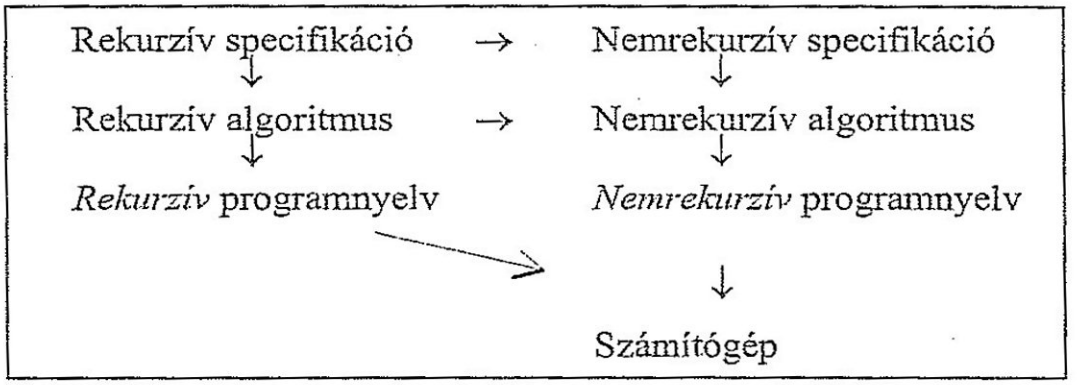
A rekurzióról

1. A lényeg (milyen lehetséges utakon juthatunk el a rekurzív specifikációtól egy –nemrekurzív– számítógépen futó programig)



2. A rekurzió megfogalmazási példái (példaalgoritmusok)

   = n!
   = Fibonacci-szám
   = Ackermann függvény
   = „n alatt a k"
   = Quicksort rendezés
   = Hanoi tornyai

3. A rekurzió megvalósítása *nemrekurzív* nyelveken (függvény-eljárás beépítése verem segítségével, paraméterátadás és lokális változók kezelése eljárások esetén; a fenti példák kódolása)

4. Rekurzió *rekurzív* nyelveken (azaz olyanokon, amelyek a rekurzív eljárások és függvények írását lehetővé teszik: Pascal, Logo)

5. Rekurzió és iteráció (átírás szükségessége, lehetősége; átírási szabályok és alkalmazásuk a példákra: rekurzív függvények, balrekurzió, jobbrekurzió)

6. Iteráció és rekurzió (rekurzívra írás szükségessége, lehetősége; átírási szabályok és alkalmazásuk példákra: ciklusok, ciklusban számolt függvények)

7. Programozási tételek rekurzívan (rekurzív specifikáció, rekurzív függvények és eljárások)

8. Gyorsrendezés — Quicksort (a rendezés variációi, összehasonlításuk)

9. Rekurzív adatszerkezetek (rekurzív adatszerkezet + rekurzív algoritmus, rekurzív adatszerkezet + nemrekurzív algoritmus, nemrekurzív adatszerkezet + rekurzív algoritmus, nemrekurzív adatszerkezet + nemrekurzív algoritmus)

Szlávi Péter *Zsakó László*

Módszeres programozás:

Rekurzió

μlógia 4

ELTE Informatikai Kar

5. bővített kiadás

Készült az *NJSZT* gondozásában
200 példányban
Felelős kiadó: dr. Kozma László
Sorozatszerkesztő: Szlávi Péter - Zsakó László

Tartalomjegyzék

Bevezetés

Ebben a füzetben a rekurzióval foglalkozunk. Ezzel kényes témába fogunk, a számítástechnikával foglalkozók ugyanis nagyon különbözően ítélik meg ezen témakör fontosságát, használhatóságát. Hogy miért is kényes? Sok érvet és ellenérvet lehet hallani, néha egymásnak ellentmondókat is, a rekurzió mellett, illetve ellen. Hogy jobban érzékelhessük, hogy milyen kemény fába vágjuk fejszénket, felsorolunk néhány rekurzióval szembeni, szokásos kifogást:

- A *rekurzió* a ciklusnál (ti. az iterációnál) bonyolultabb programszerkezet, s mindketten ugyanarra a célra szolgálnak: valamilyen tevékenységek ismételt végrehajtására.
- Mivel a számítógépek –például a gépi kódú programozás szintjén vizsgálva őket– nem tartalmaznak rekurzív lehetőségeket, ezért amit számítógéppel meg lehet oldani, azt rekurzió nélkül is meg lehet oldani.
- A rekurzió megvalósítása igen nagy központi tárigénnyel járhat.
- A rekurzív megoldás sokszor „rettenetes" futási időnövekedést okoz.
- Az, hogy a matematikusok bevett eszközei és módszerei épülnek e fogalomra, még nem indokolja, hogy a számítástechnikában is használjuk.
- A rekurzió valami nagyon bonyolult dolog, inkább ne foglalkozzunk vele!

Úgy gondoljuk, hogy határozottan két részre kell vágnia problémát. Egyrészt meg kell vizsgálni *a rekurzió gondolat-világát* (milyen problémák fölvetésénél, esetleg megoldásánál kínál kényelmes vagy –uram bocsá'– egyetlen lehetséges utat), másrészt foglalkozni kell a rekurzív algoritmusok és adatszerkezetek *számítógépes megvalósításával*! E merevnek tűnő szétválasztással meg szeretnénk akadályozni azt, hogy az esetleges megvalósítási nehézségek eleve elvegyék kedvünket egy, rekurzióval könnyen, egyszerűen megoldható feladat rekurzív megoldásától. Továbbá megadjuk azokat a feltételeket és szabályokat, amelyek ismeretében mechanikusan is átírhatjuk az eredeti (a feladathoz illő) rekurzív megoldást, a hatékonyabb, de többnyire a lényeget nem olyan világosan kifejező, klasszikusabb iteratív megfelelőjére.

E füzetben a programkészítés sok részterületéről lesz szó, amelyek különkülön is megérnének egy-egy füzetet. Most ezeket a rekurzió köti össze egy témává, így mindegyikből csak azt emeljük ki, ami a rekurzióval kapcsolatos.

1. A lényeg

A rekurzió mint eszköz felbukkan specifikációs, algoritmikus, implementációs (nyelvi) eszközként. Kezdjük a *specifikáció*nál, amellyel a matematikusok a problémák megoldását kezdik! A függvény egy igen hatékony absztrakciós eszköz, ugyanis jól kidolgozott formalizmussal rendelkezik és sokrétű, „bejáratott" operációval (függvényműveletteI) lehet építkezni. Mivel sok mindent (érts ezalatt bármilyen tevékenységsort) úgy lehet tekinteni, mint valamiféle függvényt, ami a kezdetben meglévő adatokhoz hozzárendeli a kívánt valamit, ezért nem reménytelen vállalkozás a függvény definiálást választani „általános" leíró eszközként. Ilyen ok miatt nincs mit csodálkozni azon, hogy példáink java része rekurzív függvény lesz, és elindulásként is az egyik legismertebbiket választottuk ki: a *faktoriális*t. Szokásos definíciója:

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

Ez csak az egyik függvénytípus legegyszerűbb példája. Érdemes kis kitérőt tenni a függvények „formális" világában, hogy megismerhessük mifélék lehetnek alakilag a függvények, milyen típusokba sorolhatók, vagy másként megfogalmazva: hogyan néz ki a kiszámítandó függvényt definiáló formula! Legyen a kiszámítandó függvény:

$$z = f(x)$$

Itt x a rendelkezésre álló adatot (adatokat), z: az eredményt, azaz a függvény szolgáltatta információt (adatokat), f: magát a formulát jelenti. Többnyire a függvényformula nem ilyen egyszerű szerkezetű, hanem elemibbekből építhető föl. Erre a dekompozícionálásra formailag az alábbi három lehetőség kínálkozik. Rutinosabb Olvasóink nyilván azonnal átlátnak a szitán, és kitalálják, hogy miért épp e három –valóban teljes– típusváltozatot írtuk föl. A magyarázat: épp e három szerkezet feleltethető meg a három algoritmikus szerkezetnek: szekvencia=kompozíció, alternatíva=elágazás, rekurzió=ciklus.

a, $f(x) = g \circ h(x)$

b, $f(x) = \begin{cases} g(x) & ha \ p(x) \\ h(x) & ha \ \neg p(x) \end{cases}$

c, $f(x) = \begin{cases} g(x) & ha \ p(x) \\ h \circ f \circ i(x) & ha \ \neg p(x) \end{cases}$

Ez utóbbi rekurzív formulát gyakran helyettesítik a Σ, Π, ∀, ∃ stb. jeleket tartalmazó (nem rekurzív) formulákkal. Ezt példázza a faktoriális másik, szokásosnak mondható definíciója:

$$n! = \begin{cases} \prod_{i=1}^{n} i & ha\ n > 0 \\ 1 & ha\ n = 0 \end{cases}$$

A nemrekurzív specifikációra való áttérésnél természetesen bizonyítani kell, hogy ez ekvivalens az eredeti rekurzív specifikációval. A bizonyítás módszere legtöbb esetben a teljes indukció.

A függvényt definiáló formulák természetszerűleg utalnak egyfajta lehetséges megvalósításra, „kiszámítási módra". Ezzel a rekurzió másik, fentebb említett aspektusból is felszínre került: a rekurzió mint algoritmikus eszköz. Az első két változat esetén rendkívül magától értetődő az algoritmusmegfelelő:

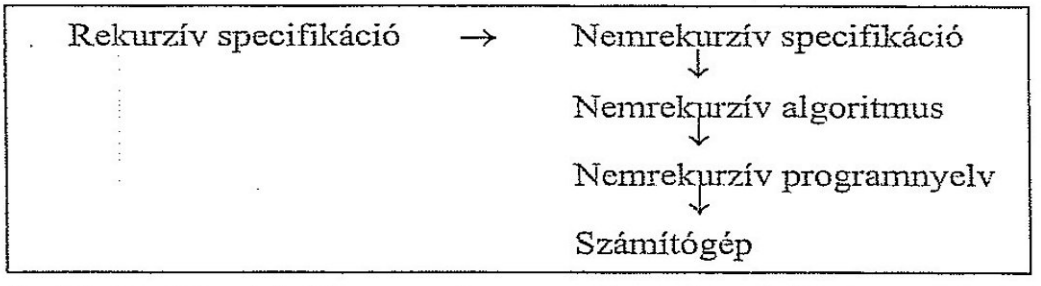
```
a,   y := h(x)
     z := g(y)

b,   Ha p(x) akkor  z:=g(x)
           különben z:=h(x)
```

azaz a függvénykompozíciónak a megvalósításban a szekvencia, a kapcsos zárójellel írt függvénynek pedig az elágazás felel meg. A rekurzív formulával definiált függvények kiszámítása azonban legtöbbször nem ebből kiindulva végezzük el, hanem a megfelelően átalakított nemrekurzív formulából. Így faktoriális számításra az *összegzés* algoritmus variánsaként (ld. [4]-ben) adódik a program (ez annak a „másik definíciónak"megfelelő algoritmus):

```
Faktoriális(n):
  f:=1 [ f=n! , ha n=0]
  Ciklus i=1-től n-ig
    f:=f*i
  Ciklus vége
  Faktoriális:=f
Függvény vége.
```

A szokásos út tehát a következő:



Ezután feltehetjük a kérdést: nem lenne-e sok esetben jobb, ha a számítógépet a rekurzív specifikációból kiindulva más úton érnénk el?

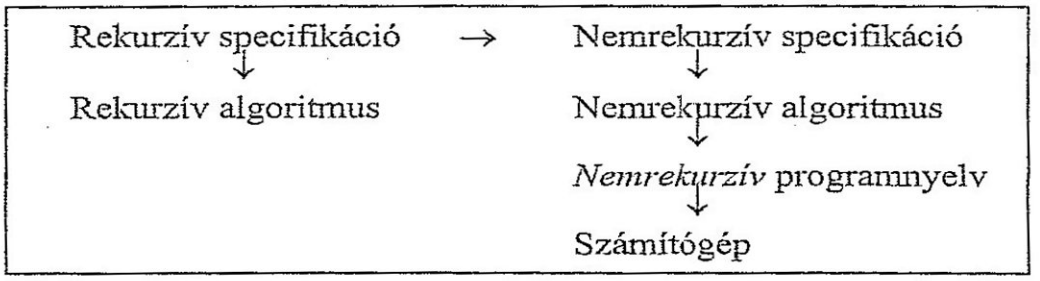
Most tehát ne térjünk ki a rekurzív definíció kihívása elől, hanem vágjunk bele bátran a definícióhoz „hű" megoldás megtalálásába, lesz ami lesz! Ragaszkodjunk az eredeti definícióhoz, arra építsük programunkat! A faktoriális definíciója, hasonlóan az általános c.-beli definícióhoz, két részre bomlik. Az egyik épít a már meglévő, és működő definícióra, és azt eggyel csökkentett értékkel (melynek előállítása az $i(x)$ függvény feladata) „újra meghívja". A másik –bízva abban, hogy valamikor „eljő az ő ideje"– kijáratot biztosít a(z előbbi) végtelenségig való önhívogatásból (a $p(x)$ logikai formula ez esetben 'x=0'). Vagyis valahogy így:

```
Faktoriális(n):
  Ha n=0 akkor  [a definíció nem rekurzív része]
                f:=1
         különben [a definíció rekurzív része]
                f:=n*Faktoriális(n-1)
  Elágazás vége
  Faktoriális:=f
Függvény vége.
```

Ezzel megkaptuk a konkrét, faktoriális függvény rekurzív formulájának megfelelő rekurzív algoritmust. (Rekurzív algoritmusokra számos példát adunk a következő fejezetben.) Ez alapján felírhatjuk a rekurzív algoritmusok általános szerkezetét is a $z=f(x)$ feladathoz:

```
c,  Ha p(x) akkor  z:=g(x)        [nemrekurzív rész]
          különben y:=f(i(x)); z:=h(y) [rekurzió]
```

Most tehát elindultunk egy másik úton a számítógép felé:



Ezzel elkészült a rekurzív algoritmus, de hogyan tovább? Ciklusokhoz szokott észjárásunk azt sugallja, hogy a kódolás ez alapján nem mehet gond nélkül. Épp ez jelzi a harmadik aspektus, a rekurzió implementációjának fontosságát. Nézzünk egy konkrét számpéldát, hogy pontosan érzékelhessük magát a problémát, és hogy rátalálhassunk a megoldás útjára! Legyen n=2 a faktoriális függvényeljárás hívásakor! Az algoritmus végrehajtását csak a ténylegesen végrehajtott utasításainak feltüntetésével követjük nyomon.

A konkrét értéket n helyére behelyettesítve, írjuk le a rekurzív algoritmus végrehajtandó részletét!

```
Faktoriális(2):
  Ha 2=0 akkor
       különben f:=2* ...
                      ... Faktoriális(2-1)
```

A rekurzió 1. szintje

Elérkezve az `f:=2*Faktoriális(1)` kifejezés kiértékeléséhez, akadunk fenn az első problémán.

1. probléma: hogy kerül ugyanazzal a kóddal megvalósított eljáráshoz most az 1 bemenő érték? (*A bemenő paraméter problematikája.*) A program algoritmusa szerint az n változó kapja értékül. Igen, de a hívó programban már kapott értéket (n=2). Nem lesz baj, ha azt elrontjuk? Folytassuk a hívott rutin megfelelő (azaz végrehajtott) részének „behelyettesítésével"! A behelyettesítés tényét azzal jelezzük, hogy zárójelek közé tesszük. E zárójelbeli rész kell, hogy végső soron egyetlen számmá (a függvény értékévé) redukálódjék. Tehát folytatódjék az f:=2* ... Faktoriális(1) ki fejezés még határozatlan részének kiértékelése a rekurzív algoritmus végrehajtandó része 1 értékkel való behelyettesítésével!

```
*(  Faktoriális(1):
      Ha 1=0 akkor
           különben f:=1* ...
                          ... Faktoriális(1-1)    )
```

A rekurzió 2. szintje

Újabb rekurzió következik, a probléma itt is ugyanaz. A részkifejezés újabb kiértékelendő részkifejezésbe torkolt. (Ezt jeleztük az újabb zárójelpár beskatulyázásával.)

```
*(  *(  Faktoriális(0):
          Ha 0=0 akkor f:=1
          Elágazás vége
          Faktoriális:=f
        Függvény vége.              )    )
```

A rekurzió 3. Szintje

A rekurziónak végeszakadt, irány: az érték visszaadása a hívó eljárásoknak!

```
*(  *(          1                   )    )
```

A rekurzió 3. szintjének Faktoriális(0) értéke

2. probléma: hogy kerül ez (ti. a függvényérték) a hívó eljárásban „felszínre", annak tudomására. Egyáltalán mi tekinthető a függvény értékének? (Ezt nevezhetjük a *függvényeljárások értékvisszaadás prob lémájának*.) Tegyük föl, a probléma

megoldható, hogy hogyan az most bennünket nem érdekel! A visszakerült függvényérték 1. Térjünk vissza a függvényhívást tartalmazó kifejezéshez!

```
* (          f:=1* (1)
          Elágazás vége                    )
```

A végrehajtás a 2. szinten folytatódik

3. probléma: az f-re vonatkozó értékadás valójában így nézett ki: f:=n*(...) Melyik n-ről van szó a sok közül? (*A lokális változók problémája.*) Majd megoldjuk!

```
* (     Faktoriális:=f
     Függvény vége.                       )
```

Visszatérés az 1. szintre

Tehát a függvény (visszaadandó) értéke:

```
* (          1                            )
```

2. szint értéke, ami az elsőn, mint a Faktoriális(1) függvény értéke kerül felszínre

Visszaérkeztünk a függvényeljárás elsőként hívott „verziójához".

```
              f:=2* (1)
  Elágazás vége
  Faktoriális:=f
Függvény vége.
```

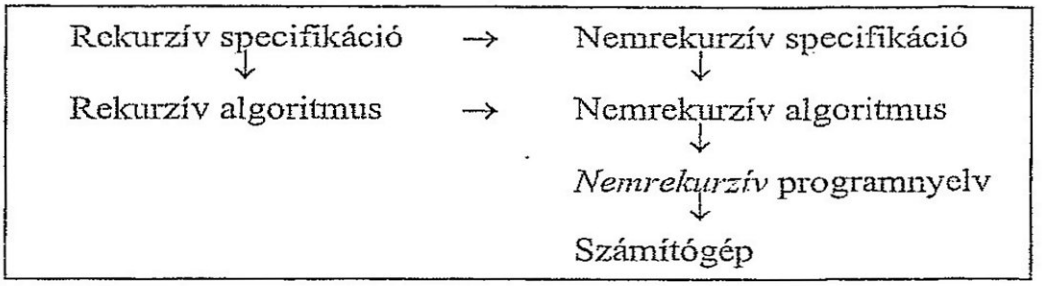
1. szint folytatása = a kiértékelt kifejezés értéke „felbukkant"

Ezzel a 2! értékét kiszámítottuk, miközben 3 megoldandó probléma fölött kellett –egyelőre– elsiklanunk, amelyek a következő pontokon jelentkeztek:

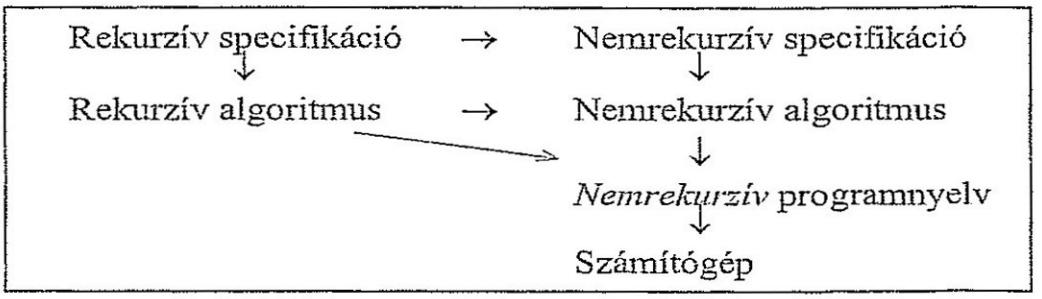
- bemenőértékhez jutásnál,
- függvényérték visszaadásánál,
- a változók egyedi voltában.

A problémák „tudatosodtak csupán", de jelen fejezetünknek célja éppen csak ennyi volt. Ezek megoldását kíséreljük meg majd a 3. fejezetben, ami előtt még néhány példán keresztül szoktatjuk magunkat a rekurzió algoritmikus eszközként való alkalmazásához.

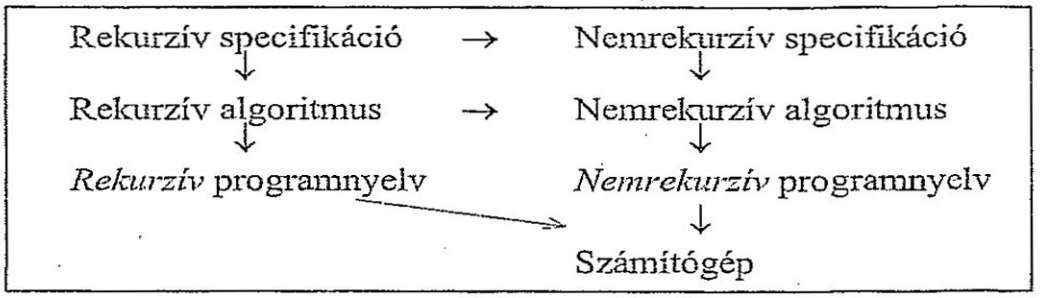
Nézzünk előre, milyen utakat követhetünk! A rekurzív algoritmusból készíthetünk nemrekurzív algoritmust egyes esetekben (5. fejezet):



A rekurzív algoritmust kódolhatjuk nemrekurzív programozási nyelven is (3. fejezet):



Végül a nemrekurzív algoritmust olyan programozási nyelven is kódolhatjuk, amelyen lehetőség van a rekurzióra (4. fejezet):



Látjuk tehát, mennyi út áll előttünk. Mindig a megoldandó feladattól függ, hogy melyik a járhatóbb, melyiken célszerű haladnunk.

Fejezetünket egy fontos gondolat megismétlésével zárjuk, amelyet amolyan közös alapnak szánunk a rekurzió vitában résztvevők számára. Ezen közös alapról szemlélve a rekurziót –azt hisszük– ha marad még egyáltalán vitatkozni való, az termékenyen tudja szolgálni a feladatmegoldás konkrét problémáit, illetőleg a nyelvek implementációs kérdéseit. Tehát a rekurzióról három szempontból érdemes beszélni:

- a specifikációban rekurzív formulával definiálhatjuk a kiszámítandó függvényt (a függvényt a legáltalánosabban értve),
- a megoldás algoritmusa lehet rekurzív,
- a rekurzív algoritmust adott számítógépen, adott programozási eszközökkel (környezetben) kell megvalósítani.

A továbbiakban a specifikációról már nem ejtünk több szót; erről az érdeklődő Olvasó a [2]-ben, [6]-ban olvashat.

2. Rekurzív algoritmusok megfogalmazási példái

Tételezzük föl, hogy a rekurzió program-megvalósítása körül semmi gond nincsen. Ebben a hiszemben vizsgáljunk meg további példákat! Így talán megbarátkozunk a nagyon sok esetben egyetlen megoldási lehetőséggel: a rekurzióval, illetve a rekurzív programmal. Elsőként a problémák feltárásához használt faktoriális függvényt említjük, mint legelemibbet, legismertebbet. Pontosabban az előző fejezetben szereplő megoldást fogalmazzuk újra.

A lényegre való összpontosítás érdekében minduntalan visszatérő szerkezetben tárgyaljuk majd az elkövetkező függvénypéldákat. Nevezetesen: a formális definícióval kezdjük, majd a tényleges rekurziót megfogalmazó függvényeljárással folytatjuk. Lássuk tehát:

2.1. A faktoriális függvény

Definíciója:

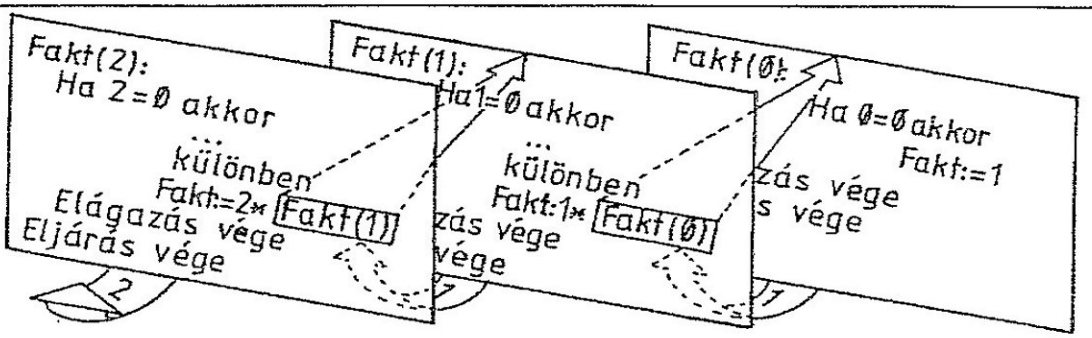
$$n! = \begin{cases} n*(n-1)! & ha\ n>0 \\ 1 & ha\ n=0 \end{cases}$$

Programja:

```
Fakt(N):
  Ha N=0 akkor          [nem rekurzív rész]
                 Fakt:=1
       különben         [rekurzív rész]
                 Fakt:=N*Fakt(N-1)
  Elágazás vége
Függvény vége.
```

2.2. A Fibonacci-számok

A rekurzív függvények matematikai elméletében éppúgy ismert, mint a biológiában a Fibonacci olasz matematikusról elnevezett számsorozat. E híres matematikus (aki egyébként a matematikának sok –mai szóhasználattal élve– ágában jeleskedett) Európában talán elsőként nyúlt egzakt eszközökhöz mindennapos –mondhatnánk „háztáji”– probléma megoldásához. Ugyanis azt vizsgálta, hogy egy nyúlpár „alapította” nyúlnemzetség adott idő alatt mekkora létszámúra




```
Fakt(N):
    Ha N=Ø akkor
            Fakt:=1
            különben
            Fakt:=N*Fakt(N-1)
    Elágazás vége
Eljárás vége
```

1. ábra

növekszik, figyelembe véve, hogy a leszármazottak is alaposan „besegítenek" a létszámnövelésbe.

Ha a szaporodás eléggé „szabályosan" történik, akkor az új generáció létszámát az előzőek ismeretében könnyen kiszámíthatjuk. Szerinte az új generáció növekedését az előző 2 generáció gyerekei teszik ki. E mögött az a feltételezés húzódik meg, hogy minden nyúlpár egyszerre éppen 2 utóddal járul a népességhez, és e „szokásukat" születésüket követő 2 egymásutáni időpontban „gyakorolják" (mert –mondjuk– mielőtt a harmadik szaporodásra sor kerülhetne, fazékba kerülnek).

Definíciója:

$$Fib(n) = \begin{cases} 0 & ha\ n = 0 \\ 1 & ha\ n = 1 \\ Fib(n-1) + Fib(n-2) & ha\ n > 1 \end{cases}$$

Példa: (a Fibonacci-szám sorozat első néhány tagja)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

Programja:

```
Fib(N):
  Elágazás
      N=0 esetén  Fib:=0
      N=1 esetén  Fib:=1
    egyéb esetben Fib:=Fib(N-1)+Fib(N-2)
  Elágazás vége
Függvény vége.
```

2.3. Az Ackermann függvény

Egy másik nevezetes függvény Ackermann nevéhez fűződik, aki a függvények kiszámíthatóságával, bonyolultságával kapcsolatosan vizsgálódott. A róla elnevezett kétváltozós függvény különös érdekessége, hogy minden „normális" függvénynél gyorsabban nő, és csak első néhány „tagjára" találtak eddig analitikus (nemrekurzív) leírást (zárt formulát). Pl. az A(0,m)=m+1, az A(1,m)=m+2, az A(2,m)=2*m+3, az $A(3,m)=2^{(m+3)}-3$ és végül az A(4,m) képlete nem is igazán tekinthető zártnak, mindenesetre egyfajta analitikus megfelelője: $2^{\alpha}-3$, ahol az α a 2-t m+2-szer tartalmazza kitevőként. Növekedésének ütemét néhány számpéldával illusztráljuk: A(0,0)=1, A(1,0)=2, A(2,0)=3, A(3,0)=5, A(4,0)=13, $A(4,1)=2^{16}-3$, ...

Definíciója:

$$A(n,m)=\begin{cases} m+1 & ha\ n=0 \\ A(n-1,1) & ha\ n>0\ és\ m=0 \\ A(n-1,A(n,m-1)) & ha\ n>0\ és\ m>0 \end{cases}$$

Programja:

```
Ack(N,M):
  Elágazás
        N=0     esetén Ack:=M+1
    N>0 és M=0  esetén Ack:=Ack(N-1,1)
      egyéb esetben    Ack:=Ack(N-1,Ack(N,M-1))
  Elágazás vége
Függvény vége.
```

2.4. Az „n alatt a k"

S végül a rekurzív függvények talán legegyszerűbbikét, a középiskolai tanulmányainkból is jól ismert „n alatt a k" (vagy binomiális együtthatók) rekurzív előállítását említjük. Meglepőnek tűnhet, de ez az additív kiszámítási mód is meglehetősen ősi. Egy Dzsamsid al-Kási nevű arab tudós foglalta írásba (nem

saját felfedezéseként beállítva) a XVI. században. Nagyon valószínűnek látszik, hogy már jóval korábban is számoltak e szabály felhasználásával. (A binomiális együtthatók háromszögszerű elrendezését hívjuk Pascal-háromszögnek.)

Definíciója:

$$B(n,k)=\begin{cases}1 & ha\ k=0\\ B(n-1,k)+B(n-1,k-1) & ha\ 0<k<n\\ 1 & ha\ k=n\end{cases}$$

Megjegyzés: Itt nem a szokásos (faktoriálissal történő) definíciót használjuk, hanem annak egy következményét. Számítástechnikai érdekessége az, hogy nincs benne szorzás, csak összeadás, így például nagyon könnyű a gépi kódú megvalósítását elkészíteni.

Programja:

```
B(N,K):
  Ha K=0 vagy K=N akkor  B:=1
                 különben B:=B(N-1,K)+B(N-1,K-1)
  Elágazás vége
Függvény vége.
```

A binomiális együtthatók egy másik rekurzív képlet segítségével is kiszámolhatók. Ebben a formulában a Pascal háromszög egyes soraiban szereplő értékeket a közvetlenül őket megelőző értékekből számítjuk ki.

Definíciója:

$$B(n,k)=\begin{cases}1 & ha\ k=0\\ B(n,k-1)*\dfrac{n-k+1}{k} & ha\ 0<k\le n\end{cases}$$

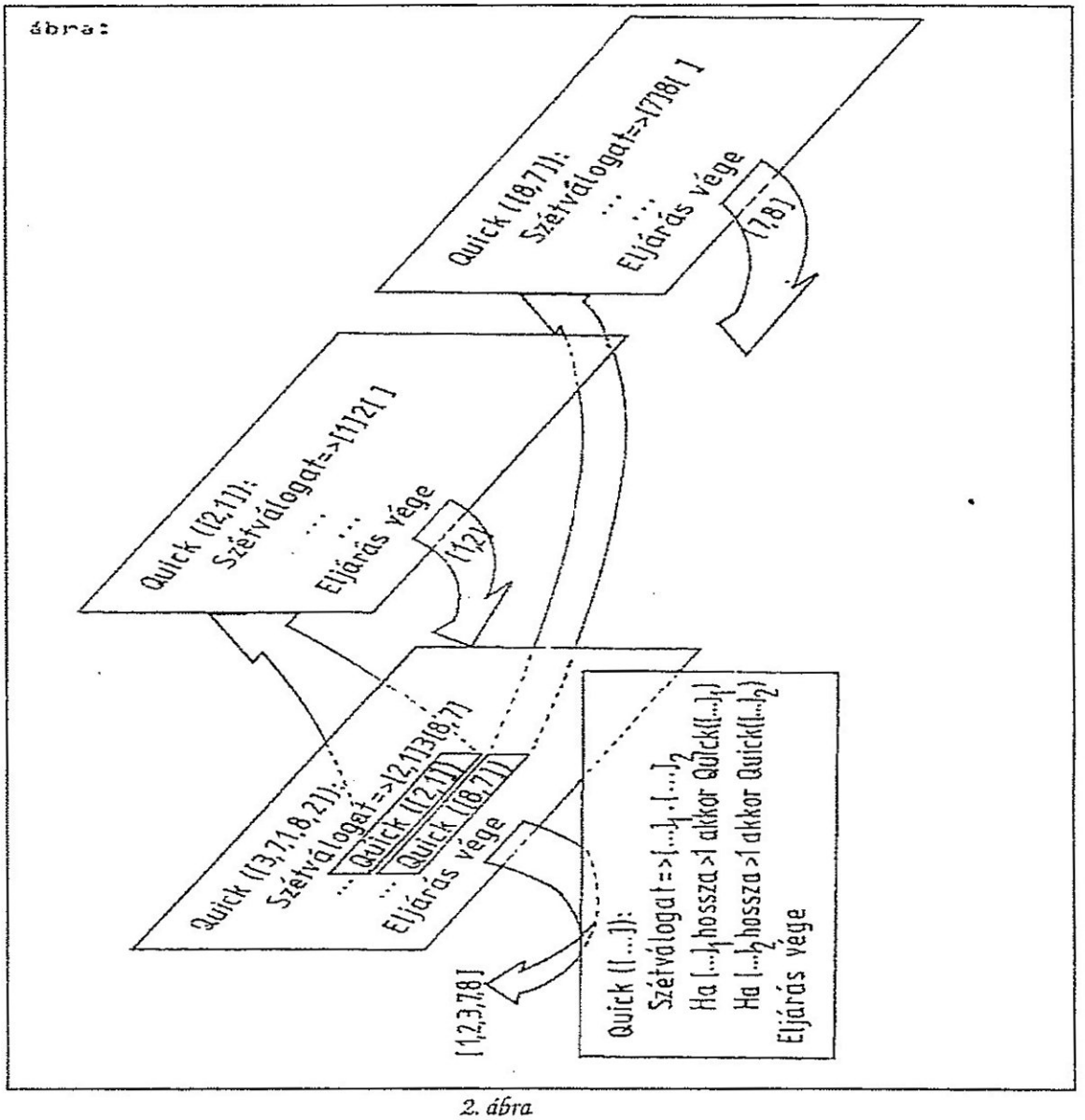
Programja:

```
B(N,K):
  Ha K=0 akkor  B:=1
       különben B:=B(N,K-1)*(N-K+1)/K
  Elágazás vége
Függvény vége.
```

2.5. A Quicksort rendezés

A következő feladatban a rekurzió nem függvény, hanem eljárás alakban bukkan föl. Ez persze nem jelent semmi nehézséget. Egyszerűen azt fejezi ki, hogy a szokásos algoritmusleíró nyelvünkben nem tudunk olyan „bonyolult" függvényt függvényként kezelni, mint a rendezés, amelynek vektor az értéke. A Hoare-féle rendezési módszer, amelyről most szó lesz, igen hatékonyan tölti be feladatát.

Lényege: Válogassuk szét úgy az A vektort, hogy az aktuális első elemnél kisebbek az elem elé kerüljenek, a nagyobbak pedig mögé. S mind az előtte, mind a mögötte levő részre végezzük el a fenti eljárást! Az 1, illetve 0 elemszámú intervallum már rendezett.



2. ábra

Programja:

```
Quick(A,E,V): [ A „rendezendő" rész Eleje, Vége]
  Szétválogat(A,E,V,K)
  Ha K-E>1 akkor Quick(A,E,K-1)
  Ha V-K>1 akkor Quick(A,K+1,V)
Eljárás vége.
```

A szétválogató eljárás az A vektor E. és V. elemei közötti részét válogatja két részre olyan átrendezéssel, hogy a kezdeti E. elemét a K. helyre teszi, a nála kisebb elemeket a K. elé, a nagyobbakat pedig a K. mögé.

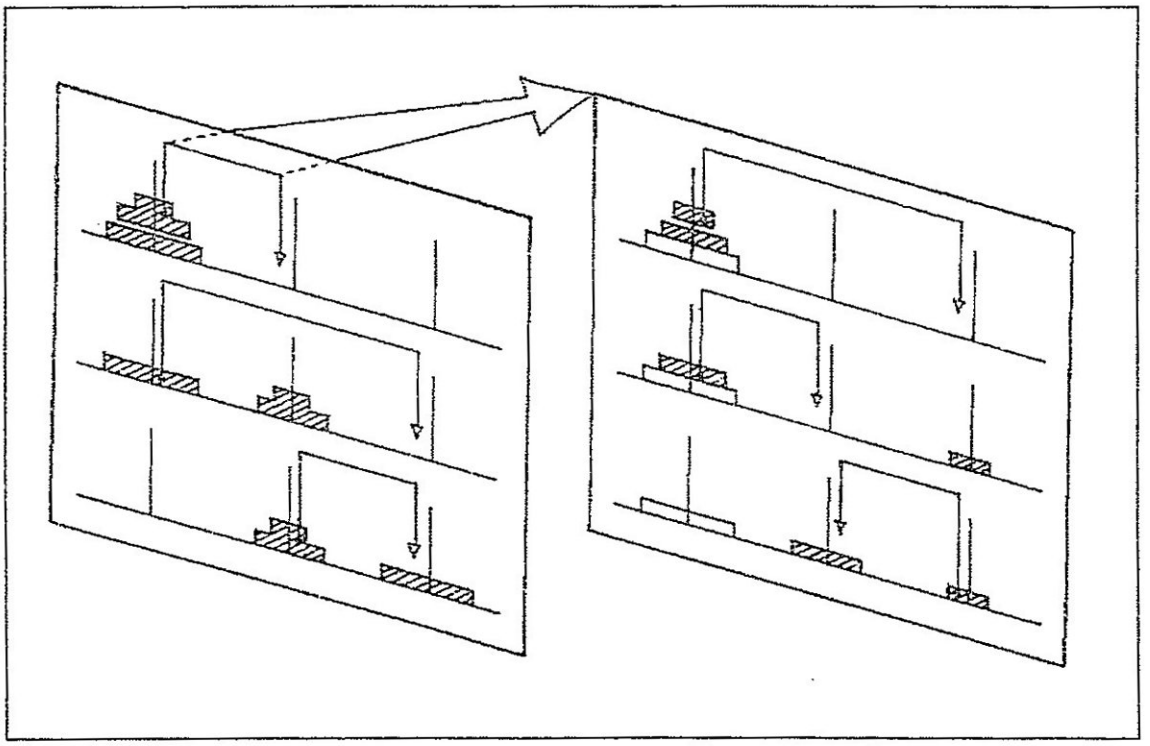
```
Szétválogat(A,E,V,K):
  K:=E; L:=V; X:=A(K)
  Ciklus amíg K<L
    Ciklus amíg K<L és A(L)≥X
      L:=L-1
    Ciklus vége
    Ha K<L akkor A(K):=A(L); K:=K+1
                 Ciklus amíg K<L és A(K)≤X
                   K:=K+1
                 Ciklus vége
                 Ha K<L akkor A(L):=A(K); L:=L-1
    Elágazás vége
  Ciklus vége
  A(K):=X
Eljárás vége.
```

2.6. Hanoi tornyai

A klasszikus példák sorában egy a klasszikusok között is klasszikusnak számító következik. A „Hanoi tornyai" nevet viselő játék -úgy gondoljuk- mindenki előtt ismert. Nézzük gyermekkorunk kedvenc intelligencia-fokmérője miként vonult be a számítástechnika „történelmébe"! Hogyan vált egy keleti kultúrjáték képzetét keltő játék a számítógépes kultúra részévé, egy gondolkodási séma iskolapéldájává? A játék ténylegesen európai: E. Lucas francia matematikus találmánya. Legfeljebb kultikus történeti alapja (ihlete) kötődik kelethez. Hagyomány szerint ehhez hasonló szertartás zajlott Benares egyik Brahma-templomában.

Lényege: Adott 3 rudacska. Egyikén (mondjuk az elsőn) egyre csökkenő sugarú korongok vannak. Az a feladat, hogy tegyük át a harmadik rudacskára a korongokat egyenként úgy, hogy az átpakolás közben és természetesen a végén is minden egyes korongon csak nála kisebb lehet. Az átpakoláshoz lehet segítségül felhasználni a középső rudacskát.

Bár az algoritmust igen röviden elintéztük (tehettük a rekurzió jóvoltából), nem árt egy mondatban megmagyarázni az algoritmus mögötti gondolatot. Íme a magyarázat: Úgy tegyük át az N db korongot az A-ról a C pálcikára (B közvetítésével), hogy először a felső N–1 darabot a B-re tesszük (C segítségével), majd az így „felszabadult" legalsót a végleges helyére (C-re). Ezután nincs is már másra szükség, csak arra, hogy a B-n levő N–1 db-ot a C-re emelgessük. (Persze ezt ugyanazzal a szisztémával, mint amivel korábban az „1. lépést" megtettük.)



3. ábra

Programja:

```
Hanoi(N,HONNAN,HOVA,MIVEL):
  Ha N>0 akkor  Hanoi(N-1,HONNAN,MIVEL,HOVA)
                Ki: N,HONNAN,HOVA
                Hanoi(N-1,MIVEL,HOVA,HONNAN)
  Elágazás vége
Eljárás vége.
```

3. A rekurzió megvalósítása *nemrekurzív* nyelveken

A korábban látott problémákra keressünk olyan megoldásokat, amelyek a rekurziót nem támogató, de eljáráshívással rendelkező nyelveken is használhatóak! Ez nemcsak önmagában érdekes ismereteket szolgáltat, hanem némi bepillantást is enged sok más programnyelv fordításának mechanizmusába is. Itt érkeztünk el a rekurzió *implementálás*ának kérdéséhez. Ennek kapcsán azt is megismerjük, hogy pl. az ALGOL-szerű nyelvek milyen módon valósítják meg a függvényeljárások hívását, illetve hogyan teszik „zárlatmentessé" (paraméterátadással, lokális változók segítségével) az eljáráshívásokat. Idézzük föl a korábban felvetődött kérdéseket! Megint a faktoriálist kiszámító rekurzív függvény példáját hívjuk segítségül.

```
Fakt(N):
  Ha N=0 akkor  f:=1
       különben f:=N*Fakt(N-1)
  Fakt:=f
Függvény vége.
```

A függvényeljárás szempontjából „tanulságos" 2! (Fakt(2)) kiszámítását követtük nyomon az első fejezetben.

1. probléma: hogyan kerül a bemenő érték ugyanazzal a kóddal megvalósított eljáráshoz, azaz a rekurzívan újból hívott hoz? (A *bemenő paraméter* problematikája.) A program algoritmusa szerint az N változó kapja értékül. Igen, de a hívó programban ugyanez már kapott értéket (N=2). Nem lesz baj, ha azt elrontjuk?

2. probléma: hogyan kerül a rekurzívan hívott függvény értéke a hívó eljárásban „felszínre", annak tudomására. Egyáltalán mi tekinthető a függvény értékének? S ez nemcsak rekurzív esetben probléma. (A *függvényeljárások értékvisszaadása*.)

3. probléma: az f-re vonatkozó értékadás valójában így nézett ki: f:=N*... . Melyik N-ről van szó a sok közül? (A *lokális változó*k problémája.)

Az 1. és 3. megoldása –mint érezhető– szorosan összefügg. Mindkettőben arra a kérdésre kell választ találnunk, hogy az eljárás egy változója –jelen esetben az N bemenő paraméter– hogyan tehető egyedivé, csak az eljáráshoz (sőt éppen annyiadik hívásához) tartozóvá. Tegyünk a következőképpen:

- Mielőtt egy újabb eljárás (függvényeljárás)hívást hajtanánk végre, jegyezzük meg a hívó rutin *bemenő paramétereit*. Gondolva arra, hogy rekurzív hívás esetén a hívó rutinhoz több ízben is meg kell jegyeznünk paramétereit,

ezért e mentésre egy *vermet* használunk föl.

- Visszatérés után, de még azelőtt, hogy az algoritmus szerint bármi mást csinálnánk, a rutinhoz tartozó paraméterek értékeit a veremből kivéve, *visszaállítjuk* azok hívást meg előző állapotát.

A 2. probléma megoldására is ugyanezt a vermet használjuk föl! Pontosabban, ha egy kifejezésben egy függvényeljárást használunk, akkor annak *értékét* maga a függvényt kiszámító rutin a *visszatérése előtt verembe* helyezi. így a kifejezést valójában elemeire bontva, részeredményenként –esetleg vermet használva– számítjuk ki. Ezt a „filozófiát" kell tükrözze pl. a BASIC-beli átírás algoritmusa is!

Például: ha a kifejezés az N*Fakt(N-1), akkor a Fakt(N-1) függvény kifejezésen belüli kiszámítása jelenti a problémát. A Fakt() függvény eredménye, mint részeredmény a verembe kerül: *verembe*(N-1 faktoriálisa) –ezt maga a Fakt() függvény teszi oda–, majd az eredeti f:=N*Fakt(N-1) kifejezés értékét –bonyolult függvényhívás helyett– a *veremből*(f); f:=N*f utasításpár adja.

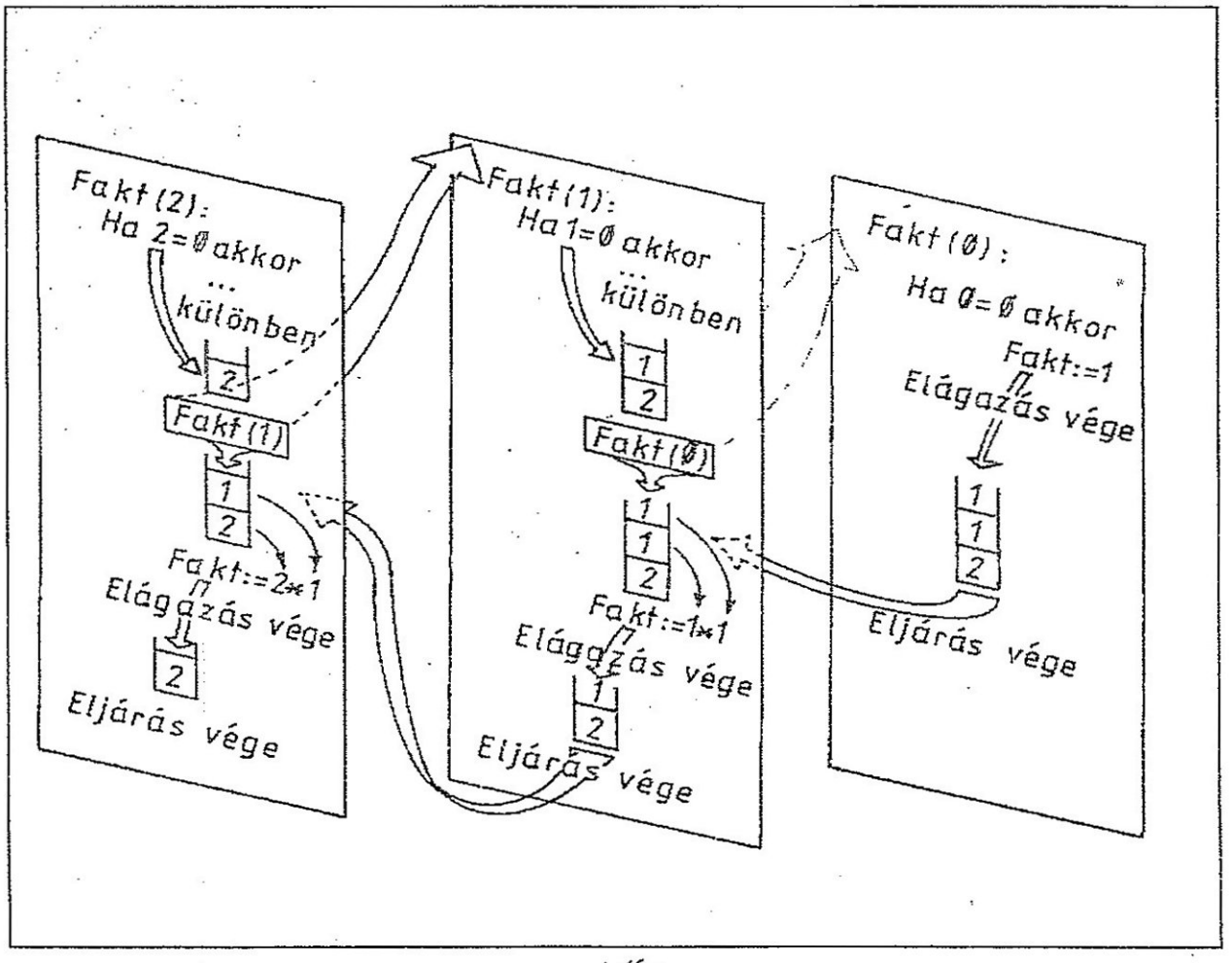
Még egy probléma merülhet föl a függvényértékkel kapcsolatosan. Hogyan ismerhető föl „mechanikusan" a függvényértéket képviselő változó, vagyis mely változó értékét kell, mint függvényértéket a verembe tenni? Ez kulcskérdése az érték verembe helyezhetőségének. A szokásos megoldás szerint –így jártunk el eddig is–az eljárásnak van egy a *függvény nevével megegyező nevű változója* –mint az értéket reprezentáló „objektuma"–, s ez kell, hogy a verembe kerüljön a visszatérés előtt! Az előbbi példára gondolva: a faktoriális „végeredményét" egy Fakt nevű változó tartalmazza.

Nézzük meg az ismertetett rekurzív feladatok ilyen értelmű átírásait! A továbbiakban tehát nem „valódi függvényes", hanem annak „utánzását végző eljárásos" megvalósításait nézzük meg a korábbi példákkal kapcsolatban. Két, veremre vonatkozó „elemi" (tehát általunk most nem részletezett) tevékenységet használunk föl az algoritmusokban:

- *Verembe* utasítás: a verembe teszi az értéke(ke)t,
- *Veremből* utasítás: értéke a verem tetején levő adat (adatok), s ezeket kiveszi a veremből.

3.1. A faktoriális függvény

```
Fakt(N):
  Ha N=0 akkor
            Fakt:=1
       különben
            Verembe(N)     [ paraméter a verembe]
            Fakt(N-1)      [ függvényhívás]
            Veremből(f)    [ értéke⇒f]
            Veremből(N)    [ paraméter a veremből]
            Fakt:=N*f      [ a függvényérték]
  Elágazás vége
  Verembe(Fakt)            [ függvényérték a verembe]
Eljárás vége.
```



4. ábra

Ezzel a megoldással még nem biztos, hogy végeztünk. Egyes nyelvekben ugyanis *nincs paraméterátadás*. Ezért ezt is magunknak kell megszerveznünk. Ezt úgy tesszük, hogy a *formális paraméter*ként szereplő változónak értékül adjuk az *aktuális paraméter*ként szereplő kifejezést. Az algoritmusban ezután meg kell különböztetnünk a Fakt eljárásnevet és a Fakt változót, az előbbit kereszteljük át

*Faktoriális*ra!

```
Faktoriális:
  Ha N=0 akkor
            Fakt:=1
        különben
            Verembe(N)      [ paraméter a verembe]
            N:=N-1          [ paraméterátadás]
            Faktoriális     [ függvényhívás]
            Veremből(f)     [ értéke⇒f]
            Veremből(N)     [ paraméter a veremből]
            Fakt:=N*f       [ a függvényérték]
  Elágazás vége
  Verembe(Fakt)             [ függvényérték a verembe]
Eljárás vége.
```

Megjegyzés: A továbbiakban a paraméterátadással nem foglalkozunk, hiszen az nemcsak a rekurzió problémája, hanem mindenfajta eljáráshasználaté is.

3.2. A Fibonacci-számok

Ebben az esetben az okoz további nehézséget, hogy a Fib:=Fib(N-1)+Fib(N-2) kifejezésben a függvényhivatkozás kétszer is szerepel. A második hívásnál tehát már nemcsak a bemenő paramétert, hanem egy lokális változót is, amely az első hívás részeredményét tárolja, verembe kell tenni, majd a hívás után kivenni. A verembe tett paraméterre is szükség van a második hívás helyes paraméterezéséhez, ezért azt is vissza kell hozni a két hívás között. Új (összetett) veremkezelő utasításokat vezetünk be a rövidebb leírhatóság érdekében:

Verembe(A,B) jelentése: *Verembe*(A); *Verembe*(B)

Veremből(B,A) jelentése: *Veremből*(B); *Veremből*(A)

azaz a később betett B-t kell előbb kivenni.

```
Fib(N):
  Elágazás
    N=0 esetén Fib:=0
    N=1 esetén Fib:=1
    egyéb esetben
      Verembe(N)          [ paraméter a verembe]
      Fib(N-1); Veremből(f1) [ 1. részeredmény⇒f1]
      Veremből(N)         [ paraméter a veremből]
      Verembe(N,f1)       [ lokális változók a verembe]
      Fib(N-2); Veremből(f2) [ 2. részeredmény⇒f2]
      Veremből(f1,N)      [ lokális változók a veremből]
      Fib:=f1+f2          [ a függvényérték kiszámítása]
  Elágazás vége
  Verembe(Fib)            [ függvényérték a verembe]
Eljárás vége.
```

3.3. Az Ackermann függvény

Itt a függvényérték a rekurzív hívásban (Ack:=Ack(N-1,Ack(N,M-1)) paraméterként szerepel, ami ha első ránézésre riasztónak is tűnik, azonban az eddigiek szem előtt tartásával –aprólékosan ugyan, de– megvalósítható.

```
Ack(N,M):
  Elágazás
        N=0      esetén Ack:=M+1
    M=0 és N>0   esetén Verembe(N,M)
                        Ack(N-1,1)
                        Veremből(Ack); Veremből(M,N)
        egyéb    esetben Verembe(N,M)
                        Ack(N,M-1); Veremből(A)
                        Veremből(M,N); Verembe(N,M)
                        Ack(N-1,A)
                        Veremből(Ack); Veremből(M,N)
  Elágazás vége
  Verembe(Ack)
Eljárás vége.
```

Az elágazás második ágát egyszerűbben is felírhattuk volna:

```
Verembe(N)
Ack(N-1,1)
Veremből(Ack); Veremből(N)
```

hiszen ezt az ágat csak M=0 esetén hajtjuk végre és M számszerűen nem érdekes a továbbiakban.

3.4. Az „n alatt a k”

Ez az eljárás a Fibonacci-számokhoz hasonló, újdonságot lényegében nem tartalmaz.

```
B(N,K):
  Ha K=0 vagy K=N akkor    B:=1
                 különben  Verembe(N,K)
                           B(N-1,K); Veremből(b1)
                           Veremből(K,N); Verembe(N,K,b1)
                           B(N-1,K-1)
                           Veremből(b2); Veremből(b1,K,N)
                           B:=b1+b2
  Elágazás vége
  Verembe(B)          [ függvényérték a verembe]
Eljárás vége.
```

A további példáink eredetileg sem rekurzív függvény-, hanem eljáráshívásokat tartalmaztak, így ez némileg egyszerűsíti az átírás metódusát. Felhasználjuk ismereteinket arra is, hogy a lokális változók mentésére vonatkozó „szabályt” nem mechanikusan alkalmazzuk, hanem csak ott építjük be a megfelelő kiegészítést,

ahol arra valóban szükség van.

3.5. A Quicksort rendezés

Itt a rendezendő A vektort úgy kezeljük, mint egy „globális" adatszerkezetet. Ezt a vektort mindenki egyformán használhatja (pontosabban a vektor paraméterek által meghatározott részét). Így a rekurzív hívások esetén ezt nem kell újra létrehozni, ezért hívás előtt nem kell menteni (s utána nem kell visszaállítani).

```
Quick(A,E,V): [A „rendezendő" rész Eleje, Vége]
  Szétválogat(A,E,V,K)
  Ha K-E>1 akkor Verembe(E,V,K)
                 Quick(A,E,K-1)
                 Veremből(K,V,E)
  Ha V-K>1 akkor Verembe(E,V,K)
                 Quick(A,K+1,V)
                 Veremből(K,V,E)
Eljárás vége.
```

A *szétválogatás* eljárást most nem ismételjük meg. Ahhoz az átírás fenti „szabályai" nem tesznek semmit hozzá.

3.6. Hanoi tornyai

A rekurzió megvalósítása ez esetben is teljesen mechanikusan történhet. Minden rekurzív hívást megelőzi a teljes lokális változógarnitúra elmentése, illetve a hívás után ugyanazok visszaállítása.

```
Hanoi(N,HONNAN,HOVA,MIVEL):
  Ha N>0 akkor Verembe(N,HONNAN,MIVEL,HOVA)
               Hanoi(N-1,HONNAN,MIVEL,HOVA)
               Veremből(HOVA,MIVEL,HONNAN,N)
               Ki: N,HONNAN,HOVA
               Verembe(N,HONNAN,MIVEL,HOVA)
               Hanoi(N-1,MIVEL,HOVA,HONNAN)
               Veremből(HOVA,MIVEL,HONNAN,N)
  Elágazás vége
Eljárás vége.
```

```
Hanoi(3,1,3,2)
      Hanoi(2,1,2,3)
            Hanoi(1,1,3,2)
                  Hanoi(0,1,2,3)
                  Eljárás vége
                  Ki: 1 1 3
                  Hanoi(0,2,3,1)
                  Eljárás vége
            Eljárás vége
            Ki: 2 1 2
            Hanoi(1,3,2,1)
                  Hanoi(0,3,1,2)
                  Eljárás vége
                  Ki: 1 3 2
                  Hanoi(0,1,2,3)
                  Eljárás vége
            Eljárás vége
      Eljárás vége
      Ki: 3 1 3
      Hanoi(2,2,3,1)
            Hanoi(1,2,1,3)
                  Hanoi(0,2,3,1)
                  Eljárás vége
                  Ki: 1 2 1
                  Hanoi(0,3,1,2)
                  Eljárás vége
            Eljárás vége
            Ki: 2 2 3
            Hanoi(1,1,3,2)
                  Hanoi(0,1,2,3)
                  Eljárás vége
                  Ki: 1 1 3
                  Hanoi(0,2,3,1)
                  Eljárás vége
            Eljárás vége
      Eljárás vége
Eljárás vége
```

5. ábra

Vegyük észre, hogy a veremben mindig az aktuális szintet megelőző szintű eljárások paraméterei találhatók! Egy újabb szint újabb bekezdésként jelenik meg az ábrán.

Megjegyzés: az 1. függelékben kitérünk a rekurzió egy másik implementációs lehetőségének ismertetésére, amely közelebb áll a ténylegesen alkalmazott technikához, bár szemlélete nem ilyen egyöntetű.

4. A rekurzió *rekurzív* nyelveken

Ezután vizsgáljunk meg néhány rekurzív eljárások, függvények írását lehetővé tevő nyelvet a rekurzió megvalósíthatósága szempontjából! Vizsgálatainkhoz amatőr programozási nyelveket választottunk ki.

Nyelvi megvalósítások esetén egy technikai probléma merül fel. A rekurzív hívás ugyanis többféleképpen történhet. Ennek illusztrálására ismételjük meg a korábbról ismert *Quicksort rendezés* algoritmusát!

```
Quick(A,E,V): [A „rendezendő" rész Eleje, Vége]
  Szétválogat(A,E,V,K)
  Ha K-E>1 akkor Quick(A,E,K-1)
  Ha V-K>1 akkor Quick(A,K+1,V)
Eljárás vége.
```

Itt a rekurzív eljárásban szerepel saját magának a hívása. Ezt nevezzük *közvetlen rekurzió*nak. A fenti eljárást azonban másképpen is elkészíthetjük:

```
Quick(A,E,V): [A „rendezendő" rész Eleje, Vége]
  Szétválogat(A,E,V,K)
  Részrendezés(A,E,K1)
  Részrendezés(A,K+1,V)
Eljárás vége.

Részrendezés(E,V):
  Ha V-E>0 akkor Quick(A,E,V)
Eljárás vége.
```

Most tehát a *Quick* eljárás önmagát nem hívja, hanem egy másik eljárást, s az hívja újra őt. Ezt nevezzük *közvetett rekurzió*nak. Az egyes programozási nyelvek a rekurzió e kétféle használatát különbözőképpen ítélik meg, s engedélyezik vagy tiltják. Az egyes nyelvekről itt nem adunk részletes leírást, magyarázatot, mindegyik példánk az adott nyelvet ismerőknekszól.

4.1. Pascal

A Pascal rendelkezik a rekurzív eljárások „létéhez" elengedhetetlen *lokális változó* fogalommal, s nem tiltja meg azt sem, hogy egy eljárás önmagát hívja. Nézzük meg, hogy a fenti eljárás hogyan kódolható Pascal nyelven! Az eljárásokon annyit módosítunk, hogy a „globális" változó szerepét játszó A vektort nem paraméterként adjuk át, hanem ténylegesen globális változóként használjuk. A megoldó Pascal programrészlet:

```
var A:array [1..100] of integer;
procedure Szétválogat(E,V: integer; var K: integer);
  ...
end;
procedure Quick(E,V: integer);
  var K:integer;
begin
  Szétválogat(E,V,K);
  if K-E>1 then Quick(E,K-1);
  if V-K>1 then Quick(K+1,V)
end;
```

A *Quick* eljárásban értékszerinti paraméterátadást használtunk, a hívott eljárás a paraméterként adott kifejezés értékét kapja meg.

Ha közvetett rekurzióval megvalósított megoldásra gondolunk, azt tapasztaljuk, hogy annak Pascal-beli megvalósítása nem megy ugyanolyan akadálymentesen, mint a közvetlen esetben ment. Ennek oka, hogy a nyelv szabályai rögzítik, hogy eljárást hívni (sőt általában bármit használni) csak akkor lehet, ha a hívott eljárást már definiáltuk (pontosabban, ha a fordító már a paraméterátadáshoz, illetve - átvételhez szükséges összes információ birtokában van: ismeri a paraméterek számát és típusukat). Ez nagy nehézséget jelent, hiszen ha az A eljárás hívja a B eljárást, akkor előbb a B eljárást kell megadnunk. Ha viszont a B is hívja az A-t, akkor pedig az A-t kell előbb megadni. Mindkettőt egyszerre nyilván nem lehet megoldani. A Pascal nyelv készítői azonban kitaláltak erre is egy megoldást, amellyel az eljárás külső szemlélő számára való definiálását és az eljárás kifejtését (azaz utasításai megadását) egymástól el lehet választani. Ezt teszi lehetővé a **forward** alapszó. (A *Szétválogat* eljárást most nem írjuk meg, a helye a *Quick* eljárás előtt van.)

```
var A:array [1..100] of integer;
procedure Szétválogat(E,V: integer; var K: integer);
  ...
end;
procedure Quick(E,V: integer);   forward;
procedure Részrendezés(E,V: integer);
begin
  if V-E>0 then Quick(E,V);
end;
```

```
procedure Quick;
  var K: integer;
begin
  Szétválogat(E,V,K);
  Részrendezés(E,K-1);
  Részrendezés(K+1,V)
end;
```

4.2. Logo

A Logo nyelv kínálja a legegyszerűbb lehetőséget a rekurzió megvalósítására. Nézzük meg például a *Fibonacci-számok*at meghatározó eljárást a Logo nyelv LCN Logo verziójában!

```
MAKE  fib
      FUNC  WITH n
            IF  equal?  n
                        0
                THEN  0
                ELSE IF equal?  n
                                1
                        THEN 1
                        ELSE sum  fib minus n
                                  1
                                  fib minus n
                                  2
```

Ha egy függvény egy másik függvény paramétereként szerepel, azt is könnyen megvalósíthatjuk a Logo nyelvben, ugyanis a függvény kiszámítása mindig a paraméterei kiszámításával kezdődik, s ha az függvény, akkor előbb a paraméterként szereplő függvény értékét kell kiszámítani. Nézzük meg az *Ackermann függvény* értékét kiszámító Logo függvényt!

```
MAKE ack
     FUNC WITH  n
                m
          IF equal? n
                    0
               THEN sum  n
                         1
               ELSE IF equal?  m
                               0
                       THEN ack  minus n
                                       1
                                 1
                       ELSE ack  minus n
                                       1
                                 ack n
                                     minus m
                                           1
```

Érdekes lehet Logo-ban a rekurzív eljárások megvalósítása, a Logo ugyanis mindent függvényszerűen definiál. Emiatt az eljárásokat is függvényekké kell alakítani.

Nézzük meg például a Hanoi tornyait függvényszerűen! Ehhez először definiálni kell az eredmény szerkezetét. Legyen az eredmény egy sorozat, amely a korongrakodásokat írja le. A sorozat egyes elemei számpárok, amelyek azt tartalmazzák, hogy melyik pálcikáról melyikre kell átteni a következő korongot.

A Logo nyelv felhasznált műveletei:

```
makelist elem1
         elem2        [a két elemből egy kételemű sorozatot készít]
sentence sorozat1
         sorozat2     [a két sorozatból egy sorozatot készít, konkatenál]
fput elem
     sorozat          [az elemet berakja a sorozat elejére]
```

```
MAKE hanoi
     FUNC  WITH  n
                 honnan
                 hova
                 mivel
           IF  greater?    n
                           0
               THEN  sentence  hanoi minus n
                                          1
                                   honnan
                                   mivel
                                   hova
                               fput makelist  honnan
                                              hova
                                    hanoi  minus  n
                                                  1
                                           mivel
                                           hova
                                           honnan
               ELSE "[]
```

5. Rekurzió és iteráció

(Rekurzív programok átírása nemrekurzívvá)

Az előzőekben megbarátkoztunk a rekurzióval, mint egy problémamegoldási stratégiával, sőt megvizsgáltunk néhány programozási nyelvet a rekurzió szempontjából (mennyire „termőtalaja" az adott nyelv a rekurzív programoknak). Úgy gondoljuk sokan elvégezték azt a kísérletet, amely a legegyszerűbb rekurzív függvénynek, a faktoriálisnak kétféle (iteratív és rekurzív) megvalósítását hasonlítja össze. Az ő tapasztalataik is –minden bizonnyal– lesújtóak voltak: már ami a végrehajtási idő lényeges meghosszabbodását illeti. (A mi tapasztalatainkat az 1. táblázat tartalmazza. Mellékeltük a kétféle nézőpont néhány jellemző futási idejét. Már ránézésre is érezhető a különbség.

N	Iteratív	Rekurzív
10	.04	0.2 másodperc
20	.08	0.42 másodperc
50	.20	1.05 másodperc
70	.28	1.5 másodperc

1. táblázat

Lesújtó véleményünk csak erősödne, ha akármelyik bonyolultabb feladatot végrehajtó iteratív, illetve rekurzív megoldást vetnénk össze. Ezzel azt is elárultuk, hogy némelyik korábban említett feladatra van egyszerű nemrekurzív megoldás. Mielőtt azonban a rekurzió feloldásának lehetőségeiről, az átírás módjáról esnék szó, érdemes elgondolkodni a futási idő meghosszabbodásának okairól.

Az N!-t kiszámító rekurzív program (N+1)-szer „aktivizálja" önmagát, ami 2*(N+1) veremműveletet jelent. Még ha a program „érdemi magja" és a rekurzió adminisztrálására szolgáló segédtevékenységek aránya nem is volna ilyen kedvezőtlen, akkor is jelentős többletidőt vinne a programfutásba. (Ugyanez a helyzet –természetesen– más programnyelvek alkalmazása esetén is fennáll, legfeljebb kisebb időszinten.) Még elszomorítóbb a kép a Fibonacci-számok rekurzív generálása esetén. Emlékezzünk: az N. Fibonacci-szám kiszámításához a megelőző két Fibonacci-számhoz kell nyúlnunk! Nem váratlan, hogy a rekurzív hívások száma meredekebben növekszik, mint tette a faktoriális esetén. Hiszen, ha r(N) jelöli az N. Fibonacci-szám kiszámításához szükséges hívások számát, akkor az algoritmus alapján magától értetődően adódik r(N)-re a képlet: r(N)=r(N-1)+r(N-2)+1. (A

+1 a kezdeti hívás miatt szükséges.) Ugyanezt kifejezhetjük a Fibonacci-számok ismeretében is: r(N)=F(N+1)+F(N)+F(N-1)-1. (Itt F(i) az i. Fibonacci-számot jelöli.) Az időigény növekedését teszi kézzelfoghatóvá a 2. táblázat, valamint a 3. táblázat.

N	F(N)	r(N)
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25
7	13	41
8	21	67
9	34	109
10	55	177
11	89	287
12	144	465

2. táblázat

Fibonacci-számok:

$F(0)=0$, $F(1)=1$

$F(i)=F(i-1)+F(i-2)$

Rekurzív hívások száma:

$r(0)=1$, $r(1)=1$

$r(i)=r(i-1)+r(i-2)+1=$

$=F(i+1)+F(i)+F(i-1)-1$

N	F(N)	rekurzív	iteratív
0	0	0.003	0.007
1	1	0.004	0.007
2	1	0.01	0.011
5	5	0.046	0.024
8	21	0.2	0.035
10	55	0.54	0.046
13	233	2.3	0.058
15	610	6.0	0.066
20	6765	67.0	0.088
21	10946	108.0	0.092

3. táblázat

Az idő- (s mélyebben belegondolva az adminisztrációval járó hely-) gondokon túl jelentkező implementációs nehézségek komolyan vetik föl a kérdést, nem lehet-e a rekurziót feloldani, iterációval helyettesíteni. Reményt ébreszthet bennünk az, hogy pl. az F(N) kiszámításakor esetleg spórolni lehet a már egyszer kiszámított részeredmények újra felhasználásával: az F(N–1)-hez generált F(N–2)-t ismételt számolás nélkül használjuk föl az F(N)-hez is.

5.1. Rekurzív formulával definiált függvények

Ha egy rekurzív függvény a következő formula felhasználásával számolható ki:

$$f(i)=\begin{cases} g(f(i-1),f(i-2),\ldots,f(i-K)) & ha\ i \geq K \\ h(i) & ha\ 0 \leq i < K \end{cases}$$

azaz minden értéke valamely korábban kiszámolható értékből számolható, akkor némi memóriafelhasználással elkészíthető a rekurziómentes változat, amelyben az egyes függvényértékeknek megfeleltetünk egy F(N) vektort. A rekurzív függvényt kiszámító eljárás:

```
f(N):
  Ha N<K akkor  f:=h(N)
        különben f:=g(f(N-1),...,f(N-K))
Függvény vége.
```

Az ennek megfelelő vektoros változat:

```
f(N):
  Ciklus I=0-tól K-1-ig
    F(I):=h(I)
  Ciklus vége
  Ciklus I=K-tól N-ig
    F(I):=g(F(I-1),...,F(I-K))
  Ciklus vége
  f:=F(N)
Függvény vége.
```

Megjegyzés: Megállapítható az is, hogy ilyen esetekben nincs szükség N elemű vektorra, hanem csak az utolsó K db értéket kell megjegyezni. (Vegyük észre, hogy ez egy K méretű sor, amelynek hatékony kezelésére ismert megoldások léteznek!) Vagyis így nemcsak időben lesz optimálisabb a megoldás, hanem a helyigénye is lényegesen csökken. A továbbiakban sem ügyelünk erre az optimalizálásra, az átírás mechanizmusára fektetjük a hangsúlyt!

Alkalmazzuk a módszert néhány korábban vizsgált függvényre!

1. N! kiszámítása

Ebben az esetben K=1, h(0)=1, g(F(I–1))=I*F(I–1). Tehát a kapott algoritmus:

```
Fakt(N):
  F(0):=1
  Ciklus I=1-től N-ig
    F(I):=I*F(I-1)
  Ciklus vége
  Fakt:=F(N)
Függvény vége.
```

2. Fibonacci-számok

Most K=2, h(0)=0, h(1)=1, g(F(I–1),F(I–2))=F(I–1)+F(I–2). Tehát a nemrekurzív algoritmus:

```
Fib(N):
  F(0):=0; F(1):=1
  Ciklus I=2-től N-ig
    F(I):=F(I-1)+F(I-2)
  Ciklus vége
  Fib:=F(N)
Eljárás vége.
```

3. N alatt a K

A feladat azért érdekes, mert most nem vektort, hanem mátrixot kell használni a megoldáshoz. B(N,K) kiszámításához B(N–1,K), illetve B(N–1,K–1) értékére van szükség. Ez azt jelenti, hogy minden olyan B(I,J)-re szükségünk lesz, amelyre I<N és J≥0 és J≥I–N+K és J≤K és J≤I. Ebből kezdőértékként kell megadnunk B(I,0) értékeit 1 és N–K között, valamint B(I,I) értékeit 1 és K között.

Példa (B(4,3) kiszámítása):

```
          1
        1 1
      1 2 1
    1 3 3 1
  1 4 6 4 1
```

```
Binom(N,K):
  Ciklus I=1-től N-K-ig   [a Pascal Δ egyik „széle"]
    B(I,0):=1
  Ciklus vége
  Ciklus I=1-től K-ig     [a Pascal Δ másik „széle"]
    B(I,I):=1
  Ciklus vége
```

```
  Ciklus I=2-től N-ig
    Ciklus J=max(1,I-N+K)-tól min(I-1,K)-ig
      B(I,J):=B(I-1,J-1)+B(I-1,J)
    Ciklus vége
  Ciklus vége
  Binom:=B(N,K)
Eljárás vége.
```

5.2. Jobbrekurzió

Gyakran speciális szerkezetű a rekurzív program, s ez megkönnyíti az átírását nemrekurzívvá. A rekurzió nemrekurzív átírását a rekurzív eljárás lokális változói teszik nehézzé. Ha maga az algoritmus olyan szerkezetű, hogy *a rekurzív hívás után nincs szükség már az eljárás lokális változóira*, akkor a rekurzív hívás minden egyéb változtatás nélkül ciklussá alakítható. Ez a *jobbrekurzió (farok-)* esete, azaz amikor a rekurzív hívás az eljárás végén található. A paraméterek egyike (X) csak a rekurzió szervezésében játszik szerepet, míg a másik (Y) az eredményt tartalmazza. Az ilyen programok általános szerkezete:

```
RekElj(X,Y):
  S(X)
  Ha p(X,Y) akkor RekElj(f(X),Y)
Eljárás vége.
```

vagy

```
RekElj(X,Y):
  Ha p(X,Y) akkor S(X,Y)
                  RekElj(f(X),Y)
Eljárás vége.
```

Mindkettőnek egyszerű a nemrekurzív változata is:

```
RekElj(X,Y):
  S(X,Y)
  Ciklus amíg p(X,Y)
    X:=f(X)
    S(X,Y)
  Ciklus vége
Eljárás vége.
```

vagy

```
RekElj(X,Y):
  Ciklus amíg p(X,Y)
    S(X,Y)
    X:=f(X)
  Ciklus vége
Eljárás vége.
```

A két alapeset összevont változata:

```
RekElj(X,Y):
  Q(X,Y)
  Ha p(X,Y) akkor S(X,Y); RekElj(f(X),Y)
Eljárás vége.
```

Az iteratívvá írása:

```
RekElj(X,Y):
  Q(X,Y)
  Ciklus amíg p(X,Y)
    S(X,Y)
    X:=f(X)
    Q(X,Y)
  Ciklus vége
Eljárás vége.
```

Nézzünk néhány példát az átírásra!

1. Egy szöveg betűinek kiírása

```
Betûk(X):
  Ha X nem üres akkor Ki: Elsõ(X)
                      Betûk(Elsõutániak(X))
Eljárás vége.
```

A ciklust tartalmazó (iteratív) megoldás:

```
Betûk(X):
  Ciklus amíg X nem üres
    Ki: Elsõ(X)
    X:=Elsõutániak(X)
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Az *Első* és az *Elsőutániak* függvények a nevük alapján értelemszerű funkcióra szolgálnak.

2. Adott egy rendezett számsorozat és egy X érték; X megtalálható a sorozatban. Határozzuk meg X sorszámát!

Az E. és a V. elem között keresünk az A vektorban. Ha a középső kisebb a keresettnél, akkor a középső mögött kell folytatni a keresést, ha nagyobb, akkor a középső előtt, egyébként pedig megtaláltuk a keresett elemet.

```
Keresés(A,X,K,E,V):
  K:=(E+V) div 2
  Elágazás
    A(K)<X esetén E:=K+1
    A(K)>X esetén V:=K-1
  Elágazás vége
  Ha A(K)≠X akkor Keresés(A,X,K,E,V)
Eljárás vége.
```

Itt most nem célszerű teljesen mechanikusan bontani ki a „rekurzív csomót", hanem némi lényeglátással a „szabályszerűen" átírtnál rövidebb iteratív megoldást is kaphatunk. Az ötlet: válasszuk ketté az S eljárást az értékadásra és az

elágazásra, s ebből az utóbbit a ciklusmag elejére beírva megcseréljük az S eljárásbeli eredeti sorrendjüket.

```
Keresés(A,X,K,E,V):
  K:=(E+V) div 2
  Ciklus amíg A(K)≠X
    Elágazás
      A(K)<X esetén E:=K+1
      A(K)>X esetén V:=K-1
    Elágazás vége
    K:=(E+V) div 2
  Ciklus vége
Eljárás vége.
```

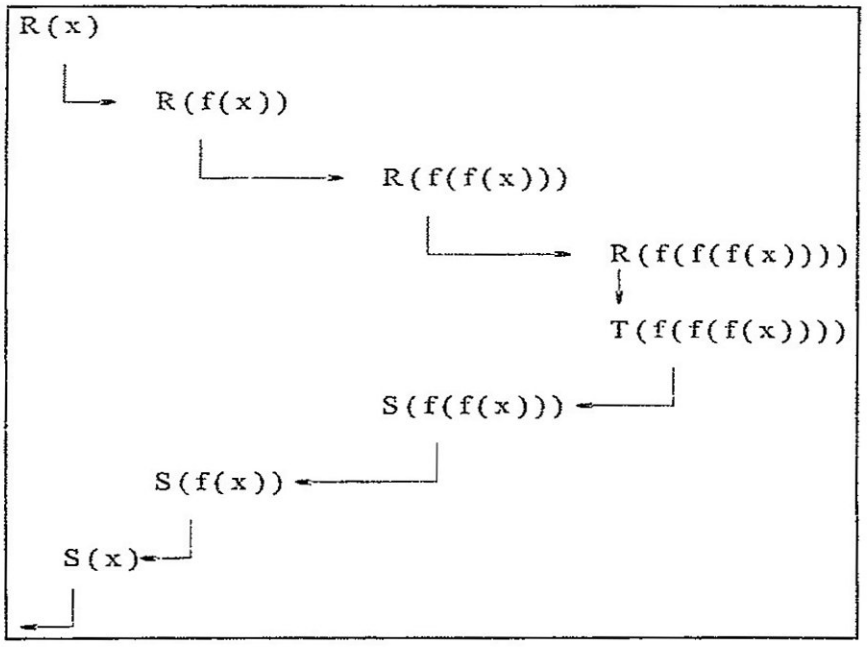
5.3. Balrekurzió

Bonyolultabb a helyzet akkor, ha a rekurzív hívás az eljárás elején található. Először nézzük meg az általános szerkezetét! (A paramétereket két részre bontottuk, X-et csak a rekurzió befejeződése érdekében változtatjuk, Y pedig ettől független.)

```
RekElj(X,Y):
  Ha p(X,Y) akkor  RekElj(f(X),Y)
                   S(X,Y)
          különben T(X,Y)
  Elágazás vége
Eljárás vége.
```

Az eljárások hívásának sorrendjét, paramétereik változását mutatja a 7. ábra.

Ez a szerkezet –célszerűen– mindig olyan feladatra utal, amelynél egy sorozatot fordított sorrendben kell feldolgozni, de a fordított sorrendű bejárás valamilyen oknál fogva nem végezhető el közvetlenül. Ekkor az f függvény az X sorozatnak az első elem nélküli részét adja, s így a rekurzív eljárás eggyel kisebb elemszámú sorozattal hívja meg önmagát.



7. ábra (R=Re(Elj)

Milyen sorozatok rendelkeznek ezzel a tulajdonsággal? Felsorolunk néhányat:

- a sorozat elemei egy soros állomány rekordjai;
- a sorozat elemeit láncolt ábrázolással tároljuk és az aktuális elemtől csak a következőt lehet elérni, az előzőt nem,
- a sorozat elemeit egy sor- vagy veremstruktúrában tároljuk,
- a sorozat elemeit mindig az előző elemből számítjuk, s a sorozat előre meg nem állapítható tagjától visszafelé kell kiírni az elemeket,
- a programozási nyelvünk csak olyan szövegkezelő függvényeket ismer, amelyek a szöveg első karakterét tudják megadni, illetve az első elhagyásával keletkezett részt.

Az aktuális részsorozat előtti elemet kell valahogyan visszakapni. Az átírás feltétele az, hogy *az S és a T eljárás nem változtatja meg X értékét.* Az S és a T eljárásnak lokális változói egyébként lehetnek, de ezek kezdőértékadásáról nekik maguknak kell gondoskodni! A lényeg tehát, hogy csak a paramétervisszaállításáról kell gondoskodni, hiszen a lokális változókat a rekurzív hívásig még nem használtuk.

A sorozat megfordítása azt igényli, hogy az elemeket sorra tároljuk valami olyan adatszerkezetben, amelynél a beírással ellentétes sorrendben is elérhetjük az adatokat. A legegyszerűbb ilyen szerkezet a verem (de egy tömb is használható erre a célra).

Ekkor a nemrekurzív változat a következő lesz (N változó számolja a végrehajtások számát):

```
RekElj(X,Y):
  N:=0
  Ciklus amíg p(X,Y)              [„leszálló" szakasz]
    Verembe(X)
    X:=f(X); N:=N+1
  Ciklus vége
  T(X,Y)
  Ciklus I=1-tõl N-ig             [„felszálló" szakasz]
    Veremből(X); S(X,Y)
  Ciklus vége
Eljárás vége.
```

Gyakran egy valódi sorozat feldolgozásáról van szó. Speciális esetben az S eljárás csak e sorozat aktuálisan első elemével dolgozik, az f pedig elhagyja az első elemét, s így a második ciklus magja egyszerűbb lehet:

```
RekElj(X,Y):
  N:=0
  Ciklus amíg p(X,Y)
    Verembe(Elsõ(X))
    X:=Elsõutániak(X)
    N:=N+1
  Ciklus vége
  T(X,Y)
  Ciklus I=1-tõl N-ig
    Veremből(A)
    X:=Elejére(A,X)
    S(X,Y)
  Ciklus vége
Eljárás vége.
```

⇐

```
Ciklus I=1-tõl N-ig
  Veremből(A); S(A,Y)
Ciklus vége
```

Ha a sorozat elemszámát előre meg tudjuk határozni, akkor az eljárás első két sora így módosul:

```
N:=elemszám(X)
Ciklus I=1-tõl N-ig
```

és a ciklusmagban (természetesen) nem kell gondoskodni N növeléséről.

Akkor van lehetőség veremnélküli ciklus alkalmazására, ha az *f(X) függvénynek van inverze*, azaz az X:=f(X) helyettesítést a rekurzív hívás helye után vissza lehet alakítani. Ha a sorozat elemeit számítani tudjuk az előző, illetve a következő elemből, akkor még erre sincs szükség (f(X) adja az X utáni, f–inverz(X) az X előtti elemet):

```
RekElj(X,Y):
  N:=0
  Ciklus amíg p(X,Y)
    X:=f(X); N:=N+1
  Ciklus vége
  T(X,Y)
  Ciklus I=1-től N-ig
    X:=f-inverz(X)
    S(X,Y)
  Ciklus vége
Eljárás vége.
```

Nézzünk példákat erre az átírásra is!

1. Egy szöveg kiírása betűnként, de fordítva (azaz hátulról előre).

```
Fordítva(X):
  Ha X nem üres akkor Fordítva(Elsőutániak(X))
                      Ki: Első(X)
  Elágazás vége
Eljárás vége.
```

Most a szöveg típusú változókra csak a fent használt kétféle műveletet engedjük meg, valamint az elemszám meghatározást.

Az átírási elv szerint a sorozat elemeit például egy veremben kell tárolni, majd fordított sorrendben kiírni.

```
Fordítva(X):
  N:=elemszám(X)
  Ciklus I=1-től N-ig
    Verembe(Első(X)); X:=Elsőutániak(X)
  Ciklus vége
  Ciklus J=1-től N-ig
    Veremből(A); Ki: A
  Ciklus vége
Eljárás vége.
```

2. Írjuk ki egy adott természetes számnál kisebb 2-hatványokat csökkenő sorrendben!

A vizsgálandó sorozat elemei a 2-hatványok, a sorozat következő tagját az aktuális elemből 2-vel szorzással állíthatjuk elő. A rekurzív megoldás (hívása, ha M-től visszafelé akarjuk kiírni a 2-hatványokat: *Hatványok*(1,M)):

```
Hatványok(K,M):
  Ha K≤M akkor Hatványok(2*K,M); Ki: K
Eljárás vége.
```

A nemrekurzív változatban a sorozat aktuálist megelőző elemét 2-vel osztással kapjuk:

```
Hatványok(K,M):
  N:=0
  Ciklus amíg K≤M
    K:=2*K; N:=N+1
  Ciklus vége
  Ciklus I=1-től N-ig
    K:=K/2; Ki: K
  Ciklus vége
Eljárás vége.
```

Végezetül felhívjuk a figyelmet a 2. függelékre, amely a fenti feladatokat megoldó programpárok futási időit adtuk meg; ezek bizonyítják az iteratív megoldás gyorsabb, hatékonyabb voltát.

Mit tehetünk, ha X egy soros állományban található? Ekkor a rekurzív eljárás is egy kicsit másképpen néz ki:

```
Fordítva(X):
  Ha nem vége(X) akkor Olvas(X,A); Fordítva(X); Ki: A
Eljárás vége.
```

Itt egy A pufferváltozóra van szükségünk a sorozat feldolgozásához, amely a rekurzív eljárásban mindig újra és újra megjelenik. Az *Olvas* utasítás végzi el az *f* függvény feladatát és őrzi meg X első elemét a pufferváltozóban. Átírása:

```
Fordítva(X):
  N:=0
    Ciklus amíg nem vége(X)
      Olvas(X,A); Verembe(A); N:=N+1
    Ciklus vége
    Ciklus J=1-től N-ig
      Veremből(A); Ki: A
  Ciklus vége
Eljárás vége.
```

6. Iteráció és rekurzió

(Nemrekurzív programok átírása rekurzívvá)

Érdekes probléma, amikor egy rekurziómentes programot rekurzívvá kell átírni. Lehet erre egyáltalán szükség vagy egy csupán önmagában érdekes gondolati játék? Sietünk az Olvasót megnyugtatni: nem öncélú agytornáról van szó, hanem a programozás gyakorlata vetette föl e problémát. Ugyanis több olyan programozási nyelv van, amelyben vagy *nem létezik ciklus utasítás*, vagy csak nagyon primitív ciklus van. Ilyenek többek között az ún. *funkcionális nyelvek*, például a Logo nyelv szövegkezelő része. Ekkor összetett feladatok esetén egyes részalgoritmusok megismétlésére *nincs más eszközünk, csak a rekurzió*.

6.1. Ciklusok átírása

A ciklust tartalmazó programok szerencsére mindig könnyen átírhatók rekurzívvá, s így nincs nehéz dolgunk. Lássuk a kétféle ciklus átírását!

```
RekElj(X):
  Ciklus amíg p(X)
    S(X)
  Ciklus vége
Eljárás vége.
```

vagy

```
RekElj(X):
  Ciklus
    S(X)
  amíg p(X)
  Ciklus vége
Eljárás vége.
```

Átírásuk:

```
R eljárás(X):
  Ha p(X) akkor
     S(X)
     RekElj(X)
  Elágazás vége
Eljárás vége.
```

illetve

```
RekElj(X):
  S(X)
  Ha p(X) akkor RekElj(X)
Eljárás vége.
```

Itt most –mint látható– az *elöl-*, illetve a *hátultesztelő ciklusok*ra gondoltunk, de mivel a *számlálásos ciklus* mindig visszavezethető ezek valamelyikére, ezért arra is vonatkozik mondanivalónk.

```
Ciklus I=kezdet-től vég-ig
  ciklusmag
Ciklus vége
```

```
I:=kezdet
Ciklus amíg I≤vég
  ciklusmag; I:=I+1
Ciklus vége
```

Azonnal kiderül, hogy ez utóbbi ciklus átírása –épp amiatt, hogy a számlálónak kezdőértéket kell adni– a fentihez képest kicsit módosult alakban áll elő.

Ha a ciklus előtt vagy mögött valamilyen algoritmusrészlet található, akkor –mivel ezen „keret"–tevékenységeket garantáltan csak egyszer kell elvégezni– külön kell választani őket a rekurzívan megadható ciklusrésztől.

```
RekElj(X):
  Y:=g(X)
  Ciklus amíg p(X,Y)
    S(X,Y)
  Ciklus vége
  X:=h(X,Y)
Eljárás vége.
```

A különválasztást az eredeti eljárás két szintűre bontásával érjük el. A „felső" szint az eredeti szekvenciát őrzi meg. Azaz:

```
RekElj(X):
  Y:=g(X)
  RekOElj(X,Y)
  X:=h(X,Y)
Eljárás vége.
```

A második szintje maga a ciklus (az *RekOElj eljárás*), amit már a megismert módon alakíthatjuk át.

```
RekOElj(X,Y)
  Ciklus amíg p(X,Y)
    S(X,Y)
  Ciklus vége
Eljárás vége.
```

Az *RekOElj eljárás* „szabványos" rekurzív párja:

```
RekOElj(X,Y):
  Ha p(X,Y) akkor S(X,Y); RekOElj(X,Y)
Eljárás vége.
```

Ennyi az „elmélet". Az alábbi példák segítségével nézzük meg, a gyakorlatban hogyan valósítható meg mindez!

1. Adott egy rendezett számsorozat és egy X érték, X megtalálható a sorozatban. Határozzuk meg X sorszámát!

A ciklust tartalmazó megoldás (az E. és a V. elem között keresünk):

```
Keresés(A,X,K,E,V):
  Ciklus
    K:=(E+V) div 2
    Elágazás
      A(K)<X esetén E:=K+1
      A(K)>X esetén V:=K-1
    Elágazás vége
  amíg A(K)≠X
  Ciklus vége
Eljárás vége.
```

A rekurzív megoldás a 2. „szabály" mechanikus alkalmazásával származtatható:

```
Keresés(A,X,K,E,V):
  K:=(E+V) div 2
  Elágazás
    A(K)<X esetén E:=K+1
    A(K)>X esetén V:=K-1
  Elágazás vége
  Ha A(K)≠X akkor Keresés(A,X,K,E,V)
Eljárás vége.
```

2. Adott egy számsorozat és egy X érték. Döntsük el, hogy X megtalálható-e a számsorozatban!

   A ciklust alkalmazó megoldás egyszerű. Vesszük sorra a sorozat elemeit, s ha valamelyik egyenlő a keresettel, akkor a válasz igenlő (IGAZ), ha egyik sem, akkor pedig tagadó (HAMIS).

```
Eldöntés(A,N,X,VAN):
  I:=1
  Ciklus amíg I≤N és A(I)≠X
    I:=I+1
  Ciklus vége
  VAN:=(I≤N)
Eljárás vége.
```

   Ebben az eljárásban a ciklus előtt és után is van tennivaló, ezért a rekurzív változat két eljárásból áll:

```
Eldöntés(A,N,X,VAN):
  I:=1
  Eld(A,N,X,I)
  VAN:=(I≤N)
Eljárás vége.

Eld(A,N,X,I):
  Ha I≤N és A(I)≠X akkor I:=I+1; Eld(A,N,X,I)
Eljárás vége.
```

6.2. Ciklusban számolt függvények átírása

Ciklusban számolt függvény esetén (legjobb a szumma példájára gondolni) a rekurzívra átírás némileg módosul. Az említett példát vizsgálva próbáljunk eljutni az ilyen típusúak általános sémájához! A *Szumma*-függvény szokásos megoldása a következő:

```
Szumma(N):
  Szumma:=0
  Ciklus I=1-től N-ig
    Szumma:=Szumma+A(I)
  Ciklus vége
Függvény vége.
```

A neki megfelelő rekurzív változat megadását az nehezíti, hogy a rekurzió most egy függvény „képében" lép föl, ami az eljárásban ráadásul két helyen is szerepel (a *Szumma:=0*, illetve a *Szumma:=Szumma+A(I)* értékadásokban). A fenti megoldás azt sugallja, hogy a szumma két dolog összegéből jön össze: az első elemnek és a többi szummájának összegéből. A többi szummája ugyanezen definíció alapján számolandó ki. E gondolatot fogalmazzuk meg rekurzívan!

```
Szumma(I,N):
  Ha I≤N akkor  Szumma:=A(I)+Szumma(I+1,N)
       különben Szumma:=0
Függvény vége.
```

A *Szumma* függvény plusz I paraméterére a továbblépés érdekében van szükség. Valójában az (I,N) paraméterkettős jelöli ki az A vektor még hátralévő szeletét. Ezt a zavaró többletparamétert szüntethetjük meg az eredeti *Szumma* függvény egy másik értelmezésű megvalósításával:

```
Szumma(N):
  Szumma:=0
  Ciklus amíg N>0
    Szumma:=Szumma+A(N); N:=N-1
  Ciklus vége
Függvény vége.
```

Ez –szemléletesen szólva– azt mondja, hogy a szumma az utolsóelemnek és az őt megelőzők szummájának összegéből jön ki. Ennek rekurzív megfelelője már nem hagy semmi „kívánnivalót" maga után.

```
Szumma(N):
  Ha N>0 akkor  Szumma:=Szumma(N-1)+A(N)
       különben Szumma:=0
Függvény vége.
```

Ekkor persze az N paramétert csökkentjük, de ennek nem lehet hatása az eljárás hívásának környezetére, ahol az eredeti N-re még szükség lehet. Ebből már körvonalazódik a ciklikusan kiszámolható függvények „általános" szerkezete. Kell, hogy legyen egy, a *függvényre vonatkozó kezdeti értékadás*, a ciklus belsejében pedig a *függvény értékét* a korábbiból a bemenő paraméter egy „részével" *módosító transzformáció*, valamint e *„bemenő részt" továbbléptető értékadás*. (A transzformáció a Szumma példáiban az összeadás volt, a továbbléptetés annak első változatában az I eggyel való növelése, a másodikban pedig az N csök-

kentése. Vegyük észre, mindkét változatban a ciklus további végrehajtása szempontjából érdekes részt jelöltük ki az (I,N)-nel, illetőleg a (0,N)-nel.)

```
FGV(X):
  FGV:=kezdőérték
  Ciklus amíg p(X)
    FGV:=FGV művelet h(X)
    X:=g(X)
  Ciklus vége
Függvény vége.
```

A rekurzív változatban a kezdőértékadást –mint láttuk– különágon kell megoldani. Ott, ahol a függvényérték–kiszámítás legelőször megtörténik, azaz a rekurzív hívás „legbelsejében".

```
FGV(X):
  Ha p(X) akkor  FGV:=h(x) művelet FGV(g(X))
         különben FGV:=kezdőérték
Függvény vége.
```

Vizsgáljuk azt a programot, amelyben el kell dönteni, hogy egy szöveg csupa magánhangzóból áll-e!

```
Mindmagán?(X):
  Mindmagán?:=IGAZ
  Ciklus amíg X nem üres
    Mindmagán?:=Mindmagán? és Magánhangzó?(első(X))
    X:=elsőutániak(X)
  Ciklus vége
Függvény vége.
```

A *Magánhangzó?* függvény IGAZ értéket ad, ha a paramétere magánhangzó, egyébként HAMISat.

```
Mindmagán?(X):
  Ha X nem üres akkor
    Mindmagán?:=(Magánhangzó?(első(X)) és
                 Mindmagán?(elsőutániak(X)))
      különben
    Mindmagán?:=IGAZ
  Elágazás vége
Függvény vége.
```

7. Programozási tételek rekurzívan

Programozási tételek rekurzívra írásakor kétféleképpen járhatunk el. Egyik lehetőség az eddig megismert *nemrekurzív specifikáció átírása* rekurzívra, majd ebből a rekurzív megoldás elkészítése. A másik út az előző fejezet alapján a *nemrekurzív algoritmus átírása* rekurzív algoritmussá.

Az utóbbi esetben garantált lesz az is, hogy a megoldó program nemcsak a feladat-, hanem a programspecifikáció szintjén is azonos, míg az előbbi esetben elképzelhető, hogy a rekurzív algoritmus a több lehetséges megoldásból mást ad meg, mint a nemrekurzív.

Megállapodás:

1, A rövidség kedvéért csak a tételbeli specifikáció *utófeltételét* fogjuk leírni, és természetesen a kiszámítást végző *függvény algoritmusát*. (A teljes specifikáció a *μLógia* 19 kötetében megtalálható.) Emlékeztetőül egy mondattal utalunk a tételben megfogalmazott feladatra.

2, Az utófeltételnek megfelelő algoritmusok használni fognak egy *globális* X tömböt, amely a feldolgozandó sorozatot tartalmazza. A „tisztességes" megoldásban az X is paraméter lenne, méghozzá érték szerinti paraméter, hiszen a rekurzív *függvény* nem változtathatja meg az *argumentuma* értékét. Ez azonban felvet egy súlyos memóriagazdálkodási problémát: az érték szerinti paramétereket az (függvény)eljárás hívásakor mindig egy újabb területre (ti. a *verem*be) kell lemásolni. Ha egy tömb érték szerinti paramétere egy rekurzív (függvény)eljárásnak, akkor pillanatok alatt elérhetjük, hogy betelik a memória (pontosabban a verem).

7.1. Sorozathoz érték rendelése

7.1.1. Sorozatszámítás

Egy *n* elemű sorozathoz (X) adjuk meg az $F(X_1,X_2,...X_n)$ értéket!

Utófeltétel: $Számít(n) = \begin{cases} F_0 & ha \quad n = 0 \\ f(Számít(n-1), X_n) & ha \quad n > 0 \end{cases}$

```
Számít(N):
  Ha N=0 akkor  Számít:=F0
         különben Számít:=f(Számít(N-1),X(N))
Függvény vége.
```

7.1.2. Eldöntés

Egy *n* elemű sorozathoz (X) adjuk meg, hogy létezik-e benne adott (T) tulajdonságú elem!

$$\text{Utófeltétel: } Eldönt(n) = \begin{cases} hamis & ha \quad n = 0 \\ igaz & ha \quad n > 0 \text{ és } T(X_n) \\ Eldönt(n-1) & egyébként \end{cases}$$

Azaz az Eldönt függvény egy 0 elemű sorozatra *hamis* értéket ad, ha egy nem üres sorozat utolsó eleme T tulajdonságú, akkor *igaz*at, egyébként pedig azt, amit az Eldönt függvény az első *n-1* elemre ad.

```
Eldönt(N):
  Elágazás
     N=0      esetén  Eldönt:=hamis
   T(X(N))    esetén  Eldönt:=igaz
    egyéb     esetben Eldönt:=Eldönt(N-1)
  Elágazás vége
Függvény vége.
```

7.1.3. Kiválasztás

Egy *n* elemű sorozatnak (X) adjuk meg egy adott (T) tulajdonságú eleme sorszámát, ha feltételezhetjük, hogy ilyen elem biztosan van!

$$\text{Utófeltétel: } Kiválaszt(n) = \begin{cases} n & ha \quad T(X_n) \\ Kiválaszt(n-1) & egyébként \end{cases}$$

Tehát *n* elemből úgy kell kiválasztani egy T tulajdonságút, hogy ha az utolsó T tulajdonságú, akkor ő a keresett elem, ha nem, akkor pedig az, amit az első *n–1* elemből tudunk kiválasztani.

```
Kiválaszt(N):
  Ha T(X(N)) akkor  Kiválaszt:=N
             különben Kiválaszt:=Kiválaszt(N-1)
Függvény vége.
```

7.1.4. Keresés

Egy n elemű sorozatnak (X) adjuk meg egy adott (T) tulajdonságú eleme sorszámát, ha van ilyen elem!

$$\text{Utófeltétel: } Keres(n) = \begin{cases} (hamis, ?) & ha \quad n = 0 \\ (igaz, n) & ha \quad T(X_n) \\ Keres(n-1) & egyébként \end{cases}$$

A Keres függvény két értékkomponenst ad eredményül. Az első azt tartalmazza, hogy van-e a keresett tulajdonsággal rendelkező elem, a második pedig azt, hogy melyik ez.

```
Keres(N):
  Elágazás
     N=0     esetén Keres:=(hamis,0 [bármi!])
    T(X(N)) esetén Keres:=(igaz,N)
     egyéb esetben Keres:=Keres(N-1)
  Elágazás vége
Függvény vége.
```

A programozási nyelvek széles körében nem használható ez a megoldás. A nyelvek ugyanis sokszor korlátozzák a függvényérték típusát, azaz csak elemi típust engednek meg függvényértékként. Ekkor nincs más lehetőségünk, mint a rekurzív függvény helyettesítése rekurzív eljárással.

```
Keresés(N,Van,Sorszám):
  Elágazás
     N=0     esetén Van:=hamis
    T(X(N)) esetén Van:=igaz; Sorszám:=N
     egyéb esetben Keresés(N-1,Van,Sorszám)
  Elágazás vége
Eljárás vége.
```

7.1.5. Megszámolás

Egy n elemű sorozathoz (X) adjuk meg egy adott (T) tulajdonsággal rendelkező elemei számát!

Utófeltétel: $Számol(n) = \begin{cases} 0 & ha \quad n = 0 \\ Számol(n-1)+1 & ha \quad T(X_n) \\ Számol(n-1) & egyébként \end{cases}$

A megoldás alapelve: n elem közötti T tulajdonságú elemek számához először határozzuk meg az első n–1 elem közötti T tulajdonságúak számát, majd ezt növeljük meg 1-gyel, ha az utolsó elem t tulajdonságú.

```
Számol(N):
  Elágazás
     N=0     esetén Számol:=0
    T(X(N)) esetén Számol:=Számol(N-1)+1
     egyéb esetben Számol:=Számol(N-1)
  Elágazás vége
Függvény vége.
```

7.1.6. Maximumkiválasztás

Egy n elemű sorozatnak (X) adjuk meg legnagyobb elemének sorszámát!

Utófeltétel: $Maximum(n) = \begin{cases} n & ha \quad X_n \geq Maximum(n-1) \\ Maximum(n-1) & egyébként \end{cases}$

A megoldás alapelve: n elem maximuma az utolsó elem, ha az nagyobb vagy egyenlő, mint az első n–1 elem maximuma, egyébként pedig az első n–1 elem maximuma.

```
Maximum(N):
  Ha X(N)≥Maximum(N-1) akkor   Maximum:=N
                       különben Maximum:=Maximum(N-1)
Függvény vége.
```

Bár a megoldás helyes, a végrehajtási idő szempontjából „borzasztó". Ha a sorozat kezdetben növekvően rendezett, akkor a ≥ reláció mindig hamis lesz, s így a Maximum függvényt a feltétel kiértékelése közben is meghívja és a különbenágon is. Ebben az esetben a Maximum függvény hívásainak száma 2^n–1. A jó megoldás lényege: a Maximum függvényt a feltétel kiértékelése előtt hívjuk meg, értékét egy változóba elhelyezzük, majd azt használjuk az elágazásban.

```
Maximum(N):
  M:=Maximum(N-1)
  Ha X(N)≥M akkor  Maximum:=N különben Maximum:=M
Függvény vége.
```

7.2. Sorozathoz sorozat rendelése

7.2.1. Másolás

Egy n elemű sorozatot (X) másoljunk le, miközben minden elemére alkalmazzuk az f függvényt!

Utófeltétel: $Másol(n) = Végére\ \left(f(X_n), Másol(n-1)\right)$

A megoldásban meggondolandó, hogy a bemenő tömbhöz hasonlóan az eredmény is globális változó legyen-e, vagy pedig függvényérték. A programozási nyelvek széles körében sajnos a tömbökhöz nem definiálnak konstrukciós függvényeket (mint pl. a specifikációban szereplő *Végére* nevű függvény), így mindenképpen más megoldás után kell néznünk. Egy olyan *eljárás*t használunk, amely minden hívásra n elemű tömböt ad vissza eredményül:

```
Másol(N,Y):
  Ha N>0 akkor Másol(N-1,Y); Y[N]:=f(X(N))
Eljárás vége.
```

Természetesen elképzelhető az is, hogy az Y tömb *globális* változó. Ekkor a megoldás minimális mértékben változik: nem jelenik meg az eljárás paraméterlistáján az Y tömb.

7.2.2. Kiválogatás

Egy n elemű sorozatnak (X) adjuk meg egy adott tulajdonsággal (T) rendelkező elemeit!

$$\text{Utófeltétel: } Kiválogat(n) = \begin{cases} \{\} & ha \quad n = 0 \\ Végére\ (X_n, Kiválogat(n-1)) & ha \quad T(X_n) \\ Kiválogat(n-1) & egyébként \end{cases}$$

Ha a specifikációt tömbökre akarjuk átfogalmazni, akkor az egyetlen sorozat típusú eredmény helyett egy tömböt és egy aktuális elemszámot kell használnunk:

$$\text{Utófeltétel: } Kiválogat(n) = \begin{cases} (0,\{\}) & ha \quad n = 0 \\ Kiválogat(n-1) \oplus (1, X_n) & ha \quad T(X_n) \\ Kiválogat(n-1) & egyébként \end{cases},$$

$$\text{ahol } (Db, Y) \oplus (1, X) = (Db+1, Végére\ (Y, X))$$

A megoldást *eljárás*ként írjuk meg, melynek paramétere a darabszám és az eredménysorozatot tartalmazó tömb:

```
Kiválogat(N,Db,Y):
  Elágazás
     N=0      esetén  Db:=0
    T(X(N))   esetén  Kiválogat(N-1,Db,Y)
                      Db:=Db+1; Y(Db):=X(N)
     egyéb eseben     Kiválogat(N-1,Db,Y)
  Elágazás vége
Eljárás vége.
```

A megoldást rövidebbre írhatjuk, ha a belső elágazás ágairól kiemeljük a közös részt:

```
Kiválogat(N,Db,Y):
  Ha N=0 akkor Db:=0
       különben Kiválogat(N-1,Db,Y)
                Ha T(X(N)) akkor Db:=Db+1; Y(Db):=X(N)
  Elágazás vége
Eljárás vége.
```

7.2.3. Rendezés maximumkiválasztással

Egy n elemű sorozatot (X) rendezzünk maximumkiválasztásos rendezéssel!

Utófeltétel: $Rendez(n,X) = \begin{cases} X & ha \quad n=1 \\ Rendez\left(n, Csere(X_{Maximum(n)}, X_n)\right) & egyébként \end{cases}$,

ahol a Csere függvény az X tömb két elemét cseréli fel.

A megoldás alapvető problémája, hogy rendezések legtöbbje helyben rendezést definiál, amit igen nehéz függvényként felírni. Így az eddigiekhez hasonlóan egy *eljárás*t írunk, amely a globális X tömb elemeit rendezi.

```
Rendez(N):
  Ha N>1 akkor Csere(X(Maximum(N)),X(N)); Rendez(N-1)
Eljárás vége.
```

7.2.4. Beillesztéses rendezés

Egy n elemű sorozatot (X) rendezzünk beillesztéses rendezéssel!

Utófeltétel: $Rendez(n,X) = \begin{cases} X & ha \quad n=1 \\ Beilleszt\left(X_n, Rendez(n-1)\right) & egyébként \end{cases}$,

ahol $Beilleszt(Y,X,n) = \begin{cases} \{Y\} & ha \quad n=0 \\ Végére \ (Y,X) & ha \quad Y \geq X_n \\ Végére \ \left(X_n, Beilleszt(Y,X,n-1)\right) & egyébként \end{cases}$

A specifikációnak megfelelően az algoritmus is két rekurzív eljárásból áll:

```
Rendez(N):
  Ha N>1 akkor Rendez(N-1); Beilleszt(X(N),N-1)
Eljárás vége.

Beilleszt(Y,N):
  Elágazás
      N=0   esetén X(1):=Y
     Y≥X(N) esetén X(N+1):=Y
    egyéb esetben  X(N+1):=X(N); Beilleszt(Y,N-1)
  Elágazás vége
Eljárás vége.
```

7.3. Sorozathoz sorozatok rendelése

7.3.1. Szétválogatás

Egy n elemű sorozatot (X) válogassuk szét egy adott tulajdonsággal (T) rendelkező, valamint azzal nem rendelkező elemekre!

Utófeltétel: $Szétválogat(n) = \begin{cases} (\{\},\{\}) & ha \quad n=0 \\ Szétválogat(n-1) \oplus (X_n,\{\}) & ha \quad T(X_n), \text{ ahol} \\ Szétválogat(n-1) \oplus (\{\},X_n) & egyébként \end{cases}$

$$(A,B)\oplus(X,\{\})=(Végére(X,A),B),\ (A,B)\oplus(\{\},X)=(A,Végére(X,B))$$

A megoldást *eljárás*ként írjuk meg, melynek paramétere két darabszám és az eredménysorozatokat tartalmazó tömbök:

```
Szétválogat(N,DbY,Y,DbZ,Z):
  Ha N=0 akkor DbY:=0; DbZ:=0
     különben Szétválogat(N-1,DbY,Y,DbZ,Z)
              Ha T(X(N)) akkor DbY:=DbY+1
                               Y(DbY):=X(N)
                         különben DbZ:=DbZ+1
                               Z(DbZ):=X(N)
              Elágazás vége
  Elágazás vége
Eljárás vége.
```

A szétválogatást megoldhatjuk helyben is, melynek eredményeképpen az X tömb elejére kerülnek a T tulajdonságú elemek, a végére pedig a nem T tulajdonságúak.

```
Szétválogat(E,U):
  Y:=X(1)
  Ha E<U akkor Hátulról_előre(E,U)
  X(U):=Y
Eljárás vége.
```

A hátulról előre, illetve az elölről hátulra mozgatás két egymást hívó eljárás, azaz egymáson keresztül közvetetten rekurzívak.

```
Hátulról_előre(E,U):
  Hátulrólkeres(E,U)
  Ha E<U akkor X(E):=X(U); Elölről_hátra(E+1,U)
Eljárás vége.

Elölről_hátra(E,U):
  Elölrőlkeres(E,U)
  Ha E<U akkor X(U):=X(E); Hátulról_előre(E,U-1)
Eljárás vége.

Hátulrólkeres(E,U):
  Ha E<U és nem T(X(U)) akkor Hátulrólkeres(E,U-1)
Eljárás vége.

Elölrőlkeres(E,U):
  Ha E<U és T(X(U)) akkor Elölrőlkeres(E+1,U)
Eljárás vége.
```

7.4. Sorozatokhoz sorozat rendelése

7.4.1. Metszet

Adjuk meg egy *n*, illetve egy *m* elemű sorozat (X,Y) közös elemeit!

Utófeltétel:

$$Metszet(X,n,Y,m)=\begin{cases}\{\} & ha \quad n=0 \, vagy \quad m=0\\ Végére(X_n, Metszet(X,n-1,Y,m)) & ha \quad Eleme(X_n,Y,m)\\ Metszet(X,n-1,Y,m) & egyébként\end{cases},$$

$$\text{ahol } Eleme(A,Y,m)=\begin{cases}hamis & ha \quad m=0\\ igaz & ha \quad A=Y_m\\ Eleme(A,Y,m-1) & egyébként\end{cases}$$

A két rekurzív képletnek megfelelően két rekurzív *eljárást* készítünk a metszet meghatározására:

```
Metszet(X,N,Y,M,Z,Db):
  Ha N=0 akkor  Db:=0
       különben Metszet(X,N-1,Y,M,Z,Db)
                Ha Eleme(X(N),Y,M) akkor Db:=Db+1
                                         Z(Db):=X(N)
  Elágazás vége
Eljárás vége.

Eleme(A,Y,M):
  Elágazás vége
     M=0    esetén Eleme:=hamis
     A=Y(M) esetén Eleme:=igaz
    egyéb esetben  Eleme:=Eleme(A,Y,M-1)
Függvény vége.
```

7.4.2. Egyesítés

Adjuk meg egy *n*, illetve egy *m* elemű sorozat (X,Y) egyesítését!

Utófeltétel:

$$Unió(X,n,Y,m)=\begin{cases}X & ha \quad m=0\\ Végére(Y_m, Unió(X,n,Y,m-1)) & ha \quad nem \quad Eleme(Y_m,X,n),\\ Unió(X,n,Y,m-1) & egyébként\end{cases}$$

$$\text{ahol } Eleme(A,X,n)=\begin{cases}hamis & ha \quad n=0\\ igaz & ha \quad A=X_n\\ Eleme(A,X,n-1) & egyébként\end{cases}$$

A két rekurzív képletnek megfelelően két rekurzív *eljárást* készítünk a metszet meghatározására:

```
Unió(X,N,Y,M,Z,Db):
  Ha M=0 akkor  Db:=N; Z:=X
       különben Unió(X,n,Y,m-1,Z,Db)
                Ha nem Eleme(Y(M),X,N) akkor
                       Db:=Db+1; Z(Db):=X(N)
                Elágazás vége
  Elágazás vége
Eljárás vége.

Eleme(A,X,N):
  Elágazás vége
      N=0   akkor Eleme:=hamis
     A=X(N) akkor Eleme:=igaz
    egyéb esetben Eleme:=Eleme(A,X,N-1)
  Elágazás vége
Függvény vége.
```

7.4.3. Visszalépéses keresés

A visszalépéses keresés egy speciális esetének feladata *n* db *n* elemû sorozatból egy-egy elem kiválasztása úgy, hogy az elemek önmagukban kiválaszthatók legyenek (*ft* függvény), valamint egymástól függõen is választhassuk õket (*fk* függvény).

Az alábbi algoritmus a feladat kiválogatásos változatát oldja meg, részben ciklussal, részben rekurzívan.[1]

```
Visszalépéses_keresés(N,Db,Y):
  Db:=0; Backtrack(1,N,X,Db,Y)
Eljárás vége.

Backtrack(I,N,X,Db,Y):
  Ha I=N+1 akkor Db:=Db+1; Y(Db):=X
  különben Ciklus J=1-től N-ig
             Ha ft(I,J) és nem Rossz(I,J,1))
               akkor X(I):=J; Backtrack(I+1,N,X,Db,Y)
           Ciklus vége
  Elágazás vége
Eljárás vége.

Rossz(I,J,K):
  Elágazás
        I=K                esetén   Rossz:=hamis
    nem fk(I,J,K,X(K))     esetén   Rossz:=igaz
     egyéb esetben                  Rossz:=Rossz(I,J,K+1)
  Elágazás vége
Függvény vége.
```

Ha a valamilyen szempontból legjobb megoldást keressük, akkor egy kicsit át kell alakítani az algoritmust. Legyen az *f* az a függvény, ami egy megoldáshoz

[1] Azért közöljük ezt a félig rekurzív változatot, mert bizonyos szempontból ez a legegyszerûbb megfogalmazása a visszalépéses keresésnek.

megadja az értékét. Tegyük fel, hogy az *f* függvény minden megoldásra nagyobb 0-nál, s azt a megoldást keressük, amelyre az *f* függvény maximális értékű.

```
Visszalépéses_keresés(N,Érték,Y):
  Érték:=0; Backtrack(1,N,X,Érték,Y)
Eljárás vége.

Backtrack(I,N,X,Érték,Y):
  Ha I=N+1 akkor Ha Érték<f(X) akkor Y:=X
  különben Ciklus J=1-től N-ig
             Ha ft(I,J) és nem Rossz(I,J,1))
               akkor X(I):=J; Backtrack(I+1,N,X,Db,Y)
           Ciklus vége
  Elágazás vége
Eljárás vége.
```

8. Gyorsrendezés — Quicksort

A Quicksort algoritmussal már foglalkoztunk az előző fejezetekben, most megvizsgáljuk a *rekurzív*, a *félig rekurzív* és a *nemrekurzív* változatát.

Az N elemű A vektort sorbarendező eredeti rekurzív változat így nézett ki:

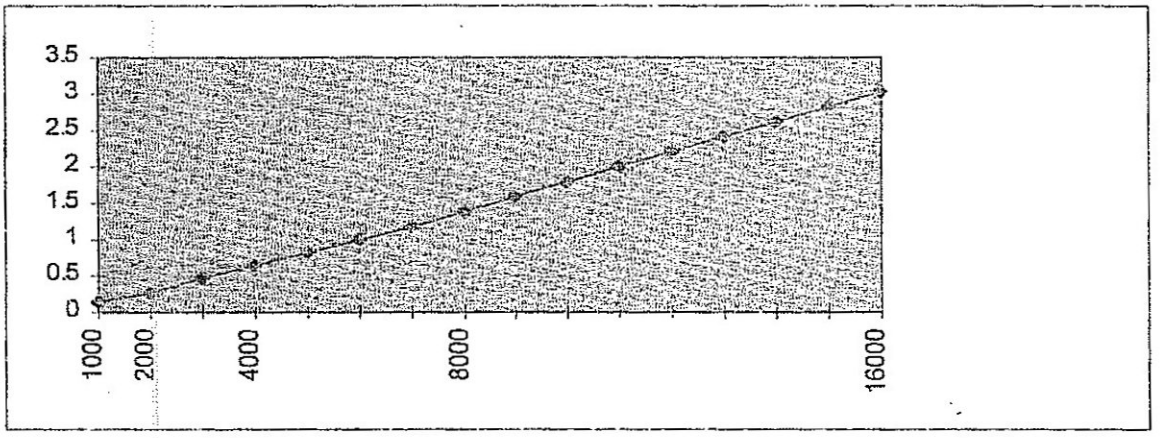
```
Quick(A,E,V): [A „rendezendő" rész Eleje, Vége]
  Szétválogat(A,E,V,K)
  Ha K-E>1 akkor Quick(A,E,K-1)
  Ha V-K>1 akkor Quick(A,K+1,V)
Eljárás vége.
```

A megoldást kétféle szempontból elemezzük ebben a fejezetben. Az egyik a rekurzív és az egyre kevesebb rekurzív hívást tartalmazó változatok hatékonysági elemzése. A másik szempontból pedig a szétválogatás különböző változatai alapján vizsgáljuk a futási időt.

Az eredeti eljárás átlagos futási ideje néhány elemszámra:

N=1000	N=2000	N=4000	N=8000	N=16000
0.14	0.27	0.65	1.38	3.02

A futási időket grafikusan ábrázolva szinte lineáris függvényt látunk:



A megoldásban szereplő két rekurzív hívásból az egyik *jobbrekurzió*, amire van egyszerű átírási szabály:

```
Quick(A,E,V):
  Ciklus
    Szétválogat(A,E,V,K)
    Ha K-E>1 akkor Quick(A,E,K-1)
    E:=K+1
  amíg V-E>0
  Ciklus vége
Eljárás vége.
```

Ez a megoldás a rekurzív hívások számát átlagosan a felére csökkenti, így elvileg hatékonyabb. A mért idők:

N=1000	N=2000	N=4000	N=8000	N=16000
0.16	0.28	0.62	1.35	2.91

Tapasztalatunk szerint a mért idők szinte ugyanazok, az elemszám növelésével kicsit jobb, mint a teljesen rekurzív változat.

A másik rekurzív hívás azonban nem fejthető ki egyszerűen ciklussá, mert sem a bal-, sem a jobbrekurzió átírási szabálya nem alkalmazható rá. Ebben az esetben nincs más megoldás, mint a *verem* alkalmazása a még rendezendő részek határainak megőrzésére.

Az iteratív megoldás alapelve: minden szétválogatás után válasszuk a baloldali részt, a jobboldali határait pedig tegyük verembe, ha még foglalkozni kell vele. Ha a baloldali részt már nem kell tovább rendezni, akkor vegyük ki a veremből az utoljára megőrzött intervallum határait, s folytassuk vele a fenti tevékenységet. A rendezés akkor fejeződik be, ha már nem kell további részekkel foglalkoznunk.

```
Quick(A,E,V):
  Verembe(0,0) [ tetszőleges értékek!]
  Ciklus
    Ciklus
      Szétválogat(A,E,V,K)
      Ha V-K>1 akkor Verembe(K+1,V)
      V:=K-1
    amíg V-E>0
    Ciklus vége
    Veremből(E,V)
  amíg E>0
  Ciklus vége
Eljárás vége.
```

A nemrekurzív változat futási ideje:

N=1000	N=2000	N=4000	N=8000	N=16000
0.14	0.34	0.77	1.5	3.23

A futási idők kicsit *rosszabb*ak, mint az előző két változatnál, pedig úgy gondolhattuk, hogy a rekurzió megszüntetése gyorsítja a programot. A magyarázat ott van, hogy a verem megvalósítás viszont lassít (eljárásokat kell hívni, tömböt indexelni stb.)

A gyorsrendezés második problémáját a szétválogatás okozza. Mivel optimális esetben a szétválogatás két egyenlő részre osztja az N elemű vektort, ezért a rendezendő szakaszok hossza felére csökken. Pontosan tudunk számolni, ha N értéke 2^K-1 alakú. Ekkor először 2 db $2^{K-1}-1$, majd 4 db $2^{K-2}-1$, a K–1. lépésben pedig

2^{K-1} db 1 elemszámú résszel kell foglalkozni. Így a rendezés lépésszáma kb. K*N, azaz $N*\log_2 N$. Ez azonban csak a fenti feltételezés esetén igaz. Ha például a sorozat eleve rendezett, akkor a rendezés lépésszáma N*(N–1)-re nő, ekkor ugyanis a szétválogatás egy üres és egy N–1 elemszámú részt eredményez.

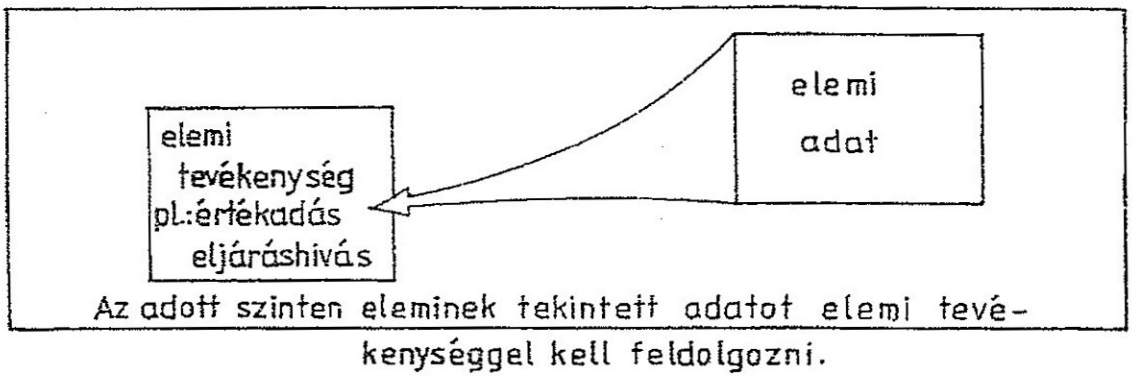
A megoldás többféle lehet, de mindegyikben a szétválogatás alapváltozatát módosítjuk. Ebben a sorozatot mindig az aktuális első elem szerint válogattuk két részre. A szétválogatás előtt alkalmazzuk az alábbiak valamelyikét:

- a középső elemet cseréljük fel az elsővel;
- egy véletlenszerűen választott elemet cseréljünk fel az elsővel;
- az első, a középső és az utolsó közül a nagyságrend szerinti középsőt cseréljük fel az elsővel (ez egy becslése az igazi középső elemnek).

8. Rekurzív adatszerkezetek és algoritmusok

E fejezetben a rekurzió és az adatszerkezetek kapcsolatáról szeretnénk elmélkedni. Talán megbocsátja nekünk a kedves Olvasó, ha egy kicsit messzebbről kezdjük vizsgálódásunkat: az algoritmusok és adatszerkezetek viszonyáról általában.

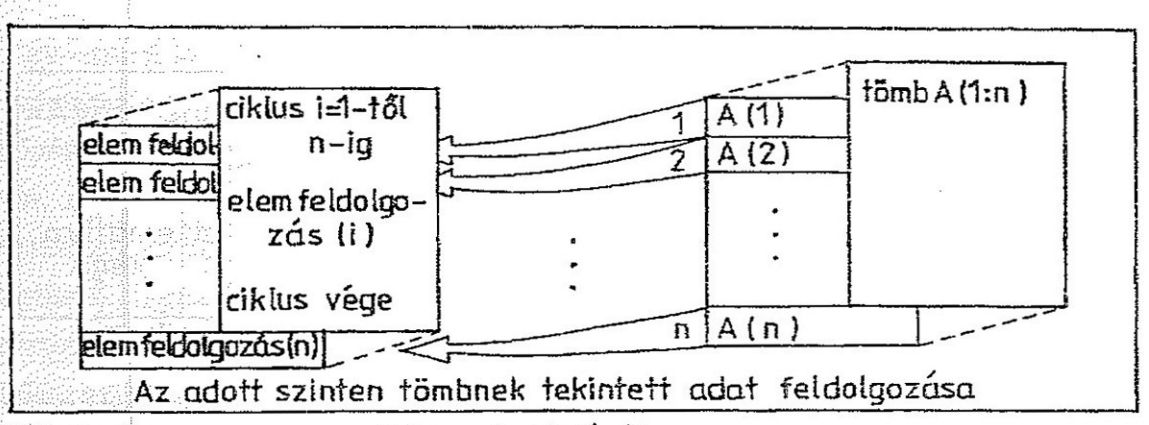
A „legelemibb" utasítástípus kétségkívül az értékadás, amely egy adatobjektum másolatát mozgatja egy(ség)ként, globálisan a célváltozóba. Így –mintegy sugallva– a megfeleltetést: az utasítás és az „arctalan", „szerkezetnélküli" adatok típusa között. (Ez a hozzárendelés a későbbiek során még nyilvánvalóbbá válik.) Megjegyezzük, hogy ennek nem mond ellent az a gyakran alkalmazott értékadásfajtánk, amelynek segítségével egy vektornak, mátrixnak, ... stb. egyetlen utasítással adtunk értéket. (Pl. A:=0.) Ennek az utasításnak a szintjén nem figyelünk az A szerkezetére, s mint egy „strukturálatlan egységet" használjuk.



8. ábra

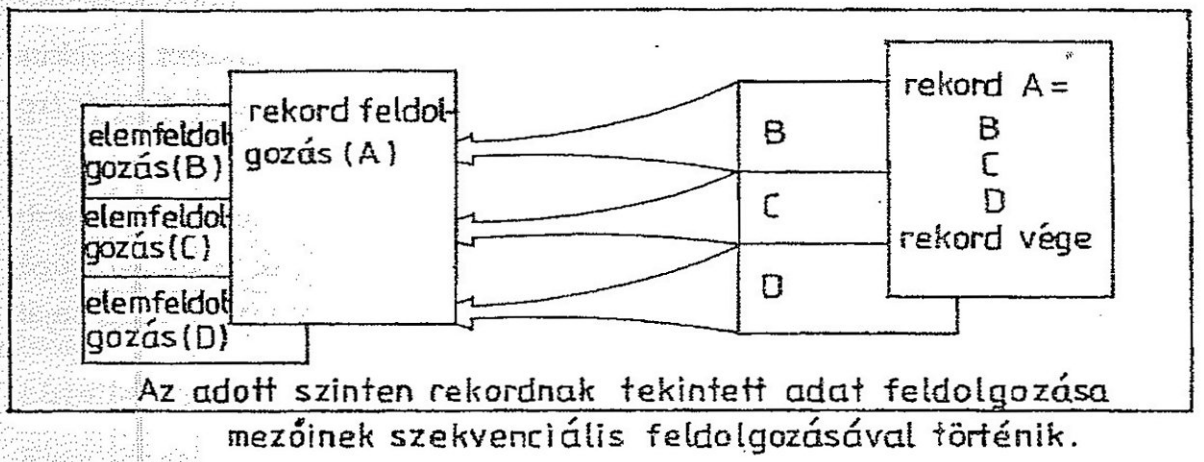
Amikor a program írása során (a program finomítása útján) eljutunk egy ilyen részeire vonatkozó vizsgálatának vagy transzformációjának megvalósításához, akkor már lényeges szerephez jut annak szerkezete.

Az adatobjektum azonos valamikre való bontása révén kapjuk a vektorokat, mátrixokat: általában mindazon struktúrákat, amelyek azonos típusú elemek sokaságából állnak. Ilyen adatok feldolgozását ciklussal végezhetjük el (ezt éppen az elemeinek azonos típusú volta teszi lehetővé).



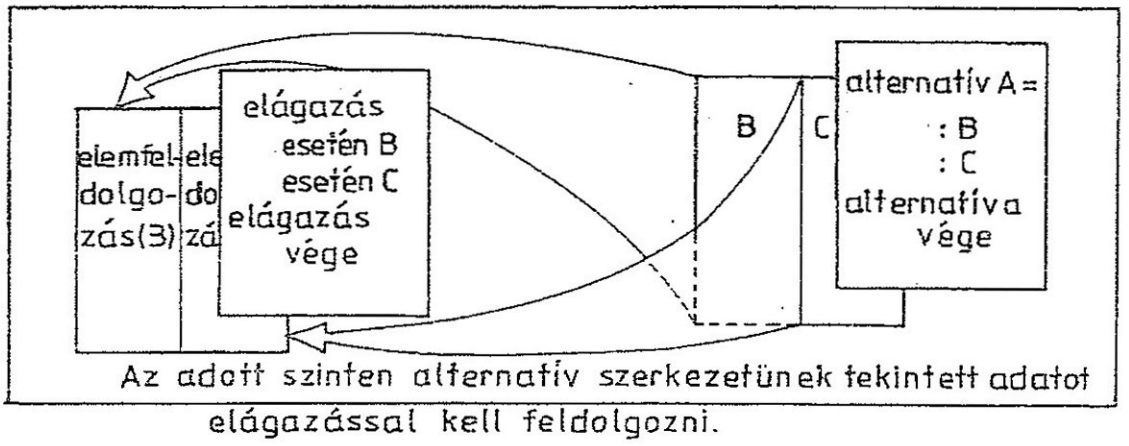
9. ábra

A különböző típusú részekre, elemibb adatokra „széteső" szerkezetek a rekordok (olyan változók, amelyek értékhalmazukat valahány halmaz direktszorzatából veszik fel). Ehhez a szerkezethez az egyes részek feldolgozását elvégző tevékenységek egymásutánja rendelhető.



10. ábra

Ha az adattípusra egyfajta „kétarcúság" jellemző, azaz vagylagosságot tartalmaz (pl.: egy személy-nyilvántartásban a hölgyeket a néven túl a leánykori nevük azonosítja, az urakat pedig katonai igazolványuk száma), akkor a feldolgozás szükségszerűen valamilyen elágazásra vezet.



11. ábra

Ahogy a feladat megoldása során egyre „mélyebbre" jutunk, úgy körvonalazódik egyre tisztábban az adataink szerkezete is. Vagyis amint az algoritmus-megfogalmazásban a finomítás legfontosabb eszköze *az algoritmus elemibbekre való lebontása*, úgy az adatok finomításának a *strukturálás*, vagyis a szerkezethozzárendelés. Semmi csodálkozni való nincs ezek után azon, hogy oly fontosnak éreztük annak meggondolását, hogy az egyik legbonyolultabb adatszerkezet –a rekurzív adatszerkezet– miként határozza meg a „hozzátartozó" programok típusát (rekurzív, nem rekuzív), szerkezetét.

Mik is azok a rekurzív adatszerkezetek? Kezdjük megint úgy, ahogy a rekurzív algoritmusok megközelítésénél tettük: induljunk ki a definiálásuk jellegzetességéből! Hogy ti. a lényeg itt is ugyanaz: az adatszerkezet leírásánál hivatkozunk saját magára, mint egy már jól definiált szerkezetre.

Példák:

$$Lista = \begin{cases} \text{üres lista} \\ elem + lista \end{cases}$$

$$Bináris\ \ fa = \begin{cases} \text{üres bináris } fa \\ \text{bináris } fa + elem + \text{bináris } fa \end{cases}$$

A „kétarcúság" miatt például a lista feldolgozása ilyeténképpen sejthető:

```
Lista feldolgozása:
  Ha üres akkor  kész
        különben ?
Eljárás vége.
```

Sejtésünk a kérdéses (?) részre vonatkozóan (rekordra gondolva):

```
?:
  Elem feldolgozása
  Lista feldolgozása
Eljárás vége.
```

Így –mint látjuk– nagyon egyszerűen, –mondhatnánk– módon lehet definiálni az adatszerkezeteket (pontosan erre ad lehetőséget az ELAN nyelv). Kódolási lehetőségek azonban egyes nyelveken (pl. a Pascalban) lényegesen szerényebbek e téren, mint várhatnánk az algoritmikus struktúrákkal vont analógiák alapján. A nehézséget az okozza, hogy az így felírt szerkezet a szokásos eszközökkel nehezen kezelhető, ugyanis a tárolásához szükséges hely fordításkor, sőt futáskor sem határozható meg fixen. Ezért a nyelvek többsége nem engedi meg az adatszerkezetekre a kifejezett (kinyilvánított) rekurziót. Ennek megkerülésére több megoldás is létezik. A legkézenfekvőbb az ún. *mutatóval való ábrázolás*. Így például a lista reprezentálása egy mutatóval és az elemek felsorolásával történik. Az elem persze a rákövetkezés kapcsolatát is kell, hogy tartalmazza (megint csak egy mutató segítségével). Az elem ilyen „kibővített" szerkezetéről nekünk kell persze rendelkezni (a „kifejezett rekurzió" hiánya miatt).

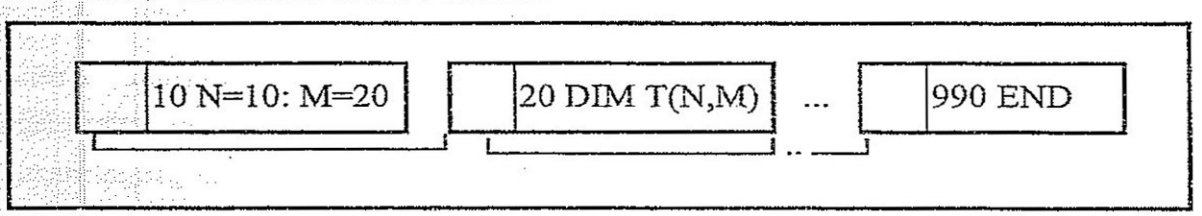
A *mutató típusú változó* tulajdonképpen kétféle információt tartalmaz: egyrészt azt a *memóriacím*et, ahol az a változó található, amire mutat; másrészt pedig ennek *a változónak a típusát*, amiből meghatározható a tárolásához szükséges memória mérete. Ebből két dolog következik:

- mivel a mutató típusú változó memóriacímet reprezentál, ezért használatával nagyon gyors memóriakezelést érhetünk el,
- „általános" mutató típusú változó nincs, hanem csak valamire (valamilyen rögzített típusúra) mutató, pl.: listaelemre mutató, bináris fára mutató stb.

Példák: *(mutató használatára)*

Lista: első elemre ***mutató*** *+* ***elemek*** *sorozata*

Elem: ***érték*** *+ következő elemre* ***mutató***

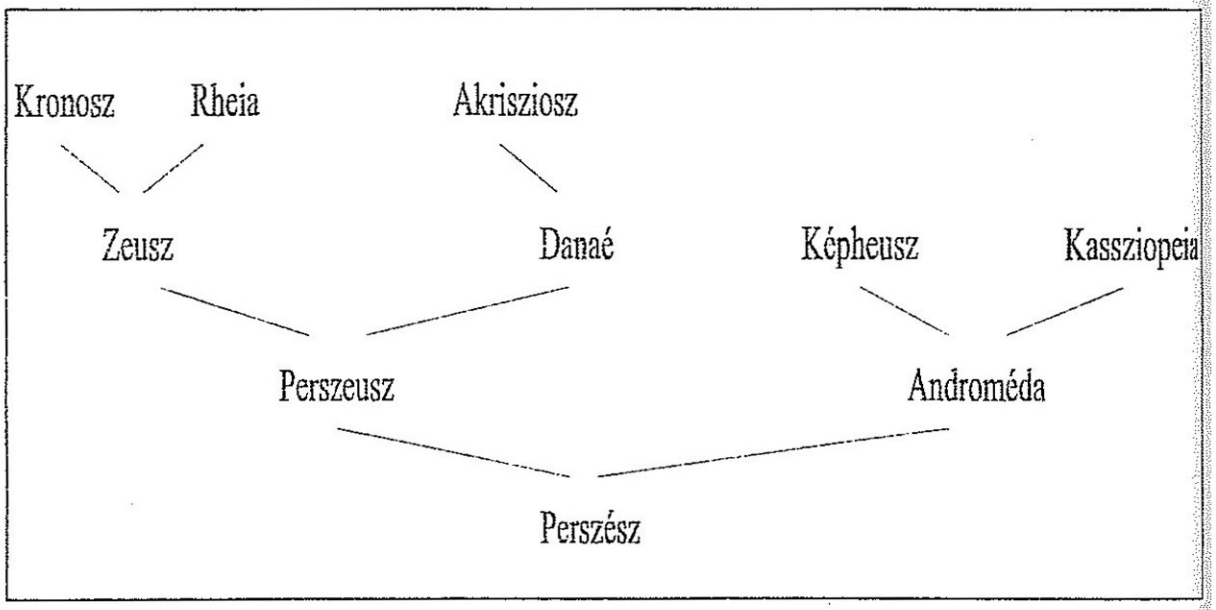


12. ábra: lista

A BASIC forrásprogramot többnyire listaszerkezettel ábrázolják.

Bináris fa: gyökérre ***mutató*** *+* ***elemek*** *sorozata*

Elem: baloldali fára ***mutató*** *+* ***érték*** *+ jobboldali fára* ***mutató***



13. ábra: bináris fa
Perszész családfája, akitől a perzsa királyok származtak

Egy olyan nyelven, ahol a kifejezett rekurzióra is van lehetőség (pl. Logo), a 13. ábrának megfelelő szerkezet:

((((Kronosz) Zeusz (Rheia)) Perszeusz ((Akrisziosz) Danaé)) Perszész ((Képheusz) Andromeda (Kassziopeia)))

Ha nincs kifejezett rekurzió, akkor minden elemhez hozzárendelünk egy (vagy több) értéket, amely csupán a szerkezetet („vázat") határozza meg, azaz megadja az egyes elemek kapcsolatát a többivel. Most azt vizsgáljuk meg, hogy a lista és a bináris fa hogyan kapcsolódhat rekurzív algoritmusokhoz. Hogyan befolyásolják az algoritmus szerkezetét, milyenségét?

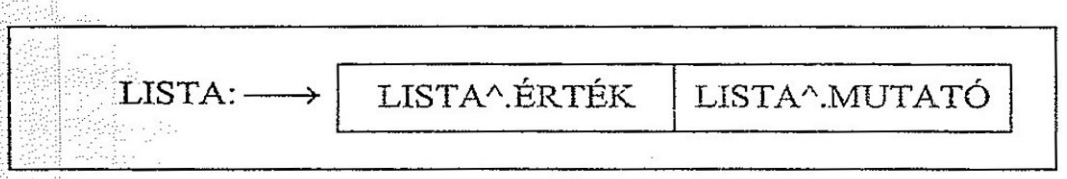
A listát is, a bináris fát is gyakran rendezési feladat megoldására használjuk. Most nézzük meg –egy valahogyan felépített adatszerkezet alapján– a rendezett adatok kiírásának rekurziómentes és rekurzív változatát, valamint a nem rekurzív adatszerkezetet használó változatokat is! Az algoritmusok megfogalmazásához kölcsönvesszük a Pascal nyelv néhány fogalmát, jelölését, mint pl. a mutató, illetve a rekord típust.

8.1. Rekurzív adatszerkezet, rekurzív algoritmus

A. Lista

Legyen LISTA egy mutató típusú változó, ami a LISTA egy elemére mutat! Ekkor *LISTA^* jelentse azt, *amire LISTA mutat.* Legyen *LISTA^.ÉRTÉK* az adott

elem *értéke*, *LISTA^.MUTATÓ* pedig a *következő elemre mutasson*! Vagyis az elemet, mint két mező (érték + mutató) rekordját képzeljük el.



14. ábra:
A lista elemeinek rekordszerkezete.

Így a listaszerkezet „bejárását" –a rekurzív definícióval összhangban– az alábbi rekurzív algoritmussal írhatjuk le. A *Bejár* eljárás meghívásakor paraméterként a gyökér mutatóját kell megadni.

```
Bejár(LISTA):
  Ha LISTA nem üres akkor Ki: LISTA^.ÉRTÉK
                          Bejár(LISTA^.MUTATÓ)
  Elágazás vége
Eljárás vége.
```

B. Bináris fa

FA legyen –hasonlóan az előbbihez– egy mutató típusú változó, amely egy fabeli csomópontra mutat! *FA^.ÉRTÉK* lesz az adott elem *értéke*, *FA^.BAL* a baloldali, *FA^.JOBB* pedig a jobboldali *részfára mutat*. A fa szerkezet bejárási módszerei közül mi most a *bal–közép–jobb* módszert valósítjuk meg rekurzívan.

```
BKJ_bejárás(FA):
  Ha FA nem üres akkor BKJ_bejárás(FA^.BAL)
                       Ki: FA^.ÉRTÉK
                       BKJ_bejárás(FA^.JOBB)
  Elágazás vége
Eljárás vége.
```

8.2. Rekurzív adatszerkezet, nemrekurzív algoritmus

A. Lista

A megoldás ebben az esetben egyszerű lesz, hiszen jobbrekurzióról van szó. Amint korábban láttuk, ilyenkor meglehetősen mechanikusan végezhető el a rekurzió kibontása, iterációvá alakítása.

```
Bejár(LISTA):
  Ciklus amíg LISTA nem üres
    Ki: LISTA^.ÉRTÉK
    LISTA:=LISTA^.MUTATÓ
  Ciklus vége
Eljárás vége.
```

B. Bináris fa

Itt egyszerű átírás nem létezik (mivel legalább egyszer minden nemterminális[1] elemhez vissza kell térni a még be nem járt részfához vezető út megtalálása érdekében), így csak verem segítségével adható meg a nemrekurzív változat. A verem játsza Ariadné fonalának szerepét. Minden egyes lépés mellett, amelyet a fa ágán teszünk előre, gombolyítunk egyet a fonalon: megjegyezzük a visszatérés legutolsó állomásának helyét. A visszalépésnél a felgombolyítás e hely címének elővételét jelenti. Ezen kívül szükség lesz egy változóra (VÉGE), amelynek segítségével megállapíthatjuk, hogy bejártuk-e már a teljes fát. (Az algoritmusban: az *üres fa* azt jelenti, hogy terminálisról „leléptünk a semmibe".)

```
BKJ_bejárás(FA):
  VÉGE:=Hamis; Verembe(üres fa)
  Ciklus amíg nem VÉGE
    Ciklus amíg FA nem üres
      Verembe(FA)  [előre a legbaloldalibb]
      FA:=FA^.BAL  [ágon a erminálishoz]
    Ciklus vége
    Veremből(FA)
    Ha FA nem üres akkor  Ki: FA^.ÉRTÉK [középső elem]
                          FA:=FA^.JOBB  [irány jobbra]
                  különben VÉGE:=IGAZ
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

8.3. Nemrekurzív adatszerkezet, rekurzív algoritmus

Ebben az esetben az adatokat és a hozzájuk tartozó mutató értékeket egymástól elszakítva külön-külön *egy-egy vektorban* (vagyis nemrekurzív szerkezetben) helyezzük el. Az egyes eljárásokat a lista logikailag első elemének, illetve a fa csúcsának vektorbeli indexével kell hívni.

[1] Nemterminálisnak hívják a fa azon csomópontjait, amelyekből vezet ki él.

A. Lista

Vegyünk fel két vektort! ÉRTÉK tartalmazza a kiírandó elemeket, MUTATÓ pedig a hozzájuk tartozó mutatókat! MUTATÓ(I)=0 álljon az utolsó elemnél! Ehhez a nemrekurzívan „tárolt" listához is megadhatunk rekurzív bejárási algoritmust. A megadás az első rekurzív algoritmusból úgy származik, hogy az ott szereplő mutatók értékei (= címek) helyett, most *vektorbeli indexek* szerepelnek.

```
Bejár(I):
  Ha I≠0 akkor Ki: ÉRTÉK(I)
               Bejár(MUTATÓ(I))
  Elágazás vége
Eljárás vége.
```

B. Bináris fa

Most három vektort kell felvennünk (a korábbi három mezős rekord mintájára)! ÉRTÉK tartalmazza a kiírandó elemeket, BAL, JOBB pedig a tőlük balra, illetve jobbra levő elem sorszámát! Ha értékük 0, az jelentse –most is– azt, hogy a fának abban az irányban vége van! A megfelelő rekurzív bejárás most is nagy hasonlatosságokat mutat a korábbi rekurzív algoritmussal.

```
BKJ_bejárás(I):
  Ha I≠0 akkor BKJ_bejárás(BAL(I))
              Ki: ÉRTÉK(I)
              BKJ_bejárás(JOBB(I))
  Elágazás vége
Eljárás vége.
```

8.4. Nemrekurzív adatszerkezet, nemrekurzív algoritmus

Ebben a változatban meg kell adni, hogy hol található az adott adatszerkezet logikailag legelső adata. Több újdonságot már nem tartalmaznak az algoritmusok, könnyen megkaphatjuk a 7.2. részben taglaltakból, a rekord mezők–vektorok, illetve mutatóérték–vektorindex megfeleltetéssel.

A. Lista

FEJ mutasson a sorrendben első elemre!

```
Bejár(FEJ):
  I:=FEJ
  Ciklus amíg I≠0
    Ki: ÉRTÉK(I)
    I:=MUTATÓ(I)
  Ciklus vége
Eljárás vége.
```

B. Bináris fa

CSÚCS mutasson a fa csúcsára!

```
BKJ_bejárás(CSÚCS):
  I:=CSÚCS; VÉGE:=Hamis; Verembe(0)
  Ciklus amíg nem VÉGE
    Ciklus amíg I≠0
      Verembe(I); I:=BAL(I)
    Ciklus vége
    Veremből(I)
    Ha I≠0 akkor  Ki: ÉRTÉK(I)
                  I:=JOBB(I)
         különben VÉGE:=IGAZ
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

Mit tehetünk akkor, ha nem az elemek kiírása a feladat, hanem valamilyen más tevékenységet kell velük elvégeznünk? Jó lenne, ha nem kellene ebben az esetben újra megírni valamelyik fenti eljárást (figyelembe véve az új transzformációt elvégző algoritmusrészt).

Sok programozási nyelv lehetőséget biztosít arra, hogy egy eljárásnak *eljárás-paramétert* adjunk, s így például a listabejáró eljárást paraméterezhetjük egy feldolgozó eljárással:

```
Bejár(LISTA,Feldolgoz):
  Ha LISTA nem üres akkor Feldolgoz(LISTA^.ÉRTÉK)
                          Bejár(LISTA^.MUTATÓ,Feldolgoz)
  Elágazás vége
Eljárás vége.
```

Érdemes összevetni a megoldásokat bonyolultsági szempontból. Tapasztalható, hogy minél többet engedünk a rekurzív megoldásból, annál bonyolultabbá válnak az algoritmusok. Ez a bonyolódás a jobbrekurziót tartalmazó listabejárásnál kicsi volt, a fabejárásnál viszont lényeges! Az egyszerűség oka –mint már tudjuk– az, hogy az adott programnyelvi környezet maga gondoskodik a rekurzió adminisztrálásáról, ára programunknak néha meglepő méretre tágulása mind „térben" (=tárban), mind időben.

Végezetül egy nagyon távolról idekapcsolódó témával foglalkozunk, amely listákhoz, bináris fákhoz (általában dinamikus adatszerkezetekhez) kapcsolható.

8.5. Dinamikus memóriakezelés

Dinamikus adatstruktúrák esetén az optimális memóriafelhasználás úgy érhetjük el, ha mindig csak annyi helyet foglalunk, amennyi az adatszerkezet pillanat-

nyi tárolásához szükséges. Ez azt jelenti, hogy ha az adatszerkezetben egy új értéket akarunk elhelyezni, akkor nekünk kell gondoskodni a szükséges *memória lefoglalásáról* (amelyet „normál" esetben az értelmező- vagy fordítóprogram megoldd). Ha pedig egy elemre már nincs szükség, akkor a mi dolgunk a számára kijelölt memóriatartomány *felszabadítása*. Erre a célra a következő két utasítás áll rendelkezésünkre:

```
Lefoglal(mutató)
Felszabadít(mutató)
```

Ezen eljárások paramétere egy-egy mutató típusú változó, amelyek meghatározzák az adott memóriaterület címét és milyenségét (=méretét).

Ezeket az eljárásokat használva készítjük el a rendezési feladat megoldására szolgáló bináris fát, majd pedig a fa lebontását.

```
Beépít(FA,ELEM):
  Ha FA üres akkor  Lefoglal(FA); FA^.ÉRTÉK:=ELEM
                    FA^.BAL:=üres ; FA^.JOBB:=üres
           különben Ha FA^.ÉRTÉK≤ELEM
                       akkor  Beépít(FA^.BAL,ELEM)
                     különben Beépít(FA^.JOBB,ELEM)
  Elágazások vége
Eljárás vége.
```

A fa lebontásához a fa *bal–jobb–közép* stratégiájú bejárására van szükség. Mivel a fa elemei csak a gyökérelemen keresztül érhetők el, ezért a gyökér megszüntetése után sem a bal- sem a jobboldal nem lenne elérhető, ami lehetetlenné tenné ezek azonosítását, kapcsolat híján pusztán a helyet foglalná a tárban.

```
Lebont(FA):
  Ha FA nem üres akkor Lebont(FA^.BAL)
                       Lebont(FA^.JOBB)
                       Felszabadít(FA)
  Elágazás vége
Eljárás vége.
```

A fenti eljárásokat használhatjuk például a következő programban, ahol a fát felépítjük, a rendezett adatokat kiírjuk, majd a fát lebontjuk.

```
Rendezés bináris fával:
  Be: N; FA:=üres
  Ciklus I=1-től N-ig
    Be: X
    Beépít(FA,X)
  Ciklus vége
  BKJ_bejárás(FA)
  Lebont(FA)
Program vége.
```

Irodalomjegyzék

A rekurzióval közvetlenül nagyon kevés magyar nyelvű könyv foglalkozik. Legtöbbször valamilyen más téma kapcsán merül fel szükségessége. Így az irodalomjegyzékben is olyan könyveket sorolunk fel, amelyek csak nagyon kis részükben foglalkoznak ezzel a témával.

1. A. Aho-J. Hopcroft-J. Ullman: Számítógép algoritmusok tervezése és analízise
   Műszaki Könyvkiadó, 1982.

2. G. Ausiello: Algoritmusok és rekurzív függvények bonyolultságelmélete
   Műszaki Könyvkiadó, 1984.

3. Z. Manna: Programozáselmélet
   Műszaki Könyvkiadó, 1981.

4. Szlávi P.-Zsakó L.: Módszeres programozás
   Műszaki Könyvkiadó, 1986.

5. B.A. Trahtenbrot: Algoritmusok és absztrakt automaták
   Műszaki Könyvkiadó, 1978.

6. N. Wirth: Algoritmusok + Adatstrukturák = Programok
   Műszaki Könyvkiadó, 1982.