

# Trabajo Práctico

## Algoritmo de búsqueda y ordenamiento

Alumnos:

- Alderete Daniel([danielalderete513@gmail.com](mailto:danielalderete513@gmail.com))
- Acosta Tadeo ([tadeoacosta14@gmail.com](mailto:tadeoacosta14@gmail.com))

Materia: Programación 1

Comisión: 6

Profesores: Cinthia Rigoni - Oscar Londero

Fecha de entrega: 09/06/2025

## Índice

1. Introducción .....	pág. 3
2. Marco Teórico .....	pág. 3
3. Caso Práctico .....	pág. 6
4. Metodología Utilizada .....	pág.9
5. Resultados Obtenidos .....	pág.10
6. Conclusiones .....	pág. 11
7. Bibliografía .....	pág. 11
8. Anexos .....	pág. 12

## 1. Introducción:

El estudio de los algoritmos de búsqueda y ordenamiento ha sido elegido para este trabajo debido a su papel esencial en la gestión y optimización de datos en el mundo del desarrollo de software. Estos algoritmos permiten organizar y recuperar información de manera eficiente, contribuyendo significativamente al rendimiento de los programas y aplicaciones.

Dentro del área de programación, la correcta implementación de estos algoritmos es clave para mejorar la velocidad y eficacia de procesos computacionales. Su aplicación abarca múltiples dominios, desde el manejo de grandes volúmenes de datos en sistemas de gestión hasta la optimización de búsquedas en estructuras complejas.

## Marco Teórico:

### **ALGORITMOS DE ORDENAMIENTO**

En la informática, los algoritmos de ordenamiento son cruciales para la optimización de una tarea, estos permiten organizar datos de manera que puedan ser accedidos y utilizados de manera más eficiente. Un algoritmo de ordenamiento permite reorganizar una lista de elementos o nodos en un orden específico, por ejemplo, de forma ascendente o descendente dependiendo de la ocasión. Entre los más comunes se encuentran:

#### 1. Ordenamiento de Burbuja (Bubble Sort):

El algoritmo de ordenamiento por burbuja es uno de los más simples, pero menos eficientes. Funciona comparando pares de elementos e intercambiándolos si están en el orden incorrecto, este proceso se hace una y otra vez hasta que la lista esté ordenada de forma correcta.

#### 2. Ordenamiento por selección:

El ordenamiento por selección es un algoritmo que te permite ordenar los valores de una lista, y tiene una complejidad de  $O(n^2)$ . Y también, cabe señalar que es más eficiente que el ordenamiento burbuja porque realiza intercambios de máximo una vez por cada recorrido, mientras que en burbuja realizamos de cero a más intercambios por recorrido.

3. Ordenamiento por inserción (Insertion Sort):

El algoritmo de ordenamiento por inserción es un algoritmo simple pero eficiente. Funciona dividiendo la lista en dos partes, una parte ordenada y otra desordenada, a medida que se recorre la lista desordenada, se insertan elementos en la posición correcta en la parte ordenada.

4. Ordenamiento rápido:

El ordenamiento rápido usa dividir y conquistar para obtener las mismas ventajas que el ordenamiento por mezcla, pero sin utilizar almacenamiento adicional. Sin embargo, es posible que la lista no se divida por la mitad. Cuando esto sucede, veremos que el desempeño disminuye.

5. Ordenamiento por mezcla:

El ordenamiento por mezcla es un algoritmo recursivo que divide continuamente una lista por la mitad. Si la lista está vacía o tiene un solo ítem, se ordena por definición (el caso base). Si la lista tiene más de un ítem, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades. Una vez que las dos mitades están ordenadas, se realiza la operación fundamental, denominada mezcla.

## ALGORITMOS DE BUSQUEDA

Los algoritmos de búsqueda permiten encontrar un elemento específico dentro de una lista. Según cómo estén organizados los datos, se elige el algoritmo más adecuado: si la lista está ordenada, se puede usar la **búsqueda binaria**, que es más eficiente. En cambio, si la lista está desordenada, se debe aplicar la **búsqueda lineal**, que revisa cada elemento uno por uno. Ambos algoritmos son fundamentales en programación y se usan frecuentemente para resolver problemas relacionados con estructuras de datos.

Además, es importante considerar la eficiencia de cada algoritmo, ya que afecta directamente al tiempo de ejecución del programa, especialmente cuando se trabaja con grandes volúmenes de datos. Por eso, elegir el algoritmo correcto puede marcar una gran diferencia en el rendimiento de una aplicación.

### BUSQUEDA LINEAL

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento

específico. Este algoritmo es muy sencillo de implementar en código, pero puede ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento.

### **Ventajas y Desventajas del Algoritmo de Búsqueda Lineal**

#### **Ventajas:**

- **Sencillez:** La búsqueda lineal es uno de los algoritmos de búsqueda más simples y fáciles de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
- **flexibilidad:** La búsqueda lineal puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.

#### **Desventajas:**

- **Ineficiencia en listas grandes:** La principal desventaja de la búsqueda lineal es su ineficiencia en listas grandes. Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.
- **No es adecuada para listas ordenadas:** Aunque puede funcionar en listas no ordenadas, la búsqueda lineal no es eficiente para listas ordenadas. En tales casos, algoritmos de búsqueda más eficientes, como la búsqueda binaria, son preferibles.

## **BUSQUEDA BINARIA**

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.

### **Ventajas y Desventajas del Algoritmo de Búsqueda Binaria**

#### **Ventajas:**

- **Eficiencia de listas ordenadas:** La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de **O (log n)**, lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.
- **Menos comparaciones:** Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que lo hace más rápido para encontrar el objetivo.

#### **Desventajas:**

- **Requiere una lista ordenada:** La búsqueda binaria sólo es aplicable a listas ordenadas. Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de usar la búsqueda binaria.
- **Mayor complejidad de implementación:** Comparado con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su naturaleza recursiva.

### 3. Caso Práctico:

#### Búsqueda Binaria

En este ejemplo, hacemos uso de un algoritmo de búsqueda binario para encontrar el número **27** en una lista de elementos ordenados, para poder encontrar el elemento que buscamos podemos hacer uso de una función recursiva, en esta función el caso base sería si el número de la lista en la posición centro es igual al número que buscamos, de ser así retornamos el valor de la variable centro este sería el índice del número, de lo contrario, dividimos la lista en dos mitades y hacemos el llamado recursivo hasta encontrar el número que buscamos pero si el número no se encuentra en la lista retornamos **-1**.

```
def binary_search(lista, objetivo, inicio, fin):
    # Condición de corte: si el inicio supera al fin, el elemento no está en la lista
    if inicio > fin:
        return -1
    # Calculamos el índice del elemento central
    centro = (inicio + fin) // 2
    # Si el valor del centro es igual al objetivo, lo encontramos
    if lista[centro] == objetivo:
        return centro
    # Si el valor del centro es menor al objetivo, buscamos en la mitad derecha
    elif lista[centro] < objetivo:
        return binary_search(lista, objetivo, centro + 1, fin)
    # Si el valor del centro es mayor, buscamos en la mitad izquierda
    else:
        return binary_search(lista, objetivo, inicio, centro - 1)

# --- Ejemplo de uso del algoritmo ---
# Lista ordenada (requisito fundamental para aplicar búsqueda binaria)
lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
# Número que queremos buscar
numero_objetivo = 27
# Definimos el rango de búsqueda inicial (toda la lista)
inicio_busqueda = 0
fin_busqueda = len(lista) - 1

# Llamamos a la función y guardamos el resultado
resultado = binary_search(lista, numero_objetivo, inicio_busqueda, fin_busqueda)
# Mostramos el resultado
if resultado != -1:
    print(f"El número {numero_objetivo} se encuentra en la posición {resultado}.")
else:
    print(f"El número {numero_objetivo} NO se encuentra en la lista.")
```

## Búsqueda Lineal

En este ejemplo de código, necesitamos buscar el número 39, para buscarlo de forma lineal simplemente recorremos la lista con la ayuda de una estructura de bucle for y luego preguntamos si el elemento actual es igual a el elemento que estamos buscando, de ser así retornamos el índice del elemento y terminamos el bucle, pero si el bucle termina y no retorno ningún elemento significa que el número que buscamos no se encuentra en la lista por lo que retornamos -1. Este algoritmo puede ser útil para recorrer listas pequeñas o listas desordenadas, pero no es eficiente para recorrer listas demasiado largas.

```
def linear_search (lista, objetivo):

    # Recorremos todos los elementos de la lista uno por uno

    for i in range(len(lista)):

        # Si el elemento actual coincide con el objetivo, devolvemos su índice

        if lista[i] == objetivo:

            return i

    # Si terminamos de recorrer la lista y no encontramos el objetivo, devolvemos -1

    return -1

# Lista de números ordenada (aunque la búsqueda lineal no necesita que lo esté)

lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]

# Número que queremos buscar

numero_objetivo = 39

# Llamamos a la función de búsqueda lineal y guardamos el resultado

resultado = linear_search (lista, numero_objetivo)

# Verificamos si el número fue encontrado y mostramos el resultado

if resultado! = -1:

    print(f"El número {numero_objetivo} se encuentra en la posición: {resultado}")

else:
```

## Ordenamiento por burbuja

En este ejemplo de ordenamiento por burbuja se desarrolla un programa en python, donde se toma el segundo elemento de la lista y con la ayuda de un bucle while intercambiamos el número actual con el número anterior mientras que el número anterior sea más grande que el número actual, este proceso se hace una y otra vez hasta que la lista esté completamente ordenada.

```
def insertion_sort(lista):
    for i in range(1, len(lista)):
        actual = lista[i]
        index = i

        while index > 0 and lista[index - 1] > actual:
            lista[index] = lista[index - 1]
            index = index - 1
        lista[index] = actual
    return lista

lista_desordenada = [39, 45, 32, 4, 2, 85, 43, 7, 18, 16, 5, 67, 32]
lista_ordenada = insertion_sort(lista_desordenada)
print(lista_ordenada) # output: [2, 4, 5, 7, 16, 18, 32, 32, 39, 43, 45, 67, 85]
```

## 4. Metodología Utilizada:

- Se realizó una búsqueda y análisis de diversas fuentes bibliográficas y sitios especializados para obtener información sobre algoritmos de ordenamiento y búsqueda.
- Se dividieron las tareas según los temas: un integrante se enfocó en los algoritmos de ordenamiento, y el otro en los de búsqueda.
- A partir del análisis teórico, se identificó un problema práctico para resolver mediante la aplicación de los algoritmos estudiados.
- Para el desarrollo del programa se utilizó el entorno de desarrollo Visual Studio Code (VSC).
- Se creó un repositorio en GitHub con el objetivo de trabajar de forma colaborativa y organizada.
- Finalmente, se realizaron pruebas sobre el código implementado para verificar su correcto funcionamiento y validar los resultados obtenidos.

## 5. Resultados Obtenidos:

### Funcionalidades que se lograron:

- **Búsqueda binaria:** permitió encontrar un número dentro de una lista ordenada de forma eficiente. Se verificó su correcto funcionamiento en varios casos, retornando el índice esperado del elemento objetivo.
- **Búsqueda lineal:** también localizó correctamente los elementos, recorriendo secuencialmente la lista.
- **Ordenamiento por burbuja:** reordenó con éxito listas desordenadas, confirmando que los elementos quedaron en orden ascendente al final del proceso.

### Casos de prueba realizados:

- Se probó la búsqueda binaria con valores presentes (como 27 o 15) y valores ausentes (como 100) para confirmar que devuelve el índice correcto o -1 si no se encuentra.
- La búsqueda lineal se testeo con el valor 39 y funcionó correctamente.
- El algoritmo de ordenamiento fue probado con una lista aleatoria de enteros, y el resultado fue una lista ordenada como se esperaba.

### Errores corregidos:

- En la implementación inicial de la búsqueda lineal, había un error de sintaxis en `if resultado! = -1:` que fue corregido a `if resultado != -1:`.
- Se ajustaron algunos print y nombres de variables para mejorar la legibilidad y el seguimiento del código.

### Evaluación de rendimiento:

- En listas pequeñas, ambos métodos de búsqueda funcionaron correctamente, pero se destaca que:
  - La **búsqueda binaria** es mucho más eficiente en listas grandes **ordenadas**, ya que reduce el número de comparaciones utilizando el enfoque de "divide y vencerás".

- La **búsqueda lineal** es útil cuando la lista no está ordenada o es muy pequeña, aunque menos eficiente en grandes volúmenes de datos.

#### **Enlace al repositorio:**

El trabajo completo, incluyendo los algoritmos implementados y casos de prueba, está disponible en el siguiente repositorio colaborativo de GitHub:

<https://github.com/Dann07-ops/TPIintegradorP1>

### **6. Conclusiones:**

La realización de este trabajo práctico nos permitió profundizar en el estudio y la implementación de algoritmos de búsqueda y ordenamiento, herramientas fundamentales dentro del mundo de la programación.

A través de la investigación y la práctica, comprendimos el funcionamiento interno de distintos algoritmos como búsqueda binaria, búsqueda lineal e inserción. Aprendimos a reconocer en qué situaciones conviene aplicar uno u otro, evaluando su eficiencia y complejidad. También fortalecimos nuestras habilidades en programación, especialmente en el uso de funciones recursivas y estructuras de control.

El conocimiento de estos algoritmos es aplicable a una amplia variedad de proyectos de software, desde simples programas de filtrado hasta sistemas que manejan grandes volúmenes de datos. Saber cómo ordenar datos eficientemente o cómo encontrarlos rápidamente es clave para optimizar el rendimiento de cualquier aplicación.

Como extensión futura del trabajo, podríamos implementar y comparar otros algoritmos como merge sort, quick sort o heap sort, y realizar pruebas de rendimiento más precisas utilizando herramientas de medición de tiempo. Además, sería interesante aplicar estos algoritmos a estructuras de datos más complejas como diccionarios o listas de objetos.

Una de las principales dificultades fue entender y aplicar correctamente la recursividad en la búsqueda binaria, así como interpretar correctamente los índices en las distintas fases del algoritmo. También tuvimos errores de sintaxis que fueron resueltos revisando el código línea por línea y utilizando el intérprete de Python para detectar y corregir problemas.

### **7. Bibliografía:**

- <https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python>
- <https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoRapido.html>

- <https://runestone.academy/ns/books/published/pythoned/SortSearch/EIOrdenamientoPorMezcla.html>

## 8. Anexos:

- Captura del programa funcionando

## Captura del caso práctico de ordenamiento por burbuja.

The screenshot shows a Python code editor interface with the following details:

- Title Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help, Search.
- File Path:** C:\Users\User>User>OneDrive>Desktop>TrabajoIntegradorP1>TPIntegradorP1>TrabajoPracticoProgramaciónEjemOrdenamiento.py ...
- Code Content:**

```
1 #Caso practico de Ordenamiento por burbuja
2 def insertion_sort(lista):
3     for i in range(1, len(lista)):
4         actual = lista[i]
5         index = i
6         while index > 0 and lista[index - 1] > actual:
7             lista[index] = lista[index - 1]
8             index = index - 1
9         lista[index] = actual
10    return lista
11
12 lista_desordenada = [39, 45, 32, 4, 2, 85, 43, 7, 18, 16, 5, 67, 32]
13 lista_ordenada = insertion_sort(lista_desordenada)
14 print(lista_ordenada) # output: [2, 4, 5, 7, 16, 18, 32, 32, 39, 43, 45, 67, 85]
```
- Terminal Output:**

```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python313/python.exe c:/Users/User/OneDrive/Desktop/TrabajoIntegradorP1/TPIntegradorP1/TrabajoPracticoProgramaciónEjemOrdenamiento.py
[2, 4, 5, 7, 16, 18, 32, 32, 39, 43, 45, 67, 85]
```
- Status Bar:** Ln 1, Col 1 | Spaces: 4 | UTF-8 | CRLF | Python | 3.13.2

Captura del caso práctico de Búsqueda Lineal.

Python

```
1 def linear_search(lista, objetivo):
2     for i in range(len(lista)):
3         if lista[i] == objetivo:
4             return i
5     return -1
6
7
8
9
10 lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
11 numero_objetivo = 39
12 resultado = linear_search(lista, numero_objetivo)
13
14 if resultado != -1:
15     print(f"El número {numero_objetivo} se encuentra en la posición: {resultado}")
```

Consola ⓘ

```
El número 39 se encuentra en la posición: 14
```

Captura de caso práctico de Búsqueda Binaria.

```
Python
1 def binary_search(lista, objetivo, inicio, fin ):
2     if inicio > fin:
3         return -1
4
5     centro = (inicio + fin) // 2
6     if lista[centro] == objetivo:
7         return centro
8     elif lista[centro] < objetivo:
9         return binary_search(lista, objetivo, centro + 1, fin)
10    else:
11        return binary_search(lista, objetivo, inicio, centro - 1)
12
13
14 # Ejemplo de uso
15 lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
```

- Aclaramos que tenemos el link del repositorio y video.  
<https://github.com/Dann07-ops/TPIintegradorP1.git>
- Se informa que dentro del repositorio se encuentran los archivos Python mostrando el funcionamiento de los casos prácticos.