Introducción a la recursividad. El problema de las ocho reinas

PERE BOTELLA

Dept. Programació de Computadors
Facultat d'Informàtica de Barcelona

Si el libro de N. Wirth "Algorithms + Data = Programs" (ref. [1]) fuese fácilmente localizable en casi todos los Centros de Cálculo de nuestro país, no tendría ningún sentido escribir este artículo, puesto que en el capítulo 3 trata extensamente, y en forma magistral, el tema que nos ocupa. Pero no es éste el caso. Y, además, está en inglés, lo cual es un "handicap" innegable para muchos profesionales.

De todas formas, sigue sin tener sentido el "reinventar" el tema cuando tengo delante una descripción que, con mis conocimientos, me es imposible mejorar. Queda claro, pues, que este artículo ni inventa ni aporta nada, sino que se limita a traducir y adaptar los párrafos 3.1, 3.2 y 3.5. Hecha esta aclaración, obligada por un mínimo de ética profesional, me excuso de referenciar continuamente los párrafos citados.

El lenguaje que utilizo como vehículo descriptor de los algoritmos que ilustran el texto es el lenguaje PASCAL. Soy consciente de su muy restringida difusión. Aún así, considero que es de los pocos lenguajes (sino el único) que pueden ser "leídos" por cualquier profesional, sin demasiadas dificultades, aparte de adaptarse perfectamente a nuestros propósitos.

Una última consideración, antes de entrar en materia. Es conveniente que el lector esté familiarizado con las construcciones de la programación estructurada (BEGIN-END, IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL, etc.) y con el diseño TOPDOWN (aproximación al problema por sucesivos refinamientos). En caso contrario, quizás encuentre alguna dificultad en la comprensión, pero esperemos que, como efecto lateral, se consiga motivar su atención hacia esta forma de trabajo.

CONCEPTOS BASICOS

Se dice que un objeto (cualquiera) es *recursivo* si consiste parcialmente o está definido en términos de sí mismo.

Supongamos que situamos dos espejos frente a frente, no totalmente paralelos, y asomamos nuestra cabeza entre los dos. Miremos a cualquiera de los dos espejos. Veremos nuestra imagen y la del espejo que tenemos detrás, el cual contiene nuestra imagen (de espaldas) y el primer espejo cuya imagen contiene nuestra imagen (de frente) y el otro espejo,...

Podemos formular la siguiente definición recursiva:

Imagen-que-veo = Mi-imagen junto a un espejo que contiene la Imagen-que-veo.

Veamos un ejemplo más clásico. Se define el factorial de un número entero *n* como el producto de todos los números enteros desde 1 hasta *n*. Es decir, (el símbolo ! indica factorial):

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

Por ejemplo, $4! = 1 \times 2 \times 3 \times 4$ y $3! = 1 \times 2 \times 3$

Pero también, $4! = 3! \times 4$
(y esta definición ya es recursiva). Sabiendo que, por convención, el factorial de cero es 1, podemos dar la definición siguiente de n! :

a) $0! = 1$
b) $n! = n \times (n-1)!$ (para $n>0$)

Precisamente la potencia de la recursividad consiste en definir en forma finita un concepto infinito (nótese la ausencia de puntos suspensivos en las dos definiciones recursivas precedentes).

¿Podemos hacer un programa para calcular el factorial de un número? Centrándonos en la primera definición, nuestro programa podría ser, para calcular el factorial de N:

```
I: = 0;   F: = 1;
WHILE  I < N DO
       BEGIN
                I: = I + 1;
                F: = F * I
       END
```

Pero éste programa no es recursivo. ¿Cómo traducir a lenguaje de programación el concepto de "consistir o estar definido en términos de sí mismo"? Obviamente, utilizando subprogramas (funciones o subrrutinas) que se llaman a sí mismos. Bajo esta óptica, podemos traducir literalmente la definición recursiva de n! a programa:

```
FUNCTION    F (N:INTEGER): INTEGER;
BEGIN
    IF     N > 0     THEN   F: = F (N-1) * N
                     ELSE   F: = 1
END
```

En un programa que aparezca una llamada a la función F tal como F(4), la función se llamará a sí misma 4 veces (F(3), F(2), F(1), F(0)), devolviendo el valor 24, de la forma siguiente:

$$F(4) = 4 \times F(3) = 4 \times 3 \times F(2) = 4 \times 3 \times 2 \times F(1) =$$
$$= 4 \times 3 \times 2 \times 1 \times F(0) = 4 \times 3 \times 2 \times 1 \times 1 =$$
$$= 4 \times 3 \times 2 \times 1 = 4 \times 3 \times 2 = 4 \times 6 = 24$$

Otro ejemplo de cómo una formulación recursiva es directamente programable consiste en la serie de números enteros de Fibonacci, en que cada elemento, de la serie, que se inicia con un cero y un uno, es la suma de los dos precedentes. Así:

f = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Su formulación recursiva:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \ (n > 1)$$

Y su programación:

```
FUNCTION FIB (N: INTEGER): INTEGER.
BEGIN
IF N = 0    THEN FIB: = 0
   ELSE     IF N = 1 THEN FIB: = 1
            ELSE FIB: = FIB (N-1) + FIB (N-2)
END
```

Llegados a este punto, nos aparecen dos tipos de problemas:

a) ¿Qué tipo de lenguajes admiten recursividad? ¿Es efectiva, en cuanto a tiempo y memoria?
b) ¿Qué tipo de problemas admiten solución recursiva? ¿En qué casos hay que usarla?

Vamos a intentar centrarlos brevemente.

De entrada, los dos lenguajes de uso más extendido en nuestro país (FORTRAN en ambientes universitarios y técnicos, y COBOL en las empresas) no admiten, normalmente, la recursividad (excepto alguna versión reciente). Una subrutina FORTRAN que se llama a sí misma provoca un error en tiempo de compilación. El COBOL no dispone de funciones, y tampoco es demasiado cómodo el uso de subprogramas.

Sólo admiten recursividad versiones recientes de FORTRAN (un ejemplo puede encontrarse en [6]) y BASIC, y todos los lenguajes de la familia ALGOL (ALGOL 60, ALGOL W, ALGOL D, ALGOL 68), incluido el PASCAL.
Aunque numéricamente, sean muchos los lenguajes que admiten esta posibilidad, su difusión y utilización es muy restringida.

Por otra parte hay que tener en cuenta que bajo un esquema recursivo subyace un esquema iterativo. Es decir, que, en principio, todo programa recursivo tiene su equivalente iterativo. Y de esta afirmación podemos sacar dos consecuencias:

1. Al igual que en una iteración, hay que preveer un final para evitar entrar en "loop". Es decir, que un programa recursivo R, deberá contener un esquema del tipo IF condición THEN R, y de un mecanismo que varíe la condición. Lo más frecuente es decrementar un contador n que se utiliza como parámetro, tal que R(n) contiene el esquema IF n > 0 THEN R(n-1). En cualquier caso lo normal es que sea más rápido el control de una iteración que el paso de parámetros a un subprograma.

2. No sólo es necesario que la "profundidad" (no de veces que la rutina recursiva se llama a sí misma) sea finita, cómo indica el párrafo anterior, sino pequeña. Cada activación recursiva de la rutina consume una cierta cantidad de memoria ya que los parámetros de la rutina y sus variables locales van apilándose en un "stack" a cada llamada, conservando su estado en un nivel determinado para reasumirlo cuando se vuelve a ese nivel.

Simplificando, digamos que si el área de datos de una rutina recursiva ocupa 1 Kbyte y la profundidad de la llamada es 10, la ocupación real de memoria será de 10 Kbytes.

Nos queda pendiente resolver las preguntas contenidas en b), pero con lo dicho en los párrafos anteriores poco queda que añadir.

¿Qué tipo de problemas admiten solución recursiva? Es obvio: todo problema que admita una formulación recursiva. Lo difícil es, en ciertos casos, dar con esta formulación.

¿En qué casos hay que usarla? Contestemos negativamente: no hay que usarla cuando exista una solución iterativa obvia. Siempre será más económica en tiempo y en memoria. Y aun cuando la solución iterativa no sea obvia, a veces será necesario encontrarla si la profundidad de la recursión hace inviable el utilizarla.

Hemos llegado, por eliminación al tipo de problemas en los que es útil la recursividad: aquellos en que la única formulación que aparece intuitivamente es la recursiva; aquellos que, aunque admitan solución iterativa, ésta es extremadamente compleja al lado de la recursiva.

Lo que es innegable es que, en cualquier caso, la solución recursiva se muestra como una salida mucho más elegante.

EL PROBLEMA DE LAS 8 REINAS: UNA SOLUCION

El problema de las 8 reinas consiste en situar ocho piezas de éste tipo en un tablero de ajedrez (que tiene 8 x 8 = 64 cuadros) de forma que no se maten entre sí.

Los movimientos posibles de una reina se muestran en la figura 1, en que R es la posición de la reina, y en donde se han marcado con asteriscos las posiciones afectadas.

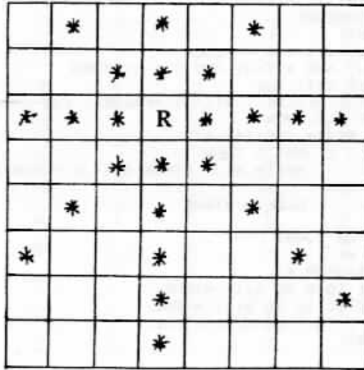

Figura 1

Este problema, como veremos, es un buen ejemplo de los citados en el último párrafo del apartado anterior: aquellos en que la aproximación recursiva aparece más clara e intuitiva que ninguna otra.

Originalmente este problema fue investigado por C. F. Gauss, en 1850, pero no llegó a resolverlo completamente. Lo cual es lógico, pues pertenece a un tipo de problemas que desafían cualquier solución de tipo analítico, y que requieren gran paciencia y exactitud si quieren resolverse manualmente.

Se popularizó al aparecer en el libro "Structured Programming" (ref. [2]), al final del artículo de Dijkstra, como ejemplo de resolución de un problema por refinamientos sucesivos.

De hecho, el trabajo original de Dijkstra (ref. [3]) no incluía este apartado que apareció originalmente en el texto que el citado profesor utilizaba como soporte de sus clases (ref. [4]). Posteriormente este problema ha aparecido en gran cantidad de textos aunque a veces presentado en forma iterativa (ref. [5]). Junto con el problema de las Torres de Hanoi (ref. [4] y [6]), me atrevo a calificarlos de "clásicos" de la recursividad.

Wirth lo resuelve en [1] mediante una aproximación por el método "trial-and-error", que consiste en avanzar en ir generando una solución mientras sea buena; si no lo es, se retrocede ("backtracking") hasta tener un camino alternativo y volver a avanzar. Este avance y retroceso se implementan recursivamente.

La primera aproximación consistirá en pensar un subprograma TRY que sitúe la I-ésima reina en el tablero.

Su esquema sería:

```
PROCEDURE TRY (I: INTEGER);
BEGIN
    inicializar selección de posiciones;
    REPEAT
        seleccionar nueva posición;
        IF posición segura
            THEN BEGIN situar-reina;
                IF I < 8
                    THEN BEGIN
                        TRY (I + 1);
                        IF no ha ido bien
                            THEN retirar-reina
                    END
                    ELSE ha ido bien
            END
    UNTIL ha ido bien OR no hay más posiciones
END
```

Intentemos clasificar este esquema con un ejemplo. Partimos de la situación presentada en la figura 2, en la que ya hemos situado 3 reinas, e intentamos situar la cuarta. Se supone que, en cada paso, situamos una reina en cada columna.

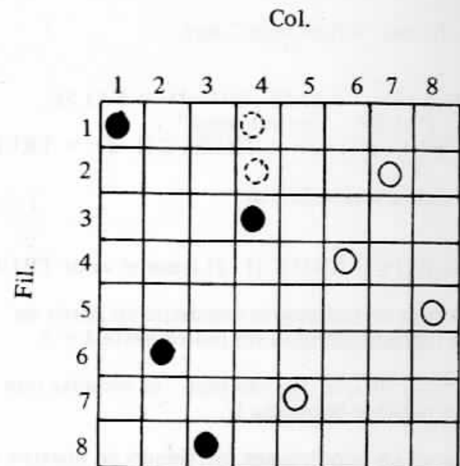

Figura 2

En nuestra hipótesis, partimos de las 3 primeras reinas, situadas respectivamente en las posiciones 1, 6 y 8, y tratamos de situar la cuarta es decir, TRY (4). Empezamos situando la cuarta reina en la primera fila (primera posición punteada). La posición no es segura, por lo que, siguiendo nuestro esquema, volvemos rápidamente a "seleccionar nueva posición", que será, de momento, en la segunda fila (segunda posición punteada). Ahora se trata de una posición "segura". Siguiendo el algoritmo, "situamos-reina", y como I es menor que 8, se intentará terminar de rellenar el tablero, lo cual se implementa con la llamada recursiva TRY (I + 1), en nuestro caso TRY (5) (la cual, si encuentra posición segura llamará a TRY (6), ésta a TRY (7) y ésta a TRY (8)). Ocurre que con la cuarta reina en la segunda fila no hay forma de terminar, luego cómo "no ha ido bien", efectuamos "retirar-reina", con lo que se vuelve a "seleccionar" la tercera fila (posición en negro), la posición es "segura", "situamos reina", y como $I < 8$ se repite TRY (5). Esta vez, en cambio, las cosas marchan bien y se consigue situar las restantes reinas (círculos en blanco en la figura 2), con lo que el algoritmo finaliza.

Podemos observar que, con el esquema recursivo se puede construir el algoritmo TRY (I), suponiendo que sabemos la solución para TRY (I + 1) (en el ejemplo anterior del factorial, construimos F (N), dando por supuesto que sabíamos calcular F (N-1)).

Para escribir el programa definitivo a partir de nuestro esquema debemos decidir la representación de tablero y reinas. Aunque lo primero que se nos ocurre es representar la posición de las reinas en un vector X_i que contendrá en su elemento X_3 la fila en que se ha situado la tercera reina. En la solución presentada en la figura 2, X = (1, 6, 8, 3, 7, 4, 2, 5).

La seguridad por columna no hace falta comprobarla ya que cada vez situamos una sola reina por columna. Para las filas nos bastará un vector A_j, de tipo booleano, otro B_k para las diagonales ↗, y otro C_k para las diagonales ↖ (notar que en las diagonales ↗, el valor I + J es constante, y oscila entre 2 = 1 + 1 y 16 = 8 + 8, y en las diagonales ↖ el valor I-J es constante y oscila entre −7 = 1 − 8 y 7 = 8 − 1).

Nuestra representación de datos queda:

X: ARRAY [1..8] OF INTEGER
A: ARRAY [1..8] OF BOOLEAN
B: ARRAY [2..16] OF BOOLEAN
C: ARRAY [−7..7] OF BOOLEAN
en donde
X (I) indica la posición de la reina en la I-ésima columna.
A (J) indica si hay reina en la fila J.
B (I + J) indica si hay reina en la diagonal ↗ (I + J) ésima
C (I−J) indica si hay reina en la diagonal ↖ (I − J) ésima.

De esta forma "situar-reina" será:

X (I): = J; A (J): = FALSE; B (I+J): = FALSE;
C (I−J): = FALSE y "retirar reina":
A (J): = TRUE; B (I+J): = TRUE; C (I−J): = TRUE

Una posición será segura si:

A (J) AND B (I+J) AND C (I−J) tiene el valor TRUE

La selección de posiciones consistirá en partir de J: = 0 e ir incrementando este indice hasta J = 8.

El indicador de que "ha ido bien" se resuelve con una simple variable booleana Q.

Estamos ya en condiciones, partiendo de nuestro esquema de escribir el programa completo (ver programa 1). En efecto, el subprograma de impresión es muy simple, y el programa principal se limitará a inicializar a TRUE los vectores A, B, C y a poner en marcha el proceso mediante una llamada TRY (1). (Recomiendo al lector la comparación del esquema y el programa viendo cómo el primero ha sido respetado y cómo se ha refinado cada una de las instrucciones redactadas en lenguaje natural). La solución hallada por el programa puede verse en la figura 3.

```
* - - - - - - -
- - - - - - * -
- - - - * - - -
- - - - - - - *
- * - - - - - -
- - - * - - - -
- - - - - * - -
- - * - - - - -
```

Figura 3. Solución obtenida mediante el programa 1

```
PROGRAM EIGHTQUEEN1(OUTPUT);
VAR I: INTEGER ; Q: BOOLEAN;
    A: ARRAY(. 1..8. ) OF BOOLEAN;
    B: ARRAY(. 2..16. ) OF BOOLEAN;
    C: ARRAY(. -7..7. ) OF BOOLEAN;
    X: ARRAY(. 1..8. ) OF INTEGER;
PROCEDURE PRINT;
VAR I,J : INTEGER;
BEGIN
   FOR I:=1 TO 8 DO WRITE(X(I):4);
   WRITELN;
   FOR I:=1 TO 8 DO
    BEGIN
       FOR J:=1 TO 8 DO
         BEGIN
           IF X(J)=I THEN WRITE ('*')
                     ELSE WRITE ('-') ;
           WRITE (' ')
         END ;
       WRITELN
     END;
   WRITELN
END; (* PRINT *)
PROCEDURE TRY(I: INTEGER ; VAR Q: BOOLEAN);
   VAR J: INTEGER;
BEGIN J:= 0 ;
   REPEAT J:=J+1 ; Q:=FALSE;
     IF A(J) AND B(I+J) AND C(I-J) THEN
       BEGIN X(I):=J;
         A(J):=FALSE ; B(I+J):=FALSE ; C(I-J):=FALSE;
         IF I<8 THEN
           BEGIN TRY(I+1,Q);
             IF NOT Q THEN
                BEGIN A(J):=TRUE; B(I+J):=TRUE; C(I-J):=TRUE END
           END
                ELSE Q:=TRUE
       END
   UNTIL Q OR (J=8)
END (* TRY *) ;
BEGIN (* MAINPGM *)
   FOR I:=1 TO 8 DO A(I):=TRUE;
   FOR I:=2 TO 16 DO B(I):=TRUE;
   FOR I:=-7 TO 7 DO C(I):=TRUE;
   TRY (1,Q);
   PAGE;
   IF Q THEN PRINT
END.
```

PROGRAMA 1

8 REINAS: TODAS LAS SOLUCIONES

Tratemos ahora de generalizar y hallar, no una, sino todas las soluciones. Se trata de una muy simple generalización del programa anterior. Volvamos al esquema. El bucle que controla el algoritmo es del tipo REPEAT-UNTIL, es decir, que termina al hallar una solución. La primera generalización consistirá en eliminar el control de si "ha ido bien", y no terminar hasta explorar todas las soluciones. Sustituiremos el REPEAT-UNTIL por un FOR J: = 1 TO 8 DO...
Sigamos. Antes retirábamos la reina sólo si no había ido bien. Ahora la deberemos retirar siempre para posibilitar seguir explorando soluciones. Y lo que habrá que hacer cada vez que se halla una solución (cuando se coloca la octava reina) será imprimir la configuración.

El esquema para hallar todas las soluciones será:

```
PROCEDURE TRY (I: INTEGER);
BEGIN
FOR todas las posiciones de la columna DO
   IF posición segura
      THEN BEGIN situar-reina
                 IF I < 8 THEN TRY (I + 1)
                          ELSE imprimir configuración
               retirar reina
           END
END
```

Sorprendentemente hemos dado con un esquema más simple, aunque algo más difícil de comprender. El programa que deriva de este esquema (programa-2) es en consecuencia también más simple que el anterior. Dado que se avanza de una forma sistemática, no hay peligro de hallar dos veces la misma solución (ver figura 4), aunque de las 92 soluciones halladas sólo 12 son significativamente distintas ya que el programa es incapaz de reconocer simetrías.

Por razones de espacio, he sustituido en el programa-2, el subprograma de impresión (PRINT) por un simple volcado del contenido de X (fig. 4).

1	5	8	6	3	7	2	4
1	6	8	3	7	4	2	5
1	7	4	6	8	2	5	3
1	7	5	8	2	4	6	3
2	4	6	8	3	1	7	5
2	5	7	1	3	8	6	4
2	5	7	4	1	8	6	3
2	6	1	7	4	8	3	5
2	6	8	3	1	4	7	5
2	7	3	6	8	5	1	4
2	7	5	8	1	4	6	3
2	8	6	1	3	5	7	4
3	1	7	5	8	2	4	6
3	5	2	8	1	7	4	6
3	5	2	8	6	4	7	1
3	5	7	1	4	2	8	6
3	5	8	4	1	7	2	6
3	6	2	5	8	1	7	4
3	6	2	7	1	4	8	5
3	6	2	7	5	1	8	4
3	6	4	1	8	5	7	2
3	6	4	2	8	5	7	1
3	6	8	1	4	7	5	2
3	6	8	1	5	7	2	4
3	6	8	2	4	1	7	5
3	7	2	8	5	1	4	6
3	7	2	8	6	4	1	5
3	8	4	7	1	6	2	5
4	1	5	8	2	7	3	6
4	1	5	8	6	3	7	2
4	2	5	8	6	1	3	7
4	2	7	3	6	8	1	5
4	2	7	3	6	8	5	1
4	2	·7	5	1	8	6	3

Figura 4. Algunas soluciones obtenidas mediante el programa 2

```
PROGRAM EIGTHQUEENS(OUTPUT);
VAR I: INTEGER;
     A: ARRAY (. 1..8 .) OF BOOLEAN;
     B: ARRAY (. 1..16.) OF BOOLEAN;
     C: ARRAY (.-7..7 .) OF BOOLEAN;
     X: ARRAY (. 1..8 .) OF INTEGER;
PROCEDURE PRINT;
     VAR K: INTEGER;
 BEGIN
     FOR K:=1 TO 8 DO WRITE(X(K):4);
     WRITELN
 END (* PRINT *) ;
PROCEDURE TRY (I:INTEGER);
     VAR J: INTEGER;
 BEGIN
     FOR J:=1 TO 8 DO
         IF A(J) AND B(I+J) AND C(I-J)
            THEN BEGIN X(I):=J;
                   A(J):=FALSE; B(I+J):=FALSE; C(I-J):=FALSE;
                   IF I<8 THEN TRY(I+1) ELSE PRINT;
                   A(J):=TRUE; B(I+J):=TRUE; C(I-J):=TRUE
                 END
END  (* TRY *) ;
BEGIN  (* MAINPGM *)
      FOR I:= 1 TO 8  DO A(I):=TRUE;
      FOR I:= 2 TO 16 DO B(I):=TRUE;
      FOR I:=-7 TO 7  DO C(I):=TRUE;
      TRY(1)
END.
```

PROGRAMA 2

Pere Botella

REFERENCIAS

[1] ALGORITHMS + DATA = PROGRAMS. NIKLAUS WIRTH, Prentice Hall, 1976

[2] STRUCTURED PROGRAMMING, DAHL, DIJKSTRA, HOARE, Academic Press, 1972

[3] NOTES ON STRUCTURED PROGRAMMING, EDSGER W. DIJKSTRA. EWD 249
Report Technological University Eindhoven, 1970

[4] A SHORT INTRODUCTION TO THE ART OF PROGRAMMING, EDSGER W. DIJKSTRA, EWD 316
Report Technological University Eindhoven, 1971

[5] AN INTRODUCTION TO PROGRAMMING AND PROBLEM SOLVING WITH PASCAL, SCHNEIDER, EWINGART, PERLMAN. JOHN WILEY, 1978

[6] LA TORRE D'HANOI. APLICACIÓ D'UN PROCEDIMENT RECURSIU PER A LA SOLUCIÓ D'UN PROBLEMA COMBINATORI. Jaume Ribera, Novática N.º 22