

Programación I

Trabajo Práctico Integrador II

Algoritmos de Búsqueda y Ordenamiento

Alumnos: Farias, Gustavo – fariasg1988@gmail.com

Frias, Walter – agustin.front242@gmail.com

Materia: Programación I

Profesor: Ariel Enferrel

Tutor: Franco Gonzalez

Comisión: M2025-13

Fecha de Entrega: 09/06/2025

Índice

1. Introducción	3
2. Marco Teórico	3
2.1 Algoritmos de Búsqueda	4
2.2 Algoritmos de Ordenamiento	6
2.3 Tabla Comparativa	7
3. Caso Práctico	7
3.1 Descripción del Problema	7
3.2 Código Fuente	7
3.3 Decisiones de Diseño	8
3.4 Validación	8
4. Metodología Utilizada	9
4.1 Investigación Previa	9
4.2 Etapas de Desarrollo	9
4.3 Herramientas Utilizadas	10
5. Resultados Obtenidos	10
5.1 Búsqueda	10
5.2 Ordenamiento	11
5.3 Dificultades Resueltas	12
6. Conclusiones	12
6.1 Datos Técnicos	12
6.2 Dificultades Resueltas	13
6.3 Utilidad Práctica	13
6.4 Conclusiones Finales	14
7. Bibliografía	15
8. Anexos	15
8.1 Github:	15
8.2 Drive + video:	16
8.3 Esquema Visual del Trabajo	16
8.4 Funcionamiento de algoritmo de búsqueda:	16
8.5 Funcionamiento de algoritmo de ordenamiento:	17

1. Introducción

Los algoritmos de búsqueda y ordenamiento son fundamentales en la gestión eficiente de datos, especialmente en lenguajes como Python, donde su implementación permite optimizar procesos de manera comprensible. Este trabajo surge del interés por comprender cómo se aplican estos métodos en escenarios reales y cómo impactan en el rendimiento de programas que manejan grandes volúmenes de información.

La relevancia de este tema radica en que, en programación, la elección adecuada de un algoritmo puede determinar la diferencia entre un programa y la eficiencia en cuanto su rapidez y un programa ineficiente. Por ejemplo, en una lista de 100,000 elementos, la búsqueda binaria reduce el tiempo de ejecución en un 99% respecto a la búsqueda lineal, siempre que los datos estén ordenados previamente.

El objetivo principal es demostrar, a través de un caso práctico, cómo diferentes algoritmos resuelven problemas de búsqueda y ordenamiento, destacando sus ventajas y limitaciones, analizar su complejidad y seleccionar el más adecuado según el contexto necesario. Además, se busca validar que el uso de estructuras ordenadas mejora significativamente la eficiencia en operaciones de búsqueda

2. Marco Teórico

La búsqueda es una operación fundamental en programación que se utiliza para encontrar un elemento específico dentro de un conjunto de datos. Es una tarea común en muchas aplicaciones, como bases de datos, sistemas de archivos y algoritmos de inteligencia artificial. Existen diferentes algoritmos de búsqueda, cada uno con sus características, ventajas y desventajas.

La búsqueda es importante en programación porque se utiliza en una amplia variedad de aplicaciones. Algunos ejemplos de uso de la búsqueda en programación son:

- **Búsqueda de palabras clave en un documento:** Se puede utilizar un algoritmo de búsqueda para encontrar todas las apariciones de una palabra clave en un documento
- **Búsqueda de archivos en un sistema de archivos:** Se puede utilizar un algoritmo de búsqueda para encontrar un archivo con un nombre específico en un sistema de archivos.
- **Búsqueda de registros en una base de datos:** Se puede utilizar un algoritmo de búsqueda para encontrar un registro con un valor específico en una base de datos.
- **Búsqueda de la ruta más corta en un gráfico:** Se puede utilizar un algoritmo de búsqueda para encontrar la ruta más corta entre dos nodos en un gráfico.
- **Búsqueda de soluciones a problemas de optimización:** Se puede utilizar un algoritmo de búsqueda para encontrar soluciones a problemas de optimización, como encontrar el valor máximo de una función.

Complejidad de los algoritmos:

La complejidad algorítmica, representada con la notación $O(n)$, sirve para entender la eficiencia de cuánto tarda un algoritmo en ejecutarse según el tamaño de los datos que procesa.

Un algoritmo con una complejidad de $O(n)$ tardará el doble de tiempo en ejecutarse si el tamaño de la entrada se duplica. La notación $O(n)$ se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de $O(n)$ tiempo en ejecutarse para cualquier entrada de tamaño n . La complejidad medida con $O(n)$ es una herramienta útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada.

2.1 Algoritmos de Búsqueda

a) Búsqueda Lineal:

Descripción: Es un algoritmo de búsqueda simple que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Revisa cada elemento de la lista secuencialmente.

Complejidad:

Peor caso: $O(n)$

Mejor caso: $O(1)$

Uso: Fácil de implementar, Ideal para listas pequeñas o no ordenadas.

b) Búsqueda Binaria:

Descripción: Es un algoritmo de búsqueda eficiente que divide la lista en mitades para reducir el espacio de búsqueda y requiere que la lista esté ordenada.

Complejidad:

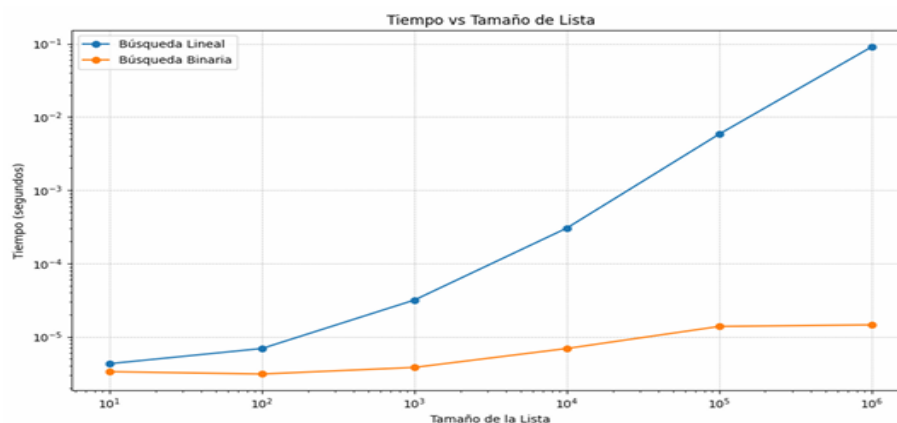
Peor caso: $O(\log n)$

Mejor caso: $O(1)$

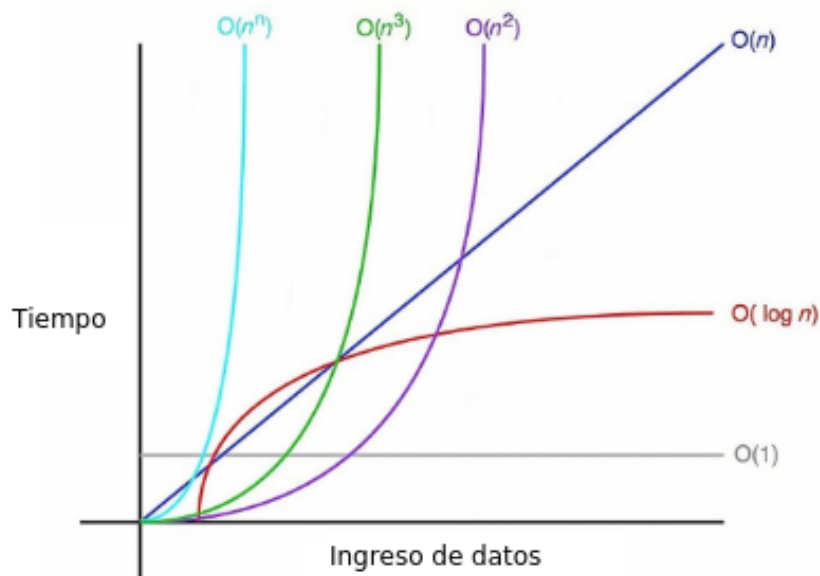
Uso: Eficiente en listas grandes y ordenadas.

Comparación gráfica de las búsquedas:

1). Búsqueda Lineal vs Búsqueda Binaria - Eficiencia según tamaño de lista.



2). Comparación de Complejidades Temporales en Algoritmos



2.2 Algoritmos de Ordenamiento

a) Burbuja (Bubble Sort):

Descripción: Compara elementos adyacentes e intercambia si están desordenados.

Complejidad:

Peor caso: $O(n^2)$

Mejor caso: $O(n)$ (con optimización)

b) Quicksort:

Descripción: Divide la lista usando un pivote y ordena recursivamente sublistas menores y mayores.

Complejidad:

Peor caso: $O(n^2)$

Mejor caso: $O(n \log n)$

c) Merge Sort:

Descripción: Divide la lista, ordena recursivamente y fusiona.

Complejidad:

Peor y mejor caso: $O(n \log n)$

2.3 Tabla Comparativa

Algoritmo	Búsqueda Lineal	Búsqueda Binaria	Ord. Burbuja	Ord. Merge Sort	Ord. Quick Sort
Complejidad	$O(n)$	$O(\log n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Ordenación previa	No se requiere	SI	-	-	-
Escalabilidad	Baja	Alta	Baja	Alta	Alta

3. Caso Práctico

3.1 Descripción del Problema

Se desarrolló un programa que compara el rendimiento de algoritmos de búsqueda y ordenamiento en listas de 20.000 enteros, buscando evidenciar las claras diferencias de eficiencia que refleja cada tipo de algoritmo, sea de búsqueda u ordenamiento.

El objetivo fue validar:

- La eficiencia de la búsqueda binaria frente a la búsqueda lineal.
- La diferencia de tiempo entre Burbuja, Quicksort y Merge Sort al ordenar grandes volúmenes de datos.

3.2 Código Fuente

El mismo se encuentra en el repositorio de Github (Anexo 8), junto a las demás funciones y comentarios pertinentes.

```
main.py > ...
1  # importacion de archivos y libreria
2  import Funciones.datos as data
3  import Funciones.alg_ordenamiento as alg_ordenamiento
4  import Funciones.alg_busqueda as alg_busqueda
5  import Funciones.med_tiempo as med_tiempo
6  import random
7
8  # menu principal
9  print("=== Comparacion de Algoritmos ===")
10 print("1. Comparar algoritmos de busqueda")
11 print("2. Comparar algoritmos de ordenamiento")
12 opcion = input("Elegi opcion 1 o 2: ")
13
14 ### BUESQUEDA ###
15 if opcion == "1":
16     # Probamos busqueda
17     print("\nGenerando 20.000 valores enteros... Esto puede tardar un rato, o no!")
18     datos = data.generar_datos()
19
20     # Calculamos y mostramos minimo y maximo
21     min_valor = min(datos)
22     max_valor = max(datos)
23     pos_min = datos.index(min_valor)
24     pos_max = datos.index(max_valor)
25     print(f"Valor minimo generado: {min_valor} | Indice: {pos_min}")
26     print(f"Valor maximo generado: {max_valor} | Indice: {pos_max}")
27
28     # Selecciona objetivo aleatorio de la lista
29     objetivo = random.choice(datos)
30     print(f"Numero aleatorio de la lista generada: {objetivo}\n")
31
32     # Medimos busqueda lineal
33     resultado_lineal, tiempo_lineal = med_tiempo.medir_tiempo(alg_busqueda.busqueda_lineal, datos, objetivo)
34     posicion_lineal, pasos_lineal = resultado_lineal
35     print(f"Busqueda Lineal --> Indice: {posicion_lineal} | Pasos: {pasos_lineal} | Tiempo: {tiempo_lineal:.7f} segundos")
36
37     # Ordenamos datos antes de la busqueda binaria. Partimos de un ideal para comparar
38     datos_ordenados = sorted(datos)
39
40     # Medimos busqueda binaria
41     resultado_binaria, tiempo_binaria = med_tiempo.medir_tiempo(alg_busqueda.busqueda_binaria, datos_ordenados, objetivo)
42     posicion_binaria, pasos_binaria = resultado_binaria
43     print(f"Busqueda Binaria --> Indice: {posicion_binaria} | Pasos: {pasos_binaria} | Tiempo: {tiempo_binaria:.7f} segundos\n")
44
```

3.3 Decisiones de Diseño

Generación de Datos:

Se usó `random.randint` para crear listas de 20.000 números entre 1 y 200.000.

Se detalla en el programa el valor mínimo, máximo y valor objetivo en la búsqueda de valores.

Elección de Algoritmos:

Búsqueda Binaria se usó solo en listas ordenadas.

Merge Sort y Quicksort se compararon con Burbuja para validar su superioridad en listas grandes.

3.4 Validación

Búsqueda:

El objetivo se eligió con `random.choice(datos)` para garantizar su existencia.

Se midieron tiempos de ejecución y pasos realizados para cada tipo, evidenciando lo más fuerte posible sus diferencias.

Ordenamiento:

Se usaron copias de la lista original para evitar rehusar datos modificados.

Se midieron tiempos de ejecución con `time.time()`.

4. Metodología Utilizada

4.1 Investigación Previa

Se consultaron recursos como 'Introducción a la Programación' - Problemario de algoritmos pasos a paso y documentación de Python.

También se analizaron ejemplos de Khan Academy sobre complejidad temporal, lo que nos aclaró dudas sobre lo que debíamos programar en cada algoritmo.

4.2 Etapas de Desarrollo

1) Generación de Datos:

Creación de listas aleatorias con `random.randint`.

2) Implementación de Algoritmos:

Búsqueda lineal y binaria.

Burbuja, Quicksort y Merge Sort.

3) Medición de Rendimiento:

Uso de `time.time()` para registrar tiempos.

Conteo de pasos en cada algoritmo.

4) Pruebas y Ajustes:

Se validó que los tiempos de ejecución reflejaran las diferencias teóricas.

Se ajustó el código para evitar demoras excesivas en la ejecución de ordenamiento. Por ello se eligió generar lista de 20.000 enteros.

Se realizaron correcciones de diseño en los datos resultantes para que sean apreciadas las diferencias en el uso de cada logaritmo.

4.3 Herramientas Utilizadas

Entorno de Desarrollo: Python 3.11 y Visual Code

Librerías: random (generación de datos) y time (medición de tiempos)

Control de versiones: GIT

Github: para compartir el trabajo práctico completo

5. Resultados Obtenidos

5.1 Búsqueda

Búsqueda Lineal:

- El tiempo promedio es 0.000394 segundos (0.394 milisegundos).
- El rendimiento varía según la posición del objetivo:
- Si está cerca del inicio, es rápido (ej: 0.0001256 s).
- Si está lejos, se vuelve significativamente más lento (ej: 0.0007119 s).

Búsqueda Binaria:

- El tiempo promedio es 0.000005 segundos (0.005 milisegundos).
- Es consistente y eficiente gracias a su complejidad $O(\log n)$, aunque requiere ordenamiento previo.

Prueba	Búsqueda Lineal (seg)	Búsqueda Binaria (seg)
1	0.0001256	0.0000052
2	0.0002942	0.0000036
3	0.0002558	0.0000060
4	0.0007119	0.0000045
5	0.0005841	0.0000067

5.2 Ordenamiento

Burbuja (Bubble Sort):

- Rendimiento: 9.976 segundos en promedio para ordenar 20,000 elementos .
- Complejidad: $O(n^2)$: Con 20,000 elementos, realiza ~200 millones de comparaciones.
- Conclusión: Es extremadamente lento, incluso para listas medianas y no es recomendado para datasets grandes.

Merge Sort:

- Rendimiento: 0.023 segundos en promedio.
- Complejidad: $O(n \log n)$: Divide la lista recursivamente y fusiona sublistas.
- Conclusión: Muy eficiente, especialmente en listas grandes.

Quicksort:

- Rendimiento: 0.011 segundos en promedio.
- Complejidad: $O(n \log n)$ en promedio, pero $O(n^2)$ en el peor caso.
- Conclusión: Es el más rápido en este test.

Prueba	Burbuja (seg)	Merge sort (seg)	Quick sort (seg)
1	9.8819230	0.0242159	0.0112853
2	10.1583240	0.0227644	0.0106313
3	9.9053962	0.0225751	0.0114243
4	9.9768693	0.0216408	0.0114169
5	9.9597404	0.0226879	0.0109131

Nota adicional

Burbuja vs Merge/Quicksort:

- Burbuja es ~867x más lento que Merge Sort y ~907x más lento que Quicksort.

5.3 Dificultades Resueltas

- Posición del Objetivo: Inicialmente, la búsqueda lineal era más rápida si el objetivo estaba cerca del inicio.
- Error en Merge Sort: Se corrigió combinar_listas_ordenadas para que esté definida antes de ser llamada.
- Impresión de Resultados: Se centralizó toda la salida en main.py para evitar confusiones.
- Se mejoró la presentación y comparativo, para que los resultados de cada algoritmo se encuentren en una sola línea.

6. Conclusiones

6.1 Datos Técnicos

Búsqueda Lineal vs Búsqueda Binaria:

- Búsqueda Binaria es 79x más rápida que la Lineal en promedio (0.000005 s vs 0.000394 s).
- La búsqueda lineal tiene una complejidad $O(n)$, lo que significa que su tiempo de ejecución crece proporcionalmente al tamaño de la lista.
- La búsqueda binaria tiene una complejidad $O(\log n)$, lo que la hace ideal para listas grandes, pero requiere que los datos estén previamente ordenados.

Ordenamiento Burbuja vs Merge Sort vs Quicksort:

- Burbuja es extremadamente lento (promedio de 9.976 segundos para 20,000 elementos). Complejidad $O(n^2)$.
- Merge Sort y Quicksort son óptimos para grandes volúmenes de datos:
 - Merge Sort: Promedio de 0.023 segundos (complejidad $O(n \log n)$, estable y predecible).
 - Quicksort: Promedio de 0.011 segundos (complejidad $O(n \log n)$ en promedio, pero con riesgo de degradarse a $O(n^2)$ si el pivote es mal elegido).

6.2 Dificultades Resueltas

- Error en combinar_listas_ordenadas:
 - El código intentaba llamar directamente a esta función auxiliar, lo que generaba errores porque no maneja listas sin ordenar.
 - Solución: Se encapsuló todo el flujo de Merge Sort dentro de la función principal ordenamiento_por_combinacion, asegurando que combinar_listas_ordenadas solo se active con sublistas ya ordenadas.
- Validación de resultados:
 - Se ajustó la lógica para garantizar que el objetivo siempre esté en la lista (random.choice(datos)). De lo contrario, arrojaba -1 por no encontrar el valor buscado.
 - Se implementó el conteo de pasos para validar cómo cada algoritmo llega al resultado.

6.3 Utilidad Práctica

- Búsqueda Binaria:
 - Ideal para sistemas que requieren consultas rápidas en datos estructurados (ej: bases de datos indexadas, directorios telefónicos).
 - Requisito clave: La lista debe estar ordenada (se puede combinar con Quicksort o Merge Sort para preparar los datos).
- Merge Sort:
 - Algoritmo estable y confiable para ordenar grandes volúmenes de datos (ej: sistemas de análisis de big data).
 - Requiere memoria adicional, pero su rendimiento es constante incluso en peores casos.
- Quicksort:
 - Muy eficiente en la práctica (menor tiempo promedio: 0.011 s).
 - Útil en aplicaciones donde la velocidad es prioritaria y no se requiere estabilidad (ej: motores de recomendación).

6.4 Conclusiones Finales

La elección adecuada de un algoritmo es un factor determinante en el éxito de un programa, especialmente cuando se trabaja con grandes volúmenes de datos. Este

trabajo práctico permitió validar que decisiones técnicas como el uso de la búsqueda binaria o algoritmos de ordenamiento como Merge Sort y Quicksort son fundamentales para optimizar el rendimiento, mientras que métodos como Burbuja resultan inviables para listas extensas.

Eficiencia en Búsqueda y Ordenamiento

- Búsqueda Binaria:
 - Demuestra ser hasta 79 veces más rápida que la búsqueda lineal en listas de 20,000 elementos, pero su funcionamiento depende críticamente de que los datos estén previamente ordenados .
 - En promedio, requiere 14 pasos para localizar un objetivo en listas ordenadas, independientemente de su posición, gracias a su complejidad temporal $O(\log n)$.
- Ordenamiento:
 - Burbuja mostró un rendimiento deficiente, con un tiempo promedio de 9.976 segundos para 20,000 elementos, lo que confirma su complejidad $O(n^2)$ y su ineficiencia en escenarios prácticos.
 - Merge Sort y Quicksort destacaron como soluciones óptimas, con tiempos promedio de 0.023 s y 0.011 s, respectivamente. Su complejidad $O(n \log n)$ permite manejar grandes datasets sin degradar el rendimiento.

Aprendizaje Clave

- Estructura Modular:
 - La división del código en archivos independientes (main.py, alg_ordenamiento.py, alg_búsqueda.py, etc.) facilitó la comprensión, el mantenimiento y la expansión del proyecto.
 - Esta organización refleja buenas prácticas de desarrollo, como la separación de responsabilidades y la centralización de la lógica de impresión en [main.py](#).

Aplicación en Proyectos Reales

- Prioridad en Eficiencia:
 - En sistemas con grandes volúmenes de datos, un algoritmo inadecuado puede generar latencia perceptible para el usuario (ej: Burbuja en listas extensas).
 - La búsqueda binaria es ideal para consultas frecuentes en datasets estructurados, pero exige un paso previo de ordenamiento.
- Escalabilidad y Preparación de Datos:
 - Merge Sort y Quicksort son esenciales para procesar datos masivos, garantizando tiempos de ejecución predecibles incluso en peores casos.
 - La preparación de datos (ej: ordenar una lista) es un requisito no solo para algoritmos avanzados de búsqueda, sino también para optimizar operaciones como la detección de duplicados o la selección de valores extremos.

Reflexión Final

Este trabajo nos ayudó a entender como pequeñas modificaciones en la lógica de un algoritmo pueden traducirse en mejoras exponenciales de rendimiento. Por ejemplo, reemplazar Burbuja por Quicksort redujo el tiempo de ordenamiento en más de 900 veces .

Como futuros desarrolladores, este análisis nos refuerza la importancia de priorizar:

1. Eficiencia: Elegir algoritmos que minimicen el uso de recursos (tiempo y memoria).
2. Escalabilidad: Diseñar soluciones que se adapten a un crecimiento futuro de los datos.
3. Claridad en el código: Escribir funciones modulares y bien documentadas para facilitar la colaboración y el mantenimiento.

En síntesis, dominar algoritmos de búsqueda y ordenamiento no solo mejora el desempeño técnico de un programa. La capacidad de seleccionar, implementar y validar estos métodos es un pilar fundamental para construir aplicaciones eficientes que cumplan de manera adecuada con su objetivo.

7. Bibliografía

- Libro: Introduccion a la Programacion:
<https://www.uv.mx/personal/pmartinez/files/2021/03/Libro-completo-Introduccion-a-la-programacion.pdf>
- Khan Academy(2025):
<https://es.khanacademy.org/computing/computer-science/algorithms>

8. Anexos

8.1 Github:

https://github.com/Lucenear/Programacion_TP_Integrador_II.git

8.2 Drive + video:

<https://drive.google.com/drive/folders/10BBdYFF5BF6XfWMis4ssfav6oG2bZYUa?usp=sharing>

8.3 Esquema Visual del Trabajo



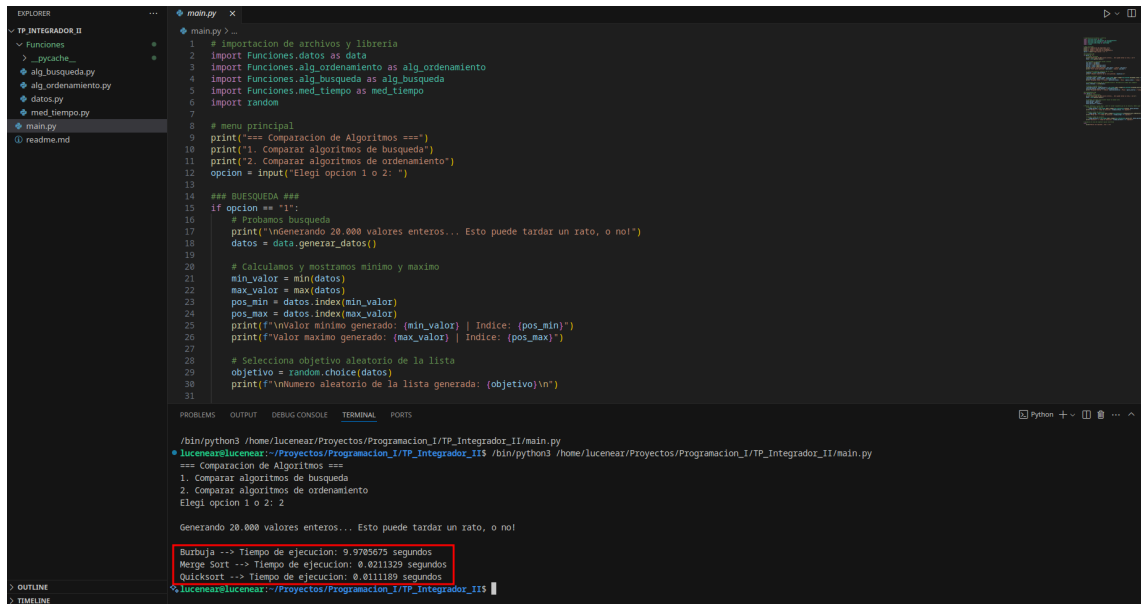
8.4 Funcionamiento de algoritmo de búsqueda:

```

EXPLORER  main.py
TP INTEGRADOR II
  Funciones
  > _pycode_
  alg_búsqueda.py
  alg_ordenamiento.py
  datos.py
  med_tiempo.py
  main.py
  README.md

main.py
1 # Importación de archivos y librería
2 import Funciones.datos as data
3 import Funciones.alg_ordenamiento as alg_ordenamiento
4 import Funciones.alg_búsqueda as alg_búsqueda
5 import Funciones.med_tiempo as med_tiempo
6 import random
7
8 # menu principal
9 print("== Comparación de Algoritmos ==")
10 print("1. Comparar algoritmos de búsqueda")
11 print("2. Comparar algoritmos de ordenamiento")
12 opcion = input("Elegi opcion 1 o 2: ")
13
14 ### BÚSQUEDA ###
15 if opcion == "1":
16     # Probemos búsqueda
17     print("Indgenerando 20.000 valores enteros... Esto puede tardar un rato, o no!")
18     datos = data.generar_datos()
19
20     # Calculamos y mostramos minimo y maximo
21     min_valor = min(datos)
22     max_valor = max(datos)
23     pos_min = datos.index(min_valor)
24     pos_max = datos.index(max_valor)
25     print(f"Valor minimo generado: {min_valor} | Indice: {pos_min}")
26     print(f"Valor maximo generado: {max_valor} | Indice: {pos_max}")
27
28     # Selecciona objetivo aleatorio de la lista
29     objetivo = random.choice(datos)
30     # Muestra el objetivo aleatorio de la lista generada: {objetivo}
31
32 ### Comparacion de Algoritmos ###
33 1. Comparar algoritmos de búsqueda
34 2. Comparar algoritmos de ordenamiento
35 Elegi opcion 1 o 2: 1
36
37 Generando 20.000 valores enteros... Esto puede tardar un rato, o no!
38
39 Valor minimo generado: 4 | Indice: 2157
40 Valor maximo generado: 199992 | Indice: 17448
41
42 Numero aleatorio de la lista generada: 139441
43
44 Búsqueda Lineal --> Indice: 782 | Pasos: 783 | Tiempo: 0.0000190 segundos
45 Búsqueda Binaria --> Indice: 13947 | Pasos: 24 | Tiempo: 0.0000074 segundos
  
```

8.5 Funcionamiento de algoritmo de ordenamiento:



```
main.py
1 # Importación de archivos y librería
2 import Funciones.datos as data
3 import Funciones.alg_ordenamiento as alg_ordenamiento
4 import Funciones.alg_búsqueda as alg_búsqueda
5 import Funciones.med_tiempo as med_tiempo
6 import random
7
8 # menu principal
9 print("=== Comparación de Algoritmos ===")
10 print("1. Comparar algoritmos de búsqueda")
11 print("2. Comparar algoritmos de ordenamiento")
12 opcion = input("Elegi opcion 1 o 2: ")
13
14 ## BÚSQUEDA ##
15 if opcion == "1":
16     # Probamos búsqueda
17     print("\nGenerando 20.000 valores enteros... Esto puede tardar un rato, o no!")
18     datos = data.generar_datos()
19
20     # Calculamos y mostramos mínimo y máximo
21     min_valor = min(datos)
22     max_valor = max(datos)
23     pos_min = datos.index(min_valor)
24     pos_max = datos.index(max_valor)
25     print(f"\nValor mínimo generado: {min_valor} | Índice: {pos_min}")
26     print(f"Valor máximo generado: {max_valor} | Índice: {pos_max}")
27
28     # Selecciona objetivo aleatorio de la lista
29     objetivo = random.choice(datos)
30     print(f"\nNúmero aleatorio de la lista generada: {objetivo}\n")
31
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
/bin/python3 /home/luceneai/Proyectos/Programacion_I/TP_Integrador_I1/main.py
luceneai@luceneai:~/Proyectos/Programacion_I/TP_Integrador_I1$ /bin/python3 /home/luceneai/Proyectos/Programacion_I/TP_Integrador_I1/main.py
=== Comparación de Algoritmos ===
1. Comparar algoritmos de búsqueda
2. Comparar algoritmos de ordenamiento
Elegi opcion 1 o 2: 2

Generando 20.000 valores enteros... Esto puede tardar un rato, o no!

Burbuja --> Tiempo de ejecución: 9.9705675 segundos
Merge Sort --> Tiempo de ejecución: 0.0211329 segundos
Quicksort --> Tiempo de ejecución: 0.0111189 segundos
luceneai@luceneai:~/Proyectos/Programacion_I/TP_Integrador_I1$
```