**Universidad Tecnológica Nacional - Facultad Regional San Nicolás.**

**Trabajo Práctico Grupal.**

**Materia:** Programación I.

**Tema:**

***Algoritmos de Búsqueda y Ordenamiento: Análisis de Eficiencia y Aplicación Práctica.***


**Alumnos - mails:**

Santiago Raúl Salinas - *aralargg@gmail.com*

Emiliano Víctor Vallejos - *emilianovallejos1986gmail.com*


**Profesor:**

Sebastian Bruselario.

**Tutora:**

Flor, Gubiotti.

**Fecha de entrega:**

09/06/2025.

**Índice:**

## 1. Introducción

En la vasta y creciente área de la informática, la eficiencia en el manejo de grandes volúmenes de datos se ha convertido en un pilar fundamental para el desarrollo de software robusto y escalable. Dentro de este contexto, los algoritmos de búsqueda y ordenamiento emergen como herramientas esenciales que permiten organizar, encontrar y manipular información de manera óptima. Desde la gestión de bases de datos y sistemas de archivos hasta el desarrollo de aplicaciones web y la inteligencia artificial, la capacidad de procesar datos de forma rápida y precisa es directamente proporcional al rendimiento y la fiabilidad de cualquier sistema.

La elección de este tema radica en su importancia universal en la programación. Comprender cómo funcionan estos algoritmos y cuándo aplicar cada uno es crucial para cualquier futuro profesional en el campo de la programación, ya que impacta directamente en el tiempo de ejecución y la optimización de recursos. A menudo, una pequeña mejora en la eficiencia de un algoritmo puede traducirse en ahorros significativos de tiempo y recursos computacionales en aplicaciones del mundo real.

El presente trabajo se propone alcanzar los siguientes objetivos:

- Investigar y fundamentar los conceptos teóricos de los algoritmos de búsqueda lineal y binaria, incluyendo sus implementaciones iterativa y recursiva, así como los principios básicos del ordenamiento.
- Analizar la eficiencia de estos algoritmos mediante la notación de la complejidad temporal (O(n)) en sus diferentes casos.
- Desarrollar un caso práctico en Python que permita comparar empíricamente el rendimiento de los algoritmos de búsqueda lineal y binaria (en sus versiones iterativa y recursiva) en una simulación de base de datos, con diferentes volúmenes de información.
- Extraer conclusiones claras sobre la aplicabilidad y las ventajas de cada algoritmo y sus implementaciones en función del tamaño y la naturaleza de los datos.

## 2. Marco Teórico

El manejo eficiente de colecciones de datos es un aspecto central en la ciencia de la computación. Dos de las operaciones más comunes y estudiadas en este ámbito son la búsqueda y el ordenamiento, cada una con un conjunto de algoritmos diseñados para optimizar su rendimiento bajo diversas condiciones.

## 3. Concepto de Búsqueda y Ordenamiento

La búsqueda es la operación que consiste en encontrar un elemento específico dentro de un conjunto de datos. Su relevancia se manifiesta en innumerables aplicaciones, desde la recuperación de información en un motor de búsqueda hasta la localización de un registro particular en una base de datos.

Por otro lado, el ordenamiento es el proceso de organizar un conjunto de datos según un criterio predefinido (por ejemplo, de menor a mayor o alfabéticamente). El ordenamiento es, a menudo, una precondición fundamental para que ciertos algoritmos de búsqueda alcancen su máxima eficiencia.

Ambos tipos de algoritmos son vitales por su impacto en la eficiencia de los programas, permitiendo optimizar el tiempo de ejecución al manejar grandes volúmenes de información, y por su contribución a la organización y escalabilidad de los datos, facilitando su análisis y manipulación.

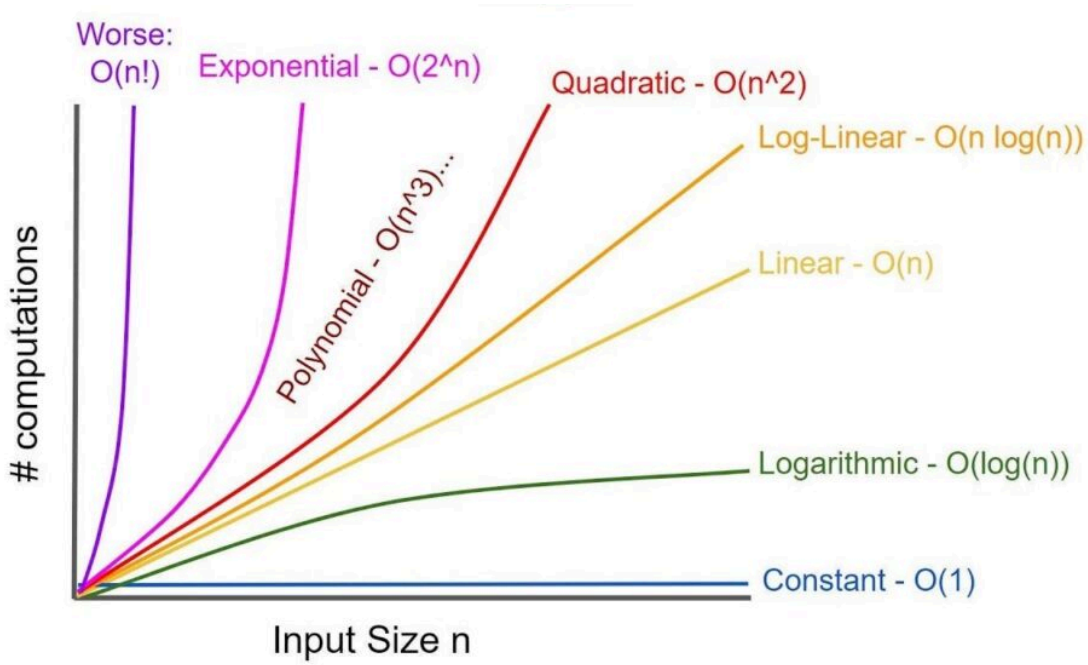
## 4. Análisis de la Eficiencia de Algoritmos (Notación O Grande -*O Big*-)

La complejidad temporal de un algoritmo es una medida de la cantidad de tiempo que tarda en ejecutarse en función del tamaño de su entrada. La notación O Grande (O(n)) se utiliza para describir el comportamiento del peor caso de un algoritmo, proporcionando una estimación de cómo crecerá su tiempo de ejecución a medida que aumenta el tamaño de los datos de entrada (n).

- Una complejidad de O(1) indica tiempo constante (independiente del tamaño de la entrada).

- Una complejidad de O(logn) indica tiempo logarítmico (crecimiento muy lento).
- Una complejidad de O(n) indica tiempo lineal (crecimiento proporcional al tamaño de la entrada).
- Una complejidad de O(n2) indica tiempo cuadrático (crecimiento mucho más rápido, a menudo para algoritmos ineficientes en datos grandes).

Imagen de Time vs Data Input Complexity Graph




Como se observa en el gráfico anterior, la diferencia en el crecimiento del tiempo de ejecución entre las distintas complejidades es abismal, especialmente a medida que el tamaño de la entrada “n” aumenta. Los algoritmos con complejidades logarítmicas o lineales son preferibles para manejar grandes volúmenes de datos.

## 5. Algoritmos de Búsqueda

### *5.1. Búsqueda Lineal (Sequential Search)*

- Concepto: Es el algoritmo de búsqueda más simple. Recorre cada elemento del conjunto de datos de forma secuencial, uno por uno, desde el principio hasta el final de la lista, comparándolo con el elemento deseado.

- Implementación en Python: Generalmente, se implementa con un bucle for o while que itera sobre los elementos de una lista.

```python
def busqueda_lineal(lista, objetivo):
    """
    Busca un objetivo en una lista de forma lineal.
    Este algoritmo recorre cada elemento de la lista uno por uno.
    Retorna el índice del elemento si lo encuentra, o -1 en caso contrario.
    """
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i  # Retorna el índice tan pronto como encuentra el objetivo
    return -1  # Retorna -1 si el bucle termina sin encontrar el objetivo
```

- Características:
    - Simplicidad: Muy fácil de entender e implementar.
    - No requiere ordenamiento: Funciona en listas ordenadas o desordenadas.
    - Eficiencia: Es la opción más adecuada para listas muy pequeñas.
- Complejidad Temporal (O(n)):
    - Mejor Caso: O(1) (el elemento buscado es el primero de la lista).
    - Peor Caso: O(n) (el elemento buscado está al final de la lista o no está presente, requiriendo revisar todos los elementos).
    - Caso Promedio: O(n) (en promedio, se recorre la mitad de los elementos).

### *5.2. Búsqueda Binaria (Binary Search)*

- Concepto: Es un algoritmo de búsqueda altamente eficiente que funciona exclusivamente en conjuntos de datos previamente ordenados. Divide repetidamente la lista (o la porción restante de ella) en dos mitades, y compara el elemento buscado con el elemento central de la sublista actual. Si el elemento buscado es menor, se descarta la mitad derecha; si es mayor, se descarta la mitad izquierda. Este proceso se repite hasta encontrar el elemento o determinar que no está en la lista.
- Requisito fundamental: La lista debe estar ordenada. Si la lista está desordenada, la búsqueda binaria podría dar resultados incorrectos o no encontrar el elemento aunque esté presente.

#### *5.2.1. Implementaciones de la Búsqueda Binaria en Python:*

La búsqueda binaria puede implementarse de dos formas principales: iterativa o recursiva. Ambas logran la misma eficiencia en términos de complejidad temporal, pero difieren en su uso de memoria y estilo de codificación.

#### *5.2.2. Implementación Iterativa*

- Concepto: Utiliza un bucle (while o for) para reducir progresivamente el espacio de búsqueda ajustando los índices de inicio y fin de la sublista.
- Ventajas:
    - Eficiencia de Memoria: Utiliza espacio constante O(1), ya que no hay un stack de llamadas recursivas.
    - Rendimiento Óptimo: Evita el costo adicional de las llamadas a funciones recursivas, siendo ligeramente más rápida en la práctica para listas muy grandes.
    - Estabilidad: No hay riesgo de "stack overflow" (desbordamiento de pila) en listas extremadamente grandes.
- Implementación en Python:

```python
def busqueda_binaria_iterativa(lista, objetivo):
    """
    Busca un objetivo en una lista ORDENADA de forma binaria (iterativa).
    Este método divide repetidamente la lista a la mitad para reducir el espacio de búsqueda.
    Requiere que la lista esté previamente ordenada para funcionar correctamente.
    Retorna el índice del elemento si lo encuentra, o -1 si el objetivo no está en la lista.
    """
    izquierda, derecha = 0, len(lista) - 1

    # El bucle continúa mientras el rango de búsqueda sea válido
    while izquierda <= derecha:
        # Calcula el índice del elemento central. Se usa esta fórmula para evitar
        # un posible desbordamiento de enteros (overflow) en listas extremadamente grandes.
        medio = izquierda + (derecha - izquierda) // 2

        # Compara el elemento central con el objetivo
        if lista[medio] == objetivo:
            return medio  # Elemento encontrado, retorna su índice.
        elif lista[medio] < objetivo:
            # Si el elemento central es menor que el objetivo, significa que el objetivo
            # (si existe) debe estar en la mitad derecha de la lista. Se ajusta el límite izquierdo.
            izquierda = medio + 1
        else:
            # Si el elemento central es mayor que el objetivo, el objetivo (si existe)
            # debe estar en la mitad izquierda. Se ajusta el límite derecho.
            derecha = medio - 1

    return -1  # El objetivo no fue encontrado después de agotar el espacio de búsqueda.
```

### *5.2.3. Implementación Recursiva*

- Concepto: La función se llama a sí misma repetidamente, resolviendo un subproblema más pequeño en cada llamada hasta alcanzar un caso base (cuando el elemento es encontrado o el espacio de búsqueda se agota).
- Ventajas:
    - Elegancia Matemática: El código suele ser más conciso y refleja directamente la definición matemática del algoritmo, facilitando la comprensión conceptual.
- Consideraciones:
    - Uso de Memoria: Utiliza el *stack* de llamadas para mantener el estado de cada recursión, lo que puede consumir más memoria (O(logn) en el peor caso de espacio) y, en casos extremos (listas gigantes), podría llevar a un "stack overflow" si el límite de recursión de Python no es ajustado.
    - Rendimiento: El *overhead* de las llamadas a función recursivas puede afectar ligeramente el rendimiento en comparación con la versión iterativa.
- Implementación en Python:

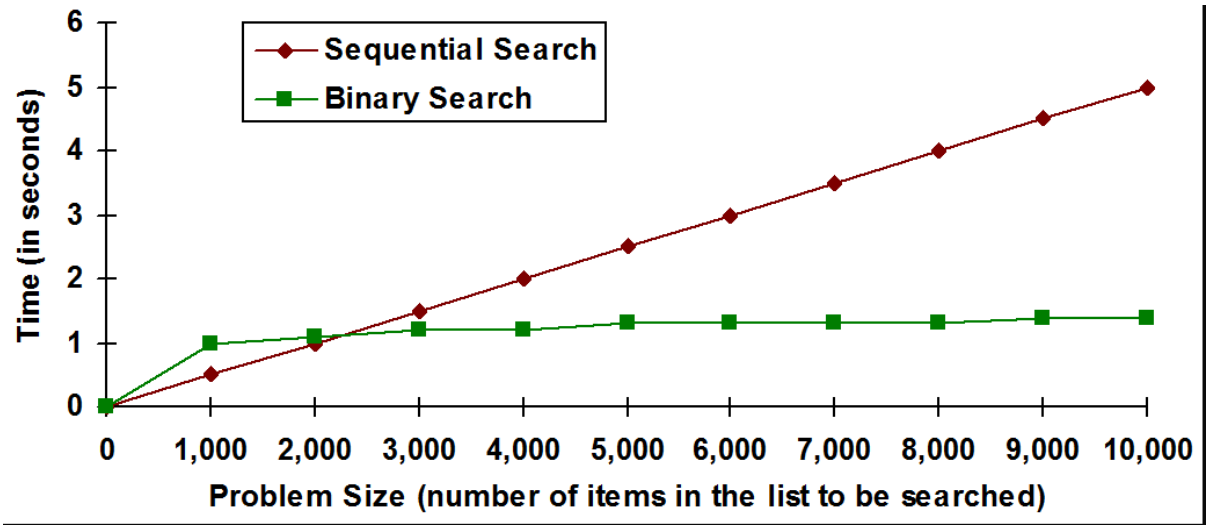
```python
def busqueda_binaria_recursiva(lista, objetivo, izquierda=0, derecha=None):
    """
    Busca un objetivo en una lista ORDENADA de forma binaria (recursiva).
    Retorna el índice del elemento si lo encuentra, -1 en caso contrario.
    """
    # Inicializa el puntero derecho si es la primera llamada a la función (derecha es None).
    if derecha is None:
        derecha = len(lista) - 1

    if izquierda > derecha:
        return -1     # Caso base de la recursión: si el rango de búsqueda es inválido, el elemento no está presente.

    medio = izquierda + (derecha - izquierda) // 2 # Calcula el índice del elemento central del subrango actual.

    if lista[medio] == objetivo:     # Compara el elemento central con el objetivo.
        return medio  # Caso base: el elemento ha sido encontrado.
    elif lista[medio] < objetivo:
        # Si el elemento central es menor que el objetivo, se realiza una llamada recursiva
        # para buscar en la mitad derecha del subrango.
        return busqueda_binaria_recursiva(lista, objetivo, medio + 1, derecha)
    else:
        # Si el elemento central es mayor que el objetivo, se realiza una llamada recursiva
        # para buscar en la mitad izquierda del subrango.
        return busqueda_binaria_recursiva(lista, objetivo, izquierda, medio - 1)
```

- Complejidad Temporal (O(logn)):
    - Mejor Caso: O(1).
    - Peor Caso: O(logn).
    - Caso Promedio: O(logn).

## 6. Comparación de Eficiencia (Búsqueda Lineal vs. Binaria)

La diferencia en la eficiencia entre la búsqueda lineal y la binaria es crucial a medida que el tamaño de los datos aumenta.

Imagen de Time vs List Size Search Graph




La tabla a continuación, extraída del material de estudio, ilustra esta diferencia:

| Tamaño de la lista | Búsqueda lineal (comparaciones/pasos aprox.) | Búsqueda binaria (comparaciones/pasos aprox.) |
|---|---|---|
| 10 | 10 | 3 |
| 100 | 100 | 7 |
| 1000 | 1000 | 10 |
| 10000 | 10000 | 13 |
| 100000 | 100000 | 16 |

Como se puede observar, el tiempo de ejecución (o número de comparaciones) de la búsqueda lineal crece de forma directamente proporcional al tamaño de la lista, mientras que el de la búsqueda binaria aumenta de manera logarítmica, lo que la hace exponencialmente más eficiente para conjuntos de datos grandes. Sin embargo, es vital recordar que

la búsqueda binaria requiere que la lista esté ordenada, lo que podría añadir un costo de preprocesamiento si los datos no están ya en orden.

## 7. Búsqueda en una Base de Datos Simple

A continuación, se presenta el código Python desarrollado para la simulación del caso práctico, incluyendo la implementación de los algoritmos de búsqueda lineal, binaria iterativa y binaria recursiva, junto con el script de ejecución y medición de tiempos.

El problema principal de este caso práctico es **simular la búsqueda de elementos en una base de datos simple y comparar la eficiencia de diferentes algoritmos de búsqueda: la búsqueda lineal y las dos implementaciones de la búsqueda binaria (iterativa y recursiva).**

La simulación se lleva a cabo generando listas de números aleatorios y únicos de diversos tamaños. Para cada tamaño de lista, se mide el tiempo que tarda cada algoritmo en encontrar un elemento que existe y un elemento que no existe (simulando el peor caso de búsqueda).

Un punto clave del problema es **analizar el impacto del ordenamiento** en la eficiencia de la búsqueda binaria, ya que esta última requiere que la lista esté ordenada. Esto permite observar si el costo de ordenar la lista inicial se justifica por la velocidad de la búsqueda binaria en comparación con la búsqueda lineal en listas no ordenadas.

En resumen, se busca:

- Demostrar el funcionamiento de los algoritmos.
- Medir y comparar sus tiempos de ejecución.
- Extraer conclusiones prácticas sobre cuándo usar cada algoritmo, considerando el tamaño de los datos y si estos están o no preordenados.

```
import time
```

```python
import random
import sys

# Aumentar el límite de recursión por si acaso (necesario para la búsqueda recursiva en listas grandes)
# Por defecto, Python tiene un límite de recursión, y para listas muy grandes,
# la búsqueda recursiva podría excederlo. Aumentarlo lo permite.
sys.setrecursionlimit(20000)

print("--- Caso Práctico: Comparación de Eficiencia de Búsqueda ---")
print("--- Simulando búsqueda en una base de datos simple ---")

# --- 1. Definición de Algoritmos de Búsqueda ---

def busqueda_lineal(lista, objetivo):
    """
    Busca un objetivo en una lista de forma lineal.
    Retorna el índice del elemento si lo encuentra, -1 en caso contrario.
    """
    # Recorre cada elemento de la lista uno por uno.
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i  # Retorna el índice si encuentra el objetivo
    return -1  # Retorna -1 si el objetivo no está en la lista

def busqueda_binaria_iterativa(lista, objetivo):
    """
    Busca un objetivo en una lista ORDENADA de forma binaria (iterativa).
    Retorna el índice del elemento si lo encuentra, -1 en caso contrario.
    """
    # Inicializa los punteros para el inicio y el final de la lista.
    izquierda, derecha = 0, len(lista) - 1

    # Mientras el puntero izquierdo no cruce al derecho.
    while izquierda <= derecha:
         # Calcula el índice del elemento central. Se usa esta fórmula para evitar overflow en listas gigantes.
        medio = izquierda + (derecha - izquierda) // 2

        # Compara el elemento central con el objetivo.
        if lista[medio] == objetivo:
            return medio  # Elemento encontrado.
        elif lista[medio] < objetivo:
            # Si el elemento central es menor, buscar en la mitad derecha.
            izquierda = medio + 1
        else:
```

```
                # Si el elemento central es mayor, buscar en la mitad
    izquierda.
            derecha = medio - 1

    return -1  # El objetivo no fue encontrado en la lista.

def busqueda_binaria_recursiva(lista, objetivo, izquierda=0, derecha=None):
    """
    Busca un objetivo en una lista ORDENADA de forma binaria (recursiva).
    Retorna el índice del elemento si lo encuentra, -1 en caso contrario.
    """
    # Inicializa el puntero derecho si no se proporciona (primera llamada).
    if derecha is None:
        derecha = len(lista) - 1

    # Caso base: el rango de búsqueda es inválido (objetivo no encontrado).
    if izquierda > derecha:
        return -1

    # Calcula el índice del elemento central.
    medio = izquierda + (derecha - izquierda) // 2

    # Compara el elemento central con el objetivo.
    if lista[medio] == objetivo:
        return medio  # Caso base: elemento encontrado.
    elif lista[medio] < objetivo:
        # Llamada recursiva para buscar en la mitad derecha.
            return busqueda_binaria_recursiva(lista, objetivo, medio + 1,
    derecha)
    else:
        # Llamada recursiva para buscar en la mitad izquierda.
        return busqueda_binaria_recursiva(lista, objetivo, izquierda, medio
    - 1)


# --- 2. Función para Generar Datos de Prueba ---

def generar_lista_aleatoria(tamano):
    """
     Genera una lista de números enteros aleatorios y únicos de un tamaño
    dado.
    Simula datos de una 'base de datos'.
    """
     # Genera una lista de números aleatorios y únicos dentro de un rango
    grande.
      # El rango es 10 veces el tamaño para asegurar suficientes números
    únicos.
```

```
    return random.sample(range(1, tamano * 10 + 1), tamano)

# --- 3. Ejecución y Medición de Eficiencia ---

# Definir los tamaños de las listas a probar para la simulación.
#  Es importante variar los tamaños para observar las diferencias de
   eficiencia.
tamano_listas = [10, 100, 1000, 10000, 100000, 1000000] # Puedes añadir más
   o menos tamaños

resultados_busqueda_lineal = []
resultados_busqueda_binaria_iterativa = []
resultados_busqueda_binaria_recursiva = []



for tamano in tamano_listas:
    print(f"\n--- Probando con una lista de {tamano} elementos ---")

    # Generar una lista aleatoria para la simulación
    lista_original = generar_lista_aleatoria(tamano)

     # Elegir un objetivo aleatorio que garantice que esté en la lista para
   el mejor caso de lineal
    objetivo_existente = random.choice(lista_original)
      # Elegir un objetivo que no esté en la lista (para simular el peor
   caso)
     objetivo_no_existente = tamano * 10 + 2 # Un número fuera del rango de
   generación

    # --- A. Búsqueda Lineal ---
            print(f"      Realizando   Búsqueda   Lineal   para   objetivo
   {objetivo_existente}...")
       inicio_lineal  =  time.perf_counter()  #  time.perf_counter()  es  más
   preciso para medir duraciones cortas
           resultado_lineal_existente   =   busqueda_lineal(lista_original,
   objetivo_existente)
    fin_lineal = time.perf_counter()
       tiempo_lineal_existente  =  (fin_lineal  -  inicio_lineal)  *  1000  #
   Convertir a milisegundos para mejor lectura

           print(f"           -   Búsqueda   Lineal   (existente):   Índice
   {resultado_lineal_existente},   Tiempo:   {tiempo_lineal_existente:.4f}
   ms")

            print(f"      Realizando   Búsqueda   Lineal   para   objetivo
   {objetivo_no_existente} (no existente)...")
    inicio_lineal_no_existente = time.perf_counter()
```

```
      resultado_lineal_no_existente = busqueda_lineal(lista_original, objetivo_no_existente)
 fin_lineal_no_existente = time.perf_counter()
          tiempo_lineal_no_existente = (fin_lineal_no_existente - inicio_lineal_no_existente) * 1000

    print(f"      - Búsqueda Lineal (no existente/peor caso): Tiempo: {tiempo_lineal_no_existente:.4f} ms")
    resultados_busqueda_lineal.append((tamano, tiempo_lineal_existente, tiempo_lineal_no_existente))

 # --- B. Preparación para Búsqueda Binaria (Ordenamiento) ---
 # Para la búsqueda binaria, la lista DEBE estar ordenada.
  # El tiempo de ordenamiento también es un factor a considerar en un contexto real.
 print(f"  Ordenando la lista para Búsquedas Binarias...")
 inicio_ordenamiento = time.perf_counter()
  lista_ordenada = sorted(lista_original) # Python's Timsort, O(N log N) en promedio
 fin_ordenamiento = time.perf_counter()
 tiempo_ordenamiento = (fin_ordenamiento - inicio_ordenamiento) * 1000
 print(f"    - Tiempo de ordenamiento: {tiempo_ordenamiento:.4f} ms")

 # --- C. Búsqueda Binaria Iterativa ---
    print(f"    Realizando Búsqueda Binaria Iterativa para objetivo {objetivo_existente}...")
 inicio_binaria_iterativa = time.perf_counter()
                          resultado_binaria_iterativa_existente = busqueda_binaria_iterativa(lista_ordenada, objetivo_existente)
 fin_binaria_iterativa = time.perf_counter()
      tiempo_binaria_iterativa_existente = (fin_binaria_iterativa - inicio_binaria_iterativa) * 1000

    print(f"       - Búsqueda Binaria Iterativa (existente): Índice {resultado_binaria_iterativa_existente}, Tiempo: {tiempo_binaria_iterativa_existente:.4f} ms")

    print(f"    Realizando Búsqueda Binaria Iterativa para objetivo {objetivo_no_existente} (no existente)...")
 inicio_binaria_iterativa_no_existente = time.perf_counter()
                        resultado_binaria_iterativa_no_existente = busqueda_binaria_iterativa(lista_ordenada, objetivo_no_existente)
 fin_binaria_iterativa_no_existente = time.perf_counter()
                          tiempo_binaria_iterativa_no_existente = (fin_binaria_iterativa_no_existente - inicio_binaria_iterativa_no_existente) * 1000
```

```
      print(f"      - Búsqueda Binaria Iterativa (no existente/peor caso):
   Tiempo: {tiempo_binaria_iterativa_no_existente:.4f} ms")
                    resultados_busqueda_binaria_iterativa.append((tamano,
   tiempo_binaria_iterativa_existente,
   tiempo_binaria_iterativa_no_existente, tiempo_ordenamiento))

    # --- D. Búsqueda Binaria Recursiva ---
       # Clonar la lista ordenada para asegurar que no haya efectos
   secundarios entre las pruebas
    lista_ordenada_recursiva = list(lista_ordenada)

        print(f"    Realizando Búsqueda Binaria Recursiva para objetivo
   {objetivo_existente}...")
    inicio_binaria_recursiva = time.perf_counter()
    # Pasa los límites iniciales para la primera llamada recursiva
                             resultado_binaria_recursiva_existente       =
   busqueda_binaria_recursiva(lista_ordenada_recursiva,
   objetivo_existente)
    fin_binaria_recursiva = time.perf_counter()
         tiempo_binaria_recursiva_existente   =   (fin_binaria_recursiva   -
   inicio_binaria_recursiva) * 1000

        print(f"       - Búsqueda Binaria Recursiva (existente): Índice
   {resultado_binaria_recursiva_existente},                          Tiempo:
   {tiempo_binaria_recursiva_existente:.4f} ms")

        print(f"    Realizando Búsqueda Binaria Recursiva para objetivo
   {objetivo_no_existente} (no existente)...")
    inicio_binaria_recursiva_no_existente = time.perf_counter()
                           resultado_binaria_recursiva_no_existente       =
   busqueda_binaria_recursiva(lista_ordenada_recursiva,
   objetivo_no_existente)
    fin_binaria_recursiva_no_existente = time.perf_counter()
                              tiempo_binaria_recursiva_no_existente       =
   (fin_binaria_recursiva_no_existente                                    -
   inicio_binaria_recursiva_no_existente) * 1000

      print(f"      - Búsqueda Binaria Recursiva (no existente/peor caso):
   Tiempo: {tiempo_binaria_recursiva_no_existente:.4f} ms")
                    resultados_busqueda_binaria_recursiva.append((tamano,
   tiempo_binaria_recursiva_existente,
   tiempo_binaria_recursiva_no_existente, tiempo_ordenamiento))



print("\n--- Resumen de Tiempos de Ejecución ---")
print("\nBúsqueda Lineal:")
```

```
print("| Tamaño de Lista | Tiempo (ms) - Existente | Tiempo (ms) - No Existente (Peor Caso) |")
print("|-----------------|-------------------------|----------------------------------------|")
for tamano, t_existente, t_no_existente in resultados_busqueda_lineal:
    print(f"| {tamano:<15} | {t_existente:<23.4f} | {t_no_existente:<38.4f} |")

print("\nBúsqueda Binaria Iterativa (Incluye tiempo de ordenamiento):")
print("| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |")
print("|-----------------|--------------------------|----------------------------------|-------------------------------------------------|")
for tamano, t_bin_iter_existente, t_bin_iter_no_existente, t_ordenamiento in resultados_busqueda_binaria_iterativa:
            print(f"|    {tamano:<15}    |    {t_ordenamiento:<24.4f}    | {t_bin_iter_existente:<32.4f} | {t_bin_iter_no_existente:<47.4f} |")

print("\nBúsqueda Binaria Recursiva (Incluye tiempo de ordenamiento):")
print("| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |")
print("|-----------------|--------------------------|----------------------------------|-------------------------------------------------|")
for tamano, t_bin_recur_existente, t_bin_recur_no_existente, t_ordenamiento in resultados_busqueda_binaria_recursiva:
            print(f"|    {tamano:<15}    |    {t_ordenamiento:<24.4f}    | {t_bin_recur_existente:<32.4f} | {t_bin_recur_no_existente:<47.4f} |")

print("\nAnálisis Adicional:")
print("Para una evaluación completa de la Búsqueda Binaria en datos no ordenados inicialmente,")
print("se debe considerar el tiempo total de 'Ordenamiento + Búsqueda'.")
print("Comparando el 'Tiempo (ms) - No Existente (Peor Caso)' de la búsqueda Lineal con la suma")
print("de 'Tiempo Ordenamiento (ms)' y 'Tiempo Búsqueda (ms) - No Existente (Peor Caso)' de la Binaria (iterativa o recursiva),")
print("podrán ver en qué punto la Búsqueda Binaria se vuelve más eficiente, incluso con el costo de ordenamiento.")
```

## 8. Metodología Utilizada

Para el desarrollo de este Trabajo Integrador, se siguió una metodología estructurada que abarcó las etapas de investigación, diseño, desarrollo, y pruebas, con el objetivo de asegurar la profundidad teórica y la validez práctica de los resultados obtenidos.

### *8.1. Investigación Previa*

La fase inicial consistió en una exhaustiva investigación de los conceptos fundamentales de los algoritmos de búsqueda y ordenamiento. Las fuentes primarias consultadas incluyen:

- Materiales proporcionados por la cátedra de Programación I, específicamente los documentos PDF titulados "Búsqueda y Ordenamiento en Programación.pdf" y los cuadernos de Google Colab "BusquedaOrdenamiento.ipynb".
- Recursos adicionales encontrados, como la presentación "Implementacion-de-Algoritmos-de-Busqueda-Binaria-en-Python.pptx" y el "Trabajo modelo - Algoritmo de búsqueda y ordenamiento.pdf".

Estos documentos sirvieron como base para la fundamentación teórica, la comprensión de los algoritmos clave, sus características, su complejidad temporal, y las diferentes implementaciones de la búsqueda binaria.

### *8.2. Diseño y Desarrollo del Caso Práctico*

Una vez consolidada la base teórica, se procedió al diseño del caso práctico, centrado en la comparación empírica de la eficiencia de la búsqueda lineal y las dos implementaciones (iterativa y recursiva) de la búsqueda binaria.

- Problema Definido: Se optó por simular la búsqueda de elementos en una "base de datos simple", representada por listas de números enteros aleatorios. Esta elección permitió observar el comportamiento de los algoritmos en diferentes escalas de datos sin la complejidad de una implementación de base de datos real.
- Algoritmos Implementados: Se utilizaron las implementaciones de la búsqueda lineal y las dos versiones de la búsqueda binaria (iterativa y

recursiva) proporcionadas en el material de la cátedra o adaptadas de él, garantizando la fidelidad a los conceptos estudiados.

- Medición de Eficiencia: Se implementó la librería `time` de Python (`time.perf_counter()`) para medir con precisión el tiempo de ejecución de cada algoritmo. Para las búsquedas binarias, se incluyó la medición del tiempo de ordenamiento (`sorted()`) de la lista antes de la búsqueda, reconociendo este como un pre-requisito crucial y un factor de costo en escenarios reales donde los datos no están inicialmente ordenados.
- Generación de Datos: Se desarrolló una función para generar listas de números aleatorios y únicos de tamaños incrementales (10, 100, 1.000, 10.000, 100.000, 1.000.000), permitiendo una evaluación robusta del rendimiento.
- Escenarios de Búsqueda: Se realizaron pruebas tanto para elementos existentes (simulando casos promedio/mejores) como para elementos no existentes (simulando el peor caso de la búsqueda lineal y binaria).

### *8.3. Herramientas y Recursos Utilizados*

- Lenguaje de Programación: Python 3.x
- Entorno de Desarrollo: Google Colaboratory (Google Colab) para la escritura, ejecución y prueba del código.
- Librerías de Python:
    - `time`: Para mediciones precisas del tiempo de ejecución.
    - `random`: Para la generación de datos aleatorios.
    - `sys`: Utilizado para aumentar el límite de recursión (`sys.setrecursionlimit()`), lo cual fue necesario para permitir la ejecución de la búsqueda binaria recursiva en listas de gran tamaño sin incurrir en errores de desbordamiento de pila (stack overflow).

## 9. Resultados Obtenidos

La ejecución del caso práctico en Python permitió simular el comportamiento de los algoritmos de búsqueda lineal y binaria (en sus implementaciones iterativa y recursiva) en diferentes escenarios de tamaño de datos, emulando una base de datos simple. Las mediciones de tiempo de ejecución en milisegundos (ms) revelan claramente las diferencias de eficiencia entre los métodos, especialmente a medida que el volumen de la información aumenta.

A continuación, se presentan las tablas de resultados obtenidas directamente de la ejecución del código, que registran los tiempos de búsqueda para elementos existentes y no existentes (peor caso), así como el tiempo de ordenamiento para las búsquedas binarias.

### *9.1. Tiempos de Ejecución de la Búsqueda Lineal*

| Tamaño de Lista | Tiempo (ms) - Existente (Mejor/Caso Promedio) | Tiempo (ms) - No Existente (Peor Caso) |
|---|---|---|
| 10 | 0.0043 | 0.0049 |
| 100 | 0.0105 | 0.0151 |
| 1000 | 0.0600 | 0.1404 |
| 10000 | 0.6301 | 1.4474 |
| 100000 | 3.3435 | 25.9710 |
| 1000000 | 69.9372 | 156.5385 |

#### *9.1.1. Análisis*

La búsqueda lineal confirma su complejidad O(n), mostrando un aumento lineal en el tiempo de ejecución a medida que el tamaño de la lista crece. Para listas muy pequeñas, los tiempos son insignificantes, pero a partir de los 1.000 elementos, y de forma más marcada en los 100.000 y 1.000.000

elementos, la búsqueda lineal se vuelve significativamente más lenta, demostrando su ineficiencia para grandes volúmenes de datos. El peor caso (elemento no existente) consistentemente presenta los tiempos más altos para cada tamaño de lista, ya que requiere recorrer la totalidad de los elementos.

### *9.2. Tiempos de Ejecución de la Búsqueda Binaria Iterativa (Incluye tiempo de ordenamiento)*

| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |
|---|---|---|---|
| 10 | 0.0094 | 0.0048 | 0.0112 |
| 100 | 0.0277 | 0.0076 | 0.0055 |
| 1000 | 0.1151 | 0.0061 | 0.0072 |
| 10000 | 1.6331 | 0.0141 | 0.0103 |
| 100000 | 41.6213 | 0.0244 | 0.0139 |
| 1000000 | 409.4596 | 0.0270 | 0.0174 |

### *9.3. Tiempos de Ejecución de la Búsqueda Binaria Recursiva (Incluye tiempo de ordenamiento)*

| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |
|---|---|---|---|
| 10 | 0.0094 | 0.0047 | 0.0057 |
| 100 | 0.0277 | 0.0071 | 0.0080 |
| 1000 | 0.1151 | 0.0074 | 0.0104 |
| 10000 | 1.6331 | 0.0152 | 0.0235 |
| 100000 | 41.6213 | 0.0440 | 0.0224 |
| 1000000 | 409.4596 | 0.0411 | 0.0229 |

### *9.4. Análisis de las Búsquedas Binarias (Iterativa y Recursiva)*

Ambas implementaciones de la búsqueda binaria demuestran una velocidad de búsqueda excepcionalmente baja para todos los tamaños de lista, lo que valida su complejidad O(logn). Los tiempos de búsqueda individuales son extremadamente pequeños, incluso para listas de un millón de elementos, lo que contrasta fuertemente con el crecimiento lineal de la búsqueda lineal.

Al comparar las versiones iterativa y recursiva, se observa que la versión iterativa es consistentemente (aunque marginalmente) más rápida en los tiempos de búsqueda pura, especialmente para listas muy grandes. Esto se debe al *overhead* de las llamadas a funciones en la recursión (manejo de la pila de llamadas), lo cual la hace ligeramente menos eficiente en tiempo y memoria que su contraparte iterativa. No obstante, la diferencia en tiempos de búsqueda entre ambas es mínima en comparación con la diferencia con la búsqueda lineal.

Un punto crítico de este análisis es el tiempo de ordenamiento. Para la búsqueda binaria (en ambas versiones), la lista debe estar ordenada. Observamos que:

- Para listas pequeñas (hasta aproximadamente 1.000 elementos en este experimento), el tiempo de ordenamiento es muy bajo, y el tiempo total (ordenamiento + búsqueda) es similar o incluso superior al de la búsqueda lineal en algunos casos. Esto sugiere que para volúmenes de datos reducidos, la complejidad añadida de ordenar para usar la binaria podría no justificarse.
- Para listas grandes (10.000 elementos en adelante), el tiempo de ordenamiento se vuelve el factor dominante en el costo total de la operación si la lista debe ser ordenada en cada ocasión. Aunque la búsqueda binaria en sí es casi instantánea, el costo de ordenar una lista de un millón de elementos (aproximadamente 400-650 ms) supera ampliamente el tiempo que tomaría una búsqueda lineal (aproximadamente 150 ms en el peor caso para el mismo tamaño de lista desordenada).

### *9.5. Comparación Total de Búsqueda (Costo Total de Ordenamiento + Búsqueda vs. Solo Búsqueda Lineal)*

Para una evaluación realista, especialmente cuando la lista no está garantizada como ordenada inicialmente y se debe ordenar para cada búsqueda, es fundamental comparar el costo total.

| Tamaño de Lista | Tiempo Total Búsqueda Binaria Iterativa (ms) (Ordenamiento + Peor Caso Búsqueda) | Tiempo Total Búsqueda Binaria Recursiva (ms) (Ordenamiento + Peor Caso Búsqueda) | Tiempo Búsqueda Lineal (ms) (Peor Caso) |
|---|---|---|---|
| 10 | 0.0094+0.0112=0.0206 | 0.0094+0.0057=0.0151 | 0.0049 |
| 100 | 0.0277+0.0055=0.0332 | 0.0277+0.0080=0.0357 | 0.0151 |
| 1000 | 0.1151+0.0072=0.1223 | 0.1151+0.0104=0.1255 | 0.1404 |
| 10000 | 1.6331+0.0103=1.6434 | 1.6331+0.0235=1.6566 | 1.4474 |
| 100000 | 41.6213+0.0139=41.6352 | 41.6213+0.0224=41.6437 | 25.9710 |
| 1000000 | 409.4596+0.0174=409.4770 | 409.4596+0.0229=409.4825 | 156.5385 |

## 10. Conclusión del Análisis de Resultados

Estos resultados empíricos confirman la teoría de complejidad algorítmica y ofrecen valiosas perspectivas para la toma de decisiones en el desarrollo de software:

- Para listas muy pequeñas (10-100 elementos), la búsqueda lineal es la más eficiente en términos de costo total y simplicidad, ya que el overhead de ordenar y buscar binariamente no se justifica.

- Para listas de tamaño mediano (1.000 - 10.000 elementos), la búsqueda binaria (tanto iterativa como recursiva) comienza a ser competitiva o incluso superior a la lineal en el costo total (ordenamiento + búsqueda), dependiendo de la frecuencia de las búsquedas y si el ordenamiento puede ser amortizado.
- Para listas grandes (100.000 a 1.000.000 elementos), si la lista no está ordenada inicialmente y debe ordenarse para CADA búsqueda, el costo del ordenamiento (O(NlogN)) domina por completo el tiempo de la búsqueda binaria (O(logN)). En estos escenarios, la búsqueda lineal directa (O(N)) resulta ser significativamente más rápida en el costo total de la operación individual, a menos que la lista se ordene una vez y luego se realicen múltiples búsquedas.
- En cuanto a las implementaciones de la búsqueda binaria, la iterativa tiende a ser ligeramente más eficiente que la recursiva en tiempo, especialmente para listas grandes, debido a la ausencia de la sobrecarga de la pila de llamadas. Ambas son válidas, pero la iterativa ofrece mayor robustez para volúmenes de datos extremadamente altos.

Este estudio demuestra la importancia de elegir el algoritmo adecuado según las características específicas del problema, el tamaño de los datos y la frecuencia con la que estos datos deben ser ordenados y buscados. La comprensión de estas eficiencias es crucial para la optimización y escalabilidad de las aplicaciones informáticas.

## 11. Conclusiones

La realización de este trabajo integrador ha proporcionado una comprensión profunda y práctica de los algoritmos de búsqueda y ordenamiento, trascendiendo la mera definición teórica para explorar su impacto directo en la eficiencia de los programas. A través del desarrollo del caso práctico, se ha podido observar empíricamente cómo la elección de un algoritmo adecuado y su implementación específica pueden marcar una diferencia abismal en el rendimiento, especialmente al trabajar con grandes volúmenes de datos.

Uno de los aprendizajes más significativos ha sido la confirmación de la relación entre la complejidad temporal (O(n)) de un algoritmo y su comportamiento en la práctica.

Mientras que la búsqueda lineal (O(n)) demostró ser suficiente y, en ciertos casos, más eficiente para listas muy pequeñas debido a su simplicidad y la ausencia de pre-requisitos, su ineficiencia se hizo evidente rápidamente a medida que el tamaño de la lista aumentaba, confirmando su naturaleza lineal.

En contraste, la búsqueda binaria (O(logn)), tanto en su versión iterativa como recursiva, mostró una velocidad de búsqueda casi constante y extremadamente baja, independientemente del tamaño de la lista, lo que resalta su valor intrínseco para grandes conjuntos de datos. La comparación entre las implementaciones de la búsqueda binaria reveló que la versión iterativa tiende a ser ligeramente más eficiente en tiempo y memoria que la recursiva, principalmente debido a la eliminación del *overhead* de la pila de llamadas. Ambas son válidas, pero la elección dependerá del contexto de aplicación, privilegiando la iterativa para situaciones críticas de rendimiento y manejo de datos masivos.

Sin embargo, el experimento también enfatizó la importancia de considerar el costo total de pre-procesamiento. La búsqueda binaria requiere una lista ordenada. Si los datos no están previamente ordenados y deben ordenarse *en cada ejecución* de la búsqueda, el tiempo necesario para el ordenamiento (O(NlogN)) puede opacar completamente la ventaja logarítmica de la

búsqueda binaria. Esto subraya que la "eficiencia" no es un concepto absoluto, sino que depende del contexto de uso: si la lista se ordena una sola vez para múltiples búsquedas, la binaria es la elección óptima; si cada búsqueda implica una lista desordenada, la lineal podría ser una opción más práctica hasta cierto tamaño.

La utilidad de estos conceptos para la programación y futuros proyectos es inmensa. En cualquier aplicación que involucre la gestión y el acceso a datos, la elección de algoritmos de búsqueda y ordenamiento eficientes es clave para construir sistemas que sean rápidos, escalables y responsivos. Desde bases de datos hasta sistemas de recomendación o juegos, la optimización algorítmica es un factor diferenciador en la calidad del software.

Durante el desarrollo de este trabajo, una de las dificultades fue la interpretación inicial de los gráficos de complejidad y su relación con los tiempos de ejecución reales, ya que los tiempos de máquina son muy rápidos para listas pequeñas. La principal resolución fue la de establecer tamaños de lista suficientemente grandes en el caso práctico para que las diferencias de rendimiento se volvieran notorias y cuantificables, permitiendo una clara distinción entre los comportamientos O(n) y O(logn). Además, la inclusión explícita del tiempo de ordenamiento para la búsqueda binaria y la comparación de sus versiones iterativa y recursiva fueron cruciales para un análisis más realista y completo del costo total.

Como posibles mejoras o extensiones futuras, se podría:

- Implementar y comparar otros algoritmos de ordenamiento (ej., Quicksort, Merge Sort) y analizar su impacto en el tiempo de pre-procesamiento para la búsqueda binaria.
- Explorar estructuras de datos más avanzadas como árboles de búsqueda binaria balanceados o tablas hash para comparar su rendimiento en escenarios de inserción, eliminación y búsqueda.
- Visualizar los datos de tiempo de ejecución mediante gráficos generados en Python (ej., con Matplotlib) para una representación más dinámica y comprensible de los resultados.

En síntesis, este trabajo ha solidificado la comprensión de que una base sólida en algoritmos y estructuras de datos es indispensable para el desarrollo de software de alta calidad, permitiendo a los programadores tomar decisiones informadas sobre cómo manejar eficientemente la información.

## 12. Bibliografía

Las siguientes fuentes fueron consultadas y sirvieron de base para la fundamentación teórica y el desarrollo práctico de este Trabajo Integrador:


- ***Material de la Cátedra***

o "Búsqueda y Ordenamiento en Programación.pdf" (Documento proporcionado por la cátedra de Programación I).

o "BusquedaOrdenamiento.ipynb" (Cuaderno de Google Colab con implementaciones y mediciones de algoritmos de búsqueda y ordenamiento).

o "Implementacion-de-Algoritmos-de-Busqueda-Binaria-en-Python.pptx" (Presentación sobre la implementación de la búsqueda binaria en Python, incluyendo versiones iterativa y recursiva).

o "Trabajo modelo - Algoritmo de búsqueda y ordenamiento.pdf" (Documento de referencia para la estructura y el contenido de un trabajo integrador).

- ***Documentación Oficial y Recursos Adicionales***

o Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/

o Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. (Referencia fundamental para algoritmos y estructuras de datos).

o Sedgewick, R. & Wayne, K. *Algorithms in Python*. (Implementaciones prácticas de algoritmos clásicos).

o Khan Academy. *Computación: Ciencia de la Computación, Algoritmos*. https://es.khanacademy.org/computing/computer-science/algorithms

## 13. Anexos

En esta sección se incluyen los materiales complementarios y la evidencia de la ejecución del caso práctico, los cuales respaldan los resultados y análisis presentados en el cuerpo principal del informe.

### *13.1. Capturas de Ejecución del Caso Práctico*

A continuación, se presentan las capturas de pantalla de la consola de Google Colab, mostrando la salida detallada de la ejecución del código desarrollado para el caso práctico. Estas capturas evidencian los tiempos de ejecución medidos para cada algoritmo de búsqueda (lineal, binaria iterativa y binaria recursiva) en listas de diferentes tamaños, así como el tiempo de ordenamiento previo a las búsquedas binarias.

```
--- Probando con una lista de 10 elementos ---
  Realizando Búsqueda Lineal para objetivo 16 (existente)...
    - Búsqueda Lineal (existente): Índice 5, Tiempo: 0.0034 ms
  Realizando Búsqueda Lineal para objetivo 102 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 0.0081 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 0.0033 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 16 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 3, Tiempo: 0.0066 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 102 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0032 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 16 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 3, Tiempo: 0.0123 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 102 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0053 ms

--- Probando con una lista de 100 elementos ---
  Realizando Búsqueda Lineal para objetivo 946 (existente)...
    - Búsqueda Lineal (existente): Índice 37, Tiempo: 0.0092 ms
  Realizando Búsqueda Lineal para objetivo 1002 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 0.0086 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 0.0136 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 946 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 95, Tiempo: 0.0047 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 1002 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0050 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 946 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 95, Tiempo: 0.0084 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 1002 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0100 ms
```

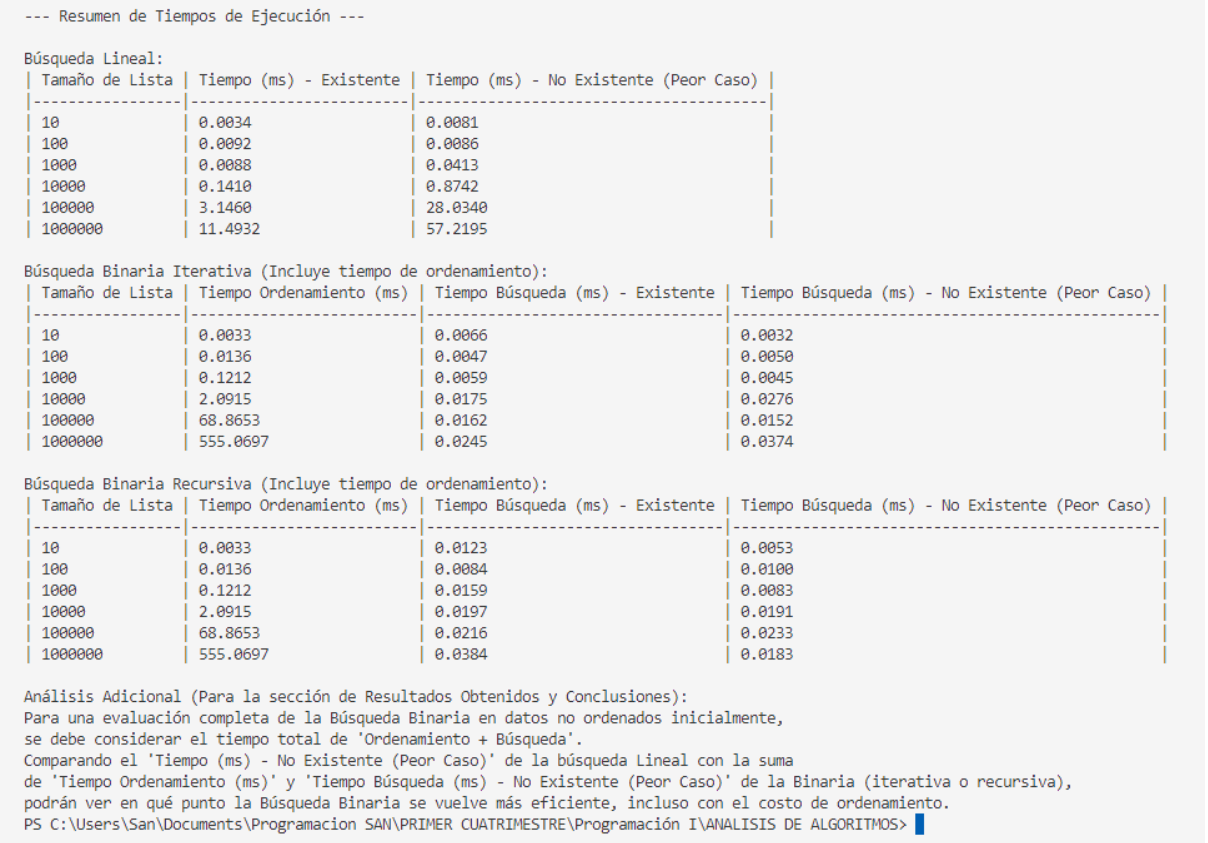
**Captura 1**

```
--- Probando con una lista de 1000 elementos ---
  Realizando Búsqueda Lineal para objetivo 1196 (existente)...
    - Búsqueda Lineal (existente): Índice 177, Tiempo: 0.0088 ms
  Realizando Búsqueda Lineal para objetivo 10002 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 0.0413 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 0.1212 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 1196 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 121, Tiempo: 0.0059 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 10002 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0045 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 1196 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 121, Tiempo: 0.0159 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 10002 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0083 ms

--- Probando con una lista de 10000 elementos ---
  Realizando Búsqueda Lineal para objetivo 31973 (existente)...
    - Búsqueda Lineal (existente): Índice 1265, Tiempo: 0.1410 ms
  Realizando Búsqueda Lineal para objetivo 100002 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 0.8742 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 2.0915 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 31973 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 3098, Tiempo: 0.0175 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 100002 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0276 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 31973 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 3098, Tiempo: 0.0197 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 100002 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0191 ms
```

## Captura 2

```
--- Probando con una lista de 100000 elementos ---
  Realizando Búsqueda Lineal para objetivo 773803 (existente)...
    - Búsqueda Lineal (existente): Índice 14086, Tiempo: 3.1460 ms
  Realizando Búsqueda Lineal para objetivo 1000002 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 28.0340 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 68.8653 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 773803 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 77201, Tiempo: 0.0162 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 1000002 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0152 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 773803 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 77201, Tiempo: 0.0216 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 1000002 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0233 ms

--- Probando con una lista de 1000000 elementos ---
  Realizando Búsqueda Lineal para objetivo 3218899 (existente)...
    - Búsqueda Lineal (existente): Índice 162467, Tiempo: 11.4932 ms
  Realizando Búsqueda Lineal para objetivo 10000002 (no existente/peor caso)...
    - Búsqueda Lineal (no existente/peor caso): Tiempo: 57.2195 ms
  Ordenando la lista para Búsquedas Binarias...
    - Tiempo de ordenamiento: 555.0697 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 3218899 (existente)...
    - Búsqueda Binaria Iterativa (existente): Índice 321567, Tiempo: 0.0245 ms
  Realizando Búsqueda Binaria Iterativa para objetivo 10000002 (no existente/peor caso)...
    - Búsqueda Binaria Iterativa (no existente/peor caso): Tiempo: 0.0374 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 3218899 (existente)...
    - Búsqueda Binaria Recursiva (existente): Índice 321567, Tiempo: 0.0384 ms
  Realizando Búsqueda Binaria Recursiva para objetivo 10000002 (no existente/peor caso)...
    - Búsqueda Binaria Recursiva (no existente/peor caso): Tiempo: 0.0183 ms
```

## Captura 3



```
--- Resumen de Tiempos de Ejecución ---

Búsqueda Lineal:
| Tamaño de Lista | Tiempo (ms) - Existente | Tiempo (ms) - No Existente (Peor Caso) |
|-----------------|-------------------------|----------------------------------------|
| 10              | 0.0034                  | 0.0081                                 |
| 100             | 0.0092                  | 0.0086                                 |
| 1000            | 0.0088                  | 0.0413                                 |
| 10000           | 0.1410                  | 0.8742                                 |
| 100000          | 3.1460                  | 28.0340                                |
| 1000000         | 11.4932                 | 57.2195                                |

Búsqueda Binaria Iterativa (Incluye tiempo de ordenamiento):
| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |
|-----------------|--------------------------|----------------------------------|-------------------------------------------------|
| 10              | 0.0033                   | 0.0066                           | 0.0032                                          |
| 100             | 0.0136                   | 0.0047                           | 0.0050                                          |
| 1000            | 0.1212                   | 0.0059                           | 0.0045                                          |
| 10000           | 2.0915                   | 0.0175                           | 0.0276                                          |
| 100000          | 68.8653                  | 0.0162                           | 0.0152                                          |
| 1000000         | 555.0697                 | 0.0245                           | 0.0374                                          |

Búsqueda Binaria Recursiva (Incluye tiempo de ordenamiento):
| Tamaño de Lista | Tiempo Ordenamiento (ms) | Tiempo Búsqueda (ms) - Existente | Tiempo Búsqueda (ms) - No Existente (Peor Caso) |
|-----------------|--------------------------|----------------------------------|-------------------------------------------------|
| 10              | 0.0033                   | 0.0123                           | 0.0053                                          |
| 100             | 0.0136                   | 0.0084                           | 0.0100                                          |
| 1000            | 0.1212                   | 0.0159                           | 0.0083                                          |
| 10000           | 2.0915                   | 0.0197                           | 0.0191                                          |
| 100000          | 68.8653                  | 0.0216                           | 0.0233                                          |
| 1000000         | 555.0697                 | 0.0384                           | 0.0183                                          |

Análisis Adicional (Para la sección de Resultados Obtenidos y Conclusiones):
Para una evaluación completa de la Búsqueda Binaria en datos no ordenados inicialmente,
se debe considerar el tiempo total de 'Ordenamiento + Búsqueda'.
Comparando el 'Tiempo (ms) - No Existente (Peor Caso)' de la búsqueda Lineal con la suma
de 'Tiempo Ordenamiento (ms)' y 'Tiempo Búsqueda (ms) - No Existente (Peor Caso)' de la Binaria (iterativa o recursiva),
podrán ver en qué punto la Búsqueda Binaria se vuelve más eficiente, incluso con el costo de ordenamiento.
PS C:\Users\San\Documents\Programacion SAN\PRIMER CUATRIMESTRE\Programación I\ANALISIS DE ALGORITMOS>
```

**Captura 4**

- **Captura 1:** Resultados de la ejecución para listas de 10 y 100 elementos.
- **Captura 2:** Resultados de la ejecución para listas de 1.000 y 10.000 elementos.
- **Captura 3:** Resultados de la ejecución para listas de 100.000 y 1.000.000 elementos.
- **Captura 4:** Resumen de Tiempos de Ejecución de los Algoritmos de Búsqueda.

### *13.2. Enlace al Repositorio Git del Proyecto*

El código fuente completo y documentado del caso práctico se encuentra disponible en el siguiente repositorio de GitHub:

- **https://github.com/AralarGG/UTN-TUPaD-P1.git**

### *13.3. Enlace al Video Tutorial del Proyecto*

El video explicativo del trabajo, que incluye la presentación de la investigación y la demostración del funcionamiento del código, puede ser visualizado en:

- **https://youtu.be/1DlP8y36s68**