# §3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

## 3.1. KHÁI NIỆM VỀ ĐỆ QUY

Ta nói một đối tượng là đệ quy nếu nó được định nghĩa qua chính nó hoặc một đối tượng khác cùng dạng với chính nó bằng quy nạp.

Ví dụ: Đặt hai chiếc gương cầu đối diện nhau. Trong chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ nhất… Ở một góc nhìn hợp lý, ta có thể thấy một dãy ảnh vô hạn của cả hai chiếc gương.

Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngồi bên máy vô tuyến truyền hình và cứ như thế…

Trong toán học, ta cũng hay gặp các định nghĩa đệ quy:

- Giai thừa của n (n!): Nếu n = 0 thì n! = 1; nếu n > 0 thì n! = n.(n-1)!
- Ký hiệu số phần tử của một tập hợp hữu hạn S là |S|: Nếu S = ∅ thì |S| = 0; Nếu S ≠ ∅ thì tất có một phần tử x ∈ S, khi đó |S| = |S\{x}| + 1. Đây là phương pháp định nghĩa tập các số tự nhiên.

## 3.2. GIẢI THUẬT ĐỆ QUY

Nếu lời giải của một bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy. Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: P' tuy có dạng giống như P, nhưng theo một nghĩa nào đó, nó phải "nhỏ" hơn P, dễ giải hơn P và việc giải nó không cần dùng đến P.

Trong Pascal, ta đã thấy nhiều ví dụ của các hàm và thủ tục có chứa lời gọi đệ quy tới chính nó, bây giờ, ta tóm tắt lại các phép đệ quy trực tiếp và tương hỗ được viết như thế nào:

Định nghĩa một hàm đệ quy hay thủ tục đệ quy gồm hai phần:

- Phần neo (anchor): Phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.
- Phần đệ quy: Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ quy giải những bài toán con đó. Khi đã có lời giải (đáp số) của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

Phần đệ quy thể hiện tính "quy nạp" của lời giải. Phần neo cũng rất quan trọng bởi nó quyết định tới tính hữu hạn dừng của lời giải.

## 3.3. VÍ DỤ VỀ GIẢI THUẬT ĐỆ QUY

### 3.3.1. Hàm tính giai thừa

```
function Factorial(n: Integer): Integer; {Nhận vào số tự nhiên n và trả về n!}
begin
  if n = 0 then Factorial := 1 {Phần neo}
  else Factorial := n * Factorial(n - 1); {Phần đệ quy}
end;
```

Ở đây, phần neo định nghĩa kết quả hàm tại n = 0, còn phần đệ quy (ứng với n > 0) sẽ định nghĩa kết quả hàm qua giá trị của n và giai thừa của n - 1.

Ví dụ: Dùng hàm này để tính 3!, trước hết nó phải đi tính 2! bởi 3! được tính bằng tích của 3 * 2!. Tương tự để tính 2!, nó lại đi tính 1! bởi 2! được tính bằng 2 * 1!. Áp dụng bước quy nạp này thêm một lần nữa, 1! = 1 * 0!, và ta đạt tới trường hợp của phần neo, đến đây từ giá trị 1 của 0!, nó tính được 1! = 1*1 = 1; từ giá trị của 1! nó tính được 2!; từ giá trị của 2! nó tính được 3!; cuối cùng cho kết quả là 6:

```
3! = 3 * 2!
     ↓
     2! = 2 * 1!
          ↓
          1! = 1 * 0!
               ↓
               0! = 1
```

### 3.3.2. Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán đặt ra như sau:

❖ Các con thỏ không bao giờ chết

❖ Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái)

❖ Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới

Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, n = 5, ta thấy:

❖ Giữa tháng thứ 1:1 cặp (ab) (cặp ban đầu)

❖ Giữa tháng thứ 2:1 cặp (ab) (cặp ban đầu vẫn chưa đẻ)

❖ Giữa tháng thứ 3:2 cặp (AB)(cd) (cặp ban đầu đẻ ra thêm 1 cặp con)

❖ Giữa tháng thứ 4:3 cặp (AB)(cd)(ef) (cặp ban đầu tiếp tục đẻ)

❖ Giữa tháng thứ 5:5 cặp (AB)(CD)(ef)(gh)(ik) (cả cặp (AB) và (CD) cùng đẻ)

Bây giờ, ta xét tới việc tính số cặp thỏ ở tháng thứ n: F(n)

Nếu mỗi cặp thỏ ở tháng thứ n - 1 đều sinh ra một cặp thỏ con thì số cặp thỏ ở tháng thứ n sẽ là:

$$F(n) = 2 * F(n - 1)$$

Nhưng vấn đề không phải như vậy, trong các cặp thỏ ở tháng thứ n - 1, chỉ có những cặp thỏ đã có ở tháng thứ n - 2 mới sinh con ở tháng thứ n được thôi. Do đó F(n) = F(n - 1) + F(n - 2) (= số cũ + số sinh ra). Vậy có thể tính được F(n) theo công thức sau:

F(n) = 1 nếu n ≤ 2

F(n) = F(n - 1) + F(n - 2) nếu n > 2

```
function F(n: Integer): Integer; {Tính số cặp thỏ ở tháng thứ n}
begin
  if n ≤ 2 then F := 1 {Phần neo}
  else F := F(n - 1) + F(n - 2); {Phần đệ quy}
end;
```

### 3.3.3. Giả thuyết của Collatz

Collatz đưa ra giả thuyết rằng: với một số nguyên dương X, nếu X chẵn thì ta gán X := X div 2; nếu X lẻ thì ta gán X := X * 3 + 1. Thì sau một số hữu hạn bước, ta sẽ có X = 1.

Ví du: X = 10, các bước tiến hành như sau:

1. X = 10 (chẵn) ⇒ X := 10 div 2; (5)
2. X = 5 (lẻ) ⇒ X := 5 * 3 + 1; (16)
3. X = 16 (chẵn) ⇒ X := 16 div 2; (8)
4. X = 8 (chẵn) ⇒ X := 8 div 2 (4)
5. X = 4 (chẵn) ⇒ X := 4 div 2 (2)
6. X = 2 (chẵn) ⇒ X := 2 div 2 (1)

Cứ cho giả thuyết Collatz là đúng đắn, vấn đề đặt ra là: Cho trước số 1 cùng với hai phép toán * 2 và div 3, hãy sử dụng một cách hợp lý hai phép toán đó để biến số 1 thành một giá trị nguyên dương X cho trước.

Ví dụ: X = 10 ta có 1 * 2 * 2 * 2 * 2 div 3 * 2 = 10.

Dễ thấy rằng lời giải của bài toán gần như thứ tự ngược của phép biến đổi Collatz: Để biểu diễn số X > 1 bằng một biểu thức bắt đầu bằng số 1 và hai phép toán "* 2", "div 3". Ta chia hai trường hợp:

Nếu X chẵn, thì ta tìm cách biểu diễn số X div 2 và viết thêm phép toán * 2 vào cuối

Nếu X lẻ, thì ta tìm cách biểu diễn số X * 3 + 1 và viết thêm phép toán div 3 vào cuối

```
procedure Solve(X: Integer); {In ra cách biểu diễn số X}
begin
  if X = 1 then Write(X) {Phần neo}
  else {Phần đệ quy}
    if X mod 2 = 0 then {X chẵn}
      begin
        Solve(X div 2); {Tìm cách biểu diễn số X div 2}
        Write(' * 2'); {Sau đó viết thêm phép toán * 2}
      end
    else {X lẻ}
      begin
        Solve(X * 3 + 1); {Tìm cách biểu diễn số X * 3 + 1}
        Write(' div 3'); {Sau đó viết thêm phép toán div 3}
      end;
```

```
end;
```

Trên đây là cách viết đệ quy trực tiếp, còn có một cách viết đệ quy tương hỗ như sau:

```
procedure Solve(X: Integer); forward; {Thủ tục tìm cách biểu diễn số X: Khai báo trước, đặc tả sau}

procedure SolveOdd(X: Integer); {Thủ tục tìm cách biểu diễn số X > 1 trong trường hợp X lẻ}
begin
  Solve(X * 3 + 1);
  Write(' div 3');
end;

procedure SolveEven(X: Integer); {Thủ tục tìm cách biểu diễn số X trong trường hợp X chẵn}
begin
  Solve(X div 2);
  Write(' * 2');
end;

procedure Solve(X: Integer); {Phần đặc tả của thủ tục Solve đã khai báo trước ở trên}
begin
  if X = 1 then Write(X)
  else
    if X mod 2 = 1 then SolveOdd(X)
    else SolveEven(X);
end;
```

Trong cả hai cách viết, để tìm biểu diễn số X theo yêu cầu chỉ cần gọi Solve(X) là xong. Tuy nhiên trong cách viết đệ quy trực tiếp, thủ tục Solve có lời gọi tới chính nó, còn trong cách viết đệ quy tương hỗ, thủ tục Solve chứa lời gọi tới thủ tục SolveOdd và SolveEven, hai thủ tục này lại chứa trong nó lời gọi ngược về thủ tục Solve.

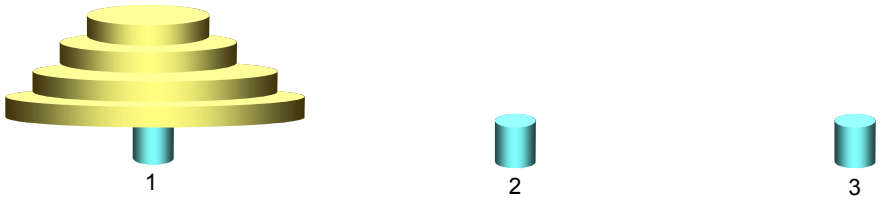
Đối với những bài toán nêu trên, việc thiết kế các giải thuật đệ quy tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa quy nạp của hàm đó được xác định dễ dàng.

Nhưng không phải lúc nào phép giải đệ quy cũng có thể nhìn nhận và thiết kế dễ dàng như vậy. Thế thì vấn đề gì cần lưu tâm trong phép giải đệ quy?. Có thể tìm thấy câu trả lời qua việc giải đáp các câu hỏi sau:

- ❖ Có thể định nghĩa được bài toán dưới dạng phối hợp của những bài toán cùng loại nhưng nhỏ hơn hay không ? Khái niệm "nhỏ hơn" là thế nào ?
- ❖ Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp tầm thường và có thể giải ngay được để đưa vào phần neo của phép giải đệ quy

### 3.3.4. Bài toán Tháp Hà Nội

Đây là một bài toán mang tính chất một trò chơi, tương truyền rằng tại ngôi đền Benares có ba cái cọc kim cương. Khi khai sinh ra thế giới, thượng đế đặt n cái đĩa bằng vàng chồng lên nhau theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt trên một chiếc cọc.




**Hình 6: Tháp Hà Nội**

Các nhà sư lần lượt chuyển các đĩa sang cọc khác theo luật:

- ❖ Khi di chuyển một đĩa, phải đặt nó vào một trong ba cọc đã cho
- ❖ Mỗi lần chỉ có thể chuyển một đĩa và phải là đĩa ở trên cùng
- ❖ Tại một vị trí, đĩa nào mới chuyển đến sẽ phải đặt lên trên cùng
- ❖ Đĩa lớn hơn không bao giờ được phép đặt lên trên đĩa nhỏ hơn (hay nói cách khác: một đĩa chỉ được đặt trên cọc hoặc đặt trên một đĩa lớn hơn)

Ngày tận thế sẽ đến khi toàn bộ chồng đĩa được chuyển sang một cọc khác.

Trong trường hợp có 2 đĩa, cách làm có thể mô tả như sau:

Chuyển đĩa nhỏ sang cọc 3, đĩa lớn sang cọc 2 rồi chuyển đĩa nhỏ từ cọc 3 sang cọc 2.

Những người mới bắt đầu có thể giải quyết bài toán một cách dễ dàng khi số đĩa là ít, nhưng họ sẽ gặp rất nhiều khó khăn khi số các đĩa nhiều hơn. Tuy nhiên, với tư duy quy nạp toán học và một máy tính thì công việc trở nên khá dễ dàng:

Có n đĩa.

- ❖ Nếu n = 1 thì ta chuyển đĩa duy nhất đó từ cọc 1 sang cọc 2 là xong.
- ❖ Giả sử rằng ta có phương pháp chuyển được n - 1 đĩa từ cọc 1 sang cọc 2, thì cách chuyển n - 1 đĩa từ cọc x sang cọc y ($1 \le x, y \le 3$) cũng tương tự.
- ❖ Giả sử ràng ta có phương pháp chuyển được n - 1 đĩa giữa hai cọc bất kỳ. Để chuyển n đĩa từ cọc x sang cọc y, ta gọi cọc còn lại là z (=6 - x - y). Coi đĩa to nhất là … cọc, chuyển n - 1 đĩa còn lại từ cọc x sang cọc z, sau đó chuyển đĩa to nhất đó sang cọc y và cuối cùng lại coi đĩa to nhất đó là cọc, chuyển n - 1 đĩa còn lại đang ở cọc z sang cọc y chồng lên đĩa to nhất.

Cách làm đó được thể hiện trong thủ tục đệ quy dưới đây:

```
procedure Move(n, x, y: Integer); {Thủ tục chuyển n đĩa từ cọc x sang cọc y}
begin
  if n = 1 then WriteLn('Chuyển 1 đĩa từ ', x, ' sang ', y)
  else {Để chuyển n > 1 đĩa từ cọc x sang cọc y, ta chia làm 3 công đoạn}
    begin
      Move(n - 1, x, 6 - x - y); {Chuyển n - 1 đĩa từ cọc x sang cọc trung gian}
      Move(1, x, y); {Chuyển đĩa to nhất từ x sang y}
      Move(n - 1, 6 - x - y, y); {Chuyển n - 1 đĩa từ cọc trung gian sang cọc y}
    end;
end;
```

Chương trình chính rất đơn giản, chỉ gồm có 2 việc: Nhập vào số n và gọi Move(n, 1, 2).

## 3.4. HIỆU LỰC CỦA ĐỆ QUY

Qua các ví dụ trên, ta có thể thấy đệ quy là một công cụ mạnh để giải các bài toán. Có những bài toán mà bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn bài toán tính giai thừa hay tính số Fibonacci. Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó, có nhiều bài toán mà việc thiết kế giải thuật đệ quy đơn giản hơn nhiều so với lời giải lặp và trong một số trường hợp chương trình đệ quy hoạt động nhanh hơn chương trình viết không có đệ quy. Giải thuật cho bài Tháp Hà Nội và thuật toán sắp xếp kiểu phân đoạn (QuickSort) mà ta sẽ nói tới trong các bài sau là những ví dụ.

Có một mối quan hệ khăng khít giữa đệ quy và quy nạp toán học. Cách giải đệ quy cho một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến (neo) rồi thiết kế làm sao để lời giải của bài toán được suy ra từ lời giải của bài toán nhỏ hơn cùng loại như thế. Tương tự như vậy, quy nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng minh tính chất đó đúng với một số trường hợp cơ sở (thường người ta chứng minh nó đúng với 0 hay đúng với 1) và sau đó chứng minh tính chất đó sẽ đúng với n bất kỳ nếu nó đã đúng với mọi số tự nhiên nhỏ hơn n.

Do đó ta không lấy làm ngạc nhiên khi thấy quy nạp toán học được dùng để chứng minh các tính chất có liên quan tới giải thuật đệ quy. Chẳng hạn: Chứng minh số phép chuyển đĩa để giải bài toán Tháp Hà Nội với n đĩa là $2^n$-1:

Rõ ràng là tính chất này đúng với n = 1, bởi ta cần $2^1 - 1 = 1$ lần chuyển đĩa để thực hiện yêu cầu

Với n > 1; Giả sử rằng để chuyển n - 1 đĩa giữa hai cọc ta cần $2^{n-1} - 1$ phép chuyển đĩa, khi đó để chuyển n đĩa từ cọc x sang cọc y, nhìn vào giải thuật đệ quy ta có thể thấy rằng trong trường hợp này nó cần $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$ phép chuyển đĩa. Tính chất được chứng minh đúng với n

Vậy thì công thức này sẽ đúng với mọi n.

Thật đáng tiếc nếu như chúng ta phải lập trình với một công cụ không cho phép đệ quy, nhưng như vậy không có nghĩa là ta bó tay trước một bài toán mang tính đệ quy. Mọi giải thuật đệ quy đều có cách thay thế bằng một giải thuật không đệ quy (khử đệ quy), có thể nói được như vậy bởi tất cả các chương trình con đệ quy sẽ đều được trình dịch chuyển thành những mã lệnh không đệ quy trước khi giao cho máy tính thực hiện.

Việc tìm hiểu cách khử đệ quy một cách "máy móc" như các chương trình dịch thì chỉ cần hiểu rõ cơ chế xếp chồng của các thủ tục trong một dây chuyền gọi đệ quy là có thể làm được. Nhưng muốn khử đệ quy một cách tinh tế thì phải tuỳ thuộc vào từng bài toán mà khử đệ quy cho khéo. Không phải tìm đâu xa, những kỹ thuật giải công thức truy hồi bằng quy hoạch động là ví dụ cho thấy tính nghệ thuật trong những cách tiếp cận bài toán mang bản chất đệ quy để tìm ra một giải thuật không đệ quy đầy hiệu quả.

**Bài tập**

Bài 1

Viết một hàm đệ quy tính ước số chung lớn nhất của hai số tự nhiên a, b không đồng thời bằng 0, chỉ rõ đâu là phần neo, đâu là phần đệ quy.

Bài 2

Viết một hàm đệ quy tính $\binom{n}{k}$ theo công thức truy hồi sau:

$$\begin{cases} \binom{n}{0} = \binom{n}{n} = 1 \\ \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}; \ \forall k: 0<k<n \end{cases}$$

(Ở đây tôi dùng ký hiệu $\binom{n}{k}$ thay cho $C_n^k$ thuộc hệ thống ký hiệu của Nga)

Bài 3

Nêu rõ các bước thực hiện của giải thuật cho bài Tháp Hà Nội trong trường hợp n = 3.

Viết chương trình giải bài toán Tháp Hà Nội không đệ quy

Lời giải:

Có nhiều cách giải, ở đây tôi viết một cách "lạ" nhất với mục đích giải trí, các bạn tự tìm hiểu tại sao nó hoạt động đúng:

```
program HanoiTower;
const
  max = 30;
var
  Stack: array[1..3, 0..max] of Integer;
  nd: array[1..3] of Integer;
  RotatedList: array[0..2, 1..2] of Integer;
  n: Integer;
  i: LongInt;

procedure Init;
var
  i: Integer;
begin
  Stack[1, 0] := n + 1; Stack[2, 0] := n + 1; Stack[3, 0] := n + 1;
  for i := 1 to n do Stack[1, i] := n + 1 - i;
  nd[1] := n; nd[2] := 0; nd[3] := 0;
  if Odd(n) then
    begin
      RotatedList[0][1] := 1; RotatedList[0][2] := 2;
      RotatedList[1][1] := 1; RotatedList[1][2] := 3;
      RotatedList[2][1] := 2; RotatedList[2][2] := 3;
    end
  else
    begin
      RotatedList[0][1] := 1; RotatedList[0][2] := 3;
      RotatedList[1][1] := 1; RotatedList[1][2] := 2;
      RotatedList[2][1] := 2; RotatedList[2][2] := 3;
    end;
end;
```

```
procedure DisplayStatus;
var
  i: Integer;
begin
  for i := 1 to 3 do
    Writeln('Peg ', i, ': ', nd[i], ' disks');
end;

procedure MoveDisk(x, y: Integer);
begin
  if Stack[x][nd[x]] < Stack[y][nd[y]] then
    begin
      Writeln('Move one disk from ', x, ' to ', y);
      Stack[y][nd[y] + 1] := Stack[x][nd[x]];
      Inc(nd[y]);
      Dec(nd[x]);
    end
  else
    begin
      Writeln('Move one disk from ', y, ' to ', x);
      Stack[x][nd[x] + 1] := Stack[y][nd[y]];
      Inc(nd[x]);
      Dec(nd[y]);
    end;
end;

begin
  Write('n = '); Readln(n);
  Init;
  DisplayStatus;
  for i := 1 to 1 shl n - 1 do
    MoveDisk(RotatedList[(i - 1) mod 3][1], RotatedList[(i - 1) mod 3][2]);
  DisplayStatus;
end.
```