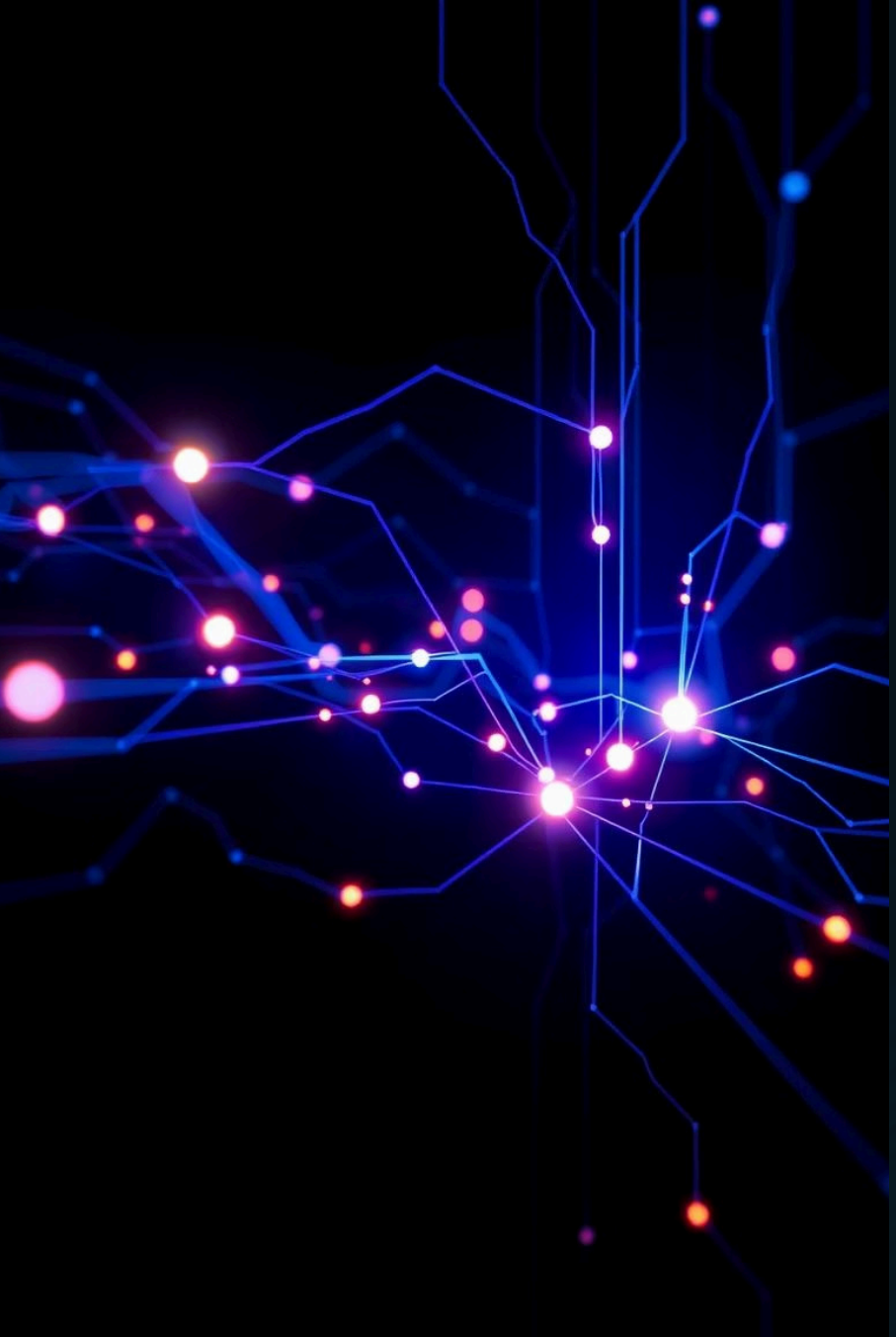

Análisis de Algoritmos: Eficiencia y Optimización

Esta presentación explora cómo el análisis de algoritmos nos ayuda a entender la rapidez y eficiencia de un programa. En Python, buscamos que el código no solo funcione correctamente, sino que también sea rápido y utilice poca memoria.

Un algoritmo es una serie de pasos para una tarea. Analizarlo implica medir el tiempo y la memoria que necesita, especialmente con grandes volúmenes de datos. Compararemos diferentes enfoques para elegir la mejor solución.

Introducción al Análisis de Algoritmos

Rapidez y Eficiencia

El análisis de algoritmos determina qué tan rápido y eficiente es un programa para resolver un problema, optimizando su rendimiento.

Uso de Recursos

Se evalúa cuánto tiempo tarda un algoritmo y cuánta memoria consume, especialmente con grandes cantidades de datos.

Implementación en Python

Python facilita la creación y comparación de algoritmos para elegir la solución óptima.

Marco Teórico: Conceptos Clave

¿Qué es un Algoritmo?

Un algoritmo es una secuencia de pasos para resolver un problema o realizar una tarea, representable con diagramas, instrucciones o código.

Análisis de Algoritmos

Estudia la velocidad y el uso de memoria de un algoritmo para asegurar su eficiencia con grandes volúmenes de datos.

Complejidad Temporal y Espacial

La complejidad temporal mide cómo el tiempo de ejecución varía con la cantidad de datos, mientras que la espacial mide el uso de memoria. Se usa la notación Big O.

Importancia

Es crucial para elegir la mejor solución, crear programas rápidos y eficientes, y entender su comportamiento con grandes datos.

Caso Práctico: Comparación de Algoritmos

Para ilustrar el análisis de algoritmos, comparamos dos funciones en Python: una que utiliza un bucle para sumar números y otra que aplica la fórmula de Gauss. El objetivo es medir y contrastar sus tiempos de ejecución.

Utilizamos el módulo **time** de Python para obtener mediciones precisas del rendimiento de cada algoritmo. Los resultados de esta ejecución se presentan a continuación, mostrando claramente las diferencias en eficiencia entre ambos enfoques.

```python
# --- Importacion de libreria ---
import time
# --- Algoritmos a comparar ---

def suma_bucle(n):
    resultado = 0
    for i in range(1, n + 1):
        resultado += i
    return resultado

def suma_gauss(n):
    return n * (n + 1) // 2


# --- Función para medir tiempos ---

def medir_tiempo(funcion, n, repeticiones=2):
    total = 0
    for _ in range(repeticiones):
        inicio = time.perf_counter()
        funcion(n)
        fin = time.perf_counter()
        total += (fin - inicio) * 1000  # milisegundos
    return total / repeticiones

# --- Tamaños de entrada (casos desde pequeños a extremos) ---

valores_n = [1, 10, 100, 1000, 10_000, 100_000, 1_000_000, 10_00

# --- Resultados de tiempos ---

resultados_bucle = []
resultados_gauss = []
resultados_cuadratico = []

print("n\tBucle(ms)\tGauss(ms)")
for n in valores_n:
    tiempo_bucle = medir_tiempo(suma_bucle, n)
    tiempo_gauss = medir_tiempo(suma_gauss, n)
    resultados_bucle.append(tiempo_bucle)
    resultados_gauss.append(tiempo_
    print(f"{n}\t{tiempo_bucle:.5f}
```

Metodología de Comparación

Definir la Comparación

Establecer claramente qué se va a comparar, por ejemplo, el tiempo de ejecución de dos funciones.

Escribir el Algoritmo

Desarrollar el código en Python que resuelve el problema planteado.

Medir el Rendimiento

Utilizar herramientas específicas para medir el tiempo de ejecución del algoritmo.

Probar con Diferentes Tamaños de Datos

Ejecutar el algoritmo con conjuntos de datos pequeños y grandes para observar cómo varía el tiempo.

Analizar Resultados

Examinar los tiempos obtenidos y determinar cuál algoritmo es más eficiente y por qué.

Resultados Obtenidos: Comparación de Sumas

Suma con Bucle

Este método tarda más tiempo a medida que el número de elementos aumenta. Es rápido para números pequeños, pero se vuelve lento para grandes cantidades de datos debido a la suma iterativa.

Suma con Fórmula de Gauss

Este algoritmo es extremadamente rápido, independientemente del tamaño de los números. Utiliza una fórmula matemática que calcula la suma directamente, sin necesidad de iteraciones repetitivas.

Conclusiones Clave

Fórmula de Gauss

Es la opción más eficiente y rápida para sumar números, superando significativamente a los métodos iterativos.

Bucle para Datos Pequeños

Los bucles son adecuados para conjuntos de datos pequeños, pero su rendimiento disminuye drásticamente con grandes volúmenes.

Evitar Bucles Anidados

Los algoritmos con bucles anidados (cuadráticos) son muy lentos y no se recomiendan para grandes cantidades de datos debido a su alta complejidad.

Recursos Adicionales y Próximos Pasos

Para profundizar en el análisis de algoritmos y la optimización, recomendamos los siguientes recursos:

- **Big-O Cheat Sheet:** Una referencia visual y resumida de las complejidades algorítmicas más comunes.
- **GeeksforGeeks – Analysis of Algorithms:** Excelente para repasar conceptos con ejemplos en pseudocódigo y Python.
- **Python time documentation:** Documentación oficial para medir tiempos con precisión.
- **Apuntes de cátedra:** Material de Programación I, Tecnicatura Universitaria en Programación, UTN - 2025.

Continuaremos explorando técnicas avanzadas de optimización y aplicando estos principios en futuros proyectos de programación.