Trabajo Práctico Integrador
Investigación aplicada en Python: Algoritmos de Búsqueda y Ordenamiento.

Alumnos: Daniela Nahir Romero (danielanahirromero@gmail.com), Juan Romero (noastranoa@outlook.es)
Materia: Programación I
Profesor: Nicolás Quirós (Comisión 20)
Tutor: Francisco Quarñolo
Fecha de Entrega: 09/06/2025

1. Introducción

El presente trabajo tiene como objetivo explorar y analizar diferentes algoritmos de búsqueda y ordenamiento, ya que son fundamentales en la creación de programas eficientes en cuanto a rendimiento. También son importantes para hacer aplicaciones escalables con una buena experiencia de usuario. Con esto en mente, se realizó un programa y se comparó los resultados que este arrojaba al cambiar los métodos de búsqueda y ordenamiento aplicados. Esto se logró cambiando también el volumen de los datos para ver el impacto producido en el rendimiento de los algoritmos. De esta manera, se logró identificar las ventajas y desventajas de cada uno de los métodos y su

aplicabilidad en el desarrollo de software.

---

2. Marco Teórico

A continuación se detallarán definiciones y conceptos relevantes para la comprensión de los algoritmos de búsqueda y ordenamiento y su análisis, que fueron necesarios para la realización de este trabajo.

Algoritmo de búsqueda: es el procedimiento por el medio del cual se puede encontrar un elemento en un conjunto de datos, y determinar si está o no presente y devolver su posición. Su importancia radica en que son la base de muchas aplicaciones que se utilizan con frecuencia, desde la búsqueda en un navegador web o en el sistema de archivos de una computadora. Ayudan a acceder a la información, por lo cual deben ser eficientes para que los programas que utilizan grandes cantidades de datos, puedan devolverlos con facilidad. Los más conocidos son incluyen:

- la búsqueda lineal (que busca el elemento de manera secuencial, recorriendo el los datos uno por uno, lo cual es efectivo pero muy lento si se tiene grandes volúmenes de datos), y
- la búsqueda binaria (que logra mayor eficiencia dividiendo de forma repetitiva los datos previamente ordenados en dos mitades, y al encontrar el dato del medio lo compara con el buscado y determina si debe seguir la búsqueda en la mitad de la derecha o de la izquierda, y luego descarta la mitad innecesaria).

Algoritmo de ordenamiento: es un conjunto de instrucciones que reorganiza los elementos de un conjunto de datos en una secuencia específica. El ordenamiento puede ayudar a permitir mejor uso de algoritmos de búsqueda, como la búsqueda binaria. Algunos de los algoritmos de ordenamiento más comunes son:

- Bubble Sort (intercambia elementos adyacentes si están en el orden incorrecto, o si no los deja en esa misma posición, y luego continua con el siguiente),
- Insertion Sort (construye una lista ordenada de a un elemento a la vez, insertandolo en la posición correcta dentro de la parte ordenada que se va construyendo),
- Quicksort (basado en la estrategia “divide y conquistaras”, se elige un elemento y se determina que de un lado irán los menores a el y del otro los mayores, y luego se aplica la misma lógica a las sublistas de cada lado, así hasta terminar con

sublistas ordenadas que forman la lista ordenada final), y
- Selection Sort (encuentra el elemento más pequeño y lo coloca al principio, y así sigue sucesivamente).

Complejidad Algorítmica (Notación Big O): Es una medida de rendimiento de un algoritmo en términos de tiempo de ejecución y uso de memoria. Se expresa en función del tamaño de los datos de entrada. Expresa como el uso de los recursos crece a medida que el volumen de datos aumenta. Esta notación se evalúa para un algoritmo en el mejor de los casos, el peor y el caso promedio. Las siguientes son las expresiones más comunes:

- O(1) = el tiempo aumenta de manera constante, independientemente del tamaño de los datos, como cuando se accede a un elemento por su índice.
- O(logn) = tiempo logarítmico que aumenta lentamente a medida que aumentan los datos, como en la búsqueda binaria.
- O(n) = el tiempo aumenta de manera proporcional al volumen de los datos, como en la búsqueda lineal
- O(nlogn) = el tiempo es log-lineal (más eficiente que el lineal, pero menos que el logarítmico), lo cual se da en el caso promedio en Quicksort y MergeSort
- O(n2) = tiempo cuadrático que aumenta drásticamente con el aumento del volumen de datos, como el peor de los casos con Bubble Sort, Insertion Sort y Selection Sort.

---

3. Caso Práctico

El caso a desarrollar en este trabajo es la creación de una pequeña aplicación web en Python que nos permitirá como usuarios experimentar interactivamente con los algoritmos de búsqueda y ordenamiento. La aplicación permitirá crear listas de datos de diferentes tamaños y elegir los algoritmos a aplicar sobre ellas. Se visualizarán los resultados en consola. De esta forma no solo podremos practicar la creación de los algoritmos, sino que también podremos derivar conclusiones acerca de su comportamiento con los resultados de nuestra aplicación y evaluar los mejores algoritmos según el conjunto de datos.

---

4. Metodología Utilizada

Para realizar este trabajo se comenzó por realizar una introducción a los temas del trabajo y sus conceptos relacionados. Para ello se utilizaron el material de la cátedra y los videos de los profesores, así como la ayuda de la inteligencia artificial para despejar dudas y lograr una mejor comprensión sobre el tema. En la etapa de diseño del problema, se tomaron en cuenta los ejercicios resueltos por los profesores en los videos y los modelos de resolución de trabajos integradores.

Para el diseño de la aplicación se tuvo en cuenta el concepto de modularización. Se buscó que el programa principal fuera lo más declarativo posible, haciendo que los detalles internos sean abstraídos con el uso de funciones específicas que encapsulan tareas concretas. De esta forma, la aplicación fue más legible y fácil de entender. Como ya se mencionó, nos centramos en la reutilización del código proporcionado en la cátedra. Así fue que incorporamos las funciones de búsqueda y ordenamiento para la base de nuestra aplicación. Estas fueron:

- burbuja(arr), selección(arr), insercion(arr) y quicksort(arr) para algoritmos de ordenamiento,
- y búsqueda:lineal(lista, objetivo), búsqueda binaria(lista, objetivo) para algoritmos de búsqueda

Por otro lado generamos las siguientes funciones para que el programa funcione:

- generate_random_list(size) para generar las listas de datos de prueba,
- elegir_ordenamiento(datos) y elegir_busqueda(datos) para la interacción con el usuario, solicitando que este elija algoritmos a utilizar,
- calcular_tiempo(funcion), donde tomamos lo mostrado en los videos acerca del uso de la librería time para medir el tiempo que pasa entre el inicio y el fin de la ejecución de una función y creamos una función para reutilizar ese método en varias ocasiones,
- y probar_algoritmos, la cual es la que determina el flujo de la aplicación desde la solicitud de entrada del usuario, hasta la ejecución y la presentación de los resultados,

En cuanto al diseño del código, elegimos empezar a pensar el programa desde los componentes más grandes y la estructura principal. Decidimos que lo mejor era que el usuario elija qué algoritmos usar: 1 de busqueda y 1 de ordenamiento por vez (de esta forma nos pareció que el usuario podía ver cuando tiempo se tardaba, y luego volver a correr el programa y elegir otros algoritmos y comprobar los resultados). A medida que íbamos identificando operaciones complejas o repetitivas, sacamos esos

procesos del flujo principal y los encapsulamos en funciones aparte. Esto facilitó el desarrollo del código, porque pudimos depurarlo poco a poco, revisando que cada subtarea funcione de manera esperada.

El rendimiento de los algoritmos fue la pregunta que inspiró la creación del programa. Para aprender acerca de ello, se realizaron pruebas específicas con el programa con diferentes tipos de datos: listas vacías, ordenadas, y con diferentes tamaños de datos. De esta forma pudimos conocer de primera mano el rendimiento de los algoritmos de búsqueda y ordenamiento inclusive en casos borde. Además, para medir de manera precisa el tiempo de ejecución de los algoritmos, repetimos las mismas ejecuciones con el mismo tipo y tamaño de datos múltiples veces, para tomar la medición promedio. Esto se logró reemplazando temporalmente las lineas que generaban listas aleatorias:

```
  def probar_algoritmos():
    print("Bienvenido a la prueba de algoritmos")
    cantidad = int(input("Ingrese la cantidad aproximada de datos a
procesar: "))
    datos = generate_random_list(cantidad)
```

A estas las reemplazamos en un algoritmo exactamente igual, a excepción del hecho de que permitia la inserción directa de listas específicas como argumento:

```
def probar_algoritmos_prueba(datos):
```

Dicho algoritmo se gurdo en un archivo pruebas.py y se lo utilizó para hacer todas la experimentaciones controladas necesarias para este trabajo.

Por otro lado, en relación a las herramientas utilizadas para el desarrollo de la aplicación, podemos mencionar que se utilizó VS Code y Python 3.11.9. La ejecución de las pruebas se hizo desde la terminal de Powershell y se hicieron operaciones de control de versiones con Git y Github.

El trabajo fue resuelto de manera colaborativa por ambos integrantes, distribuyendo las tareas, en líneas generales, de la siguiente manera:

- Daniela Romero: responsable de la redacción del marco teórico y la descripción del diseño del programa, la edición del video.

- Juan Romero: responsable de la redacción de la introducción y la sección de resultados obtenidos y la creación del anexo.

- Ambos integrantes se reunieron a través de videollamadas de Discord y editaron de manera conjunta lo previamente redactado individualmente utilizando la plataforma de Google Docs. A su vez, utilizaron la plataforma de videollamadas para realizar la creación y ejecución del programas en vivo, permitiendo esto que ambos estudiantes participaran en la solución de los desafíos técnicos que a medida que iba surgiendo y de las pruebas del código.

---

5. Resultados Obtenidos

Para asegurarnos que nuestro programa funcionará de manera adecuada y lograr hacer las comparaciones del rendimiento de los algoritmos, tomamos listas de datos fijas (que no cambiaban en cada ejecución) y aplicamos diferentes algoritmos. De esta forma pudimos ver que en ciertos casos, el volumen de los datos, la posición del elemento buscado y otros factores, afectaba el resultado. A continuación se detallan algunas de nuestras múltiples pruebas con el programa.

a) **Experimentación con búsqueda binaria y diferentes métodos de ordenamiento:** En esta prueba medimos los tiempos de ejecución (en milisegundos) de la búsqueda binaria y de cuatro algoritmos de ordenamiento. Todas las mediciones se realizaron sobre listas de 10.000 elementos, manteniendo siempre el mismo elemento como objetivo de la búsqueda. A continuación se muestra los resultados junto con el número de captura de pantalla correspondiente:

Dato	Algoritmo de búsqueda	Tiempo Busqueda (ms)	Algoritmo de ordenamiento	Tiempo Ordenamiento (ms)	Captura de pantalla n°
un elemento entre 10000 (siempre el mismo elemento)	Binaria	0.00	Quicksort	23.9933	1
	Binaria	0.00	Inserción	2876.8942	2
	Binaria	0.00	Selección	2986.64	3
	Binaria	0.00	Burbuja	7892.9632	4

b) **Experimentación con búsqueda lineal:** se observó cómo al buscar un elemento en una lista de 10000 datos, la búsqueda lineal tomaba más tiempo a medida que el elemento tenía un índice mayor en la lista.

Dato	Algoritmo de búsqueda	Tiempo (ms)	Captura de pantalla n°
elemento cerca del principio de la lista de 10000 elementos	Lineal	0.00	5
elemento cerca del medio de la lista de 10000 elementos	Lineal	1.0018	6
elemento cerca del final de la lista de 10000 elementos	Lineal	1.0099	7

c) **Experimentación con ordenamiento con diferentes algoritmos:** A continuación se muestra los tiempos de ejecución (en milisegundos) de cuatro algoritmos de ordenamiento aplicados a una lista desordenada de 10.000 elementos:

Dato	Algoritmo de ordenamiento	Tiempo (ms)	Captura de pantalla n°
lista 10000 elementos desordenada	Quicksort	54.5449	8
	Burbuja	6881.8009	9
	Selección	2348.2361	10
	Inserción	2651.9980	11

d) **Experimentación con ordenamiento con listas vacías:** Para completar el análisis de desempeño de los algoritmos de ordenamiento, evaluamos aquí el escenario límite en el que la lista de entrada no contiene ningún elemento. Aunque en todos los casos deberían detectar rápidamente que no hay nada que ordenar, es útil medir la sobrecarga práctica de la llamada a función, los bucle o comparaciones internas. A continuación, se muestran los tiempos obtenidos (promedio de múltiples pruebas de ejecución en un entorno controlado).

Dato	Algoritmo de ordenamiento	Tiempo	Captura de pantalla n°

lista vacía	Quicksort	0.0000	12
lista vacía	Burbuja	0.0000	13
lista vacía	Selección	0.0000	14
lista vacía	Inserción	0.0000	15

A continuación se detallan algunos de las dificultades o errores que debemos corregir en la creación de nuestro código para que funcione de manera óptima y poder hacer las evaluaciones deseadas:

- uso de milisegundos en vez de milisegundos: al principio los resultados que el programa arrojaba no tenían el nivel de precisión que necesitábamos para poder ver una diferencia en la ejecución de diferentes algoritmos. Esto fue porque utilizamos segundos como medida de tiempo. Una sencilla corrección con una fórmula matemática nos permitió cambiar la unidad de medida a milisegundos.
- error en funciones como argumento de otra función: en un principio tuvimos problemas identificando porque nuestro código no respondía. Lo que sucedía era que le estábamos pidiendo a la función calcular_tiempo() que mida el tiempo de ejecución de una función que necesitaba argumentos (por ejemplo, burbuja(datos)). El problema es que calcular_tiempo estaba preparada para tomar como argumento una función sin argumentos. Lambda fue una sugerencia de la IA para solventar este problema. Siendo que no ha sido abordado aún en la materia, nos sirvió el análisis de la inteligencia artificial para entender mejor porque necesitábamos en ese lugar a una función anónima (lambda). Esta función envuelve la llamada al algoritmo de búsqueda u ordenamiento, que ya tiene sus argumentos fijados, y permite a calcular_tiempo invocarla sin pasarle nada.

---

6. Conclusiones

Nuestro trabajo mostró claramente cómo se comportan distintos algoritmos de búsqueda y ordenamiento en la práctica.

Gracias a un diseño modular, pudimos separar cada parte del programa (búsqueda, ordenamiento y medición de tiempo, menúes interactivos y funcionalidades principales), lo que hizo más fácil probar cosas de forma independiente y encontrar errores rápidamente.

Probamos diferentes escenarios como listas vacías, ordenadas y totalmente desordenadas. También hicimos pruebas con los dos tipos de búsqueda (lineal y binaria) con varios tamaños de datos (100, 1000 y 10.000). En aquellas con volúmenes de datos pequeños los resultados arrojados eran mínimos y no se podía apreciar su diferencia, así que optamos por un valor de dato más grande (10.000). Esto nos permitió ver cómo la teoría se refleja en los tiempos de ejecución reales:

- La Búsqueda Binaria: Siempre fue más rápida y mantuvo tiempo casi iguales sin importar donde estuviera el elemento.

- La Búsqueda Lineal: Tardó cada vez más a medida que el elemento seleccionado estaba más lejos, lo cual se condice con lo que dice el material, cuando explica que este tipo de búsqueda tiene complejidad O(n).

- En los ordenamientos: Los algoritmos como Quicksort fueron más rápidos con listas grandes que los algoritmos como Burbuja , Inserción y Selección. Solo en casos extremos en nuestras pruebas con Quicksort sobre listas muy grandes, ocasionalmente obtuvimos un error. Esto sucede porque, en Python, la implementación recursiva por defecto solo admite unas 1000 llamadas anidadas. Cuando la lista está muy desbalanceada (por elegir mal el pivote), Quicksort termina llamándose a sí mismo demasiadas veces seguidas. Para solucionarlo, aumentamos el límite de la recursión antes de ejecutar Quicksort. De este modo permitimos que la recursión llegue más profundo y evitemos ese error en las listas grandes.

Además, corregimos errores como medir en segundos en lugar de milisegundos o pasar mal las funciones a nuestra función para medir el tiempo, lo cual nos ayudó a pulir buenas prácticas como validar cada parte, mantener el código limpio y comprobar siempre que los resultados tengan sentido.
En resumen, esta “mini aplicación” no solo nos ayudó a entender mejor la teoría detrás de cada algoritmo, sino que también nos dio herramientas prácticas para experimentar y medir su rendimiento en distintos escenarios.

---

7. Bibliografía

- Universidad Tecnológica Nacional (UTN). 2025. Programación. Material de Cátedra de la Tecnicatura en Programación a Distancia.

8. Anexos

Enlace a vídeo de youtube.

Haga click aquí para ver el video explicativo de este trabajo.

Código Python

A continuación se adjunta el código del archivo de Python main.py:

```
import time
import random
import sys

#Aumentamos el límite de recursión para evitar RecursionError en pruebas grandes
sys.setrecursionlimit(10000)

# Algoritmos de ordenamiento
# Burbuja, Selección, Inserción, Quicksort

def burbuja(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]

def seleccion(arr):
    n = len(arr)
    for i in range(n):
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Intercambiar el elemento mínimo con el elemento actual
        arr[i], arr[min_index] = arr[min_index], arr[i]

def insercion(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

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        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)


# Algoritmos de búsqueda
## Lineal, Binaria

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

#funciones para el programa principal
# Generar una lista aleatoria de números enteros
def generate_random_list(size):
    return [random.randint(1, 1000) for _ in range(size)]
```


```
# Función para calcular el tiempo de ejecución de una función
def calcular_tiempo(funcion):
    inicio = time.time()
    funcion()
    fin = time.time()
    duracion_ms = (fin - inicio) * 1000  # Convertir a milisegundos
    return duracion_ms

# Funciones para elegir el algoritmo de ordenamiento y búsqueda
def elegir_ordenamiento(datos):
        print("¿Qué método de ordenamiento deseas usar? (ingresa el
número)")
        print("1. Burbuja")
        print("2. Selección")
        print("3. Inserción")
        print("4. Quicksort")
        input_ordenamiento = int(input("Ingrese el número del método: "))
        match input_ordenamiento:
            case 1:
                print("Usando el método de ordenamiento Burbuja")
                return calcular_tiempo(lambda: burbuja(datos)), "Burbuja"
            case 2:
                print("Usando el método de ordenamiento Selección")
                return calcular_tiempo(lambda: seleccion(datos)),
"Selección"
            case 3:
                print("Usando el método de ordenamiento Inserción")
                return calcular_tiempo(lambda: insercion(datos)),
"Inserción"
            case 4:
                print("Usando el método de ordenamiento Quicksort")
                return calcular_tiempo(lambda: quicksort(datos)),
"Quicksort"
            case _:
                print("Método no válido, se usará burbuja por defecto")

def elegir_busqueda(datos):
    print("¿Qué método de búsqueda deseas usar? (ingresa el número)")
    print("1. Lineal")
    print("2. Binaria")
    input_busqueda = int(input("Ingrese el número del método: "))
    elemento_buscado = int(input("Ingrese el elemento a buscar: "))
```

```
    match input_busqueda:
        case 1:
            print("Usando el método de búsqueda Lineal")
            return calcular_tiempo(lambda: busqueda_lineal(datos,
elemento_buscado)), "Lineal"
        case 2:
            print("Usando el método de búsqueda Binaria. Si no se hizo
antes, la lista se ordena para realizarla.")
            datos_ordenados= quicksort(datos.copy())
            return calcular_tiempo(lambda:
busqueda_binaria(datos_ordenados, elemento_buscado)), "Binaria"
        case _:
            print("Método no válido, se usará búsqueda lineal por defecto")

#programa principal

def probar_algoritmos():
    print("Bienvenido a la prueba de algoritmos")
    cantidad = int(input("Ingrese la cantidad aproximada de datos a
procesar: "))
    datos = generate_random_list(cantidad)
    print(f"Datos generados: {datos}")
    ordenamiento = input("¿Deseas ordenarlos? Y/N ").lower()
    tiempo_ordenamiento = 0
    tiempo_busqueda = 0
    tipo_busqueda = "ninguno"
    tipo_ordenamiento = "ninguno"
    if ordenamiento == 'y':
        tiempo_ordenamiento, tipo_ordenamiento = elegir_ordenamiento(datos)
    else:
        print("No se eligio ordenar los datos.")
    busqueda = input("¿Deseas probar un algoritmo de busqueda? Y/N
").lower()
    if busqueda == 'y':
        tiempo_busqueda, tipo_busqueda = elegir_busqueda(datos)
    else:
        print("No se eligio buscar un elemento en los datos.")
    print(f"Tiempo de ordenamiento: {tiempo_ordenamiento:.4f} ms, tipo:
{tipo_ordenamiento}")
    print(f"Tiempo de búsqueda: {tiempo_busqueda:.4f} ms, tipo:
{tipo_busqueda}")
```

```
#ejecutar el programa principal
probar_algoritmos()
```

A continuación se adjunta el algoritmo que utilizamos para las pruebas controladas del archivo de Python pruebas.py:

```
from app import *

#Programa modificado para probar algoritmos con listas definidas, no
aleatorias

def probar_algoritmos_prueba(datos):
    print("Bienvenido a la prueba de algoritmos")
    ordenamiento = input("¿Deseas ordenarlos? Y/N ").lower()
    tiempo_ordenamiento = 0
    tiempo_busqueda = 0
    tipo_busqueda = "ninguno"
    tipo_ordenamiento = "ninguno"
    if ordenamiento == 'y':
        tiempo_ordenamiento, tipo_ordenamiento = elegir_ordenamiento(datos)
    else:
        print("No se eligio ordenar los datos.")
    busqueda = input("¿Deseas probar un algoritmo de busqueda? Y/N
").lower()
    if busqueda == 'y':
        tiempo_busqueda, tipo_busqueda = elegir_busqueda(datos)
    else:
        print("No se eligio buscar un elemento en los datos.")
    print(f"Tiempo de ordenamiento: {tiempo_ordenamiento:.4f} ms, tipo:
{tipo_ordenamiento}")
    print(f"Tiempo de búsqueda: {tiempo_busqueda:.4f} ms, tipo:
{tipo_busqueda}")
```

Capturas de pantalla de pruebas y validación

Pruebas con búsqueda binaria con 10000 números:

- Figura n°1: búsqueda binaria con ordenamiento quicksort

```
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 4
Usando el método de ordenamiento Quicksort
¿Deseas probar un algoritmo de busqueda? Y/N y
¿Qué método de búsqueda deseas usar? (ingresa el número)
1. Lineal
2. Binaria
Ingrese el número del método: 2
Ingrese el elemento a buscar: 4767
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 23.9933 ms, tipo: Quicksort
Tiempo de búsqueda: 0.0000 segundos, tipo: Binaria
PS C:\Users\usuario\Desktop\utn\1-PROGRAMACION I\TP INTEGRADOR>
```

- Figura n°2: búsqueda binaria con ordenamiento inserción

```
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

y"
Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 3
Usando el método de ordenamiento Inserción
¿Deseas probar un algoritmo de busqueda? Y/N y
1. Lineal
2. Binaria
Ingrese el número del método: 2
Ingrese el elemento a buscar: 4767
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 2876.8942 ms, tipo: Inserción
Tiempo de búsqueda: 0.0000 segundos, tipo: Binaria
PS C:\Users\usuario\Desktop\utn\1-PROGRAMACION I\TP INTEGRADOR> & C:/Users/usuario/AppData/Local
```

- Figura n°3: búsqueda binaria con ordenamiento selección

```
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

¿Qué método de búsqueda deseas usar? (ingresa el número)
1. Lineal
2. Binaria
Ingrese el número del método: 2
Ingrese el elemento a buscar: 4767
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 2986.64 ms, tipo: Selección
Tiempo de búsqueda: 0.0 segundos, tipo: Binaria
1. Lineal
2. Binaria
Ingrese el número del método: 2
Ingrese el elemento a buscar: 4767
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 2986.64 ms, tipo: Selección
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 2986.64 ms, tipo: Selección
Tiempo de búsqueda: 0.0 segundos, tipo: Binaria
```

- Figura 4: búsqueda binaria con ordenamiento burbuja

```
pruebas.py > ...
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 1
Usando el método de ordenamiento Burbuja
¿Deseas probar un algoritmo de busqueda? Y/N y
¿Qué método de búsqueda deseas usar? (ingresa el número)
1. Lineal
2. Binaria
Ingrese el número del método: 2
Ingrese el elemento a buscar: 4757
Usando el método de búsqueda Binaria. Si no se hizo antes, la lista se ordena para realizarla.
Tiempo de ordenamiento: 7892.9632 ms, tipo: Burbuja
Tiempo de búsqueda: 0.0000 segundos, tipo: Binaria
```

Experimentación con búsqueda lineal

- Figura 5: búsqueda lineal sin ordenar de elemento al principio de la lista

```
Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N n
No se eligio ordenar los datos.
¿Deseas probar un algoritmo de busqueda? Y/N y
2. Binaria
Ingrese el número del método: 1
Ingrese el elemento a buscar: 6875
Usando el método de búsqueda Lineal
Tiempo de ordenamiento: 0.00 ms, tipo: ninguno
Tiempo de búsqueda: 0.0 segundos, tipo: Lineal
```

- Figura 6: búsqueda lineal sin ordenar de elemento en el medio de la lista

```
Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N n
No se eligio ordenar los datos.
¿Deseas probar un algoritmo de busqueda? Y/N y
2. Binaria
Ingrese el número del método: 1
Ingrese el elemento a buscar: 3036
Usando el método de búsqueda Lineal
Tiempo de ordenamiento: 0.00 ms, tipo: ninguno
Tiempo de búsqueda: 1.0018348693847656 segundos, tipo: Lineal
```

- Figura 7: búsqueda lineal sin ordenar de elemento al final de la lista

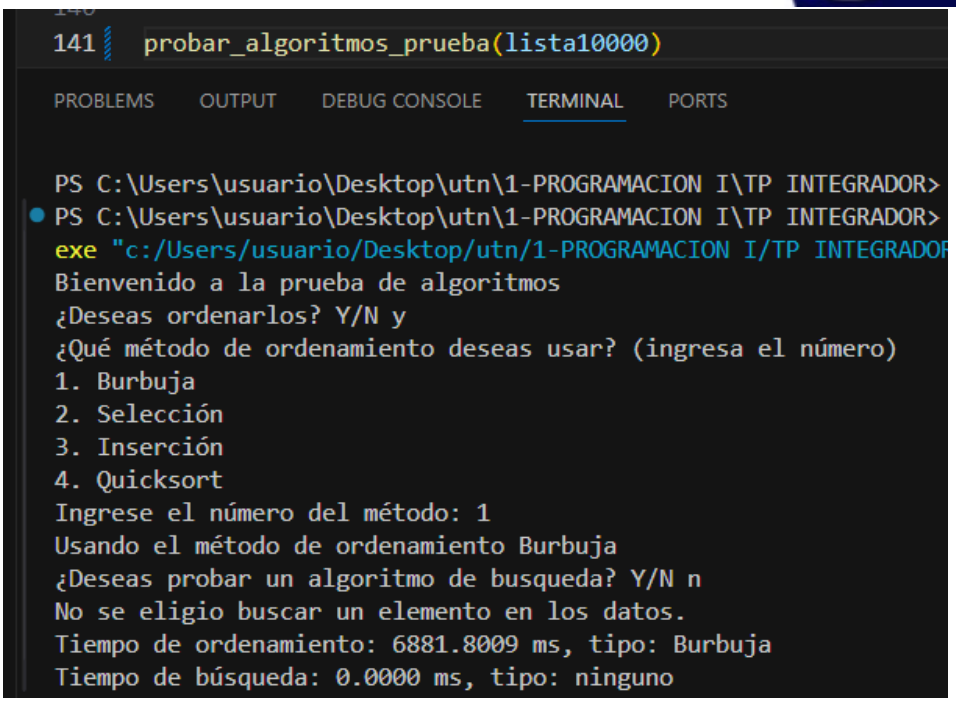
```
Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N n
No se eligio ordenar los datos.
¿Deseas probar un algoritmo de busqueda? Y/N y
2. Binaria
Ingrese el número del método: 1
Ingrese el elemento a buscar: 4767
Usando el método de búsqueda Lineal
Tiempo de ordenamiento: 0.00 ms, tipo: ninguno
Tiempo de búsqueda: 1.0099411010742188 segundos, tipo: Lineal
```

Experimentación con ordenamiento con diferentes algoritmos

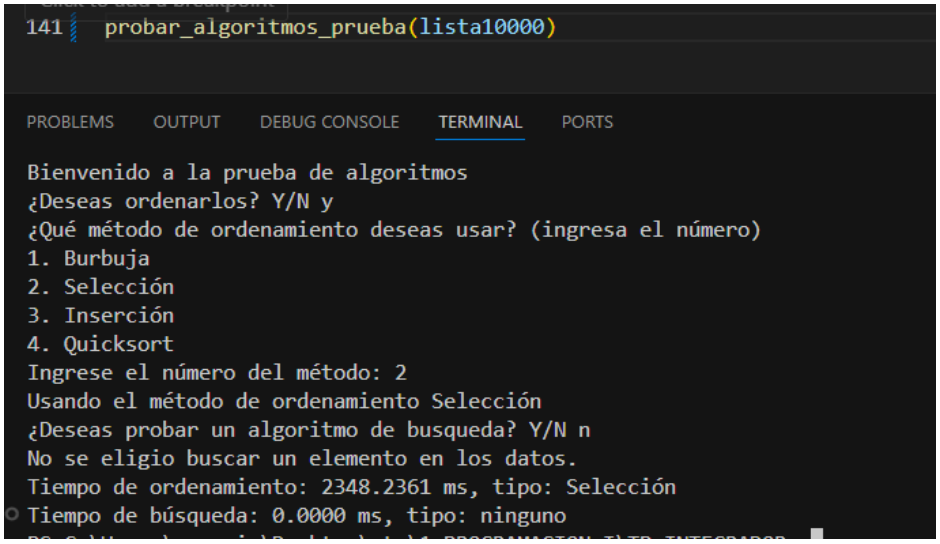
- Figura 8: ordenamiento de 10000 elementos con Quicksort:

```
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

exe "c:/Users/usuario/Desktop/utn/1-PROGRAMACION I/TP INTEGRADOR/pruebas.py"
Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 4
Usando el método de ordenamiento Quicksort
¿Deseas probar un algoritmo de busqueda? Y/N n
No se eligio buscar un elemento en los datos.
Tiempo de ordenamiento: 54.5449 ms, tipo: Quicksort
Tiempo de búsqueda: 0.0000 ms, tipo: ninguno
PS C:\Users\usuario\Desktop\utn\1-PROGRAMACION I\TP INTEGRADOR>
```

- Figura 9: ordenamiento de 10000 elementos con Burbuja:



- Figura 10: ordenamiento de 10000 elementos con selección:



- Figura 11: ordenamiento de 10000 elementos con Inserción:

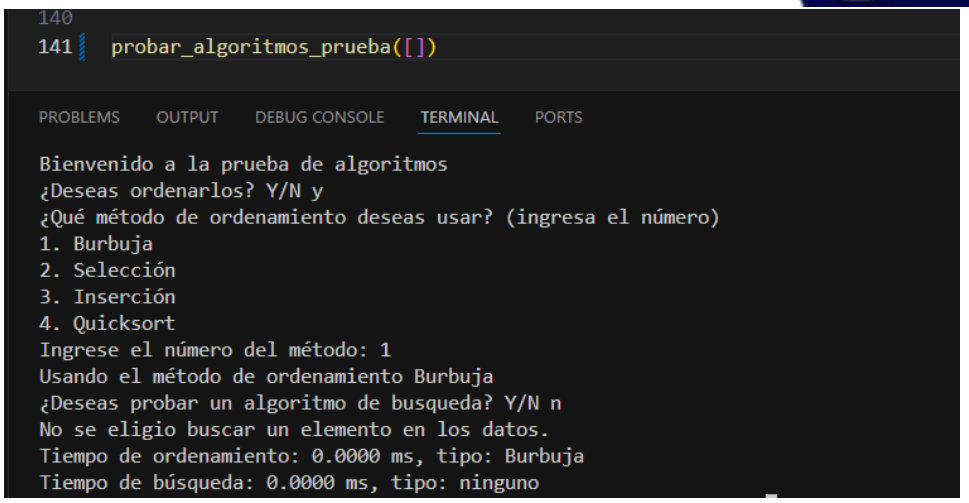
```
141  probar_algoritmos_prueba(lista10000)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 3
Usando el método de ordenamiento Inserción
¿Deseas probar un algoritmo de busqueda? Y/N n
No se eligio buscar un elemento en los datos.
Tiempo de ordenamiento: 2651.9980 ms, tipo: Inserción
Tiempo de búsqueda: 0.0000 ms, tipo: ninguno
```

Experimentación con ordenamiento con listas vacías

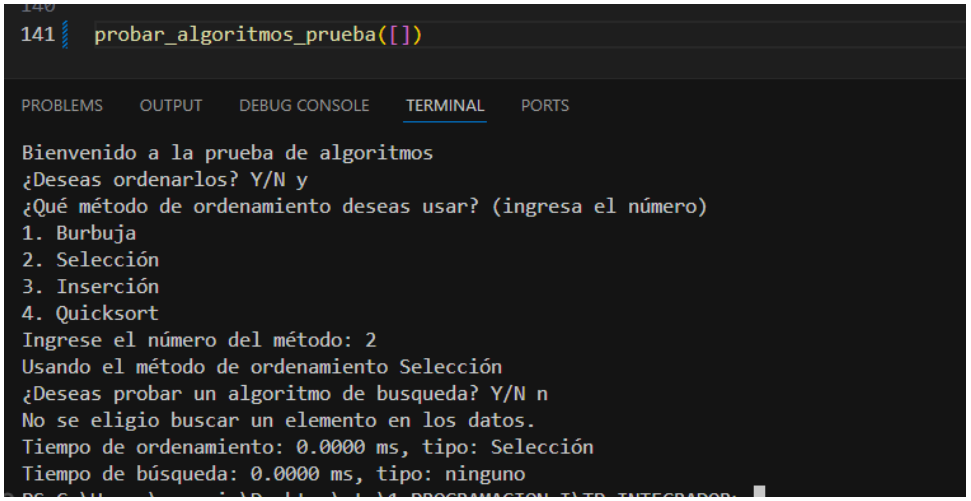
- Figura 12: ordenamiento en listas vacías con Quicksort

```
141  probar_algoritmos_prueba([])

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Bienvenido a la prueba de algoritmos
¿Deseas ordenarlos? Y/N y
¿Qué método de ordenamiento deseas usar? (ingresa el número)
1. Burbuja
2. Selección
3. Inserción
4. Quicksort
Ingrese el número del método: 4
Usando el método de ordenamiento Quicksort
¿Deseas probar un algoritmo de busqueda? Y/N n
No se eligio buscar un elemento en los datos.
Tiempo de ordenamiento: 0.0000 ms, tipo: Quicksort
Tiempo de búsqueda: 0.0000 ms, tipo: ninguno
```

- Figura 13: ordenamiento en listas vacías con Burbuja



- Figura 14: ordenamiento en listas vacías con Selección



- Figura 15: ordenamiento en listas vacías con Inserción