Trabajo Práctico Integrador

Algoritmos de Ordenamiento

Programación I

Ayrton Caldo – ayrton06caldo@gmail.com

Agustín Lago – lagoagustindev@gmail.com

Profesor/a: Ariel Enferrel & Cinthia Rigoni

Fecha de entrega: 09/06/2025

1. Introducción

El siguiente trabajo tiene como objetivo el estudio de los ***algoritmos de ordenamiento***, fundamentales en el desarrollo de software y en el procesamiento eficiente de datos. A través de una investigación teoría y una implementación en ***Python***, se buscará analizar el comportamiento y desempeño de ***3 algoritmos de ordenamiento*** diferentes frente al procesamiento de ***2 conjuntos***.

2. Marco Teórico

¿Qué es un Algoritmo?

Un ***algoritmo*** es un conjunto de instrucciones o reglas definidas, ordenadas y finitas, diseñadas para la resolución de un problema o realización de una tarea. Cada paso de un ***algoritmo*** debe ser preciso, ya que debe ser interpretado y ejecutado de manera consistente.

Dentro del desarrollo de software, la utilización de algoritmos está presente en todas las etapas de procesamiento de datos. Un ejemplo de ellos son los ***algoritmos de ordenamiento.***

¿Qué es un algoritmo de ordenamiento?

Un ***algoritmo de ordenamiento*** es un conjunto de instrucciones que organiza elementos de un conjunto, siguiendo un criterio específico de orden. Estos ***algoritmos*** facilitan otras tareas como la búsqueda, visualización de patrones y optimización de operaciones.

Existen múltiples ***algoritmos de ordenamiento***, cada uno con sus características de ***eficiencia.*** Entre ellos podemos encontrar:

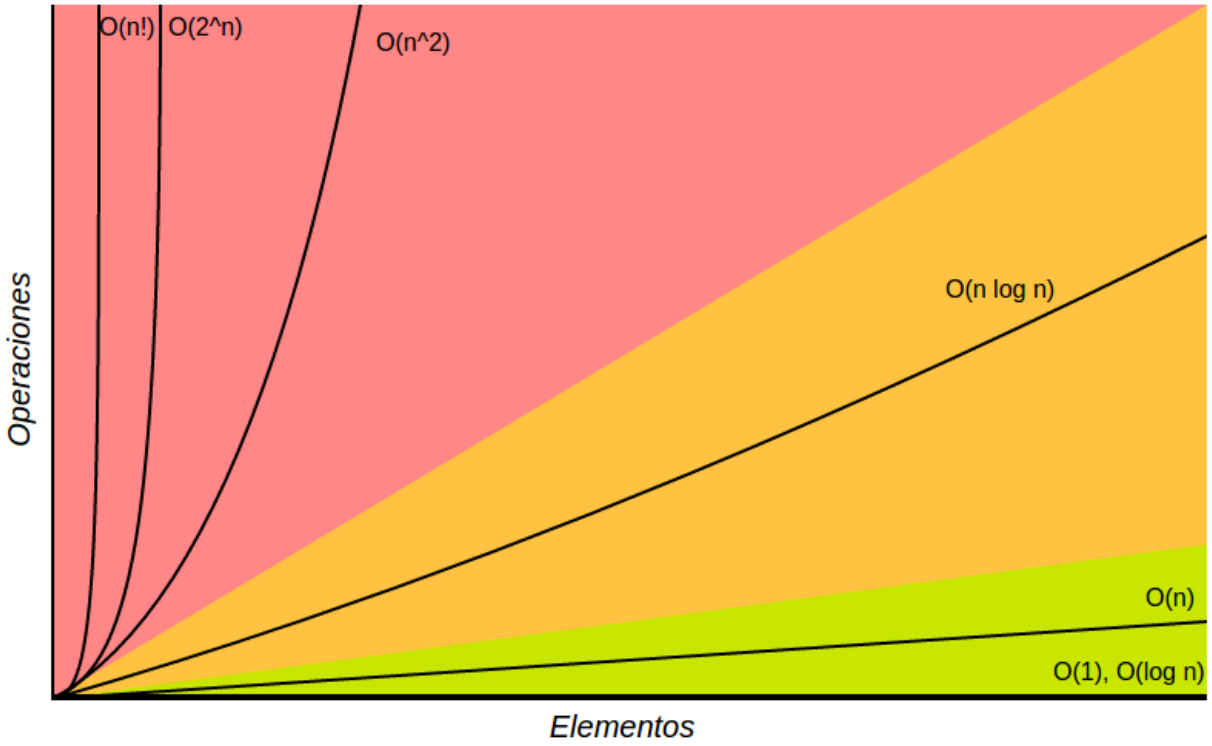
- ***Quick sort:*** Este ***algoritmo*** consiste en seleccionar un elemento pivote y particionar el conjunto original en dos subconjuntos: uno con elementos menores o iguales al elemento seleccionado y otro con elementos mayores. Se aplica ***recursivamente*** el mismo proceso sobre cada subconjunto.
- ***Bubble sort:*** Este ***algoritmo*** compara pares de elementos adyacentes y los intercambia si su orden no es correcto. Este proceso es iterativo hasta que el conjunto quede ordenado.
- ***Random sort o Bogosort:*** Este ***algoritmo*** está basado en la generación aleatoria de cambios en el conjunto de datos hasta obtener una lista ordenada. Se utiliza principalmente con fines demostrativos, ya que, aunque se trate de un ***algoritmo*** correcto, su alta ineficiencia resalta en el análisis de complejidad mediante ***Análisis Asintótico***.

¿Cómo medimos la eficiencia de un algoritmo?

La ***eficiencia*** de un ***algoritmo*** se mide en función del tiempo que tarda en ejecutarse y del uso de memoria. Para esto se realiza un ***Análisis Asintótico***, que permite describir el comportamiento del ***algoritmo*** al crecer la cantidad de datos en la entrada.

El análisis se expresa mediante la notación ***Big-O,*** representando el peor caso de tiempo de ejecución. Entre algunos resultados podemos observar:

- ***O(1)*****:** tiempo constante.
- **O(log n)**: tiempo logarítmico
- **O(n)**: tiempo lineal
- **O(n log n)**: tiempo log-lineal
- **O(n²)**: tiempo cuadrático



3. Caso Práctico

En esta sección desarrollaremos los distintos ***algoritmos de ordenamiento*** planteados anteriormente con el objetivo de analizar y comparar su comportamiento. Mostraremos tanto el código correspondiente a cada ***algoritmo*** como su ***análisis asintótico.*** A su vez se realizaron cálculos de tiempos de ejecución para una misma lista.

1. ***Quick sort:***

Código fuente:

```
def quicksort(array):
    # Condición de corte de recursividad para evitar que se ejecute infinitamente
    if len(array) <= 1:
        return array

    # Tomamos el primer elemento de la lista como elemento de referencia
    target = array[0]

    # Definimos las listas vacías que contendrán los elementos menores y mayores al
elemento de referencia
    lessThanTarget = []
    greaterThanTarget = []

    # Recorremos la lista y comparamos cada elemento con el elemento de referencia
    for i in array[1:]:
        if i <= target:
            lessThanTarget.append(i)
        else:
            greaterThanTarget.append(i)

    # Invocamos la función recursivamente para ordenar el resto de elementos
comprendidos en las listas de menores y mayores
    return quicksort(lessThanTarget) + [target] + quicksort(greaterThanTarget)
```

Eficiencia O(n log n)

En su análisis asintótico se detectan particiones razonablemente balanceadas para la mayoría de los casos prácticos, lo que lleva a un número de niveles logarítmico en el árbol de recursión y un costo lineal en cada nivel.

2. ***Bubble sort:***

Código fuente:

```
def bubblesort(array):
    # Definimos número de elementos de la lista
    lenght = len(array)

    # Recorremos la lista
    for i in range(lenght):
        # En cada iteración los elementos mayores que el elemento actual se mueve
hacia el final de la lista
        for j in range(0, lenght - i - 1):
# Se compara el elemento actual con el siguiente, intercambiandolo si es necesario
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]

    return array
```

Eficiencia O(n²)

En su análisis asintótico se observa que la mayoría de los elementos requieren múltiples comparaciones e intercambios, lo que implica un número cuadrático de operaciones en función del tamaño de la lista.

3. ***Bogosort:***

Código fuente:

```
def bogosort(array):
# Mientras que la lista no se encuentre ordenada, se vuelve a ordenar
aleatoriamente
    while not isSorted(array):
        random.shuffle(array)

    # Retornamos la lista ordenada
    return array

# Definimos una función verificadora de orden
def isSorted(array):
    for i in range(len(array) - 1):
        if array[i] > array[i + 1]:
            return False
    return True
```

Eficiencia O(n!)

En su análisis asintótico se observa que la generación aleatoria de cambios implica un número de intentos proporcional a n!, debido al crecimiento exponencial del espacio de búsqueda de casos posibles.

4. Metodología Utilizada

1. Se investigaron diferentes algoritmos de ordenamiento y su funcionamiento.
2. Se recopilaron datos sobre el análisis asintótico de los algoritmos seleccionados.
3. Se implementaron los algoritmos en lenguaje Python.
4. Se midió el tiempo de ejecución de cada algoritmo utilizando la función `perf_counter()` para dos casos diferentes, una lista de 10 elementos y una de 100.

```
# Iniciamos el contador de tiempo
startTime = time.perf_counter()
# Invocación de algoritmo de ordenamiento
numbers = [i for i in range(10)] || [i for i in range(100)]
sortedNumbers = algoritmo(numbers)
# Calculamos y mostramos el tiempo de ejecución
```

```
endTime = time.perf_counter()
print("Execution time: ", endTime - startTime)
```

5. El experimento se repitió tres veces para obtener resultados consistentes.
6. Se analizaron los resultados obtenidos para extraer conclusiones sobre el desempeño de cada algoritmo.

5. Resultados Obtenidos

Input	`[i for i in range(10)]`		
Algoritmo de ordenamiento	***Tiempo de ejecución registrado (segundos)***		
	Iteración 1	***Iteración 2***	***Iteración 3***
Quick sort:	6.8200e-5	5.0299e-5	5.3299e-5
Bubble sort:	6.3700e-5	4.4800e-5	4.9799e-5
Bogo sort:	0.0218	0.0009	0.0030

Input	`[i for i in range(100)]`		
Algoritmo de ordenamiento	***Tiempo de ejecución registrado (segundos)***		
	Iteración 1	***Iteración 2***	***Iteración 3***
Quick sort:	0.000164	0.000186	0.000144
Bubble sort:	0.000318	0.000283	0.000304
Bogo sort:	***~~~***	***~~~***	***~~~***

	Promedio (segundos)	
Algoritmo de ordenamiento	***Lista pequeña***	***Lista de 100 elementos***
Quick sort:	~0.0000573	~0.0001647
Bubble sort:	~0.0000528	~0.0003017
Bogo sort:	~0.00857	No ejecutado (impracticable)

6. Conclusiones

Los resultados analizados reflejaron las diferencias esperadas entre los ***3 algoritmos*** analizados. En lista pequeñas, ***Bubblesort*** otorgó tiempos de ejecución inferiores a ***Quicksort***. Sin embargo, al aumentar el tamaño de la lista a 100 elementos, ***Quicksort*** presentó una clara superioridad. Por su parte, ***Bogosort*** resulta impráctico para listas de pequeño tamaño e inaplicable a listas de mayor tamaño.

Este experimento confirma los comportamientos planteados en el ***análisis asintótico*** para cada algoritmo y evidencia la importancia de seleccionar el ***algoritmo de ordenamiento*** correcto en función del tamaño y características del conjunto de datos a analizar, priorizando ***algoritmos*** eficientes como el ***Quicksort*** en aplicaciones reales.