# Carpeta de Investigación: Análisis de Algoritmos en Python

Título del proyecto: **ANÁLISIS DE ALGORITMOS**

Alumnos:

Francisco Gutiérrez

Gaspar Inacio

Materia: Programación I

Profesor: Ariel Enferrel

Fecha de Entrega: 9 de Junio de 2025

## Introducción

Este trabajo se dedica a estudiar el **análisis de algoritmos**, un tema muy importante en el mundo de la programación. Se eligió este tema porque es esencial para crear programas que no solo funcionen, sino que lo hagan bien y rápido, incluso cuando tienen que manejar mucha información. Hoy en día, como usamos cada vez más datos y necesitamos aplicaciones que respondan al instante, saber si un programa es eficiente o no es una habilidad clave para cualquier persona que programe.

La principal utilidad de analizar algoritmos es que nos permite **anticipar cómo se va a comportar un programa**: cuánto tiempo tardará en hacer su tarea y cuánta memoria va a necesitar, especialmente si le damos una gran cantidad de datos. Entender esto ayuda a tomar mejores decisiones sobre cómo diseñar nuestros programas para resolver un problema específico.

Si no le prestamos atención a este análisis, podríamos crear un programa que funcione bien con pocos datos, pero que se vuelva exageradamente lento o consuma demasiados recursos cuando se enfrente a situaciones reales. Esto puede llevar a que los usuarios se frustren, que mantener el programa sea muy caro o, incluso, a que el sistema falle por completo.

En este TP se trata de plasmar de manera teórica y gráfica a través de la comparación de algoritmos, cómo se puede analizar su comportamiento.

## Marco Teórico

El análisis de algoritmos evalúa formalmente la eficiencia de los procedimientos computacionales, centrándose en el tiempo de ejecución y el uso de memoria en función del tamaño de la entrada. Para esto, se emplea la notación asintótica, especialmente Big O (O), que describe el comportamiento limitante o la tasa de crecimiento del consumo de recursos. Esta notación permite comparar la escalabilidad de diferentes algoritmos de manera abstracta, independientemente del hardware. Se analizan casos como el peor, mejor y promedio para entender el rendimiento bajo diversas condiciones. El fin último es seleccionar el algoritmo más adecuado para optimizar el rendimiento de las soluciones de software.

Estudio de la eficiencia de un algoritmo :

La eficiencia es la medida del coste de recursos que necesita el algoritmo para llevar a cabo su tarea

Entre los recursos más importantes se encuentran:

- Tiempo de ejecución
- Espacio de almacenamiento (memoria)

Tipos de análisis:

- Empírico
- Teórico

***Análisis empírico:***

- Se realiza la comparación de eficiencia de dos o mas algoritmos mediante gráficos que muestran los recursos (tiempo y memoria)
- Deben implementarse los algoritmos, por tal motivo se consume tiempo de trabajo
- Se debe utilizar el mismo entorno para ejecutar los diferentes algoritmos
- Los resultados podrían no ser representativos para todas las posibles entradas

***Análisis teórico:***

- Se trata de un análisis que permite clasificar y comparar funciones temporales de algoritmos sin necesidad de implementar un entorno de prueba (software/hardware)
- La función temporal T(n) representa el número de operaciones que deben ser ejecutadas para una entrada de tamaño n
- Al ser una función, es posible considerar todas las entradas posibles

El análisis teórico se basa en contabilizar las operaciones/instrucciones y asignarles una unidad de tiempo para luego llegar a un resultado final y poder comparar las funciones temporales entre sin necesidad de implementarlas de manera práctica

Dentro de las operaciones/instrucciones se pueden encontrar:

- Operaciones primitivas
- Bucles
- Operaciones selectivas o condicionales

- Operaciones primitivas
  - Asignar valor a una variable -> x=2
  - Indexar un elemento en un array -> vector[3]
  - Devolver un valor en una función -> return x
  - Evaluar una expresión aritmética -> x+5
  - Evaluar una expresión lógica -> 0<i<index

En el caso de que una operación/instrucción esté compuesta de varias operaciones primitivas, se debe descomponer y separar cada una para su evaluación

Ejemplo:

>return vector[3] + vector[6]

En este caso se puede descomponer de la siguiente manera:

- 1 operación de acceso vector[3]
- 1 operación de acceso vector[6]
- 1 operación de suma

- 1 operación return

La función temporal de las operaciones primitivas son constantes y no dependen del tamaño de la entrada

Bucles

- while condicion: operacion
- for x in ... : operacion

La función temporal de los bucles se calcula como el número de veces que se ejecuta el bucle por la función temporal de su bloque interno

Ejemplo:

```
>for i in range(1, n+1): # (n + 1)
>    result = result + i   # 2
```

#### En este caso, el número de iteraciones del bucle es n y la función temporal dentro del bloque es 2. Por lo tanto T(n) = n * 2

Bucles anidados

Se calcula como el producto del número de iteraciones de cada bucle

Ejemplo:

```
>for i in range(n)
>    for i in range(n):
>        print(i*j)    # T(n) = 2
```

Por lo tanto el resultado del ejemplo anterior es T(n) = 2*n*n = 2(n**2)

Condicionales

- if/else

En este tipo de operaciones, sólo un bloque se ejecutará mientras que los restantes solo se evaluarán de manera lógica

En este caso la función temporal se definirá como T(n) = max(Tb1(n), Tb2(n), Tbk(n))

Ejemplo:

```
>if opc == 1:              # B1
>    n = n + 1
>elif opc == 2:            # B2
>    n = n + 2
>elif opc == 3:            # B3
>    for i in range(1, n + 1):
>        print(i)
>else:                     # B4
>    print("Error")
```

El bloque con la mayor función temporal es B3, por lo tanto:

- B1 = 1 (Evaluar la condición lógica)
- B2 = 1 (Evaluar la condición lógica)
- B3 = 1 (Evaluar la condición lógica) + n (Bucle del bloque interno)

T(n) = 3 + n

## Notación Big-O

Esta notación se utiliza para describir el comportamiento asintótico de las funciones permitiendo ver como crecen a medida que el tamaño de entrada crece (tendiendo a infinito). Se utiliza para poder simplificar la comparación de las funciones temporales eliminando constantes y términos de menor orden.

Pasos para simplificar funciones utilizando notación Big-O:

- Identificar dentro de la función temporal el término de mayor crecimiento
- Eliminar constantes y coeficientes

Ejemplo:

- T(n) = 3(n**2) + 2n + 5 = O(n**2)
- T(n) = 7n + 13 = O(n)
- T(n) = 10 = O(1)

### *Órdenes de complejidad*

#### Órdenes de complejidad

| Notación | Nombre | Descripción |
|---|---|---|
| O(1) | Constante | El tiempo no depende de la entrada. |
| O(log n) | Logarítmica | Típico en algoritmos de búsqueda binaria. |
| O(n) | Lineal | Tiempo proporcional al tamaño de la entrada. |
| O(n log n) | Linealogarítmica | Típico en algoritmos de ordenación eficientes. |
| O(n²) | Cuadrática | Típico en bucles anidados. |
| O(2^n) | Exponencial | Típico en algoritmos de fuerza bruta. |
| O(n!) | Factorial | Típico en problemas de permutaciones. |

## Caso Práctico

Análisis de algoritmos:

Se definen dos funciones para sumar los n primeros números que se pasen como argumento. La primera utiliza un bucle para ir sumando los números y guardarlos en una variable para luego retornar al final.

La segunda utiliza una operación (suma de Gauss)

```python
# Comparación de funciones que suman los primeros n números
def sumar_n(n):
    resultado = 0           # 1
    for i in range(n + 1):  # n
        resultado += i      # 2
    return resultado        # 1
# T(n) = 1 + 2 * n + 1 ⇒ T(n) = 2n + 2
```

```python
# Suma de Gauss
def suma_gauss(n):
    return n * (n + 1) // 2
# T(n) = 1 + 1 + 1 + 1 ⇒ T(n) = 4
```

Luego se define una función que toma como argumento las dos funciones en cuestión y utiliza las librerías timeit y matplotlib para graficar el procesamiento a medida que aumentan los valores en función del tiempo

```python
def graficar_comparación(func1, func2):
    # Valores de entrada
    valores_n = list(range(1, 1001, 100))

    # Medición de tiempos
    tiempos_func1 = []
    tiempos_func2 = []

    for n in valores_n:
        tiempo_1 = timeit.timeit(lambda: func1(n), number=10)
        tiempo_2 = timeit.timeit(lambda: func2(n), number=10)
        tiempos_func1.append(tiempo_1)
        tiempos_func2.append(tiempo_2)

    # Gráfico de resultados
    plt.figure(figsize=(10, 6))
    plt.plot(valores_n, tiempos_func1, label="sumar n", color='blue')
    plt.plot(valores_n, tiempos_func2, label="suma gauss", color='red')
    plt.xlabel("n")
    plt.ylabel("Tiempo de ejecución (s)")
    plt.legend()
    plt.grid(True)
    plt.show()
```

## Resultados Obtenidos

Al ejecutar el código, se observa que la función sumar_n, que utiliza un bucle, presenta un aumento lineal en el tiempo de ejecución a medida que crece el valor de n, lo que corresponde a una complejidad O(n)O(n)O(n). En cambio, la función suma_gauss, basada en una fórmula matemática, mantiene un tiempo de ejecución constante O(1)O(1)O(1). El gráfico generado muestra claramente esta diferencia, evidenciando la eficiencia superior de la solución algorítmica directa frente a la iterativa. Esto demuestra la importancia de elegir algoritmos eficientes en función del problema a resolver.

## Conclusiones

- El análisis de algoritmos permite comparar la complejidad temporal de un algoritmo y comparar algoritmos que resuelven el mismo problema para proponer la mejor solución.

- El análisis empírico se basa en la medición de tiempos de ejecución de algoritmos para distintas entradas. La dificultad de este análisis radica en la necesidad de ejecutar el algoritmo y compararlo con los demás en las mismas condiciones, dependiendo del software y el hardware que se utilice.

- El análisis teórico se ejecuta contabilizando el número de instrucciones del algoritmo para configurar una funcion temporal

- La obtención de la función temporal (T(n)) facilita el análisis ya que en base a las cotas superiores obtenidas (o funciones Big-O) permite comparar gráficamente las funciones temporales sin necesidad de implementar o requerir entornos de pruebas.

## Bibliografía


- Documentación oficial Python time: https://docs.python.org/3/library/time.html
- Big O Cheat Sheet: https://www.bigocheatsheet.com/