Documentación del Programa de Comparación y Análisis de Algoritmos de Búsqueda y Ordenamiento

---

Objetivo General

Este programa permite comparar el rendimiento de diferentes **algoritmos de ordenamiento** y **algoritmos de búsqueda** aplicados sobre una lista de canciones generadas aleatoriamente. Evalúa su eficiencia a través de tiempos de ejecución y muestra los resultados por consola.

Estructura del Código

---

1. Importación de módulos

```
import time
import random
import sys
```

- `time`: Para medir tiempos de ejecución.
- `random`: Para generar datos aleatorios.
- `sys`: Para aumentar el límite de recursión (necesario para listas grandes con Quick Sort).

---

2. Configuración inicial

```
sys.setrecursionlimit(5000)
```

Se aumenta el límite de recursión para evitar errores de stack overflow con `quick_sort` en listas grandes.

---

3. Funciones de ordenamiento

`bubble_sort(lista, clave)`

- Algoritmo de burbujeo, comparando elementos adyacentes.
- Ineficiente en listas grandes (`O(n²)`).

`insertion_sort(lista, clave)`

- Inserta cada elemento en su posición correcta en la parte ordenada.
- Mejor rendimiento que Bubble Sort, pero también `O(n²)`.

`selection_sort(lista, clave)`

- Selecciona el mínimo de la lista restante y lo coloca en orden.
- También de orden cuadrático.

`quick_sort(lista, clave)`

- Divide y conquista: elige un punto central o pivote y separa elementos menores y mayores.
- Tiene una complejidad promedio de `O(n log n)`.

*Todas las funciones devuelven una copia ordenada de la lista sin modificar la original.

---

4. Funciones de búsqueda

`busqueda_lineal(lista, clave, valor)`

- Recorre la lista secuencialmente buscando coincidencias exactas (`==`).
- Funciona en cualquier lista (ordenada o no).
- Retorna una lista de coincidencias.

`busqueda_binaria(lista, clave, valor)`

- Requiere una lista ordenada por la clave.

- Busca el valor utilizando el algoritmo binario (`O(log n)`).
- Expande desde el elemento encontrado hacia ambos lados para capturar múltiples coincidencias.

---

5. Generación de lista de canciones

`def generar_lista_canciones(n):`

- Crea una lista de `n` diccionarios con claves: `nombre`, `artista`, `genero`.
- Los valores son aleatorios:
    - Nombre: `"CanciónX"`
    - Artista: `"ArtistaX"` (repetidos cada 10 canciones)
    - Géneros aleatorios entre Pop, Rock, Jazz, Rap, Clásica, Indie.

---

6. Mostrar resultados

`def mostrar_resultados(titulo, resultados):`

- Muestra hasta 10 canciones coincidentes.
- Imprime título y número de resultados encontrados.
- Si hay más de 10 resultados, imprime `...` como indicación.

---

7. Ejecución de búsquedas y ordenamientos

`def ejecutar_busquedas(valor_busqueda, lista):`

- Ejecuta **búsqueda lineal** en los campos `nombre`, `artista` y `genero`.
- Mide tiempo y muestra resultados para cada campo.
- Para **búsqueda binaria**, primero ordena la lista con cada uno de los 4 algoritmos, luego realiza la búsqueda binaria:
    - Se mide el tiempo de ordenamiento.

    - Se mide el tiempo de búsqueda binaria.
    - Se muestran ambos resultados.

---

8. Menú de interacción con el usuario

`def menu():`

- Menú interactivo por consola con dos opciones:
    - `1. Buscar`: solicita el tamaño de la lista (10, 100, 1000, 10000) y un valor a buscar.
    - `2. Salir`: finaliza el programa.
- Valida entradas del usuario.
- Llama a `ejecutar_busquedas` en la opción 1.

---

9. Punto de entrada del programa

```
if __name__ == "__main__":
    menu()
```

- Inicia el programa ejecutando el menú principal si el archivo es ejecutado directamente.

Propósito de análisis

Este programa está diseñado para:

- **Comparar la eficiencia** de algoritmos de ordenamiento clásicos.
- **Evaluar el impacto** del ordenamiento sobre la búsqueda binaria.
- **Visualizar diferencias** entre búsqueda lineal y binaria.
- **Medir tiempos reales** de ejecución en listas de distintos tamaños.
- **Reflexionar sobre la elección adecuada** de algoritmos según el contexto y la cantidad de datos.