PRO1: Lista Recursividad

Emma Rollón
© Departamento CS, UPC

13 de noviembre de 2020

Introducción

Antes de resolver algunos ejercicios, recordad que las características que ha de tener una acción/función recursiva es:

- Ha de tener al menos un caso BASE.
- Ha de tener al menos un caso RECURSIVO. Estos casos han de cumplir:
  - La llamada recursiva ha de ser sobre un problema más pequeño.
  - Se ha de llegar siempre a uno de los casos base.

La idea para encontrar el caso recursivo dado un problema original es:

- Resuelve un problema un poco más pequeño que el que te plantean
- Usa ese resultado para solucionar el problema original

La idea para encontrar el caso base es:

- Piensa qué problema original es muy sencillo de resolver ...
- ... teniendo en cuenta que una cascada de llamadas recursivas tienen que acabar llegando siempre a ese problema sencillo

Problema P51222

En este problema tenemos que implementar una función recursiva tal que dado un número natural retorne cuántos dígitos tiene:

```
// Pre: n natural
// Post: retorna el número de dígitos de n
int numero_digitos(int n);
```

Vamos a ver cómo se aplica las ideas para implementar funciones recursivas a este ejercicio:

- Problema original: queremos saber el número de dígitos de, por ejemplo, 468.
- Para hallar el caso recursivo, vamos a suponer que la función numero_digitos está implementada. ¿Cómo podemos utilizar una llamada a esa función con un número diferente a 468 pero que nos ayude a saber el número de dígitos de 468?. Fíjate que el número de dígitos de 468 es 1 más el número de dígitos de 46 (es decir, 468/10). Por tanto,

  `numero_digitos(468) = 1 + numero_digitos(46)`

- Para hallar el caso base, tenemos que pensar para qué número(s) es muy sencillo calcular su número de dígitos y tal que una cascada de llamadas recursivas como la anterior se vayan acercando a ese caso. Cuando se hace una llamada recursiva se quita uno de los dígitos del número original. Por tanto, vamos a llegar a números con 1 único dígito. ¿Es fácil detectar cuándo un número tenga 1 dígito? Sí, la expresión $x < 10$ evalúa a true si es así, y a false en caso contrario.

En notación matemática, la definición de esta función sería:

$$\text{numero\_digitos(n)} = \begin{cases} 1 & \text{si } n < 10 \\ 1 + \text{numero\_digitos(n/10)} & \text{si } n \geq 10 \end{cases}$$

Traducirlo a código C++ es fácil:

```
// Pre: n natural
// Post: retorna el número de dígitos de n
int numero_digitos(int n) {
    if (n < 10) return 1;
    else return 1 + numero_digitos(n/10);
}
```

Problema P96965

Vamos a resolver primero la función:

```
// Pre: n es natural
// Post: retorna la suma de los dígitos de n
int suma_digitos(int n);
```

Igual que antes:

- Problema original: queremos saber la suma de dígitos de, por ejemplo, 468.
- Caso recursivo: suponemos que suma_digitos está implementada y la podemos utilizar para saber la suma de dígitos de 468 (lógicamente suma_digitos(468) no es una opción porque haríamos una llamada con el mismo número original). Fíjate que si supiera la suma de dígitos de 46 (es decir, 468/10) podría fácilmente completarlo para saber la suma de 468. Sólo faltaría sumarle el último dígito de 468 (es decir, 468 %10).

  `suma_digitos(468) = 8 + suma_digitos(46)`

- Caso base: ¿de qué número(s) es fácil saber su suma de dígitos y además se llegue a él/ellos al ir haciendo la llamada recursiva anterior que quita el dígito de menos peso del número original? Sí. Y podemos optar por dos opciones:
  - Si el número sólo tiene un dígito, la suma de sus dígitos es él mismo.
  - Si el número es 0, la suma de sus dígitos es 0.

En notación matemática, las dos opciones serían:

$$\text{suma\_digitos(n)} = \begin{cases} n & \text{si } \ n < 10 \\ n\,\%10 + \text{suma\_digitos(n/10)} & \text{si } \ n \geq 10 \end{cases}$$

$$\text{suma\_digitos(n)} = \begin{cases} 0 & \text{si } \ n = 0 \\ n\,\%10 + \text{suma\_digitos(n/10)} & \text{si } \ n \neq 0 \end{cases}$$

La traducción de las dos opciones a código C++ es:

```
// Pre: n natural
// Post: retorna la suma de dígitos de n
int suma_digitos(int n) {
    if (n < 10) return n;
    else return n%10 + numero_digitos(n/10);
}
```

```
// Pre: n natural
// Post: retorna la suma de dígitos de n
int suma_digitos(int n) {
    if (n == 0) return 0;
    else return n%10 + numero_digitos(n/10);
}
```

Ahora, vamos a solucionar la función reduccion_digitos de forma iterativa:

```
// Pre: n natural
// Post: retorna la reducción de los dígitos de n
int reduccion_digitos(int n) {
    while (n >= 10) {
        n = suma_digitos(n);
    }
    return n;
}
```

Piensa cómo sería su implementación recursiva. Te damos el caso base:

- si $n < 10$, retornar n.

Problema P22467

Tenemos que implementar de forma recursiva la función:

```
// Pre: n natural
// Post: retorna true si n es primo perfecto, false en caso contrario
bool es_primer_perfecte(int n);
```

En este ejercicio vamos a hacer el razonamiento de forma más abstracta con un número n cualquiera. Substituye n por un número concreto para obtener un ejemplo particular.

Según el enunciado, un número n es primo perfecto si la secuencia de números

$$\text{n, suma\_digitos(n), suma\_digitos(suma\_digitos(n)), } \ldots,$$

son números primos. Fíjate que si los números de la subsecuencia:

$$\text{suma\_digitos(n), suma\_digitos(suma\_digitos(n)), } \ldots,$$

son primos, eso quiere decir que el número n' = suma_digitos(n) es primero perfecto. Por tanto, para que que n sea primero perfecto, tiene que cumplir dos condiciones:

- n tiene que ser primero
- suma_digitos(n) tiene que ser primero perfecto

Si suponemos que la función es_primer_perfecte está implementado, los pasos que daremos para saber si n es primero perfecto serán:

1. si es_primer(n) entonces
   1.1 si es_primer_perfecte(suma_digitos(n)) entonces return true
   1.2 si no return false
2. si no return false

Fíjate que en este razonamiento hemos obtenido un caso que no es recursivo: cuando es_primer(n) es falso. En palabras, cuando *encontramos* un número de la secuencia que no es primo, ya sabemos que la secuencia no es la de un número primo perfecto y no es necesario que sigamos analizándola.

¿Necesitamos algún caso base más o con el que hemos descubierto es suficiente? Fíjate que el caso base que hemos descubierto identifica a una secuencia de un número que no es primero perfecto. Por tanto, nos queda identificar cuándo la secuencia sí que es la de un número primero perfecto.

Si nos fijamos en la llamada recursiva, se hace sobre la suma de dígitos de n. Si encadenamos llamadas recursivas, quiere decir que iremos reduciendo progresivamente el valor del número sobre el que hacemos la llamada recursiva. Por ejemplo, cuando el número inicial es 977, la primera llamada recursiva será sobre 23 y la segunda llamada recursiva será sobre 5.

Fíjate que una vez llegamos a tener un número con un único dígito, ya hemos tratado toda la secuencia. Por tanto, cuando $n < 10$, entonces:

1. si es_primer(n) entonces return true
2. si no return false

La primera versión de nuestro código es:

```
// Pre: n natural
// Post: retorna true si n es primo perfecto, false en caso contrario
bool es_primer_perfecte(int n) {
    if (n < 10) {
        if (es_primer(n)) return true;
        else return false;
```

```
    }
    else {
        if (es_primer(n)) {
            if (es_primer_perfecte(suma_digits(n))) return true;
            else return false;
        }
        else return false;
    }
}
```

En esta primera versión identificamos estructuras de condicional que se pueden simplificar:

```
// Pre: n natural
// Post: retorna true si n es primo perfecto, false en caso contrario
bool es_primer_perfecte(int n) {
    if (n < 10) return es_primer(n));
    else return es_primer(n) and es_primer_perfecte(suma_digits(n));
}
```

Una pregunta que nos deberíamos hacer es en qué cambiaría el comportamiento del código si utilizamos la siguiente rama else:

```
    else return es_primer_perfecte(suma_digits(n)) and es_primer(n);
```

Es decir, si cambiamos el orden de comprobación en esa rama del condicional, ¿cambia algo?. La respuesta es que en este caso estaríamos haciendo un *recorrido* sobre la secuencia de números que vamos generando (y no una *búsqueda*). Esto quiere decir que no se pararía de analizar la secuencia cuando uno de sus elementos es no primo. Haz la ejecución de los dos códigos con el ejemplo n inicial 978 para comprobarlo.