Análise de Algoritmos de Ordenação

Giovani Candido - RA: 191021601
Luis Henrique Morelli - RA: 181027097

Importação de módulos

Vamos importar alguns módulos necessários.

In [1]:
```
import random
import heapq
```

Funções Utilitárias

Precisaremos de uma função para gerar arrays para nossos experimentos.

A função generate_array é a responsável por criar um array com valores de 0 a upper_bound-1. Ademais, a flag sorted_arr indica se o array deve ser ordenado ou ter uma distribuição aleatória de valores.

In [2]:
```
def generate_array(sorted_arr, upper_bound):
    arr = list(range(0, upper_bound))

    if not sorted_arr:
        random.shuffle(arr)

    return arr
```

Também vamos precisar de uma função para transformar os nossos arrays em strings.

In [3]:
```
def truncate_array_str(size, arr):
    if size <= 10:
        return str(arr)

    first_slice = [str(x) for x in arr[0:5]]
    last_slice = [str(x) for x in arr[-5:]]

    new_arr = first_slice + ['...'] + last_slice

    new_arr_str = '['

    for c in new_arr[:-1]:
        new_arr_str += c + ', '

    new_arr_str += new_arr[-1] + ']'

    return new_arr_str
```

Implementação dos Algoritmos

Aqui, implementamos todos os algoritmos de ordenação que iremos utilizar nos experimentos.

Bubble sort

In [4]:
```
def bubble_sort(size, arr):
    swapped = True
    i = 0

    while i < size-1 and swapped:
        swapped = False
        j = 0

        while j < size-1-i:
            if arr[j] > arr[j+1]:
                aux = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = aux

                swapped = True

            j += 1

        i += 1
```

Insertion sort

In [5]:
```
def insertion_sort(size, arr):
    for i in range(1, size):
        aux = arr[i]
        j = i - 1

        while j >= 0 and aux < arr[j]:
            arr[j + 1] = arr[j]

            j -= 1

        arr[j + 1] = aux
```

Merge sort

In [6]:
```
def merge_sort(begin, end, arr):
    if begin == end:
        return

    middle = int((begin+end)/2)

    merge_sort(begin, middle, arr)
    merge_sort(middle+1, end, arr)

    aux = []

    i = begin
```

```
    j = middle+1

    while i < middle+1 or j < end+1:
        if i == middle+1:
            aux.append(arr[j])
            j += 1
        elif j == end+1:
            aux.append(arr[i])
            i += 1
        elif arr[i] < arr[j]:
            aux.append(arr[i])
            i += 1
        else:
            aux.append(arr[j])
            j += 1

    for i in range(begin, end+1):
        arr[i] = aux[i-begin]
```

Quick sort

In [7]:

```
def quick_sort(begin, end, arr):
    pivot = arr[int((begin+end)/2)]

    i = begin
    j = end

    while i < j:
        while arr[i] < pivot:
            i += 1

        while arr[j] > pivot:
            j -= 1

        if i <= j:
            aux = arr[i]
            arr[i] = arr[j]
            arr[j] = aux

            i += 1
            j -= 1

    if j > begin:
        quick_sort(begin, j, arr)

    if i < end:
        quick_sort(i, end, arr)
```

Heap sort

In [8]:

```
def heap_sort(size, arr):
    aux = list(arr)
    heapq.heapify(aux)

    for i in range(0, size):
        arr[i] = heapq.heappop(aux)
```

Experimentos

Nosso protocolo de experimentação consiste em rodar determinado algoritmo de ordenação para cada array gerado, tencionando obter o tempo de execução. Para cada algoritmo de ordenação, consideramos duas situações: arrays ordenados e arrays com valores distruibuídos aleatoriamente. Outrossim, para observarmos melhor como o tamanho da entrada afeta o desempenho dos algoritmos, decidimos considerar múltiplos tamanhos de array para cada situação. Sendo assim, adotamos arrays com 10, 100, 1.000 e 10.000 valores inteiros.

Para obtermos o tempo de execução de cada experimento, utilizaremos o comando %timeit.

Tamanho = 10

Arrays ordenados:

Bubble sort

In [9]:
```
size = 10

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 3.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [10]:
```
size = 10

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Tempo de execução: 8.61 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [11]:

```
size = 10

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 29.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [12]:

```
size = 10

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 75.1 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [13]:

```
size = 10

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)
```

```
print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 8.47 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Arrays não ordenados

Bubble sort

In [14]:

```
size = 10

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [1, 6, 8, 9, 5, 7, 2, 4, 3, 0]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 42.9 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [15]:

```
size = 10

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [3, 4, 8, 7, 0, 5, 9, 1, 2, 6]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 13 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [16]:

```
size = 10
```

```
arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [1, 5, 4, 8, 6, 9, 7, 2, 3, 0]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 26.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [17]:

```
size = 10

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [1, 0, 2, 7, 5, 4, 6, 3, 8, 9]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 12 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [18]:

```
size = 10

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [5, 9, 1, 0, 4, 7, 3, 2, 8, 6]
Array após a ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de execução: 8.09 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Tamanho = 100

Arrays ordenados

Bubble sort

In [19]:
```
size = 100

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 18 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [20]:
```
size = 100

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 34.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [21]:
```
size = 100

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))
```

```
print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 327 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [22]:

```
size = 100

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 80.4 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [23]:

```
size = 100

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 45.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Arrays não ordenados

Bubble sort

In [24]:

```
size = 100

arr = generate_array(False, upper_bound = size)
```

```
print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [45, 61, 66, 80, 30, ..., 2, 42, 83, 16, 20]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 4.93 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [25]:

```
size = 100

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [44, 80, 17, 55, 52, ..., 30, 13, 68, 7, 96]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 401 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [26]:

```
size = 100

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [21, 77, 66, 15, 95, ..., 90, 57, 37, 5, 91]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 1.5 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [27]:
```
size = 100

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [64, 57, 62, 35, 34, ..., 86, 9, 73, 69, 76]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 465 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [28]:
```
size = 100

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [34, 49, 42, 78, 32, ..., 47, 89, 17, 13, 69]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 95, 96, 97, 98, 99]

Tempo de execução: 164 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Tamanho = 1.000

Arrays ordenados

Bubble sort

In [29]:
```
size = 1000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 420 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [30]:

```
size = 1000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 582 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [31]:

```
size = 1000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 4.82 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [32]:

```
size = 1000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))
```

```
exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 8.74 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [33]:

```
size = 1000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 1.69 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Arrays não ordenados

Bubble sort

In [34]:

```
size = 1000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [860, 57, 797, 114, 368, ..., 183, 111, 684, 477, 930]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 156 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [35]:
```
size = 1000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [271, 585, 260, 217, 403, ..., 375, 190, 361, 664, 571]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 56.7 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [36]:
```
size = 1000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [355, 236, 799, 825, 959, ..., 11, 704, 147, 592, 498]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 32.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [37]:
```
size = 1000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [286, 756, 758, 595, 898, ..., 92, 337, 809, 291, 1]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 1.78 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [38]:
```
size = 1000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [652, 256, 920, 3, 283, ..., 389, 198, 520, 881, 475]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]

Tempo de execução: 892 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Tamanho = 10.000

Arrays ordenados

Bubble sort

In [39]:
```
size = 10000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 2.99 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [40]:
```
size = 10000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)
```

```
print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 18.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [41]:

```
size = 10000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 141 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [42]:

```
size = 10000

arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 20.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [43]:

```
size = 10000
```

```
arr = generate_array(True, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 5.28 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Arrays não ordenados

Bubble sort

In [44]:

```
size = 10000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o bubble_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [4876, 8713, 1728, 4342, 5786, ..., 2011, 7188, 914, 486, 9140]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 16.8 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Insertion sort

In [45]:

```
size = 10000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o insertion_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [4745, 3650, 5196, 491, 119, ..., 4447, 5864, 4744, 1305, 3623]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 5.33 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Merge sort

In [46]:
```
size = 10000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o merge_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [8495, 4385, 9907, 821, 2587, ..., 9896, 7563, 6017, 8905, 7005]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 70.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Quick sort

In [47]:
```
size = 10000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o quick_sort(0, size-1, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [1757, 2387, 1899, 8967, 4045, ..., 3804, 5372, 3617, 2531, 1062]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 38 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Heap sort

In [48]:
```
size = 10000

arr = generate_array(False, upper_bound = size)

print('Array original: ' + truncate_array_str(size, arr))

exec_time = %timeit -n1 -r1 -q -o heap_sort(size, arr)

print('Array após a ordenação: ' + truncate_array_str(size, arr))

print('\nTempo de execução: ' + str(exec_time))
```

```
Array original: [6271, 7171, 3996, 1433, 3406, ..., 4649, 479, 2901, 2398, 2057]
Array após a ordenação: [0, 1, 2, 3, 4, ..., 9995, 9996, 9997, 9998, 9999]

Tempo de execução: 7.48 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Resultados

Em relação ao Bubble e ao Insertion Sort, algoritmos de ordenação iterativos, podemos inferir que o desempenho é bom quando se trata de arrays já ordenados ou até mesmo de não ordenados com até 100 valores. Esses algoritmos apresentam os melhores tempos de execução nas situações descritas. No entanto, para arrays não ordenados de tamanhos superiores a 100, os algoritmos começam a perder o lugar para algoritmos mais robustos.

Os algoritmos Merge Sort e Quick Sort, que apresentam uma abordagem recursiva, também possuem suas peculiaridades. O Merge apresentou bons resultados para arrays ordenados e de tamanhos menores ou iguais a 100, e para arrays não ordenados com tamanhos maiores que 100. Por outro lado, o Quick Sort não apresentou resultados satisfatórios para arrays já ordenados e de tamanhos pequenos, mas destacou-se com o aumento do tamanho em arrays não ordenados.

O destaque entre os algoritmos de ordenação é o Heap Sort. Este algoritmo conseguiu resultados satisfatórios para arrays ordenados e não ordenados, e de todos os tamanhos utilizados nos experimentos. Mesmo que não tenha resultado no melhor tempo de execução em todas as configurações, o algoritmo é sem dúvidas o mais versátil.

Abaixo, há uma tabela que sintetiza nossos achados.

	Tamanho = 10		**Tamanho = 100**		**Tamanho = 1000**		**Tamanho = 10000**	
	Ordenado	Não Ordenado	Ordenado	Não Ordenado	Ordenado	Não Ordenado	Ordenado	Não Ordenado
Bubble Sort	3.8 us	42.9 us	18 us	4.93 ms	420 us	156 ms	2.99 ms	16.8 s
Insertion Sort	8.61 us	13 us	34.8 us	401 us	582 us	56.7 ms	18.3 ms	5.33 s
Merge Sort	29.8 us	26.6 us	327 us	1.5 ms	4.82 ms	32.4 ms	141 ms	70.1 ms
Quick Sort	75.1 us	12 us	80.4 us	465 us	8.74 ms	1.78 ms	20.9 ms	38 ms
Heap Sort	8.47 us	8.09 us	45.6 us	164 us	1.69 ms	892 us	5.28 ms	7.48 ms