UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERIA DE MECANICA ELECTRICA, ELECTRONICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERIA DE SISTEMAS

Práctica de Laboratorio: Análisis de Complejidad y Algoritmos de Ordenamiento

ANALISIS Y DISEÑO DE ALGORITMOS

INGENIERA:

INGALUQUE ARAPA MARGA ISABEL

PRESENTADO POR:

QUISPE CHATA CARLOS DAVID

2024 - I

PUNO – PERU

Parte 1: Implementación de Algoritmos de Ordenamiento (40 minutos)

1. **Ordenamiento por Burbuja**:

    - Implementar el algoritmo de **ordenamiento por burbuja** y aplicarlo en la lista siguiente: **5,2,9,1,5,6**.

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

lista = [5, 2, 9, 1, 5, 6]
print(burbuja(lista))

```

Lista Ordenada:

```
[Running] python -u "d:\Documentos\PRACTICAS\Burbuja.py"
[1, 2, 5, 5, 6, 9]
```

- **Analizar cuántas iteraciones son necesarias para ordenar completamente la lista.**

  Las iteraciones necesarias son 3 iteraciones para ordenar la lista dada

- Probar con otras listas y reflexionar sobre su eficiencia en el **mejor caso** y **peor caso**.

  Primera lista peor caso:

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

  Segunda lista mejor caso:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Respuestas:

- **Número de iteraciones (peor caso)**: _____36 iteraciones_____
- **Número de iteraciones (mejor caso)**: _____1 iteración_____

13. **Ordenamiento por Inserción**:

    - Implementar el **algoritmo de inserción** y aplicarlo en la lista siguiente: **4,3,2,10,12,1,5**.

```python
#insercion
def insercion(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

lista = [4, 3, 2, 10, 12, 1, 5]
print(insercion(lista))

```

Lista ordenada:

```
[Running] python -u "d:\Documentos\PRACTICAS\Insercion.py"
[1, 2, 3, 4, 5, 10, 12]
```

- **Observar cuántas operaciones son necesarias para que el número 10 alcance su posición correcta.**

  Operaciones necesarias: 14 operaciones serian necesarias para el numero 10 este en su posición correcta ósea el índice 5.

- **Probar con otras listas y analizar la eficiencia en diferentes escenarios.**

  Voy ha probar una lista en el peor de los casos y una lista que este desordenada aleatoriamente.

  Lista en el peor de los casos :

```
lista_peor_caso = [5, 4, 3, 2, 1]
```

el tiempo de ejecución nos da: 0.451 seconds

Lista ordena de manera random:

```
lista = [7, 2, 9, 4, 1, 6, 3]
el tiempo de ejecución nos da: in 0.525 seconds
```

Respuestas:

- **Posición final del 10**:La posición final del numero 10 es en el índice 5

28. **Ordenamiento por Selección**:
    - Implementar el **algoritmo de selección** y aplicarlo en la lista siguiente: **64,25,12,22,11**.

```python
def seleccion(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

lista = [64, 25, 12, 22, 11]

print(seleccion(lista))
```

Analizar cómo queda la lista después de cada iteración y reflexionar sobre su eficiencia comparado con los otros algoritmos.

- **Primera iteración (i = 0)**:

    - Lista actual: [64, 25, 12, 22, 11]
    - Busca el mínimo entre los elementos restantes (desde 64 hasta 11). El mínimo es 11.
    - Intercambia 64 con 11.
    - **Lista después de la primera iteración**: [11, 25, 12, 22, 64]

Segunda iteración (i = 1):

- Lista actual: [11, 25, 12, 22, 64]
- Busca el mínimo entre los elementos restantes (desde 25 hasta 64). El mínimo es 12.
- Intercambia 25 con 12.
- **Lista después de la segunda iteración**: [11, 12, 25, 22, 64]

Tercera iteración (i = 2):

- Lista actual: [11, 12, 25, 22, 64]
- Busca el mínimo entre los elementos restantes (desde 25 hasta 64). El mínimo es 22.
- Intercambia 25 con 22.
- **Lista después de la tercera iteración**: [11, 12, 22, 25, 64]

Cuarta iteración (i = 3):

- Lista actual: [11, 12, 22, 25, 64]
- Busca el mínimo entre los elementos restantes (desde 25 hasta 64). El mínimo es 25, por lo que no hay intercambios.
- **Lista después de la cuarta iteración**: [11, 12, 22, 25, 64]

 Quinta iteración (i = 4):

- Lista actual: [11, 12, 22, 25, 64]
- El último elemento ya está en su posición, por lo que no se hace nada.
- **Lista después de la quinta iteración**: [11, 12, 22, 25, 64]

  -

Respuestas:

  - **Lista después de la primera iteración**: La lista quedaría de la siguiente manera después de la primera iteración buscando el menor valor y intercambiarlo con el índice 0.

    Lista después de la primera iteración: [11, 25, 12, 22, 64]

Parte 2: Análisis de Complejidad (30 minutos)

1. **Complejidad Temporal**:
   - Para cada algoritmo, describe su **complejidad temporal** en el **mejor**, **peor** y **caso promedio**.
   - Reflexiona sobre cuál algoritmo es más eficiente en cada caso y cuál deberías usar en situaciones como listas grandes, listas ordenadas o listas con desorden moderado.

Respuestas:

  - **Burbuja (complejidad)**:
    - **Peor caso: O(n^2)** Esto Sucede cuando la lista está en orden inverso. En este caso, cada elemento tiene que intercambiarse múltiples veces.
    - **Mejor caso: O(n)** Esto ocurre cuando la lista ya está ordenada. En este caso, el algoritmo hará solo una pasada por la lista y se detendrá.
    - **Promedio: O(n^2)** En promedio, el algoritmo burbuja tendrá que hacer comparaciones e intercambios en aproximadamente la mitad de los elementos en cada pasada.
  - **Inserción (complejidad)**:
    - **Mejor caso**: **O(n)**Este escenario ocurre cuando la lista ya está ordenada.
    - **Peor caso**: **O(n^2)**El peor caso ocurre cuando la lista está en orden inverso. Cada elemento tendrá que moverse hasta el principio de la lista, lo que implica hacer comparaciones y movimientos múltiples para cada elemento.

- **Caso promedio**: **O(n^2)**En una lista desordenada de manera aleatoria, el algoritmo tendrá que hacer movimientos y comparaciones para alrededor de la mitad de los elementos.

- **Selección (complejidad):**

- **Mejor caso**: **O(n^2)** El algoritmo de selección **no se ve afectado** por si la lista ya está ordenada. Siempre necesita recorrer la lista entera para encontrar el mínimo en cada iteración.
- **Peor caso**: **O(n^2)** El peor caso ocurre cuando la lista está en el peor orden esto no afecta la cantidad de comparaciones que realiza, ya que sigue buscando el mínimo en cada iteración.
- **Caso promedio**: **O(n^2)** Al igual que en los otros casos, el algoritmo de selección siempre realiza el mismo número de comparaciones y movimientos, sin importar el nivel de desorden de la lista.

Parte 3: Comparación y Reflexión Final (20 minutos)

1. **Comparación de Algoritmos**:

   - Reflexiona sobre cuál de los algoritmos implementarías en una aplicación que trabaja con grandes volúmenes de datos. Justifica tu elección basándote en la **complejidad temporal** de cada uno.

Respuesta:

Implementaría el algoritmo de Inserción es el más eficiente cuando los datos ya están ordenados o casi ordenados, con una mejor complejidad en el mejor caso (**O(n)**).

2. **Conclusión sobre el Uso de Algoritmos de Ordenamiento**:

   - ¿Qué consideraciones tomarías al elegir un algoritmo de ordenamiento para una aplicación en tiempo real, donde tanto el tiempo de ejecución como el uso de memoria son críticos?

Respuesta:

Al elegir un algoritmo de ordenamiento para una aplicación en tiempo real consideraría lo siguiente:

- Complejidad Temporal
- Complejidad Espacial
- Estabilidad
- Tamaño y el tipos de datos

Enlace al código: https://github.com/carlos5845/Implementacion-de-los-algoritmos