Módulo 6 – Algoritmos de búsqueda

Curso de Introducción a Estructuras de Datos y Algoritmos

21 de agosto de 2025

Contents

1 Módulo 6 – Algoritmos de Búsqueda: Paradigmas y Análisis de Eficiencia

1.1 0. Introducción: El Problema Fundamental de la Recuperación de Información

La recuperación de información es una de las operaciones computacionales más ubicuas y fundamentales. Desde la consulta de un registro en un sistema de gestión de bases de datos hasta la resolución de símbolos en un compilador, la eficiencia con la que se localiza

un dato es un factor determinante en el rendimiento global de un sistema. Un **algoritmo de búsqueda** es, por tanto, un procedimiento formal diseñado para localizar la existencia y posición de un elemento, o un conjunto de elementos, dentro de una estructura de datos.

Este módulo presenta un análisis comparativo de los principales paradigmas de búsqueda, evaluando sus fundamentos teóricos, su complejidad computacional y los contextos en los que cada uno resulta óptimo. Se examinarán desde métodos de fuerza bruta hasta técnicas logarítmicas y de tiempo constante amortizado, ilustrando el profundo nexo entre la **organización de los datos** y la **eficiencia algorítmica**.

1.2 1. Búsqueda Lineal (Secuencial)

La búsqueda lineal constituye el algoritmo más elemental. Procede mediante el examen exhaustivo y secuencial de cada elemento en una colección hasta que se encuentra el elemento objetivo o se agota el espacio de búsqueda.

1.2.1 Análisis Formal

Dada una colección C de n elementos, el algoritmo verifica la condición $C[i] = x$ para $i = 0, 1, \ldots, n-1$.

- **Complejidad Temporal**:
  - **Peor caso**: El elemento no se encuentra o está en la última posición, requiriendo n comparaciones. La complejidad es, por tanto, $O(n)$.
  - **Caso promedio**: Asumiendo una distribución uniforme de la probabilidad de que el elemento se encuentre en cualquier posición, el número esperado de comparaciones es $\frac{n+1}{2}$, lo que también resulta en una complejidad de $O(n)$.

1.2.1.1 Explicación del análisis Formal

La expresión $\frac{n+1}{2}$ es el **valor esperado** (la media) del número de comparaciones que se realizan en una búsqueda lineal, bajo dos supuestos clave:

1. El elemento que se busca **está presente** en la colección.
2. El elemento tiene la **misma probabilidad** de estar en cualquiera de las n posiciones (distribución uniforme).

La deducción es la siguiente:

- Para encontrar el elemento en la **1ª posición**, se necesita 1 comparación.
- Para encontrarlo en la **2ª posición**, se necesitan 2 comparaciones.
- ...
- Para encontrarlo en la **n-ésima posición**, se necesitan n comparaciones.

El número promedio de comparaciones es la suma de todas las posibles comparaciones dividida por el número de posiciones:

$$\text{Promedio} = \frac{1 + 2 + 3 + \cdots + n}{n}$$

La suma de los primeros n enteros tiene una fórmula conocida: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Sustituyendo esta fórmula en la ecuación del promedio:

$$\text{Promedio} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

1.2.2 Conexión con la Complejidad O(n)

Aunque el caso promedio sea $\frac{n+1}{2}$, en el análisis de complejidad asintótica (notación Big O), los factores constantes y los términos de menor orden se descartan.

La expresión $\frac{n+1}{2}$ se puede reescribir como $\frac{1}{2}n + \frac{1}{2}$. El término dominante es $\frac{1}{2}n$. Al eliminar el coeficiente constante ($\frac{1}{2}$), la complejidad sigue siendo **lineal**, es decir, $O(n)$. Esto significa que, en promedio, el tiempo de ejecución crece linealmente con el tamaño de la entrada.

- **Ventaja**: Su principal mérito es la **universalidad**. No impone precondiciones sobre la estructura o el orden de los datos.
- **Desventaja**: Su escalabilidad es pobre, haciéndolo impracticable para conjuntos de datos de gran magnitud.

1.2.3 Pseudocódigo

```
función busquedaLineal(colección, objetivo):
    para cada elemento en colección:
        si elemento == objetivo:
            devolver posición(elemento)

    devolver no_encontrado
```

1.3 2. Búsqueda Binaria

Este algoritmo representa un cambio de paradigma fundamental, pero introduce un prerrequisito indispensable: la colección de datos debe estar **totalmente ordenada**. Su

estrategia se basa en el principio de **"divide y vencerás"**, reduciendo el espacio de búsqueda a la mitad en cada iteración.

1.3.1 Análisis de Complejidad Temporal: Deducción de O(log n)

La eficiencia de la búsqueda binaria se modela mediante una **relación de recurrencia**. Sea $T(n)$ el tiempo de ejecución del algoritmo para una entrada de tamaño n. En cada paso, se realiza una comparación de coste constante, c, y el problema se reduce a un subproblema de tamaño $n/2$. La recurrencia es $T(n) = T(n/2) + c$.

Resolviendo esta recurrencia, por ejemplo, mediante el Teorema Maestro o sustitución iterativa, se demuestra que el número de pasos es logarítmico. Después de k pasos, el tamaño del problema es $n/2^k$. El algoritmo termina cuando $n/2^k = 1$, lo que implica $k = \log_2(n)$. Por lo tanto, la complejidad temporal es: $T(n) \in O(\log n)$

1.3.2 Pseudocódigo (Versión Iterativa)

```
función busquedaBinaria(array_ordenado, objetivo):
    bajo ← 0
    alto ← longitud(array_ordenado) - 1

    mientras bajo ≤ alto:
        medio ← bajo + piso((alto - bajo) / 2)

        si array_ordenado[medio] == objetivo:
            devolver medio
        sino si array_ordenado[medio] < objetivo:
            bajo ← medio + 1
        sino:
            alto ← medio - 1

    devolver no_encontrado
```

1.4 3. Tablas Hash (Hashing)

Las tablas hash intentan **calcular la posición** de un elemento de forma directa a través de una **función hash**, $h(k)$, que mapea una clave a un índice de un array.

1.4.1 Componentes y Análisis

- **Función Hash**: Debe ser determinista, rápida y distribuir las claves de forma **uniforme** sobre los índices para minimizar colisiones.
- **Gestión de Colisiones**: Dado que $h(k_1) = h(k_2)$ para $k_1 \neq k_2$ es posible, se usan estrategias como el **encadenamiento separado** o el **direccionamiento abierto**.
- **Complejidad**:
  - **Caso Promedio**: Si el **factor de carga** ($\alpha = n/m$) se mantiene constante, la complejidad es $O(1)$ **amortizado**.
  - **Peor Caso**: Todas las claves colisionan en el mismo índice, degradando la búsqueda a $O(n)$.

1.4.2 Pseudocódigo (Búsqueda con Encadenamiento Separado)

```
función buscarEnTablaHash(tabla_hash, clave):
    // Calcular el índice usando la función hash
    índice ← hash(clave) % tamaño(tabla_hash)

    // Obtener la lista enlazada (o bucket) en esa posición
    bucket ← tabla_hash[índice]

    // Buscar linealmente dentro del bucket
    para cada par (k, v) en bucket:
        si k == clave:
            devolver v // Valor encontrado

    devolver no_encontrado
```

1.5 4. Búsqueda en Árboles Binarios de Búsqueda (BST)

Un BST mantiene sus claves ordenadas de forma implícita. Para cualquier nodo x, las claves en su subárbol izquierdo son menores y las del subárbol derecho son mayores. La complejidad depende de la altura del árbol, h. En un **árbol auto-balanceado** (ej. AVL, Rojo-Negro), $h \in O(\log n)$, garantizando una búsqueda eficiente.

1.5.1 Pseudocódigo (Versión Recursiva)

```
función buscarEnBST(nodo, clave_objetivo):
    // Caso base: el subárbol está vacío o encontramos la clave
```

```
    si nodo == null o nodo.clave == clave_objetivo:
        devolver nodo

    // Decisión recursiva
    si clave_objetivo < nodo.clave:
        devolver buscarEnBST(nodo.izquierdo, clave_objetivo)
    sino:
        devolver buscarEnBST(nodo.derecho, clave_objetivo)
```

1.6 5. Búsqueda en Grafos (BFS y DFS)

En grafos, la búsqueda se refiere a algoritmos de recorrido para determinar la existencia de un vértice o un camino hacia él.

- **BFS (Breadth-First Search)**: Explora por niveles adyacentes. Es el algoritmo canónico para encontrar **caminos de mínima longitud** en grafos no ponderados.
- **DFS (Depth-First Search)**: Explora una rama hasta su máxima profundidad antes de retroceder. Útil en **detección de ciclos** y búsqueda exhaustiva.

Ambos algoritmos tienen una complejidad temporal de $O(|V| + |E|)$.

1.6.1 Pseudocódigo (BFS para encontrar un objetivo)

```
función buscarBFS(grafo, origen, objetivo):
    crear cola Q
    crear conjunto de visitados V

    encolar(origen, Q)
    añadir(origen, V)

    mientras Q no esté vacía:
        actual ← desencolar(Q)

        si actual == objetivo:
            devolver encontrado

        para cada vecino de actual en grafo:
            si vecino no está en V:
                añadir(vecino, V)
                encolar(vecino, Q)
```

```
    devolver no_encontrado
```

1.7 6. Comparación de algoritmos de búsqueda

Algoritmo	Datos ordenados	Complejidad	Ventajas	Desvent
Lineal	No	$O(n)$	Simple, universal	Lento e
Binaria	Sí	$O(\log n)$	Muy rápida	Requier
Hash	No	$O(1)$ promedio	Extremadamente rápida	Depend
BST	Sí (orden implícito)	$O(\log n)$ en balanceados	Inserción y borrado dinámicos	Puede d
BFS/DFS	No necesario	$O(V + E)$	Exploran redes y caminos	Más co

---

1.8 7. Aplicaciones

- **Lineal**: búsqueda en pequeños arrays o listas cortas.
- **Binaria**: diccionarios, búsqueda en bases de datos ordenadas.
- **Hash**: tablas de símbolos en compiladores, índices en bases de datos.
- **BST**: diccionarios dinámicos, sistemas de ficheros.
- **BFS/DFS**: rutas en mapas, redes sociales, análisis de grafos biológicos.

1.9 8. Conclusiones

- La elección del algoritmo depende de la **estructura de datos** y del **problema**.
- Lineal es universal pero lenta.
- Binaria aprovecha el orden para lograr $O(\log n)$.
- Hash es rapidísima, ideal para búsquedas directas.
- BST permite colecciones dinámicas ordenadas.
- BFS y DFS extienden la búsqueda a redes y grafos complejos.

1.10 9. Ejercicios de autoevaluación

1. ¿Cuál es la complejidad promedio y peor caso de una búsqueda en tabla hash?
2. ¿Por qué no se puede usar búsqueda binaria en una lista no ordenada?
3. Inserta los valores 10, 20, 5, 15 en un BST y muestra cómo buscarías el valor 15.

4. Aplica BFS en un grafo con vértices A–B–C–D y aristas {A–B, A–C, B–D}. ¿En qué orden se visitan los vértices desde A?
5. Explica en qué contextos preferirías un BST frente a una tabla hash.
6. Diseña un caso práctico donde BFS sea más útil que DFS.

1.11 Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms.* MIT Press.
- Weiss, M. A. *Data Structures and Algorithm Analysis.* Pearson.
- Sedgewick, R., & Wayne, K. *Algorithms.* Addison-Wesley.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. *Data Structures and Algorithms in Java.* Wiley.