UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE CRATEÚS

CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

TRABALHO PRÁTICO I

IMPLEMENTAÇÃO DOS ALGORITMOS DE ORDENAÇÃO

CRATEÚS, CE.

FRANCISCO HENRIQUE DA COSTA SILVA
FRANCISCO LUCAS SOARES BATISTA
IUDE KILDARE DE MENESES
JEOVÁ CAÇULA DE AGUIAR JUNIOR

TRABALHO PRÁTICO - I

IMPLEMENTAÇÃO DOS ALGORITMOS DE ORDENAÇÃO

Relatório apresentado como pré-requisito para aprovação na disciplina de Projeto e Análise de Algoritmos.

Professor: Rafael Martins Barros.

CRATEÚS, CE

SUMÁRIO

1. Introdução

O presente relatório visa apresentar as implementações dos algoritmos de ordenação, a saber, o *insertion sort, merge sort, quick sort* e *selection sort.* Assim como, traçar um comparativo entre os tempos de execução dos mesmos para determinadas entradas, a fim de avaliarmos a performance dos mesmos.

Também vamos apresentar uma solução híbrida que combina um pouco dos algoritmos anteriores, a fim de que ele possa ter um desempenho satisfatório mantendo uma complexidade aceitável.

Adicionalmente, este relatório também inclui o desenvolvimento e a implementação de uma solução híbrida que combina características de alguns dos algoritmos previamente mencionados. A ideia é projetar um algoritmo que alie um desempenho prático satisfatório a uma complexidade aceitável, atendendo assim às necessidades de aplicações em diferentes escalas de dados.

Os resultados obtidos serão apresentados em forma de gráficos e análises detalhadas, com o objetivo de verificar se os comportamentos observados estão alinhados com as expectativas baseadas em suas complexidades assintóticas teóricas.

2. Implementação dos Algoritmos de ordenação

I. Bubble Sort

```
    def bubbleSort(arr):
    for n in range(len(arr)-1,0,-1):
        for i in range(n):
            if arr[i]>arr[i+1]:
                temp = arr[i]
                arr[i] = arr[i+1]
                arr[i+1] = temp
```

II. Insertion Sort

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

    for i in range(n):
        aux = arr[i]
        j = i-1

        while j >= 0 and arr[j] > aux:
```

```
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = aux
```

III. Quick Sort

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quicksort(left) + middle + quicksort(right)
```

IV. Merge Sort

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    leftHalf = arr[:mid]
    rightHalf = arr[mid:]

    sortedLeft = mergeSort(leftHalf)
    sortedRight = mergeSort(rightHalf)

    return merge(sortedLeft, sortedRight)

def merge(left, right):
    result = []
    i = j = 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
```

```
    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

3. Implementação de Algoritmo de ordenação híbrido

Optamos, com base nos resultados obtidos pelos testes, que mesclar o *insertion sort* com o *quick sort* geraria um algoritmo que possibilita resultados satisfatórios.

```
def insertionQuickHibrid(arr):
    if len(arr) <= 100:
        insertion_sort(arr)
        return arr
    else:
        quicksort(arr)
        return arr
```

Como o *insertion sort* se comporta bem para ordenação de vetores pequenos(na nossa análise o tamanho ficou em torno de 100) decidimos fazer uma verificação inicial para ver se o vetor é menor ou igual a 100. Caso seja, vai ser tratado pelo função *insertion sort*. Do contrário (ele ser maior que 100), adotamos a estratégia do *quick sort*, que vai ter um bom desempenho para os casos maiores que 100. Desta forma, conseguimos um algoritmo que possui bom desempenho e mantém uma complexidade. n log n no melhor caso e no caso médio e, no pior caso, n ao quadrado.

4. Testes realizados para validar as implementações

Geração de números aleatórios

```
import numpy as np


n_sizes = [10, 10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8]
```

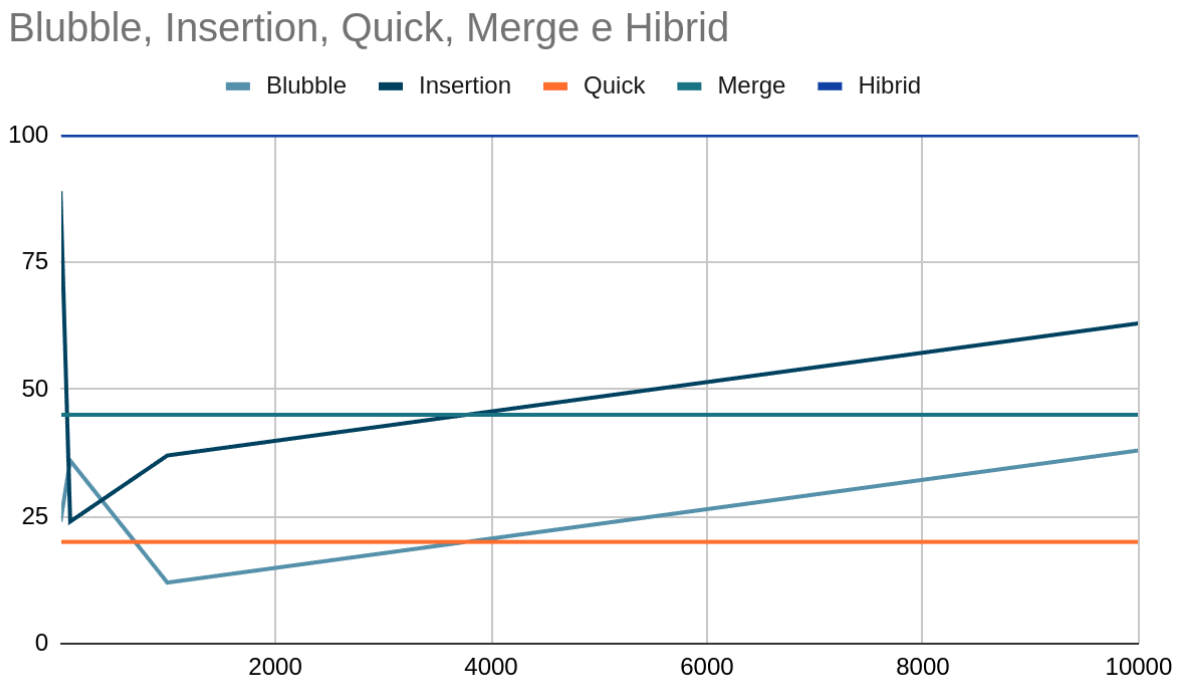
```
data = {}

rng = np.random.default_rng()


for n in n_sizes:

      data[n] = rng.integers(low=0, high=100, size=n)
```

Função para mensurar o tempo em segundos

```
import time


def measure_execution_time(func, arr):

    times = []

    for _ in range(10):

        temp = arr.copy()

        start = time.time()

        func(temp)

        end = time.time()

        times.append(end - start)

    return sum(times) / len(times)
```

Resultado em gráficos

Devido a restrições de processamento e sistemas operacionais relacionados ao uso de recursos pelo python3, não podemos trazer resultados para arrays de tamanho na casa dos milhões. Pode-se observar no gráfico abaixo o comportamento dos algoritmos para entradas suficientemente grandes de arrays.



5. Conclusão

A nossa análise constatou que o algoritmo *bubble sort* apresentou o pior desempenho, principalmente para valores de ordem mais elevada. Já os algoritmos *quick* e *merge sort* apresentaram bom desempenho para valores de ordem maior (>=100000). O algoritmo híbrido, consequentemente obteve um resultado bastante satisfatório, já que combinou os melhores cenários. Utiliza o *insertion sort* para valores menores que 100 e o *quick sort* para valores maiores que 100.