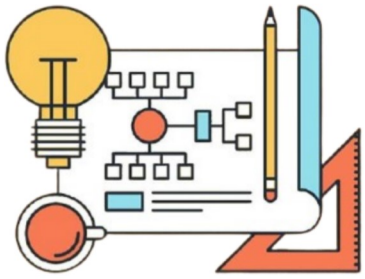
Trabajo Práctico Integrador

Análisis de algoritmos aplicado a algoritmos de ordenamiento



Materia: Programación I
Profesor: Ariel Enferrel
Tutor: Luciano Chiroli

Alumnos:
Maria Sol Couchot – solcouchot@gmail.com
Nicolas Macaris – nicolas.macaris@gmail.com

Fecha de Entrega: 09 de junio de 2025

Índice

Introducción

En este trabajo práctico se aborda el análisis de algoritmos, con un caso práctico aplicado a algoritmos clásicos de ordenamiento de datos.
A través de ejemplos simples, se busca entender cómo el rendimiento de un algoritmo puede variar en función de la cantidad de datos procesados.
El objetivo principal es comparar distintos enfoques para resolver un mismo problema (en este caso, ordenar una lista), analizando los resultados obtenidos y comprendiendo la importancia de elegir un algoritmo adecuado según el contexto.
Este análisis se realiza sobre tres algoritmos: Bubble Sort, Insertion Sort y Merge Sort.

Marco Teórico

Algoritmo: Los algoritmos son conjuntos estructurados de instrucciones diseñadas para resolver problemas específicos o realizar tareas concretas. Funcionan mediante una serie de pasos bien definidos, cada uno de los cuales contribuye al objetivo final. Debido a que hay varias maneras de resolver un problema, debe haber una manera de evaluar estas soluciones o algoritmos en términos de rendimiento y eficiencia (el tiempo que tardará tu algoritmo en ejecutarse y la cantidad total de memoria que consumirá).

Esto es fundamental para que los programadores se aseguren de que sus aplicaciones se ejecuten correctamente y les ayuden a escribir código limpio.

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, ...).
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.

Big O: es una métrica para determinar la eficiencia de un algoritmo. Le permite estimar cuánto tiempo se ejecutará su código en diferentes conjuntos de entradas y medir la eficacia con la que su código escala a medida que aumenta el tamaño de su entrada. Representa la complejidad del peor de los casos de un algoritmo. Utiliza términos algebraicos para describir la complejidad de un algoritmo.

Big O define el tiempo de ejecución necesario para ejecutar un algoritmo identificando cómo cambiará el rendimiento de su algoritmo a medida que crece el tamaño de la entrada. Pero no le dice qué tan rápido es el tiempo de ejecución de su algoritmo.

La notación Big O mide la eficiencia y el rendimiento de su algoritmo usando la complejidad del tiempo y el espacio.

Ejemplos de casos:

O(n): el tiempo crece en forma proporcional a la cantidad de datos.

O(n^2): el tiempo crece mucho más rápido (por ejemplo, el doble de datos → cuatro veces más tiempo).

O(log n): crece lentamente, es más eficiente.

Algoritmo de ordenamiento por burbuja: Bubble sort. Hace múltiples pasadas a lo largo de una lista. Compara los ítems adyacentes e intercambia los que no están en orden. Cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada ítem “burbujea” hasta el lugar al que pertenece.

Algoritmo de ordenamiento por inserción: Insertion sort. Su funcionamiento consiste en el recorrido por la lista seleccionando en cada iteración un valor como clave y compararlo con el resto insertándose en el lugar correspondiente.

Algoritmo de ordenamiento por mezcla: Merge sort. Divide el conjunto de datos en mitades más pequeñas, las ordena y luego las fusiona de nuevo en una sola lista ordenada.

Caso Práctico

- Se implementan los algoritmos de ordenamiento.
- Se compara el rendimiento de los tres algoritmos al ordenar listas de distintos tamaños.
- Cada algoritmo se corre 5 veces sobre la misma lista y se saca el promedio de tiempo de ejecución.

```
import time
import random

# Bubble sort
# https://www.w3schools.com/python/python_dsa_bubblesort.asp
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Insertion sort
# https://www.w3schools.com/python/python_dsa_insertionsort.asp
def insertion_sort(arr):
    n = len(arr)
    for i in range(1,n):
        insert_index = i
        current_value = arr.pop(i)
        for j in range(i-1, -1, -1):
            if arr[j] > current_value:
                insert_index = j
        arr.insert(insert_index, current_value)
```

```python
# Merge sort
# https://www.w3schools.com/python/python_dsa_mergesort.asp
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    leftHalf = arr[:mid]
    rightHalf = arr[mid:]

    sortedLeft = merge_sort(leftHalf)
    sortedRight = merge_sort(rightHalf)

    return merge(sortedLeft, sortedRight)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

```python
# Función para medir tiempos
# utilizamos time.perf_counter() para medir el tiempo de ejecución porque es más preciso
# repetimos la ejecucion 5 veces y calculamos el promedio
def medir_tiempo_promedio(func, datos, repeticiones=5):
    total = 0
    for _ in range(repeticiones):
        lista_copia = datos.copy()
        inicio = time.perf_counter()
        func(lista_copia)
        fin = time.perf_counter()
        total += (fin - inicio)
    return total / repeticiones

# Creación de listas aleatorias y llamado a funciones
# se formatea a 4 decimales
tamaño = 1000
datos = [random.randint(1, 10000) for _ in range(tamaño)]

print(f"Tamaño de lista: {tamaño}")

tiempo_promedio_bubble = medir_tiempo_promedio(bubble_sort, datos)
print(f"Bubble Sort: {tiempo_promedio_bubble:.4f} segundos")

tiempo_promedio_insertion = medir_tiempo_promedio(insertion_sort, datos)
print(f"Insertion Sort: {tiempo_promedio_insertion:.4f} segundos")

tiempo_promedio_merge = medir_tiempo_promedio(merge_sort, datos)
print(f"Merge Sort: {tiempo_promedio_merge:.4f} segundos")
```

Resultado:

```
PS C:\Users\Nicolás\Downloads\py\repos\TPI-Programacion1-Analisis-de-Algoritmos>
/py/repos/TPI-Programacion1-Analisis-de-Algoritmos/analisis-de-algoritmos.py
Tamaño de lista: 100
Bubble Sort: 0.0002 segundos
Insertion Sort: 0.0001 segundos
Merge Sort: 0.0001 segundos
PS C:\Users\Nicolás\Downloads\py\repos\TPI-Programacion1-Analisis-de-Algoritmos>
/py/repos/TPI-Programacion1-Analisis-de-Algoritmos/analisis-de-algoritmos.py
Tamaño de lista: 500
Bubble Sort: 0.0061 segundos
Insertion Sort: 0.0025 segundos
Merge Sort: 0.0005 segundos
PS C:\Users\Nicolás\Downloads\py\repos\TPI-Programacion1-Analisis-de-Algoritmos>
/py/repos/TPI-Programacion1-Analisis-de-Algoritmos/analisis-de-algoritmos.py
Tamaño de lista: 1000
Bubble Sort: 0.0262 segundos
Insertion Sort: 0.0100 segundos
Merge Sort: 0.0011 segundos
PS C:\Users\Nicolás\Downloads\py\repos\TPI-Programacion1-Analisis-de-Algoritmos>
```

¿Por qué utilizamos perf_counter() y no time()?

- Es más preciso para medir tiempos muy cortos.
- A diferencia de time(), está diseñado para medir duraciones, no para obtener la hora actual.
- Permite diferenciar mejor la eficiencia de algoritmos rápidos como merge_sort, que con time() a veces aparecen con tiempo 0.

Ejecución con **time()**:

```
Tamaño de lista: 100
Bubble Sort: 0.0002 segundos
Insertion Sort: 0.0002 segundos
Merge Sort: 0.0000 segundos
```

¿Por qué ejecutamos cada algoritmo varias veces?

Las mediciones pueden variar de una ejecución a otra. Esto pasa por varias razones, entre otras:

- El sistema operativo puede estar haciendo otras tareas al mismo tiempo (actualizaciones, antivirus, etc.),
- El procesador puede estar más (o menos) ocupado.

Por esto decidimos:

1. Ejecutar cada algoritmo 5 veces con la misma lista.
2. Medir cuánto tardó en cada ejecución.
3. Calcular el promedio de esos 5 tiempos.

De esta manera, evitamos una sola medición poco precisa.

Metodología Utilizada

➔ Se implementaron los algoritmos de ordenamiento en python
  - Bubble sort
  - Insertion sort
  - Merge sort

➔ Se implementó función para medir el tiempo de ejecución promedio (5 ejecuciones)

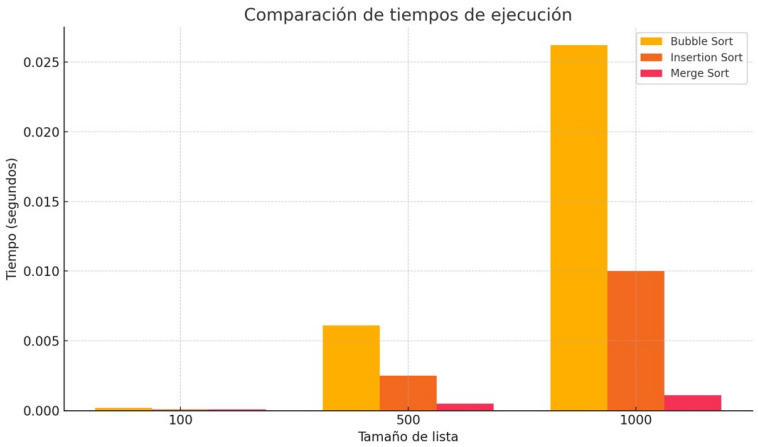
➔ Se generaron listas de distintos tamaños con números aleatorios.

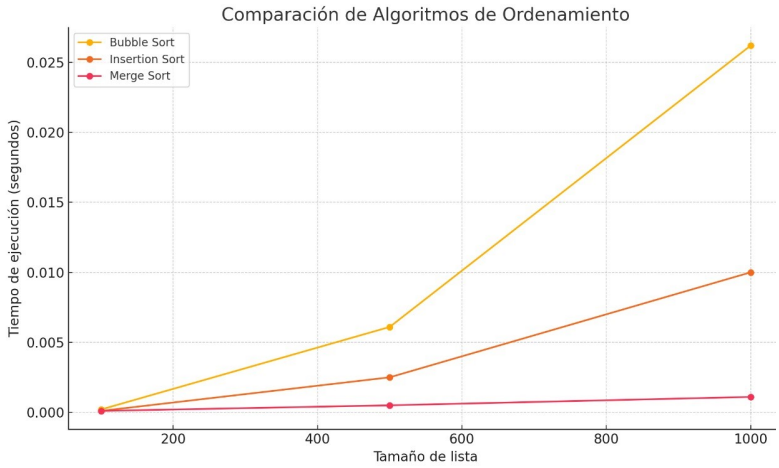
➔ Se midió el tiempo de ejecución de cada algoritmo.

➔ Se compararon los resultados.

Resultados Obtenidos

Tamaño de lista	Bubble Sort (s)	Insertion Sort (s)	Merge Sort (s)
100	0.0002 segundos	0.0001 segundos	0.0001 segundos
500	0.0061 segundos	0.0025 segundos	0.0005 segundos
1000	0.0262 segundos	0.0100 segundos	0.0011 segundos





- **Bubble sort** fue el más lento en todas las pruebas, lo cual es coherente con la ineficiencia de su método de comparación repetitiva
- **Insertion Sort** mostró mejor desempeño que Bubble Sort, especialmente a partir de 500 elementos. (*Aunque ambos son O(n²), Insertion Sort es más eficiente que Bubble Sort cuando la lista ya está ordenada o casi ordenada, o cuando la cantidad de datos es pequeña.)*
- **Merge Sort** fue significativamente más rápido en todas las pruebas, gracias a su complejidad O(n log n).

Conclusiones

En este trabajo práctico pudimos observar cómo diferentes algoritmos resuelven un mismo problema (en este caso, ordenar una lista), pero con rendimientos muy distintos dependiendo del tamaño de los datos y del tipo de algoritmo utilizado.

Aunque Bubble Sort e Insertion Sort funcionaron correctamente, sus tiempos de ejecución crecieron significativamente con el tamaño de la lista. Algo que se notó mucho cuando hicimos una prueba con una lista de 10.000 items.

Esto demuestra que *no todos los algoritmos son igual de eficientes, especialmente con listas más grandes.*

- **Merge Sort** resultó mucho más eficiente que los otros dos métodos, especialmente con listas de tamaño 500 y 1000.
- A pesar de tener la misma complejidad que Bubble sort en el peor caso (**$O(n^2)$**), Insertion Sort fue más rápido. Esto muestra cómo no solo importa la complejidad teórica, sino también el comportamiento práctico de los algoritmos.
- Utilizar **perf_counter()** y calcular el promedio de varias ejecuciones permitió obtener mediciones más estables y menos afectadas por el sistema operativo u otros procesos.
- No existe un único algoritmo “mejor”. Si se sabe que la lista suele estar casi ordenada, Insertion Sort puede ser una buena opción. Si se trabaja con grandes volúmenes de datos, algoritmos más complejos como Merge Sort son preferibles.

El análisis de algoritmos es una herramienta fundamental. A través de este trabajo práctico aprendimos que dos soluciones a un mismo problema pueden tener diferencias enormes en rendimiento, **especialmente cuando se escalan a grandes volúmenes de datos.**
Entender cómo y por qué un algoritmo es más eficiente que otro nos permite tomar decisiones más inteligentes.

Bibliografía

DataCamp. (s.f.). *¿Qué es un algoritmo?* DataCamp. https://www.datacamp.com/es/blog/what-is-an-algorithm

Díaz, N. E. (s.f.). *Estructuras de Datos Dinámicas I - Capítulo 1*. Universidad del Cauca. http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm

freeCodeCamp. (2022, septiembre 27). *Hoja de trucos de la notación Big O*. https://www.freecodecamp.org/espanol/news/hoja-de-trucos-big-o/

Python Software Foundation. (s.f.). *Recetario de ordenamientos*. En *Documentación de Python 3.13*. https://docs.python.org/es/3.13/howto/sorting.html

Runestone Academy. (s.f.). *El ordenamiento burbuja*. En *Programación en Python*. https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoBurbuja.html

Juncotic. (2021, junio 1). *Ordenamiento por inserción - Algoritmos de ordenamiento*. https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento/

W3Schools. (s.f.). *Python Bubble Sort*. https://www.w3schools.com/python/python_dsa_bubblesort.asp

W3Schools. (s.f.). *Python Insertion Sort*. https://www.w3schools.com/python/python_dsa_insertionsort.asp

W3Schools. (s.f.). *Python Merge Sort*. https://www.w3schools.com/python/python_dsa_mergesort.asp

Fagonzalezo. (2016, noviembre 10). *Algoritmos de ordenamiento*. GitHub Gist. https://gist.github.com/fagonzalezo/460b7311f84a94750414

Python Software Foundation. (s.f.). *time — Funciones relacionadas con el tiempo*. En *Documentación de Python 3.11*. https://docs.python.org/es/3.11/library/time.html

Angulo, P. (s.f.). *Algoritmos de ordenación*. Universidad Autónoma de Madrid. https://verso.mat.uam.es/~pablo.angulo/doc/laboratorio/b2s2.html