Partiziona e la sua ottimizzazione

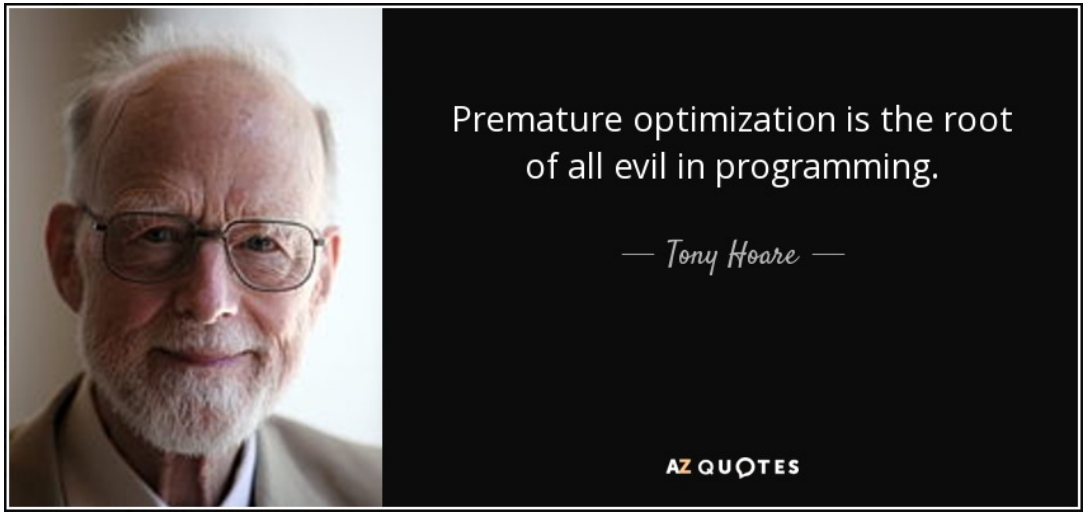
corso di laurea in **Matematica**

Informatica Generale, Lezione **10 (b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

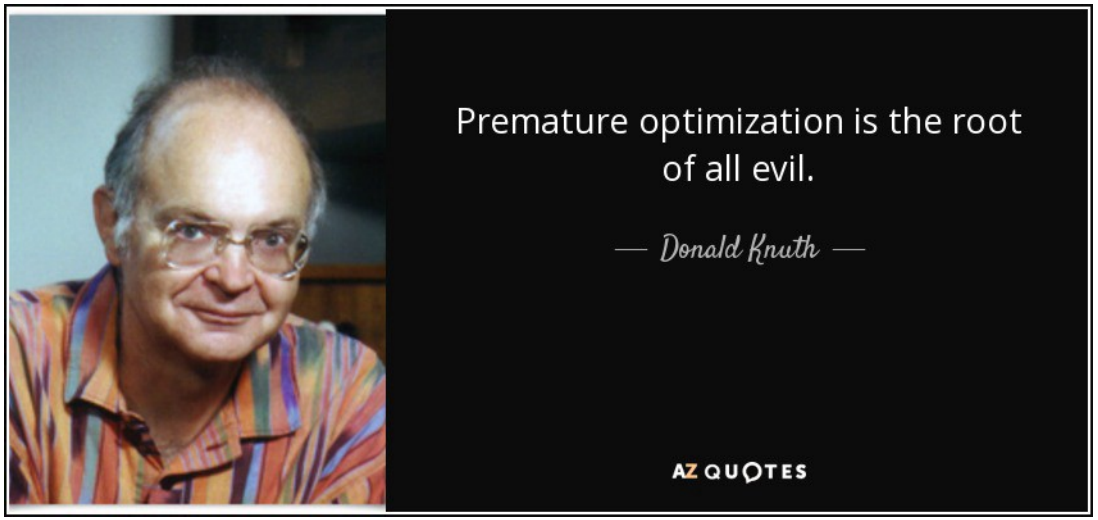
Intermezzo: Attribuzioni Difficili



D'istinto la attribuirei a Tony Hoare…

… ma sembra più accreditata l'ipotesi Donald Knuth…

Ma se fosse di Tony Hoare, la sua funzione partiziona sarebbe in contraddizione con questo pensiero!



Morale: **non fidatevi dei siti di citazioni…**

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 ignorante

Un modo semplice per fare partiziona è il seguente:

1. **Calcolare l'indice *m*** dove si troverà il valore perno $p = v[n\text{-}1]$. Si può fare facilmente usando una funzione **`contaMinori(v, p)`**;
2. **Riempire un vettore *z***, copiando **i minori di *p* a partire da 0** e i **maggiori a partire di *m*+1**;
3. ricopiare *z* in *v*.

Osserviamo che questa procedura è $\Theta(n)$, per cui è sufficiente per le analisi di complessità fatte finora…

Dopo **`contaminori`**, con qualche riflessione, si riesce anche a scrivere una versione in-place (➧ **Esercizio**)

… ma **`quickSort`** non sarebbe affatto quick!

Partiziona: versione naïf

Vediamo una versione naïf che partiziona un vettore $v[0, n)$ con n = len(v) **usando un vettore d'appoggio** z.

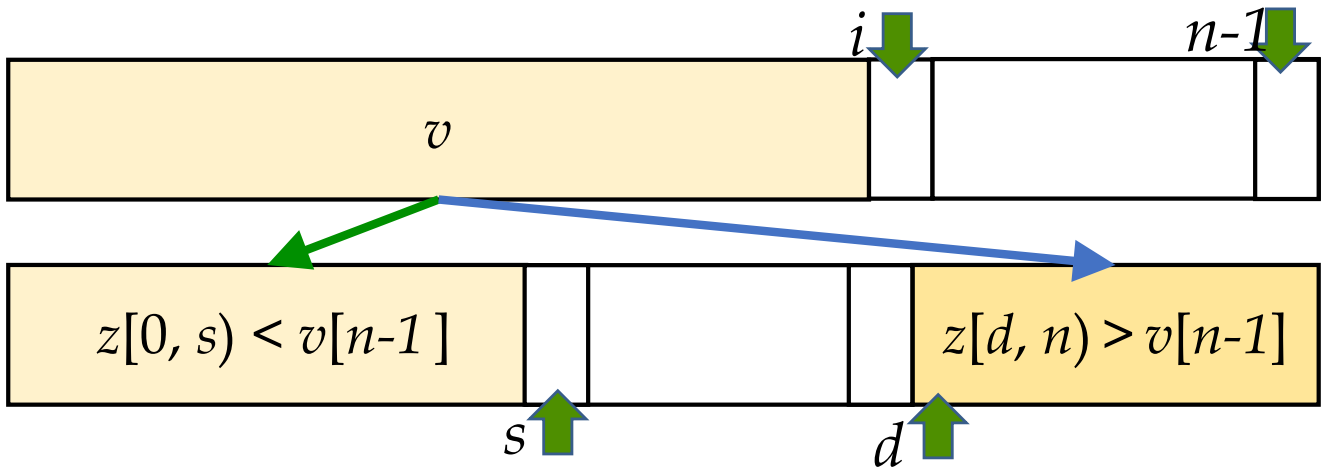
Prendiamo come pivot, un elemento p **qualsiasi** di v (per comodità $p = v[n\text{-}1]$) e carichiamo un vettore $z[0, n)$ che soddisfa:

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

senza calcolare preventivamente m.

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

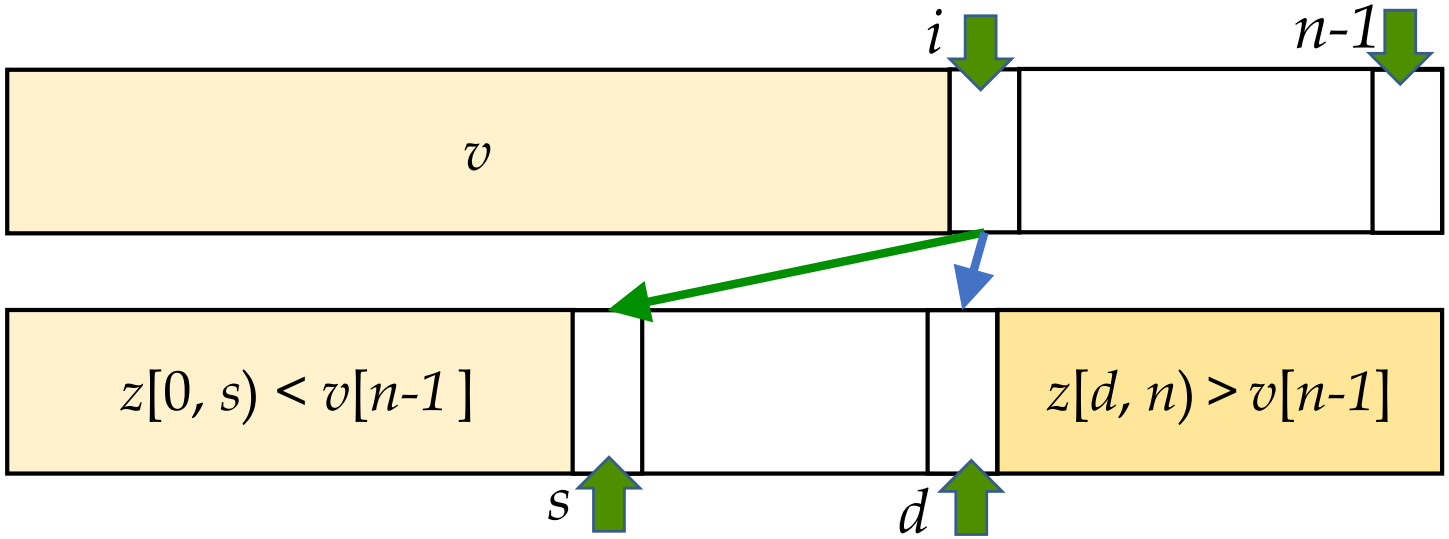
L'**invariante** sarà:

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

$\varphi(0, 0, n)$ vale banalmente da cui inizializzo $i = 0$, $s = 0$ e $d = n\text{-}1$.

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 $z[d]$ decrementando d. Anche $\boldsymbol{d - s = n - i - 1}$ è **invariante** (le prossime celle libere dove scrivere in z sono s e d).

Quando $i = n - 1$ esco, $s - d = 0$ e quindi l'ultima cella libera è $s = d$ 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.

Poi presenteremo le versioni di partiziona che rendono **quickSort** degno del suo nome, ma osservate che **le analisi di complessità asintotica** necessitano solo di una funzione ***partiziona*** **lineare**.

```
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, inf)
    return s
```

partiziona in-place di Nico Lomuto

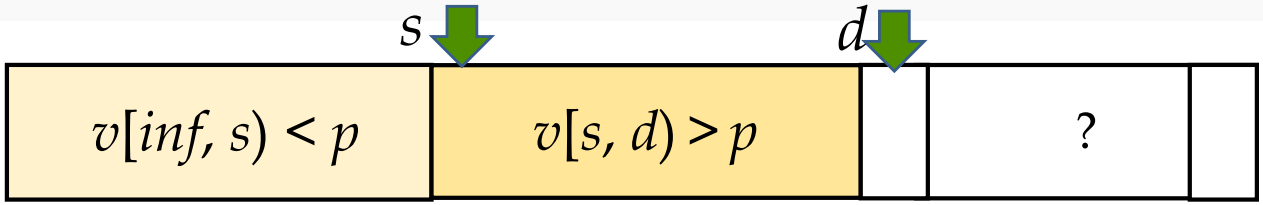
È facile da capire. Immaginiamo a una generica iterazione di essere nella situazione nell'immagine.

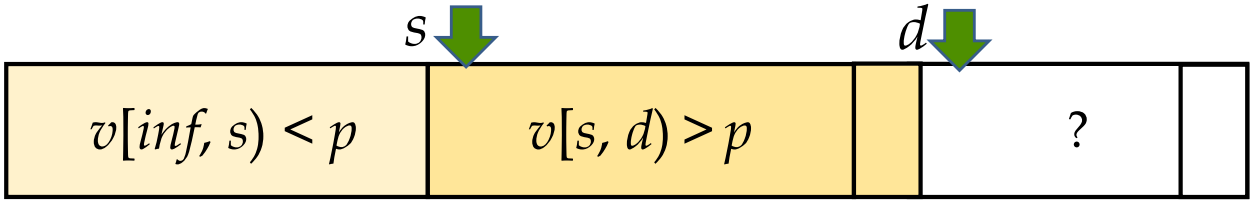
Se $v[d] > v[n\text{-}1]$ devo incrementare d per mantenere l'invariante.

Se $v[d] \leq v[n\text{-}1]$, pare non esserci posto nella parte chiara… ma in realtà basta **scambiare $v[d]$ con $v[s]$** e incrementare sia d che s.

L'ultimo scambio porterà il perno $p = v[sup - 1]$ al posto giusto e basterà tornare s come risultato.

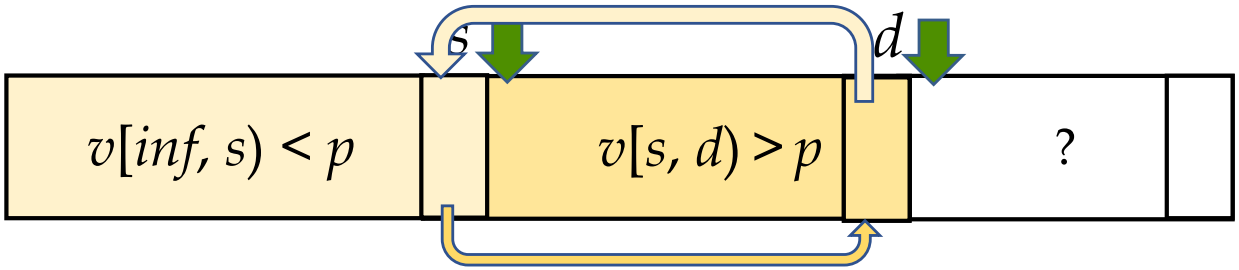
iterazione generica





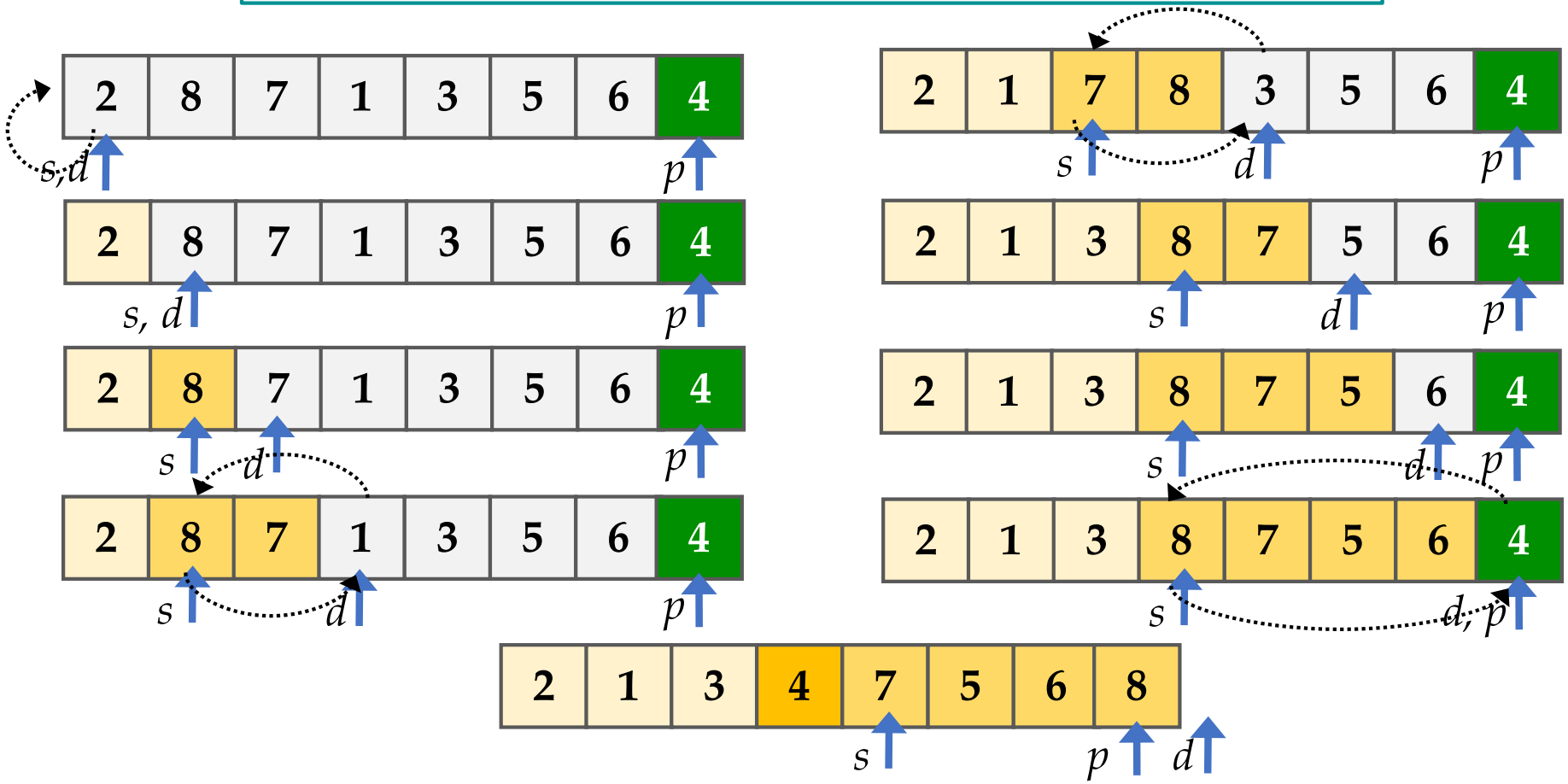
caso $v[d] > p$

caso $v[d] \leq p$



pseudocodice ed esecuzione

```
def partizionaLomuto(v, inf, sup):
    s, p = inf, v[sup-1]
    for d = inf to sup-1:
        if v[d] ≤ p: v[s], v[d], s = v[d], v[s], s+1
    return s-1
```



partiziona di Hoare

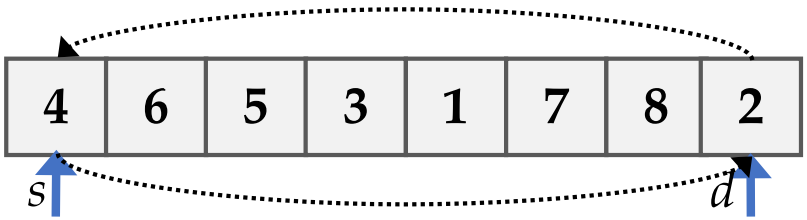
```
def partizionaH(v, inf, sup):
    s, d, p = inf, sup-1, v[inf]
    while True:
        while v[d] > p: d = d-1
        while v[s] < p: s = s+1
        if d < s:  return s
        if d == s: return s+1
        v[s], v[d] = v[d], v[s]
        s, d = s+1, d-1
```

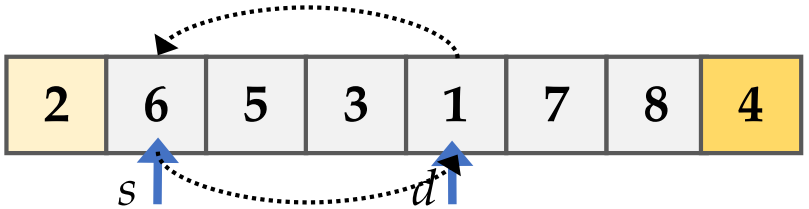
`partizionaH` di Tony Hoare **non isola il perno**: divide il vettore in due metà in cui è vero che:

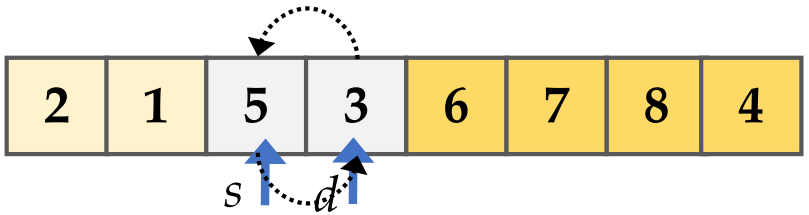
$$v[inf, m) \leq p \leq v[m, sup)$$

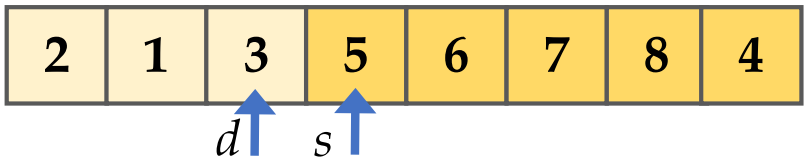
È una funzione **efficientissima**, ma **dagli equilibri delicatissimi** (notate che non controlla neanche di uscire dai limiti del vettore)

Esercizio: modificare **`quickSort`** per usare **`partizionaH`**.









partiziona di Hoare classica

Classicamente, **partizionaH** di Tony Hoare viene presentata con cicli di tipo **repeat** C **until** B: e l'eleganza ne guadagna.

In **C** c'è il ciclo **do** C **while** B che è simile, ma ha le condizioni rovesciate: si esce quando B è falsa, mentre nel repeat-until quando B è vera: **do** C **while** B **è equivalente a** **repeat** C **until** not(B).

Notare che, siccome le prime assegnazioni si fanno sempre e comunque, occorre inizializzare sia s che d "fuori dal vettore".

In pseudo-Python potrei scrivere **repeat** C **until** B (tanto è uno pseudo linguaggio) ma volendo rimanere aderente a Python, la traduzione più fedele usa **break** come segue:

```
while True:
    C
    if B: break
```

```
def partizionaH(v, inf, sup):
    s, d, p = inf-1, sup, v[inf]
    while True:
        repeat: d = d-1 until v[d] ≤ p
        repeat: s = s+1 until v[s] ≤ p:
        if d ≤ s:  return s
        v[s], v[d] = v[d], v[s]
```

L'opinione di Jon L. Bentley

Partiziona di Nico Lomuto **è particolarmente facile** da scrivere e imparare.

To partition the array around the value T we'll use a simple scheme that I learned from Nico Lomuto of Alsys, Inc. There are faster programs for this job†, but this routine is so easy to understand that it's hard to get wrong, and it is by no means slow. Given the value T, we are to rearrange $X[A..B]$ and

Non è particolarmente difficile scrivere partiziona seguendo l'idea di Hoare (scorrere il vettore da due lati): ciò che è veramente arduo è farlo **senza** ad esempio **controllare** le **condizioni di fine vettore**.

† Most presentations of Quicksort use a partitioning scheme based on two approaching indices, like the one described in Problem 3. Although the basic idea of that scheme is simple, I have always found the details tricky — I once spent the better part of two days chasing down a bug hiding in a short partitioning loop. A reader of a preliminary draft complained that the standard two-index method is in fact simpler than Lomuto's, and sketched some code to make his point; I stopped looking after I found two bugs.

Considerazioni finali su part&quick

La fortuna di **`quickSort`** è dovuta all'estrema efficienza di **`partiziona`**.

La versione Lomuto è comunque molto efficiente, e ha diversi vantaggi tra cui segnalo: può partizionare il vettore rispetto a un qualsiasi valore, anche **non appartenente all'array** (è molto più arduo estendere partiziona di Hoare a questo fine).

L'esistenza di una **funzione lineare per *k*-mediana** rende possibile scrivere **`quickSort`** in **tempo pessimo** $\Theta(n \log n)$ (➧ **Esercizio**), ovviamente rallentandolo pesantemente.

Analogamente, il **modo migliore** per fare ***k*-mediana** è l'algoritmo visto basato su **`partiziona`**, di tempo di esecuzione atteso $\Theta(n)$.

Una miglioria molto studiata è **scegliere** come **pivot** la **mediana** di un insieme di **elementi estratti a caso**: sperimentalmente la scelta migliore pare sia fare la mediana di **3** (ancora una volta, **ogni miglioramento sembra rallentare** **`quickSort`**).

Ci sono infinite varianti: segnalo che a volte (quando i vettori hanno **molti elementi uguali**) è preferibile partizionare come segue:

$v[inf, s) < p$	$v[s, d) = p$	$v[d, sup) > p$