UNIVERSIDAD PRIVADA DOMINGO SAVIO
FACULTAD DE INGENIERIA

Actividad

Título: investigación sobre búsqued
Binaria

Docente: Ing.Jimmy Nataniel Requena Llorentty

Materia: Programación 2

Estudiante: LORA COLODRO FABRIZZIO

Junio del 2025

Santa Cruz – Bolivia

Búsqueda Lineal (Linear Search)

Concepto:

Recorre **secuencialmente** cada elemento de una lista (ordenada o desordenada) hasta encontrar el valor deseado o verificar su ausencia.

**Características: **

- **Complejidad temporal: **
  - **Peor caso: ** \(O(n)\) (elemento al final o ausente).
  - **Mejor caso: ** \(O(1)\) (elemento en la primera posición).
  - **Caso promedio: ** \(O(n)\).
- **Complejidad espacial: ** \(O (1)\) (no usa memoria adicional).
- **No requiere datos ordenados. **

**Algoritmo (Pseudocódigo): **

```python
def busqueda_lineal(arr, objetivo):
    for i in range(len(arr)):
        if arr[i] == objetivo:
            return i # Devuelve el índice si lo encuentra
    return -1 # No encontrado
```

**Pasos: **

1. Inicia desde el primer elemento.
2. Compara cada elemento con el objetivo.
3. Si coincide, retorna el índice.
4. Si termina el recorrido sin éxito, retorna -1.

**Ventajas: **

- Sencillo de implementar.

- Funciona con listas no ordenadas.

**Desventajas: **

- Ineficiente para listas grandes (ej.: 1 millón de elementos requiere 1 millón de comparaciones en el peor caso).

**Búsqueda Binaria (Binary Search) **

**Concepto: **

Divide repetidamente una lista **ordenada** a la mitad, comparando el elemento central con el objetivo para descartar una mitad en cada iteración.

**Características: **

- **Complejidad temporal: ** \ (O (\log n) \) en todos los casos (peor, mejor y promedio).

- **Complejidad espacial: **

  - **Iterativa: ** \ (O (1) \).

  - **Recursiva: ** \ (O (\log n) \) (por la pila de llamadas).

- **Requiere datos ordenados. **

**Algoritmo (Iterativo - Pseudocódigo): **

```python
def busqueda_binaria(arr, objetivo):
    izquierda, derecha = 0, len(arr) - 1

    while izquierda <= derecha:
```

```
        medio = (izquierda + derecha) // 2
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            izquierda = medio + 1 # Busca en la mitad derecha
        else:
            derecha = medio - 1 # Busca en la mitad izquierda
    return -1 # No encontrado
```

**Pasos: **

1. Define los límites `izquierda` (0) y `derecha` (n-1).
2. Calcula el índice `medio` del rango actual.
3. Compara `arr[medio]` con el objetivo:
   - **Igual: ** Retorna `medio`.
   - **Objetivo mayor: ** Actualiza `izquierda = medio + 1`.
   - **Objetivo menor: ** Actualiza `derecha = medio - 1`.
4. Repite hasta que `izquierda > derecha` (no encontrado).

**Ventajas: **

- **Extremadamente eficiente** para listas grandes (ej: 1 millón de elementos requiere solo 20 comparaciones, ya que \(\log_2(10^6) \approx 20\)).
- Óptimo para datos estáticos que se buscan repetidamente.

**Desventajas: **

- Requiere que la lista esté **ordenada** (el ordenamiento inicial tiene costo \ (O (n \log n) \)).
- Más complejo de implementar que la búsqueda lineal.

Comparación Clave

Característica	**Búsqueda Lineal**	**Búsqueda Binaria**
Lista ordenada	No necesaria	Obligatoria
Complejidad Temporal	\(O(n)\)	\(O(\log n)\)
Complejidad Espacial	\ (O (1) \)	\(O(1)\) (iterativa)
Implementación	Sencilla	Moderadamente compleja
Eficiencia	Ineficiente para \(n\) grande	Óptima para \(n\) grande
Casos de uso	Listas pequeñas o desordenadas	Listas grandes y ordenadas

---

Ejemplo Visual de Búsqueda Binaria

**Lista ordenada: ** ` [2, 5, 8, 12, 16, 23, 38, 56]`, objetivo = `23`.

1. **Paso 1: ** Medio = índice 3 (valor `12`). `23 > 12` → busca en ` [16, 23, 38, 56]`.

2. **Paso 2: ** Medio = índice 5 (valor `23`). ¡Encontrado!

Conclusión

- **Usa búsqueda lineal** si la lista es pequeña, desordenada o solo se busca una vez.

- **Usa búsqueda binaria** si la lista es grande, está ordenada y se realizarán múltiples búsquedas.

- **¡Nunca uses búsqueda binaria en datos no ordenados! ** Primero ordénalos (\ (O (n \log n) \)).