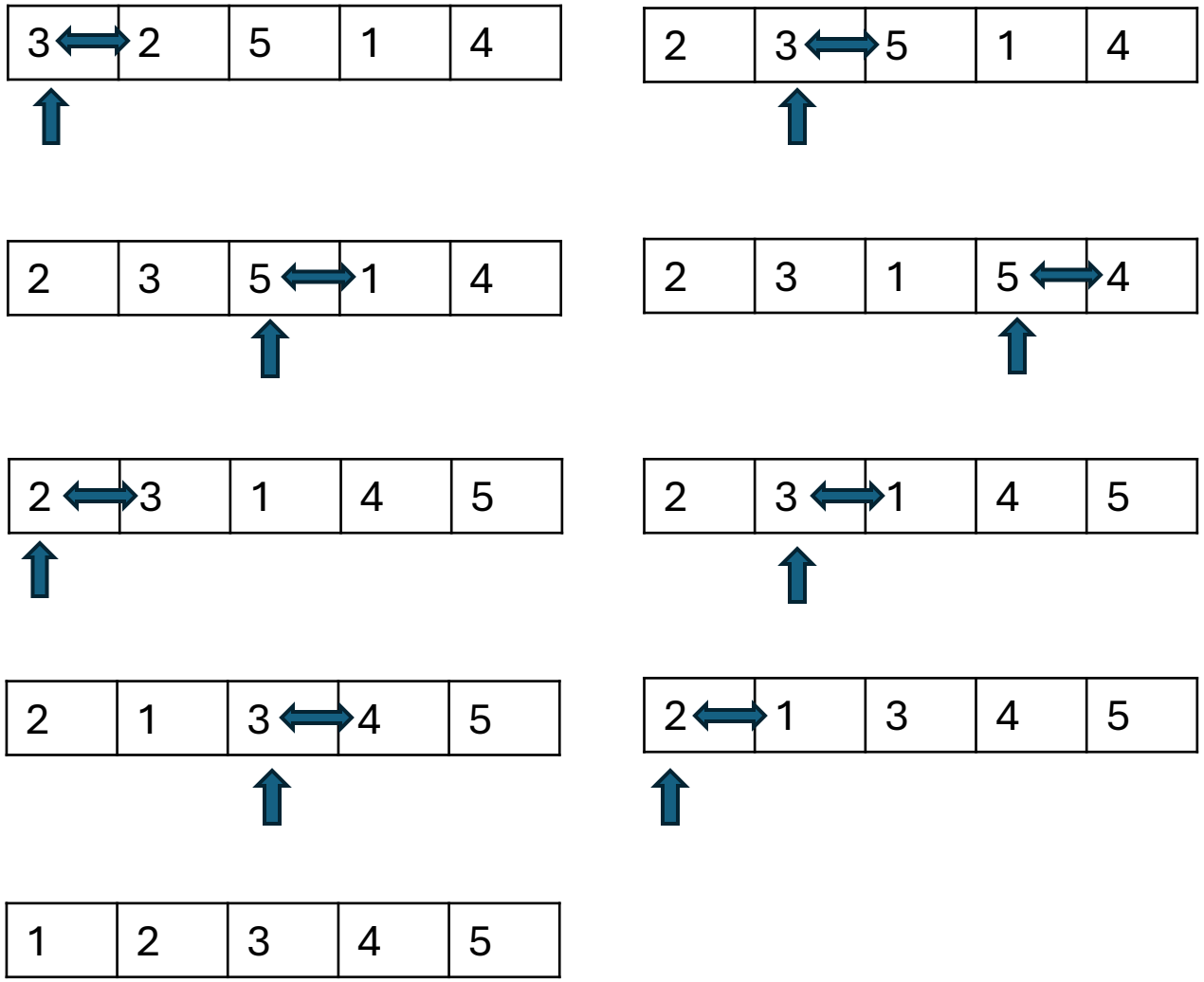
PROGRAMMAZIONE INFORMATICA

ALGORITMI DI ORDINAMENTO

BUBBLE SORT



COMPLESSITA’ COMPUTAZIONALE

La **complessità computazionale** è un'area dell'informatica che studia le risorse computazionali (come tempo e memoria) necessarie per eseguire un algoritmo, in funzione della dimensione dell'input. L'obiettivo principale è valutare l'efficienza di un algoritmo nel risolvere un problema, tenendo conto del **tempo di esecuzione** e dello **spazio di memoria** richiesti.

L'analisi della complessità computazionale permette di:

- **Valutare l'efficienza** di un algoritmo.
- **Confrontare algoritmi diversi** per risolvere lo stesso problema.
- **Prevedere la scalabilità**: Come le prestazioni dell'algoritmo cambiano con l'aumentare dell'input.
- **Identificare punti di miglioramento**: Migliorare algoritmi inefficienti in termini di tempo o spazio.

COMPLESSITA' COMPUTAZIONALE

Il **comportamento asintotico** di un algoritmo descrive come la complessità (in termini di tempo o spazio) cresce all'aumentare della dimensione dell'input, n, ignorando i dettagli minori come costanti o fattori non dominanti.

La **notazione di Landau**, meglio conosciuta come **notazione Big-O e sue varianti**, è utilizzata per descrivere il comportamento asintotico di funzioni e algoritmi.

NOTAZIONE DI LANDAU

- **Notazione Big-O:** fornisce un limite superiore asintotico del tempo di esecuzione di un algoritmo.

$$Of(n)$$

- **Notazione Ω**: fornisce un limite inferiore asintotico, indicando la crescita minima che un algoritmo impiega nel caso migliore.

$$\Omega f(n)$$

CLASSI DI COMPLESSITA’

1	costante
$\log(n)$	logaritmica
n	lineare
$n\log(n)$	log-lineare o pseudolineare
n^2	quadratica
n^3	cubica
n^k	polinomiale

VALUTAZIONE DELLA COMPLESSITA’

- **Identificazione delle Operazioni Dominanti**: identificare le parti del codice che contribuiscono maggiormente al tempo di esecuzione. Le operazioni come cicli annidati e chiamate ricorsive sono spesso le più significative.

- **Conteggio delle Operazioni**: si conta quante volte vengono eseguite queste operazioni in funzione di n. Per esempio, se abbiamo un ciclo che scorre un array di dimensione n, contribuisce n operazioni.

- **Determinazione della complessità**: trovare la funzione di complessità e associarla ad una delle classi.

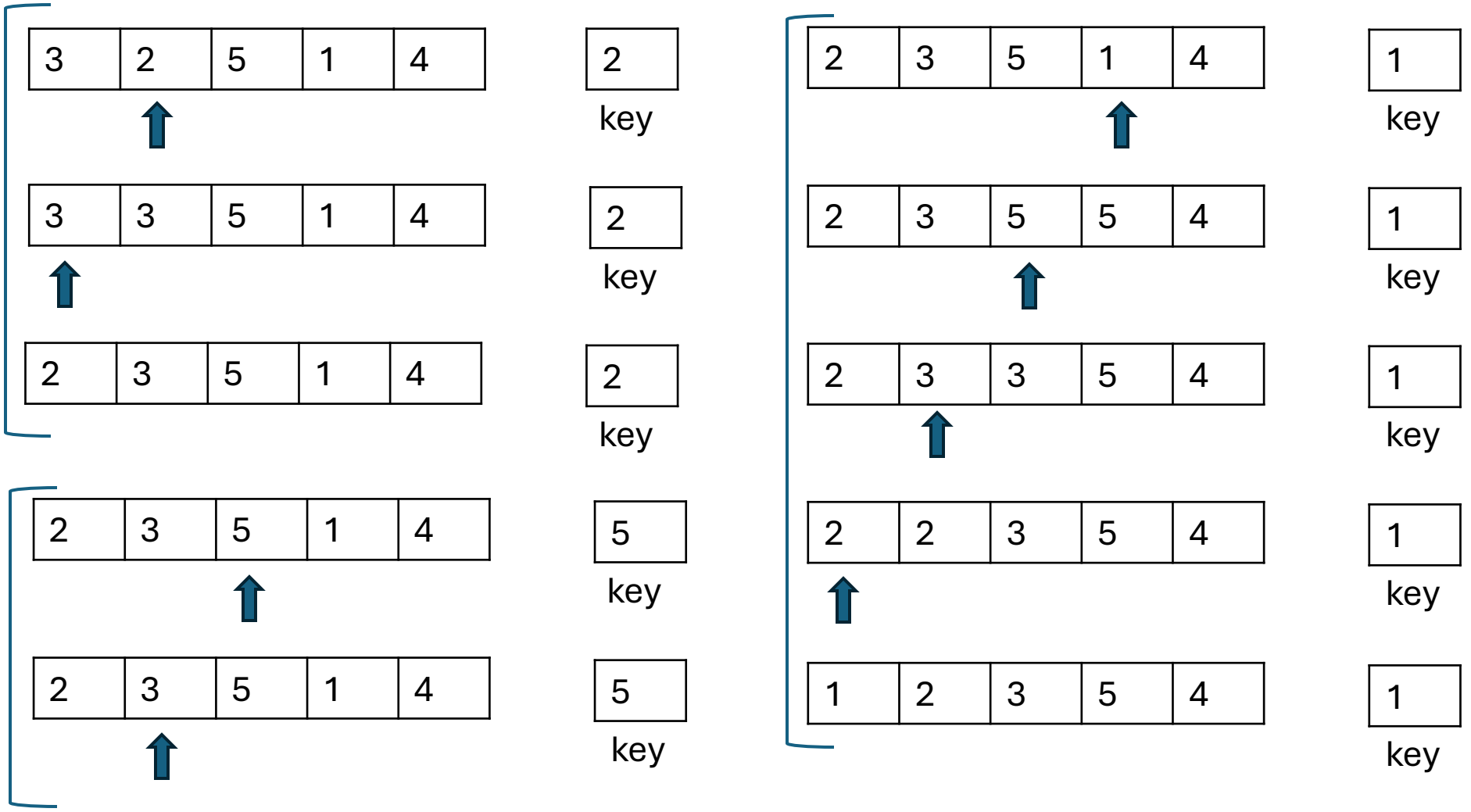
BUBBLE SORT

```
for(i in seq(0,n)) {
        scambio = False
        for(j in  seq(0,(n-(i+1)))) {
                if(v[j] > v[j+1]) {
                        x = v[j+1]
                        v[j+1] = v[j]
                        v[j] = x
                        scambio = True
                }
        }
        if(!scambio)  break
}
```

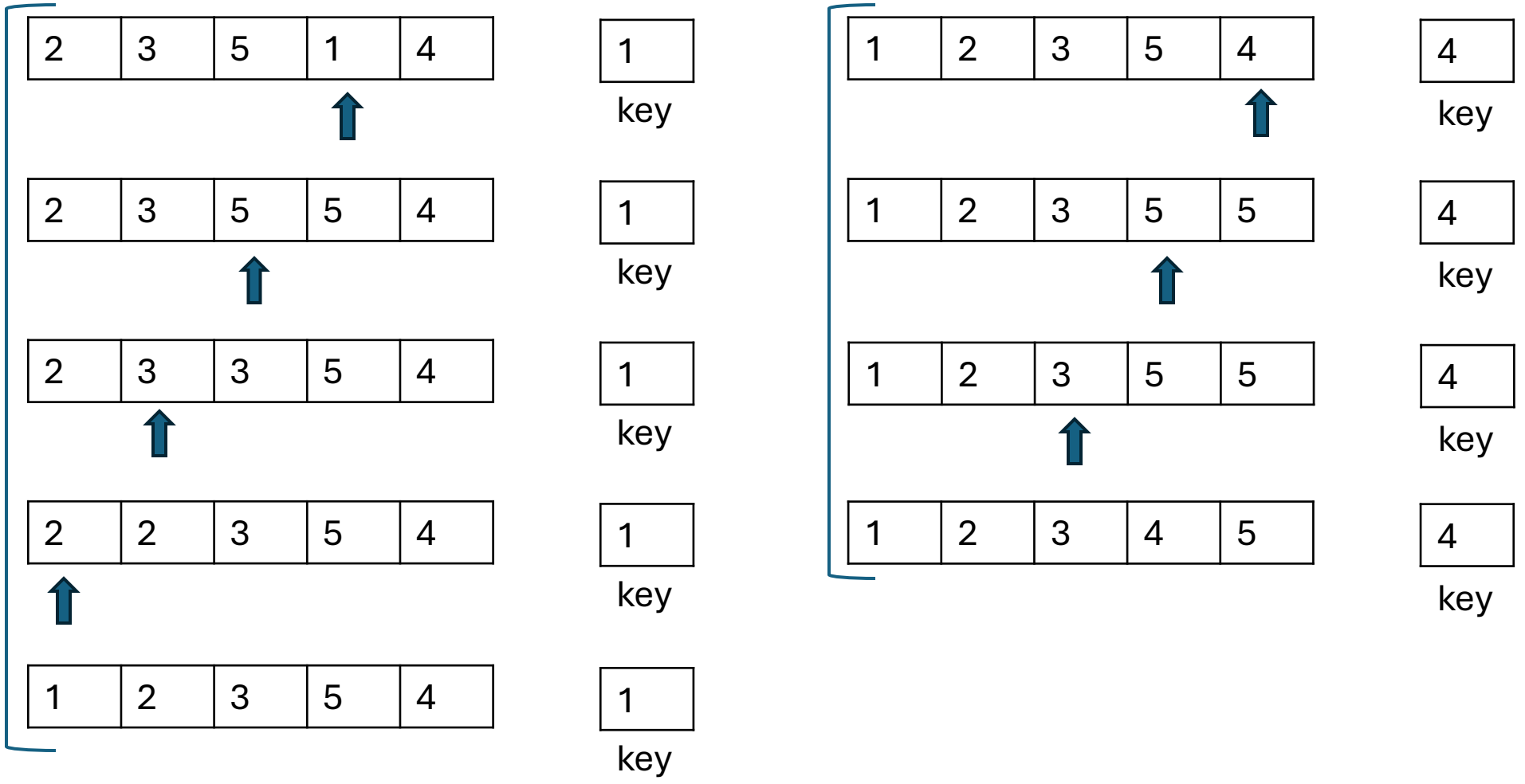
$$On^2$$

$$\Omega n$$

INSERTION SORT



INSERTION SORT



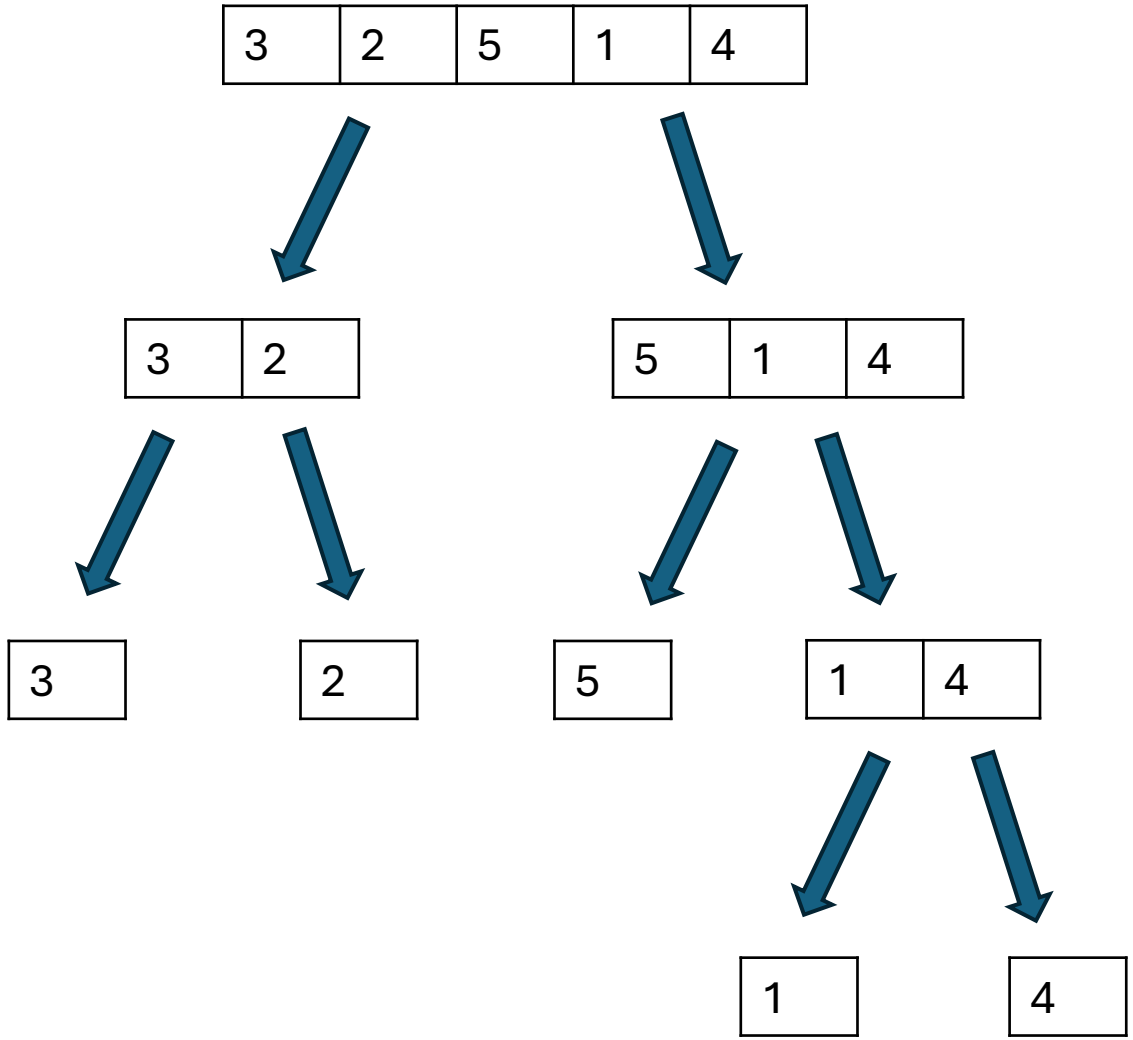
INSERTION SORT

```
for(i in seq(1,n-1)) {
        key = v[i]
        j = i – 1
        while(j >= 0 and v[j] > key) {
                v[j+1] = v[j]
                j = j - 1
        }
        v[j+1] = key
}
```

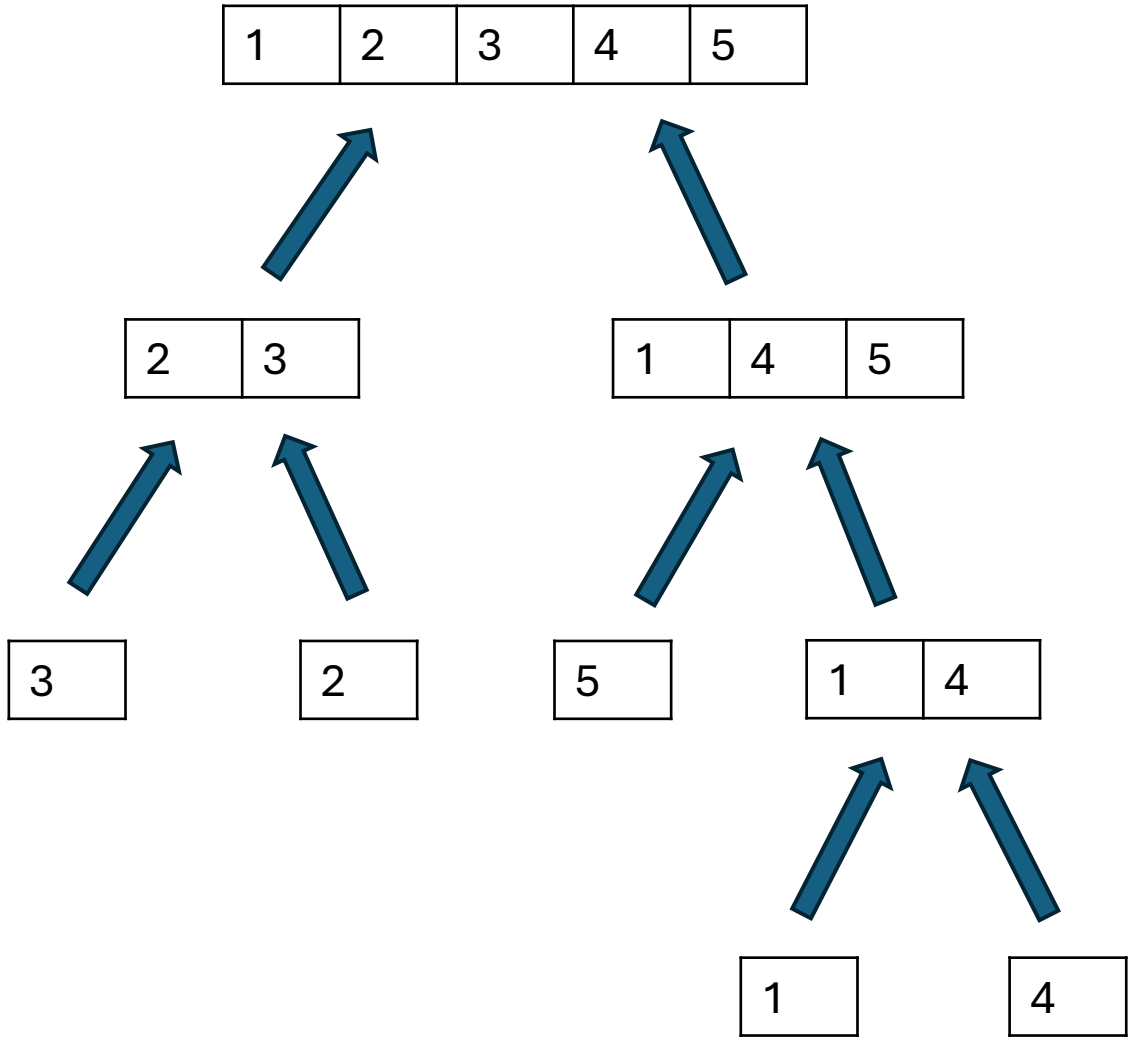
$O n^2$

Ωn

MERGE SORT



MERGE SORT

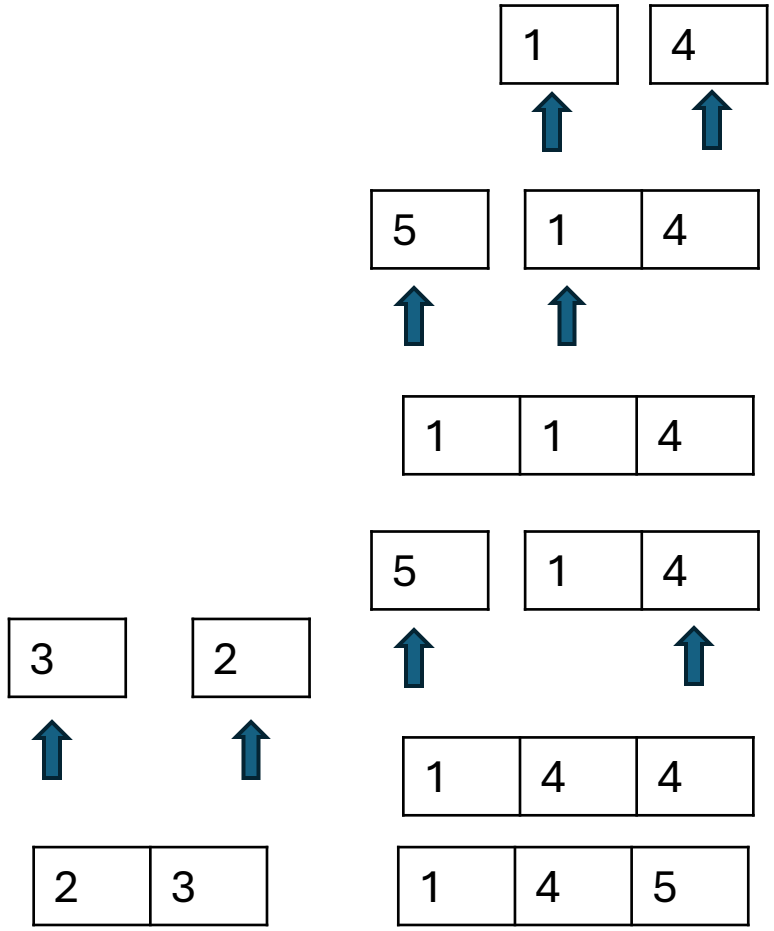


MERGE SORT

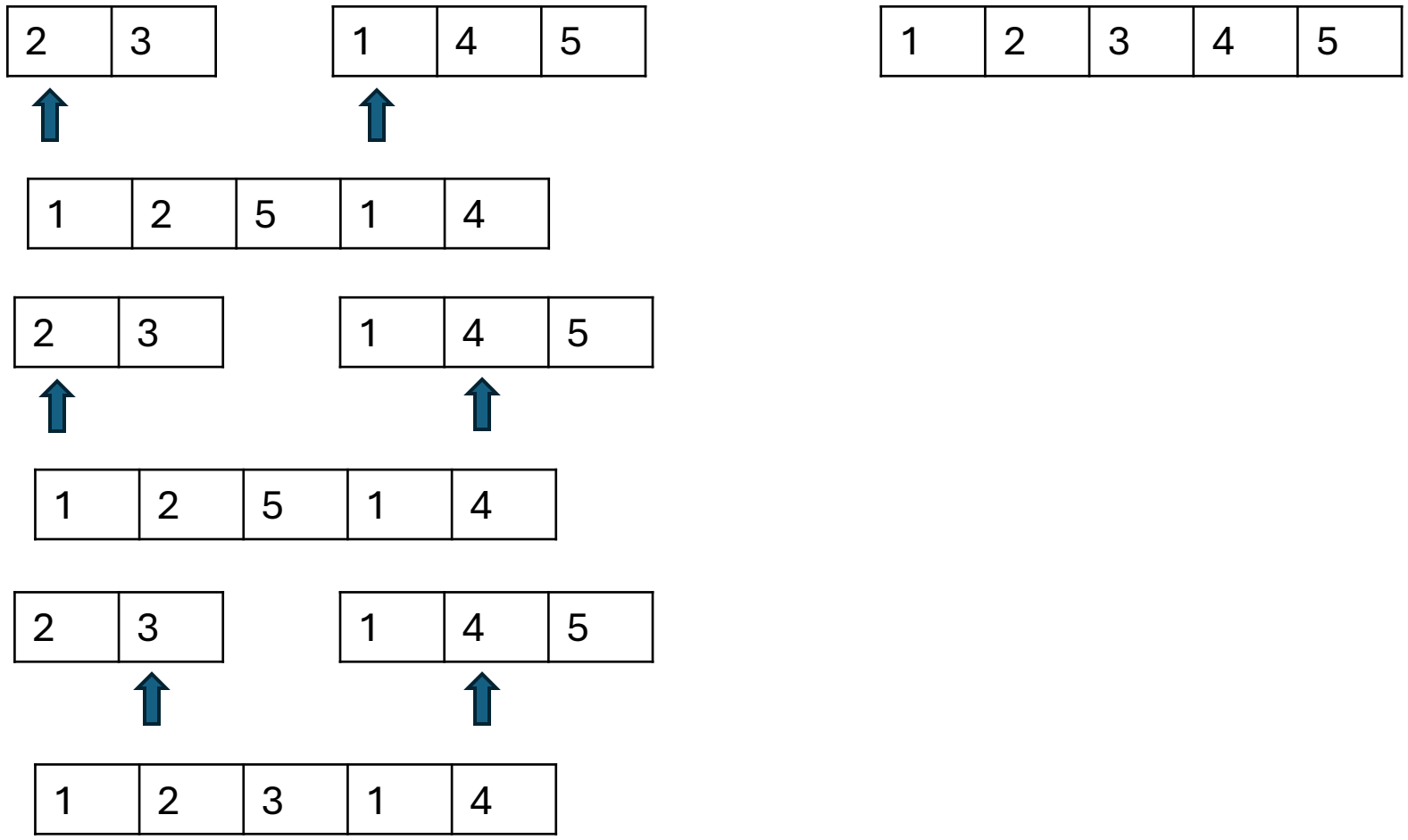
```
def mergesort(v) {
        n = len(v)
        if(n <= 1) return(v)
        mid = n / 2
        left = v[0:(mid-1)]
        right = v[mid:(n-1)]
        left = mergesort(left)
        right = mergesort(right)
        Ordinamento left-right...
        return(v_ordinato)
}
```

FUNZIONE RICORSIVA

MERGE SORT



MERGE SORT



MERGE SORT

```
int i = 0, j = 0, k = 0
while(i < len(left) and j < len(right)) {
        if(left[i] < right[j]) {
                v[k] = left[i]
                i = i+1
        } else {
                v[k] = right[j]
                j = j+1
        }
        k = k+1
}
```

MERGE SORT

```
int i = 0, j = 0, k = 0
while(i < len(left) and j < len(right)) {
        if(left[i] < right[j]) {
                v[k] = left[i]
                i = i+1
        } else {
                v[k] = right[j]
                j = j+1
        }
        k = k+1
}
```

```
while(i < len(left)) {
        v[k] = left[i]
        i = i+1
        k = k+1
}

while(j < len(right)) {
        v[k] = right[j]
        j = j+1
        k = k+1
}
```

MERGE SORT

```
def mergesort(v) {
        n = len(v)
        if(n <= 1) return(v)
        mid = n / 2
        left = v[0:(mid-1)]
        right = v[mid:(n-1)]
        left = mergesort(left)
        right = mergesort(right)
```

```
int i = 0, j = 0, k = 0
while(i < len(left) and j < len(right)) {
        if(left[i] < right[j]) {
                v[k] = left[i]
                i = i+1
        } else {
                v[k] = right[j]
                j = j+1
        }
        k = k+1
}
```

```
while(i < len(left)) {
        v[k] = left[i]
        i = i+1
        k = k+1
}

while(j < len(right)) {
        v[k] = left[j]
        j = j+1
        k = k+1
}

return(v) }
```

MERGE SORT

def mergesort(v) {

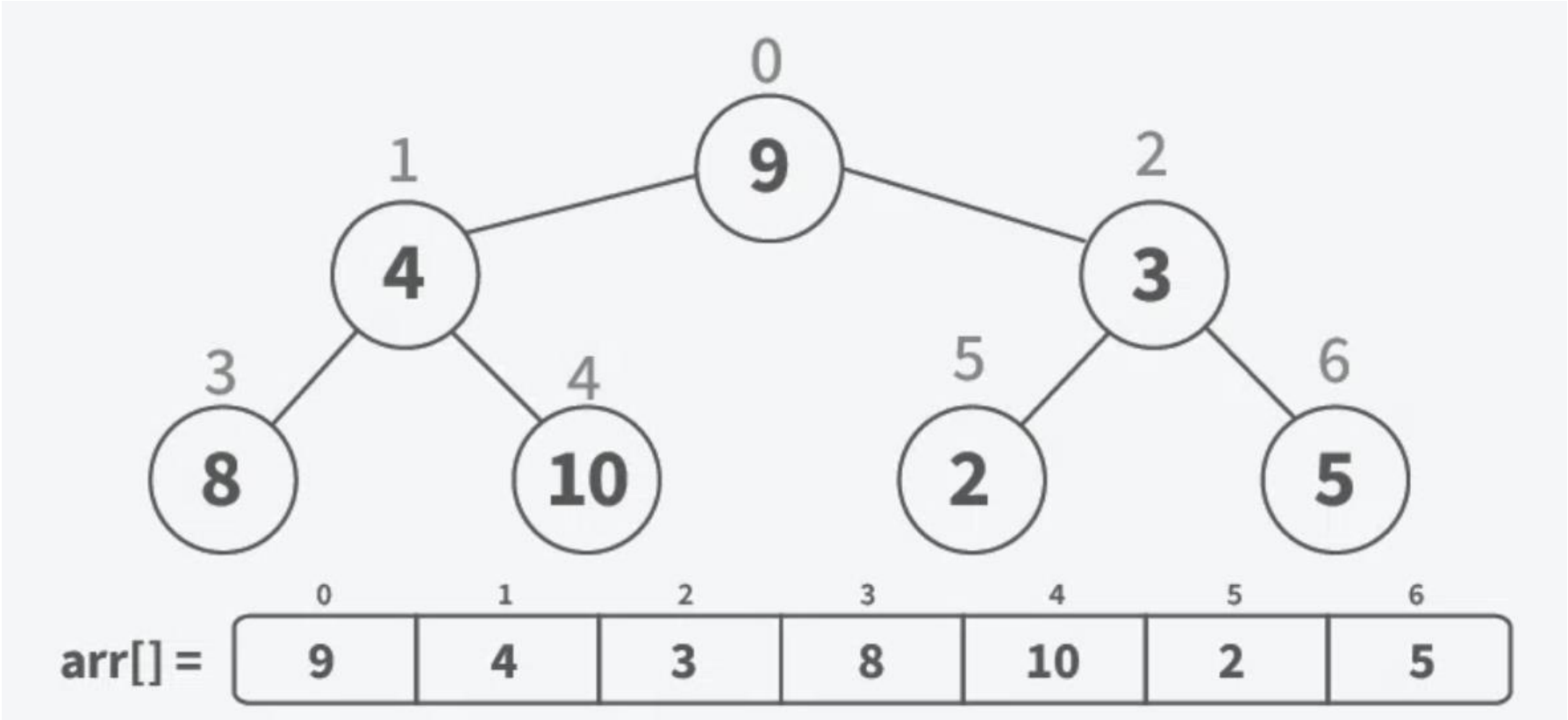
PARTE RICORSIVA ⟶ $\log n$

CICLI WHILE ⟶ n

}

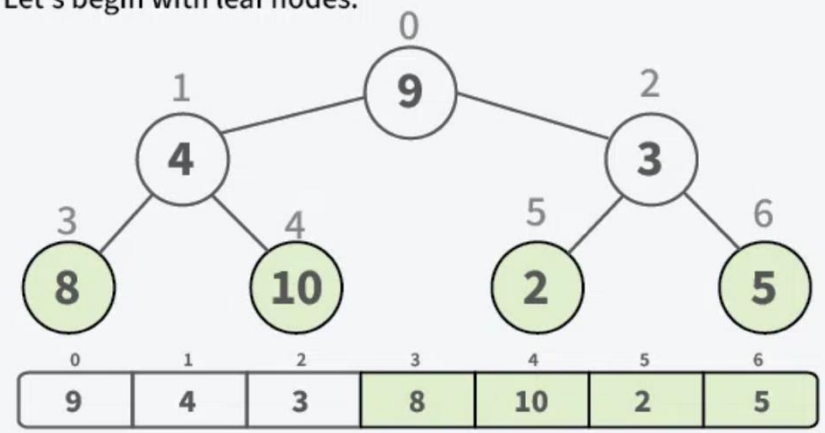
$$On \log n$$
$$\Omega n \log n$$

HEAP SORT

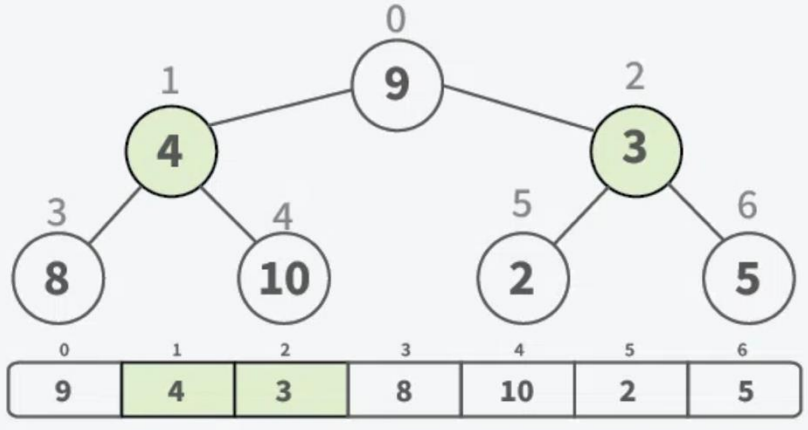


HEAP SORT

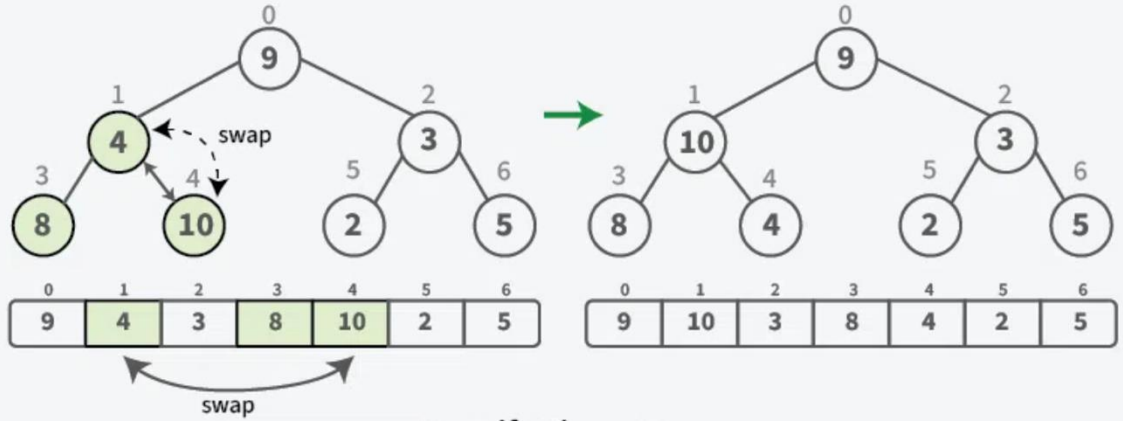
01 Step | Compare each node with its children, ensuring parent nodes are larger. This causes smaller nodes to bubble down and larger nodes to rise to the top. Let's begin with leaf nodes.



02 Step | Let's look in the next upper level (node 4 and 3)

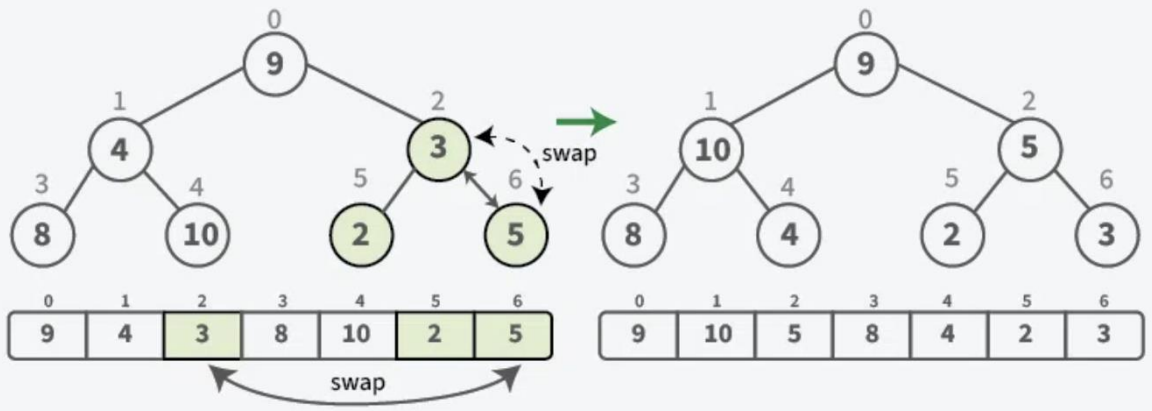


03 Step | Node 4 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.

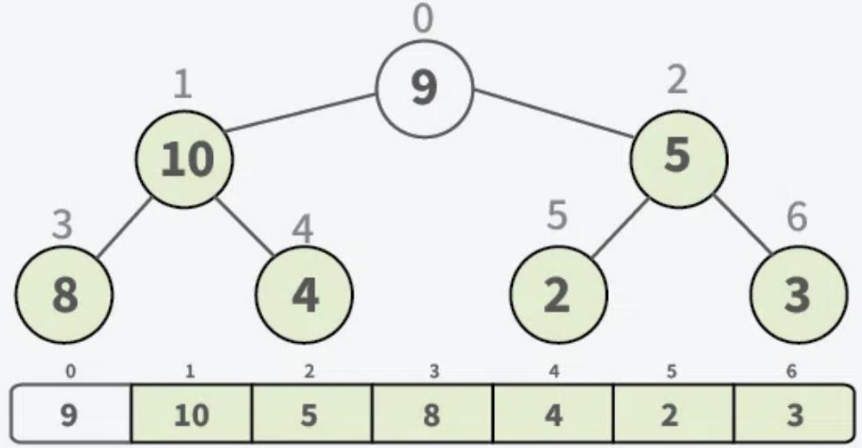


HEAP SORT

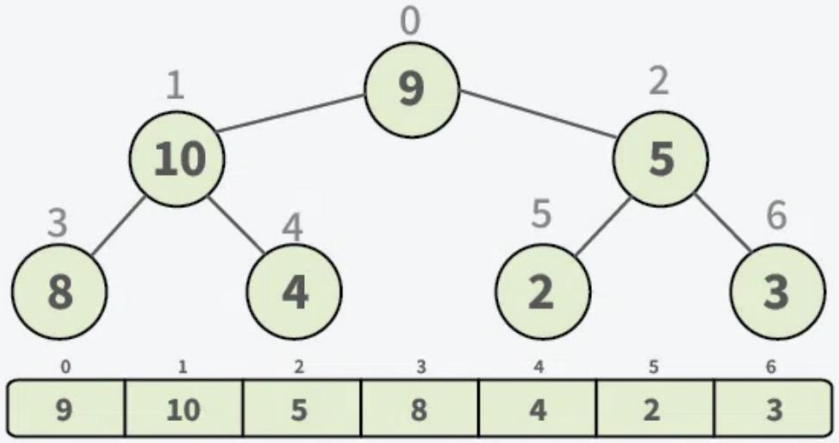
04 Step — Check for the next node (3) in current level. Since, it has a larger child, so swap it to ensure the parent has a larger value.



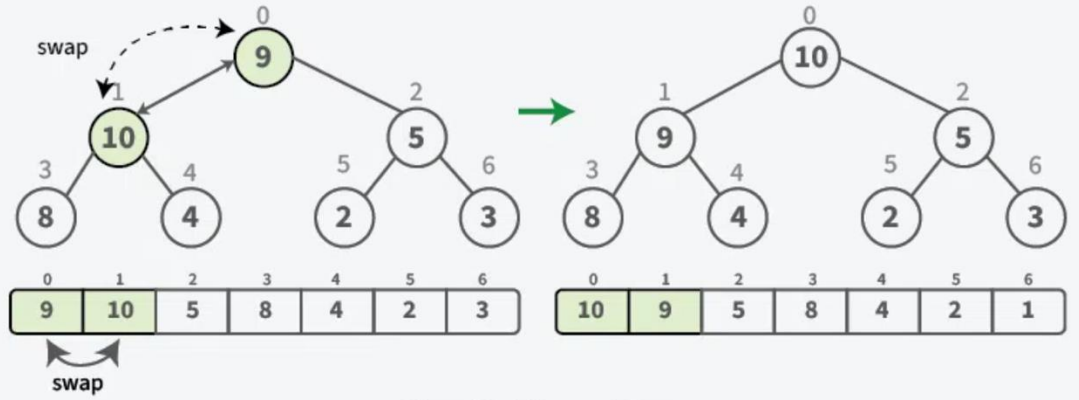
05 Step — After the completion of current level, we have now two smaller valid max heap



06 Step — Let's move to the next upper level. Here we've node 9 at the root.

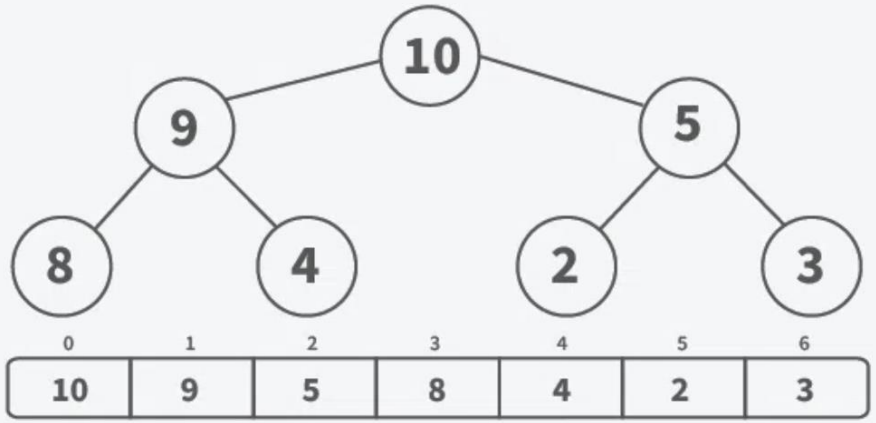


07 Step — Node 9 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.

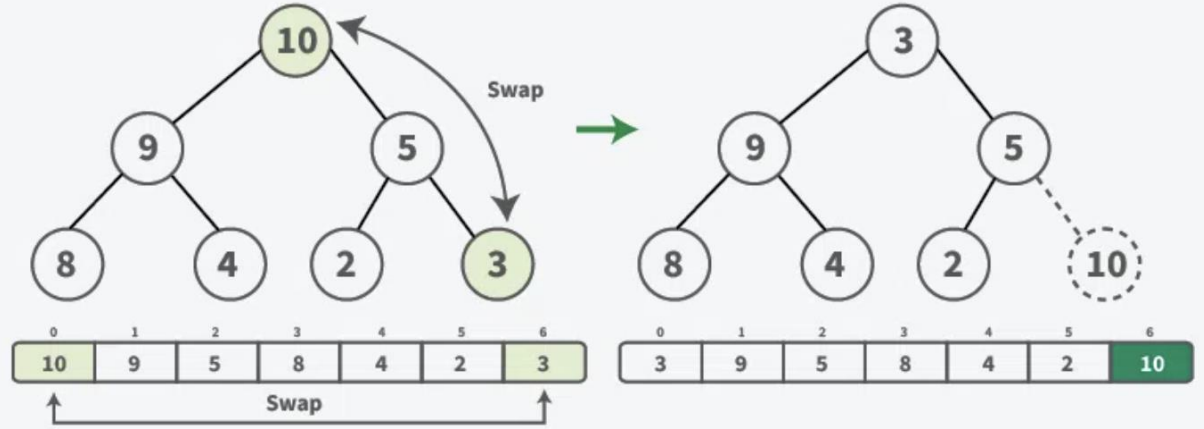


HEAP SORT

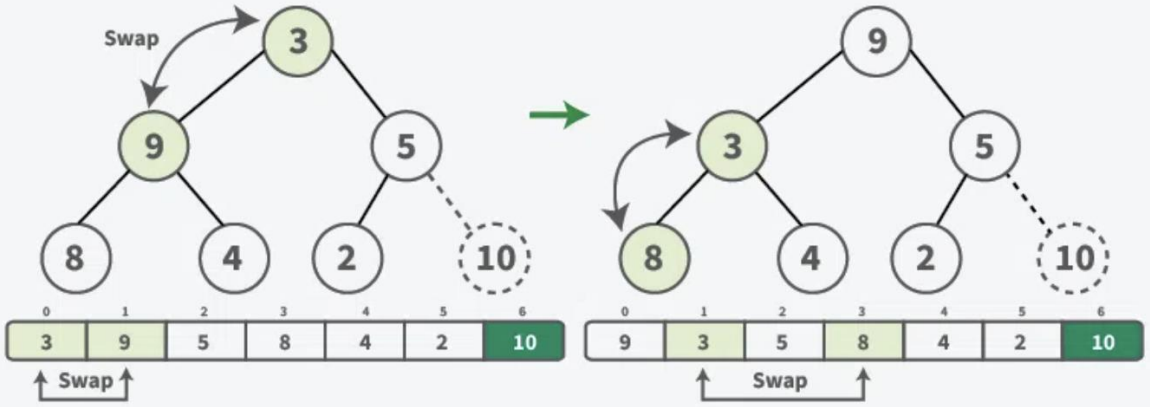
01 Step | Let's assume we have transformed the given array to follow the max heap property. Here's how our array would look in max heap form.



02 Step | Swap the maximum element (10) with the last element (3) in the unsorted array. Decrease the size of the heap by one (ignore the last element, as it is now sorted).

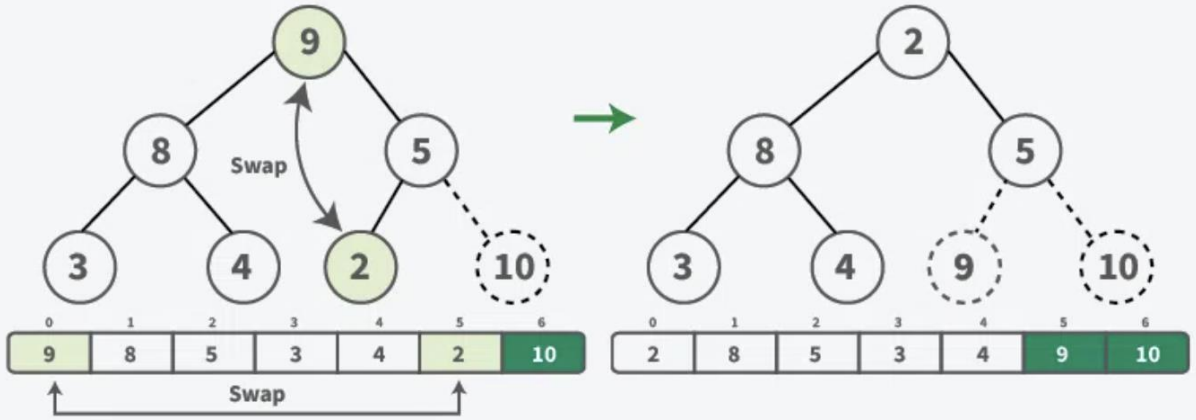


03 Step | Now, root violate the max-heap property, So, heapify it.
Swap node 3 with its largest child (9).
Still node 3 have larger children, so swap it with largest one (node 8).

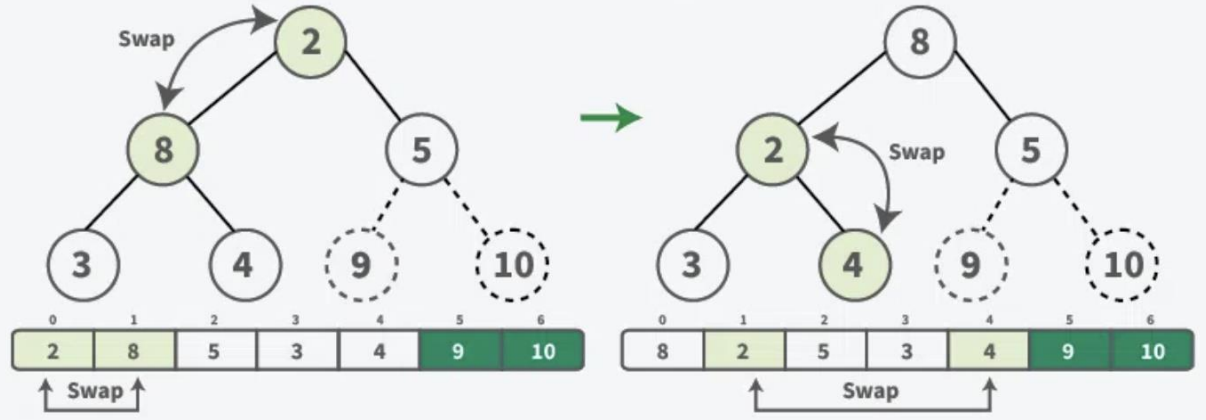


HEAP SORT

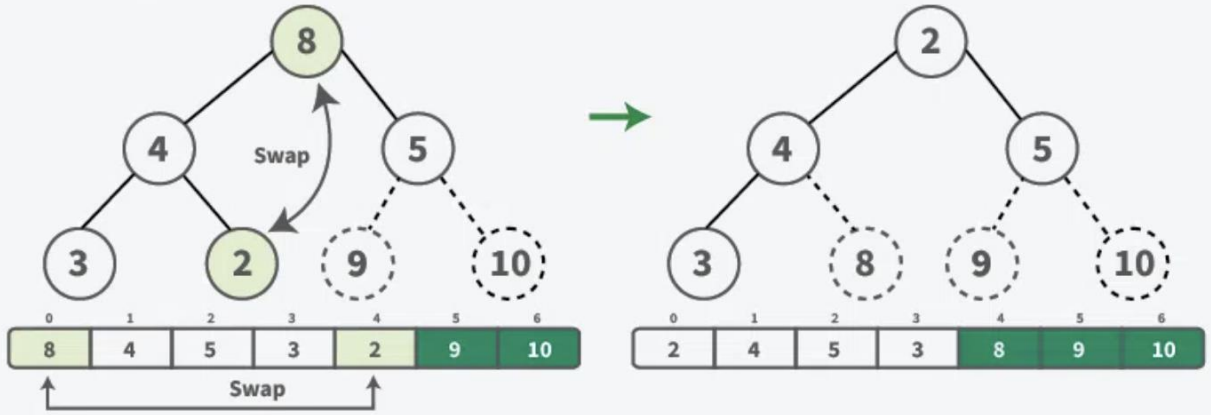
04 Step Now, we have a max heap. Swap the maximum element (9) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the second last element as it's now sorted).



05 Step Now, root violate the max-heap property, So, heapify it.
Swap node 2 with its largest child (8).
Still node 2 have larger children, so swap it with largest one (node 4)



06 Step Now, we have a max heap. Swap the maximum element (8) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the third last element as it's now sorted).



HEAP SORT

```
def heapify(arr, n, i) {
        par = i
        l = 2*i + 1
        r = l + 1
        if(l < n and arr[l] > arr[par]) par = l
        if(r < n and arr[r] > arr[par]) par = r
        if(par != i) {
                x = arr[i]
                arr[i] = arr[par]
                arr[par] = x
                heapify(arr, n, par)
        }
        return(arr)
}
```

HEAP SORT

```
int i = n/2 - 1
while(i >=0) {
        heapify(arr, n, i)
        i = i-1
}
i = n-1
while(i >0) {
        x = arr[i]
        arr[i] = arr[0]
        arr[0] = x
        heapify(arr, n, i)
        i = i-1
        n = n - 1
}
```

HEAP SORT

HEAPIFY → $\log n$

RICOSTRUZIONE → n

$O n \log n$

Ωn