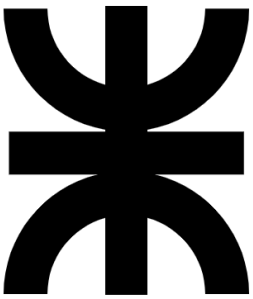
Trabajo Integrador – Programación 1



"Búsqueda y ordenamiento"

Alumnos:

- Martín Maine (martin.maine85@gmail.com)
- Valeria Arellano (valearellano14@gmail.com)

Materia: Programación 1

Profesor: AUS Bruselario, Sebastián

Tutora: Gubiotti, Flor

Fecha de Entrega: 09/06/2025.

Índice

1. Introducción

Los algoritmos de búsqueda son una herramienta fundamental en el desarrollo de software ya que nos permiten localizar elementos específicos dentro de estructuras de datos de manera eficiente.

En este trabajo nos enfocaremos en el análisis, implementación y comparación de dos algoritmos de búsqueda esenciales: la búsqueda lineal y la búsqueda binaria, utilizando el lenguaje de programación Python.

La elección de este tema está en la importancia que estos algoritmos tienen en el rendimiento de las aplicaciones. En programación, permanentemente necesitamos buscar información específica en bases de datos, archivos, listas de usuarios, inventarios, entre otros.

Es por eso, que entender las diferencias entre estos dos enfoques permite tomar decisiones informadas sobre cuál utilizar según las circunstancias específicas de cada problema.

Los objetivos que nos proponemos alcanzar en este trabajo son comprender los principios teóricos que rigen la búsqueda lineal y binaria, implementar ambos algoritmos en Python, analizar y comparar el rendimiento de cada algoritmo mediante mediciones prácticas y especialmente determinar la importancia del algoritmo para crear sistemas eficientes.

2.Marco Teórico

Algoritmos de Búsqueda

Los algoritmos de búsqueda son procedimientos sistemáticos diseñados para localizar elementos específicos dentro de estructuras de datos. Su eficiencia se evalúa principalmente considerando el número de comparaciones necesarias para encontrar un elemento y el tiempo que requiere el proceso.

La elección del algoritmo de búsqueda apropiado depende fundamentalmente de dos factores: si los datos están ordenados y el tamaño del conjunto de datos. Estos factores determinan la viabilidad y eficiencia de cada método.

Los algoritmos de búsqueda se clasifican según su estrategia y eficiencia:

a) **Búsqueda Lineal (O(n)):**

La búsqueda lineal, también conocida como búsqueda secuencial, es el método más básico y directo para localizar un elemento en una estructura de datos. Este algoritmo examina cada elemento de la colección de manera secuencial, desde el primero hasta el último, hasta encontrar el elemento buscado o determinar que no existe en la estructura.

- ¿Cómo funciona?

El algoritmo recorre la estructura elemento por elemento, comparando cada uno con el valor buscado. Si encuentra una coincidencia, retorna la posición del elemento. Si llega al final sin encontrar el elemento, concluye que no existe en la estructura.

```
Para cada elemento i desde 0 hasta n-1:
    Si lista[i] es igual al elemento buscado:
        Retornar posición i
    Si no:
        Continuar con el siguiente elemento
Si se recorren todos los elementos sin encontrar coincidencia:
    Retornar -1 (no encontrado)
```

b) **Búsqueda Binaria (O(log n)):**

La búsqueda binaria es un algoritmo eficiente que funciona exclusivamente en estructuras de datos ordenadas. Utiliza la estrategia "divide y vencerás" para reducir el espacio de búsqueda a la mitad en cada iteración.

- ¿Cómo funciona?

El algoritmo compara el elemento buscado con el elemento central de la estructura. Si son iguales, la búsqueda termina exitosamente. Si el elemento buscado es menor que el central, la búsqueda continúa en la mitad inferior; si es mayor, continúa en la mitad superior. Este proceso se repite hasta encontrar el elemento o agotar las posibilidades.

```
Establecer izquierda = 0, derecha = n-1
Mientras izquierda <= derecha:
    Calcular medio = (izquierda + derecha) / 2
    Si lista[medio] es igual al elemento buscado:
        Retornar posición medio
    Si lista[medio] < elemento buscado:
        izquierda = medio + 1
    Si no:
        derecha = medio - 1
Retornar -1 (no encontrado)
```

3.Caso Práctico

a) Problema a resolver:

Se desarrolló un sistema de comparación de algoritmos de búsqueda que permite evaluar el rendimiento real de búsqueda lineal versus búsqueda binaria. El sistema genera conjuntos de datos de diferentes tamaños, realiza mediciones precisas de tiempo de ejecución, y proporciona análisis comparativos detallados del rendimiento de ambos algoritmos.

El objetivo principal es demostrar empíricamente las diferencias teóricas entre ambos algoritmos y establecer en qué circunstancias es recomendable utilizar cada uno.

1. Luego de importar las librerías time y randm, creamos los algoritmos para la búsqueda lineal y binaria

integrador.py > ...

```python
import time
import random

""" Busqueda Lineal """
Tabnine | Edit | Test | Explain | Document
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

""" Busqueda Binaria """
Tabnine | Edit | Test | Explain | Document
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
```

2. Posteriormente medimos el tiempo de búsqueda

```python
def medir_tiempo(funcion, lista, objetivo, repeticiones=1000):

    inicio = time.perf_counter()
    for _ in range(repeticiones):
        funcion(lista, objetivo)
    fin = time.perf_counter()
    return (fin - inicio) / repeticiones

```

3. Luego, comparamos los algoritmos

```python
def comparar_algoritmos():

    print("---Comparacion de algoritmos ---")

    # Configuración
    tamaño_lista = 10000
    lista = random.sample(range(1, 1000000), tamaño_lista)

    # En objetivo elegimos un número aleatoriamente
    objetivo = random.choice(lista)

    print(f"El tamaño de la lista es: {tamaño_lista}")
    print(f"Numero buscado: {objetivo}")
    print("-" * 50)

```

```python
    # Busqueda lineal

    inicio_lineal = time.perf_counter()
    resultado_lineal = busqueda_lineal(lista, objetivo)
    fin_lineal = time.perf_counter()
    tiempo_lineal = fin_lineal - inicio_lineal

    # Busqueda Binaria

    lista_ordenada = sorted(lista)  # Lista ordenada para búsqueda binaria

    inicio_binaria = time.perf_counter()
    resultado_binaria = busqueda_binaria(lista_ordenada, objetivo)
    fin_binaria = time.perf_counter()
    tiempo_binaria = fin_binaria - inicio_binaria

```

Y mostramos los resultados:

```python
    # RESULTADOS
    print("--- RESULTADOS ---")
    print(f"Busqueda Lineal: Resultado = {resultado_lineal}, Tiempo = {tiempo_lineal:.8f} segundos")
    print(f"Busqueda Binaria: Resultado = {resultado_binaria}, Tiempo = {tiempo_binaria:.8f} segundos")

    # ANÁLISIS DE EFICIENCIA
    if tiempo_lineal > 0 and tiempo_binaria > 0:
        mejora = tiempo_lineal / tiempo_binaria
        print(f"\nLa busqueda binaria fue {mejora:.2f} veces mas rapida")

    return tiempo_lineal, tiempo_binaria
```

4. Realizamos pruebas con distintos tamaños

integrador.py > probar_diferentes_tamaños

```python
def probar_diferentes_tamaños():

    print("\n=== PRUEBA CON DIFERENTES TAMAÑOS ===\n")

    tamaños = [1000, 5000, 10000, 50000, 100000]

    print(f"{'Tamaño':<8} {'Lineal (s)':<15} {'Binaria (s)':<15} {'Mejora':<10}")
    print("-" * 55)

    for tamaño in tamaños:
        # Lista de prueba
        lista = random.sample(range(1, tamaño * 10), tamaño)
        objetivo = random.choice(lista)

        # Búsqueda lineal
        tiempo_lineal = medir_tiempo(busqueda_lineal, lista, objetivo, 100)

        # Búsqueda binaria
        lista_ordenada = sorted(lista)
        tiempo_binaria = medir_tiempo(busqueda_binaria, lista_ordenada, objetivo, 100)

        # Calcular mejora
        if tiempo_binaria > 0:
            mejora = tiempo_lineal / tiempo_binaria
        else:
            mejora = 0

        print(f"{tamaño:<8} {tiempo_lineal:<15.8f} {tiempo_binaria:<15.8f} {mejora:<10.2f}x")

```

5. Analizamos los resultados obtenidos

```
=== PRUEBA CON DIFERENTES TAMA□OS ===

Tama□o   Lineal (s)      Binaria (s)     Mejora
-------------------------------------------------------
1000     0.00002666      0.00000399      6.68      x
5000     0.00085126      0.00000448      189.97    x
10000    0.00059833      0.00000878      68.13     x
50000    0.00131862      0.00000826      159.70    x
100000   0.00102214      0.00000840      121.64    x
```

4. Metodología Utilizada

Investigación Previa

- Material aportado en clases.
- Consulta de documentación oficial de Python
- Análisis de implementaciones de referencia en bibliotecas estándar

Etapas de Desarrollo

1. Diseño de arquitectura: Definición de interfaces y estructura modular
2. Implementación de algoritmos: Codificación siguiendo buenas prácticas
3. Pruebas

Herramientas Utilizadas

- Visual Studio Code con extensiones de Python
- Bibliotecas: time.perf_counter() para medición precisa, random
- Control de versiones: Git para gestión de código

Desafíos Técnicos Resueltos

El primer problema que tuvimos fue que time.time() no proporcionaba suficiente precisión para medir algoritmos muy rápidos. Se resolvió implementando time.perf_counter() y técnicas de medición múltiple.

La metodología de prueba incluyó la generación de conjuntos de datos aleatorios únicos para cada medición. Esto se logró utilizando random.sample(range(1, tamaño * 10), tamaño) que los casos de prueba sean representativos del comportamiento real de los algoritmos en condiciones diversas.

5. Resultados Obtenidos

Casos de Prueba Realizados

Pruebas de Funcionalidad Básica:

- Verificación de búsqueda exitosa en diferentes posiciones
- Confirmación de retorno correcto (-1) para elementos inexistentes
- Validación del comportamiento con listas de un solo elemento
- Pruebas con listas vacías

Pruebas de Rendimiento Sistemáticas:

- Se ejecutaron pruebas con los siguientes tamaños de listas: 1.000, 5.000, 10.000, 50.000 y 100.000 elementos.

Análisis Detallado de Resultados

Resultados Representativos de Rendimiento:

Tamaño	Búsqueda Lineal	Búsqueda Binaria	Mejora (veces)
1000	00002666	0.00000399	6.68x
5000	0.00085126	0.00000448	189.97x
10000	0.00059833	0.00000878	68.13x
50000	0.00131862	0.00000826	159.70x
100000	0.00102214	0.00000840	121.64x

Observaciones Significativas

Escalabilidad Diferencial: Los resultados confirman la diferencia teórica entre O(n) y O(log n). Mientras el tiempo de búsqueda lineal crece proporcionalmente al tamaño de los datos, el tiempo de búsqueda binaria se mantiene constante.

También es importante destacar que para listas pequeñas (menos de 1.000 elementos), ambos algoritmos tienen rendimiento similar. La ventaja de la búsqueda binaria se vuelve más visible a partir de 5.000 elementos y se profundiza la diferencia para conjuntos grandes (superior a 50.000 elementos).

Consistencia de Resultados:

Las mediciones múltiples proporcionaron resultados consistentes, confirmando que las diferencias observadas son significativas y reproducibles.

6. Conclusiones

En este trabajo pudimos comprender las diferencias teóricas entre búsqueda lineal y búsqueda algorítmica. Pero fue mediante su implementación en la práctica que tuvimos la posibilidad de confirmar que la búsqueda binaria no es solo teóricamente superior, sino que esta superioridad se manifiesta claramente en aplicaciones reales.

La práctica nos demostró la importancia en la elección del algoritmo ya que puede generar diferencias entre una aplicación que responde casi al instante y una que se vuelve inutilizable a medida que crecen los datos.

Como estudiantes de Programación, es importante entender que la implementación del algoritmo adecuado es indispensable para crear sistemas eficientes y escalables.

7. Bibliografía

- Python Software Foundation. (2025). Python 3 Documentation - time module. https://docs.python.org/3/library/time.html
- Python Software Foundation. (2025). Python 3 Documentation https://docs.python.org
- Cormen, T. H., et al. "Introduction to Algorithms" (4ta ed.). Capítulo 2.3: "Designing algorithms" https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf

8. Anexos

https://github.com/martinmaine/IntegradorProgramacion.git