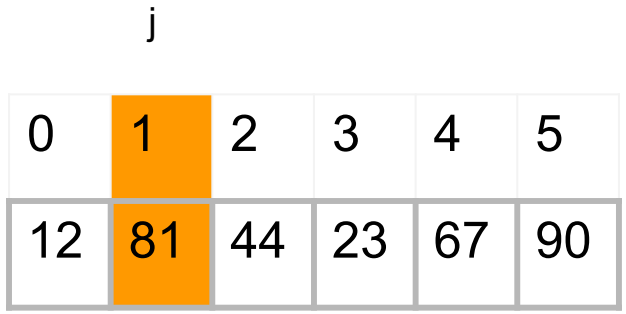
Algoritmos de Ordenação

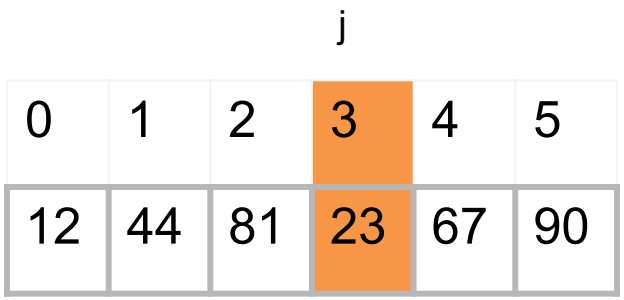
Problema Computacional: Ordenação

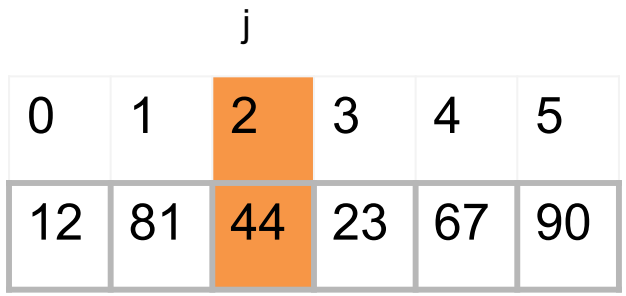
- Problema computacional que surge em diversas situações.
- Definição:
  - ✓ Input: Uma sequência de n números:$\{a_1\ a_2, a_3, \ldots., a_n\}$
  - ✓ Output: Uma reordenação da sequência em $\{a'_1, a'_2, a'_3 \ldots a'_n\}$ tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

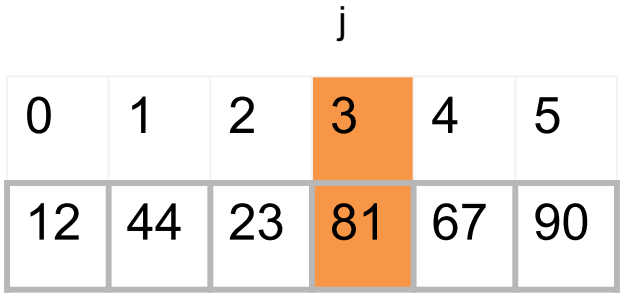
Bubble Sort

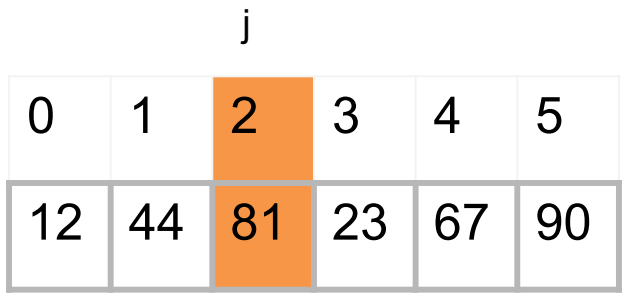
```
def bubble_sort(L):
    swap = False
    while not swap:
        print('bubble sort: ' + str(L))
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

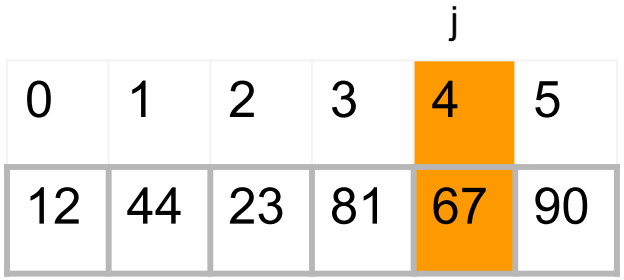


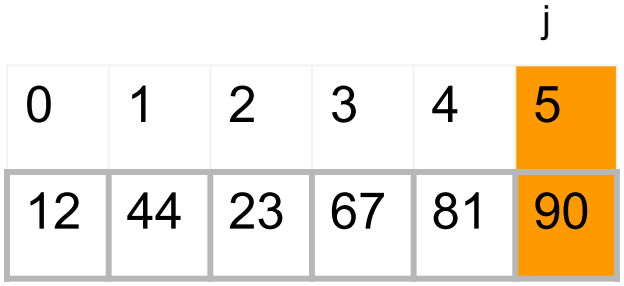






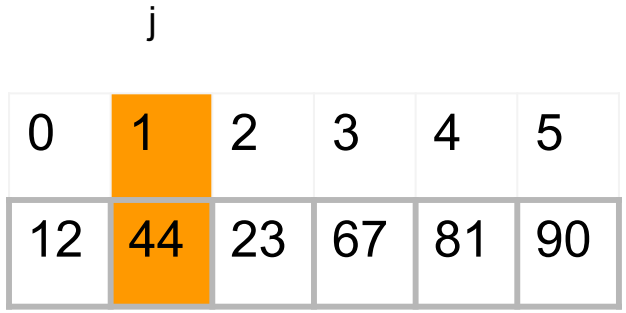


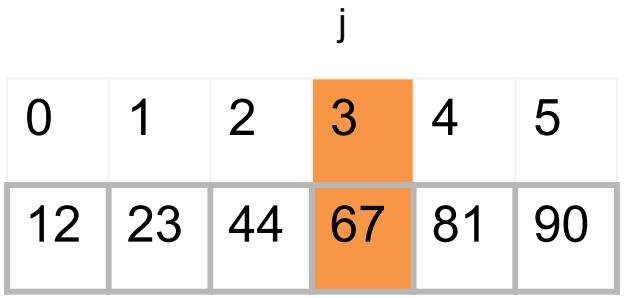


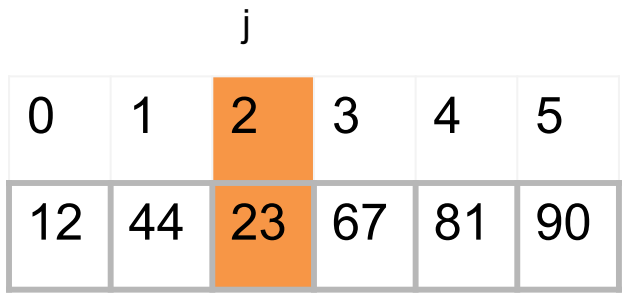


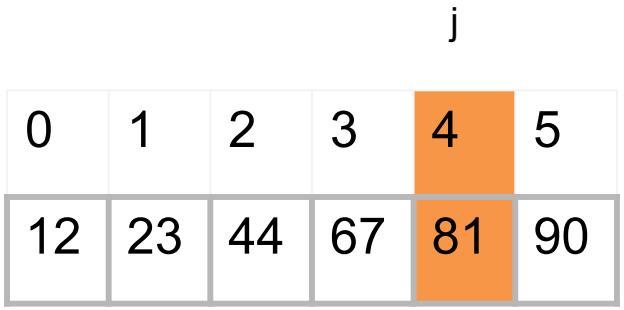
Bubble Sort

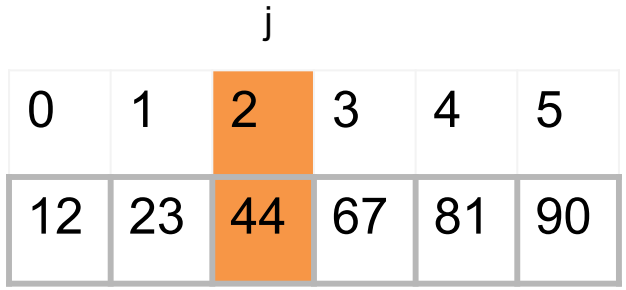
```
def bubble_sort(L):
    swap = False
    while not swap:
        print('bubble sort: ' + str(L))
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

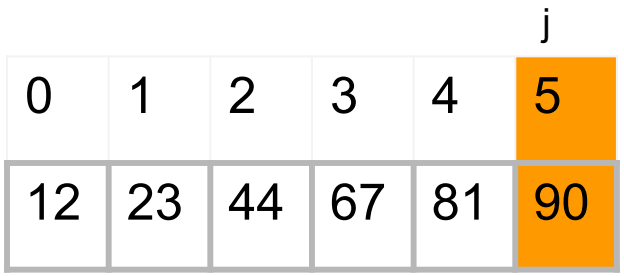


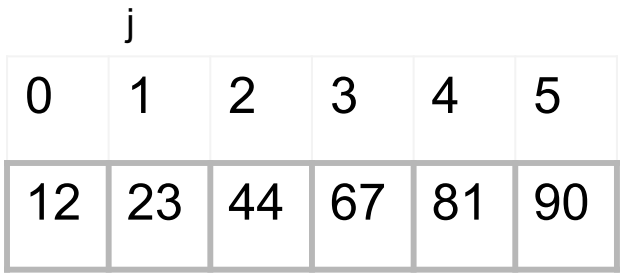












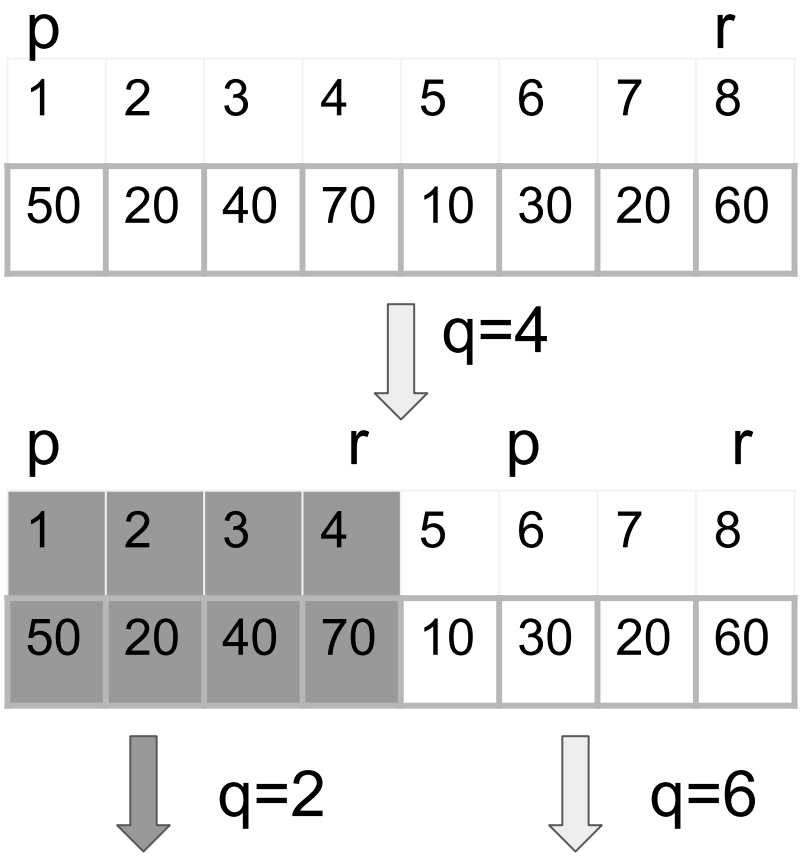
Bubble Sort

```
def bubble_sort(L):
    swap = False
    while not swap:                          O(n)
        print('bubble sort: ' + str(L))
        swap = True
        for j in range(1, len(L)):           O(n)
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

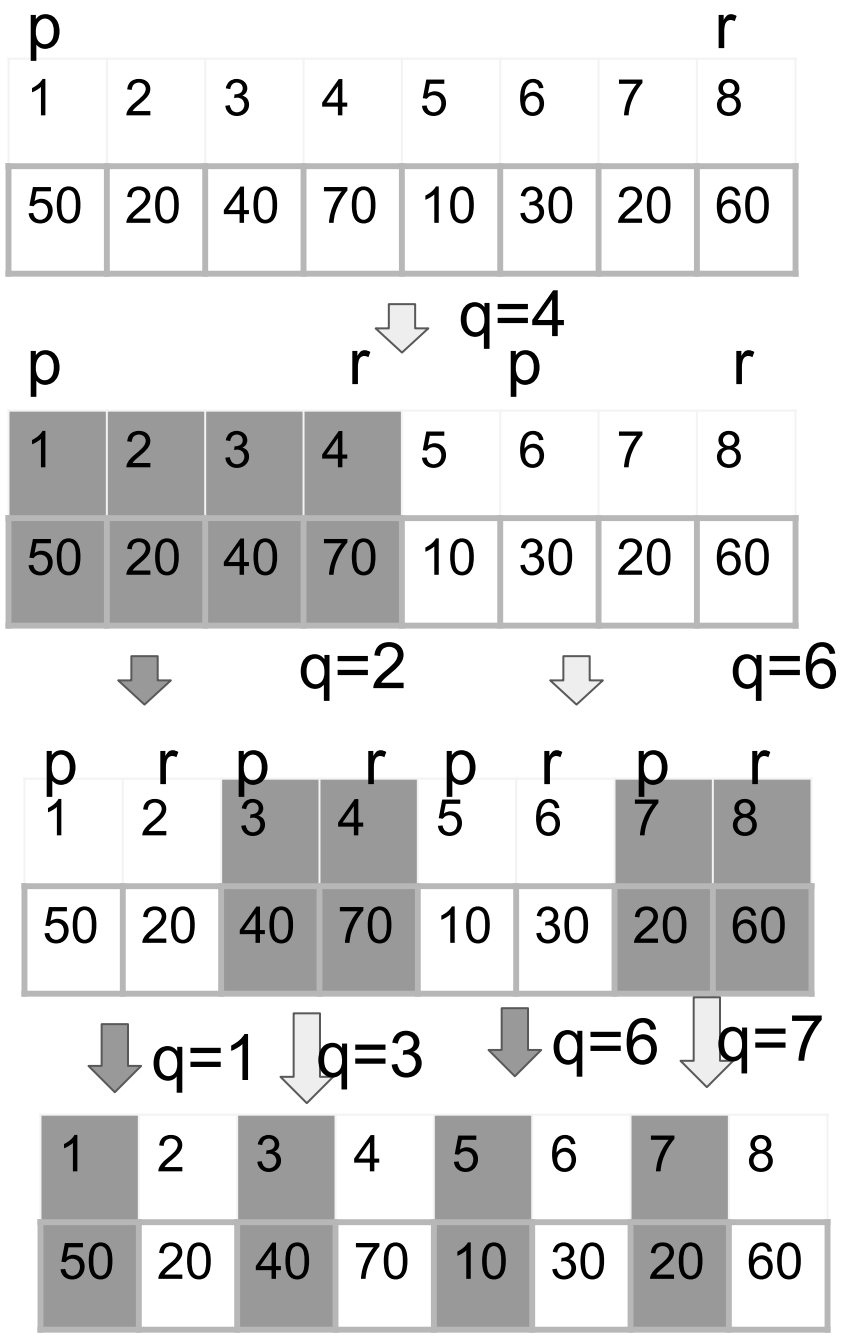
$O(n^2)$ no pior caso para n = len(L)

Merge sort

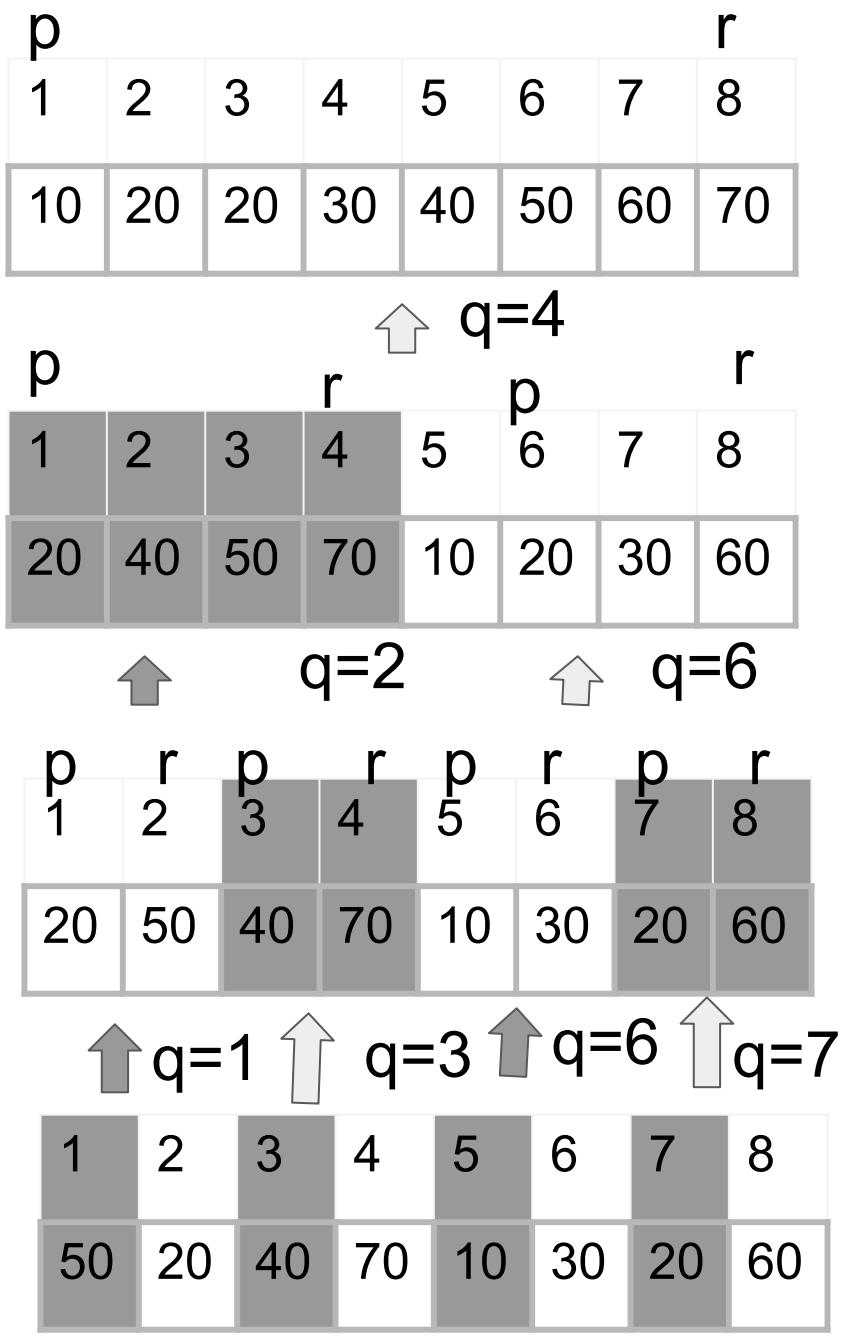
```
def merge_sort(L):
    print('merge sort: ' + str(L))
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```



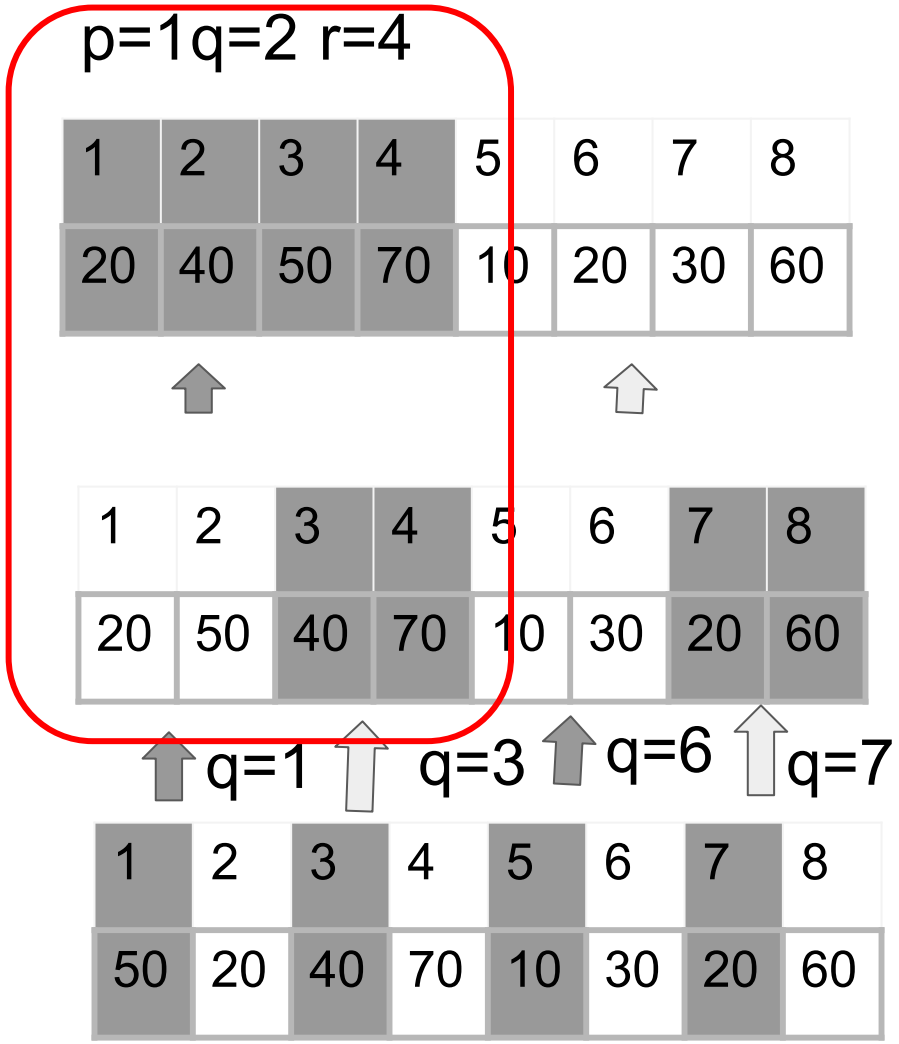
```
def merge_sort(L):
    print('merge sort: ' + str(L))
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```



```
def merge_sort(L):
    print('merge sort: ' + str(L))
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```



```
def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    print('merge: ' + str(left) + '&' + str(right) + ' to ' +str(result))
    return result
```



Analisando Merge sort

- T (n) : tempo no pior caso (n = r − p + 1).

```
def merge_sort(L):
    print('merge sort: ' + str(L))
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

D(n): Tempo para dividir o problema

C(n): Tempo para combinar o problema

- Relação de recorrência:

1.T (1) = $\Theta(1)$ n=1.

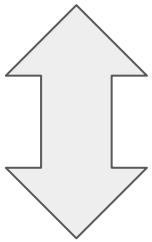
2.T (n) = 2.T(n/2)+D(n)+C(n) n>1.

Analisando Merge sort

- Relação de recorrência:

1.T (n) = $\Theta(1)$ n=1.

2.T (n) = 2.T(n/2)+D(n) + C(n) n>1.



$$T(n) = \begin{cases} \Theta(1) & n=1 \\ 2T(n/2) + \Theta(n) & n>1 \end{cases}$$

T(n) = O(nlgn)

Analisando o Heapsort(A)

```
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
```

O(n)
Θ(n)
Θ(n)
Θ(n)
nO(lgn)

Pior caso: O(nlgn)