

# TRABAJO PRÁCTICO INTEGRADOR

## Algoritmos de búsqueda y ordenamiento

ALUMNOS:

Neto, Franco - fnetoaguirre@gmail.com

Noguera Ríos, Joana - noguerariosjoana@gmail.com

MATERIA:

Programación 1

PROFESORA:

Julieta Trapé

FECHA DE ENTREGA:

Lunes, 9 de Junio de 2025

TEMA ELEGIDO:

Algoritmos de búsqueda y ordenamiento

## índice

1. Introducción .....	página 2
2. Marco Teórico .....	página 3
3. Caso Práctico .....	página 19
4. Metodología Utilizada .....	página 21
5. Resultados Obtenidos .....	página 27
6. Conclusiones .....	página 29
7. Bibliografía .....	página 30
8. Anexos .....	página 30

## Introducción

En el desarrollo de Software, los algoritmos de búsqueda y ordenamiento son herramientas fundamentales que permiten organizar y manejar datos de manera eficiente. Este trabajo se centra en estos algoritmos porque son esenciales para resolver problemas en casi cualquier programa que procese listas, bases de datos o estructuras más complejas.

El tema fue elegido porque resulta muy importante comprender cómo funcionan estos algoritmos para poder aplicarlos correctamente y mejorar el rendimiento de las aplicaciones. Además, aprender sobre estos temas ayuda a tener una base sólida que servirá para enfrentar problemas más avanzados en el futuro.

En la programación, el uso adecuado de estos algoritmos puede optimizar el tiempo de ejecución de programas y escribir códigos más eficientes y escalables. Saber elegir entre distintos métodos de búsqueda y ordenamiento según el caso es una habilidad que cualquier programador necesita desarrollar para enfrentar proyectos reales.

El objetivo principal de este trabajo es analizar y comparar diferentes algoritmos de búsqueda y ordenamiento, evaluando su rendimiento y sus características. Se busca aplicar estos conocimientos a través de un proyecto práctico en Python, para entender cómo se comportan en la práctica y poder elegir el más adecuado según las necesidades. De esta manera, se pretende reforzar lo aprendido en la materia y mejorar la capacidad para aplicar estos conceptos en situaciones reales.

## Marco teórico

En programación informática, la búsqueda y la ordenación son técnicas esenciales para organizar y manipular datos. La búsqueda implica encontrar un elemento específico dentro de un conjunto de datos, mientras que la ordenación implica organizar los elementos de un conjunto de datos en un orden específico. Ambas técnicas desempeñan un papel vital en el rendimiento y la eficiencia de los programas informáticos.

### Eficiencia de un algoritmo

La complejidad temporal o complejidad algorítmica, se centra en entender y cuantificar cómo el tiempo de ejecución (o el número de operaciones) de un algoritmo crece en relación con el tamaño de la entrada. En otras palabras, es una medida que nos indica cuánto tiempo tardará un algoritmo en ejecutarse en función del tamaño de su entrada.

#### Complejidad Temporal

examina cómo cambia el tiempo de ejecución a medida que el tamaño de la entrada varía. Se interesa en patrones de crecimiento y cómo estos se relacionan con la eficiencia del algoritmo. La notación Big O (O) es comúnmente utilizada para expresar la complejidad temporal

#### Escalabilidad

A medida que tus proyectos crecen en tamaño y complejidad, la eficiencia de tus algoritmos se vuelve aún más crítica. Comprender la complejidad temporal te permite seleccionar algoritmos que escalen bien con un aumento en el tamaño de la entrada.

#### Cómo funciona la notación Big O (O)

Estas clases de complejidad temporal son herramientas cruciales para entender cómo se comportan los algoritmos en diferentes situaciones:

- $O(1)$  constante:

## Trabajo Integrador - Programación 1

Si un algoritmo tiene complejidad constante, significa que su rendimiento no cambia, independientemente del tamaño de la entrada. Es como decir que toma siempre la misma cantidad de tiempo, sin importar cuántos elementos estén involucrados.

- $O(n)$  lineal:

Si la complejidad es lineal, el tiempo de ejecución aumenta proporcionalmente al tamaño de la entrada. Si tienes 10 elementos, tomará el doble de tiempo que con 5 elementos.

- $O(\log n)$ : Logarítmica

El tiempo de ejecución aumenta logarítmicamente con el tamaño de la entrada. A medida que el tamaño de la entrada se duplica, el tiempo de ejecución solo aumenta en una cantidad constante. Por ejemplo; si tienes una lista ordenada y realizas una búsqueda binaria (dividir la lista a la mitad en cada paso), el tiempo de ejecución seguirá siendo manejable incluso para conjuntos de datos grandes.

- $O(n^2)$  cuadrática:

Si la complejidad es cuadrática, el tiempo de ejecución aumenta cuadráticamente con el tamaño de la entrada. Si tienes 10 elementos, tomará 100 veces más tiempo que con 1 elemento.

## Búsqueda y ordenamiento

### Algoritmos de búsqueda

La búsqueda es una operación fundamental en programación que se utiliza para encontrar un elemento específico dentro de un conjunto de datos. Es una tarea común en muchas aplicaciones, como bases de datos, sistemas de archivos y algoritmos de inteligencia artificial. Consiste en localizar un elemento en un conjunto de datos. Los dos métodos más comunes son:

#### Búsqueda Lineal

Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero puede ser

## Trabajo Integrador - Programación 1

lento para conjuntos de datos grandes.

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

Python Software Foundation. (2024). Python (Versión 3.12). <https://www.python.org/>

Este código es funcional pero poco eficiente, tardaría mucho tiempo en buscar un valor dentro de una lista.

Complejidad temporal:

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.

### Busqueda binaria

Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.

Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento buscado con el elemento central. Si el elemento buscado es menor que el central, se descarta la mitad derecha; si es mayor, se descarta la mitad izquierda.

```
def busqueda_binaria(lista, objetivo):  
  
    izquierda, derecha = 0, len(lista) - 1  
  
    while izquierda <= derecha:  
  
        medio = (izquierda + derecha) // 2
```

## Trabajo Integrador - Programación 1

```
if lista[medio] == objetivo:
    return medio
elif lista[medio] < objetivo:
    izquierda = medio + 1
else:
    derecha = medio - 1
return -1
```

Python Software Foundation. (2024). Python (Versión 3.12). <https://www.python.org/>

Complejidad temporal:

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.

### Algoritmos de ordenamiento

El ordenamiento organiza los datos de acuerdo a un criterio, como de menor a mayor o alfabéticamente. Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Objetivo:

El objetivo de los algoritmos de ordenamiento, es ordenar una lista de datos para realizar búsquedas o cualquier otra operación que se requiera con estas.

Importancia:

- Eficiencia: Mejoran el tiempo de ejecución de programas que manejan grandes cantidades de datos.

## Trabajo Integrador - Programación 1

- Organización: Permiten trabajar con datos de forma más clara y estructurada. Los algoritmos de búsqueda y ordenamiento son herramientas fundamentales en programación, aportando soluciones eficientes para organizar y recuperar información. Su relevancia se basa en aspectos como:
- Eficiencia: Al optimizar el acceso y la manipulación de datos, estos algoritmos mejoran significativamente el tiempo de ejecución de programas, especialmente aquellos que manejan grandes volúmenes de información.
- Organización: Facilitan la estructuración y presentación de datos de manera coherente, simplificando su análisis y comprensión.
- Escalabilidad: Su diseño permite manejar conjuntos de datos de diferentes tamaños, adaptándose a las necesidades cambiantes de un programa.
- Precisión: Garantizan la recuperación de resultados exactos y relevantes, evitando errores y ambigüedades en la búsqueda de información.
- Versatilidad: Se aplican en una amplia gama de contextos y dominios, desde bases de datos y sistemas de archivos hasta aplicaciones web y motores de búsqueda.

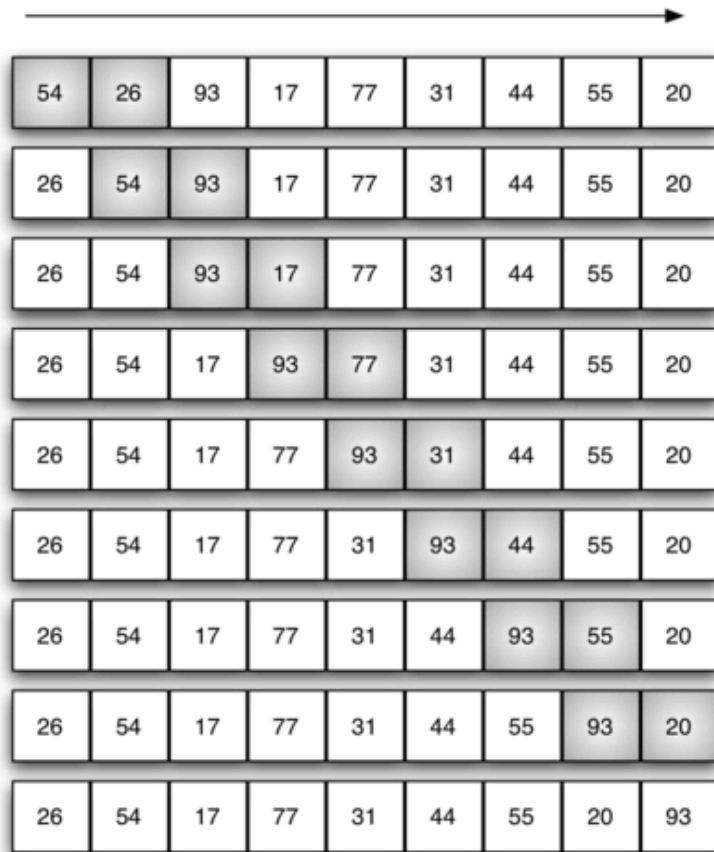
Existen muchos algoritmos de ordenamiento diferentes, cada uno con sus propias ventajas y desventajas. Algunos de los algoritmos de ordenamiento más comunes incluyen:

### Ordenamiento de Burbuja o bubble (Bubble Sort)

El ordenamiento de burbuja es un algoritmo de ordenamiento simple pero menos eficiente, que funciona intercambiando repetidamente elementos adyacentes si están en el orden incorrecto. El algoritmo comienza al principio de la lista y compara cada par de elementos adyacentes. Si no están en el orden correcto, se intercambian y el algoritmo continúa hasta que no se necesiten más intercambios. El ordenamiento de burbuja tiene una complejidad temporal de  $O(n^2)$  y no es adecuado para conjuntos de datos grandes.



## Trabajo Integrador - Programación 1



Ventajas y Desventajas del Ordenamiento de Burbuja (Bubble Sort):

Ventajas:

- Simplicidad: El algoritmo de burbuja es fácil de entender e implementar, lo que lo convierte en una buena opción para introducir conceptos de ordenamiento en la programación.
- Implementación sencilla: Requiere poca cantidad de código y no involucra estructuras de datos complejas.

Desventajas:

- Lento para listas grandes: Debido a su complejidad cuadrática el algoritmo de burbuja se vuelve lento en la práctica para listas de tamaño considerable. No considera el orden

## Trabajo Integrador - Programación 1

- - parcial: A diferencia de otros algoritmos, el algoritmo de burbuja realiza el mismo número de comparaciones e intercambios sin importar si la lista ya está en gran parte ordenada.

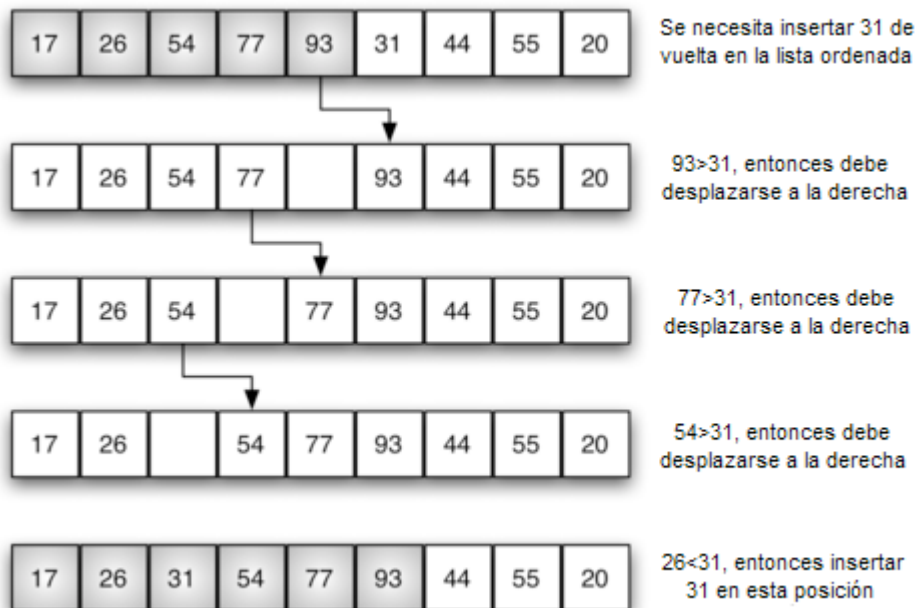
Complejidad temporal:

- 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:  $O(n^2)$ .

### Ordenamiento por Inserción

El ordenamiento por inserción funciona de una manera ligeramente diferente. Siempre mantiene una sublista ordenada en las posiciones inferiores de la lista. Cada ítem nuevo se “inserta” de vuelta en la sublista previa de manera que la sublista ordenada sea un ítem más largo. La siguiente imagen muestra el proceso de ordenamiento por inserción. Los ítems sombreados representan las sublistas ordenadas a medida que el algoritmo lleva a cabo cada pasada.

## Trabajo Integrador - Programación 1



Ventajas y Desventajas del Ordenamiento por inserción (Insertion Sort)

## Trabajo Integrador - Programación 1

### Ventajas:

- Baja sobrecarga: Requiere menos comparaciones y movimientos que algoritmos como el ordenamiento de burbuja, lo que lo hace más eficiente en términos de intercambios de elementos.
- Simplicidad: el ordenamiento por inserción es uno de los algoritmos de ordenamiento más simples de implementar y entender. Esto lo hace adecuado para enseñar conceptos básicos de ordenamiento.

### Desventajas:

- Ineficiencia en listas grandes: A medida que el tamaño de la lista aumenta, el rendimiento del ordenamiento por inserción disminuye. Su complejidad cuadrática de  $O(n^2)$  en el peor caso lo hace ineficiente para las listas grandes.
- No escalable: Al igual que otros algoritmos de complejidad cuadrática, el ordenamiento por inserción no es escalable para listas grandes, ya que su tiempo de ejecución aumenta considerablemente con el tamaño de la lista.

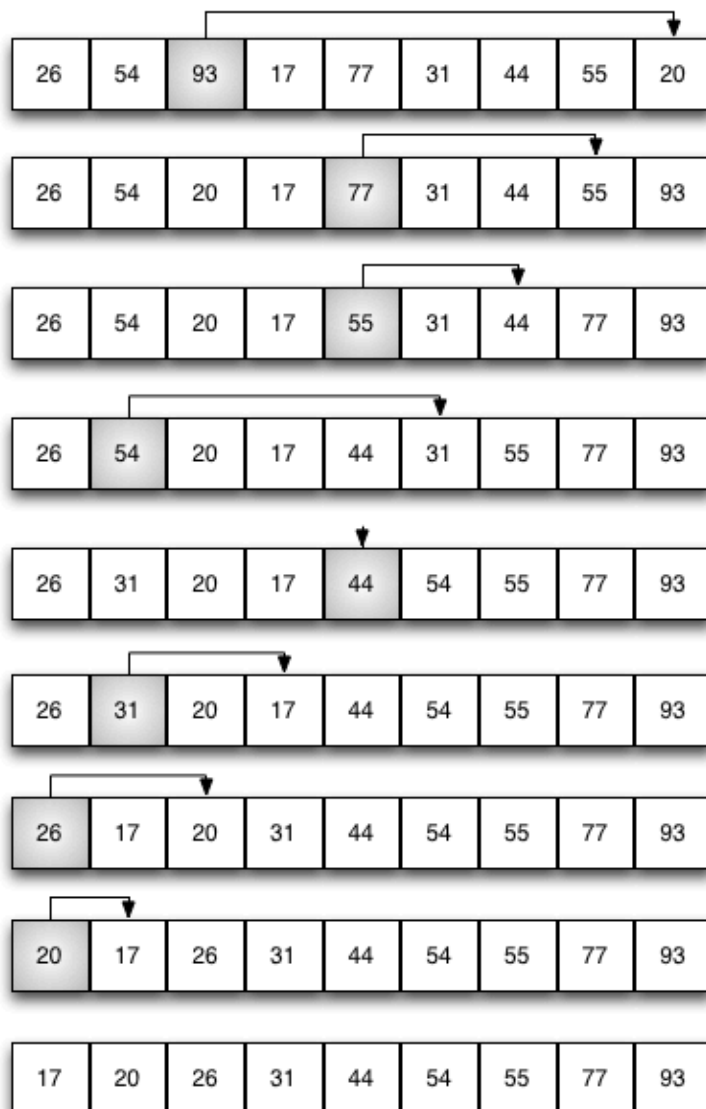
### Complejidad temporal:

- Peor caso:  $O(n^2)$ , cuando la lista está en orden inverso.
- Mejor caso:  $O(n)$ , cuando la lista ya está ordenada
- Caso promedio:  $O(n^2)$ .

### Ordenamiento por selección

El ordenamiento por selección es otro algoritmo de ordenamiento simple que funciona buscando repetidamente el elemento más pequeño de la parte no ordenada de la lista e intercambiándolo con el primer elemento. A continuación, el algoritmo pasa al segundo elemento, encuentra el elemento más pequeño de los elementos no ordenados restantes y lo intercambia con este, y así sucesivamente hasta que toda la lista esté ordenada. El ordenamiento por selección también tiene una complejidad temporal de  $O(n^2)$  y no es adecuado para conjuntos de datos grandes.

## Trabajo Integrador - Programación 1



### Ventajas

- Fácil de entender e implementar
- No requiere memoria adicional
- Realiza el mínimo número posible de intercambios
- Su comportamiento no cambia con el orden inicial de los datos

### Desventajas

## Trabajo Integrador - Programación 1

- No es eficiente para las listas grandes (su complejidad es siempre  $O(n^2)$ , sin importar el caso).
- Puede cambiar el orden de elementos iguales.
- No aprovecha listas parcialmente ordenadas.

Complejidad temporal:

- Peor caso:  $O(n^2)$ , ya que siempre realiza comparaciones para encontrar el mínimo.
- Mejor caso:  $O(n^2)$ , incluso si la lista está ordenada.
- Caso promedio:  $O(n^2)$

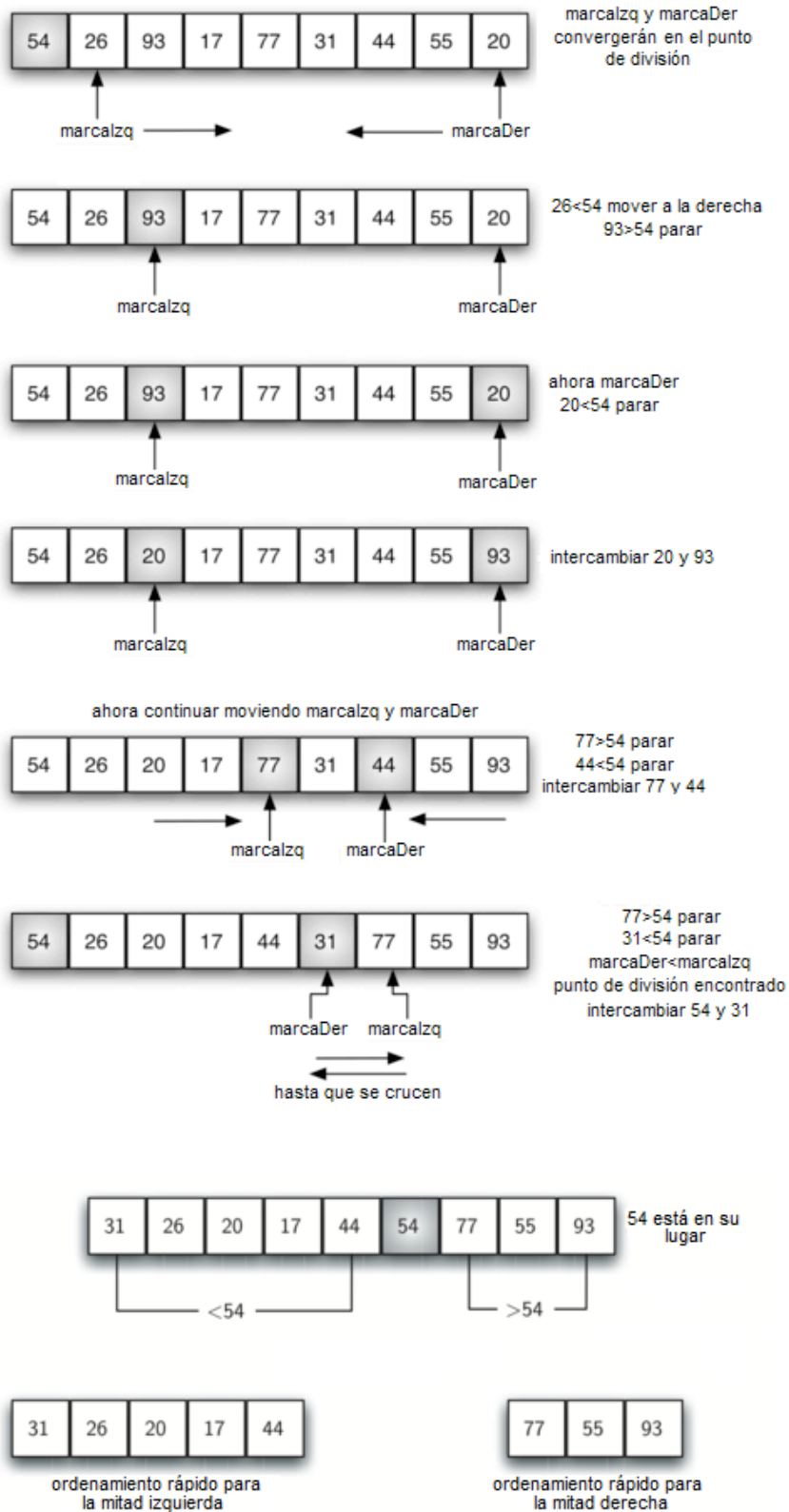
### Quicksort (Ordenamiento rápido)

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.

Un ordenamiento rápido primero selecciona un valor, que se denomina el valor pivote. Aunque hay muchas formas diferentes de elegir el valor pivote, simplemente usaremos el primer ítem de la lista. El papel del valor pivote es ayudar a dividir la lista. La posición real a la que pertenece el valor pivote en la lista final ordenada, comúnmente denominado punto de división, se utilizará para dividir la lista para las llamadas posteriores a la función de ordenamiento rápido.



## Trabajo Integrador - Programación 1



## Trabajo Integrador - Programación 1

### Ventajas

- Muy rápido en la práctica para listas grandes
- Requiere poca memoria adicional.
- Es el algoritmo más utilizado en muchas bibliotecas estándar.

### Desventajas

- En el peor caso puede ser lento
- No es estable (puede cambiar el orden de elementos iguales)
- Rendimiento sensible a la elección del pivote

### Complejidad temporal:

- 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:  $O(n \log n)$ .

Este ordenamiento es muy útil para listas con una mayor cantidad de datos. Tener en cuenta que para listas más pequeñas es más eficiente usar el ordenamiento de selección o inserción.

### Comparación entre Algoritmos de Ordenamiento

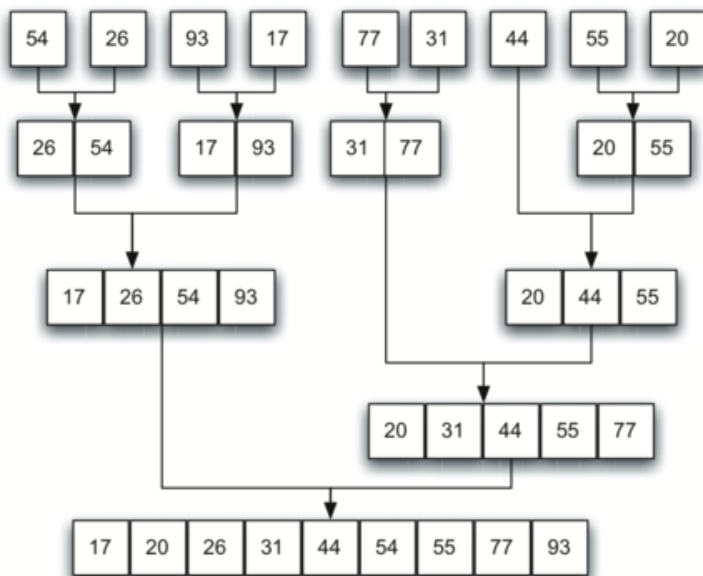
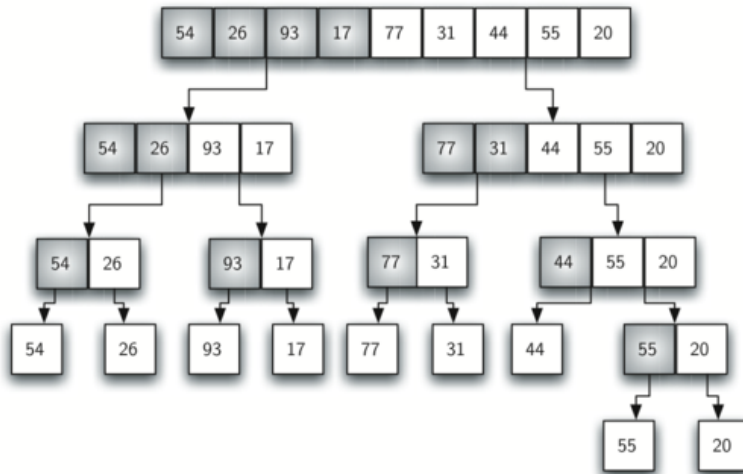
#### MergeSort (Ordenamiento por Mezcla):

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.



## Trabajo Integrador - Programación 1

para ambas mitades. Una vez que las dos mitades están ordenadas, se realiza la operación fundamental, denominada mezcla. La mezcla es el proceso de tomar dos listas ordenadas más pequeñas y combinarlas en una sola lista nueva y ordenada



Ventajas

## Trabajo Integrador - Programación 1

- Rendimiento consistente, incluso en el peor caso.
- Estable (mantiene el orden de elementos iguales).
- Ideal para ordenar listas enlazadas o archivos muy grandes.

### Desventajas

- Requiere memoria adicional  $O(n)$
- Puede ser más lento que Quicksort en listas pequeñas por la sobrecarga de llamadas recursivas y copias.

### Complejidad temporal

- Peor caso:  $O(n \log n)$
- Mejor caso:  $O(n \log n)$
- Caso promedio:  $O(n \log n)$

### Tabla de Complejidad Temporal y Escalabilidad

La siguiente tabla resume la complejidad temporal y la escalabilidad de los principales algoritmos de búsqueda y ordenamiento explicados anteriormente.

Esta comparación permite visualizar cuál es el comportamiento esperado de cada algoritmo según el tamaño de los datos y su eficiencia relativa.

## Trabajo Integrador - Programación 1

Algoritmo	Complejidad Temporal	Escalabilidad	Característica Clave
Búsqueda Lineal	Mejor: $O(1)$ Peor/Promedio: $O(n)$	Baja	Recorre todos los elementos uno por uno. Ineficiente en listas grandes
Búsqueda Binaria	Mejor: $O(1)$ Peor/Promedio: $O(\log n)$	Alta	Muy eficientes con listas ordenadas. Reduce la lista a la mitad en cada paso
Ordenamiento de Burbuja (Bubble Sort)	Mejor: $O(n)^*$ Peor/Promedio: $O(n^2)$	Muy baja	Simplicidad extrema, pero rendimiento muy pobre en listas grandes
Ordenamiento por Inserción (Insertion Sort)	Mejor: $O(n)$ Peor/Promedio: $O(n^2)$	Media-baja	Eficiente para listas pequeñas o casi ordenadas.
Ordenamiento por Selección (Selection Sort)	$O(n^2)$ en todos los casos	Baja	Siempre realiza las mismas comparaciones, Poco eficiente
Ordenamiento Rápido (Quicksort)	Mejor/Promedio: $O(n \log n)$ Peor: $O(n^2)$	Alta	Muy rápido en la práctica, sensible en la elección del pivote
Ordenamiento por Mezcla (Merge Sort)	$O(n \log n)$ en todos los casos	Alta	Estable y confiable para grandes volúmenes, requiere más memoria

## Caso Práctico

Para entender mejor cómo funcionan algunos algoritmos de ordenamiento y búsqueda, realizamos un proyecto práctico que consistió en desarrollar un programa en Python. Este programa nos permitió probar diferentes técnicas para ordenar listas de números y buscar elementos dentro de esas listas. El objetivo principal fue observar cómo se comportan estos algoritmos en distintos escenarios, qué tan eficientes son y qué tan fáciles son de implementar y entender.

Durante el desarrollo de este trabajo, tuvimos la oportunidad de aplicar conocimientos teóricos vistos en clase y comprobar su funcionamiento de forma real. Al trabajar con código, logramos visualizar con más claridad qué hace cada algoritmo paso a paso y cómo afectan las decisiones de programación al rendimiento y la claridad del código. El enfoque que adoptamos fue modular: dividimos cada algoritmo en una función separada y probamos todo con ejemplos concretos.

El programa fue diseñado para realizar dos tareas principales:

1. Ordenar listas de números utilizando dos algoritmos conocidos: bubble sort (ordenamiento por burbuja) y selection sort (ordenamiento por selección). Ambos organizan los números de menor a mayor, pero lo hacen de formas diferentes. Mientras que bubble sort intercambia elementos adyacentes repetidamente si están en orden incorrecto, selection sort busca el número más pequeño de toda la lista y lo pone al principio, repitiendo el proceso hasta ordenar toda la lista.
2. Buscar un número específico en la lista utilizando dos métodos: búsqueda lineal, que recorre la lista de principio a fin comparando uno por uno, y búsqueda binaria, que es más rápida pero necesita que la lista esté previamente ordenada. Este método divide la lista por la mitad y determina si el número está en la parte izquierda o derecha, repitiendo el proceso hasta encontrarlo o determinar que no está.

Para que el código fuera más fácil de leer y de mantener, implementamos cada algoritmo como una función distinta. Esto nos permitió probar cada uno de forma independiente, corregir errores sin afectar el resto del programa y, seguramente, reutilizar el código en otros ejercicios.

Descubrimos que no todos los algoritmos funcionan igual de bien en todos los casos. Por ejemplo, bubble sort fue más lento en listas muy desordenadas, mientras que selection sort mantuvo un rendimiento más constante, aunque tampoco fue muy rápido. La búsqueda binaria, por otro lado,

## Trabajo Integrador - Programación 1

fue muy eficiente en listas ordenadas, pero no se podía usar en listas desordenadas sin ordenarlas primero.

## Metodología utilizada

La elaboración del trabajo se realizó en las siguientes etapas:

- Recolección de información teórica en documentación confiable.
- Implementación en Python de los algoritmos estudiados y pruebas con diferentes conjuntos de datos. .

```
# ORDENAMIENTO - BURBUJA

def bubble_sort(arr):
    """
    Ordenamiento por burbuja: compara elementos adyacentes e intercambia si están
    desordenados.
    """
    n = len(arr)
    for i in range(n):
        # Últimos i elementos ya están ordenados
        for j in range(0, n - i - 1):
            # Si el elemento actual es mayor que el siguiente, los intercambia
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# ORDENAMIENTO - SELECCIÓN

def selection_sort(arr):
    """
```

Ordenamiento por selección: busca el elemento más pequeño y lo coloca al principio.

```
"""  
  
n = len(arr)  
  
for i in range(n):  
  
    min_idx = i # Se asume que el índice mínimo es el actual  
  
    for j in range(i + 1, n):  
  
        # Si encuentra un elemento menor, actualiza el índice mínimo  
  
        if arr[j] < arr[min_idx]:  
  
            min_idx = j  
  
    # Intercambia el elemento actual con el mínimo encontrado  
  
    arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

# BÚSQUEDA - LINEAL

```
def busqueda_lineal(arr, objetivo):  
  
    """  
  
    Búsqueda lineal: recorre todos los elementos hasta encontrar el objetivo.  
  
    """  
  
    for i in range(len(arr)):  
  
        # Si encuentra el objetivo, devuelve su posición  
  
        if arr[i] == objetivo:  
  
            return i
```

```
# Si no lo encuentra, devuelve -1

return -1

# BÚSQUEDA - BINARIA

def busqueda_binaria(arr, objetivo):
    """
    Búsqueda binaria: requiere que la lista esté ordenada.
    Divide y conquista hasta encontrar el valor.
    """
    inicio = 0
    fin = len(arr) - 1

    while inicio <= fin:

        medio = (inicio + fin) // 2 # Calcula el punto medio

        if arr[medio] == objetivo:

            return medio # Elemento encontrado

        elif arr[medio] < objetivo:

            inicio = medio + 1 # Busca en la mitad derecha

        else:

            fin = medio - 1 # Busca en la mitad izquierda

    return -1 # Elemento no encontrado
```



```
# PROGRAMA PRINCIPAL

if __name__ == "__main__":

    # Lista de datos de ejemplo

    datos = [34, 7, 23, 32, 5, 62]

    print(f"Lista original: {datos}")

    # Ordenamiento por burbuja (usando una copia para no modificar el original)

    ordenada_burbuja = datos.copy()

    bubble_sort(ordenada_burbuja)

    print(f"Lista ordenada con burbuja: {ordenada_burbuja}")

    # Ordenamiento por selección (otra copia de la lista original)

    ordenada_seleccion = datos.copy()

    selection_sort(ordenada_seleccion)

    print(f"Lista ordenada con selección: {ordenada_seleccion}")

    # Valor a buscar

    objetivo = 100

    # Búsqueda lineal sobre la lista original

    pos_lineal = busqueda_lineal(datos, objetivo)
```

## Trabajo Integrador - Programación 1

```

if pos_lineal != -1:

    print(f"Búsqueda lineal: {objetivo} encontrado en la posición {pos_lineal}")

else:

    print("No encontrado (con búsqueda lineal)")

# Búsqueda binaria sobre la lista ordenada con burbuja

pos_binaria = busqueda_binaria(ordenada_burbuja, objetivo)

if pos_binaria != -1:

    print(f"Búsqueda binaria: {objetivo} encontrado en la posición {pos_binaria}")

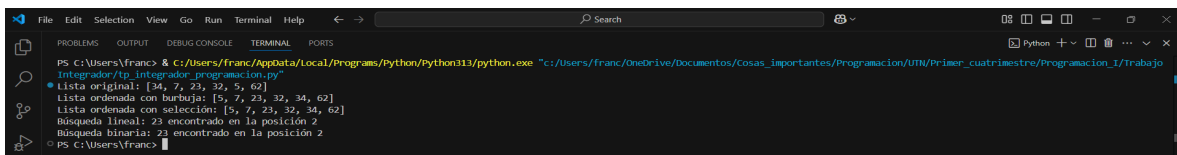
else:

    print("No encontrado (con búsqueda binaria)")

```

- Registro de resultados y validación de funcionalidad.

Cuando el elemento buscado (23) se encuentra en la misma posición tanto en la lista original como en la lista ordenada, la búsqueda lineal y la búsqueda binaria devuelven el mismo resultado. Esto ocurre porque el ordenamiento no altera la ubicación del elemento en este caso puntual.

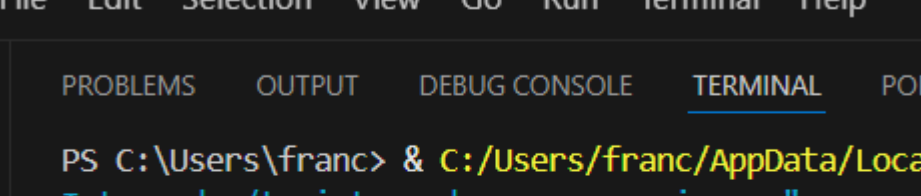


```

PS C:\Users\franc> & c:\Users\franc\AppData\Local\Programs\Python\Python311\python.exe "c:/Users/franc/OneDrive/Documentos/cosas_importantes/Programacion/UNT/Primer_cuatrimestre/Programacion_1/trabajo
Integrador/tp_integrador_programacion.py"
• Lista original: [34, 7, 23, 32, 5, 62]
Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
Búsqueda lineal: 23 encontrado en la posición 2
Búsqueda binaria: 23 encontrado en la posición 2
PS C:\Users\franc>

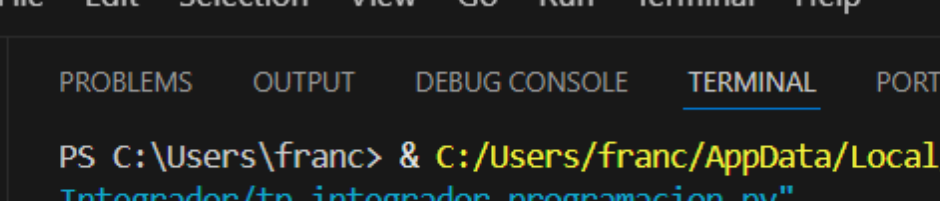
```

Cuando el elemento buscado (5) está en diferentes posiciones según si miramos la lista original o la lista ordenada, la búsqueda lineal y la búsqueda binaria pueden devolver resultados distintos. Esto resalta que, para que la búsqueda binaria funcione correctamente, es indispensable que la lista



```
PS C:\Users\franc> & C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/Integrador/tp_integrador_programacion.py
• Lista original: [34, 7, 23, 32, 5, 62]
  Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
  Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
  No encontrado (con búsqueda lineal)
  No encontrado (con búsqueda binaria)
○ PS C:\Users\franc>
```

esté previamente ordenada.



The screenshot shows a Windows terminal window with the following content:

```
PS C:\Users\franc> & C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/Python.exe C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/Integrador/tp_integrador_programacion.py
```

- Lista original: [34, 7, 23, 32, 5, 62]
- Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
- Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
- Búsqueda lineal: 5 encontrado en la posición 4
- Búsqueda binaria: 5 encontrado en la posición 0

```
PS C:\Users\franc>
```

Cuando el elemento buscado (100) no existe en la lista, de tal modo que no es posible determinar su índice

- Elaboración de este informe y preparación de anexos.

## Resultados obtenidos

A lo largo de nuestras pruebas, obtuvimos una gran cantidad de datos útiles que nos permitieron sacar conclusiones claras sobre el comportamiento de los algoritmos. A continuación, se resumen los resultados más relevantes:

### Ordenamiento:

#### Bubble sort:

Muy intuitivo y fácil de programar.

Hace muchas comparaciones e intercambios.

En listas pequeñas funcionó bien, pero en listas grandes fue lento.

Sensible al orden inicial: si la lista ya está casi ordenada, hace menos pasos.

#### Selection sort:

También fácil de entender.

Hace menos intercambios, pero siempre la misma cantidad de comparaciones.

Más predecible que bubble sort.

No mejora si la lista ya está ordenada.

### Búsqueda:

#### Búsqueda lineal:

Siempre funciona, sin importar cómo esté ordenada la lista.

Muy fácil de programar en comparación con otros algoritmos.

Poco eficiente en listas largas si el número está al final o no está.

## Trabajo Integrador - Programación 1

Búsqueda binaria:

Muy rápida y eficiente, sobre todo en listas grandes.

Requiere que la lista esté ordenada.

Ideal para búsquedas repetidas en la misma lista.

Resultados de pruebas adicionales:

En listas vacías, todos los algoritmos manejaron bien el caso sin errores.

En listas con un solo elemento, los algoritmos no generaron problemas.

En listas con números repetidos, tanto ordenamiento como búsqueda funcionaron correctamente.

En listas grandes (más de 100 elementos), las diferencias de eficiencia fueron notorias.

Se midió el tiempo de ejecución con `time.time()` en listas grandes, mostrando que la búsqueda binaria fue varias veces más rápida que la lineal.

Además, se tomaron capturas de pantalla del código y las ejecuciones. Esto ayudó a documentar y analizar los resultados, además de servir como prueba del funcionamiento correcto del programa.

Las imágenes mostraban:

- La lista original.
- El estado de la lista después de cada ordenamiento.
- La posición donde se encontró el número buscado (o mensaje si no estaba).

## Conclusión

A lo largo del trabajo se evidenció que los algoritmos de búsqueda y ordenamiento son piezas clave en la construcción de programas más eficientes y escalables. Pudimos observar y comprender que cada algoritmo tiene comportamientos distintos según el caso, lo que brinda la oportunidad de elegir correctamente cuál usar dependiendo de factores como el tamaño de los datos, si están o no ordenados y la necesidad de eficiencia.

Comprender cómo y cuándo aplicar un algoritmo te permite optimizar el rendimiento y ahorrar recursos y tiempo de ejecución.

En síntesis, dominar estas técnicas mejora la lógica y organización del código, y también nos permite desarrollar soluciones adaptadas a las necesidades reales de cada proyecto. Elegir bien es tan importante como saber programar

## Bibliografía


Material de la cátedra

Introducción Algoritmos de búsqueda y ordenamiento-Integrador:

[https://www.youtube.com/watch?v=u1OuRbx-\\_x4&ab\\_channel=Tecnicatura](https://www.youtube.com/watch?v=u1OuRbx-_x4&ab_channel=Tecnicatura)

Búsqueda: <https://www.youtube.com/watch?v=gJlQTq80llg>

Ordenamiento: <https://www.youtube.com/watch?v=xntUhrhtLaw>

 BusquedaOrdenamiento.ipynb

<https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing>

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

[https://www.apinem.com/complejidad-temporal-de-algoritmos/?utm\\_source=chatgpt.com](https://www.apinem.com/complejidad-temporal-de-algoritmos/?utm_source=chatgpt.com)

Python Software Foundation. (2024). Python (Versión 3.12). <https://www.python.org/>

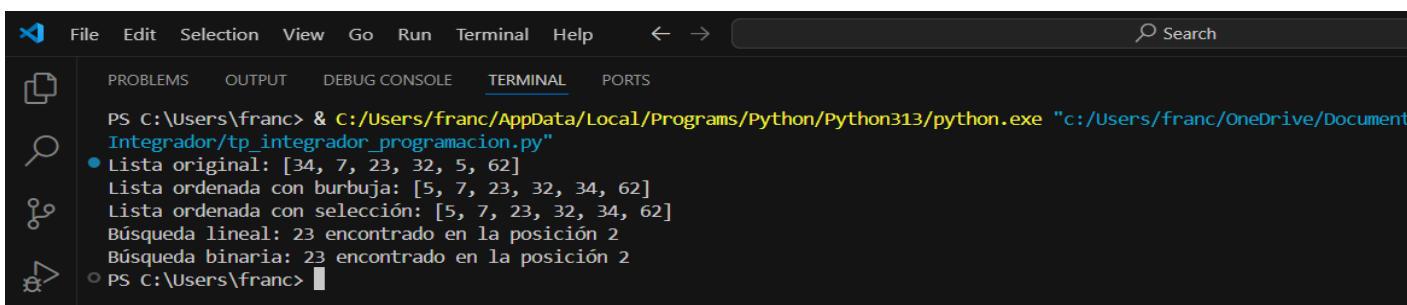
## Anexos

Link video en Youtube: <https://www.youtube.com/watch?v=THmEDZXfT8o>

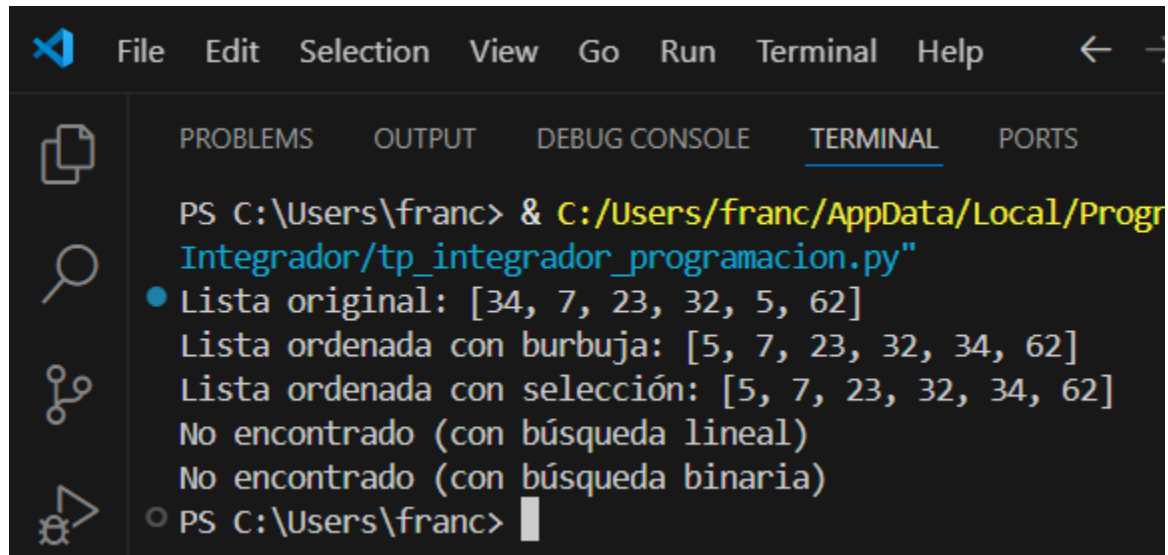
Link Presentación:

<https://gamma.app/docs/Programacion-1-doflya4nqwax7zu?mode=present#card-83hqp91ce3c95uq>

Capturas:



```
PS C:\Users\franc> & C:/Users/franc/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Franc/OneDrive/Document
Integrador/tp_integrador_programacion.py"
• Lista original: [34, 7, 23, 32, 5, 62]
Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
Búsqueda lineal: 23 encontrado en la posición 2
Búsqueda binaria: 23 encontrado en la posición 2
PS C:\Users\franc>
```

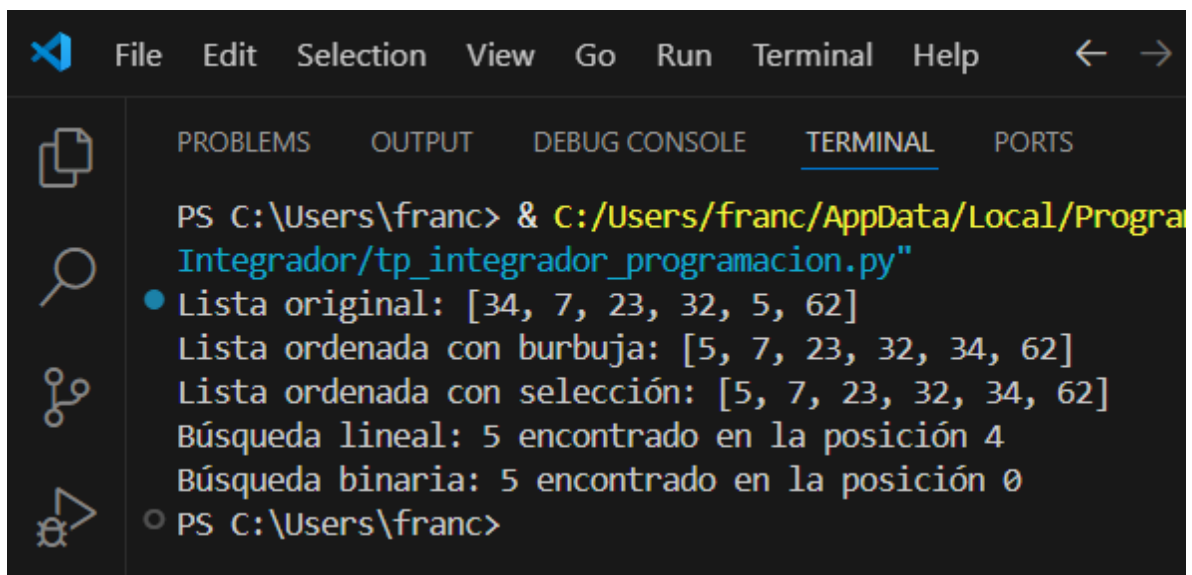


```

File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\franc> & C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Python.exe C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/python C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/tp_integrador_programacion.py
• Lista original: [34, 7, 23, 32, 5, 62]
  Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
  Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
  No encontrado (con búsqueda lineal)
  No encontrado (con búsqueda binaria)
○ PS C:\Users\franc>

```



```

File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\franc> & C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Python.exe C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/python C:/Users/franc/AppData/Local/Programs/Python/Python39-64/Scripts/tp_integrador_programacion.py
• Lista original: [34, 7, 23, 32, 5, 62]
  Lista ordenada con burbuja: [5, 7, 23, 32, 34, 62]
  Lista ordenada con selección: [5, 7, 23, 32, 34, 62]
  Búsqueda lineal: 5 encontrado en la posición 4
  Búsqueda binaria: 5 encontrado en la posición 0
○ PS C:\Users\franc>

```