PROGRAMACIÓN 1

1º AÑO

CLASE Nº 9

ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN EN ARREGLOS

CONTENIDO

En la clase de hoy trabajaremos los siguientes temas:

- Ordenación ascendente
- Ordenación descendente
- Búsqueda Lineal
- Búsqueda Binaria
- Introducción a la Programación Orientada a objetos

1. PRESENTACIÓN

Bienvenidos a la clase N° 9 del bloque Programación 1. En la clase pasada hemos trabajado aspectos básicos de listas y estructuras iterativas. En esta clase desarrollaremos los temas de ordenamiento y búsqueda de datos, fundamentales para resolver problemas de forma eficiente, y además haremos una primera introducción a la Programación Orientada a Objetos (POO).

Este conjunto de conceptos es importante porque permite organizar, manipular y recuperar información dentro de grandes volúmenes de datos y estructurar mejor el código para futuros proyectos.

DESARROLLO

Los conceptos que estudiaremos en esta clase se relacionan con lo que vimos anteriormente en la clase N° 8 sobre listas y estructuras de control. Además, introduciremos uno de los paradigmas más usados en el desarrollo de software: la Programación Orientada a Objetos.

Ordenación ascendente con sorted() y sort()

sorted() es una función incorporada que toma un iterable (como una lista) y devuelve una nueva lista ordenada. sort() es un método que se aplica directamente a una lista para ordenarla en su lugar.

```
# Usando sorted()
original_list = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_list = sorted(original_list)
print(sorted_list)  # Resultado: [1, 1, 2, 3, 4, 5, 6, 9]
print(original_list)  # Resultado: [3, 1, 4, 1, 5, 9, 2, 6] (sin cambios)

# Usando sort()
original_list.sort()
print(original_list)  # Resultado: [1, 1, 2, 3, 4, 5, 6, 9] (modificado)
```

Explicación:

1- Creamos una lista llamada original_list que contiene números desordenados: [3, 1, 4, 1, 5, 9, 2, 6].

2- Usamos la función sorted(original_list) para ordenar los elementos de original_list y crear una nueva lista ordenada llamada sorted_list. Esta función devuelve una nueva lista sin modificar la lista original.

3- Imprimimos la lista sorted_list, que ahora contiene los números ordenados en forma ascendente: [1, 1, 2, 3, 4, 5, 6, 9]. Luego, imprimimos la lista original_list para demostrar que no ha sido modificada: [3, 1, 4, 1, 5, 9, 2, 6].

4- A continuación, utilizamos el método sort() directamente en la lista original_list. Esto ordenará los elementos de la lista original en su lugar, es decir, modificará la lista original directamente.

5- Finalmente, imprimimos la lista original_list una vez más para mostrar que ahora está ordenada: [1, 1, 2, 3, 4, 5, 6, 9].

En resumen, tanto la función sorted() como el método sort() se utilizan para ordenar listas, pero sorted() crea una nueva lista ordenada sin modificar la original, mientras que sort() modifica la lista original directamente.

Ordenación descendente con el argumento reverse

Tanto sorted() como sort() pueden aceptar el argumento reverse=True para realizar una ordenación descendente.

```python
original_list = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_descending = sorted(original_list, reverse=True)
print(sorted_descending)  # Resultado: [9, 6, 5, 4, 3, 2, 1, 1]

original_list.sort(reverse=True)
print(original_list)  # Resultado: [9, 6, 5, 4, 3, 2, 1, 1]
```

Explicación:

1- Creamos una lista llamada original_list que contiene números desordenados: [3, 1, 4, 1, 5, 9, 2, 6].

2- Usamos la función sorted(original_list, reverse=True) para ordenar los elementos de original_list en orden descendente. El argumento reverse=True indica que queremos ordenar en sentido descendente.

3- Imprimimos la lista sorted_descending, que ahora contiene los números ordenados en orden descendente: [9, 6, 5, 4, 3, 2, 1, 1].

4- A continuación, utilizamos el método sort(reverse=True) directamente en la lista original_list. Esto ordenará los elementos de la lista original en orden descendente.

5- Finalmente, imprimimos la lista original_list una vez más para mostrar que ahora está ordenada en orden descendente: [9, 6, 5, 4, 3, 2, 1, 1].

En resumen, al usar el argumento reverse=True con las funciones sorted() y sort(), podemos ordenar una lista en sentido descendente en lugar del orden ascendente predeterminado.

Búsqueda Lineal

La búsqueda lineal recorre uno a uno los elementos de una lista para encontrar un valor. Es sencilla pero ineficiente en listas muy grandes.

Ejemplo en Python:

```
lista = [3, 5, 2, 8, 6]
buscado = 8

for i in range(len(lista)):
    if lista[i] == buscado:
        print("Elemento encontrado en la posición", i)
```

EJERCICIO PARA REFLEXIONAR

Supongamos que tenés una lista de 20.000 productos. ¿Creés que usar búsqueda lineal es lo más adecuado? Argumentá tu respuesta.

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para buscar en listas ordenadas. Divide repetidamente la lista a la mitad y compara el valor deseado con el elemento central. Aquí tienes un ejemplo de implementación en Python:

```python
def busqueda_binaria(lista, elemento):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == elemento:
            return medio
        elif lista[medio] < elemento:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

# Ejemplo de uso
numeros = [1, 3, 5, 7, 9, 11, 13, 15]
elemento_buscado = 7
indice = busqueda_binaria(numeros, elemento_buscado)
print(indice)  # Salida: 3
```

En este ejemplo, la función recibe una lista ordenada y un elemento a buscar. Utiliza una estrategia de dividir y conquistar para reducir el espacio de búsqueda a la mitad en cada iteración. Compara el elemento deseado con el elemento central y ajusta los límites según corresponda. Si encuentra una

coincidencia, devuelve el índice correspondiente. Si no se encuentra el elemento, se devuelve -1.

ACTIVIDAD CON INSTANCIA DE IA:

Usá ChatGPT para que te genere una lista de 10 nombres ordenados alfabéticamente y pedile que te muestre paso a paso la búsqueda binaria del nombre "Lucia".

Prompt sugerido: "Generame una lista ordenada de 10 nombres y mostrá paso a paso la búsqueda binaria del nombre 'Lucia'."

Introducción a la Programación Orientada a Objetos (POO)

La POO es un paradigma de programación que permite modelar conceptos del mundo real mediante "clases" y "objetos". Los cuatro pilares fundamentales de la POO son:

1. **Abstracción**: Representar entidades del mundo real mediante clases.
2. **Encapsulamiento**: Ocultar los detalles internos del funcionamiento de un objeto, exponiendo solo lo necesario.

3. **Herencia**: Permite que una clase herede atributos y métodos de otra.
4. **Polimorfismo**: Posibilidad de que métodos con el mismo nombre se comporten de manera diferente según el contexto o clase.

Componentes básicos de una clase:

- **Atributos**: Representan las características de un objeto.
- **Métodos**: Son las acciones o comportamientos que puede realizar.
- **Constructor `__init__()`**: Se ejecuta al crear un objeto, inicializando sus atributos.

Ejemplo simple:

```python
class Perro:
    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza

    def ladrar(self):
        print(f"{self.nombre}, un {self.raza}, dice: Guau!")

mi_perro = Perro("Firulais", "Labrador")
mi_perro.ladrar()
```

Para reforzar lo aprendido sobre Programación Orientada a Objetos, mirá este video:

https://www.youtube.com/watch?v=knnhFa75xXU

ACTIVIDAD PARA APLICAR CON IA:

Pedile a ChatGPT:

"Mostrame una clase en Python que represente a un libro y tenga atributos título, autor y año, con un método para mostrar esa información."

Luego, agregale un método `es_antiguo()` que devuelva `True` si el libro fue publicado antes del año 2000.

INSTANCIA DE AUTOEVALUACIÓN

- **Cuestionario Moodle:** Preguntas de opción múltiple y Verdadero o Falso sobre ordenamiento y tipos de búsqueda.
    - Criterios:
        - Interpreta código Python.
        - Comprende la relación entre ordenamiento y búsqueda.

INSTANCIA DE FORO

- **Participación en Foro:** Video explicativo sobre una estructura de programación elegida por el alumno.
    - Criterios:
        - Interpreta código Python.
        - Comprende la estructura de programación según el caso a resolver.

ACTIVIDAD

a. Implementar la búsqueda lineal en una lista de números desordenados.

b. Aplicar la búsqueda binaria en una lista ordenada de palabras.

3. CONCLUSIÓN

Búsqueda Lineal

- Es sencillo de implementar y funciona en cualquier tipo de lista, sin necesidad de que esté ordenada.
- Sin embargo, su eficiencia es lineal y puede ser lenta en listas grandes.
- Es útil cuando no se tiene información adicional sobre la lista o cuando la lista es pequeña.

Búsqueda Binaria

- Requiere que la lista esté ordenada, pero ofrece una eficiencia logarítmica, lo que significa que puede encontrar el elemento en menos iteraciones.
- Es más eficiente que la búsqueda lineal en listas grandes.
- Es especialmente útil cuando la lista está ordenada y se necesita una búsqueda rápida.

En general, la elección del algoritmo de búsqueda depende del contexto y los requisitos específicos del problema.

Si se tiene una lista ordenada y se necesita una búsqueda eficiente, la búsqueda binaria es una buena opción. Si no se tiene información adicional sobre la lista o si la lista es pequeña, la búsqueda lineal puede ser suficiente.
La búsqueda de salto y la búsqueda hash son opciones intermedias que pueden ser útiles en determinados escenarios.

Es importante considerar el tamaño de los datos, la frecuencia de las búsquedas, la necesidad de ordenamiento y otros factores al seleccionar el algoritmo de búsqueda adecuado.

Recuerda que cada algoritmo tiene sus ventajas y desventajas, y es importante evaluar el compromiso entre eficiencia y complejidad para encontrar la mejor solución en cada caso.

BIBLIOGRAFÍA OBLIGATORIA:

- FUNDAMENTOS DE PROGRAMACIÓN. Algoritmos, estructura de datos y objetos-Cuarta edición - Luis Joyanes Aguilar- McGrawHill - 2008 - ISBN 978-84-481-6111-8

BIBLIOGRAFÍA SUGERIDA DE LA UNIDAD:

- "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- "Algorithms, Part I" de Robert Sedgewick y Kevin Wayne.
- "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, y Michael H. Goldwasser.
- "Algorithms, Part II" de Robert Sedgewick y Kevin Wayne.
- "Introduction to the Design and Analysis of Algorithms" by Anany Levitin.