Trabalho 1 - Análise de Algoritmos
Algoritmos de Ordenação

Davi Augusto Neves Leite - RA: 191027383
Luiz Fernando Sementille - RA: 191021032

---

1. Introdução

Este trabalho consiste na aplicação de alguns algoritmos de ordenação comumente utilizados, sendo eles: Bubble Sort, Insertion Sort, Merge Sort, QuickSort, HeapSort; em dois tipos de listas (ordenadas e aleatórias) com tamanho distintos, a fim de se avaliar a complexidade temporal dos mesmos.

Para tanto, as seções foram divididas em: Funções Auxiliares, a qual contem algumas funções utilizadas para facilitar a criação e manipulação das listas; Implementações dos Algoritmos, em que. como o próprio nome diz, traz a implementação na linguagem Python 3 de cada um dos algoritmos de ordenação; Experimentações, cujo intuito está em trazer o tempo médio de execução dos algoritmos tendo em vista os dois tipos de listas com tamanhos distintos; Resultados e Conclusões, na qual traz os resultados obtidos e as considerações finais acerca do tema.

Dessa forma, utilizou-se de dois código-fonte separados: um denominado de *T1_Ordenacao.py* e outro de *T1_Ordenacao_Testes.py*. O primeiro contém uma classe principal denominada *Ordenacao*, a qual estão contidas as implementações dos algoritmos de ordenação; enquanto que o segundo contém as funções auxiliares e a realização dos testes em si. A escolha para trabalhar-se com uma classe separada se deu por conta do conceito de reuso de código e facilidade de implementação posterior, bastando apenas realizar a chamada instância da classe. Por fim, necessitou-se de três bibliotecas para a melhor implementação dos algoritmos de ordenação e das medições de tempo.

Especificamente, abaixo tem-se a importação da biblioteca *heapq*, a qual possui funções primárias de construção e manipulação de uma árvore do tipo Heap (sendo utilizado no HeapSort), e está contida no arquivo *T1_Ordenacao.py* junto da classe principal *Ordenacao*.

In [1]:
```
# Para a utilização da "Heapify" (construir uma Árvore Heap a partir de uma lista) n
import heapq
```

Já as bibliotecas *random* e *time* são utilizadas no arquivo de testes, sendo fundamentais para a geração de listas e demarcação do tempo de execução de cada algoritmo de ordenação, respectivamente.

In [2]:
```
# Importação da Classe "Ordenacao" para uso das funções estabelecidas no outro arqui
from T1_Ordenacao import Ordenacao

import random
import time
```

2. Funções Auxiliares

Para facilitar na criação e manipulação de listas, bem como na medição e exibição dos tempos de processamento dos algoritmos, foram desenvolvidas três funções auxiliares no código-fonte *T1_Ordenacao_Testes.py*.

A primeira, denominada de *geraLista*, é responsável por gerar e retornar uma lista de tamanho previamente especificado e que contenha elementos de 0 ao número equivalente a esse tamanho não incluso (por exemplo: considere um *tamanho = 10*, a lista a ser gerada conterá os elementos de 0 a 9, ordenados ou não). Além disso, possui um parâmetro denominado *ordenado*, o qual representa se a lista deve ser gerada de maneira ordenada (*ordenado=True*) ou de maneira aleatória (*ordenado=False*).

In [3]:
```
def geraLista(limiteSuperior, ordenado=True):
    """
        Função para gerar uma lista, ordenada ou aleatória, de tamanho especificado
        contendo elementos de 0 a um limite superior
    """
    lista = list(range(0, limiteSuperior))  # Geração da lista ordenada

    # Caso seja pra lista ser aleatória
    if ordenado == False:
        random.shuffle(lista)

    return lista
```

A segunda, denominada de *printaLista*, como o próprio nome sugere, é responsável por exibir os valores de uma lista (sendo restrito aos cinco primeiros e cinco últimos) e o tempo de processamento do respectivo algoritmo de ordenação aplicado.

In [4]:
```
def printaLista(lista, tempoProcesso=0.0, tipo=True):
    """
        Função simples para printar uma lista,
        limitando-se nos primeiros cinco e nos últimos cinco elementos

        Além disso, mostra também o tempo de processamento do algoritmo...
    """

    tamanho = len(lista)  # Tamanho da lista

    # Manipular a lista no caso do tamanho ser maior que 10
    if tamanho > 10:
        esquerda = [i for i in lista[0:5]]
        direita = [i for i in lista[-5:]]

        lista = esquerda + ['...'] + direita

    # O tipo define se a lista é ordenada ou não: True - Antes da Ordenação; False -
    if tipo == True:
        print(f'Lista antes da ordenação: {lista}')
    else:
        print(f'Lista depois da ordenação: {lista}\n')
        print(f'Tempo de Processamento: {tempoProcesso:0.4f} milissegundos (ms)\n')
```

A terceira e última, denominada de *executaOrdenacao*, é responsável por gerenciar as duas

funções auxiliares anteriores bem como aplicar um algoritmo de ordenação (especificado por parâmetro) e calcular o seu respectivo tempo de processamento (em milissegundos), exibindo as listas antes e após a ordenação.

In [5]:
```
def executaOrdenacao(algOrd, lista):

    printaLista(lista)

    t0 = time.perf_counter()

    algOrd(lista)

    t1 = time.perf_counter()

    tempoTotal = (t1 - t0) * 1000

    printaLista(lista, tempoTotal, False)
```

---

3. Implementação dos Algoritmos

Essa seção se restringe a exibição separada dos algoritmos de ordenação analisados neste trabalho e na respectiva classe final *Ordenacao*. Para tanto, utilizou-se as melhores implementações de cada um para a linguagem Python 3.

BubbleSort

In [1]:
```
    def bubbleSort(self, lista):
        """
            Versão otimizada do BubbleSort: contém uma variável booleana que
            verifica se a lista já está ordenada.
            Complexidade do Algoritmo: O(n²)
        """
        trocas = True
        tamanhoVetor = len(lista) - 1
        while tamanhoVetor > 0 and trocas:
            trocas = False
            for i in range(tamanhoVetor):
                if lista[i] > lista[i + 1]:
                    trocas = True
                    # Variável Temporária de Troca -> Gasto Espacial
                    lista[i], lista[i + 1] = lista[i + 1], lista[i]
            tamanhoVetor -= 1
```

InsertionSort

In [6]:
```
    def insertionSort(self, lista):
        """Algoritmo de ordenação InsertionSort
        Complexidade do algoritmo: O(n²)

        Args:
            lista (Lista): lista a ser ordenada
        """
        for i in range(1, len(lista)):
            valor = lista[i]
            j = i - 1
```

```
            while j >= 0 and valor < lista[j]:
                lista[j + 1] = lista[j]
                j -= 1

            lista[j + 1] = valor
```

MergeSort

In [7]:

```
    def mergeSort(self, lista):
        """Algoritmo de ordenação MergeSort.
        Complexidade do Algoritmo: O(n*log(n))

        Args:
            lista (List): Lista a ser ordenada
        """
        # Caso-base: vetor com um elemento não é processado
        if len(lista) > 1:
            # Divisão inteira para pegar a posição do meio
            meio = len(lista) // 2
            # Salva a lista do início até o meio (esquerda)
            esquerda = lista[:meio]
            direita = lista[meio:]  # Salva a lista do meio até o fim (direita)

            # Aplica o algoritmo novamente a fim de se obter a menor: Divisão e Conq
            self.mergeSort(esquerda)
            self.mergeSort(direita)

            # Auxiliares para percorrer as sub-listas
            i, j, k = 0, 0, 0

            # Percorrendo as sub-listas e ordenando...
            while i < len(esquerda) and j < len(direita):
                if esquerda[i] < direita[j]:
                    # Salva o elemento ordenado na lista
                    lista[k] = esquerda[i]
                    i += 1
                else:
                    lista[k] = direita[j]
                    j += 1
                k += 1

            # Percorrendo o restante da esquerda
            while i < len(esquerda):
                lista[k] = esquerda[i]
                i += 1
                k += 1

            # Percorrendo o restante da direita
            while j < len(direita):
                lista[k] = direita[j]
                j += 1
                k += 1
```

QuickSort

In [7]:

```
    def _quickSortAuxiliar(self, lista, prim, ult):

        i = prim
        j = ult

        pivo = lista[(i + j) // 2]
```

```
        while i < j:
            while lista[i] < pivo:
                i += 1

            while lista[j] > pivo:
                j -= 1

            if i <= j:
                lista[i], lista[j] = lista[j], lista[i]

                i += 1
                j -= 1

        if j > prim:
            self._quickSortAuxiliar(lista, prim, j)

        if i < ult:
            self._quickSortAuxiliar(lista, i, ult)


    def quickSort(self, lista):
        """Algoritmo de ordenação QuickSort que utiliza a recursão.
        Complexidade do algoritmo: O(n²)

        Args:
            lista (List): lista a ser ordenada
        """

        self._quickSortAuxiliar(lista, 0, len(lista) - 1)
```

HeapSort

In [9]:

```
    def heapSort(self, lista):
        """Algoritmo de ordenação HeapSort.
        Complexidade do algoritmo: O(n*log(n))

        Args:
            lista (List): Lista a ser ordenada
        """
        tamanho = len(lista)

        # Construindo uma HEAP-MÁXIMA a partir da lista
        heapMaxima = list(lista)    # Gasto espacial
        # Construção da HEAP-MAXIMA e é devolvida na variável
        heapq.heapify(heapMaxima)

        for i in range(0, tamanho):
            # Remove o elemento da raiz da heap ordenada e coloca na lista
            lista[i] = heapq.heappop(heapMaxima)
```

Classe *Ordenacao*

In [6]:

```
# Para a utilização da "Heapify" (construir uma Árvore Heap a partir de uma lista) n
import heapq


class Ordenacao():
    """
        Classe para algoritmos de ordenação de vetores by Davi & Luiz
    """
```

```python
    # def __init__(self):
    # print("Classe ''Ordenação'' iniciada com sucesso!\n")

    def bubbleSort(self, lista):
        """
            Versão otimizada do BubbleSort: contém uma variável booleana que
            verifica se a lista já está ordenada.
            Complexidade do Algoritmo: O(n²)
        """
        trocas = True
        tamanhoVetor = len(lista) - 1
        while tamanhoVetor > 0 and trocas:
            trocas = False
            for i in range(tamanhoVetor):
                if lista[i] > lista[i + 1]:
                    trocas = True
                    # Variável Temporária de Troca -> Gasto Espacial
                    lista[i], lista[i + 1] = lista[i + 1], lista[i]
            tamanhoVetor -= 1

    def insertionSort(self, lista):
        """Algoritmo de ordenação InsertionSort
        Complexidade do algoritmo: O(n²)

        Args:
            lista (Lista): lista a ser ordenada
        """
        for i in range(1, len(lista)):
            valor = lista[i]
            j = i - 1

            while j >= 0 and valor < lista[j]:
                lista[j + 1] = lista[j]
                j -= 1

            lista[j + 1] = valor

    def mergeSort(self, lista):
        """Algoritmo de ordenação MergeSort.
        Complexidade do Algoritmo: O(n*log(n))

        Args:
            lista (List): Lista a ser ordenada
        """
        # Caso-base: vetor com um elemento não é processado
        if len(lista) > 1:
            # Divisão inteira para pegar a posição do meio
            meio = len(lista) // 2
            # Salva a lista do início até o meio (esquerda)
            esquerda = lista[:meio]
            direita = lista[meio:]  # Salva a lista do meio até o fim (direita)

            # Aplica o algoritmo novamente a fim de se obter a menor: Divisão e Conq
            self.mergeSort(esquerda)
            self.mergeSort(direita)

            # Auxiliares para percorrer as sub-listas
            i, j, k = 0, 0, 0

            # Percorrendo as sub-listas e ordenando...
            while i < len(esquerda) and j < len(direita):
                if esquerda[i] < direita[j]:
                    # Salva o elemento ordenado na lista
                    lista[k] = esquerda[i]
                    i += 1
```

```
            else:
                lista[k] = direita[j]
                j += 1
            k += 1

        # Percorrendo o restante da esquerda
        while i < len(esquerda):
            lista[k] = esquerda[i]
            i += 1
            k += 1

        # Percorrendo o restante da direita
        while j < len(direita):
            lista[k] = direita[j]
            j += 1
            k += 1


def _quickSortAuxiliar(self, lista, prim, ult):

    i = prim
    j = ult

    pivo = lista[(i + j) // 2]

    while i < j:
        while lista[i] < pivo:
            i += 1

        while lista[j] > pivo:
            j -= 1

        if i <= j:
            lista[i], lista[j] = lista[j], lista[i]

            i += 1
            j -= 1

    if j > prim:
        self._quickSortAuxiliar(lista, prim, j)

    if i < ult:
        self._quickSortAuxiliar(lista, i, ult)


def quickSort(self, lista):
    """Algoritmo de ordenação QuickSort que utiliza a recursão.
    Complexidade do algoritmo: O(n²)

    Args:
        lista (List): lista a ser ordenada
    """

    self._quickSortAuxiliar(lista, 0, len(lista) - 1)

def heapSort(self, lista):
    """Algoritmo de ordenação HeapSort.
    Complexidade do algoritmo: O(n*log(n))

    Args:
        lista (List): Lista a ser ordenada
    """
    tamanho = len(lista)

    # Construindo uma HEAP-MÁXIMA a partir da lista
```

```
        heapMaxima = list(lista)    # Gasto espacial
        # Construção da HEAP-MAXIMA e é devolvida na variável
        heapq.heapify(heapMaxima)

        for i in range(0, tamanho):
            # Remove o elemento da raiz da heap ordenada e coloca na lista
            lista[i] = heapq.heappop(heapMaxima)
```

---

4. Experimentações

Os experimentos foram divididos da seguinte forma: primeiramente, é definido um tamanho para a geração de listas, sendo utilizado tamanhos variando de 10, 100, 1000 e 10000 para a análise neste trabalho. Após isso, é definido o tipo de lista a ser gerada: aleatória ou ordenada, sendo o primeiro utilizado a função *shuffle* da biblioteca *random* para embaralhar os elementos. Por fim, é aplicado um algoritmo de ordenação e exibido o respectivo tempo de processamento (em milissegundos).

Desta forma, pode-se mensurar, em termos de complexidade temporal, as principais diferenças entre os algoritmos de ordenação utilizados neste trabalho.

Tamanho = 10

In [7]:
```
tamanho = 10
```

Não-Ordenado

In [8]:
```
tipo = False
```

BubbleSort

In [9]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [2, 1, 5, 3, 0, 6, 4, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0074 milissegundos (ms)
```

InsertionSort

In [10]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [9, 6, 2, 0, 5, 7, 1, 8, 3, 4]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0307 milissegundos (ms)
```

MergeSort

In [11]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [3, 9, 0, 6, 2, 5, 8, 7, 1, 4]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0371 milissegundos (ms)
```

QuickSort

In [12]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [8, 6, 5, 9, 0, 1, 2, 7, 3, 4]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0204 milissegundos (ms)
```

HeapSort

In [13]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [0, 4, 7, 5, 3, 8, 9, 6, 1, 2]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0120 milissegundos (ms)
```

Ordenado

In [14]:
```
tipo = True
```

BubbleSort

In [15]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0100 milissegundos (ms)
```

InsertionSort

In [16]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0071 milissegundos (ms)
```

MergeSort

In [17]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0766 milissegundos (ms)
```

QuickSort

In [18]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0275 milissegundos (ms)
```

HeapSort

In [19]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois da ordenação: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Tempo de Processamento: 0.0101 milissegundos (ms)
```

Tamanho = 100

In [20]:
```
tamanho = 100
```

Não-Ordenado

In [21]:
```
tipo = False
```

BubbleSort

In [22]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [18, 0, 95, 23, 47, '...', 7, 4, 69, 33, 10]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 1.0631 milissegundos (ms)
```

InsertionSort

In [23]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [35, 36, 94, 32, 97, '...', 76, 24, 99, 33, 95]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
```

```
Tempo de Processamento: 0.5454 milissegundos (ms)
```

MergeSort

In [24]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [16, 21, 73, 45, 95, '...', 71, 89, 63, 75, 64]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.5697 milissegundos (ms)
```

QuickSort

In [25]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [40, 26, 22, 21, 27, '...', 80, 12, 52, 16, 64]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.1806 milissegundos (ms)
```

HeapSort

In [26]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [45, 25, 40, 60, 59, '...', 88, 87, 69, 57, 90]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.0882 milissegundos (ms)
```

Ordenado

In [27]:
```
tipo = True
```

BubbleSort

In [28]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.0154 milissegundos (ms)
```

InsertionSort

In [29]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.0127 milissegundos (ms)
```

MergeSort

In [30]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.1931 milissegundos (ms)
```

QuickSort

In [31]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.0559 milissegundos (ms)
```

HeapSort

In [32]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 95, 96, 97, 98, 99]

Tempo de Processamento: 0.0298 milissegundos (ms)
```

Tamanho = 1000

In [33]:
```
tamanho = 1000
```

Não-Ordenado

In [34]:
```
tipo = False
```

BubbleSort

In [35]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [705, 996, 528, 33, 601, '...', 926, 584, 606, 448, 796]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 76.7258 milissegundos (ms)
```

InsertionSort

In [36]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [490, 228, 968, 298, 985, '...', 715, 424, 179, 269, 578]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 63.9077 milissegundos (ms)
```

MergeSort

In [37]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [665, 683, 342, 599, 819, '...', 620, 905, 598, 817, 201]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 10.8624 milissegundos (ms)
```

QuickSort

In [38]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [389, 242, 35, 372, 665, '...', 63, 344, 41, 246, 752]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 3.5113 milissegundos (ms)
```

HeapSort

In [39]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [746, 70, 841, 483, 618, '...', 206, 685, 613, 552, 292]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 0.6015 milissegundos (ms)
```

Ordenado

In [40]:
```
tipo = True
```

BubbleSort

In [41]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 0.2351 milissegundos (ms)
```

InsertionSort

In [42]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
```

```
Tempo de Processamento: 0.2723 milissegundos (ms)
```

MergeSort

In [43]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 4.5194 milissegundos (ms)
```

QuickSort

In [44]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 1.2644 milissegundos (ms)
```

HeapSort

In [45]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 995, 996, 997, 998, 999]

Tempo de Processamento: 0.5686 milissegundos (ms)
```

Tamanho = 10000

In [46]:
```
tamanho = 10000
```

Não-Ordenado

In [47]:
```
tipo = False
```

BubbleSort

In [48]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [2275, 7589, 2488, 7183, 9272, '...', 906, 4975, 3825, 73
7, 2555]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 15562.7740 milissegundos (ms)
```

InsertionSort

In [49]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [6319, 7128, 1039, 9396, 5506, '...', 1959, 2684, 4997, 85
63, 22]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 6122.4051 milissegundos (ms)
```

MergeSort

In [50]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [4441, 3163, 6626, 3410, 1047, '...', 8209, 8066, 4120, 83
26, 5683]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 74.3101 milissegundos (ms)
```

QuickSort

In [51]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [9599, 6840, 765, 5592, 2939, '...', 6740, 4314, 4908, 328
4, 1446]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 42.6471 milissegundos (ms)
```

HeapSort

In [52]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [1483, 1643, 6106, 1617, 8920, '...', 6834, 5796, 7993, 74
53, 4399]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 9.6814 milissegundos (ms)
```

Ordenado

In [53]:
```
tipo = True
```

BubbleSort

In [54]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().bubbleSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
```

```
Tempo de Processamento: 1.5928 milissegundos (ms)
```

InsertionSort

In [55]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().insertionSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 6.0687 milissegundos (ms)
```

MergeSort

In [56]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().mergeSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 62.2840 milissegundos (ms)
```

QuickSort

In [57]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().quickSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 28.1851 milissegundos (ms)
```

HeapSort

In [58]:
```
lista = geraLista(tamanho, tipo)
executaOrdenacao(Ordenacao().heapSort, lista)
```

```
Lista antes da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]
Lista depois da ordenação: [0, 1, 2, 3, 4, '...', 9995, 9996, 9997, 9998, 9999]

Tempo de Processamento: 7.5465 milissegundos (ms)
```

---

5. Resultados e Conclusões

Para inferir sobre o desempenho dos algoritmos é necessário ter em vista que os algoritmos avaliados compartilham, em algumas situações, técnicas semelhantes como o aninhamento de laços de repetição, dividir para conquistar e lista de prioridades.

Tabela de Tempo (em ms)	Tamanho = 10		Tamanho = 100		Tamanho = 1000		Tamanho = 10000	
	Não Ordenado	Ordenado	Não Ordenado	Ordenado	Não Ordenado	Ordenado	Não Ordenado	Ordenado
BubbleSort	0,0074	0,0100	1,0631	0,0154	76,7258	0,2351	15562,7740	1,5928
InsertionSort	0,0307	0,0071	0,5454	0,0127	63,9077	0,2723	6122,4051	6,0687
MergeSort	0,0371	0,0766	0,5697	0,1931	10,8624	4,5194	74,3101	62,2840
QuickSort	0,0204	0,0275	0,1806	0,0559	3,5113	1,2644	42,6471	28,1851
HeapSort	0,0120	0,0101	0,0882	0,0298	0,6015	0,5686	9,6814	7,5465

Os algoritmos BubbleSort e InsertionSort, que utilizam o aninhamento de laços de repetição (daí a complexidade $O(n^2)$), performaram melhor em relação aos outros algoritmos, quando testados com listas de todos os tamanhos desde que já ordenadas, visto que apenas a execução de uma iteração completa sobre a lista é necessária para identificar que a lista já está ordenada. Agora, em se tratando de listas não ordenadas, esses algoritmos performam bem apenas em listas de até 100 elementos.

Os algoritmos de MergeSort e QuickSort, que utilizam a abordagem de dividir para conquistar, tendem a ter uma performance aceitável com vetores de tamanhos menores, mas o melhor desempenho desses algoritmos foram extraídos com listas maiores e não ordenadas, visto que dividem a lista maior em sublistas menores.

Já o algoritmo de HeapSort, que utiliza a estrutura de dados Heap (Fila de Prioridades), performou de forma excelente em listas não ordenadas de qualquer tamanho e performou de forma aceitável em listas já ordenadas de qualquer tamanho. Diante disso, é possível afirmar que esse algoritmo foi o mais "equilibrado" dentre os analisados.

Logo, é possível concluir que não existe um único algoritmo que tenha um desempenho excelente em todos os casos e sob todas as variáveis, mas é possível definir quais algoritmos desempenham melhor em cada situação desde que se conheça a natureza da lista previamente.