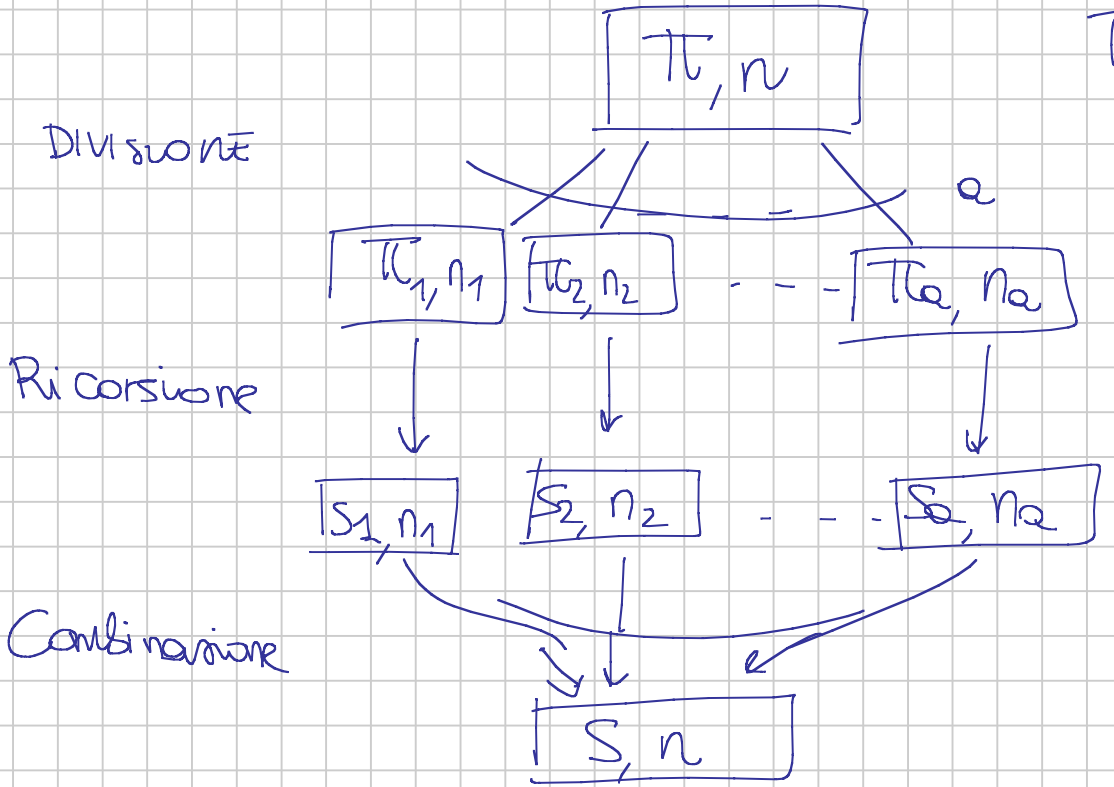
PARADIGMA del Divide et Impera

- **DIVISIONE** dividi il problema in sottoproblemi che operano su input di dimensione inferiore
- **RICORSIONE**
  - risolvi i sottoproblemi ricorsivamente
  - oppure direttamente se la loro dimensione è sufficientemente piccola
- **COMBINAZIONE** si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema originale



Π: problema
$n = |I|$ dim. input

$\forall i, \quad 1 \le i \le a$
$n_i < n$

$$T(n) = \begin{cases} \Theta(1) & n \le n_0 \\ D(n) + T(n_1) + T(n_2) + \dots + T(n_a) + C(n) & n > n_0 \end{cases}$$

$D(n)$: costo della divisione

$T(n_1) + T(n_2) + \dots + T(n_a)$: costo della ricorsione (risoluzione dei sottoproblemi)

$C(n)$: costo della ricombinazione

↳

$$T(n) = \begin{cases} \Theta(1) & n < n_0 \\ D(n) + a\,T\left(\frac{n}{b}\right) + C(n) & \end{cases}$$

b costante, $b > 1$

gli a sottoproblemi operano tutti su input di dimensione $\frac{n}{b}$

Dim. di correttezza → INDUZIONE

BASE correttezza su istanze di dim. sull. piccola
[caso base della ricorsione]

IP. INDUTTIVA: si assume che l'algoritmo risolva correttamente i sottoproblemi

PASSO si dimostra la correttezza dell'algoritmo relativo al problema generale esaminando la combinazione dei risultati parziali

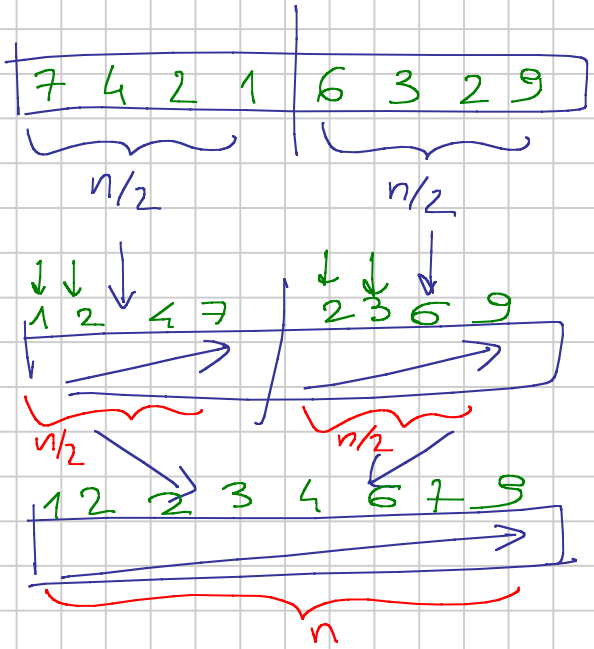
MERGE SORT

(1945, Von Neumann)

Divide et Impera

Ordinamento di un array di n interi $|I| = n$

n $n \geq 2$



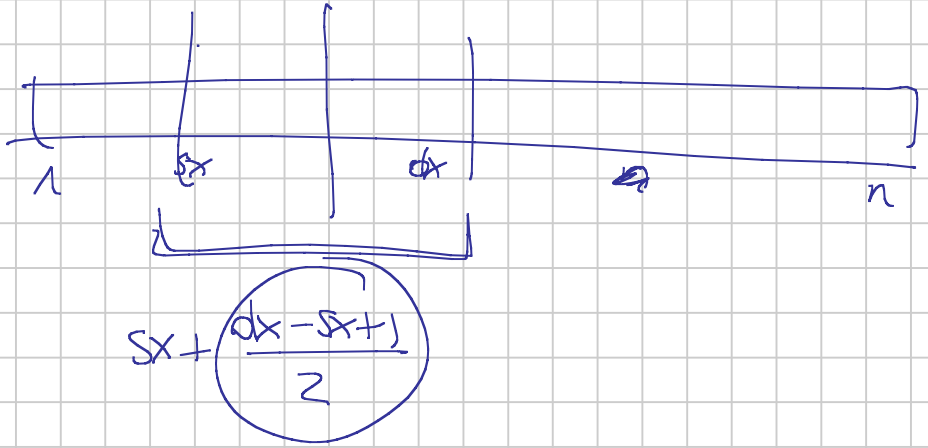
DIVISIONE

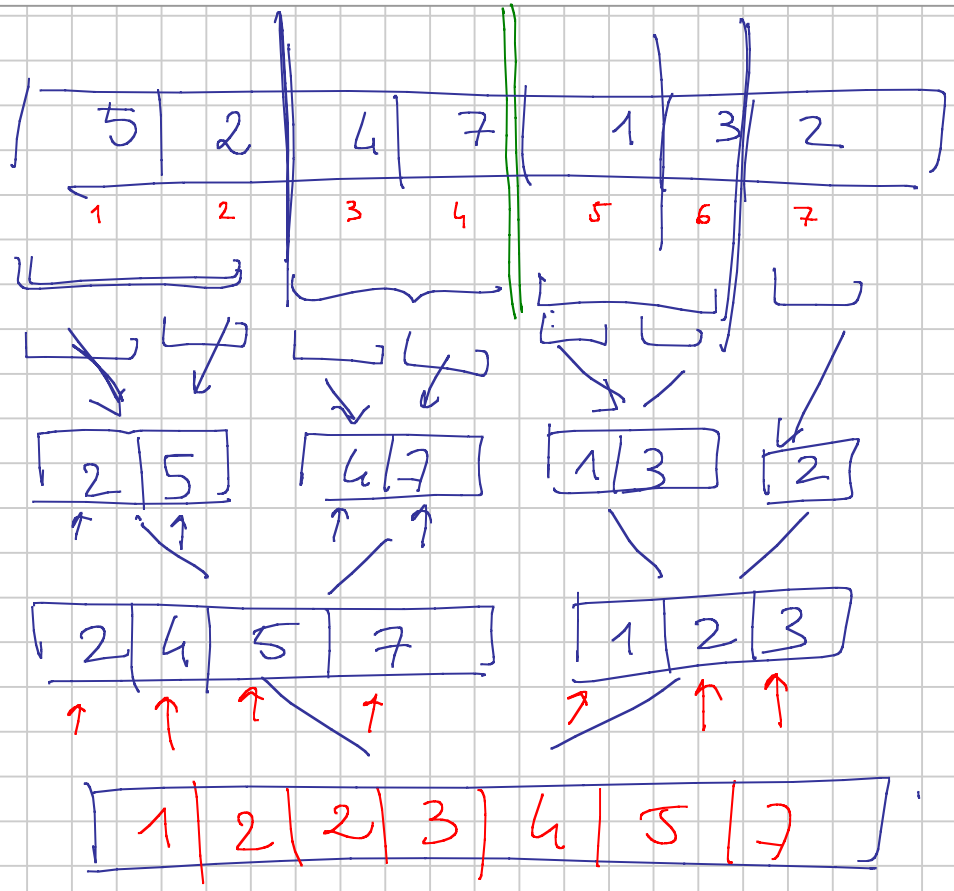
COMBINAZIONE (merge, fusione)

MergeSort(a, sx, dx)

```
if (sx < dx) {
    cx = ⌊(sx+dx)/2⌋
    MergeSort(a, sx, cx);
    MergeSort(a, cx+1, dx);
    Merge(a, sx, cx, dx);
}
```

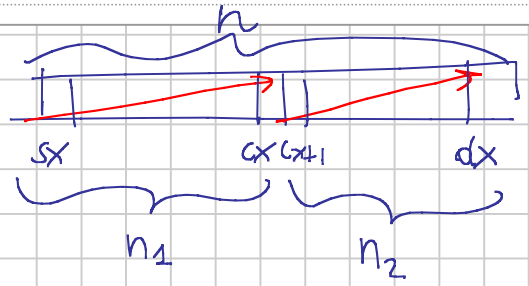
// ci sono almeno 2 elementi in a[sx...dx]





MS(a, 1, 7)
cx=4
MS(a, 1, 4)
cx=2
S(a, 1, 2)
cx=1
MS(a, 1, 1)
MS(a, 2, 2)
M(a, 1, 1, 2)
MS(a, 3, 4)
cx=3
MS(a, 3, 3)
MS(a, 4, 4)
M(a, 3, 3, 4)
M(a, 1, 2, 4)
MS(a, 5, 7)
M(a, 1, 4, 7)

Merge (a, sx, cx, dx) // $n = dx - sx + 1$;



$\Theta(1)$ {
- $n_1 = cx - sx + 1$ // # elementi in $a[sx..cx]$
- $n_2 = dx - cx$ // # elementi in $a[cx+1,\ldots,dx]$
- L = nuovo array di n_1+1 elementi
- R = nuovo array di n_2+1 elementi

}

$\Theta(n_1)$ { for $i = 1$ to n_1
$L[i] = a[sx + i - 1]$; // copio $a[sx..cx]$ in $L[1..n_1]$ }

$\Theta(n_2)$ { for $i = 1$ to n_2
$R[i] = a[cx + i]$; // copio $a[cx+1..dx]$ in $R[1..n_2]$ }

$\Theta(1)$ {
- $L[n_1+1] = \infty$
- $R[n_2+1] = \infty$

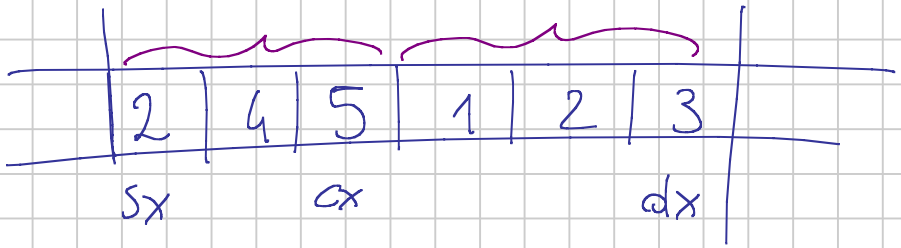
}

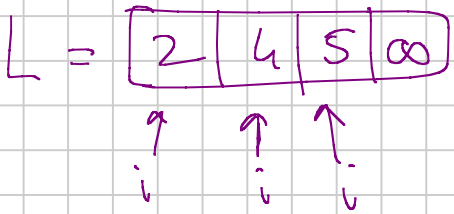
$\Theta(1)$ { `i = 1` // per scorrere L
`j = 1` // per scorrere R

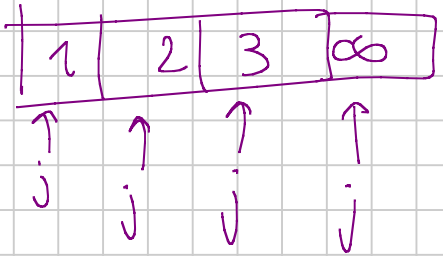
$\Theta(n_1+n_2)$ {
`for k = sx to dx {`
`if (L[i] ≤ R[j]) { a[k] = L[i]; i++; }`
`else { a[k] = R[j]; j++; }`
}

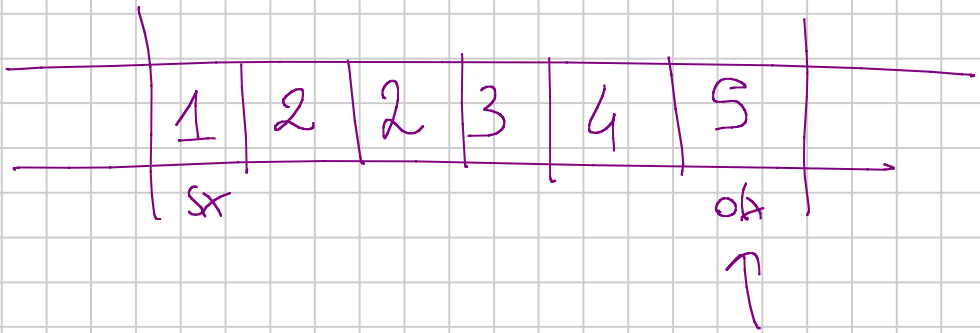
$n = n_1 + n_2$

$T_{merge}(n) = \Theta(1) + \Theta(n_1) + \Theta(n_2) + \Theta(n_1+n_2)$
$= \Theta(n_1+n_2) = \Theta(n)$


2 4 5 1 2 3
sx
cx
dx


L =
2 4 5 ∞
i
i
i


R =
1 2 3 ∞
j
j
j
j


1 2 2 3 4 5
sx
dx

```
MergeSort(a, sx, dx)                                   // n = dx - sx + 1
if (sx < dx) {
    cx = (sx + dx) / 2                                 } Θ(1)
    MergeSort(a, sx, cx)                               ~> T(⌈n/2⌉)
    MergeSort(a, cx+1, dx)                             ~> T(⌊n/2⌋)
    Merge(a, sx, cx, dx)                               } Θ(n)
}
```

caso base — divisione — ricorsione — combinazione

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ \underbrace{\Theta(1)}_{D(n)} + T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + \underbrace{\Theta(n)}_{C(n)} & n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ 2T\left(\tfrac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Analisi di correttezza

MERGE SORT

Per induzione su n (grandezza input)

CASO BASE $n=0$, $n=1$

array banalmente già ordinato

IPOTESI INDUTTIVA

MergeSort corretto su array di k elementi, $\forall k < n$

PASSO

x ipotesi induttiva dopo le 2 chiamate ricorsive,
$a[sx \dots cx]$ e $a[cx+1 \dots dx]$ sono ORDINATI
(contengono $n/2 < n$ elementi)

↝ la correttezza della funzione Merge garantisce che alla fine tutto a è ordinato

Correttezza MERGE

Invariante di ciclo (for triple)

All'inizio di ogni iterazione, $a[sx \dots k-1]$ contiene i $k-sx$ elementi più piccoli di $L[1 \dots n_1]$ e $R[1 \dots n_2]$, ORDINATI
inoltre $L[i]$ e $R[j]$ sono gli elementi più piccoli di L e R che NON sono ancora stati copiati in a

Inizializzazione

$k = sx$

↳ $a[sx \dots sx-1] = \emptyset$

L e R sono ordinati
⇒ $L[1]$ e $R[1]$ sono gli elementi più piccoli OK

Conservazione

il for seleziona l'elemento più piccolo di L e R che non è ancora stato copiato in A e lo scrive in posizione A[k]

$\Longrightarrow$ A[sx ... k] contiene i k−sx+1 elementi più piccoli di L e R

L[i] e R[j] sono gli elementi più piccoli di L e R non ancora copiati in A.

k → k+1

Conclusione

$k = dx + 1$ all'uscita del ciclo for

$a[sx \dots k-1] = a[sx \dots dx]$ contiene

i $k - sx = dx + 1 - sx = n$ elementi più piccoli

di L e R, ordinati,

in L e R sono rimaste le sentinelle

□