Trabajo Integrador Programación 1.

Alvarez Andres
Alvarez Emiliano
Comisión 24
Mayo 2025

Índice

Introducción.

Imagina que sos el coordinador académico de una escuela secundaria que tiene 10 cursos distintos, y cada uno con 100 estudiantes. Un día, necesitás encontrar rápidamente a “Sofía Martínez”, una alumna de la cual no sabés ni el curso exacto ni mucho menos su posición en la lista.

Podrías empezar curso por curso, nombre por nombre, revisando las listas de asistencia: ese sería el método más simple, pero también el más lento.

Ahora, pensá si en cambio tuvieras todas las listas ordenadas alfabéticamente: podrías buscar mucho más rápido, tal vez usando un método que descarte la mitad de los nombres en cada paso. Y si, además, cada nombre tuviera asignado un código único como un número de legajo que te permite ir directamente a su ubicación exacta, la búsqueda sería casi instantánea.

Esta situación ilustra un problema central en la programación: la búsqueda y el ordenamiento de datos. Estos dos procesos son fundamentales en el diseño de algoritmos, ya que permiten organizar y acceder a la información de forma eficiente. En programación, trabajar con datos desordenados o aplicar algoritmos inadecuados puede representar diferencias significativas en términos de rendimiento, especialmente a medida que el volumen de datos crece.

Por ello, el estudio de los algoritmos de búsqueda y ordenamiento no solo implica aprender cómo funcionan, sino comprender cuándo conviene usar cada uno, cómo se comportan según el tamaño y la estructura de los datos, y cuál es su complejidad computacional.

Estas tareas están presentes en casi todos los programas que manejan información, desde bases de datos escolares hasta redes sociales, juegos o sistemas bancarios.

Elegir el método adecuado puede hacer una diferencia enorme en eficiencia, sobre todo cuando los datos son muchos.

Marco teórico.

1. ¿Qué es la búsqueda?

La búsqueda es el proceso de localizar un elemento específico dentro de una colección de datos. En programación, esta operación es esencial y se encuentra en tareas tan diversas como consultar bases de datos, encontrar archivos en un sistema operativo o localizar rutas en un mapa digital.

Tipos de algoritmos de búsqueda

1. **Búsqueda lineal**
   Recorre los elementos uno por uno hasta encontrar el deseado o llegar al final.
   - Ventajas: Simplicidad, no requiere datos ordenados.
   - Desventajas: Ineficiente en listas grandes.
   - Ejemplo aplicado: buscar manualmente en una lista de 1000 alumnos el que tiene promedio 10.
2. **Búsqueda binaria**
   Requiere que la lista esté ordenada previamente. Divide el conjunto de datos en mitades sucesivas hasta hallar el valor o descartar su existencia.
   - Ventajas: Mucho más rápido que la lineal en listas grandes.
   - Desventajas: No funciona con listas desordenadas.
   - Ejemplo aplicado: buscar por DNI a un alumno en una lista de alumnos ya ordenados por número de documento.
3. **Búsqueda por interpolación**
   Estima la posición del valor deseado considerando su distribución en la lista. Es útil cuando los datos están uniformemente distribuidos.
   - Ejemplo aplicado: buscar el alumno que tiene un promedio de 9.50 en una lista ordenada de promedios.
4. **Búsqueda con hash**
   Utiliza una función de dispersión (hash) para ubicar cada elemento en una tabla hash.

- Ventajas: Acceso extremadamente rápido.
- Desventajas: Requiere estructuras específicas y puede haber colisiones.
- Ejemplo aplicado: acceso directo al legajo de un alumno en una base de datos bien diseñada.

2. ¿Qué es el ordenamiento?

El ordenamiento consiste en organizar los datos de acuerdo a un criterio específico, como de menor a mayor, alfabéticamente o por fecha. Es un paso previo importante para muchas operaciones, incluida la búsqueda eficiente.

Beneficios del ordenamiento:

- Permite aplicar algoritmos más rápidos (como la búsqueda binaria).
- Facilita el análisis de datos (por ejemplo, para detectar máximos o mínimos).
- Mejora la legibilidad y la presentación de la información.

Algoritmos comunes de ordenamiento

1. **Bubble Sort ($O(n^2)$)**
   Compara elementos adyacentes e intercambia si están en orden incorrecto.
   - Muy simple, pero ineficiente para listas grandes.
   - Ejemplo aplicado: ordenar los promedios de un curso de 20 alumnos.
2. **Selection Sort ($O(n^2)$)**
   Encuentra el menor y lo coloca al principio, repitiendo hasta ordenar todo.
   - Fácil de implementar, pero lento en grandes volúmenes.
3. **Insertion Sort ($O(n^2)$, pero $O(n)$ en listas casi ordenadas)**
   Inserta cada nuevo elemento en la posición correcta.
   - Ideal para listas pequeñas o parcialmente ordenadas.
4. **Merge Sort ($O(n \log n)$)**
   Divide la lista en mitades, ordena cada mitad y las fusiona.
   - Muy eficiente y estable, útil para grandes volúmenes de datos.
5. **Quick Sort ($O(n \log n)$ promedio)**
   Selecciona un “pivote”, divide según ese valor y ordena recursivamente.

    - Rápido, aunque puede tener un peor caso de $O(n^2)$ si el pivote no se elige bien.

Elección del algoritmo

- Para listas pequeñas: algoritmos simples como Bubble o Insertion Sort pueden ser adecuados.
- Para listas grandes: Quick Sort o Merge Sort ofrecen mejor rendimiento.
- Si se requiere ordenamiento estable (mantener el orden relativo de elementos iguales), Merge Sort es preferido.

3. Relación entre búsqueda y ordenamiento

El ordenamiento potencia la eficiencia de las búsquedas. En listas ordenadas, se puede aplicar la búsqueda binaria o interpolación, que son mucho más rápidas que la búsqueda lineal. Por ejemplo, si se desea localizar a todos los alumnos con promedios superiores a 9, tener la lista ordenada permite detener la búsqueda tan pronto como se alcance un promedio inferior.

4. Importancia en la programación

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales de la ciencia de la computación:

- **Eficiencia**: Optimizar el acceso a la información.
- **Escalabilidad**: Adaptarse a grandes volúmenes de datos.
- **Versatilidad**: Se utilizan en bases de datos, motores de búsqueda, sistemas operativos, redes, inteligencia artificial, entre otros.
- **Precisión**: Garantizan resultados exactos, minimizando errores en el manejo de información.

Algoritmo

Este proyecto implementa y compara tres algoritmos clásicos de ordenamiento: Bubble Sort, Merge Sort y Quick Sort, utilizando el lenguaje Python.

A través de una interfaz en consola sencilla, el usuario puede generar una lista de números aleatorios y elegir cuál algoritmo utilizar para ordenarla. También se puede comparar el tiempo que tarda cada uno en ordenar la misma lista, lo que permite visualizar las diferencias de eficiencia entre un algoritmo lento (como Bubble Sort) y otros más rápidos y eficientes (como Merge y Quick Sort).

El código está estructurado de forma clara, con funciones independientes para cada algoritmo, una función para medir el tiempo de ejecución y un menú principal que organiza toda la interacción. Esta implementación fue pensada para experimentar con conceptos fundamentales de la programación y el análisis de algoritmos de búsqueda y ordenamiento y su eficiencia en tiempo.

```
import random
import time

#  Algoritmo 1: Bubble Sort
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

#  Algoritmo 2: Merge Sort
def merge_sort(lista):
    if len(lista) > 1:
        medio = len(lista) // 2
        izquierda = lista[:medio]
        derecha = lista[medio:]

        merge_sort(izquierda)
        merge_sort(derecha)

        i = j = k = 0
        while i < len(izquierda) and j < len(derecha):
            if izquierda[i] < derecha[j]:
                lista[k] = izquierda[i]
                i += 1
            else:
                lista[k] = derecha[j]
```

```
                j += 1
            k += 1

        while i < len(izquierda):
            lista[k] = izquierda[i]
            i += 1
            k += 1

        while j < len(derecha):
            lista[k] = derecha[j]
            j += 1
            k += 1

#  Algoritmo 3: Quick Sort
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if x < pivote]
        mayores = [x for x in lista[1:] if x >= pivote]
        return quick_sort(menores) + [pivote] + quick_sort(mayores)

#  Función para medir el tiempo que tarda cada algoritmo
def medir_tiempo(algoritmo, lista):
    inicio = time.time()
    algoritmo(lista)
    fin = time.time()
    return fin - inicio

#  Menú principal
def menu():
    while True:
        print("\n--- Comparador de Algoritmos de Ordenamiento ---")
        print("1. Ordenar lista con Bubble Sort")
        print("2. Ordenar lista con Merge Sort")
        print("3. Ordenar lista con Quick Sort")
        print("4. Comparar los tres algoritmos")
        print("5. Salir")
        opcion = input("Elegí una opción (1-5): ")

        if opcion in ["1", "2", "3", "4"]:
            try:
```

```
                tamaño = int(input("¿Cuántos números aleatorios querés
generar? "))
                lista_original = [random.randint(1, 10000) for _ in
range(tamaño)]
            except:
                print("Ingresaste un valor no válido.")
                continue

        if opcion == "1":
            lista = lista_original.copy()
            t = medir_tiempo(bubble_sort, lista)
            print("Bubble Sort completado.")
            print(f"Tiempo: {t:.6f} segundos.")

        elif opcion == "2":
            lista = lista_original.copy()
            t = medir_tiempo(merge_sort, lista)
            print("Merge Sort completado.")
            print(f"Tiempo: {t:.6f} segundos.")

        elif opcion == "3":
            lista = lista_original.copy()
            t = medir_tiempo(quick_sort, lista)
            print("Quick Sort completado.")
            print(f"Tiempo: {t:.6f} segundos.")

        elif opcion == "4":
            for alg in [bubble_sort, merge_sort, quick_sort]:
                lista = lista_original.copy()
                t = medir_tiempo(alg, lista)
                print(f"{alg.__name__} → {t:.6f} segundos.")

        elif opcion == "5":
            print("¡Gracias por usar el comparador!")
            break

        else:
            print("Opción no válida. Por favor, elegí entre 1 y 5.")

#  Ejecutar el menú
menu()
```

Metodología Empleada

Para el desarrollo de este trabajo práctico se plantearon tres etapas bien definidas:

1. Parte Teórica: Analizando información tanto de los materiales y videos ofrecidos en la unidad de la cursada, especialmente sobre el tema Búsqueda y Ordenamiento asi como tambien parte de analisis algoritmos, que nos serian útiles tanto en la fundamentación del marco teórico como del diseño y análisis posterior del caso práctico.

2. Parte Práctica: Habiendo planteado el objetivo del algoritmo propuesto nos enfocamos en traer un ejemplo que cumpliera con dicho objetivo de forma ordenada y simple que pudiéramos lograr. Incluimos en este documento también las pruebas finales que nos permitieron realizar el análisis comparativo deseado. Proveer una interfaz acorde a la prueba que queríamos realizar.

3. Análisis Final: Evaluamos el nivel de concreción de los objetivos planteados y consignas que se nos solicitaban para este trabajo. Para finalizar acercamos nuestras conclusiones producto del trabajo grupal y las experiencias realizando el mismo.

Resultados obtenidos

- El programa cumplió con realizar correctamente los 3 tipos de ordenamientos solicitados sobre los diferentes tamaños de lista.
- Nos fue posible implementar y consultar la medición del tiempo de ejecución de los diferentes métodos.
- Comparamos los datos obtenidos para comprender en qué situaciones tendremos que elegir cada método de ordenamiento, de acuerdo al tamaño de los datos de entrada.
- La interfaz de usuario funciono de acorde a los esperado.

Conclusión

Sobre la temática: Tal como se planteó en el marco teórico, el ordenamiento de los datos es una de las tantas operaciones importantes para iniciarnos en el manejo de grandes cantidades de datos con la que estaremos trabajando durante nuestra etapa de formación y trabajo como programadores, especialmente teniendo en cuenta lo expuesto acerca de la necesidad de ordenar los datos antes de utilizar algoritmos de búsqueda que necesitan ordenamiento.

Por otro lado, haber podido realizar comparaciones sobre el desempeño de diferentes algoritmos nos hace reflexionar acerca de que no solo nos tenemos que ocupar de encontrar soluciones a los problemas que se nos presenten, sino también que la eficiencia de la estrategia a implementar tiene que ser tenida en cuenta.

Sobre los resultados obtenidos: Los datos obtenidos varían considerablemente teniendo en cuenta la estrategia empleada (ANEXOS CON EJEMPLOS) podemos verificar cuales son

las eficiencias de los métodos en función del tamaño de la lista de datos, tal como se indicaba teóricamente:

DIMENSIÓN DE LA LISTA	ALGORITMO		
	BUBBLE SORT	MERGE SORT	QUICK SORT
Pequeña	MEJOR	MEDIA	MEDIA
Mediana	MEDIA	BUENA	BUENA
Grande	MALA	MEJOR	MEJOR

Bibliografía:

- **Apuntes de Programación 1 – UTN – 2025**
  Tecnicatura Universitaria en Programación

ANEXOS

LINK VIDEO EXPLICATIVO
https://youtu.be/YzBsgGEvSxc

LINK DEL REPOSITORIO CON EL PROGRAMA
https://github.com/emiready/Programacion1/blob/main/IntegradorProgramacion2.py

PRUEBAS DE VERIFICACIÓN DEL FUNCIONAMIENTO: (se imprimieron listas de tamaño pequeño por comodidad de la visualización, para ello se utilizó un comando print() de forma provisoria a la salida del cálculo de tiempo de cada método)

```
Elegí una opción (1-5): 1
¿Cuántos números aleatorios querés generar? 20
Lista desordenada -> [4406, 6940, 1072, 8671, 2648, 9033, 7232, 8027, 6226, 4663, 6741, 6416, 6671, 5281, 9097, 1236, 1619, 8457, 1622, 8299]
Lista ordenada -> [1072, 1236, 1619, 1622, 2648, 4406, 4663, 5281, 6226, 6416, 6671, 6741, 6940, 7232, 8027, 8299, 8457, 8671, 9033, 9097]
Bubble Sort completado.
Tiempo: 0.000058 segundos.
```

```
Elegí una opción (1-5): 2
¿Cuántos números aleatorios querés generar? 20
Lista desordenada -> [8786, 5008, 262, 74, 6523, 9319, 2215, 7329, 4305, 4645, 5821, 3874, 129, 4214, 9369, 1278, 1031, 3623, 8299, 7462]
Lista ordenada -> [74, 129, 262, 1031, 1278, 2215, 3623, 3874, 4214, 4305, 4645, 5008, 5821, 6523, 7329, 7462, 8299, 8786, 9319, 9369]
Merge Sort completado.
Tiempo: 0.000074 segundos.
```

```
¿Cuántos números aleatorios querés generar? 20
Lista desordenada -> [1666, 372, 7057, 2492, 9512, 2515, 4355, 3992, 8149, 2252, 6759, 8598, 5017, 8156, 8996, 821, 2062, 1573, 7756, 9845]
lista ordenada-> [372, 821, 1573, 1666, 2062, 2252, 2492, 2515, 3992, 4355, 5017, 6759, 7057, 7756, 8149, 8156, 8598, 8996, 9512, 9845]
Quick Sort completado.
Tiempo: 0.000184 segundos.
```

PRUEBAS DE RESULTADOS COMPARATIVOS

```
¿Cuántos números aleatorios querés generar? 10
bubble_sort -> 0.000031 segundos.
merge_sort -> 0.000044 segundos.
quick_sort -> 0.000043 segundos.
```

```
Elegí una opción (1-5): 4
¿Cuántos números aleatorios querés generar? 1000
bubble_sort -> 0.071187 segundos.
merge_sort -> 0.002361 segundos.
quick_sort -> 0.001640 segundos.
```

```
Elegí una opción (1-5): 4
¿Cuántos números aleatorios querés generar? 10000
bubble_sort -> 6.176161 segundos.
merge_sort -> 0.038090 segundos.
quick_sort -> 0.027780 segundos.
```