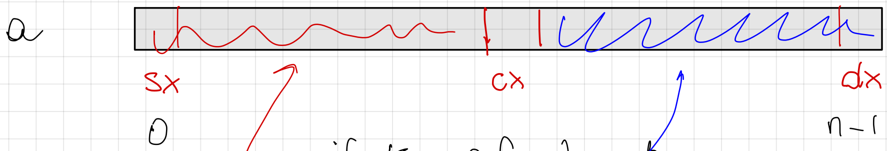
Ricerca Binaria Ricorsiva

array a ordinato

int k: chiave



$k \overset{?}{\in} a$

n = a.length

if k==a[cx] return cx;
else if k < a[cx] → cerco in a[sx... cx-1]
else // k > a[cx] → cerco in a[cx+1 .. dx]

```
int RicercaBinariaR(int[] a, int k, int sx, int dx)
    // array a ordinato, sx e dx estremi del segmento
    // di lavoro, k: chiave da cercare in a   [ k ∈? a[sx...dx] ]

    if (sx > dx) return -1;        // segmento vuoto

O(1) {
    if ( sx == dx) {
        if ( k == a[sx]) return sx;
        else return -1;
    }
    // se ci sono almeno 2 elementi:
O(1)    cx = ⌊(sx+dx)/2⌋          // divisione intera

O(1)    if (k == a[cx]) return cx;                           T(n/2)
        else if (k < a[cx]) return RicercaBinariaR(a, k, sx, cx-1);
        else // k > a[cx]                  T(n/2)
            return RicercaBinariaR(a, k, cx+1, dx)
```

Ricerca nell'intero array a : a[0 . . . . n-1]   n = a.length :

RicercaBinariaR(a, k, 0, n-1)

T(n): costo della ricerca in un segmento di n elementi

T(n/2): " " di n/2 "

$$T(n) \leq T(n/2) + O(1)$$

DIS. Eq. di Ricorrenze

Ricorrenza (valutazione della complessità in tempo di algoritmi ricorsivi)

Equazione, o disequazione, che esprime il costo in tempo di un algoritmo su input di dimensione n in funzione del costo in tempo dello stesso algoritmo su input di dimensione inferiore

$$T_{RB}(n) \leq T_{RB}(n/2) + O(1)$$

Soluzione (in forma chiusa): $T(n) = O(\log n)$

[DA DIMOSTRARE]

Algoritmi di ordinamento per confronti

- Algoritmi che effettuano l'ordinamento utilizzando solo confronti tra gli elementi di input, senza fare uso di altre primitive (operazioni aritmetiche, logiche, o altro)
- L'operazione dominante è il confronto tra elementi
  - ➡ il costo in tempo è proporzionale al numero di confronti effettuati

Θ

InsertionSort
SelectionSort

$O(n^2)$ confronti

andamento peggiore: corrisponde a fare tutti i possibili confronti tra le coppie di oggetti

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

InsertionSort: analisi

Caso pessimo

$$C(n) = \frac{n(n-1)}{2} = \binom{n}{2} \quad \in \Theta(n^2)$$

😔

fa tutti i confronti possibili!

Quando si verifica?

Caso ottimo

$$C(n) = n - 1 \quad \in \Theta(n)$$

🙂

Quando si verifica?

Caso medio

$$C(n) \simeq \frac{1}{2}\binom{n}{2} \quad \in \Theta(n^2)$$

😔

Costo in spazio

S(n) = O(1), ordina in loco

🙂

SelectionSort: analisi

Caso ottimo, medio e pessimo

😔

$$C(n) = \frac{n(n-1)}{2} = \binom{n}{2} \quad \in \Theta(n^2)$$

*fa **sempre** tutti i confronti possibili!*

Costo in spazio

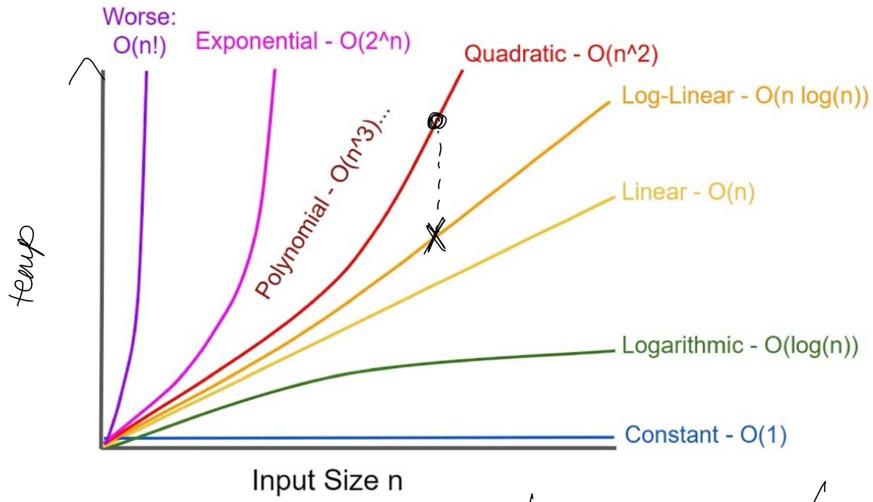
S(n) = O(1), ordina in loco

🙂

MergeSort
[redacted]

~~Θ~~$(n \log n)$ confronti

> *è quanto di meglio possiamo sperare di ottenere in un modello basato su confronti*



n	10	100	1000	10^6	10^9
$n \log n$	~ 33	~ 665	~ 10^4	~ $2 \cdot 10^7$	~ $3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Metodo Divide et Impera

problema Π, di dimensione n

Π, n $\quad T(n)$?

$\forall i, \ 1 \leq i \leq a$

$n_i < n$

DIVISIONE

si divide il problema in sottoproblemi che operano su istanze di dimensione ~~[illegible]~~ inferiore

$D(n)$ costo della divisione

$\Pi_1, n_1 \quad \Pi_2, n_2 \quad \dots \quad \Pi_a, n_a$

RICORSIONE

si risolvono i sottoproblemi ricorsivamente, con la stessa tecnica.

o direttamente se la dimensione è sufficientemente piccola

$T(n_1) + T(n_2) + \dots T(n_a)$

$S_1, n_1 \quad S_2, n_2 \quad S_a, n_a$

COMBINAZIONE

$C(n)$ costo della combinazione

S, n

si combinano le soluzioni dei sotto problemi, per ottenere la soluzione del problema originale

$$T(n) = \begin{cases} O(1) & n \leq n_0 \\ D(n) + \underbrace{T(n_1) + T(n_2) + \ldots + T(n_a)}_{\text{costo della ricorsione}} + C(n) & \end{cases}$$

$n \leq n_0$: costo della risoluzione diretta dei sottoproblema ~~el~~ elementare (di dim. piccole)

$D(n)$: costo della divisione

$C(n)$: costo della combinazione

costo delle operazioni svolte al di fuori delle chiamate ricorsive

Se i sottoproblemi ~~[illegible]~~ operano su istanze di pari dimensione:

$$T(n) = \begin{cases} O(1) & n \leq n_0 \\ aT(n/b) + f(n) & n > n_0 \end{cases}$$

a = # di sottoproblemi (a intero positivo)

$n/b \rightarrow$ dimensione dell'input dei sottoproblemi

$f(n)$: costo ~~del~~ della divisione e della combinazione
($f(n) = D(n) + C(n)$)

$$\frac{n}{b} < n \Rightarrow b > 1 \quad !!$$

Merge Sort

(Von Neumann, 1945)

A: array di n interi $n = A.length$

basato sul metodo D&I.

DIVISIONE

se $n < 2$, l'array è già (banalmente) ordinato

se $n \geq 2$: dividiamo la sequenza da ordinare a metà, in due sottosequenze di $n/2$ elementi ciascuna

A =

7	4	3	1	6	2	2	9

$n/2$ $n/2$

RICORSIONE

ordiniamo ricorsivamente le due sottosequenze

1	3	4	7	2	2	6	9

COMBINAZIONE

MERGE (fusione)

si fondono le due sottosequenze ordinate in un'unica sequenza ordinata

1	2	2	3	4	6	7	9

n

MergeSort(A, p, r)

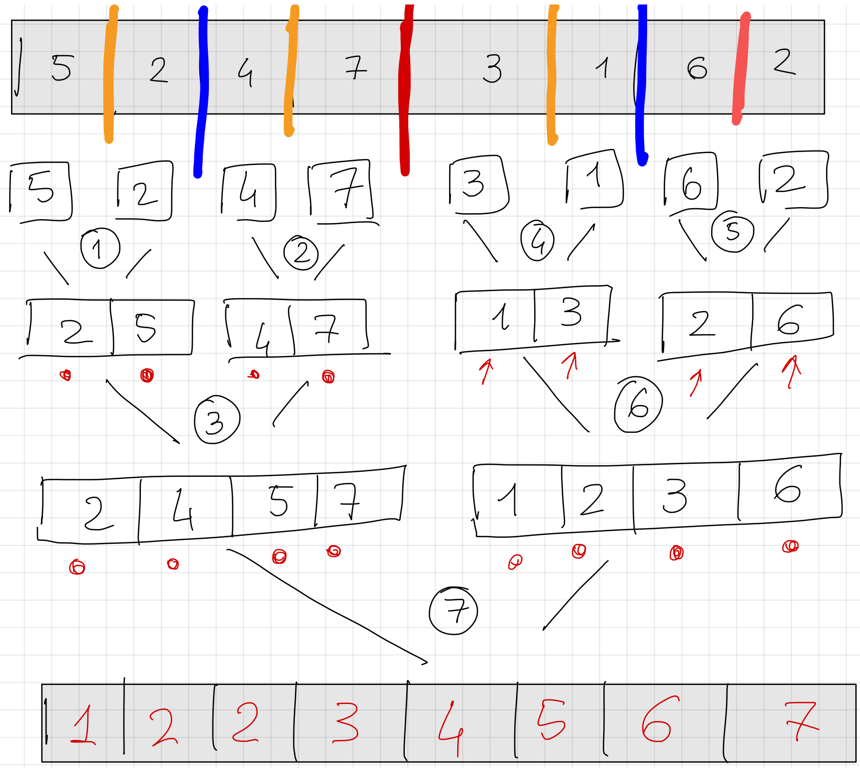
CASO BASE   if ($p < r$) {   // chiusura caso base: // lavoriamo solo se ci sono // almeno 2 elementi in A[p...r]

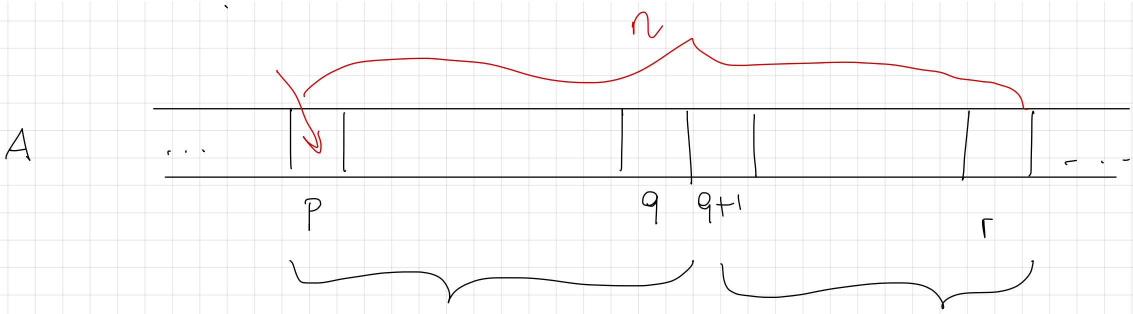
DIVISIONE $q = \left\lfloor \frac{p+r}{2} \right\rfloor$;

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

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

}





$n_L = q - p + 1$

↓

copiamo gli n_L elementi di $A[p \ldots q]$ in un array L

$n_R = r - (q+1) + 1 = r - q$

↓

copiamo gli n_R elementi di $A[q+1 \ldots r]$ in un array R

$n = n_L + n_R$ = # di elementi nel segmento $A[p \ldots r]$

Merge (A, p, q, r)

```
nL = q-p+1;        // dim di A[p...q]
nR = r-q ;         // dim 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 A[p...q] in L[0...nL-1]

for (j=0; j<nR; j++) { R[j] = A[q+1+j]; }  // copia A[q+1...r] in R[0...nR-1]

i=0;   // scorre l'array L
j=0    // scorre l'array R
k=p    // scorre e scrive su A[p...r]
while (i < nL && j < nR) {   // se ci sono ancora elementi da fondere in L e in R
    if (L[i] <= R[j]) { A[k] = L[i]; i++; }     copia l'elemento + piccolo in A[k]
    else { A[k] = R[j]; j++; }

    k++;
}
while (i < nL) { A[k] = L[i]; i++; k++; }
while (j < nR) { A[k] = R[j]; j++; k++; }
```

se uno dei due array L o R è stato esaminato completamente, si copiano gli elementi rimasti nell'array alla fine di A[p...r]