Trabajo Integrador - Búsqueda y Ordenamiento - Cátedra de Programación I

Datos Generales

- Algoritmo de búsqueda y orden en Python
- Alumnos:
  - Rodrigo Daniel Montes Sare – rodrii.montesrms@gmail.com
  - Gabriel Maximiliano Montaña - gabrielm.montana@gmail.com
- Materia: Programación I
- Profesor/a: Julieta Trapé
- Fecha de Entrega: 09/06/2025

Índice

---

1. Introducción

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación, ya que permiten localizar y organizar información de manera eficiente. A medida que crece la cantidad de datos, se vuelve imprescindible implementar métodos que reduzcan el tiempo y los recursos necesarios para su procesamiento. En este trabajo exploramos distintos tipos de algoritmos de búsqueda (lineal y binaria) y ordenamiento (burbuja, selección, inserción, mezcla y rápido), analizando sus características, ventajas, desventajas, y ejemplos de implementación en Python.

---

2. Marco Teórico

Algoritmos de Búsqueda

En programación, los métodos de búsqueda son técnicas fundamentales que permiten localizar información específica dentro de un conjunto de datos, como listas, arreglos o bases de datos. La eficiencia en la búsqueda es clave en cualquier sistema, ya que a medida que la cantidad de información crece, encontrar un elemento manualmente se vuelve inviable. Los algoritmos de búsqueda automatizan esta tarea y optimizan los recursos del sistema.

Existen diferentes algoritmos de búsqueda, cada uno con sus ventajas y limitaciones. La elección del método adecuado depende de factores como el tamaño del conjunto de datos, si están ordenados o no, y la frecuencia de las búsquedas.

Búsqueda Lineal

La búsqueda lineal es un algoritmo que localiza un valor específico dentro de una lista recorriendo cada elemento uno por uno. Comienza por el primer elemento, lo compara con el valor objetivo, y continúa hasta encontrarlo o llegar al final de la lista.

Este método no requiere que los datos estén ordenados, por lo que se utiliza principalmente en listas desordenadas. Es ideal para conjuntos de datos pequeños o cuando se requiere una búsqueda puntual.

Ventajas:

- Muy fácil de entender e implementar.
- Funciona en cualquier estructura secuencial.

Desventajas:

- Es ineficiente en listas grandes, ya que compara elemento por elemento. Su complejidad temporal es O(n).

Implementaciones en Python:

Método iterativo:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Método recursivo:

```
def linear_search_recursive(arr, target, index=0):
    if index >= len(arr):
        return -1
    if arr[index] == target:
        return index
    return linear_search_recursive(arr, target, index+1)
```

Cuando usarla: listas chicas, búsqueda puntual, datos desordenados.

Cuando evitarla: en conjuntos de datos grandes, ya que se vuelve ineficiente.

Complejidad:

- Tiempo: O(n)
- Espacio: O(1) para la versión iterativa y O(n) para la versión recursiva.

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para encontrar un valor en listas ordenadas. Su idea principal es dividir el conjunto a la mitad en cada paso. Esto permite reducir significativamente la cantidad de comparaciones necesarias.

Es fundamental que los datos estén ordenados para que el algoritmo funcione correctamente. De lo contrario, los resultados serán incorrectos.

Ventajas:

- Muy eficiente en listas grandes, con complejidad O(log n).
- Ideal para búsquedas frecuentes sobre datos ordenados.

Desventajas:

- Solo funciona en listas ordenadas.
- Puede presentar errores si no se maneja correctamente el índice medio.

Implementación típica en Python:

Método iterativo:

```
def binary_search_iterative(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Explicación: el algoritmo comienza con los límites izquierdo y derecho. Calcula el punto medio, compara con el objetivo, y si no coincide, descarta la mitad en la que no puede estar. Repite este proceso hasta encontrar el valor o agotar las posibilidades.

Método recursivo:

```
    def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1,
right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)
```

Complejidad:

- Tiempo: O(log n)
- Espacio: O(1) para la versión iterativa y O(log n) para la versión recursiva.

Cuando usarla: listas ordenadas, búsquedas frecuentes, grandes volúmenes de datos.

Errores comunes:

- Usar búsqueda binaria en listas desordenadas.
- No manejar el caso en el que el valor no se encuentra.
- Olvidar que los índices en Python empiezan en 0.

Resumen: la búsqueda lineal es útil para listas pequeñas o sin ordenar, mientras que la búsqueda binaria es la mejor opción para listas grandes y ordenadas donde se busca eficiencia.

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento permiten reorganizar elementos en una lista según un criterio específico, como de menor a mayor. Son fundamentales porque permiten búsquedas más rápidas y mejoran el procesamiento y análisis de datos.

Beneficios de ordenar datos:

- Permite aplicar algoritmos de búsqueda más eficientes como la búsqueda binaria.
- Facilita el análisis y visualización de datos.
- Optimiza operaciones como fusiones, filtros y eliminación de duplicados.

Ordenamiento por burbuja (Bubble Sort)

Este método compara cada elemento con su siguiente y los intercambia si están en el orden incorrecto. Es simple pero poco eficiente para listas grandes.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
```

```
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Ordenamiento por selección (Selection Sort)

Encuentra el valor mínimo en la lista y lo intercambia con el primer elemento.

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Ordenamiento por inserción (Insertion Sort)

Inserta cada elemento en su posición correcta dentro de una lista ordenada previamente.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

Ordenamiento rápido (Quick Sort)

Divide la lista en dos partes usando un pivote y ordena recursivamente.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    less = [x for x in arr[1:] if x <= pivot]
```

```
    greater = [x for x in arr[1:] if x > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)
```

Ordenamiento por mezcla (Merge Sort)

Divide la lista en mitades, las ordena y luego las fusiona.

```
def mergeSort(nums):
    if len(nums) <= 1:
        return nums
    mid = len(nums) // 2
    left_list = mergeSort(nums[:mid])
    right_list = mergeSort(nums[mid:])
    return merge(left_list, right_list)


def merge(left_list, right_list):
    sorted_list = []
    i = j = 0
    while i < len(left_list) and j < len(right_list):
        if left_list[i] <= right_list[j]:
            sorted_list.append(left_list[i])
            i += 1
        else:
            sorted_list.append(right_list[j])
            j += 1
    sorted_list.extend(left_list[i:])
    sorted_list.extend(right_list[j:])
    return sorted_list
```

3. Marco Practico

Descripción del problema

Para aplicar los algoritmos de búsqueda y ordenamiento estudiados, desarrollamos una simulación de gestión de productos para un negocio genérico. El sistema permite generar una lista de productos ficticios, cada uno con un código, nombre y precio, y realizar sobre dicha lista operaciones de **ordenamiento** y **búsqueda** por cualquiera de estos tres campos.

El objetivo principal es observar cómo afectan distintos algoritmos al rendimiento del sistema y cómo elegir la técnica adecuada según el contexto. El proyecto incluye medición de tiempos de ejecución, validación funcional y comparación de resultados.

Funcionalidades implementadas

- Generación dinámica de listas de productos (con cantidad elegible por el usuario).
- Ordenamiento por código, nombre o precio, utilizando el algoritmo **Merge Sort**.
- Búsqueda de un producto por campo, utilizando **búsqueda binaria** (cuando la lista está ordenada adecuadamente).
- Medición de tiempos de ejecución para evaluar eficiencia.
- Validación de la búsqueda (mostrar resultados encontrados).
- Posibilidad de buscar **todos** los productos que coincidan con un valor (ej: todos los productos con el mismo precio).

Funciones principales

A continuación, se resumen los fragmentos más representativos del código desarrollado.

Generación de productos:

```
def generar_productos(n):

    return [{'codigo': i, 'nombre': f'Prod{i}', 'precio':
random.randint(100, 1000)} for i in range(n)]
```

Genera una lista de n productos, cada uno con código numérico, nombre automático y precio aleatorio.

Ordenamiento con Merge Sort

```
def merge_sort(lista, campo):
    if len(lista) <= 1:
        return lista
    mid = len(lista) // 2
    izquierda = merge_sort(lista[:mid], campo)
    derecha = merge_sort(lista[mid:], campo)
    return merge(izquierda, derecha, campo)


def merge(izq, der, campo):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i][campo] <= der[j][campo]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado.extend(izq[i:])
    resultado.extend(der[j:])
    return resultado
```

Se ordena por el campo seleccionado (nombre, precio, código) y se mide el tiempo de ejecución.

Búsqueda binaria:

```
def busqueda_binaria(lista, campo, valor):
    izq, der = 0, len(lista) - 1
    while izq <= der:
        medio = (izq + der) // 2
        actual = lista[medio][campo]
        if str(actual) == str(valor):
            return lista[medio]
        elif str(actual) < str(valor):
            izq = medio + 1
        else:
            der = medio - 1
    return None
```

Permite encontrar un producto cuando la lista está ordenada por el campo correspondiente. Se incluye también la variante busqueda_binaria_todos para múltiples coincidencias.

Decisiones de diseño

Elección del algoritmo de ordenamiento: Merge Sort

Se optó por implementar el algoritmo **Merge Sort** para ordenar la lista de productos por un campo elegido por el usuario (código, nombre o precio). La decisión se basó principalmente en su **eficiencia algorítmica y comportamiento estable ante listas grandes**.

- **Complejidad:** Merge Sort tiene complejidad **O(n log n)** en todos los casos (mejor, promedio y peor), lo que lo hace predecible y confiable, incluso con grandes volúmenes de datos.
- **Comparación con otros algoritmos simples:**
    - Bubble Sort, Selection Sort e Insertion Sort tienen complejidad **O(n^2)**, por lo que se vuelven extremadamente lentos a medida que la lista crece.
    - En pruebas realizadas, estos métodos tardaron más de 7 a 26 segundos con 10.000 elementos, mientras que Merge Sort los ordenó en menos de **0.05 segundos**.
- **Por qué no Quicksort:** Si bien Quicksort suele ser el más rápido en promedio, su peor caso es también O(n^2). Merge Sort, al mantener un rendimiento constante y eficiente, fue preferido para este contexto.
- **Paralelismo con Python:** Python utiliza **Timsort** en su método interno sorted(), que combina Merge Sort e Insertion Sort, justamente por sus buenos resultados en la práctica.

Conclusión: Para listas medianas o grandes (más de 1.000 elementos), Merge Sort resultó ser el algoritmo más adecuado. Los otros métodos son útiles para comprender el funcionamiento de los algoritmos, pero no se recomiendan en sistemas reales que requieren rendimiento.

```
¿Cuántos productos querés generar? (ej: 1000): 10000
¿Por qué campo querés ordenar? (nombre, precio, codigo):
1. Nombre
2. Precio
3. Codigo
Elegí una opción (1-3): 2

Midiendo tiempos de ordenamiento por precio
Burbuja: 26.5935 seg | Selección: 10.1577 seg | Inserción: 7.2411 seg | MergeSort: 0.0474 seg
```

```
¿Cuántos productos querés generar? (ej: 1000): 10000
¿Por qué campo querés ordenar? (nombre, precio, codigo):
1. Nombre
2. Precio
3. Codigo
Elegí una opción (1-3): 1

Midiendo tiempos de ordenamiento por nombre
Burbuja: 13.3207 seg | Selección: 14.7041 seg | Inserción: 1.7058 seg | MergeSort: 0.0842 seg
```

```
¿Cuantos productos queres generar? (ej: 1000): 10000
¿Por qué campo querés ordenar? (nombre, precio, codigo):
1. Nombre
2. Precio
3. Codigo
Elegí una opción (1-3): 3

Midiendo tiempos de ordenamiento por codigo
Burbuja: 10.9498 seg | Selección: 10.3753 seg | Inserción: 0.0030 seg | MergeSort: 0.0325 seg
```

Elección del algoritmo de búsqueda: Búsqueda Binaria

El sistema permite buscar productos por distintos campos, utilizando **búsqueda binaria** siempre que la lista esté ordenada por ese mismo campo. Si no lo está, el sistema advierte al usuario.

Se utilizaron **dos variantes** de búsqueda binaria:

1. **Buscar uno solo:** útil cuando se busca por campos únicos como el código del producto o cuando se necesita verificar rápidamente si existe un valor.
2. **Buscar todos:** ideal cuando se buscan valores repetidos, como múltiples productos con el mismo precio.

Ventajas frente a la búsqueda lineal:

- La búsqueda lineal tiene complejidad O(n), ya que recorre todos los elementos.
- La búsqueda binaria tiene complejidad O(log n) y en la variante "todos" se suma O(k), siendo k la cantidad de coincidencias.
- En pruebas realizadas, la binaria respondió en **milisegundos (incluso microsegundos)**, mostrando una diferencia sustancial frente a la búsqueda lineal.

Decisión de diseño adicional: se incluyó una opción para que el usuario indique si desea encontrar un solo resultado o **todos** los que coincidan con el valor buscado. Esta distinción permite optimizar aún más el tiempo de respuesta dependiendo de la necesidad puntual.

Ejemplos prácticos:

- Buscar por código: se espera un único resultado → se usa binaria simple.
- Buscar por precio: puede haber múltiples coincidencias → se usa binaria “todos”.

- Buscar por nombre: si no se repite, binaria simple basta; si hay nombres repetidos, se puede usar la versión “todos”.

Conclusión práctica:

- En listas grandes, usar **búsqueda binaria** siempre que sea posible permite acelerar significativamente las búsquedas.
- La variante "buscar todos" demuestra un excelente rendimiento cuando hay múltiples coincidencias consecutivas.
- La clave para aprovechar la búsqueda binaria es **asegurar que la lista esté ordenada por el campo de búsqueda**, condición que el sistema valida automáticamente antes de

```
¿Por qué campo querés buscar?
1. Nombre
2. Precio
3. Codigo
Elegí una opción (1-3): 2
Ingresá el valor a buscar en 'precio': 106
Desea buscar todos? (s/n)s
Búsqueda Lineal: 0.002028 seg
Se encontraron 9 resultados:

Primeros 9 productos ordenados por 'precio':
{'codigo': 3395, 'nombre': 'Prod3395', 'precio': 106}
{'codigo': 3461, 'nombre': 'Prod3461', 'precio': 106}
{'codigo': 3637, 'nombre': 'Prod3637', 'precio': 106}
{'codigo': 5319, 'nombre': 'Prod5319', 'precio': 106}
{'codigo': 5389, 'nombre': 'Prod5389', 'precio': 106}
{'codigo': 6676, 'nombre': 'Prod6676', 'precio': 106}
{'codigo': 8025, 'nombre': 'Prod8025', 'precio': 106}
{'codigo': 9081, 'nombre': 'Prod9081', 'precio': 106}
{'codigo': 9104, 'nombre': 'Prod9104', 'precio': 106}
Desea buscar todos? (s/n)s
Búsqueda Binaria (lista ordenada): 0.000000 seg
Se encontraron 9 resultados:

Primeros 9 productos ordenados por 'precio':
{'codigo': 3395, 'nombre': 'Prod3395', 'precio': 106}
{'codigo': 3461, 'nombre': 'Prod3461', 'precio': 106}
{'codigo': 3637, 'nombre': 'Prod3637', 'precio': 106}
{'codigo': 5319, 'nombre': 'Prod5319', 'precio': 106}
{'codigo': 5389, 'nombre': 'Prod5389', 'precio': 106}
{'codigo': 6676, 'nombre': 'Prod6676', 'precio': 106}
{'codigo': 8025, 'nombre': 'Prod8025', 'precio': 106}
{'codigo': 9081, 'nombre': 'Prod9081', 'precio': 106}
{'codigo': 9104, 'nombre': 'Prod9104', 'precio': 106}
```

ejecutarla.

4. Metodología Utilizada

El desarrollo del trabajo integrador se realizó siguiendo una serie de etapas que incluyeron investigación, diseño, implementación, prueba y análisis de resultados. El objetivo fue comprender en profundidad los algoritmos de búsqueda y ordenamiento, y aplicarlos de forma práctica en un sistema funcional.

Investigación previa

Como primer paso, se realizó una investigación bibliográfica sobre los algoritmos de búsqueda y ordenamiento más comunes. Se consultaron fuentes en línea como **DataCamp** y "**El Aula Virtual"**, además de documentación oficial de Python y guías facilitadas por los profesores

Diseño e implementación

Se diseñó una aplicación en **Python** que simula la gestión de productos de un negocio. Cada producto contiene un código, nombre y precio. A partir de este diseño:

- Se implementaron los algoritmos de ordenamiento **Bubble Sort**, **Selection Sort**, **Insertion Sort** y **Merge Sort**.
- Para la búsqueda, se desarrollaron versiones **lineales** y **binarias**, incluyendo variantes para encontrar **uno solo** o **todos los elementos** coincidentes.
- Se integraron funciones para medir el tiempo de ejecución de cada algoritmo, con el fin de evaluar su rendimiento bajo distintas condiciones.

El código se desarrolló utilizando el IDE **Visual Studio Code y Intellij**, y se hizo uso de librerías estándar de Python como random para generar datos y time para medir el rendimiento.

Pruebas y análisis

Una vez implementados los algoritmos, se realizaron múltiples pruebas con listas de distintos tamaños: desde 10 elementos hasta más de 10.000. Se midió el tiempo de ejecución de cada algoritmo para evaluar su eficiencia, y se documentaron los resultados obtenidos (Las pruebas se pueden volver a observar en el marco practico).

Durante las pruebas se observaron diferencias notables en los tiempos de ordenamiento, particularmente en listas grandes. Esto permitió justificar la elección final de **Merge Sort** como algoritmo óptimo para el caso práctico, debido a su rendimiento constante y eficiente. De forma similar, se observó que la **búsqueda binaria** resultó mucho más rápida que la búsqueda lineal cuando se cumplía el requisito de tener la lista ordenada.

Herramientas y recursos utilizados

- **Lenguaje:** Python
- **Entorno de desarrollo:** Visual Studio Code y Intellij
- **Librerías estándar utilizadas:** random, time
- **Control de versiones:** Git (repositorio compartido en GitHub)

Trabajo colaborativo

El trabajo fue desarrollado en equipo, distribuyendo las tareas de la siguiente manera:

- **Investigación y redacción del marco teórico**
- **Diseño e implementación del código en Python**
- **Pruebas, capturas y validación del caso práctico**
- **Documentación y redacción final del informe**

5. Resultados Obtenidos

El desarrollo del caso práctico permitió aplicar y validar los conocimientos adquiridos sobre algoritmos de búsqueda y ordenamiento. A través de la implementación en Python, se logró construir una herramienta funcional capaz de:

- Generar automáticamente listas de productos con campos código, nombre y precio.
- Ordenar dichas listas por cualquier campo elegido utilizando el algoritmo Merge Sort.
- Buscar productos por distintos criterios utilizando búsqueda binaria, siempre que la lista estuviera previamente ordenada por el campo correspondiente.

Casos de prueba realizados

Se realizaron pruebas con listas de:

- 10 elementos (listas pequeñas).
- 1.000 y 10.000 elementos (listas grandes y realistas).
- Se intentó generar una lista con 50.000 productos, pero las computadoras utilizadas **colapsaron por uso excesivo de memoria y CPU**, lo que obligó a reducir la escala de pruebas. Este error evidenció la importancia de adaptar el procesamiento a los recursos disponibles.

Evaluación de rendimiento

Durante las pruebas de rendimiento, se midió el tiempo de ejecución de distintos algoritmos de ordenamiento. Los resultados más destacados fueron:

- **Merge Sort:** ordenó 10.000 elementos en menos de **0.005 segundos**.
- **Bubble Sort:** tardó más de **26 segundos** en la misma tarea.
- **Selection Sort:** aproximadamente **10 segundos**.

- **Insertion Sort:** alrededor de **7 segundos**.

Estos resultados confirmaron que Merge Sort es claramente superior para listas grandes, cumpliendo con una complejidad de O(n log n), frente a los otros métodos que presentan $O(n^2)$.

En cuanto a la búsqueda:

- **La búsqueda binaria** demostró una velocidad casi instantánea (menos de **0.000001 segundos** en muchos casos), siempre y cuando la lista estuviera correctamente ordenada por el campo correspondiente.
- **La búsqueda binaria "todos"**, que encuentra todas las coincidencias de un valor, también funcionó de forma eficiente gracias a la combinación de búsqueda logarítmica y recorrido lineal mínimo.
- Se implementó un sistema de validación que impide usar búsqueda binaria si la lista no está correctamente ordenada.

Errores

- Se identificó y resolvió un problema importante al intentar procesar una lista excesivamente grande (50.000 productos), que causaba la detención del sistema. Esto llevó a limitar el tamaño de las listas de prueba y optimizar el código para eficiencia.
- Olvidar que los índices en Python empiezan en 0.
- Usar búsqueda binaria en listas NO ordenadas.
- No manejar el valor de retorno cuando el elemento no se encuentra.
- También se mejoró el control de errores al validar que la búsqueda binaria solo se ejecute si la lista está ordenada por el campo de búsqueda, evitando resultados incorrectos.

6. Conclusiones

Este trabajo integrador permitió al grupo afianzar los conceptos teóricos de algoritmos de búsqueda y ordenamiento mediante su aplicación directa en un caso práctico. Más allá de la implementación técnica, se logró comprender en profundidad **cuándo y por qué conviene utilizar un algoritmo por sobre otro**, según el tipo de datos y el objetivo buscado.

Aprendizajes clave

- Se aprendió a implementar, probar y comparar distintos algoritmos en Python.
- Se entendió la importancia de la **eficiencia algorítmica** y cómo esta puede impactar en el rendimiento de una aplicación.
- Se comprobó que **elegir el algoritmo correcto** puede significar una mejora de cientos o miles de veces en tiempo de ejecución.

Utilidad del trabajo

Este tipo de conocimientos resulta fundamental para cualquier desarrollo que implique manejo de datos, desde bases de datos hasta aplicaciones web. Saber elegir el algoritmo correcto permite construir sistemas más rápidos, robustos y escalables. También sienta las bases para proyectos más complejos, como motores de búsqueda o inteligencia artificial.

Dificultades y soluciones

- Una de las principales dificultades fue la sobrecarga de recursos al generar listas demasiado grandes, lo que se resolvió limitando el tamaño y midiendo el rendimiento de manera más progresiva.
- También se ajustó el flujo del programa para mejorar la interacción con el usuario, evitando errores por búsquedas inválidas.

Posibles mejoras

- Implementar otros algoritmos
- Agregar una interfaz gráfica para mejorar la interacción del usuario.
- Exportar resultados y tiempos a un archivo para su análisis externo.
- Permitir ordenamiento múltiple (por ejemplo, primero por precio y luego por nombre).

7. Bibliografía

- DataCamp. (s.f.). *Linear Search in Python*. Recuperado de: https://www.datacamp.com/es/tutorial/linear-search-python
- EIPOS Grados. (s.f.). *Tipos de algoritmos de ordenación en Python*. Recuperado de: https://eiposgrados.com/blog-python/tipos-de-algoritmos-de-ordenacion-en-python/

---

8. Anexos

- Código fuente completo.
- Capturas de pantalla de pruebas realizadas.
- Comparación de tiempos de ejecución (si corresponde).
- Enlace al repositorio GitHub:

  https://github.com/gabito00/UTN-TUPAD-TPI