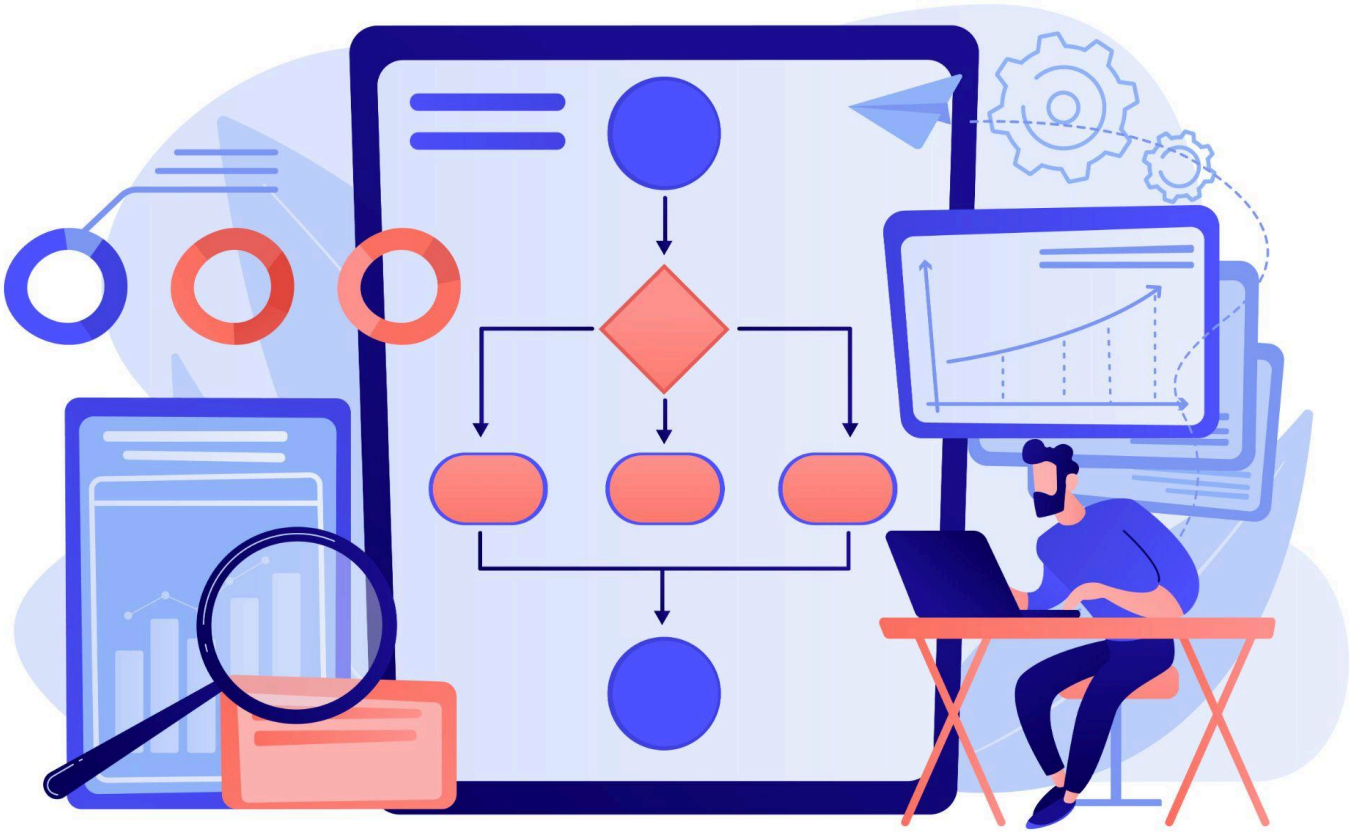
ANÁLISIS DE ALGORITMOS



Alumnos:
Herrera Florencia (herreraflorencia874@gmail.com)
Quinteros Federico (fedequinterod30@gmail.com)
Materia: Programación I
Profesor: Sebastián Bruselario
Tutora: Virginia Cimino
Fecha de Entrega: 9 de mayo de 2025

ÍNDICE

Introducción

Este trabajo práctico tiene como objetivo analizar y comparar dos algoritmos distintos utilizados para encontrar el número más grande en una lista. La elección de este tema surgió del interés de saber cómo dos algoritmos pueden resolver un mismo problema con distintos niveles de eficiencia y claridad. Además,nos permite profundizar en conceptos fundamentales de la programación como el uso de estructuras de control, eficiencia computacional y análisis de algoritmos.

La importancia de esto en programación es que la búsqueda de valores extremos (como el mayor o el menor) es una operación básica y frecuente en numerosos programas y aplicaciones. Comprender distintas formas de abordarla no solo fortalece el pensamiento lógico, sino que también permite al programador tomar decisiones más acertadas en función del contexto o las necesidades del programa, como la optimización del tiempo de ejecución o la claridad del código.

Con el desarrollo de este trabajo buscamos alcanzar los siguientes objetivos:

- Comprender el funcionamiento de dos algoritmos diferentes para resolver el mismo problema.
- Comparar su eficiencia en términos de cantidad de pasos o instrucciones necesarias.

Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es un conjunto finito, ordenado y bien definido de pasos o instrucciones que permiten resolver un problema o realizar una tarea específica. Estos pasos se diseñan de manera lógica para que, al ejecutarse, conduzcan a un resultado deseado a partir de ciertos datos de entrada.

Análisis de Algoritmos

El análisis de algoritmos es una herramienta para hacer la evaluación del diseño de un algoritmo, permite establecer la calidad de un programa y compararlo con otros que puedan resolver el mismo problema, sin necesidad de desarrollarlos. El análisis de algoritmos estudia, desde un punto de vista teórico, los recursos computacionales que requiere la ejecución de un programa, es decir su eficiencia (tiempo de CPU, uso de memoria, ancho de banda, etc). Además de la eficiencia en el desarrollo de software existen otros factores igualmente relevantes: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad y aún el propio costo de programación.

El análisis de algoritmos se basa en:

- El análisis de las características estructurales del algoritmo que respalda el programa.
- La cantidad de memoria que utiliza para resolver un problema.
- La evaluación del diseño de las estructuras de datos del programa, midiendo la eficiencia de los algoritmos para resolver el problema planteado

Determinar la eficiencia de un algoritmo nos permite establecer lo que es factible en la implementación de una solución de lo que es imposible.

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

Los criterios para evaluar programas son diversos: eficiencia, portabilidad, eficacia, robustez, etc. El análisis de complejidad está relacionado con la eficiencia del programa. La eficiencia mide el uso de los recursos del computador por un algoritmo. Por su parte, el análisis de complejidad mide el tiempo de cálculo para ejecutar las operaciones (complejidad en tiempo) y el espacio de memoria para contener y manipular el programa más los datos (complejidad en espacio). Así, el objetivo del análisis de complejidad es

cuantificar las medidas físicas: tiempo de ejecución y espacio de memoria y comparar distintos algoritmos que resuelven un mismo problema.

El tiempo de ejecución de un programa depende de factores como:

- Los datos de entrada del programa.
- La calidad del código objeto generado por el compilador.
- La naturaleza y rapidez de las instrucciones de máquina utilizadas.
- La complejidad en tiempo del algoritmo base del programa.

El tiempo de ejecución debe definirse como una función que depende de la entrada; en particular, de su tamaño. El tiempo requerido por un algoritmo expresado como una función del tamaño de la entrada del problema se denomina complejidad en tiempo del algoritmo y se denota T(n). El comportamiento límite de la complejidad a medida que crece el tamaño del problema se denomina complejidad en tiempo asintótica. De manera análoga se pueden establecer definiciones para la complejidad en espacio y la complejidad en espacio asintótica.

El tiempo de ejecución depende de diversos factores. Se tomará como más relevante el relacionado con la entrada de datos del programa, asociándo se a un problema entero llamado tamaño del problema, el cual es una medida de la cantidad de datos de entrada.

Tenemos dos tipos de análisis:

Análisis Empírico: El análisis empírico consiste en medir el rendimiento real de un algoritmo, ejecutándose en la práctica con distintos datos de entrada, y observando cuánto tiempo tarda y cuanta memoria usa en la computadora.

Análisis Teórico: Consiste en evaluar su eficiencia (en tiempo y espacio) utilizando fórmulas matemáticas, expresiones simbólicas (como notación Big O), y sin necesidad de implementarlo o probarlo en una computadora.

¿Qué es Big O?

Big O notation o notación Big O es una forma de medir la eficiencia de un algoritmo en términos de cuánto tiempo o espacio necesita para ejecutarse en función del tamaño de la entrada.

La clave es la relación entre el tiempo o espacio y el tamaño de la entrada.

¿Por qué Big O notation es importante?
Cuando desarrollamos una aplicación, es probable que se tenga que trabajar con grandes cantidades de datos. Por ejemplo, una base de datos con millones de registros o un archivo de texto con miles de líneas. En estos casos, la eficiencia de tus algoritmos es crucial para que tu aplicación funcione correctamente y no se quede colgada.
Big O ayuda a:
Evaluar la eficiencia de tus algoritmos: ¿Cuánto tiempo o espacio necesita tu algoritmo para ejecutarse en función del tamaño de la entrada?

Comparar algoritmos: ¿Cuál es el algoritmo más eficiente para resolver un problema?
Optimizar tu código: ¿Cómo puedes mejorar la eficiencia de tu algoritmo?

Cualquier milisegundo o incluso micro segundo que puedas ahorrar en la ejecución de tu algoritmo, puede hacer una gran diferencia en la experiencia del usuario.

Las notaciones de Big O más comunes son:

- O(1) - Notación constante
- O(log n) - Notación logarítmica
- O(n) - Notación lineal
- O(n log n) - Notación lineal-logarítmica
- O(n^2) - Notación cuadrática
- O(2^n) - Notación exponencial
- O(n!) - Notación factorial

Caso Práctico

COMPARACIÓN DE DOS ALGORITMOS PARA ENCONTRAR EL NÚMERO MÁS GRANDE EN UNA LISTA

Realizamos la comparación teórica de 2 algoritmos cuya finalidad es encontrar el número de mayor valor dentro de una lista.

Código principal

```
import time
import random

numeros0 = [random.randint(1, 1000000) for i in range(1000)]
numeros1 = [random.randint(1, 1000000) for i in range(10000)]
numeros2 = [random.randint(1, 1000000) for i in range(100000)]
numeros3 = [random.randint(1, 1000000) for i in range(1000000)]
numeros4 = [random.randint(1, 1000000) for i in range(10000000)]
numeros = [numeros0,numeros1,numeros2,numeros3,numeros4]

# Algoritmo 1: Decimos que es un algoritmo ‘’manual’’ porque recorre y compara cada elemento de la lista para luego darnos un resultado final.
Condicional reemplazando la variable “número” siempre que se encuentre un número mayor.

def max_manual(lista):
    if not lista:                #1
        return None              #1
    maximo = lista[0]            #1
    for numero in lista:         #2N          T(n)= 1+1+1+1+(2N) = 4+(2*n)
        if numero > maximo:      #1
```

```
            maximo = numero         #1
    return maximo                   #1
```

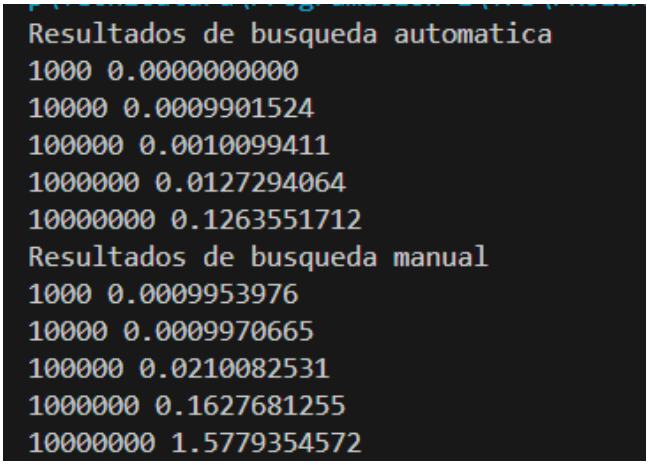
Algoritmo 2: Función max(). Decimos que es un algoritmo ‘’automático’’ ya que devuelve el número mayor.

```
def max_auto(lista):
    if not lista:                   #1              T(n)= 1+1+2 = 4
        return None                 #1
    return max(lista)               #2

print("Resultados de busqueda automatica")

for i in range(5):
    inicio = time.time()
    maximo_auto = max_auto(numeros[i])
    tiempo = time.time() - inicio
    print(f"{10**(i+3)} {tiempo:.10f}")

print("Resultados de busqueda manual")
for i in range(5):
    inicio = time.time()
    maximo_manual = max_manual(numeros[i])
    tiempo = time.time() - inicio
    print(f"{10**(i+3)} {tiempo:.10f}")
```

Tipos de análisis

Análisis empírico:

Realizamos la medición de los tiempos de ejecución de cada algoritmo, a través de la función time, comparando los tiempos con las diferentes listas, que van desde 1.000 a 10.000.000 de elementos. Obteniendo como output los siguientes resultados:



Análisis Teórico:

Determinamos la función T(n) de cada algoritmo para establecer la cantidad de operaciones que realiza cada uno, para una entrada de tamaño n.

Algoritmo manual: T(n)= 1+1+1+1+(2N) = 4+(2*n)

Algoritmo automático: T(n)= 1+1+2 = 4

Notación Big O:

Tomamos la entrada que más crece de cada algoritmo:

Algoritmo manual = 2N
Algoritmo automático = 2

Quitamos el valor constante que acompaña a N. Quedando la notación del algoritmo automático O(2). Por lo tanto, es una notación constante y la notación del algoritmo manual O(n) es una notación lineal.

Metodología Utilizada

• Análisis teórico de la complejidad de cada algoritmo.
• Generación de datos:Se generaron cinco listas de números enteros aleatorios, con tamaños crecientes Los valores fueron generados de forma aleatoria dentro del rango de 1 a 1.000.000.
• Para cada lista, se midió el tiempo de ejecución de cada función usando el módulo time.
• Los resultados se registraron en la terminal y se compararon para observar las diferencias de rendimiento entre ambas funciones.
• Comparación de resultados de tiempo real.
• Documentación del proceso en repositorio GitHub.

Resultados Obtenidos

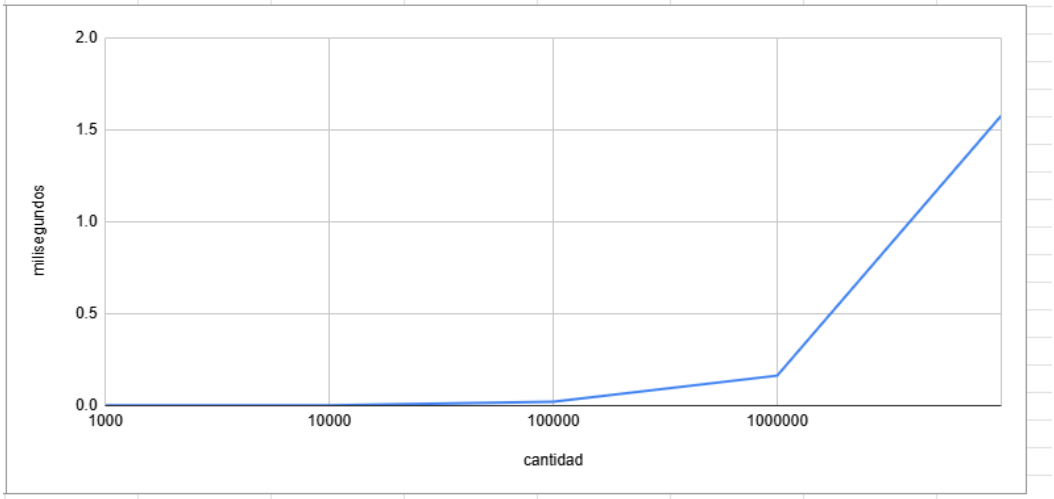
• Ambos algoritmos devuelven el mismo resultado correcto.
• La opción automática es mucho más rápida y eficiente que la manual a medida que aumenta el tamaño de la lista.

Resultados de la búsqueda manual:

Resultados de búsqueda manual

Tabla_2

cantidad	milisegund
1000	0.0009953976
10000	0.0009970665
100000	0.0210082531
1000000	0.1627681255
10000000	1.577935457

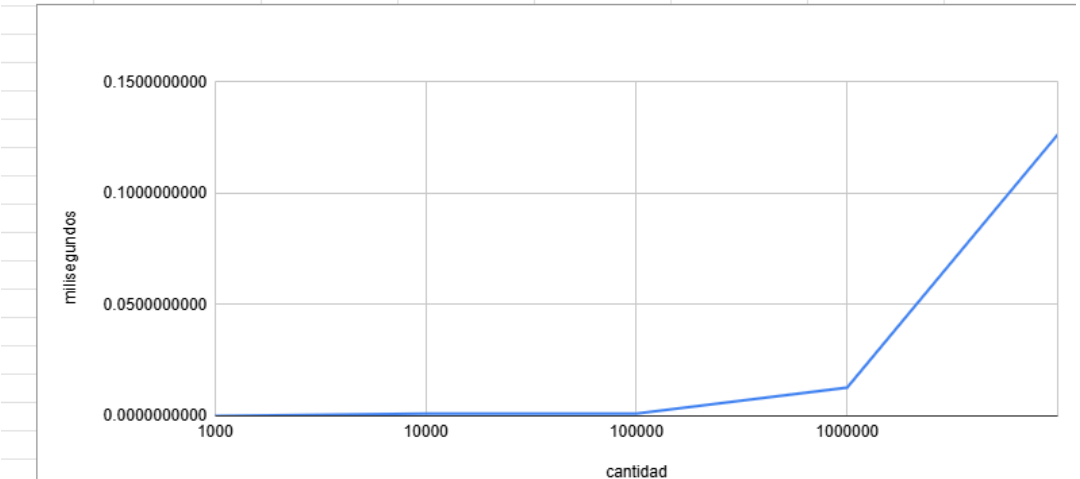


Resultados de la búsqueda automática:

Resultados de búsqueda automática

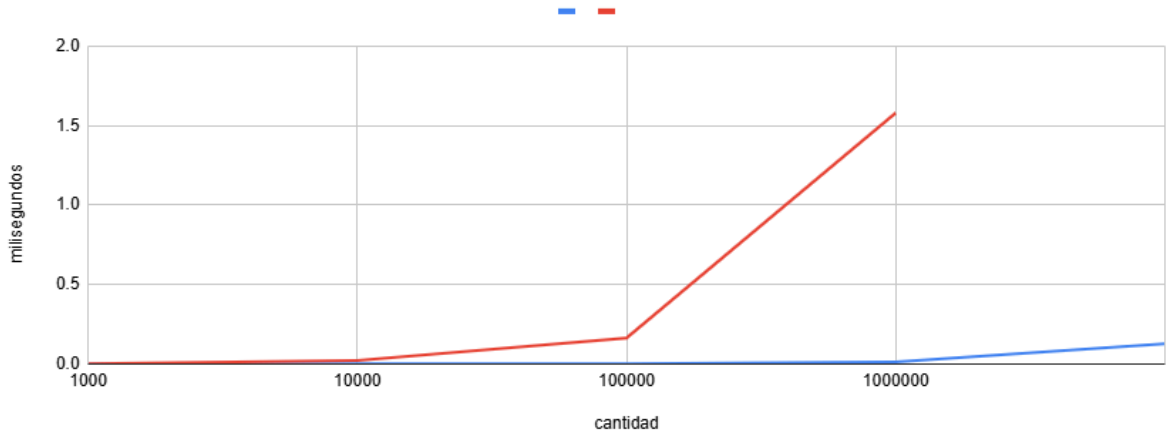
Tabla_1

cantidad	milisegund
1000	0.0000000000
10000	0.0009901524
100000	0.0010099411
1000000	0.0127294064
10000000	0.1263551712



Comparación de resultados

- Algoritmo manual
- Algoritmo automático



Conclusiones

En este trabajo se compararon dos algoritmos para resolver el mismo problema: encontrar el número más grande en una lista. Uno de ellos fue implementado manualmente mediante un recorrido con un bucle for, mientras que el otro utilizó la función automática max() de Python.

Los resultados mostraron que, si bien ambos algoritmos tienen una complejidad lineal O(n) y realizan la misma tarea, la función max() fue consistentemente más rápida. Esto se debe a que está optimizada a nivel interno en el lenguaje Python, lo que le permite realizar la operación con mayor eficiencia.

En conclusión, siempre que sea posible, es recomendable utilizar funciones integradas del lenguaje como max(), ya que no sólo simplifican el código, sino que además ofrecen un mejor rendimiento, especialmente en estructuras de datos grandes. Este tipo de comparaciones permite desarrollar un pensamiento crítico sobre la eficiencia de distintas soluciones y tomar decisiones fundamentadas a la hora de programar.

Bibliografía

• https://developero.io/blog/big-o
• http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm
• Learn Big-O Notation and Algorithm Analysis with Examples.
•https://ude.edu.uy/que-son-algoritmos/#:~:text=Se%20puede%20entender%20un%20algoritmo.pueden%20ver%20como%20un%20algoritmo.
• Contenido del Aula Virtual

Anexos

Repositorio en GitHub:
https://github.com/Federico-Quinteros/TP-Integrador-Programacion-1.git
Video explicativo :
https://drive.google.com/file/d/1h5RVDvTrbPZndBHCH4-8uywDdsb5efTA/view?usp=drive_link