Análisis de la eficiencia algorítmica en Python

Alumnos:

Lorenzo Pattini – lorenzopattini78@gmail.com
Uriel Palma – palmaurieldev@gmail.com

Materia: Programación I

Profesor: Sebastián Bruselario

Tutora: Virginia Cimino

Fecha de Entrega: 20-06-2025

Introducción

En este trabajo planteamos como objetivo el análisis de dos algoritmos y realizamos una comparativa entre los tiempos que requieren ambos para llevar a cabo su función (Encontrar el número de menor valor dentro de una lista).
Hicimos esta investigación para poder analizar la eficiencia de un algoritmo, esta misma se refiere a cómo se utilizan los recursos computacionales (tiempo y memoria) para resolver un problema. El análisis de algoritmos nos permite comparar diferentes soluciones y elegir la más óptima según el contexto. Comprender cómo se comportan frente al aumento de datos es esencial para desarrollar aplicaciones eficientes y escalables.

En esta investigación se estudian los conceptos de Eficiencia Temporal (Tiempo) y Espacial (Memoria), se implementan ejemplos prácticos y se presentan las conclusiones.

Marco Teórico

¿Qué es el Análisis de Algoritmos?

El análisis de algoritmos es una herramienta cuyo objetivo es evaluar el diseño de un algoritmo, permite comparar entre sí algoritmos que cumplan con la resolución del mismo objetivo sin necesidad de desarrollarlos. Generalmente se evalúa su rendimiento en términos de eficiencia (Tiempo y Espacio) y se busca determinar cómo se comporta a medida que el tamaño de sus entradas crece.

De forma teórica, el análisis de algoritmos estudia los recursos computacionales que requiere cualquier programa para ejecutarse, básicamente su eficiencia (tiempo de CPU, Uso de memoria, ancho de banda). También existen otros factores igual de relevantes como su funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad y hasta el costo de programación o desarrollo.

El análisis de algoritmos nos permite:

Evaluar la eficiencia: Determinar la cantidad de recursos que necesita el algoritmo para su ejecución.

Comparar algoritmos: Ayuda a elegir el algoritmo más eficiente en base a los resultados de la comparativa.

Entender el comportamiento: Permite predecir cómo se comportará un algoritmo con diferentes tamaños de entradas.

Conceptos Clave:

- Complejidad temporal: Mide el tiempo de ejecución de un algoritmo en función del tamaño de entrada.
- Complejidad Espacial: Mide la cantidad de memoria que utiliza un algoritmo en función del tamaño de la entrada.
- Notación Big O: Se utiliza para expresar la complejidad de un algoritmo de forma asintótica (a medida que el tamaño de la entrada crece)

Tipos de Análisis(Teórico y Empírico):

El análisis de algoritmos tiene dos enfoques complementarios, estos son:

Análisis teórico: El análisis teórico consiste en estimar el comportamiento de un algoritmo sin necesidad de realizar su ejecución, usando herramientas como la notación Big O.

Análisis empírico: El análisis empírico es el estudio del rendimiento en el caso real del algoritmo, evaluando tiempos de ejecución sobre datos concretos y analizando tiempos y costos.

Tipos de Análisis (Peor, Mejor y Promedio):

- **Peor caso (usualmente): T(n) =** Tiempo máximo necesario para un problema de tamaño n. Se refiere a la situación inicial de los datos que genera una ejecución del algoritmo con una complejidad computacional mayor.(El más utilizado)
- **Mejor caso (engañoso): T(n) =** Tiempo menor para un problema cualquiera de tamaño n. Se refiere a la situación inicial de los datos que genera una ejecución del algoritmo con una menor complejidad computacional.
- **Caso medio (a veces): T(n) =** Tiempo esperado para un problema cualquiera de tamaño n. La situación inicial de los datos no sigue ningún patrón preestablecido que aporte ventajas o desventajas. Se puede considerar, por tanto, la situación típica de ejecución del algoritmo.

¿Por qué se utiliza con mayor frecuencia el análisis del peor caso?

Caso promedio: El análisis de caso promedio no es fácil de realizar en la mayoría de los casos prácticos y rara vez se realiza. En el análisis de caso promedio,

debemos considerar cada entrada, su frecuencia y el tiempo que tarda, lo cual puede no ser posible en muchos escenarios.

Mejor caso: El análisis del mejor caso se considera falso. Garantizar un límite inferior en un algoritmo no proporciona información, ya que, en el peor de los casos, un algoritmo puede tardar años en ejecutarse.

Peor caso: es más fácil que el caso promedio y proporciona un límite superior que constituye información útil para analizar productos de software.

Los criterios utilizados para evaluar programas son varios: Eficiencia, portabilidad, eficacia,robustez, etc. El análisis de complejidad está directamente relacionado con la eficiencia del programa, esta mide el uso de los recursos del dispositivo por un algoritmo mientras que el análisis de complejidad mide el tiempo de cálculo para ejecutar operaciones y el espacio en memoria para contener y manipular el programa más los datos.

En resumen, el objetivo del análisis de complejidad es cuantificar las medidas físicas: tiempo de ejecución y espacio de memoria y comparar distintos algoritmos que resuelven el mismo problema.

En cuanto al tiempo de ejecución, este debe definirse como una función que depende de la entrada, en particular, del tamaño de la misma. El tiempo requerido por un algoritmo expresado como una función del tamaño de la entrada del problema se denomina complejidad en tiempo del algoritmo y se denota T(n). El comportamiento (límite) de la complejidad a medida que crece el tamaño del problema se lo denomina complejidad en tiempo asintótica.

El tiempo de ejecución de un programa depende de factores tales como:

- Los datos de entrada del programa.
- La calidad del código objeto generado por el compilador.
- La naturaleza y rapidez de las instrucciones de máquina.
- La complejidad en tiempo del algoritmo base del programa.

Notación Big O

La notación “Big O” es una técnica común utilizada en el análisis algorítmico para describir la tasa de crecimiento del tiempo o del espacio requerido a medida que el tamaño del problema aumenta en las estructuras de datos. Por ejemplo, un algoritmo con una complejidad de O(n) requerirá aproximadamente n operaciones para un problema de tamaño n. Un algoritmo con una complejidad de O(n^2) requerirá aproximadamente n^2 operaciones para un problema de tamaño n.

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, esto le permite a los desarrolladores y analistas de datos una evaluación y comparación clara y concisa del

rendimiento de diferentes algoritmos. La notación Big O se centra siempre en el peor de los casos y expresa el tiempo de ejecución asintótico a medida que la entrada aumenta hasta el infinito.

¿Qué significa la "O" de Big-O?

El significado de la letra O es importante para comprender el símbolo y aplicarlo en el análisis. En este contexto, se refiere al tamaño o tasa de crecimiento de la función que describe la complejidad temporal del algoritmo a analizar. Cuando usamos la notación Big-O nos centramos en el comportamiento asintótico del algoritmo. La letra O sirve como símbolo clave que resalta la relación entre el tiempo de ejecución y el volumen de datos, lo que permite que los desarrolladores evalúen de forma rápida y eficiente el rendimiento de los algoritmos y tomar decisiones informadas sobre su implementación.

Beneficios de Big O

Big O tiene muchos beneficios al implementarse en las decisiones de los desarrolladores, las principales son la evaluación de la eficiencia de los algoritmos, la comparativa entre algoritmos con un mismo objetivo y la optimización de código y recursos.

Notaciónes Big O más utilizadas:

- O(1) - Notación constante
- O(log (n)) - Notación logarítmica
- O(n) - Notación lineal
- O(n log (n)) - Notación lineal-logarítmica
- O(n^2) - Notación cuadrática
- O(2^n) - Notación exponencial
- O(n!) - Notación factorial

Caso Práctico

Se analiza y se comparan dos algoritmos cuya función es encontrar el número de menor valor dentro de una lista

Código principal:

```
import time
import random


# Generación de listas de tamaños crecientes con números aleatorios


# Se generan listas que crecen desde los 1 mil, 10 mil, 100 mil, 1 millón y 10
millones
# Cada número es un entero aleatorio entre 1 y 1.000.000
```

```
numeros_0 = [random.randint(1, 1000000) for i in range(1000)]
numeros_1 = [random.randint(1, 1000000) for i in range(10000)]
numeros_2 = [random.randint(1, 1000000) for i in range(100000)]
numeros_3 = [random.randint(1, 1000000) for i in range(1000000)]
numeros_4 = [random.randint(1, 1000000) for i in range(10000000)]



#Se guardan todas las listas dentro de otra lista para un fácil procesamiento
numeros_total = [numeros_0, numeros_1, numeros_2, numeros_3,
numeros_4]

# Definición de funciones

def min_manual(lista):

    #Encuentra el valor mínimo recorriendo la lista manualmente.
    #Complejidad temporal: O(n)  |  Complejidad espacial: O(1)

if not lista:
        return None        #Si la lista está vacía retorna un "None"
    minimo = lista[0]     #Se asume que el primer número es el mínimo
    for numero in lista:
        if numero < minimo:
            minimo = numero
    return minimo

def min_auto(lista):

    #Encuentra el valor mínimo usando la función min() de Python.
    #Complejidad temporal: O(n)  |  Complejidad espacial: O(1)

    if not lista:
        return None        #Devuelve None si la lista está vacía
    return min(lista)      #Usa la función interna optimizada de Python
```

```
#Resultados con la función min() (Automática)

print("Resultados de la búsqueda del mínimo automático")

#Se recorre cada lista y se mide el tiempo que tarda "min_auto" en encontrar
el número menor
for i in range(5):
    inicio = time.time()
    minimo_auto = min_auto(numeros_total[i])
    tiempo = time.time() - inicio
    print(f"{10**(i+3)} elementos: {tiempo:.10f} segundos")

#Resultados con la función manual

print("Resultados de la búsqueda del mínimo manual")

#Se recorre cada lista y se mide el tiempo que tarda "min_manual" en
encontrar el número menor
for i in range(5):
    inicio = time.time()
    minimo_manual = min_manual(numeros_total[i])
    tiempo = time.time() - inicio
    print(f"{10**(i+3)} elementos: {tiempo:.10f} segundos")
```

Resultado Aproximado:

Búsqueda del mínimo automático:

Resultados:

1000 0.0000000000

10000 0.0000000000

100000 0.0010347366

1000000 0.0059986115

10000000 0.0670058727

Búsqueda del mínimo manual:

Resultados:

1000 0.0000000000

10000 0.0000000000

100000 0.0010268688

1000000 0.0139691830

10000000 0.1250259876

Metodología Utilizada

Para el análisis de la eficiencia de nuestros algoritmos utilizamos los dos enfoques del análisis algorítmico.

Análisis Teórico:

Realizamos un estudio de la complejidad computacional de ambos algoritmos en tiempo y en espacio, utilizando la notación Big O.

La función min_manual() y la función min auto() tienen complejidad temporal O(n), ya que recorren la totalidad de la lista una vez.

En memoria, ambas tienen complejidad espacial O(1), ya que utilizan una cantidad constante de variables auxiliares.

Análisis Empírico:

Complementamos el enfoque teórico con mediciones reales de rendimiento. Para eso implementamos los dos algoritmos en Python.

Generamos listas de prueba con tamaños crecientes: 1.000, 10.000, 100.000, 1.000.000 y 10.000.000 elementos.

Registramos los tiempos de ejecución de cada función utilizando la función time.time(), que nos permitió calcular el tiempo de ejecución de cada algoritmo.

Comparamos los resultados obtenidos, analizando cómo se comporta cada algoritmo frente al aumento del tamaño de los datos.

Este enfoque nos permite hacer estimaciones de la eficiencia tanto desde la teoría como desde las pruebas prácticas, esto es útil y eficiente para entender cómo elegir algoritmos para su uso real.

Resultados Obtenidos

Al realizar la ejecución de ambos algoritmos sobre las listas crecientes, vemos los siguientes resultados:

Búsqueda del mínimo automático:

Resultados:

1000 0.0000000000

10000 0.0000000000

100000 0.0010347366

1000000 0.0059986115

10000000 0.0670058727

Búsqueda del mínimo manual:

Resultados:

1000 0.0000000000

10000 0.0000000000

100000 0.0010268688

1000000 0.0139691830

10000000 0.1250259876

Podemos interpretar de estos algoritmos que ambos devuelven correctamente el mínimo y reportan el tiempo de ejecución, pero hay diferencias claras en el tiempo de ejecución, sobre todo cuando el tamaño de la lista crece.

Hasta los 10.000 elementos, la diferencia es prácticamente imperceptible pero a partir de los 100.000, la función automática min() toma una ventaja significativa en cuanto al rendimiento y tiempo de ejecución contra la función manual, tanto que llegando a los 10 millones la funcion automatica tarda la mitad de tiempo que la manual.

Este comportamiento coincide con el análisis teórico, ya que las dos funciones tienen la complejidad O(n). Sin embargo podemos ver en el análisis empírico una evidente ventaja de la función min(), esto se debe a que posiblemente la función min() tenga una optimización interna con otro lenguaje de programación como lo puede ser C.

Conclusiones

El análisis de algoritmos es esencial para optimizar programas, especialmente en aplicaciones con grandes volúmenes de datos. Se ve más reflejado en las grandes empresas, donde encontrar la eficiencia en los algoritmos es un criterio de aceptación de suma importancia.

En el desarrollo de software, la eficiencia de los algoritmos no es un lujo, sino una necesidad crítica. A medida que las aplicaciones manejan volúmenes masivos de datos (big data, inteligencia artificial, sistemas distribuidos), elegir un algoritmo ineficiente puede resultar en:

- Tiempos de ejecución prohibitivos (horas en lugar de segundos).
- Consumo excesivo de memoria (colapsando servidores o dispositivos).
- Mala experiencia de usuario (lentitud en aplicaciones web/móviles).

Este trabajo nos permitió entender cómo dos algoritmos que, en el análisis teórico tienen la misma complejidad, pueden mostrar rendimientos diferentes en la práctica.

A través del análisis empírico observamos que la función min() de python, a pesar de recorrer la totalidad de la lista al igual que la función manual, esta tiene un tiempo de ejecución mucho menor gracias a la optimización interna con la que cuenta.

Como aprendizaje y reflexión, tras la investigación e implementación de conceptos, podemos afirmar que el análisis de algoritmos no está limitado a la teoría, la experimentación y las pruebas prácticas son muy importantes y fundamentales para tomar decisiones más eficientes en el desarrollo.

Anexos

- Captura de resultados de la ejecución de ambos algoritmos y su tiempo de iteración en la terminal:

```
Resultados de la búsqueda del minimo automático
1000 0.0000000000
10000 0.0000000000
100000 0.0010015965
1000000 0.0062534809
10000000 0.0590045452
////////////////////////////////////////////////
Resultados de la búsqueda del minimo manual
1000 0.0009980202
10000 0.0000000000
100000 0.0010004044
1000000 0.0110981464
10000000 0.1110434532
PS C:\Users\Usuario>
```

Gráfico representando el mínimo automatico

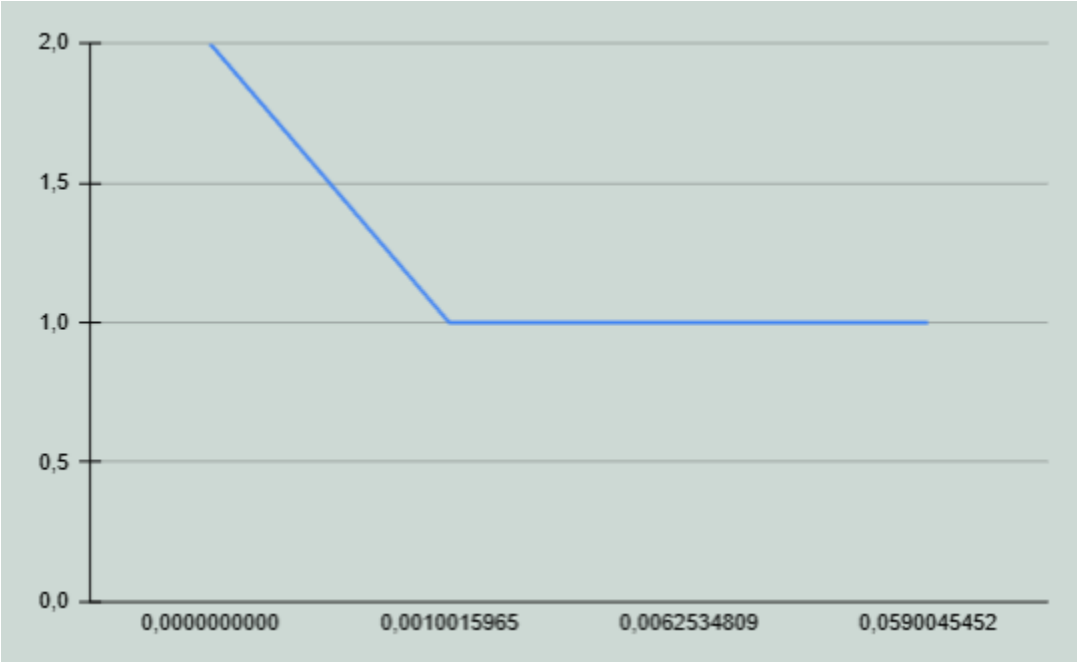
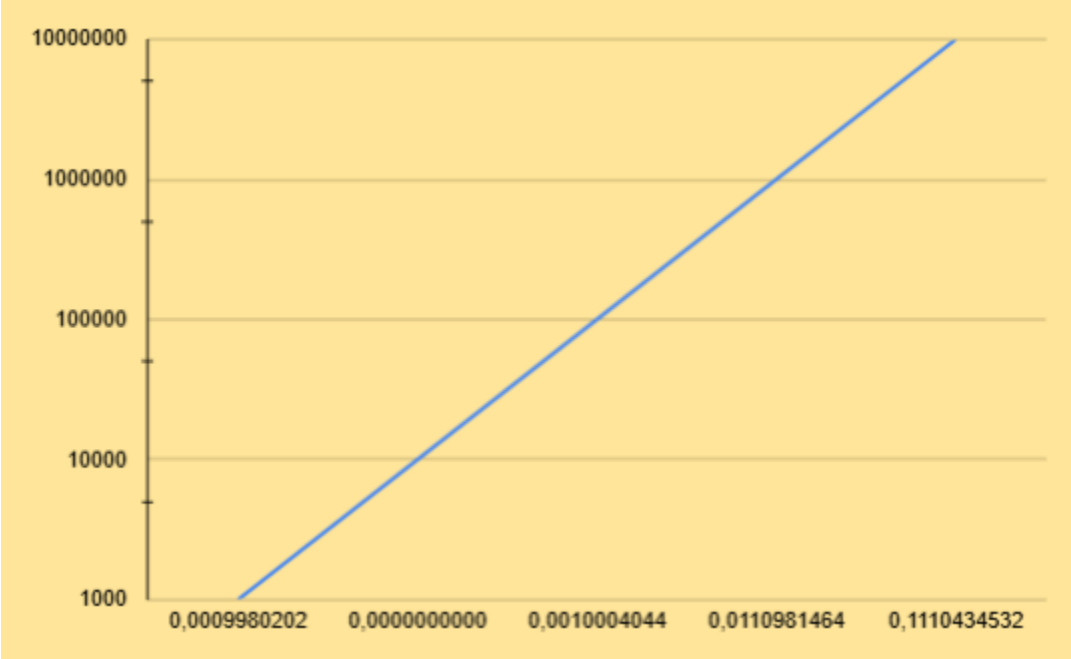


Gráfico representando el mínimo manual



- Repositorio en GitHub: https://github.com/PalmaUriel/Prog1-Integrador-Uriel-Lorenzo

Bibliografía

- Documentación y material extraído de la Unidad: Análisis de algoritmos, TUPAD de la UTN.
- https://www.bigocheatsheet.com
- http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm
- https://bookdown.org/nury_farelo/notas_de_clase_estructuras_de_datos_y_anlisis_de_algoritmos/Notas%20de%20clase.html
- https://msmk.university/big-o-notation/

Video explicativo

URL: https://youtu.be/Z0uC6aWLlfs