UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Integrador Final

Algoritmos de Búsqueda y Ordenamiento

Estudiante: Camilo Quiroga

Materia: Programación 1

Tema: Algoritmo de Búsqueda y Ordenamiento.

Caso Practico: Sistema de gestión de turnos médicos.

Comisión: 19

Carrera: Tecnicatura Universitaria en Programación

Año: 2025

UNIVERSIDAD TECNOLOGICA NACIONAL

Índice

Introducción

En programación, los algoritmos de búsqueda y ordenamiento son herramientas clave para trabajar de forma eficiente con grandes volúmenes de datos. Están presentes en todo tipo de aplicaciones: desde buscadores, sistemas de inventario, hasta plataformas como redes sociales. Saber cómo funcionan y cuándo conviene usarlos puede marcar una gran diferencia en el rendimiento de un programa.
Este trabajo tiene como objetivo entender y poner en práctica algunos de los algoritmos más conocidos y usados del área: búsqueda lineal, búsqueda binaria (muy útil en listas ordenadas), y tres métodos de ordenamiento que se enseñan en todos los niveles de programación. Vamos a implementarlos en Python y el objetivo es ver cómo se comportan estos algoritmos en la práctica. Para eso vamos a medir el tiempo que tardan, cuánta memoria usan y qué diferencias aparecen al aplicarlos sobre listas de distintos tamaños.
Con esta información buscamos sacar conclusiones reales y aprender a elegir el algoritmo más adecuado según el contexto. Porque más allá de saber “cómo se hace”, lo importante es saber cuándo conviene usar cada uno.

Marco Teórico

Cuando trabajamos con listas o grandes cantidades de datos, una de las tareas más comunes es ordenar los elementos o buscar algo dentro de ellos. Para eso usamos lo que se llaman algoritmos de búsqueda y ordenamiento. Son técnicas ya conocidas que permiten resolver estos problemas de forma rápida y precisa, dependiendo del caso.

Ordenamiento

Ordenar significa poner los datos en cierto orden lógico, por ejemplo de menor a mayor, por nombre, por fecha, etc. Esto no solo sirve para que se vean más prolijos, sino que también hace más fácil y rápida la búsqueda después. Hay muchos tipos de algoritmos de ordenamiento. En este trabajo usamos tres:

Bubble Sort: compara de a pares los elementos y los va cambiando de lugar si están mal ubicados. Es muy fácil de entender, pero es lento si la lista es grande.

Quick Sort: elige un número (llamado pivote), divide la lista en dos (menores y mayores que ese número), y repite el proceso. En la mayoría de los casos es rápido y eficiente.

Merge Sort: parte la lista en mitades, las ordena por separado y después las junta. Funciona muy bien incluso cuando la lista es larga, pero usa más memoria.

Búsqueda

Buscar significa encontrar un dato dentro de una lista. Según cómo estén organizados los datos, podemos usar distintos métodos:

- **Búsqueda lineal**: revisa cada elemento uno por uno hasta encontrar lo que buscamos. Es fácil de hacer, pero si la lista es muy larga, puede tardar bastante.
- **Búsqueda binaria**: sólo funciona si la lista ya está ordenada. Mira el elemento del medio y decide si tiene que buscar a la izquierda o a la derecha. Va achicando la lista hasta encontrar el valor o darse cuenta de que no está. Es mucho más rápida que la lineal.

Comparación rápida

Algoritmo	**Tipo**	**Complejidad promedio**
Bubble Sort	Ordenamiento	**O(n²)**
Quick Sort	Ordenamiento	**O(n log n)**
Merge Sort	Ordenamiento	**O(n log n)**
Búsqueda lineal	Búsqueda	**O(n)**
Búsqueda binaria	Búsqueda	**O(log n)**

¿Cómo medimos los resultados?

Para probar cómo se comportan los algoritmos en la práctica, usamos Python y algunas herramientas específicas. El objetivo es ver con datos concretos cuál rinde mejor según el caso.
Tiempo de ejecución: lo medimos con el módulo time, que nos permite saber cuánto tarda cada algoritmo en terminar.
Uso de memoria: usamos memory_profiler para ver cuánta memoria consume cada uno mientras corre.
Visualización de resultados: con matplotlib generamos gráficos que muestran las diferencias de rendimiento entre los algoritmos cuando procesan listas de distintos tamaños.
Estas pruebas nos ayudan a ir más allá de la teoría, comparando realmente el rendimiento en situaciones concretas. Así podemos entender mejor cuál conviene usar en cada caso, según el tipo de datos y el contexto del problema.

UNIVERSIDAD TECNOLOGICA NACIONAL

Caso Práctico: Gestión de Turnos en una Clínica

Este caso práctico consiste en simular un pequeño sistema de gestión de turnos para una clínica médica. El sistema permite organizar los turnos de atención según el horario asignado y buscar pacientes por su número de documento. Esta solución apunta a optimizar la atención al paciente, evitando demoras o confusiones por desorden en la agenda. La simplicidad del sistema permite enfocarnos en los conceptos clave de la programación vistos durante la cursada, al mismo tiempo que se resuelve un problema cotidiano.

Objetivos del desarrollo:

- Implementar el algoritmo de Insertion Sort para ordenar los turnos por hora de manera ascendente.
- Utilizar una búsqueda lineal para encontrar rápidamente un paciente en la lista, usando su número de DNI como identificador.
- Mostrar los resultados por consola y validar que las funciones respondan correctamente en distintas situaciones.

Se eligió este ejemplo porque refleja una necesidad real de muchas instituciones: administrar turnos de forma organizada y permitir búsquedas rápidas ante consultas imprevistas. Al resolverlo con estructuras básicas en Python, logramos aplicar de forma concreta los conocimientos adquiridos.

Contenidos aplicados del curso:

Este caso integra varios temas abordados en la materia, incluyendo:

- **Estructuras Secuenciales:** ejecución paso a paso del flujo lógico del programa.
- **Estructuras Condicionales (if):** para tomar decisiones dentro de la búsqueda y el ordenamiento.
- **Estructuras Repetitivas (for, while):** utilizadas en ambos algoritmos, tanto para recorrer listas como para comparar valores.
- **Listas y Diccionarios**: como forma de almacenar los datos de los pacientes (DNI, nombre y hora de turno).

- **Funciones:** para dividir el problema en partes reutilizables, como ordenar la lista o realizar una búsqueda.
- **Lógica aplicada**: comprensión del problema, análisis de la mejor solución posible y validación de los resultados.

El desarrollo se realizó en Python por su sintaxis simple y clara, ideal para este tipo de problemas. Se realizaron pruebas con distintos conjuntos de datos para comprobar el correcto funcionamiento del ordenamiento y de la búsqueda.

Código:

```
!pip install -q memory_profiler          # Para que corra memory_profiler en colab

from datetime import datetime            # Para convertir y comparar horarios
from time import time                    # Para medir tiempo de ejecución
from memory_profiler import memory_usage  # Para medir uso de memoria
import matplotlib.pyplot as plt          # Para graficar los resultados
import random                            # Para generar turnos aleatorios
# Función auxiliar: convierte "HH:MM" a datetime para poder comparar horas
def hora_a_datetime(hora_str):
    return datetime.strptime(hora_str, "%H:%M")
# Algoritmo Insertion Sort adaptado a horarios de turnos
def insertion_sort_por_hora(turnos):
    for i in range(1, len(turnos)):
        actual = turnos[i]               # Turno actual a ubicar en su lugar correcto
        j = i - 1                        # Índice anterior
        # Mientras no llegamos al inicio y el turno anterior es posterior al actual
        while j >= 0 and hora_a_datetime(turnos[j]["hora"]) >
hora_a_datetime(actual["hora"]):
            turnos[j + 1] = turnos[j]    # Desplazamos el turno hacia la derecha
            j -= 1
        turnos[j + 1] = actual           # Insertamos el turno actual en la posición
correcta
    return turnos
```

```
# Búsqueda lineal por número de documento (DNI)
def buscar_por_dni(turnos, dni):
    for turno in turnos:
        if turno["dni"] == dni:
            return turno                # Devuelve el turno si lo encuentra
    return None                         # Si no está, devuelve None
# Lista de turnos fija, simulando una base de datos chica
turnos = [
    {"dni": 34567123, "nombre": "Lucía López", "hora": "10:30"},
    {"dni": 29876123, "nombre": "Carlos Pérez", "hora": "09:00"},
    {"dni": 41678901, "nombre": "Ana Toro", "hora": "11:15"},
    {"dni": 37890123, "nombre": "Julián Quiroga", "hora": "08:45"},
    {"dni": 30987654, "nombre": "Laura González", "hora": "09:45"}
]
# Medimos el tiempo y el uso de memoria de la ordenación
start_time = time()
mem_before = memory_usage()[0]
turnos_ordenados = insertion_sort_por_hora(turnos)
mem_after = memory_usage()[0]
end_time = time()
# Mostramos los turnos ya ordenados por hora
print("Turnos ordenados por hora:")
for turno in turnos_ordenados:
    print(f'{turno["hora"]} - {turno["nombre"]} ({turno["dni"]})')
# Imprimimos el tiempo de ejecución y el uso de memoria
print(f"\n⏱ Tiempo de ejecución: {end_time - start_time:.6f} segundos")
print(f" Memoria utilizada: {mem_after - mem_before:.4f} MiB")
# Búsqueda por DNI
dni_a_buscar = 30987654
resultado = buscar_por_dni(turnos_ordenados, dni_a_buscar)
```

```
print("\n Resultado de la búsqueda:")
if resultado:
    print(f'Turno encontrado: {resultado["hora"]} - {resultado["nombre"]}')
else:
    print("Paciente no encontrado.")

# -------- #

# Genera turnos aleatorios para simular bases de datos más grandes
def generar_turnos(n):
    turnos = []
    for i in range(n):
        hora = f"{random.randint(8, 18)}:{random.choice(['00', '15', '30', '45'])}"
        turnos.append({
            "dni": random.randint(20000000, 45000000),
            "nombre": f"Paciente {i+1}",
            "hora": hora
        })
    return turnos

# Lista de tamaños a probar y lista para guardar los tiempos medidos
tamaños = [10, 100, 500, 1000, 2000]
tiempos = []

# Medimos el tiempo que tarda Insertion Sort en cada caso
for n in tamaños:
    datos = generar_turnos(n)
    inicio = time()
    insertion_sort_por_hora(datos)
    fin = time()
    tiempos.append(fin - inicio)
```

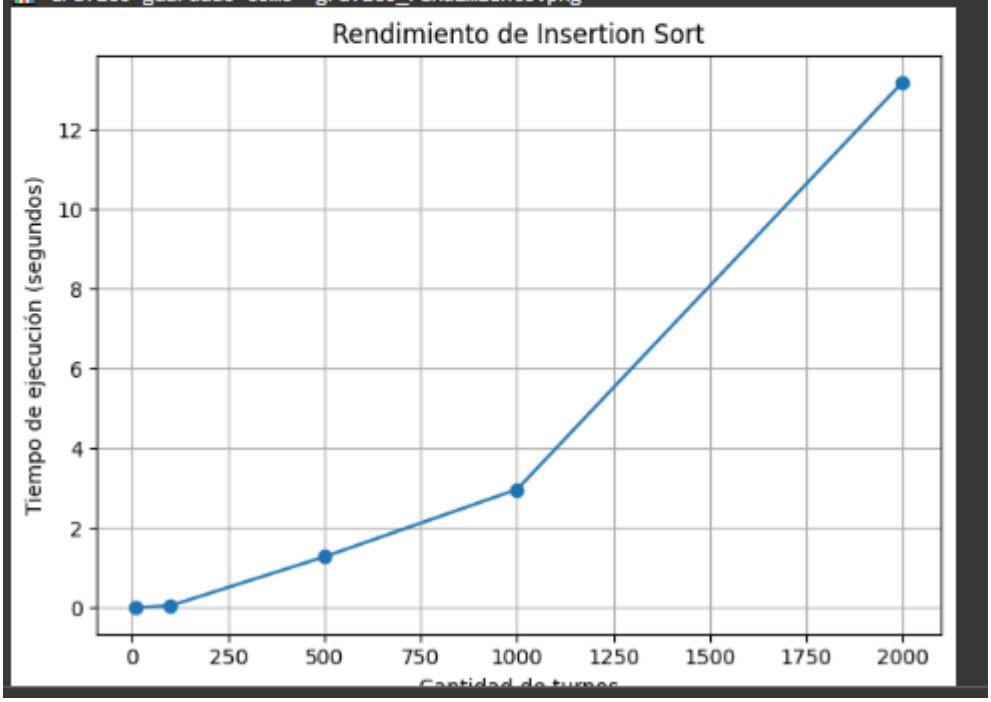
```
# Graficamos los resultados para visualizar la eficiencia
plt.plot(tamaños, tiempos, marker='o')
plt.title("Rendimiento de Insertion Sort")              # Título del gráfico
plt.xlabel("Cantidad de turnos")                        # Eje X
plt.ylabel("Tiempo de ejecución (segundos)")            # Eje Y
plt.grid(True)
plt.tight_layout()
plt.savefig("grafico_rendimiento.png")
print("📊 Gráfico guardado como 'grafico_rendimiento.png'")
```

Resultados en la consola

```
Turnos ordenados por hora:
08:45 - Julián Quiroga (37890123)
09:00 - Carlos Pérez (29876123)
09:45 - Laura González (30987654)
10:30 - Lucía López (34567123)
11:15 - Ana Toro (41678901)

⏱ Tiempo de ejecución: 0.202028 segundos
 Memoria utilizada: 0.0000 MiB

 Resultado de la búsqueda:
Turno encontrado: 09:45 - Laura González
📊 Gráfico guardado como 'grafico_rendimiento.png'
```



Metodología Utilizada

La realización de este trabajo integrador se organizó en una serie de etapas que combinan teoría, práctica y análisis aplicado:

Selección del tema: Se optó por trabajar con algoritmos de búsqueda y ordenamiento por su relevancia en la resolución de problemas concretos relacionados con estructuras de datos.
Recolección de información teórica: Se consultaron libros, documentación oficial de Python y materiales provistos en clase para comprender el funcionamiento detallado de los algoritmos seleccionados (Insertion Sort y búsqueda lineal).
Diseño del caso práctico: Se diseñó un sistema de gestión de turnos médicos como contexto aplicado. Se planteó ordenar pacientes según su horario y buscar rápidamente un turno por DNI, simulando un entorno real de uso.
Implementación en Python: Se programaron los algoritmos utilizando listas, diccionarios y funciones modulares. El desarrollo inicial se realizó en Visual Studio Code para tener mayor control del entorno local.
Pruebas con distintos datos: Se realizaron ensayos con volúmenes variables de información (listas de turnos de diferente tamaño), observando cómo afectaban el rendimiento de los algoritmos en escenarios pequeños y grandes.
Medición de rendimiento: Se integraron las librerías time y memory_profiler para calcular con precisión el tiempo de ejecución y el uso de memoria, y comparar la eficiencia del algoritmo de ordenamiento.
Visualización de resultados: Se utilizó matplotlib para graficar los tiempos de ejecución de Insertion Sort frente a distintas cantidades de turnos. Esto permitió visualizar de forma clara su crecimiento temporal en función del tamaño de entrada.
Validación en Google Colab: Para complementar el entorno de desarrollo local, también ejecutamos el código en Google Colab, lo que permitió verificar su funcionamiento en un entorno virtual limpio, evitando errores por configuraciones específicas del sistema.
Esto resultó útil para validar la portabilidad del código en distintos entornos, asegurando que la solución fuera funcional más allá del equipo personal de desarrollo.

Documentación del trabajo: Se registraron todas las etapas del proceso en este informe, incluyendo el código fuente debidamente comentado, los resultados de las mediciones, los gráficos generados, y capturas de pantalla. Además, se elaboró un video explicativo como complemento visual para facilitar la comprensión del proyecto.

Resultados Obtenidos

El programa ordenó de manera precisa los turnos médicos por horario utilizando el algoritmo Insertion Sort, permitiendo organizar la agenda de atención de forma lógica y accesible.

La búsqueda lineal fue efectiva al localizar pacientes por número de documento (DNI), incluso cuando se trabajó con listas de mayor tamaño, demostrando su utilidad en consultas rápidas.

Se comprobó que, aunque Insertion Sort no es el algoritmo más eficiente para grandes volúmenes de datos, su simplicidad lo convierte en una excelente opción para entornos con pocas entradas o necesidades de implementación inmediata.

El análisis del tiempo de ejecución y el consumo de memoria evidenció que el rendimiento de los algoritmos está directamente influenciado por el tamaño de la lista y el orden previo de los datos.

Esta experiencia reforzó la importancia de estructurar correctamente los datos, especialmente en aplicaciones donde el tiempo de respuesta y la precisión de la búsqueda son factores clave.

Además, el desarrollo permitió afianzar conceptos fundamentales como estructuras de control, funciones, listas y diccionarios en Python, aplicando estos conocimientos en un contexto realista con impacto práctico.

Conclusiones

Trabajar con algoritmos de búsqueda y ordenamiento en un caso real como la gestión de turnos en una clínica nos permitió comprender su importancia más allá de lo teórico. Pudimos ver que aplicar el algoritmo adecuado mejora notablemente la eficiencia de un programa, especialmente cuando se trata de manejar datos que cambian constantemente.

La búsqueda lineal demostró ser sencilla de implementar y útil en listas pequeñas, mientras que el ordenamiento con Insertion Sort fue suficiente para mantener el orden cronológico de los turnos, aunque no sería la mejor opción con una base de datos más grande.

Este trabajo no solo ayudó a afianzar conceptos como listas, funciones, estructuras condicionales y repetitivas, sino que también mostró cómo estas herramientas se integran para resolver problemas concretos. Además, practicar la medición del rendimiento nos dio una mirada más crítica sobre cómo pensar y escribir código eficiente. Entendimos que elegir bien el algoritmo no es solo una cuestión técnica, sino una decisión estratégica que depende del contexto, del volumen de datos y de lo que el sistema necesite resolver.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Python Software Foundation. (2024). *Python Language Reference*. Disponible en: https://docs.python.org/3/
- Khan Academy. *Algoritmos en Ciencias de la Computación*. Disponible en: https://es.khanacademy.org/computing/computer-science/algorithms
- Apuntes de la cátedra y materiales audiovisuales provistos en el aula virtual.
- Video explicativo de búsqueda binaria: https://www.youtube.com/watch?v=haF4P8kF4Ik&ab_channel=Tecnicatura

Anexos

- **Repositorio GitHub** https://github.com/camiloquirogadev/UTN-TUPaD-P1/tree/main/13%20Trabajo%20Integrador
- **Video explicativo del trabajo integrador**: https://youtu.be/6f-WdVHx6c4
- **Código en colab**: https://colab.research.google.com/drive/17kbn9hq1GXX-TXEH4Pbh7OTKj_d6EnKD