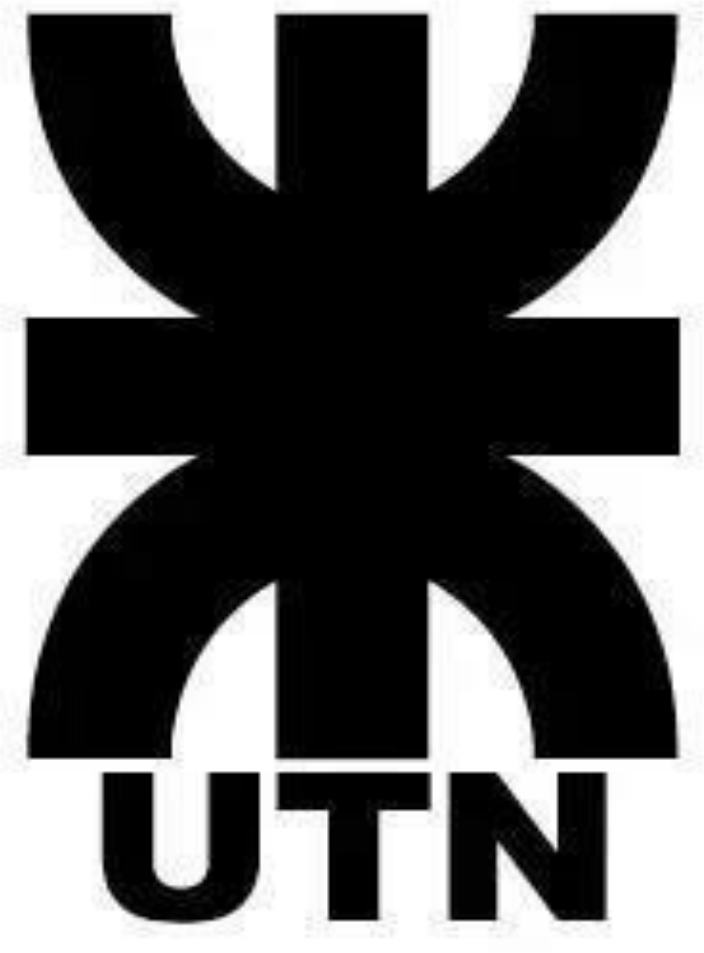
Trabajo Práctico Integrador de Programación

Estudio Comparativo de Algoritmos de Búsqueda y Ordenamiento en Python



Alumnos: Keyla Valdes – valdeskeyla05@gmail.com

Andres Vizone – andresvizone@gmail.com

Materia: Programación I

Profesor: Sebastián Bruselario

Fecha de Entrega: 9 de junio del 2025

ÍNDICE

1. Introducción

En este trabajo se realiza un análisis comparativo de diferentes algoritmos de búsqueda y ordenamiento, implementados en el lenguaje de programación Python. Se analizan tanto sus complejidades algorítmicas como su rendimiento práctico, evaluando tiempos de ejecución sobre listas de gran tamaño. El objetivo es comprender cómo la elección de un algoritmo adecuado puede influir significativamente en la eficiencia de un programa, especialmente en contextos donde se procesan grandes volúmenes de datos.

2. Marco Teórico

2.1 Algoritmos

Los algoritmos son procedimientos o conjuntos de instrucciones que permiten resolver problemas específicos, brindando la posibilidad de obtener grandes ahorros en tiempo y recursos computacionales. Su diseño cuidadoso es fundamental para la resolución eficiente de problemas complejos. Elegir el algoritmo adecuado puede implicar análisis matemáticos sofisticados y es esencial conocer los recursos que consumen antes de su implementación práctica.

2.2 Algoritmos de Búsqueda

El desarrollo tecnológico ha generado un acceso masivo a la información, por lo que la capacidad de buscar de forma eficiente dentro de grandes volúmenes de datos es fundamental. Los algoritmos de búsqueda permiten localizar elementos específicos dentro de estructuras de datos y son esenciales para muchas aplicaciones informáticas, influyendo directamente en el avance científico y tecnológico. Existen distintos tipos de algoritmos de búsqueda, entre los que destacan la búsqueda lineal y la búsqueda binaria:

- **Búsqueda Lineal (Linear Search):**

  Es el método más simple para buscar un elemento dentro de un conjunto de datos. Consiste en examinar secuencialmente cada elemento desde el inicio hasta encontrar una coincidencia con el valor buscado. Una vez localizado el elemento, la búsqueda finaliza inmediatamente.

  La eficiencia de este algoritmo está relacionada con la forma en que se insertan los elementos en la lista o estructura de datos. Por ejemplo, si la prioridad es insertar elementos rápidamente sin importar el orden, se puede añadir cada nuevo elemento en la última posición de un arreglo o al inicio de una lista enlazada. Esto implica que el conjunto resultante

estará ordenado según el momento de inserción y no por el valor de sus claves, lo cual puede afectar la velocidad de búsqueda. Sin embargo, la búsqueda lineal funciona correctamente tanto en listas ordenadas como desordenadas, lo que la hace sencilla y flexible, aunque no siempre eficiente para conjuntos grandes.

Este algoritmo tiene una complejidad temporal de O(n) en el peor caso, ya que en el peor escenario podría recorrer toda la lista para encontrar el elemento o confirmar que no está presente.

- **Búsqueda Binaria (Binary Search):**

Es un algoritmo eficiente que se utiliza para encontrar un elemento dentro de una lista ordenada. Funciona dividiendo repetidamente el intervalo de búsqueda a la mitad. Si el valor buscado es menor que el del medio, se continúa en la mitad inferior; si es mayor, en la mitad superior. Este proceso se repite hasta que se encuentra el elemento o el intervalo se reduce a cero.

Según la definición del *Dictionary of Algorithms and Data Structures* del Instituto Nacional de Estándares y Tecnología (NIST):
"La búsqueda binaria trabaja dividiendo repetidamente el intervalo de búsqueda a la mitad hasta que se encuentra el valor o el intervalo esté vacío" (Black, 2022).

Este algoritmo pertenece a la categoría de métodos de **divide y vencerás**, y presenta una **complejidad temporal de O(log n)** en el peor de los casos, lo que lo convierte en una opción muy eficiente frente a otros métodos como la búsqueda lineal.

2.3 Algoritmos de Ordenamiento

El ordenamiento es el proceso de reorganizar una secuencia de elementos para establecer un orden lógico, como ascendente o descendente. Es un paso fundamental para optimizar el procesamiento y la visualización de datos. En sistemas computacionales, los algoritmos de ordenamiento se utilizan ampliamente para mejorar la eficiencia de operaciones posteriores, como búsquedas y agrupamientos. Se compararán:

Ordenamiento de Burbuja (Bubble Sort)

Bubble Sort es un algoritmo de ordenamiento simple que se basa en comparar elementos adyacentes y permutarlos si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista queda completamente ordenada. Su nombre se debe a la forma en que los elementos "suben" en la lista durante los intercambios, como burbujas en el agua.

Características

- Es un **algoritmo de comparación** basado en intercambios directos.
- Su **complejidad temporal** es **$O(n^2)$** en el peor y promedio de los casos, lo que lo hace poco eficiente para grandes conjuntos de datos.
- Es útil en casos donde los datos están casi ordenados, ya que en su mejor caso puede tener una complejidad **O(n)**.

Desventajas

- Su rendimiento es inferior al de otros algoritmos como **quicksort** o **ordenamiento por inserción**.
- Requiere múltiples pasadas por el arreglo, lo que lo vuelve ineficiente en comparación con alternativas más optimizadas.

Ordenamiento Rápido (Quicksort)

Quicksort es un algoritmo eficiente de ordenamiento que funciona dividiendo repetidamente una lista en dos sublistas basadas en un **elemento pivote**. Los elementos menores que el pivote quedan a su izquierda, y los mayores a su derecha. Luego, el proceso se repite recursivamente hasta que toda la lista está ordenada.

Características

- Se basa en el concepto de **dividir y conquistar**.
- En el mejor y promedio de los casos, su **complejidad temporal** es **O(n log n)**, lo que lo hace eficiente para grandes conjuntos de datos.
- En el peor caso (cuando el pivote elegido no divide equilibradamente los elementos), la complejidad puede ser **$O(n^2)$**.

Ventajas

- Es mucho más rápido que el ordenamiento de burbuja en la mayoría de los escenarios.

- Es ampliamente utilizado en sistemas informáticos debido a su eficiencia.
- Puede ser optimizado mediante diferentes estrategias de selección de pivote para evitar el peor caso.

2.4 Notación Big O

La notación Big O es un estándar para expresar la tasa de crecimiento del tiempo de ejecución de un algoritmo a medida que aumenta el tamaño de la entrada. Por ejemplo, un algoritmo con complejidad O(n) implica que el tiempo de ejecución crece linealmente con n, mientras que uno con complejidad O(log n) crece de forma logarítmica, siendo más eficiente para grandes valores de n

3. Desarrollo Práctico

El procesamiento eficiente de datos es una necesidad fundamental en informática. Para lograrlo, los algoritmos de **búsqueda** y **ordenamiento** juegan un papel crucial en la manipulación de grandes volúmenes de información. En este ejemplo práctico, se implementaron y compararon diferentes algoritmos con el objetivo de analizar su rendimiento en términos de eficiencia temporal.

Código en Python:

```
import time import
random

# Algoritmos de Búsqueda

# Búsqueda lineal def busqueda_lineal(lista,
objetivo):
for i in range(len(lista)):        if
lista[i] == objetivo:
return i     return -1

# Búsqueda binaria  def
busqueda_binaria(lista, objetivo):
izquierda, derecha = 0, len(lista) - 1
```

```
while izquierda <= derecha:
medio = (izquierda + derecha) // 2
if lista[medio] == objetivo:
        return medio          elif
lista[medio] < objetivo:
izquierda = medio + 1         else:
derecha = medio - 1     return -1

# Algoritmos de Ordenamiento

# Bubble Sort def
bubble_sort(lista):     n =
len(lista)     for i in range(n):
for j in range(0, n - i - 1):
if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]     return
lista

# Quick Sort def quick_sort(lista):     if len(lista) <= 1:
return lista     pivote = lista[len(lista) // 2]     izquierda = [x for x
in lista if x < pivote]     centro = [x for x in lista if x == pivote]
derecha = [x for x in lista if x > pivote]     return
quick_sort(izquierda) + centro + quick_sort(derecha)

# Función auxiliar para medir tiempos

def medir_tiempo(funcion,
*args):     inicio =
time.perf_counter()     resultado =
funcion(*args)     fin =
time.perf_counter()     return
resultado, fin - inicio
# Programa principal de prueba
```

```
if __name__ == "__main__":
 # Crear una lista grande con 10,000 números aleatorios     tamano = 10_000
lista_original = [random.randint(1, 100_000) for _ in range(tamano)]     objetivo =
lista_original[tamano // 2]  # Usamos un número que sabemos que está

    print("\nORDENAMIENTO")

    # Bubble Sort      lista_copia_bubble = lista_original.copy()
    _, tiempo_bubble = medir_tiempo(bubble_sort, lista_copia_bubble)
print(f"Lista ordenada con Bubble Sort (primeros 20 elementos):
{lista_copia_bubble[:20]}")     print(f"Bubble Sort:
{tiempo_bubble:.4f} segundos")



    # Quick Sort      lista_copia_quick = lista_original.copy()      lista_ordenada_quick,
tiempo_quick = medir_tiempo(quick_sort, lista_copia_quick)     print(f"Lista ordenada
con Quick Sort (primeros 20 elementos):
{lista_ordenada_quick[:20]}")     print(f"Quick Sort:
{tiempo_quick:.4f} segundos")



    print("\nBÚSQUEDA")

    # Búsqueda Lineal     indice_lineal, tiempo_lineal = medir_tiempo(busqueda_lineal,
lista_original, objetivo)     print(f"Búsqueda Lineal: índice {indice_lineal}, tiempo
{tiempo_lineal:.6f} segundos")

    # Búsqueda Binaria      indice_binaria, tiempo_binaria =
medir_tiempo(busqueda_binaria,     lista_ordenada_quick,
objetivo)
  print(f"Búsqueda Binaria: índice {indice_binaria}, tiempo {tiempo_binaria:.6f} segundos")

    print("\nCONCLUSIÓN")     print("• Quick Sort fue más eficiente que Bubble
Sort.")     print("• Búsqueda Binaria fue más rápida que la lineal, pero requiere
una lista
```

ordenada.")

Resultado esperado (salida aproximada):

ORDENAMIENTO

Lista ordenada con Bubble Sort (primeros 20 elementos): [2, 2, 4, 24, 36, 42, 52, 53, 54, 60, 61, 76, 98, 99, 103, 105, 130, 143, 152, 153]

Bubble Sort: 3.6596 segundos

Lista ordenada con Quick Sort (primeros 20 elementos): [2, 2, 4, 24, 36, 42, 52, 53, 54, 60, 61, 76, 98, 99, 103, 105, 130, 143, 152, 153]

Quick Sort: 0.0130 segundos

BÚSQUEDA

Búsqueda Lineal: índice 5000, tiempo 0.000145 segundos

Búsqueda Binaria: índice 649, tiempo 0.000006 segundos

CONCLUSIÓN

- Quick Sort fue más eficiente que Bubble Sort.
- Búsqueda Binaria fue más rápida que la lineal, pero requiere una lista ordenada.

3.2 Comparación de Tiempos de Ejecución
Para evaluar el rendimiento de los algoritmos implementados, se realizaron pruebas de tiempo sobre una lista de 10.000 números aleatorios. Se compararon dos métodos de búsqueda: búsqueda lineal y búsqueda binaria, así como dos algoritmos de ordenamiento: Bubble Sort y Quick Sort.

Los resultados obtenidos reflejan lo siguiente:

- La búsqueda binaria fue más rápida que la búsqueda lineal, lo cual se debe a que reduce el espacio de búsqueda dividiéndolo sucesivamente.

- El Quick Sort demostró ser más eficiente que Bubble Sort, ya que su enfoque de dividir y conquistar permite ordenar los elementos en menos pasos y con menor tiempo de ejecución.

4. Resultados Obtenidos

- El programa logró ordenar correctamente una lista de 10.000 números generados aleatoriamente.
- La búsqueda binaria identificó con rapidez el número objetivo, siempre que la lista estuviera previamente ordenada.
- Se observaron diferencias claras en la eficiencia entre los algoritmos: Quick Sort y búsqueda binaria fueron más rápidos que Bubble Sort y búsqueda lineal, respectivamente.
- Se comprendió la relevancia de contar con una lista ordenada para aplicar búsqueda binaria de forma efectiva.

5. Conclusiones

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación y el manejo eficiente de datos. A través de este trabajo se evidenció que la elección del algoritmo adecuado puede impactar significativamente en el rendimiento de una aplicación. La búsqueda binaria resultó ser notablemente más eficiente que la búsqueda lineal, aunque requiere listas ordenadas. Asimismo, Quick Sort demostró su ventaja sobre Bubble Sort gracias a su estrategia de divide y vencerás. Este ejercicio permitió afianzar conocimientos sobre eficiencia algorítmica y su importancia práctica en la resolución de problemas.

Metodología Utilizada

La elaboración del trabajo se llevó a cabo en las siguientes etapas:

- **Recolección de información teórica** a partir de documentación confiable sobre algoritmos de búsqueda y ordenamiento.
- **Implementación en Python** de los algoritmos estudiados, incluyendo **búsqueda lineal**, **búsqueda binaria**, **Bubble Sort** y **Quick Sort**.
- **Generación de conjuntos de datos aleatorios** utilizando el módulo random para simular escenarios de prueba representativos.
- **Medición del tiempo de ejecución** de cada algoritmo con el módulo time, registrando los resultados para su análisis.

- **Comparación y validación de los resultados**, evaluando la eficiencia y aplicabilidad de cada método.
- **Elaboración del informe final**, organizando los hallazgos y preparando anexos con el código fuente y resultados obtenidos.

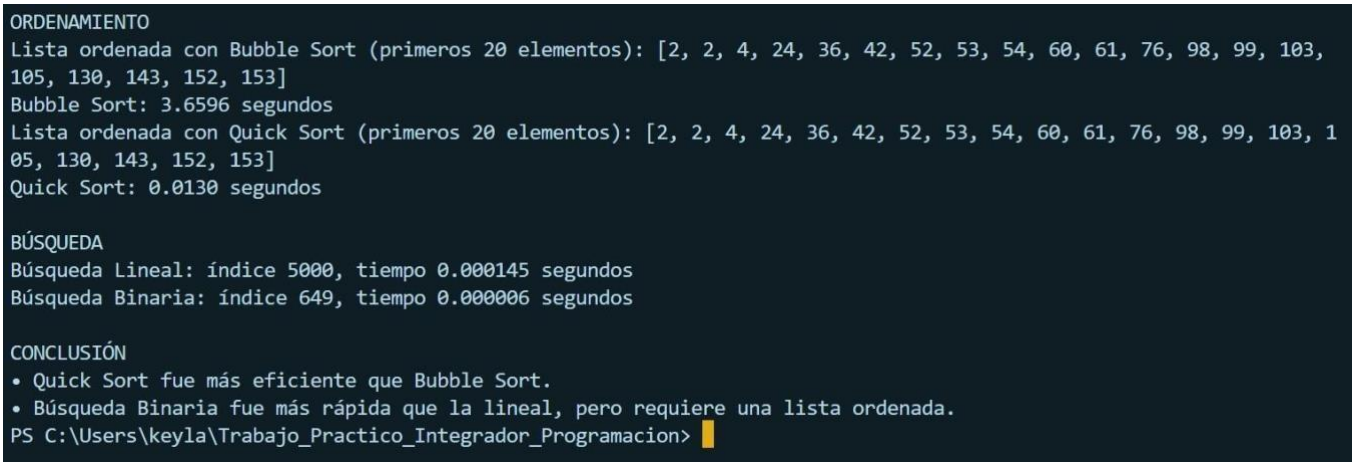
Bibliografia:

Black, P. E. (2022). *Búsqueda binaria*. En *Dictionary of Algorithms and Data Structures* [en línea]. National Institute of Standards and Technology. Recuperado el 6 de junio de 2025, de https://www.nist.gov/dads/HTML/binarySearch.html

García Flores, A. (s.f.). *Algoritmos*. Universidad Nacional Autónoma de México (UNAM). Recuperado el 6 de junio de 2025 de: https://www.paginaspersonales.unam.mx/app/webroot/files/6345/Garcia_Flores_Arturo_-_Algoritmos.pdf

Anexos

Captura de pantalla del programa funcionando:



Repositorio en GitHub:

https://github.com/KEY1A/Trabajo_Practico_Integrador_Programacion

Video explicativo: https://youtu.be/gEXvBJ4uvp4?si=uoOXaXEH_WtQwBS6