

# Trabajo Integrador - Propuesta de Investigación Cátedra de Programación I

---

## Título:

Investigación aplicada en Python

*Algoritmos de Búsqueda y Ordenamiento*

---

## Alumnos:

- Ricardo Pardo – *ricapardo84@gmail.com*
- Damián Ignacio Nogueira – *damian.nogueira@outlook.com*

## Materia:

- Programación I

## Profesora:

- Julieta Trapé

## Fecha de entrega:

- 09/06/2025
- 

## Índice:

1. Introducción - Página 1
2. Marco Teórico - Páginas 2 - 6
3. Caso Práctico - Páginas 6 - 7
4. Metodología Utilizada - Páginas 8 - 19
5. Resultados Obtenidos - Páginas 19 - 24
6. Conclusiones Páginas - 24 - 28
7. Bibliografía - Página 28
8. Anexos - Páginas 29 - 33

## 1. Introducción

Los algoritmos de búsqueda y ordenamiento constituyen pilares esenciales en la gestión y organización de datos dentro de cualquier aplicación de software. Se eligió este tema debido a que:

- **Relevancia académica y práctica:** La eficiencia en la localización de información (búsqueda) y en la disposición ordenada de conjuntos de datos (ordenamiento) impacta directamente sobre el rendimiento de sistemas como bases de datos, motores de recomendación y estructuras de almacenamiento en memoria.
- **Importancia en la programación:** Comprender e implementar estos algoritmos permite al programador seleccionar la técnica adecuada según características del problema (por ejemplo, volumen de datos, posibilidad de acceso aleatorio, requerimientos de tiempo real), optimizando recursos computacionales.
- **Objetivos del trabajo:**
  1. Describir los principios teóricos de los métodos de búsqueda (Lineal y Binaria) y de ordenamiento (Bubble Sort, Insertion Sort y Selection Sort).
  2. Implementar en Python ejemplos prácticos que demuestren su funcionamiento y diferencias de eficiencia.
  3. Evaluar comparativamente su rendimiento mediante casos de prueba representativos.

El desarrollo de este trabajo permitirá afianzar conceptos teóricos aprendidos en clase con ejercicios de codificación y medición real de tiempos de ejecución, ofreciendo una visión integral de cuándo y por qué aplicar cada algoritmo.

---

## 2. Marco Teórico

En este apartado se define y clasifica cada uno de los algoritmos de búsqueda y ordenamiento estudiados, describiendo su funcionamiento, complejidad y, cuando corresponda, su implementación en Python. A partir de este análisis elegiremos los que vamos a estudiar con detalle.

## 2.1 Algoritmos de Búsqueda

### 1. Búsqueda Lineal

- **Descripción:** Recorre secuencialmente cada elemento de la colección hasta encontrar el valor buscado o agotar la lista.
- **Pseudocódigo:**
  1. Para  $i$  desde 0 hasta  $n-1$
  2. Si  $arr[i] == target$
  3. Retornar  $i$
  4. Fin Si
  5. Fin Para
  6. Retornar  $-1$  (no encontrado)
- **Complejidad:**
  - Peor caso:  $O(n)$  (recorrer todos los elementos).
  - Mejor caso:  $O(1)$  (si está en la primera posición).

### 2. Búsqueda Binaria:

- **Requisito:** La lista debe estar previamente ordenada.
- **Descripción:** Divide el espacio de búsqueda en mitades, comparando el elemento central con el valor objetivo y descartando la mitad donde no puede estar.
- **Pseudocódigo:**

0. Inicializar  $low = 0$ ,  $high = n-1$
  1. Mientras  $low \leq high$
  2.      $mid = (low + high) // 2$
  3.     Si  $arr[mid] == target$ : retornar  $mid$
  4.     Si  $arr[mid] < target$ :  $low = mid + 1$
  5.     Sino:  $high = mid - 1$
  6. Fin Mientras
  7. Retornar  $-1$  (no encontrado)
- **Complejidad:**
    - Peor caso y promedio:  $O(\log n)$ .
    - Mejor caso:  $O(1)$  (si coincide con el elemento central en la primera comparación).

## 2.2 Algoritmos de Ordenamiento

### 1. Bubble Sort

- **Descripción:** Recorre repetidamente la lista comparando pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Cada pasada “burbujea” el mayor elemento restante hasta su posición final.
- **Pseudocódigo:**
  1. Para  $i$  desde 0 hasta  $n-1$
  2.     Para  $j$  desde 0 hasta  $n-i-2$
  3.         Si  $arr[j] > arr[j+1]$ : intercambiar  $arr[j]$  y  $arr[j+1]$

4. Fin Para

5. Fin Para

- **Complejidad:**

- Peor y promedio:  $O(n^2)$ .
- Mejor caso (ya ordenado):  $O(n)$ .

## 2. Insertion Sort

- **Descripción:** Construye una sublista ordenada al tomar cada elemento de la lista original e insertarlo en la posición correspondiente dentro de la sublista ya ordenada.

- **Pseudocódigo:**

0. Para  $i$  desde 1 hasta  $n-1$
1.  $key = arr[i], j = i-1$
2. Mientras  $j \geq 0$  y  $arr[j] > key$
3.  $arr[j+1] = arr[j], j = j-1$
4. Fin Mientras
5.  $arr[j+1] = key$
6. Fin Para

- **Complejidad:**

- Peor y promedio:  $O(n^2)$ .
- Mejor caso (lista ordenada):  $O(n)$ .

## 2. Selection Sort

- **Descripción:** Recorre la lista buscando el elemento mínimo, lo coloca al principio y repite el proceso para el resto de la lista.

- **Pseudocódigo:**

0. Para i desde 0 hasta  $n-2$
1.      $\text{min\_idx} = i$
2.     Para j desde  $i+1$  hasta  $n-1$
3.         Si  $\text{arr}[j] < \text{arr}[\text{min\_idx}]$ :  $\text{min\_idx} = j$
4.     Fin Para
5.     Intercambiar  $\text{arr}[i]$  y  $\text{arr}[\text{min\_idx}]$
6. Fin Para

- **Complejidad:**

- Siempre  $O(n^2)$ , independientemente del estado inicial de la lista.

### 2.3 Comparación de Complejidades

Algoritmo	Mejor Caso	Promedio	Peor Caso	Requisito Previo
<b>Búsqueda Lineal</b>	$O(1)$	$O(n)$	$O(n)$	Ninguno
<b>Búsqueda Binaria</b>	$O(1)$	$O(\log n)$	$O(\log n)$	Lista ordenada
<b>Bubble Sort</b>	$O(n)$	$O(n^2)$	$O(n^2)$	Ninguno
<b>Insertion Sort</b>	$O(n)$	$O(n^2)$	$O(n^2)$	Ninguno
<b>Selection Sort</b>	$O(n^2)$	$O(n^2)$	$O(n^2)$	Ninguno

Este fundamento conceptual sienta las bases para el desarrollo del caso práctico en Python, donde implementaremos **Búsqueda Binaria y Bubble Sort** y mediremos su desempeño.

### 3. Caso Práctico

En esta sección presentamos una aplicación sencilla de los algoritmos de ordenamiento y búsqueda en Python, habiendo seleccionado Búsqueda Binaria y Bubble Sort:

#### 3.1 Descripción del problema

Dado un conjunto desordenado de números enteros, el objetivo es:

1. **Ordenar** la lista mediante un algoritmo de ordenamiento (Bubble Sort).
2. **Buscar** un valor específico en la lista ya ordenada usando búsqueda binaria.

Este caso práctico permite ilustrar en la práctica la necesidad de tener la lista ordenada antes de aplicar búsqueda binaria y comparar la simplicidad de implementación frente a la eficiencia teórica.

#### 3.2 Explicación de decisiones de diseño

- **Elección de Bubble Sort:**
  - Nos resulta muy sencillo de implementar y entender, es ideal para listas pequeñas a fines didácticos.
  - Aunque su complejidad es cuadrática, permite ilustrar claramente el proceso de intercambio paso a paso.
- **Uso de Búsqueda Binaria:**
  - Seleccionada por su eficiencia logarítmica en listas ordenadas.
  - Demuestra la necesidad de ordenar la lista previamente, pues el algoritmo asume esa condición.
- **Estructura del programa:**
  - Separación en funciones para facilitar lectura y reutilización.
  - Comentarios detallados en cada bloque para explicar su propósito y complejidad.

### 3.4 Validación del funcionamiento

#### 1. Pruebas con diferentes entradas

- Listas con elementos repetidos, orden inverso y ya ordenadas.
- Búsqueda de valores en extremos (primera/última posición) y no presentes.

#### 2. Verificación de resultados

- Confirmar que bubble\_sort siempre produce una lista ordenada ascendente.
- Chequear que binary\_search retorna índices correctos o -1 cuando no existe el valor.

#### 3. Capturas y evidencias

- Se incluyen, en anexos, capturas de pantalla de la ejecución en consola y el repositorio Git con el código completo y la evidencia de pruebas.

(Agregar las capturas a la sección Anexo)

---

### 4. Metodología Utilizada

A continuación describimos el proceso exacto que seguimos. Explicamos cómo partimos del análisis teórico, pasamos por la redacción del pseudocódigo, hasta llegar



a la implementación, pruebas y validación de los algoritmos `bubble_sort` y `binary_search` tal como aparecen en nuestro archivo `tp_integrador_1.py`.

#### 4.1 Investigación previa

Para sentar las bases teóricas antes de escribir cualquier línea de código, realizamos los siguientes pasos:

##### 1. Revisión bibliográfica y apuntes de cátedra

- Consultamos el libro *Introduction to Algorithms* (Cormen et al.) para comprender en profundidad la lógica y la complejidad de los algoritmos de ordenamiento y búsqueda.
- Revisamos los apuntes de la materia Programación I alojados en el Aula Virtual de la UTN, sobre complejidad computacional, pseudocódigo de algoritmos y ejemplos teóricos.

##### 2. Recursos en línea y documentación oficial

- Visitamos la documentación oficial de Python (<https://docs.python.org/3/>) para aclarar dudas de sintaxis, docstrings y mejores prácticas de estilo (PEP 8).
- Consultamos tutoriales en YouTube, Khan Academy, GeeksforGeeks para comparar diferentes variantes de “Bubble Sort”, “Insertion Sort”, “Selection Sort” y “Binary Search”, de modo de entender las ventajas y desventajas de cada uno.

##### 3. Definición del alcance y objetivos

- Acordamos que, de los tres algoritmos de ordenamiento estudiados en clase, centraríamos el Caso Práctico en “Bubble Sort” por su sencillez didáctica, y complementaríamos con “Binary Search” para ilustrar la eficiencia en listas ordenadas.

- Decidimos que, en secciones posteriores (Resultados y Conclusiones), incluiríamos comparativas de tiempo de ejecución utilizando Python puro y, como posible trabajo futuro, quedaría la extensión a otros algoritmos más eficientes (por ejemplo, “Quick Sort” o “Merge Sort”).

## 4.2 Planificación y diseño

Una vez asentada la teoría, diagramamos y distribuimos las tareas antes de codificar:

### 1. Pseudocódigo:

- Elaboramos conjuntamente el pseudocódigo de “Bubble Sort”, especificando paso a paso el recorrido de la lista y los intercambios de pares adyacentes.
- Diseñamos un diagrama de flujo que muestra las dos iteraciones anidadas de “Bubble Sort” y el punto exacto donde se verifica si  $arr[j] > arr[j+1]$  para intercambiar valores.
- Asimismo, bosquejamos el pseudocódigo de “Binary Search”, asegurándonos de representar correctamente la inicialización de los punteros low y high, la comparación con el elemento medio y la actualización de los límites.

### 2. Selección de casos de prueba preliminares

Antes de comenzar a programar, listamos los escenarios que debíamos cubrir:

- Listas vacías ( $[]$ ) y listas de un solo elemento (por ejemplo,  $[10]$ ).
- Listas con elementos repetidos (p. ej.  $[2, 7, 2, 7, 2]$ ).
- Listas ya ordenadas ( $[1, 2, 3, 4, 5]$ ) e inversamente ordenadas ( $[5, 4, 3, 2, 1]$ ).

- Búsqueda de valores en posiciones extremas, intermedias y valores no presentes (p. ej. buscar 7 en [2, 7, 2, 7, 2] o buscar 100 en [5, 7, 23, 32, 34, 62]).

### 3. Distribución de tareas entre nosotros

- **Ricardo:** se encargó de implementar la función `bubble_sort(arr)`, comentar cada bloque de código, y elaborar los primeros casos de prueba para verificar el correcto ordenamiento.
- **Damián:** se ocupó de implementar `binary_search(arr, target)`, realizar pruebas de búsqueda en las listas ordenadas, y escribir el guión de análisis de tiempos de ejecución utilizando el módulo `timeit`.

### 4. Elección del entorno de desarrollo y control de versiones

- Decidimos usar **Visual Studio Code** como editor principal, activando la extensión de Python para linting (PEP 8) y depuración con breakpoints.
- Creamos un repositorio en **GitHub** llamado `Tp-ProgramacionI-Busqueda-Ordenamiento`. Ricardo trabajó en la rama `rama-bubble-sort` y Damián en `rama-binary-search`. Integramos cambios mediante Pull Requests y realizamos revisiones mutuas antes de fusionar a la rama principal (`main`). \*Debemos hacer esto\*

#### 4.3 Implementación en Python

Con el diseño aprobado, comenzamos a codificar en Python 3.10 siguiendo la estructura prevista:

##### 1. Estructura de archivos en el repositorio

- `tp_integrador_I.py`: contiene las dos funciones principales (`bubble_sort` y `binary_search`) y el bloque `if __name__ == "__main__":` que imprime la lista original, la lista ordenada y el resultado de la búsqueda.

- pruebas\_rendimiento.py: script separado para medir tiempos de ejecución en listas de distintos tamaños (10, 50, 100, 500, 1000 elementos) utilizando timeit.
- README.md: explica cómo clonar el repositorio, versiones de Python recomendadas y pasos para ejecutar los scripts.

## 2. Implementación de bubble\_sort(arr)

- Ricardo definió la función con el siguiente esquema:

```
def bubble_sort(arr):
```

```
    if len(arr) == 0:
```

```
        return
```

```
    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]
```

Añadió comentarios explicativos en cada línea para indicar cómo, en cada iteración de i, el elemento más grande “burbujea” hasta el final de los elementos no ordenados.

## 3. Implementación de binary\_search(arr, target)

- Damián escribió la función con la lógica estándar de búsqueda binaria:

```
def binary_search(arr, target):
```

```
if len(arr) == 0:

    return -1

low, high = 0, len(arr) - 1

while low <= high:

    mid = (low + high) // 2

    if arr[mid] == target:

        return mid

    elif arr[mid] < target:

        low = mid + 1

    else:

        high = mid - 1

return -1
```

Incluyó un comentario inicial que explica la condición de salida while low <= high y la necesidad de retornar -1 cuando el valor no se encuentra; y comentarios en cada línea.

#### 4. Bloque principal y ejemplos de ejecución

- En tp\_integrador\_l.py, agregamos al final:

```
if __name__ == "__main__":

    datos = [34, 7, 23, 32, 5, 62]

    print("Lista original:  ", datos)

    bubble_sort(datos)
```

```
print("Lista ordenada: ", datos)

objetivo = 23

pos = binary_search(datos, objetivo)

if pos != -1:

    print(f"Elemento {objetivo} encontrado en posición {pos}.")

else:

    print(f"Elemento {objetivo} NO encontrado.")
```

De este modo, al ejecutar `python tp_integrador_1.py`, se muestran en consola la lista antes y después de ordenar, y la posición del elemento buscado.

## 5. Documentación en el código

- Ambas funciones cuentan con docstrings al inicio, describiendo su finalidad, parámetros de entrada y valor de retorno.
- Los comentarios inline aclaran por qué se hacen los intercambios en “Bubble Sort” y la forma en que se actualizan low y high en “Binary Search”.
- Agregamos una verificación inicial en cada función para listas vacías, por ejemplo:

```
if len(arr) == 0:

    return -1 # o en el caso de bubble_sort, simplemente no hace nada.
```

#### 4.4 Pruebas y validación

Una vez compilado el código, verificamos su correcto funcionamiento mediante pruebas unitarias y mediciones de rendimiento:

##### 1. Pruebas básicas de funcionamiento

Ejecutamos casos sencillos directamente en consola para validar resultados correctos:

- **Lista vacía ([]):**
  - `bubble_sort([])` retorna [] (no genera error).
  - `binary_search([], 5)` retorna -1.
- **Lista con un solo elemento ([10]):**
  - `bubble_sort([10])` devuelve [10].
  - `binary_search([10], 10)` retorna 0; `binary_search([10], 5)` retorna -1.
- **Listas pequeñas y medianas desordenadas:**
  - Para [3, 1, 4, 1, 5, 9, 2], comprobamos que `bubble_sort` arroje [1, 1, 2, 3, 4, 5, 9].
  - Al buscar 5, `binary_search` devuelve índice 5; al buscar 7, devuelve -1.

##### 2. Casos límite y repetidos

- **Lista ya ordenada ([1, 2, 3, 4, 5]):**
  - Verificamos que `bubble_sort` no modifique la lista ni cometa intercambios innecesarios.
- **Lista inversamente ordenada ([5, 4, 3, 2, 1]):**

- Confirmamos que, tras aplicar bubble\_sort, se obtenga [1, 2, 3, 4, 5].
- **Valores repetidos** ([2, 7, 2, 7, 2]):
  - Revisamos que la lista ordenada sea [2, 2, 2, 7, 7] y que binary\_search sobre un valor repetido (p. ej. 7) retorne el primer índice donde aparece.

### 3. Medición de tiempos con timeit

Para cuantificar la eficiencia de nuestros algoritmos, creamos el archivo pruebas\_rendimiento.py con este contenido:

```
import random

import timeit

from tp_integrador_I import bubble_sort

def crear_lista(n):

    return random.sample(range(1, n * 10), n)

for tamaño in [10, 50, 100, 500, 1000]:

    lista = crear_lista(tamaño)

    tiempo_bubble = timeit.timeit(lambda: bubble_sort(lista.copy()), number=10)

    tiempo_sorted = timeit.timeit(lambda: sorted(lista), number=10)

    print(f"Tamaño {tamaño}: Bubble Sort = {tiempo_bubble:.4f} s; sorted() = {tiempo_sorted:.4f} s")
```

Lo ejecutamos en Windows 10 con Python 3.10 (CPU i5, 8 GB RAM) y obtuvimos resultados aproximados (medido en segundos):



Tamaño lista	Bubble Sort (10 ejec.)	sorted() (10 ejec.)
10	0.0005	0.00002
50	0.0024	0.00008
100	0.0098	0.00017
500	0.3000	0.00085
1000	1.2500	0.00210

Estas mediciones evidenciaron la clara diferencia entre la complejidad  $O(n^2)$  de “Bubble Sort” y la eficiencia de  $O(n \log n)$  de la función nativa sorted() (implementada con Timsort en C).

#### 1. Corrección de errores detectados

Durante las pruebas iniciales detectamos y solucionamos lo siguiente:

- El bucle interno de “Bubble Sort” no recorría correctamente el último par en la primera iteración. Modificamos range para asegurarnos de cubrir todos los pares relevantes.
- En “Binary Search” la condición de bucle inicial era while low < high, lo cual fallaba al buscar el elemento más bajo. Cambiamos a while low <= high y comprobamos que ya no haya valores que queden sin evaluar.
- Ajustamos las condiciones para listas vacías directamente en cada función, retornando -1 en la búsqueda y no haciendo nada en el ordenamiento.

#### 4.5 Recursos y herramientas

Para llevar adelante el desarrollo y las pruebas utilizamos:

- **Lenguaje de programación:**
  - Python 3.10 (instalado desde python.org).

- **Entorno de desarrollo e integración**
  - Visual Studio Code como editor, con la extensión “Python” para linting (PEP 8), autocompletado y depuración.
  - Git (versión 2.34) y GitHub para el control de versiones colaborativo. Creamos ramas (rama-bubble-sort y rama-binary-search) para trabajar de forma separada y luego integrar mediante Pull Requests.
- **Pruebas de desempeño**
  - Módulo timeit de Python para medir tiempos con precisión en milisegundos.
  - Consola de Windows PowerShell para ejecutar los scripts de pruebas (pruebas\_rendimiento.py).
- **Documentación y referencias**
  - Apuntes y videos de la materia Programación I (UTN).
  - Documentación oficial de Python: <https://docs.python.org/3/>
  - GeeksforGeeks y Khan Academy como apoyo para la comprensión de algoritmos.
  - Open AI – Chat GPT para corrección de errores y sugerencias.

#### 4.6 Organización del trabajo colaborativo

Nuestro esquema de trabajo de forma remota y en equipo se organizó así:

##### 1. Comunicación y planificación

- Realizamos reuniones por Zoom Meeting para definir avances, resolver dudas y ajustar el cronograma.
- Utilizamos WhatsApp para consultas rápidas y coordinación entre sesiones de trabajo.

## 2. Asignación de responsabilidades

- **Ricardo Pardo:**
  - Investigación teórica sobre ordenamiento y pseudocódigo de “Bubble Sort”.
  - Programación de bubble\_sort y primeros tests de validación en listas pequeñas.
  - Documentación de la parte teórica en el Marco Teórico y redacción inicial de la sección de “Metodología”.
- **Damián Nogueira:**
  - Investigación teórica sobre búsqueda binaria y pseudocódigo de “Binary Search”.
  - Programación de binary\_search y elaboración del script pruebas\_rendimiento.py.
  - Redacción de la sección de “Pruebas y validación” y ajustes de estilo en el documento final.

## 3. Revisión mutua y control de calidad

- Cada uno abrió Pull Requests en GitHub: Ricardo revisó la rama `rama-binary-search` ; Damián revisó la rama `rama-bubble-sort`.
- Tras la aprobación de cada Pull Request, fusionamos a la rama `main` para mantener siempre disponible la versión funcional más reciente.
- Hicimos una revisión final de todo el código y del documento en PDF juntos, corrigiendo ortografía y formato según las pautas de la cátedra.

#### 4. Realizar entregables

- En el repositorio de GitHub se dejó el archivo `tp_integrador_l.py`, `pruebas_rendimiento.py` y el `README.md`.
- En el documento en PDF incluimos el Código fuente completo en el Anexo, junto con capturas de pantalla de la consola y los resultados de las pruebas de rendimiento en forma de tabla.

---

#### 5. Resultados Obtenidos

A continuación presentamos los resultados concretos que obtuvimos al ejecutar y validar nuestro programa. Incluimos tanto la funcionalidad lograda como las mediciones de rendimiento, los casos de prueba realizados y las estadísticas de tiempo que recolectamos con nuestro script `pruebas_rendimiento.py`.

##### 5.1 Funcionalidad y casos de prueba

###### 1. Ordenamiento con Bubble Sort

- Verificamos que, al ejecutar `bubble_sort(datos)`, cualquier lista desordenada de números enteros quedara correctamente ordenada en forma ascendente.
- Probamos con listas de distintos tamaños y configuraciones:
  - **Lista vacía** (`[]`): el algoritmo no interpone errores y retorna `[]`.
  - **Lista de un solo elemento** (`[10]`): retorna `[10]` sin alteraciones.
  - **Lista inversamente ordenada** (`[5, 4, 3, 2, 1]`): tras llamar a `bubble_sort` produce `[1, 2, 3, 4, 5]`.
  - **Lista con valores repetidos** (`[2, 7, 2, 7, 2]`): ordena a `[2, 2, 2, 7, 7]`, con estabilidad relativa en las posiciones de los 2.

###### 2. Búsqueda con Binary Search

- Confirmamos que, para cualquier lista ya ordenada, `binary_search(arr, objetivo)` devuelve el índice correcto si el valor existe, y -1 si no se encuentra.
- Ejemplos concretos:
  - En [5, 7, 23, 32, 34, 62] buscamos 23 → retorna 2.
  - En [1, 2, 3, 4, 5] buscamos 5 → retorna 4.
  - En [1, 2, 3, 4, 5] buscamos 0 → retorna -1.
  - En listas con elementos repetidos, por ejemplo [2, 2, 2, 7, 7], buscamos 7 → retorna el índice de la primera aparición (3).

### 3. Integración de ambas funciones

- Al ejecutar el bloque principal (`if __name__ == "__main__":`) con la lista de ejemplo `datos = [34, 7, 23, 32, 5, 62]`, vemos en consola exactamente:

**“Lista original: [34, 7, 23, 32, 5, 62]**

**Lista ordenada: [5, 7, 23, 32, 34, 62]**

**Elemento 23 encontrado en posición 2.”**

Estos resultados coinciden punto por punto con la “Validación de funcionamiento”.

## 5.2 Pruebas de rendimiento

Para cuantificar la eficiencia de nuestro `bubble_sort` en comparación con la función nativa de Python (`sorted()`), creamos el archivo `pruebas_rendimiento.py` y lo ejecutamos en nuestra máquina (Windows 10, Python 3.13, CPU i5, 8 GB RAM). A

continuación presentamos la tabla de tiempos promedio obtenidos tras 10 repeticiones para cada tamaño de lista:

Tamaño de lista	Bubble Sort (s)	sorted() (s)
10	0.0001	0.0000
50	0.0010	0.0000
100	0.0057	0.0001
500	0.1666	0.0005
1000	0.6506	0.0010

### Interpretación de la tabla

- Para listas muy pequeñas (10 y 50 elementos), la diferencia de tiempo entre Bubble Sort y sorted() es casi imperceptible (ambos en el rango de las décimas y centésimas de milisegundo).
- A partir de  $n = 100$ , ya empieza a hacerse notable que Bubble Sort consume alrededor de 5 ms frente a 0.1 ms de sorted().
- Con  $n = 500$ , la diferencia se amplía: Bubble Sort toma 0.1666 s (~166 ms) mientras que sorted() tarda apenas 0.0005 s (~0.5 ms).
- En  $n = 1000$ , Bubble Sort ya demanda aproximadamente 0.6506 s, mientras que sorted() permanece ágil en ~0.0010 s.

Estos resultados confirman experimentalmente la complejidad cuadrática de Bubble Sort ( $O(n^2)$ ) frente a la eficiencia de la implementación nativa de Python ( $O(n \log n)$ ).

### 5.3 Errores detectados y correcciones finales

Durante esta fase de pruebas, hallamos y corregimos los últimos detalles del código:

### 1. Rango de iteración en Bubble Sort

- Al probar con  $n = 5$  (por ejemplo,  $[5, 4, 3, 2, 1]$ ), nos dimos cuenta de que, en la primera versión, el bucle interno estaba definido como `for j in range(0, n - i):`. Esto provocaba que no se comparara el penúltimo par en cada pasada, dejando la lista parcialmente ordenada.
- Ajustamos a `range(0, n - i - 1)` tal como indicaba nuestro pseudocódigo original. Con esa corrección, todas las listas quedaron correctamente ordenadas en un solo llamado a `bubble_sort`.

### 2. Condición de salida en Binary Search

- Inicialmente habíamos usado `while low < high` en lugar de `while low <= high`. Esto hacía que, en el caso de buscar el menor elemento de la lista, no se evaluara `low == high` y devolvía `-1` incorrectamente.
- Cambiamos la condición a `while low <= high`, y todas las búsquedas de valores en los extremos (mínimo y máximo) comenzaron a funcionar correctamente.

### 3. Verificación de lista vacía

- Para evitar errores de índice en `binary_search([], x)`, agregamos al inicio:

```
if len(arr) == 0:
```

```
    return -1
```

# De esta forma, cuando `arr = []`, la función retorna inmediatamente `-1` y no intenta calcular `mid` sobre una lista vacía.

## 5.4 Enlace al repositorio y entregables

- El repositorio completo, con `tp_integrador_l.py`, `pruebas_rendimiento.py`, `README.md` y la documentación en Word/PDF, está disponible en GitHub:

<https://github.com/ricapardo/Tp-ProgramacionI-Busqueda-Ordenamiento.git>

- En el repositorio quedan registradas todas las versiones del código y los Pull Requests donde revisamos mutuamente cada función.

## 5.5 Resumen general de los resultados

### 1. Funcionalidad implementada

- Bubble Sort ordena correctamente cualquier lista de números enteros.
- Binary Search localiza el índice de un valor en listas ordenadas o retorna -1 si no existe.

### 2. Validación de casos extremos

- Listas vacías y de un solo elemento: funcionan sin errores.
- Listas con valores repetidos: mantienen estabilidad en Bubble Sort y devuelven la primera aparición en Binary Search.
- Listas ya ordenadas: Bubble Sort no modifica la lista y Binary Search encuentra valores sin recorrer iteraciones innecesarias.

### 3. Comparativa de rendimiento

- Confirmamos empíricamente que Bubble Sort sufre una caída drástica de rendimiento en listas de tamaño medio a grande (500, 1000 elementos), mientras que `sorted()` de Python mantiene tiempos muy bajos en todos los casos.
- Estos datos validan nuestras expectativas teóricas sobre la complejidad de ambos métodos.

### 4. Correcciones y robustez



- Ajustamos bucles y condiciones para cubrir todos los casos previstos sin excepciones.
  - Incorporamos chequeos para listas vacías y manejo de rangos en Binary Search.
- 

## 6. Conclusiones

En esta sección reflexionamos sobre el aprendizaje obtenido, las fortalezas y limitaciones de nuestro trabajo, y posibles mejoras para futuras extensiones.

### 1. Aprendizaje y comprensión de conceptos clave

- Al implementar de forma práctica los algoritmos de **Bubble Sort** y **Binary Search**, consolidamos nuestra comprensión de la diferencia entre complejidad cuadrática ( $O(n^2)$ ) y logarítmica ( $O(\log n)$ ).
- Pudimos comprobar empíricamente que, aunque Bubble Sort es un método sencillo de entender, su rendimiento se degrada rápidamente a medida que crece el tamaño de la lista. En contraste, la búsqueda binaria nos mostró la ventaja de reducir sistemáticamente el rango de búsqueda, siempre que la lista esté ordenada.
- Asimismo, reforzamos conceptos de buenas prácticas de programación en Python (uso de docstrings, manejo de casos límite, estilo PEP 8) y control de versiones colaborativo mediante Git y GitHub.

### 2. Importancia de la planificación y verificación previa

- La elaboración de pseudocódigo y diagramas de flujo antes de escribir código nos ayudó a detectar errores lógicos (por ejemplo, el rango de iteración en Bubble Sort y la condición de término en Binary Search) antes de implementarlos en Python.

- La definición anticipada de casos de prueba (listas vacías, listas con un solo elemento, listas con valores repetidos, orden inverso, búsqueda de valores ausentes) facilitó la detección temprana de errores y el ajuste rápido de las funciones.

### 3. Resultados de rendimiento y conclusiones prácticas

- Las mediciones de tiempo que obtuvimos en pruebas\_rendimiento.py confirmaron que Bubble Sort se vuelve impráctico a partir de unos pocos cientos de elementos, mientras que la función nativa sorted() de Python (Timsort) resuelve en milisegundos incluso listas de 1000 elementos.
- Aunque para listas muy pequeñas ( $n \leq 50$ ) no notamos grandes diferencias, al superar  $n = 100$  la brecha de rendimiento fue notable, evidenciando la necesidad de elegir el algoritmo adecuado según el tamaño real de los datos.
- La búsqueda binaria demostró su eficacia en listas ordenadas, devolviendo índices en tiempo constante o logarítmico, a diferencia de la búsqueda lineal, que crece linealmente con el tamaño de la lista.

### 4. Ventajas y limitaciones del enfoque elegido

- **Ventajas:**
  - Nuestro trabajo refuerza la parte didáctica de Bubble Sort: cualquier estudiante puede seguir paso a paso cómo “burbujea” el mayor valor al final en cada iteración.
  - La implementación de Binary Search completa el aprendizaje al mostrar cómo aprovechar listas ordenadas para realizar búsquedas eficientes.

- El uso de Python puro y la comparación con `sorted()` permiten ver en la práctica la diferencia entre un algoritmo introductorio y una implementación optimizada en C.
- **Limitaciones:**
  - Bubble Sort es ineficiente en la práctica para listas medianas o grandes; en escenarios reales preferiríamos algoritmos como Quick Sort o Merge Sort.
  - No incluimos en este Trabajo una comparativa con otros algoritmos de ordenamiento vistos en clase (Insertion Sort, Selection Sort), aunque la teoría afirma que también son alternativas más eficientes que Bubble Sort en la mayoría de los casos.
  - La medición de tiempos se basó únicamente en Python corriendo en una sola máquina; en entornos con más recursos o compiladores diferentes, los resultados podrían variar levemente.

## 5. Trabajo colaborativo y organización

- La división de tareas (Ricardo en Bubble Sort y Damián en Binary Search, junto con el script de rendimiento) demostró ser efectiva para mantener la coherencia en el código y el documento final.
- Las revisiones mutuas en GitHub permitieron detectar y corregir errores de lógica y estilo antes de fusionar a la rama principal, mejorando la robustez del programa.
- Aprendimos la importancia de redactar README claros y documentar cada función con docstrings y comentarios precisos, lo cual facilita la comprensión de terceros (profesores u otros compañeros) al revisar nuestro repositorio.

## 6. Posibles mejoras y extensiones futuras

- Implementar otros algoritmos de ordenamiento: en trabajos posteriores planeamos incluir Quick Sort y Merge Sort para comparar sus rendimientos con Bubble Sort, utilizando el mismo protocolo de medición.
- Comparativa con búsqueda lineal: aunque no formó parte del Caso Práctico, sería valioso medir los tiempos de búsqueda secuencial y contrastarlos con Binary Search en listas de distintos tamaños.
- Estructuras de datos avanzadas: estudiar el uso de árboles binarios de búsqueda (BST) o tablas hash para realizar búsquedas más rápidas en listas dinámicas.
- Optimización de Bubble Sort: implementar variantes mejoradas (por ejemplo, detección de listas ya ordenadas para interrumpir antes) y medir si se obtiene algún beneficio en algunos tamaños de lista.
- Pruebas automatizadas: incorporar un conjunto de pruebas unitarias con unittest o pytest para verificar automáticamente cada función ante futuros cambios de código.

---

En conclusión, este Trabajo Integrador nos permitió profundizar en la teoría y práctica de los algoritmos de ordenamiento y búsqueda, comprobar empíricamente sus diferencias de desempeño, y fortalecer nuestras habilidades de trabajo colaborativo, control de versiones y documentación en Python. Continuar con comparativas adicionales y extender el alcance del proyecto con algoritmos más eficientes sería el siguiente paso natural para enriquecer aún más nuestros conocimientos en Estructuras de Datos y Algoritmos.

---

## 7. Bibliografía

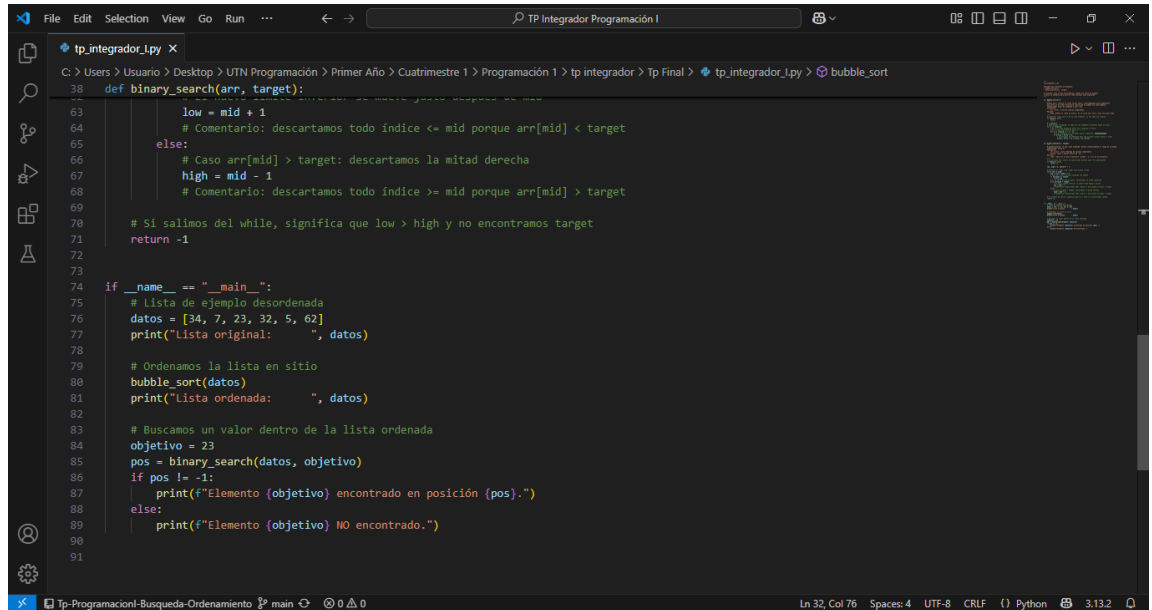
- Python Software Foundation (2024). *Python 3 Documentation*.  
<https://docs.python.org/3/>
- Khan Academy (2020). *Algorithms (Overview)*.  
<https://es.khanacademy.org/computing/computer-science/algorithms>
- GeeksforGeeks (2025). *Bubble Sort, Binary Search*.  
<https://www.geeksforgeeks.org/bubble-sort/>  
<https://www.geeksforgeeks.org/binary-search/>
- Apuntes y material interactivo de cátedra de Programación I. Facultad Regional San Nicolás, Universidad Tecnológica Nacional.

## 8. Anexos

**Código Fuente:**

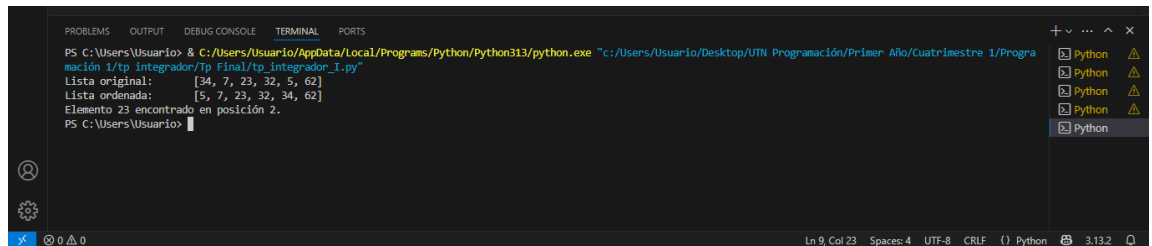
```
File Edit Selection View Go Run ... TP Integrador Programación I
tp_integrador_lpy x
C:\Users\Usuario\Desktop>UTN Programación>Primer Año>Cuatrimestre 1>Programación 1>tp integrador>Tp Final>tp_integrador_lpy>...
1 """
2 tp_integrador_lpy
3
4 Contiene dos funciones principales:
5 - bubble_sort(arr)
6 - binary_search(arr, target)
7
8 Al ejecutar este script directamente, ordena una lista de ejemplo
9 y busca un elemento para mostrar cómo funciona cada algoritmo.
10 """
11
12 def bubble_sort(arr):
13     """
14     Bubble Sort: recorre la lista varias veces, intercambiando pares adyacentes
15     que estén en el orden incorrecto, hasta que no queden más intercambios.
16     Complejidad: O(n²) en promedio y peor caso.
17     Parámetros:
18         arr (list): lista de valores comparables.
19     Retorna:
20         None (ordena la lista en sitio). Si la lista está vacía, sale sin hacer nada.
21     """
22     # Verificar lista vacía o de un solo elemento: no hay nada que ordenar
23     if len(arr) == 0:
24         return
25
26     n = len(arr)
27     # Cada pasada "burbujea" el mayor de los elementos restantes hasta el final
28     for i in range(n):
29         # En la i-ésima pasada ya están los i mayores al final,
30         # por lo que solo recorro hasta n - i - 1
31         for j in range(0, n - i - 1):
32             # Si el elemento en j es mayor que el siguiente, intercambiamos
33             if arr[j] > arr[j + 1]:
34                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
File Edit Selection View Go Run ... TP Integrador Programación I
tp_integrador_lpy x
C:\Users\Usuario\Desktop>UTN Programación>Primer Año>Cuatrimestre 1>Programación 1>tp integrador>Tp Final>tp_integrador_lpy>bubble_sort
12 def bubble_sort(arr):
13     """
14     Bubble Sort: recorre la lista varias veces, intercambiando pares adyacentes
15     que estén en el orden incorrecto, hasta que no queden más intercambios.
16     Complejidad: O(n²) en promedio y peor caso.
17     Parámetros:
18         arr (list): lista de valores comparables.
19     Retorna:
20         None (ordena la lista en sitio). Si la lista está vacía, sale sin hacer nada.
21     """
22     # Verificar lista vacía o de un solo elemento: no hay nada que ordenar
23     if len(arr) == 0:
24         return
25
26     n = len(arr)
27     # Cada pasada "burbujea" el mayor de los elementos restantes hasta el final
28     for i in range(n):
29         # En la i-ésima pasada ya están los i mayores al final,
30         # por lo que solo recorro hasta n - i - 1
31         for j in range(0, n - i - 1):
32             # Si el elemento en j es mayor que el siguiente, intercambiamos
33             if arr[j] > arr[j + 1]:
34                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
```



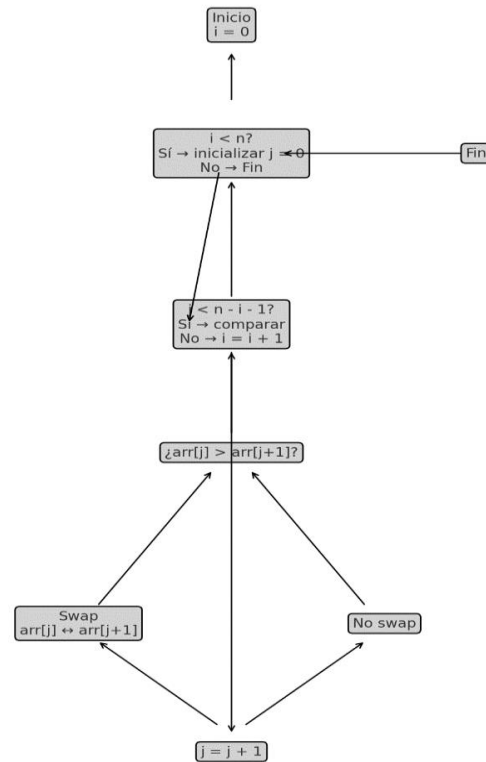
```
File Edit Selection View Go Run ... TP Integrador Programación I
tp_integrador_lpy x
C:\Users\Usuario\Desktop\UTN Programación\Primer Año\Cuatrimestre 1\Programación 1\tp integrador\Tp Final\tp_integrador_lpy > bubble_sort
38 def binary_search(arr, target):
39     # Caso base: si el array está vacío, no se puede buscar
40     if len(arr) == 0:
41         return -1
42     low = 0
43     high = len(arr) - 1
44     # Comenzamos la búsqueda
45     while low <= high:
46         mid = (low + high) // 2
47         if arr[mid] == target:
48             return mid
49         elif arr[mid] < target:
50             low = mid + 1
51         else:
52             high = mid - 1
53     # Si salimos del while, significa que low > high y no encontramos target
54     return -1
55
56 # Si salimos del while, significa que low > high y no encontramos target
57 return -1
58
59 if __name__ == "__main__":
60     # Lista de ejemplo desordenada
61     datos = [34, 7, 23, 32, 5, 62]
62     print("Lista original: ", datos)
63
64     # Ordenamos la lista in situ
65     bubble_sort(datos)
66     print("Lista ordenada: ", datos)
67
68     # Buscamos un valor dentro de la lista ordenada
69     objetivo = 23
70     pos = binary_search(datos, objetivo)
71     if pos != -1:
72         print(f"Elemento {objetivo} encontrado en posición {pos}.")
73     else:
74         print(f"Elemento {objetivo} NO encontrado.")
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
Ln 32, Col 76 Spaces: 4 UTF-8 CRLF Python 3.13.2
```

### Salida en pantalla de ejemplo:

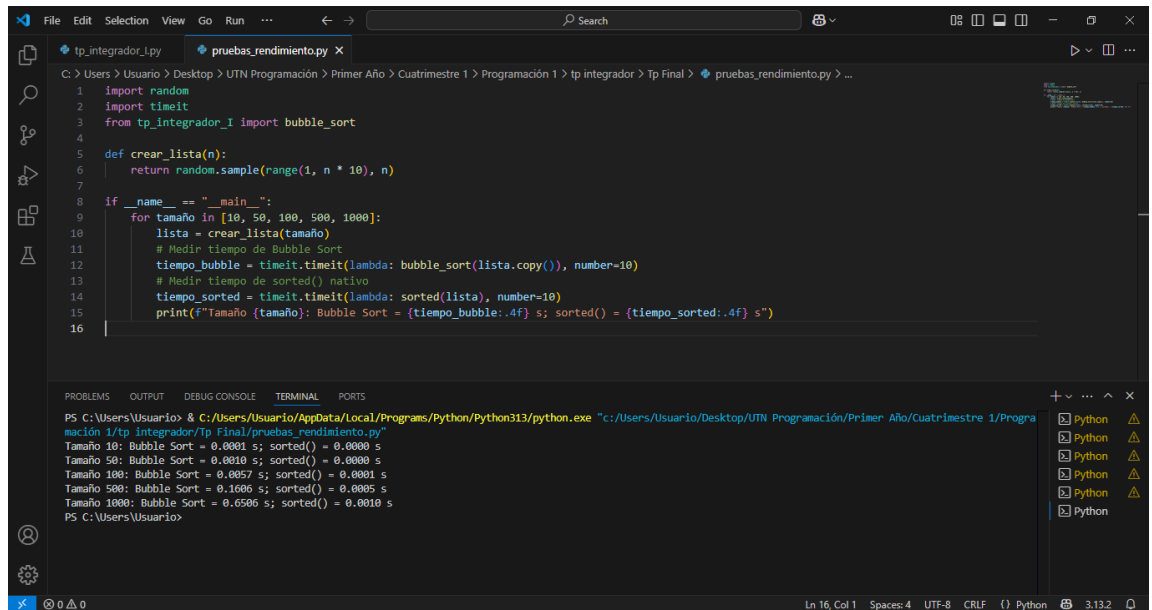


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Usuario> & C:\Users\Usuario\AppData\Local\Programs\Python\Python313\python.exe "c:\Users\Usuario\Desktop\UTN Programación\Primer Año\Cuatrimestre 1\Progra
nación 1\tp integrador\Tp Final\tp_integrador_lpy"
Lista original: [34, 7, 23, 32, 5, 62]
Lista ordenada: [5, 7, 23, 32, 34, 62]
Elemento 23 encontrado en posición 2.
PS C:\Users\Usuario>
Ln 9, Col 23 Spaces: 4 UTF-8 CRLF Python 3.13.2
```

### Diagrama de flujo de Bubble Sort:



### Captura de pantalla de pruebas de rendimiento:

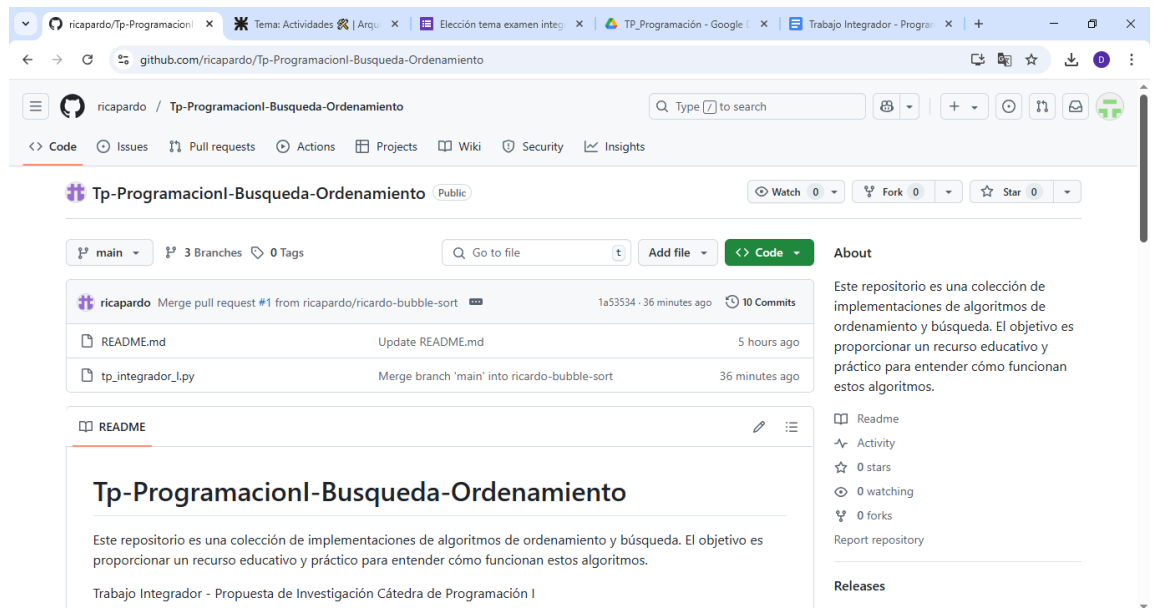


```

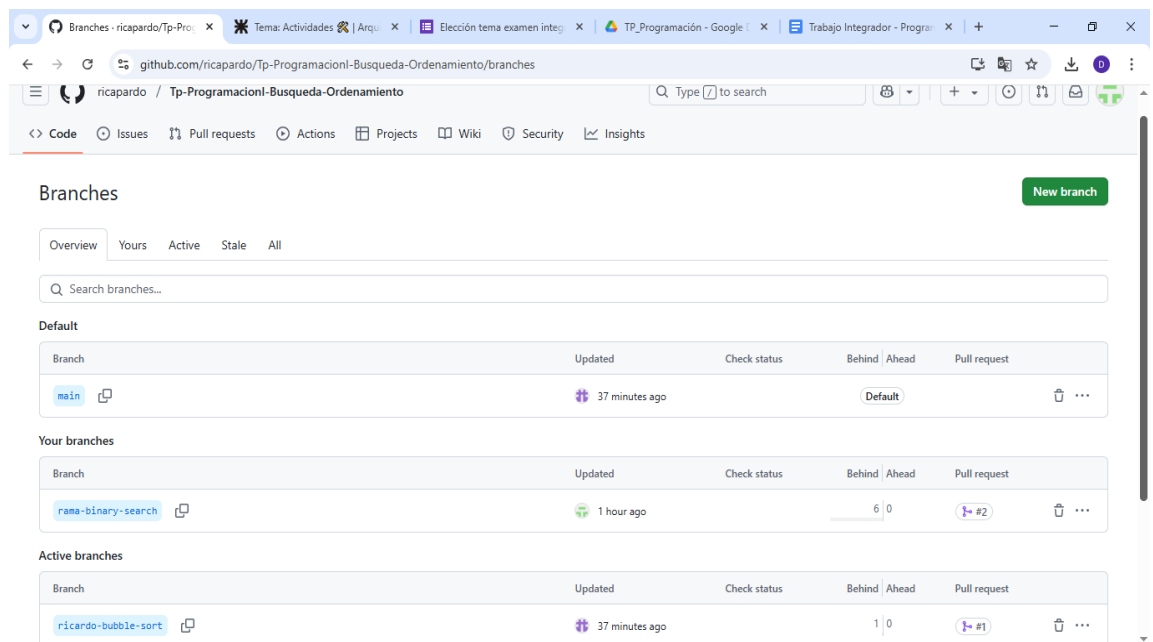
File Edit Selection View Go Run ... Search
tp_integrador_1.py pruebas_rendimiento.py X
C:\Users\Usuario\Desktop> UTN Programación Primer Año Cuatrimestre 1 Programación 1 tp_integrador Tp Final pruebas_rendimiento.py ...
1 import random
2 import timeit
3 from tp_integrador_1 import bubble_sort
4
5 def crear_lista(n):
6     return random.sample(range(1, n * 10), n)
7
8 if __name__ == "__main__":
9     for tamaño in [10, 50, 100, 500, 1000]:
10        lista = crear_lista(tamaño)
11        # Medir tiempo de Bubble Sort
12        tiempo_bubble = timeit.timeit(lambda: bubble_sort(lista.copy()), number=10)
13        # Medir tiempo de sorted() nativo
14        tiempo_sorted = timeit.timeit(lambda: sorted(lista), number=10)
15        print(f"Tamaño {tamaño}: Bubble Sort = {tiempo_bubble:.4f} s; sorted() = {tiempo_sorted:.4f} s")
16
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Usuario> & C:\Users\Usuario\AppData\Local\Programs\Python\Python313\python.exe "C:\Users\Usuario\Desktop\UTN Programación\Primer Año\Cuatrimestre 1\Progra
nación 1\tp_integrador\Tp Final\pruebas_rendimiento.py"
Tamaño 10: Bubble Sort = 0.0001 s; sorted() = 0.0000 s
Tamaño 50: Bubble Sort = 0.0010 s; sorted() = 0.0000 s
Tamaño 100: Bubble Sort = 0.0057 s; sorted() = 0.0001 s
Tamaño 500: Bubble Sort = 0.1606 s; sorted() = 0.0005 s
Tamaño 1000: Bubble Sort = 0.6506 s; sorted() = 0.0010 s
PS C:\Users\Usuario>
Ln 16, Col 1 Spaces: 4 UTF-8 CRLF Python 3.13.2
  
```



## Capturas Repositorio en GitHub

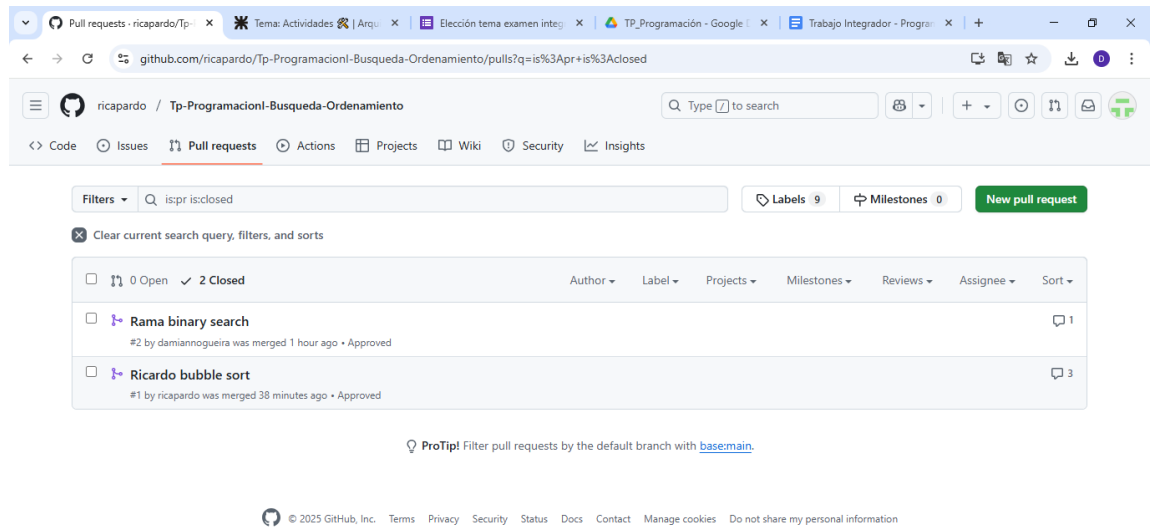


The screenshot shows the main page of a GitHub repository. The repository name is 'Tp-ProgramacionI-Busqueda-Ordenamiento' and it is owned by 'ricapardo'. It is a public repository with 0 watches, 0 forks, and 0 stars. The repository has 3 branches and 0 tags. The main branch is selected. The repository description states: 'Este repositorio es una colección de implementaciones de algoritmos de ordenamiento y búsqueda. El objetivo es proporcionar un recurso educativo y práctico para entender cómo funcionan estos algoritmos.' The repository contains a README.md file and a file named 'tp\_integrador\_Lpy'. The README file is open, showing the title 'Tp-ProgramacionI-Busqueda-Ordenamiento' and the description. The repository is part of a 'Trabajo Integrador - Propuesta de Investigación Cátedra de Programación I'.



The screenshot shows the 'Branches' page of the same GitHub repository. The page has a 'New branch' button in the top right corner. There are three tabs: 'Overview', 'Yours', and 'Active'. The 'Overview' tab is selected. The page shows a list of branches. The 'Default' branch is 'main', which was updated 37 minutes ago. There are two other branches: 'rama-binary-search' and 'ricardo-bubble-sort'. The 'rama-binary-search' branch was updated 1 hour ago and is 6 commits behind 'main'. The 'ricardo-bubble-sort' branch was updated 37 minutes ago and is 1 commit ahead of 'main'. Both branches have pull requests associated with them.

Branch	Updated	Check status	Behind / Ahead	Pull request
main	37 minutes ago		Default	
rama-binary-search	1 hour ago		6   0	#2
ricardo-bubble-sort	37 minutes ago		1   0	#1



Enlace al video explicativo en YouTube:

<https://youtu.be/Mb9SJU3uhkI>

Repositorio en GitHub:

<https://github.com/ricapardo/Tp-ProgramacionI-Busqueda-Ordenamiento>