UNIVERSIDAD TECNOLÓGICA NACIONAL
Facultad Regional Avellaneda
Tecnicatura Universitaria en Programación
Modalidad a Distancia – Ciclo Lectivo 2025

TRABAJO INTEGRADOR FINAL SOBRE ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO EN PYTHON

Alumnos: Matías D'Agostino, Agustina Milagros Cruz.

Correos: matias.dagostino@tupad.utn.edu.ar, agustina.cruz@tupad.utn.edu.ar.

Materia: Programación I

Tutor: Luciano Chiroli.

Profesor: Ariel Enferrel.

UNIVERSIDAD TECNOLÓGICA NACIONAL
Facultad Regional Avellaneda
Tecnicatura Universitaria en Programación
Modalidad a Distancia – Ciclo Lectivo 2025

Fecha de entrega: 9/6/2025

Índice

1. Introducción

Este trabajo se centra en el estudio de algoritmos de búsqueda y ordenamiento, fundamentales en programación para organizar y acceder eficientemente a datos. Se eligió este tema debido a su aplicabilidad práctica en sistemas reales y su importancia en estructuras de datos y rendimiento computacional. El objetivo principal es comprender, implementar y comparar diferentes algoritmos de búsqueda en Python, como la búsqueda binaria (iterativa y recursiva) y la búsqueda lineal, además del ordenamiento necesario para optimizar estas búsquedas.

2. Marco Teórico

Los algoritmos de ordenamiento y búsqueda son fundamentales en la programación, ya que permiten organizar datos y acceder a ellos de forma eficiente. Su correcta elección impacta directamente en el rendimiento de un programa, especialmente cuando se trabaja con grandes volúmenes de datos.

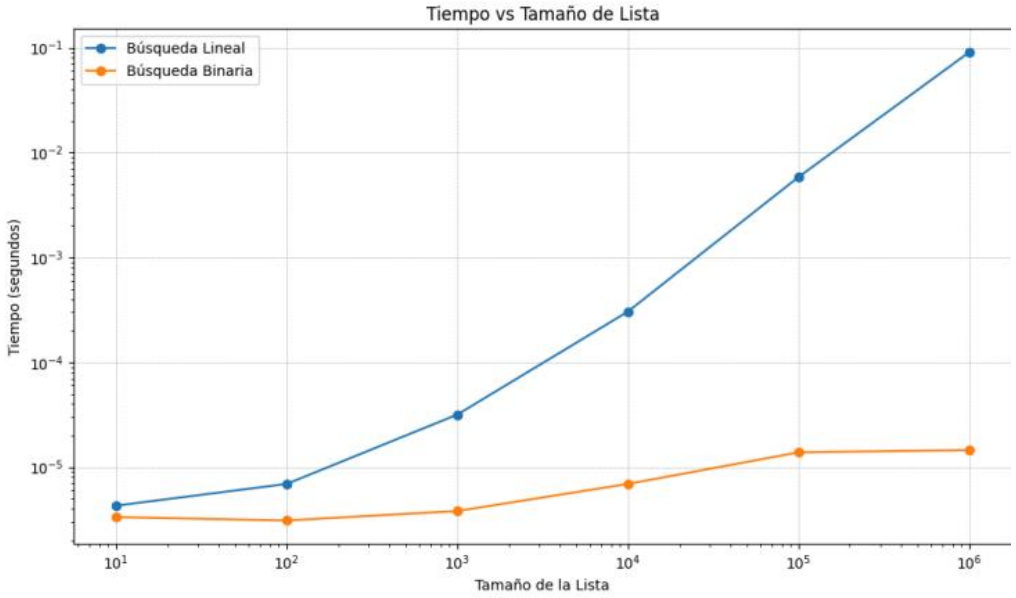
Algoritmos de ordenamiento

Un algoritmo de ordenamiento organiza una colección de elementos según un criterio, por ejemplo, de menor a mayor. Uno de los métodos más sencillos es Bubble Sort, que compara pares de elementos adyacentes e intercambia aquellos que estén desordenados. Su principal ventaja es la simplicidad, aunque su desventaja es la baja eficiencia en listas grandes, ya que tiene una complejidad temporal de $O(n^2)$.

Algoritmos de búsqueda

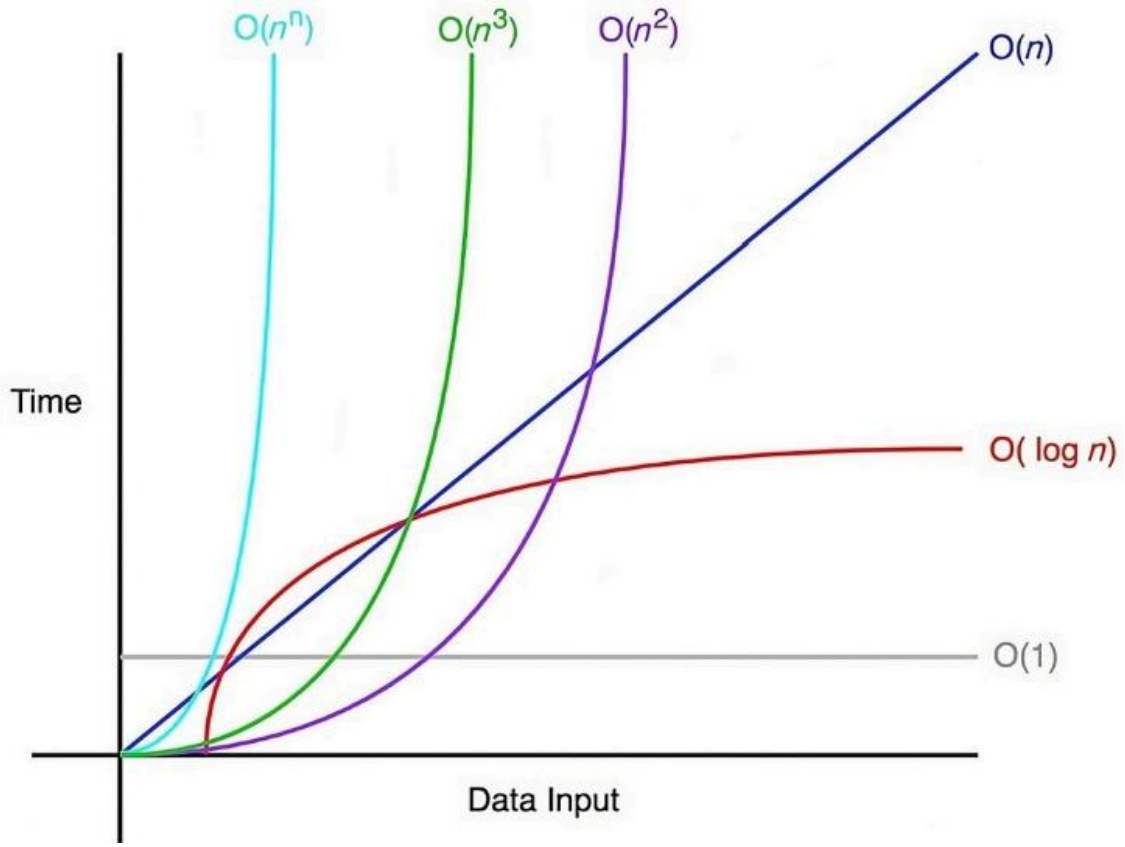
Los algoritmos de búsqueda permiten localizar un valor dentro de una estructura de datos.

- La búsqueda lineal revisa todos los elementos uno por uno hasta encontrar el deseado. Es simple y no necesita que la lista esté ordenada, pero es poco eficiente en listas largas. Su complejidad es O(n).
- La búsqueda binaria, en cambio, es mucho más rápida. Solo funciona en listas previamente ordenadas y consiste en dividir el espacio de búsqueda por la mitad en cada paso. Puede implementarse de forma iterativa o recursiva, y su complejidad es O(log n).



Comparación de complejidad:

La complejidad medida con O(n) es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada. Por ejemplo, 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.



Los algoritmos de búsqueda lineal tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado. Los algoritmos de búsqueda binaria tienen un tiempo de ejecución de O(log n), lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.

La siguiente tabla muestra el tiempo de ejecución de los algoritmos de búsqueda lineal y binaria para diferentes tamaños de lista:

Tamaño de la lista	Búsqueda lineal	Búsqueda binaria
10	10	3
100	100	7
1000	1000	10
10000	10000	13
100000	100000	16

En la tabla, podemos ver que el tiempo de ejecución de la búsqueda lineal aumenta mucho más rápidamente que el tiempo de ejecución de la búsqueda binaria a medida que aumenta el tamaño de la lista. Esto hace que la búsqueda binaria sea mucho más eficiente para listas grandes.

En general, el tamaño de la lista es un factor importante a tener en cuenta al elegir un algoritmo de búsqueda. Si la lista es pequeña, es probable que la búsqueda lineal sea más eficiente. Sin embargo, si la lista es grande, es probable que la búsqueda binaria sea más eficiente.

Algoritmo	Complejidad	¿Requiere lista ordenada?
Búsqueda lineal	$O(n)$	No
Búsqueda binaria	$O(\log n)$	Sí
Bubble Sort	$O(n^2)$	No

3. Caso Práctico

En este trabajo se desarrollaron e implementaron tres algoritmos principales: uno de ordenamiento (Bubble Sort) y dos de búsqueda (búsqueda binaria en su versión iterativa y recursiva). Todos fueron programados en Python y probados sobre listas numéricas. A continuación, se detallan cada uno de ellos.

Ordenamiento con Bubble Sort (algoritmo iterativo)

Antes de aplicar los algoritmos de búsqueda, fue necesario ordenar la lista. Para esto se utilizó el algoritmo Bubble Sort, que recorre la lista varias veces, comparando elementos adyacentes e intercambiándolos si están desordenados. Aunque no es el método más eficiente en términos de tiempo, su implementación es sencilla y adecuada para listas pequeñas.

Código en Python:

```
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista
```

Búsqueda binaria iterativa

Una vez ordenada la lista, se aplicó la **búsqueda binaria en su forma iterativa**. Esta versión del algoritmo es más eficiente en memoria y más fácil de depurar que su equivalente recursiva. Funciona dividiendo el espacio de búsqueda a la mitad en cada paso.

Código en Python:

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

Búsqueda binaria recursiva

También se implementó una versión recursiva de la búsqueda binaria. Esta forma del algoritmo resulta más elegante desde el punto de vista lógico, ya que sigue el principio de "divide y vencerás", llamándose a sí misma sobre subconjuntos más pequeños. Sin embargo, puede consumir más memoria debido al uso del stack de llamadas.

Código en Python:

```
def busqueda_binaria_recursiva(lista, objetivo, inicio=0, fin=None):
    if fin is None:
        fin = len(lista) - 1
    if inicio > fin:
        return -1
```

```
    medio = (inicio + fin) // 2
    if lista[medio] == objetivo:
        return medio
    elif lista[medio] < objetivo:
        return busqueda_binaria_recursiva(lista, objetivo, medio + 1, fin)
    else:
        return busqueda_binaria_recursiva(lista, objetivo, inicio, medio - 1)
```

Prueba de funcionamiento del programa

A continuación, se presenta un ejemplo práctico de uso de los algoritmos. Primero se ordena una lista con Bubble Sort y luego se aplica la búsqueda binaria iterativa para encontrar un valor específico.

Código en Python:

```
datos = [34, 7, 23, 32, 5, 62]
bubble_sort(datos)
print("Lista ordenada:", datos)
posicion = busqueda_binaria(datos, 23)
print("Elemento encontrado en la posición:", posicion)
```

Este programa devuelve la lista ordenada y la posición del número buscado. Se verificó que, al estar la lista ordenada previamente, la búsqueda binaria funciona de manera rápida y efectiva.

4. Metodología Utilizada

El trabajo se desarrolló en lenguaje Python 3.x, implementando diferentes versiones del algoritmo de búsqueda binaria: una iterativa, una recursiva y una versión robusta con manejo de errores. Además, se compararon estas implementaciones con la búsqueda lineal, aplicándolas sobre listas ordenadas de distintos tamaños para evaluar su comportamiento.

Durante el proceso, se realizaron pruebas exhaustivas para verificar el funcionamiento de los algoritmos en diferentes escenarios: listas vacías, valores existentes y no existentes, y elementos duplicados. Se utilizó bibliografía especializada y documentación oficial para fundamentar las decisiones técnicas.

El trabajo fue realizado en equipo, distribuyendo tareas de investigación, programación, validación y documentación. También se utilizó un repositorio en GitHub para el control de versiones y colaboración.

5. Resultados Obtenidos

Las pruebas demostraron que la búsqueda binaria fue mucho más rápida que la búsqueda lineal en listas grandes, especialmente en la versión iterativa, que resultó ser la más eficiente en cuanto al uso de memoria. Ambas versiones (iterativa y recursiva) devolvieron los mismos resultados correctamente, validando su funcionalidad.

La implementación robusta permitió identificar errores comunes, como listas vacías o no ordenadas, mejorando la fiabilidad del algoritmo. También se confirmó que, para aplicar búsqueda binaria, es fundamental que los datos estén ordenados previamente, por lo que se utilizó Bubble Sort como paso previo.

6. Conclusiones

El análisis realizado nos permitió comprender el funcionamiento, las ventajas y las limitaciones de los algoritmos de búsqueda. Se comprobó que la búsqueda binaria es altamente eficiente en listas ordenadas, y que la versión iterativa es preferible cuando se busca optimizar memoria.

Se valoró la importancia de aplicar el algoritmo adecuado según el tipo de datos y el contexto. Además, se aprendió a identificar errores comunes en la implementación y a documentar correctamente el código.

Como mejora futura, se propone utilizar algoritmos de ordenamiento más eficientes que Bubble Sort, como QuickSort o Timsort, e investigar estructuras más complejas como árboles binarios para ampliar las posibilidades de búsqueda.

7. Bibliografía

Videos:

1. Video sobre algoritmos de búsqueda y ordenamiento (YouTube).
   Disponible en:
   https://www.youtube.com/watch?v=gJlQTq80lIg&feature=youtu.be
2. Explicación de algoritmos de búsqueda y ordenamiento (YouTube).
   Disponible en: https://www.youtube.com/watch?v=xntUhrhtLaw
3. Tutorial de casos prácticos de búsqueda y ordenamiento en programación (YouTube).
   Disponible en: https://www.youtube.com/watch?v=u1QuRbx-_x4

Documentos y Materiales Escritos:

4. PDF: *Búsqueda y Ordenamiento en Programación - Casos Prácticos y Algoritmos*.
   Archivo local:
   Downloads/Búsqueda%20y%20Ordenamiento%20en%20Programación%20.pdf

5. Google Colab: Notebook con ejemplos y prácticas de algoritmos de búsqueda y ordenamiento.
   Disponible en:
   https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing

8. Anexos

- Capturas del programa funcionando.
- Código fuente en GitHub: https://github.com/codewtato/UTN-TUPaD-P1/tree/main/10%20B%C3%BAsqueda%20y%20ordenamiento
- Link del video explicativo:
https://youtu.be/BQmNQFKR354?si=iYNRxf9vIKmo5-Aq

```python
#Importe time para medir cuanto tiempo tarda cada algoritmo
import time

# ORDENAMIENTO POR BURBUJA
#Definimos el ordenamiento por burbuja que recibe lista como parametro
def bubble_sort(lista):
    #Obtenemos el tamaño de la lista
    n = len(lista)
    #Pasamos al primer bucle que controla la cantidad de pasadas
    for i in range(n):
        #Este bucle compara los elementos adyacentes desde el principio hasta la parte que ya esta ordenada
        for j in range(0, n - i - 1):
            #Formula del ordenamiento por burbuja, si el elemento es mayor al siguiente los intercambia
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                #Devuelvo lista
    return lista
```

```python
# PRUEBAS


lista = list(range(1, 10001))
objetivo = 10000

# ORDENAMIENTO
print("=== ORDENAMIENTO (Bubble Sort) ===")
#Desordeno la lista para que trabaje el ordenamiento
lista_desordenada = lista[::-1]
#Marcamos el inicio
inicio = time.perf_counter()
#Aplicamos el ordenamiento por burbuja sobre una copia de la lista
ordenada = bubble_sort(lista_desordenada.copy())
#Marcamos el final
fin = time.perf_counter()
#Mostramos por pantalla la lista ordenada
print("Lista ordenada:", ordenada)
#Muestro cuanto tardo, con 10 decimales para ver bien
print(f"Tiempo de ordenamiento: {fin - inicio:.10f} segundos\n")

# BÚSQUEDA
print("=== BÚSQUEDA (Interpolación) ===")
#Marcamos el inicio
inicio = time.perf_counter()
#Buscamos el numero en la lista ya ordenada y le pasamos un objetivo como parametro
posicion = busqueda_interpolacion(ordenada, objetivo)
#Marcamos el final
fin = time.perf_counter()
#Mostramos que numero se esta buscando
print(f"Elemento buscado: {objetivo}")
#Mostramos si se encontro
print(f"Encontrado en posición: {posicion}" if posicion != -1 else "No encontrado.")
#Mostramos cuanto tardo en hacer la busqueda
print(f"Tiempo de búsqueda: {fin - inicio:.10f} segundos")
```

```
PS C:/Users/code/Desktop/Pruebas>
Tiempo de búsqueda: 0.0000026000 segundos
Encontrado en posición: 89
Elemento buscado: 90
=== BÚSQUEDA (Interpolación) ===

Tiempo de ordenamiento: 0.0004169000 segundos
, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80
Lista ordenada: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
=== ORDENAMIENTO (Bubble Sort) ===
PS C:/Users/code/Desktop/Pruebas> & C:\Users\code\AppData\Local\Microsoft\WindowsApps\python3.11.exe c:\Users\code\Downloads\busquedayordenamiento.py
```

```python
# BÚSQUEDA POR INTERPOLACIÓN
#Defino busqueda por interpolacion y pasamos lista y objetivo como parametros
def busqueda_interpolacion(lista, objetivo):
    #Inicializo izquierda y derecha
    izquierda = 0
    derecha = len(lista) - 1
    #Verificamos si hay elementos para revisar y que el numero este dentro del rango
    while izquierda <= derecha and lista[izquierda] <= objetivo <= lista[derecha]:
        #Revisamos si queda solo un numero y comparamos si es el que buscamos
        if izquierda == derecha:
            if lista[izquierda] == objetivo:
                return izquierda
            #Devuelve -1 si no esta
            return -1
        #Formula de busqueda por interpolacion, calcula masomenos donde puede estar.
        pos = izquierda + ((objetivo - lista[izquierda]) * (derecha - izquierda) //
                           (lista[derecha] - lista[izquierda]))
        #Si justo en esa posición calculada esta el numero que pedimos como objetivo, significa que lo encontramos y nos devuelve el numero.
        if lista[pos] == objetivo:
            return pos
        #Verificamos si el numero que en pos es menor al objetivo
        elif lista[pos] < objetivo:
            izquierda = pos + 1
        #Verificamos si el numero que en pos es mayor al objetivo
        else:
            derecha = pos - 1
            #Devuelvo -1 si todo eso no pasa
    return -1
```

```
, 9637, 9638, 9639, 9640, 9641, 9642, 9643, 9644, 9645, 9646, 9647, 9648, 9649, 9650, 9651, 9652, 9653, 9654, 9655, 9656, 9657, 9658, 9659, 9660, 9661, 9662, 9663,
 9664, 9665, 9666, 9667, 9668, 9669, 9670, 9671, 9672, 9673, 9674, 9675, 9676, 9677, 9678, 9679, 9680, 9681, 9682, 9683, 9684, 9685, 9686, 9687, 9688, 9689, 9690,
9691, 9692, 9693, 9694, 9695, 9696, 9697, 9698, 9699, 9700, 9701, 9702, 9703, 9704, 9705, 9706, 9707, 9708, 9709, 9710, 9711, 9712, 9713, 9714, 9715, 9716, 9717, 9
718, 9719, 9720, 9721, 9722, 9723, 9724, 9725, 9726, 9727, 9728, 9729, 9730, 9731, 9732, 9733, 9734, 9735, 9736, 9737, 9738, 9739, 9740, 9741, 9742, 9743, 9744, 97
45, 9746, 9747, 9748, 9749, 9750, 9751, 9752, 9753, 9754, 9755, 9756, 9757, 9758, 9759, 9760, 9761, 9762, 9763, 9764, 9765, 9766, 9767, 9768, 9769, 9770, 9771, 977
2, 9773, 9774, 9775, 9776, 9777, 9778, 9779, 9780, 9781, 9782, 9783, 9784, 9785, 9786, 9787, 9788, 9789, 9790, 9791, 9792, 9793, 9794, 9795, 9796, 9797, 9798, 9799
, 9800, 9801, 9802, 9803, 9804, 9805, 9806, 9807, 9808, 9809, 9810, 9811, 9812, 9813, 9814, 9815, 9816, 9817, 9818, 9819, 9820, 9821, 9822, 9823, 9824, 9825, 9826,
 9827, 9828, 9829, 9830, 9831, 9832, 9833, 9834, 9835, 9836, 9837, 9838, 9839, 9840, 9841, 9842, 9843, 9844, 9845, 9846, 9847, 9848, 9849, 9850, 9851, 9852, 9853,
9854, 9855, 9856, 9857, 9858, 9859, 9860, 9861, 9862, 9863, 9864, 9865, 9866, 9867, 9868, 9869, 9870, 9871, 9872, 9873, 9874, 9875, 9876, 9877, 9878, 9879, 9880, 9
881, 9882, 9883, 9884, 9885, 9886, 9887, 9888, 9889, 9890, 9891, 9892, 9893, 9894, 9895, 9896, 9897, 9898, 9899, 9900, 9901, 9902, 9903, 9904, 9905, 9906, 9907, 99
9854, 9855, 9856, 9857, 9858, 9859, 9860, 9861, 9862, 9863, 9864, 9865, 9866, 9867, 9868, 9869, 9870, 9871, 9872, 9873, 9874, 9875, 9876, 9877, 9878, 9879, 9880, 9
9854, 9855, 9856, 9857, 9858, 9859, 9860, 9861, 9862, 9863, 9864, 9865, 9866, 9867, 9868, 9869, 9870, 9871, 9872, 9873, 9874, 9875, 9876, 9877, 9878, 9879, 9880, 9
9854, 9855, 9856, 9857, 9858, 9859, 9860, 9861, 9862, 9863, 9864, 9865, 9866, 9867, 9868, 9869, 9870, 9871, 9872, 9873, 9874, 9875, 9876, 9877, 9878, 9879, 9880, 9
881, 9882, 9883, 9884, 9885, 9886, 9887, 9888, 9889, 9890, 9891, 9892, 9893, 9894, 9895, 9896, 9897, 9898, 9899, 9900, 9901, 9902, 9903, 9904, 9905, 9906, 9907, 99
08, 9909, 9910, 9911, 9912, 9913, 9914, 9915, 9916, 9917, 9918, 9919, 9920, 9921, 9922, 9923, 9924, 9925, 9926, 9927, 9928, 9929, 9930, 9931, 9932, 9933, 9934, 993
5, 9936, 9937, 9938, 9939, 9940, 9941, 9942, 9943, 9944, 9945, 9946, 9947, 9948, 9949, 9950, 9951, 9952, 9953, 9954, 9955, 9956, 9957, 9958, 9959, 9960, 9961, 9962
, 9963, 9964, 9965, 9966, 9967, 9968, 9969, 9970, 9971, 9972, 9973, 9974, 9975, 9976, 9977, 9978, 9979, 9980, 9981, 9982, 9983, 9984, 9985, 9986, 9987, 9988, 9989,
 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Tiempo de ordenamiento: 3.9407446000 segundos

=== BÚSQUEDA (Interpolación) ===
Elemento buscado: 10000
Encontrado en posición: 9999
Tiempo de búsqueda: 0.0000058000 segundos
```