Algoritmos de Ordenación

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

July 7, 2025

Descripción de la unidad

Análisis de distintos algoritmos de ordenación, técnicas de solución y eficiencia.

Contents

1 Algoritmos de ordenación

1.1 Definición y Aplicaciones

Ordenar un conjunto de datos significa reorganizarlo en una secuencia específica, ya sea ascendente o descendente, según una relación de orden predefinida. Esta relación de orden define un conjunto totalmente ordenado, donde para cualquier par de elementos, se puede determinar cuál es mayor o menor.

Las aplicaciones de la ordenación son vastas:

- **Bases de datos:** La ordenación es fundamental para indexar datos y acelerar las consultas.
- **Generación de reportes:** Los datos ordenados facilitan la creación de informes claros y concisos.
- **Búsqueda:** La búsqueda de elementos en un conjunto ordenado es mucho más eficiente (e.g., búsqueda binaria).
- **Compresión de datos:** Algunos algoritmos de compresión se benefician de datos ordenados.

Características clave:

- **Datos:** Un conjunto de elementos a ordenar.
- **Relación de orden:** Define cómo se comparan los elementos entre sí.

1.2 Comparación de ADTs

Para comparar ADT, los lenguajes de programación ofrecen mecanismos específicos:

- **Java:** La interfaz `Comparable` permite definir un orden natural para los objetos de una clase. La clase debe implementar el método `compareTo()`, que devuelve un valor negativo, cero o positivo si el objeto es menor, igual o mayor que el objeto con el que se compara.
  Interface `Comparable<T>`
  Interface `Comparator<T>`
- **C#:** La interfaz `IComparable` cumple una función similar a `Comparable` en Java. Las clases que implementan `IComparable` deben proporcionar una implementación del método `CompareTo()`.
  `IComparable<T>` Interface
  `IComparer<T>` Interface
- **Python:** La comparación de objetos se basa en métodos especiales como `__lt__` (menor que), `__le__` (menor o igual que), `__eq__` (igual a), `__ne__` (no igual a), `__gt__` (mayor que) y `__ge__` (mayor o igual que). Al implementar estos métodos, se define cómo se comparan los objetos de una clase. Por ejemplo, implementar `__le__` permite utilizar el operador `<=` para comparar instancias de la clase.
  Rich comparison methods: `__lt__`, `__le__`, `__gt__`, `__ge__`.
  Unravelling rich comparison operators

1.3 Ordenación por selección

La ordenación por selección funciona encontrando el elemento mínimo en la parte no ordenada del arreglo y luego intercambiándolo con el elemento en la posición actual. En cada iteración, el algoritmo selecciona el elemento más pequeño restante y lo coloca en la posición correcta. Ver algoritmo 1.

Algorithm 1 Ordenación por selección

Require: $T[0 \dots n-1]$ es un vector de objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
1: function SelectionSort(T[0...n-1])
2:     for i ← 0 to n − 2 do
3:         minj ← i
4:         minx ← T[i]
5:         for j ← i + 1 to n − 1 do
6:             if T[j] < minx then
7:                 minj ← j
8:                 minx ← T[j]
9:             end if
10:        end for
11:        T[minj] ← T[i]
12:        T[i] ← minx
13:    end for
14: end function
```

Análisis de la eficiencia de la ordenación por selección:

La ordenación por selección tiene una complejidad temporal de $O(n^2)$ en todos los casos (peor, promedio y mejor). Esto se debe a que siempre realiza dos bucles anidados para encontrar el elemento mínimo y colocarlo en su posición correcta.

Características de la ordenación por selección:

- **Cuadrático en comparaciones:** Realiza un número de comparaciones proporcional a n^2.
- **Lineal en intercambios (accesos al arreglo):** Realiza un número de intercambios proporcional a n. Esto la hace útil cuando los intercambios son costosos.
- **No adaptativo:** Su rendimiento no se ve afectado por el orden inicial de los datos.
- **In-situ:** No requiere memoria adicional.

1.4 Ordenación por inserción

La ordenación por inserción funciona construyendo una sublista ordenada desde el principio del arreglo. En cada iteración, toma un elemento del arreglo no ordenado y lo inserta en la posición correcta dentro de la sublista ordenada. El elemento se compara con sus predecesores y se inserta en la posición donde debe estar para mantener el orden. Ver algoritmo 2.

Análisis de la eficiencia de la ordenación por inserción:

- **Peor caso:** $O(n^2)$. Ocurre cuando el arreglo está ordenado en orden inverso.

Algorithm 2 Ordenación por inserción

Require: $T[0 \dots n-1]$ es un vector objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
1: function InsertionSort(T[0...n-1])
2:     for i ← 1 to n-1 do
3:         x ← T[i]
4:         j ← i-1
5:         while j > 0 and x < T[j] do
6:             T[j+1] ← T[j]          ▷ Los elementos mayores a x se desplazan a la derecha
7:             j ← j-1
8:         end while
9:         T[j+1] ← x
10:    end for
11: end function
```

- **Mejor caso:** $O(n)$. Ocurre cuando el arreglo ya está ordenado. En este caso, solo realiza una comparación en cada iteración del bucle principal.
- **Caso promedio:** $O(n^2)$.

Características de la ordenación por inserción:

- **Adaptativo:** Su rendimiento mejora si el arreglo está parcialmente ordenado.
- **Estable:** Preserva el orden relativo de los elementos con claves iguales.
- **In-situ:** No requiere memoria adicional.
- **Simple de implementar:** Es un algoritmo relativamente fácil de entender e implementar.

1.5 Ordenación shellsort

Shellsort es una generalización de la ordenación por inserción que permite el intercambio de elementos que están lejos. La idea es organizar los elementos del arreglo de tal manera que, comenzando con un gran tamaño de salto, todos los elementos separados por ese tamaño de salto estén ordenados. Luego, el tamaño del salto se reduce para ordenar los elementos en grupos más pequeños. A medida que el tamaño del salto final se reduce a 1, el ordenamiento se convierte esencialmente en un ordenamiento por inserción, pero para entonces, el arreglo ya está parcialmente ordenado, lo que hace que el ordenamiento sea más eficiente. Ver algoritmo 3.

Análisis de la eficiencia de la ordenación Shellsort:

El análisis de la complejidad temporal de Shellsort es complejo y depende de la secuencia de incrementos utilizada. No se conoce una fórmula exacta para su complejidad en todos los casos.

- **Complejidad empírica:** En la práctica, Shellsort muestra un rendimiento significativamente mejor que la ordenación por selección e inserción, especialmente para arreglos de tamaño mediano. Con la secuencia de incrementos original de Shell $(n/2, n/4, ..., 1)$, la complejidad es $O(n^2)$. Con otras secuencias de incrementos, se puede lograr una complejidad de $O(n^{3/2})$ o incluso mejor.

Algorithm 3 Ordenación shellsort

Require: $T[0 \dots n-1]$ es un vector objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
1: function ShellSort(T[0...n-1])
2:     h ← 1
3:     while h < N/3 do
4:         h ← 3h + 1
5:     end while
6:     while h ≥ 1 do
7:         for i ← h to n − 1 do
8:             x ← T[i]
9:             j ← i − h
10:            while j ≥ 0 and x < T[j] do
11:                T[j + h] ← T[j]          ▷ Los elementos mayores a x se desplazan a la derecha
12:                j ← j − h
13:            end while
14:            T[j + h] ← x
15:        end for
16:        h ← h/3
17:    end while
18: end function
```

Características de la ordenación Shellsort:

- **No estable:** No preserva el orden relativo de los elementos con claves iguales.
- **In-situ:** No requiere memoria adicional.
- **Adaptativo:** Su rendimiento puede variar dependiendo del orden inicial de los datos y de la secuencia de incrementos utilizada.

1.6 Ordenación por fusión (mergesort)

Mergesort es un algoritmo de ordenación basado en la técnica de divide y vencerás. Divide el vector en mitades recursivamente hasta que cada subvector contenga un solo elemento (que se considera ordenado). Luego, fusiona (merge) las mitades ordenadas para obtener un vector ordenado más grande. Este proceso de fusión continúa hasta que se obtiene el vector completo ordenado. Ver algoritmo 4.

Análisis de la eficiencia de la ordenación por fusión:

La ordenación por fusión tiene una complejidad temporal de $O(n \log n)$ en todos los casos (peor, promedio y mejor). Esto se debe a que divide el arreglo en mitades recursivamente hasta que cada subarreglo contiene un solo elemento, y luego fusiona los subarreglos ordenados en un solo arreglo ordenado.

Características de la ordenación por fusión:

- **Complejidad temporal:** $O(n \log n)$.

Algorithm 4 Ordenación por fusión

Require: $T[0 \dots n-1]$ es un vector objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
1: function MERGE(a[0...n-1], aux[0...n-1], lo, mid, hi)
2:     for k ← lo to hi do
3:         aux[k] ← a[k]
4:     end for
5:     i ← lo
6:     j ← mid + 1
7:     for k ← low to hi do
8:         if i > mid then
9:             a[k] ← aux[j]
10:            j ← j + 1
11:        else if j > hi then
12:            a[k] ← aux[i]
13:            i ← i + 1
14:        else if aux[j] < aux[i] then
15:            a[k] ← aux[j]
16:            j ← j + 1
17:        else
18:            a[k] ← aux[i]
19:            i ← i + 1
20:        end if
21:    end for
22: end function

23: function MERGESORT(T[0...n-1])
24:    if n ≤ n_0 then
25:        ahdoc(T)
26:    else
27:        U ← T[0...⌊n/2⌋]
28:        V ← T[⌊n/2⌋ + 1...n-1]
29:        MERGESORT(U)
30:        MERGESORT(V)
31:        T ← MERGE(U, V)
32:    end if
33: end function
```

- **Complejidad espacial:** $O(n)$. Requiere memoria adicional para el arreglo auxiliar utilizado en la fusión.
- **Estable:** Preserva el orden relativo de los elementos con claves iguales.
- **No adaptativo:** Su rendimiento no se ve afectado por el orden inicial de los datos.

1.7 Ordenación rápida (quicksort)

Quicksort es un algoritmo de ordenación muy eficiente que también utiliza la técnica de divide y vencerás. Selecciona un elemento del vector como "pivote" y luego particiona el vector en dos subvectores: uno con elementos menores que el pivote y otro con elementos mayores que el pivote. El pivote queda en su posición final ordenada. Los subvectores se ordenan recursivamente. La eficiencia de Quicksort depende en gran medida de la elección del pivote. Ver el algoritmo 5.

Análisis de la eficiencia de la ordenación rápida:
La eficiencia de Quicksort depende en gran medida de la elección del pivote.

- **Peor caso:** $O(n^2)$. Ocurre cuando el pivote es siempre el elemento más pequeño o el más grande del arreglo. En este caso, la partición divide el arreglo en un subarreglo de tamaño 0 y otro de tamaño $n-1$, lo que lleva a una recursión de profundidad n.
- **Mejor caso:** $O(n \log n)$. Ocurre cuando el pivote divide el arreglo en dos subarreglos de tamaño aproximadamente igual. En este caso, la recursión tiene una profundidad de $\log n$.
- **Caso promedio:** $O(n \log n)$. Con una buena elección de pivote (e.g., elegir un elemento aleatorio), Quicksort tiene un rendimiento promedio muy bueno.

Características de la ordenación rápida:

- **In-situ:** No requiere memoria adicional (aparte de la pila de recursión).
- **No estable:** No preserva el orden relativo de los elementos con claves iguales.
- **Adaptativo:** Su rendimiento puede variar dependiendo del orden inicial de los datos y de la elección del pivote.
- **Muy eficiente en la práctica:** A pesar de su complejidad en el peor caso, Quicksort es uno de los algoritmos de ordenación más rápidos en la práctica, especialmente para arreglos grandes.

Algorithm 5 Ordenación rápida

Require: $T[0 \ldots n-1]$ es un vector objetos comparables
Ensure: $T[0 \ldots n-1]$ el vector de entrada ordenado ascendentemente

```
1: function PARTITION(a[0...n-1], lo, hi)
2:     i ← lo
3:     j ← hi + 1
4:     v ← a[lo]
5:     while True do
6:         while a[i] < v do
7:             i ← i + 1
8:             if i = hi then
9:                 break
10:            end if
11:        end while
12:        while v < a[j] do
13:            j ← j − 1
14:            if j = lo then
15:                break
16:            end if
17:        end while
18:        if i ≥ j then
19:            break
20:        end if
21:        exchange a[i] ↔ a[j]
22:    end while
23:    exchange a[lo] ↔ a[j]
24:    return j
25: end function

26: function QUICKSORT(a[0...n-1], lo, hi)
27:    if hi ≤ lo then
28:        return
29:    end if
30:    j ← PARTITION(a,lo,hi)
31:    QUICKSORT(a,lo,j − 1)
32:    QUICKSORT(a,j + 1,hi)
33: end function
```

2 Análisis por medio de árbol de llamados recursivos

El análisis por medio de árbol de llamados recursivos es una técnica que permite visualizar y analizar el costo de un algoritmo recursivo. Se construye un árbol donde cada nodo representa una llamada a la función recursiva. La raíz del árbol representa la llamada inicial, y los hijos de cada nodo representan las llamadas recursivas realizadas por ese nodo.

Para analizar el costo del algoritmo, se contabiliza la operación seleccionada como modelo de costo para el algoritmo en cada nodo del árbol. El costo total del algoritmo es la suma de los costos de todos los nodos del árbol.

Ejemplo Árbol de llamados recursivos para el algoritmo mergesort

Consideremos el algoritmo mergesort para ordenar una lista de n elementos. El árbol de llamadas recursivas tendrá una estructura donde cada nivel representa una división del arreglo en dos mitades. En cada nodo, los modelos de costo dominante son la comparación de elementos y los accesos al arreglo en los llamados a la operación `merge`. En la figura 1 se ilustra el cálculo del número de comparaciones de `mergesort` usando el árbol de llamados recursivos.

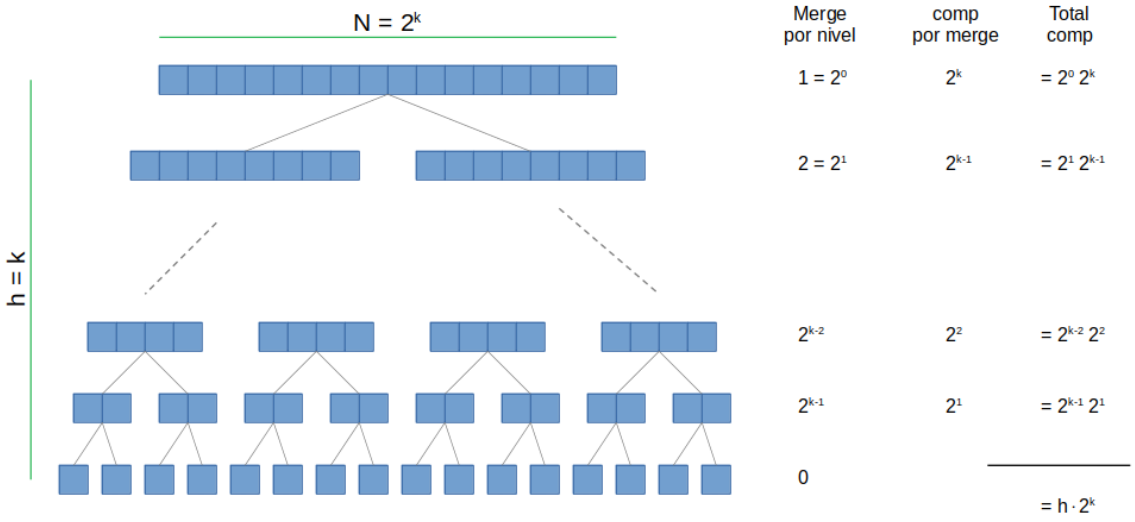

Figure 1: Árbol de invocaciones recursivas para `mergesort`

Limitaciones de esta metodología: Esta metodología puede ser compleja de aplicar si el árbol de llamadas no sigue un patrón regular, o si el costo de las operaciones en cada nodo varía significativamente. La figura 2 ilustra una posible secuencia de llamadas recursivas para `quicksort`, mostrando las irregularidades del árbol.

3 Análisis por medio de recurrencias

El análisis por medio de recurrencias es una técnica que permite describir el costo de un algoritmo recursivo en función del tamaño de la entrada. Una recurrencia es una ecuación que define una función en términos de sí misma, con uno o más casos base.

La recurrencia describe la cantidad de operaciones (modelo de costo) en función de una invocación del algoritmo más los llamados recursivos que se hacen.

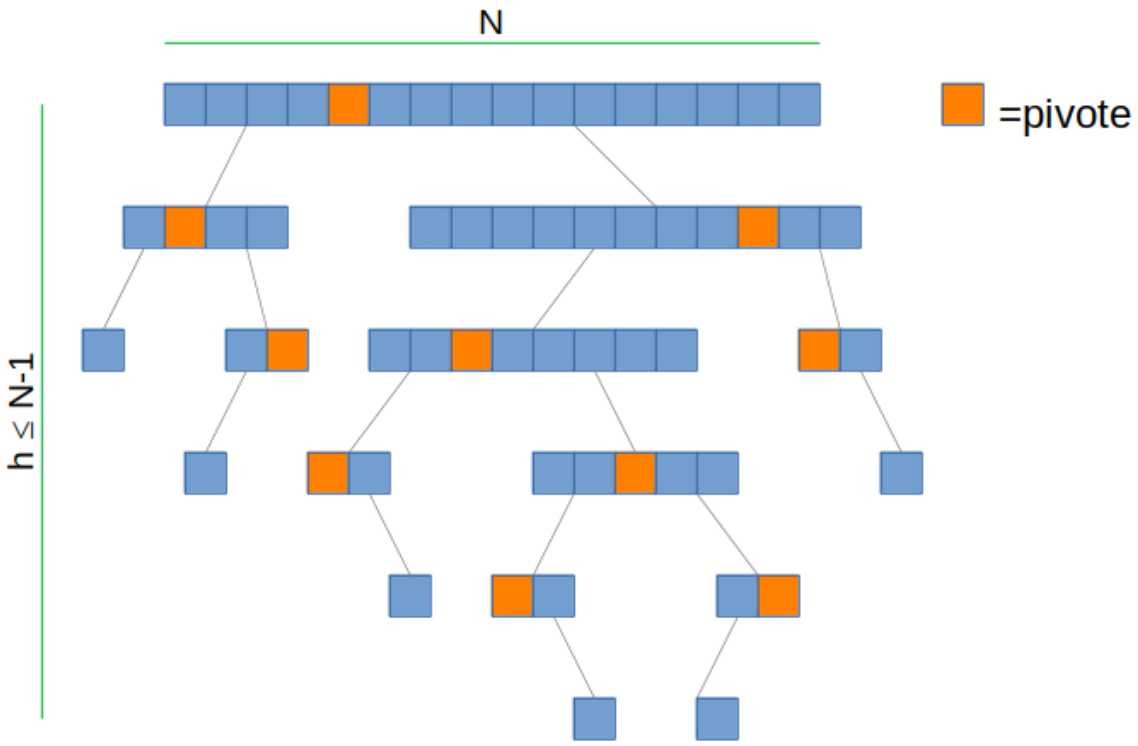

Figure 2: Árbol de invocaciones recursivas para `quicksort`

Método de sustitución Este método permite "resolver" la recurrencia, es decir, encontrar una fórmula no recursiva para la misma función. El método de sustitución implica desarrollar la recurrencia hacia abajo hasta el caso base, y luego sustituir las respuestas del caso anterior hasta obtener la respuesta del llamado inicial.

Ejemplo Análisis por medio de recurrencias para el algoritmo `mergesort`

Se puede definir la funcion $C(N)$ como el número de comparaciones que hace `mergesort` en un intervalo de tamaño N. Se observa que al invocar el algoritmo en el intervalo de tamaño N este se subdivide en dos intervalos de tamaños $\lceil N/2 \rceil$ y $\lfloor N/2 \rfloor$, para los cuales se haran $C(\lceil N/2 \rceil)$ y $C(\lfloor N/2 \rfloor)$ comparaciones respectivamente. Considerando además que en el llamado a `merge` se hacen como máximo N comparaciones, se obtiene la recurrecia:

$$C(N) = N + C\left(\left\lceil \frac{N}{2} \right\rceil\right) + C\left(\left\lfloor \frac{N}{2} \right\rfloor\right)$$

Haciendo el cambio de variable $N = 2^H$ y agrupando se obtiene:

$$C(2^H) = 2^H + 2C\left(2^{H-1}\right)$$

Que se puede resolver aplicando el método de sustitución, y teniendo presente que en el caso base del algoritmo se hacen 0 comparaciones, es decir, $C(1) = 0$.

$$
\begin{aligned}
C(2^H) &= 2^H + 2C\left(2^{H-1}\right) \\
C(2^{H-1}) &= 2^{H-1} + 2C\left(2^{H-2}\right) \\
&\vdots \\
C(2^1) &= 2^1 + 2C\left(2^{H-H}\right) = 2^1 \\
C(2^2) &= 2^2 + 2C\left(2^1\right) = 2^2 + 2(2^1) \\
C(2^3) &= 2^3 + 2C\left(2^2\right) = 2^3 + 2(2^2 + 2^2) \\
&\vdots \\
C(2^H) &= 2^H + 2(2^{H-1} + \ldots + 2^{H-1}) = \underbrace{2^H + \ldots + 2^H}_{\text{H veces}} \\
C(2^H) &= 2^H \cdot H \\
C(N) &= N\lceil \lg(N) \rceil
\end{aligned}
$$

Ejemplo Análisis por medio de recurrencias para el algoritmo `quicksort`

El análisis de Quicksort por medio de recurrencias es más complejo debido a que el tamaño de los subproblemas depende de la elección del pivote. En el mejor de los casos, el pivote divide el arreglo en dos subarreglos de tamaño aproximadamente igual. En el peor de los casos, el pivote resulta ser el elemento más pequeño o el más grande, lo que lleva a un subarreglo de tamaño 0 y otro de tamaño $n-1$.

Peor caso En el peor caso, la recurrencia para el número de comparaciones $C(n)$ es:

$$C(n) = C(n-1) + n$$

Esto se debe a que, en cada paso, se realiza una partición que requiere n comparaciones y se genera un subproblema de tamaño $n-1$. Desarrollando la recurrencia:

$$
\begin{aligned}
C(n) &= C(n-1) + n \\
&= C(n-2) + (n-1) + n \\
&= C(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= C(1) + 2 + 3 + \ldots + n
\end{aligned}
$$

Dado que $C(1) = 0$, la suma resultante es la suma de los primeros n enteros menos 1:

$$C(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

Mejor caso En el mejor caso, cada partición divide el arreglo en dos subarreglos de tamaño aproximadamente $n/2$. La recurrencia para el número de comparaciones $C(n)$ es:

$$C(n) = 2C(n/2) + n$$

Esta recurrencia es similar a la de Mergesort. Aplicando el teorema maestro, con $a = 2$, $b = 2$, y $f(n) = n$, se tiene que $n^{\log_b a} = n^{\log_2 2} = n$. Como $f(n) = \Theta(n)$, estamos en el Caso 2 del teorema maestro, y la solución es $C(n) = \Theta(n \lg n)$.

Caso promedio El análisis del caso promedio es más intrincado, pero se puede demostrar que también tiene un costo de $\Theta(n \ln n)$. La recurrencia para el caso promedio es:

$$C(n) = n + \frac{1}{n}\sum_{i=1}^{n}(C(i-1) + C(n-i))$$

Esta recurrencia refleja que cada posible pivote tiene la misma probabilidad de ser elegido, y se suman los costos de las dos subllamadas recursivas resultantes. La solución de esta recurrencia es $C(n) = \Theta(n \ln n)$.

4 Teorema maestro

El teorema maestro proporciona una solución directa para recurrencias de la forma $T(n) = aT(n/b) + f(n)$, donde $a \geq 1$ y $b > 1$ son constantes, y $f(n)$ es una función asintóticamente positiva. Este teorema es útil para determinar el orden de crecimiento de algoritmos recursivos.

Expresión para el teorema maestro y distintos casos de aplicación:

El teorema maestro establece que, dependiendo de la relación entre $f(n)$ y $n^{\log_b a}$, se pueden determinar tres casos:

- Caso 1: Si $f(n) = O(n^{\log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_b a})$.
- Caso 2: Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$.
- Caso 3: Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante $\epsilon > 0$, y si $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y n suficientemente grande, entonces $T(n) = \Theta(f(n))$.

Ejemplo: Aplicación del teorema maestro para el algoritmo `mergesort`

Para el algoritmo `mergesort`, la recurrencia es $C(n) = 2C(n/2) + O(n)$. Aquí, $a = 2$, $b = 2$, y $f(n) = O(n)$. Por lo tanto, $n^{\log_b a} = n^{\log_2 2} = n$. Como $f(n) = \Theta(n)$, estamos en el Caso 2 del teorema maestro, y la solución es $C(n) = \Theta(n \log n)$.

Limitaciones del teorema maestro: El teorema maestro no se puede aplicar a todas las recurrencias. Tiene limitaciones en cuanto a la forma de la recurrencia y las condiciones que deben cumplir la función $f(n)$ y las constantes a, b.

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms*. Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy*. CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.