Trabajos Integradores – Programación I

Datos Generales

- **Título del trabajo**: Algoritmos de Búsqueda y Ordenamiento en Python”
- **Alumnos**: Matias Perez y Marcos Pousada
- **Materia**: Programación I
- **Profesor**: Nicolás Quirós
- **Tutor**: Carla Bustos
- **Comisión**: 19
- **Fecha de Entrega**: 09/06/2025

Índice

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

---

1. Introducción

Se eligió este tema para entender más en profundidad cómo funcionan estos algoritmos, cuándo aplicarlos y cuáles son sus ventajas y desventajas según el contexto, ayuda a escribir código más eficiente, ordenado y escalable. Además, son ampliamente laborales.

En la programación los algoritmos de búsqueda y ordenamiento son herramientas fundamentales para gestionar información de manera eficiente. Permiten organizar, acceder y manipular datos de forma óptima, lo cual es esencial en una gran variedad de aplicaciones prácticas.

Los objetivos del trabajo es usar el conocimiento obtenido en el cursado de la

materia, con el fin de implementar un programa en Python que simule una situación real: registrar la nota de un alumno, previa búsqueda de su nombre en una lista de estudiantes. Para ello, se aplicarán técnicas de ordenamiento y búsqueda, evaluando su funcionamiento y utilidad práctica. A través de este desarrollo, se busca comprender el funcionamiento de algoritmos de ordenamiento y búsqueda, y aplicar estructuras de datos apropiadas como listas y diccionarios.

---

2. Marco Teórico

Búsqueda

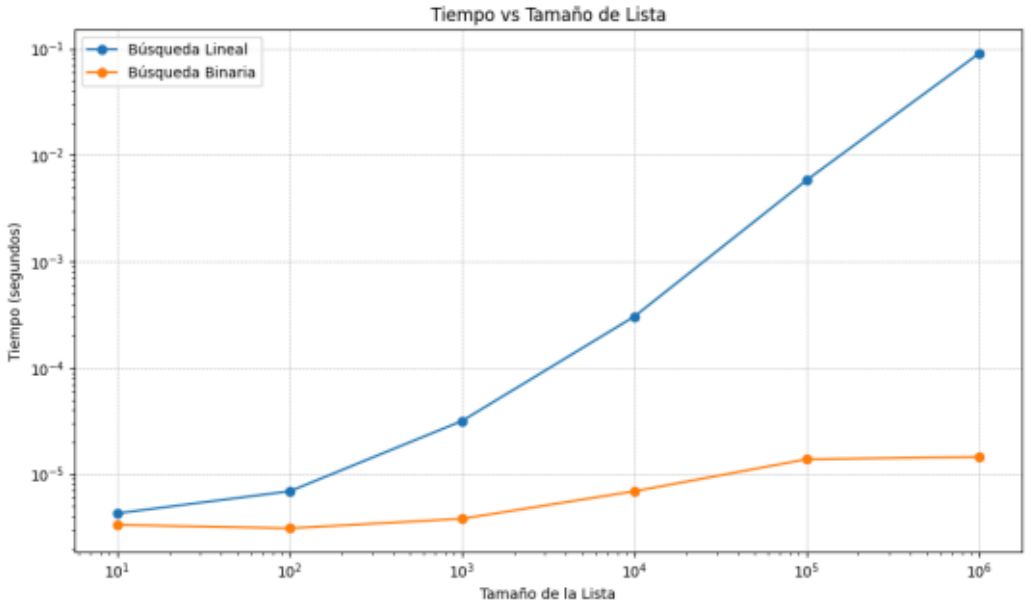
Un algoritmo de búsqueda permite encontrar un elemento específico dentro de una estructura de datos. Existen diferentes algoritmos de búsqueda, cada uno con sus propias ventajas y desventajas. En el trabajo se abordará las siguientes búsquedas :

- **Búsqueda linea**l: 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.

- **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.

Comparativa de algoritmos de búsqueda:

Algoritmo	Orden requerido	Complejidad	Ventaja principal
Búsqueda lineal	No	O(n)	Simplicidad
Búsqueda binaria	Sí	O(log n)	Eficiencia con datos ordenados



Ordenamiento

El ordenamiento organiza los datos de acuerdo a un criterio, como de menor a mayor o alfabéticamente.

Algunos de los beneficios de utilizar algoritmos de ordenamiento incluyen:

- 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

Algunos de los algoritmos de ordenamiento más comunes incluyen:

1. 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.

2. 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.

3. 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.

4. 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.

3. Comparación entre Algoritmos de Ordenamiento

Algoritmo	Mejor Caso	Peor Caso	Estable	Complejidad Espacial	Características principales
Bubble Sort	O(n)	$O(n^2)$	Sí	O(1)	Fácil de entender e implementar.
Insertion Sort	O(n)	$O(n^2)$	Sí	O(1)	Eficiente en listas pequeñas o casi ordenadas.
Selection Sort	$O(n^2)$	$O(n^2)$	No	O(1)	Realiza menos intercambios que Bubble Sort.
Quick Sort	O(n log n)	$O(n^2)$	No	O(log n)	Muy eficiente en la práctica, pero no estable.

Fuentes clave:

-Cátedra de Programación I. (2025). Tecnicatura Universitaria en Programación a Distancia.

-Python Software Foundation (https://docs.python.org)

4. Caso Práctico

Se desarrolló un programa en Python que solicita el nombre y la nota(entre 0 y 10) de un alumno, obtenido en un parcial de la universidad, de una lista inicial desordenada de 500 estudiantes, luego busca si está en la lista y si existe se guarda su nota.
Se realiza un ordenamiento de la lista para realizar una comparativa entre la búsqueda lineal y binaria

Algoritmo Final después de realizar pruebas

```
# Funcion que genera una lista de nombres únicos de alumnos y la desordena aleatoriamente
def generar_lista_alumnos(cantidad):
    import random
    lista = [f"Alumno{i:03}" for i in range(1, cantidad + 1)]
    random.shuffle(lista)
    return lista

# Funcion ordena la lista alfabéticamente usando el algoritmo Bubble Sort
def ordenar_lista(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista

# Funcion realiza búsqueda binaria en una lista ordenada y retorna la posición del elemento
def busqueda_binaria(lista, objetivo):
    inicio, fin = 0, len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

import random
random.seed(42)  # Fijamos la semilla para obtener siempre la misma lista desordenada
alumnos = generar_lista_alumnos(500)  # Crear y desordenar 500 nombres de
```

```
alumnos
alumnos = ordenar_lista(alumnos)  # Ordenar alfabéticamente la lista de alumnos
notas = {}  # Diccionario para registrar las notas asignadas


# Solicita nombre del alumno
nombre = input("Ingrese el nombre del alumno (ej. Alumno001): ")
valido = False  # Controla que se ingrese una nota válida
while not valido:
    entrada = input("Ingrese la nota del alumno (entre 0 y 10): ")
    try:
        nota = float(entrada)  # Convierte la entrada a número decimal
        if 0 <= nota <= 10:
            valido = True
        else:
            print("La nota debe estar entre 0 y 10.")
    except ValueError:
        print("La nota debe ser un número decimal válido.")

indice_binaria = busqueda_binaria(alumnos, nombre)

indice = indice_binaria  # Se usa la posición devuelta por la búsqueda binaria
if indice != -1:
    notas[nombre] = nota
    print(f"Nota registrada para {nombre}: {nota}")  # Confirmación de registro
else:
    print("El alumno no se encuentra en la lista.")  # Mensaje si no se encuentra el
alumno
```

5.Metodología Utilizada

Investigación previa: Se analizo los diferentes temas propuestos para el trabajo, consultaron materiales de apoyo proporcionados por la cátedra y documentación oficial de Python. Esto permitió comprender a fondo los distintos temas y elegir un tema teniendo en cuenta su funcionamiento y la utilidad de los algoritmos seleccionados.

Etapas de diseño y prueba: Primero se definió el problema a resolver y se decidió trabajar con una lista de alumnos predefinida. Luego se desarrolló la función de ordenamiento utilizando Bubble Sort y posteriormente se incorporó la búsqueda binaria. Se realizaron múltiples pruebas para validar el comportamiento esperado ante distintas entradas (correctas, incorrectas, existentes y no existentes).

- Herramientas y recursos utilizados (IDE, librerías, control de versiones, etc.).
- Trabajo colaborativo (reparto de tareas si se realizó en grupo).

Herramientas y recursos utilizados:

- IDE: Visual Studio Code
- Lenguaje: Python
- Control de versiones: Git (repositorio privado de práctica)
- Sistema operativo: Windows/Linux
- Medios de Comunicación: Discord y WhatsApp

Trabajo colaborativo: El desarrollo fue realizado de manera grupal, entre los integrantes en donde la programación se dividió en funciones para realizar una programación modular, validación de datos y comentarios en el código. Mientras que la redacción del informe se dividió en secciones, realizando Matias de la 1 a la 3 y Marcos de la 4 a la 8. La edición de video fue realizada en conjunto. La comunicación constante facilitó la integración fluida del trabajo.

.

5.Resultados Obtenidos

- El algoritmo ordenó correctamente la lista de nombres mediante el uso de Bubble Sort.
- La búsqueda binaria localizó de manera precisa los nombres buscados, siempre que existieran en la lista.
- El programa validó correctamente los datos ingresados por el usuario, rechazando valores inválidos y manteniendo la integridad de los registros.
- Se observó que, aunque Bubble Sort no es el más eficiente, cumplió su propósito en listas moderadas como la utilizada (500 elementos).
- Se logró un manejo robusto de errores de entrada, tanto en nombres inexistentes como en notas fuera del rango permitido.
- El trabajo colaborativo y el uso de herramientas como Discord y WhatsApp mejoraron la coordinación y resolución de dudas durante el desarrollo.
- Se cumplieron los objetivos planteados inicialmente, integrando correctamente teoría y práctica.

Casos de prueba realizados:

- Búsqueda de alumnos ubicados al inicio, mitad y final de la lista.

- Ingresos válidos e inválidos de notas (letras, valores fuera de rango, vacíos).
- Simulación de múltiples registros secuenciales para verificar acumulación de datos en el diccionario.

Evaluación de rendimiento:

Comparación de Pruebas Realizadas medidas en tiempo entre búsqueda lineal y búsqueda binaria en una lista ordena.

Búsqueda del primer elemento:

```
Ingrese el nombre del alumno (ej. alumno001): alumno001
Ingrese la nota del alumno (entre 0 y 10): 3
El alumno no se encuentra en la lista.
Tiempo de búsqueda binaria: 0.00001600 segundos
Tiempo de búsqueda lineal: 0.00001700 segundos
```

En este caso que se busca el primer elemento de la lista, el mejor caso para la búsqueda lineal, se puede observar que el tiempo de la búsqueda binaria es menor a la búsqueda lineal

```
Ingrese el nombre del alumno (ej. alumno001): alumno250
Ingrese la nota del alumno (entre 0 y 10): 8
El alumno no se encuentra en la lista.
Tiempo de búsqueda binaria: 0.00001570 segundos
Tiempo de búsqueda lineal: 0.00002650 segundos
```

En este caso donde se busca un elemento medio, se puede observar que el tiempo de la búsqueda binaria es menor a la búsqueda lineal

```
Ingrese el nombre del alumno (ej. alumno001): alumno500
Ingrese la nota del alumno (entre 0 y 10): 9
Nota registrada para alumno500: 9.0
Tiempo de búsqueda binaria: 0.00001330 segundos
Tiempo de búsqueda lineal: 0.00003910 segundos
```

En este caso donde se busca el último elemento, se puede observar que el tiempo de la búsqueda binaria es menor a la búsqueda lineal, denotando que para el caso más la diferencia eficacia a favor de la búsqueda binaria.

Detalla qué se logró con el caso práctico, qué aspectos funcionaron correctamente y qué dificultades se presentaron.

Se pueden incluir:

- Casos de prueba realizados.
- Errores corregidos.
- Evaluación de rendimiento (si aplica, por ejemplo, comparar el tiempo de ejecución entre algoritmos).
- Enlace a repositorio si el trabajo está subido a GitHub u otra plataforma.

6.Conclusiones

En la realización del trabajo se aprendió a diseñar algoritmos simples pero funcionales para resolver problemas concretos. Además, se fortalecieron habilidades de programación estructurada, validación de entradas y toma de decisiones técnicas fundamentadas.

La utilidad que tiene el tema trabajado para la programación o para otros proyectos, es que los algoritmos de búsqueda y ordenamiento son esenciales en la mayoría de los sistemas informáticos. Este conocimiento permite desarrollar soluciones eficientes, aplicables en bases de datos, motores de búsqueda, interfaces gráficas, y más.

Una de las dificultades fue decidir qué algoritmo utilizar en cada etapa, lo que se resolvió con pruebas prácticas acompañadas del marco teorioco. También se enfrentaron validaciones de entradas numéricas, que se resolvieron con estructuras de control apropiadas para asegurar la robustez del programa.

Posibles mejoras o extensiones futuras:

- Incorporar algoritmos de ordenamiento más eficientes como Merge Sort o Quick Sort.
- Agregar una lista de datos mediante archivos o bases de datos.
- Implementar una interfaz gráfica para mejorar la interacción con el usuario.

7.Bibliografía

- Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/
- Sweigart, A. (2019). *Automate the Boring Stuff with Python*. No Starch Press.
- Cátedra de Programación I. (2025). *Trabajo Integrador* . Tecnicatura Universitaria en Programación a Distancia.

8.Anexos

- Link al video explicativo.

  https://youtu.be/27B5rEiD5MY

- Link Código completo GITHUB.

  https://github.com/matiasplm/Trabajo-Integrador-Programacion1.git