Trabajo Práctico Integrado:

“Algoritmos de Búsqueda y Ordenamiento en Python”

Alumnos:

Bascuñan Martin Ivan / ivanbascunan7@gmail.com

Marcia Cristina Alvarez / marciaalavarez13@gmail.com

Materia:

Programación I

Profesor/a:

AUS Bruselario, Sebastián

Carbonari, Verónica

Fecha de Entrega: 09/06/2025

Indice

1. Introducción

De las tres temáticas expuestas por la cátedra, la selección del tema: “Búsqueda y el ordenamiento de Python” es un tema interesante y en la programación permitirá la vida mucho más fácil. Pero ¿De qué manera? Por ejemplo, supongamos que se quiere encontrar un libro de “Programación I” en la biblioteca de nuestra estantería, en este caso con los Algoritmos ayudará a encontrarlos mucho más rápido con “Búsqueda” que revisa libro por libro uno por uno, es como si alguien se preguntara en casa a cada miembro si lo ha visto. En cuanto a ordenamiento, supongamos que se tiene los libros no sólo en una estantería, sino por toda la casa dispersos, en este caso los Algoritmos de Ordenamiento permitirá a ponerlos a todos en orden. Por ejemplo, el “ordenamiento por burbuja” compara un elemento con su “vecino” siguiente y el elemento cambia de lugar si es que se encuentra en el orden incorrecto.
Por lo tanto, la selección del tema de “Búsqueda y Ordenamiento de Python” permite la inmersión como una herramienta poderosa para el desarrollo de aplicaciones eficientes, esto mejora a sumergirse en el mundo de la simplicidad, versatilidad y accesibilidad de Python. Enfatizando el beneficio de organizar datos, optimización de búsquedas y mejora de la experiencia del usuario. El lenguaje de programación de Python es excelente alternativa para el análisis de los datos, la computación exploratoria e interactiva, así como en la visualización de datos, convirtiéndolo en una alternativa sólida para las tareas de manipulación de datos.
Algunos de los objetivos que se presentaran en el siguiente TPI, son:
Comprensión solida de los conceptos fundamentales de los algoritmos de búsqueda y ordenamiento.
Desarrollo de al menos una de la aplicación técnica clave como la búsqueda lineal, binaria y principales métodos de ordenamiento.
Diseñar y desarrollar algoritmos sencillos para realizar búsquedas y ordenar datos mediante Python.
Capacidad de resolución a problemas computacionales mediante códigos claros y eficientes.
Los invito a explorar el siguiente “Trabajo Integrador de Programación I”.

2. Marco Teórico

A. “Búsqueda y Ordenamiento en Programación:

A.1 Búsqueda.

La búsqueda es una operación fundamental en Programación, utilizada para encontrar un elemento especifico dentro de un conjunto de datos. Es una tarea común en muchas aplicaciones, como base de datos, sistemas de archivos y algoritmos de inteligencia artificial.

Se presentan diferentes algoritmos de búsquedas, cada uno con sus propias ventajas y desventajas. Algunos de los algoritmos de búsqueda 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 lento para conjuntos de datos grandes. Es aquella que tal vez surgiría naturalmente por lógica.

- Búsqueda 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.

- Búsqueda de interpolación: Es un algoritmo de búsqueda que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Puede ser más eficiente que la búsqueda binaria para conjuntos de datos grandes con una distribución uniforme de valores.

- Búsqueda de hash: Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

La búsqueda es importante en programación ya que se utiliza en una amplia variedad de aplicaciones.

Algunos de los ejemplos de uso de la búsqueda en programación son:

- Búsqueda de palabras clave en un documento: Se puede utilizar un algoritmo de búsqueda para encontrar todas las apariciones de una palabra clave en un documento.

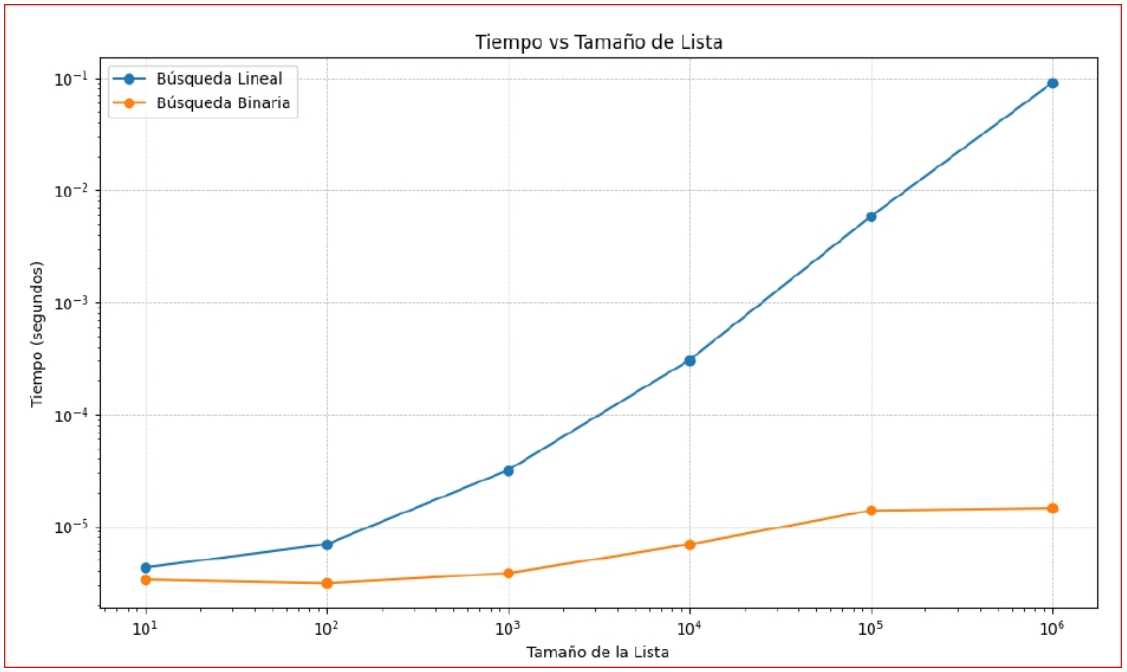
- Búsqueda de archivos en un sistema de archivos: Se puede utilizar un algoritmo de búsqueda para encontrar un archivo con un nombre específico en un sistema de archivos.

- Búsqueda de registros en una base de datos: Se puede utilizar un algoritmo de búsqueda para encontrar un registro con un valor específico en una base de datos.

- Búsqueda de la ruta más corta en un gráfico: Se puede utilizar un algoritmo de búsqueda para encontrar la ruta más corta entre dos nodos en un gráfico.

- Búsqueda de soluciones a problemas de optimización: Se puede utilizar un algoritmo de búsqueda para encontrar soluciones a problemas de optimización, como encontrar el valor máximo de una función.

Asimismo, al comprender los diferentes algoritmos de búsqueda y cómo utilizarlos, se puede mejorar el rendimiento y la eficiencia de sus programas.
Se observa en el siguiente gráfico, dos ejemplos: búsqueda lineal y de búsqueda binaria.



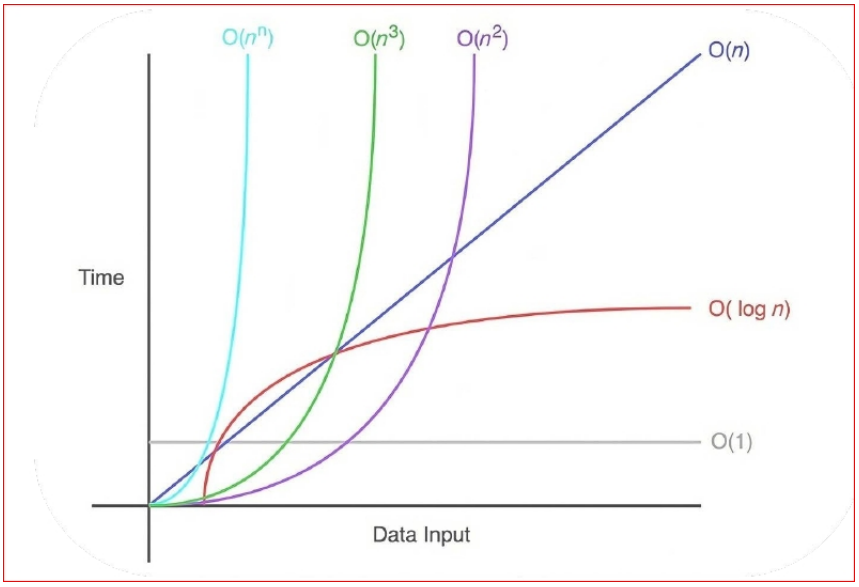
El tamaño de la lista tiene un impacto significativo en el **tiempo** que tardan los algoritmos de búsqueda en encontrar el objetivo. Cuanto más grande sea la lista, **más tiempo** tardará el algoritmo en encontrar el elemento deseado. Esto se debe a que el algoritmo debe comprobar cada elemento de la lista hasta encontrar el que busca.

A.1.1Complejidad de los algoritmos

La complejidad medida con O(n) es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.
Por ejemplo, un algoritmo con una complejidad de O(n) tardará el doble de tiempo en ejecutarse si el tamaño de la entrada se duplica.

La notación O(n) se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de O(n) tiempo en ejecutarse para cualquier entrada de tamaño n.

La complejidad medida con O(n) es una herramienta útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada.



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 significaría 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 significaría que, si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.

Tiempo de ejecución de los *algoritmos de búsqueda lineal y binaria* para diferentes tamaños de lista:

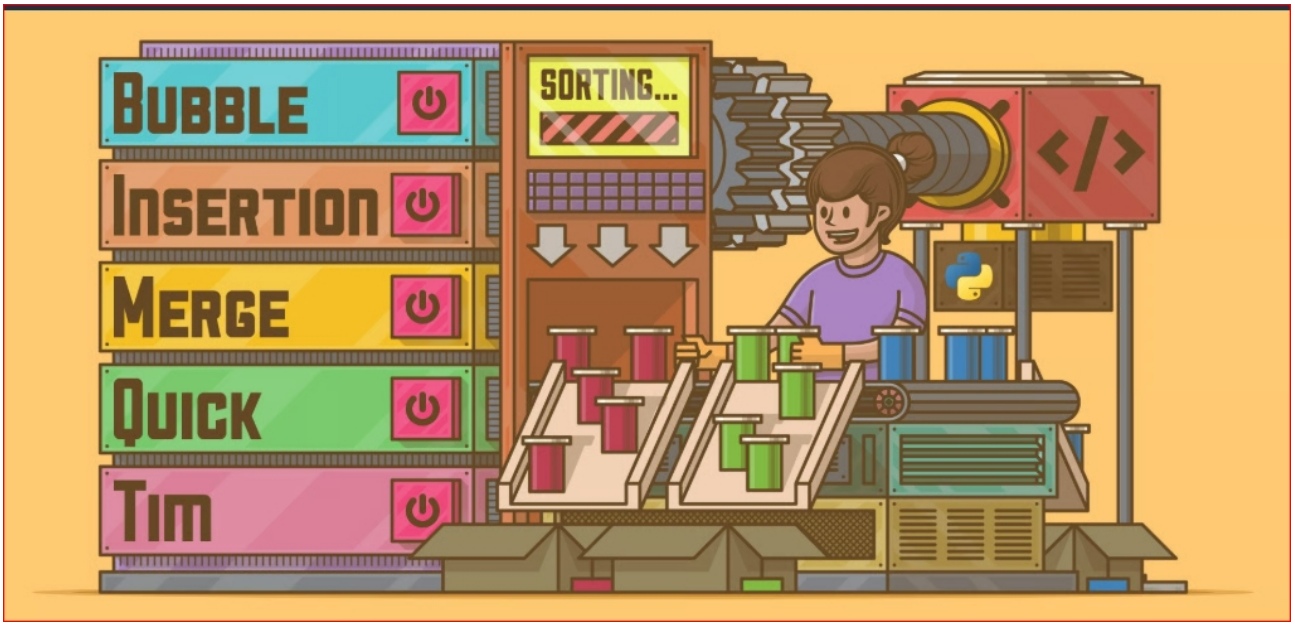
Tamaño de la lista	Búsqueda lineal	Búsqueda binaria
10	10	3
100	100	7
1000	1000	10
10000	10000	13
100000	100000	16

Lo que se observa en la tabla, el tiempo de ejecución de la *búsqueda lineal* aumenta mucho más rápidamente que el tiempo de ejecución de la *búsqueda binaria* a medida que aumenta el tamaño de la lista**. Esto hace que la *búsqueda binaria* sea mucho más eficiente para listas grandes.**

El tamaño de la lista es un factor importante para tener en cuenta al elegir un algoritmo de búsqueda:

- ✓ Si la lista es pequeña, es probable que la búsqueda lineal sea más eficiente.
- ✓ Si la lista es grande, es probable que la búsqueda binaria sea más eficiente.

A.2 Ordenamiento



El ordenamiento (ordenar) organiza los datos de acuerdo con un criterio, como por ejemplo 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.

Alguno de los beneficios de utilizar algoritmos de ordenamiento son:

- Búsqueda más eficiente: Una vez que los datos están ordenados, es mucho más fácil buscar un elemento específico. Esto se debe a que se puede utilizar la Búsqueda binaria, que es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal.

- Análisis de datos más fácil: Los datos ordenados pueden ser analizados más fácilmente para identificar patrones y tendencias. Por ejemplo, si se tienen datos sobre las ventas de una empresa, se pueden ordenar por producto, región o fecha para ver qué productos se venden mejor, en qué regiones se venden más productos o cómo cambian las ventas con el tiempo.

- Operaciones más rápidas: Muchas operaciones, como fusionar dos conjuntos de datos o eliminar elementos duplicados, se pueden realizar de manera más rápida y eficiente en datos ordenados

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

- Ordenamiento por burbuja: Es un algoritmo de ordenamiento simple y fácil de implementar. Funciona comparando cada elemento de la lista con el siguiente elemento y luego intercambiando los elementos si están en el orden incorrecto.

- Ordenamiento por selección: Es otro algoritmo de ordenamiento simple que funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista estén ordenados.

- Ordenamiento por inserción: Es un algoritmo de ordenamiento que funciona insertando cada elemento de la lista en su posición correcta en la lista ordenada.

- Ordenamiento rápido: Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes y luego ordenando cada parte de forma recursiva.

- Ordenamiento por mezcla: Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas.

En resumen, podemos decir que, la selección del algoritmo de ordenamiento adecuado depende de varios factores, como el tamaño de la lista, el tipo de datos y los requisitos de rendimiento:

I. Bubble Sort (Ordenamiento por burbuja):
   - ✓ Compara elementos adyacentes y los intercambia si están en el orden incorrecto.
   - ✓ Es fácil de entender, pero no muy eficiente para listas grandes.

II. Quick Sort (Ordenamiento rápido):
   - ✓ Divide y conquista: selecciona un "pivote" y organiza los elementos menores a un lado y los mayores al otro.
   - ✓ Es mucho más rápido que el Bubble Sort en la mayoría de los casos.

III. Selection Sort (Ordenamiento por selección):
   - ✓ Encuentra el elemento más pequeño de la lista y lo coloca al inicio. Repite el proceso con el resto de la lista.
   - ✓ Es más eficiente que el Bubble Sort, pero sigue siendo lento para listas grandes.

IV. Insertion Sort (Ordenamiento por inserción):
   - ✓ Construye la lista ordenada elemento por elemento, insertando cada nuevo elemento en la posición correcta.
   - ✓ Es eficiente para listas pequeñas o parcialmente ordenadas.

B. En resumen.

Importancia de los algoritmos.

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:

- Organización: Facilitan la estructuración y presentación de datos de manera coherente, simplificando su análisis y comprensión. Permiten trabajar con datos de forma más clara y estructurada.

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

- 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.

3. Caso Práctico

Objetivo

Simular un sistema de inventario que permita buscar un producto de forma rápida usando el algoritmo búsqueda binaria, el cual es más eficiente que una búsqueda tradicional si la lista está ordenada.

1. Importación de la librería

- **Se importa la librería random para generar números aleatorios que representen códigos de productos.**

```
import random
```

2. Función para generar el inventario

- **Crea una lista de 1000 códigos únicos entre 1000 y 1999 (ambos incluidos), luego los ordena.**
- **La búsqueda binaria solo funciona en listas ordenadas**

```
def generar_inventario():
#Crear una lista de 1000 códigos únicos ordenados (simulando el inventario)
    return sorted(random.sample(range(1000, 2000), 1000))
```

3. Función de búsqueda binaria

- **Se definen los límites iniciales: izquierda y derecha.**

```
def busqueda_binaria(inventario, codigo):
#Función de búsqueda binaria"""
    izquierda, derecha = 0, len(inventario) - 1
```

- Se calcula el índice del elemento central de la lista actual.

```
while izquierda <= derecha:
    medio = (izquierda + derecha) // 2
```

- Si el elemento central es igual al que buscamos, lo encontramos.

```
if inventario[medio] == codigo:
    return medio  # Código encontrado
```

- Si el código buscado es mayor que el central, nos vamos a la mitad derecha.

```
elif inventario[medio] < codigo:
    izquierda = medio + 1
```

- Si es menor, nos vamos a la mitad izquierda.

```
else:
    derecha = medio - 1
```

- Si no lo encuentra después de dividir la lista varias veces, devuelve -1.

```
return -1  # Código no encontrado
```

4. Generación del inventario

- Se genera la lista de códigos automáticamente al ejecutar el programa.

```
inventario = generar_inventario()
```

5. Mostrar información inicial

- Se muestran los primeros 20 códigos como ejemplo y el tamaño total del inventario (1000 productos).

```
# Mostrar parte del inventario
print("Inventario generado (primeros 20 códigos):", inventario[:20])
print("Total de códigos en el inventario:", len(inventario))
```

6. Solicitar un código al usuario

- El usuario escribe el código que desea buscar.

```
try:
    codigo_buscado = int(input("Ingrese el código de producto a buscar: "))
```

7. Ejecutar la búsqueda

- Se llama a la función y se guarda el resultado.

```
resultado = busqueda_binaria(inventario, codigo_buscado)
```

8. Mostrar resultados

- Se informa al usuario si el código fue encontrado o no.

```
print("Código buscado:", codigo_buscado)
if resultado != -1:
    print(f"El código {codigo_buscado} se encuentra en la posición {resultado} del inventario.")
else:
    print(f"El código {codigo_buscado} no se encuentra en el inventario.")
```

9. Manejo de errores

- Evita que el programa se rompa si el usuario escribe letras o caracteres no válidos.

```
except ValueError:
    print("Por favor, ingrese un número entero válido.")
```

Objetivo logrado

El caso práctico permitió implementar un sistema de búsqueda eficiente sobre un inventario simulado de productos, utilizando el algoritmo de búsqueda binaria. Se logró:

- Generar una lista de códigos de inventario único y ordenado.
- Aplicar correctamente la lógica de búsqueda binaria.
- Interactuar con el usuario mediante input () para realizar búsquedas personalizadas.
- Validar entradas con manejo de errores (try/except).

Aspectos que funcionaron correctamente

1. **Generación del inventario**:
   - Se generó una lista de 1000 códigos únicos con random.sample y se ordenó correctamente.
2. **Lógica de búsqueda binaria**:
   - Se implementó con éxito, dividiendo la lista en mitades hasta encontrar el código o determinar su ausencia.
3. **Interacción con el usuario**:
   - El usuario pudo ingresar un código de forma manual y recibir resultados claros y precisos.
4. **Manejo de errores**:
   - Se evitaron caídas del programa si el usuario ingresaba letras o caracteres no válidos.

Ordenamiento por Inserción

¿Qué hace?

Este algoritmo **ordena una lista comparando cada elemento con los anteriores** y ubicándolo en su lugar correcto. Es como ordenar cartas de una en una.

Paso a Paso del Código

1. Definición del algoritmo

- Se define una función que recibe una lista desordenada como parámetro.

```
def insercion(lista):
```

2. Recorrido desde el segundo elemento

- El bucle comienza en la posición 1 (segundo elemento), porque asumimos que el primer elemento ya está "ordenado".

```
for i in range(1, len(lista)):
```

3. Guardar el valor a insertar

- Se guarda el valor actual, que intentaremos insertar en la posición correcta.

```
insertar = lista[i]
```

4. Buscar hacia atrás en la lista

- Se toma la posición anterior al elemento actual.

```
anterior = i - 1
```

5. Comparar e intercambiar si es necesario

- Mientras el índice esté dentro de la lista y el elemento anterior sea mayor, se lo mueve hacia adelante.
- Esto **abre espacio** para que el valor actual se inserte en su lugar correcto.

```
while anterior >= 0 and lista[anterior] > insertar:
    lista[anterior + 1] = lista[anterior]
    anterior -= 1
```

6. Insertar el valor en su posición correcta

- Una vez que se encontró la posición, se inserta el valor guardado.

```
lista[anterior + 1] = insertar
```

```
# Este código implementa el algoritmo de ordenamiento por inserción.
if __name__ == "__main__":
    lista = [5, 2, 9, 1, 7, 6]
    print("Lista original:", lista)
    lista_ordenada = insercion(lista)
    print("Lista ordenada:", lista_ordenada)
```

Resultado esperado:

Ordenamiento por inserción

- antes de ordenar: [73, 6, 7, 21, 1, 10]
- después de ordenar: [1, 6, 7, 10, 21, 73]

¿Qué se logró con el caso práctico?

- Se implementó exitosamente el algoritmo de ordenamiento por inserción, que permite ordenar listas de manera ascendente.
- Se utilizó una lista desordenada de números enteros para validar su funcionamiento.

- Se comprendió el mecanismo interno del algoritmo: comparación hacia atrás e inserción en la posición adecuada.
- Se reforzó el conocimiento sobre estructuras de control como bucles for y while.

Aspectos que funcionaron correctamente

- El algoritmo ordenó correctamente la lista [73, 6, 7, 21, 1, 10], dando como resultado [1, 6, 7, 10, 21, 73].
- No se presentaron errores de ejecución.
- El código es legible, modular y fácil de probar con diferentes datos.

Video Trabajo Integrador:

TPI-Programación1.mp4

Enlace:
https://drive.google.com/file/d/1PIAzvJcLTqZhvfrqgaeJHMrHZeONuf3A/view?usp=drive_web

6. Conclusiones

Concluimos mediante este documento evaluativo, que se ha demostrado la importancia teórica y técnica de los algoritmos de ordenamiento y búsqueda en el

ámbito de la programación. Estos algoritmos son esenciales para la manipulación y búsquedas eficientes de datos.

Ejemplo de aplicación incluyen la organización de correo junto con las empresas de mensajería, se ordena el correo, paquetes por códigos postales para una entrega eficiente. Otro ejemplo, sería los estudiantes de una universidad que se ordenan por sus apellidos, por los números de legajos o bien por méritos para el ingreso de una plataforma on line. La ordenación y búsqueda son tareas frecuentes en el procesamiento de datos, tanto en computadoras como en sistemas de información.

Por lo tanto, una de las principales conclusiones que se puede extraer del TPI es que, en primer lugar, un algoritmo de ordenamiento sea el más rápido para cualquier conjunto de datos a ordenar debe ser tanto de la jerarquía de memoria del computador como de la forma de ordenarse de los elementos. En cuanto a los algoritmos de búsqueda son esenciales ya que, ahorra demasiado tiempo en la búsqueda de información, por lo cual es de vital importancia conocer cómo funcionan, como se pudo apreciar en este trabajo integrador.

7. Bibliografía

Introducción a Python para geociencias (Villie Morocho) *versión On-line* ISSN 2663-3981*versión impresa* ISSN 0080-2085https://www.scielo.org.mx/scielo.php?pid=S2663-39812024000200137&script=sc

Uso de Python para el análisis de datos aplicado en la Investigación. REVISTA INCAING ISSN 2448 9131.
https://www.researchgate.net/publication/366157405_Uso_de_Python_para_el_analisis_de_datos_aplicado_en_la_investigacion

www.Python.com/google/https://docs.python.org/es/3.13/contents.html
Documentos y Librería

Visual Studio Codehttps://code.visualstudio.com/

¿Qué es el método sort() en Python?
https://www.freecodecamp.org/espanol/news/ordenar-listas-en-python-como-ordenar-por-descendente-o-ascendente/