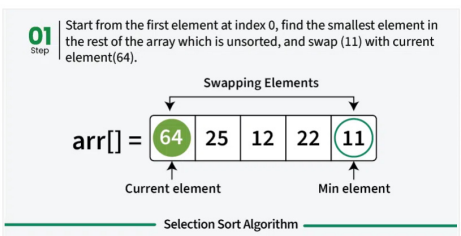
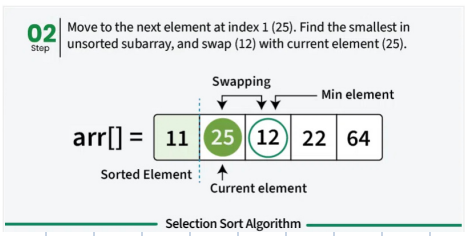
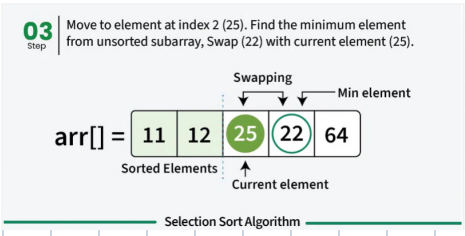
SELECTION SORT

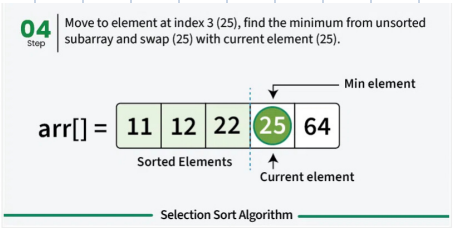
SCORRIAMO TUTTO L'ARRAY INIZIANDO DALLA PRIMA POSIZIONE E ASSUMENDO CHE L'ELEMENTO IN QUELLA POSIZIONE SIA IL MINIMO, DOPODICHÈ SCORRIAMO L'ARRAY PARTENDO DALL'ELEMENTO SUCCESSIVO, SE TROVIAMO UN ELEMENTO PIÙ PICCOLO DEL NOSTRO MINIMO ALLORA LI SCAMBIAMO.

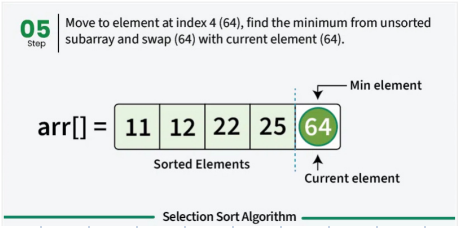
AD OGNI ITERAZIONE ASSUMIAMO INIZIALMENTE CHE L'ELEMENTO IN CUI SIAMO SIA IL MINIMO.
SE L'ELEMENTO IN CUI SIAMO È GIÀ IL MINIMO NON FACCIAMO NESSUNO SCAMBIO.

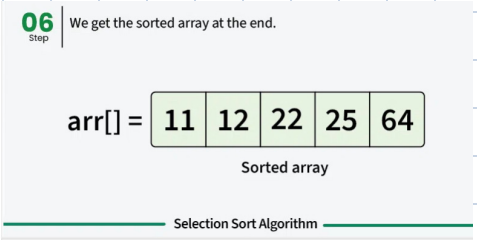













```python
def selectionSort(array):

    size = len(array)

    for i in range(0,size):

        #assumiamo che il minimo sia l'elemento corrente
        min_pos = i

        #cerchiamo il minimo tra gli elementi successivi, se non esiste min_pos rimane invariato
        for j in range(i+1,size):

            if array[j] < array[min_pos]:

                min_pos = j #assegnamo la posizione del nuovo minimo

        #scambiamo i due elementi, se min_pos non è cambiato si scambia con se stesso quindi non cambia nulla
        swap = array[i] #variabile di appoggio
        array[i] = array[min_pos]
        array[min_pos] = swap

    return array
```

ALCUNI ALGORITMI ARRIVANO AL PENULTIMO ELEMENTO CON RANGE(SIZE - 1) PER EVITARE L'ITERAZIONE SULL'ULTIMO ELEMENTO CHE SARÀ GIÀ ORDINATO.

DESCRIZIONE MIGLIORE DI SELECTION SORT

SCORRIAMO MANO MANO L'ARRAY, AD OGNI ITERAZIONE ASSUMIAMO CHE L'ELEMENTO IN CUI SIAMO SIA IL PIÙ PICCOLO, SALVIAMO QUINDI LA SUA POSIZIONE IN minPos, ADESSO CON UN SECONDO CICLO ANDIAMO A CERCARE L'ELEMENTO PIÙ PICCOLO

↳ COME? ⟶ CONFRONTIAMO OGNI ELEMENTO A[J] CON A[minPos], SE A[J] È PIÙ PICCOLO SALVIAMO minPos = J QUINDI STIAMO SALVANDO LA POSIZIONE DELL'ELEM PIÙ PICCOLO.

ALLA FINE DEL CICLO INTERNO SCAMBIAMO A[i] e A[minPos], NEL CASO ABBIAMO TROVATO UN ELEMENTO MINORE DI A[i] ALLORA LO STIAMO METTENDO AL POSTO DI A[i] CHE È PIÙ PICCOLO, SE NON ABBIAMO TROVATO NESSUN ELEM. PIÙ PICCOLO minPos SARÀ UGUALE AD i, E QUINDI SCAMBIAMO L'ELEMENTO CON SE STESSO.

UPPER BOUND: $T(m) \leq 5 \cdot m^2 \cdot O(1) \longrightarrow T(m) = O(m^2)$, PER VIA DEI LOOP INNESTATI

MA T(m) POTREBBE ESSERE $\Theta(m^2)$? LO ANDIAMO A VERIFICARE STIMANDO IL:

LOWER BOUND:

NUMERO DI ITERAZIONI CHE FA IL CICLO INTERNO

NUMERO DI ITERAZIONI CHE FA IL CICLO INTERNO

$$T(m) \geq \sum_{k=0}^{m-2} (m-k-1)$$ CHE POSSO RISCRIVERE COME $$\sum_{k=1}^{m-1} k = m(m-1)/2 = \Theta(m^2)$$

ATTENZIONE ARRIVIAMO A m-2 PERCHÈ UTILIZZIAMO L'ALGORITMO CHE EVITA L'ULTIMO CONFRONTO

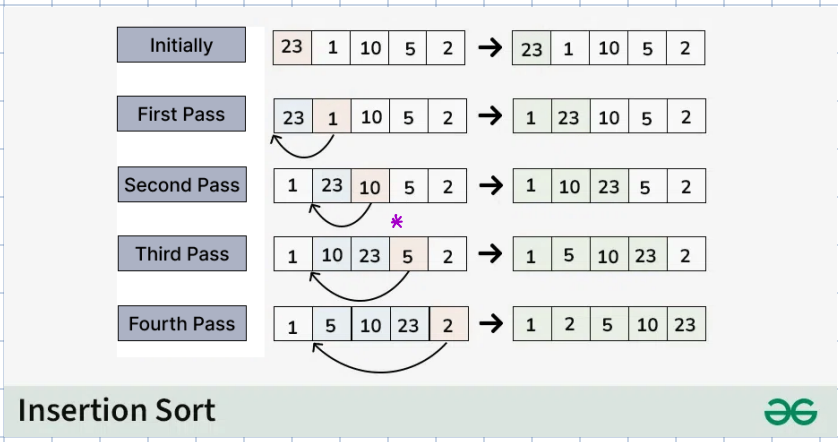
SE CI FACCIAMO CASO È LA SOMMATORIA DEI PRIMI m NUMERI REALI

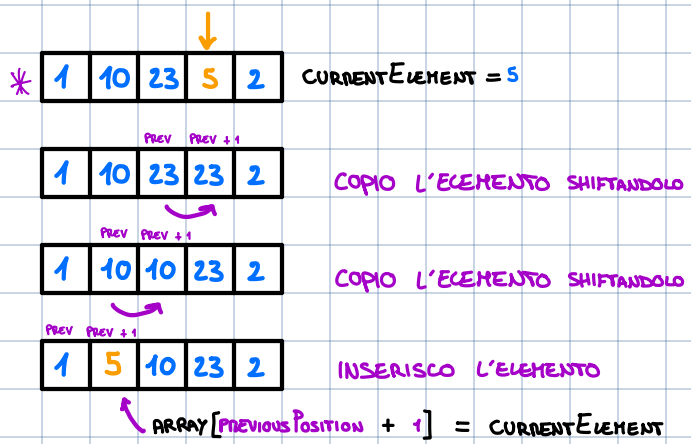
$$T(m) = \Theta(m^2)$$ ✓

INSERTION SORT

INIZIAMO DALL' ELEMENTO IN SECONDA POSIZIONE, SE QUESTO DOVESSE ESSERE MINORE DI QUELLO NELLA POSIZIONE PRECEDENTE ALLORA FACCIAMO UNO SHIFT E INSERIAMO L'ELEMENTO NELLA POSIZIONE PRECEDENTE.

ANDANDO AVANTI NELLE ITERAZIONI QUELLO CHE FACCIAMO È PRENDERE L'ELEMENTO CORRENTE, SCORRENDO ALL' INDIETRO OGNI VOLTA CHE TROVIAMO UN ELEMENTO MAGGIORE DEL NOSTRO LO SHIFTIAMO A DESTRA, QUANDO NON TROVEREMO PIÙ ELEMENTI MAGGIORI DEL NOSTRO CI INSERIAMO IL NOSTRO ELEMENTO CHE AVEVAMO SALVATO IN UNA VARIABILE INIZIALE.





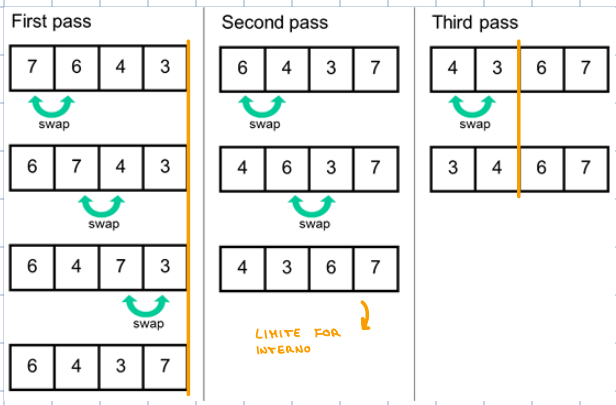
```python
def insertionSort(array):

    # Itero l'array a partire dal secondo elemento
    for i in range(1, len(array)):

        # Memorizzo l'elemento corrente e la posizione precedente
        currentElement = array[i]
        previousPosition= i - 1

        # Itero l'array all'indietro fino a che ci sono elementi più grandi di quello corrente
        # o fino a che non arrivo alla fine (verso sinistra)
        while previousPosition >= 0 and currentElement < array[previousPosition]:

            # Sposto l'elemento più grande di una posizione a destra (shift a destra)
            array[previousPosition + 1] = array[previousPosition]

            # Decremento la posizione precedente andando sempre più indietro (verso sinistra)
            previousPosition -= 1

        # Se sono qui, significa che ho trovato la posizione corretta per l'elemento corrente
        # perchè tutti gli elementi a sinistra sono più piccoli di quello corrente

        # Quindi inserisco l'elemento corrente nella posizione corretta (ovvero previousPosition + 1)
        array[previousPosition + 1] = currentElement
```

$O(n^2)$ ✓

BUBBLE SORT

IL BUBBLE SORT UTILIZZA LA TECNICA A "BOLLA" XCHÉ APPUNTO AD OGNI CICLO FA SALIRE L'ELEMENTO PIÙ GRANDE ALL'ULTIMA POSIZIONE A DESTRA.

CON UN PRIMO CICLO ESTERNO EFFETTUA n ITERAZIONI, AD OGNI ITERAZIONE CON UN SECONDO CICLO EFFETTUA $n - i - 1$ ITERAZIONI OVVERO OGNI VOLTA CHE VIENE FATTO IL CICLO ESTERNO (PRIMO FOR), IL CICLO INTERNO DECREMENTA DI 1 IL VALORE A CUI ARRIVARE, SE IL CICLO ESTERNO FA 4 ITERAZIONI ALLORA QUELLO INTERNO NE FARÀ: 3, 2, 1, 0.

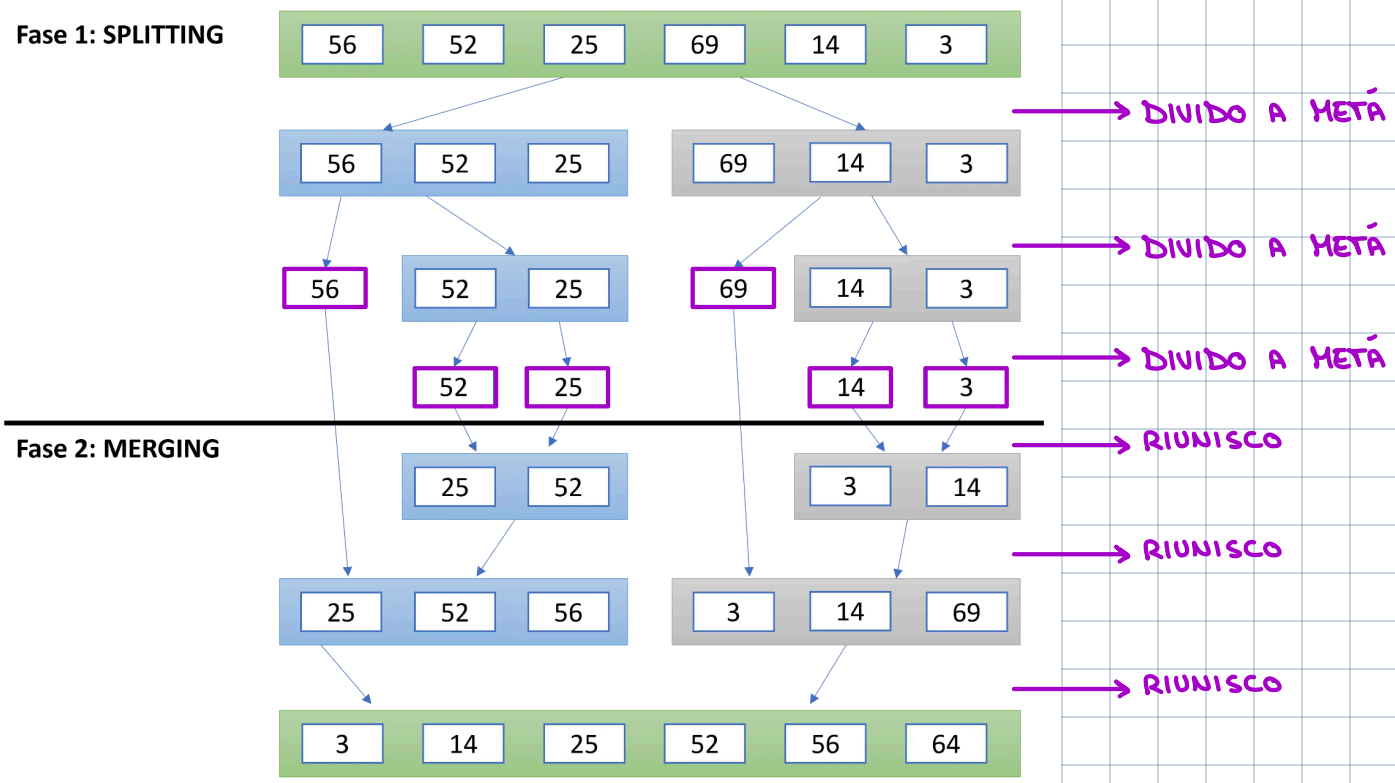



```python
def bubble_sort(arr):

    n = len(arr)

    # Ciclo n volte l'array
    for i in range(n):

        # Ogni iterazione sposta il numero più grande alla fine, ogni volta ciclo fino all'elemento n - i - 1
        # Quindi sempre meno elementi da confrontare
        for j in range(0, n - i - 1):

            # Confronta l'elemento corrente con quello successivo
            if arr[j] > arr[j + 1]:

                # Scambia se l'elemento corrente è maggiore del successivo
                temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
    return arr
```

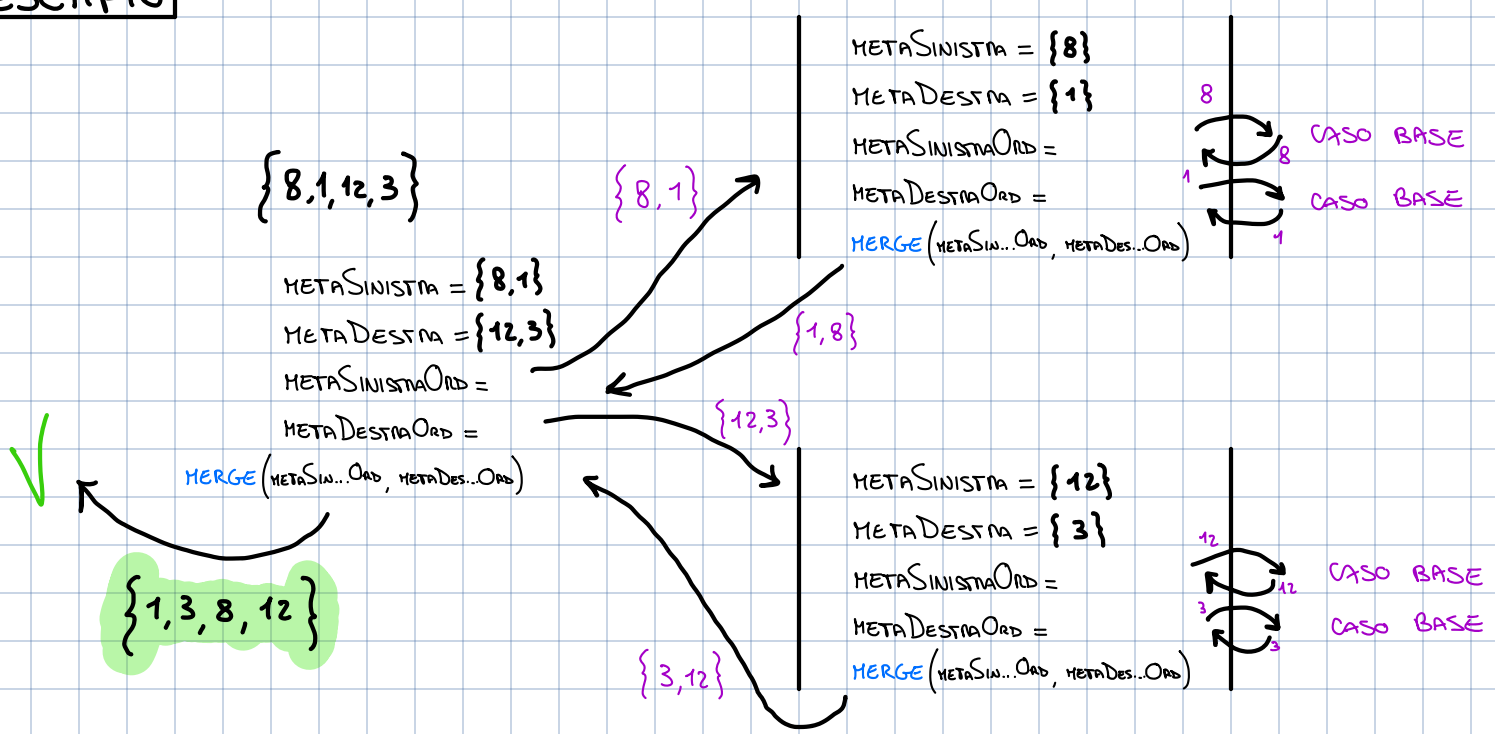
$O(n^2)$ ✓

MERGE SORT

IL MERGE SORT UTILIZZA LA TECNICA DIVIDI ET IMPERA, L'ARRAY VIENE DIVISO SEMPRE DI PIÙ A METÀ FINO A CHE NON SI RAGGIUNGE UNA DIMENSIONE == 1, QUANDO SI RAGGIUNGE QUESTO I SINGOLI ELEMENTI VENGONO RIORDINATI E POI RIUNITI.



ESEMPIO



```python
def merge_sort(arr):

    # Condizione di base: se la lista ha uno o zero elementi, è già ordinata
    if len(arr) <= 1:
        return arr

    # Dividi la lista in due metà
    meta = len(arr) // 2
    metaSinistra = arr[:meta] # Da 0 a meta-1
    metaDestra = arr[meta:] # Da meta a len(arr)-1

    # Ordina ricorsivamente le due metà
    metaSinistraOrdinata = merge_sort(metaSinistra)
    metaDestraOrdinata = merge_sort(metaDestra)

    # Unisci le due metà ordinate
    return merge(metaSinistraOrdinata, metaDestraOrdinata)

```

```python

def merge(left, right):
    sorted_array = []
    i = j = 0

    # Confronta gli elementi di 'left' e 'right' e ad ogni iterazione aggiungi il minore
    while i < len(left) and j < len(right):

        if left[i] < right[j]:

            # Se l'elemento di 'left' è minore, aggiungilo alla lista ordinata
            # e incrementa l'indice per scorrere 'left'
            sorted_array.append(left[i])
            i += 1

        else:

            # Se l'elemento di 'right' è minore, aggiungilo alla lista ordinata
            # e incrementa l'indice per scorrere 'right'
            sorted_array.append(right[j])
            j += 1

    # Aggiungi gli elementi rimanenti, se ce ne sono, da 'left' e 'right'
    sorted_array.extend(left[i:])
    sorted_array.extend(right[j:])

    return sorted_array
```

ANALISI COMPLESSITÀ

ABBIAMO UNA COMPLESSITÀ DATA DALLA RELAZIONE DI RICORRENZA:

$$T(m) = \underbrace{2 \cdot T(m/2)}_{\text{AD OGNI CHIAMATA ABBIAMO 2 NODI FIGLI DI DIMENSIONE } \frac{m}{2}} + \underbrace{O(m)}_{\text{COSTO DI FUSIONE PER OGNI CHIAMATA (APPARTE IL CASO BASE)}}$$

CHE USANDO IL TEOREMA MASTER SI TRASFORMA IN:

$$T(m) = \underbrace{2 \cdot T(m/2)}_{a \cdot T(\frac{m}{b})} + \underbrace{O(m)}_{f(m)}$$

CON $a = 2$ e $b = 2$ E QUINDI: $m^{\log_b a} = m^{\log_2 2} = m^1$

ABBIAMO QUINDI: m vs $m^{\log_2 2}$ ($\nearrow 1$) E SAPPIAMO CHE $m = \Theta(m)$ SIAMO QUINDI NEL CASO 2 DEL TEOREMA MASTER, E QUINDI:

$$T(m) = m^{\log_b a} \log m$$

OVVERO

$$m^{\log_2 2} \log m \longrightarrow T(m) = \Theta(m \log m) \checkmark$$

ANALISI MEMORIA UTILIZZATA

LA MEMORIA UTILIZZATA DAL MERGE SORT È $\Theta(m)$ POICHÉ VIENE UTILIZZATO UN ARRAY AUSILIARIO DI DIMENSIONE m PER LA FUSIONE DEI SUB-ARRAY ORDINATI.

STACK CALL MERGESORT

```python
def merge_sort(arr):

    # Condizione di base: se la lista ha uno o zero elementi, è già ordinata
    if len(arr) <= 1:
        return arr

    # Dividi la lista in due metà
    meta = len(arr) // 2
    metaSinistra = arr[:meta] # Da 0 a meta-1
    metaDestra = arr[meta:] # Da meta a len(arr)-1

    # Ordina ricorsivamente le due metà
    metaSinistraOrdinata = merge_sort(metaSinistra)
    metaDestraOrdinata = merge_sort(metaDestra)

    # Unisci le due metà ordinate
    return merge(metaSinistraOrdinata, metaDestraOrdinata)

def merge(left, right):
    sorted_array = []
    i = j = 0

    # Confronta gli elementi di 'left' e 'right' e ad ogni iterazione aggiungi il minore
    while i < len(left) and j < len(right):

        if left[i] < right[j]:

            # Se l'elemento di 'left' è minore, aggiungilo alla lista ordinata
            # e incrementa l'indice per scorrere 'left'
            sorted_array.append(left[i])
            i += 1

        else:

            # Se l'elemento di 'right' è minore, aggiungilo alla lista ordinata
            # e incrementa l'indice per scorrere 'right'
            sorted_array.append(right[j])
            j += 1

    # Aggiungi gli elementi rimanenti, se ce ne sono, da 'left' e 'right'
    sorted_array.extend(left[i:])
    sorted_array.extend(right[j:])

    return sorted_array
```

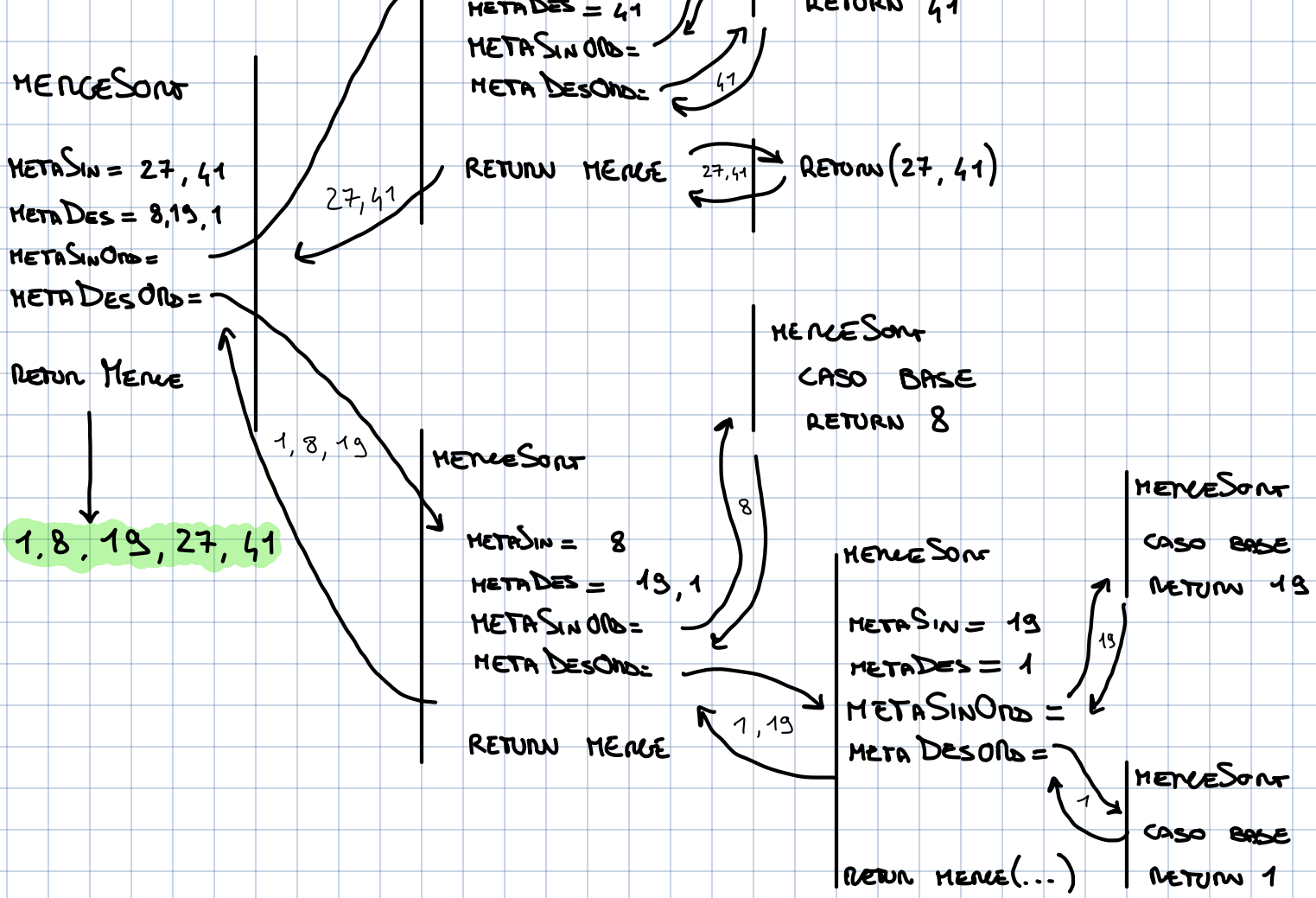
27, 41, 8, 19, 1

MERGESORT
CASO BASE
RETURN 27

MERGESORT
CASO BASE

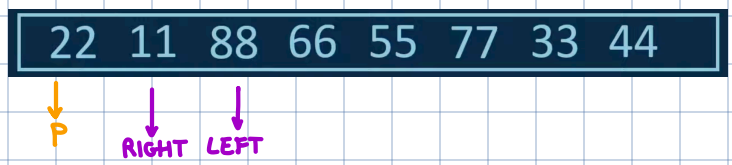
MERGESORT
METASIN = 27

27



QUICK SORT

VIENE SCELTO UN PIVOT, DI SOLITO IL PRIMO ELEMENTO, DOPO DI CHÉ SI PRENDONO DUE CONTATORI LEFT e RIGHT CHE SONO I CONTATORI CHE USEREMO PER ITERARE L'ARRAY.



CON LEFT ANDIAMO A SCORRERE GLI ELEMENTI DA SX A DX FINCHÉ NON TROVIAMO UN ELEMENTO MAGGIORE DEL PERNO, CON RIGHT FACCIAMO L'INVERSO, USCITI DAI CICLI AVREMO:

- LEFT ⟶ POSIZIONE DEL PRIMO ELEM. DA SX, MAGGIORE DEL PIVOT
- RIGHT ⟶ POSIZIONE DEL PRIMO ELEM. DA DX, MINORE DEL PIVOT

ORA LI SCAMBIAMO, FACENDO CIÓ STIAMO PORTANDO UN ELEMENTO > PIVOT NELLA PARTE DX DELL' ARRAY e L'ELEMENTO < PIVOT NELLA PARTE SX DELL' ARRAY.

QUANDO LEFT NON SARÀ PIÚ MINORE DI RIGHT SIGNIFICA CHE I CONTATORI SI

SARANNO INVERTITI E CHE QUINDI AVREMO UN PUNTO MEDIO ALLA QUALE A SX CI SARANNO GLI ELEM. MINORI DEL PIVOT E A DX QUELLI MAGGIORI.

INFINE SCAMBIA IL PRIMO ELEMENTO DELL'ARRAY CON L'ELEMENTO IN POSIZIONE RIGHT COSÌ METTENDO IL PIVOT AL CENTRO.

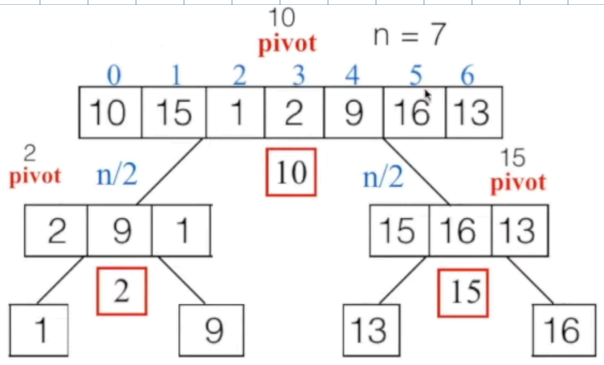
RICHIAMIAMO RICORSIVAMENTE IL QUICK SORT DUE VOLTE, LA PRIMA CHIAMATA SULLA PARTE A SX DEL PIVOT e LA SECONDA SU QUELLI MAGGIORI DEL PIVOT.

- Scegli il perno
- Scorri l'array "in parallelo" da sinistra verso destra e da destra verso sinistra
  - da sinistra verso destra, ci si ferma su un elemento maggiore del perno
  - da destra verso sinistra, ci si ferma su un elemento minore del perno
- Scambia gli elementi e riprendi la scansione
- Fermati quando i due indici si sono incrociati

"PARTIZIONE IN LOCO"
OVVERO SUDDIVIDE L'ARRAY SENZA L'UTILIZZO DI MEMORIA AGGIUNTIVA. !!

QuickSort (A, i, f)

1. **if** (i < f) **then**
2. m=Partition(A,i,f)
3. QuickSort(A,i,m-1)
4. QuickSort(A, m +1,f)



PRIMA POSIZ. — POSIZ. FINALE (+1)

```
PARTITION (l, h):
    PIVOT = A[l];   → PRIMO ELEM.
    i = l;   j = h;

    WHILE (i < j):
        DO:
            i++;
        WHILE (A[i] ≤ PIVOT);

        DO:
            j--;
        WHILE (A[j] > PIVOT);

        IF (i < j):
            SWAP (A[i], A[j]);

    SWAP (A[l], A[j]);
    RETURN j;
```

INCREMENTA i FINCHÉ NON TROVI UN ELEMENTO MAGGIORE DEL PIVOT

DECREMENTA J FINCHÉ NON TROVI UN ELEMENTO MINORE DEL PIVOT

SE GLI INDICI NON SONO INVERTITI, SCAMBIA A[i] e A[j] PORTANDO UN ELEM. > PIVOT A DX E UN ELEM ≤ PIVOT A SX.

METTI AL CENTRO IL PIVOT

ANALISI DELLA COMPLESSITÀ

LA COMPLESSITÀ DEL QUICK SORT PUÒ VARIARE IN BASE AL PIVOT SCELTO.

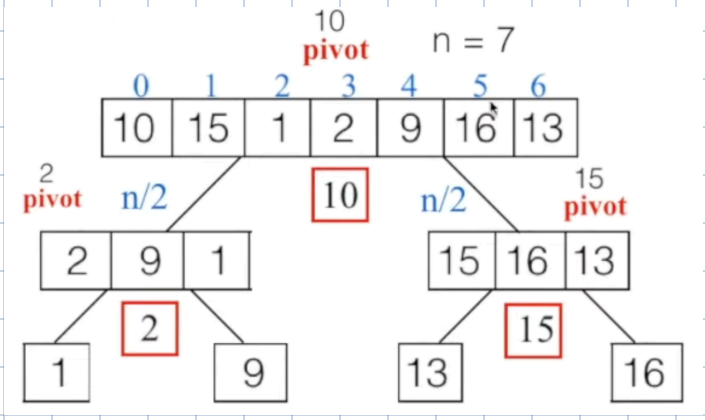
BEST CASE: IL MIGLIOR CASO SI VERIFICA QUANDO IL PIVOT SCELTO DIVIDE L'ARRAY IN DUE PARTI DI DIMENSIONI QUASI UGUALI, CAPITA QUANDO L'ARRAY È GIÀ BEN DISTRIBUITO O SE SI RIESCE A SCEGLIERE UN PIVOT CENTRATO.

SE OGNI PARTIZIONE DIVIDE L'ARRAY IN DUE METÀ QUASI UGUALI, IL QUICK SORT CREERÀ UN ALBERO DI RICORSIONE BILANCIATO CON PROFONDITÀ LOGARITMICA.

IN QUESTO CASO ABBIAMO:

- $O(m)$ CHE RAPPRESENTA IL TEMPO SPESO PER OGNI LIVELLO
- $\log m$ CHE RAPPRESENTA LA PROFONDITÀ DELL'ALBERO

QUINDI: $O(m \cdot \log m)$



WORST CASE: IL CASO PEGGIORE SI VERIFICA QUANDO IL PIVOT SCELTO È IL MASSIMO O IL MINIMO ELEMENTO IN OGNI PARTIZIONE, ACCADE AD ESEMPIO QUANDO L'ARRAY È GIÀ ORDINATO O INVERSAMENTE ORDINATO E SI SCEGLIE COME PIVOT IL PRIMO O L'ULTIMO ELEMENTO.

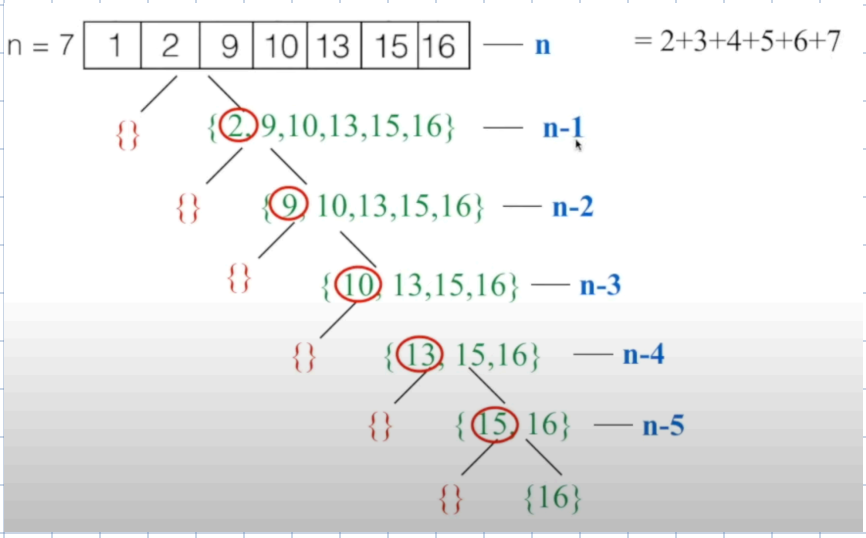
IN QUESTO SCENARIO QUANDO VIENE PARTIZIONATO L'ARRAY QUESTO VIENE PARTIZIONATO IN MANIERA SBILANCIATA PERCHÉ AVREMO DA UNA PARTE UN SOLO ELEMENTO E DALL'ALTRA GLI $m-1$ ELEMENTI

LA PROFONDITÀ DELL' ALBERO DIVENTA COSÌ LINEARE PERCHÉ OGNI CHIAMATA RICORSIVA RIDUCE L'ARRAY DI <u>UN SOLO ELEMENTO ALLA VOLTA</u>.

AVENDO QUINDI:

- $O(m)$ CHE RAPPRESENTA IL TEMPO SPESO PER OGNI LIVELLO
- m CHE RAPPRESENTA LA PROFONDITÀ DELL' ALBERO

QUINDI: $O(m^2)$



QUICK SORT RANDOMIZZATO

IL PIVOT VIENE SCELTO RANDOMICAMENTE COSÌ DA AVERE UNA PROBABILITÀ RANDOMICA, COSÌ PER CERCARE DI EVITARE IL CASO PEGGIORE.

SELECTION SORT

↘ SELEZIONA IL MINORE E LO SCAMBIA CON QUELLO IN POSIZIONE ATTUALE $\Theta(m^2)$

INSERTION SORT

↘ PARTE DAL SECONDO ELEMENTO, PER OGNI ELEMENTO CONTROLLA SE TRA GLI ELEMENTI PRECEDENTI C'È QUALCHE ELEMENTO MAGGIORE, SE SÌ SHIFTA A DESTRA PER OGNI ELEMENTO MAGGIORE, INSERISCE L'ELEMENTO SU CUI ERA NELL' ULTIMO "BUCO" A SINISTRA LASCIATO DALL'ULTIMO SHIFT DESTRO. $O(m^2)$

BUBBLE SORT

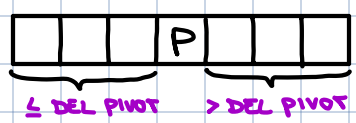
↘ "BOLLA CHE SALE VERSO L'ALTO", CON QUESTA TECNICA PER OGNI ELEMENTO PORTA IL PIÙ GRANDE ELEMENTO VERSO L'ALTO OVVERO A DESTRA.
AD OGNI ITERAZIONE L'ELEMENTO PIÙ GRANDE PUÒ SALIRE DI UNA POSIZIONE IN MENO RISPETTO IL CICLO PRECEDENTE. $O(m^2)$

MERGE SORT

↘ L'ARRAY VIENE SUDDIVISO RICORSIVAMENTE FINO AD OTTENERE SOTTO-SEQUENZE DI UN SOLO ELEMENTO (GIÀ ORDINATO).
LE SOTTOSEQUENZE VENGONO RIUNITE ORDINATAMENTE IN MODO CRESCENTE UTILIZZANDO LA PROCEDURA DI MERGE.
IL PROCESSO CONTINUA FINO A RICOSTRUIRE L'ARRAY ORDINATO. $O(m \log m)$

QUICK SORT

↘ SCEGLIE UN ELEMENTO CHIAMATO PIVOT, AD ESEMPIO IL PRIMO ELEMENTO.
UTILIZZA DUE CONTATORI i & J, CON i CERCA DA SX A DX NUMERI MAGGIORI DEL PIVOT FERMANDOSI SULLA POSIZIONE QUANDO NE TROVA UNO, CON J CERCA DA DX A SX NUMERI MINORI DEL PIVOT FERMANDOSI SULLA POSIZIONE QUANDO NE TROVA UNO, SE $i < J$ SIGNIFICA CHE ABBIAMO TROVATO UN ELEM MAGGIORE DEL PIVOT CHE DOVREBBE STARE NELLA PARTE DI DX, POI UN ELEM MINORE DEL PIVOT CHE DOVREBBE STARE NELLA PARTE SINISTRA, SCAMBIAMO QUINDI I DUE ELEMENTI.
SCAMBIA IL PIVOT CON A[J], IN A[J] CI SARÀ L'ULTIMO ELEMENTO MINORE UGUALE DEL PIVOT, DOPO A[J] CI SONO GLI ELEMENTI MAGGIORI.
PONTO QUINDI IL PIVOT AL CENTRO AVENDO UNA SITUAZIONE DEL TIPO:

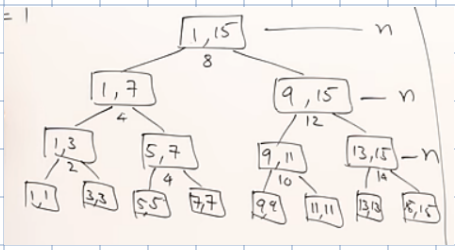


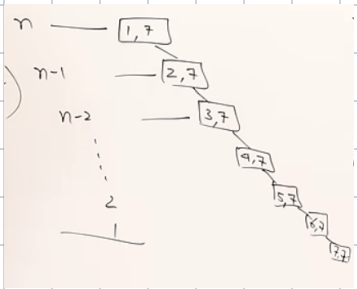
RESTITUISCO J (M) CHE SAREBBE IL PUNTO MEDIO, CHIAMO QUICKSORT RICORSIVAMENTE SULLA PARTE SINISTRA $i \ldots M-1$ E SULLA PARTE DESTRA $M+1 \ldots h$

$O(m \cdot \log m)$ BEST CASE $\quad O(m^2)$ WORST CASE

IL PIVOT DIVIDE L'ARRAY QUASI EQUAMENTE

IL PIVOT È IL MAX O MIN DIVIDENDO L'ARRAY IN MODO SBILANCIATO





- $\log m$ LIVELLI
- m TEMPO SPESO NEL PARTITION
- TOTALE $O(m \log m)$

- m LIVELLI
- m TEMPO SPESO PER PARTITION
- TOTALE $O(m^2)$