S Z E M L E

A rekurzív algoritmusok tanításáról

A rekurzív algoritmusok fontos eszközei a számítógépek programozásának. A mai programozási nyelvek és a mai hardver lehetőségek megszüntették a rekurzív hívások hátrányait, ezért használatuk nagyon megkönnyíti a programozó munkáját.

Rekurzív összefüggésekkel a matematikában gyakran találkozunk. Egyszerű példa erre a *Fibonacci*-sorozat meghatározása:

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, ha $>n$ 1.

Könnyű belátni, hogy a sorozat első tagjai: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 stb. Az F_n értékére a következő képlet adódik:

$$F_n = \frac{1}{\sqrt{5}}(\Phi^n - \overline{\Phi}^n), \quad \text{ahol} \quad \Phi = \frac{1+\sqrt{5}}{2} \quad \text{és} \quad \overline{\Phi} = \frac{1-\sqrt{5}}{2}$$

Nyilvánvaló, hogy a sorozatot úgy is meghatározhatnánk, hogy megadjuk a fenti képletet az F_n kiszámítására. A rekurzív definíció azonban sokkal egyszerűbb. Van tehát értelme annak, hogy rekurzív definíciókat, rekurzív képleteket adjunk meg.

Hasonló a helyzet a programozási nyelvekben is. Ha olyan eljárást vagy függvényt írunk, amely – valamilyen módon – önmagát hívja, akkor rekurzív hívásról beszélünk. Itt azonban bonyolultabb a helyzet, mivel vannak olyan programozási nyelvek, amelyek nem engedik meg a rekurzív hívást. Ekkor feltétlenül más megoldást kell választanunk. De ha a használt nyelv meg is engedi a rekurzív hívást, akkor is felvetődik a kérdés, érdemes-e használni, mivel a tárigény sokkal nagyobb, és a futási idő is megnőhet.

A rekurzió tanítását nem érdemes nagyon egyszerű feladatokkal kezdeni, mert akkor nem látszik a rekurzió fontossága, hasznossága. Olyan feladatot kell választani, amelynek nem rekurzív megoldása nem nyilvánvaló. Például nem érdemes a faktoriális kiszámításra rekurzív hívást alkalmazni, hiszen nyilvánvaló, hogy könnyen kiszámítható az első n szám összeszorzásával, egy egyszerű ciklusban (különben a faktoriális definíciója is ezt sugallja). Mégis sok programozási könyvben ezzel illusztrálják a rekurzív hívást!

Jó feladatnak tartom a rekurzív hívásra a Hanoi tornyai néven ismert feladatot. Hanoi egyik temploma előtt három oszlop található: egy arany, egy ezüst és egy réz oszlop. Az arany oszlopon száz darab könnyű korong van, nagyság szerint csökkenő sorrendben. Az egyik szerzetes azt a feladatot kapja, hogy helyezze át a korongokat az arany oszlopról a réz oszlopra úgy, hogy bármelyik oszlopot használhatja, de sohasem tehet nagyobb korongot kisebbre.

A szerzetes úgy gondolkodik, hogy ha a legokosabb tanítványát megkéri, hogy 99 korongot helyezzen át az ezüst oszlopra, akkor ő majd áthelyezi az utolsót az arany oszlopról a réz oszlopra, majd ismét megkéri a tanítványt, hogy most pedig helyezze át a 99 korongot az ezüst oszlopról a réz oszlopra. Ezzel a feladatot megoldotta. A legokosabb tanítvány hasonló módon jár el az ő legokosabb tanítványával, akivel áthelyeztet 98 korongot és így tovább. A megoldást a következőképpen írhatjuk le:

```
ELJÁRÁS Hanoi (n, A, E, R)
Ha n > 0 akkor   Hanoi (n-1, A, R, E)
                 Helyezd át: A –> R
                 Hanoi (n-1, E, A, R)
(Ha) vége
ELJÁRÁS VÉGE.
```

Az eljárás definíciós sorában n a korongok számát jelenti, A, E, R az arany, ezüst, illetve réz oszlopot. A Hanoi (n, A, E, R) jelentése: n korongot áthelyez A-ról E segítségével R-re.

A fenti megoldás azonnal adódik az ismertetett módszerből. Nagyon egyszerű, könnyen megérthető, és nem nyilvánvaló, hogy másképp, nem rekurzív hívással hogyan kellene megoldani. Illusztrálni lehet – adott n esetén – az eljáráshívásokat. Például, ha n = 3, akkor eredeti feladatunk:

Hanoi (3, A, E, R).

A fenti eljárás alapján ezt helyettesíteni lehet a következővel:

Hanoi (2, A, R, E)
A –> R
Hanoi (2, E, A, R).

Ugyancsak a fenti eljárás alapján Hanoi (2, A, R, E) helyettesíthető a következővel:

Hanoi (1, A, E, R)
A –> R
Hanoi (1, R, A, E).

Hasonlóképpen Hanoi (2, E, A, R) helyettesíthető:

Hanoi (1, E, R, A)
E –> R
Hanoi (1, A, E, R).

Mivel pl. Hanoi (1, A, E, R) egyenértékű az A –> R áthelyezéssel, behelyettesítve a fenti eljáráshívásokat az eredetibe, a következő áthelyezéseket kapjuk:

A –> R, A –> E, R –> E
A –> R
E –> A, E –> R, A –> R.

Az áthelyezések száma 7, általános esetben 2^n-1. Ez utóbbit könnyű igazolni a következő rekurzív összefüggés alapján:

H(n) = 2H(n-1), ha n > 1
H(1) = 1.

[H(n) az áthelyezések száma n korong esetében.]

A feladat könnyen programozható, például Turbo Pascalban:

```
program Hanoi tornyai;
var n:integer;
procedure Hanoi (n:integer; a, b, c:char); {a –>c, b segítségével}
begin
      if n > 0 then
      begin
          Hanoi (n-1, a, c, b);
          write (a, '–>', c, '  ');
          Hanoi (n-1, b, a, c);
      end;
end; {Hanoi}
BEGIN
      write ('Korongok száma: '); readln(n);
      Hanoi (n, 'A', 'B', 'C');
      readln;
END.
```

Természetesen érdekesebb bemutatni a feladatot grafikusan, amikor a lépéseket el is végezzük, megfelelően mozgatva a képernyőn a korongokat. Ez a program azonban sokkal hosszabb és bonyolultabb, ezért nem térünk ki rá.

Másik érdekes feladat, amelyiket szintén érdemes bemutatni, az *m* elem összes permutációját előállító feladat. Ezt lépésről lépésre építjük fel. Ha egy elemünk van, természetesen egyetlenegy permutáció lehetséges. Két elem permutációit úgy kaphatjuk meg, hogy a második elemet az első elé, majd utána helyezzük. Így megkapjuk az összes kételemű permutációt. Három elem esetében, mindegyik kételemű permutációból úgy kapunk három-három háromeleműt, hogy a harmadik elemet az első elé, az első és a második közé, majd a második után helyezzük. Így például az *ab* permutációból a *cab, acb, abc* permutációk nyerhetők.

Általában, ha van egy *n-1* elemű permutációnk, akkor az *n*-dik elemet sorra az első elé, az első és második közé, a második és harmadik közé stb. helyezzük, s így *n* újabb

n elemű permutációt kapunk. A következő eljárás egy $a_1, a_2, ..., a_{n-1}$ permutációból indul, és megadja az összes *n* elemű permutációt, majd mindegyiket tovább folytatja, ameddig megkapja az összes *m* elemű permutációt (elemekként a természetes számokat használjuk):

```
ELJÁRÁS perm (n,a)
Ha n < m akkor
    Minden i = 1, 2,....n értékre
        bk: = ak minden k = 1,2,...,i-1 értékre
        bi: = n
        bk: = ak-1 minden k = i+1, i+2,...,n értékre perm (n+1, b)
    (Minden)vége
(Ha) vége
ELJÁRÁS VÉGE.
```

A következő program egy a tömbben megőrzi az ábécé első m nagybetűjét, fordított sorrendben (hogy az első permutáció pl. ABCD, és ne DCBA legyen).

```
program permutálás; {m elem permutációja}
uses Crt;
type sor = array [1...20] of char;
var m, i: integer; {m globális változó}
    a:sor;
procedure perm(n:integer; b:sor);
var k,i: integer;
    c:sor;
begin {perm}
    if n<=m then
        begin
            for i: = 1 to n do
            begin
                for k: = 1 to i-1 do c[k]: = b [k];
                c [i]: = a [n]; {n-dik nagybetű}
                for k: = i+1 to n do c[k]: = b[k-1];
                perm(n+1,c);
            end;
        end
    else
        begin
            for k: = 1 to m do write (b[k]);
            writeln;
        end;
end; {perm}
BEGIN
    ClrScr;
    writeln ('Permutál m elemet');
    repeat write ('m='); readln(m) until m in [1...20];
    writeln;
    for i:=m downto 1 do a [m-i+1]:=Chr(64+i); {az abc nagybetűi}
    perm (2,a);
    repeat until KeyPressed;
END.
```

Ha m = 3 akkor a hívások a következőképpen alakulnak:

```
perm(2,a): BC perm(3,b): ABC
                         BAC
                         BCA
CB perm(3,b): ACB
              CAB
              CBA.
```

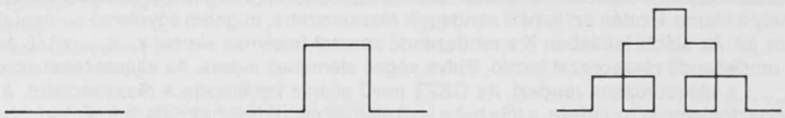
A második példában egyszerű rekurzív hívás szerepelt, amikor az eljárás (vagy más esetben függvény) önmagát hívja egyetlenegy helyen. A Hanoi tornyai elnevezésű feladatban a Hanoi eljárás kétszer hívta önmagát. Vannak olyan esetek is, amikor egy eljárás (vagy függvény) több helyen hívja önmagát, esetleg más eljárásokon (vagy függvényeken) keresztül.

Nézzünk meg néhány példát!

A *gyorsrendezés* (angolul quicksort) néven ismert algoritmus úgy rendez egy adott sorozatot (pl. növekvő sorrendbe), hogy először kettéosztja a sorozatot úgy, hogy az első részsorozat bármelyik eleme kisebb (esetleg egyenlő), mint a második részsorozat bármelyik eleme. Ezután ezt ismétli mindegyik részsorozatra, mígnem egyelemű sorozatokhoz jut. Az alábbi leírásban X a rendezendő sorozat (melynek elemei $x_1, x_2, \ldots x_n$), *b és j* a rendezendő részsorozat kezdő, illetve végső elemének indexe. Az eljárás tehát az x_b, $x_{b+1}, \ldots, x_j$ részsorozatot rendezi. Az OSZT nevű eljárás kettéosztja a részsorozatot, *k* a választóelemnek az indexe, a tőle balra levő elemek mind kisebbek nála, míg a tőle jobbra levők mind nagyobbak. Ez az elem tehát már a helyén van, eljárásunkat újra hívjuk az $x_b, \ldots, x_{k-1}$ és $x_{k+1}, \ldots, x_j$ részsorozatokra.

```
ELJÁRÁS GYORS (x, b, j);
    Ha b < j akkor
        OSZT (x, b, j, k);
        GYORS (x, b, k-1);
        GYORS (x, k+1, j);
    (Ha) vége
ELJÁRÁS VÉGE.
A teljes program Turbo Pascalban a következő:
program rendez;
const m = 50;
type sorozat = array [1...m] of integer;
var n,i : integer;
    x : sorozat;
Procedure GYORS (var x:sorozat; b,j:integer);
Var k : integer;
    Procedure OSZT (var x:sorozat; b,j : integer; var k : integer);
    Var y : integer;
    begin {OSZT}
        y : = x[b]; k : = b;
        While b<j do begin
            While (y  = x[j] and (b<j) do j : = j-1;
            x[k] : = x[j]; k : = j; if b<j then b : = b+1;
            While (x[b] <=y) and (b) do b : = b+1;
            x[k] : = x[b]; k : = b; if b<j then j : = j-1;
        end {while};
        x[k] : = y;
    end; {OSZT}
begin {GYORS}
    if b < j then begin
        OSZT (x, b, j, k);
        GYORS (x, b, k-1);
        GYORS (x, k+1, j);
    end {if}
end {GYORS}
BEGIN
    writeln('Sorozat rendezése növekvő sorrendbe');
    repeat write ('n='); readln (n) until n in [1...m];
    for i : = 1 to n do
        begin write ('x(', i, ')='); readln (x[i]) end;
    writeln ('Eredeti sorozat:');
    for i : = 1 to n do write (x[i], ' ');
    writeln;
    GYORS (x, 1, n);
    writeln ('Rendezett sorozat:');
    for i : = 1 to n do write (x[i], ' ');
    READLN;
END.
```

Rekurzió segítségével nagyon könnyen rajzolhatók ún. fraktálok. Vizsgáljuk meg, hogyan rajzolhatnánk le az ábrán látható S_1, S_2, S_3 görbéket! Hogyan általánosíthatjuk tetszőleges n-re? (Ezek egy adott fraktál különböző szintjei.)



S_1 S_2 S_3

Látható, hogy S_3 az S_2 görbéből, és annak elforgatásából könnyen előállítható. Ha A-val jelöljük azt az eljárást, amelyik S_1-et rajzolja le balról jobbra haladva, B-vel azt, amelyik a 90-kal elforgatott S_1-et, C-vel, illetve D-vel a 180°-kal, illetve 270°-kal elforgatott S_1-et lerajzolót, akkor feladatunkat könnyen leírhatjuk, figyelembe véve, hogy egy adott szint miként hívja az előbbi szint eljárásait.

A(n) : A(n-1), B(n-1), → D(n-1), A(n-1)
B(n) : B(n-1), C(n-1), ↑ A(n-1), B(n-1)
C(n) : C(n-1), D(n-1), ← B(n-1), C(n-1)
D(n) : D(n-1), A(n-1), ↓ C(n-1), D(n-1),

ahol a nyilak egy-egy, az adott iránnyal megrajzolt, összekötő szakaszt jelölnek. A Pascal-program a következő:

```
program rajz;
uses Graph;
var n, i, h, x, y: integer;
      Gd, Gm : integer;
procedure B(i : integer); forward;
procedure C(i : integer); forward;
procedure D(i : integer); forward;
procedure A(i : integer);
begin
      if i > 0 then
      begin   A(i-1);
              B(i-1); LineRel (h,0);
              D(i-1);
              A(i-1);
      end
end; {A}
procedure B;
begin
      if i > 0 then
      begin   B(i-1);
              C(i-1); LineRel (0,-h);
              A(i-1);
              B(i-1);
      end
end; {B}
procedure C;
begin
      if i > 0 then
      begin   C(i-1);
              D(i-1);
              B(i-1);
              C(i-1);
      end
end; {C}
procedure D;
begin
      if i > 0 then
      begin   D(i-1);
```

```
                A(i-1); LineRel (0,h);
                C(i-1);
                D(i-1);
        end
end; {D}
BEGIN
        repeat
            write ('n='); readln (n) (n a szintszám)
        until n in [1...9];
        Gd:=Detect; InitGraph (Gd, Gm, 'c:\tp\bgi');
        h:= GetMaxY; for i:=1 to n-1 do h:=h div 2;
        x:=1;
        for i:=1 to n-1 do x:=2*x;
        x:= GetmaxX div 2 – (x-1)*h-h div 2;
        y:= GetmaxY – 20;
        MoveTo (x,y); {x, y a kezdőpont koordinátái}
        A(n);
        readln;
        CloseGraph;
END.
```

Wirth könyvében még számos ehhez hasonló fraktál leírása megtalálható.

Ezek a rajzok nagyon könnyen elkészíthetők LOGO-ban is, nem egyszer sokkal egyszerűbb leírással. Itt jegyezzük meg, hogy a rekurzió fogalmának kialakításában a LOGO rendkívül előnyös.

A rekurzió tárgyalásakor mindenképpen meg kell említeni a visszalépéses (backtracking) algoritmust a feladatok megoldására. Ennek lényege, hogy a feladatot próbálkozással oldja meg, sorra megvizsgálva a lehetőségeket, amennyiben zsákutcába jut, visszalép addig a pontig, ahonnan újabb lehetőség választható. Ilyen feladat például lóugrással bejárni a sakktáblát, hogy minden mezőt csak egyszer érintsünk, vagy elhelyezni a sakktáblán nyolc királynőt úgy, ne üssék egymást stb. Mindkét ajánlott könyvünkben erre is több példa található.

Nem érdemes rekurzív algoritmust használni akkor, amikor a feladat egyszerűen megoldható iterációval. Vannak esetek, amikor pedig nem szabad rekurzívan megoldani egy feladatot, akkor sem, ha az történetesen rekurzívan van megadva. Jó példa erre a Fibonacci-sorozat. Ha egyszerűen alkalmazzuk a rekurzív képletet, bizonyos Fibonacci-számokat többször is ki fogunk számítani, pedig ez fölösleges. A feladat könnyen átírható nem rekurzív alakra. Kezdetben beállítjuk a P:=0 és R:=1 értékeket, majd az S := P+R, P :=R, R :=S ismétlésével tetszőleges Fibonacci-szám kiszámítható.

IRODALOM

C. H. A. Koster: Programozás felülnézetben. Műszaki Könyvkiadó, Bp., 1988.
N. Wirth: Algoritmusok + Adatstruktúrák = Programok. Műszaki Könyvkiadó, Bp., 1982.

KÁSA ZOLTÁN