

Trabajo Práctico Integrador

Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos

Zampar Facundo, correo: facuzamparutn@gmail.com

Zampieri Pamela, correo: pamezampieri@gmail.com

**Tecnicatura Universitaria en Programación - Universidad
Tecnológica Nacional.**

Programación I

Docente Titular

Nicolás Quirós

Docente Tutor

Flor Camila Gubiotti

9 de Junio de 2025

Índice

INTRODUCCIÓN.....	3
MARCO TEÓRICO.....	4
Algoritmos de búsqueda.....	4
Búsqueda lineal (secuencial).....	4
Búsqueda binaria.....	4
Algoritmos de ordenamiento.....	5
Bubble Sort (ordenamiento burbuja).....	5
Selection Sort (ordenamiento por selección).....	6
Insertion Sort.....	6
Merge Sort.....	7
Quick Sort.....	7
Algoritmos en python.....	8
Aplicación en el Trabajo Práctico.....	8
CASO PRÁCTICO.....	9
METODOLOGÍA UTILIZADA.....	13
RESULTADOS OBTENIDOS.....	15
Pruebas con lista pequeña (10 cursos).....	15
Pruebas con lista grande (1000 cursos).....	19
Pruebas con gran volumen de datos (10000 cursos).....	21
CONCLUSIONES.....	23
BIBLIOGRAFÍA.....	24
ANEXOS.....	25
Código.....	25
Link del video demostrativo del entorno y ejecución del código.....	31
Link del repositorio remoto en GitHub.....	31

Introducción

En el desarrollo de software y la resolución de problemas computacionales, los algoritmos de búsqueda y ordenamiento juegan un papel fundamental. Estas técnicas permiten organizar, manipular y recuperar datos de manera eficiente, lo cual es esencial en diversas aplicaciones, desde bases de datos hasta motores de búsqueda, sistemas de recomendación o inteligencia artificial.

Aunque muchos lenguajes modernos como Python ofrecen funciones integradas para ordenar y buscar elementos en listas, comprender el funcionamiento interno de estos algoritmos brinda a los desarrolladores las herramientas necesarias para optimizar procesos y enfrentar retos de programación con mayor fundamento.

En este trabajo práctico, se abordó un caso de estudio aplicado a la gestión de cursos educativos. A través de una interfaz de consola, se desarrolló un programa que:

- ❖ Genera una lista de cursos con atributos como nombre, profesor, tema, duración y calificación. Para dar realismo, los nombres de los profesores se generaron utilizando la librería Faker.
- ❖ Ordena los cursos según diferentes criterios (nombre, profesor, tema, duración o calificación) mediante los algoritmos Bubble Sort y Quick Sort.
- ❖ Realiza búsquedas por distintos campos (nombre, profesor o tema), implementando tanto búsqueda lineal como búsqueda binaria.
- ❖ Mide y compara los tiempos de ejecución de cada algoritmo, permitiendo evaluar su eficiencia práctica en listas de distintos tamaños (10, 1,000 y 10,000 elementos).

Este enfoque práctico no solo consolidó el conocimiento teórico sobre algoritmos de búsqueda y ordenamiento, sino que también permitió visualizar su impacto en un escenario real, evaluando sus ventajas, limitaciones y criterios para seleccionar la técnica adecuada según el contexto.

Marco Teórico

Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten localizar un elemento específico dentro de una estructura de datos, como una lista o conjunto. Son fundamentales en la informática, ya que muchas tareas requieren encontrar información de forma eficiente.

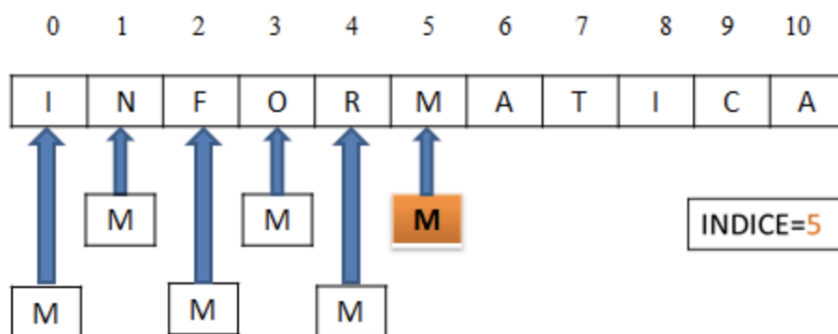
Búsqueda Lineal (o Secuencial)

Consiste en recorrer la lista elemento por elemento hasta encontrar el valor buscado o llegar al final.

- ❖ **Ventajas:** Funciona sobre listas desordenadas.
- ❖ **Desventajas:** Es poco eficiente en listas grandes.
- ❖ **Complejidad temporal:** $O(n)$ en el peor de los casos.

Ejemplo:

Si se busca "M" en la siguiente palabra:



Fuente: <https://jorgecontrerasp.wordpress.com/unidad-i/algoritmos-de-busqueda-y-ordenamiento/>

Búsqueda Binaria

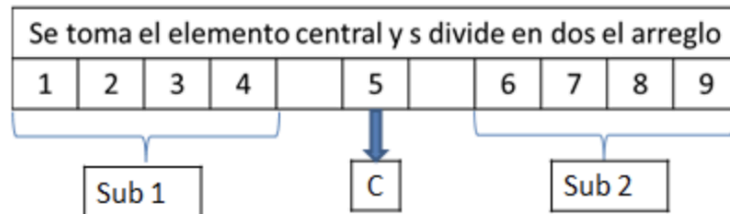
Requiere que la lista esté previamente ordenada. En cada paso, compara el elemento buscado con el del medio y descarta la mitad que no lo contiene.

- ❖ **Ventajas:** Muy eficiente para listas grandes.
- ❖ **Desventajas:** No funciona en listas desordenadas.
- ❖ **Complejidad temporal:** $O(\log n)$.

TRABAJO PRÁCTICO INTEGRADOR

Ejemplo:

Se intenta buscar el elemento 3 en el arreglo {1,2,3,4,5,6,7,8,9}, se realizarán los siguientes pasos.



Como el elemento buscado (3) es menor que el central (5), debe estar en el sub 1 {1,2,3,4}

Se vuelve a dividir el sub 1 en dos {1} - 2 - {3,4}, entonces como el elemento buscado es mayor que el central, debe estar en segundo subarreglo {3,4}, y se vuelve a dividir en dos { } - 3 - {4}, como el elemento buscado coincide con el central, lo hemos encontrado.

Fuente: <https://jorgecontrerasp.wordpress.com/unidad-i/algoritmos-de-busqueda-y-ordenamiento/>

Algoritmos de Ordenamiento

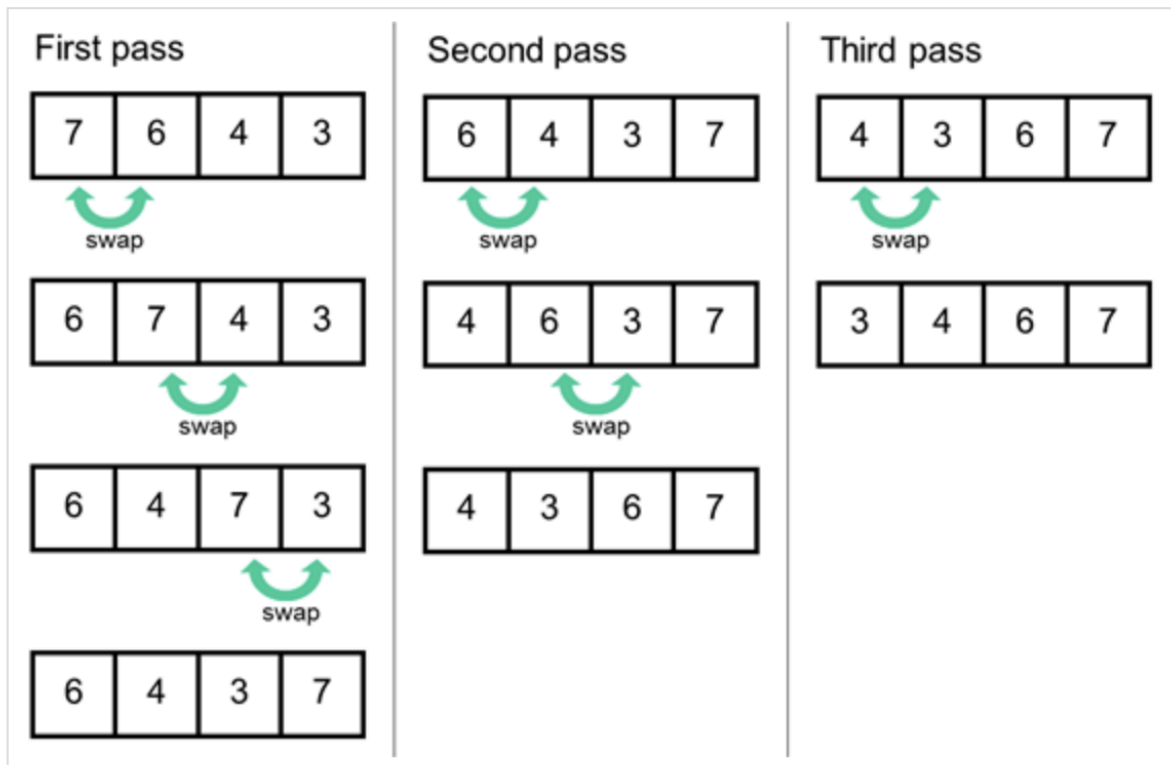
El ordenamiento es el proceso de reorganizar elementos en una estructura de datos según un criterio (por ejemplo, por nombre, duración o calificación). Existen múltiples algoritmos de ordenamiento, cada uno con ventajas y limitaciones:

Bubble Sort (Ordenamiento Burbuja)

Compara pares de elementos adyacentes e intercambia aquellos que están en el orden incorrecto. El proceso se repite hasta que toda la lista está ordenada.

- ❖ **Ventajas:** Fácil de implementar.
- ❖ **Desventajas:** Muy ineficiente para listas grandes.
- ❖ **Complejidad:** $O(n^2)$.

Ejemplo:



Fuente: <https://www.computersciencebytes.com/sorting-algorithms/bubble-sort/>

El ejemplo anterior ordena 4 números en orden numérico ascendente. Como se puede ver, se requieren 3 ($n-1$) pasadas, ya que hay 4 datos. Se puede observar que los números más grandes se expanden hacia la parte superior.

Selection Sort (Ordenamiento por Selección)

Selecciona el mínimo (o máximo) de la lista en cada iteración y lo coloca en su posición definitiva.

- ❖ **Ventajas:** Simple de entender.
- ❖ **Desventajas:** Ineficiente en grandes volúmenes de datos.
- ❖ **Complejidad:** $O(n^2)$.

Insertion Sort (Ordenamiento por Inserción)

Inserta cada elemento en su posición correcta dentro de una sublista ya ordenada.

- ❖ **Ventajas:** Bueno para listas pequeñas o casi ordenadas.
- ❖ **Desventajas:** No escala bien para listas grandes.

- ❖ **Complejidad:** $O(n^2)$.

Merge Sort

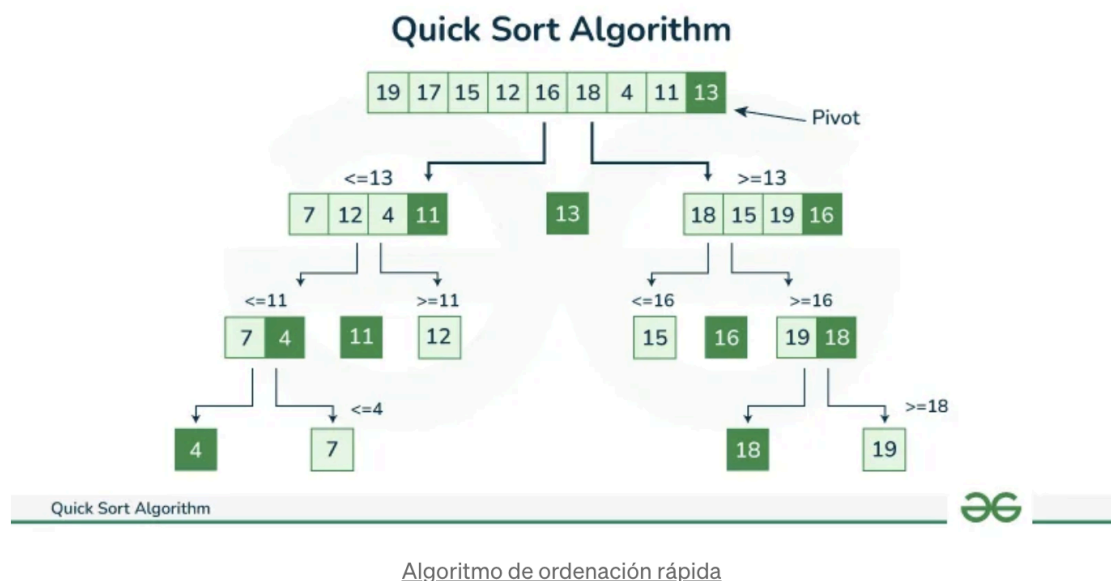
Divide la lista en mitades, las ordena recursivamente y luego las fusiona. Es un algoritmo muy eficiente y estable.

- ❖ **Ventajas:** Excelente rendimiento en listas grandes.
- ❖ **Complejidad:** $O(n \log n)$.

Quick Sort

Es un algoritmo de ordenación eficiente y ampliamente utilizado que emplea un enfoque de "divide y vencerás" para organizar los elementos de un array. El algoritmo selecciona un elemento pivote y reorganiza los elementos de forma que todos los valores mayores que el pivote se coloquen en un lado y los menores en el otro. Este proceso se repite hasta que se ordena todo el array. QuickSort es especialmente ventajoso al trabajar con grandes conjuntos de datos gracias a su capacidad de ordenación rápida y eficaz. Aplica recursión para ordenar ambas mitades.

- ❖ **Ventajas:** Muy rápido en la práctica.
- ❖ **Desventajas:** En el peor caso (lista ya ordenada), puede degradarse.
- ❖ **Complejidad:** Promedio $O(n \log n)$, peor caso $O(n^2)$.



Fuente:

<https://medium.com/@el.elhamhashemi/sorting-algorithms-quicksort-explained-b5bb814df35b>

Algoritmos en Python

Python proporciona funciones integradas como `.sorted()` y el método `.sort()`, que utilizan el algoritmo **Timsort**, una combinación eficiente de **Merge Sort** e **Insertion Sort**.

No obstante, implementar manualmente algoritmos como los que se presentan en este trabajo permite:

- ❖ Comprender su lógica interna.
- ❖ Comparar rendimientos en distintos escenarios.
- ❖ Elegir el algoritmo más adecuado según el tipo y tamaño de datos.

Aplicación en el Trabajo Práctico

En este trabajo práctico se implementan **dos algoritmos de búsqueda**:

- ❖ Búsqueda lineal
- ❖ Búsqueda binaria

Y **dos algoritmos de ordenamiento**:

- ❖ Bubble Sort
- ❖ QuickSort

El caso práctico consiste en generar una lista de cursos con atributos como **nombre**, **docente**, **duración** y **calificación**. A partir de estos datos, se aplican los algoritmos para:

- ❖ Buscar cursos según distintos criterios.
- ❖ Ordenarlos por distintos criterios.
- ❖ Comparar tiempos de ejecución y eficiencia entre algoritmos.

Caso Práctico

Tema elegido:

Implementación de algoritmos de búsqueda y ordenamiento sobre una lista de cursos online generados automáticamente.

En este trabajo práctico se desarrolló un sistema interactivo en Python que simula una base de datos de cursos online, con atributos como nombre del curso, nombre del profesor, tema, duración y calificación. El objetivo principal fue aplicar y analizar algoritmos clásicos de búsqueda (lineal y binaria) y ordenamiento (Bubble Sort y Quick Sort), midiendo sus tiempos de ejecución y observando su eficiencia en distintos volúmenes de datos.

Para la generación automática de datos, se utilizó la biblioteca **Faker** para crear nombres realistas de profesores, y la biblioteca **random** para asignar temáticas, niveles de dificultad, duración en horas y calificaciones aleatorias, lo que permitió crear un conjunto amplio y variado de cursos.

Preparación del entorno

Se trabajó sobre un entorno virtual en macOS, asegurando una ejecución aislada y controlada del proyecto:

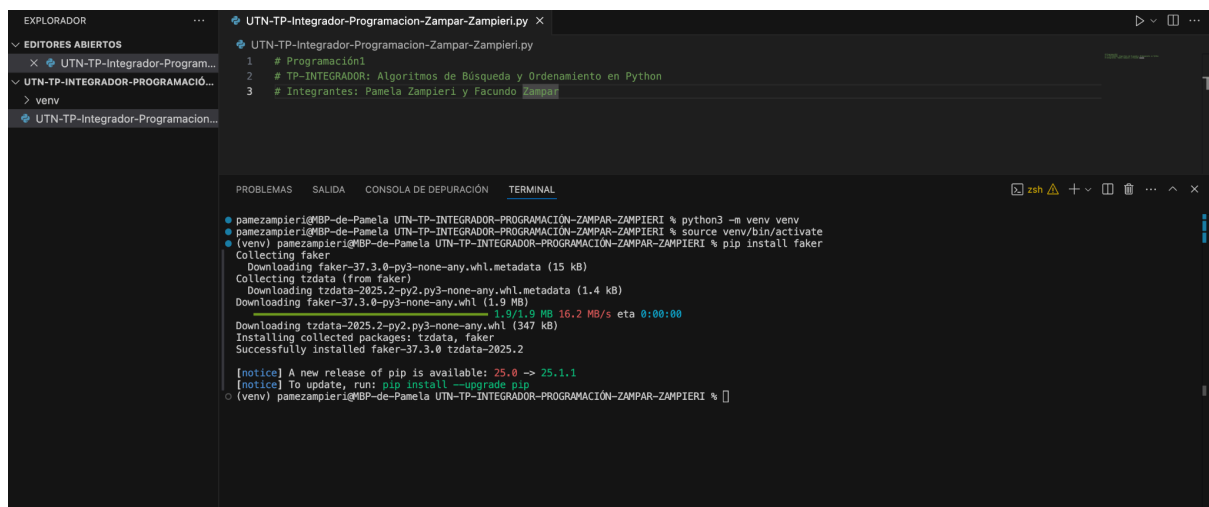
```
python3 -m venv venv
```

```
source venv/bin/activate
```

```
pip install faker
```

Esto garantizó la instalación reproducible de dependencias, facilitando las pruebas.

Instalación de Faker:



```
EXPLORADOR
  UTN-TP-Integrador-Programacion-Zampar-Zampieri.py x
  UTN-TP-Integrador-Programacion-Zampar-Zampieri.py
  1 # Programación1
  2 # TP-INTEGRADOR: Algoritmos de Búsqueda y Ordenamiento en Python
  3 # Integrantes: Pamela Zampieri y Facundo Zampar

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
  pamezampieri@MBP-de-Pamela UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI % python3 -m venv venv
  pamezampieri@MBP-de-Pamela UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI % source venv/bin/activate
  (venv) pamezampieri@MBP-de-Pamela UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI % pip install faker
  Collecting faker
    Downloading faker-37.3.0-py3-none-any.whl.metadata (15 kB)
  Collecting tzdata (from faker)
    Downloading tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)
    Downloading faker-37.3.0-py3-none-any.whl (11.9 MB)
    Downloading tzdata-2025.2-py2.py3-none-any.whl (347 kB)
    Installing collected packages: tzdata, faker
    Successfully installed faker-37.3.0 tzdata-2025.2
  [notice] A new release of pip is available: 25.0 -> 25.1.1
  [notice] To update, run: pip install --upgrade pip
  (venv) pamezampieri@MBP-de-Pamela UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI %
```

Funcionalidades del sistema

El sistema cuenta con un **menú interactivo** que permite al usuario realizar diferentes operaciones:

- ❖ **Generación de cursos aleatorios:**
 - Se generan dinámicamente tantos cursos como indique el usuario, con atributos aleatorios pero controlados.
 - Se visualizan todos los cursos si son pocos, o un resumen con los **5 primeros y 5 últimos** si se superan los 50 elementos.
- ❖ **Ordenamiento de cursos:**
 - Permite ordenar los cursos por cualquier campo (nombre, profesor, tema, duración o calificación).
 - Se aplican y comparan los algoritmos **Bubble Sort** y **Quick Sort**, mostrando los tiempos de ejecución de cada uno sobre la misma lista.
- ❖ **Búsqueda de cursos:**
 - Se puede buscar por **nombre, profesor o tema**.
 - Se ejecutan en paralelo la **búsqueda lineal** (sobre la lista tal cual) y la **búsqueda binaria** (solo si la lista ya fue ordenada por ese campo), comparando tiempos y resultados.

Análisis práctico:

1. Generación de cursos con datos realistas

Se creó una función específica para generar automáticamente una lista de cursos, con atributos como nombre, profesor, tema, duración y calificación.

Para obtener nombres de profesores realistas, se utilizó la biblioteca **Faker**, y se combinaron listas predefinidas con funciones de la biblioteca **random** para los demás campos.

Este enfoque permitió construir una base de datos simulada, estructurada pero aleatoria, adecuada para aplicar técnicas de búsqueda y ordenamiento.

```
# Programación1
# TP-INTEGRADOR: Algoritmos de Búsqueda y Ordenamiento en Python
# Integrantes: Pamela Zampieri y Facundo Zampar
# Implementación de búsqueda y ordenamiento en una lista de cursos online generados automáticamente

from faker import Faker # Importa la librería Faker para generar datos falsos
import random # Importa random para generar números aleatorios
import time # Importa time para medir tiempos de ejecución
import unicodedata # Importa el módulo estándar unicodedata de Python, que permite trabajar con caracteres Unicode

# Crea una instancia de Faker configurada para español de España
fake = Faker('es_ES')
```

```
def generar_cursos(cantidad):
    lista = []
    for _ in range(cantidad):
        tema = random.choice(list(temas_cursos.keys()))
        nombre_base = random.choice(temas_cursos[tema])
        nombre = f"{nombre_base} {random.choice(niveles)}"
        profesor = fake.name()
        duracion = random.randint(5, 50)
        calificacion = round(random.uniform(1.0, 5.0), 1)
        lista.append(Curso(nombre, profesor, tema, duracion, calificacion))
    return lista
```

2. Implementación de algoritmos de ordenamiento

Se desarrollaron dos funciones para ordenar listas de cursos:

- ❖ **Bubble Sort:** útil para comprender la lógica de ordenamiento paso a paso.
- ❖ **Quick Sort:** más eficiente, implementado con recursividad.

Ambas funciones aceptan como criterio de ordenamiento cualquiera de los atributos del curso, gracias al uso de funciones lambda para acceso dinámico a campos.

```
# Métodos de ordenamiento
# Bubble Sort (Ordenamiento burbuja)
def bubble_sort(lista, funcion_clave):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if funcion_clave(lista[j]) > funcion_clave(lista[j + 1]):
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

# QuickSort (Ordenamiento rápido)
def quick_sort(lista, funcion_clave):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if funcion_clave(x) <= funcion_clave(pivote)]
        mayores = [x for x in lista[1:] if funcion_clave(x) > funcion_clave(pivote)]
        return quick_sort(menores, funcion_clave) + [pivote] + quick_sort(mayores, funcion_clave)
```

3. Presentación parcial de listas extensas

Para evitar salidas excesivamente largas en consola, se diseñó una función que **muestra solo los primeros y últimos 5 elementos** de cualquier lista de cursos, facilitando la visualización sin perder contexto.

```
# Si son más de 50 cursos(umbral elegido) se muestra un resumen
def mostrar_resumen_cursos(lista):
    print("... Primeros 5 cursos ...")
    for curso in lista[:5]:
        print(curso)
    print("... Últimos 5 cursos ...")
    for curso in lista[-5:]:
        print(curso)
```

4. Implementación de algoritmos de búsqueda

Se crearon métodos de búsqueda que permiten localizar cursos por distintos criterios:

- ❖ **Búsqueda Lineal:** utilizada sobre listas desordenadas.
- ❖ **Búsqueda Binaria:** aplicada sobre listas ordenadas previamente por el atributo buscado.

Ambos métodos fueron diseñados para aceptar como parámetro el campo a buscar, permitiendo reutilizar el código con diferentes tipos de búsquedas.

```
# Métodos de búsqueda
# Búsqueda binaria (lista previamente ordenada por la función clave)
# Devuelve el índice de una coincidencia (no necesariamente la primera si hay varias)
def busqueda_binaria(items, objetivo, funcion_clave):
    izquierda = 0
    derecha = len(items) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        valor_medio = funcion_clave(items[medio])
        if valor_medio == objetivo:
            return medio # devuelve el índice
        elif valor_medio < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return None # no encontrado

# Búsqueda lineal (lista no ordenada)
# Devuelve el índice de la primera coincidencia encontrada
def busqueda_lineal(items, objetivo, funcion_clave):
    for i in range(len(items)):
        if funcion_clave(items[i]) == objetivo:
            return i
    return None
```

Propósito de esta sección

El objetivo de este análisis fue documentar cómo se construyó la lógica del sistema: desde la generación estructurada de datos hasta la implementación modular y reutilizable de algoritmos clásicos.

Los resultados, pruebas de rendimiento y comparaciones empíricas se presentan en la siguiente sección (*Resultados Obtenidos*), donde se muestran los tiempos de ejecución, eficiencia relativa y comportamiento ante distintos volúmenes de datos.

Metodología Utilizada

Generación de Datos

Para evitar la carga manual de nombres de profesores, se utilizó la biblioteca **Faker**, que permite generar nombres realistas y aleatorios, simulando datos del mundo real.

El resto de los atributos del curso (nombre del curso, tema, nivel de dificultad, duración y calificación) se generaron combinando **listas predefinidas** con valores aleatorios generados mediante la biblioteca **random**. Esto permitió crear cursos diversos, con datos no triviales, ideales para evaluar algoritmos.

Ventajas de esta metodología de generación de datos:

- ❖ **Ahorro de tiempo:** Se evita la carga manual de datos uno por uno.
- ❖ **Datos realistas:** Faker aporta verosimilitud al simular nombres de personas comunes.
- ❖ **Pruebas en escala:** Se pueden generar cientos o miles de cursos fácilmente.
- ❖ **Variedad estructurada:** La combinación con **random** permite simular datos heterogéneos, pero controlados.

Implementación de Algoritmos

Búsqueda

- ❖ **Búsqueda Lineal:**
Recorre todos los elementos uno por uno. Se usa cuando no se garantiza que la lista esté ordenada.
- ❖ **Búsqueda Binaria:**
Requiere que la lista esté previamente ordenada por el campo buscado. Divide el rango de búsqueda a la mitad en cada paso, mejorando significativamente el rendimiento en listas grandes.

TRABAJO PRÁCTICO INTEGRADOR

Ordenamiento

❖ **Bubble Sort:**

Algoritmo sencillo, utilizado con fines pedagógicos. Poco eficiente para listas grandes, ya que tiene complejidad cuadrática.

❖ **Quick Sort:**

Más eficiente en la práctica, con un enfoque de "divide y vencerás" que reduce considerablemente el tiempo de ordenamiento para grandes volúmenes de datos.

Evaluación del Rendimiento

- ❖ Para cada operación de búsqueda u ordenamiento, se mide el **tiempo de ejecución** utilizando `time.time()` antes y después del algoritmo.
- ❖ Se trabaja sobre **copias idénticas de la lista original**, asegurando condiciones equitativas entre algoritmos.
- ❖ Cuando se ordena, la lista principal del sistema se actualiza, permitiendo luego aplicar búsquedas binarias eficientes sobre el mismo campo.

Resultados Obtenidos

Durante la ejecución del programa se realizaron múltiples pruebas de inserción, ordenamiento y búsqueda sobre una lista generada automáticamente de cursos online.

Se utilizaron los algoritmos de **búsqueda lineal y binaria**, y los algoritmos de **ordenamiento Bubble Sort y QuickSort**, registrando el tiempo de ejecución de cada uno mediante la biblioteca `time`.

Pruebas con lista pequeña (10 cursos)

Prueba 1: Búsqueda por nombre en lista desordenada (10 cursos)

Para esta primera prueba se generaron **10 cursos aleatorios**, simulando un conjunto reducido de datos con fines de verificación funcional.

```

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
/Users/pamezampieri/Desktop/UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI/venv/bin/python /Users/pamezampieri/Desktop/UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI/UTN-TP-Integrador-Programación1-Zampar-Zampieri.py
○ venv/pamezampieri@MBP-de-Pamela: UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI % /Users/pamezampieri/Desktop/UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI/venv/bin/python /Users/pamezampieri/Desktop/UTN-TP-INTEGRADOR-PROGRAMACIÓN-ZAMPAR-ZAMPIERI/UTN-TP-Integrador-Programación1-Zampar-Zampieri.py

¡Bienvenido! ¿Cuántos cursos desea generar? 10

--- CURSOS GENERADOS (DESORDENADOS) ---
Estadística Intermedio | Victoriano Rosales Solana | Matemáticas | 34 hs | 4.0
Python Intermedio | Aurelio de Caparrós | Programación | 31 hs | 4.0
Álgebra Avanzado | Vanesa Montoya Mármol | Matemáticas | 38 hs | 1.8
Algoritmos Básico | Iván Vergara Piñol | Programación | 42 hs | 3.3
Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3
Cálculo Intermedio | Heriberto Eustaquio Esparza Ríos | Matemáticas | 13 hs | 1.1
Historia de América Intermedio | Godofredo de Castell | Historia | 31 hs | 2.9
Escultura Avanzado | Jose Ignacio Sotelo Canelas | Arte | 21 hs | 2.0
Diseño Industrial Avanzado | Anacleto Santamaria Río | Diseño | 12 hs | 3.3
Marketing Digital Avanzado | Clímaco Briones Falcón | Negocios | 16 hs | 2.2
  
```

Una vez generada la lista, se seleccionó la opción **Buscar curso**, eligiendo el campo nombre y proporcionando el valor "Diseño UX Básico".

Dado que la lista estaba en estado **desordenado**, el sistema aplicó el algoritmo de **búsqueda lineal**, recorriendo uno por uno los elementos hasta encontrar una coincidencia exacta.

Resultado:

El curso fue encontrado correctamente mediante búsqueda lineal, con un **tiempo de respuesta de 0.000046 segundos**.

Este resultado demuestra que, en listas pequeñas, la búsqueda lineal es eficiente y suficiente.

```

--- MENU ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Diseño UX Básico

Resultado búsqueda lineal: Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3 (en 0.000046 segundos)
  
```

Este tipo de prueba resulta útil para verificar el correcto funcionamiento del sistema en escenarios simples, asegurando que los algoritmos de búsqueda responden como se espera cuando aún no se ha aplicado ningún ordenamiento.

Prueba 2: Ordenamiento de cursos por nombre (Bubble Sort vs Quick Sort)

Luego de realizar una búsqueda en la lista desordenada, se optó por aplicar un **ordenamiento alfabético por nombre del curso**. Esta operación es clave para habilitar posteriormente búsquedas binarias eficientes y para mejorar la presentación de los datos.

Al seleccionar la opción "**Ordenar cursos**" y elegir el campo **nombre**, el sistema ejecutó **dos algoritmos de ordenamiento clásicos**: Bubble Sort y Quick Sort, permitiendo comparar su rendimiento incluso sobre una lista pequeña.

Resultados de ordenamiento:

Bubble Sort

Tiempo de ejecución: **0.000648 segundos**

La lista resultante fue:

- ❖ Álgebra Avanzado
- ❖ Algoritmos Básico
- ❖ Cálculo Intermedio
- ❖ Diseño Industrial Avanzado
- ❖ Diseño UX Básico
- ❖ Escultura Avanzado
- ❖ Estadística Intermedio
- ❖ Historia de América Intermedio
- ❖ Marketing Digital Avanzado
- ❖ Python Intermedio

Quick Sort

Tiempo de ejecución: **0.000528 segundos**

La lista ordenada es **idéntica**, lo que valida que ambos algoritmos funcionan correctamente en cuanto a resultado.

TRABAJO PRÁCTICO INTEGRADOR

```
--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 1

--- Ordenar cursos ---
Ordenar por (nombre/profesor/tema/duracion/calificacion): nombre
Lista ordenada con Bubble Sort por nombre en 0.000648 segundos:

Álgebra Avanzado | Vanesa Montoya Mármol | Matemáticas | 38 hs | 1.8
Algoritmos Básico | Iván Vergara Piñol | Programación | 42 hs | 3.3
Cálculo Intermedio | Heriberto Eustaquio Esparza Ríos | Matemáticas | 13 hs | 1.1
Diseño Industrial Avanzado | Anacleto Santamaría Río | Diseño | 12 hs | 3.3
Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3
Escultura Avanzado | Jose Ignacio Sotelo Cañellas | Arte | 31 hs | 2.0
Estadística Intermedio | Victoriano Rosales Solana | Matemáticas | 34 hs | 4.0
Historia de América Intermedio | Godofredo de Castell | Historia | 31 hs | 2.9
Marketing Digital Avanzado | Climaco Briones Falcón | Negocios | 16 hs | 2.2
Python Intermedio | Aurelio de Caparrós | Programación | 31 hs | 4.0

Lista ordenada con Quick Sort por nombre en 0.000528 segundos:

Álgebra Avanzado | Vanesa Montoya Mármol | Matemáticas | 38 hs | 1.8
Algoritmos Básico | Iván Vergara Piñol | Programación | 42 hs | 3.3
Cálculo Intermedio | Heriberto Eustaquio Esparza Ríos | Matemáticas | 13 hs | 1.1
Diseño Industrial Avanzado | Anacleto Santamaría Río | Diseño | 12 hs | 3.3
Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3
Escultura Avanzado | Jose Ignacio Sotelo Cañellas | Arte | 31 hs | 2.0
Estadística Intermedio | Victoriano Rosales Solana | Matemáticas | 34 hs | 4.0
Historia de América Intermedio | Godofredo de Castell | Historia | 31 hs | 2.9
Marketing Digital Avanzado | Climaco Briones Falcón | Negocios | 16 hs | 2.2
Python Intermedio | Aurelio de Caparrós | Programación | 31 hs | 4.0
```

Análisis:

Aunque con una lista de solo 10 elementos las diferencias de tiempo son mínimas, ya se puede observar que **Quick Sort es ligeramente más eficiente**, como era esperable dado su menor complejidad algorítmica en promedio.

Esta prueba también permite comprobar que el sistema muestra correctamente **ambas versiones ordenadas**, facilitando una comparación visual y cuantitativa entre algoritmos.

Prueba 3: Búsqueda de curso por nombre en lista ordenada

Con la lista previamente ordenada por nombre, se realizó una nueva búsqueda del curso **"Diseño UX Básico"** para comparar la eficiencia entre los métodos de búsqueda lineal y búsqueda binaria.

Resultados obtenidos:

- ❖ **Búsqueda lineal** (sobre lista ordenada): encontró el curso en **0.000104 segundos**
- ❖ **Búsqueda binaria** (sobre lista ordenada): encontró el curso en **0.000011 segundos**

```
--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Diseño UX Básico

Resultado búsqueda lineal: Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3 (en 0.000104 segundos)
Resultado búsqueda binaria: Diseño UX Básico | René de Nevado | Diseño | 19 hs | 3.3 (en 0.000011 segundos)
```

Análisis:

Aunque la búsqueda lineal también encontró correctamente el curso, la búsqueda binaria demostró ser mucho más rápida, reduciendo significativamente el tiempo de búsqueda.

TRABAJO PRÁCTICO INTEGRADOR

Esto se debe a que la búsqueda binaria aprovecha que la lista está ordenada, dividiendo el espacio de búsqueda a la mitad en cada paso, lo que es especialmente beneficioso cuando el tamaño de la lista crece.

Este resultado refuerza la importancia de ordenar previamente los datos para optimizar las búsquedas cuando se trabaja con grandes volúmenes de información.

Pruebas con lista grande (1000 cursos)

Prueba 1: Búsqueda por nombre en lista desordenada (100 cursos)

Se generaron 1,000 cursos aleatorios para probar el comportamiento de los algoritmos con un tamaño de lista mayor.

```
iBienvenido! ¿Cuántos cursos desea generar? 1000

--- CURSOS GENERADOS (DESORDENADOS) ---
... Primeros 5 cursos ...
Historia Universal Intermedio | Jacinto Bernabé Gálvez Uriarte | Historia | 5 hs | 3.4
Estadística Intermedio | Berto Urbano Gelabert Novoa | Matemáticas | 43 hs | 4.1
Desarrollo Web Básico | Piedad Ordóñez Fernández | Programación | 8 hs | 2.0
Desarrollo Web Avanzado | Yolanda Tere Alegre Iglesia | Programación | 33 hs | 3.5
Inglés Avanzado | Amor del Escamilla | Idiomas | 41 hs | 2.0
... Últimos 5 cursos ...
Fotografía Intermedio | Lara Roselló Hernández | Arte | 9 hs | 4.5
Desarrollo Web Intermedio | Sebastián Badía Porcel | Programación | 40 hs | 4.8
Finanzas Avanzado | Julio Orozco-Bayona | Negocios | 15 hs | 3.4
Español para Extranjeros Avanzado | Severino Casal Zabaleta | Idiomas | 6 hs | 3.0
Cálculo Intermedio | Camila Acosta Solana | Matemáticas | 34 hs | 4.0

--- MENU ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Cálculo Intermedio

Resultado búsqueda lineal: Cálculo Intermedio | Humberto Ricardo Cuevas Pizarro | Matemáticas | 27 hs | 4.8 (en 0.001740 segundos)
```

Se realizó una búsqueda por nombre utilizando el valor "**Cálculo Intermedio**" sobre la lista desordenada.

- ❖ La búsqueda lineal encontró el curso en **0.001740 segundos**.

Como la lista no estaba ordenada, solo se pudo realizar búsqueda lineal, la cual, aunque efectiva, presenta un tiempo mayor al esperado en listas grandes.

Prueba 2: Ordenamiento de cursos por nombre (Bubble Sort vs Quick Sort)

Se ordenó la lista de 1,000 cursos por **nombre** utilizando dos algoritmos clásicos:

- **Bubble Sort:** completó la ordenación en **5.840608 segundos**.
La salida muestra los primeros y últimos 5 cursos ordenados alfabéticamente, evidenciando que el orden fue correctamente aplicado.
- **Quick Sort:** completó la ordenación en **0.387819 segundos**, mostrando un desempeño significativamente mejor que Bubble Sort para este tamaño de lista.
También se muestran los primeros y últimos 5 cursos, confirmando la correcta ordenación.

Este resultado evidencia la diferencia sustancial en eficiencia entre ambos algoritmos en listas grandes, siendo Quick Sort mucho más rápido y adecuado para grandes volúmenes de datos.

TRABAJO PRÁCTICO INTEGRADOR

```
--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 1

--- Ordenar cursos ---
Ordenar por (nombre/profesor/tema/duracion/calificacion): nombre

Lista ordenada con Bubble Sort por nombre en 5.840608 segundos:

... Primeros 5 cursos ...
Alemán Avanzado | Rolando Agapito Aznar Gonzalez | Idiomas | 36 hs | 1.6
Alemán Avanzado | Juliana Cantero Toledo | Idiomas | 39 hs | 2.6
Alemán Avanzado | Telmo Coello-Puerta | Idiomas | 28 hs | 4.7
Alemán Avanzado | Alejo Abad Marqués | Idiomas | 33 hs | 2.1
Alemán Avanzado | Inés Vilanova Mateo | Idiomas | 13 hs | 1.4
... Últimos 5 cursos ...
Python Intermedio | Rosa del Español | Programación | 9 hs | 2.7
Python Intermedio | Estefanía Mora Andrade | Programación | 10 hs | 3.3
Python Intermedio | Benigno Sierra-Montenegro | Programación | 6 hs | 1.1
Python Intermedio | Timoteo Sanmartín Manzanares | Programación | 46 hs | 4.6
Python Intermedio | Leonel Pedrero | Programación | 8 hs | 1.4

Lista ordenada con Quick Sort por nombre en 0.387819 segundos:

... Primeros 5 cursos ...
Alemán Avanzado | Melania Santamaría Company | Idiomas | 31 hs | 3.1
Alemán Avanzado | Augusto Camps Sobrino | Idiomas | 35 hs | 1.6
Alemán Avanzado | Eduardo Luján | Idiomas | 35 hs | 3.5
Alemán Avanzado | Rolando Alegre Gordillo | Idiomas | 37 hs | 1.1
Alemán Avanzado | Toni Arribas Bonilla | Idiomas | 35 hs | 2.9
... Últimos 5 cursos ...
Python Intermedio | Benigno Sierra-Montenegro | Programación | 6 hs | 1.1
Python Intermedio | Estefanía Mora Andrade | Programación | 10 hs | 3.3
Python Intermedio | Rosa del Español | Programación | 9 hs | 2.7
Python Intermedio | Celso Zabala Mateos | Programación | 31 hs | 1.3
Python Intermedio | Ciriaco Lope Iriarte Artigas | Programación | 11 hs | 4.9
```

Prueba 3: Búsqueda de curso por nombre en lista ordenada

Se realizó una búsqueda por **nombre** del curso "Cálculo Intermedio" en la lista ya ordenada:

- ❖ **Búsqueda lineal** encontró un resultado en **0.000669 segundos**.
- ❖ **Búsqueda binaria**, aprovechando la ordenación previa, encontró otro resultado en **0.000040 segundos**, demostrando ser mucho más rápida.

Dado que la lista contiene múltiples cursos con el mismo nombre, ambos métodos encontraron diferentes instancias, evidenciando que la búsqueda binaria es más eficiente cuando se trabaja sobre listas ordenadas.

```
--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Cálculo Intermedio

Resultado búsqueda lineal: Cálculo Intermedio | Humberto Ricardo Cuevas Pizarro | Matemáticas | 27 hs | 4.8 (en 0.000669 segundos)
Resultado búsqueda binaria: Cálculo Intermedio | Perlita Izquierdo Millán | Matemáticas | 17 hs | 4.5 (en 0.000040 segundos)
```

TRABAJO PRÁCTICO INTEGRADOR

Pruebas con gran volumen de datos (10.000 cursos)

Prueba 1: Búsqueda por nombre en lista desordenada (10.000 cursos)

Se realizó una búsqueda por **nombre** del curso "Español para Extranjeros Básico" en una lista desordenada de 10,000 cursos generados aleatoriamente.

- ❖ La **búsqueda lineal** encontró el curso en **0.001758 segundos**.

Este resultado ilustra que la búsqueda lineal, aunque sencilla y funcional en listas desordenadas, consume más tiempo a medida que aumenta la cantidad de elementos.

```
iBienvenido! ¿Cuántos cursos desea generar? 10000

--- CURSOS GENERADOS (DESORDENADOS) ---
... Primeros 5 cursos ...
Finanzas Avanzado | Camila Morena Reguera Colomer | Negocios | 25 hs | 2.7
Escultura Básico | Ani Cámara Moraleda | Arte | 48 hs | 2.8
Historia del Arte Básico | Leticia Azcona Mascaró | Arte | 18 hs | 2.7
Álgebra Avanzado | María Del Carmen Lorenzo Ruiz | Matemáticas | 39 hs | 3.2
Gestión Empresarial Básico | Faustino Terrón-Alegre | Negocios | 8 hs | 2.9
... Últimos 5 cursos ...
Español para Extranjeros Básico | Ambrosio Oliva Grau | Idiomas | 20 hs | 5.0
Historia de América Avanzado | Julie Codina-Fábreas | Historia | 23 hs | 1.2
Historia Medieval Básico | Rosario Huerta Báez | Historia | 25 hs | 4.0
Diseño UX Básico | Lara Maza Batlle | Diseño | 37 hs | 1.3
Algoritmos Básico | Innocencio Porras Luján | Programación | 47 hs | 5.0

--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Español para Extranjeros Básico

Resultado búsqueda lineal: Español para Extranjeros Básico | Vera Oliva Aguilera | Idiomas | 9 hs | 3.3 (en 0.001758 segundos)
```

Prueba 2: Ordenamiento de cursos por nombre (Bubble Sort vs Quick Sort)

Se ordenó una lista de 10,000 cursos generados aleatoriamente por el campo **nombre**, utilizando dos algoritmos: Bubble Sort y Quick Sort.

- ❖ El ordenamiento con Bubble Sort tomó 569.065710 segundos en completarse.
- ❖ El ordenamiento con Quick Sort tardó significativamente menos, 24.254292 segundos.

Este resultado evidencia la gran diferencia en eficiencia entre ambos métodos, donde Bubble Sort, con complejidad cuadrática, es poco práctico para grandes volúmenes de datos, mientras que Quick Sort, con complejidad promedio $O(n \log n)$, ofrece un rendimiento mucho más adecuado.

TRABAJO PRÁCTICO INTEGRADOR

```
--- Ordenar cursos ---
Ordenar por (nombre/profesor/tema/duracion/calificacion): nombre
Lista ordenada con Bubble Sort por nombre en 569.065710 segundos:

... Primeros 5 cursos ...
Alemán Avanzado | Maricela Román Porras | Idiomas | 11 hs | 4.5
Alemán Avanzado | Javier Téllez Marín | Idiomas | 20 hs | 4.1
Alemán Avanzado | Judith Cabezas Morera | Idiomas | 32 hs | 4.5
Alemán Avanzado | Pepito Romero | Idiomas | 39 hs | 4.8
Alemán Avanzado | Fátima Sola Carmona | Idiomas | 14 hs | 1.8
... Últimos 5 cursos ...
Python Intermedio | Marcio de Cortes | Programación | 32 hs | 3.4
Python Intermedio | Silvestre Arroyo Lobato | Programación | 13 hs | 2.8
Python Intermedio | Soraya Pascuala Amat Salamanca | Programación | 39 hs | 3.0
Python Intermedio | Ximena del Tellez | Programación | 30 hs | 2.0
Python Intermedio | Merche Montserrat Recio | Programación | 22 hs | 1.3

Lista ordenada con Quick Sort por nombre en 24.254292 segundos:

... Primeros 5 cursos ...
Alemán Avanzado | Jordi del Ferrero | Idiomas | 16 hs | 3.7
Alemán Avanzado | Maura Morán Santamaría | Idiomas | 9 hs | 2.5
Alemán Avanzado | Jenaro Marín Amador | Idiomas | 9 hs | 3.1
Alemán Avanzado | Che Cortes Padilla | Idiomas | 24 hs | 5.0
Alemán Avanzado | Andrés Tormo Zabala | Idiomas | 13 hs | 1.6
... Últimos 5 cursos ...
Python Intermedio | Noé Jáuregui Diego | Programación | 7 hs | 1.5
Python Intermedio | Daniela Barranco Pérez | Programación | 31 hs | 1.7
Python Intermedio | Epifanio Isern Muñoz | Programación | 11 hs | 4.0
Python Intermedio | Julio César Salvá Ibarra | Programación | 15 hs | 1.6
Python Intermedio | Marcelo Galán Figueroa | Programación | 8 hs | 4.4
```

Prueba 3: Búsqueda de curso por nombre en lista ordenada

Luego de ordenar la lista de 10,000 cursos por nombre, se realizó nuevamente la búsqueda del curso **"Español para Extranjeros Básico"**.

- ❖ La búsqueda lineal encontró el curso en 0.024379 segundos.
- ❖ La búsqueda binaria, posible gracias a la lista ordenada, lo encontró en 0.000067 segundos.

Este resultado demuestra la ventaja de la búsqueda binaria en listas ordenadas, mostrando un tiempo de respuesta mucho menor comparado con la búsqueda lineal, especialmente evidente en grandes volúmenes de datos.

```
--- MENÚ ---
1. Ordenar cursos
2. Buscar curso
3. Salir
Seleccione una opción: 2

--- Buscar curso ---
Buscar por (nombre/profesor/tema): nombre
Ingrese el valor a buscar en nombre: Español para Extranjeros Básico

Resultado búsqueda lineal: Español para Extranjeros Básico | Vera Oliva Aguilera | Idiomas | 9 hs | 3.3 (en 0.024379 segundos)
Resultado búsqueda binaria: Español para Extranjeros Básico | Pacífica América Verdú Montesinos | Idiomas | 21 hs | 4.1 (en 0.000067 segundos)
```

Conclusiones

Este trabajo práctico permitió simular una base de datos de cursos online, generando aleatoriamente diferentes atributos de los cursos, mientras que los nombres de los profesores fueron creados específicamente utilizando la librería Faker en un entorno virtual configurado en macOS. Esto facilitó la generación de datos realistas para los profesores sin necesidad de ingreso manual, haciendo posible realizar múltiples pruebas sobre estructuras de distinto tamaño.

Se implementaron y compararon distintos algoritmos de búsqueda (lineal y binaria) y de ordenamiento (Bubble Sort y QuickSort), evaluando su comportamiento en listas con volúmenes crecientes de datos: 10, 1,000 y 10,000 cursos.

Las pruebas permitieron concluir que:

- ❖ La búsqueda binaria es significativamente más rápida que la búsqueda lineal, siempre que la lista esté previamente ordenada.
- ❖ En una lista ordenada, la búsqueda por nombre puede arrojar múltiples resultados coincidentes, reflejando escenarios reales donde varios cursos pueden compartir el mismo nombre.
- ❖ Bubble Sort, aunque sencillo de implementar, se vuelve altamente ineficiente a medida que crece el tamaño de la lista, llegando a demorar varios minutos en ordenar 10,000 cursos.
- ❖ QuickSort demostró un rendimiento notablemente superior, ordenando 10,000 cursos en apenas segundos, siendo una opción mucho más práctica para grandes volúmenes de datos.
- ❖ Mostrar solo los primeros y últimos cinco elementos de una lista ordenada es una solución práctica para visualizar resultados sin saturar la consola en listas grandes.

Además de aplicar conceptos clave de algoritmos, esta experiencia reforzó la importancia de elegir la herramienta adecuada según el contexto y el tamaño de los datos, así como la utilidad de medir el desempeño para tomar decisiones informadas en la optimización de código.

Bibliografía

- Faker. (s.f.). *Faker Documentation (v18.13.0)*. Recuperado de <https://faker.readthedocs.io/en/master/>
- Computer Science Bytes. (s.f.). *Bubble Sort – Sorting Algorithms*. Recuperado de <https://www.computersciencebytes.com/sorting-algorithms/bubble-sort/>
- Contreras, J. (s.f.). *Algoritmos de búsqueda y ordenamiento*. Recuperado de <https://jorgecontrerasp.wordpress.com/unidad-i/algoritmos-de-busqueda-y-ordenamiento/>
- Hashemi, E. E. (2023). *QuickSort Explained*. Medium. Recuperado de <https://medium.com/@el.elhamhashemi/sorting-algorithms-quicksort-explained-b5bb814df35b>
- Python Oficial: <https://docs.python.org/3/library/>

Anexos

Código:

```
# Programación1
# TP-INTEGRADOR: Algoritmos de Búsqueda y Ordenamiento en Python
# Integrantes: Pamela Zampieri y Facundo Zampar
# Implementación de búsqueda y ordenamiento en una lista de cursos online generados
# automáticamente

from faker import Faker # Importa la librería Faker para generar datos falsos
import random # Importa random para generar números aleatorios
import time # Importa time para medir tiempos de ejecución
import unicodedata # Importa el módulo estándar unicodedata de Python, que permite
# trabajar con caracteres Unicode

# Crea una instancia de Faker configurada para español de España
fake = Faker('es_ES')

# Métodos de búsqueda
# Búsqueda binaria (lista previamente ordenada por la función clave)
# Devuelve el índice de una coincidencia (no necesariamente la primera si hay
# varias)
def busqueda_binaria(items, objetivo, funcion_clave):
    izquierda = 0
    derecha = len(items) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        valor_medio = funcion_clave(items[medio])
        if valor_medio == objetivo:
            return medio # devuelve el índice
        elif valor_medio < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return None # no encontrado

# Búsqueda lineal (lista no ordenada)
# Devuelve el índice de la primera coincidencia encontrada
def busqueda_lineal(items, objetivo, funcion_clave):
    for i in range(len(items)):
        if funcion_clave(items[i]) == objetivo:
            return i
    return None
```

TRABAJO PRÁCTICO INTEGRADOR

```
# Métodos de ordenamiento
# Bubble Sort (Ordenamiento burbuja)
def bubble_sort(lista, funcion_clave):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if funcion_clave(lista[j]) > funcion_clave(lista[j + 1]):
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

# QuickSort (Ordenamiento rápido)
def quick_sort(lista, funcion_clave):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if funcion_clave(x) <=
funcion_clave(pivote)]
        mayores = [x for x in lista[1:] if funcion_clave(x) > funcion_clave(pivote)]
        return quick_sort(menores, funcion_clave) + [pivote] + quick_sort(mayores,
funcion_clave)

# Diccionario con temas y nombres posibles de cursos relacionados
temas_cursos = {
    "Programación": ["Python", "Java", "C++", "Desarrollo Web", "Algoritmos"],
    "Arte": ["Pintura al Óleo", "Escultura", "Historia del Arte", "Fotografía"],
    "Historia": ["Historia Universal", "Historia de América", "Historia Medieval"],
    "Diseño": ["Diseño Gráfico", "Diseño UX", "Diseño Industrial"],
    "Negocios": ["Marketing Digital", "Gestión Empresarial", "Finanzas"],
    "Idiomas": ["Inglés", "Francés", "Alemán", "Español para Extranjeros"],
    "Matemáticas": ["Álgebra", "Cálculo", "Estadística", "Matemáticas Discretas"]
}

# Niveles de dificultad
niveles = ["Básico", "Intermedio", "Avanzado"]

class Curso:
    def __init__(self, nombre, profesor, tema, duracion, calificacion):
        self.nombre = nombre
        self.profesor = profesor
        self.tema = tema
        self.duracion = duracion
        self.calificacion = calificacion

    def __str__(self):
```

TRABAJO PRÁCTICO INTEGRADOR

```
        return f"{self.nombre} | {self.profesor} | {self.tema} | {self.duracion} hs  
| {self.calificacion:.1f}"  
  
# Método para generar cursos  
def generar_cursos(cantidad):  
    lista = []  
    for _ in range(cantidad):  
        tema = random.choice(list(temas_cursos.keys()))  
        nombre_base = random.choice(temas_cursos[tema])  
        nombre = f"{nombre_base} {random.choice(niveles)}"  
        profesor = fake.name()  
        duracion = random.randint(5, 50)  
        calificacion = round(random.uniform(1.0, 5.0), 1)  
        lista.append(Curso(nombre, profesor, tema, duracion, calificacion))  
    return lista  
  
# Método para obtener el campo a buscar  
def obtener_funcion_clave_campo(campo):  
    campo = campo.lower()  
    if campo == "nombre":  
        return lambda c: c.nombre  
    elif campo == "profesor":  
        return lambda c: c.profesor  
    elif campo == "tema":  
        return lambda c: c.tema  
    elif campo == "duracion":  
        return lambda c: c.duracion  
    elif campo == "calificacion":  
        return lambda c: c.calificacion  
    else:  
        return None  
  
# Mostrar los cursos  
def mostrar_cursos(lista):  
    for curso in lista:  
        print(curso)  
  
# Si son más de 50 cursos(umbral elegido) se muestra un resumen  
def mostrar_resumen_cursos(lista):  
    print("... Primeros 5 cursos ...")  
    for curso in lista[:5]:  
        print(curso)  
    print("... Últimos 5 cursos ...")  
    for curso in lista[-5:]:  
        print(curso)
```

TRABAJO PRÁCTICO INTEGRADOR

```
# Inicio
while True:
    try:
        print("-----")
        cantidad = int(input("¡Bienvenid@! ¿Cuántos cursos desea generar? "))
        if cantidad > 0:
            break
        else:
            print("Ingrese un número mayor que cero.")
    except ValueError:
        print("Por favor, ingrese un número válido.")

# Convierte el texto a minúsculas y elimina acentos para facilitar comparaciones y ordenamientos.
def normalizar_texto(texto):
    texto = texto.lower()
    texto = unicodedata.normalize('NFD', texto)
    texto = ''.join(c for c in texto if unicodedata.category(c) != 'Mn')
    return texto

cursos = generar_cursos(cantidad)
criterio_ordenamiento = ""

print("\n--- CURSOS GENERADOS (DESORDENADOS) ---")
mostrar_resumen_cursos(cursos) if len(cursos) > 50 else mostrar_cursos(cursos)

while True:
    print("\n--- MENÚ ---")
    print("1. Ordenar cursos")
    print("2. Buscar curso")
    print("3. Salir")

    opcion = input("Seleccione una opción: ")

    if opcion == "1":
        print("\n--- Ordenar cursos ---")
        campo = input("Ordenar por (nombre/profesor/tema/duracion/calificacion): ")
        campo = campo.strip().lower()

        funcion_clave_original = obtener_funcion_clave_campo(campo)
        if funcion_clave_original is None:
            print("Campo no válido. Intente de nuevo.")
            continue
```

TRABAJO PRÁCTICO INTEGRADOR

```
# Normalización si corresponde
if campo in ["nombre", "profesor", "tema"]:
    funcion_clave = lambda c: normalizar_texto(funcion_clave_original(c))
else:
    funcion_clave = funcion_clave_original

lista_bubble = cursos.copy()
inicio = time.time()
bubble_sort(lista_bubble, funcion_clave)
tiempo_bubble = time.time() - inicio

lista_quick = cursos.copy()
inicio = time.time()
lista_quick_ordenada = quick_sort(lista_quick, funcion_clave)
tiempo_quick = time.time() - inicio

print(f"\nLista ordenada con Bubble Sort por {campo} en {tiempo_bubble:.6f} segundos:\n")
mostrar_resumen_cursos(lista_bubble) if len(lista_bubble) > 50 else
mostrar_cursos(lista_bubble)

print(f"\nLista ordenada con Quick Sort por {campo} en {tiempo_quick:.6f} segundos:\n")
mostrar_resumen_cursos(lista_quick_ordenada) if len(lista_quick_ordenada) >
50 else mostrar_cursos(lista_quick_ordenada)

cursos = lista_bubble # Ahora cursos está ordenado por este campo
criterio_ordenamiento = campo

elif opcion == "2":
    print("\n--- Buscar curso ---")
    campo = input("Buscar por (nombre/profesor/tema): ").strip().lower()
    if campo not in ["nombre", "profesor", "tema"]:
        print("Campo no válido.")
        continue

    valor = input(f"Ingrese el valor a buscar en {campo}: ").strip()
    funcion_clave_original = obtener_funcion_clave_campo(campo)

    # Normalización
    funcion_clave = lambda c: normalizar_texto(funcion_clave_original(c))
    valor = normalizar_texto(valor)

    # Búsqueda lineal
    inicio = time.time()
```

TRABAJO PRÁCTICO INTEGRADOR

```
indice_lineal = busqueda_lineal(cursos, valor, funcion_clave)
tiempo_lineal = time.time() - inicio

if criterio_ordenamiento == campo:
    # Búsqueda binaria solo si la lista está ordenada por el campo
    inicio = time.time()
    indice_binaria = busqueda_binaria(cursos, valor, funcion_clave)
    tiempo_binaria = time.time() - inicio
else:
    indice_binaria = None
    tiempo_binaria = None

# Mostrar resultados
if indice_lineal is not None:
    print(f"\nResultado búsqueda lineal: {cursos[indice_lineal]} (en
{tiempo_lineal:.6f} segundos)")
else:
    print(f"No se encontró curso con búsqueda lineal (en {tiempo_lineal:.6f}
segundos)")

if tiempo_binaria is not None:
    if indice_binaria is not None:
        print(f"Resultado búsqueda binaria: {cursos[indice_binaria]} (en
{tiempo_binaria:.6f} segundos)")
    else:
        print(f"No se encontró curso con búsqueda binaria (en
{tiempo_binaria:.6f} segundos)")

elif opcion == "3":
    print("Saliendo del programa.")
    break

else:
    print("Opción inválida. Intente nuevamente.")
```

TRABAJO PRÁCTICO INTEGRADOR

Link del video demostrativo del entorno y ejecución del código:

<https://youtu.be/bMcE4Mmg8O0>

Link del repositorio remoto en GitHub:

<https://github.com/PamesZampieri/UTN-TP-Integrador-Programacion1-Zampar-Zampieri.git>