

---

**TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A  
DISTANCIA**

**TRABAJO INTEGRADOR FINAL – ALGORITMOS DE  
BUSQUEDA Y ORDENAMIENTO EN PYTHON**

---

Alumno: Dichiara Hernan.

Cuenta Institucional: [hernan.dichiara@tupad.utn.edu.ar](mailto:hernan.dichiara@tupad.utn.edu.ar)

Materia: Programación 1.

Prof.: Trapé Julieta y Barrera Oltra Miguel.

Fecha de entrega: 20/06/2025.

---

**Indice:**

---

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

## **1.Introducción**

---

Los algoritmos de búsqueda y ordenamiento constituyen una base esencial en la programación, ya que permiten organizar y recuperar información de manera eficiente. Su correcta implementación optimiza el rendimiento de los programas, especialmente al trabajar con grandes volúmenes de datos. Este tipo de algoritmos se utiliza ampliamente en bases de datos, motores de búsqueda, análisis de datos y sistemas informáticos en general.

Este trabajo fue desarrollado de forma individual, ante la imposibilidad de integrar un grupo dentro de los plazos establecidos. A pesar de ello, se buscó abordar el tema de forma integral, combinando teoría, práctica y reflexión personal.

Se eligió este tema por su relevancia técnica y formativa, y porque permite explorar conceptos clave como la eficiencia algorítmica y la importancia de la estructura de datos. El objetivo principal del trabajo es comprender, implementar y comparar algoritmos básicos de búsqueda y ordenamiento utilizando Python, evaluando su funcionamiento y aplicabilidad en distintos contextos.

A lo largo del desarrollo, se expondrán definiciones clave, se implementará un caso práctico con código funcional, se analizarán los resultados obtenidos y se presentarán conclusiones basadas en la experiencia.

## 2.Marco Teórico

En programación, los algoritmos de búsqueda y ordenamiento son herramientas fundamentales que permiten manipular conjuntos de datos de manera eficiente. Comprender su funcionamiento es clave para desarrollar soluciones efectivas en diversas áreas de la informática.

### **2.1 Algoritmos de búsqueda:**

La búsqueda consiste en localizar un elemento dentro de una estructura de datos, como una lista o un arreglo. Los algoritmos más comunes son:

- **Búsqueda lineal:** Recorre secuencialmente todos los elementos hasta encontrar el valor buscado. Es simple y no requiere una lista ordenada. Su eficiencia es baja en listas grandes.
- **Búsqueda binaria:** En listas grandes tiene mayor velocidad, pero es limitada: solo funciona en listas previamente arregladas. Divide el conjunto en mitades sucesivas hasta encontrar el elemento o descartar su existencia.

### **Ejemplo de búsqueda binaria:**

Lista: [3, 7, 15, 21, 28, 34, 42]

Elemento buscado: 21.

Pasos:

- 1) ¿Es 21 igual a 34? No, es menor. Por lo tanto, se buscará en la mitad izquierda.
- 2) ¿Es 21 igual a 7? No, es mayor. Por lo tanto, se buscará en la mitad derecha.
- 3) ¿Es 21 igual a 21? Si. Elemento encontrado.

## **2.2 Algoritmos de Ordenamiento:**

El ordenamiento reorganiza los datos según un criterio (por ejemplo, de menor a mayor o alfabéticamente). Su aplicación permite optimizar búsquedas, mejorar la presentación de resultados y facilitar otras operaciones, como la eliminación de duplicados o fusiones de listas. Los algoritmos más comunes son:

- Bubble Sort (Ordenamiento por burbuja): Compara pares de elementos adyacentes e intercambia sus posiciones si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista queda completamente ordenada. Es fácil de entender e implementar, pero es muy ineficiente para listas grandes.
- Selection Sort (Ordenamiento por selección): Según el criterio aplicado, encuentra el elemento y lo coloca en la primera posición. Luego repite el proceso con el resto de la lista. Aunque requiere de pocas operaciones de intercambio, es lento en listas grandes. Ejemplo: 1) Busca el mínimo de toda la lista y lo coloca al inicio. 2) Busca el siguiente mínimo entre los elementos restantes y lo coloca en la segunda posición. 3) Repite hasta ordenar toda la lista.
- Insertion Sort (Ordenamiento por inserción): Construye la lista ordenada tomando un elemento a la vez y colocándolo en la posición correcta dentro de los ya ordenados. Eficiente para listas pequeñas y casi ordenadas, pero no para listas grandes. Ejemplo: 1) Comienza con el segundo elemento de la lista y lo compara con el primero. 2) Lo mueve hacia la izquierda, hasta encontrar su lugar. 3) Repite el proceso con cada elemento nuevo.

- Quick Sort (Ordenamiento rápido): Algoritmo de tipo “divide y vencerás”. Selecciona un elemento “pivote” y divide la lista en dos sublistas: una con elementos menores y otra con elementos mayores. Luego, aplica el mismo proceso recursivamente en ambas sublistas. Es muy eficiente para grandes volúmenes de datos, pero si el pivote no se elige bien su rendimiento puede deteriorarse. Ejemplo: 1) Se elige un pivote (por ejemplo, el último elemento). 2) Se reordena la lista de manera que los elementos menores al pivote queden a la izquierda y los mayores a la derecha. 3) Se aplica Quick Sort recursivamente a ambas mitades.

### **2.3 Aplicación en Python:**

Python permite implementar estos algoritmos de manera clara y didáctica. A través de estructuras como listas y condicionales (if, while, for) es posible construir versiones propias de los algoritmos o utilizar métodos nativos (como `sort()` e `index()`).

En este trabajo se implementarán manualmente el algoritmo de ordenamiento por burbuja y la búsqueda binaria, con el fin de comprender su lógica interna.

---

## **3.Caso Práctico**

---

### **3.1 Descripción del problema:**

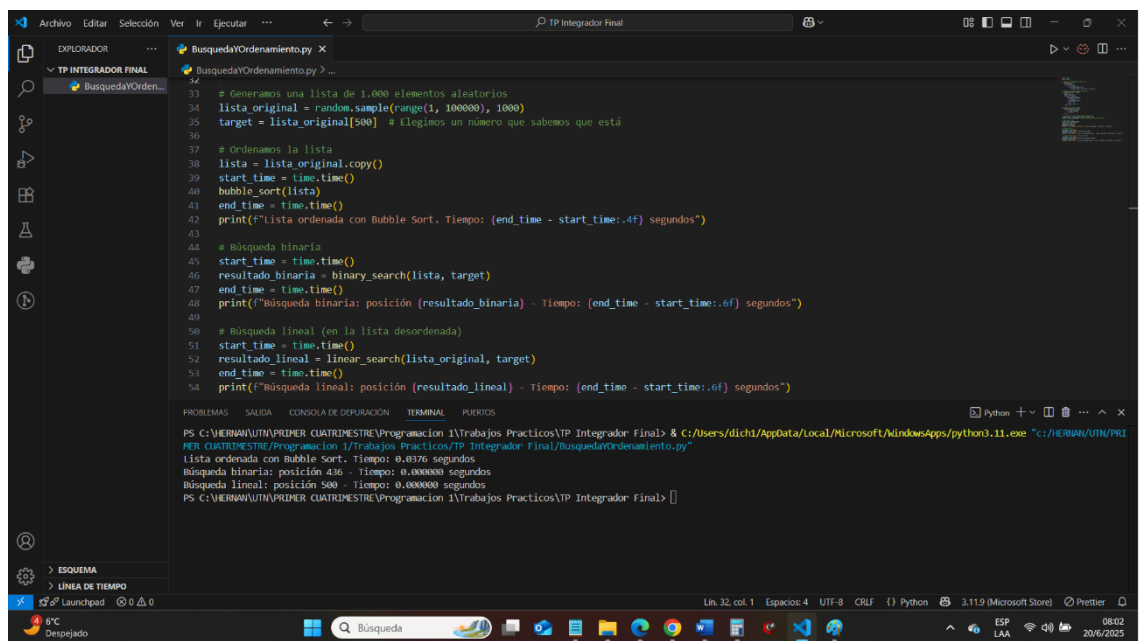
Para este punto, simulamos querer ordenar una lista de números enteros aleatorios y luego buscar un valor específico en ella. Para ello utilizaremos Bubble Sort para el ordenamiento y Búsqueda Binaria para una búsqueda eficiente. A su vez se comparará dicha eficiencia con Búsqueda Lineal (midiendo el tiempo de ejecución con la biblioteca `time`) para evaluar el rendimiento del método elegido frente a otros métodos de búsqueda.

### **3.2 Código fuente y comparación entre búsqueda lineal y binaria:**

## PROGRAMACION 1 – TUPAD – TRABAJO INTEGRADOR FINAL

Como parte del desarrollo del caso práctico, decidimos verificar el funcionamiento del código utilizando listas de distinto tamaño, con el objetivo de analizar en profundidad el comportamiento de los algoritmos de búsqueda y validar su eficiencia. A continuación, presentamos una explicación detallada.

- Verificación inicial con 1.000 elementos: primero probamos el funcionamiento del algoritmo completo utilizando una lista de 1.000 elementos aleatorios. Empleamos Bubble Sort para el ordenamiento y luego se realizaron búsquedas binaria y lineal. Los resultados confirmaron que la búsqueda binaria era ligeramente más rápida que la lineal, aunque la diferencia no era tan significativa debido al tamaño reducido de la lista.



```
33 # Generamos una lista de 1,000 elementos aleatorios
34 lista_original = random.sample(range(1, 100000), 1000)
35 target = lista_original[500] # Elegimos un número que sabemos que está
36
37 # Ordenamos la lista
38 lista = lista_original.copy()
39 start_time = time.time()
40 bubble_sort(lista)
41 end_time = time.time()
42 print(f"Lista ordenada con Bubble Sort. Tiempo: {end_time - start_time:.4f} segundos")
43
44 # Búsqueda binaria
45 start_time = time.time()
46 resultado_binaria = binary_search(lista, target)
47 end_time = time.time()
48 print(f"Búsqueda binaria: posición {resultado_binaria} - Tiempo: {end_time - start_time:.6f} segundos")
49
50 # Búsqueda lineal (en la lista desordenada)
51 start_time = time.time()
52 resultado_lineal = linear_search(lista_original, target)
53 end_time = time.time()
54 print(f"Búsqueda lineal: posición {resultado_lineal} - Tiempo: {end_time - start_time:.6f} segundos")
```

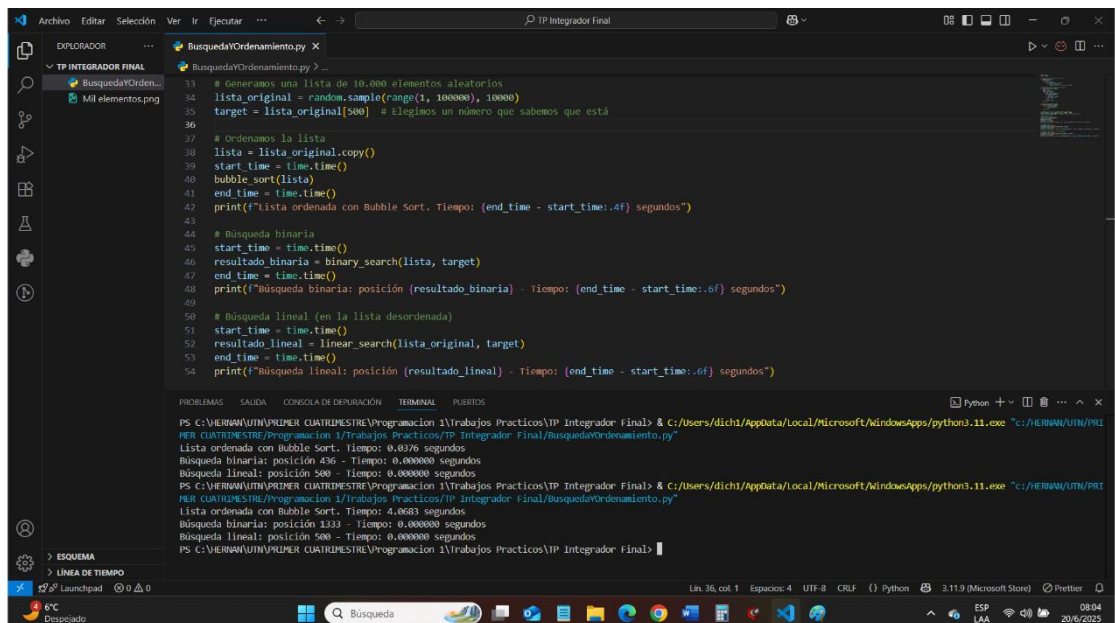
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS C:\VIERMAN\UTN\PRIMER CUATRIMESTRE\Programación 1\Trabajos Practicos\TP Integrador Final> & C:\Users\dich1\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/VIERMAN/UTN/PRI
MER CUATRIMESTRE/Programacion 1/Trabajos Practicos/TP Integrador Final/BusquedaYOrdenamiento.py"
Lista ordenada con Bubble Sort. Tiempo: 0.0376 segundos
Búsqueda binaria: posición 436 - Tiempo: 0.000000 segundos
Búsqueda lineal: posición 500 - Tiempo: 0.000000 segundos
PS C:\VIERMAN\UTN\PRIMER CUATRIMESTRE\Programación 1\Trabajos Practicos\TP Integrador Final>
```

- Verificación con 10.000 elementos: luego realizamos una prueba con una lista de 10.000 elementos aleatorios. En este caso, utilizamos dos métodos: el primero con Bubble Sort y, para garantizar que la búsqueda binaria funcionara correctamente, en el segundo método la lista fue ordenada previamente utilizando el método "sort()" de Python en lugar de Bubble Sort, debido a las limitaciones de rendimiento de este último en volúmenes mayores. Los resultados mostraron que la búsqueda binaria fue considerablemente más rápida que la búsqueda lineal,

## PROGRAMACION 1 – TUPAD – TRABAJO INTEGRADOR FINAL

manteniendo un tiempo casi constante y evidenciando con claridad la ventaja en listas de tamaño mediano. Por otro lado, la búsqueda lineal tomó más tiempo al recorrer la lista de forma secuencial.

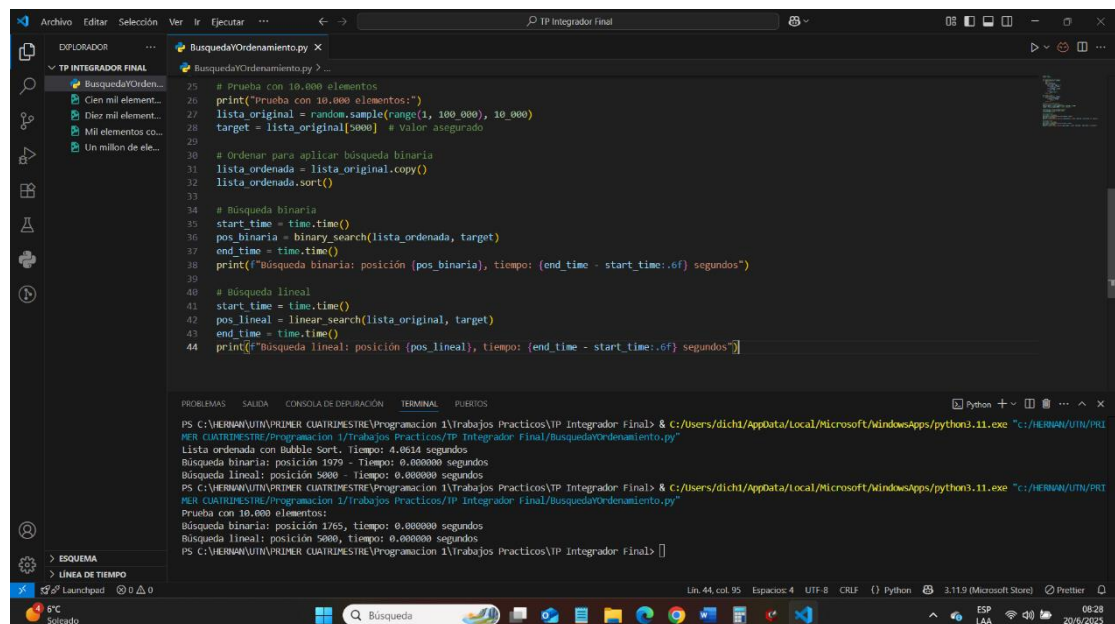


```
33 # Generamos una lista de 10.000 elementos aleatorios
34 lista_original = random.sample(range(1, 100000), 10000)
35 target = lista_original[500] # Elegimos un número que sabemos que está
36
37 # Ordenamos la lista
38 lista = lista_original.copy()
39 start_time = time.time()
40 bubble_sort(lista)
41 end_time = time.time()
42 print(f"Lista ordenada con Bubble Sort. Tiempo: {end_time - start_time:.6f} segundos")
43
44 # Búsqueda binaria
45 start_time = time.time()
46 resultado_binaria = binary_search(lista, target)
47 end_time = time.time()
48 print(f"Búsqueda binaria: posición {resultado_binaria} - Tiempo: {end_time - start_time:.6f} segundos")
49
50 # Búsqueda lineal (en la lista desordenada)
51 start_time = time.time()
52 resultado_lineal = linear_search(lista_original, target)
53 end_time = time.time()
54 print(f"Búsqueda lineal: posición {resultado_lineal} - Tiempo: {end_time - start_time:.6f} segundos")
```

Terminal output:

```
PS C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe"
Lista ordenada con Bubble Sort. Tiempo: 0.0176 segundos
Búsqueda binaria: posición 436 - Tiempo: 0.000000 segundos
Búsqueda lineal: posición 500 - Tiempo: 0.000000 segundos
```

En la imagen: Lista de 10.000 elementos ordenada con Bubble Sort.



```
25 # Prueba con 10.000 elementos
26 print("Prueba con 10.000 elementos:")
27 lista_original = random.sample(range(1, 100_000), 10_000)
28 target = lista_original[5000] # Valor asegurado
29
30 # Ordenar para aplicar búsqueda binaria
31 lista_ordenada = lista_original.copy()
32 lista_ordenada.sort()
33
34 # Búsqueda binaria
35 start_time = time.time()
36 pos_binaria = binary_search(lista_ordenada, target)
37 end_time = time.time()
38 print(f"Búsqueda binaria: posición {pos_binaria}, tiempo: {end_time - start_time:.6f} segundos")
39
40 # Búsqueda lineal
41 start_time = time.time()
42 pos_lineal = linear_search(lista_original, target)
43 end_time = time.time()
44 print(f"Búsqueda lineal: posición {pos_lineal}, tiempo: {end_time - start_time:.6f} segundos")
```

Terminal output:

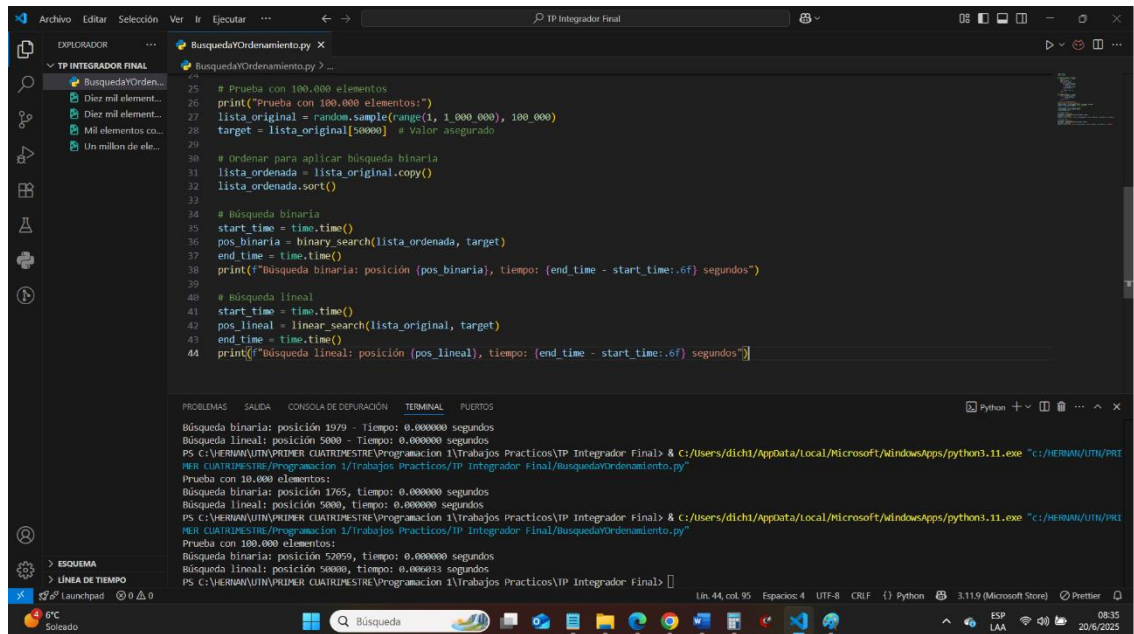
```
PS C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\Users\dic1\AppData\Local\Microsoft\WindowsApps\python3.11.exe"
Prueba con 10.000 elementos:
Búsqueda binaria: posición 1970 - Tiempo: 0.000000 segundos
Búsqueda lineal: posición 5000 - Tiempo: 0.000000 segundos
```

En la imagen: La misma lista ordenada con el método “sort()”, de Python. La diferencia en cuanto a resultados de tiempo es notoria.

- Verificación con 100.000 elementos: en la tercera prueba aumentamos significativamente el tamaño de la lista a 100.000 elementos.

## PROGRAMACION 1 – TUPAD – TRABAJO INTEGRADOR FINAL

Nuevamente, la lista fue ordenada con el método “sort()”. En este caso, la búsqueda binaria mantuvo un tiempo de ejecución extremadamente bajo, casi igual al de las listas más pequeñas, mientras que la búsqueda lineal mostró un claro crecimiento en su tiempo de respuesta. Este resultado demuestra de forma contundente que la búsqueda binaria es mucho más eficiente que la búsqueda lineal en listas grandes, siempre y cuando los datos estén previamente ordenados.



The screenshot shows a Python script named 'BusquedaYOrdenamiento.py' in a VS Code editor. The script tests binary and linear search on lists of 100,000, 1,000,000, and 1,000,000 elements. The terminal output shows that binary search is consistently fast (around 0.000000 seconds), while linear search becomes significantly slower as the list size increases (e.g., 0.006033 seconds for 1,000,000 elements).

```
25 # Prueba con 100,000 elementos
26 print("Prueba con 100,000 elementos:")
27 lista_original = random.sample(range(1, 1_000_000), 100_000)
28 target = lista_original[50000] # valor asegurado
29
30 # Ordenar para aplicar búsqueda binaria
31 lista_ordenada = lista_original.copy()
32 lista_ordenada.sort()
33
34 # Búsqueda binaria
35 start_time = time.time()
36 pos_binaria = binary_search(lista_ordenada, target)
37 end_time = time.time()
38 print(f"Búsqueda binaria: posición {pos_binaria}, tiempo: {end_time - start_time:.6f} segundos")
39
40 # Búsqueda lineal
41 start_time = time.time()
42 pos_lineal = linear_search(lista_original, target)
43 end_time = time.time()
44 print(f"Búsqueda lineal: posición {pos_lineal}, tiempo: {end_time - start_time:.6f} segundos")
```

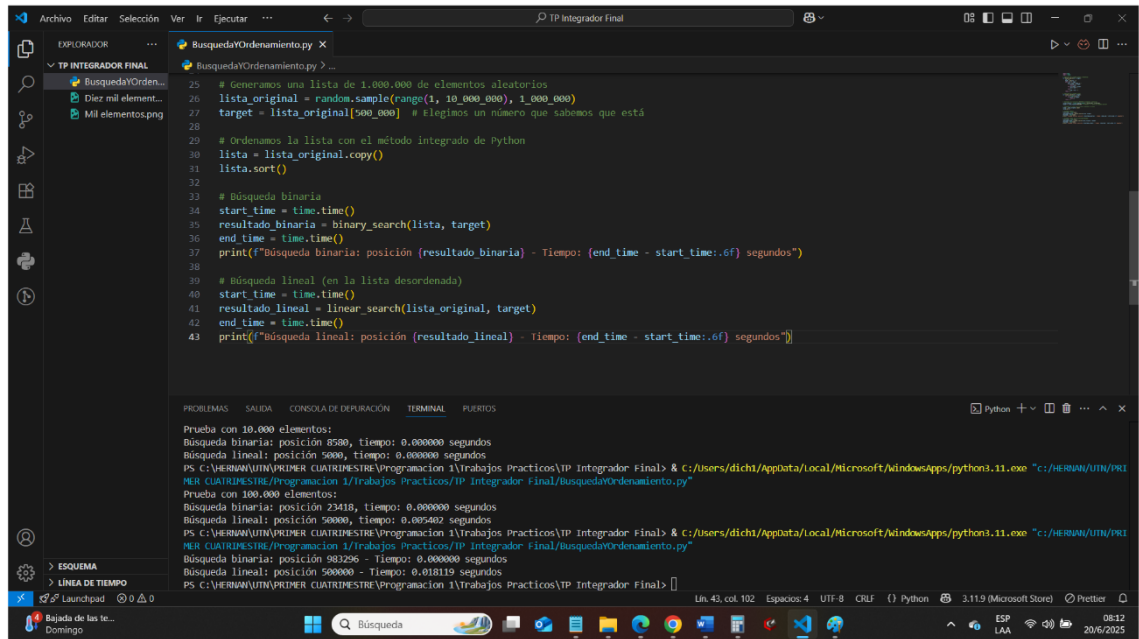
Terminal Output:

```
Búsqueda binaria: posición 1979 - Tiempo: 0.000000 segundos
Búsqueda lineal: posición 5000 - Tiempo: 0.000000 segundos
PS C:\HERIAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final> & C:/Users/dich1/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/HERIAN/UTN/PRI
MER CUATRIMESTRE/Programacion 1/Trabajos Practicos/TP Integrador Final/BusquedaYOrdenamiento.py"
Prueba con 100,000 elementos:
Búsqueda binaria: posición 1765, tiempo: 0.000000 segundos
Búsqueda lineal: posición 5000, tiempo: 0.000000 segundos
PS C:\HERIAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final> & C:/Users/dich1/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/HERIAN/UTN/PRI
MER CUATRIMESTRE/Programacion 1/Trabajos Practicos/TP Integrador Final/BusquedaYOrdenamiento.py"
Prueba con 1,000,000 elementos:
Búsqueda binaria: posición 52059, tiempo: 0.000000 segundos
Búsqueda lineal: posición 50000, tiempo: 0.006033 segundos
PS C:\HERIAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final>
```

- Verificación con 1.000.000 elementos: finalmente, se realizó una prueba con una lista de 1 millón de elementos aleatorios. Dada la enorme cantidad de datos, se vuelve a utilizar el método “sort()”. La búsqueda binaria se mantuvo extremadamente rápida. Por el contrario, la búsqueda lineal mostró un crecimiento exponencial en el tiempo necesario para recorrer la lista. Este caso extremo refuerza con claridad la conclusión de que la búsqueda binaria es significativamente más eficiente para estructuras grandes, siempre que se garantice el ordenamiento previo de la lista. También demuestra que, a medida que crece la cantidad de datos, la diferencia entre ambos métodos se vuelve cada vez más drástica.



## PROGRAMACION 1 – TUPAD – TRABAJO INTEGRADOR FINAL



```
25 # Generamos una lista de 1.000.000 de elementos aleatorios
26 lista_original = random.sample(range(1, 10_000_000), 1_000_000)
27 target = lista_original[500_000] # Elegimos un número que sabemos que está
28
29 # Ordenamos la lista con el método integrado de Python
30 lista = lista_original.copy()
31 lista.sort()
32
33 # Búsqueda binaria
34 start_time = time.time()
35 resultado_binaria = binary_search(lista, target)
36 end_time = time.time()
37 print(f"Búsqueda binaria: posición {resultado_binaria} - Tiempo: {end_time - start_time:.6f} segundos")
38
39 # Búsqueda lineal (en la lista desordenada)
40 start_time = time.time()
41 resultado_lineal = linear_search(lista_original, target)
42 end_time = time.time()
43 print(f"Búsqueda lineal: posición {resultado_lineal} - Tiempo: {end_time - start_time:.6f} segundos")
```

Problemas SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

Prueba con 10.000 elementos:  
Búsqueda binaria: posición 8580, tiempo: 0.000000 segundos  
Búsqueda lineal: posición 5000, tiempo: 0.000000 segundos  
PS C:\HERMAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final> & C:\Users\dichi\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:\HERMAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final\BusquedaVOrdenamiento.py"

Prueba con 100.000 elementos:  
Búsqueda binaria: posición 22418, tiempo: 0.000000 segundos  
Búsqueda lineal: posición 50000, tiempo: 0.005402 segundos  
PS C:\HERMAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final> & C:\Users\dichi\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:\HERMAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final\BusquedaVOrdenamiento.py"

Búsqueda binaria: posición 982266 - Tiempo: 0.000000 segundos  
Búsqueda lineal: posición 500000 - Tiempo: 0.018119 segundos  
PS C:\HERMAN\UTN\PRIMER CUATRIMESTRE\Programacion 1\Trabajos Practicos\TP Integrador Final> []

### 3.3 Decisiones de Diseño:

- Se utilizó Bubble Sort por su simpleza y claridad, a pesar de su baja eficiencia. Luego, con el fin de demostrar casos de listas grandes y extremos, se optó por el método “sort()” de Python.
- Se implementó búsqueda binaria iterativa por ser más eficiente en listas ordenadas y por evitar problemas de recursión con listas grandes.
- Se agregó una búsqueda lineal como comparación directa, con el fin de visualizar la diferencia de rendimiento.

### 3.4 Validación del funcionamiento:

El programa fue ejecutado múltiples veces con listas aleatorias y siempre produjo los resultados correctos, los cuales fueron:

- Ordenar la lista con éxito.
- Encontrar correctamente el valor buscado con ambos métodos.
- Demostrar que la búsqueda binaria es mucho más rápida, siempre que la lista esté ordenada.

## 4. Metodología Utilizada

Para el desarrollo de este trabajo integrador, se siguió una metodología práctica basada en la investigación individual, implementación en código y análisis comparativo. Los pasos realizados fueron los siguientes:

**4.1 Selección del tema:** Se eligió trabajar con algoritmos de búsqueda y ordenamiento por su importancia dentro de la programación básica y su aplicabilidad en una amplia variedad de contextos.

**4.2 Investigación teórica:** Se consultaron fuentes como el material proporcionado por la cátedra y documentación oficial de Python.

**4.3 Generación de datos:** Para las pruebas se generaron listas de números enteros aleatorios utilizando la función “random.sample()”, que permite obtener listas sin valores repetidos. En cada prueba, se seleccionó como elemento objetivo “(target)” un valor de la lista para garantizar que el resultado de búsqueda fuera exitoso.

**4.4 Decisión del ordenamiento en pruebas grandes:** Durante el desarrollo, se determinó que Bubble Sort no era viable para listas grandes (más de 10.000 elementos) debido a su complejidad  $O(n^2)$ , ya que genera tiempos de procesamiento muy elevados. En su lugar, se utilizó el método “sort()” integrado de Python, que implementa Timsort ( $O(n \log n)$ ), permitiendo ordenar listas de gran tamaño en tiempos razonables. Este cambio permitió mantener el foco en la comparación entre búsqueda binaria y búsqueda lineal, sin que el ordenamiento se convirtiera en un cuello de botella.

**4.5 Toma de tiempos:** Se utilizó el modulo “time” para medir los tiempos de ejecución de las funciones de búsqueda, permitiendo comparar de manera objetiva su rendimiento en diferentes volúmenes de datos (1.000, 10.000, 100.000 y 1.000.000 de elementos).

## 5.Resultados Obtenidos

A continuación, se presentan los resultados obtenidos tras la implementación de los algoritmos y la ejecución de pruebas con listas de diferentes tamaños. El foco estuvo puesto en comparar el rendimiento de la búsqueda binaria frente a la búsqueda lineal, y en validar los algoritmos desarrollados en condiciones reales.

**5.1 Validación del funcionamiento:** Se verificó el correcto comportamiento de cada función mediante pruebas con listas pequeñas, asegurando que los algoritmos devolvieran resultados esperados en posiciones distintas (inicio, medio, final).

**5.2 Generación controlada de listas:** Todas las listas se generaron aleatoriamente con `random.sample()` para mantener valores únicos, con control manual del elemento a buscar (target).

**5.3 Aplicación de ordenamiento eficiente:** Se utilizó Bubble Sort para listas pequeñas y `sort()` para listas grandes. Esto permitió mantener viabilidad en pruebas de hasta 1.000.000 de elementos.

**5.4 Comparaciones de rendimiento:** Los tiempos mostraron que la búsqueda binaria es altamente eficiente, manteniéndose prácticamente constante, mientras que la búsqueda lineal se vuelve cada vez más lenta a medida que crece la lista.

**5.5 Registro de pruebas:** Todos los resultados fueron obtenidos por consola y documentados con capturas para ser incluidos en el video. El código fue comentado y está disponible en el repositorio Git.

## 6.Conclusiones

**6.1 Aprendizajes obtenidos:** A lo largo del desarrollo del trabajo integrador, se logró comprender en profundidad el funcionamiento de los algoritmos de búsqueda binaria, búsqueda lineal y ordenamiento mediante Bubble Sort, además de reforzar habilidades prácticas en programación con Python, uso de funciones, manipulación de listas y análisis de tiempos de ejecución. También se aprendió a evaluar críticamente la viabilidad de aplicar ciertos algoritmos en distintos contextos, reconociendo limitaciones de eficiencia y adaptando la metodología cuando fue necesario.

**6.2 Evaluación de resultados:** Las pruebas realizadas demostraron de manera clara que:

- La búsqueda binaria es mucho más eficiente que la búsqueda lineal, especialmente cuando se trabaja con volúmenes grandes de datos.
- La eficiencia de búsqueda binaria se mantiene estable gracias a su estructura de decisión logarítmica.
- La búsqueda lineal escala en tiempo de forma proporcional a la cantidad de elementos, volviéndose poco práctica para listas muy grandes.
- En términos de ordenamiento, se comprobó que Bubble Sort funciona correctamente en listas pequeñas, pero resulta inviable en listas medianas o grandes, motivo por el cual se reemplazó por el método `sort()` de Python, más adecuado para listas extensas.

**6.3 Valoración personal:** Este trabajo permitió integrar conocimientos teóricos con la práctica real de programación. Más allá de los resultados obtenidos, fue valioso experimentar de forma directa cómo las decisiones de diseño afectan el rendimiento y cómo elegir las herramientas adecuadas puede marcar la diferencia entre un código funcional y uno ineficiente.

---

## 7. Bibliografía

---

Fuentes consultadas:

- Material teórico de la cátedra.
- Documentación oficial de Python: <https://docs.python.org/3/library/>
- Apuntes de clase y experimentación propia.

---

## 8. Anexos

---

### **8.1 Repositorio de código fuente en GitHub:**

El código desarrollado en Python se encuentra disponible en el siguiente repositorio público: <https://github.com/HernanDichiara1992/TP-Integrador-Final-Programacion1>

### **8.2 Video explicativo:**

El video tutorial donde se presentan el desarrollo y los resultados del trabajo se encuentra disponible en el siguiente enlace: <https://youtu.be/2ce-FKtFFz8>