**TRABAJO INTEGRADOR – PROGRAMACIÓN**

**Análisis de algoritmos**

**Alumnos:**

Vega Luis Fernando (luis.vega@tupad.utn.edu.ar)

Gabriel Valdez Arce (gabrielvaldezarce@gmail.com)

**Materia:** Programación I

**Profesor/a:** Nicolás Quirós

**Fecha de Entrega:** 09 de Junio del 2025

# Índice

# 1. Introducción.

Este trabajo se enfoca en el análisis de algoritmos, un tema que forma parte de los conceptos clave en programación. Se eligió porque ayuda a entender cómo funcionan los algoritmos por dentro y por qué algunos son más rápidos o eficientes que otros. No siempre basta con que un programa funcione. Muchas veces también es importante que lo haga de forma rápida, especialmente, si se trabaja con muchos datos o se quiere que el programa escale bien.

El **análisis de algoritmos** permite comparar distintas soluciones de un mismo problema sin necesidad de probar cada una por separado. Esto es útil porque se puede anticipar qué tan bien se comportará un algoritmo solo con mirar su estructura y su lógica.

El objetivo principal de este trabajo es aprender a reconocer esas diferencias aplicando ejemplos concretos y viendo cómo cambian los resultados según el tipo de algoritmo. Al finalizar, se espera haber adquirido herramientas para elegir o construir algoritmos más eficientes en futuros proyectos.

# 2. Marco Teórico.

### ¿Qué es un algoritmo?

Un algoritmo es una secuencia de pasos definidos que permite resolver un problema o realizar una tarea. En programación, cada función o conjunto de instrucciones que se escribe puede considerarse un algoritmo, ya que tiene un inicio, un desarrollo y un resultado. Es la base de cualquier programa, sin importar el lenguaje de programación.

### ¿Qué es el análisis de algoritmos?

El análisis de algoritmos es el estudio de qué tan eficiente es un algoritmo. Esto significa analizar cuánto tiempo tarda en ejecutarse y cuánta memoria necesita para funcionar. Este análisis permite comparar distintas soluciones a un mismo problema y elegir la más adecuada según el contexto.

## Eficiencia temporal y espacial

- **Eficiencia temporal**: se refiere al tiempo que tarda un algoritmo en dar una respuesta.

- **Eficiencia espacial**: se refiere a la cantidad de memoria adicional que necesita durante su ejecución.

Ambas medidas son importantes, especialmente si se trabaja con grandes volúmenes de datos o en dispositivos con pocos recursos.

## Tipos de análisis

- Mejor caso: es la situación más favorable, donde el algoritmo realiza la menor cantidad de operaciones posibles.

- Peor caso: es el escenario más desfavorable, donde el algoritmo necesita el mayor esfuerzo para resolver el problema.

- Caso promedio: Es una estimación del comportamiento del algoritmo en situaciones normales.

¿Qué es la notación Big-O?

La notación Big-O es una forma de representar cómo crece el tiempo de ejecución de un algoritmo en función del tamaño de los datos. No se enfoca en cuánto tarda exactamente, sino en cómo se comporta cuando la cantidad de datos aumenta mucho.

Algunos **ejemplos** comunes son:

- O(1) – Tiempo constante: no cambia según la cantidad de datos (por ejemplo, acceder a un elemento de una lista).

- O(n) – Tiempo lineal: el tiempo crece en proporción al número de datos (por ejemplo, recorrer una lista).

- $O(n^2)$ – Tiempo cuadrático: el tiempo crece rápidamente, usado en algoritmos como el bubble sort.

- O(log n) – Tiempo logarítmico: muy eficiente en listas ordenadas, como en la búsqueda binaria.

**Importancia en la programación**

Conocer la eficiencia de un algoritmo permite escribir programas más rápidos y que usen menos recursos. Esto es especialmente útil cuando se trabaja en aplicaciones reales, como motores de búsqueda, juegos, inteligencia artificial o procesamiento de datos. El análisis de algoritmos ayuda a tomar decisiones antes de programar, y evita perder tiempo en soluciones que no escalan bien.

# 3. Caso Práctico

Se desea comparar la eficiencia de dos algoritmos que determinen si una palabra es palindromo, uno utilizando un bucle for y el otro usando el operador [ : : -1 ]

**a.** Se desarrollan dos códigos, que comparen si una palabra ingresada por el usuario es palíndroma.

Algoritmo 1: se verifica una palabra mediante un bucle el cual itera n veces.

Algoritmo 2: se verifica la palabra mediante [: :-1], el cual devuelve una copia de la palabra pero en orden inverso. esto quiere decir que recorre toda la cadena n veces y crea una nueva con n caracteres.

**b.** se realiza el cálculo de la función temporal T(n)

Algoritmo 1: **2+n4**

```
def es_palindromo_bucle(palabra):
    n = len(palabra) # O(1)
    for i in range(n // 2): # O(n)
        if palabra[i] != palabra[n - 1 - i]: # O(3)
            return False #O(1)
    return True #O(1)
#T(n)= 2+n4
```

Algoritmo 2: **3n+1**

```
def es_palíndromo(palabra):
    return palabra == palabra[::-1] # O(2n) de [::-1], O(n) de la
       comparación y O(1) del return
#T(n)=2n+n+1 = 3n+1
```

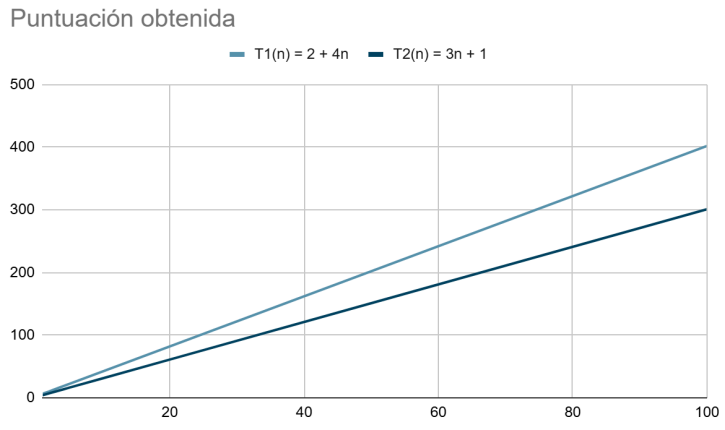
**c.** Calcular Big-O, identificando el término de mayor crecimiento e eliminando constantes y coeficientes.

**T1(n)= 2+n4 → O(n).**

**T2(n)= 3n+1 → O(n).**

**d.** Comparación de algoritmos mediante gráfica.

| n | T1(n) = 2 + 4n | T2(n) = 3n + 1 |
|---|---|---|
| 1 | 6 | 4 |
| 2 | 10 | 7 |
| 3 | 14 | 10 |
| 4 | 18 | 13 |
| 5 | 22 | 16 |
| 10 | 42 | 31 |
| 20 | 82 | 61 |
| 50 | 202 | 151 |
| 100 | 402 | 301 |




# 4. Metodología Utilizada.

- Se utilizó el cálculo de la función temporal T(n), mediante las operaciones primitivas.
- Se calculó la notación Big-O, se transformaron los datos primitivos, eliminando constantes y coeficientes.
- Se realizó un cuadro comparativo de ambos algoritmos, donde se detectó la eficiencia de los mismos

## 5.Resultados Obtenidos.

Los dos algoritmos son de tipo O(n) lineal, podemos observar que T2 (utilizando el slicing), es ligeramente más eficiente porque tiene un menor coeficiente en n, siendo esta más rápida. mientras que T1 tiene un coeficiente mayor, es ligeramente más lento y es más eficiente procesando muchos más datos que T2.

## 6. Conclusión.

a través del análisis de ambos algoritmos, mediante el cálculo de la función temporal T(n) y la Notación Big-O, tuvimos una visión más completa del comportamiento algorítmico, ayudó a tomar decisiones en el uso del diseño y la elección del algoritmo según el tipo y volumen de datos a procesar.

## 7. Bibliografía

-Bhargava, A. (2016). Grokking Algorithms: An illustrated guide for programmers and other curious people. Manning Publications.

-Python Software Foundation. (2024). Documentación oficial del módulo time en Python. https://docs.python.org/3/library/time.html

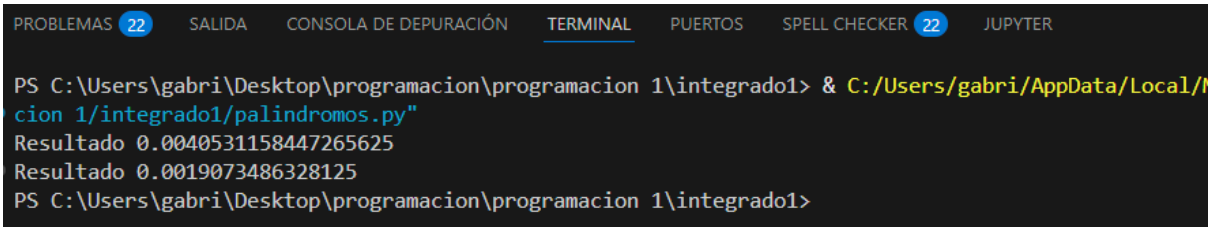
-Big-O Cheat Sheet. (2024). Complexity chart for common algorithms. https://www.bigocheatsheet.com/
-Cátedra Programación I – UTN FRBA. (2025). Análisis de Algoritmos – Fundamentos y Técnicas de Optimización.
-Cátedra Programación I – UTN FRBA. (2025). Apunte: Análisis de algoritmos (PDF)

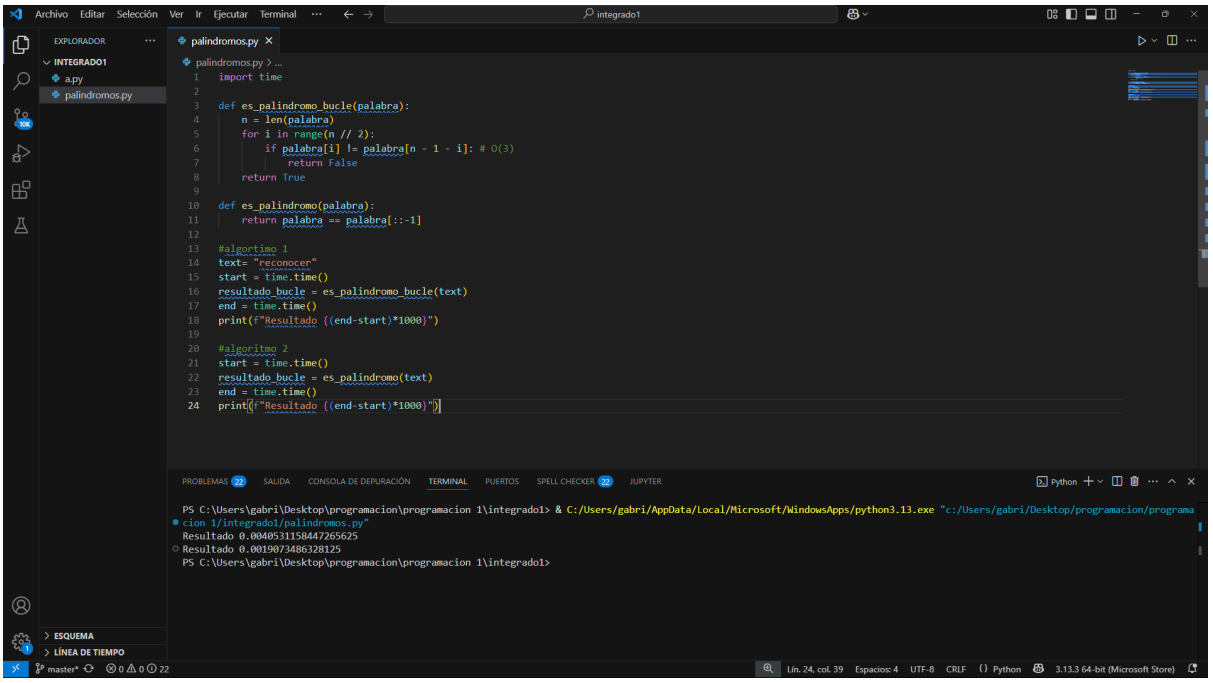
# 8. **Anexos**

A continuación, se presenta la captura de pantalla de los resultados obtenidos al medir los tiempos de ejecución de los dos algoritmos analizados:

```python
import time

def es_palindromo_bucle(palabra):
    n = len(palabra)
    for i in range(n // 2):
        if palabra[i] != palabra[n - 1 - i]: # O(3)
            return False
    return True

def es_palindromo(palabra):
    return palabra == palabra[::-1]

```

```python
#algortimo 1
text= "reconocer"
start = time.time()
resultado_bucle = es_palindromo_bucle(text)
end = time.time()
print(f"Resultado {(end-start)*1000}")

#algoritmo 2
start = time.time()
resultado_bucle = es_palindromo(text)
end = time.time()
print(f"Resultado {(end-start)*1000}")
```




## Enlace video:

https://drive.google.com/drive/u/0/folders/1cWODn8cEw09WVBB13BgqLmXFfETTk8iS