Taller-6: Algoritmos de Búsqueda y Ordenamiento

*Javier Eduardo Barreto Rojas Cod: 506231722 Estudiante: Nicolas Saavedra Arciniegas Cod: 506221076

I. Objetivos del taller

- Comprender y diferenciar los métodos de búsqueda binaria y búsqueda lineal.
- Implementar y analizar el desempeño de los algoritmos de ordenamiento Quicksort y Mergesort.
- Comparar la eficiencia de estos métodos con algoritmos de ordenamiento más simples como el Bubble Sort. Realizar un profiling de los algoritmos para entender mejor su comportamiento en términos de tiempo y eficiencia de memoria.
- Aplicar los principios de resolución de problemas de dividir el problema, resolver partes individuales y unir los resultados, con especial énfasis en la recursividad.
- Generar una tabla comparativa de la complejidad algorítmica de cada método de búsqueda y ordenamiento utilizado en el taller.

Parte 2: Búsqueda Lineal vs. Búsqueda Binaria

II. Actividad 1: Implementación de Búsqueda Lineal.

Descripción Teórica: Una búsqueda lineal es un método para encontrar un valor objetivo dentro de una lista. Ésta comprueba secuencialmente cada elemento de la lista para el valor objetivo hasta que es encontrado o hasta que todos los elementos hayan sido comparados.

Implementación: Código escrito en lenguaje Python que implemente la búsqueda lineal para llevar a cabo el ejercicio.

```python
import numpy as np

random_array = np.random.randint(low=0, high=5000, size=3000)
random_number = np.random.randint(0, 5000)

flag = 0
step = 0


for i in range(len(random_array)):
    step += 1
    if random_number == i:
        flag = 1
        print("Se encontró el número", random_number, "en la posición", random_array[i], "en", step, "iteraciones.")


if not flag:
    print("No se encontró el número", random_number)
```

Fig. 1. Codigo creado para el ejercicio de implementación

Prueba: Se Utiliza dentro de la busqueda Lineal esta función para buscar elementos en una lista no ordenada de números (númeroa aleatorios) y se registra el número de comparaciones realizadas por medio de capturas.

```
Se encontró el número 267 en la posición 2732 en 268 iteraciones.


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 2. Prueba de escritorio 1

```
Se encontró el número 920 en la posición 1147 en 921 iteraciones.


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 3. Prueba de escritorio 2

```
No se encontró el número 4405


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 4. Prueba donde no se encontró número

III. Actividad 2: Implementación de Búsqueda Binaria

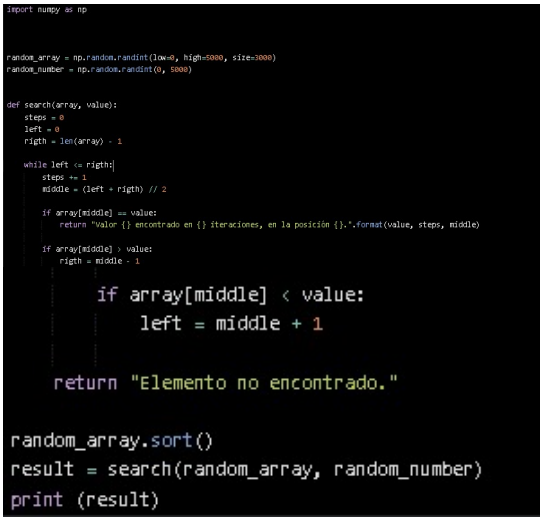
Descripción Teórica: Es un algoritmo que se emplea para localizar un elemento dentro de una lista o array, ordenada en el que se empieza tomando el valor del punto medio de la misma, comparándolo con el valor buscado, de modo que si ambos no coinciden se determina en cuál de las dos mitades puede encontrarse el valor, desechándose aquella en la que no puede estar y repitiendo el proceso con la otra, hasta dar con dicho valor o concluir que este no se encuentra en la lista. Es importante que este ordenado la lista o array, dado que entonces la búsqueda no sería efectiva, ya que se siempre se compara la cabeza de la mitad para determinar si podría estar o no en esa parte, si estuviera desordenado no podría hacerse correctamente.

Implementación: Código escrito en lenguaje Python que implemente la búsqueda binaria para llevar a cabo el ejercicio.

Prueba: Se Utiliza dentro de la busqueda binaria esta función para buscar elementos en una lista no ordenada de números (númeroa aleatorios) y se registra el número de comparaciones realizadas por medio de capturas.

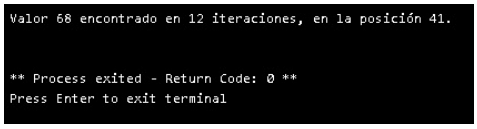
IV. Preguntas de Comparación:

¿En qué casos es más eficiente cada tipo de búsqueda?.



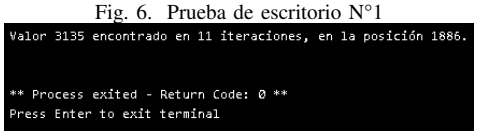
```
import numpy as np


random_array = np.random.randint(low=0, high=5000, size=3000)
random_number = np.random.randint(0, 5000)


def search(array, value):
    steps = 0
    left = 0
    rigth = len(array) - 1

    while left <= rigth:
        steps += 1
        middle = (left + rigth) // 2

        if array[middle] == value:
            return "Valor {} encontrado en {} iteraciones, en la posición {}.".format(value, steps, middle)

        if array[middle] > value:
            rigth = middle - 1

            if array[middle] < value:
                left = middle + 1


    return "Elemento no encontrado."

random_array.sort()
result = search(random_array, random_number)
print (result)
```

Fig. 5. Codigo creado para el ejercicio de implementación



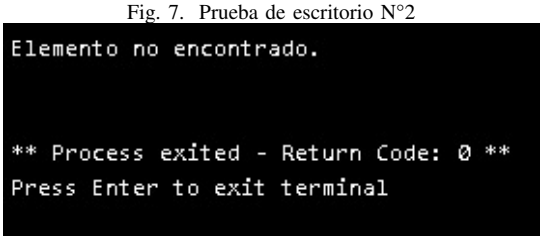
```
Valor 68 encontrado en 12 iteraciones, en la posición 41.


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 6. Prueba de escritorio N°1



```
Valor 3135 encontrado en 11 iteraciones, en la posición 1886.


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 7. Prueba de escritorio N°2



```
Elemento no encontrado.


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Fig. 8. Prueba donde no se encontró número

Búsqueda Lineal:
La búsqueda lineal es más eficiente cuando los datos no están ordenados o cuando se necesita encontrar múltiples valores en una lista desordenada.

Es útil cuando no se conoce la posición del elemento que se busca y se necesita recorrer toda la lista.

En el peor caso, la búsqueda lineal puede requerir revisar todos los elementos de la lista, lo que la hace menos eficiente para listas grandes o cuando se busca un único elemento en una lista ordenada.

Búsqueda Binaria:
La búsqueda binaria es más eficiente en listas ordenadas, ya que reduce significativamente el número de comparaciones necesarias para encontrar un elemento.

Es ideal para buscar un único elemento en una lista ordenada, ya que divide la lista en mitades sucesivas hasta encontrar el valor deseado.

En el peor caso, la búsqueda binaria tiene una complejidad de O(log n), lo que la hace altamente eficiente para listas grandes y ordenadas.

¿Cómo afecta el tamaño de la lista al rendimiento de cada algoritmo de búsqueda?.

El tamaño de la lista afecta cómo se realiza la búsqueda porque en diveros tipos de algoritmos se cuenta con maneras distintas de buscar en ella.

Por ejemplo, en caso de tener una lista pequeña, se realiza un recorrido cosa a cosa, asi hasta encontrar el elemento buscado, lo cual es lo que hace la Búsqueda Lineal. Por otro lado, en caso de tener una lista muy grande, hacer esto sería muy lento. En dicho caso, usar la Búsqueda Binaria, que divide la lista a la mitad y busca en la mitad correcta, es mucho más rápido y efectivo porque reduce el número de cosas por las cuales se deberia recorrer.

En conclusion, la búsqueda lineal se vuelve más lenta a medida que la lista crece, mientras que la búsqueda binaria sigue siendo rápida incluso con listas enormes. Asi que realmente depende mas que todo del tipo de operacion de busqueda que se desee realizar.

***Parte 2: Comparación de Algoritmos de Ordenamiento.**

V. Actividad 3: Implementación de Quicksort y Mergesort.

Descripción Teórica: Explicar cómo funcionan el Quicksort y el Mergesort, enfocándose en cómo estos algoritmos aplican los principios de dividir el problema, resolver de manera recursiva, y unir los resultados.

Quicksort:
Quicksort es un algoritmo de división y conquista que utiliza un elemento de pivote para particionar los datos de entrada en dos sub-arreglos, uno con elementos menores que el pivote y el otro con elementos mayores que el pivote. El elemento de pivote se coloca entonces en su posición final.

Quicksort:
Dividir el problema:

- Quicksort selecciona un elemento de la lista, llamado pivote, y particiona la lista en dos subconjuntos: uno con elementos menores que el pivote y otro con elementos mayores que el pivote.

Resolver de manera recursiva:

- Los dos subconjuntos se ordenan recursivamente utilizando el mismo proceso de división y partición.
- Esto se hace hasta que los subconjuntos sean lo suficientemente pequeños para ser ordenados eficientemente por otro método, como la inserción directa.

Unir los resultados:

- Como Quicksort ordena los subconjuntos in situ (en la misma lista), no es necesario unir los resultados explícitamente.
- Una vez que todos los subconjuntos se ordenan, la lista original estará ordenada.

Mergesort:
Mergesort también es un algoritmo de división y conquista que funciona dividiendo el arreglo de entrada en dos mitades, ordenándolas y luego fusionándolas de nuevo. El proceso de fusión es donde el algoritmo obtiene su nombre, ya que fusiona dos sub-arreglos ordenados en un solo sub-arreglo ordenado.

Mergesort:
Dividir el problema:

- Mergesort divide la lista en dos mitades de tamaño aproximadamente igual.

Resolver de manera recursiva:

- Cada mitad se ordena recursivamente utilizando el mismo algoritmo (Mergesort).
- Esto se repite hasta que las sublistas sean lo suficientemente pequeñas y estén ordenadas por completo.

Unir los resultados:

- Después de que las sublistas estén ordenadas, se fusionan (merge) para crear una lista ordenada más grande.
- La fusión se realiza comparando los elementos en las sublistas ordenadas y colocándolos en orden en una nueva lista.

Comparación de los Principios:
Dividir el problema:

- Ambos algoritmos dividen la lista en partes más pequeñas, lo que simplifica el problema y permite abordarlo de manera más efectiva.

Resolver de manera recursiva:

- Ambos utilizan la recursión para ordenar las sublistas más pequeñas.
- La recursión se aplica hasta que las sublistas sean lo suficientemente pequeñas como para ser ordenadas de manera eficiente.

Unir los resultados:

- Quicksort no requiere una etapa explícita de fusión, ya que los subconjuntos se ordenan in situ.
- Mergesort, por otro lado, fusiona las sublistas ordenadas para reconstruir la lista ordenada final.

Implementación: Escribe funciones en Python para cada algoritmo. Asegúrate de comentar el código para indicar las partes clave de cada algoritmo.

```
do

  swapped = false

  for i = 1 to indexOfLastUnsortedElement-1

    if leftElement > rightElement

      swap(leftElement, rightElement)

      swapped = true; ++swapCounter

while swapped
```

Fig. 9. Visu Algo: Implementación de Quicksor.

```
split each element into partitions of size 1

recursively merge adjacent partitions

  for i = leftPartIdx to rightPartIdx

    if leftPartHeadValue <= rightPartHeadValue

      copy leftPartHeadValue

    else: copy rightPartHeadValue; Increase InvIdx

copy elements back to original array
```

Fig. 10. Visu Algo: Implementación de Mergesort.

```
# Para cada particion no ordenada
for partition in partitions:

    # Selecciona el primer elemento como pivote
    pivot = partition[0]

    # Inicializa el índice de almacenamiento
    store_index = pivot_index + 1

    # Itera sobre los elementos desde el pivote hasta el último
    for i in range(pivot_index + 1, rightmost_index + 1):

        # Compara el elemento actual con el pivote
        if a[i] < a[pivot] or (a[i] == a[pivot] and random.randint(0, 1) == 0):

            # Intercambia el elemento actual con el índice de almacenamiento y avanza el
                índice de almacenamiento
            a[i], a[store_index] = a[store_index], a[i]
            store_index += 1

    # Intercambia el pivote con el elemento en store_index - 1
    a[pivot], a[store_index - 1] = a[store_index - 1], a[pivot]
```

Fig. 11. Implementación de Quicksor.

```
# Divide cada elemento en particiones de tamaño 1
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_partition = merge_sort(arr[:mid])
    right_partition = merge_sort(arr[mid:])

    # Fusiona recursivamente particiones adyacentes
    return merge(left_partition, right_partition)

# Fusiona particiones adyacentes
def merge(left_partition, right_partition):
    merged_arr = []
    left_idx, right_idx = 0, 0

    while left_idx < len(left_partition) and right_idx < len(right_partition):
        if left_partition[left_idx] <= right_partition[right_idx]:
            merged_arr.append(left_partition[left_idx])
            left_idx += 1
        else:
            merged_arr.append(right_partition[right_idx])
            right_idx += 1

    merged_arr.extend(left_partition[left_idx:])
    merged_arr.extend(right_partition[right_idx:])
    return merged_arr
```

Fig. 12. Implementación de Mergesort.

VI. Actividad 4: Implementación de Bubble Sort

Descripción Teórica: Descripción del Bubble Sort y por qué generalmente es menos eficiente.

Bubble Sort es un algoritmo de ordenación simple que funciona comparando cada elemento de la lista con su vecino adyacente y realizando intercambios si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista esté completamente ordenada. A continuación, proporciono una descripción teórica del Bubble Sort y explico por qué generalmente es menos eficiente que otros algoritmos de ordenación.

```
do

  swapped = false

  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true; ++swapCounter

while swapped
```

Fig. 13. Visu Algo: Implementación de Bubble Sort.

```python
def bubble_sort(arr):
    n = len(arr)  # Longitud del arreglo
    swap_counter = 0  # Contador de intercambios
    swapped = True  # Inicializa swapped como True para entrar en el bucle

    while swapped:  # Ejecuta mientras swapped sea True
        swapped = False  # Reinicia swapped al inicio de cada pasada
        for i in range(1, n):  # Itera desde el segundo elemento hasta el último no ordenado
            if arr[i - 1] > arr[i]:  # Compara elementos adyacentes
                arr[i - 1], arr[i] = arr[i], arr[i - 1]  # Intercambia si están en el orden incorrecto
                swapped = True  # Marca que se ha realizado un intercambio
                swap_counter += 1  # Incrementa el contador de intercambios
        n -= 1  # Reduce el rango de elementos a verificar

    return arr, swap_counter  # Devuelve el arreglo ordenado y el número de intercambios

# Ejemplo de uso del código
arr = [64, 34, 25, 12, 22, 11, 90]  # Define un arreglo desordenado
sorted_arr, swap_counter = bubble_sort(arr)  # Llama a la función bubble_sort
print("Sorted array:", sorted_arr)  # Imprime el arreglo ordenado
print("Number of swaps:", swap_counter)  # Imprime el número de intercambios realizados
```

Fig. 14. Code: Implementación de Bubble Sort.

La razón principal por la que el Bubble Sort es generalmente menos eficiente en comparación con otros algoritmos de ordenamiento es su complejidad de tiempo, que es de O(n2).

Implementación: Escribe una función para Bubble Sort.

VII. Actividad 5:

Comparación de Rendimiento y Profiling
Creación de Lista para Pruebas: Genera una lista de 2000 datos aleatorios para las pruebas de ordenamiento.

Prueba: Ordena esta lista utilizando los tres algoritmos.

Registro de Tiempos y Profiling: Utiliza la biblioteca time de Python para medir el tiempo de ejecución y herramientas como cProfile para realizar un análisis de perfil del rendimiento de cada algoritmo.

VIII. Preguntas de Reflexión

¿Cómo varía el rendimiento de cada algoritmo en función del tipo de datos de entrada?.

El rendimiento de cada algoritmo varía significativamente en función del tipo de datos de entrada y su tamaño. La elección del algoritmo adecuado depende de la estructura de los datos y de las operaciones que se necesitan realizar sobre ellos.

Búsqueda Lineal:

La búsqueda lineal es un algoritmo simple que busca un elemento en una lista revisando cada elemento uno a uno,

```python
import time
import numpy as np

random_array = np.random.randint(low=0, high=5000, size=2000)
random_number = np.random.randint(0, 5000)

inicio = time.time()
step = 0
flag = 0
for i in range(len(random_array)):
    step += 1
    if random_number == i:
        fin = time.time()
        flag = 1
        print("Lineal: Se encontró el número", random_number, "en la posición", random_array[i], "en", step, "iteraciones.", fin-inicio)
if not flag:
    fin = time.time()
    print("Lineal: Elemento no encontrado.", fin-inicio)

def search(array, value):
    steps = 0
    left = 0
    rigth = len(array) - 1

    while left <= rigth:
        steps += 1
        middle = (left + rigth) // 2
        if array[middle] == value:
            return "Binario: Valor {} encontrado en {} iteraciones, en la posición {}.".format(value, steps, middle)
        if array[middle] > value:
            rigth = middle - 1
        if array[middle] < value:
            left = middle + 1
    return "Binario: Elemento no encontrado."

random_array.sort()
inicio1 = time.time()
result = search(random_array, random_number)
fin1 = time.time()
print (result, fin1-inicio1)
```

Fig. 15. Código: Rendimiento de Tiempos y Profiling.

desde el principio hasta el final. Su complejidad es O(n), lo que significa que el tiempo de ejecución crece linealmente con el tamaño de la lista. Este algoritmo es adecuado para listas pequeñas o cuando no se conoce la ordenación de la lista.

Búsqueda Binaria:

La búsqueda binaria, por otro lado, es un algoritmo más eficiente que busca un elemento en una lista ordenada. Divide la lista en dos partes y compara el elemento buscado con el elemento medio. Si el elemento buscado es menor que el medio, busca en la parte izquierda; si es mayor, busca en la parte derecha. Su complejidad es O(log n), lo que significa que el tiempo de ejecución crece logarítmicamente con el tamaño de la lista. Este algoritmo es adecuado para listas grandes y ordenadas.

¿Por qué algunos algoritmos funcionan mejor en ciertos tipos de listas que en otros?.

Algunos algoritmos funcionan mejor que otros dependiendo del tipo de lista que se esté usando. Esto se debe a que cada algoritmo tiene su propia forma de trabajar con los datos.

Por ejemplo, hay algoritmos que son buenos para ordenar listas que ya están casi ordenadas, pero no funcionan tan bien con listas muy desordenadas. Otros algoritmos pueden ser mejores para listas grandes que están muy revueltas.

También importa si la lista es de un tipo que permite hacer cambios fácilmente, como agregar o quitar elementos en cualquier parte. Algunos algoritmos funcionan mejor con

```python
import random
# Generate a list of 2000 random numbers
data = random.sample(range(1, 10000), 2000)

```

Fig. 16. Code: Random List (2000 Datos Aleatorios)

```python
# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

```

Fig. 17. Code: Ordenamiento Quick Sort

```python
# Merge Sort
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    return merge(merge_sort(left), merge_sort(right))

def merge(left, right):
    merged = []
    left_index = 0
    right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
    merged += left[left_index:]
    merged += right[right_index:]
    return merged
```

Fig. 18. Code: Ordenamiento Merge Sort

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

```

Fig. 19. Code: Ordenamiento Bubble Sort

```python
# Sort the data using Bubble Sort
print("Bubble Sort:")
bubble_sort(data)
print(data)

# Sort the data using Quick Sort
print("Quick Sort:")
data = random.sample(range(1, 10000), 2000)
quick_sort(data)
print(data)

# Sort the data using Merge Sort
print("Merge Sort:")
data = random.sample(range(1, 10000), 2000)
merge_sort(data)
print(data)
```

Fig. 20. Code: Ordenamiento Print Result

listas que permiten hacer estos cambios de forma sencilla.

IX. Parte 3: Tabla de Complejidad Algorítmica

Generación de la Tabla: Crea una tabla que liste todos los algoritmos utilizados en el taller, tanto de búsqueda como de ordenamiento, indicando su complejidad algorítmica en el mejor, promedio y peor caso. La tabla deberá estar organizada del más eficiente al menos eficiente según la complejidad en el caso promedio.

Complejidad Algortimica: O(n2)A simple vista parece algo muy simple, pero a medida que un programa crece, se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la

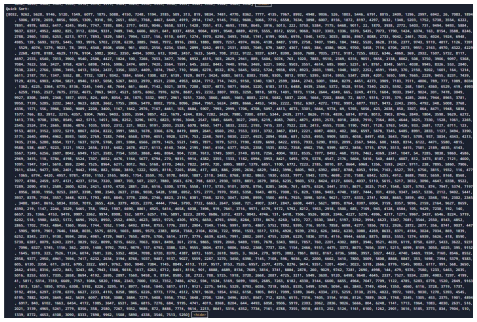


Fig. 21. Code: List Result Quick Sort

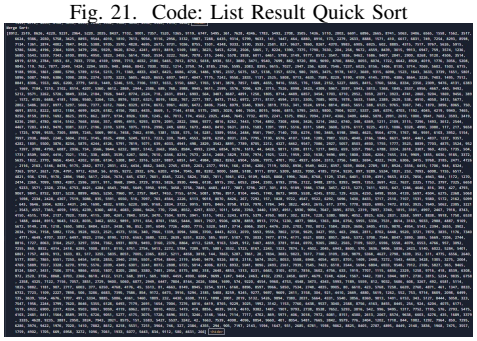


Fig. 22. Code: List Result Merge Sort

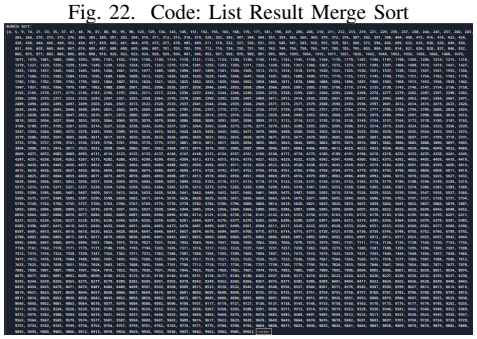


Fig. 23. Code: List Result Bubble Sort

```python
import time
import numpy as np

random_array = np.random.randint(low=0, high=5000, size=30000)
random_number = np.random.randint(0, 5000)

```

Fig. 24. Código: Registro de Tiempos y Profiling Base

```
Lineal: Se encontró el número 1997 en la posición 1073 en 1998 iteraciones. 0
    .0003223419189453125
Binario: Valor 1997 encontrado en 12 iteraciones, en la posición 11972. 1
    .5020370483398438e-05

=== Code Execution Successful ===

Lineal: Se encontró el número 1602 en la posición 1969 en 1603 iteraciones. 0
    .0001986026763916016
Binario: Valor 1602 encontrado en 13 iteraciones, en la posición 9764. 1
    .33514404296875e-05

=== Code Execution Successful ===

Lineal: Se encontró el número 2227 en la posición 147 en 2228 iteraciones. 0
    .0002586841583251953
Binario: Valor 2227 encontrado en 11 iteraciones, en la posición 13401. 1
    .3828277587890625e-05

=== Code Execution Successful ===

Lineal: Se encontró el número 2433 en la posición 869 en 2434 iteraciones. 0
    .0003528594970703125
Binario: Valor 2433 encontrado en 11 iteraciones, en la posición 14573. 2
    .5033950805664062e-05

=== Code Execution Successful ===
```

Fig. 25. Código: Registro de Tiempos y Profiling Pruebas.

Algoritmos Utilizados	Complejidad Algoritmica O(n2)		
	Menor	Promedio	Mejor
Busqueda Lineal	O(n)	O(n)	O(1)
Busqueda Binaria	O(log n)	O(log n)	O(1)
Quicksort	O(n^2)	O(n log n)	O(n log n)
Mergesort	O(n log n)	O(n log n)	O(n log n)
Bubble Sort	O(n^2)	O(n^2)	O(n)

Fig. 26. Tabla Algoritmo Búsqueda De Ordenamiento.

eficiencia teórica del programa, al estudio de estos casos se denomina Complejidad Algorítmica.

La búsqueda binaria: En este caso es más eficiente que la búsqueda lineal, ya que su complejidad en el caso promedio es O(log n), mientras que la búsqueda lineal tiene una complejidad de O(n).

La búsqueda binaria: Requiere que el arreglo esté ordenado previamente, lo que puede agregar un costo adicional de O(n log n) si se utiliza un algoritmo de ordenamiento como quicksort o mergesort.
Sin embargo, si el arreglo ya está ordenado, la búsqueda binaria es claramente más eficiente que la búsqueda lineal.

Quick Sort También tiene una complejidad algorítmica de O(n log n) en el caso promedio, pero en el peor caso puede llegar a O(n2) si los datos están ordenados de forma creciente o decreciente.

Merge Sort Tiene una complejidad algorítmica de O(nlogn) en todos los casos, lo que significa que el tiempo de ejecución no depende del orden en que se encuentren los datos.

Bubble Sort Es el algoritmo de ordenamiento menos eficiente de los tres, con una complejidad algorítmica de O(n2) en todos los casos. Aunque es fácil de implementar, no es práctico para ordenar grandes conjuntos de datos.

REFERENCES

[1] Taller-6, GitLab. [En línea]. Disponible en: https://gitlab.com/konrad_lorenz/data-structures-2024-1/taller_6. [Consultado: 18-may-2024].

REFERENCES

[1] Visualising data structures and algorithms through animation - VisuAlgo, Visualgo.net. [En línea]. Disponible en: https://visualgo.net/en. [Consultado: 18-may-2024].