Trabajo Práctico Integrador

Programación I

Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos

Maximiliano Rao – rao.maximiliano.l@gmail.com

Mauricio López – rinaldi.el@hotmail.com

Profesor: Prof. Nicolás Quirós

Profesor: Prof. Sebastián Bruselario

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Junio 2025

índice

1- Introducción 3
2- Marco Teórico 4
3- Caso Práctico 9
4- Metodología Utilizada 11
5- Resultados Obtenidos 12
6- Conclusiones 13
7- Bibliografía 14
8- Anexos 14

1- Introducción

El presente trabajo se centra en el estudio de los algoritmos de búsqueda y ordenamiento, herramientas fundamentales dentro del área de la programación. Se eligió este tema debido a su importancia en la resolución de problemas cotidianos en el desarrollo de software, como la gestión eficiente de grandes volúmenes de datos, la optimización del rendimiento de aplicaciones y la mejora en la experiencia del usuario.

En programación, la búsqueda permite localizar información específica dentro de un conjunto de datos, mientras que el ordenamiento organiza dichos datos para facilitar su acceso y análisis. Estos procesos son esenciales en una amplia variedad de contextos, desde motores de búsqueda, bases de datos y sistemas de archivos, hasta algoritmos de inteligencia artificial y visualización de datos.

El objetivo de este trabajo es comprender el funcionamiento de distintos algoritmos de búsqueda y ordenamiento, analizar sus ventajas y desventajas en términos de eficiencia, uso de memoria y escalabilidad, y determinar en qué situaciones resulta más conveniente utilizar cada uno. Además, se busca aplicar estos conocimientos en la implementación de un programa práctico en Python que combine ambos tipos de algoritmos.

A través de este análisis, se pretende destacar la relevancia de seleccionar adecuadamente los métodos de búsqueda y ordenamiento en función del problema a resolver, contribuyendo así al diseño de soluciones más efectivas y robustas en el ámbito del desarrollo de software.

2- Marco Teórico

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en ciencias de la computación y programación. Su correcta aplicación permite mejorar el rendimiento de los programas, optimizar recursos y garantizar la escalabilidad de las soluciones desarrolladas.

1. Concepto de Búsqueda

La búsqueda es el proceso mediante el cual se localiza un elemento dentro de un conjunto de datos. En programación, este conjunto suele estar representado por listas, arreglos u otras estructuras.

- **Búsqueda lineal**

  La búsqueda lineal, también llamada secuencial, recorre uno a uno los elementos de la lista hasta encontrar el valor buscado o llegar al final sin hallarlo.

```
1 def busqueda_lineal(lista, objetivo):
2     for i in range(len(lista)):
3         if lista[i] == objetivo:
4             return i
5     return -1
```

Complejidad:

- **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.

- **Búsqueda binaria**

  La búsqueda binaria requiere que los datos estén ordenados. Divide la lista a la mitad en cada iteración, descartando la mitad que no contiene el objetivo.

```
1 def busqueda_binaria(lista, objetivo):
2     izquierda, derecha = 0, len(lista) - 1
3     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
```

Complejidad:

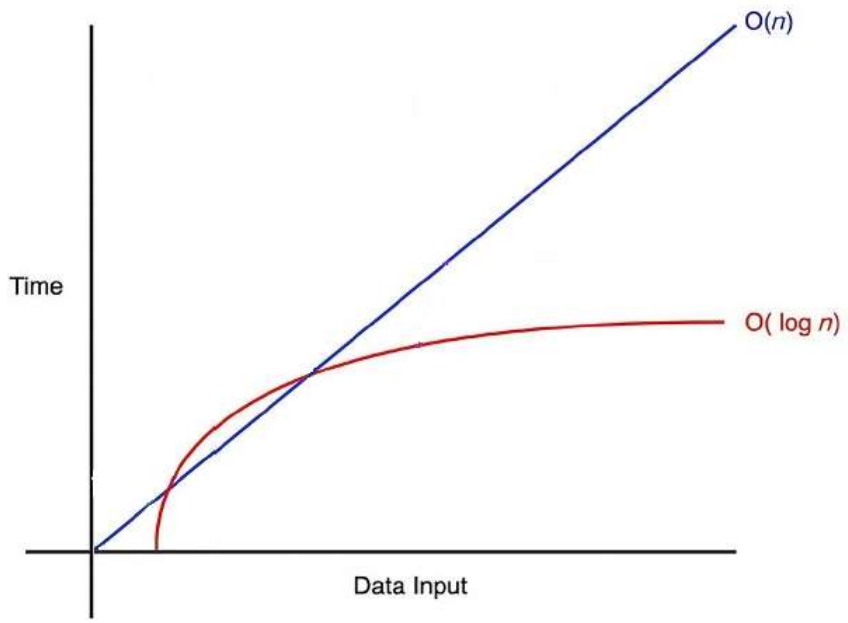
- **Peor caso: O(log n),** donde n es el número de elementos en la lista. Esto ocurre cuando el elemento no está presente o está en una de las divisiones finales.
- **Mejor caso: O(1),** cuando el elemento buscado está justo en el centro de la lista.
- **Caso promedio: O(log n),** porque el algoritmo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

Comparación entre algoritmos de búsquedas:

Algoritmo	Complejidad	Ventajas	Desventajas	Uso recomendado
Búsqueda lineal	O(n)	-No requiere lista ordenada. -Implementación simple.	-Poco eficiente con grandes volúmenes de datos.	Listas pequeñas o desordenadas
Búsqueda binaria	O(log n)	-Eficiente para grandes listas ordenadas.	-Requiere datos previamente ordenados -Más compleja de implementar	Listas grandes y ordenadas

Los algoritmos de búsqueda lineal tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado.

Los algoritmos de búsqueda binaria tienen un tiempo de ejecución de O(log n), lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.



2. Concepto de Ordenamiento

El ordenamiento organiza datos según un criterio (por ejemplo, de menor a mayor). Es crucial para acelerar búsquedas, detectar duplicados o visualizar información.

- **Bubble Sort (Ordenamiento por burbuja):**

  Algoritmo simple que compara elementos adyacentes y los intercambia si están en orden incorrecto.

```
1 def bubble_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         for j in range(0, n - i - 1):
5             if arr[j] > arr[j + 1]:
6                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Complejidad:

- **Peor caso: $O(n^2)$,** 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).
- **Caso promedio: O(n^2).**

- **Quick Sort (Ordenamiento rápido):**

  Divide la lista en dos sublistas, menores y mayores al pivote, y ordena recursivamente.

```
1 def quicksort(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         pivot = arr[0]
6         less = [x for x in arr[1:] if x <= pivot]
7         greater = [x for x in arr[1:] if x > pivot]
8         return quicksort(less) + [pivot] + quicksort(greater)
```

Complejidad:

- **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.
- **Caso promedio: O(n log n).**

- **Selection Sort (Ordenamiento por selección):**

  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.

```
1  def selection_sort(arr):
2      n = len(arr)
3      for i in range(n):
```

```
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Intercambiar el elemento mínimo con el elemento actual
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Complejidad:

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

- **Insertion Sort (Ordenamiento por inserción):**

  Construye una lista ordenada insertando cada elemento en su posición correcta.

```
def insertion_sort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    j = i-1
    while j >=0 and key < arr[j] :
        arr[j+1] = arr[j]
        j -= 1
    arr[j+1] = key
```

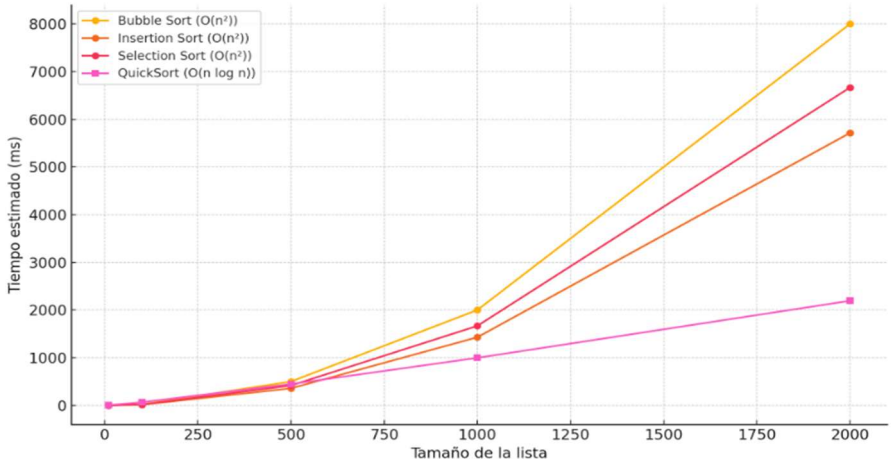
Complejidad:

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

Comparación entre algoritmos de ordenamiento

Algoritmo	Caso Promedio	Eficiencia	Uso recomendado
Burbuja	O(n²)	Baja	Listas pequeñas o casi ordenadas
Selección	O(n²)	Baja	Listas pequeñas
Inserción	O(n²)	Media	Listas pequeñas, casi ordenadas
QuickSort	O(n log n)	Alta	Listas grandes, alto rendimiento

Gráfico comparativo de tiempo estimado de ejecución para diferentes algoritmos de ordenamiento según el tamaño de la lista:



3- Caso Práctico

Descripción del problema

Se simula una situación donde una tienda online posee un inventario con productos. El objetivo es implementar un sistema simple que permita:

- Buscar un producto por su nombre.
- Ordenar los productos por su precio para facilitar su visualización.

A continuación, se comparten fragmentos representativos del código. El código completo puede consultarse en el repositorio compartido al final del documento.

Lista de productos utilizada:

```
# Lista simulada de productos en forma de diccionario
productos = [
    {"nombre": "Auriculares", "precio": 100000},
    {"nombre": "Teclado", "precio": 25000},
    {"nombre": "Monitor", "precio": 200000},
    {"nombre": "Mouse", "precio": 10000},
    {"nombre": "Notebook", "precio": 700000}
]
```

Implementación de la búsqueda lineal:

```
# Búsqueda lineal: se recorre la lista comparando cada nombre de producto
def buscar_producto(nombre_buscado, lista):
    for producto in lista:
        # Convierte ambos nombres a minúsculas para comparar sin importar
mayúsculas/minúsculas
        # Si hay coincidencia, se retorna el producto
        if producto["nombre"].lower() == nombre_buscado.lower():
            return producto
    # Si no se encuentra el producto, se retorna None
    return None
```

Implementación del algoritmo de ordenamiento QuickSort:

```
# Ordenamiento QuickSort por precio de menor a mayor
def quicksort(lista):
    # Caso base: si la lista tiene un solo elemento o está vacía, ya está ordenada
    if len(lista) <= 1:
        return lista
    else:
        # Toma el primer elemento como pivote
        pivote = lista[0]
        # Lista con los elementos cuyo precio es menor o igual al del pivote
        menores = [x for x in lista[1:] if x["precio"] <= pivote["precio"]]
        # Lista con los elementos cuyo precio es mayor al del pivote
        mayores = [x for x in lista[1:] if x["precio"] > pivote["precio"]]
        # Aplicar QuickSort de forma recursiva y combinar los resultados
        return quicksort(menores) + [pivote] + quicksort(mayores)
```

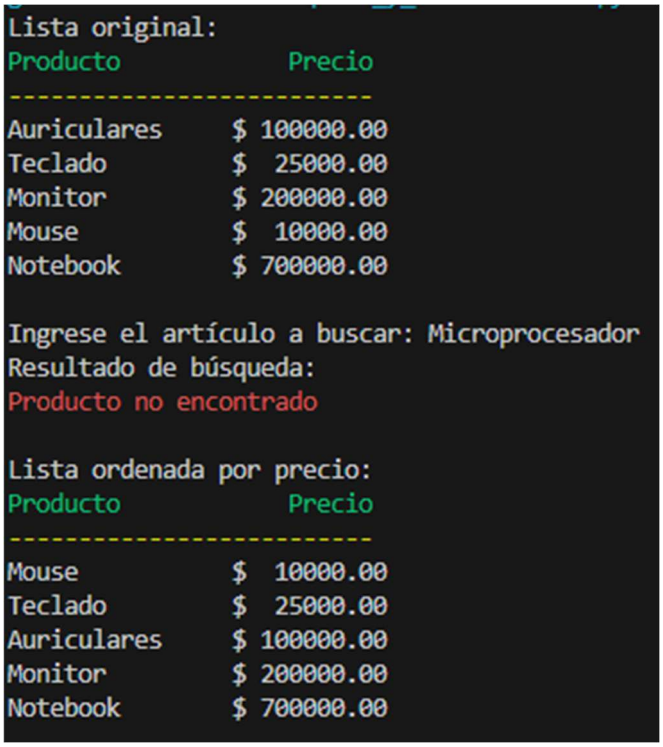
Ejemplo de ejecución cuando el producto es encontrado en el listado:

```
Lista original:
Producto            Precio
-------------------------
Auriculares     $ 100000.00
Teclado         $  25000.00
Monitor         $ 200000.00
Mouse           $  10000.00
Notebook        $ 700000.00

Ingrese el artículo a buscar: Mouse
Resultado de búsqueda:
Producto encontrado:
Producto            Precio
-------------------------
Mouse           $  10000.00


Lista ordenada por precio:
Producto            Precio
-------------------------
Mouse           $  10000.00
Teclado         $  25000.00
Auriculares     $ 100000.00
Monitor         $ 200000.00
Notebook        $ 700000.00
```

Ejemplo de ejecución cuando el producto no es encontrado en el listado:



Se eligió el algoritmo de **búsqueda lineal** debido a que la lista de productos es pequeña, y este método no requiere que los datos estén previamente ordenados.

Para el ordenamiento, se implementó el algoritmo QuickSort ya que es un algoritmo eficiente en la mayoría de los casos y tiene mejor rendimiento promedio que Bubble Sort o Insertion Sort.

4- Metodología Utilizada

1-Se realizó una investigación teórica sobre los principales algoritmos de búsqueda y ordenamiento en programación, con énfasis en sus ventajas, desventajas y casos de uso. Las fuentes consultadas incluyeron:

- Apuntes de clase.
- Documentación oficial de Python (docs.python.org).

2-Se definió un problema concreto: buscar y ordenar productos simulados en una lista de diccionarios.
El diseño del código en Python se dividió en funciones específicas para:

- Realizar una búsqueda lineal (por nombre del producto).
- Ordenar los productos usando el algoritmo **QuickSort**, adaptado para trabajar con listas de diccionarios.

También se definió una función adicional para mostrar los resultados de manera clara en consola.

3- Se ejecutaron distintas pruebas con:

- Productos existentes y no existentes en la lista (para validar la búsqueda).
- Productos con precios variados (para verificar el ordenamiento).

5- Resultados Obtenidos

El desarrollo del caso práctico permitió aplicar de manera efectiva los conceptos de búsqueda lineal y ordenamiento QuickSort en una lista de productos simulada.

Funcionalidades logradas

- Búsqueda exitosa de productos por nombre (sin importar mayúsculas/minúsculas).
- Ordenamiento correcto de productos por precio, de menor a mayor, utilizando el algoritmo QuickSort.
- Visualización mejorada en consola gracias al uso de la librería colorama.
- Medición de tiempo de ejecución de ambas operaciones con el uso de la librería time, aunque por el tamaño reducido de la lista los tiempos fueron muy bajos (cercanos a 0 segundos).

Se probaron distintas situaciones para validar el funcionamiento del código:

- Ingreso de productos existentes (ej.: "Mouse", "Teclado") → Producto encontrado correctamente.

- Ingreso de productos no existentes (ej.: "Tablet") → Mensaje correcto: "Producto no encontrado".
- Prueba de ordenamiento con productos ya ordenados o con precios iguales → Orden correcto mantenido.

Durante el proceso se detectaron y resolvieron algunos problemas menores:

- Comparación sensible a mayúsculas en la búsqueda → Se solucionó convirtiendo ambos valores a minúsculas con .lower().
- Tiempos de ejecución siempre en cero → Se entendió que el problema era la baja complejidad del conjunto de datos. Para pruebas más precisas sería necesario trabajar con listas mucho más grandes.

El código completo, documentado y funcional se encuentra disponible en el siguiente repositorio de GitHub: https://github.com/MaximilianoRao/Trabajo-Integrador---Programaci-n-I---UNC

6- Conclusiones

A lo largo del desarrollo del trabajo, el grupo pudo afianzar conceptos fundamentales de programación relacionados con algoritmos de búsqueda y ordenamiento, se logró una implementación funcional de búsqueda lineal y ordenamiento QuickSort sobre una lista de productos simulada. Esto permitió ver en acción cómo se comportan los algoritmos frente a diferentes entradas y cómo el rendimiento puede variar según la estructura de datos utilizada y el volumen de información.

Entre las dificultades encontradas, se destaca la interpretación de los tiempos de ejecución, que resultaron muy bajos debido a la poca cantidad de datos. Esta situación permitió reflexionar sobre la importancia de realizar pruebas con diferentes tamaños de datos para obtener resultados más realistas. También fue necesario ajustar detalles como la sensibilidad de las búsquedas a mayúsculas/minúsculas, lo cual se resolvió con funciones como .lower().

Como mejora futura, se propone:

- Implementar búsqueda binaria para listas ordenadas.
- Comparar tiempos de ejecución con volúmenes de datos mayores.
- Aplicar otros algoritmos de ordenamiento como InsertionSort para analizar diferencias de rendimiento.

7- Bibliografía

- Python Software Foundation. (2024). *Built-in Types — Python 3.12.3 documentation*. https://docs.python.org/es/3.13/library/stdtypes.html
- Colorama documentation. (2023). *Python Package Index (PyPI)*. https://pypi.org/project/colorama/
- Aula Virtual Institucional. (2025). *Apuntes de clase: Algoritmos de búsqueda y ordenamiento*. Material proporcionado por el docente.

8- Anexos

Link repositorio en GitHub:

https://github.com/MaximilianoRao/Trabajo-Integrador---Programaci-n-I---UNC

Link video:

https://www.youtube.com/watch?v=GRhRCQ_Z9qA