# 7. Algoritmos de Búsqueda

Los **algoritmos de búsqueda** son procedimientos diseñados para encontrar la ubicación de uno o más elementos dentro de una estructura de datos, o para determinar si el elemento está presente o no. La eficiencia de un algoritmo de búsqueda es crucial, especialmente cuando se trabaja con grandes volúmenes de datos.

—

## 0.1. Tipos Principales de Algoritmos de Búsqueda

Los algoritmos de búsqueda se clasifican generalmente según cómo exploran la estructura de datos:

- **Búsqueda Lineal (o Secuencial):**
  - Recorre la estructura de datos elemento por elemento desde el principio hasta el final, comparando cada elemento con el valor buscado.
  - Es el algoritmo de búsqueda más simple y no requiere que la estructura de datos esté ordenada.
- **Búsqueda Binaria:**
  - Un algoritmo mucho más eficiente que la búsqueda lineal, pero con la estricta condición de que la estructura de datos (generalmente un array o lista) debe estar **ordenada**.
  - Opera dividiendo repetidamente por la mitad la porción de la lista donde podría estar el elemento.
- **Búsqueda por Hash:**
  - Utiliza una función hash para calcular directamente un índice o clave, permitiendo un acceso muy rápido (casi $\mathcal{O}(1)$ en promedio) al elemento deseado.
  - Requiere una tabla hash (Hash Map/Hash Table) para su implementación.
- **Búsqueda en Árboles (BFS/DFS):**
  - Algoritmos utilizados para buscar nodos en estructuras de datos tipo árbol o grafo, como la búsqueda en anchura (Breadth-First Search - BFS) o la búsqueda en profundidad (Depth-First Search - DFS).

—

## 0.2. Ejemplos de Algoritmos de Búsqueda en C++

Veamos implementaciones de los algoritmos de búsqueda lineal y binaria.

**Búsqueda Binaria ($\mathcal{O}(\log N)$)**

Busca un elemento en un array **ordenado** dividiendo repetidamente el intervalo de búsqueda por la mitad.

Listing 1: Implementación de Búsqueda Binaria

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::sort (si el array no estuviera ordenado)

using namespace std; // Incluyendo namespace std como se solicit

// Funci n de b squeda binaria (iterativa)
int busquedaBinaria(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Evita desbordamiento de int para (left + right)

        if (arr[mid] == target) {
            return mid; // Elemento encontrado
        } else if (arr[mid] < target) {
            left = mid + 1; // Buscar en la mitad derecha
        } else { // arr[mid] > target
            right = mid - 1; // Buscar en la mitad izquierda
        }
    }
    return -1; // Elemento no encontrado
}

int main() {
    vector<int> numeros_ordenados = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int valor_a_buscar_1 = 60;
    int valor_a_buscar_2 = 35;

    cout << "Array ordenado: ";
    for (int x : numeros_ordenados) {
        cout << x << " ";
    }
    cout << endl;

    int indice1 = busquedaBinaria(numeros_ordenados, valor_a_buscar_1);
    if (indice1 != -1) {
```

```
39          cout << "El valor " << valor_a_buscar_1 << " se
                encontr  en el  ndice : " << indice1 << endl;
40      } else {
41          cout << "El valor " << valor_a_buscar_1 << " no
                se encontr  en el array." << endl;
42      }
43
44      int indice2 = busquedaBinaria(numeros_ordenados,
            valor_a_buscar_2);
45      if (indice2 != -1) {
46          cout << "El valor " << valor_a_buscar_2 << " se
                encontr  en el  ndice : " << indice2 << endl;
47      } else {
48          cout << "El valor " << valor_a_buscar_2 << " no
                se encontr  en el array." << endl;
49      }
50      return 0;
51  }
```

—

## 0.3. Análisis de Eficiencia de los Algoritmos de Búsqueda

La eficiencia de los algoritmos de búsqueda varía drásticamente dependiendo de si la estructura de datos está ordenada y del método utilizado.

- **Búsqueda Lineal:**
  - **Complejidad Temporal:** $\mathcal{O}(N)$ en el peor y caso promedio, donde $N$ es el número de elementos. En el mejor caso (elemento al principio), es $\mathcal{O}(1)$.
  - **Complejidad Espacial:** $\mathcal{O}(1)$.
- **Búsqueda Binaria:**
  - **Complejidad Temporal:** $\mathcal{O}(\log N)$ en el peor y caso promedio. Esto se debe a que el espacio de búsqueda se reduce a la mitad en cada paso. Es extremadamente eficiente para grandes conjuntos de datos.
  - **Complejidad Espacial:** $\mathcal{O}(1)$ para la versión iterativa, y $\mathcal{O}(\log N)$ para la versión recursiva debido a la pila de llamadas.
- **Búsqueda por Hash:**
  - **Complejidad Temporal:** Promedio $\mathcal{O}(1)$, pero puede degradarse a $\mathcal{O}(N)$ en el peor caso si hay muchas colisiones y una mala función hash.
  - **Complejidad Espacial:** $\mathcal{O}(N)$ para almacenar la tabla hash.

—

## 0.4. Consideraciones al Elegir un Algoritmo de Búsqueda

La elección del algoritmo de búsqueda depende principalmente de dos factores:

- **¿La estructura de datos está ordenada?:** Si no lo está, la búsqueda lineal es la única opción directa a menos que se ordene primero (lo que añade un costo de ordenación). Si está ordenada, la búsqueda binaria es casi siempre preferible.
- **Frecuencia de las búsquedas:** Si se van a realizar muchas búsquedas en la misma estructura de datos, justificaría el costo inicial de ordenar o construir una tabla hash.
- **Restricciones de memoria:** La búsqueda binaria y lineal son eficientes en memoria. La búsqueda por hash requiere más memoria.

Comprender la eficiencia de los algoritmos de búsqueda es fundamental para el diseño de sistemas de recuperación de información y bases de datos. En la práctica por simplicidad usaremos la funcion find, upperbound() y lowerbound() de la biblioteca algorithms. Sin embargo es importante saber como funcionan.