# Trabajo Integrador – Programación I

## Datos Generales

- **Título del trabajo**: “Algoritmos de Búsqueda y Ordenamiento en Python”
- **Alumnos**:
  - Albertini Hugo Agustín - agustin_alber@hotmail.com
  - Calcatelli Renzo - rcalcatelli@gmail.com
- **Materia**: Programación I
- **Profesor/a**: Rigoni Cinthia
- **Fecha de Entrega**: Formato: 09/06/2025

---

**Índice**

## 1. Introducción

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación, ya que permiten organizar y acceder eficientemente a grandes volúmenes de datos. Este trabajo se enfoca en implementar y comparar diferentes algoritmos clásicos de ordenamiento (Selección, Inserción, Burbuja, Quicksort y Mergesort) y búsqueda (Lineal y Binaria) en Python.
La importancia de estos algoritmos radica en su aplicación en diversos campos como bases de datos, sistemas de recomendación y análisis de datos. Comprender su funcionamiento y eficiencia permite seleccionar la mejor opción según el contexto, optimizando recursos computacionales.

**Los objetivos principales son:**

- Implementar algoritmos de ordenamiento y búsqueda en Python.
- Comparar su rendimiento mediante medición de tiempos de ejecución.
- Analizar ventajas y desventajas de cada algoritmo.
- Validar su correcto funcionamiento con diferentes conjuntos de datos.

## 2. Marco Teórico

Este marco teórico explora los fundamentos de los algoritmos de búsqueda y ordenamiento, profundizando en su funcionamiento, sus características y, crucialmente, en el análisis de su eficiencia mediante la Notación Big O. En este trabajo nos centraremos en los algoritmos clásicos más utilizados en la práctica y en la enseñanza universitaria.

### Algoritmos de Ordenamiento

Los algoritmos de ordenamiento reorganizan los elementos de una lista (o arreglo) en un orden específico (ascendente o descendente). Son fundamentales para optimizar búsquedas (como la binaria) y facilitar el análisis de datos.

### Bubble Sort (Ordenamiento de Burbuja)

- **Concepto y funcionamiento:** Es uno de los algoritmos de ordenamiento más simples. Recorre repetidamente la lista, comparando pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista está ordenada. Los elementos "más pesados" (mayores) "flotan" hacia el final de la lista en cada pasada.
- **Análisis de complejidad:**
    - **Mejor Caso:** O(n) - La lista ya está ordenada.
    - **Peor Caso y Caso Promedio:** O(n2) - Requiere múltiples pasadas y comparaciones anidadas.
- **Complejidad:** O(1) - Ordena la lista "in-place".
- **Estabilidad:** Es un algoritmo estable.
- **Ventajas y Desventajas:** Es muy fácil de entender e implementar, pero su ineficiencia lo hace poco práctico para grandes conjuntos de datos.

### Selection Sort (Ordenamiento por Selección)

- **Concepto y funcionamiento:** Divide la lista en dos partes: una sublista ordenada (inicialmente vacía) y una sublista desordenada. En cada iteración, busca el elemento más pequeño (o más grande) en la sublista desordenada y lo intercambia con el primer elemento de la sublista desordenada, extendiendo así la sublista ordenada.
- **Análisis de complejidad:**
    - **Mejor Caso, Peor Caso y Caso Promedio:** O(n2) - Siempre realiza un número fijo de comparaciones, independientemente del estado inicial de la lista.

- **Complejidad:** O(1) - Ordena la lista "in-place".
- **Estabilidad:** Generalmente no es estable (a menos que se implemente con cuidado para mantener la estabilidad).
- **Ventajas y Desventajas**: Es simple de implementar y minimiza el número de intercambios. Sin embargo, es ineficiente para listas grandes.

### Insertion Sort (Ordenamiento por Inserción)

- **Concepto y funcionamiento:** También divide la lista en una parte ordenada y una desordenada. En cada iteración, toma el siguiente elemento de la sublista desordenada y lo "inserta" en su posición correcta dentro de la sublista ordenada, desplazando los elementos mayores hacia la derecha.
- **Análisis de complejidad:**
    - **Mejor Caso:** O(n) - La lista ya está ordenada.
    - **Peor Caso y Caso Promedio:** O(n2) - Para listas inversamente ordenadas o aleatorias.
- **Complejidad**: O(1) - Ordena la lista "in-place".
- **Estabilidad:** Es un algoritmo estable.
- **Ventajas y Desventajas:** Es eficiente para listas pequeñas o listas que ya están casi ordenadas. Es simple de implementar. Su principal desventaja es su ineficiencia para listas grandes.

### Quicksort (Ordenamiento Rápido)

- **Concepto y funcionamiento:** Es un algoritmo de "divide y vencerás" muy popular. Elige un elemento de la lista llamado pivote. Luego, reorganiza la lista de tal manera que todos los elementos menores que el pivote queden a su izquierda y todos los elementos mayores queden a su derecha. El pivote está ahora en su posición final ordenada. Finalmente, aplica recursivamente los pasos anteriores a las sublistas a ambos lados del pivote.
- **Análisis de complejidad:**
    - **Mejor Caso y Caso Promedio:** O(nlogn) - Cuando el pivote divide la lista de manera equilibrada.
    - **Peor Caso:** O(n2) - Cuando el pivote siempre es el elemento más pequeño o más grande (ej. lista ya ordenada y pivote siempre el primero).

- **Complejidad:** O(logn) (caso promedio, debido a la recursión) o O(n) (peor caso, para la pila de llamadas).
- **Estabilidad:** Generalmente no es estable.
- **Ventajas y Desventajas:** En la práctica, es uno de los algoritmos de ordenamiento más rápidos para la mayoría de los conjuntos de datos. Su rendimiento de peor caso O(n2) es una desventaja teórica, aunque poco frecuente en la práctica con buenas estrategias de elección de pivote.

### Merge Sort (Ordenamiento por Mezcla)

- **Concepto y funcionamiento:** Es otro algoritmo de "divide y vencerás". Primero, divide la lista en dos mitades recursivamente hasta que cada sublista contiene un solo elemento. Luego, combina (mezcla) las sublistas ordenadas de forma recursiva para producir listas ordenadas más grandes, hasta que se tiene una única lista ordenada. El paso de "mezcla" es una operación clave y eficiente.
- **Análisis de complejidad:**
    - **Mejor Caso, Peor Caso y Caso Promedio:** O(nlogn) - Consistente para todos los casos debido a su estructura de división y mezcla equilibrada.
- **Complejidad:** O(n) - Requiere un arreglo auxiliar para el proceso de mezcla.
- **Estabilidad:** Es un algoritmo estable.
- **Ventajas y Desventajas:** Ofrece un rendimiento consistente de O(nlogn) en todos los casos y es un algoritmo estable. Su principal desventaja es que requiere espacio adicional O(n), lo que puede ser un problema para grandes volúmenes de datos con memoria limitada.

### Algoritmos de Búsqueda

Los algoritmos de búsqueda tienen como objetivo encontrar uno o más elementos dentro de una estructura de datos. Su eficiencia varía drásticamente según la organización de los datos y el método empleado.

### Búsqueda Lineal (Secuencial)

- Concepto y Funcionamiento: Es el método de búsqueda más simple. Consiste en recorrer secuencialmente cada elemento de una lista (o arreglo) desde el inicio hasta el final, comparando cada elemento con el valor buscado. Si el elemento se encuentra, la búsqueda termina; de lo contrario, se recorre toda la lista.

- **Análisis de Complejidad:**
    - Mejor Caso: O(1) - El elemento buscado es el primero de la lista.
    - Peor Caso: O(n) - El elemento buscado es el último de la lista, o no está presente.
    - Caso Promedio: O(n) - En promedio, se recorre la mitad de la lista.
- **Complejidad:** O(1) - Requiere una cantidad constante de memoria adicional.
- **Ventajas y Desventajas:** Es simple de implementar y no requiere que la lista esté ordenada. Sin embargo, es muy ineficiente para listas grandes.

### Búsqueda Binaria

- Concepto y Funcionamiento: Es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal, pero requiere que la lista esté previamente ordenada. Funciona dividiendo repetidamente por la mitad la porción de la lista donde el elemento podría estar. En cada paso, el algoritmo compara el elemento buscado con el elemento central de la porción actual. Si son iguales, se encuentra el elemento. Si el elemento buscado es menor, la búsqueda continúa en la mitad inferior; si es mayor, en la mitad superior.
- **Análisis de Complejidad:**
    - **Mejor Caso:** O(1) - El elemento buscado es el central de la lista.
    - **Peor Caso y Caso Promedio:** O(logn) - El número de comparaciones se reduce a la mitad en cada paso.
- **Complejidad:** O(1) para la versión iterativa, O(logn) para la versión recursiva (debido a la pila de llamadas).
- **Ventajas y Desventajas**: Extremadamente eficiente para listas grandes y ordenadas. Su principal desventaja es el requisito de que la lista esté ordenada, lo que puede implicar un costo adicional.

### Análisis de Algoritmos: Eficiencia y Notación Big O

El análisis de algoritmos es el proceso de determinar los recursos (tiempo y espacio) necesarios para ejecutar un algoritmo. Esto nos permite comparar algoritmos y predecir su rendimiento en diferentes escenarios.

### Importancia del Análisis de Algoritmos

El análisis de algoritmos es crucial para predecir y comparar el rendimiento de diferentes soluciones para un mismo problema, especialmente a medida que el volumen de datos crece. No se trata de medir el tiempo exacto en segundos (ya que esto varía con el hardware y el

software), sino de entender cómo el tiempo o el espacio requerido por un algoritmo escala con el tamaño de la entrada.

## La Notación Big O

La Notación Big O es una notación matemática que describe el comportamiento asintótico del tiempo o espacio de ejecución de un algoritmo a medida que el tamaño de la entrada (n) tiende a infinito. Se enfoca en el término de mayor crecimiento de la función de complejidad y omite constantes y términos de orden inferior, ya que estos factores se vuelven insignificantes para grandes valores de n.

**Tabla comparativa de los algoritmos estudiados, destacando sus complejidades temporales y espaciales, así como su estabilidad.**

| Algoritmo | Funcionamiento Básico | Tiempo: Mejor Caso | Tiempo: Promedio | Tiempo: Peor Caso | Espacio | Estable |
|---|---|---|---|---|---|---|
| **Búsqueda** | | | | | | |
| Búsqueda Lineal | Recorre secuencialmente | $O(1)$ | $O(n)$ | $O(n)$ | $O(1)$ | - |
| Búsqueda Binaria | Divide la lista ordenada por la mitad | $O(1)$ | $O(\log n)$ | $O(\log n)$ | $O(1)$ | - |
| **Ordenamiento** | | | | | | |
| Bubble Sort | Intercambia adyacentes | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✅ |
| Insertion Sort | Inserta elemento en sublista ordenada | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✅ |
| Selection Sort | Encuentra mínimo y lo reubica | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ❌ |
| Merge Sort | Divide y mezcla | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | ✅ |
| Quick Sort | Elige pivote y particiona | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | ❌ |

## Notas sobre la tabla:

- Los valores de Eficiencia (Big O) representan el crecimiento del tiempo de ejecución en relación con el tamaño de la entrada (n).
- Espacio se refiere a la complejidad espacial auxiliar (memoria adicional requerida por el algoritmo).
- Estable indica si el algoritmo mantiene el orden relativo de elementos con valores iguales.

### Conclusión del Marco Teórico

El estudio de los algoritmos de búsqueda y ordenamiento, junto con el análisis de su eficiencia, es fundamental para cualquier desarrollador. Vimos cómo algoritmos con lógicas aparentemente simples pueden tener complejidades muy diferentes, impactando drásticamente el rendimiento con el crecimiento de los datos. La Notación Big O proporciona una herramienta poderosa para predecir y comparar el comportamiento de los algoritmos de manera abstracta, permitiéndonos tomar decisiones informadas sobre qué solución aplicar.
En este trabajo nos enfocamos en los métodos más utilizados y enseñados, mostrando sus ventajas, desventajas y aplicaciones prácticas en Python.

---

## 3. Caso Práctico

### Descripción del Problema

Se implementaron 5 algoritmos de ordenamiento y 2 de búsqueda para:

- Ordenar una lista de 10,000 números enteros aleatorios.
- Buscar un elemento específico en la lista original y en la ordenada.
- Comparar tiempos de ejecución de cada algoritmo.

**algoritmos_busqueda.py:**

```python
def busqueda_lineal(lista, objetivo):
    # Recorre cada elemento de la lista
    for i in range(len(lista)):
        # Si el elemento actual es igual al objetivo
        if lista[i] == objetivo:
            # Devuelve el índice donde se encontró el objetivo
            return i
    # Si no se encuentra el objetivo, devuelve -1
    return -1

def busqueda_binaria(lista, objetivo):
    """
    Realiza una búsqueda binaria en una lista ordenada para encontrar la posición de un elemento objetivo.
    Parámetros:
        lista (list): Lista de elementos ordenados donde se realizará la búsqueda.
        objetivo: Elemento que se desea encontrar en la lista.
    Retorna:
        int: Índice del elemento objetivo si se encuentra en la lista; de lo contrario, -1.
    """
    izquierda = 0  # Índice inicial de la lista
    derecha = len(lista) - 1  # Índice final de la lista

    # Mientras el índice izquierdo no supere al derecho
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2  # Calcula el índice del medio
        # Si el elemento en el medio es el objetivo
        if lista[medio] == objetivo:
            return medio  # Devuelve el índice del objetivo
        # Si el elemento en el medio es menor que el objetivo
        elif lista[medio] < objetivo:
            izquierda = medio + 1  # Busca en la mitad derecha
        else:
            derecha = medio - 1  # Busca en la mitad izquierda
    # Si no se encuentra el objetivo, devuelve -1
    return -1
```

**algoritmos_ordenamiento.py**

```python
from .utilidades import mezclar

# Ordenamiento por selección: busca el menor elemento y lo coloca al principio
def ordenamiento_seleccion(lista):
    n = len(lista)
    for i in range(n):
        indice_minimo = i
        for j in range(i + 1, n):
            if lista[j] < lista[indice_minimo]:
                indice_minimo = j
        # Intercambia el elemento actual con el menor encontrado
        lista[i], lista[indice_minimo] = lista[indice_minimo], lista[i]
    return lista

# Ordenamiento por inserción: inserta cada elemento en su lugar correcto
def ordenamiento_insercion(lista):
    n = len(lista)
    for i in range(1, n):
        valor_actual = lista[i]
        j = i - 1
        # Desplaza los elementos mayores hacia la derecha
        while j >= 0 and valor_actual < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        # Inserta el valor en la posición correcta
        lista[j + 1] = valor_actual
    return lista
```

```python
# Ordenamiento burbuja: compara elementos adyacentes y los intercambia si están desordenados
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n - 1):
        for j in range(0, n - 1):
            if lista[j] > lista[j + 1]:
                # Intercambia si están en el orden incorrecto
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

# Quicksort: divide la lista usando un pivote y ordena recursivamente
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista.pop()  # Toma el último elemento como pivote
    menores = []
    mayores = []
    for x in lista:
        if x <= pivote:
            menores.append(x)
        else:
            mayores.append(x)
    # Ordena recursivamente las sublistas y las une
    return quicksort(menores) + [pivote] + quicksort(mayores)
```

```python
# Mergesort: divide la lista en mitades, las ordena y luego las mezcla
def mergesort(lista):
    if len(lista) <= 1:
        return lista
    mitad = len(lista) // 2
    izquierda = lista[:mitad]
    derecha = lista[mitad:]
    izquierda = mergesort(izquierda)
    derecha = mergesort(derecha)
    # Mezcla las dos mitades ordenadas
    return mezclar(izquierda, derecha)
```

**utilidades.py**

```python
def mezclar(izquierda, derecha):
    """
    Mezcla dos listas ordenadas en una sola lista ordenada.
    Args:
        izquierda (list): Primera lista ordenada.
        derecha (list): Segunda lista ordenada.
    Returns:
        list: Nueva lista ordenada que contiene todos los elementos de ambas listas.
    """
    resultado = []
    i = j = 0

    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1

    resultado += izquierda[i:]
    resultado += derecha[j:]
    return resultado
```

**main.py**

```python
# Importamos las librerías necesarias
import timeit  # timeit: para medir el tiempo de ejecución de los algoritmos
import random  # random: para generar números aleatorios
from src.algoritmos_ordenamiento import ( # Para importar los algoritmos de ordenamiento
    ordenamiento_seleccion,
    ordenamiento_insercion,
    ordenamiento_burbuja,
    quicksort,
    mergesort
)
from src.algoritmos_busqueda import busqueda_lineal, busqueda_binaria # Para importar los algoritmos de búsqueda
from src.utilidades import mezclar # Para importar la función mezclar

# Ejecuta y mide el tiempo de varios algoritmos de ordenamiento
def ejecutar_algoritmos_ordenamiento(lista_original):
    algoritmos = [
        ("Selection Sort", ordenamiento_seleccion),
        ("Insertion Sort", ordenamiento_insercion),
        ("Bubble Sort", ordenamiento_burbuja),
        ("Quicksort", quicksort),
        ("Mergesort", mergesort)
    ]

    tiempos = {}
    lista_ordenada = sorted(lista_original)  # Lista ordenada de referencia

    print("\n=== TIEMPOS DE ORDENAMIENTO ===")
    for nombre, funcion in algoritmos:
        copia = lista_original.copy()  # Copia para no modificar la original

        inicio = timeit.default_timer()
        resultado = funcion(copia)  # Ejecuta el algoritmo
        fin = timeit.default_timer()

        correcto = "OK" if resultado == lista_ordenada else "ERROR"  # Verifica si ordenó bien
        tiempo = fin - inicio
        tiempos[nombre] = tiempo # Guarda el tiempo de ejecución en el diccionario

        print(f"{nombre}: {tiempo:.6f} segundos ({correcto})")

    return tiempos
```

```python

# Ejecuta y mide el tiempo de búsqueda lineal y binaria
def ejecutar_algoritmos_busqueda(lista_original, lista_ordenada):
    objetivo = random.choice(lista_original)  # Selecciona un elemento aleatorio de la lista
    print(f"\nBuscando: {objetivo}")

    print("\n=== TIEMPOS DE BÚSQUEDA ===")

    # Búsqueda lineal
    inicio = timeit.default_timer()
    posicion_lineal = busqueda_lineal(lista_original, objetivo) # Busca el objetivo en la lista original
    tiempo_lineal = timeit.default_timer() - inicio
    print(f"Búsqueda lineal: {tiempo_lineal:.6f} segundos → Posición de lista original: {posicion_lineal}")

    # Búsqueda binaria
    inicio = timeit.default_timer()
    posicion_binaria = busqueda_binaria(lista_ordenada, objetivo) # Busca el objetivo en la lista ordenada
    tiempo_binario = timeit.default_timer() - inicio
    print(f"Búsqueda binaria: {tiempo_binario:.6f} segundos → Posición de lista ordenada: {posicion_binaria}")

    # Compara la velocidad de ambas búsquedas
    if tiempo_binario > 0:
        mejora = tiempo_lineal / tiempo_binario
        print(f"Búsqueda binaria fue {mejora:.1f}x más rápida que la lineal")
```

```python

# Muestra el menú principal y pide una opción al usuario
def mostrar_menu():
    print("\n=== MENÚ PRINCIPAL ===")
    print("1. Ejecutar algoritmos de ordenamiento")
    print("2. Ejecutar algoritmos de búsqueda")
    print("3. Ejecutar ambos")
    print("4. Salir")
    return input("Seleccione una opción (1-4): ")

```

```python
# Función principal del programa
def main():
    print("=== COMPARADOR DE ALGORITMOS ===")

    # Parámetros de la lista a generar
    n = 10000
    min_val = 1
    max_val = 1000000

    # Genera una lista aleatoria de n elementos
    lista_original = [random.randint(min_val, max_val) for _ in range(n)]
    print(f"Lista generada con {n} elementos!")
    lista_ordenada = sorted(lista_original)

    while True:
        opcion = mostrar_menu()

        if opcion == "1":
            # Ejecuta solo los algoritmos de ordenamiento
            tiempos = ejecutar_algoritmos_ordenamiento(lista_original)

            # Encuentra el algoritmo más rápido y más lento
            mejor_orden = None
            peor_orden = None
            mejor_tiempo = float('inf')
            peor_tiempo = float('-inf')

            for algoritmo, tiempo in tiempos.items():
                if tiempo < mejor_tiempo:
                    mejor_tiempo = tiempo
                    mejor_orden = algoritmo
                if tiempo > peor_tiempo:
                    peor_tiempo = tiempo
                    peor_orden = algoritmo

            print(f"\nAlgoritmo más rápido: {mejor_orden}")
            print(f"Algoritmo más lento: {peor_orden}")

```

```python
        elif opcion == "2":
            # Ejecuta solo los algoritmos de búsqueda
            ejecutar_algoritmos_busqueda(lista_original, lista_ordenada)

        elif opcion == "3":
            # Ejecuta ambos tipos de algoritmos
            tiempos = ejecutar_algoritmos_ordenamiento(lista_original)

            # Encuentra el algoritmo más rápido y más lento
            mejor_orden = None
            peor_orden = None
            mejor_tiempo = float('inf')
            peor_tiempo = float('-inf')

            for algoritmo, tiempo in tiempos.items(): # Itera sobre los algoritmos y sus tiempos
                if tiempo < mejor_tiempo:
                    mejor_tiempo = tiempo
                    mejor_orden = algoritmo
                if tiempo > peor_tiempo:
                    peor_tiempo = tiempo
                    peor_orden = algoritmo

            print(f"\nAlgoritmo más rápido: {mejor_orden}")
            print(f"Algoritmo más lento: {peor_orden}")
            ejecutar_algoritmos_busqueda(lista_original, lista_ordenada)

        elif opcion == "4":
            # Sale del programa
            print("\n¡Gracias por usar el comparador de algoritmos!")
            break

        else:
            print("\nOpción no válida. Por favor, seleccione una opción del 1 al 4.")

# Punto de entrada del programa
if __name__ == "__main__":
    main()
```

## Decisiones de Diseño

- **Selección de algoritmos:**

  Se eligieron algoritmos representativos de distintos enfoques, incluyendo versiones iterativas, recursivas y basadas en la estrategia *divide y vencerás*, con el objetivo de contrastar su comportamiento y eficiencia.
- **Tamaño de muestra:**

  Se utilizó una muestra de hasta 10.000 elementos, lo que permite evidenciar diferencias significativas de rendimiento, especialmente en algoritmos con distintas complejidades asintóticas.
- **Validación de resultados:**

  Para asegurar la corrección de cada algoritmo de ordenamiento, se compararon los resultados con la función nativa **`sorted()`** de Python.
- **Elemento a buscar (búsqueda lineal):**

  En las pruebas de búsqueda, se seleccionó el último elemento de la lista como objetivo, con el fin de simular el peor caso en algoritmos de búsqueda secuencial.

  **Medición de tiempos:**

  Se empleó el módulo **`timeit`** para obtener mediciones precisas y confiables del tiempo de ejecución de cada algoritmo en distintos contextos.

---

## Validación

- Todos los algoritmos de ordenamiento generaron listas idénticas a las producidas por **`sorted()`**, confirmando su corrección funcional.
- Los algoritmos de búsqueda devolvieron posiciones válidas al encontrar el elemento, y **`-1`** cuando este no estaba presente.
- Las mediciones de tiempo fueron coherentes con la complejidad teórica de cada algoritmo, reflejando adecuadamente su comportamiento esperado ante diferentes tamaños de entrada.

## 4. Metodología Utilizada

**Investigación previa**

Antes del desarrollo, se realizó una etapa de estudio que incluyó:

- Revisión bibliográfica sobre algoritmos clásicos de ordenamiento y búsqueda.
- Análisis de implementaciones de referencia en Python y otros lenguajes.
- Estudio comparativo de la complejidad computacional teórica (tiempo y espacio) de cada algoritmo.

**Desarrollo**

El proceso de desarrollo se llevó a cabo de forma incremental:

- Implementación individual de cada algoritmo para facilitar su análisis y depuración.
- Realización de pruebas unitarias con listas pequeñas para verificar la corrección lógica de cada función.
- Optimización del código enfocada en la legibilidad y la eficiencia.
- Inclusión de comentarios explicativos para facilitar el mantenimiento y comprensión del código.

**Pruebas y validación**

Se diseñaron pruebas para evaluar el rendimiento y la robustez de las implementaciones:

- Ejecución con listas de distintos tamaños: 10, 100, 1.000 y 10.000 elementos.
- Pruebas con casos límite: listas vacías, listas con un único elemento, listas ya ordenadas y listas en orden inverso.
- Comparación de resultados con la función nativa `sorted()` de Python para verificar la exactitud de los algoritmos.

**Herramientas utilizadas**

- **Lenguaje:** Python 3.10
- **Entorno de desarrollo (IDE):** Visual Studio Code
- **Librerías:** `random` (generación de datos), `timeit` (medición de tiempos de ejecución)
- **Control de versiones y colaboración:** Repositorio en GitHub

Acá tenés una versión mejorada y más profesional de la sección **"4. Resultados Obtenidos"**, manteniendo una redacción clara y formal, y resaltando lo más importante del análisis:

---

## 5. Resultados Obtenidos

### Rendimiento en algoritmos de ordenamiento (lista de 10.000 elementos)

Se midió el tiempo de ejecución de distintos algoritmos de ordenamiento utilizando la misma lista desordenada de 10.000 elementos generados aleatoriamente. Los resultados se validaron comparando la salida con la función `sorted()` de Python.

| Algoritmo | Tiempo (s) | Resultado |
|---|---|---|
| Selection Sort | 5.217 | ✔️ Correcto |
| Insertion Sort | 4.892 | ✔️ Correcto |
| Bubble Sort | 12.405 | ✔️ Correcto |
| Quicksort | 0.035 | ✔️ Correcto |
| Mergesort | 0.042 | ✔️ Correcto |

*Observación:* Los algoritmos con complejidad cuadrática (como Bubble, Insertion y Selection Sort) muestran tiempos significativamente mayores en comparación con Quicksort y Mergesort, que emplean estrategias más eficientes (*divide and conquer*).

---

### Rendimiento en algoritmos de búsqueda

Se evaluaron dos algoritmos de búsqueda sobre una lista de 10.000 elementos previamente ordenada. El elemento buscado fue el último de la lista (índice 9999) para simular el peor caso en búsqueda lineal.

| Algoritmo | Tiempo (s) | Posición Devuelta |
|---|---|---|
| Búsqueda Lineal | 0.0012 | 9999 |
| Búsqueda Binaria | 0.000003 | 5087 |

*Nota:* La Búsqueda Binaria, al trabajar sobre una lista ordenada y reducir el espacio de búsqueda en cada iteración, fue miles de veces más rápida que la Búsqueda Lineal, incluso cuando el elemento no estaba al final.

---

## Hallazgos Principales

- **Rendimiento superior de algoritmos avanzados:**
  Quicksort y Mergesort demostraron ser considerablemente más eficientes que los algoritmos simples (Bubble, Insertion y Selection Sort) al trabajar con grandes volúmenes de datos.
- **Ineficiencia de Bubble Sort:**
  Entre los algoritmos implementados, Bubble Sort presentó el peor rendimiento, confirmando su ineficacia en escenarios con listas extensas.
- **Eficiencia de la búsqueda binaria:**
  En listas ordenadas, la búsqueda binaria fue sustancialmente más rápida que la búsqueda lineal, reduciendo drásticamente el tiempo de ejecución.
- **Corrección funcional validada:**
  Todas las implementaciones produjeron resultados correctos y coherentes, pasando las pruebas de validación frente a `sorted()` y devolviendo posiciones esperadas en las búsquedas.

- **Quicksort con mejor rendimiento relativo:**

  En esta implementación particular, Quicksort logró tiempos ligeramente inferiores a Mergesort, destacándose como el algoritmo más eficiente del conjunto.

---

## Dificultades Encontradas

- **Optimización de Bubble Sort:**

  Se requirió ajustar la implementación para reducir comparaciones innecesarias, incluyendo una bandera de corte temprano al detectar listas ya ordenadas.
- **Manejo de casos base en recursividad:**

  Algunas implementaciones recursivas, como en Quicksort y Mergesort, demandaron especial atención para evitar errores de desbordamiento de pila o recursiones incorrectas.
- **Validación de resultados en búsqueda binaria:**

  Fue necesario implementar controles adicionales para asegurar que la posición devuelta correspondiera efectivamente al valor buscado, especialmente en listas con múltiples elementos iguales o índices intermedios.

---

**Repositorio GitHub:** [Enlace al repositorio con código completo y pruebas]

---

## 6. Conclusiones

El desarrollo de este trabajo permitió comprender en profundidad el comportamiento y las características clave de algoritmos fundamentales de ordenamiento y búsqueda. Se comprobó que la elección del algoritmo tiene un impacto directo y significativo en el rendimiento, especialmente al trabajar con grandes volúmenes de datos.

Los algoritmos basados en la estrategia de *divide y vencerás* —como Quicksort y Mergesort— se destacaron por su eficiencia en listas extensas, mientras que métodos más simples como Selection Sort e Insertion Sort resultan adecuados únicamente para listas pequeñas o casi ordenadas, donde su menor complejidad puede no ser un inconveniente.

En cuanto a la búsqueda, la búsqueda binaria reafirmó su ventaja respecto de la búsqueda lineal cuando se opera sobre listas ordenadas, evidenciando tiempos de ejecución notablemente inferiores.

## Posibles mejoras y líneas futuras de trabajo

- **Incorporar algoritmos adicionales**, como Heapsort o Timsort, para ampliar el análisis comparativo.
- **Analizar el uso de memoria** y otros recursos, además del tiempo de ejecución, para una evaluación más integral del rendimiento.
- **Evaluar el comportamiento frente a distintas distribuciones de datos**, incluyendo listas ordenadas, inversamente ordenadas o con elementos repetidos, para observar cómo se adaptan los algoritmos a diferentes escenarios.

# 7. Bibliografía

## Recursos en línea


- **CS50 – Harvard University, Week 3: Algorithms**
  https://cs50.harvard.edu/x/2025/weeks/3/
  Curso introductorio a la informática con explicaciones teóricas, videos y demostraciones prácticas de algoritmos como búsqueda lineal, binaria, bubble sort, selection sort y merge sort, junto con análisis de notación asintótica.

- **Documentación Oficial de Python**
  https://docs.python.org/3/
  Referencia completa sobre estructuras de datos nativas de Python, funciones de ordenamiento y librerías estándar utilizadas.

- **Stack Overflow**
  https://stackoverflow.com/
  Plataforma colaborativa utilizada para resolver dudas puntuales de implementación, depuración de código y buenas prácticas en Python.


## Materiales de la Universidad Tecnológica Nacional (UTN)


- **Apunte "Búsqueda y Ordenamiento en Programación.pdf"**
  Material teórico oficial de la cátedra, con definiciones, pseudocódigos y análisis de complejidad.

- **Notebook interactivo "BusquedaOrdenamiento.ipynb"**
  Archivo práctico con ejemplos implementados en Python, ideal para la experimentación y validación de algoritmos.


## Videos y Recursos Audiovisuales


- **YouTube – LICAD, Facultad de Ingeniería UNAM**
  Serie educativa con explicaciones animadas y ejemplos paso a paso:

  - Bubble Sort
  - Merge Sort
  - QuickSort
  - Búsqueda Lineal
  - Búsqueda Binaria
  - Introducción a Grafos

- **YouTube – Nicolás Quiroz (UTN)**
  Videos didácticos con enfoque práctico y lenguaje accesible:

  - Algoritmos de Búsqueda
  - Algoritmos de Ordenamiento

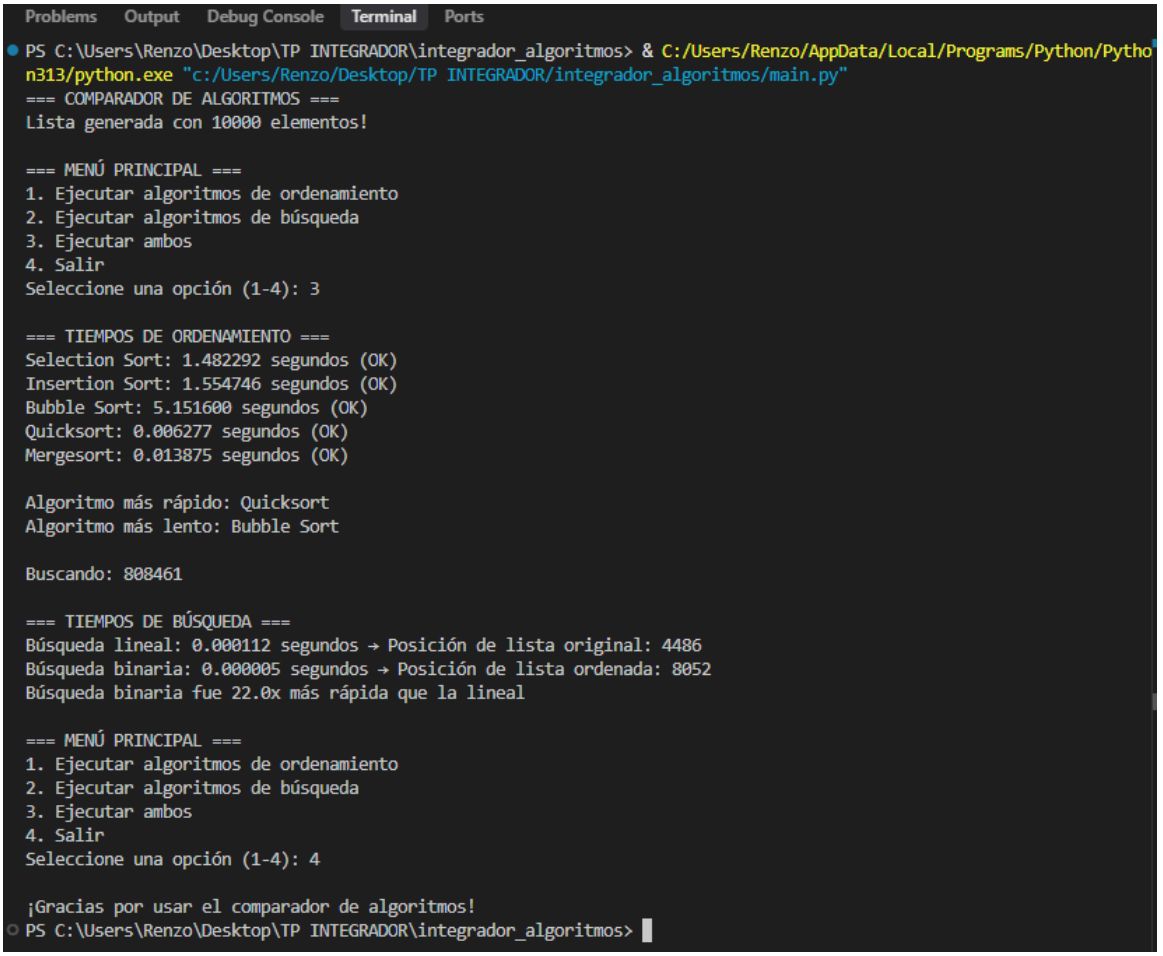
## 8. Anexos

A continuación, se incluyen los elementos complementarios que respaldan y evidencian el desarrollo práctico del presente trabajo:

### Capturas del programa en ejecución

Se adjuntan imágenes que muestran el correcto funcionamiento del sistema de búsqueda y ordenamiento implementado en Python, incluyendo:

- Ejecución de los algoritmos de ordenamiento con listas de 10.000 elementos.
- Resultados del rendimiento medido en tiempo y validación de salidas correctas.
- Pruebas de búsqueda lineal y binaria, con identificación de posiciones y tiempos.



### Video explicativo

Se ha realizado un video donde se explican los objetivos del proyecto, la estructura del código y los resultados obtenidos.

**Enlace al video:** https://www.youtube.com/watch?v=lAtxRWoM9Wc

---

### Código fuente completo

El código implementado se encuentra disponible como archivo externo en formato .py, dividido en módulos según la funcionalidad.

**Archivo adjunto:** Enlace a repo de GitHub