EJEMPLOS RECURSIVIDAD

1. Factorial

La definición recursiva del factorial de un número $n > 0$, considerando que $0! = 1$ por definición, se obtiene como consecuencia de aplicar la propiedad asociativa de la multiplicación, es decir:

$$n! = n*(n-1)*(n-2)*... *2*1$$

asociando los factores de la siguiente forma

$$n! = n * [(n-1)*(n-2)* ... *2*1]$$

se tiene que

$$n! = n * (n-1)!$$

Así pues, la definición recursiva de factorial es:

$$n! = \begin{cases} n*(n-1)! & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Por lo tanto, la función recursiva que calcula el factorial de un número puede escribirse como sigue:

```
{
Función: factorial
Datos de entrada: El número n del que calcular su factorial, que debe
cumplir la precondición (n > 0).
Datos de salida: El factorial del número, es decir, factorial(n) = n!
}
Funcion factorial (E n: Entero): Entero
     Inicio
                              {la solución para el caso base es directa}
          Si (n = 0)
               Entonces Retorna 1;
                          {mientras que para el caso general es recursiva}
               Sino     Retorna n*factorial(n-1);
          FinSi;
     FinFuncion
```

2. **Multiplicación entera**

La definición recursiva de la multiplicación de dos números a y b, se deriva de la definición de la multiplicación como una suma abreviada y la aplicación de la propiedad asociativa de la suma, es decir:

$$a * b = a + a + ^{(b \text{ veces}} \ldots + a$$

de modo que asociándose los sumandos como sigue:

$$a * b = a + [\ a + ^{(b\text{-}1 \text{ veces}} \ldots + a]$$

se tiene que

$$a * b = a + [a * (b\text{-}1)]$$

Así pues, la definición recursiva de la multiplicación es:

$$a * b = \begin{cases} a + (a * (b - 1)) & \text{si } b > 0 \\ 0 & \text{si } b = 0 \end{cases}$$

Por lo tanto, la función recursiva que calcula el producto de dos números puede escribirse como sigue:

```
{
Función: mult
Datos de entrada: Los números a multiplicar: a y b. Como precondición
se considera que los números deben ser no negativos, a, b >= 0
Datos de salida: El valor del producto de los dos factores pasados, es
decir, mult(a, b) = a*b
}
Funcion mult (E a, b: Entero): Entero
     Inicio
                        {la solución para el caso general es recursiva}
          Si (b > 0)
               Entonces Retorna 0;
                            {mientras que para el caso base es directa}
               Sino     Retorna a + mult(a, b-1);
          FinSi;
     FinFuncion
```

3. **Exponenciación entera**

La definición recursiva de la operación exponenciación entera, es decir, calcular la potencia a^b, se deriva de la definición de la potencia como una multiplicación abreviada y la aplicación de la propiedad asociativa de la multiplicación (el proceso es similar al del ejemplo anterior), es decir:

$$a^b = a * a * {}^{(b \text{ veces}} \ldots * a$$

de modo que asociándose los factores como sigue:

$$a^b = a * [\, a * {}^{(b\text{-}1 \text{ veces}} \ldots * a]$$

se tiene que

$$a^b = a * a^{b\text{-}1}$$

Así pues, la definición recursiva de la exponenciación es:

$$a^b = \begin{cases} a * a^{b-1} & \text{si } b > 0 \\ 0 & \text{si } b = 0 \end{cases}$$

Este ejemplo, es también interesante porque ilustra cómo se pueden buscar otras formas de definir recursivamente la operación propuesta, y obtener beneficios (en términos de eficiencia del algoritmo) de la nueva descomposición.

En el caso de la exponenciación, cuando el exponente b de la operación es par, se tiene que:

$$a^b = [\, a * \ \ldots {}^{(b/2 \text{ veces}} \ \ldots * a\,]*[a * \ldots {}^{(b/2 \text{ veces}} \ldots * a] = a^{b/2} * a^{b/2}$$

Considerándose este hecho, puede definirse recursivamente la operación potencia como:

$$a^b = \begin{cases} a^{b/2} * a^{b/2} & \text{si } b > 0 \wedge \text{EsPar}(b) \\ a * a^{b-1} 0 & \text{si } b > 0 \wedge \text{EsImpar}(b) \\ 1 & \text{si } b > 0 \end{cases}$$

Esta definición, permite la implementación del denominado algoritmo de exponenciación rápida, que realiza menos multiplicaciones si se calcula sólo una vez el término $a^{b/2}$

```
{
Función: expR
Datos de entrada: La base y el exponente, a y b, respectivamente. Como
precondición se considera que a y b son no negativos, y que no pueden
ser 0 simultáneamente, es decir, (a=0 AND b>0) OR (a>0 AND b>=0)
Datos de salida: El resultado de elevar a a b, es decir, expR(a, b)= a^b
}
Funcion expR (E a, b: Entero): Entero
     Variables
          temp: Entero;
     Inicio
                                {la solución para el caso base es directa}
          Si (b = 0)
               Entonces Retorna 1;
                        {mientras que para el caso general es recursiva}
               Sino     Si ( b MOD 2 = 0 )
                             Entonces temp ← expR(a, b DIV 2);
                                      Retorna temp*temp;
                             Sino     Retorna a*expR(a, b-1);
                        FinSi
          FinSi;
     FinFuncion
```

4. **Máximo Común Divisor**

Todos los ejemplos anteriores se basan en la solución del mismo problema para un caso más sencillo (factorial(*n*-1), mult(*a*, *b*-1) y expR(*a*, *b* DIV 2) o expR(*a*, *b*-1), respectivamente). La solución parcial obtenida no es la solución final, sino que debe combinarse de algún modo con los datos actuales (multiplicando por *n*, sumando *a*, multiplicándose por sí mismo o por el término *a*, respectivamente). Cuando la recursividad tiene esta característica, es decir, cuando es necesario combinar los resultados parciales con los datos actuales para obtener la solución final, se dice que la recursividad es NO FINAL.

Existen ejemplos de recursiones finales, como por ejemplo, el caso que se trata. El cálculo del máximo común divisor se basa en la siguiente propiedad de los números enteros:

$$m.c.d.(a,b)=\begin{cases} m.c.d.(a-b,b) & \text{si } a \geq b \\ m.c.d.(a,b-a) & \text{si } b > a \\ a & \text{si } b = 0 \\ b & \text{si } a = 0 \end{cases}$$

Esta definición, permite la definición de la siguiente función recursiva para el cálculo del máximo común divisor de dos números, que como se aprecia es FINAL.

```
{
Función: mcd
Datos de entrada: Los dos números de los que calcular su MCD, a y b.
Como precondición se considera que a y b son no negativos y no pueden
ser 0 simultáneamente, es decir, (a=0 AND b>0) OR (a>0 AND b>=0)
Datos de salida: El máximo común divisor de los dos números, es decir,
mcd(a, b)= M.C.D.(a, b)
}
Funcion mcd (E a, b: Entero): Entero
  Inicio
                              {la solución para los casos bases son directas}
      Si (b = 0)
          Entonces Retorna a
          Sino  Si ( a = 0)
                      Entonces       Retorna b;
                      {mientras que para los casos generales son recursivas}
                      Sino  Si ( a >= b )
                                  Entonces  Retorna mcd(a-b, b);
                                  Sino      Retorna mcd(a, b-a);
                              FinSi
                  FinSi;
      FinSi
  FinFuncion
```

5. **Suma de los elementos de un vector**

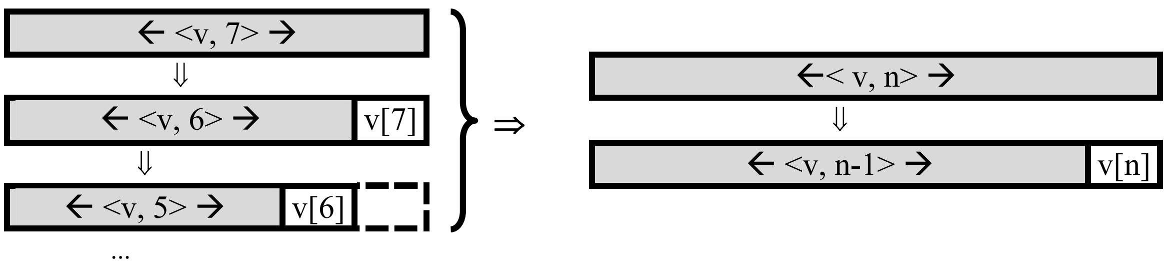
La suma de los elementos de un vector puede expresarse, también, recursivamente. Para ello, considérese que la suma de tales elementos,

$$\sum_{i=1}^{n} v_i = v_1 + \ldots + v_{n-1} + v_n$$

puede rescribirse como sigue, sin más que considerar la propiedad asociativa de la suma:

$$\sum_{i=1}^{n} v_i = (v_1 + \ldots + v_{n-1}) + v_n = \sum_{i=1}^{n-1} v_i + v_n$$

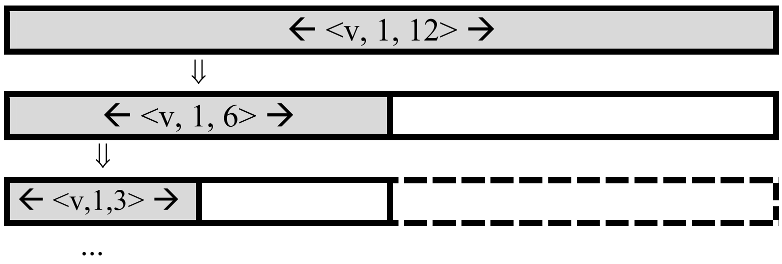
Por lo tanto, la definición recursiva de la suma de los elementos de un vector se basa en reducir en un elemento el vector original, y así sucesivamente. Los datos que representan un vector deben admitir este proceso. Si se considera como representación de un vector un `array` *v*, y un índice *n* que mantiene en todo momento, el número de elementos que, comenzando desde la primera posición del `array`, se consideran efectivamente en el vector, puede reducirse el vector por la derecha, sin más que decrementar en 1 el valor del índice *n*, es decir, si el par <*v*, *n*> representa un vector de *n* elementos, los datos <*v*, *n*-1> representa el mismo vector, pero con un elemento menos por la derecha. Gráficamente,



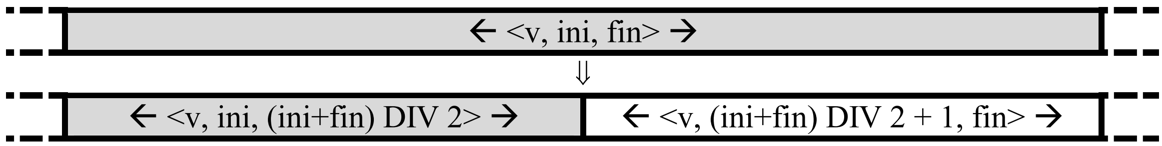
Así pues, la función recursiva que calcula la suma de los elementos de un vector es:

```
{
Función: suma_vec
Datos de entrada: Un vector de números (enteros), representado por la
pareja de datos <v, n>. Como precondición se considera que el vector
ya tiene cargados un conjunto de valores válido y que n es un valor
comprendido entre 0 (el vector puede estar vacío) y NMAX (la dimensión
máxima del array
Datos de salida: La suma de los elementos del vector, es decir,
suma_vec(v, n) = ∑v[i], i= 1,..., n
}
Función suma_vec (E v: Tarray; E n: Entero): Entero
  Inicio
                                {la solución para el caso base es directa}
     Si (n = 0)
        Entonces Retorna 0;
                             {la solución para el caso general es recursiva}
        Sino  Retorna suma_vec(v, n-1) + v[n];
     FinSi
  FinFunción
```

Como se ha visto en el Ejemplo 3, los datos admiten diferentes descomposiciones recursivas que conducen a diferentes soluciones recursivas. En el caso de la suma de los elementos de un vector, puede considerarse primero calcular la suma de las dos mitades del vector, para después calcular la suma de todos los elementos del vector. En este caso, se hace necesario considerar subvectores en vez de vectores, es decir, es necesario poder representar cualquier subconjunto de elementos consecutivos de un vector. Así pues, debe pasarse de la representación de un vector, dada por el par <*v*, *n*>, a la de un subvector, que viene dada por la terna <*v*, *ini*, *fin*>. Gráficamente,



En general, se tiene que:



Así pues, otra versión de la función que suma los elementos de un vector puede ser:

```
{
Función: suma_vec2
Datos de entrada: Un subvector de números (enteros), representado por
la tripleta de datos <v, ini, fin>. Como precondición se considera que
el vector ya tiene cargados un conjunto de valores válido y que el
número de elementos del subvector es cero si fin < ini, y el valor de
la expresión fin - ini + 1, en caso contrario
Datos de salida: La suma de los elementos del subvector, es decir,
suma_vec2(v, ini, fin) = ∑v[i], i= ini,..., fin
}
Funcion suma_vec2 (E v: Tarray; E ini, fin: Entero): Entero
 Variables
    cen: Entero;
 Inicio
                              {la solución para el caso base es directa}
    Si (fin < ini)
       Entonces Retorna 0;
                         {la solución para el caso general es recursiva}
       Sino  cen ← (ini+fin) DIV 2;
             Retorna suma_vec2(v, ini, cen) + suma_vec2(v, cen+1, fin)
    FinSi
 FinFuncion
```

Asimismo, este ejemplo a diferencia de los anteriores, es un ejemplo de recursividad NO LINEAL, es decir, una definición recursiva en la cual cada invocación externa a la función genera más de una llamada recursiva (en concreto, dos). Un último comentario se refiere al hecho de cómo invocar a la función definida para sumar los elementos de un vector <v, n>. En este caso, bastaría con realizar la invocación: `suma_vec2(v, 1, n);`

6. **Búsqueda Binaria**

Como ya se vio en el Tema 2, la especificación formal de la búsqueda binaria es la siguiente:

$\{\ x = X \wedge \langle v, n\rangle = \langle \mathbf{V}, N\rangle \wedge \forall k\colon 1 \leq k < N\colon v[k] \leq v[k+1] \wedge v[0] \leq x < v[n+1]\ \}$
busqueda binaria
$\{\ x = X \wedge \langle v, n\rangle = \langle \mathbf{V}, N\rangle \wedge (0 \leq i < N+1) \wedge (v[i] \leq x < v[i+1])\ \}$

Esta especificación se basa en las siguientes suposiciones:

- El vector $\langle v, n\rangle$ está inicializado y sus elementos están ordenados.
- Existen dos componentes ficticias en el vector: $v[0] = -\infty$ y $v[n+1] = +\infty$. De esta forma se garantiza que para cualquier valor x que se busque, se cumple que $v[0] \leq x < v[n+1]$.
- Que como resultado de la búsqueda se determina la posición i en la que debería encontrarse el elemento buscado x, dentro del vector (manteniendo el orden, por supuesto, y suponiendo que en caso de repetición se colocase después de todas las repeticiones), es decir, $(0 \leq i < N+1) \wedge (v[i] \leq x < v[i+1])$.

Pues bien la solución recursiva se fundamenta en las mismas hipótesis, aunque aplicando el enfoque recursito.

Como clave para la solución recursiva de la búsqueda cabe citar que, en cada paso, se considera el elemento central del vector (con lo que éste se DIVIDE en dos mitades) y dependiendo de la comparación de x con $v[cen]$ se continúa el mismo proceso por la mitad correspondiente.
Para poder llevar a cabo esta reducción del problema, es necesario que la representación de datos utilizada admita la misma. Por lo tanto, en este caso será necesario tratar subvectores en vez de un vector, por lo que se considerará como dato de entrada a la función recursiva, un subvector dado por la terna $\langle v, ini, fin\rangle$. Teniendo en cuenta las componentes ficticias . Si se define la siguiente interfaz para la función recursiva:

```
Funcion BusBinR(E v: TArrray; E ini, fin: Entero; E x: Entero): Entero
```

La solución recursiva al problema puede expresarse como:

```
cen ← (ini +fin) DIV 2;
Si (x < v[cen])
      Entonces    Retorna BusBinR(v, ini, cen, x);
      Sino        Retorna BusBinR(v, cen, fin, x);
FinSi;
```

El caso base se deduce de la postcondición de la búsqueda, es decir, $(0 \leq i < N+1) \wedge (v[i] \leq x < v[i+1])$, por lo que necesariamente se debe reducir el subvector hasta que el último subvector considerado siempre tenga dos componentes consecutivas. Así pues, el caso base se alcanza cuando las posiciones inicial y final del subvector son consecutivas ($ini+1 = fin$), en cuyo caso, el resultado de la búsqueda viene dado por el valor de la posición *ini*. Nótese, que en el caso extremo de que el vector esté vacío, se cumple inmediatamente la condición del caso base, gracias al hecho de considerar las dos posiciones ficticias $v[0]$ y $v[n+1]$.

Por lo tanto, la definición de la función completa, podría ser la siguiente:

```
{
Función: BusBinR
Datos de entrada: Un subvector de números (enteros), representado por
la tripleta de datos <v, ini, fin>. Como precondición se considera que
el vector ya tiene cargados un conjunto de valores válido, que estos
valores están ordenados, y que el número de elementos del subvector es
cero si fin < ini, y el valor de la expresión fin – ini + 1, en caso
contrario.
Además, se tiene que x es el elemento a buscar dentro del subvector.
Datos de salida: La posición i en la que debería encontrarse el
elemento x dentro del subvector, de forma que se cumple que v[i] <= x
< v[i+1]
}
Funcion BusBinR (E v: Tarray; E ini, fin: Entero; E x: Entero): Entero
  Variables
     cen: Entero;
  Inicio
                                {la solución para el caso base es directa}
     Si (fin < ini)
        Entonces Retorna ini;
                           {la solución para el caso general es recursiva}
        Sino  cen ← (ini+fin) DIV 2;
              Si (x < v[cen] )
                    Entonces    Retorna BusBinR(v, ini, cen, x);
                    Sino        Retorna BusBinR(v, cen, fin, x);
              FinSi
     FinSi
  FinFuncion
```

Con el fin de adaptar la función recursiva definida a una función que reciba el elemento *x* a buscar y un vector <*v*, *n*> bastaría con definirse la siguiente función, a modo de adaptador entre diferentes interfaces.

```
{
Función: BusBin
Datos de entrada: Un vector de números (enteros), representado por el
par de datos <v, n>. Como precondición se considera que el vector ya
tiene cargados un conjunto de valores válido, que estos valores están
ordenados, y que el número de elementos es mayor o igual a cero, n>=0.
Además, se tiene que x es el elemento a buscar dentro del subvector.
Datos de salida: La posición i en la que debería encontrarse el
elemento x dentro del vector, de forma que se cumple que v[i] <= x <
v[i+1]
}
Funcion BusBin (E v: Tarray; E n: Entero; E x: Entero): Entero
  Inicio
     Retorna BusBinR(v, 0, n+1, x);
  FinFuncion
```

7. **QuickSort**

El algoritmo de ordenación rápida o *quicksort* es un algoritmo recursivo de ordenación que se basa en la siguiente idea. Dado un vector a ordenar, se considera un elemento cualquiera que se denominará pivote, por ejemplo, el primer elemento. El proceso de ordenación *quicksort* se basa en un proceso subordinado de partición consistente en recolocar los elementos del vector de forma que los elementos que sean menores o iguales que el pivote se sitúen antes que éste en el vector resultante, y que los mayores que el pivote se sitúen después de éste en el vector resultante. Como consecuencia de este proceso, el vector no está completamente ordenado, pero sí que se cumple que: el pivote está ya ordenado y que los elementos por debajo de éste son menores o iguales, y los que están por encima son mayores. Gráficamente, el resultado de particionar un vector, considerando como pivote el primer elemento sería el siguiente:

posición inicial del pivote (pos_pivote = 1)
↓

13	2	5	2	54	3	42	7	38	9	10	3	23	4	56	4

⇓
Particionar
⇓

10	2	5	2	4	3	4	7	3	9	**13**	38	23	42	56	54

← menores o iguales que el pivote → ↑ ← mayores que el pivote →

Posición del pivote después de particionar (*pos_pivote* = 11)

Una vez realizado el proceso de partición, el método de ordenación rápida se vuelve a invocar para ordenar cada una de las dos partes resultantes. Como puede apreciarse, para poder llevar a cabo este proceso es necesario considerar subvectores en vez de vectores, con el fin de que los datos manejados por el algoritmo recursivo admitan la descomposición manejada. Para poder particionar el vector, es imprescindible que al menos haya dos elementos, por lo que la condición del caso base del algoritmo quicksort se cumplirá cuando el subvector tenga 0 ó 1 elementos.

Supuesta la definición de la acción Particionar, cuya interfaz se incluye a continuación,

```
{
Procedimiento: Particionar
Datos de entrada: Un subvector de números (por ejemplo, enteros), representado
por la tripleta de datos <v, ini, fin>. Como precondición se considera que el
vector ya tiene cargados un conjunto de valores válido y la posición del
elemento por el que se particiona el vector, pos_pivote, que cumple que izq <=
pos_pivote <= der
Datos de salida: La nueva posición del pivote, que se recolocará de forma que
los elementos menores o iguales que el pivote estén por debajo de dicha
posición y los mayores por encima. El subvector de entrada con sus elementos
recolocados.
}
Procedimiento Particionar (  E/S v: Tarray; E ini, fin: Entero;
                             E/S pos_piv: Entero)
```

el algoritmo *quicksort* quedaría como sigue:

```
{
Procedimiento: QuickSort
```

```
Datos de entrada: Un subvector de números (por ejemplo, enteros), representado
por la tripleta de datos <v, ini, fin>. Como precondición se considera que el
vector ya tiene cargados un conjunto de valores válido
Datos de salida: El subvector de entrada con sus elementos ordenados
}
Procedimiento QuickSort (E/S v: Tarray; E ini, fin: Entero)
  Variables
      pos_pivote: Entero;
  Inicio
      Si (der>izq)
          Entonces  pos_pivote ← ini;
                    Particionar(v, izq, der, pos_pivote);
                    QuickSort(v, izq, pos_pivote-1);
                    QuickSort(v, pos_pivote+1, der);
      FinSi
  FinProcedimiento
```

Así pues, para ordenar el vector <*v*, *n*>, bastaría con invocar al procedimiento QuickSort de la siguiente forma: Quicksort(*v*, 1, *n*).

Por lo que se refiere al algoritmo particionar, éste se basa en un algoritmo iterativo que trata de recorrer alternativamente el subvector con dos índices. Un índice recorre el subvector de izquierda a derecha, comenzando desde el principio, tratando de localizar un elemento mayor que el pivote. Una vez encontrado, se recorre el vector de derecha a izquierda, comenzando por el último, con la intención de localizar un elemento menor o igual que el pivote. Una vez localizados estos dos elementos, que están mal posicionados en el vector resultante, éstos se intercambian y se repite el proceso hasta que se cruzan los índices. En detalle, el algoritmo seguido es el siguiente:

```
Procedimiento Particionar ( E/S v: Tarray; E ini, fin: Entero;
                            E/S pos_piv: Entero)
  Variables
      izq, der, pivote: Entero;
  Inicio
      {1. Garantizar que el pivote ocupa la primera posición}
          pivote ← v[pos_piv];
          Intercambiar[1](v, ini, pos_piv);
      {2. Inicialización índices}
          izq ← ini; der ← fin + 1;
      {3. Buscar primer elemento mal posicionado por la izquierda}
          Repetir izq ← izq + 1 Hasta ((v[izq]>pivote) OR (izq >= der));
                 {se controla que izq < der, puede no haber eltos. > que pivote}
      {4. Buscar primer elemento mal posicionado por la derecha}
          Repetir der ← der -1 Hasta (v[der]<=pivote)
                        {no se controla que der < izq, seguro que v[der]<=pivote}
      {5. Mientras estén separados, continuar colocándoles bien}
          Mientras (izq < der) Hacer
              {5.1 Intercambiar elementos descolocados en cada lado}
                  Intercambiar(v, izq, der);
              {5.2 Buscar siguiente elemento mal posicionado por la izquierda}
                  Repetir izq ← izq + 1 Hasta (v[izq] > pivote);
              {5.3 Buscar siguiente elemento mal posicionado por la derecha}
                  Repetir der ← der - 1 Hasta (v[der] <= pivote);
          FinMientras;
      {6. Colocar pivote en su sitio (posición final recorrido por la derecha)}
          Intercambiar(v, ini, der);
      {7. Actualizar posición pivote}
          pos_pivote ← der;
FinProcedimiento;
```

[1] `Intercambiar(v, i, j)` simplemente intercambia los elementos del (sub)vector *v* que ocupan las posiciones *i* y *j*, respectivamente.

8. **Torres de Hanoi**

En el momento de la creación del mundo, los sacerdotes del templo de Brama recibieron una plataforma de bronce sobre la cual había tres agujas de diamante. En la primera aguja estaban apilados sesenta y cuatro discos de oro, cada uno ligeramente menor que el que estaba debajo. A los sacerdotes se les encomendó la tarea de pasarlos todos de la primera aguja a la tercera, con dos condiciones: sólo puede moverse un disco a la vez, y ningún disco podrá ponerse encima de otro más pequeño. Se dijo a los sacerdotes que, cuando hubieran terminado de mover los sesenta y cuatro discos, llegaría el fin del mundo.

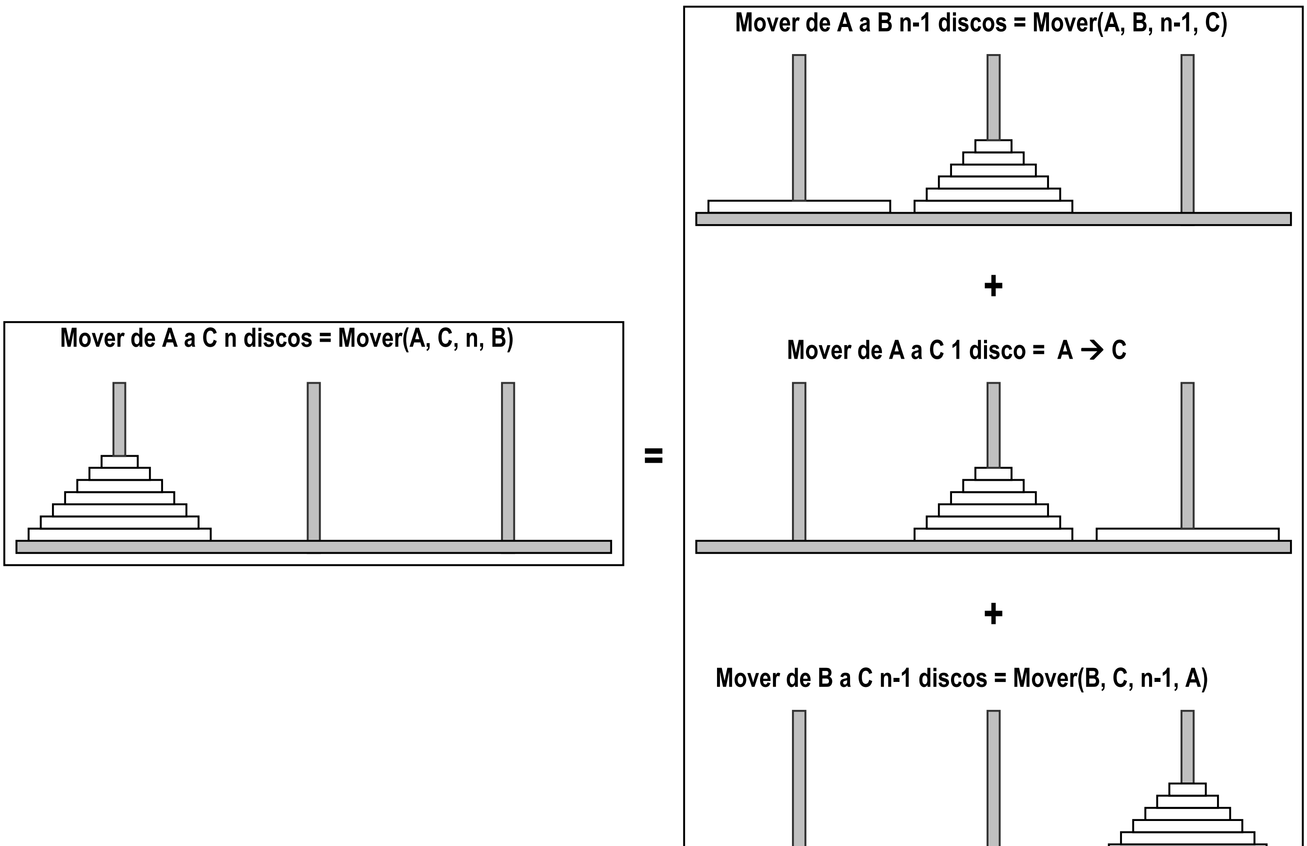
Se trata de realizar un procedimiento recursivo que indique los movimientos (válidos según las reglas indicadas) para mover *n* discos de la primera a la tercera aguja.

Con el fin de tratar inferir una solución general, se incluye a continuación, la lista de movimientos a realizar para mover 1, 2 ó 3 discos.

Mover 1 disco: A → C
Mover 2 discos: [A → B]; A→ C; [B → C]
Mover 3 discos: [A → C; A → B; C→ B]; A→ C; [B → A; B → C; A → C]

En general, para mover *n* discos son necesarios los siguientes pasos:
[Mover *n*-1 discos de A a B]; A→ C; [Mover *n*-1 discos de B a C]

Gráficamente,



Por lo tanto, un procedimiento recursivo que resuelve el problema de las torres de Hanoi puede ser el siguiente:

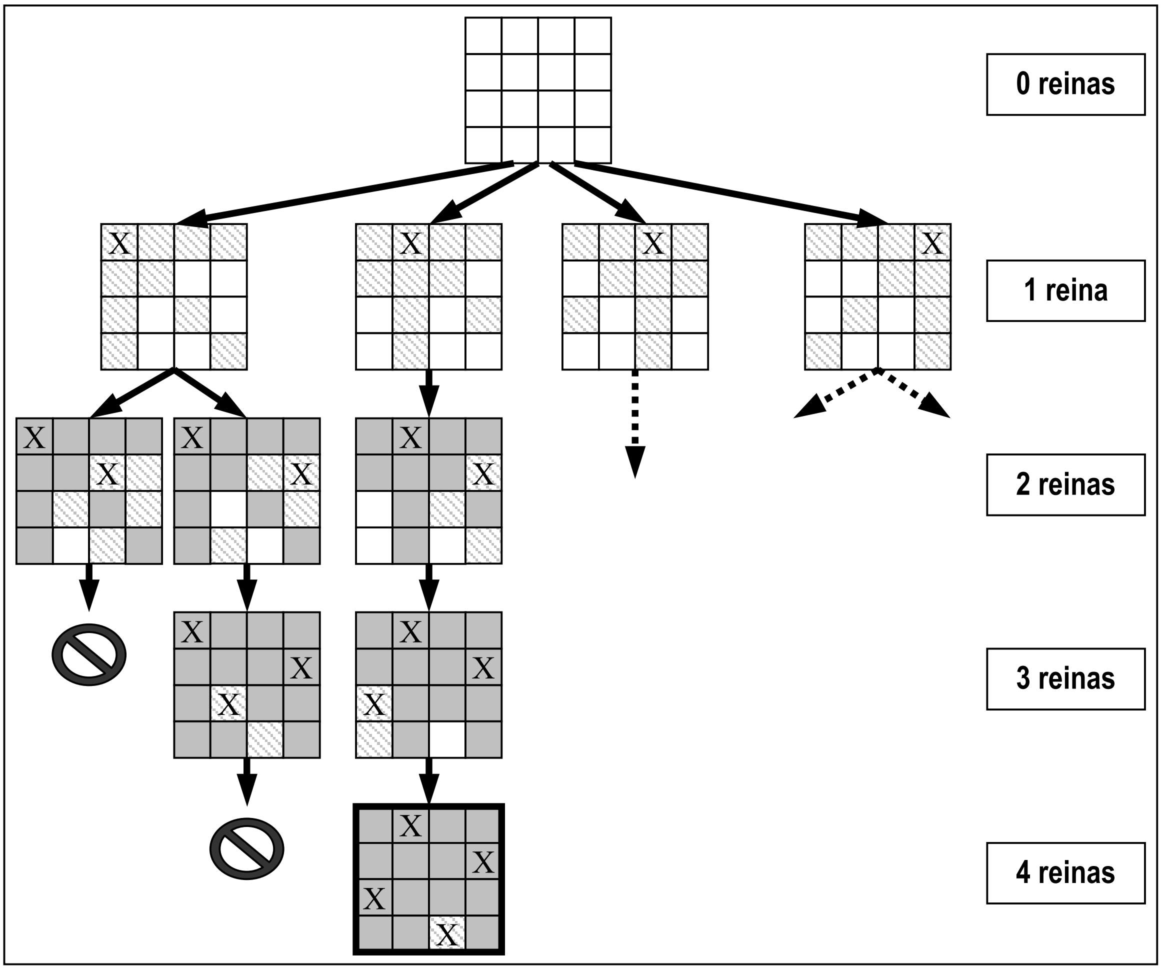
```
{
Procedimiento: hanoi
Datos de entrada: Una letra (A) que identifica la aguja origen de los
discos, una letra (C) que identifica la aguja destino de los discos,
el número de discos (n) a mover, y una letra (B) que identifica la
aguja intermedia utilizada para el movimiento de los discos. Como
precondición se considera que el número de discos debe ser no
negativo, es decir, n >= 0.
Datos de salida: No tiene, ya que el procedimiento escribe
directamente la secuencia de movimientos (de discos individuales) que
hay que realizar y no calcula ningún dato
}
Procedimiento hanoi (E A, C: Caracter; E n: Entero; E B: Caracter);
 Inicio
                        {la solución para el caso general es recursiva}
    Si (n > 0)
       Entonces
             hanoi(A, B, n-1, C);
             Escribir(A, ' → ', C, '; ');
             hanoi(B, C, n-1, A);
    FinSi
       {en el caso base, el procedimiento no hace nada, simplemente se
                                         finaliza la llamada recursiva}
 FinProcedimiento
```

9. **Las 8-reinas**

Este ejemplo ilustra el uso de la recursividad para solucionar problemas basándose en la técnica conocida como *backtracking* o algoritmos de "vuelta atrás". Esta técnica no es más que la aplicación de una estrategia de prueba y error sistemáticamente, para la resolución de problemas para los cuales no se tiene un método específico (algoritmo) para su resolución. Esta situación se da en problemas como el de la 8-reinas (consistente en colocar 8 reinas en un tablero de ajedrez de dimensión 8 x 8 sin que se amenacen) encontrar la salida en un laberinto, etc.

Todos estos problemas se caracterizan porque no existe un algoritmo explícito que los resuelva. Ahora bien, en cada estado intermedio del proceso se tiene un conjunto de posibles acciones a realizar (situar una reina en una de las columnas disponibles, elegir un camino u otro en el laberinto). Pues bien, la técnica de *backtracking* selecciona una de las posibles opciones y actualiza el estado, como si tal acción se hubiese realizado. Si siguiendo este método, se alcanza la solución (se colocan las reinas en todas las filas o se alcanza la salida del laberinto) el algoritmo termina. Si por el contrario, se alcanza un callejón sin salida (no es posible colocar una reina en una columna porque ya están todas las posiciones dominadas por reinas colocadas en pasos anteriores, no se tienen caminos por los que seguir o se cierra un ciclo en el camino), el algoritmo es capaz de, mediante el uso de la recursividad (realmente, mediante el uso de la pila de ejecución), recuperar los estados anteriores (deshaciendo las operaciones previas) y probar otras alternativas.

Indirectamente, la solución del problema se plantea como un problema de búsqueda, que recorre el espacio de todas las posibles soluciones. En el caso del problema de las 8-reinas suponiendo que: partiendo de una situación inicial en la que el tablero está vacío y que se trata de ir colocando una reina fila a fila (comenzando por la primera fila), el espacio de búsqueda viene dado por un árbol. Este árbol puede observarse en la siguiente gráfica que, por motivos de simplicidad, considera el mismo problema, pero para 4 reinas en vez de 8.



Como se ilustra en el gráfico, un algoritmo podría ir colocando las reinas fila a fila. En cada paso (en cada fila), el algoritmo dispondrá de un conjunto de casillas habilitadas para colocar una reina. En caso de haber varias, probaría la primera disponible y actualizaría el estado del tablero. Es decir, actualizaría el estado de las casillas en relación con que se pueda o no colocar otra reina en dicha casilla. Si en un punto, no hay una casilla disponible donde colocar una reina y no se han colocado todas, se llega a una vía muerta, con lo que hay que deshacer el último movimiento y probar el siguiente.

Como se ha visto, la implementación del algoritmo depende del estado del proceso en todo momento, el cual incluye:

- la posición en la que está colocada una reina.
- saber si una casilla del tablero está libre o no.

Para representar la posición en la que está colocada una reina, se supone que se utilizará un `array` de dimensión 8 (denominado "reina"), donde el índice determina la fila del tablero y su valor, es decir, `reina[fila]`, la columna ocupada por la reina en la fila especificada. Así pues, se supone la definición del siguiente `array`:

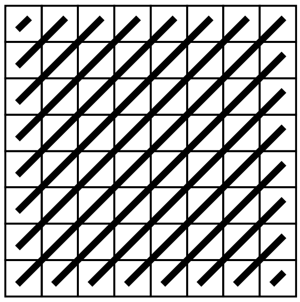
```
Reina: Array [1..8] de Entero;
```

Para saber si una casilla está libre o no, en vez de considerar casillas individuales, es mejor saber si una fila, columna o diagonal está o no, ya dominada por una reina. En el caso de las filas, no es necesario mantener el estado, pues se van colocando fila a fila, y una vez colocada una reina en una fila, esta fila no vuelve a considerarse. Por el contrario, si es necesario controlar el estado de las columnas, para lo cual se define el siguiente `array`:

```
EC : Array [1..8] de Lógico;
```

de forma que `EC[j]` almacenará el valor `Verdad` si ya está dominada esa columna por otra reina, o `Falso`, si no lo está.

En el caso de las diagonales, se necesita diferenciar entre diagonales directas (/) y diagonales inversas (\). Las diagonales directas aparecen en el siguiente gráfico,



Si se observa, los elementos de una misma diagonal directa se caracterizan porque la suma de sus coordenadas (i, j), $i + j$, es igual a una cantidad fija, para cada diagonal. En concreto, si las consideramos de arriba-abajo, y de izquierda a derecha, las coordenadas de las casillas de la primera diagonal suman 2, las de las segunda suman 3 y así, sucesivamente, hasta la última que suman 16. Es decir, hay 15 diagonales directas numeradas desde 2 hasta 16. Por lo tanto, puede definirse el `array`:

```
EDD: Array[2..16] de Lógico;
```

de forma que, `EDD[d]` almacenará un valor de `Verdad`, si la diagonal directa "*d*" ya está dominada por una reina y `Falso`, en caso contrario.

Análogamente, las diagonales inversas, se caracterizan porque las coordenadas (i, j) de los elementos de una misma diagonal cumplen que i – j es igual a una cantidad fija. En concreto, las diagonales pueden numerarse (de arriba-abajo y de derecha a izquierda) desde –7 hasta 7 (como se aprecia, también hay 15 diagonales inversas). Así pues, se define el `array`:

```
EDI: Array[-7..7] de Lógico;
```

de modo que, `EDD[d]` almacenará un valor de `Verdad`, si la diagonal inversa "*d*" ya está dominada por una reina y `Falso`, en caso contrario.

Por lo tanto, el algoritmo que resuelve el problema de las 8 reinas podría ser el siguiente:

```
Algoritmo ocho_reinas;

Constantes
   LIBRE = Falso;
   OCUPADA = Verdad;
Tipos
   TSolucion = Array [1..8] de Entero;
   TEstado = Registro
      EC: Array[1..8] de Logico;
      EDD: Array[2..16] de Logico;
      EDI: Array[-7..7] de Logico;
   FinRegistro;


Funcion Ensayar(  E fil: Entero;
                  E/S sol: TSolucion;
                  E/S estado: TEstado): Logico;
Variables
   col: Entero;
   exito: Logico;
Inicio
   Si (fil > 8)
      Entonces  Retorna Verdad;             {Se han colocado todas las reinas}
      Sino
            exito ← Falso;
            col ← 1;
                              {Mientras no se agoten todas las posibilidades y
                                                     no se haya terminado ...}
            Mientras (col<=8) AND (NOT (exito) ) Hacer
                              {si es una posición no dominada por otras reinas ...}
               Si (estado.EC[col]=LIBRE) AND (estado.EDD[fil+col]=LIBRE) AND
                  (estado.EDI[fil-col]=LIBRE)
                  Entonces      {Posicionar reina fila "fil" en la columna "col"}
                     sol[fil] ← col;
                     estado.EC[col] ← OCUPADA;
                     estado.EDD[fil+col] ← OCUPADA;
                     estado.EDI[fil-col] ← OCUPADA;
                     exito ← Ensayar(fil+1, sol, estado);
                                          {Si no se llega a una solución ... }
                     Si NOT(exito)
                        Entonces
                                                    {deshacer posicionamiento}
                           sol[fil] ← 0;
                           estado.EC[col] ← LIBRE;
                           estado.EDD[fil+col] ← LIBRE;
                           estado.EDI[fil-col] ← LIBRE;
                                                    {probar siguiente columna}
                           col ← col + 1;
                     FinSi;
               FinSi;
            FinMientras;
            Retorna exito;
   FinSi;
FinFuncion;
```

```
Procedimiento InicializarSolucion (S sol: TSolucion);
   Variables
      fil: Entero;
   Inicio
      Desde fil←1 Hasta 8 Hacer
         sol[fil] ← 0;
      FinDesde;
   FinProcedimiento;


Procedimiento InicializarEstado (S estado: TEstado);
   Variables
      col, diag: Entero;
   Inicio
      Desde col ← 1 Hasta 8 Hacer
         estado.EC[col] ← LIBRE;
      FinDesde;
      Desde diag ← 2 Hasta 16 Hacer
         Estado.EDD[diag] ← LIBRE;
      FinDesde;
      Desde diag ← -7 Hasta 7 Hacer
         Estado.EDI[diag] ← LIBRE;
      FinDesde;
   FinProcedimiento;


Procedimiento EscribirSolucion(E sol: Tsolucion);
   Variables
      fil, col: Entero;
   Inicio
      Desde fil ← 1 hasta 8 Hacer
         Desde col ← 1 hasta 8 Hacer
            Si (col = sol[fil])
               Entonces  Escribir('[X]')
               Sino      Escribir('[ ]');
            FinSi;
         FinDesde
         Escribir(SALTO_LINEA);
      FinDesde;
   FinProcedimiento;


Variables                                    {Variables globales del algoritmo}
   exito: Logico;
   Reina: TSolucion;
   Estado: TEstado;
                                                         {Algoritmo principal}
Inicio
   InicializarSolucion(Reina);
   InicializarEstado(Estado);
   exito ← Ensayar(1, Reina, Estado);
   Si (exito)
      Entonces  EscribirSolucion(Reina)
      Sino      Escribir('No se ha encontrado solución');
   FinSi;
Fin
```