Oltre il muro dei limiti inferiori

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **10(b)** **[31/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Ordinamenti "quasi" lineari

Abbiamo visto che ogni algoritmo di ordinamento **basato su confronti** deve essere $\Omega(n \log n)$.

Vedremo degli ordinamenti di complessità "**quasi lineare**" (**attenzione alla complessità**!) che ordinano **senza fare confronti**, ma al prezzo di **fare ipotesi più forti** sui valori del vettore.

In particolare, nel primo, il **Counting Sort**, si assume che i **valori** siano in un **intervallo di interi** $[0, k)$.

In tal caso, si riesce a ordinare gli elementi di un vettore in **tempo** $\Theta(n + k)$ usando **memoria** $\Theta(k)$.

Di conseguenza, questo metodo risulta **efficiente** quando, a sua volta, abbiamo $k \in \Theta(n)$.

Counting Sort: idea

Idea: dato un vettore $v[0, n)$ a **valori nell'intervallo** $[0, k)$ (ossia, visto v come insieme, ho che $v[0, n) \subseteq [0, k)$: v può contenere ripetizioni) **contare** $\forall i \in [0, k)$ **quanti elementi** hanno valore i.

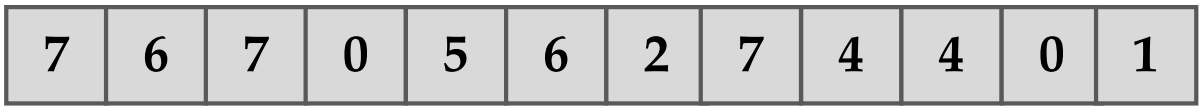
Quest'informazione può essere calcolata e memorizzata in un **vettore di contatori** $c[0, k)$ che avrà la seguente proprietà:

$$c[i] = \#\{\, j \in [0, n) \mid v[j] = i \,\}$$

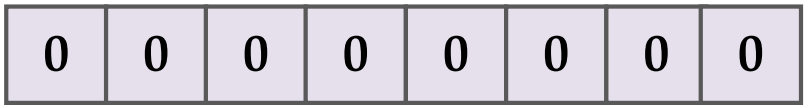
Nel vettore ordinato v', un intero $i \in [0, k)$ viene dopo tutti gli interi nell'intervallo $[0, i)$, il cui numero è $\sum_{j \in [0, i)} c[j] = \mathbf{S}(c[0, i))$. Osservate infine che $\sum_{j \in [0, k)} c[j] = n$, perché gli elementi di v sono per ipotesi tutti nell'intervallo $[0, k)$.

Quindi, usando il vettore c possiamo facilmente **ricostruire** il **vettore** v' **ordinato**: sarà sufficiente, per ogni $\forall i \in [0, k)$ scrivere nel vettore v l'intero i per $c[i]$ volte.

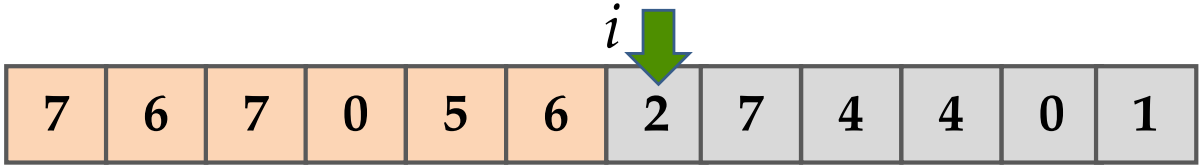
countingSort: esecuzione e invarianti



$v, n = 12, k = 8$

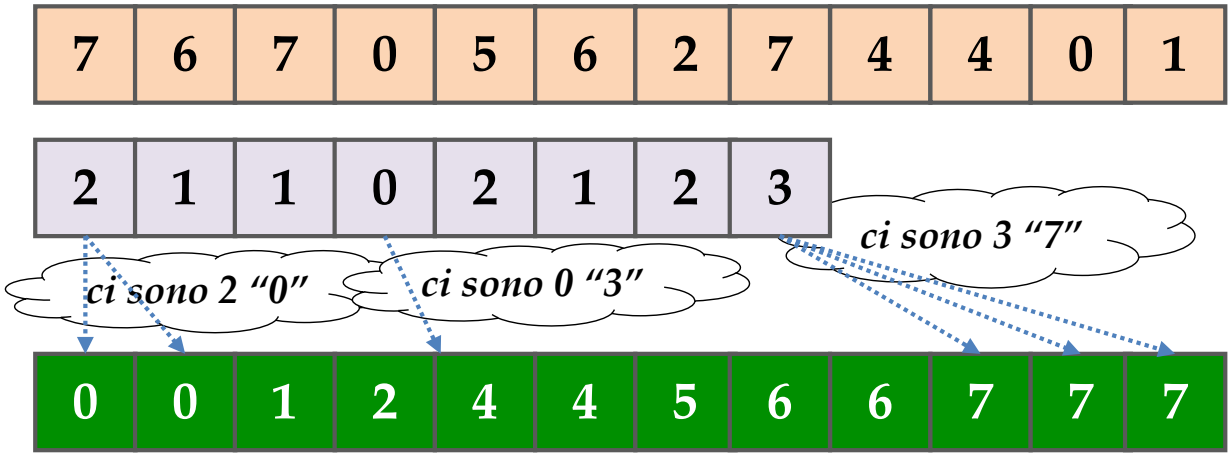


All'inizio si azzera c



Leggendo v, si aggiorna il vettore di contatori c mantenendo l'inv.:

$c[l]=\#\{j \in [0, i) \mid v[j]=l\}$



Finito il conteggio:

$c[l]=\#\{j \in [0, n) \mid v[j]=l\}$

Alla fine, l'informazione registrata su c, permette di **ricostruire** v

Counting Sort: pseudocodice

Da quanto detto, è sufficiente **produrre** il **vettore** c di **contatori**…

… e poi usarlo poi per **ricostruire** v ordinato.

```
def countingSort(v, k):
# REQ: v di interi in [0,k)
# ENS: v ordinato
    c = conta(v, k)
    ricostruisciCS(v, c)
```

```
def conta(v, k):
# REQ: v di interi in [0,k)
# ENS: c[j]=#{l ∈ [0,n)|v[l]=k}
    c, n = allZero(k), len(v)
    for i = 0 to n - 1:
    # INV:c[j]=#{l ∈ [0,i)|v[l]=j}
        c[v[i]] = c[v[i]] + 1
    return c
```

L'operazione di conteggio è facile.

Per ricostruire, è sufficiente copiare il valore i per $c[i]$ volte nel vettore v, andando sempre a destra.

$$\text{INV} \equiv v(\mathbf{S}(c[0, i)) = v \cap_{\text{ms}} [0, i)$$

```
def ricostruisciCS(v, c):
# REQ: c conteggio v
# ENS: v ordinato
    k, j = len(c), 0
    for i = 0 to k - 1:
        do c[i] times:
            v[j], j = i, j+1
```

Counting Sort: complessità

L'operazione di conteggio è chiaramente lineare nella lunghezza n di v: è un ciclo che fa n iterazioni, ciascuna di costo $\theta(1)$.

Da notare che l'allocazione e azzeramento del vettore di contatori è $\theta(k)$, anche se con costanti molto basse. Totale $\boldsymbol{\theta(n+k)}$ per ***conta***.

Un'**analisi affrettata** di ***ricostruisciCS*** potrebbe far pensare che la complessità sia $\theta(k \cdot n)$, essendo k il numero di iterazioni del **for** esterno ed n il massimo numero in un $c[i]$: tuttavia il ciclo interno viene eseguito $\mathbf{S}(c[0,k]) = n$ volte (**analisi aggregata**).

Quindi anche la complessità è $\boldsymbol{\theta(n+k)}$ anche per ***ricostruisciCS***.

$\theta(n+k)$

```
def conta(v, k):
# REQ: v di interi in [0,k)
# ENS: c[i]=#{l∈[0,n)|v[i]=l}
    c, n = allZero (k), len(v)
    for i = 0 to n - 1:
        c[v[i]] = c[v[i]] + 1
    return c
```

$\theta(k)$

$\theta(n)$

$\theta(n+k)$

```
def ricostruisciCS(v, c):
# REQ: c conteggio v
# ENS: v ordinato
    k, n = len(c), len(v)
    j = 0
    for i = 0 to k - 1:
        do c[i] times:
            v[j], j = i, j+1
```

$\theta(n+k)$

Counting Sort: dati satellite

Finora abbiamo sempre considerato il problema di **ordinare vettori di interi**, ritenendolo sufficientemente **generale**.

Ciò è senza dubbio **vero** per gli **algoritmi basati** su **scambi**, dove si scambiano tutte le informazioni contenute. **Qualche riflessione** in più è necessaria per ***`countingSort`***.

Usualmente nei **dati strutturati** (immaginate ad esempio i vostri dati nel database della Sapienza) c'è un **campo chiave** (ad esempio il numero di matricola) e una certa quantità di **dati satellite**.

Inoltre: posso essere interessato a **ordinare** un vettore **rispetto a diverse grandezze** utili in diverse applicazioni (matricola, alfabetico cognome, data di nascita, numero di esami o media etc.).

Se avete notato, ***`countingSort`*** **si limita a ricopiare interi**, senza tener conto di **eventuali dati satellite** presenti.

È possibile usare ***`countingSort`*** per ordinare vettori con dati satellite?

Counting Sort con dati satellite

È sufficiente riflettere sulle **post-condizioni** assicurate dalla funzione di **conteggio**.

Siccome $c[i] = \#\{\, j \in [0, n) \mid v[j]=i \,\}$, si deduce immediatamente anche:

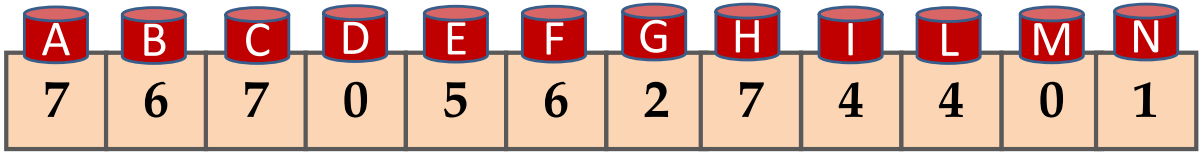
$$\mathbf{S}(c[0, l)) = \#\{\, j \in [0, n) \mid v[j] < l \,\}$$

e quindi i dati con chiave l vanno scritti **nel vettore ordinato** nelle $c[l]$ posizioni **[S**$(c[0, l))$, **S**$(c[0, l+1))$**)**. Questo intervallo è vuoto se $l \notin v$, e in tal caso infatti $c[l] = 0$ e **S**$(c[0, l)) =$ **S**$(c[0, l+1))$.

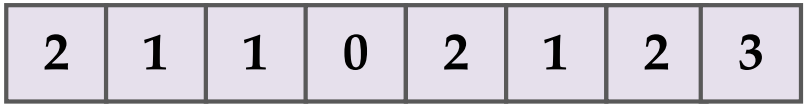
Dobbiamo scrivere le funzioni **`sommeCumulate`** e **`ricostruisciCS-DS`**.

```
def countingSort-DS(v, k):
# REQ: v di interi in [0,k)
# ENS: v ordinato
    c = contaKeys(v, k)
        -- uguale a conta, ma campo chiave
    sommeCumulate(c)
    ricostruisciCS-DS(v, c)
```

countingSort: esecuzione e invarianti

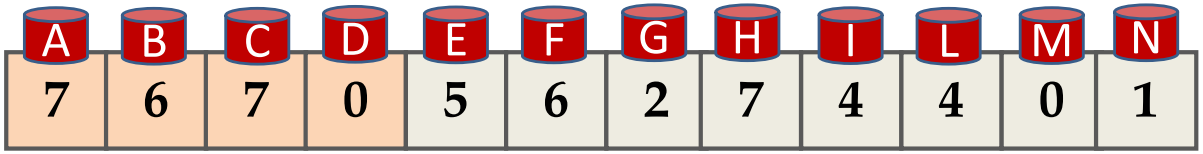


$v, n = 12, k = 8$

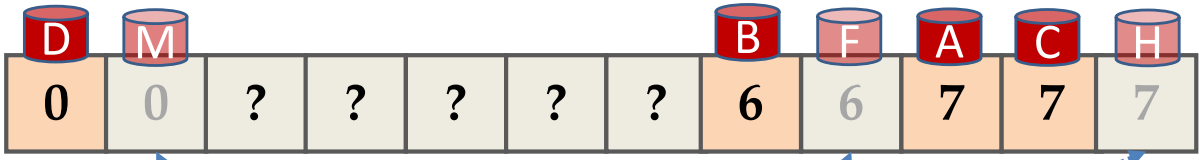


conteggio delle occorrenze di ciascuna chiave in *c* come prima

vettore *c* con le somme cumulate



dopo aver processato i primi 4 elementi…



… li ho messi al posto giusto usando *c*

c va **riaggiornato** in modo che sia sempre *c*[*i*] il **prossimo posto giusto** per un elemento di chiave *i*.

Somme cumulate

Facciamo questa operazione mantenendo una variabile s per cui valga sempre l'invariante $\varphi(j) \equiv s = \mathbf{S}(c[0, j))$.

Quindi, necessariamente $c[j]$ prende il valore di s.

Per mantenere l'invariante, dev'essere $s' = s + c[j]$. Anche questo programmino beneficia molto dell'uso dell'assegnamento parallelo (**altrimenti dovrei gestire una variabile di appoggio**).

Quello che gli **algoritmisti non capiscono**: "*sì, ma nell'assegnamento parallelo ci sono variabili temporanee invisibili al programmatore*".

- **Esercizio**: scrivere l'assegnamento parallelo in C.

```
def sommeCumulate(c):
# REQ: c conteggio
# ENS: c' | c'[i]=S(c[0,i))
    k = len(v)
    c[0], s = 0, c[0]
    for j=1 to k-1
        c[j], s = s, s+c[j]
```

Ricostruzione con dati satellite

Per la ricostruzione, scorrendo v con l'indice j, la chiave l = key[v[j]] ci dirà di scrivere i dati in posizione c[l].

Siccome il valore l **può non essere unico**, incremento c[l] affinchè sia mantenuto l'invariante che c[l] sia l'indice dove scrivere il prossimo elemento di chiave l.

Quest'operazione è distruttiva e quindi sono costretto a usare un vettore d'appoggio u.

Alternative? **Esercizio**.

```
def ricostruisciCS-DS(v, c):
# REQ: v di interi in [0,k)
# ENS: v ordinato
    n = len(v)
    u = alloca(n)
    for j=0 to n-1
        u[c[v[j].key]] = v[j]
        c[v[j].key] = c[v[j].key] + 1
    copia(u, v)
    dealloca(u)
```

Estensioni di Counting Sort: Radix Sort

Stabilità degli ordinamenti

Un algoritmo di ordinamento è **stabile** se **preserva** l'**ordine** di elementi con **chiavi uguali**.

Detto v il vettore originario e v' il vettore ordinato, dobbiamo quindi avere che:

$$key(v[i]) = key(v[j]) \;\&\; \mathbf{i < j} \;\&\; data(v'[i']) = data[v[i]] \;\&\; data(v'[j']) = data(v[j]) \;\textbf{allora}\; i' < j'$$

Esercizio: Verificare quali siano gli algoritmi stabili e quali no tra quelli studiati finora.

Esercizio (diseducativo): vedere se riuscite a modificare gli algoritmi stabili per renderli non stabili.

Esercizio: Riflettere se sia possibile (e come) rendere stabili quelli non stabili.

Esercizio: studiare una semplice strategia per assicurarsi che l'ordinamento sia stabile (indipendentemente dall'algoritmo).

Radix Sort e stabilità

Viene introdotto dal Cormen con un oscuro esempio di ordinamento di schede perforate (!? neanch'io le ho mai viste!!!).

In realtà è utile quando dobbiamo ordinare vettori **rispetto a più campi chiave** (e, vedremo, adattabile al caso generale).

⏵ **Esempio**: immaginate di voler ordinare dei valori rispetto alla **data**, rappresentata con **3 valori**: anno, mese e giorno.

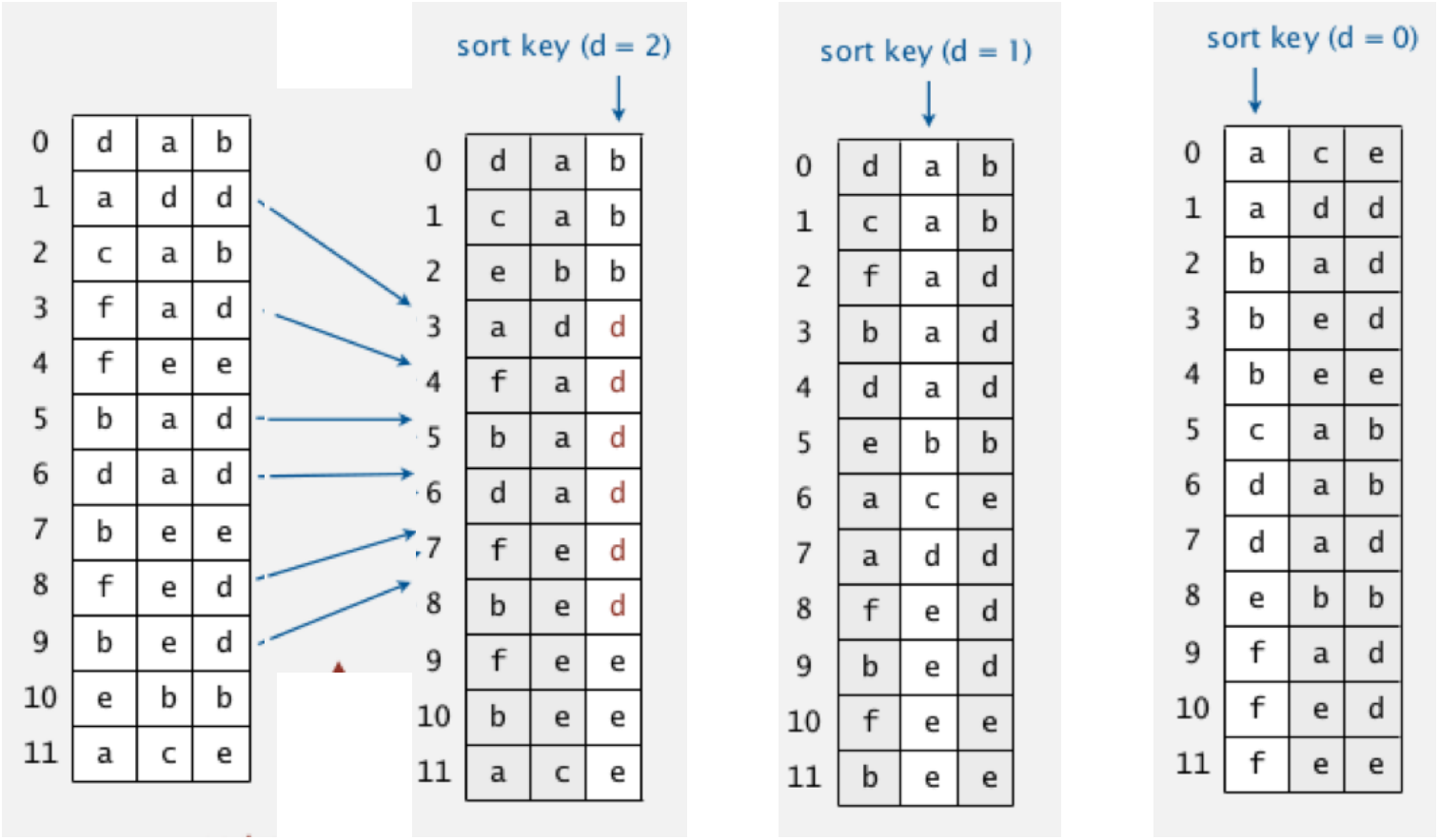
Ovviamente, posso definire una funzione **≤ sulle date** che prima guarda l'anno, in caso di parità il mese e infine il giorno in caso di ulteriore parità e **procedere con un qualsiasi** algoritmo di **ordinamento** già visto.

Ma possiamo fare un'altra cosa: **ordinare 3 volte** il vettore: **prima** rispetto **al giorno** (aspetto **contro-intuitivo**), poi rispetto al mese e **infine rispetto all'anno**: se l'algoritmo di ordinamento **è stabile** dopo l'ultima passata, in **caso di parità di anno**, rimarranno **prima elementi con mese minore** e in caso di ulteriore parità elementi con giorno minore (grazie agli ordinamenti precedenti e alla stabilità).

Radix Sort: esempio di esecuzione

Vediamo un esempio in cui si ordinano sequenze di caratteri (di ugual lunghezza).

Vedete anche graficamente cosa significa **stabilità**: in caso di parità non viene **mai invertito l'ordine** (le **frecce degli spostamenti di elementi uguali non si intersecano** – nella figura i **d**).



Correttezza & pseudocodice

Immaginiamo di ordinare rispetto a un vettore di chiavi *key* di lunghezza d (la chiave **più significativa** in posizione **0**).

Teorema: *Radix Sort ordina correttamente rispetto all'ordine lessicografico delle chiavi.*

Dim: Induzione su d. Se $d = 1$, la correttezza discende dalla correttezza dell'ordinamento usato.

Se $d > 0$, assumiamo radixSort abbia ordinato correttamente il vettore fino alla chiave $d - 1$.

Se $key_d(v[i]) < key_d(v[j])$ allora $i < j$ per correttezza di **`stableSort`**.

Se $key_d(v[i]) = key_d(v[j])$, la correttezza deriva dall'ipotesi induttiva di correttezza e dalla stabilità dell'algoritmo di ordinamento usato. □

```
def radixSort(v, d):
    k = len d
    for i = k-1 downto 0:
        stableSort(v, ≤d[i])
```

Radix Sort: Complessità

Siccome è ragionevole immaginare che le singole chiavi abbiano un insieme ragionevolmente piccolo k di valori, ogni passata può essere fatta con Counting Sort in tempo $\Theta(n + k)$.

La complessità è quindi $\Theta(d\ (n + k))$.

Possiamo usare Radix Sort anche sfruttando le b cifre, o caratteri o anche bit di ogni singola chiave: se $b < \log n$ avremo che Radix Sort è asintoticamente migliore in tempo degli algoritmi ottimi basati su confronti, come **`mergeSort`**.

Però se **$b < \log n$**, possiamo anche usare direttamente Counting Sort, perché in tal caso $2^b < 2^{\log n} = n$ quindi in questo caso **$\Theta(n)$**.

Se $b > \log n$ possiamo **raggruppare le cifre binarie** a gruppi di r bit. In tal caso, il costo di Counting Sort è $\Theta(n + 2^r)$ e occorrono b/r passate di Counting Sort a Radix Sort, ottenendo una complessità $\Theta((b/r)(n + 2^r))$.

Una scelta ragionevole è scegliere $r = \log n$, ottenendo un tempo di esecuzione $\Theta(bn/\log n)$: per valori maggior di r, il termine 2^r cresce rapidamente in relazione a n.

Utilizzo e Discussione

Va osservato che le costanti sono diverse però, e in particolare **Counting Sort** con dati satellite **non può essere fatto in place**.

Sotto ipotesi ragionevoli, è probabile che **`quickSort`** resti il migliore!

Esempio: se consideriamo la codifica binaria delle chiavi, nel caso in cui le chiavi **siano univoche**, è necessario che siano lunghe **almeno** $\log n$ (per avere chiavi tutte diverse), ma molto probabilmente molto di più (raramente si cerca una biiezioni – es: **Codice Fiscale**).

Viceversa, ci sono molti casi pratici in cui le **chiavi da ordinare** sono valori con **molte ripetizioni** (come visto nell'esempio delle **date**).

Morale: ogni "buon" algoritmo può avere uno specifico campo di applicazione in cui risulta il migliore.

Difficilmente ci sono algoritmi tanto generali da essere sempre i migliori in ogni applicazione.

Quindi: conoscere la Teoria e Pratica degli algoritmi non serve solo a progettare nuovi algoritmi, ma a usare con criterio quelli noti.

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **10(b)** **[31/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA