UNIVERSIDAD AMERICANA

FACULTAD DE INGENIERÍA Y ARQUITECTURA

Metodología de la Investigación

Análisis de eficiencia de métodos de ordenamiento

Estudiante:

ISAAC MISAEL MORA CARCAMO

Denis Gabriel Molina Arauz

Yader Vilcent Garcia Bravo

Geordany Efrain Valdez Caceres

Manuel Alejandro Jiron Martinez

Docente:

Jose Duran Garcia

Fecha:

Miércoles 18 de Junio 2025

Análisis de Eficiencia de Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son fundamentales en la informática, ya que permiten organizar datos de manera que puedan ser procesados o buscados de forma más eficiente. Existen diversos algoritmos de ordenamiento, cada uno con distintas características y niveles de eficiencia según el tipo de datos, su tamaño y la cantidad de elementos desordenados. El análisis de eficiencia de estos algoritmos se basa en su complejidad temporal y espacial, las cuales determinan el rendimiento del algoritmo en diferentes escenarios.

Metodos de ordenamiento

Bubble Sort (Ordenamiento de burbuja)

Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Repite este proceso muchas veces hasta que todos estén ordenados, como si las “burbujas” más grandes fueran subiendo al final de la lista.

Ejemplo:

Lista original: [5, 3, 8, 1]

- Paso 1: [3, 5, 1, 8]
- Paso 2: [3, 1, 5, 8]
- Paso 3: [1, 3, 5, 8]

Mejor caso: $O(n)$ (si ya está ordenado)

Peor caso: $O(n^2)$

Ejercicio propuesto:

Tienes una lista de calificaciones de estudiantes en una prueba (valores del 0 al 100). Tu tarea es ordenar esta lista de menor a mayor utilizando el algoritmo de Bubble Sort, implementado manualmente (sin usar sort() ni sorted() de Python).

Insertion Sort (Ordenamiento por inserción)

Toma los elementos uno por uno y los inserta en la posición correcta dentro de una sublista que ya está ordenada. Es como ordenar cartas en tu mano.

Ejemplo:

Lista original: `[4, 2, 7, 1]`

- Empezamos con `4` (ordenado)
- Insertamos **`2`**: **`[2, 4]`**
- Insertamos **`7`**: **`[2, 4, 7]`**
- Insertamos **`1`**: **`[1, 2, 4, 7]`**

`Mejor caso: O(n)`

`Peor caso: O(n²)`

Selection Sort (Ordenamiento por selección)

Busca el menor elemento de la lista y lo intercambia con el primero. Luego busca el segundo menor y lo pone en la segunda posición, y así sucesivamente.

Ejemplo:

Lista original: [7, 3, 1, 4]

- Encuentra el mínimo (1), lo pone al inicio: [1, 3, 7, 4]
- Encuentra el siguiente mínimo (3), ya está: [1, 3, 7, 4]
- Luego 4 y 7 se intercambian: [1, 3, 4, 7]

Siempre: O(n²)

Shell Sort (Ordenamiento por Shell)

Es una mejora del Insertion Sort. La idea principal es comparar elementos que están separados por cierta distancia o “gap” y luego reducir ese gap progresivamente hasta llegar a 1, donde ya se aplica un ordenamiento final tipo inserción.

Este algoritmo fue propuesto por Donald Shell en 1959.

ejemplo:

Lista: `[10, 3, 7, 4, 8, 2]`
Paso 1: gap = 3

- Compara y ordena: `[10, 3, 7, 4, 8, 2]` → se ordenan pares como (10, 4), (3, 8), (7, 2)

Paso 2: gap = 1

- Ahora se aplica una especie de Insertion Sort
- Resultado final: `[2, 3, 4, 7, 8, 10]`

`Peor caso: O(n²)`

`Mejor casó (con mejores secuencias): O(n log² n) o incluso mejor`

Quicksort (Ordenamiento rápido)

También usa dividir y conquistar. Escoge un pivote, separa la lista en dos partes (menores y mayores que el pivote), y repite el proceso recursivamente.

Ejemplo:

Lista original: [6, 3, 8, 5], pivote = 6

- Menores: [3, 5]
- Mayores: [8]
- Aplica recursión en ambas partes

Resultado: [3, 5, 6, 8]

Mejor caso: $O(n \log n)$
Peor caso: $O(n^2)$ (si elige mal el pivote)

Análisis de eficiencia

Análisis teórico: Estudiar la notación Big-O, sin ejecutarlo.

Análisis empírico: Probar el algoritmo con datos reales o simulados y medir el tiempo.

Análisis asintótico: Observar cómo crece el tiempo/memoria con el tamaño de entrada (n).

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso	Complejidad Espacial	Estable	In-place
Bubble Sort	**O(n)**	**O(n²)**	**O(n²)**	**O(1)**	**✅ Sí**	**✅ Sí**
Insertion Sort	**O(n)**	**O(n²)**	**O(n²)**	**O(1)**	**✅ Sí**	**✅ Sí**
Selection Sort	**O(n²)**	**O(n²)**	**O(n²)**	**O(1)**	**❌ No**	**✅ Sí**
Quick Sort	**O(n log n)**	**O(n log n)**	**O(n²)**	**O(log n)**	**❌ No**	**✅ Sí**
Shell Sort	**O(n log n)***	**~O(n log² n)***	**O(n²)**	**O(1)**	**❌ No**	**✅ Sí**

Ejercicio propuesto

Una tienda en línea necesita mostrar sus productos ordenados por precio, desde el más bajo hasta el más alto. Tiene listas de productos de diferentes tamaños (pequeñas, medianas y grandes), que pueden estar:

- Totalmente desordenadas
- Parcialmente ordenadas
- Ya ordenadas (por ejemplo, cuando alguien vuelve a ver la misma categoría)

El equipo de programación quiere saber qué algoritmo de ordenamiento es más eficiente para aplicar en su sistema según cada tipo de lista.