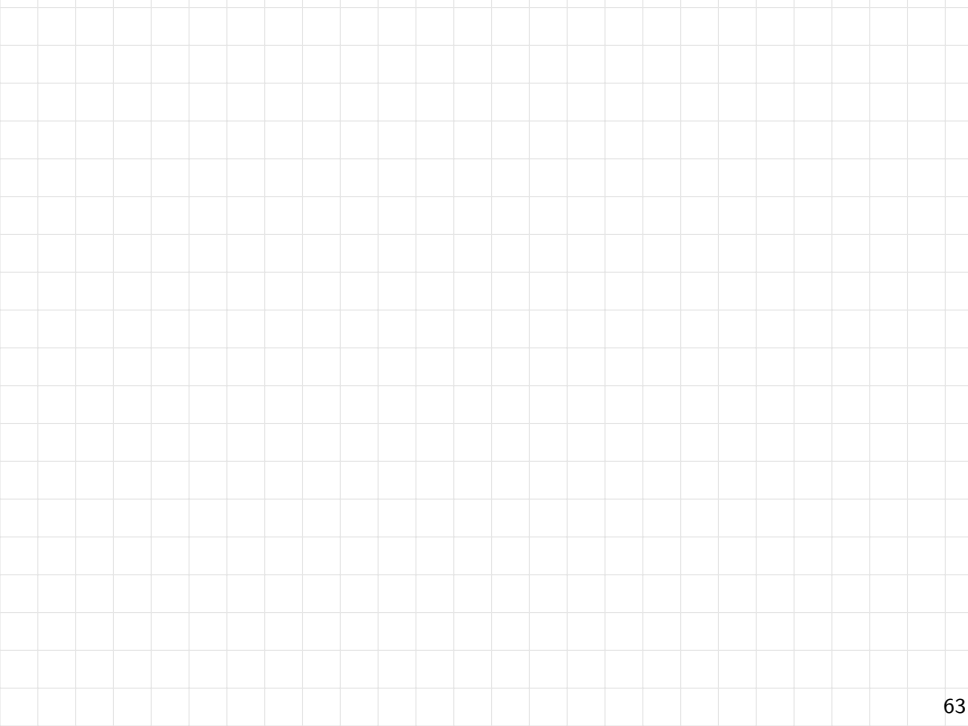
$S = (3, A)(7, D)(4, E)(3, K)(3, P)(8, J)(4, H)$.

Dopo il I ciclo for



Dopo il ciclo for

$S : (3, A)(3, K)(3, P)(4, E)(4, H)(7, D)(8, J)$



Complessità di BucketSort

Proposizione

La complessità di BucketSort(S) è $\Theta(n + N)$, dove n è il numero di entry in S e $[0, N - 1]$ è il range delle chiavi.

Dimostrazione

I ciclo for : $\Theta(n)$ operazioni assumendo che l'inserimento di una entry in un bucket richiede $\Theta(1)$ operazioni.

II ciclo for : detto $n_k = \# \text{entry in } B[k]$, si ha che

$$\sum_{k=0}^{N-1} n_k = n$$

* L'iterazione k del II ciclo richiede $\Theta(n_k + 1)$ operazioni (il "+1" serve se $n_k = 0$)

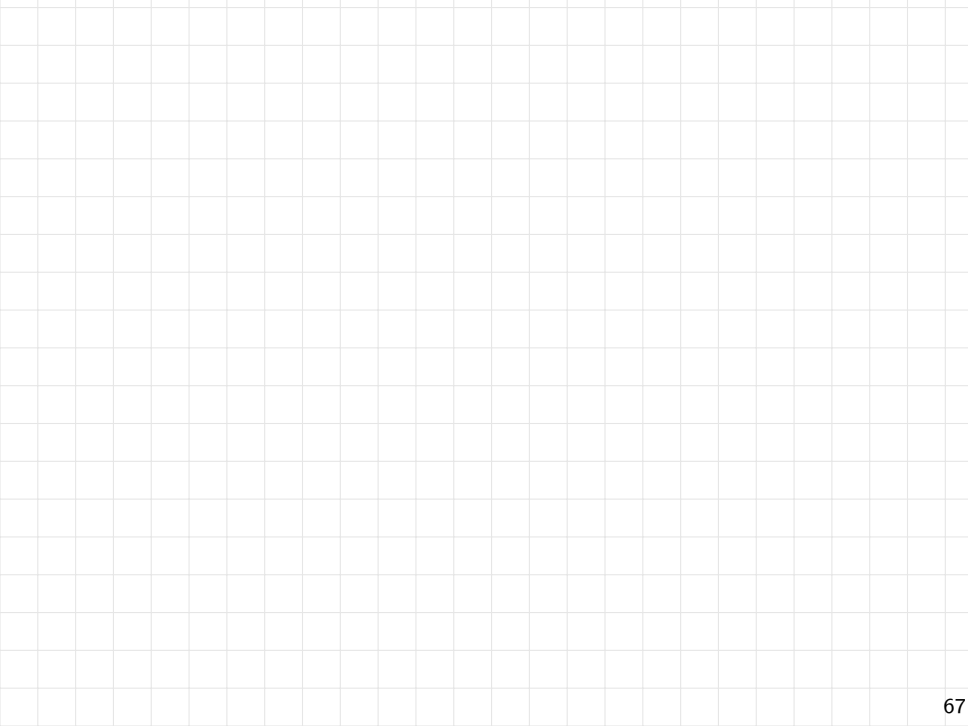
$\Rightarrow$ Il II ciclo richiede un numero di operazioni che è

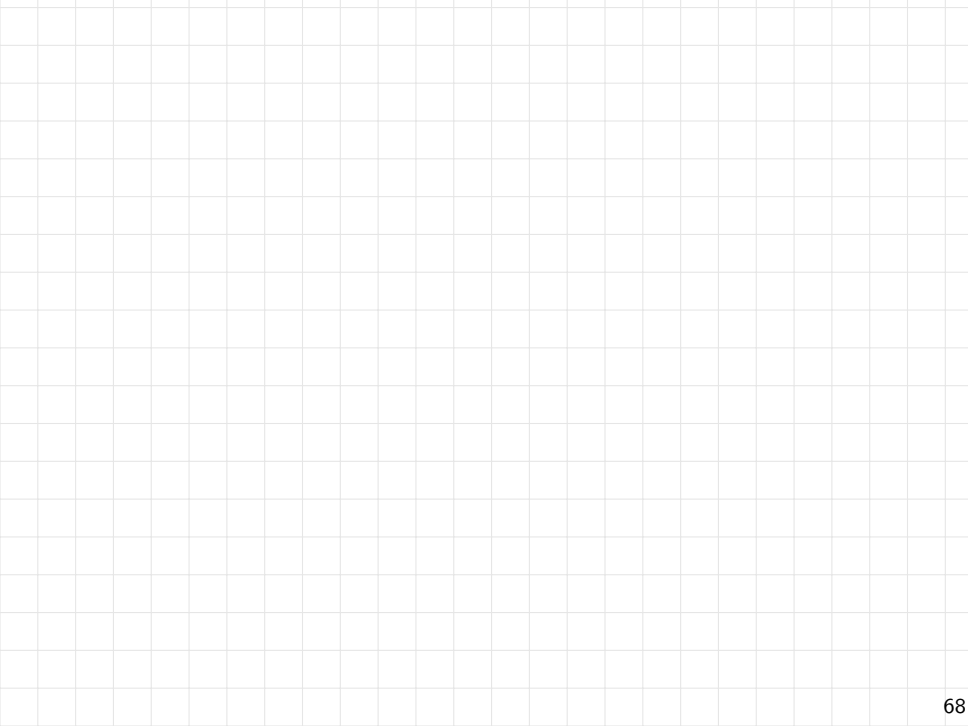
$$\Theta\left(\sum_{k=0}^{N-1} (n_k + 1)\right) = \Theta\left(\left(\sum_{k=0}^{N-1} n_k\right) + N\right)$$
$$= \Theta(n + N)$$

$\Rightarrow$ la complessità di BucketSort è $\Theta(n + N)$

OSSERVAZIONI

- * BucketSort è eff. vcente quando N non è troppo piccolo
- * Se $N = o(n \log n)$ allora BucketSort risulta ASINTOTICAMENTE PIÙ EFFICIENTE di qualsiasi algoritmo basato su confronti.





Ordinamento stabile

Definizione

Un algoritmo di ordinamento per sequenze di entry si dice **stabile** (**stable sorting**) se coppie di entry con la stessa chiave mantengono in output lo stesso ordine relativo che avevano in input

Proposizione

BucketSort è stabile

Dimostrazione

Conseguenza immediata della politica FIFO di accesso ai bucket.



RadixSort

- Generalizzazione di BucketSort (estende il range di linearità)
- Molto usato in applicazioni scientifiche
- **Input:** n entry con chiavi costituite da d cifre in $[0, N - 1]$, $c_{d-1}, c_{d-2}, \dots, c_1, c_0$, da ordinare *lessicograficamente*.
- **Idea:**
 - d fasi, dove l'output di una fase è input per quella successiva.
 - Ogni fase: **ordinamento stabile** su una cifra diversa.
 - L'ordine in cui vengono scelte le cifre è importante!

Qual è l'ordine giusto?

Esempio $n=4$, $d=2$, $N=10$

$$S = \overset{c_1}{\downarrow} \overset{c_0}{\downarrow} (29, A) (72, F) (63, H) (27, K)$$

OPZIONE 1: Partire dalla c-ke più significativa (c_1)

* I Fek: ordinamento stabile rispetto a c_1

$$\Rightarrow (29, A) (27, K) (63, H) (72, F)$$

* II Fek: ordinamento stabile rispetto a c_0

$$\Rightarrow (72, F) (63, H) (27, K) (29, A) \quad :-C$$

NON DÀ L'OUTPUT CORRETTO

OPZIONE 2: Partiti dalle cifre meno significative (C_0)

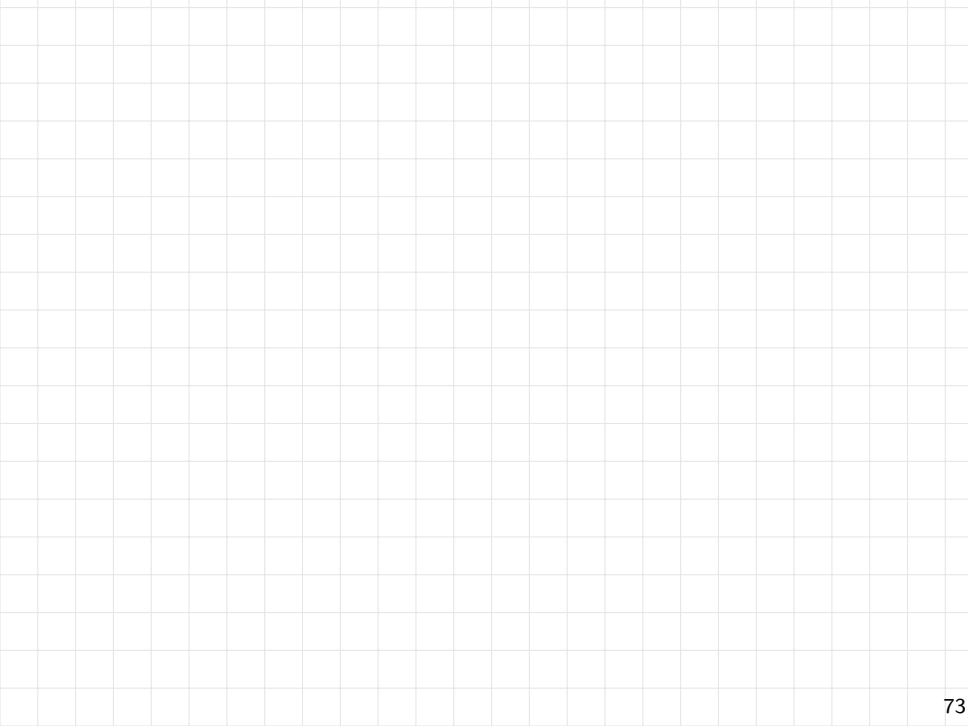
* I Fax: ordinamento stabile rispetto a C_0

$$\Rightarrow (72, F)(63, H)(27, K)(29, A)$$

* II Fax: ordinamento stabile rispetto a C_1

$$\Rightarrow (27, K)(29, A)(63, H)(72, F) \quad : -)$$

DÀ L'OUTPUT CORRETTO



RadixSort

Algoritmo RadixSort(S)

Input: sequenza S di n entry con chiavi:

$(c_{d-1}, c_{d-2}, \dots, c_1, c_0)$, dove $c_i \in [0, N-1]$, $\forall i$

Output: sequenza S con le entry ordinate lessicograficamente
per chiave

for $i \leftarrow 0$ **to** $d-1$ **do** ordina S in modo stabile rispetto a c_i ;

use stable BucketSort

N.B.: Le chiavi possono essere viste come interi in $[0, N^d - 1]$,
rappresentati in base N

Esercizio

Sia S una sequenza di 5 entry con le seguenti chiavi:

513, 116, 273, 506, 018. Far vedere l'ordine in cui compaiono le chiavi dopo ogni iterazione del ciclo for di RadixSort. Sfruttando l'intuizione ottenuta, dire, nel caso generale, che ordinamento sussiste tra le entry alla fine della Iterazione i del for

$$n = 5 \quad d = 3 \quad N = 10$$

Dopo l'iterazione i del for

0

513 273 116 506 018

1

506 513 116 018 273

2

018 116 273 506 513

l'esercizio suggerisce il seguente intervento che deve venire alla fine di ciascuna iterazione i del for , $i=0, 1, \dots, d-1$

le entry di S sono ordinate lessicograficamente rispetto alle cifre $c_i c_{i-1} \dots c_1 c_0$ delle chiavi

Correttezza di RadixSort

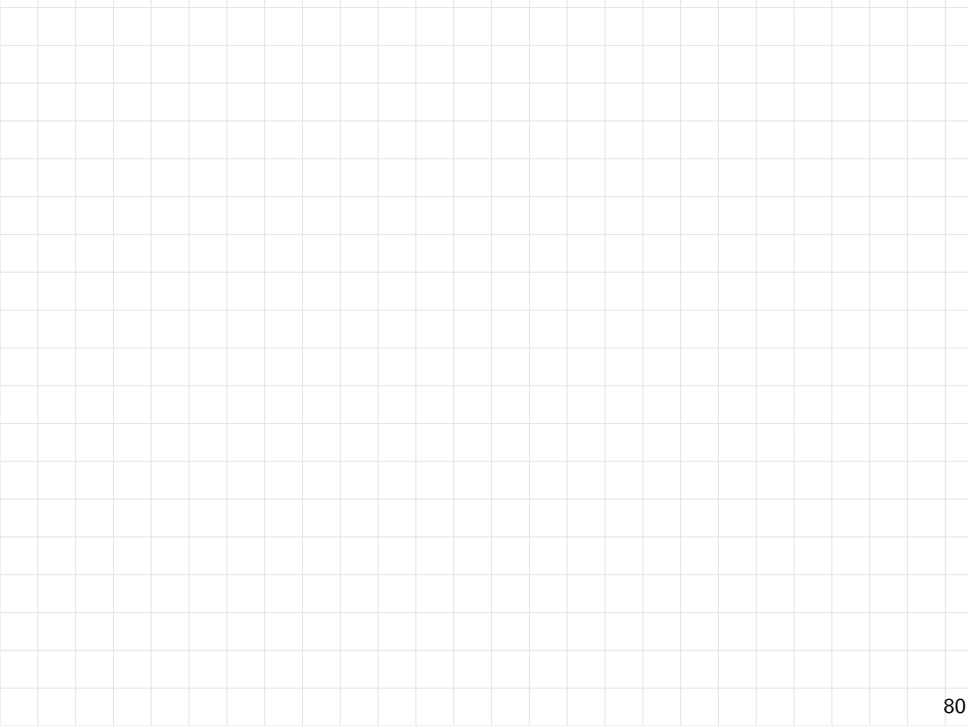
Proviamo a dimostrare tramite l'induzione della slide precedente

- * All'inizio del p (fine delle iterazioni -1) l'intervallo vale per vacuità
- * Supponiamo che valga alla fine delle iterazioni i e da questo le entry di S siano ordinate rispetto a $C_i C_{i-1} \dots C_1 C_0$
- Alle fine delle successive iterazioni $i+1$ si ha che

- 2 entry con valori diversi di C_{i+1} risultano unicamente ordinate rispetto a C_{i+1} e di conseguenza anche rispetto a $C_{i+1}, C_i, \dots, C_1, C_0$
- 2 entry con lo stesso valore di C_{i+1} risultano ordinate rispetto a $C_{i+1}, C_i, \dots, C_1, C_0$ grazie all'invariante che vale alla fine delle iterazioni i , e grazie all'ordinamento stabile rispetto a C_{i+1}

$\Rightarrow$ l'invariante vale alla fine delle iterazioni $i+1$

* Se l'invariante vale alla fine dell'ultima iterazione ($i = d-1$) è immediato vedere che la sequenza S è ordinata correttamente $\square$



Complessità di RadixSort

Supponendo di usare BucketSort per ordinare S in ciascuna iterazione del for, la prova della seguente proposizione risulta immediata.

Proposizione

La complessità di RadixSort(S) è $\Theta(d(n + N))$, dove n è il numero di entry in S e le chiavi sono costituite da d cifre in $[0, N - 1]$.

Osservazione: $d = O(1)$ e $N = O(n) \Rightarrow$ complessità $\Theta(n)$

In generale, se $d = \Theta(1)$ e $N = o(n \log n)$

Radix Sort può ordinare n entry con chiavi costituite da d cifre in $[0, N-1]$ (che possono essere viste come rappresentazioni di interi in $[0, N^d-1]$) con complessità $o(n \log n)$, batteando qualunque algoritmo basato su confronti.

Esempio

Stimiamo le complessità relative di MergeSort e RadixSort per ordinare tutti i cittadini italiani per codice fiscale

- Numero di cittadini $n \simeq 6 \cdot 10^7 \simeq 2^{26}$
- Codice Fiscale: stringa di 16 caratteri da un alfabeto di 36 (che associamo con gli interi in $[0, 35]$)

MergeSort: $\simeq 26 \cdot n$

RadixSort: Vediamo il CF composto da d cifre in $[0, N - 1]$.

Consideriamo due casi:

- ① $d = 16$, $N = 36$: la complessità è $\simeq 16 \cdot (n + N) \simeq 16 \cdot n$
- ② $d = 4$, $N = 36^4$: la complessità diventa $\simeq d \cdot (n + N) < 5 \cdot n$

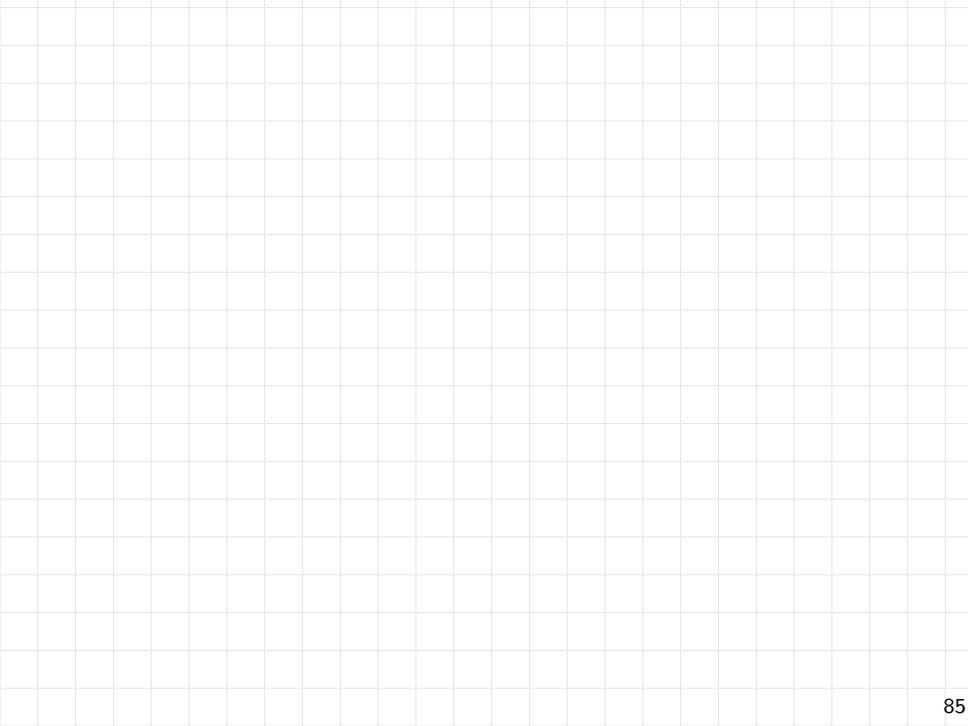
IN OGNI CASO VINCE RADIXSORT!

Il primo caso è ovvio.

Per il secondo caso si raggruppano i 16 caratteri del codice fiscale in 4 gruppi di 4



e ciascun gruppo, costituito da 4 caratteri, e quindi da 4 interi in $[0, 35]$ è visto come una cifra intera nel range $[0, 36^4 - 1]$



Esercizio C-13.39 [GTG14]

Dato un array A di n interi nell'intervallo $[0, n^2 - 1]$, descrivere un metodo semplice per ordinare A in tempo $O(n)$.

Valutiamo diverse alternative

1) Algoritmo basato su confronti: $\Omega(n \lg n)$

$\Rightarrow$ No

2) Bucket Sort: $\Theta(n + N)$

$N = n^2 \Rightarrow \Theta(n^2) \Rightarrow$ No

3) Radix Sort: $\Theta(d(n + N))$ d, N ?

Consideriamo diverse opzioni per d e N

- $d=1$, $N=n^2 \Rightarrow$ 2. ce d e un su
Bucket sort $\Rightarrow N^2$

- Se rappresentiamo gli interi con d cifre in $[0, N-1]$

Sappiamo che il massimo intero rappresentabile è
 $N^d - 1$, e quindi otteniamo scegliere N e d in
modo che $N^d \geq n^2$, che sarà vero

$$\Leftrightarrow d \log_2 N \geq 2 \log_2 n$$

$$\Leftrightarrow d \geq \frac{2 \log_2 n}{\log_2 N}$$

Allora, fissato N , conviene scegliere d più piccolo possibile
e quindi: $d = \lceil 2 \log_2 n / \log_2 N \rceil$

Esempio 1: $N=2 \Rightarrow d = \lceil 2 \log_2 n \rceil$ e

la complessità di Radix Sort diventa

$$\Theta(d(n+N)) = \Theta(n \log n)$$

che non va bene

Esempio 2 $N=n \Rightarrow d=2$ e la complessità
di Radix Sort diventa

$$\Theta(d(n+N)) = \Theta(2(n+n)) = \Theta(n) !!$$

Quindi l'algoritmo richiesto opera come segue:

- 1 Trovare ogni chiave k in S in una coppia di cifre (c_1, c_0) ciascuna in $[0, n-1]$, con

$$\left. \begin{array}{l} c_0 = k \bmod n \\ c_1 = \lfloor \frac{k}{n} \rfloor \bmod n \end{array} \right\} \begin{array}{l} \text{in generale:} \\ c_i = \lfloor \frac{k}{n^i} \rfloor \bmod n \end{array}$$

- 2 Ordinare S (con le nuove chiavi) usando Radix Sort

- 3 Ricostruire le chiavi della sequenza S ordinata in quelle originali: $c_1 c_0 \Rightarrow c_1 \cdot n + c_0$

Esercizio

Sia S una sequenza di n chiavi distinte $\in [0, 10n]$. Progettare un algoritmo di complessità lineare che restituisca l'indice i della *mediana*, cioè tale che esistono esattamente $\lfloor n/2 \rfloor$ chiavi più piccole di $S[i]$ in S .

Esercizio C-13.40 [GTG14]

Siano $S_1, S_2, \dots, S_k$ k sequenze non vuote di entry con chiavi nel range $[0, N - 1]$, per un qualche $N \geq 2$. Descrivere un algoritmo che ordini separatamente tutte le sequenze in tempo $O(n + N)$, dove n è la somma delle taglie delle sequenze. (In output le sequenze devono rimanere distinte e quindi l'esercizio non chiede di ordinare l'unione delle sequenze, che sarebbe banale.)

Esercizio

Siano $S_1[0 \div n - 1]$ and $S_2[0 \div n - 1]$ due array ordinati di n chiavi intere ciascuno. In ogni array le chiavi sono distinte, ma una chiave può comparire in entrambi gli array.

- 1 Progettare un algoritmo CountNeg, **contenente un solo ciclo**, che determina il *numero di chiavi negative distinte* che compaiono in S_1 e/o S_2 . Ad esempio, se $S_1 = [-3, -2, 0, 34]$ e $S_2 = [-7, -2, 34, 45]$ l'algoritmo restituisce 3 (chiavi negative distinte: -7, -3, -2). Per avere punteggio pieno, l'algoritmo deve utilizzare spazio costante, oltre a S_1 e S_2 .
- 2 Specificare un opportuno invariante che serva per provare la correttezza dell'algoritmo.

Riepilogo

- Algoritmi di ordinamento basati su confronti
 - Paradigma Divide-and-Conquer
 - MergeSort: specifica, complessità con albero della ricorsione
 - QuickSort deterministico: specifica (quasi) in-place, complessità con albero della ricorsione
 - InsertionSort: specifica, complessità in funzione del numero di chiavi e del numero di inversioni
 - Randomized QuickSort: specifica, complessità in alta probabilità e in media
 - Lower bound per algoritmi di ordinamento basati su confronti
- Algoritmi di ordinamento non basati su confronti
 - BucketSort
 - Ordinamento Stabile
 - RadixSort