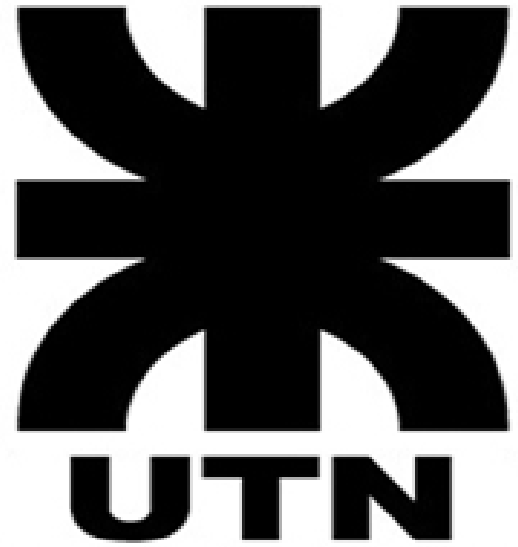
UNIVERSIDAD TECNOLÓGICA NACIONAL

FACULTAD SAN NICOLÁS



TECNICATURA EN PROGRAMACIÓN

ANÁLISIS DE ALGORITMOS

Profesores:

- Nicolás Quiros
- Flor Camila Gubiotti

Integrantes: **Correos:**

- Francisco Ezequiel Alarcón franciscoalarcon.ez@gmail.com
- Leandro David Zalazar david26692@gmail.com

Fecha de Entrega:

09/06/2025

Contenido:

Introducción 3
Marco teórico 4
Caso Práctico.....9
Metodología Utilizada:.....17
Resultados Obtenidos: 18
Conclusiones 19
Bibliografía 20
Anexo 20

Introducción

El análisis de algoritmos permite medir y comparar la eficiencia de los programas en términos de tiempo de ejecución y uso de memoria. Comprender cómo se comporta un algoritmo frente al aumento de datos es esencial para desarrollar aplicaciones eficientes y escalables. En este trabajo el enfoque es comparar la eficiencia de dos algoritmos que cumplen el mismo propósito, pero con diferencias en la implementación, en este caso se ha de comparar la función max() de Python con una función desarrollada en forma manual. también este trabajo se centrará en como cada algoritmo, permite no solo cumplir la misma función si no cual nos conviene usar, en las diferentes situaciones del ámbito de la programación tanto laboral como educativo.

Marco teórico

¿Qué es un algoritmo?

El algoritmo es en esencia una secuencia de pasos, que se crea para poder solucionar, una problemática, realizar cálculos, procesar datos, llevar a cabos otras tantas tareas. Los algoritmos tampoco escapan a la vida cotidiana, porque se usan esencialmente casi todos los días para resolver cuestiones tan mundanas, como el cocinar, que es directamente proporcional a seguir instrucciones para tener determinados resultados, otro ejemplo sería un manual de uso de un electrodoméstico, entre otras cosas, las multiplicaciones y divisiones que usamos a diario para resolver cálculos en nuestra vida.

Características de un algoritmo:

* Tienes reglas definidas

* No es ambiguo

* Esta de manera ordenada

* Es finito

¿Qué es el análisis de algoritmos?

El término análisis de algoritmos fue acuñado por Donald Knuth, científico y matemático, norteamericano y se refiere al proceso de encontrar la complejidad computacional de un algoritmo que resuelva un problema computacional dado, con el objetivo de proveer estimaciones teóricas de los recursos que necesita.

Usualmente, los recursos a los cuales se hace referencia son el tiempo (complejidad temporal) y el almacenamiento (complejidad espacial). Mientras que la complejidad temporal involucra determinar una función que relaciona la longitud o el tamaño de la entrada del algoritmo con el número de pasos que realiza, la complejidad espacial busca la cantidad de ubicaciones de almacenamiento que utiliza. Distintos algoritmos pueden utilizarse para resolver un mismo problema y a su vez los algoritmos pueden estudiarse de forma independiente del lenguaje de programación a utilizar y de la

máquina donde se ejecutará. Esto significa que se necesitan técnicas que permitan comparar la eficiencia de los algoritmos antes de su implementación, porque de esta manera comparando distintos algoritmos, podes llegar a saber cual conviene, utilizar en lo que queremos implementar.

¿Que es un Algoritmos de búsqueda?

Un algoritmo de búsqueda es un conjunto de indicaciones, que literalmente están diseñadas para buscar elementos, con ciertas propiedades dentro de una estructura de datos, como puede ser una lista, o dentro de una tabla de datos entre otras.

Los algoritmos de búsqueda cuentan con varios tipos como puede ser binaria, secuencial, estas son unas de las más usadas.

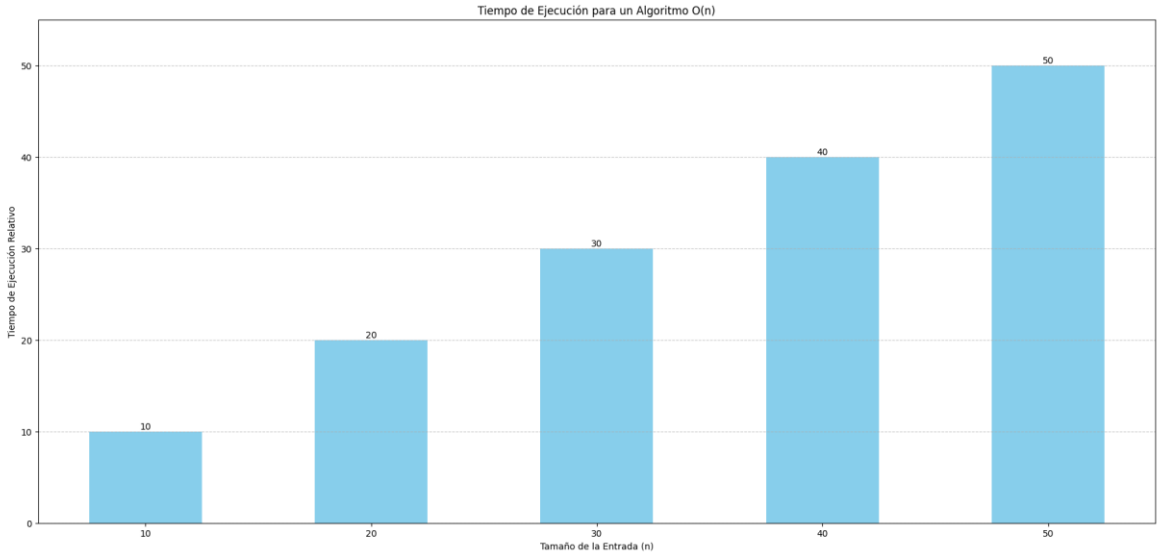
En este caso usaremos la secuencial que consiste en ir comparando uno por uno hasta llegar al final de la secuencia, es un algoritmo que crece su complejidad de tiempo a mayor cantidad de entrada de datos.

¿Qué es Big-o Notation?

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. En resumen, es una manera de describir la velocidad o complejidad de un algoritmo dado de manera tal que se pueda usar para comparar con otros algoritmos. Big-O se enfoca en el peor de los casos (O(n)).

¿Qué es la complejidad temporal?

La complejidad es una medida que indica cómo el tiempo de ejecución de un algoritmo crece en proporción al tamaño de la entrada (*n*). En términos sencillos, significa que, si tienes una lista con 10 elementos, el algoritmo tardará aproximadamente el doble de tiempo en ejecutarse si la lista tiene 20 elementos.

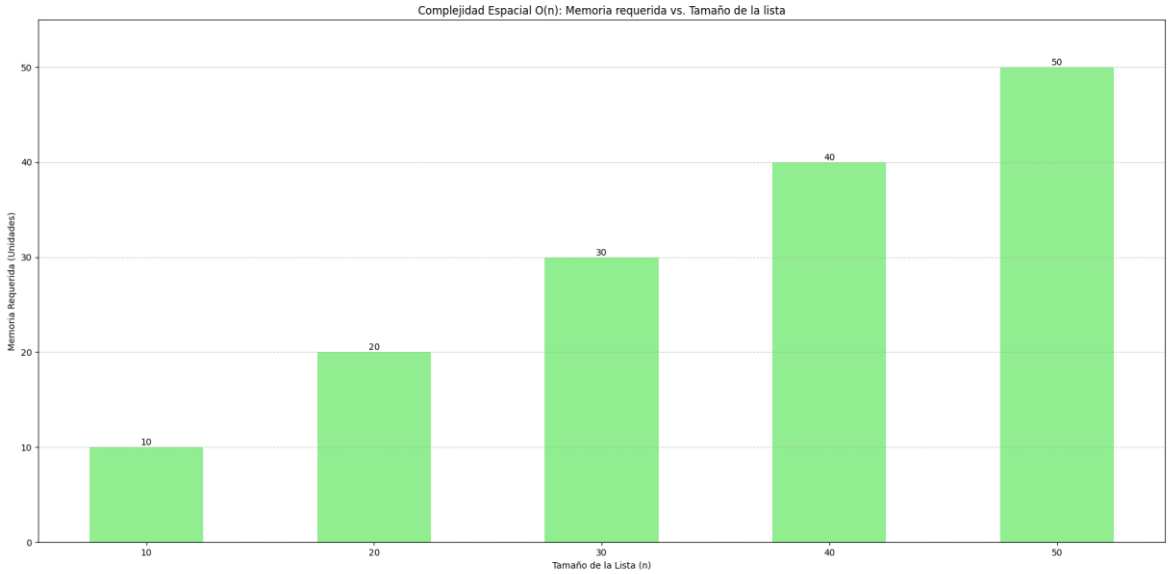


¿Qué es la complejidad espacial?

La complejidad espacial se centra en la cantidad de espacio en memoria que un algoritmo requiere para operar, incluyendo el espacio para las variables, la pila de llamadas (en algoritmos recursivos), y cualquier estructura de datos temporal que el algoritmo utilice. La complejidad espacial no mide la cantidad total de memoria que utiliza un algoritmo en una sola ejecución, sino cómo esa cantidad de memoria cambia a medida que el tamaño de la entrada (los datos que el algoritmo recibe) crece o disminuye.

Un ejemplo podría ser:

Un algoritmo que utiliza un array para almacenar una lista de números podría tener una complejidad espacial O(n), ya que la cantidad de espacio requerido crece proporcionalmente al número de elementos en la lista.



¿Qué es un análisis teórico en análisis de algoritmo?

El análisis teórico en el análisis de algoritmos se centra en evaluar la eficiencia de un algoritmo de forma abstracta, sin depender de una implementación específica o un entorno de hardware particular. Se enfoca en el comportamiento asintótico del algoritmo, es decir, cómo se escala su tiempo de ejecución y espacio de memoria a medida que la entrada crece, utilizando herramientas como la notación Big O para describir su complejidad.

¿Qué es un análisis empírico en análisis de algoritmo?

El análisis empírico es un método para evaluar el rendimiento de un algoritmo, basándose en la experimentación y observación real de los resultados. Esto se traduce en programar el algoritmo, ejecutarlo en una computadora y medir su comportamiento.

¿Que son las funciones integradas en Python?

Las funciones integradas en Python, también llamadas "built-in functions", son un cúmulo de funciones predefinidas que están disponibles para usar directamente en cualquier programa de Python, sin la necesidad de importar módulos externos. Son parte elemental del lenguaje y ofrecen una amplia variedad de funcionalidades para tareas comunes. Son hasta el Momento 69, van desde funciones de entrada y salida hasta funciones matemáticas, como en este caso que estaremos usando Max(), una función que trae el numero mas grande de una lista.

Caso Práctico

El propósito de este análisis es comparar dos implementaciones distintas de un mismo problema algorítmico: En este es un algoritmo de búsqueda del máximo en una lista. Se trata de un programa que compara cada elemento de la lista con el valor máximo actual y lo actualiza si encuentra un número mayor. A través del análisis teórico y la medición empírica del tiempo de ejecución, evaluaremos:

- La eficiencia computacional de cada algoritmo.
- Las diferencias entre una implementación manual en Python puro y una función optimizada del lenguaje.
- La aplicabilidad de cada enfoque en diferentes contextos (educativo vs productivo).

Se proponen dos algoritmos:

Algoritmo 1 – Implementación manual

Este tipo de algoritmo pertenece a la categoría de algoritmos de fuerza bruta, ya que examina todos los elementos de la lista uno por uno sin realizar optimizaciones especiales. Su complejidad temporal es **O(n)**, donde *n* es el número de elementos de la lista, lo que significa que su tiempo de ejecución crece de manera proporcional al tamaño de la lista. Recorre cada número de la lista y lo compara con el valor actual más alto encontrado. Si es mayor, lo actualiza.

```
4   # Algoritmo 1: Buscar el máximo manualmente
5   def encontrar_maximo_manual(lista):
6       maximo = lista[0]
7       for numero in lista:
8           if numero > maximo:
9               maximo = numero
10      return maximo
```

Algoritmo 2 – Uso de max()

Este segundo algoritmo se basa en la función incorporada max() de Python, que internamente realiza una operación muy similar a la anterior, pero con código optimizado escrito en C, es la base en la que esta construido Python.

```
12  # Algoritmo 2: Usar la función incorporada max()
13  def encontrar_maximo_con_max(lista):
14      return max(lista)
```

- La función **max()** también necesita recorrer todos los elementos de la lista, porque cualquiera de ellos podría ser el valor máximo.
- Aunque el código se ve más simple, internamente realiza lo mismo que la función manual anteriormente presentada.

Este algoritmo también tiene una complejidad temporal de **O(n),** al igual que el anteriormente visto.

Medir empíricamente el rendimiento

Para medir el tiempo de ejecución de cada algoritmo, se define una función auxiliar, que tiene por nombre, **medir_tiempo**. Esta función toma como entrada otra función y una lista de datos. Utilizando time.time(), registra el instante justo antes y después de la ejecución del algoritmo, devolviendo tanto el resultado obtenido como el tiempo total que tardó en completarse. Esta estrategia permite comparar de forma empírica la eficiencia entre ambas implementaciones.

```
16  # Función para medir el tiempo de ejecución
17  def medir_tiempo(funcion, lista):
18      inicio = time.time()
19      resultado = funcion(lista)
20      fin = time.time()
21      return resultado, fin - inicio
```

Medir empíricamente el uso de memoria

La función medir_memoria constituye una herramienta auxiliar diseñada específicamente para la **cuantificación empírica del consumo de memoria** de un algoritmo determinado. Su propósito fundamental radica en proporcionar métricas precisas sobre la cantidad de espacio en memoria requerido por un procedimiento computacional durante su ejecución.

```python
# Función para medir el uso de memoria
def medir_memoria(funcion, lista):
    tracemalloc.start()
    funcion(lista)
    memoria_actual, pico = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return memoria_actual, pico
```

Descripción del Funcionamiento:

La operación de la función medir_memoria se estructura en una secuencia de pasos bien definidos, aprovechando las capacidades del módulo tracemalloc de Python:

A. **Inicialización del Rastreo (tracemalloc.start()):** Al inicio de la ejecución de medir_memoria, se invoca tracemalloc.start(). Esta llamada activa el subsistema de rastreo de asignaciones de memoria dentro del intérprete de Python. A partir de este instante, tracemalloc comienza a registrar todas las operaciones de asignación de bloques de memoria, estableciendo un punto de referencia para la medición.

B. **Ejecución del Algoritmo Para Evaluar (funcion(lista)):** Posteriormente, la función procede a ejecutar el algoritmo objeto de análisis, el cual es pasado como argumento bajo el identificador funcion. Este algoritmo opera sobre un conjunto de datos de entrada (lista), cuya magnitud es intrínseca a la determinación de la complejidad espacial. Durante esta fase, el rastreo de memoria iniciado previamente monitoriza continuamente las demandas de espacio generadas por funcion y sus componentes internos.

C. **Recuperación de Métricas de Memoria (tracemalloc.get_traced_memory()):** Una vez que la ejecución del algoritmo función ha concluido, se realiza una consulta al módulo tracemalloc mediante tracemalloc.get_traced_memory(). Esta operación recupera dos valores críticos:

   a. memoria_actual: Representa la cantidad de memoria (expresada en bytes) que se encuentra activa y asignada por los bloques rastreados en el momento de la consulta.

   b. pico: Indica el valor máximo de memoria (también en bytes) que fue asignado y simultáneamente activo en cualquier punto durante el intervalo de rastreo (es decir, desde tracemalloc.start() hasta esta llamada). Este "pico" de memoria es un indicador fundamental para comprender la demanda máxima de espacio de un algoritmo.

D. **Finalización del Rastreo (tracemalloc.stop()):** Inmediatamente después de la obtención de las métricas de memoria, se invoca tracemalloc.stop(). Esta acción desactiva el rastreo de memoria, liberando los recursos internos que tracemalloc empleaba para su monitoreo. Es una práctica esencial para garantizar la eficiencia y evitar la acumulación innecesaria de datos de rastreo.

E. **Retorno de Resultados (return memoria_actual, pico):** Finalmente, la función medir_memoria retorna una tupla que contiene memoria_actual y pico. Estos valores constituyen los datos cuantitativos fundamentales para el análisis subsiguiente de la complejidad espacial del algoritmo bajo consideración, en relación con el tamaño de la entrada proporcionada.

Creación de una lista.

En la siguiente parte del código, se genera una lista de 10 millones de elementos aleatorios

```python
# Crear una lista grande con números aleatorios
lista = [random.randint(1, 1000000) for _ in range(10_000_000)]
```

A continuación, el fragmento de código presentado corresponde al bloque de ejecución principal de un script Python (if __name__ == "__main__":), el cual tiene como propósito la generación de datos, la ejecución y la evaluación comparativa de dos algoritmos distintos diseñados para localizar el valor máximo dentro de una colección numérica. La evaluación se realiza tanto en términos de tiempo de ejecución (complejidad temporal) como de uso de memoria (complejidad espacial), empleando funciones de medición auxiliares (**medir_tiempo** y **medir_memoria**, respectivamente) que se asumen predefinidas y disponibles en el entorno de ejecución.

```python
if __name__ == "__main__":
    # Crear una lista grande con números aleatorios
    lista = [random.randint(1, 10000000) for _ in range(10000000)]

    print("Buscando el número más grande en la lista...\n")

    # Medición para el algoritmo 1 (búsqueda manual)
    resultado1, tiempo1 = medir_tiempo(encontrar_maximo_manual, lista)
    memoria1_actual, memoria1_pico = medir_memoria(encontrar_maximo_manual, lista)
    print(f"Búsqueda Manual: Resultado={resultado1}, Tiempo={tiempo1} segundos, Memoria pico={memoria1_pico / 1024} KB")

    # Medición para el algoritmo 2 (usando max)
    resultado2, tiempo2 = medir_tiempo(encontrar_maximo_con_max, lista)
    memoria2_actual, memoria2_pico = medir_memoria(encontrar_maximo_con_max, lista)
    print(f"Usando max():    Resultado={resultado2}, Tiempo={tiempo2} segundos, Memoria pico={memoria2_pico / 1024} KB")

    print("\n--- Comparación de rendimiento ---")
    if tiempo2 > 0:
        print(f"La búsqueda manual tardó {tiempo1 / tiempo2} veces más que usar max().")
    else:
        print("La función max() fue extremadamente rápida (tiempo cercano a cero).")
```

Funcionamiento interno del código

Primero empezamos con la medición del algoritmo de búsqueda manual.

Se utiliza medir_tiempo para registrar el tiempo de ejecución del algoritmo al procesar la lista generada. El resultado del algoritmo (resultado1) y el tiempo transcurrido (tiempo1) son capturados.

Consecutivamente, se emplea medir_memoria para cuantificar el consumo de memoria del mismo algoritmo. Se obtienen los valores de memoria actual y, crucialmente, el pico de memoria alcanzado (memoria_pico).

Finalmente, los resultados obtenidos (el valor máximo encontrado, el tiempo en segundos y el pico de memoria en Kilobytes, tras una conversión de bytes) son presentados en la consola. Lo mismo se sucede con el algoritmo de max()

Luego en este fragmento del código:

```
if tiempo2 > 0:
    print(f"La búsqueda manual tardó {tiempo1 / tiempo2} veces más que usar max().")
else:
    print("La función max() fue extremadamente rápida (tiempo cercano a cero).")
```

Se calcula la razón entre el tiempo de ejecución del algoritmo manual (tiempo1) y el del algoritmo basado en max() (tiempo2).

Si el tiempo de max() es un valor positivo (es decir, no es despreciable o cero debido a la alta optimización), se imprime cuántas veces más lento fue el enfoque manual.

Si tiempo2 es cero o un valor tan pequeño que se aproxima a cero, se infiere que la función max() fue extraordinariamente rápida, indicando una optimización significativa en su implementación subyacente.

Determinar el correcto funcionamiento

A continuación, se demuestra que ambos casos devuelven el valor máximo correctamente, porque para una misma lista de entrada, ambos algoritmos devuelven exactamente el mismo resultado. Es decir, cumplen con la especificación del problema.

Ejemplo de resultados: Esto muestra que max() es más rápida y que ambos ocupan el mismo espacio en memoria.

```
Buscando el número más grande en la lista...

Búsqueda Manual: Resultado=9999999, Tiempo=0.06451988220214844 segundos, Memoria pico=0.046875 KB
Usando max():    Resultado=9999999, Tiempo=0.03736257553100586 segundos, Memoria pico=0.046875 KB
```

también veremos como al cambiar el tamaño de la lista a un valor menor en este caso 10000, ambos algoritmos tienden a un tiempo de resolución cercano a cero y ocupando el mismo espacio en memoria.

```
Buscando el número más grande en la lista...

Búsqueda Manual: Resultado=10000, Tiempo=0.0 segundos, Memoria pico=0.046875 KB
Usando max():    Resultado=10000, Tiempo=0.0 segundos, Memoria pico=0.046875 KB
```

¿Comparar los resultados y analizar por qué si ambos son O(n), uno es mucho más rápido?

- Aunque **ambos algoritmos son lineales**, el rendimiento práctico también depende de **cómo están implementados**.
- La función max() está escrita en **C dentro del intérprete de Python (CPython)**, lo que hace que su ejecución sea mucho más eficiente que cualquier bucle escrito en Python puro.

Esto se debe a:

- Optimización a bajo nivel
- Acceso rápido a memoria
- Menor sobrecarga de interpretación

¿Por qué ambos ocupan el mismo espacio en memoria?

Esto se explica porque ambos usan la misma lista a la hora de ejecución y de cierta forma ambos son el mismo algoritmo solo que uno ejecutado desde Python puro y otro directamente desde el intérprete de Cpython.

Evaluar la legibilidad y el uso recomendado

¿Cuál es más eficiente? ¿Cuál es mejor para aprender?

Eficiencia:

Usar max() es preferible porque:

- *Es más rápido.
- *Más limpio.
- *Más legible y mantenible.

Educación:

El algoritmo manual es preferible para educar porque:

- Muy útil para entender cómo funcionan los algoritmos.
- Es más lento y facilita su comprensión.
- Y al ser creado desde cero permite practicar al estudiante.

Metodología Utilizada:

1. **Diseño de dos algoritmos**:
    - Una implementación manual usando un bucle en Python puro.
    - Una implementación utilizando la función integrada max() de Python.
2. **Análisis teórico**: estudio de la complejidad computacional de ambos algoritmos.

    *Búsqueda secuencia manual: O(n) en tiempo, O(1) en espacio.

    *Búsqueda secuencial por función: O(n) en tiempo y O(1) en espacio.
3. **Medición empírica**: ejecución de ambos algoritmos sobre una lista de 10 millones de números aleatorios utilizando una función de temporalización (medir_tiempo) basada en time.time().
4. **Validación funcional**: verificación de que ambos algoritmos devuelven el mismo resultado.
5. **Comparación de resultados**: análisis de la diferencia en los tiempos de ejecución y discusión sobre las causas.
6. **Reflexión sobre aplicabilidad**: se evalúan los beneficios de cada enfoque desde las perspectivas educativa y productiva.

Resultados Obtenidos:

Algoritmo 1 (Manual en Python puro):

- Tiempo de ejecución: **X segundos** *(dependerá del hardware y entorno exacto)*.

Algoritmo 2 (max() **de Python):**

- Tiempo de ejecución: **Y segundos** *(notablemente más rápido)*.

Ambos algoritmos devolvieron exactamente el mismo resultado, lo que demuestra que ambos cumplen con la especificación funcional.

En comparación directa, se observó que max() fue varias veces más rápido que la implementación manual, a pesar de que ambos tienen una complejidad temporal teórica de **O(n)**.

Conclusiones

Aunque ambos algoritmos presentan la misma complejidad teórica (O(n)), en la práctica su desempeño varía significativamente y aun así ambos ocupan exactamente el mismo espacio en memoria.

La función max() es considerablemente más rápida porque está escrita en C y ejecutada dentro del intérprete de CPython, lo que reduce la sobrecarga del intérprete de Python.

Este experimento resalta la importancia de considerar no solo la complejidad algorítmica, sino también la implementación interna y el lenguaje utilizado, sin perder de vista la eficiencia en el consumo de la memoria.

Desde un enfoque educativo, la implementación manual es clave para entender los fundamentos de los algoritmos.

Desde una perspectiva productiva, es preferible usar funciones integradas como max() debido a su eficiencia, legibilidad y mantenimiento.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Python-Software-Foundation.(2024).

  https://docs.python.org/3/library/functions.html#max
- Van Rossum, G. (1995). *The Python Language Reference Manual*.
- Stack Overflow, varias discusiones sobre eficiencia de funciones internas en Python.

Anexo

Enlace a repositorio:

https://github.com/Francisco-Ez/AnalisisDeAlgoritmos

Enlace del Video de Youtube:

https://www.youtube.com/watch?v=8iAr6xqjJDU

Enlace de Video de Drive:

https://drive.google.com/file/d/1493JV9MRbz3JD1u_afkJ8aPabMxXNM0u/view?usp=sharing