

Trabajo Integrador: Programacion 1

Datos Generales

Título del trabajo: Busqueda y Ordenamiento de Datos

- **Alumnos:** Tiziano Caamaño (tizianocaamano@gmail.com), Lorenzo Blanco (blancolorenzo139@gmail.com)
 - **Materia:** Programacion 1
 - **Profesor/a:** Brian Esteban Lara Campos
 - **Fecha de Entrega:** 09/06/2025
-

Índice

1. Introducción
 2. Marco Teórico
 3. Caso Práctico
 4. Resultados Obtenidos
 5. Conclusiones
 6. Bibliografía
-

1. Introducción

Este trabajo se enfoca en los algoritmos de **búsqueda y ordenamiento**, herramientas clave en la programación para organizar y encontrar datos de forma eficiente. Se eligió este tema por su aplicación directa en el desarrollo de todo tipo de sistemas y por la importancia que tiene optimizar el manejo de la información.

El objetivo es comprender cómo funcionan estos algoritmos, analizar sus diferencias y ventajas, y aplicarlos en un caso práctico utilizando Python. A través de esta investigación, se busca valorar su impacto en la resolución de problemas cotidianos dentro del mundo del desarrollo de software.

2. Marco Teórico

una lista o un arreglo. Dependiendo de si la lista está ordenada o no, se pueden aplicar distintos métodos:

- **Búsqueda lineal:** recorre los elementos uno por uno hasta encontrar el valor deseado. Es simple, pero poco eficiente en listas grandes.
- **Búsqueda binaria:** solo funciona en listas ordenadas. Divide el conjunto en mitades sucesivas, descartando la mitad donde no puede estar el elemento. Es mucho más rápida que la búsqueda lineal, con una complejidad logarítmica ($O(\log n)$).
- **Búsqueda de interpolación:** Es un algoritmo de búsqueda que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Puede ser más eficiente que la búsqueda binaria para conjuntos de datos grandes con una distribución uniforme de valores.
- **Búsqueda de hash:** Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

Complejidad de los algoritmos

La complejidad medida con $O(n)$ es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.

Los algoritmos de ordenamiento organizan elementos en un orden específico (por ejemplo, de menor a mayor). Algunos de los más conocidos son:

- **Ordenamiento de Burbuja:** compara elementos adyacentes y los intercambia si están en el orden incorrecto. Es fácil de implementar, pero ineficiente en listas grandes.
- **Inserción:** inserta cada elemento en su lugar correspondiente dentro de una lista ordenada parcial. Es más eficiente que Bubble en listas casi ordenadas.
- **Selección:** divide la lista en partes más pequeñas, las ordena y luego las combina. Usa recursión y tiene una buena eficiencia ($O(n \log n)$).
- **QuickSort:** elige un "pivot" y reordena la lista según si los elementos son mayores o menores que él. Es uno de los algoritmos más usados por su velocidad promedio.

Elegir el algoritmo correcto depende del tamaño y estado de la lista, así como del tipo de datos. Mientras algunos algoritmos priorizan la simplicidad, otros ofrecen mejor rendimiento. En aplicaciones reales, como motores de búsqueda, bases de datos o comercio electrónico, el uso adecuado de estas técnicas marca una gran diferencia en la experiencia del usuario.

3. Caso Práctico

Una institución educativa desea realizar dos operaciones básicas sobre una lista de estudiantes:

1. **Ordenar los estudiantes según sus calificaciones** de mayor a menor.
 2. **Buscar la calificación de un estudiante específico** usando su nombre.
-
- Implementar una **búsqueda lineal** en una lista de tuplas.



```
# Lista de estudiantes (nombre, calificación)
estudiantes = [
    ("Ana", 85),
    ("Luis", 90),
    ("Carlos", 78),
    ("Marta", 92),
    ("Jorge", 88)
]

# 1. Ordenar estudiantes por calificación de mayor a menor usando Quicksort
def quicksort_calificación(lista):
    if len(lista) <= 1:
        return lista
    pivot = lista[0]
    mayores = [x for x in lista[1:] if x[1] > pivot[1]]
    iguales = [x for x in lista if x[1] == pivot[1]]
    menores = [x for x in lista[1:] if x[1] < pivot[1]]
    return quicksort_calificación(mayores) + iguales + quicksort_calificación(menores)

estudiantes_ordenados = quicksort_calificación(estudiantes)

print("Estudiantes ordenados por calificación (mayor a menor):")
for nombre, calificación in estudiantes_ordenados:
    print(f"{nombre}: {calificación}")

# 2. Quicksort para ordenar alfabéticamente por nombre
def quicksort_nombres(lista):
    if len(lista) <= 1:
        return lista
    pivot = lista[0]
    menores = [x for x in lista[1:] if x[0] < pivot[0]]
    iguales = [x for x in lista if x[0] == pivot[0]]
    mayores = [x for x in lista[1:] if x[0] > pivot[0]]
    return quicksort_nombres(menores) + iguales + quicksort_nombres(mayores)

estudiantes_por_nombre = quicksort_nombres(estudiantes)

# Búsqueda binaria por nombre (requiere lista ordenada alfabéticamente)
def búsqueda_binaria(lista_ordenada, nombre_buscar):
    izquierda = 0
    derecha = len(lista_ordenada) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        nombre_actual = lista_ordenada[medio][0]
        if nombre_actual == nombre_buscar:
            return f"{nombre_actual} tiene una calificación de {lista_ordenada[medio][1]}."
        elif nombre_actual < nombre_buscar:
            izquierda = medio + 1
        else:
            derecha = medio - 1
```

4. Resultados Obtenidos

- Logros alcanzados

Se logró **ordenar correctamente** la lista de estudiantes por calificación, desde la más alta a la más baja, utilizando Quicksort

La función de **búsqueda binaria** funcionó correctamente al verificar si un estudiante está presente en la lista y devolver su calificación.

Se aplicaron con éxito **dos operaciones básicas pero fundamentales**: ordenamiento y búsqueda, que forman parte de los pilares del procesamiento de datos en programación.

Ordenamiento de estudiantes:

- Lista original: Ana (85), Luis (90), Carlos (78), Marta (92), Jorge (88).
- Resultado esperado: Marta (92), Luis (90), Jorge (88), Ana (85), Carlos (78).
- El resultado fue el esperado.

Búsqueda de estudiante existente:

- Entrada: "Carlos"
 - Resultado esperado: "Carlos tiene una calificación de 78."
 - Resultado correcto.
-

5. Conclusiones

Ordenación eficiente: Se utilizó `sorted()` con `key=lambda x: x[1]` y `reverse=True` para ordenar por calificación descendente. Esto permitió visualizar rápidamente a los estudiantes con mejor calificación.

Búsqueda simple pero útil: La función `buscar_estudiante()` implementa una **búsqueda binaria** con la cual se encuentra de manera muy efectiva y rápida el estudiante en cuestión.

Aplicabilidad práctica: Este ejemplo puede escalar fácilmente a sistemas más complejos como bases de datos o sistemas escolares si se reemplaza la lista por estructuras más robustas (listas de objetos, diccionarios, archivos CSV, etc.).

6. Bibliografía

GeeksforGeeks en Español. (2025). *Algoritmos de búsqueda en Python*.

<https://es.geeksforgeeks.org/algoritmos-de-busqueda-en-python/>

GeeksforGeeks en Español. (2025). *Algoritmos de ordenamiento en Python*.

<https://es.geeksforgeeks.org/algoritmos-de-ordenamiento-en-python/>

W3Schools en Español. (2025). *Tutorial de Python – Búsqueda y Ordenamiento*.

<https://www.w3schools.com/python/default.asp>