Meglio veloci che ottimi

corso di laurea in **Matematica**

Informatica Generale, Lezione **10** [**1/4/22**]

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Da mergeSort a quickSort

I punti deboli di mergeSort sono:

- **non può lavorare in place** (ha **sempre** bisogno almeno di un **vettore ausiliario** della stessa dimensione di quello da ordinare)
- anche ottimizzato, necessita di **numerosi "trasferimenti"** di valori durante l'operazione di **fusione**.

Se la suddivisione potesse garantire l'**asserzione più forte**:

$$v[inf, m) \leq v[m, sup)$$

non servirebbe alcun **lavoro di ricombinazione** (fatto dalla funzione `fondi` in `mergeSort`) dei risultati, in quanto gli elementi della parte **sinistra** sarebbero **tutti più piccoli** della parte **destra**.

Ciò è possibile, **investendo** tempo nella fase di **suddivisione** in sottoproblemi (ovviamente non potrà più essere $\theta(1)$ come il semplice calcolo del punto medio).

quickSort

[C. A. R. (Tony) **Hoare**, **1959**]

Abbiamo bisogno di una funzione **`partiziona(v, inf, sup)`** che soddisfi alle seguenti post-condizioni:

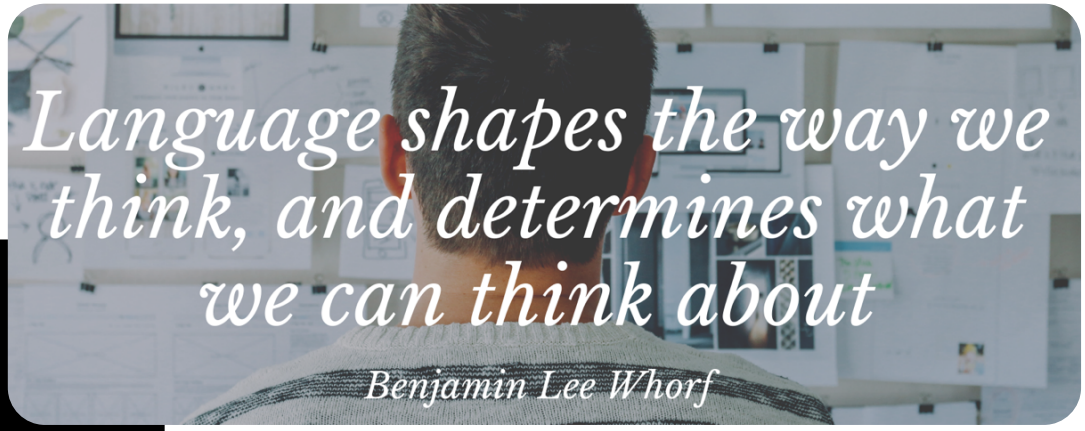
- **ritorna** come risultato un **indice** m, $inf \le m \le sup$
- **modifica** v in modo che soddisfi: $v[inf, m) \le v[m] \le v[m+1, sup)$

Assumendo **`partiziona`** soddisfi le specifiche sopra, possiamo scrivere una procedura **divide et impera**, semplice e geniale, conosciuta come **`quickSort`** (**ordinamento veloce**).

Termina perché **`partiziona`** "**separa**" **sempre almeno un elemento**, quindi i vettori da ordinare ricorsivamente si scorciano almeno di 1.


```
def quickSort(v, inf, sup):
    if sup - inf >= 2:
        m = partiziona(v, inf, sup)
        quickSort(v, inf, m)
        quickSort(v, m+1, sup)
```

Intermezzo: Linguaggio e Pensiero



"Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors."

- TONY HOARE

[ABOUT ALGOL 60] DUE CREDIT MUST BE PAID TO THE GENIUS OF THE DESIGNERS OF ALGOL 60 WHO INCLUDED RECURSION IN THEIR LANGUAGE AND ENABLED ME TO DESCRIBE MY INVENTION [QUICKSORT] SO ELEGANTLY TO THE WORLD.

- C. A. R. HOARE -

LIBQUOTES.COM

partiziona: specifica

Vediamo di visualizzare **cosa fa** `partiziona`, cioè la sua **specifica**.

Discuteremo l'implementazione (cioè **come lo fa**) dopo, vedendo diversi modi di **implementare** questa specifica. Dato un array…

1. prende un elemento qualsiasi p, detto **perno** o **pivot,**
2. calcola il **posto giusto** m in cui p si troverà dopo l'ordinamento di v.
3. Mette p in $v[m]$
4. fa alcuni scambi, in modo che il prefisso $v[\text{inf}, m)$ contenga **elementi minori** di p e il suffisso $v[m, sup)$ contenga **elementi maggiori** di p.


```
def partiziona(v, inf, sup):
# ENS: torna un indice m ∈ [inf, sup)
# MOD: v : v[inf, m) ≤ v[m] ≤ v[m,sup)
```

Partiziona: versione naïf

Per semplicità, vediamo prima una versione naïf che partiziona un vettore $v[0, n)$ con $n = \text{len}(v)$ **usando un vettore d'appoggio** z.

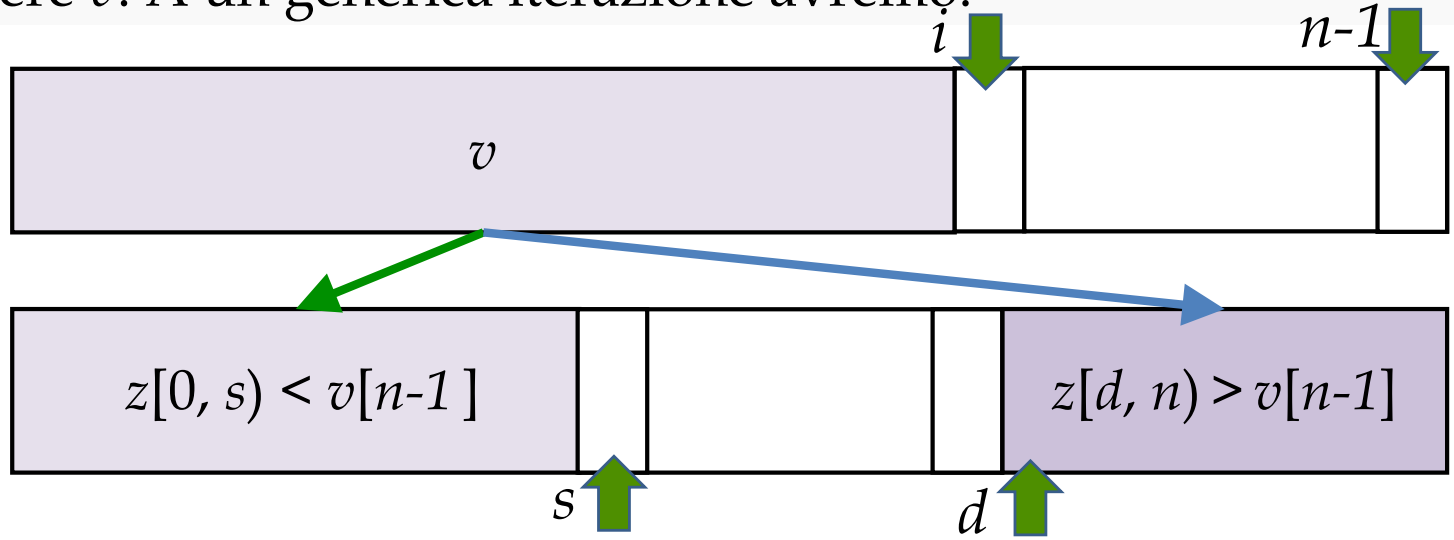
Prendiamo un elemento p **qualsiasi** di v detto **perno** o **pivot** (per comodità $p = v[n\text{-}1]$) e carichiamo un vettore $z[0, n)$ che soddisfa:

$$z[0, m) \leq z[m] = p \leq z[m+1, n) \;\&\; z =_{\text{set}} v$$

Non conosciamo m e **non vogliamo calcolarlo** (costo $\theta(n)$).

Se **riempiamo** z a partire da **sinistra coi minori** di $v[n\text{-}1]$, e da **destra coi maggiori** di $v[n\text{-}1]$, avremo anche l'indice m cercato.

Usiamo le variabili s, d per le zone ricopiate di z e l'indice i per scorrere v. A un generica iterazione avremo:



partiziona naïf: invariante

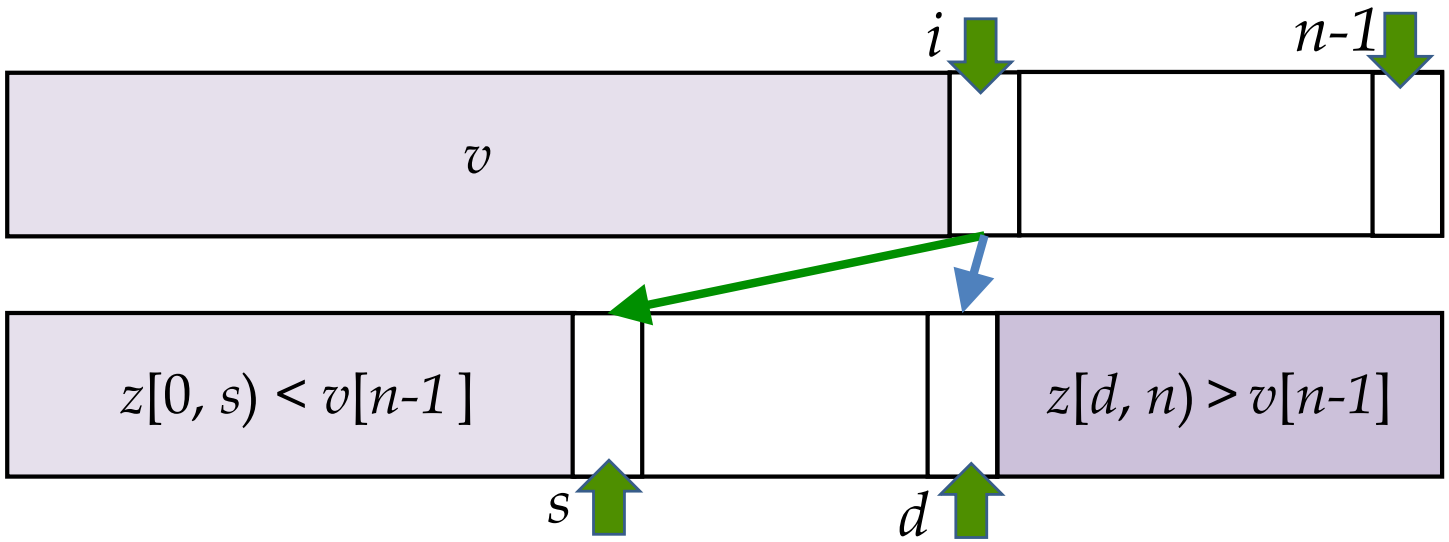
L'invariante sarà:

$\varphi(i, s, d) \equiv v[0, i) = z[0, s) \cup z[d, n)$ & $z[0, s) \leq v[n\text{-}1]$ & $z[d, n) \geq v[n\text{-}1]$

$\varphi(0, 0, n)$ vale banalmente da cui inizializzo $i = 0$, $s = 0$ e $d = n$.

Scorrendo v, se $v[i] \leq v[n\text{-}1]$ lo ricopio in $z[s]$ incrementando s, e se $v[i] \geq v[n\text{-}1]$ lo ricopio $z[s]$ decrementando s. Anche ***d - s = n – i*** è **invariante** (le prossime celle libere dove scrivere in z sono s e d - 1).

Quando $i = n - 1$ esco, $s - d = 1$ e quindi l'ultima cella libera è $s = d - 1$ che è il valore cercato da tornare. Chiudo ricopiando $v[n\text{-}1]$ in $z[s]$.



partiziona naïf: pseudocodice

Abbiamo bisogno di una versione generica opportunamente **parametrizzata** per lavorare su una **porzione di vettore**.

Gli invarianti si adattano, sostituendo n con *sup* - *inf*, 0 con *inf*.

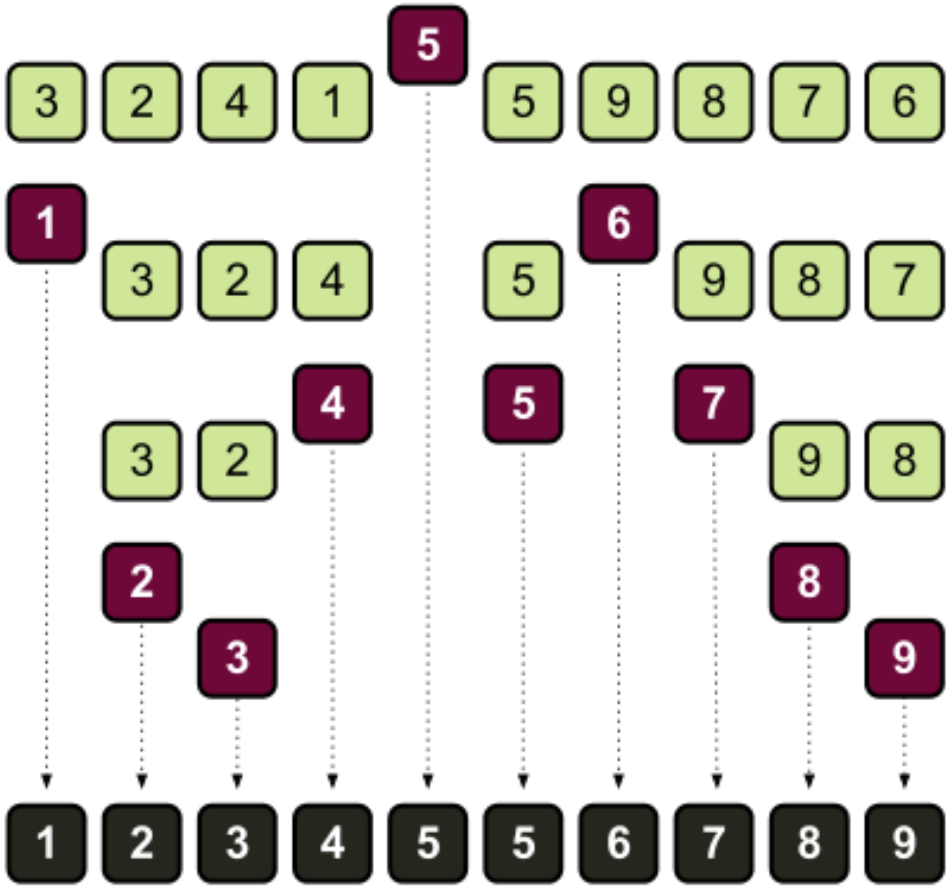
Passiamo z come parametro perché, come per `mergesort`, è chiaro che possiamo usare solo un vettore e lo passiamo.

Presenteremo versioni ottimizzate dopo l'analisi di `quickSort`.

Ora ci basta sapere che `partiziona` è $\Theta(n)$.

```
def partiziona(v, inf, sup, z):
    s, d = inf, sup
    for i = inf to sup-2:
        if v[i] ≤ v[sup-1]: z[s], s = v[i], s+1
                    else: d, z[d] = d-1, v[i]
    z[s] = v[sup-1]
    copia(z, inf, sup, v)
    return s
```

quickSort: esempio di esecuzione



Osservate che il **perno** dopo un'esecuzione di partiziona **finisce definitivamente nel posto giusto.**

Non ci sono più confronti né tantomeno **scambi** tra elementi che sono finiti in due **partizioni diverse**.

Nell'esempio, come nella nostra procedura, si sceglie l'**ultimo elemento** di v come **perno**.

quickSort: casi ottimo/pessimo

Analizzare ***quickSort*** non è affatto facile, perché a differenza di ***mergeSort***, il suo comportamento **dipende dalla scelta del perno**. La sua relazione di ricorrenza è (k è m – *inf* ed n = *sup* – *inf*):

$$T(n) = T(k) + T(n - k - 1) + \Theta(n) \qquad T(1) = \Theta(1)$$

- La **scelta ottima** sarebbe un perno che divide il vettore in **due partizioni** di dimensioni **uguali**.

Se questo fosse il caso, l'equazione di ricorrenza di *quickSort* sarebbe la stessa di ***mergeSort***, e quindi la complessità $\Theta(n \log n)$. Purtroppo, per ***quickSort***, questo è solo il caso **ottimo**.

- Il **caso pessimo** viceversa si ha quando una delle due partizioni è **sempre vuota**. In tal caso, l'equazione di ricorrenza diventa quella di ***insertionSort***: $T(n)=T(n - 1)+\Theta(n)$ con soluzione $T(n)=\Theta(n^2)$.

Osserviamo che scegliendo come **perno** il **primo** o l'**ultimo** elemento, questo caso si verifica curiosamente quando il **vettore è già ordinato** (o quasi).

- Assumendo le permutazioni di ingresso equiprobabili, si può dimostrare con un po' di conti che il **caso medio** è $\Theta(n \log n)$.

Il problema della k-mediana

Prima di addentrarci nell'analisi di ***quickSort***, è utile considerare il problema della ***k*-mediana**, cioè il calcolo del **valore** km in un vettore v che ha **esattamente** k **elementi minori** in v (per evitare dettagli tanto fastidiosi, assumiamo che v contenga valori **tutti distinti**).

Noi abbiamo già calcolato in tempo $\Theta(n)$ la **0-mediana** (che è il **minimo** del vettore) e la **(n-1)-mediana** (che è il **massimo**).

Non è altrettanto ovvio calcolare in tempo lineare la mediana o più in generale la k-mediana.

➧ **Potendo modificare** il vettore, e conoscendo ***mergeSort***, è viceversa evidente un algoritmo $\Theta(n \log n)$: si ordina il vettore e poi si va a prendere il valore in posizione k.

Ma **non è chiaro** se **ordinare il vettore** sia veramente **necessario**.

```
def kMediana(v, k):
# ENS: torna km : #{i : v[i] < km} = k-1
# MOD: ordina v
    mergeSort(v)
    return v[k]
```

Possiamo sfruttare partiziona?

partiziona calcola una m-mediana, ma **non possiamo scegliere m**. Ma possiamo sfruttare questo fatto?

Dobbiamo valutare se $m = k$, $m > k$, oppure $m < k$.

Nel caso $m = k$ abbiamo **finito**, yep!

Nel caso $m > k$, per il comportamento di partiziona, la k-mediana è finita **nella parte sinistra** del vettore $v[inf, m)$.

Simmetricamente, nel caso $m < k$, la k-mediana è finita nella **parte destra** del vettore. **Attenzione** che nella parte destra, la k-mediana si trova in posizione $k - m$.

```
def kMediana(v, k, inf, sup):
# ENS: torna km : #{i : v[i] < km } = k-1
# MOD: modifica v ma senza ordinarlo complet.
    m = partiziona(v, inf, sup)
    if m==k: return v[m]
    if m >k: return kMediana(v, k, inf, m)
    return kMediana(v, k-m, m, sup)
```

Analisi dell'algoritmo

Siamo in una situazione simile a ***`quickSort`***: **dipende dal bilanciamento** delle partizioni, ma stavolta andiamo ricorsivamente **su una partizione sola**.

L'equazione di ricorrenza è (ponendo $n = sup - inf$ e $h = m - inf$):

$T(n) = T(h) + \Theta(n)$ oppure $T(n) = T(n - h) + \Theta(n)$ e $T(1) = \Theta(1)$

Ancora una volta, può diventare **quadratica** se le partizioni sono molto **sbilanciate** $T(n) = T(n - 1) + \Theta(n)$ con soluzione $T(n) = (n^2)$, come **`insertSort`** ricorsivo (*prossima slide*) mentre se si **divide esattamente in due** si ha $T(n) = T(n/2) + \Theta(n)$, con soluzione [identica al **Torneo**] $T(n) = \Theta(n)$.

Siccome la k-mediana va cercata in **una sola delle due** partizioni (differentemente da *`quickSort`*), eventuali **partizioni sbilanciate favorevoli accelerano** il calcolo.

Analisi casi medi di k-mediana e quickSort

corso di laurea in **Matematica**

Informatica Generale, Lezione **11.1**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Alcune intuizioni su quickSort (1)

- **Domanda**: Quante volte viene chiamata ***`partiziona`*** in ***`quickSort`***?

Esattamente n volte: ogni elemento diventa perno almeno una volta (al limite su vettori lunghi 1) e poi **sparirà dalle chiamate ricorsive**.

- **Domanda**: Quali (e quanti) confronti farà ***`quickSort`***?

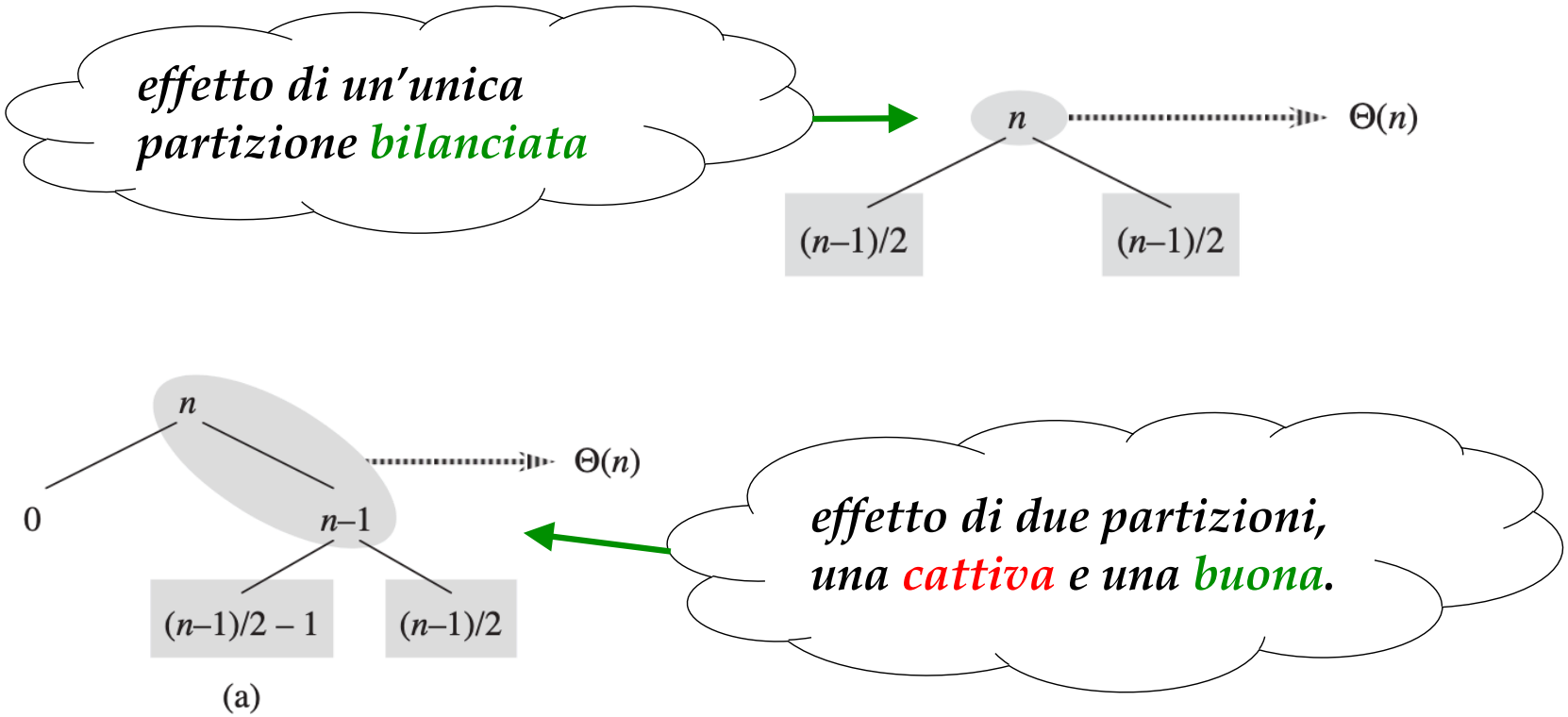
Il perno viene confrontato con **tutti gli altri elementi della partizione** (al primo giro $\mathcal{O}(n)$ confronti). Tuttavia se due elementi finiscono in partizioni diverse, non saranno **mai più** confrontati.

Quindi, se la partizione è bilanciata, **elimino di colpo** $(n/2)^2$ **confronti** dagli $n^2/2$ possibili. **Nessun confronto viene ripetuto**. Se la partizione esce sbilanciata, elimino solo n - 1 confronti.

Altre intuizioni su quickSort

Domanda: Cosa succede se ***`quickSort`*** alterna una partizione bilanciata a una sbilanciata?

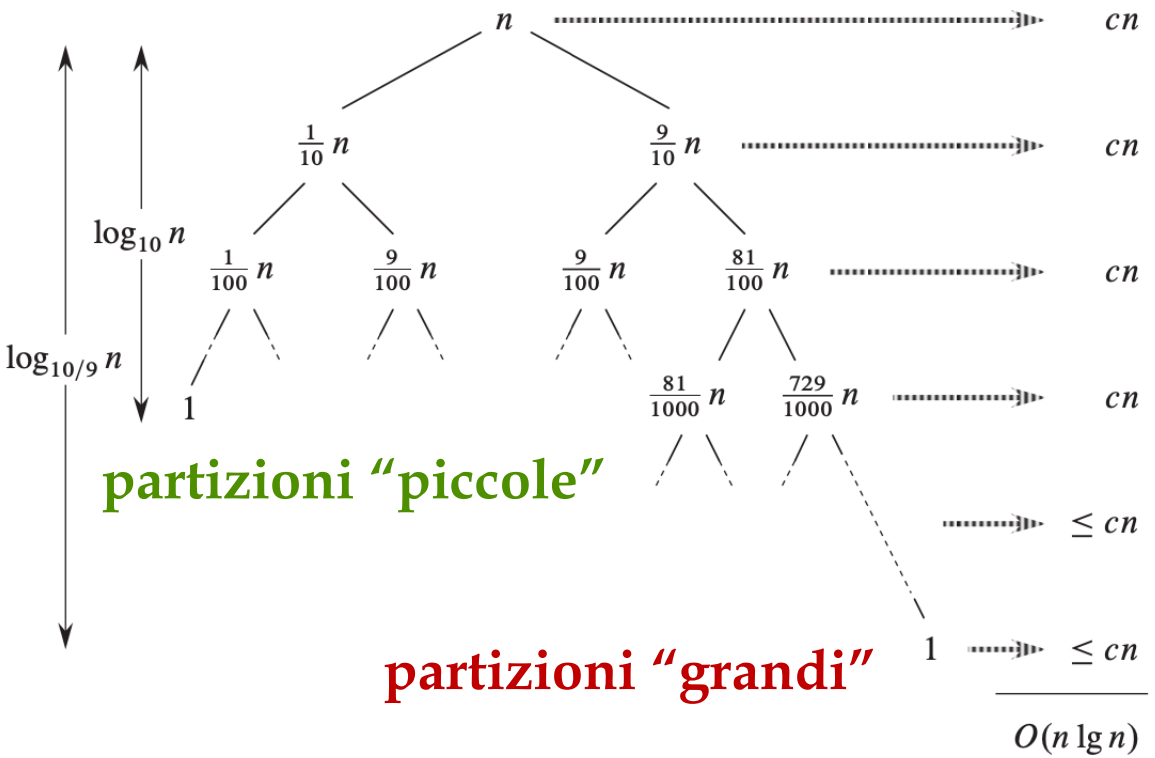
Niente. Divide in 3 sotto-vettori di cui uno vuoto e due bilanciati, come se **la partiziona buona "assorbisse" subito l'errore dovuto a quella cattiva**.



Ultime intuizioni su quickSort

Domanda: Cosa succede se ***`quickSort`*** partiziona il vettore **sistematicamente** in due parti di lunghezze in rapporto 9 a 1?

Occorre risolvere la relazione di ricorrenza $T(n) = T(9n/10) + T(n/10) + cn$. Impegnativo, ma aiutandosi col metodo dell'albero…



k-mediana: partizioni εn / $(1 - \varepsilon)n$

Supponiamo di avere partizioni **sempre nelle stesse proporzioni** εn e $(1 - \varepsilon)n$, con $0 < \varepsilon < ½$ e di dover cercare la k-mediana sempre **nella più grande**. La relazione di ricorrenza diventa:

$$T(n) = T((1 - \varepsilon)n) + \Theta(n) \qquad T(1) = \Theta(1)$$

Applichiamo il metodo iterativo, cercando una maggiorazione:

$$\begin{aligned} T(n) &\leq cn + T((1 - \varepsilon)n) \\ &\leq cn + (1 - \varepsilon)cn + T((1 - \varepsilon)^2 n) \\ &\leq cn + (1 - \varepsilon)cn + (1 - \varepsilon)^2 cn + T((1 - \varepsilon)^3 n) \leq \\ &\leq \ldots = \\ &\leq \sum_{i=0,\ldots,k} (1 - \varepsilon)^i cn \\ &\leq cn \sum_{i=0,\ldots,\infty} (1 - \varepsilon)^i = \{\text{serie geometrica convergente}\} \\ &= cn\,(1/\varepsilon) = \mathcal{O}(n) \end{aligned}$$

Ottenendo **sempre** una **complessità lineare** che **cresce allo sbilanciarsi** delle partizioni (al decrescere di ε).

Partiziona e kMediana randomizzate

Immaginiamo ora che ***partiziona*** **scelga in modo casuale** il pivot (**algoritmo randomizzato**), ottenendo ***kMedianaRand***.

Fissiamo una qualsiasi costante ε, per comodità $\varepsilon = 1/4$ e diciamo che l'algoritmo **è in fase *j*** se la **dimensione della partiziona** in cui si sta cercando è **compresa tra** $(3/4)^j n$ **e** $(3/4)^{j+1} n$. Qual è **il numero medio** di esecuzioni in cui ***kMedianaRand*** in **ciascuna fase**?

C'è probabilità $1/2$ di scegliere un buon pivot (che divide lo spazio di ricerca in due parti $1/4\ n$ e $3/4\ n$): il valor medio delle chiamate di ***partiziona*** per passare dalla fase j alla fase j+1 è $\sum_{i \in [0, k)} i \cdot 1/2^i \leq \sum_{i \in [0, \infty)} i \cdot 1/2^i =$ **2** (vedi problema del **confrontatore** dell'esonero).

Qual è il numero di passi fa ***kMedianaRand*** randomizzata? Sarà la somma di $X_1, \ldots, X_h$, dove X_j è il numero di passi fatti in fase j.

Il **valor medio** di passi fa ***kMedianaRand*** sarà $E[\sum_j X_j] =$ (per linearità del valor medio) $\sum_j E[X_j] = \sum_j 2c\,(3/4)^j n = 2cn \sum_j (3/4)^j =$ ***8cn*** (ancora serie geometrica convergente a $1/(1 - 3/4) = 4$). Abbiamo quindi:

Teorema. *Il valor medio atteso della costo computazionale di* `kMedianaRand` *è* $O(n)$.

quickSort randomizzato

Modifichiamo **`quickSort`** come segue: si partiziona random finché non c'è un **perno buono**, che divida il vettore in almeno $^1/_4\,n$ - $^3/_4\,n$.

Questa operazione ha senso se il vettore ha almeno 4 elementi, quindi consideriamo **caso base i vettori di al più 3 elementi**.

Se il perno non è "buono", **buttiamo via la partizione** e ripetiamo la stessa chiamata a **`quickSort`** con gli stessi parametri (osservare che questa procedura **non termina**, con **probabilità 0**).

```
def quickSortRand(v, inf, sup):
    if sup - inf <= 3: # caso base: 3 elem.
            ordina3(v)
            return
    m = partizionaRand(v, inf, sup)
    if inf + (sup – inf) / 4 > m
          or inf + 3 * (sup – inf) / 4 < m:
       # butto via tutto e rifaccio
       quickSortRand(v, inf, sup)
    quickSortRand(v, inf, m)
    quickSortRand(v, m+1, sup)
```

Conseguenze: caso medio quickSort

Osserviamo che:

- come prima, in media **troviamo un buon perno ogni 2 tentativi**
- ogni **valore** giocherà da **perno 1 sola volta**
- in fase j, ci sono al più $(4/3)^{j+1}$ istanze di dimensione $(3/4)^j n$

La complessità attesa della fase j è globalmente $\mathcal{O}(n)$ [come nell' albero con partizioni 1/10 – 9/10] e si attraversano **al più $\log_{4/3} n$ fasi** cioè $\Theta(\log n)$, quindi il **tempo di esecuzione atteso** è $\Theta(n \log n)$.

Osservate che qualsiasi versione **randomizzata** o **deterministica** su **vettore** con valori distribuiti uniformemente **causale** **che però non butta via le partizioni calcolate** ci mette un **tempo inferiore** (è comunque meglio una cattiva partizione, che rifare la partizione).

Quindi, **`quickSortRand`** (che è essenzialmente uno **strumento concettuale**) stabilisce un **limite superiore** al **valor medio** del tempo di esecuzione di **`quickSort`**. Quindi:

Teorema. *Il valor medio atteso del costo computazionale di* `quickSort` *è* $\mathcal{O}(n \log n)$.