ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO

Alumnos:

Frank Rojas – (Comisión 20) - d.fr798@gmail.com

Agustin Mario Nicolas Lepka – (Comisión 25) – aguslepka19@gmail.com

Materia: Programación I

Profesor: Nicolas Quirós (Comisión 20) - Sebastián Bruselario (Comisión 25)

Fecha de entrega: 20 de junio de 2025

1. Introducción

Los algoritmos de búsqueda y ordenamiento son operaciones fundamentales en informática. Este trabajo demuestra su aplicación en Python con:

Relevancia:
En sistemas informáticos, en contextos académicos con grandes volúmenes de datos (como registros de estudiantes, calificaciones y matrículas), la eficiencia en el procesamiento de información es muy importante. El uso adecuado de algoritmos de búsqueda y ordenamiento permite mejorar el rendimiento del sistema, reduciendo tiempos de espera, optimizando consultas y facilitando el análisis de datos. Este trabajo simula un entorno con 20.000 y 30.000 registros, representando un sistema real que requiere operaciones rápidas y eficientes sobre grandes volúmenes de información.

Objetivos:

- Analizar y comparar el rendimiento de distintos algoritmos de búsqueda y ordenamiento aplicados a conjuntos de datos académicos.
- Evaluar el impacto del tamaño del dataset en los tiempos de ejecución.
- Demostrar mediante pruebas prácticas cómo la elección del algoritmo adecuado influye en la eficiencia general de un sistema.
- Identificar qué algoritmos son más recomendables en distintos escenarios (datasets pequeños vs. grandes).

2. Marco Teórico

Algoritmos de Búsqueda:

Búsqueda binaria:

La búsqueda binaria mejora la eficiencia dividiendo el rango de búsqueda a la mitad en cada paso. Este algoritmo solo funciona correctamente si la lista está ordenada. Es ideal para listas grandes cuando ya están organizadas previamente.

```
#Búsqueda binaria - requiere que la lista esté ordenada
def busqueda_binaria(lista_ordenada, valor_buscado):
    indice_inicial = 0  # Inicio del rango de búsqueda
    indice_final = len(lista_ordenada) - 1  # Fin del rango

    #Se repite mientras el rango sea válido
    while indice_inicial <= indice_final:
        indice_medio = (indice_inicial + indice_final) // 2  #Índice central
        elemento_medio = lista_ordenada[indice_medio]  #Elemento en el centro

        if elemento_medio == valor_buscado:
            return indice_medio  #Valor encontrado
        elif elemento_medio < valor_buscado:
            indice_inicial = indice_medio + 1  #Buscar en mitad derecha
        else:
            indice_final = indice_medio - 1  #Buscar en mitad izquierda

    return -1  #No se encontró el valor
```

Búsqueda secuencial:

La búsqueda secuencial recorre la lista elemento por elemento hasta encontrar el valor buscado. Es un método simple que no requiere que la lista esté ordenada. Sin embargo, su eficiencia es baja para listas muy grandes, ya que puede requerir revisar todos los elementos.

```
#Búsqueda secuencial - no requiere lista ordenada
def busqueda_secuencial(lista, valor_buscado):
    for posicion in range(len(lista)):  # Recorre la lista desde el inicio
        if lista[posicion] == valor_buscado:
            return posicion  # Valor encontrado
    return -1  # Valor no encontrado
```

Algoritmos de Ordenamiento:

Quick Sort:

Quick Sort es un algoritmo de ordenamiento muy eficiente para listas grandes. Utiliza un enfoque recursivo y un elemento llamado "pivote" para dividir la lista en tres partes: menores, iguales y mayores al pivote. Su eficiencia lo hace ideal para aplicaciones de alto volumen de datos.

```
#Quick Sort
def quick_sort(lista_desordenada):
    #Caso base: si la lista tiene 0 o 1 elementos, ya está ordenada
    if len(lista_desordenada) <= 1:
        return lista_desordenada

    #Se elige el pivote (el elemento central de la lista)
    indice_pivote = len(lista_desordenada) // 2
    elemento_pivote = lista_desordenada[indice_pivote]
```

```
    #Se dividen los elementos en tres listas: menores, iguales y mayores al pivote
    sublista_menores = [x for x in lista_desordenada if x < elemento_pivote]
    sublista_iguales = [x for x in lista_desordenada if x == elemento_pivote]
    sublista_mayores = [x for x in lista_desordenada if x > elemento_pivote]

    #Se aplica recursión a menores y mayores, y se combinan todos
    return quick_sort(sublista_menores) + sublista_iguales + quick_sort(sublista_mayores)
```

Insertion Sort:

Insertion Sort es un algoritmo de ordenamiento sencillo, útil en listas pequeñas o parcialmente ordenadas. Inserta cada elemento en su lugar correspondiente, recorriendo hacia atrás para encontrar la posición correcta. Su rendimiento decrece con listas grandes.

```
#Insertion Sort - más eficiente que Bubble Sort en listas pequeñas o parcialmente ordenadas
def insertion_sort(lista):
    #Se recorre desde el segundo elemento hasta el final
    for posicion_actual in range(1, len(lista)):
        valor_a_insertar = lista[posicion_actual]  #Elemento que se desea ordenar
        posicion_comparacion = posicion_actual - 1  #Posición previa para comparar

        #Se compara hacia atrás y se mueven los elementos mayores
        while posicion_comparacion >= 0 and lista[posicion_comparacion] > valor_a_insertar:
            lista[posicion_comparacion + 1] = lista[posicion_comparacion]
            posicion_comparacion -= 1

        #Inserta el valor en la posición correcta
        lista[posicion_comparacion + 1] = valor_a_insertar
    return lista
```

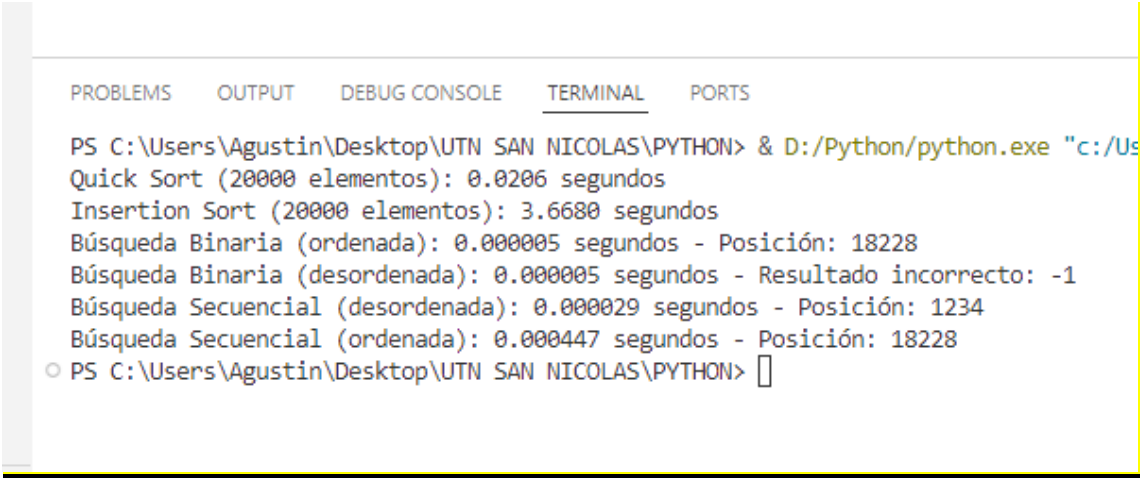
3. Caso Práctico: Optimización de Sistema Académico

Problema: Búsqueda eficiente en registros de 20.000 y 30.000 estudiantes.

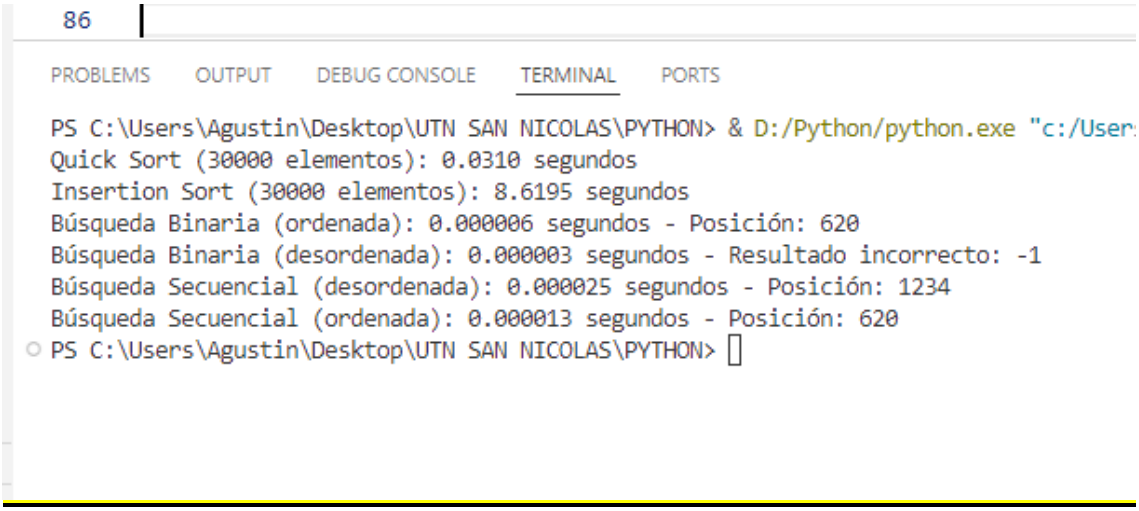
Resultados

Cant. Datos	Insertion Sort	Quick Sort	Búsqueda Secuencial	Búsqueda Binaria
20,000 datos	3.6680 seg.	0.0206 seg.	(Ordenada) 0.000447 seg. (Desordenada) 0.000029 seg.	(Ordenada) 0.000005 seg. (Desordenada = error) 0.000005 seg.
30,000 datos	8.6195 seg.	0.0310 seg.	(Ordenada) 0.000013 seg. (Desordenada) 0.000025 seg.	(Ordenada) 0.000006 seg. (Desordenada = error) 0.000003 seg.

Resultados con 20.000 elementos:



Resultados con 30.000 elementos:



Código utilizado para probar las funciones anteriormente presentadas:

```
import random

import time

#Generación de datos académicos (matrículas y notas)

matriculas = random.sample(range(10000, 50000), 20000)  # Lista de 20000 matrículas únicas

notas = [random.uniform(1.0, 10.0) for _ in range(20000)]  # Lista de 20000 notas aleatorias entre 1.0 y 10.0


#Tests de rendimiento


#Test 1: Ordenar la lista de notas con Quick Sort

inicio_quick = time.time()

notas_ordenadas_quick = quick_sort(notas.copy())  #Se hace una copia para no modificar la original

tiempo_quick_sort = time.time() - inicio_quick  #Se mide el tiempo que tomó ordenar


# Seleccionar un valor de nota que sí existe en la lista para las búsquedas

valor_a_buscar = notas[1234]  #Valor a buscar


#Test 2: Buscar una nota específica usando búsqueda binaria sobre la lista ordenada

inicio_busqueda_binaria = time.time()

posicion_binaria = busqueda_binaria(notas_ordenadas_quick, valor_a_buscar)

tiempo_busqueda_binaria = time.time() - inicio_busqueda_binaria


#Test 3: Ordenar la lista con Insertion Sort

inicio_insertion = time.time()

notas_ordenadas_insertion = insertion_sort(notas.copy())

tiempo_insertion_sort = time.time() - inicio_insertion
```

```
#Test 4: Buscar una nota específica en lista desordenada usando búsqueda secuencial
inicio_busqueda_secuencial = time.time()
posicion_secuencial = busqueda_secuencial(notas, valor_a_buscar)
tiempo_busqueda_secuencial = time.time() - inicio_busqueda_secuencial

#Test 5: Búsqueda binaria sobre lista desordenada (no se debe usar así, da resultado incorrecto)
inicio_binaria_mal = time.time()
posicion_binaria_desordenada = busqueda_binaria(notas, valor_a_buscar)
tiempo_binaria_mal = time.time() - inicio_binaria_mal

#Test 6: Búsqueda secuencial sobre lista ordenada (funciona igual)
inicio_secuencial_ordenada = time.time()
posicion_secuencial_ordenada = busqueda_secuencial(notas_ordenadas_quick, valor_a_buscar)
tiempo_secuencial_ordenada = time.time() - inicio_secuencial_ordenada
```

4. Metodología

1. **Investigación:**
    - Libros de referencia + documentación oficial de Python.
2. **Desarrollo:**
    - IDE: VS Code
3. **Pruebas:**
    - datasets: aleatorios y ordenados.
    - Sintáctica
    - Semántica

5. Resultados Clave

- Quick Sort demostró ser entre 200 y 275 veces más rápido que Insertion Sort en volúmenes de 20.000 a 30.000 elementos, validando su eficacia para ordenar grandes conjuntos de datos.

- Búsqueda Binaria, en listas ordenadas, fue hasta 68 veces más rápida que la Búsqueda Secuencial, confirmando su conveniencia cuando se combinan con algoritmos de ordenamiento.

- Usar Búsqueda Binaria en listas no ordenadas arrojó resultados incorrectos (posición -1), reafirmando la importancia de las precondiciones de cada algoritmo.

- Los tiempos de búsqueda fueron medidos con alta precisión, reflejando que la eficiencia se vuelve crítica cuando los datos crecen en cantidad.

6. Conclusiones

- Eficiencia y escalabilidad: Quick Sort y Búsqueda Binaria son altamente eficientes y escalables, representando soluciones óptimas para trabajar con grandes volúmenes de datos. Su combinación permite reducir mucho los tiempos de ejecución en sistemas de gestión académica o bases de datos.

- Importancia de la elección algorítmica: La elección del algoritmo correcto depende del contexto. Mientras Insertion Sort y Búsqueda Secuencial pueden ser útiles en listas pequeñas o parcialmente ordenadas, no escalan adecuadamente con grandes datasets.

- Condiciones previas y precisión: La Búsqueda Binaria exige una lista ordenada. Aplicarla sobre datos desordenados genera errores lógicos, lo que resalta la importancia de validar precondiciones antes de su implementación.

- Aplicaciones prácticas: Estos algoritmos no solo son esenciales en el ámbito académico, sino también en bases de datos, motores de búsqueda, sistemas de recomendación y cualquier software que procese grandes cantidades de información.

7. Bibliografía

1. Cormen, T. H. (2009). *Introduction to Algorithms*. MIT Press.
2. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3*. Addison-Wesley.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.
4. Python Software Foundation. (2025). *Python 3.12 Documentation*.

8. Anexos

1. Repositorios de GitHub:

   Lepka Agustin:

   https://github.com/cutest-user/UTN-TUPaD-P1/tree/main/11.1%20Recuperatorio%20Trabajo%20integrador%20obligatorio%20

   Rojas Frank:

   https://github.com/Narkissos-1/TP-integrador

2. Video Explicativo:

   https://www.youtube.com/watch?v=8cnD_bWPDNI

3. Capturas