IMPORTANTE:

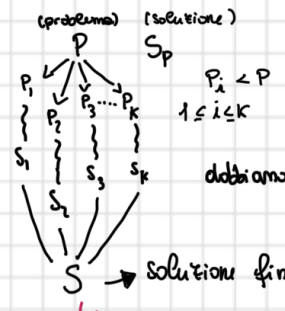
LA RICORSIONE

Ottimizzazione: caratteristica del problema che esiste a prescindere dalla soluzione
↳ un problema è di ottimizz. quando ci sono più soluzioni ma non tutte sono uguali. (se c'è una sola soluzione non è un prob. di ott.)
Tra tutte prendo la migliore (non è detto sia la più veloce)

GNU IS NOT UNIX → GNU acronimo ricorsivo

La ricorsione: deve essere effettuata su un problema dello stesso tipo ma deve diventare sempre più piccolo. Così posso risolverlo facilmente.
↳ chiama se stessa

(problema) (soluzione)



Passo della suddivisione

$P_i < P$ $\quad 1 \le i \le k$

dobbiamo solo suddividere il problema in sottoproblemi per poi unire le soluzioni dei sottoproblemi per formare la soluzione S finale.

S → soluzione finale.

L'importante è suddividere il problema in sottoproblemi più piccoli e più semplici da risolvere.

↳ Passo dell'unione

esempi FATTORIALE $F(n) = n!$ moltiplicatività dei naturali da 1 a n
$1 \cdot 2 \cdot 3 \cdot \dots \cdot n$

struttura generale del metodo ricorsivo: $F(n) \begin{cases} 1 & \text{se } n=1 \\ n \cdot F(n-1) & \text{se } n>1 \end{cases}$

F(n)
↓
F(n-1)
↓
n-1 !
↓
in

MUL (x, y) dimensione del problema può essere qualunque x in questo caso
per la moltiplicazione posso usare la somma

$x \times y = y + y + \dots + y$

$MUL(x,y) = \begin{cases} y & \text{se } x=1 \\ y + MUL(x-1, y) & \text{se } x>1 \end{cases}$

$POW(y,x) = \underbrace{y \times y \times y \times y \times \dots \times y}_{x}$ $\quad x$ dimensione

$POW(y,x) = \begin{cases} 1 & x=0 \\ y \times POW(y, x-1) & \text{se } x \ge 0 \end{cases}$

La RICORSIONE si può applicare anche alla struttura dati (es. albero binario di ricerca)
a destra elementi + grandi, a sinistra più piccoli.

(caso base)

BST { NULL
BST BST

definizione albero binario ricorsiva

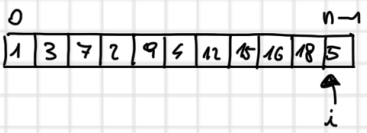
anche gli array

A { [ ]
□ ∼ A
alla concatenata all'array
tolgo una cella e mi rimane un sottoarray

concatenazione all'inizio o alla fine

IMPORTANTE CHE LA DIMENSIONE DIMINUISCA PIANO PIANO

- **Selection Sort**



Costo nella suddivisione

```
for i ← n-1 down to 1 (l'ultima iterazione inutile mi fermo ad 1) do
    j ← findmax(0, i)
    swap(i,j)
```

- **SelectionSort ricorsivo** (n) ← dimensione array

```
• IF n = 1 then return (caso base)
• j = findmax(0, n-1)
  swap(j, n-1)
  SelectionSort(n-1)
```

Soluzione più elegante ma ha lo stesso effetto

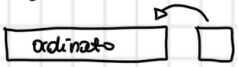
- **InsertionSort** (a sinistra parte ordinata, a poco a poco prendo elementi e li ordino)

```
for i ← 1 to n-1 do
    j ← i
    while j > 0 and A[j] < A[j-1] do
        swap(j, j-1)
        j ← j-1
```

costo nell'unione

da ripetere per n volte.

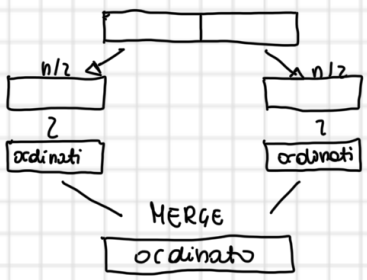
- **InsertionSort ricorsivo** (n) dim array (elimino da destra perché inserisco a sinistra)




```
• IF n = 1 return (caso base)
• INSERTIONSORT (n-1)   → più complesso perché devo avere una parte già ordinata quindi devo inserire chiamata ricorsiva prima
  j ← n-1
  while j > 0 AND A[j] < A[j-1] do
      SWAP(j, j-1)
      j ← j-1
```

- **Merge Sort**

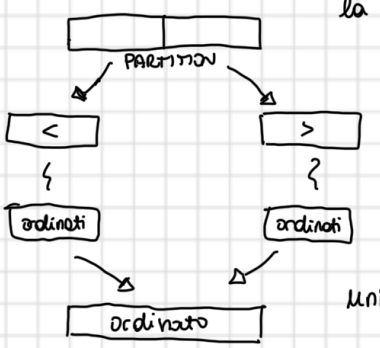
divide in 2 sott. della stessa dimensione



da ordinare un array (problema difficile)
a unire 2 array ordinati (problema facile)

costo nell'unione.

- **Quick Sort** a sinistra più piccoli, a destra più grandi
la suddivisione può essere sbilanciata (scambia sul posto gli elementi)



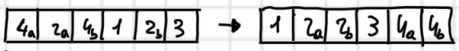
costo (perdi più tempo) nella suddivisione

unire non mi costa nulla già fatta

algoritmo di ordinamento

- lavora in loco quando non usa memoria aggiuntiva rispetto a quella fornita
  MERGESORT non in loco

- STABILITÀ DI UN ALGORITMO
  ↳ mantiene posizione relativa delle chiavi doppie.



es. insertionsort.
selectionsort (≥ perché il 1° verrà sostituito dal 2°)
- quicksort è l'unico che non è stabile.

- alcuni algoritmi sono adattivi: sulla base dell'input si adatta
  ↳ es. insertionsort.

$O(n \cdot k)$ dove k è la distanza massima