Quick Sort

- Ordinamento in loco
- Tempo di esecuzione nel caso peggiore: $\Theta(n^2)$
- Tempo di esecuzione nel caso medio: $\Theta(n \lg n)$
- I fattori costanti nascosti nella notazione Θ sono abbastanza piccoli
- Basato sul paradigma *divide et impera*:

Per ordinare un sottoarray $A[p...r]$:

Divide: $A[p...r]$ è ripartito (e risistemato) in due sottoarray non vuoti $A[p...q]$ e $A[q+1...r]$, in modo che ogni elemento del primo sia minore o uguale a ogni elemento del secondo. *L'indice q viene calcolato dalla procedura di partizionamento.*

Impera: i due sottoarray $A[p...q]$ e $A[q+1...r]$ sono ordinati, ricorsivamente.

Combina: non c'è niente da fare: $A[p...r]$ è ordinato.

PARTITION(A,p,r) risistema il sottoarray $A[p...r]$ e riporta l'indice q:

Quick Sort - Algoritmo

```
QUICKSORT(A)
1. QUICK(A,1,length[A])
```

```
QUICK(A,p,r)
1. if p<r
2.    ;; il sottoarray contiene almeno 2 elementi
3.    then q <- PARTITION(A,p,r)
4.         QUICK(A,p,q)
5.         QUICK(A,q+1,r)
```

Partizionamento dell'array (I)

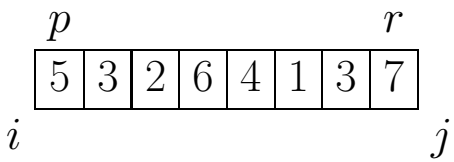
Il sottoarray $A[p...r]$ viene risistemato in loco.

Il primo elemento del sottoarray, $A[p]$ viene scelto come *pivot*: alla fine, tutti gli elementi in $A[p...q]$ conterranno elementi minori o uguali a $A[p]$ e tutti gli elementi in $A[q+1...r]$ conterranno elementi maggiori o uguali a $A[p]$

```
x <- A[p]      ;; x e' il pivot
```

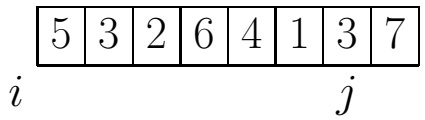
Le due regioni di sinistra e di destra vengono costruite per stadi: inizialmente sono entrambe vuote, e si usano gli indici i e j per i due estremi

```
i <- p-1       ;; la regione A[p...i] e' vuota
j <- r+1       ;; la regione A[j...r] e' vuota
```



INIZIO CICLO: ripetuto finché $i < j$. L'indice j viene spostato verso sinistra, finché non si trova un elemento minore o uguale al pivot ($A[j] \leq x$):

```
repeat j <- j-1
until A[j] <= x
```



Simmetricamente, l'indice i viene spostato a destra, fino a trovare un elemento maggiore o uguale al pivot ($A[i] \geq x$):

Partizionamento dell'array (II)

```
repeat i <- i+1
until A[i] >= x
```

Inizialmente il ciclo termina subito con $i = p$, perché $A[p] \geq x$.

5	3	2	6	4	1	3	7
i						j	

Ora si ha $A[i] \geq x$ alla sinistra di $A[j] \leq x$: i due elementi vengono scambiati

3	3	2	6	4	1	5	7
i						j	

Poiché ancora $i < j$, il ciclo riprende: l'indice j si sposta a sinistra finché $A[j] \leq x$ e l'indice i si sposta a destra finché $A[i] \geq x$:

3	3	2	6	4	1	5	7
			i		j		

Poiché ancora $i < j$, si scambiano di posto $A[i]$ e $A[j]$:

3	3	2	1	4	6	5	7
			i		j		

Ancora si ha $i < j$ quindi il ciclo ricomincia:

3	3	2	1	4	6	5	7
				j	i		

FINE CICLO: Ora $i > j$, non si effettua nessuno scambio e il ciclo termina.

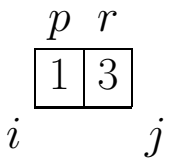
L'indice riportato q è j.

L'algoritmo Partition

```
PARTITION(A,p,r)
 1. x <- A[p]
 2. i <- p-1
 3. j <- r+1
 4. while i < j
 5.    do repeat j <- j-1
 6.       until A[j] <= x
 7.       repeat i <- i+1
 8.       until A[i] >= x
 9.       if i < j
10.         then scambia A[i] <-> A[j]
11. return j
```

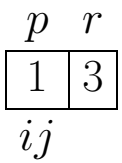
Partition - Caso base I

Primo caso: array composto da due elementi ordinati.



Pivot = 1

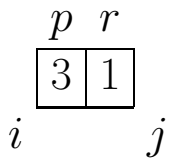
```
while i<j      (SI)
   repeat j<-j-1    SI FERMA  CON j=p
   until A[j]<=1

   repeat i<-i+1    SI FERMA SUBITO CON i=p
   until A[i]>=1

if i<j ...(NO)
```




```
while i<j ... (NO)  ---> FINE PARTIZIONE
```

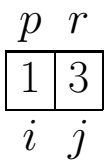
Partition - Caso base II

Secondo caso: array composto da due elementi non ordinati.



Pivot = 3

```
while i<j       (SI)
   repeat j<-j-1    SI FERMA SUBITO CON j=r
   until A[j]<=3
   repeat i<-i+1    SI FERMA SUBITO CON i=p
   until A[i]>=3
if i<j
   then scambia A[i] <-> A[j]
```



Alla successiva iterazione gli indici i e j si accavallano e viene restituito $q = j = 1$, come nel caso precedente.

Il tempo di esecuzione di Partition

Se n è il numero di elementi del sottoarray $A[p...r]$

$$T_{PARTITION}(n) \in \Theta(n)$$

Per ogni posizione del sottoarray viene eseguita almeno un'assegnazione ad i oppure una a j. Quindi il costo di PARTITION è $\Omega(n)$.

Oltre all'assegnazione, per ogni elemento del sottoarray vengono eseguiti al massimo due test (uno per ciascuno dei due cicli `until`) ed eventualmente il test e lo scambio dell'istruzione `if` che segue: tutte operazioni a costo costante.
Quindi il costo del ciclo `while` è $O(n)$.

L'inizializzazione di i, j e l'assegnazione a x richiedono tempo costante.

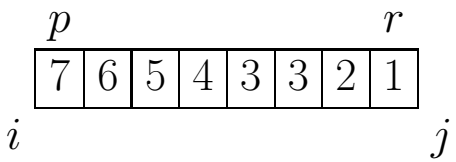
Quindi il costo dell'algoritmo è anche $O(n)$.

Esercizi

1. Che cosa succederebbe nel QUICKSORT se PARTITION(A,p,r) restituisse un valore q uguale a r?
2. Illustrare le operazioni di PARTITION sull'array A $= \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$ (8.1-1)
3. Quale valore restituisce PARTITION se tutti gli elementi dell'array A[p...r] hanno lo stesso valore? (8.1-2)
4. Illustrare le operazioni di PARTITION su un array già ordinato.
5. Illustrare le operazioni di PARTITION su un array ordinato in senso decrescente.

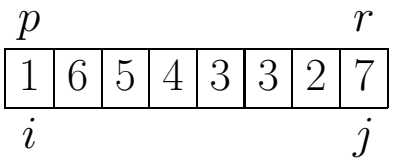
Partizionamento sbilanciato

Esempio 1: gli elementi sono ordinati al contrario



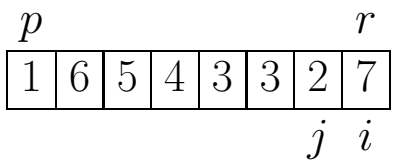
Pivot = 7

```
while i<j
   repeat j<-j-1     SI FERMA SUBITO CON j=r
   until A[j]<=7

   repeat i<-i+1     SI FERMA  CON i=p
   until A[i]>=7

if i<j
   then scambia A[i] <-> A[j]
```

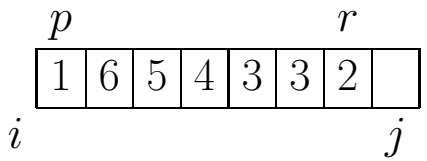



```
while i<j
```

```
   repeat j<-j-1     SI FERMA SUBITO CON j=r-1
   until A[j]<=7

   repeat i<-i+1     SI FERMA CON i=r
   until A[i]>=7

if i<j ... (NO)
```

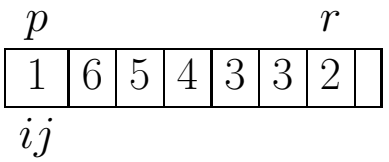


La prima partizione produce i sottoarray $A[p \ldots j]$ e $A[i \ldots r]$.



Pivot = 1

```
while i<j
   repeat j<-j-1     SI FERMA  CON j=p
   until A[j]<=1

   repeat i<-i+1     SI FERMA SUBITO CON i=p
   until A[i]>=1                                  if i<j ... (NO)
```

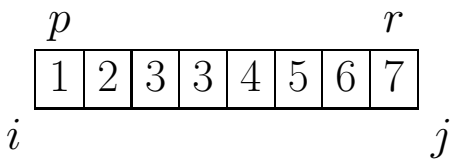


- La seconda partizione produce i sottoarray $A[p \ldots j]$ e $A[i+1 \ldots r]$.
- Anche la terza partizione viene effettuata su un sottoarray ordinato in senso inverso.

Complessità del Quick Sort - Analisi del caso peggiore (II)

Partizionamento sbilanciato

Esempio 2: gli elementi sono già ordinati



Il pivot è 1.

```
while i<j
   repeat j<-j-1     SI FERMA  CON j=p
   until A[j]<=1

   repeat i<-i+1     SI FERMA SUBITO CON i=p
   until A[i]>=1

if i<j ... (NO)
```

Quando l'array è già ordinato in senso crescente il pivot viene sempre preso a sinistra.

Ogni partizione di una regione di n elementi genera, una regione con un solo elemento e l'altra con $n-1$.

Essendo il costo della partizione $\Theta(n)$ nel caso peggiore l'equazione di ricorrenza diventa:

$$\begin{aligned} T_{quick}(0) &= a \\ T_{quick}(n+1) &= T_{quick}(n) + \Theta(n) \end{aligned}$$

Con $\Theta(n)$ polinomio di primo grado.
Applicando il primo teorema per la soluzione delle equazioni di ricorrenza:

$$\begin{aligned} T(0) &= a \\ T(n+1) &= T(n) + g(n) \end{aligned}$$

allora

$$T(n) = a + \sum_{k=0}^{n-1} g(k)$$

si ha

$$T_{quick}(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$$

Quindi, $T_{quick}(n)$ è $\Theta(n^2)$ (nel caso peggiore)

Ad ogni partizionamento il pivot definisce due regioni di uguale dimensione (+-1).

$$\begin{aligned} T_{quick}(0) &= c_0 \\ T_{quick}(n) &= 2 \cdot T_{quick}(n/2) + \Theta(n) \end{aligned}$$

Per il secondo caso del Master Theorem, ponendo:

$a = 2, \quad b = 2, \quad \Theta(n) = c \cdot n = p(n^k), \quad k = 1$ ed essendo $a = b^k$

il tempo d'esecuzione risulta $T_{quick}(n) = \Theta(n \, lg \, n)$

MASTER THEOREM

Se

$$T(n) = \begin{cases} c_0 & \text{se } n = 0 \\ a\,T(n/b) + p(n^k) & \text{se } n > 0 \end{cases}$$

per $a, b \geq 1$ e $p(n^k)$ un polinomio di grado k.

Allora:

caso 1: $T(n) = \Theta(n^{log_b a})$ se $a > b^k$.

caso 2: $T(n) = \Theta(n^k lg\, n)$ se $a = b^k$.

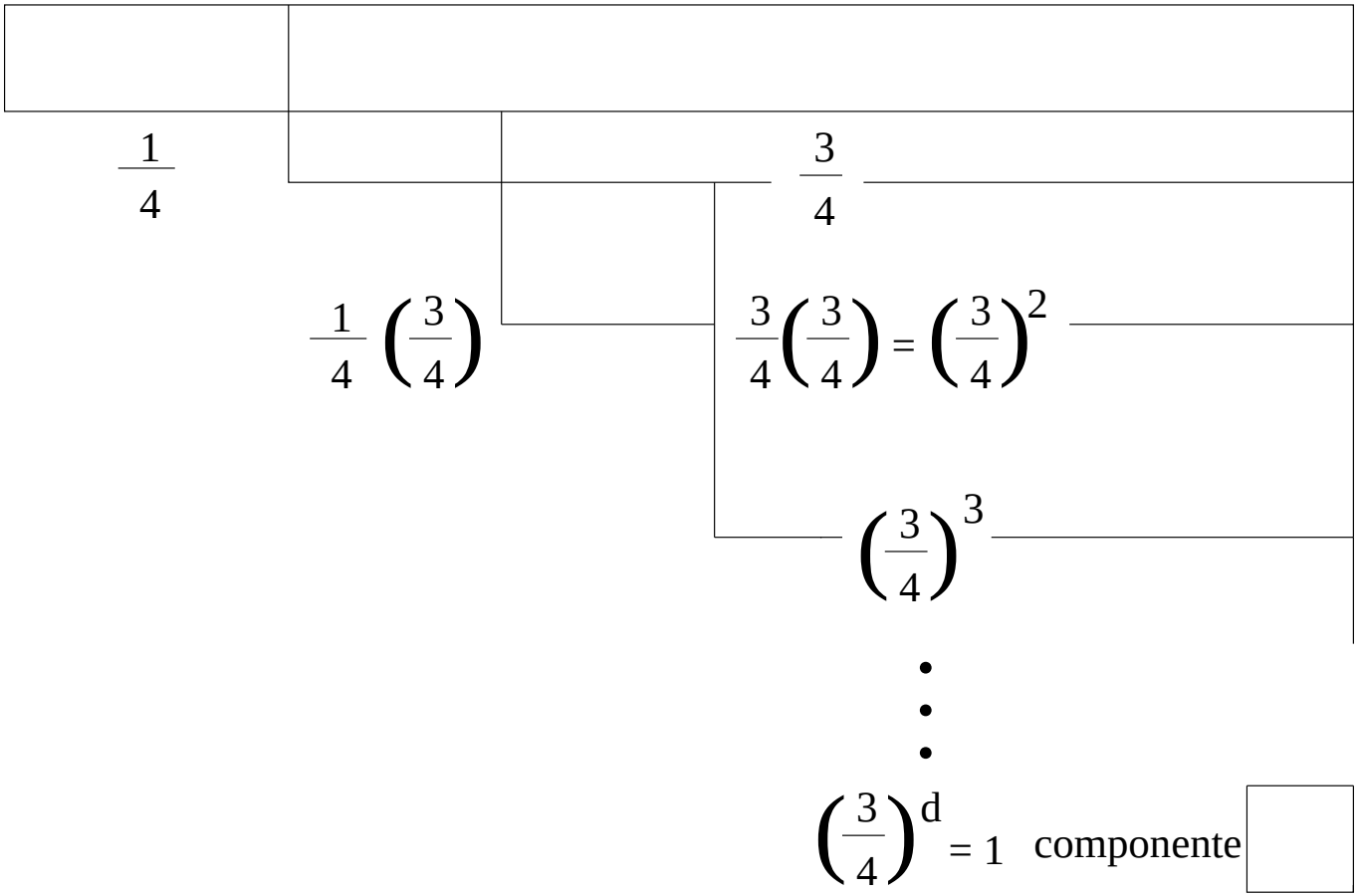
caso 3: $T(n) = \Theta(n^k)$ se $a < b^k$.

Complessità del Quick Sort - Caso medio (I)

Idea intuitiva: Distribuzione dei nodi nei due sottoarray - Caso generale

Il pivot partiziona sempre l'array in $1/4$ *e* $3/4$

$$T(n)_{quick} = \Theta(d \cdot n) = \Theta(n \cdot logn)$$



Complessità del Quick Sort - Caso medio (II)

Sia d il numero di partizioni che vengono effettuate: Per quale valore di d, $(3/4)^d n = 1$?

O equivalentemente (poiché $k = 1$ sse $log_2 k = 0$):

Per quale valore di d, $log_2 \left(\left(\frac{3}{4}\right)^d n \right) = 0$?

$$log_2 \left(\left(\frac{3}{4}\right)^d n \right) = 0$$

$$log_2 \left(\frac{3}{4}\right)^d + log_2 n = 0$$

$$d \cdot log_2 \left(\frac{3}{4}\right) + log_2 n = 0$$

$$d \cdot (-0.4) + log_2 n = 0$$

$$log_2 n = d \cdot 0.4$$

$$2.5 \cdot log_2 n = d \qquad (2.5 = \frac{1}{0.4})$$

$$d = 2.5 \cdot log_2 n$$

$$T(n)_{quick} = \Theta(d \cdot n) = \Theta(2.5n \cdot logn) = \Theta(n \cdot logn)$$

Come evitare il caso peggiore

- Si può modificare l'array imponendo una disposizione degli elementi che sia il più casuale possibile;
- quando si esegue la partition si sceglie a caso un elemento che viene scambiato col primo e che diventa il pivot.

`pseudorandom-number-generator: int*int --> int`

Applicato a due interi a e b, restituisce un numero nell'intervallo [a,b].

Nel caso del quick sort a e b sono gli indici minimo e massimo del sottoarray che si sta ordinando.

Poichè il costo della funzione `pseudorandom-number-generator` *è* $\Theta(1)$ *il costo complessivo del quick sort non cambia.*

Linear Congruential Method

X_0 valore iniziale $X_0 \geq 0$
a moltiplicatore $a \geq 0$
c incremento $c \geq 0$
m modulo $m > X_0,\ m > a,\ m > c$

$$X_{n+1} = (aX_n + c) \ \ mod \ \ m \qquad (*)$$

TH: La sequenza generata da $(_*)$ ha periodo di lunghezza m sse

i) $gcd(c, m) = 1$ (c e m sono relativamente primi tra loro);

ii) $b = a - 1$ è un multiplo di p per ogni p che divide m;

iii) b è un multiplo di 4 se m è un multiplo di 4.