Tecnicatura en Programación

Trabajo Práctico de Integración

“Búsqueda Binaria vs Lineal: Análisis Comparativo con Python”

Materia: Programación I

Integrantes:

- Cozzi Pablo Osvaldo
- Davidenco Martin

09 de Junio, 2025

ÍNDICE

1. INTRODUCCIÓN

Este trabajo práctico tiene como finalidad comparar el rendimiento, las ventajas y las limitaciones de dos algoritmos fundamentales en programación: búsqueda lineal y búsqueda binaria. A partir de esta comparación, se busca comprender en qué contextos conviene aplicar cada uno, y cómo influyen factores como el orden previo de los datos, el tamaño de la lista y el enfoque de implementación (iterativo o recursivo).

Para sostener la comparación, se desarrollaron diversos scripts que permiten medir y evidenciar con datos reales tanto la eficiencia como el costo oculto de utilizar uno u otro algoritmo. Además, se incorporó un análisis de rendimiento entre métodos de ordenamiento, ya que este aspecto influye directamente en la aplicabilidad de la búsqueda binaria. También se analizó cómo el diseño del algoritmo (iterativo o recursivo) impacta en el rendimiento real dentro del lenguaje Python.

Objetivos del trabajo

- Comparar el comportamiento de la búsqueda binaria y la búsqueda lineal en distintos tamaños de listas.
- Analizar las condiciones necesarias para aplicar búsqueda binaria y su impacto en tiempo de ejecución.
- Evaluar el costo computacional de ordenar una lista antes de aplicar búsqueda binaria.
- Medir el rendimiento de los algoritmos en versiones iterativas y recursivas.
- Determinar cuándo resulta más eficiente utilizar búsqueda binaria y cuándo no.
- Aplicar conceptos teóricos de complejidad algorítmica (O(n) vs. O(log n)) en situaciones prácticas.
- Reflexionar sobre las decisiones técnicas a tomar al elegir algoritmos según el contexto del problema.
-

2. MARCO TEÓRICO:

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales de la programación, utilizados en sistemas que requieren localizar o reordenar datos de manera eficiente. Comprender su funcionamiento y sus implicancias en el rendimiento permite tomar decisiones técnicas más acertadas en el desarrollo de software.

Este trabajo se centra en comparar dos estrategias de búsqueda: búsqueda lineal y búsqueda binaria, evaluando sus ventajas, limitaciones y condiciones de uso. También se analiza cómo el rendimiento puede verse afectado por la necesidad de ordenar los datos previamente y por el tipo de implementación (recursiva o iterativa).

2.1 Búsqueda lineal

La búsqueda lineal (o secuencial) consiste en recorrer una lista elemento por elemento hasta encontrar el objetivo, o hasta agotar todos los elementos. Es una estrategia simple, no requiere orden previo, y funciona correctamente en cualquier lista.
Complejidad temporal: O(n)
Ventajas:

- Funciona en listas desordenadas
- Fácil de implementar

Desventajas:

- Poco eficiente en listas grandes
- Tiempo de ejecución crece linealmente con el tamaño de los datos

Es útil en contextos donde el volumen de datos es bajo o cuando no se puede garantizar el orden de los mismos.

2.2 Búsqueda binaria

La búsqueda binaria es un algoritmo más eficiente, pero con una condición: la lista debe estar previamente ordenada. Su lógica se basa en dividir el espacio de búsqueda en mitades sucesivas hasta encontrar el valor deseado o descartar su existencia.
Complejidad temporal: O(log n)

Ventajas:

- Muy rápida para grandes volúmenes de datos
- Crece logarítmicamente (más lenta la tasa de crecimiento del costo computacional)

Desventajas:

- Requiere que la lista esté ordenada
- El costo de ordenar puede superar la ganancia si se hace una sola búsqueda

La búsqueda binaria es ideal cuando ya se cuenta con datos ordenados o si se van a realizar muchas búsquedas sobre el mismo conjunto.

2.3 Complejidad algorítmica

La notación Big O permite comparar el crecimiento del tiempo de ejecución de un algoritmo en función del tamaño de la entrada (n).

Algoritmo	Complejidad Temporal	Crecimiento
Búsqueda Lineal	O(n)	Lineal
Búsqueda Binaria	O(log n)	Logarítmico
Ordenamiento Burbuja	$O(n^2)$	Cuadrático
Ordenamiento Rápido	O(n log n)	Log-lineal

Esta notación es clave para comprender que no solo importa si un algoritmo es rápido en la práctica, sino cómo se comporta al aumentar el tamaño de los datos.

2.4 Algoritmos de ordenamiento

Los algoritmos de ordenamiento permiten acomodar datos bajo un criterio determinado (numérico, alfabético, etc.). Son fundamentales para aplicar búsqueda binaria y mejorar la eficiencia en la manipulación de datos.

Bubble Sort: Comparación repetida entre elementos adyacentes. Fácil de entender pero muy ineficiente para listas grandes ($O(n^2)$).

QuickSort: Utiliza el enfoque “divide y vencerás” seleccionando un pivote. Muy rápido y eficiente en la práctica (O(n log n)). Es el algoritmo más utilizado para el ordenamiento general.

2.5 Implementación: iterativa vs. recursiva

Muchos algoritmos pueden implementarse en versiones iterativas (con bucles) o recursivas (con llamadas a sí mismos).

En Python, las versiones iterativas suelen ser más rápidas, ya que la recursión implica un mayor consumo de memoria y tiempo por el llamado constante a funciones.

Sin embargo, la recursividad suele resultar más intuitiva o elegante para ciertos problemas, como la búsqueda binaria o el ordenamiento rápido.

En este trabajo se analizaron ambas versiones para evaluar el impacto real del enfoque de implementación sobre el rendimiento del programa.

3. CASO PRÁCTICO:

Para poner en práctica los conceptos teóricos abordados, se desarrollaron una serie de scripts en Python que permiten simular diferentes escenarios y medir el rendimiento de distintos algoritmos de búsqueda y ordenamiento. A través de estas pruebas controladas, se buscó responder de forma empírica a la pregunta central del trabajo: ¿Cuándo conviene realmente usar la búsqueda binaria?

Los resultados se organizaron en tres grandes bloques de análisis: eficiencia teórica, eficiencia práctica considerando el ordenamiento, y diferencias entre enfoques iterativos y recursivos.

3.1 Análisis teórico de operaciones (archivo: analisis_teorico.py)

Este script demuestra experimentalmente la diferencia de operaciones que realiza cada algoritmo cuando se trabaja sobre una lista ya ordenada. Se generaron listas de distintos tamaños y se midió la cantidad de comparaciones necesarias para encontrar un elemento con: Búsqueda lineal y Búsqueda binaria.

Resultado: La búsqueda lineal realiza un número de operaciones proporcional al tamaño de la lista. La búsqueda binaria realiza un número de operaciones muy reducido, incluso en listas de tamaño grande, lo que confirma su eficiencia teórica O(log n). Este experimento evidencia que, si la lista está ordenada, la búsqueda binaria es significativamente más rápida en términos de operaciones necesarias.

3.2 Análisis práctico de rendimiento (archivo: analisis_rendimiento.py)

El objetivo de este script fue mostrar el “costo oculto” de la búsqueda binaria: el tiempo que lleva ordenar una lista antes de aplicar este algoritmo. Para eso se compararon dos estrategias:

Estrategia 1: aplicar búsqueda lineal directamente sobre la lista desordenada

Estrategia 2: ordenar la lista con QuickSort y luego aplicar búsqueda binaria

Se midió el tiempo total de ejecución para cada estrategia en listas de diferentes tamaños.

Resultado: En listas pequeñas, la búsqueda lineal fue más rápida. En listas grandes, la eficiencia de QuickSort + búsqueda binaria comenzó a superar a la búsqueda lineal directa. En general, si se realiza una sola búsqueda sobre datos desordenados, conviene usar búsqueda lineal. Pero si los datos ya están ordenados o se van a hacer muchas búsquedas, la búsqueda binaria es la mejor opción. Este experimento aporta una visión realista al análisis, considerando no solo la eficiencia del algoritmo de búsqueda, sino el costo total de preparación de los datos.

3.3 Comparación de enfoque iterativo vs. recursivo (archivo: comparacion_recursiva.py)

Este script compara el rendimiento entre las versiones iterativas y recursivas de los algoritmos de búsqueda (tanto lineal como binaria). Se midieron los tiempos de ejecución en listas de tamaños variados.

Resultado: En todos los casos, las versiones iterativas fueron más rápidas. La diferencia es más notable a medida que crece la lista. Esto no se debe al algoritmo en sí, sino al comportamiento interno del lenguaje Python: las llamadas recursivas generan mayor carga computacional (overhead).

Este análisis permitió reflexionar sobre cómo la elección del enfoque de implementación afecta el rendimiento, incluso cuando el algoritmo lógico es el mismo.

3.4 Comparación entre métodos de ordenamiento (archivo: comparacion_ordenamiento.py)

Para complementar el análisis, se compararon dos algoritmos de ordenamiento:

- Bubble Sort: sencillo pero ineficiente ($O(n^2)$)
- QuickSort: más complejo, pero altamente eficiente ($O(n \log n)$)

Se ejecutaron ambos algoritmos sobre listas de distintos tamaños y se midieron los tiempos.

Resultado: QuickSort fue significativamente más rápido que Bubble Sort en todos los tamaños. La diferencia se hizo más evidente a medida que aumentaba la cantidad de elementos. Esta información es clave para entender por qué se eligió QuickSort como método de ordenamiento previo a la búsqueda binaria en los experimentos de rendimiento.

3.5 Funciones utilizadas (archivo: algoritmos.py)

Todas las funciones de búsqueda y ordenamiento utilizadas fueron centralizadas en un archivo único, algoritmos.py. Desde allí se importaron en los distintos scripts de análisis. Esto permitió:

- Evitar duplicación de código
- Facilitar el mantenimiento
- Modularizar el sistema, siguiendo buenas prácticas

Entre las funciones destacadas se encuentran: busqueda_lineal() , busqueda_lineal_recursiva(), busqueda_binaria_iterativa() y busqueda_binaria_recursiva(), ordenamiento_burbuja() y ordenamiento_rapido() (QuickSort)

A continuación, adjuntamos el enlace al video realizado, donde se puede ver el desarrollo del trabajo práctico: [https://www.youtube.com/watch?v=0_SaU-NGsoQ]

4. METODOLOGÍA UTILIZADA:

El desarrollo del presente trabajo integrador se estructuró en una secuencia lógica de etapas, que permitieron avanzar desde la formulación de la pregunta inicial hasta la obtención de conclusiones respaldadas por pruebas empíricas.

El enfoque adoptado fue práctico, iterativo y exploratorio: se diseñaron scripts de prueba para validar hipótesis sobre la eficiencia de distintos algoritmos, se analizaron los resultados obtenidos, y se documentó el proceso con una mirada reflexiva.

Etapa 1: Planteamiento del problema

Objetivo de la etapa: Delimitar el alcance del trabajo y diseñar una estrategia basada en pruebas concretas, no solo en teoría.

- Formulación de la pregunta central: ¿Cuándo conviene realmente usar la búsqueda binaria?

- Revisión teórica de algoritmos de búsqueda, ordenamiento y complejidad.

- Identificación de factores relevantes para comparar: tiempo, cantidad de operaciones, orden previo, enfoque iterativo vs. recursivo.

Etapa 2: Desarrollo de funciones modulares

Implementación de funciones de búsqueda y ordenamiento en un módulo central (algoritmos.py).

Creación de versiones iterativas y recursivas para cada algoritmo.

Validación individual del funcionamiento correcto de cada función con listas de prueba.

El objetivo fue centralizar las funciones permite utilizarlas en múltiples experimentos, facilitando la organización del código y evitando redundancias.

Etapa 3: Construcción de scripts de análisis

analisis_teorico.py: comparó la cantidad de operaciones realizadas por cada búsqueda.

analisis_rendimiento.py: midió tiempos reales de búsqueda lineal vs. ordenar+buscar binaria.

comparacion_recursiva.py: comparó rendimiento de versiones iterativas y recursivas.

comparacion_ordenamiento.py: comparó tiempos de ordenamiento con Bubble Sort y QuickSort.

Objetivo de la etapa: Producir evidencia experimental para contrastar y complementar los resultados teóricos.

Etapa 4: Análisis e interpretación de resultados.

Ejecución de los scripts con listas de diferentes tamaños.

Registro de tiempos, número de comparaciones y veredictos automáticos.

Comparación directa de estrategias en distintos escenarios.

Reflexión sobre condiciones reales de uso y rendimiento en Python.

Etapa 5: Redacción del informe y preparación del video.

Redacción del trabajo práctico en formato académico.

Síntesis de conclusiones con base en la evidencia recolectada.

Organización de ideas para el guión del video explicativo.

Planificación de las diapositivas de apoyo y división de roles.

Trabajo colaborativo:

El desarrollo del código estuvo a cargo de uno de los integrantes, mientras que la documentación, el análisis y la presentación fueron coordinados entre ambos. Se utilizó una carpeta compartida y GitHub como repositorio de respaldo.

5. RESULTADOS:

Diferencias entre implementación iterativa y recursiva: El script comparacion_recursiva.py permitió medir el tiempo de ejecución de las versiones iterativas y recursivas tanto en búsqueda binaria como en búsqueda lineal. En todos los casos, las versiones iterativas fueron más rápidas. La recursividad introduce un overhead (costo adicional por llamadas a funciones) que afecta el rendimiento real, aunque no la lógica del algoritmo.

Esta sección permite comprender que no solo importa el algoritmo, sino también cómo se lo implementa, especialmente en lenguajes como Python.

Comparación entre algoritmos de ordenamiento: Con el script comparacion_ordenamiento.py, se midió el rendimiento de: Bubble Sort, un algoritmo simple pero ineficiente, de complejidad $O(n^2)$, QuickSort, un algoritmo más sofisticado y con rendimiento O(n log n) en la mayoría de los casos. El resultado fue que QuickSort fue notablemente más rápido en todos los tamaños de lista. Bubble Sort demoró varios segundos más incluso en listas de tamaño moderado (1.000 elementos). Esta comparación justificó la elección de QuickSort como método de ordenamiento previo en los experimentos con búsqueda binaria.

Conclusión general de los resultados

- La búsqueda binaria es más rápida sólo si los datos están ordenados, o si se hacen muchas búsquedas sobre los mismos datos.
- La búsqueda lineal sigue siendo más práctica cuando se hace una sola consulta sobre datos desordenados.
- En Python, las versiones iterativas de los algoritmos funcionan mejor que las recursivas, en cuanto a rendimiento.
- El ordenamiento influye directamente en la eficiencia general de los sistemas de búsqueda.

6. CONCLUSIÓN:

El presente trabajo permitió comprender de manera teórica y práctica el funcionamiento, las ventajas y las limitaciones de los algoritmos de búsqueda más comunes: búsqueda lineal y búsqueda binaria. A través del desarrollo de distintos scripts de análisis, se evaluaron condiciones de uso, tiempos de ejecución y diferencias de implementación, lo que llevó a conclusiones claras sobre su aplicabilidad real.

Si bien la búsqueda binaria es altamente eficiente en listas ordenadas, su rendimiento depende directamente de que los datos ya estén organizados. En contraposición, la búsqueda lineal, aunque más lenta en promedio, no requiere ningún procesamiento previo, lo que la hace útil en escenarios donde se realiza una única búsqueda sobre datos sin ordenar.

El trabajo también permitió observar que el tipo de implementación (iterativa o recursiva) impacta significativamente en el tiempo de ejecución, especialmente en Python, donde las llamadas recursivas tienen un costo adicional.

Aprendizajes:

- La eficiencia teórica de un algoritmo no siempre se traduce en eficiencia práctica si no se consideran las condiciones previas del conjunto de datos.
- Implementaciones recursivas pueden resultar más elegantes, pero en algunos lenguajes como Python son menos eficientes que sus versiones iterativas.
- Es fundamental considerar el contexto del problema antes de elegir un algoritmo: una buena elección depende de los datos, la cantidad de búsquedas y la necesidad o no de ordenar.
- La medición experimental es una herramienta clave para validar o refutar suposiciones teóricas en programación.

Posibles mejoras:

- Incorporar un análisis gráfico (por ejemplo con matplotlib) para representar visualmente los resultados de tiempos y comparaciones.

- Añadir una interfaz gráfica simple que permita al usuario ejecutar pruebas sin necesidad de modificar código.
- Extender los análisis a otros algoritmos de búsqueda (como búsqueda exponencial o por interpolación).
- Incluir un sistema de registro automático de resultados en archivos .csv para facilitar su análisis posterior.
- Explorar el comportamiento en otros lenguajes (como C, JavaScript o Go) para comparar cómo influyen el lenguaje y el entorno de ejecución en el rendimiento.

Este trabajo no solo reforzó conceptos esenciales sobre algoritmos, sino que también brindó herramientas concretas para tomar decisiones informadas al momento de desarrollar sistemas que involucren operaciones de búsqueda u ordenamiento. La experiencia fue enriquecedora tanto desde lo técnico como desde el trabajo colaborativo..

7. BIBLIOGRAFÍA:

McDowell, G. (2020). Cracking the Coding Interview (6.ª ed.). CareerCup. Capítulos sobre análisis de complejidad y estructuras de datos aplicadas.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley. Enfoque práctico y matemático sobre algoritmos clásicos de búsqueda y ordenamiento.

Python Software Foundation. (2024). The Python Standard Library Documentation. https://docs.python.org/3/library/

DigitalOcean. (2023). Measuring Algorithm Performance in Python. https://www.digitalocean.com/community/tutorials

UTN – Facultad Regional San Nicolás. (2025). Material de Programación I – Tecnicatura Universitaria en Programación.

TutorialsPoint. (2024). Difference Between Recursive and Iterative Approaches. https://www.tutorialspoint.com/

GeeksforGeeks. (2024). QuickSort Algorithm with Python Examples. https://www.geeksforgeeks.org/