Recursividad a fondo

Entonces la recursividad es una cosa. Es algo muy confuso, sin duda, pero también es una herramienta realmente poderosa, cuando la entendemos bien. El problema, para la mayoría de nosotros, es que no tendemos a pensar en la recursividad en nuestra vida diaria, pero la usamos, más de lo que nos damos cuenta.

Puede ser complicado de entender por definición. Esto es lo que dice Merriam Webster al respecto:

Definición de Recursión:

2: la determinación de una sucesión de elementos (como números o funciones) por operación sobre uno o más elementos precedentes según una regla o fórmula que implica un número finito de pasos.

Así que pasamos una serie de elementos a una operación, aplicamos una regla, y luego volvemos a aplicar esa misma regla al subconjunto resultante, hasta un límite finito. Claro como el barro. Veamos un ejemplo concreto de recursividad, y uno que hacemos sin pensar en ello.

Exponentes como recursividad

¿Recuerdas cuando, al ver un problema de matemáticas que contenía algo así como "3 a la potencia de 7" (a menudo expresado en las calculadoras como 3^7)? Realmente no pensamos en ello, porque en nuestras cabezas leemos que como: 3*3*3*3*3*3*3 (es decir, tres multiplicado por sí mismo siete veces).

Pero es una recursividad, en un abrir y cerrar de ojos. Observa.

```
3^7 = 3 * (3^6)
   = 3 * (3 * (3^5) )
   = 3 * (3 * (3 * (3^4) ) )
   = 3 * (3 * (3 * (3 * (3^3) ) ) )
   = 3 * (3 * (3 * (3 * (3 * (3^2) ) ) ) )
   = 3 * (3 * (3 * (3 * (3 * (3 * (3^1) ) ) ) ) )
   = 3 * (3 * (3 * (3 * (3 * (3 * 3) ) ) ) )
   = 3 * (3 * (3 * (3 * (3 * (9) ) ) ) )
   = 3 * (3 * (3 * (3 * (27) ) ) )
   = 3 * (3 * (3 * (81) ) )
   = 3 * (3 * (243) )
   = 3 * (729)
   = 2187
```

Entonces, ¿qué acaba de pasar? Todo eso pasa, hasta cierto punto, en nuestro cerebro. Puede que no lo formalicemos así, pero ese es el patrón: empezar desde el exterior, simplificando la cosa hasta llegar a la expresión más interna, aplicando la misma funcionalidad cada vez, luego reducir esa expresión más interna y pasarla de vuelta, un nivel cada vez, hasta que tengamos un único valor restante.

¡Esta es una gran aplicación de recursividad! Hablando programáticamente, deberíamos poder ver que podemos simplificar eso un poco. Escribamos una función externa que se parezca MUCHO a lo que hicimos anteriormente: multiplique nuestro número base por una llamada interna 'recursiva'. Llamémosla nthPower:

```
function nthPower(number, exponent){
  return number * nthPower(number, exponent-1);
}
```

Eso es literalmente lo que sucede cada vez que anidamos dentro de nuestro 3 ^ 7: multiplicamos 3 * (3 ^ 6), que es simplemente nuestro número, por nuestro número a un exponente menos.

¡No intente eso en casa!

Si ejecutas esa función ahora, harás explotar tu motor de javascript (ya sea una consola o una caja de arena de código). ¿Por qué? Porque nos perdimos una parte crucial de nuestra definición:

Definición de Recursión:

2: la determinación de una sucesión de elementos (como números o funciones) por operación sobre uno o más elementos precedentes según una regla o fórmula que ***implica un número finito de pasos***.

Se agregó negrita y cursiva para enfatizar. Perdimos el número finito de pasos. Mira, la clave de la recursividad es que, en un punto dado, dejamos de hacer un bucle y comenzamos a SALIR. En algún momento, llegamos a una condición que nos dice que dejemos de anidar y que invirtamos la dirección: necesitamos colapsar todo ese anidamiento.

¿Cuál es ese punto, en nuestro caso? Mira hacia atrás en el bloque largo que describe 3 ^ 7: dejamos de profundizar cuando llegamos a esta línea:

= 3 * (3 * (3 * (3 * (3 * (3 * (3^1))))))

Es decir, cuando nuestro exponente es uno, hemos profundizado tanto como necesitamos. En ese punto, simplemente comenzamos a evaluar la llamada a la función más interna (en este caso, el paréntesis más interno, y lo devolvemos a la siguiente función externa (de nuevo, en este caso, el siguiente paréntesis externo):

= 3 * (3 * (3 * (3 * (3 * (3 * 3)))))

Mirando hacia atrás a nuestra función de nthPower, sabemos que necesitamos una "condición de parada", y sabemos que esa condición de parada es cuando el exponente llega a uno. Añadamos eso:

```
function nthPower(number, exponent){
  if(exponent===1){
    // cualquier número elevado a uno es... ese número.
    // ¡Y nuestro punto de parada!!
    return number;
  } else {
    // Tenemos que seguir profundizando, reduciendo el exponente cada vez.
    return number * nthPower(number, exponent-1);
  }
}
```

Así es como se vería esa función, en una pila de ejecución hipotética:

```
// El exponente no es 1 todavía, así que
nthPower(3, 7) = (3 * nthPower(3, 6) )
// Todavía no es uno, sigue profundizando hasta que esté.
nthPower(3, 7) = (3 * (3 * nthPower(3, 5) ) )
//... y sigue entrando hasta que lleguemos a
nthPower(3,7) = (3 * (3 * (3 * (3 * (3 * ( 3 * nthPower(3, 1) ) ) ) ) ) )
// En este punto, el exponente es uno, así que pasamos nuestro número a la
última función:
nthPower(3,7) = (3 * (3 * (3 * (3 * (3 * ( 3 * 3 ) ) ) ) ) )
// Ahora ten en cuenta que cada paréntesis era una llamada de función, por lo
que cada vez pasamos el valor del paréntesis interno al anterior:
nthPower(3,7) = (3 * (3 * (3 * (3 * (3 * 9 ) ) ) ) )
nthPower(3,7) = (3 * (3 * (3 * (3 * 27 ) ) ) )
nthPower(3,7) = (3 * (3 * (3 * 81 ) ) )
nthPower(3,7) = (3 * (3 * 243 ) )
nthPower(3,7) = (3 * 729 )
nthPower(3,7) = 2187
```

¿Qué acaba de pasar?

Empezamos sabiendo lo que queríamos: un número elevado a un exponente dado. Podemos ver que se trata de una "operación recursiva", en la que se aplicará la misma serie de pasos una y otra vez, hasta que se cumpla una determinada condición. En nuestro caso, esa condición era cuando el exponente se reduce a uno.

Hemos creado una función que podemos aplicar, y que también se puede aplicar a sí misma. Esa es la base de la recursividad - una función se aplica a sí misma, una y otra vez, y finalmente devolviendo sus valores.

Y, en mi opinión, lo más importante es que examinamos cómo pensamos. En lugar de hacer el trabajo de codificar nuestro proceso recursivo, que es a menudo como lo haremos en nuestra cabeza, analizamos lo que significa la operación, de una manera más abstracta. Haciéndolo así vimos cómo podría convertirse en 'autorreferencial' o recursivo.

Espero que esto ayude a aclarar la idea de la recursividad y aclare el proceso de pensamiento detrás de por qué la recursividad es una herramienta útil para comprender. Por favor comenta o pregunta. ¡Estaré encantado de continuar la conversación!

Edición: Hubo una pregunta acerca de escribir esto como una función de 'fat arrow', usando una convención de ES6 y escribiendo un código mínimo y simplificado. Aquí está la misma función, haciendo exactamente lo mismo, en un abrir y cerrar de ojos:

```
const nthPower = (num, exp) => exp===1? num : nthPower(num, exp-1) );
```

Sip. Lo mismo, lo mismo. Esa línea dice "crear una función que, si exp == 1, devuelva num y si exp! == 1, devuelva nthPower (num, exp-1)". Esto hace **exactamente** lo mismo que nuestra versión más detallada, de una forma resumida al estilo ES6.

Fuente: https://portfoliostuff-parenttobias.codeanyapp.com/2020/01/29/recursion-all-the-way-down/