Tecnicatura universitaria en programación a distancia

Materia: **Programación I**

Alumnos: **Matías Facundo Herrera, Andres Oscar Bonelli**

Profesoras: Julieta Trapé / Prof. Cinthia Rigoni

Fecha de entrega: 09/06/2025

Título: Análisis de algoritmos - eficiencia y optimización

Índice

1. Introducción

2. Marco Teórico

3. Caso Práctico

4. Metodología Utilizada

5. Resultados Obtenidos

6. Conclusiones

7. Bibliografía

8. Anexos

1 - Introducción

El análisis de algoritmos y su eficiencia es una de las bases fundamentales de la programación y la informática en general. Se eligió este tema porque permite comprender no sólo cómo funcionan los programas, sino qué tan bien funcionan. En un mundo donde los datos crecen exponencialmente y la velocidad de respuesta es crucial, la eficiencia del código es una necesidad. Abordar este tema es clave para quienes buscan no solo resolver problemas, sino hacerlo de manera inteligente y escalable.

¿Qué importancia tiene en la programación?

En programación, escribir un código que funcione no siempre es suficiente. También debe ser eficiente, optimizado y escalable. La eficiencia de un algoritmo impacta directamente en el rendimiento del software: puede ser la diferencia entre una app usable y una inutilizable. Entender la eficiencia permite a los programadores: elegir la mejor solución entre varias posibles, ahorrar recursos del sistema (tiempo de ejecución, memoria, energía), garantizar que una aplicación pueda manejar grandes volúmenes de datos sin colapsar, prevenir cuellos de botella y mejorar la experiencia del usuario.

2 - Marco teórico

Complejidad entre el uso de memoria y la eficiencia del tiempo de ejecución

El tiempo de ejecución y el uso de memoria son dos métricas clave para evaluar la eficiencia de un algoritmo. El tiempo de ejecución se refiere al tiempo que tarda un algoritmo en completarse, mientras que el uso de memoria se refiere a la cantidad de espacio que el algoritmo requiere en la memoria del ordenador.

Comprender las compensaciones entre el uso de memoria y la eficiencia del tiempo de ejecución es crucial en la optimización de algoritmos. Piense en ello como un balancín: por un lado, está el uso de memoria y, por el otro, la eficiencia en tiempo de ejecución. Si se centra demasiado en reducir el uso de memoria, es posible que el algoritmo se ejecute más lento porque obtiene constantemente datos que podría haber almacenado. Por el contrario, si prioriza la velocidad y almacena todo en la memoria, es posible que se quede sin espacio o use más memoria de la necesaria. El objetivo es encontrar un término medio en el que el algoritmo sea rápido y eficiente en cuanto a memoria.

La complejidad del espacio se refiere a la cantidad de memoria que necesita un algoritmo en relación con el tamaño de los datos de entrada. Es importante entender cómo se escala el algoritmo. Por ejemplo, un algoritmo con una complejidad de espacio de O(n) significa que la memoria requerida crece linealmente con el tamaño de la entrada. A veces, el uso de espacio adicional puede acelerar significativamente un algoritmo (Esto se conoce como una compensación espacio-temporal). Sin embargo, siempre debe preguntarse si hay una manera de reducir el uso de espacio sin un aumento sustancial en el tiempo de ejecución.

La complejidad del tiempo es una medida de la cantidad de tiempo que tarda un algoritmo en completarse a medida que crece el tamaño de la entrada. Por lo general, se prefieren los algoritmos con menor complejidad de tiempo porque pueden manejar entradas más grandes de manera más eficiente. Sin embargo, lograr una menor complejidad de tiempo a veces significa usar más memoria. Para equilibrar esto, puede buscar optimizaciones que

reduzcan el número de operaciones o mejoren la forma en que se accede a los datos y se procesan, sin aumentar necesariamente la superficie de memoria.

Dependencia del tamaño de entrada

El tiempo de ejecución de un algoritmo a menudo depende del tamaño de la entrada. En algunos casos, el tiempo de ejecución puede crecer linealmente con el tamaño de la entrada, mientras que en otros casos puede crecer exponencialmente, cuadrática, o con otras relaciones matemáticas.

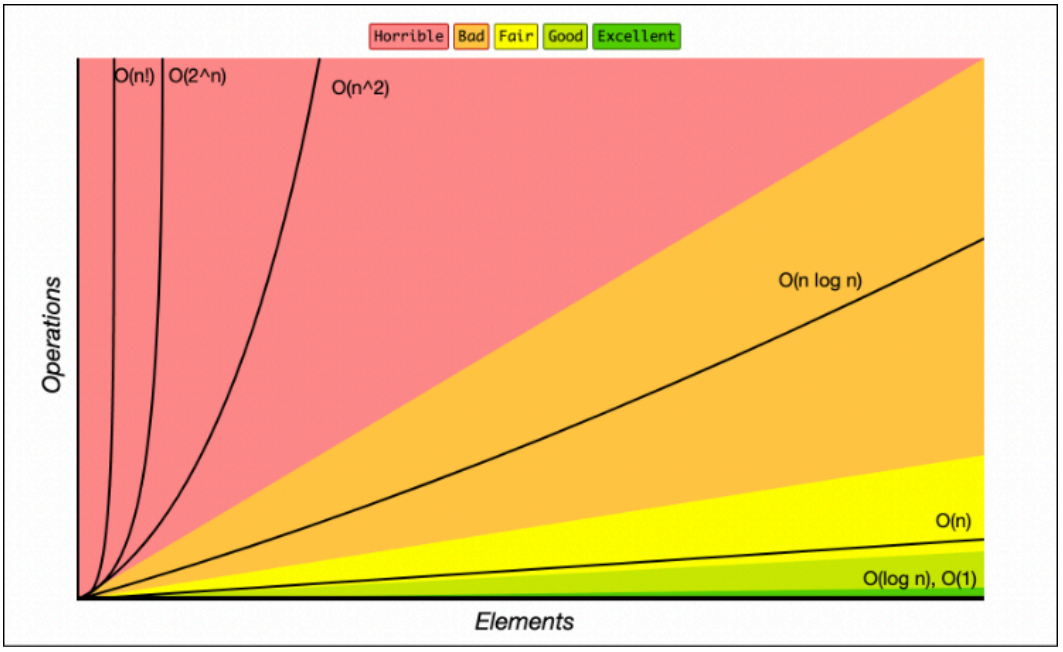
Se utiliza la notación Big O para describir el crecimiento del tiempo de ejecución en función del tamaño de la entrada. Por ejemplo, un algoritmo de complejidad O(n) tarda un tiempo proporcional al tamaño de la entrada, mientras que un algoritmo de complejidad O(n^2) tarda un tiempo proporcional al cuadrado del tamaño de la entrada.

Además del tamaño de la entrada, el tiempo de ejecución también puede verse afectado por factores externos como la velocidad del procesador, la cantidad de memoria disponible y la eficiencia del código fuente.

Notación Big O

La notación Big O es un concepto fundamental en informática que se utiliza para analizar y describir el rendimiento de los algoritmos. Esto muestra la eficiencia temporal del algoritmo en términos de crecimiento relativo del tamaño de entrada.

La base de la notación Big O es proporcionar una forma sistemática y estandarizada de expresar la eficiencia temporal de un algoritmo en función del tamaño de entrada. Es un lenguaje matemático que simplifica la complejidad de los algoritmos, permitiendo a los desarrolladores y analistas de datos evaluar y comparar de forma clara y concisa el rendimiento de diferentes algoritmos. La notación Big O se centra en el peor de los casos y expresa el tiempo de ejecución asintótico a medida que la entrada aumenta hasta el infinito. La letra “O” representa el orden de magnitud y los términos entre paréntesis representan la función de complejidad del tiempo. Desde O(1) para algoritmos de tiempo constante hasta O(n!) para algoritmos de complejidad factorial, estos fundamentos son esenciales para tomar decisiones informadas al diseñar y seleccionar algoritmos en el desarrollo de software. Esto ayuda a los programadores a identificar y comprender completamente el peor de los casos y el tiempo de ejecución o la memoria requerida de un algoritmo.



Representación de la complejidad Big O

Complejidad espacial en resumen

La complejidad espacial de un algoritmo se refiere a la cantidad de memoria que un algoritmo requiere para funcionar, medida en función del tamaño de la entrada. Es una métrica crucial para evaluar la eficiencia de un algoritmo, especialmente en sistemas con limitaciones de memoria.

Esta medida considera tanto el espacio que ocupa el código del algoritmo como el espacio utilizado para los datos de entrada y los datos auxiliares que genera durante la ejecución.

La complejidad espacial se suele expresar utilizando la notación Big O, similar a la complejidad temporal, para describir cómo el uso de memoria aumenta con el tamaño de la entrada.

En cuanto a su relación con la complejidad temporal, en algunos casos se puede optimizar la complejidad espacial a expensas de la complejidad temporal, o viceversa, como se ha mencionado previamente.

Complejidad temporal en resumen

La complejidad temporal de un algoritmo se refiere al tiempo de computación que toma para ejecutarse en función del tamaño de la entrada.

Acá la notación Big O se utiliza para describir el crecimiento asintótico del tiempo de ejecución de un algoritmo. Por ejemplo, O(n) representa un crecimiento lineal, $O(n^2)$ un crecimiento cuadrático y O(log n) un crecimiento logarítmico.

3 - Caso práctico

Para ejemplificar el análisis de eficiencia y optimización de algoritmos, se utilizó el cálculo de la serie de Fibonacci, un problema que permite ilustrar claramente diferencias de complejidad temporal y espacial entre distintas implementaciones.

Descripción matemática del problema:

Sea $n \in \mathbb{N}$, $n \geq 0$, la función de Fibonacci se define como:

$$F(n) = \begin{cases} 0 & \text{si } n=0 \\ 1 & \text{si } n=1 \\ F(n-1)+F(n-2) & \text{si } n>1 \end{cases}$$

La entrada del algoritmo es un número entero *n*, y la salida es el valor F(*n*), es decir, el número de Fibonacci correspondiente a la posición *n*.

Se desarrollaron cuatro versiones distintas del algoritmo en Python:

- Versión recursiva pura: Genera números de Fibonacci usando recursión. La complejidad de tiempo es exponencial O(2^n) debido a la recursión repetida. La complejidad espacial aumenta también exponencialmente debido a la pila de llamadas.

- Versión con *memorización*: Esta función usa una forma de cacheo para optimizar la recursión. Almacena los números que ya fueron computados y su posición correspondiente en un diccionario para evitar así realizar cálculos redundantes. La complejidad temporal es

0(n), ya que cada número de Fibonacci es calculado sólo una vez, reduciendo significativamente el tiempo de ejecución. Sin embargo, también usa recursión, con lo cual se puede llegar a alcanzar el límite de pila de llamadas en números muy altos.

- Versión con tabulación: Esta versión implementa una solución basada en programación dinámica. Se construye la secuencia de forma iterativa en lugar de recursiva, desde el primer número hasta el número pasado como parámetro, almacenando los resultados en una lista. La complejidad temporal es lineal 0(n) ya que no usamos recursión, y la complejidad espacial también es 0(n) ya que vamos almacenando la lista completa.

- Versión iterativa optimizada: Esta versión es una optimización de la anterior, que calcula el número sin necesidad de cacheo. Tiene complejidad temporal 0(n) al igual que los anteriores, pero complejidad espacial constante 0(1) ya que solo se almacenan los últimos 2 números computados en lugar de la secuencia completa, reduciendo también al mínimo indispensable el uso de memoria.

Cada versión se ejecutó para calcular el número de Fibonacci en la posición 30. Se midió el tiempo de ejecución de cada implementación utilizando la función time.time() antes y después de su ejecución, para luego comparar los resultados.

4 - Metodología utilizada

Diseño de las funciones: Cada versión del algoritmo fue implementada respetando su paradigma correspondiente

Medición del tiempo: Se utilizó una función medir_tiempo() que encapsula la ejecución del algoritmo y calcula la duración utilizando la diferencia entre los *timestamps* de inicio y fin.

Misma entrada de prueba: Todas las funciones calcularon el mismo número de la serie (posición 30), para mantener consistencia en la comparación.

Entorno controlado: Las pruebas se ejecutaron en el mismo entorno de ejecución, con las mismas condiciones de hardware/software, para evitar variaciones externas en el tiempo de ejecución.

Esta metodología permite observar claramente cómo diferentes enfoques en la implementación de un algoritmo pueden tener un impacto significativo en la eficiencia.

5 - Resultados obtenidos

Fibonacci recursivo: Tiempo=0.08366609 segundos.
Fibonacci con memo: Tiempo=0.00002289 segundos.
Fibonacci con tabulación: Tiempo=0.00001001 segundos.
Fibonacci optimizado: Tiempo=0.00000167 segundos.

A partir de los resultados obtenidos hemos observado que la diferencia más significativa entre las distintas implementaciones fue el tiempo de ejecución, donde podemos apreciar como la versión iterativa optimizada fue la más eficiente en cuanto a performance temporal, ocupando apenas una fracción de milisegundo para calcular el resultado. Las versiones iterativa y recursiva con cacheo resultaron entre 10 a 20 veces más lentas, mientras que la version recursiva sin ningun tipo de optimizacion se encontro en varios ordenes de magnitud más lento (más de cien mil veces más lento)
Estos resultados confirman que la elección del algoritmo tiene un impacto considerable en el rendimiento, especialmente cuando se manejan entradas grandes.

6 - Conclusiones

Este trabajo permitió comprobar cómo la eficiencia de un algoritmo depende tanto del enfoque de resolución como del tipo de estructura utilizada. Aunque todos los métodos calculan el mismo valor, las diferencias en complejidad temporal y espacial son notables.

Los algoritmos recursivos simples, aunque fáciles de comprender a simple vista, no siempre son eficientes para problemas donde existen llamadas repetidas. En cambio, las técnicas de cacheo mejoran significativamente el rendimiento. La versión optimizada iterativa se destacó por su bajo consumo de recursos y velocidad, lo que la convierte en la opción más adecuada en la mayoría de los escenarios.

Este análisis refuerza la importancia de comprender no sólo cómo resolver un problema, sino también de buscar la forma más eficiente, teniendo en cuenta los recursos del sistema y la capacidad de escalabilidad del software.

7 - Bibliografía

Notación Big O, The university of Science & Technology, 2024, msmk
url: https://msmk.university/big-o-notation/

Algoritmos de eficiencia, The university of Science & Technology, 2024, msmk
url: https://msmk.university/algorithm-efficiency/

¿Cómo se pueden equilibrar las compensaciones entre el uso de memoria y la eficiencia del tiempo?, LinkedIn, 2025, Usuarios de ingenieria/algoritmos
url:
https://www.linkedin.com/advice/0/how-can-you-balance-trade-offs-between-memory-usage-ys2sf?lang=es&originalSubdomain=es

Guía: Notación Big O - Gráfico de complejidad de tiempo, freeCodeCamp, 2024, Autor: Joel Olawanle
url:
https://www.freecodecamp.org/espanol/news/hoja-de-trucos-big-o/#:~:text=La%20notaci%C3%B3n%20Big%20O%20es,el%20tama%C3%B1o%20de%20su%20entrada.

8 - Anexos

Link al repositorio: https://github.com/andresbonelli/UTN-TUPaD-Programacion_1-Trabajo-Integrador

Link al vídeo: https://www.youtube.com/watch?v=o6Weov3TFaY