Capítulo 3

Divide y vencerás

El planteamiento "divide y vencerás" es una estrategia para resolver problemas complejos dividiéndolos en problemas más pequeños y manejables. La idea es bastante intuitiva: en lugar de afrontar el problema completo, se divide en partes más pequeñas, hasta el punto en que su resolución puede ser trivial, y luego se combinan las soluciones de esas partes para obtener la solución final. A pesar de su simpleza conceptual, este enfoque es muy potente porque permite abordar una gran cantidad de problemas, no sólo computacionales.

3.1. Esquema general

Los algoritmos de divide y vencerás se basan en tres etapas principales:

- **División:** Dividir el problema original en subproblemas más pequeños que son instancias del mismo tipo de problema.
- **Resolución:** Cuando un subproblema es lo suficientemente pequeño, se resuelve de forma directa.
- **Combinación:** Se combinan las soluciones de los subproblemas para obtener la solución del problema original.

No todas estas etapas se utilizan en cualquier problema. Como veremos más adelante, en algunos casos, alguna de las etapas no es necesaria o permanece implícita en la solución.

La mayoría de los enfoques computacionales basados en divide y vencerás se plantean mediante recursividad, ya que permite abstraer más fácilmente las distintas etapas. Un ejemplo genérico siguiendo este principio sería:

```
función dyv(n)
    si trivial(n)
        devuelve resolver(n)

    subproblemas := división(n)
    para cada subproblema en subproblemas
        soluciones += dyv(subproblema)

    devuelve combinar(soluciones)
```

donde `n` representa, de manera genérica, la entrada (haciendo referencia a la talla) del problema.

3.1.1. Ejemplo: búsqueda binaria

Vamos a identificar los distintos pasos del esquema de "divide y vencerás" con la ya conocida búsqueda binaria. Como ya vimos en el capítulo anterior, el algoritmo funciona dividiendo repetidamente el rango de búsqueda a la mitad, hasta encontrar el valor objetivo o determinar que no está presente. Recordemos su pseudocódigo:

```
función búsqueda_binaria(v, z):
    si |v| = 0:
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v[m] = z
        devuelve m
    si v[m] > z
        devuelve búsqueda_binaria(v[:m], z)
    si_no
        devuelve búsqueda_binaria(v[m:], z)
```

En este algoritmo, podemos identificar las tres etapas mencionados anteriormente:

- **División**. Se realiza al calcular el índice medio (`m`) y dividir la lista en dos partes: la parte izquierda y la parte derecha.
- **Resolución**. El problema es trivial cuando la lista tiene tamaño cero (`|v| = 0`) o el elemento objetivo está en la posición central (`si v[m] = z`).

- **Combinar**. Este paso es más o menos implícito, ya que el subproblema resuelto proporciona directamente la solución final. En este caso, la combinación sería simplemente propagar la instrucción `devuelve` en la pila de llamadas recursivas.

La complejidad temporal de la búsqueda binaria corresponde a la relación de recurrencia $T(n) = T(\frac{n}{2}) + O(1)$, donde n es el número de elementos en el vector. Como ya vimos en el capítulo anterior, su complejidad es del orden $\mathcal{O}(\log n)$.

3.2. Algoritmos de ordenación

Los algoritmos de ordenación son de vital importancia en multitud de aplicaciones. Es por ello que a lo largo de los años se han realizado numerosas propuestas para encontrar algoritmos eficientes de ordenación. En este capítulo vamos a explorar dos de los algoritmos de ordenación más conocidos y eficientes, que siguen el esquema de divide y vencerás: MergeSort y QuickSort.

3.2.1. MergeSort: Ordenación por mezcla

La "ordenación por mezcla", más conocida como MergeSort, es un algoritmo de ordenación que utiliza el principio de divide y vencerás. Como veremos después, MergeSort garantiza una complejidad temporal del orden $\mathcal{O}(n \log n)$ en todos los casos.

MergeSort se basa en el principio de que es más fácil construir una lista ordenada a partir de dos sublistas ya ordenadas. Para ello, el algoritmo divide la lista a ordenar recursivamente hasta que llegamos a sublistas cuya ordenación es trivial (listas con máximo un elemento). Después, siguiendo el orden de las llamadas recursivas, va mezclando (operación `mezclar`) las sublistas ordenadas hasta obtener la lista original ordenada.

Un pseudocódigo del algoritmo sería el siguiente:

```
función mergesort(v):
    si |v| <= 1
        devuelve v
    medio = |arr| / 2
    izquierda = mergesort(arr[:medio])
    derecha = mergesort(arr[medio:])
    return mezclar(izquierda, derecha)
```

Con carácter general, podemos identificar las etapas que componen el esquema de divide y vencerás:

1. **División.** Dividir la lista en dos mitades iguales.
2. **Resolución.** Cuando la lista es de tamaño menor o igual a 1, ya está ordenada.
3. **Combinación.** Combinar las dos mitades ordenadas en una sola lista ordenada, mediante la operación *mezclar*.

En el pseudocódigo anterior, hemos utilizado una función de mezcla (`mezclar`). Esta función asume que se reciben dos listas ya ordenadas y devuelve la unión de éstas, también ordenada. Esta función tendría un pseudocódigo como el siguiente:

```
función mezclar(a, b):
    resultado = []
    i, j := 0, 0
    mientras i < |a| y j < |b|
        si a[i] < b[j]
            resultado += a[i]
            i += 1
        else:
            resultado += b[j]
            j += 1
    resultado += a[i:]
    resultado += b[j:]
    devuelve resultado
```

Básicamente, la función `mezclar` va construyendo la lista a devolver (*resultado*) concatenando en cada momento el siguiente menor número de las listas de entrada. Esto se implementa eficientemente manteniendo un índice para cada lista (i, j), que apuntan a la siguiente posición a comparar, respectivamente. Cuando alguno de los índices sobrepasa el tamaño de la lista correspondiente, se concatenan los elementos restantes de la otra lista, que ya estará ordenada (condición del algoritmo).

MergeSort no tiene mejor o peor caso, dado que ni el número de llamadas ni las operaciones a ejecutar dependen del contenido de la lista. Teniendo en cuenta que la operación `mezclar` tiene un coste del orden $\mathcal{O}(n)$, la relación de recurrencia de MergeSort se define como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + 2T(\frac{n}{2}) & \text{si } n > 1 \end{cases} \tag{3.1}$$

Esta relación de recurrencia indica un coste perteneciente a $\mathcal{O}(n \log n)$.

3.2.2. Quicksort: Ordenación rápida

La "ordenación rápida", más conocida como *QuickSort*, es un algoritmo de ordenación eficiente que sigue el paradigma de Divide y Vencerás. Este algoritmo divide una lista en dos partes a partir de un elemento "pivote". Este elemento es uno de los valores de la lista y sirve para organizar el resto de elementos en dos listas disjuntas: una para los elementos que son mayores que el pivote y otra para los elementos que son menores que el pivote. Una vez posicionado el pivote en el lugar que le corresponde en la lista ordenada final, QuickSort llama recursivamente para ordenar las dos (sub)listas restantes.

Siguiendo el paradigma de este capítulo, podemos describir el proceso de QuickSort mediante los siguientes pasos:

1. **División:** Elegir un elemento como "pivote" y dividir el vector de manera que todos los elementos menores que el pivote queden a un lado de la lista y los mayores al otro. Esto posiciona el pivote en el lugar que le corresponde en la lista final. A continuación, se ordenan ambos lados de la lista mediante llamadas recursivas.

2. **Resolución:** La resolución es implícita cuando la entrada es un vector de tamaño 1.

3. **Combinación:** Tras la llamada recursiva, el vector queda ordenado puesto que el elemento pivote está en la posición que le corresponde y las otras dos partes se han ordenado recursivamente.

Vamos a asumir por ahora que el pivote se elige aleatoriamente de entre todos los elementos del vector (abajo se comentará más sobre esta decisión). Siendo así, un pseudocódigo para QuickSort sería el siguiente:

```
función qsort(v):
    si |v| <= 1
        devuelve v
    si no
        pivote := aleatorio(1,|v|)
```

```
    izq, der := partición(v,pivote)
    devuelve [qsort(izq), pivote, qsort(der)]
```

El elemento pivote se utiliza para la función `partición`, la cual devuelve dos listas con los valores menores y mayores que el pivote, respectivamente.[1] Esta función tendría un coste perteneciente a $\mathcal{O}(n)$, ya que únicamente tendría que recorrer la lista una vez para dividirla en los dos conjuntos correspondientes.

QuickSort tiene una complejidad temporal diferente según la lista a ordenar. Es decir, sí presenta un mejor y peor caso. El mejor caso ocurre cuando el pivote elegido es siempre la mediana de los elementos a ordenar, lo que permite dividir la lista en dos partes iguales en cada nivel de recursión. Esto conlleva una relación de recurrencia tal que:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + 2T(\frac{n}{2}) & \text{si } n > 1 \end{cases} \tag{3.2}$$

Este caso arroja una complejidad de orden $\mathcal{O}(n \log n)$. Sin embargo, el peor caso, que ocurre cuando el pivote es siempre el elemento o más grande o más pequeño (casos degenerados), lleva a una relación de recurrencia tal que:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + T(1) + T(n-1) & \text{si } n > 1 \end{cases} \tag{3.3}$$

Esto se debe a que en cada paso sólo se reduce el tamaño de la lista en uno, lo que genera una profundidad de recursión máxima, que sumado a que los pasos no recursivos tienen complejidad $\mathcal{O}(n)$, lleva a una complejidad temporal perteneciente a $\mathcal{O}(n^2)$.

Elección del pivote

Como acabamos de ver, la elección del pivote es crucial para el rendimiento de QuickSort. En la práctica, se selecciona el pivote de forma aleatoria o casi-aleatoria (por ejemplo, utilizando estrategias como el "mediana de N", que elige el elemento mediana de N elementos seleccionados al azar). Estadísticamente, esto lleva a un caso promedio con una complejidad esperada de $\mathcal{O}(n \log n)$.[2]

[1] Una de las dos listas incluirá los elementos iguales al propio pivote (sin incluirlo). Cuál de las dos lo haga es irrelevante para el algoritmo.

[2] Como vimos en el Capítulo 2, el caso promedio se refiere a la esperanza (matemática) del coste para una instancia cualquiera, y no al promedio entre el mejor y el peor caso.

También es posible forzar la búsqueda de la mediana exacta en cada partición, lo cual garantiza siempre el mejor caso. Sin embargo, encontrar la mediana exacta tiene un coste adicional de $\mathcal{O}(n)$. Asintóticamente, esto asegura siempre el mejor caso pero puede no ser eficiente en muchas aplicaciones.

En resumen, aunque existen métodos para mejorar la elección del pivote, QuickSort es generalmente eficiente con estrategias de pivote simples.

3.3. Consideraciones adicionales

Para poder aplicar el esquema de Divide y Vencerás, se deben identificar una serie de requisitos. El primero es encontrar una forma de descomponer un problema en partes más pequeñas (reduciendo la talla) y que cada descomposición acerque el problema a un punto en el cual se pueda resolver de manera directa. Además, es necesario que exista una forma de combinar las soluciones de los subproblemas de manera que se obtenga la solución del problema original.

Aunque, a priori, la idea de Divide y Vencerás sea muy general, no siempre el problema se puede descomponer o dicha descomposición no implica necesariamente que nos acerquemos a poder resolverlo de manera directa. Del mismo modo, no siempre es posible resolver el problema original a partir de la resolución de los subproblemas.

Un ejemplo clásico de un problema que no es fácilmente resoluble mediante el esquema de Divide y Vencerás es el "Problema del viajante de comercio". Este problema consiste en encontrar el camino más corto que permite a un viajante visitar un conjunto de ciudades, pasando por cada una de ellas exactamente una vez y regresando al punto de partida. En este caso, es imposible dividir el problema en subproblemas sin que haya interdependencias entre estos subproblemas. Es decir, que resolver los subproblemas independientemente y luego combinarlos no garantiza la solución óptima. Veremos este problema en profundidad en el Capítulo 6 (Vuelta atrás).