Merge Sort

MergeSort (A, p, r)

BASE: if (p < r) {

DIVISIONE: $q = \left\lfloor \frac{p+r}{2} \right\rfloor$

RICORSIONE: MergeSort (A, p, q)
MergeSort (A, q+1, r)

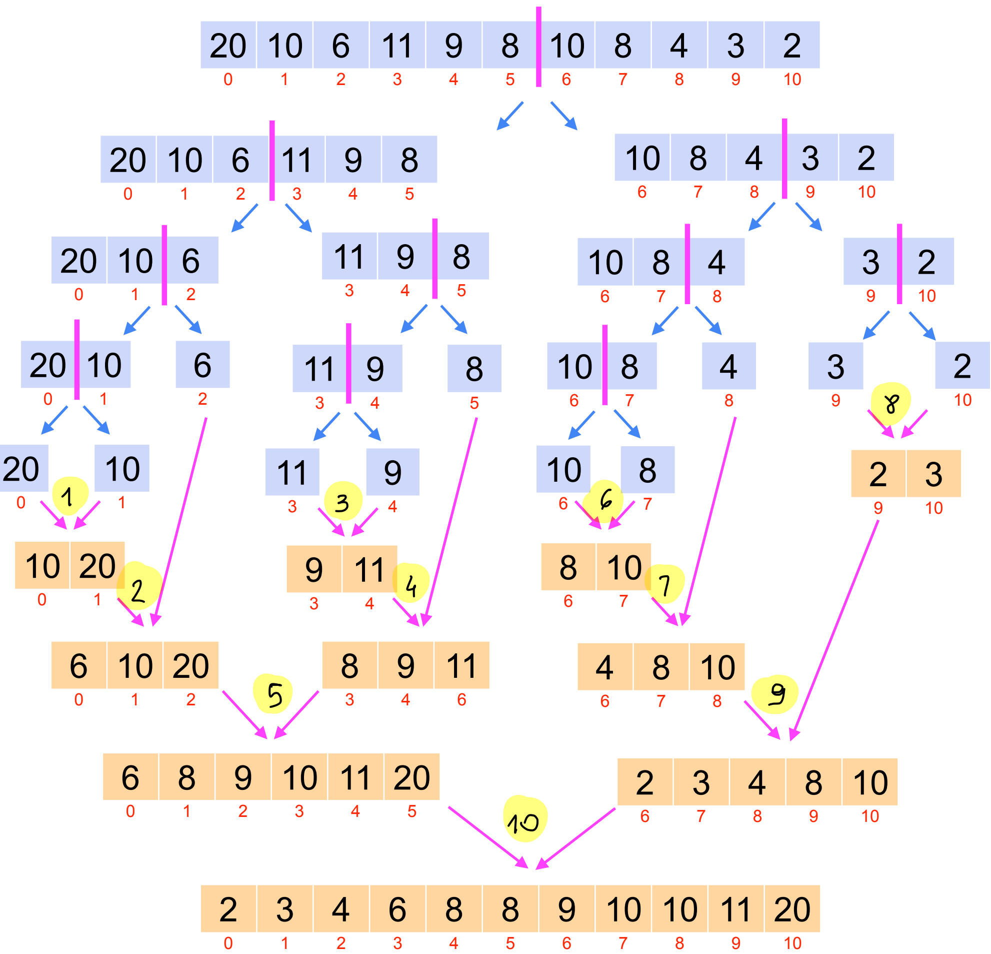
COMBINAZIONE: Merge(A, p, q, r)

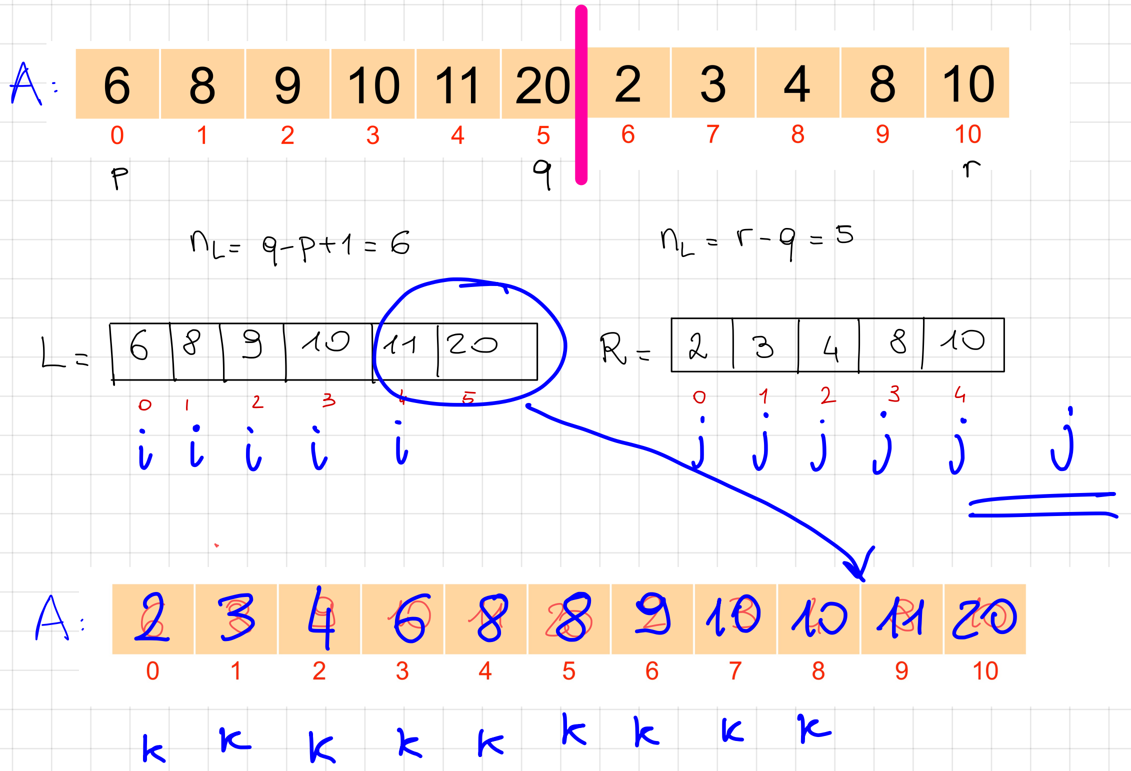
}

// chiusura caso base:
// lavoriamo solo se ci sono
// almeno 2 elementi in A[p...r]

Prima chiamata, per ordinare tutti gli elementi di A:

MergeSort (A, 0, n − 1)   n = A.length






```
MERGE(A, p, q, r)          // A: array di interi, p, q, r interi
  nL = q - p + 1           // dimensione di A[p...q]
  nR = r - q               // dimensione di A[q+1...r]
  int[] L = new int[nL]
  int[] R = new int[nR]
  for (i=0; i<nL; i++) { L[i] = A[p+i]; }      // copia di A[p...q] in L
  for (j=0; j<nR; j++) { R[j] = A[q+1+j]; }    // copia di A[q+1...r] in R
  i=0; // scorre L
  j=0; // scorre R
  k=p; // scorre A[p...r]
  while (i<nL && j<nR) {   // ci sono ancora elementi da fondere in L e in R
      if (L[i] ≤ R[j]) { A[k]=L[i]; i++; }    } si copia l'elemento più
      else { A[k]=R[j]; j++; }                 } piccolo ancora da fondere
      k++;                                       in A
  }
  while (i<nL) { A[k]=L[i]; i++; k++; }    } uno dei due array L e R è
  while (j<nR) { A[k]=R[j]; j++; k++; }    } stato completamente esaminato,
                                             si copiano gli elementi
                                             rimasti nell'altro alla fine di
                                             A[p..r]
```

```
MERGE ( A, p, q, r)                              n = nL + nR
nL = q - p + 1                          } O(1)
nR = r - q
int[] L = new int[nL]                                        } Θ(nL+nR)
int[] R = new int[nR]
for (i=0; i < nL; i++) { L[i] = A[p+i]; }        Θ(nL)
for (j=0; j < nR; j++) { R[j] = A[q+1+j]; }      Θ(nR)
i=0;
j=0;              } O(1)
k=p;
while ( i < nL && j < nR) {
        if ( L[i] ≤ R[j]) { A[k] = L[i]; i++; }
        else { A[k] = R[j]; j++; }                   Θ(nL+nR)
        k++;
}
while (i < nL) { A[k] = L[i]; i++; k++; }
while (j < nR) { A[k] = R[j]; j++; k++; }
```

Merge: analisi

$$n = n_L + n_R$$

$$T_{Merge}(n) = \Theta(1) + \underbrace{\Theta(n_L + n_R)}_{\text{copia in } L \text{ e } R} + \underbrace{\Theta(n)}_{\text{fusione in } A}$$

$$= \Theta(n)$$

- i primi due cicli for hanno un costo $\Theta(n_L + n_R)$
- ogni iterazione dei 3 cicli while copia un elemento da L o da R in A
- ogni valore è copiato esattamente una volta

↳ i 3 while complessivamente eseguono n iterazioni, ciascuna di costo costante

↳ $\Theta(n)$

COSTO in SPAZIO di MERGE : $S(n) = \Theta(n)$

usa gli array L e R di appoggio

Merge: correttezza

Invariante di ciclo: (primo while)

all'inizio di ogni iterazione del primo while, il segmento di array $A[p \dots k-1]$ contiene i $k-p$ elementi più piccoli di L e R, ordinati, inoltre $L[i]$ e $L[j]$ sono gli elementi più piccoli di L e R non ancora copiati in A

Merge Sort: correttezza

per induzione generalizzata su n

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

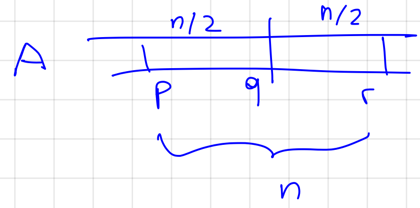
Merge Sort è corretto perché A è banalmente ordinato
(non fa nulla)

IPOTESI INDUTTIVA Merge Sort corretto su array di $k<n$ elementi

PASSO per ipotesi induttiva, dopo le due chiamate ricorsive $A[p \dots q]$ e $A[q+1 \dots r]$ sono ORDINATI
(contengono $n/2 < n$ elementi)
$\rightarrow$ la correttezza di Merge garantisce che alla fine A è ordinato

$\square$

MergeSort: analisi



MergeSort (A, p, r)

if (p < r) {

DIV. $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ → $O(1)$

MergeSort(A, p, q) → $T_{MS}(\lceil n/2 \rceil)$

MergeSort(A, q+1, r) → $T_{MS}(\lfloor n/2 \rfloor)$

Merge(A, p, q, r) → $\Theta(n)$

}

$$T_{MS}(n) = \begin{cases} O(1) & n \leq 1 \\ \underbrace{T_{MS}(\lceil n/2 \rceil) + T_{MS}(\lfloor n/2 \rfloor)}_{\text{costo della ricorsione (chiamate ricorsive)}} + \underbrace{\Theta(n)}_{\text{costo combinazione}} + \underbrace{O(1)}_{\text{costo divisione}} & n > 1 \end{cases}$$

trascuriamo i "dettagli tecnici" ($\lceil\,\rceil$, $\lfloor\,\rfloor$ ...)
e studiamo la ricorrenza semplificata

$$T_{MS}(n) = \begin{cases} O(1) & n \leq 1 \\ 2T_{MS}(n/2) + \Theta(n) & n > 1 \end{cases}$$

ipotesi: n sia una potenza di 2

$$T(n) = \begin{cases} c_0 & n \leq 1 \\ 2T(n/2) + c \cdot n & n > 1 \end{cases}$$

c costante

$T(n)$

$$cn$$

$$T(n/2) \qquad T(n/2)$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}$$

$$cn$$

$$c\cdot\frac{n}{2} \qquad cn/2$$

$$T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right)$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}$$

livello 0 $\qquad cn \qquad\qquad cn$

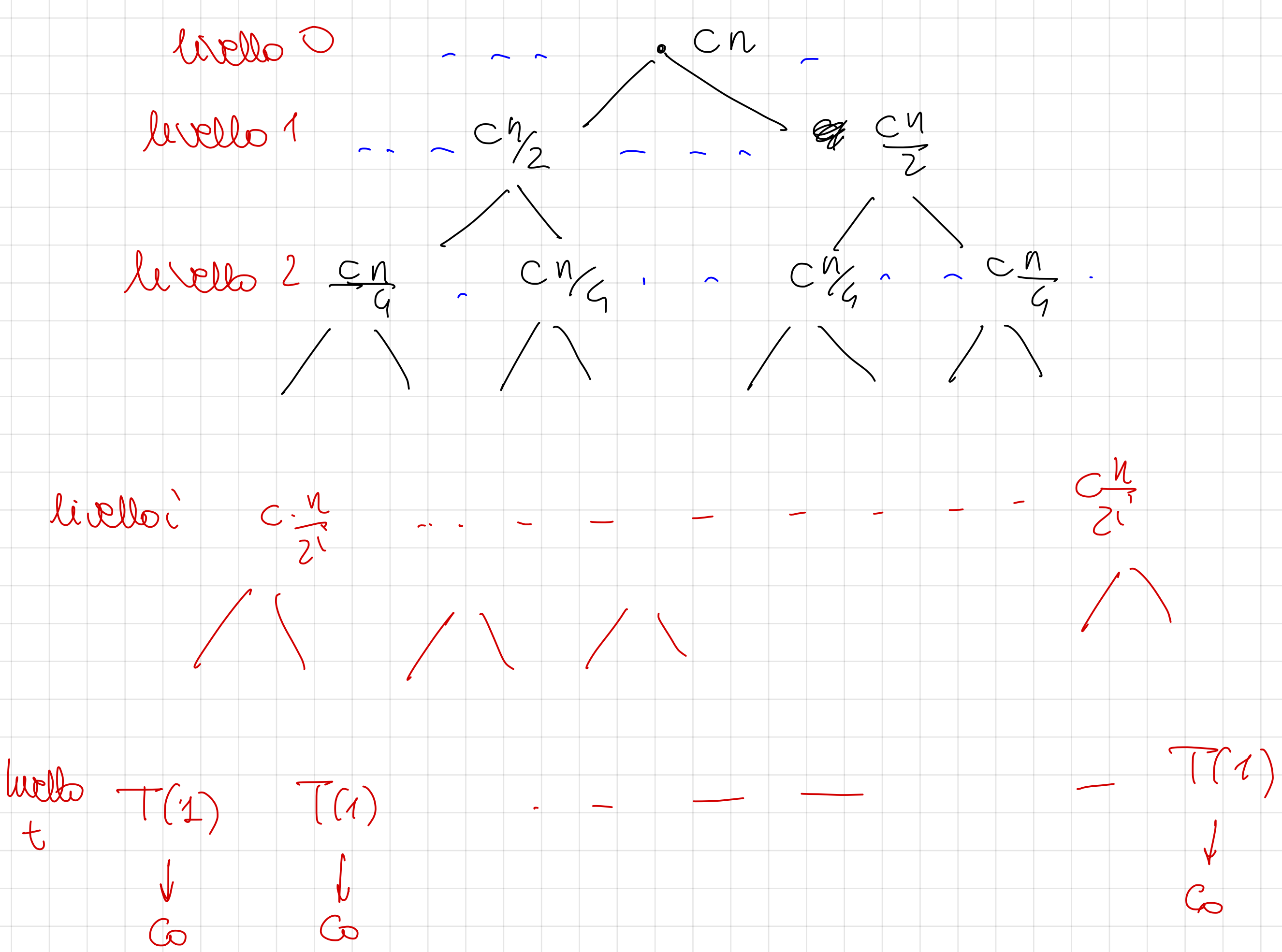
livello 1 $\qquad cn/2 \qquad \frac{cn}{2} \qquad\qquad 2\cdot\frac{c\cdot n}{2} = cn$

livello 2 $\qquad \frac{cn}{4} \quad cn/4 \quad cn/4 \quad \frac{cn}{4} \qquad\qquad 4\cdot\frac{cn}{4} = cn$

livello i : 2^i nodi, che rappresentano le 2^i chiamate ricorsive ~~che~~ su porzioni di $\frac{n}{2^i}$ elementi

ogni chiamata costa $c\cdot\frac{n}{2^i}$

costo sul livello i $\qquad 2^i\cdot\frac{c\cdot n}{2^i} = cn$



dopo t livelli si raggiunge il caso base: $T(1)$

t è t.c. $\frac{n}{2^t} = 1$

$$\Rightarrow \quad n = 2^t$$

$$\log_2 n = \underbrace{\log_2 2^t}_{t}$$

$$\Rightarrow \quad t = \log_2 n$$

lo sviluppo dell'albero si arresta nel livello

$t = \log_2 n$ (tutti i sottoproblemi hanno raggiunto il caso base)

nodi sul livello t: $2^t = 2^{\log_2 n} = n$ nodi

$T(1) = C_0$, ogni nodo sull'ultimo livello rappresenta un contributo C_0 al costo complessivo dell'algoritmo

$\Rightarrow$ $t = \log_2 n$ livelli di costo $c \cdot n$

\+ l'ultimo livello di costo $C_0 \cdot n$

$$T(n) = c \cdot n \cdot \underline{\log_2 n} + C_0 \, n = \Theta(n \log n)$$

Soluzione col metodo iterativo

$$T(n) = 2\, T\left(\tfrac{n}{2}\right) + c \cdot n = 2\left(2T\left(\tfrac{n}{4}\right) + c \cdot \tfrac{n}{2}\right) + cn$$

$$= 4T\left(\tfrac{n}{4}\right) + \cancel{2} \cdot c \cdot \tfrac{n}{\cancel{2}} + cn = 4T\left(\tfrac{n}{4}\right) + 2cn$$

$$= 4\left(2T\left(\tfrac{n}{8}\right) + c \cdot \tfrac{n}{4}\right) + 2cn$$

$$= 8T\left(\tfrac{n}{8}\right) + cn + 2cn = 2^3 T\left(\tfrac{n}{2^3}\right) + 3cn$$

$$= \dots = 2^i T\left(\tfrac{n}{2^i}\right) + i \cdot cn \underset{i = \log_2 n}{=}$$

$$= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot c \cdot n$$

$$= n\, T(1) + n \cdot c \cdot \log_2 n = nC_0 + nc \log n$$

$$\Rightarrow \quad \underset{\text{c.s.}}{T(n)} = \Theta(n \log n)$$

Merge Sort vs Insertion Sort

C_1, 1982, commodore 64 — 0.5 milioni ops/sec

$0.5 \cdot 10^6$ ops/sec

C_2, processore moderno — 10 miliardi ops/sec

10^{10} ops/sec

$$\text{tempo in secondi} = \frac{\#\text{ operazioni}}{\#\text{ ops/sec}} \nearrow \frac{n\log n}{0.5 \cdot 10^6} \text{ ops su } C_1 \quad \searrow \frac{n^2}{10^{10}} \text{ 1s su } C_2$$

C_1: Merge Sort — $n\log n$

C_2: Insertion Sort — n^2

A array da ordinare, A.length = n

n	C_1 merge Sort	C_2 Insertion Sort
10^4	~ 0.26 sec	~ 0.01 sec
10^5	~ 3.32 sec	~ 1 sec
10^6	~ 39 sec	~ 100 sec
10^7	~ 8 minuti	~ 167 minuti (3 ore)
10^8	~ 88 minuti	~ 11 giorni

Costo della ricerca binaria (ricorsiva)

$$T(n) \leq \begin{cases} c_0 & n = 0 \\ T(n/2) + c & n > 0 \end{cases}$$

Studiamo il caso della ricerca senza successo

c

$T(n/2)$

c

c

$T(n/4)$

c

c

c

livello i $\quad T\left(\frac{n}{2^i}\right)$ $\quad c \cdot (\log_2 n + 1)$

$i = \log_2 n + 1 \quad T(0) = c_0 \quad + \; c_0$

$$\left\lfloor \frac{n}{2^i} \right\rfloor = 0 \; ?$$

divisione intera

$$n < 2^i \qquad \frac{n}{2^i} = 0$$

$$\boxed{\log_2 n < i} \qquad \boxed{i = \log_2 n + 1}$$

si raggiunge il caso base

$$T(n) \leq c_0 + c(\log n + 1) \Rightarrow T(n) = O(\log n)$$

(vale $\Theta(\log n)$ nel caso della ricerca senza successo)

Metodo iterativo

$$T(n) \leq T(n/2) + c \leq \left(T(n/4) + c \right) + c = T\left(\frac{n}{4}\right) + 2c$$

$$\leq \left(T\left(\frac{n}{8}\right) + c \right) + 2c = T\left(\frac{n}{2^3}\right) + 3c \leq$$

$$= \dots \leq T\left(\frac{n}{2^i}\right) + ic \leq$$

$$i = \log_2 n + 1$$

$$\leq T\left(\frac{n}{2^{\log_2 n + 1}}\right) + (\log_2 n + 1) \cdot c =$$

$$= T\left(\frac{n}{2n}\right) + c(\log_2 n + 1) = T(0) + c \cdot (\log_2 n + 1)$$

$$= c_0 + c(\log_2 n + 1)$$