Análisis de Algoritmos

- **Alumnos:**
  - **Nahuel Urciuoli Zabala –** Nahuel_zabala@live.com
  - **Martin Rotolo –** martinrotolo97@gmail.com
- **Profesor:**
  - **Bruselario, Sebastián**
- **Tutora:**
  - **Gubiotti, Flor**

- **Fecha de entrega:**
  - **09/06/2025**

CONTENIDO

Introducción

Ya que los algoritmos son la base de la programación pensamos que lograr algoritmos eficientes aportan también mucho peso al código escrito, pero. **¿Qué es un algoritmo?** Un algoritmo es un conjunto de instrucciones bien definidas y ordenadas diseñadas para resolver un problema específico. El análisis de algoritmos evalúa su rendimiento, permitiendo comprender cómo se comportan frente a grandes volúmenes de datos. Para ello, se utilizan dos métricas fundamentales:

Eficiencia temporal: Tiempo de ejecución del algoritmo en función del tamaño de entrada (n).

Eficiencia espacial: Cantidad de memoria utilizada durante su ejecución.

Estas métricas son herramientas indispensables para desarrollar software escalable y optimizado, especialmente en sistemas que manejan datos masivos o requieren alta velocidad de procesamiento.

Este trabajo tiene como objetivo comparar algoritmos de distinta complejidad, evidenciando mediante métricas cuantitativas (tiempo de ejecución y uso de memoria) por qué algunos enfoques son más eficientes que otros

Marco Teórico

Los algoritmos son esenciales en las ciencias de la computación porque proporcionan métodos sistemáticos para resolver problemas complejos de manera eficiente y reproducible.

Hay ciertas cualidades importante que un algoritmo debe cumplir:

- Tener un número finito de pasos claramente especificado
- Aceptar cero o más entradas
- Producir al menos una salida

Y estos en lo posible deberían ser:

- Correctos
  - Resolver el problema de manera precisa.
- Robusto
  - Manejar situaciones inesperadas.
- Eficiente
  - Utilizar los recursos de manera óptima, especialmente en términos de tiempo de ejecución y uso de memoria.

La elección de un algoritmo eficiente puede marcar la diferencia en aplicaciones con grandes volúmenes de datos o alta demanda de procesamiento.

¿Y cómo los identificamos?

Con el análisis de algoritmos. Este análisis es el estudio formal del rendimiento de los algoritmos midiendo sus métricas de:

- Eficiencia Temporal (Tiempo de ejecución)
- Eficiencia Espacial (Uso de memoria)

Algunas maneras de medir la Eficiencia Temporal son a través de un Análisis empírico o un Análisis teórico (notación Big-O)

El Análisis empírico mide la eficiencia de un algoritmo mediante la observación del tiempo de ejecución de un algoritmo y cuanto tarda en resolverlo mediante diferentes inputs.

Mientras que el análisis teórico es un enfoque matemático sin necesidad de ejecutar el algoritmo basándose en pseudocódigo y permite calcular una función temporal que representa el número de operaciones que realiza el algoritmo

Existen diferentes tipos de Análisis:

- Peor caso (usualmente): T(n) = Tiempo máximo necesario para un problema de tamaño (n)
- Caso medio (a veces): T(n) = Tiempo esperado para un problema cualquiera de tamaño (n)
  · (Requiere establecer una distribución estadística)
- Mejor caso (engañoso): T(n) = Tiempo menor para un problema cualquiera de tamaño (n)

Análisis empírico

Este método implica implementar el algoritmo y medir cuánto tiempo tarda en resolver un problema para diferentes tamaños de entrada.

Como realizar un análisis empírico:

1. Implementar el algoritmo
   - Escribir el código del algoritmo
2. Instrumentación
   - Incluir instrucciones para medir el tiempo de ejecución
3. Diferente Inputs
   - Ejecutar el algoritmo con datos de entrada de diferentes tamaños
4. Resultados
   - Comparar y medir resultados obtenidos
   - Pueden ser gráficos como tablas

Ventajas:

- Permite obtener gráficas que muestran el tiempo de ejecución de un Algoritmo
- Facilita la comparación visual

Desventajas:

- Es necesario implementar el algoritmo, lo que implica una inversión de tiempo y recursos.
- Para que la comparación sea válida, los algoritmos deben ejecutarse bajo las mismas condiciones de hardware y software.
- Los resultados obtenidos pueden no reflejar el comportamiento general del algoritmo, ya que solo se analizan un conjunto limitado de casos de prueba.

Análisis teórico

A diferencia del empírico no depende del hardware o software específico ya que no es necesario correr el código. También permite considerar todas las posibles entradas, no solo un conjunto limitado. Es más general y abstracto que el análisis empírico.

Este Análisis se realiza con unas estructuras de control, estas son reglas que definen el "valor" de cada algoritmo y al final se las suma. Por ejemplo:

- Secuencia: Si nuestro algoritmo se compone de varios bloques, su función T(n), será la suma de las funciones T(n) de cada bloque.

T(n)=3+2+3
Return T(n)

Cada operación (+, -, *,/ ,%,return) tienen un "valor" de 1. En el ejemplo anterior tenemos dos sumas y un return por lo tanto el "valor" del algoritmo es 3 (O(3))

- Condicionales (if-else): Como solo uno de los bloques se ejecuta o se ejecuta el if o el else se tomará el valor del algoritmo que se encuentre "debajo" de la condicional

```
If T(n)==5
        T(n)=3+5-1
Else:
        T(n)=1+5
```

En este ejemplo si es T==5 se ejecuta T(n)=3+5-1 teniendo un "valor" de 2(una suma y una resta) mientras que el Else un valor de 1 (solo una suma) dependiendo cual se ejecute será el valor que se tomara

- Bucles: Se calculará el número de veces que se ejecuta el bucle. Si dentro de un bucle se encuentra una función con un "valor" de 3 y el bucle se ejecuta 3 veces
  Seria 3 x 3(veces)= 9
  Tendría un valor de 9 O(9)

  For i in range (n) se ejecutará n veces
  Suma=2+1-3
  La variable suma se ejecutará 2 veces y como la variable suma tiene un "valor" de 2 (una suma y una resta) 2 se ejecutará n veces quedando así O(2n)
- Bucles anidados: Es el producto del número de iteraciones de cada bucle

  For i in range (n) se ejecutará n veces
  For j in range (n) se ejecutará n veces
  Suma=2+1-3
  En este caso "Suma" se ejecutará n veces, 2 veces, 1 en bucle J y otra en bucle I. Quedando así n x n = n^2 y como suma tiene un "valor" de 2, Suma se ejecutará n^2 veces quedando así $2n^2$
- Notación Big-O: siempre tomara el peor caso posible de una función.
  Se identifica el término de mayor crecimiento en T(n) y luego se eliminan constantes y coeficientes
  $T(n) = 5n^2 - 1n + 3 = O(n^2)$ (Se eliminaron las constantes (sumas y retas) y el menor valor de n)
  T(n) = 7n - 5 = O(n)
  T(n) =9+9-3 = O(2).

Complejidades Temporales

N=N+3-5 O(3) - Constante

Tiempo independiente del tamaño de entrada.

O(log n) – Logarítmica

Crecimiento muy lento. Usada en búsqueda binaria

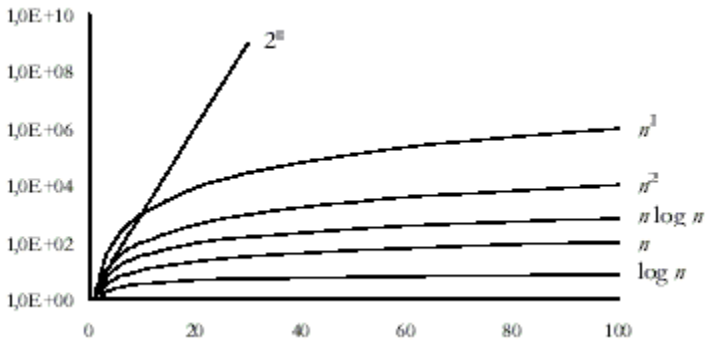
For i in range (n) O(n) – Lineal

Tiempo proporcional al tamaño de n

Doble bucle for $O(n^2)$ - Cuadrática

Crecimiento cuadrático

En este grafico se comparan los principales órdenes de complejidad. Se puede apreciar el excesivo costo que representa una complejidad de orden 2^n, frente a la estabilidad de log n y la constante siendo la menor de todas (cercana al eje x)



Complejidad en espacio

La idea utilizada para medir la complejidad temporal de un algoritmo también se aplica al análisis de su complejidad espacial. Cuando se dice que un programa tiene una complejidad de O(n) en espacio, significa que el uso de memoria crece de forma proporcional al tamaño del problema.

Es decir, si el tamaño del problema se duplica, también lo hace la memoria requerida. La complejidad del espacio también se expresa usando la notación O grande, pero se refiere al espacio adicional utilizado por el algoritmo, no al espacio total.

Por ejemplo, O(1) significa que el algoritmo usa una cantidad constante de espacio adicional Mientras que en el caso de una complejidad de $O(n^2)$, el consumo de memoria aumenta con el cuadrado del tamaño del problema, por lo que duplicar el tamaño del problema implicaría utilizar cuatro veces más memoria $2^2=4$.

Caso Práctico

Problema: Buscar un número dentro de una lista ordenada.

Algoritmos comparados:

- Búsqueda lineal: recorre la lista elemento por elemento → O(n)
- Búsqueda binaria: divide y conquista sobre lista ordenada → O(log n)

Codigo principal (Analisis.py):

```
import time

# Búsqueda lineal

def busqueda_lineal(lista, objetivo):

    for i, valor in enumerate(lista):

        if valor == objetivo:

            return i

    return -1

# Búsqueda binaria

def busqueda_binaria(lista, objetivo):

    izquierda = 0

    derecha = len(lista) - 1

    while izquierda <= derecha:

        medio = (izquierda + derecha) // 2

        if lista[medio] == objetivo:

            return medio

        elif lista[medio] < objetivo:

            izquierda = medio + 1

        else:

            derecha = medio - 1

    return -1

# Llamadas a las funciones para diferentes tamaños

for n in [100, 1000, 10000, 1000000]:

    lista = list(range(n))

    objetivo = n - 1
```

```
inicio = time.time()

busqueda_lineal(lista, objetivo)

fin = time.time()

print(f"Lineal - Tamaño {n}: {fin - inicio:.6f} segundos")

inicio = time.time()

busqueda_binaria(lista, objetivo)

fin = time.time()

 print(f"Binaria - Tamaño {n}: {fin - inicio:.6f} segundos")
```

Se probaron ambos algoritmos sobre listas de tamaño creciente (100, 1.000, 10.000, 1.000.000) y se midieron los tiempos de ejecución.

Metodología Utilizada

- Investigación teórica basada en el material de la cátedra.
- Implementación y prueba de los algoritmos en Python 3.11.
- Medición de tiempos con time.time().
- Comparación y análisis de resultados.
- Redacción colaborativa entre integrantes.
- Uso de capturas para evidencia del análisis empírico.

Resultados Obtenidos

Se observaron diferencias notables a medida que aumentó el tamaño de la lista:

- En listas pequeñas, ambos algoritmos se comportaron de forma similar.
- A partir de 10.000 elementos, la búsqueda binaria fue claramente más rápida.
- En listas de 1.000.000 de elementos, la búsqueda binaria fue más de 100 veces más veloz.

Conclusiones

A partir del caso práctico analizado, quedó demostrado cómo el análisis de algoritmos permite tomar decisiones fundamentadas sobre qué solución implementar según el contexto. La comparación entre búsqueda lineal y búsqueda binaria mostró que, aunque ambas cumplen el mismo objetivo, el tiempo de ejecución y el uso de recursos difieren drásticamente cuando se incrementa el tamaño de los datos.

Este trabajo permitió aplicar de forma concreta los conceptos teóricos vistos en clase, como la notación Big-O, el análisis empírico y teórico, y la clasificación de complejidades. También evidenció que un buen diseño algorítmico no solo es cuestión de funcionalidad, sino de eficiencia.

Comprender la lógica detrás de cada tipo de análisis nos ayuda a desarrollar software optimizado, escalable y robusto. En definitiva, el análisis de algoritmos no es solo una herramienta académica, sino una habilidad indispensable para cualquier programador profesional.

Bibliografía

- Python Software Foundation. (2024). https://docs.python.org/3/
- Sweigart, A. (2019). *Automate the Boring Stuff with Python*. No Starch Press
- Apuntes UTN: Análisis de algoritmos, Notación Big-O, Análisis teórico y empírico.
- Documentación y presentaciones de la cátedra (2025).

Anexos

Implementación en Python:

```python
import time
# Búsqueda lineal
def busqueda_lineal(lista, objetivo):
    for i, valor in enumerate(lista):
        if valor == objetivo:
            return i
    return -1
# Búsqueda binaria
def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
# Llamadas a las funciones para diferentes tamaños
for n in [100, 1000, 10000, 1000000]:
    lista = list(range(n))
    objetivo = n - 1

    inicio = time.time()
    busqueda_lineal(lista, objetivo)
    fin = time.time()
    print(f"Lineal - Tamaño {n}: {fin - inicio:.6f} segundos")

    inicio = time.time()
    busqueda_binaria(lista, objetivo)
    fin = time.time()
    print(f"Binaria - Tamaño {n}: {fin - inicio:.6f} segundos")
```

Capturas del análisis empírico:

Lista de 100 elementos: Se ejecuta la búsqueda lineal y binaria sobre una lista de 100 elementos. Las diferencias son mínimas o imperceptibles.

```
PS C:\Users\Martin> & C:/Users/Martin/AppData/Local/
Lineal - Tamaño 100: 0.000005 segundos
Binaria - Tamaño 100: 0.000005 segundos
```

Lista de 1.000 elementos: Se repite el proceso sobre una lista de 1.000 elementos. La búsqueda binaria empieza a mostrar ventaja en tiempo.

```
Lineal - Tamaño 1000: 0.000027 segundos
Binaria - Tamaño 1000: 0.000005 segundos
```

Lista de 10.000 elementos: A esta escala se evidencian diferencias marcadas.

```
Lineal - Tamaño 10000: 0.000242 segundos
Binaria - Tamaño 10000: 0.000003 segundos
```

Lista de 1.000.000 de elementos: La eficiencia de la búsqueda binaria es contundente.

```
Lineal - Tamaño 1000000: 0.025467 segundos
Binaria - Tamaño 1000000: 0.000015 segundos
```

Repositorio en GitHub: https://github.com/NahuelZa/TRABAJO-INTEGRADOR-PROGRAMACION-I

Video explicativo : https://www.youtube.com/watch?v=ofUfYSqTr6c&ab_channel=Nahuel