Trabajo Integrador - Análisis de Algoritmos en el Cálculo de Fibonacci

Alumnos:

Andres Francisco Muñoz

Juan Marcelo Barbero

Comisión N°25 - Tecnicatura Universitaria en Programación a Distancia - UTN

Materia: Programación 1

Profesor: Bruselario, Sebastian

Tutor: Carbonari, Veronica

Fecha de Entrega: 09 de Junio de 2025

Introducción

El análisis de algoritmos es una práctica fundamental en la informática que nos permite medir y predecir el rendimiento de un programa. Entender cómo la elección de un algoritmo impacta el tiempo de ejecución y el consumo de memoria es crucial para construir software escalable y eficiente.

En este trabajo, se analiza esta idea a través de un problema clásico: el cálculo de la sucesión de Fibonacci. Se analizarán dos enfoques muy diferentes: uno recursivo simple y otro iterativo optimizado. A través de la implementación y medición de ambos, se demostrará cuantitativamente cómo una mejor aproximación algorítmica puede llevar a mejoras de rendimiento exponenciales y significativas.

Marco Teórico

El análisis de algoritmos es el estudio teórico y práctico del rendimiento de los algoritmos. Su objetivo principal es optimizar dos recursos computacionales clave:

- **Tiempo de ejecución (Complejidad Temporal):** Cuánto tarda un algoritmo en completarse en función del tamaño de la entrada.
- **Uso de memoria (Complejidad Espacial):** Cuánta memoria adicional necesita un algoritmo para ejecutarse.

Conceptos Clave:

- **Notación Big-O:** Es el lenguaje estándar para describir la complejidad de un algoritmo. Se enfoca en el "peor caso" y describe cómo crece el tiempo de ejecución a medida que aumenta el tamaño de la entrada (por ejemplo: O(1) - constante, O(n) - lineal, O(n2) - cuadrático, O(2n) - exponencial).

- **Tiempo de ejecución real:** Es el tiempo medido en segundos que un programa tarda en ejecutarse. Se puede medir en Python con librerías como time.
- **Complejidad espacial:** Se refiere a la cantidad de memoria adicional (variables, estructuras de datos) que el algoritmo utiliza, sin contar la memoria de los datos de entrada.

Caso Práctico

Se analiza y compara dos algoritmos para calcular el n-ésimo número de la sucesión de Fibonacci.

Código principal propuesto (analisis_fibonacci.py)

```
import time


def fibonacci_recursivo(num):
    if num <= 1:
        return num
    return fibonacci_recursivo(num-1) + fibonacci_recursivo(num-2)


def fibonacci_iterativo(num):
    if num <= 1:
        return num
    a, b = 0, 1
    for _ in range(num - 1):
        a, b = b, a + b
    return b


def medir_tiempo(funcion, num):
    """Mide el tiempo de ejecución de una función dada."""
    inicio = time.time()
```

```
    resultado = funcion(num)

    fin = time.time()

    return resultado, fin - inicio


if __name__ == "__main__":


    num_peq = 40


    print(f"Calculando Fibonacci para n = {num_peq}...\n")


    resultado1, tiempo1 = medir_tiempo(fibonacci_recursivo,num_peq)

    print(f"Fibonacci Recursivo: Resultado={resultado1},
Tiempo={tiempo1:.6f} segundos")


    resultado2, tiempo2 = medir_tiempo(fibonacci_iterativo,num_peq)

    print(f"Fibonacci Iterativo: Resultado={resultado2},
Tiempo={tiempo2:.6f} segundos")


    n_grande = 1000000

    print(f"\nCalculando Fibonacci para n = {n_grande} con el método
iterativo...")

    _,tiempo_grande = medir_tiempo(fibonacci_iterativo, n_grande)

    print(f"Fibonacci Iterativo (n={n_grande}):
Tiempo={tiempo_grande:.6f} segundos")
```

Resultados Obtenidos

- Ambos algoritmos calculan correctamente el n-ésimo número de Fibonacci.
- El tiempo de ejecución del algoritmo **recursivo crece de forma exponencial**. Para un n cercano a 40, ya tarda varios segundos, mientras que para n=50

sería impracticable.
- El algoritmo **iterativo es drásticamente más rápido**. Su tiempo de ejecución crece linealmente con n y puede calcular términos muy altos (como n=1,000,000) en segundos.
- Los resultados prácticos confirman de manera contundente el análisis teórico de la notación Big-O.

Conclusiones

La elección del algoritmo es determinante para el rendimiento de una aplicación. Como se demostró con el cálculo de Fibonacci, una solución recursiva puede ser computacionalmente inviable, mientras que una solución iterativa, aunque quizás menos directa al principio, ofrece un rendimiento y una escalabilidad muy superiores. Antes de implementar una solución, especialmente una recursiva, es importante analizar si realiza cálculos redundantes. En esos casos, técnicas como la **iteración** o la **memoización** (un paso intermedio que guarda resultados previos) son alternativas mucho más eficientes. Este proyecto subraya la importancia de no solo resolver un problema, sino de resolverlo eficientemente.

Bibliografía

- Fundamentals of Data Structures in C++ E. Horowitz, S. Sahni, D. Mehta Computer Science Press, 1995
- Python Time Documentacion Oficial https://docs.python.org/3/library/time.html
- Notacion Big-O https://msmk.university/big-o-notation/