# gianfrancojoel_condoriluna_aula4Pratico

September 3, 2024

## 1 Aula 04 - Exercício prático Ordenação n log n

### 1.0.1 1) Execute os códigos do Shellsort, Heapsort, Quicksort e Mergesort num vetor de 100.000 elementos preenchidos aleatoriamente, em ordem crescente e decrescente e mostre o tempo de execução que cada algoritmo levou para ordenar o vetor.

**Solução:**

[16]:
```
# SHELLSORT
# Fonte: ChatGPT

def generate_gaps(n):
    gaps = []
    k = 1
    gap = 1

    # Generar gaps usando la fórmula: gap = (3^k - 1) / 2
    while gap < n:
        gaps.append(gap)
        k += 1
        gap = (3**k - 1) // 2

    # Retornar la lista de gaps en orden descendente
    return gaps[::-1]

# Generar la lista de gaps para 100,000 elementos
#n = 100000
#gaps = generate_gaps(n)

def shell_sort(arr, gaps):
    n = len(arr)

    # Itera sobre cada valor de h (gap)
    for gap in gaps:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            # Realiza la comparación y el intercambio dentro del gap
```

```
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp

    return arr

# Ejemplo de uso
arr = [22, 35, 2, 1, 13, 7]
gaps = generate_gaps(len(arr))
sorted_arr = shell_sort(arr, gaps)
print("Arreglo ordenado:", sorted_arr)
```

```
Arreglo ordenado: [1, 2, 7, 13, 22, 35]
```

[8]:
```
# QUICKSORT
# Fonte: ChatGPT

def quicksort(arr, low, high):
    if low < high:
        # Particionamos el array y obtenemos el índice del pivote
        pivot_index = partition(arr, low, high)

        # Ordenamos las dos mitades del array de forma recursiva
        quicksort(arr, low, pivot_index - 1)
        quicksort(arr, pivot_index + 1, high)

def partition(arr, low, high):
    pivot = arr[low]  # Elegimos el primer elemento como pivote
    i = low + 1  # Índice para recorrer desde la izquierda
    j = high  # Índice para recorrer desde la derecha

    while True:
        # Avanzamos con `i` hasta encontrar un elemento mayor que el pivote
        while i <= j and arr[i] <= pivot:
            i += 1

        # Retrocedemos con `j` hasta encontrar un elemento menor que el pivote
        while i <= j and arr[j] >= pivot:
            j -= 1

        # Si `i` ha pasado `j`, terminamos la partición
        if i > j:
            break

        # Intercambiamos los elementos que están fuera de lugar
        arr[i], arr[j] = arr[j], arr[i]
```

```
    # Intercambiamos el pivote con el elemento en la posición `j`
    arr[low], arr[j] = arr[j], arr[low]

    return j  # Devolvemos el índice del pivote

# Ejemplo de uso
arr = [3, 6, 8, 10, 1, 2, 1]
quicksort(arr, 0, len(arr) - 1)
print("Arreglo ordenado:", arr)
```

```
Arreglo ordenado: [1, 1, 2, 3, 6, 8, 10]
```

[9]:
```
# HEAPSORT
# Fonte: ChatGPT

def heapify(arr, n, i):
    largest = i  # Inicializamos el nodo raíz como el más grande
    left = 2 * i + 1  # Hijo izquierdo
    right = 2 * i + 2  # Hijo derecho

    # Si el hijo izquierdo es más grande que la raíz
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Si el hijo derecho es más grande que el más grande hasta ahora
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Si el más grande no es la raíz
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Intercambiar
        heapify(arr, n, largest)  # Recursivamente hacer heapify en el subárbol␣
 ↪afectado

def heapsort(arr):
    n = len(arr)

    # Construimos el max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extraemos elementos del heap uno por uno
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Mover la raíz actual al final
        heapify(arr, i, 0)  # Llamar a heapify en el heap reducido
```

```
# Ejemplo de uso
arr = [12, 11, 13, 5, 6, 7]
heapsort(arr)
print("Arreglo ordenado:", arr)
```

```
Arreglo ordenado: [5, 6, 7, 11, 12, 13]
```

[10]:
```
# MERGESORT
# Fonte: ChatGPT

def merge_sort(arr):
    if len(arr) > 1:
        # Encuentra el punto medio del arreglo
        mid = len(arr) // 2

        # Divide el arreglo en dos mitades
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Ordena cada mitad
        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        # Une las dos mitades ordenadas en un solo arreglo
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Verifica si quedan elementos en la mitad izquierda
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # Verifica si quedan elementos en la mitad derecha
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

```
# Ejemplo de uso
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Arreglo ordenado:", arr)
```

```
Arreglo ordenado: [5, 6, 7, 11, 12, 13]
```

[70]:
```
# vetor de 100.000 elementos preenchidos aleatoriamente,
# Se trocou para só 1000 elementos porque o quicksort não executaba
# Library
import numpy as np
from datetime import datetime
# Semilla
np.random.seed(42)

# Create a vector with 100,000 random elements in the range 2 to 1,000
vector_aleatorio = np.random.randint(2, 1001, size=1000)

print("Size vector: ", len(vector_aleatorio))
vector_aleatorio[:10]
```

```
Size vector:  1000
```

[70]:
```
array([104, 437, 862, 272, 108,  73, 702,  22, 616, 123])
```

[71]:
```
vector_ordenado_asc = sorted(vector_aleatorio)
vector_ordenado_asc[:10]
```

[71]:
```
[2, 3, 3, 3, 6, 6, 9, 10, 10, 11]
```

[72]:
```
vector_ordenado_desc = sorted(vector_aleatorio, reverse=True)
vector_ordenado_desc[:10]
```

[72]:
```
[1000, 1000, 998, 998, 997, 997, 994, 994, 993, 991]
```

### 1.0.2 Teste com o Shellsort

**Vetor random**

[61]:
```
# GAP
gaps = generate_gaps(1000)
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    shell_sort(vector_aleatorio, gaps)
```

```
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.002811
```

**Vetor ordem crescente**

[62]:
```
# GAP
gaps = generate_gaps(1000)
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    shell_sort(vector_ordenado_asc, gaps)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.001219
```

**Vetor ordem decrescente**

[63]:
```
# GAP
gaps = generate_gaps(1000)
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    shell_sort(vector_ordenado_desc, gaps)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.001714
```

### 1.0.3 Teste com o Quicksort

NOTA: Reinicializamos as variáveis "vector_aleatorio", "vector_ordenado_asc" e "vector_ordenado_desc".

**Vetor random**

[55]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    quicksort(vector_aleatorio, 0, len(vector_aleatorio) - 1)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.043990
```

**Vetor ordem crescente**

[56]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    quicksort(vector_ordenado_asc, 0, len(vector_ordenado_asc) - 1)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.028637
```

**Vetor ordem decrescente**

[57]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    quicksort(vector_ordenado_desc, 0, len(vector_ordenado_desc) - 1)
```

```
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.028583
```

### 1.0.4 Teste com o Heapsort

NOTA: Reinicializamos as variáveis "vector_aleatorio", "vector_ordenado_asc" e "vector_ordenado_desc".

**Vetor random**

[67]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    heapsort(vector_aleatorio)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.009989
```

**Vetor ordem crescente**

[68]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    heapsort(vector_ordenado_asc)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.007916
```

**Vetor ordem decrescente**

[69]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    heapsort(vector_ordenado_desc)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.008632
```

### 1.0.5 Teste com o Mergesort

NOTA: Reinicializamos as variáveis "vector_aleatorio", "vector_ordenado_asc" e "vector_ordenado_desc".

**Vetor random**

[73]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    merge_sort(vector_aleatorio)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.006366
```

**Vetor ordem crescente**

[74]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    merge_sort(vector_ordenado_asc)
```

```
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.003989
```

**Vetor ordem decrescente**

[75]:
```
# Almazena o tempo
array_time = []

for i in range(10):
    # Initial time
    tiempo_inicial = datetime.now()
    merge_sort(vector_ordenado_desc)
    # End time
    tiempo_final = datetime.now()
    diferencia_tiempo = tiempo_final - tiempo_inicial
    array_time.append(diferencia_tiempo)

print("Time average: ", np.mean(array_time))
```

```
Time average:  0:00:00.002687
```

| Algoritmo | Vetor random | Vetor ordenado crescente | Vetor ordenado decrescente |
|---|---|---|---|
| Shellsort | 00.002811 s | 00.001219 s | 00.001714 s |
| Quicksort | 00.043990 s | 00.028637 s | 00.028583 s |
| Heapsort | 00.009989 s | 00.007916 s | 00.008632 s |
| Mergesort | 00.006366 s | 00.003989 s | 00.002687 s |

NOTA: A quantidade do array teve que ser reduzida para apenas 1.000 elementos porque o quicksort não foi executado nem uma vez.

Comente as questões a seguir:

Qual desses vc considera que seria o melhor algoritmo n log n?

- O melhor de acordo com os tempos que obtivemos nesses testes seria o Shellsort.

Considere diferentes pivôs no Quicksort, houve alguma diferença no tempo?

- Sim, varia dependendo da escolha do pivô. A escolha do pivô influencia a eficiência do algoritmo

Considere diferentes valores h no Shellsort, houve alguma diferença no tempo?

- Sim, a escolha dos valores de h no Shellsort pode ter um impacto significativo no tempo de execução do algoritmo. O desempenho do Shellsort é fortemente influenciado pela escolha desses valores de gap.