Trabajo Práctico – Búsqueda y Ordenamiento

Programación I

Profesora
Julieta Trapé

Fecha de Entrega: 09 de junio de 2025

Alumnos

Florencia Paolazzi florpaolazzi@gmail.com

Rosa Lourdes, Paco Laura lourdes.rosapaco@gmail.com

Índice

1. Introducción

En el ámbito de la programación, los algoritmos de búsqueda y ordenamiento son fundamentales para el manejo eficiente de datos, permitiendo organizar y recuperar información de manera rápida y estructurada. Estos métodos no solo optimizan el rendimiento de aplicaciones, sino que también son esenciales en diversos campos, como bases de datos, inteligencia artificial y sistemas de archivos.

El tema fue elegido por su relevancia tanto teórica como práctica, y se desarrolla a partir de un caso inspirado en el arbolado urbano. A través de este ejemplo concreto, se busca demostrar que la elección del tipo de búsqueda **no es universal**, sino que depende directamente del tipo de dato que se desea consultar y de la estructura en la que se encuentra almacenado.

En este trabajo práctico nos centramos en analizar los algoritmos más utilizados para la búsqueda y ordenamiento, destacando sus características, ventajas y limitaciones.

2. Marco Teórico

2.1 Algoritmos de Búsqueda

Un algoritmo de búsqueda tiene como objetivo encontrar un elemento específico dentro de un conjunto de datos, como una lista o un diccionario. Existen distintos tipos de búsqueda, cuya eficiencia depende de la estructura del dato y del tipo de consulta.

a) Búsqueda Lineal

Consiste en recorrer secuencialmente todos los elementos hasta encontrar el valor deseado. Tiene una **complejidad temporal de O(n)** y no requiere orden previo. Es simple pero poco eficiente para listas grandes.

b) Búsqueda Binaria

Requiere que los datos estén **previamente ordenados** según el criterio de búsqueda (por ejemplo, una dirección). En cada paso divide la lista a la mitad, reduciendo significativamente la cantidad de comparaciones. Tiene una **complejidad de O(log n)**. No es aplicable a atributos con múltiples valores por registro o sin orden natural, como una lista de tareas que se repiten.

c) Búsqueda mediante Índice Hash

Utiliza una estructura de tipo diccionario o hash table que permite acceder directamente a un elemento mediante una clave. Tiene una **complejidad promedio de O(1)**, siendo extremadamente rápida. Es ideal para búsquedas por valores únicos como direcciones, pero requiere una etapa previa de construcción del índice. En el caso de valores no únicos (como tareas), se debe indexar cada clave con una lista de elementos asociados.

2.2 Algoritmos de Ordenamiento

El **ordenamiento de datos** es un paso previo necesario para aplicar ciertos algoritmos de búsqueda, como la binaria. Además, puede mejorar la legibilidad, agrupamiento o accesibilidad de los datos.

Algunos algoritmos de ordenamiento relevantes son:

a) Ordenamiento por Burbuja (Bubble Sort)

Es simple de implementar, pero ineficiente para grandes volúmenes de datos, con una complejidad de **$O(n^2)$**. Se utiliza principalmente con fines didácticos.

b) Ordenamiento por Inserción o Selección

Tienen también complejidad **$O(n^2)$** en el peor caso. Son útiles solo para listas pequeñas o casi ordenadas.

c) Ordenamientos eficientes (Merge Sort, Quick Sort, TimSort)

- **Merge Sort** y **Quick Sort** tienen una complejidad promedio de **O(n log n)**.
- **TimSort**, utilizado por defecto en Python (`sorted()`), combina los beneficios de varios algoritmos y está optimizado para casos reales.

El ordenamiento se vuelve relevante cuando se necesita aplicar búsqueda binaria o cuando se desea facilitar la navegación manual o visual sobre los datos.

2.3 Estructura de Datos y Tipo de Consulta

En el caso del arbolado urbano, la estructura de los datos influye directamente en la elección del algoritmo :

- La **dirección** es un atributo **único por árbol**, ideal para búsquedas con hash o binaria (si está ordenado).
- Las **tareas** pueden repetirse en distintos árboles, por lo que requieren estructuras que permitan asociar **una misma clave a múltiples valores** (como un diccionario de listas).

2.4 Complejidad Computacional

La **complejidad temporal** mide cuántas operaciones realiza un algoritmo en función del tamaño de la entrada:

- **Búsqueda lineal**: O(n)
- **Búsqueda binaria**: O(log n)
- **Búsqueda hash**: O(1) promedio
- **Ordenamiento**: O(n log n) en métodos eficientes

También debe considerarse la **complejidad espacial**, especialmente en los algoritmos que requieren estructuras auxiliares (como índices o diccionarios adicionales).

3. Caso Práctico

Como caso práctico, se propone el análisis y la optimización de algoritmos de búsqueda sobre una base de datos simulada de arbolado urbano. En este contexto, cada árbol posee una **dirección única** y puede tener asignadas **múltiples tareas, como poda, extracción o reparación de vereda.** Considerando la estructura de los datos y los distintos criterios de consulta, el objetivo es identificar qué algoritmos de búsqueda y ordenamiento resultan más eficientes según el tipo de búsqueda a realizar.

3.1 Tipo de búsquedas abordadas

- **Búsqueda por dirección** (Ubicación del árbol buscado _ **valor único**).
  - Búsqueda lineal
  - Búsqueda binaria
  - Incide Hush

- **Búsqueda por tareas** (Poda, extracción, corte de raíz,etc _**no hay un solo valor**).
  - Búsqueda lineal
  - Búsqueda binaria
  - Incide Hush

3.1.1 Busqueda por direccion

Búsqueda lineal

```
def buscar_por_direccion_lineal(arboles, direccion_objetivo):
    for i in range(len(arboles)):
        if arboles[i]['direccion'] == direccion_objetivo:
            return i
    return -1
```

Búsqueda binaria (requiere ordenamiento previo)

```
# Función para obtener la dirección del árbol
def obtener_direccion(arbol):
    return arbol["direccion"]

# Ordenar la lista de árboles por dirección (requisito para búsqueda
binaria)
arboles_ordenados = sorted(arboles, key=obtener_direccion)

# Función de búsqueda binaria adaptada para buscar por dirección
def busqueda_binaria_por_direccion(arboles_ordenados, direccion_objetivo):
    izquierda, derecha = 0, len(arboles_ordenados) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        direccion_actual = arboles_ordenados[medio]["direccion"]
        if direccion_actual == direccion_objetivo:
            return medio
        elif direccion_actual < direccion_objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Búsqueda con Índice Hash

```
indice_direccion = {arbol["direccion"]: arbol for arbol in arboles}
arbol = indice_direccion.get(direccion_objetivo)
```

3.1.2 Búsqueda por tarea

Búsqueda lineal

```
# Buscar todos los árboles que tengan una tarea específica

tarea_objetivo ="poda" #si la tarea buscada fuera poda


def busqueda_lineal_por_tarea(arboles, tarea_objetivo):
    resultado = []
    for arbol in arboles:
        if tarea_objetivo in arbol["tareas"]:
            resultado.append(arbol)
    return resultado
```

Búsqueda binaria

```
tareas_lista = []
for arbol in arboles:
    for tarea in arbol["tareas"]:
        tareas_lista.append({"tarea": tarea, "arbol": arbol})

# Ordenar por tarea
def obtener_tarea(item):
    return item["tarea"]

tareas_ordenadas = sorted(tareas_lista, key=obtener_tarea)
def busqueda_binaria_por_tarea(lista, tarea_objetivo):
    resultados = []
    izquierda, derecha = 0, len(lista) - 1

    # Encontrar una ocurrencia con búsqueda binaria clásica
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
```

```
        tarea_actual = lista[medio]["tarea"]
        if tarea_actual == tarea_objetivo:
            # Buscar hacia ambos lados para encontrar todos los árboles con
esa tarea
            i = medio
            while i >= 0 and lista[i]["tarea"] == tarea_objetivo:
                resultados.append(lista[i]["arbol"])
                i -= 1
            i = medio + 1
            while i < len(lista) and lista[i]["tarea"] == tarea_objetivo:
                resultados.append(lista[i]["arbol"])
                i += 1
            break
        elif tarea_actual < tarea_objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    return resultados
```

Búsqueda con Índice Hash

```
indice_tareas = {}
for arbol in arboles:
    for tarea in arbol["tareas"]:
        if tarea not in indice_tareas:
            indice_tareas[tarea] = []
        indice_tareas[tarea].append(arbol)
```

4. Metodología Utilizada

Este trabajo se centró en el análisis, implementación y comparación de distintos **algoritmos de búsqueda y ordenamiento** aplicados a un conjunto de datos representativo del **arbolado urbano**. Para ello se adoptó una estrategia práctica dividida en varias etapas que permitieron evaluar la eficiencia de los métodos aplicados según el tipo de dato y el objetivo de la búsqueda.

4.1 Definición del conjunto de datos

Se diseñó una estructura de datos simulada en formato de lista de diccionarios, donde cada diccionario representa un árbol urbano con los siguientes atributos:

- *id*: identificador numérico único del árbol.
- *direccion:* cadena de texto que representa la ubicación exacta del árbol (**única**).

- *tareas:* lista de tareas asociadas al árbol, que puede incluir poda, extracción, corte de raíz o reparación de vereda **(no necesariamente únicas ni exclusivas**).

4.2 Implementación de algoritmos de búsqueda

Se implementaron tres métodos diferentes de búsqueda para comparar su comportamiento en relación al tipo de dato buscado:

- **Búsqueda lineal**: recorre uno a uno los elementos de la lista para encontrar coincidencias.

- **Búsqueda binaria**: requiere una lista previamente ordenada por dirección; permite localizar rápidamente un árbol en base a este atributo único.

- **Búsqueda mediante índice hash**: utiliza un diccionario para mapear claves (direcciones) a sus respectivos valores. En el caso de las tareas, el índice apunta a listas de árboles que comparten una misma tarea. Por eso utilizaremos **Búsqueda con Índice Invertido**, cuya estructura es un diccionario donde cada **clave** (una tarea) apunta a un **conjunto de registros** que contienen esa clave.

Se diferenciaron claramente dos escenarios:

- **Búsqueda por dirección**: dado que cada dirección es única, se usaron los tres métodos (lineal, binaria e índice hash) para comparar rendimiento y precisión.

- **Búsqueda por tarea**: un árbol puede tener múltiples tareas y una misma tarea puede estar asignada a múltiples árboles, por esta razón, la búsqueda binaria no se considera recomendable y se priorizaron la búsqueda lineal y el índice hash.

4.3 Implementación de ordenamientos

Para que la búsqueda binaria fuera posible, se ordenaron previamente los árboles por dirección utilizando la función *sorted()* de Python.

Se decidió **no implementar manualmente algoritmos clásicos de ordenamiento** como burbuja, inserción, selección o quicksort, ya que *sorted()* los supera en rendimiento y simplicidad, especialmente para los fines del presente trabajo.

4.4 Medición de rendimiento

Con el módulo *time* de Python se midió el tiempo de ejecución de cada búsqueda (lineal, binaria e índice hash) en milisegundos, permitiendo comparar objetivamente su

eficiencia. Se utilizó *random* en las etapas de prueba, para generar listas aleatorias y distintos valores de entrada tanto para direcciones como para tareas.

5. Resultados Obtenidos

5.1 Búsqueda por Dirección (valor único):

Tipo de búsqueda	Complejidad Temporal	Tiempo de ejecución (segundos)	Observaciones
Búsqueda Lineal	O(n)	0.034752 s	Más lenta que la binaria, más rapida que por busqueda hash.
Búsqueda Binaria	O(log n)	0.000047 s	Fue la más rápida. Requiere ordenamiento previo para funcionar.
Búsqueda con Índice Hash	O(1) (promedio)	0.331869 s	Teóricamente rápida, pero fue mucho más lenta que binaria y lineal. Incluye tiempo de construccion del indice.

```
Búsqueda Lineal: Resultado = 521189, Tiempo = 0.034752 segundos
Búsqueda Binaria: Resultado = 467991, Tiempo = 0.000047 segundos
Búsqueda Hash: Árbol encontrado = {'direccion': 'Calle 521189'}, Tiempo = 0.331869 segundos
```

5.2 Búsqueda por tareas (valor no único)

En este caso, un árbol puede tener más de una tarea, y una misma tarea puede estar asignada a múltiples árboles, por ejemplo un árbol puede tener asignado poda y reparación de vereda o son más de uno los árboles destinados a extracción, osea una misma tarea. Esto cambia el enfoque de búsqueda:

Tipo de búsqueda	Complejidad temporal	Tiempo de ejecución (segundos)	Observaciones
Búsqueda lineal	O(n)	0.000005	Fue rápida pero no es el mejor tiempo
Búsqueda binaria	No recomendable	—	No aplicable: no se puede ordenar por tarea única
Búsqueda con índice hash (invertida)	O(1) acceso, O(n) para construir	0.000001	Se creó una tabla hash donde cada clave es una tarea, y el valor asociado es una lista de árboles que tienen esa tarea. Muy rapido, con complejidad cercana a O(1) para acceder a la lista

```
Búsqueda Lineal por tarea: 2 árboles encontrados. Tiempo = 0.000005 segundos
{'id': 1, 'direccion': 'Av. Callao 800', 'tareas': ['poda', 'extraccion']}
{'id': 4, 'direccion': 'Av. Córdoba 900', 'tareas': ['poda']}
Búsqueda Binaria por Tarea:2 árboles encontrados. Tiempo = 0.000005 segundos.
{'id': 1, 'direccion': 'Av. Callao 800', 'tareas': ['poda', 'extraccion']}
{'id': 4, 'direccion': 'Av. Córdoba 900', 'tareas': ['poda']}
Búsqueda Hash por tarea 'poda':2 árboles encontrados .Tiempo = 0.000001 segundos.
{'id': 1, 'direccion': 'Av. Callao 800', 'tareas': ['poda', 'extraccion']}
{'id': 4, 'direccion': 'Av. Córdoba 900', 'tareas': ['poda']}
```

6. Conclusiones

A partir de los resultados de las búsquedas sobre una base de datos simulada del arbolado urbano, se pueden extraer las siguientes conclusiones:

6.1 La elección del algoritmo de búsqueda depende del tipo de dato y la estructura de la información:

- En el caso de la **búsqueda por dirección**, al tratarse de un dato **único e irrepetible**, resultó más efectiva la **búsqueda binaria** (con la lista previamente ordenada). La **búsqueda por índice hash,** si se descarta el tiempo de construcción de la tabla hash, debería mejorar el tiempo de búsqueda.

- En cambio, la **búsqueda por tareas**, donde un árbol puede tener **múltiples tareas**, requiere una estrategia diferente. Aquí, la **búsqueda lineal** fue simple de implementar, pero la **tabla hash invertida** (índice de tareas) ofreció un rendimiento mucho más eficiente para búsquedas repetidas.

6.2 Comparación de rendimiento:

- La **búsqueda binaria por dirección** fue la más rápida (tiempos cercanos a 0.000000 s) cuando la lista estaba ordenada, y es ideal para datos únicos con acceso secuencial.

- La **búsqueda por índice hash** fue también muy eficiente (0.000001 s) para ambos tipos de búsquedas, pero el tiempo es fuertemente afectado por el tiempo de **creación del índice en el caso de búsqueda por dirección**, que es un costo fijo.

- La **búsqueda lineal**, si bien es funcional y fácil de programar, tiene tiempos notablemente mayores (0.03 a 0.06 s en listas grandes) y **genera errores** a medida que aumenta el volumen de datos.

6.3 Costo vs. beneficio de estructuras auxiliares:

- Las estructuras como **índices hash** o listas ordenadas **requieren procesamiento adicional y memoria**, pero **mejoran significativamente el tiempo de consulta** cuando se realizan muchas búsquedas.

- En cambio, si solo se realiza una búsqueda puntual sobre una lista pequeña, la **búsqueda lineal puede ser suficiente y más simple**.

6.4 Limitaciones y aplicabilidad:

- No todos los algoritmos pueden aplicarse a todos los casos. Por ejemplo, la **búsqueda binaria no puede utilizarse** con campos múltiples como las tareas, porque **no hay un orden único** posible.

- La **búsqueda por índice invertida** es ideal para campos no únicos (como tareas), permitiendo acceder directamente a todos los árboles asociados a una clave.

- **Posible mejora**: Actualmente, el índice hash se crea en tiempo de ejecución. Una mejora sería **almacenarlo y reutilizarlo**, reduciendo el tiempo de carga en futuras búsquedas.

7. Bibliografía

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3ra ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Python* (1st ed.). Wiley.

Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

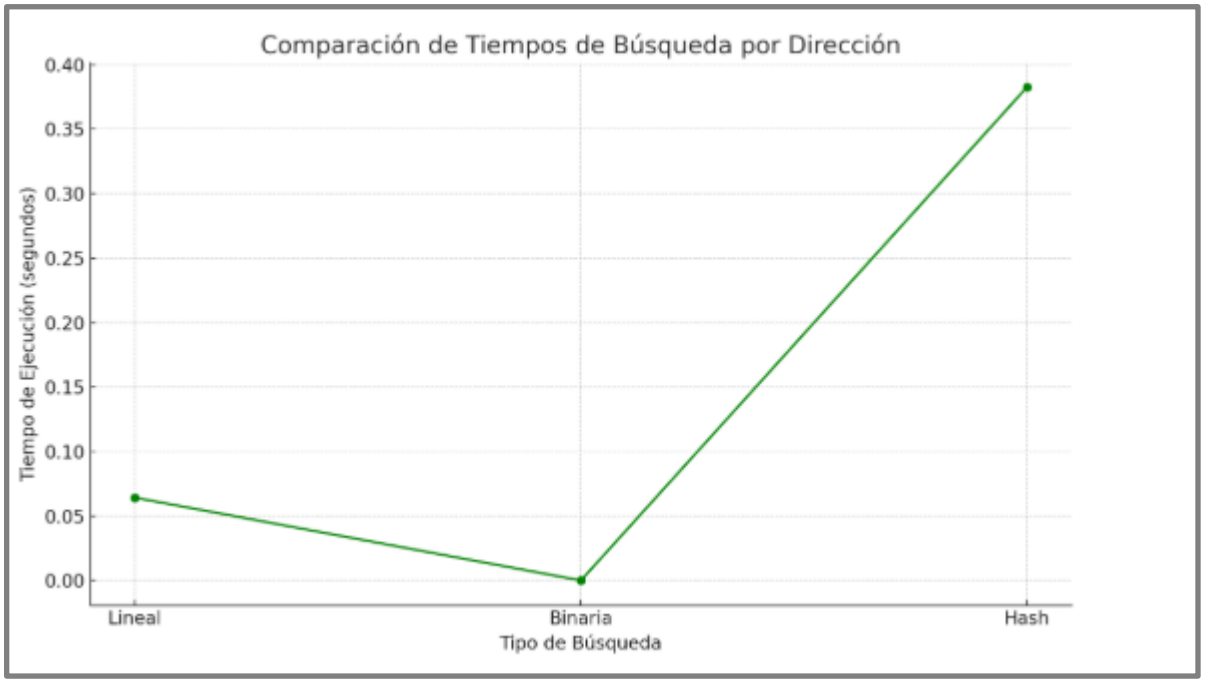
Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

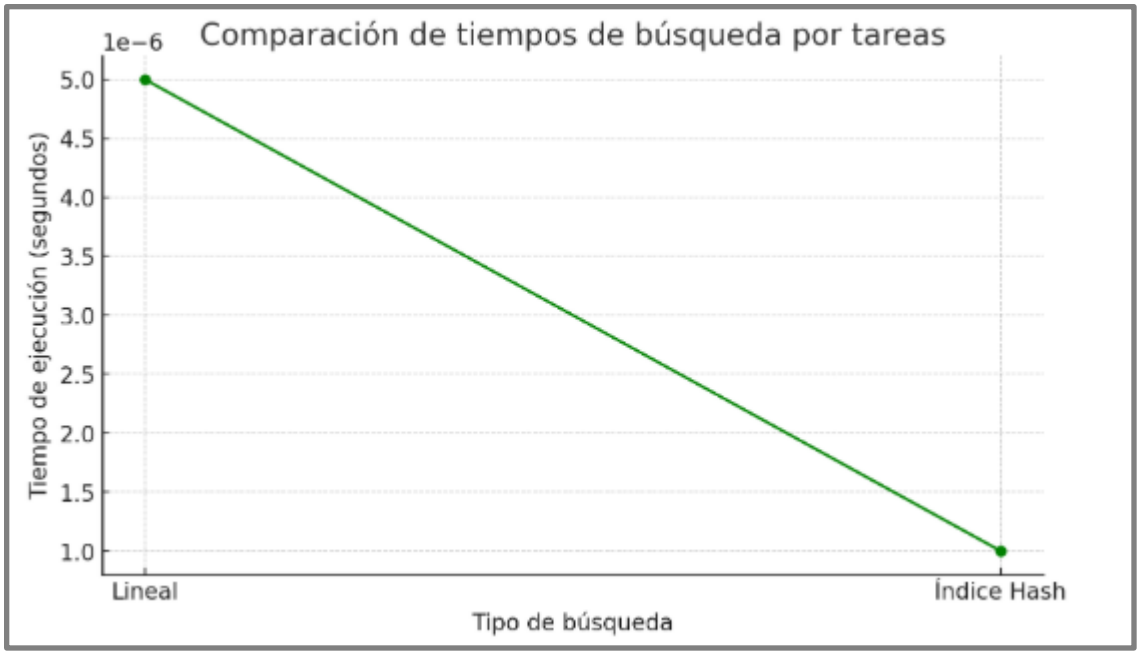
Van Rossum, G., & Drake Jr, F. L. (2009). *The Python Language Reference Manual*. Network Theory Ltd.

Downey, A. (2012). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.

8. Anexos

Gráfico de tiempo de ejecución en función de los tipos de búsqueda





Búsqueda de Índice Hash con la corrección de solo la búsqueda (no cuenta la creación del índice)

```python
# ANEXO : INDIDE HUSH SIN MEDICION DE TIEMPO EN CONSTRUCCION DEL INDICE
# Construir índice (solo una vez)
indice_direccion = {arbol["direccion"]: arbol for arbol in arboles}

# Medir solo la búsqueda
inicio_hash = time.time()
arbol = indice_direccion.get(direccion_objetivo)
fin_hash = time.time()

tiempo_hash2 = fin_hash - inicio_hash
```

Resultado de performance en busqueda por direccion (Busqueda hash2)

```
PS C:\Users\user> & C:/Users/user/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/user/OneDrive - Facultad de Agronomía - Universidad de Buenos Aires/Es
ritorio/mis proyectos/TP_busqueda_y_ordenamiento/busqueda_por direccion.py"
Búsqueda Lineal: Resultado = 533248, Tiempo = 0.078996 segundos
Búsqueda Binaria: Resultado = 481390, Tiempo = 0.000000 segundos
Búsqueda Hash: Árbol encontrado = {'direccion': 'Calle 533248'}, Tiempo = 0.546999 segundos
.....
Búsqueda Hash2: Árbol encontrado = {'direccion': 'Calle 533248'}, Tiempo = 0.000000 segundos
PS C:\Users\user>
```

9- **Enlace al video**

https://www.youtube.com/watch?v=awhiZGXb9cI&ab_channel=FlorenciaPaolazzi