Recursividad

Recursión es, en ciencias de computación, una forma de atajar y solventar problemas. De hecho, recursión es una de las ideas centrales de ciencia de computación. Resolver un problema mediante recursión significa que la solución depende de las soluciones de pequeñas instancias del mismo problema.

El poder de la recursión evidentemente se fundamenta en la posibilidad de definir un conjunto infinito de objetos con una declaración finita. Igualmente, un número infinito de operaciones computacionales puede describirse con un programa recursivo finito, incluso en el caso de que este programa no contenga repeticiones explícitas.

La mayoría de los lenguajes de programación dan soporte a la recursión permitiendo a una función llamarse a sí misma desde el texto del programa. Los lenguajes imperativos definen las estructuras de loops como while y for que son usadas para realizar tareas repetitivas. Algunos lenguajes de programación funcionales no definen estructuras de loops sino que posibilitan la recursión llamando código de forma repetitiva. La teoría de la computabilidad ha demostrado que estos dos tipos de lenguajes son matemáticamente equivalentes, es decir que pueden resolver los mismos tipos de problemas, aunque los lenguajes funcionales carezcan de las típicas estructuras while y for.

Algoritmos recursivos

Es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden N, cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que N. De esta forma, al ir reduciendo progresivamente la complejidad del problema que resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva). En esa situación diremos que estamos ante un caso base de la recursividad.

Las claves para construir un subprograma recurrente son:

Cada llamada recurrente se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).
Ha de existir al menos un caso base para evitar que la recurrencia sea infinita.
Es frecuente que los algoritmos recurrentes sean más ineficientes en tiempo que los iterativos aunque suelen ser mucho más breves en espacio.

Un método frecuente para simplificar es dividir un problema en problemas derivados de menor tamaño del mismo tipo. Esto se conoce como dialecting. Como técnica de programación se denomina divide y vencerás y es pieza fundamental para el diseño de muchos algoritmos de importancia, así como parte esencial de la programación dinámica.

Ejemplos de subrutinas definidas recursivamente (recursión generativa)
Un ejemplo clásico de una subrutina recursiva es la función usada para calcular el factorial de un entero.

Definición de la función:

$$\text{fact}(n) = \begin{cases} si & n = 0 & \longrightarrow & 1 \\ si & n > 0 & \longrightarrow & n \cdot \text{fact}(n-1) \end{cases}$$

Pseudocódigo (recursivo):

```
función factorial:

input: entero n de forma que n >= 0

output: [n × (n-1) × (n-2) × … × 1]


    1. if n es 0, return 1
    2. else, return [ n × factorial(n-1) ]


end factorial
```

Una relación recurrente es una ecuación que relaciona términos posteriores en la secuencia con términos previos.

Relación recurrente de un factorial:

$$b_n = nb_{n-1}$$
$$b_0 = 1$$

Computando la relación recurrente para n = 4:

$$\begin{aligned} b_4 &= 4 * b_3 \\ &= 4 * 3 * b_2 \\ &= 4 * 3 * 2 * b_1 \\ &= 4 * 3 * 2 * 1 * b_0 \\ &= 4 * 3 * 2 * 1 * 1 \\ &= 4 * 3 * 2 * 1 \\ &= 4 * 3 * 2 \\ &= 4 * 6 \\ &= 24 \end{aligned}$$

Referencias

https://es.wikipedia.org/wiki/Recursi%C3%B3n_(ciencias_de_computaci%C3%B3n)