Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos:

Federico Durussel – federico.durussel@tupad.utn.edu.ar

Fernando Javier Elichiribehety - fernando.elichiribehety@tupad.utn.edu.ar

Materia: Programación I

Profesor: Ariel Enferrel

Fecha de Entrega: 20/06/2025

Índice

Introducción

Para nuestro trabajo integrador elegimos el eje temático **búsqueda y ordenamiento**. Los algoritmos para resolver este tipo de problemas los encontramos en las plataformas, aplicaciones y desarrollos web que más usamos. Nos encontramos con ellos en cada momento al navegar por internet o utilizar una aplicación. Si revisamos con atención siempre nos encontraremos buscando algun chat, alguna página web, información, datos, etc. Esos algoritmos nos permiten a nosotros encontrar lo que buscamos y para ello también deben estar ordenados. Acá nos aparece el ordenamiento como un pilar importante para estas tareas de búsqueda. El manejo eficiente de datos nos ayuda a optimizar, organizar y mejorar las búsquedas. Algunos ejemplos los encontramos en plataformas de e.commerce o

buscadores. Queda claro que estos deben ser algoritmos más complejos, pero como primera aproximación es importante ver donde se utilizan y que aplicación real se puede observar. Por eso elegimos este tema. Tiene gran presencia en nuestra cotidianidad digital.

La búsqueda y el ordenamiento nos permite recuperar información de manera mucho más rápida y eficiente. No solo por los tiempos de espera a nivel usuario sino encontrar lo que se necesita. Es importante tener en cuenta que aprender este tipo de algoritmos nos acerca a entender como operar con datos e información. El crecimiento de la información digital cada día es más amplio y vasta.

En el siguiente trabajo se encontrará con el estudio que hemos realizado sobre el tema. Mediante la manipulación de una hoja de cálculos con diferentes funciones hemos encontrado que tipo y para qué sirven algunos de los algoritmos usados. Nos permitirá entender y organizar nuestro entendimiento sobre el tema.

Marco Teórico

Los algoritmos de ordenamiento son importantes para la gestión de grandes o pequeñas cantidades de datos. Estos nos permiten de manera eficaz mantener listas grandes de datos de manera ordenada. Existen diferentes algoritmos que pueden ser usados para diferentes cantidades de datos. Los algoritmos de búsqueda nos permiten encontrar los datos necesarios. Estos últimos deben estar ordenados (o no, dependiendo el tipo de búsqueda). El ordenamiento es un concepto central ya que esto repercute directamente en la experiencia de usuario y rendimiento del código.[1]

Debemos tener en cuenta tres conceptos a la hora de analizar los algoritmos. El **tiempo de búsqueda** que es la cantidad de tiempo en lo que tardar en encontrar el dato buscado. La cantidad de **memoria usada**, eso hace que el programa donde usamos ese algoritmo sea más pesado o no y por lo tanto se repercuta en el rendimiento del dispositivo que opera. Y la escalabilidad que hace referencia a cómo el rendimiento de un algoritmo de ordenamiento se comporta a medida que aumenta el tamaño de los datos.

Vamos a clasificar los algoritmos ordenamiento desarrollados y probados: [2][4]

Ordenamiento de burbuja (o bubble sort)

Compara elementos adyacentes e intercambia aquellos que están en el orden incorrecto. Simple de entender, pero ineficiente para grandes conjuntos de datos. **Ideal para listas pequeñas**.

- **Peor caso**: O(n²), donde n es el número de elementos en la lista. Esto ocurre cuando la lista está en orden inverso.
- **Mejor caso:** O(n), cuando la lista ya está ordenada (con una optimización que detecta si no hubo intercambios).

Código de ejemplo.

Ordenamiento por inserción (o insertion sort)

Construye una lista ordenada insertando cada elemento en su posición correcta. Eficiente para listas pequeñas y casi ordenadas. El algoritmo construye una parte ordenada de la lista uno por uno, insertando cada elemento en su posición correcta.

- **Peor caso**: O(n²), cuando la lista está en orden inverso.
- **Mejor caso**: O(n), cuando la lista ya está ordenada

Código de ejemplo.

Ordenamiento por selección (o selection sort)

El algoritmo de selección funciona encontrando el elemento más pequeño en la lista no ordenada y colocándolo en la primera posición. Luego, repite este proceso para el resto de la lista, encontrando el siguiente elemento más pequeño y colocándolo en la siguiente posición, y así sucesivamente hasta que toda la lista esté ordenada. **Puede ser más eficiente cuando la lista está casi ordenada, ya que realiza menos comparaciones en ese caso.**

- **Peor caso**: O(n²), ya que siempre realiza comparaciones para encontrar el mínimo.
- **Mejor caso**: O(n²), incluso si la lista está ordenada.

Código de ejemplo.

Ordenamiento por Quicksort

Quicksort es un algoritmo eficiente que utiliza la técnica de "divide y vencerás". Funciona seleccionando un elemento pivot y particionando la lista en dos sublistas: una con elementos menores que el pivot y otra con elementos mayores. Luego, se aplica Quicksort recursivamente a cada sublista. **Generalmente es más eficiente que inserción y selección para listas grandes, especialmente cuando se implementa de manera eficiente.**

- **Peor caso**: O(n^2), cuando el pivote seleccionado es el menor o mayor elemento en cada partición (por ejemplo, si la lista ya está ordenada).
- **Mejor caso**:O(n log n), cuando el pivote divide la lista en dos partes aproximadamente iguales.

Código de ejemplo.

Ahora calcificamos los algoritmos ordenamiento desarrollados y probados: [3][4]

Búsqueda Lineal

Método simple que revisa cada elemento de la lista. Este código es funcional pero poco eficiente, tardaría mucho tiempo en buscar un valor dentro de una lista larga, por ejemplo.

- **Peor caso:** O(n), donde n es el número de elementos en la lista. Esto ocurre cuando el elemento buscado está al final de la lista o no está presente.
- **Mejor caso:** O(1), cuando el elemento buscado es el primero de la lista.
- **Caso promedio:** O(n), porque en promedio se recorren la mitad de los elementos.

Código de ejemplo.

Búsqueda binaria

Método eficiente que requiere que la lista esté ordenada. Esto quiere decir que la lista tiene que ser ordenada primero.

Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento buscado con el elemento central. Si el elemento buscado es menor que el central, se descarta la mitad derecha; si es mayor, se descarta la mitad izquierda.

- **Peor caso:** O(log n), donde n es el número de elementos en la lista. Esto ocurre cuando el elemento no está presente o está en una de las divisiones finales.
- **Mejor caso:** O(1), cuando el elemento buscado está justo en el centro de la lista.
- **Caso promedio:** O(log n), porque el algoritmo divide la lista en mitades en cada iteración.

Código de ejemplo.

Caso Práctico

Breve descripción del problema a resolver:

Nuestro objetivo fue crear un programa capaz de buscar información rápidamente dentro de una lista muy grande de vehículos. Esta lista, contenida en un archivo llamado patentes.csv, tiene **100.000 registros**, donde cada uno incluye datos como la patente, marca, modelo y año del automóvil. El desafío principal era encontrar los detalles de un vehículo en particular usando solo su número de patente, haciéndolo de la manera más eficiente posible.

Explicación de decisiones de diseño:

Para resolver el problema de buscar rápidamente en una lista de 100.000 vehículos, decidimos cargar los datos del archivo CSV en una lista de diccionarios Python, lo que permite un acceso claro a la información de cada vehículo. La clave de nuestra eficiencia fue la combinación de dos algoritmos: primero, utilizamos **Quicksort** para ordenar la lista por patentes, ya que los métodos de ordenamiento más simples resultaron ser demasiado lentos para este volumen de datos. Una vez ordenada, empleamos la **Búsqueda Binaria**, un algoritmo extremadamente rápido para encontrar un vehículo específico. Finalmente, nos aseguramos de que el programa

funcionara en cualquier sistema operativo usando el módulo os para gestionar las rutas de los archivos de manera universal.

Código Fuente:

```
import csv
import time # Para medir el tiempo
import os

lista_patentes = []

script_dir = os.path.dirname(os.path.abspath(__file__))
csv_path = os.path.join(script_dir, 'patentes.csv')

# Se abre el archivo utilizando la ruta construida.
with open(csv_path, 'r', newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        lista_patentes.append(fila)

# El archivo patentes.csv tiene 100.000 elementos (patentes) con sus datos
correspondientes del automovil.

def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivot = lista[0]
        # Las comparaciones <= y > funcionan lexicográficamente con cadenas
        less = [x for x in lista[1:] if x['patente'] <= pivot['patente']]
        greater = [x for x in lista[1:] if x['patente'] > pivot['patente']]
        return quicksort(less) + [pivot] + quicksort(greater)

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        patente_actual = lista[medio]['patente']
        if patente_actual == objetivo:
            return lista[medio]
        elif patente_actual < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

```
def mostrar_vehiculo(vehiculo):
    print(f"El automóvil patente {vehiculo['patente']}, marca
{vehiculo['marca']}, "
            f"modelo {vehiculo['modelo']}, año {vehiculo['anio']}, color
{vehiculo.get('color', 'desconocido')} "
            f"se encuentra en la base de datos.")

print("--- Ordenamiento con Quicksort de patentes en lista ---")
print("Primeros 3 vehículos sin ordenar:")
for v in lista_patentes[:3]:
    mostrar_vehiculo(v)

# Se ordena la lista
inicio_tiempo_quicksort = time.time() # Comienza a medir tiempo
lista_patentes_ordenada = quicksort(lista_patentes)
fin_tiempo_quicksort = time.time() # Termina contador

print("Primeros 3 vehículos ordenados por patente:")
for v in lista_patentes_ordenada[:3]:
    mostrar_vehiculo(v)

print(f"---Tiempo de ordenamiento: {(fin_tiempo_quicksort -
inicio_tiempo_quicksort):.6f} seg---")

print("\n" + "/" * 40 + "\n")

print("---Busqueda BINARIA de patentes en lista.---")
print("Ejemplo: NYG296 o PNZ619")
patente = input("Ingrese la patente interesada en buscar: ")

inicio_lineal = time.time() # Comienza a medir tiempo
resultado = busqueda_binaria(lista_patentes_ordenada, patente)
if resultado != -1:
    mostrar_vehiculo(resultado)
else:
    print("Patente no encontrada.")

fin_lineal = time.time() # Termina contador
print(f"---Tiempo de busqueda: {(fin_lineal - inicio_lineal):.6f} seg---")
```

Captura de pantalla del funcionamiento del programa:

Patente existente

```
--- Ordenamiento con Quicksort de patentes en lista ---
Primeros 3 vehículos sin ordenar:
El automóvil patente DCM043, marca CitroÃ«n, modelo C3, año 2010, color Azul se encuentra en la base de datos.
El automóvil patente OY234AG, marca CitroÃ«n, modelo C5 Aircross, año 2010, color Verde se encuentra en la base de datos.
El automóvil patente IQV868, marca Mercedes-Benz, modelo Clase G, año 2018, color Negro se encuentra en la base de datos.
Primeros 3 vehículos ordenados por patente:
El automóvil patente AA003SN, marca Volkswagen, modelo Polo, año 1998, color Gris se encuentra en la base de datos.
El automóvil patente AA019ML, marca Chevrolet, modelo Cruze, año 2021, color Blanco se encuentra en la base de datos.
El automóvil patente AA097BS, marca Nissan, modelo Kicks, año 2022, color Verde se encuentra en la base de datos.
---Tiempo de ordenamiento: 1.625933 seg---

//////////////////////////////////////

---Busqueda BINARIA de patentes en lista.---
Ejemplo: NYG296 o PNZ619
Ingrese la patente interesada en buscar: NYG296
El automóvil patente NYG296, marca Mercedes-Benz, modelo GLA, año 2016, color Gris se encuentra en la base de datos.
---Tiempo de busqueda: 0.000337 seg---
```

Patente no existente

```
--- Ordenamiento con Quicksort de patentes en lista ---
Primeros 3 vehículos sin ordenar:
El automóvil patente DCM043, marca CitroÃ«n, modelo C3, año 2010, color Azul se encuentra en la base de
 datos.
El automóvil patente OY234AG, marca CitroÃ«n, modelo C5 Aircross, año 2010, color Verde se encuentra en
 la base de datos.
El automóvil patente IQV868, marca Mercedes-Benz, modelo Clase G, año 2018, color Negro se encuentra en
 la base de datos.
Primeros 3 vehículos ordenados por patente:
El automóvil patente AA003SN, marca Volkswagen, modelo Polo, año 1998, color Gris se encuentra en la ba
se de datos.
El automóvil patente AA019ML, marca Chevrolet, modelo Cruze, año 2021, color Blanco se encuentra en la
base de datos.
El automóvil patente AA097BS, marca Nissan, modelo Kicks, año 2022, color Verde se encuentra en la base
 de datos.
---Tiempo de ordenamiento: 1.531005 seg---

//////////////////////////////////////

---Busqueda BINARIA de patentes en lista.---
Ejemplo: NYG296 o PNZ619
Ingrese la patente interesada en buscar: ZZZ123
Patente no encontrada.
---Tiempo de busqueda: 0.000240 seg---
```

Metodología Utilizada

La metodología de trabajo se puede ordenar de la siguiente manera:

1. Primero nos informamos de los temas para el trabajo.

2. Realizamos las actividades relacionadas con el tema en la plataforma de la UTN. Y nos encargamos de buscar más información en algunas páginas web.

3. Creamos el repositorio donde trabajar y desarrollamos los algoritmos para las pruebas tabla obtenida por IA.

4. Realizamos la prueba y posteriormente luego de verificar, desarrollamos el programa principal para la hoja de cálculo.

5. Escribimos los archivos de texto: README.md en el repositorio y el presente trabajo.

6. Se crea el video explicativo.

Resultados Obtenidos

Los resultados obtenidos se pueden resumir en:

1. Se realizaron las pruebas correctamente y se pudieron elegir los algoritmos correctos para la hoja de cálculos. Se anotaron y se pueden ver en la ejecución los tiempos de respuesta.

2. El programa principal ordena de manera correcta las patentes y los datos asociados al vehículo. Mostrando la información completa al ingresar la patente.

3. Se pudo encontrar la solución al ejecutarse en diferentes sistemas operativos.

4. La búsqueda binaria y el ordenamiento por quick sort fueron los seleccionados para el programa. Ambos se comportan de manera eficaz y sin interrupciones. Teniendo un tiempo de respuesta para 100.000 entradas muy bajo.

5. Creamos repositorio donde se encuentra el código fuente, el programa principal y el video explicativo. Compartimos enlace: https://github.com/AIT-4/Busqueda-y-Ordenamiento-UTN-TUPaD-P1

Conclusiones

Este proyecto integrador ha sido una experiencia sumamente valiosa y enriquecedora para nuestro equipo, consolidando los conocimientos teóricos de Programación 1 con su aplicación práctica en un escenario real.

Qué se aprendió al hacer el trabajo:

Durante la realización de este proyecto, logramos una comprensión práctica de los algoritmos de ordenamiento, como Quicksort, y de búsqueda, como la Búsqueda Binaria. Aprendimos a implementar su lógica y, lo más importante, experimentamos de primera mano cómo su eficiencia es crucial al manejar grandes volúmenes de datos, como los 100.000 registros del archivo patentes.csv. Comprobamos que algoritmos más simples son inviables para estos tamaños, mientras que Quicksort permite procesar la información rápidamente. También nos familiarizamos con el manejo de datos estructurados en Python a través de archivos CSV y el uso de diccionarios. Además, fue fundamental aprender a escribir código portátil utilizando funciones del módulo os para gestionar rutas de archivos de manera compatible con cualquier sistema operativo. Finalmente, la colaboración con Git y GitHub nos proporcionó una experiencia práctica indispensable en el control de versiones y el trabajo en equipo.

Qué utilidad tiene el tema trabajado para la programación o para otros proyectos:

El dominio de los algoritmos de búsqueda y ordenamiento es un pilar fundamental en la programación. Su utilidad es inmensa y transversal a casi cualquier proyecto que involucre datos, ya que son la base para construir aplicaciones rápidas y eficientes, desde motores de búsqueda y bases de datos hasta sistemas de recomendación y plataformas de comercio electrónico. Estos algoritmos permiten

organizar la información para su posterior análisis, visualización o procesamiento, y la elección del algoritmo adecuado tiene un impacto directo en la escalabilidad y la experiencia del usuario de un sistema. Comprender estos algoritmos simples sienta las bases para abordar estructuras de datos y algoritmos más sofisticados en el futuro.

Dificultades que surgieron y cómo se resolvieron:

Durante el desarrollo del proyecto, surgieron varias dificultades que nos permitieron aprender y aplicar soluciones prácticas. La principal fue el rendimiento excesivo de algoritmos de ordenamiento como burbuja, inserción o selección con el dataset de 100.000 registros; esto nos llevó a la clara conclusión de su inviabilidad para grandes volúmenes de datos y nos impulsó a elegir y optimizar algoritmos con mejor complejidad temporal, como Quicksort, para el programa principal, validando así la teoría. Otra dificultad fueron las rutas de archivo multiplataforma, que se resolvieron eficientemente con el uso de las funciones os.path.dirname, os.path.abspath y os.path.join de Python.

En síntesis, este proyecto no solo ha mejorado nuestras habilidades de codificación en Python, sino que también nos ha proporcionado una perspectiva más profunda sobre la importancia de la eficiencia algorítmica y las mejores prácticas en el desarrollo de software colaborativo.

Bibliografía

1. Introducción Búsqueda y Ordenamiento. UTN-TUP. Programación 1. Video. Enlace al video.
2. Clase de ordenamiento. UTN-TUP. Programación 1. Video. Enlace al video.
3. Clase de búsqueda. UTN-TUP. Programación 1. Video. Enlace al video.
4. Búsqueda y ordenamiento. Colab Google. Enlace.
5. Estructura de datos. Algoritmos de búsqueda y ordenamiento. W3school.com. Enlace.

Anexos

1. Carpeta digital. Enlace.

2. Video explicativo. Parte uno. Enlace. Parte dos. Enlace.