Trabajo Integrador : Programación I
Búsqueda y Ordenamiento

Datos Generales

Algoritmos de Búsqueda en Programación

Alumnos:

Gomez Julian - Gomezjulian1995@gmail.com

Girelli William – Girelli.william@gmail.com

Materia: Programación I

Profesor: Ariel Enferrel

Fecha de Entrega: 09/06/2025

Video: https://www.youtube.com/watch?v=wl1h2A5ydg8

Índice

1. Introducción

2. Marco Teórico

3. Caso Práctico

4. Metodología Utilizada

5. Resultados Obtenidos

6. Conclusiones

7. Bibliografía

8. Anexos

1. Introducción

En el desarrollo de soluciones computacionales, la elección adecuada de los métodos empleados es crucial para garantizar un procesamiento ágil y efectivo de la información. Entre ellos, ciertos procedimientos destinados a localizar y organizar datos resultan especialmente relevantes, ya que su implementación impacta directamente en la capacidad del sistema para manejar grandes cantidades de información.

El estudio de estas técnicas permite comprender cómo operan y cómo su diseño incide en su desempeño. Este análisis se centra en cómo el tiempo de ejecución varía según el tamaño de entrada, lo cual es clave para elegir estrategias más adecuadas según el contexto. De esta forma, se logra no solo un uso eficiente de los recursos disponibles, sino también sistemas más responsivos y confiables.

2. Marco Teórico

Complejidad Computacional

En el análisis de algoritmos, uno de los factores más relevantes es su eficiencia computacional, evaluada generalmente a través del tiempo de ejecución frente al tamaño de la entrada. Esta estimación, conocida como complejidad temporal, es esencial para seleccionar soluciones adecuadas según el contexto del problema.

Para expresarla se utiliza la notación asintótica, siendo la más común la notación **Big O** (O grande), que describe cómo se comporta el algoritmo en su caso más desfavorable. A continuación, se comparan dos algoritmos clásicos de búsqueda bajo esta perspectiva.

Búsqueda Lineal

La búsqueda lineal es uno de los métodos más sencillos para encontrar un elemento dentro de una colección. Simplemente consiste en recorrer secuencialmente cada valor desde el principio de la lista hasta hallar el elemento buscado o agotar la lista sin éxito.

Presenta una complejidad temporal de O(n) tanto en el caso promedio como en el peor escenario, donde *n* representa la cantidad de elementos. Esto implica que el tiempo de ejecución crece de forma proporcional al tamaño de los datos, lo que la vuelve poco eficiente al trabajar con listas extensas.

Entre sus principales ventajas se destacan su facilidad de implementación, el bajo consumo de recursos y su capacidad para operar sobre listas no ordenadas.

No obstante, su mayor desventaja es la ineficiencia en escenarios con grandes volúmenes de información, ya que no aprovecha ninguna estructura previa ni permite aplicar técnicas de optimización en la búsqueda. Suponiendo que la lista sea muy grande y el elemento buscado sea el último, o directamente no exista, este método de búsqueda recorrerá la lista completa hasta llegar a él, siendo ineficiente para esos casos.

Ejemplo de recorrido lineal en una lista de 1000 elementos:

Complejidad temporal: O(1000)

Mejor caso: si el elemento está en la primera posición → **1 paso**.

Peor caso: si está al final o no está → **1000 pasos**.

Promedio: (1000 / 2) ≈ **500 pasos**.

Búsqueda binaria

La búsqueda binaria es un algoritmo eficiente cuya estrategia consiste en dividir el conjunto de búsqueda por la mitad en cada paso, descartando sistemáticamente la mitad en la que no puede encontrarse el elemento buscado.

Lógica del algoritmo:

1. Al dividir la lista a la mitad, ubicamos el índice para la comparación en el elemento del medio (o el de la mitad - 1 en caso de ser una lista con longitud impar)
2. Una vez hecho esto, comparamos el valor apuntado actualmente (el del medio) con el valor buscado

   a. Si el elemento buscado es mayor al apuntado, se actualizan los índices de búsqueda para buscar en la sublista ubicada a la derecha del elemento del medio, pues la lista izquierda contiene todos elementos menores y es descartada.
   b. Si el elemento buscado es menor al apuntado, se pasa a buscar en la sublista izquierda, descartando la sublista derecha.
3. Se repite el proceso con cada sublista hasta encontrar el elemento buscado.

Como habremos notado, este algoritmo funciona bajo la condición de que la lista esté ordenada, ya que de lo contrario la búsqueda nos dará errores al momento de hacer las comparativas de valor.

Esta técnica tiene una complejidad temporal de O(log n) en el peor caso, lo que representa una mejora significativa frente a métodos con crecimiento lineal, especialmente cuando se trabaja con grandes volúmenes de información.

Entre sus principales ventajas se destacan su rapidez y la escasa cantidad de comparaciones necesarias para localizar un valor.
Sin embargo, su aplicación está condicionada al orden de los datos, lo que puede implicar un costo adicional si estos cambian con frecuencia o deben organizarse previamente para aplicar el algoritmo.
Conviene utilizarlo en listas ordenadas grandes y resulta no tan efectivo en listas más pequeñas, o en los casos donde el elemento buscado está cerca del comienzo

Ejemplo de búsqueda binaria con una lista de 1000 elementos:

Requisitos: La lista debe estar **ordenada**.

Complejidad temporal (O):
$O(\log_2 n)$
Para n = 1000 → $\log_2(1000) \approx$ **9.97**, o sea, **≈ 10 pasos** como máximo.

Número mínimo de pasos:

- **Mejor caso:** si el elemento está justo en el centro en el primer intento → **1 paso**.
- **Peor caso:** $\log_2(1000) \approx$ **10 pasos**.

Ordenamiento

El ordenamiento es el proceso de organizar una colección de datos en un orden específico, como ascendente o descendente. Esta operación es fundamental en muchas aplicaciones, ya sea para facilitar búsquedas, análisis de datos o mejorar la presentación de información.

Bubble Sort (Ordenamiento burbuja)

El ordenamiento burbuja es un algoritmo sencillo que consiste en comparar pares de elementos adyacentes e intercambiarlos si no se encuentran en el orden deseado. Este proceso se repite hasta que toda la lista esté ordenada. Su principal ventaja es su simplicidad, lo que lo convierte en una buena herramienta para introducir conceptos básicos de algoritmos. Sin embargo, su desempeño es notablemente ineficiente, con una complejidad temporal de $O(n^2)$ tanto en el peor caso como en el promedio, lo que lo hace poco adecuado para conjuntos de datos grandes.

Insertion Sort (Ordenamiento por inserción)

Este algoritmo construye una lista ordenada desde el inicio, insertando cada elemento en su posición correcta dentro de la parte ya ordenada del arreglo. Es especialmente eficiente cuando se trabaja con listas pequeñas o casi ordenadas, mostrando buen desempeño en ciertos casos específicos. Además, es un algoritmo estable, lo que significa que mantiene el orden relativo de los elementos iguales. No obstante, al igual que Bubble Sort, tiene una complejidad de $O(n^2)$ en el peor caso, lo que limita su uso en aplicaciones que manejan grandes volúmenes de información.

Selection Sort (Ordenamiento por selección)

El método de selección funciona identificando repetidamente el elemento más pequeño (o más grande) de la parte no ordenada del arreglo y colocándolo al inicio de esta sección. Se repite este proceso hasta que todos los elementos estén en su lugar correspondiente. Su implementación es sencilla y requiere una cantidad reducida de intercambios, exactamente

O(n) a lo largo de toda la ejecución. Sin embargo, su rendimiento general es deficiente, con una complejidad temporal fija en $O(n^2)$ para todos los casos, lo que lo hace poco viable en escenarios donde se exige eficiencia. Además, no es un algoritmo estable ni adaptativo.

Quicksort (Ordenamiento rápido)

Quicksort es un algoritmo recursivo basado en el paradigma de "divide y vencerás". Su funcionamiento consiste en seleccionar un elemento pivote y dividir el conjunto en dos sublistas: una con elementos menores al pivote y otra con elementos mayores. Luego aplica el mismo proceso recursivamente a cada sublista. En promedio, Quicksort ofrece un excelente rendimiento con una complejidad temporal de $O(n \log n)$, lo que lo convierte en una opción muy eficiente incluso para grandes volúmenes de datos. Otra ventaja importante es que puede implementarse *in-place* , usando poco espacio adicional de memoria. Sin embargo, en el peor caso su complejidad puede elevarse a $O(n^2)$, aunque esto puede evitarse en gran medida mediante estrategias adecuadas de selección del pivote. Por otro lado, no es estable salvo que se realicen modificaciones explícitas en su implementación.

Merge Sort (Ordenamiento por mezcla)

> Merge Sort es otro algoritmo basado en el paradigma de "divide y vencerás", utilizado para ordenar listas de forma eficiente. Su estrategia consiste en dividir recursivamente la lista en mitades, ordenar cada una de ellas y luego combinar las sublistas ya ordenadas en una única lista final, también ordenada.

Este algoritmo garantiza un rendimiento predecible, con una complejidad temporal de O(n log n) en el mejor, peor y caso promedio. Por esta razón, resulta especialmente útil cuando se requiere estabilidad en los tiempos de ejecución, independientemente del estado inicial de los datos.

Una de sus principales características es que se trata de un algoritmo estable, lo que significa que conserva el orden relativo entre elementos iguales, algo relevante en ciertos contextos como bases de datos o estructuras ordenadas jerárquicamente.

Sin embargo, a diferencia de Quicksort, Merge Sort no es in-place, ya que requiere espacio adicional de memoria proporcional al tamaño de la lista, lo que puede representar una desventaja en sistemas con recursos limitados. Aun así, es ampliamente utilizado en muchas

bibliotecas estándar de programación por su buen rendimiento general y comportamiento confiable incluso con listas muy grandes o parcialmente ordenadas.

A continuación, se muestra una tabla con una comparación general de los algoritmos de ordenamiento que vimos en este trabajo. Esta tabla te va a ayudar a entender de forma rápida cuáles son sus principales características, cuándo conviene usar cada uno y qué cosas hay que tener en cuenta a la hora de elegir uno u otro.

Algoritmo	Complejidad Temporal	Ventajas	Desventajas	Estabilidad	Uso recomendado
Bubble Sort	$O(n^2)$	Muy simple de implementar	Ineficiente para listas grandes	Sí	Educativo / listas pequeñas
Insertion Sort	$O(n^2)$	Bueno para listas pequeñas o casi ordenadas	Poco eficiente en listas desordenadas	Sí	Listas pequeñas o semiordenadas
Selection Sort	$O(n^2)$	Fácil de entender, pocos intercambios	Siempre lento, no es estable	No	Educativo / pocos swaps
Quicksort	O(n log n) promedio	Muy rápido, bajo uso de memoria	Puede ser $O(n^2)$, no es estable	No	Datos grandes, sin necesidad de estabilidad
Timsort (sort())	O(n log n) promedio	Excelente en casos reales, estable	Complejo de implementar manualmente	Sí	Uso general en producción
Merge Sort	O(n log n)	Eficiente y estable, buen rendimiento en grandes volúmenes	Uso de memoria adicional, más complejo que algoritmos simples	Sí	Listas grandes donde se requiere estabilidad

3. Caso Práctico

Para el presente trabajo práctico, implementamos un programa en python que simula un juego de adivinanza de números, donde se generan una listas con valores aleatorios, de las que se toma un elemento al azar y el usuario debe adivinarlo ingresando por consola números en cada intento.

Allí, el programa lo primero que hace es buscar el elemento objetivo en la lista (no lo busca por cada intento), de dos maneras:

1. Aplicando búsqueda lineal
2. Aplicando búsqueda binaria

En cada uno de los casos, mide el tiempo en milisegundos que le toma al programa ubicar el elemento secreto a adivinar.

El tipo de ordenamiento utilizado ha sido la implementación del método incluido en la librería estándar de python *sort( )*.

Este método realiza un ordenamiento del tipo timsort, que combina:

- Merge Sort (eficiente para grandes volúmenes)
- Insertion Sort (eficiente para pequeñas secuencias o cuando hay orden parcial)

Fue diseñado para ser rápido con datos reales, especialmente aquellos que ya están parcialmente ordenados.

Link repositorio GitHub con código fuente: https://github.com/juliandg1995/UTN-TUPaD-P1/tree/main/12%20B%C3%BAsqueda%20y%20ordenamiento/archivos_python

4. Metodología Utilizada

Utilizamos el lenguaje Python, aplicando algunas de las técnicas y conceptos aprendidos durante la cursada para armar el caso práctico, utilizando diversas funciones y módulos para segmentar el código, hacerlo más visible y reutilizable.
Es así como el repositorio cuenta con dos archivos, uno con el programa principal (***juego_adivinanza.py***) y otro con las funciones implementadas (***funciones.py***).

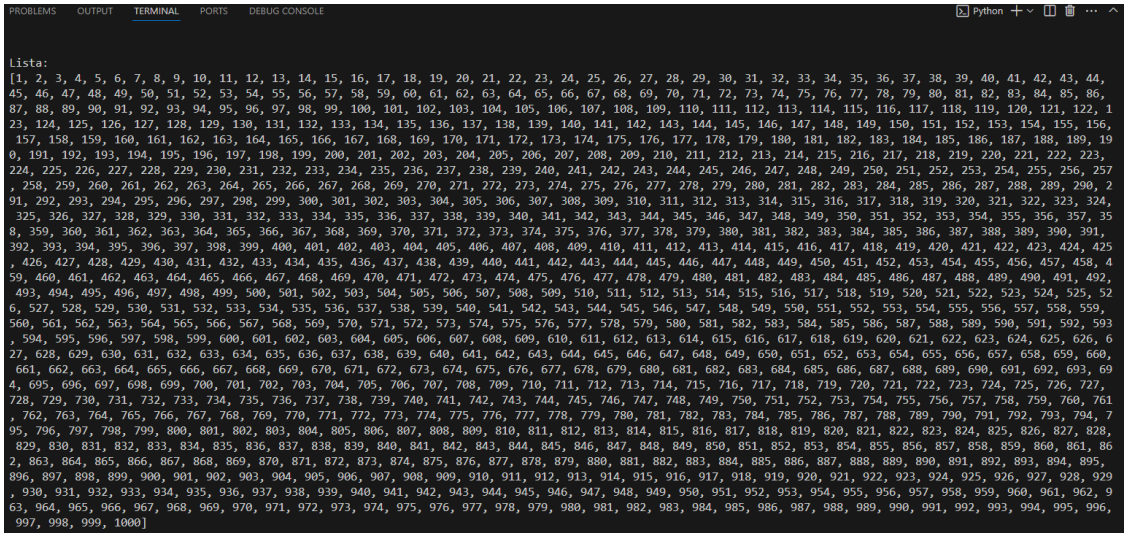
Básicamente, en el programa se plasma el código que llama a las funciones con la lógica operacional y todo lo referido a inputs/outputs por consola, a modo de interfaz de comunicación con el usuario.

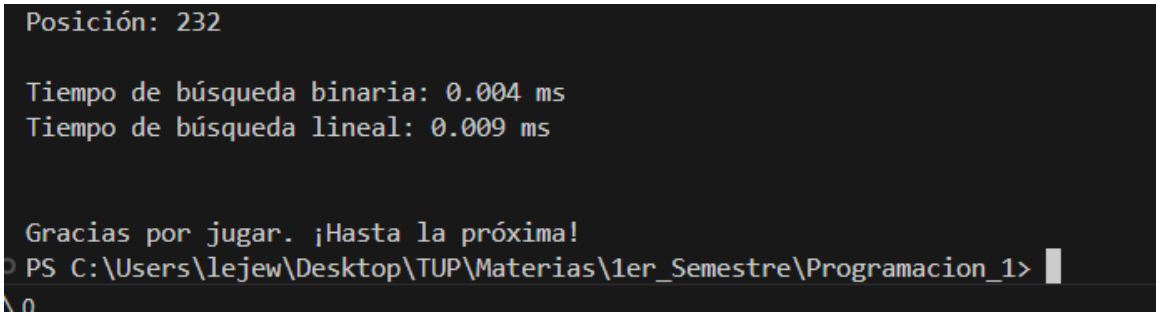
Algunas funciones devuelven múltiples valores (tuplas), y es por eso que dejamos comentarios en el código explicando el funcionamiento de algunas secciones donde la sintaxis es un tanto más compleja.

Para el control de versiones, utilizamos git y GitHub como repositorio.

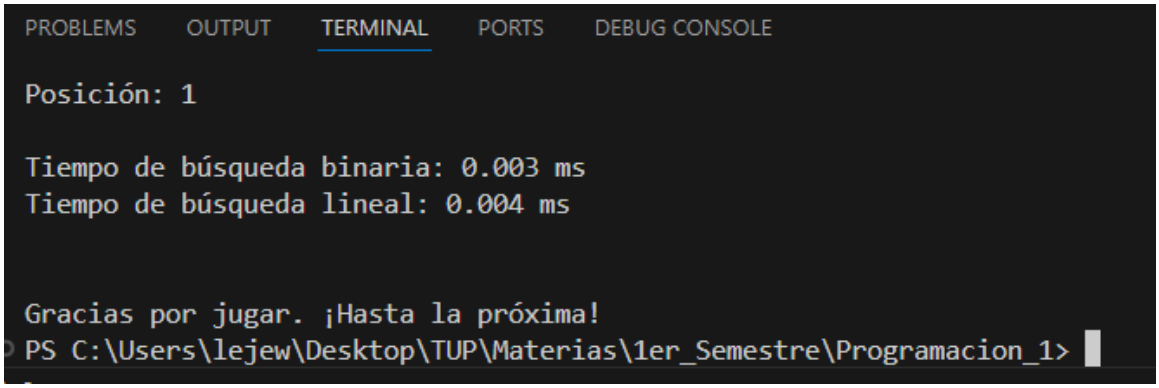
5. Resultados Obtenidos

Esto, a modo de una simple prueba para evidenciar que en los casos donde el usuario desea crear una lista grande, la búsqueda binaria resulta mucho más eficiente, ya que lo que demora en encontrar el elemento es en un tiempo mucho menor a la búsqueda lineal.





Sin embargo, si comenzamos a achicar los tamaños de las listas vemos claramente cómo la búsqueda binaria empieza a perder eficiencia frente a la lineal.



6. Conclusiones

Como conclusión, podemos decir que para grandes volúmenes de datos, resulta crucial la utilización de algoritmos eficientes de búsqueda como el binary search, ya que las aplicaciones modernas requieren de agilidad y tiempos de respuesta muy cortos, y si dependieramos únicamente de búsquedas lineales o no contaramos con algoritmos de ordenamiento de listas, esta tarea sería mucho más costosa en tiempo y recursos.

También vimos que usar el algoritmo Timsort, que es el que Python usa por defecto, es una buena opción para ordenar listas, ya que combina lo mejor de Merge Sort e Insertion Sort.

En resumen, este proyecto nos ayudó a entender cómo se aplican los algoritmos de búsqueda y ordenamiento en un ejemplo real, y nos mostró la importancia de elegir métodos eficientes al resolver problemas en programación.

7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. *https://docs.python.org/3/*

-Complejidad Computacional 101: Big O, Búsqueda Lineal y Búsqueda Binaria, Victor Sanz, *https://www.youtube.com/watch?v=JBm5OXbReQE*

-Khan Academy. (s.f.). *Un juego de adivinanzas, https://es.khanacademy.org/computing/computer-science/algorithms/intro-to-algorithms/a/a-guessing-game*

-Clase de Programación: Búsquedas y Ordenamiento en Python, https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing#scrollTo=iaTCzGLnsrmA

-Corrector de texto alternativo, *https://www.correctoronline.es/*

8. Anexos

- Enlace al repositorio en GitHub: https://github.com/juliandg1995/UTN-TUPaD-P1/tree/main/12%20B%C3%BAsqueda%20y%20ordenamiento/archivos_python
- Enlace al video de Youtube explicativo: https://www.youtube.com/watch?v=wl1h2A5ydg8