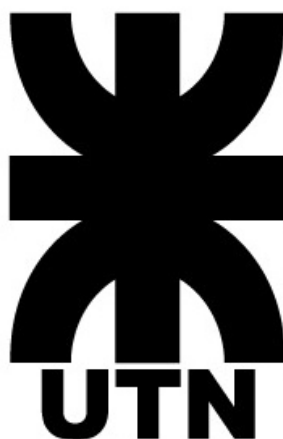


Universidad Tecnológica Nacional  
Tecnatura Universitaria en Programación



**TRABAJO PRÁCTICO INTEGRADOR**  
**“ANÁLISIS DE ALGORITMOS”**

**Integrantes:** Lisandro Alarcón y Manuel Barrera.

**Año:** 2025 - 1° Cuatrimestre

**Materia:** Programación I

**Profesora:** Cinthia Rigoni

**Fecha de Entrega:** 09/06/2025

**Correos de los alumnos:**

Manuel Barrera - manubaterobarrera@gmail.com

Lisandro Alarcón - lichualarconn@gmail.com

**Índice**

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

**Introducción:**

Elegimos este tema porque creemos que es una herramienta y conocimiento fundamental al permitirnos evaluar y comparar el rendimiento de distintas soluciones a un mismo problema. Si bien los algoritmos pueden resolverse de múltiples maneras, no significa que todos lo hagan con la misma eficiencia.

En la programación, el análisis de algoritmos es muy importante porque permite anticipar el consumo de recursos (principalmente tiempo y memoria) sin necesidad de ejecutar el código (en caso del análisis teórico). Esto toma una enorme importancia en sistemas grandes donde el rendimiento es crucial.

Con este trabajo se busca entender cómo se analiza el rendimiento de los algoritmos, tanto desde un enfoque teórico como empírico. Se busca aprender a calcular funciones de tiempo  $T(n)$  y a interpretar su comportamiento mediante la notación Big-O, entendiendo cómo las

distintas estructuras de control afectan el crecimiento del tiempo de ejecución. Además, se tiene como objetivo aplicar estos conocimientos en la práctica, mediante la implementación del análisis empírico y teórico (con la notación Big-O), la medición de los tiempos de ejecución de dos algoritmos, y la comparación entre dos distintas soluciones para un mismo problema.

## **Marco Teórico**

### **¿Qué es un algoritmo?**

Un algoritmo es un conjunto de operaciones que busca resolver un problema determinado a través de secuencias lógicas. Este procedimiento esquemático emplea una serie de pasos, como una receta, los cuales pueden ser formulados de diferentes maneras cuidando que en dicha combinación no se produzca una ambigüedad.

La principal utilización de estas fórmulas es que su resultado muestre el punto o destino buscado a través de las secuencias de instrucciones previas, definidas por su programador.

### **Características de un algoritmo**

- Preciso: cada paso y su orden de realización deben ser claros y concretos
- Definido: se deben obtener resultados delimitados a las órdenes y estos siempre deben ser los mismos
- Finito: su diseño debe tener un número limitado de pasos
- Ordenado: la secuencia de pasos debe seguir un orden que no puede ser alterado
- Eficiente: debe utilizar los recursos de manera óptima, especialmente en términos de tiempo de ejecución y uso de memoria.
- Robusto: Debe manejar situaciones inesperadas sin fallar.

### **¿Qué es el análisis de algoritmos?**

El análisis de algoritmos se ocupa de comparar algoritmos con base en la cantidad de recursos computacionales que utiliza cada uno. Queremos ser capaces de considerar dos algoritmos y decir que uno es mejor que el otro, porque es más eficiente en su uso de esos recursos o simplemente tal vez porque utiliza una menor cantidad

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

## Conceptos clave del análisis de algoritmos

Complejidad temporal: El tiempo que requiere el algoritmo para resolver un problema dado.

Complejidad espacial: La cantidad de memoria que requiere el algoritmo para resolver un problema dado

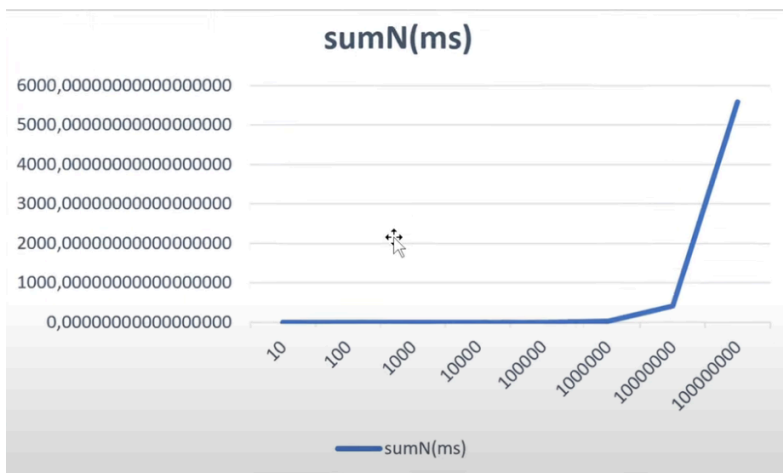
### ¿Qué es el análisis empírico?

El análisis empírico es un método para evaluar el rendimiento de un algoritmo ejecutándolo en un conjunto de entradas y midiendo el tiempo o los recursos que consume. El análisis empírico puede ayudar a identificar las fortalezas y debilidades de un algoritmo, probar su escalabilidad y eficiencia, y encontrar posibles cuellos de botella y errores. También permite obtener gráficas del tiempo de ejecución y poder realizar una comparación visual de los algoritmos

En este análisis no se evalúan todas las posibles entradas, requiere implementación y depende del software y hardware donde se ejecuta.

### ¿Cómo hacer un análisis empírico?

1. Escribir el algoritmo (o los algoritmos) en Python (ya que trabajaremos con este lenguaje)
2. Incluir la función `perf_counter()` del módulo `time` para obtener el tiempo de una manera precisa. Se debe utilizar antes y después de la ejecución del llamado a la función a analizar. Luego, se deberá restar el tiempo final al del inicio para obtener el tiempo que tardó en ejecutarse.
3. Ejecutar el programa con entradas de diferentes tamaños y obtener sus tiempos de ejecución.
4. Importar los resultados a una hoja de cálculo y crear un gráfico de líneas (X-Y) para analizar el comportamiento del tiempo de ejecución en función del tamaño de la entrada.



## ¿Qué es el análisis teórico?

El análisis teórico es un enfoque matemático para evaluar la eficiencia de un algoritmo sin necesidad de implementarlo. Este método se basa en el pseudocódigo del algoritmo y permite calcular una función temporal  $T(n)$ , que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño  $n$ .

El análisis teórico no depende del software o hardware y permite considerar todas las entradas posibles. A diferencia del análisis empírico, éste es más general y abstracto.

## ¿Cómo se calcula?

- Se cuentan operaciones primitivas como asignaciones, comparaciones, sumas, accesos a arrays, etcétera.
- Se analizan estructuras de control:
  - Secuencias: se suman los  $T(n)$  de cada bloque.
  - Condicionales: se toma el peor caso (el máximo  $T(n)$ ).
  - Bucles: se multiplican las iteraciones por el costo del bloque interno.
  - Bucles anidados: se multiplica el número de iteraciones de cada bucle.

## Caso promedio, peor y mejor de un algoritmo

Peor caso: En el análisis del peor caso, calculamos el límite superior del tiempo de ejecución de un algoritmo. Debemos conocer el caso que provoca la ejecución de un número máximo de operaciones.

Caso promedio: En el análisis de caso promedio, tomamos todas las entradas posibles y calculamos el tiempo de cálculo para cada una de ellas. Sumamos todos los valores calculados y dividimos la suma entre el número total de entradas.

Mejor caso: En el análisis del mejor caso, calculamos el límite inferior del tiempo de ejecución de un algoritmo. Debemos conocer el caso que requiere la ejecución de un número mínimo de operaciones.

## ¿Qué es la 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 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.

Mientras el análisis teórico calcula la función  $T(n)$ , Big-O abstrae y simplifica ese resultado, enfocándose en lo que realmente importa: cómo escalan los recursos necesarios (tiempo o espacio) al aumentar el tamaño del problema.

### Complejidades comunes (Big-O)

| Notación Big-O | 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   |
| $O(n \log n)$  | Lineal-Logarítmica | Típico en algoritmos de ordenación eficientes |
| $O(N^2)$       | 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

Consigna que nos planteamos:

Desarrollar un programa en Python que compare el rendimiento de dos formas de buscar un valor en una lista: uno mediante una función manual que recorre secuencialmente la lista con un bucle for, y otro utilizando el operador in. El código debe crear una lista grande, definir casos de búsqueda correspondientes al mejor, promedio y peor escenario, medir los tiempos de ejecución de ambos métodos en cada caso, y finalmente imprimir los resultados para su análisis comparativo. Además, se debe aplicar la notación Big-O para analizar la complejidad teórica de ambas implementaciones y relacionarla con los resultados empíricos obtenidos.

Descripción del caso práctico:

El objetivo de este caso práctico es comparar el rendimiento de dos formas de buscar un valor dentro de una lista en Python. Por un lado, se implementó una búsqueda manual recorriendo la lista con un for, y por otro, se utilizó el operador in, que es la forma más común y directa en Python para este tipo de tareas. Para el análisis teórico, se aplicó la notación Big-O, que permite expresar la complejidad del algoritmo en función del tamaño de la entrada. En este caso, ambas implementaciones presentan una complejidad  $O(n)$  en el peor caso, lo que significa que el tiempo de ejecución crece de forma lineal a medida que aumenta el tamaño de la lista. El propósito del trabajo es evaluar si esa equivalencia teórica también se refleja en la práctica o si hay diferencias de rendimiento medibles entre ambas formas de búsqueda.

Para hacerlo más completo, se medirán tiempos en tres situaciones distintas:

- Cuando el valor está al principio de la lista (mejor caso)
- Cuando está en el medio (caso promedio)
- Cuando no está (peor caso)

Estos tres casos nos permitirán observar cómo varía el comportamiento del algoritmo según el caso.

A partir de los tiempos obtenidos se construirá un gráfico, para comparar visualmente el rendimiento de ambas funciones. Esto no solo nos permitirá analizar la teoría, sino también ver cómo se comportan estos casos en la práctica.

Código del programa (integrador\_programacion.py)

```
import time
```

```
def encontrar_valor(lista, valor):
```

```
    # Recorrer la lista secuencialmente
```

```
    for numero in lista: # se recorre n veces
```

```
        if numero == valor: # 1 operación n veces -> O(n)
```

```
            return True # O(1)
```

```
    return False # O(1)
```

```
def esta_en_lista(lista, valor):
```

```
    # El operador in realiza internamente una búsqueda secuencial optimizada
```

```
    return valor in lista # O(n)
```

```
# Creamos una lista con un rango de 10 millones
```

```
n = 10000000
```

```
mi_lista = list(range(n))
```



# Definimos los casos en un diccionario para luego recorrer cada caso.

```
casos = {

    "Mejor caso": mi_lista[0], #O(1) -> está al inicio

    "Caso promedio": mi_lista[n//2], #O(n) -> está a la mitad

    "Peor caso": -1 # O(n) -> inexistente

}
```

for caso, valor\_a\_buscar in casos.items():

# Búsqueda manual con la primera función -> O(n)

inicio = time.perf\_counter() # Se guarda el tiempo preciso del inicio

rta\_encontrar = encontrar\_valor(mi\_lista, valor\_a\_buscar)

fin = time.perf\_counter() # Se guarda el tiempo preciso al finalizar

tiempo\_encontrar = (fin - inicio)\*1000 # Convertir a ms

# Búsqueda con la segunda función -> O(1)

inicio = time.perf\_counter()

rta\_lista = esta\_en\_lista(mi\_lista, valor\_a\_buscar)

fin = time.perf\_counter()

tiempo\_esta\_en\_lista = (fin - inicio) \* 1000

# Imprimir los resultados e informar si se encontró o no el valor.

```
print(f"{caso}-> 1ra Función: {tiempo_encontrar:.6f} ms | 2da Función:
{tiempo_esta_en_lista:.6f} ms | ¿Encontrado? -> {rta_encontrar}")
```

### **Metodología Utilizada:**

- Selección del tema análisis de algoritmos
- Realizar la investigación previa al desarrollo del trabajo práctico, incluyendo fuentes internas y externas al Aula Virtual.
- Organizar el trabajo de manera colaborativa, repartiendo las tareas entre los integrantes del grupo.
- Desarrollo del marco teórico del trabajo práctico
- Diseño y desarrollo del código para resolver el problema que nos planteamos resolver en el caso práctico. Se utilizó Visual Studio Code con el lenguaje Python.
- Se grabó la prueba del código para tener los datos necesarios para realizar la comparación entre los algoritmos. Se desarrollaron las partes “Resultados obtenidos” y “Conclusión”. (incluido el gráfico del análisis empírico y la comparación)
- Uso de la aplicación Zoom/Celular para grabar con cámara la presentación de los integrantes.
- Grabar el audio para el video explicativo utilizando la función de audios en Whatsapp.
- Utilizar la aplicación CapCut para la edición del video.
- Subir el video explicativo a YouTube de manera que pueda acceder quien tenga el link.
- Crear un repositorio remoto en GitHub para el trabajo práctico integrador.
- Documentar el trabajo en el README del repositorio.
- Utilizar la herramienta de control de versiones Git para subir los archivos de nuestro repositorio local al repositorio remoto.

## Resultados Obtenidos

Se logró implementar y ejecutar satisfactoriamente la comparación entre una búsqueda manual y una búsqueda con el operador in para verificar la existencia de un valor en una lista grande. Todos los bloques del código funcionaron correctamente, y los tiempos de ejecución se midieron de forma precisa en los 3 casos planteados.

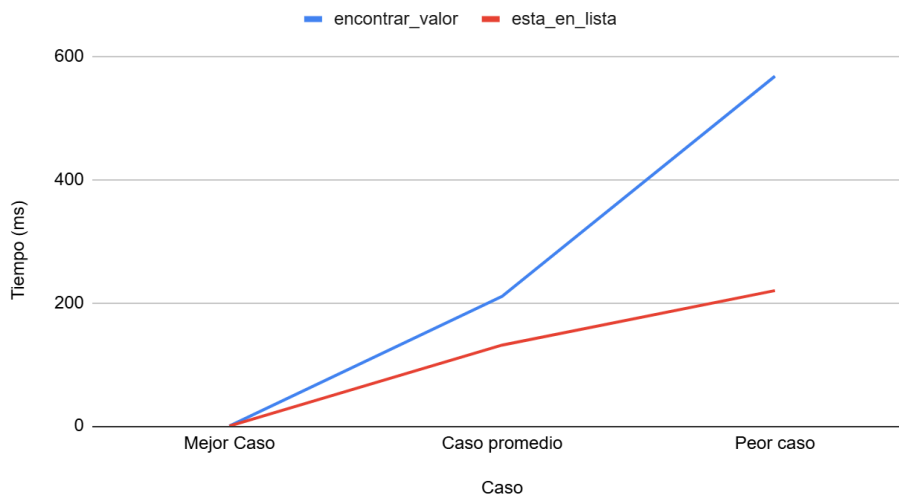
No sugieren dificultades técnicas durante el desarrollo ni en la ejecución del código, por lo que podemos decir que el caso práctico fue un éxito.

Se realizó un caso de prueba donde se obtuvieron los siguientes resultados:

```
Mejor caso-> 1ra Función: 0.003300 ms | 2da Función: 0.004200 ms | ¿Encontrado? -> True
Caso promedio-> 1ra Función: 210.827700 ms | 2da Función: 131.462000 ms | ¿Encontrado? -> True
Peor caso-> 1ra Función: 568.292400 ms | 2da Función: 219.974200 ms | ¿Encontrado? -> False
```

> Se graficó el análisis empírico:

Tiempos de ejecución por tipo de caso



Los resultados obtenidos en la ejecución nos permiten observar claras diferencias en el rendimiento de ambas funciones, a pesar de tener la misma complejidad  $O(n)$  en el peor caso.

En el mejor caso, donde el valor a buscar se encuentra al comienzo de la lista, ambas funciones tienen tiempos de ejecución demasiado bajos y similares (0.0033 ms para la búsqueda manual

y 0.0042 ms para el operador in). Esto se debe a que detienen la búsqueda tras la primera comparación, por lo que era una respuesta esperable -> Se tiene un comportamiento muy óptimo ( $O(1)$ )

En el caso promedio, el rendimiento se diferencia claramente. En este caso, la búsqueda manual tardó aproximadamente 210.83 ms, mientras que el operador in lo resolvió en 131.46 ms.

En el peor caso, donde el valor no está dentro de la lista y se requiere recorrerla completa, la diferencia se aprecia mucho mejor que en el caso promedio. La función manual tardó 568.29 ms, mientras que la del operador in tardó 219.97 ms, menos de la mitad de lo que tardó la búsqueda manual. Este caso deja evidente como la optimización puede marcar una diferencia considerable en la ejecución de nuestros programas, incluso cuando la complejidad común es la misma.

Aunque ambas soluciones son lineales, es decir, su tiempo de ejecución crece proporcionalmente al tamaño de la entrada, el análisis empírico demuestra que la implementación de "in" resulta mucho más eficiente que la versión manual. El rendimiento de un algoritmo no depende únicamente de su complejidad común, sino también de cómo implementamos ese algoritmo y qué herramientas o funciones del lenguaje utilizamos. Es recomendable utilizar recursos optimizados del lenguaje, ya que puede significar una mejora en la eficiencia del programa.

## **Conclusiones:**

A lo largo del desarrollo de este trabajo profundizamos en el análisis de algoritmos desde una perspectiva tanto teórica como práctica. Se aprendió a medir tiempos de ejecución en distintos casos (mejor, promedio y peor caso), a observar cómo la cantidad de elemento dentro de una estructura de datos puede afectar el rendimiento, y a comparar dos implementaciones distintas pero con el mismo objetivo.

Este tema también resulta muy útil en contextos reales de programación, donde frecuentemente es necesario optimizar algoritmos para mejorar el rendimiento del software. Evaluar el comportamiento de distintas implementaciones nos va a permitir tomar mejores decisiones al momento de querer desarrollar una solución eficiente. Sabemos que el rendimiento tiene un

impacto muy grande en la experiencia del usuario, por lo que comprender el tema de análisis de algoritmos es algo importante para los programadores.

Como posible mejora, sería buena idea automatizar la repetición de pruebas varias veces por cada caso y el cálculo del promedio de los tiempos obtenidos. Esto permitiría obtener resultados más precisos de cada caso.

## Bibliografía

GeeksforGeeks. (s.f.). *Time complexity and space complexity*. GeeksforGeeks.

<https://www.geeksforgeeks.org/time-complexity-and-space-complexity/>

GeeksforGeeks. (s.f.). *Worst, average and best case analysis of algorithms*. GeeksforGeeks.

<https://www.geeksforgeeks.org/worst-average-and-best-case-analysis-of-algorithms/>

LinkedIn. (s.f.). *¿Cómo puedes medir el tiempo real de ejecución de un algoritmo usando Python?*

LinkedIn. <https://www.linkedin.com/advice/0/how-can-you-measure-actual-running-time-algorithm-using?lang=es&originalSubdomain=es#%C2%BFqu%C3%A9-es-el-an%C3%A1lisis-emp%C3%ADrico?>

MSMK University. (s.f.). *Notación Big O: qué es y cómo aplicarla*.

<https://msmk.university/big-o-notation/>

NIC Argentina. (s.f.). *¿Qué es un algoritmo?* NIC Argentina.

<https://nic.ar/es/enterate/novedades/que-es-algoritmo>

Python Software Foundation. (s.f.). *time — Time access and conversions*. Python 3.13 documentation. [https://docs.python.org/3/library/time.html#time.perf\\_counter](https://docs.python.org/3/library/time.html#time.perf_counter)

Python Software Foundation. (s.f.). *Entrada y salida*. Tutorial de Python 3.13.

<https://docs.python.org/es/3.13/tutorial/inputoutput.html>

Runestone Academy. (s.f.). *¿Qué es el análisis de algoritmos?* En *PythonED: Análisis de algoritmos*.

<https://runestone.academy/ns/books/published/pythoned/AlgorithmAnalysis/QueEsAnalisisDeAlgoritmos.html>

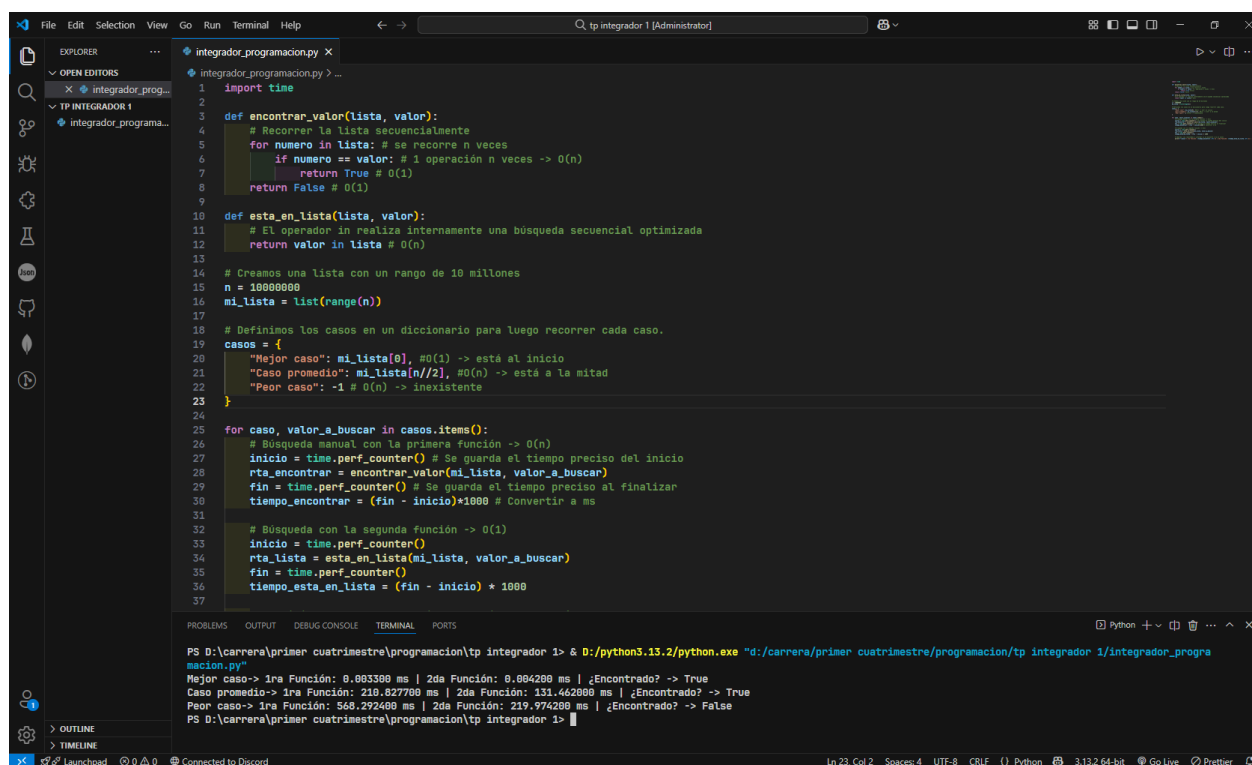
Universidad Tecnológica Nacional – Facultad Regional San Nicolás. (2025). *Análisis de algoritmo empírico* [Archivo PDF].

Universidad Tecnológica Nacional – Facultad Regional San Nicolás. (2025). *Análisis de algoritmo teórico y Big O* [Archivo PDF].

Universidad Tecnológica Nacional – Facultad Regional San Nicolás. (2025). *Notación Big O* [Archivo PDF].

## Anexos

- Captura del programa funcionando:



The screenshot shows a VS Code editor with a Python file named `integrador_programacion.py`. The code defines two functions: `encontrar_valor` (sequential search) and `esta_en_lista` (optimized search). It then creates a list of 10 million elements and tests both functions with different cases. The terminal output shows the execution results, including execution times and whether the value was found.

```

1  import time
2
3  def encontrar_valor(lista, valor):
4      # Recorrer la lista secuencialmente
5      for numero in lista: # se recorre n veces
6          if numero == valor: # 1 operación n veces -> O(n)
7              return True # O(1)
8      return False # O(1)
9
10 def esta_en_lista(lista, valor):
11     # El operador in realiza internamente una búsqueda secuencial optimizada
12     return valor in lista # O(n)
13
14 # Creamos una lista con un rango de 10 millones
15 n = 10000000
16 mi_lista = list(range(n))
17
18 # Definimos los casos en un diccionario para luego recorrer cada caso.
19 casos = {
20     "Mejor caso": mi_lista[0], #0(1) -> está al inicio
21     "Caso promedio": mi_lista[n//2], #O(n) -> está a la mitad
22     "Peor caso": -1 # O(n) -> inexistente
23 }
24
25 for caso, valor_a_buscar in casos.items():
26     # Búsqueda manual con la primera función -> O(n)
27     inicio = time.perf_counter() # Se guarda el tiempo preciso del inicio
28     rta_encontrar = encontrar_valor(mi_lista, valor_a_buscar)
29     fin = time.perf_counter() # se guarda el tiempo preciso al finalizar
30     tiempo_encontrar = (fin - inicio)*1000 # Convertir a ms
31
32     # Búsqueda con la segunda función -> O(1)
33     inicio = time.perf_counter()
34     rta_lista = esta_en_lista(mi_lista, valor_a_buscar)
35     fin = time.perf_counter()
36     tiempo_esta_en_lista = (fin - inicio) * 1000
37

```

Terminal Output:

```

PS D:\carrera\primer cuatrimestre\programacion\tp integrador 1> & D:/python3.13.2/python.exe "d:/carrera/primer cuatrimestre/programacion/tp integrador 1/integrador_progra
nacion.py"
Mejor caso-> 1ra Función: 0.003309 ms | 2da Función: 0.004200 ms | ¿Encontrado? -> True
Caso promedio-> 1ra Función: 210.827700 ms | 2da Función: 131.462800 ms | ¿Encontrado? -> True
Peor caso-> 1ra Función: 560.292400 ms | 2da Función: 219.974200 ms | ¿Encontrado? -> False
PS D:\carrera\primer cuatrimestre\programacion\tp integrador 1>

```

- Enlace al video explicativo: [LINK](#)
- Repositorio en GitHub: [https://github.com/kiensovoo/P1\\_TP\\_Integrador](https://github.com/kiensovoo/P1_TP_Integrador)
- Código completo: <https://pastebin.com/EcqNWTU9>

- Cuadro comparativo:

| Caso          | encontrar_valor() | esta_en_lista() |
|---------------|-------------------|-----------------|
| Mejor Caso    | 0,0033 ms         | 0,0042 ms       |
| Caso promedio | 210,8277 ms       | 131,462 ms      |
| Peor caso     | 568,2924 ms       | 219,9742 ms     |