SZLÁVI PÉTER * ZSAKÓ LÁSZLÓ

MÓDSZERES PROGRAMOZÁS

HIBAJEGYZÉK

Szlávi-Zsakó: Módszeres programozás

A 82.–84. oldalon felsorolt programtranszformációkhoz a következő megjegyzések tartoznak.

1.-hez (ha U1 nem változtatja meg értékét.)

3.-hoz (Abban az esetben, ha F2 ellenőrzése bonyolult és F1 elég gyakran hamis értékű.)

4.-hez (Abban az esetben, ha F1 és F2 teljes feltételrendszer, azaz minden esetben közülük pontosan az egyik teljesül.)

5.-höz (Abban az esetben, ha U1 végrehajtása nem befolyásolja F értékét.)

6.-hoz (Ha U1 és U2 független egymástól és U3-tól, továbbá CF-re csak U3-nak van hatása – azaz pl. számlálásos ciklusról van szó.)

8.-hoz (Abban az esetben, ha F értéke a ciklus végrehajtása során nem változhat.)

9.-hez (Abban az esetben ha U1 végrehajtása független a ciklustól, és saját maga korábbi végrehajtásától is)

A 77. oldal 2. algoritmusa $\sqrt{N}$-ig fut.

A 89. oldal 2.(3) szintű programjában a második ciklus N+D1–1-ig tart.

SZLÁVI PÉTER–ZSAKÓ LÁSZLÓ

MÓDSZERES PROGRAMOZÁS

MŰSZAKI KÖNYVKIADÓ
BUDAPEST, 1986

Lektorálta:

LEKTORÁLTA:
MEGYERI JÓZSEF
OKL. VILLAMOSMÉRNÖK

© SZLÁVI PÉTER–ZSAKÓ LÁSZLÓ
BUDAPEST, 1986

ETO: 519.683
ISBN: 963 10 6820X

FELELŐS SZERKESZTŐ:
VOTISKY ZSUZSA

KIADJA A MŰSZAKI KÖNYVKIADÓ
FELELŐS KIADÓ: FISCHER HERBERT IGAZGATÓ

86-75 SYLVESTER JÁNOS NYOMDA, SZOMBATHELY
FELELŐS VEZETŐ: HANUSZEK BÉLA IGAZGATÓ

MŰSZAKI VEZETŐ: KŐRIZS KÁROLY
MŰSZAKI SZERKESZTŐ: LOVÁSZ GYÖRGY
A KÖNYV ÁBRÁIT RAJZOLTA: MELEGH JUDIT, ORLAY ERNŐ
A KÖNYV FORMÁTUMA: B 5
ÍVTERJEDELME: 10,15 (A5)
ÁBRÁK SZÁMA: 16
PAPÍR MINŐSÉGE: 80 G OFSZET
BETŰCSALÁD ÉS -MÉRET: PRESS ROMAN 10/11 PONTRA
AZONOSSÁGI SZÁM: 61370
MŰ: 3876–i–8688
A KÉZIRAT LEZÁRVA: 1985. OKTÓBER
KÉSZÜLT AZ MSZ 5601 ÉS 5602 SZERINT
A SZEDÉS A MŰSZAKI KÖNYVKIADÓ COMPOSER RÉSZLEGÉBEN KÉSZÜLT

TARTALOM

Bevezetés 7
Módszerek nélkül 9
Problémák a programkészítés körül 11
A programozási elvek 13
Első példa a módszeres programozásra (a kitűzött feladat módszeres megoldása) 22
Második példa a módszeres programozásra (hasonló problémák, hasonló programok) 31
Programozási tételek 33
Harmadik példa a módszeres programozásra (a tételek alkalmazhatósága) 49
Tesztelés 53
Példa a tesztadatok választására 59
Programhiba-keresés 61
Példák a hibakeresésre 64
Helyességbizonyítás 68
Negyedik példa a módszeres programozásra (programok összehasonlíthatóságáról) 73
Programok hatékonysága 76
Programcsaládok 86
Példa a feladatmeghatározás bonyodalmaira 91
Néhány szó a dokumentációról 94
Utószó 97
1. Függelék 98
2. Függelék 101
3. Függelék 108

BEVEZETÉS

Könyvünkkel a számítástechnika nem hivatásszerű alkalmazóinak szeretnénk segítséget nyújtani. Körükben is el kívánjuk terjeszteni azokat a *programozási elveket*, amelyek a számítástechnika szakemberei előtt már régóta ismertek és használtak. Egyrészt azokhoz szólunk, akiknek némi gyakorlatuk van már a programozásban. Számukra minden bizonnyal ismerősek lesznek a problémák. Lehet, hogy a kezdő programozók nem kellő átéléssel olvassák majd az első, a *Problémák a programkészítés körül* c. fejezetet, de higgyék el nekünk, hogy az elvek be nem tartása kétségkívül a leírt bonyodalmakhoz vezet.

Mondanivalónkat az elvek kialakulásához vezető *problémákkal* kezdjük, majd ennek orvoslására egy nyelvet vezetünk be, amelynek elsajátítása nem jelent különösebb gondot a nem számítástechnikai szakember számára sem. Mint kiderül, ennek mintájára bárki kialakíthatja a saját nyelvjárását, amit persze úgy célszerű megalkotni, hogy másoknak is könnyen érthető legyen.

A problémák után néhány feltétlenül megszívlelendő *programozási* (helyesebben programírási) *elvet* sorolunk fel. Már itt sietünk felhívni a figyelmet arra, hogy az elvek betartása nem lesz könnyű. Ez főleg azokra vonatkozik, akik már úgymond *rutinból* programoznak. Gondoljanak keserű tapasztalataikra, amikor kísértést éreznek letérni az elvek útjáról!

Az elvek ismertetése után bemutatjuk a *módszeres programozás gyakorlatát*. Ekkor válnak természetessé és magától értetődővé az elvek. Példáinkat kellően összetettnek választottuk, hogy az elvek használandóságára fény derüljön. Egy *szelíd* program születésének lehetnek szemtanúi az első példában. A másodikban megtanulhatják, hogy miért fontos (s hogyan lehet) új feladatok megoldása során a régiekkel hasonlóságot keresni. Majd egy későbbi példában többféle megoldást vizsgálva eldönthető lesz, hogy melyik megoldás a jobb, s ez milyen értelemben mondható róla. A *hatékonyság dimenzióit* kutatjuk: a helyfoglalást, a végrehajtási időt, a bonyolultságot.

Közben olyan szabályokkal is találkozunk, amelyeket tapasztalt programozók fedeztek fel. Ezeket gyakoriságuknál és általánosságuknál fogva helyesebb lenne programozási típusalgoritmusoknak, vagy – ahogy a szakirodalomban hivatkoznak rájuk – *programozási tételeknek* nevezni. Nem garantálják ugyan a programozás egzakt, matematikai felépítését, de a felmerülő feladatok igen nagy részére alkalmazhatók.

A programozás gyakorlatáról ennyivel még nem lenne teljes a kép. Tegyük fel, hogy megszületett a mű: a program. A következő feladat: meg kell győződni arról, hogy vajon tényleg jó-e? Ha nem, akkor hogyan lehet kijavítani úgy, hogy ami eddig jó volt, az jó is maradjon. Itt fogjuk tisztázni, hogy mik a program helyességének formális követelményei, s ízelítőt adunk a *helyességbizonyítás* témaköréből is.

Sokszor hasznos, ha egy feladat megoldása előtt a feladattal rokon, s már megoldott problémák között kutatva bukkanunk rá a megoldásra. Így mintegy *(program-) családi kapcsolataink* juttatnak el a cél közvetlen közelébe. E családi kapcsolatok feltárásában,

valamint a már *önálló* lépések megtételében nyújt segítséget a *Programcsaládok* c. fejezet. Ugyanitt más beállításban is felvetődik a *rokonság* figyelembevétele. Sokszor ui. a feladatra nem egy, hanem épp egy egész programcsalád ad kézenfekvő megoldást.

Végül a feladatmegoldás folyamatában oly mostohán kezelt, és e folyamat alapjaként tekintendő lépést, a *feladatmeghatározást* (vagy *specifikálást*) tárgyaljuk. Bár ez a programkészítés egyik első tevékenysége, mégis a végére hagytuk. Indokunk: kezdők számára kevéssé tudjuk a szükségességét korábban hangsúlyozni, és az amúgy is nehéz első fejezetek mellé nem akartunk még egy mély meggondolást igénylő részt helyezni.

A függelékekben egy javasolt *nyelvet* írunk le. Megadjuk, hogy mit s hogyan, azaz a nyelvtant (szintaxist) és a jelentést (szemantikát). Ezután összefoglaljuk, amit a BASIC-ről tudni kell: a kész programot milyen, ún. *kódolási szabályokkal* lehet átültetni a BASIC nyelvre.

Befejezésül még két gondolatot kell felvetnünk!

Könyvünk hangvétele némi előzetes magyarázatra szorul. Úgy gondoltuk, hogy a pontos, definíciókra épített tárgyalásmód helyett eredményesebb, ha az elvek kialakulása felől, szabadabb stílussal közelítjük meg a programozás problematikáját. Így sokszor használunk olyan fogalmakat, amelyek félrevezetően túl tudományosnak tűnhetnek, s mégsem definiálunk. Ilyen pl. a *frontális módszer,* a *módszeres program* stb. Ezeket a szó köznapi értelmezése szerint kell érteni, használata közben a hangulatára célszerű gondolni. Például a frontális módszer: a program teljes *arcvonalán* támadó módszer, amely a feladat minden egyes pontján egyszerre szeretne sikert aratni anélkül, hogy alkalmazkodna a feladat sajátos logikájához. Világos, hogy nagyobb lélegzetű probléma esetén ez a megoldási stratégia már nem lehet sikeres.

A programírás folyamatában feltételezzük a közvetlen ember–gép kapcsolatot. Mondanivalónk elsősorban ilyen kapcsolatot biztosító számítógépekre és nyelvekre vonatkozik. Így a személyi számítógépek mindegyikére. Ezeknek ma még kétségkívül legelterjedtebb programnyelve a BASIC. Ezért mi is e nyelv Magyarországon leggyakrabban előforduló *nyelvjárásaira* szorítkoztunk a kódolási szabályok megadásánál (ABC80, HT–1080Z, Commodore 64, ZX Spectrum, ZX–81). Nem titkolt szándékunk ezzel mintát adni más nyelvek kódolási szabályainak összeállításához.

Végül köszönetet mondunk mindazon munkatársainknak az Eötvös Loránd Tudományegyetem Számítástechnikai Tanszékén, akik munkájukkal sok segítséget adtak könyvünk elkészítéséhez, valamint Megyeri József lektornak lelkiismeretes munkájáért.

MÓDSZEREK NÉLKÜL

Nagyon nehéz feladat előtt állunk. Meg kellene mutatnunk, hogy milyen, és hogyan készül a rossz program. A rossz programoknak azonban az az egyik sajátossága, hogy nem látszik rajtuk a készítés menete. Milyen a rossz program? A listáján semmi feltűnőt, a vadhajtásoknak semmi látható jelét nem tapasztaljuk. Semmi stílusról nem árukodik, kusza, szövevényes. Mondanivalóját rettentő nehéz kihámozni a szövegéből. A most következő feladat a könyv néhány fejezetét végigkíséri, így érdemes alaposan meggondolni.

Feladat: Adott egy csillagtérkép, ahol a csillagokat a koordinátáikkal adjuk meg (háromdimenziós térkép). Csillagsűrűsödéseknek nevezzük azokat a lehető legnagyobb csoportokat, amelyekben legalább S0 darab csillag van és minden ott levő csillagra igaz, hogy el lehet jutni tőle a csoport bármely tagjához olyan úton, amely a csoport tagjain át vezet és a lépések hossza nem nagyobb S1 fényévnél (elég sűrűn vannak a csillagok). Határozzuk meg a térképen a csillagsűrűsödések számát!

1. ábra

Felhívjuk az Olvasó figyelmét, hogy mielőtt folytatná az olvasást, győződjön meg arról, hogy az ábra alapján megértette-e a feladatot, s csak ezután lapozzon tovább! Az 1. ábra alapján N = 11, S0 = 3, S1 = ____, a sűrűsödések száma = 2. A megértésben segítséget adhat a könyv vége felé szereplő *Példa a feladatmeghatározás bonyodalmaira* c. fejezet is.

Nézzük tehát a feladatot megoldó *vadon nőtt* programot! (A továbbiakban, ahol a számítógép típusát nem jelezzük, ott a HT–1080Z nyelvjárását használjuk)

```
10        DIM X(100),Y(100),Z(100)
20        INPUT N,S0,S1
30        S=0
40        FOR I=1 TO N
50        INPUT X(I),Y(I),Z(I)
60        NEXT I
```

```
70          FOR I=1 TO N
80          IF X(I)=0 THEN 200
90          X(I)=-X(I)
100         A=0
110         FOR J=1 TO N
120         IF J=I OR X(J)<=0 THEN 140
130         IF SQR((-X(I)-X(J))**2+(Y(I)-Y(J))**2+
(Z(I)-Z(J))**2) <= S1 THEN X(J)=-X(J) : A=A+1
140         NEXT J
150         X(K)=0
160         K=K+1
170         IF K>N THEN 190
180         IF X(K)>=0 THEN K=K+1 : GOTO 170 ELSE 110
190         IF A>S0 THEN S=S+1
200         NEXT I
210         PRINT S
220         STOP
```

Minden tiszteletünk azé az Olvasónké, aki:
- megértette a feladat megoldását,
- talált benne rejtett feltételezéseket,
- megtalálta benne a hibákat,
- ki tudta javítani a hibákat,
- és végül használni tudta a programot.

Ne keseredjen el, ha netán nem sikerült mindezt megtennie, hiszen ez talán a szerzőknek sem sikerülne. A könyvet továbbolvasva meg fogja találni az előbbi program szelídített változatát, és megismerheti a megoldásokat programkészítési problémáira.

PROBLÉMÁK A PROGRAMKÉSZÍTÉS KÖRÜL

A programozók gyakori hibája, hogy a feladatot kézhez kapva, rögtön nekilátnak a program megírásának, pontosabban a program kódolásának. Vegyük szemügyre a feladat elleni ilyen *frontális támadás* hibáit!

A feladatmegoldás efféle megközelítése igen gyakran kudarcra ítélt: *zavaros visszalépésekkel* tagolt. A visszalépések oka az előrelátás hiánya. Ez azt jelenti, hogy a problémát csak fellépésekor érzékeli, akkor, amikor már kitérni nem tud, csak visszakozni!

Nem a visszalépés ténye rossz, hiszen nem lehet belelátni a feladatba első pillantásra. Kicsit bele kell élnünk magunkat a feladat sajátos problémavilágába ahhoz, hogy biztonságosan tudjunk tájékozódni a megoldás kidolgozása során. Gyanút csak a gyakori és zavaros visszalépések kell, hogy ébresszenek.

A visszalépések megtagadása nem vezet célra. Az újabb és újabb *csalafintaságok*, a trükkök hatására csak még inkább belebonyolódunk a programba, s később még fájóbb az újrakezdés.

De tételezzük fel, hogy kudarcot nem ismerve, megküzdünk a feladattal: és elkészül a program. Azt hihetnénk, hogy megpróbáltatásainknak ezzel vége. Koránt sincs így! További próbatételek várnak ránk, s programunkra.

Használni kell ui., de a program nem működik! Pedig mennyi leleményt és fortélyt alkalmaztunk! Talán éppen ezek a szunnyadó, rosszkor ébredező ötletek okozzák tragédiánkat?! A lecke fel van adva: ki kell javítani a programot! Ehhez először meg kell találni a hibát, ami nem is egyszerű dolog egy ilyen pokolian ravasz *szerkezet* esetén. Kérdés, hogy beszélhetünk-e egyáltalán szerkezetről, van-e a programnak szerkezete? S ha megtaláltuk a hibákat, milyen újabb varázslat szükséges a program működtetéséhez?

Készüljünk fel a legjobbra! A program elkészült, sőt – láss csodát! – működik is. Telik-múlik az idő, hasonló problémával találkozunk. Elővesszük *jól megírt,* kész programunkat abban a hiszemben, hogy használni tudjuk. Azonban hamar kiderül, hogy a hasonlatosság, bár valós, de nem elegendő ahhoz, hogy a program magától megbirkózzék a problémával. Például egy kicsit más formában kéri az adatokat vagy/és egy kicsit másként értelmezi, mint a mostani feladat igényelné. Nyilván a programot kell a feladathoz idomítani. Csalóka ábránd, hogy nagy nehezen meglelve a kritikus, kicserélendő részt, átalakítás után működő programot kapunk. Szomorúan ugyan, de kénytelenek leszünk tudomásul venni, hogy egyetlen kiút maradt: újra írni a programot!

Hasonló megrázkódtatás, ha programunk használatára nem nekünk, hanem barátunknak, munkatársunknak vagy – elképzelni is szörnyű – főnökünknek van szüksége. De ezen a ponton hagyjuk is abba a programírás fejetlenségéből származó gondok ecsetelését.

Mindezek a frontális módszer hibáiból származó problémák. A módszeresen írt programokkal is lehetnek gondok. Ezek csak technikai problémák, de egyáltalán nem jelentéktelenek.

Lássuk tehát, jól megírt programunk használata milyen követelményeket támaszt írójával szemben!

A programot elindítva még a legvakmerőbb felhasználónak is inába száll a bátorsága, ha a program első ténykedéseként egy kérdőjel jelenik meg a képernyőn. Legyen bármily fantáziadús is az illető, koránt sem biztos, hogy tudja, mit s hogyan feleljen a kérdésre. Tételezzük fel, hogy a kérdőjel ennek jelzése céljából került ki a képernyőre. Persze feltehető, hogy van elképzelése arról, mi is lehet a kérdés. Bár, ha sejti, hogy a programnak milyen paraméterekre van szüksége, ezek megadási sorrendjében akkor sem lehet biztos. Így aztán a felhasználó vagy lemond a program segítségéről, vagy sietve megkeresi a program készítőjét. A bátrabbak, a kihívást elfogadva, automatizált rejtvénynek tekintik az adódott problémát, s kísérleti alapon próbálják megfejteni, kitalálni a várt választ.

A felhasználó reménytelen eset! E kijelentéssel arra a kellemetlen kötelességre utalunk, hogy a felhasználó minden ravaszkodására, jó vagy rosszindulatú tévedésére fel kell készülni. Kerülni kell annak eshetőségét, hogy nem természetes módon – általunk nem betervezetten – álljon le a program amiatt, hogy a felhasználó nem azt az adatot, vagy nem akkor adta, amit, ill. amikor kellett volna. Előfordulhat az is, hogy nem abban a sorrendben, esetleg nem egészen abban a formában írja be a választ, ahogyan a program várja.

A borzasztó programok kategóriájába tartozik az a program is, amely eltitkolja használója elől, hogy éppen mit csinál. Gondoljunk csak arra a bizonytalan érzésre, ami akkor keríti hatalmába az embert, amikor a program hosszú időn át semmi jelzést nem ad működéséről, s nem tudjuk, hogy valami hiba miatt áll, vagy pedig helyesen működik.

Ne is beszéljünk azokról a gyötrelmekről, amikben akkor van részünk, ha más jól-rosszul megírt, de az előbbiekben ecsetelt, barátságtalan programját kényszerülünk használni.

Foglaljuk össze, milyen, a programírás módszeréből fakadó, ill. milyen, a program használhatóságát érintő problémákról beszéltünk!

1. A feladat elleni frontális támadás módszere

a) Visszalépésekkel sűrűn tagolt. Ez hasonló probléma, mint a dzsungel ösvényein saját lépteink nyomát keresni hazafelé.

b) Visszakozás helyett további útkeresés a sűrűben. A csalafintaság dzsungelkésével vágott úton nem gondoskodunk a hazatalálást lehetővé tevő útjelzőkről.

c) A hibakeresés megfelelő tájékozódási pontok nélkül nehézkes, bizonytalan vállalkozás.

d) A hibakeresésnek nincs vége. Hiba hibát szül: a sok csalafintaság, megfontolatlan betoldás a programot megmenti ugyan, de más, a legváratlanabb helyeken bukkannak fel új, eddig nem tapasztalt hibajelenségek.

e) A hibajavításnak nincs vége. Pontosabban nem tudjuk, hogy mikor van vége, mert a hibajelenségek hiánya még nem jelent hibátlanságot.

f) A program módosíthatatlan. Így két lehetőségünk van: vagy a feladatot idomítjuk a meglevő programhoz, vagy újraírjuk a programot, vagyis a programot kérészéletűvé tesszük. (Hosszú fejlődési szakasz, és röpke termékeny időszak.)

g) A program használhatatlan másnak, mert képtelenség, hogy az íróját kivéve bárki a program lelkivilágába beleélhesse magát. Csak idő kérdése, hogy az írójának is gondot jelentsen a használata.

2. A módszeresen írt program kényelmetlen vonásai

a) A felhasználónak fogalma sincs a program használatának mikéntjéről, ha maga a program nem igazítja el ebben.

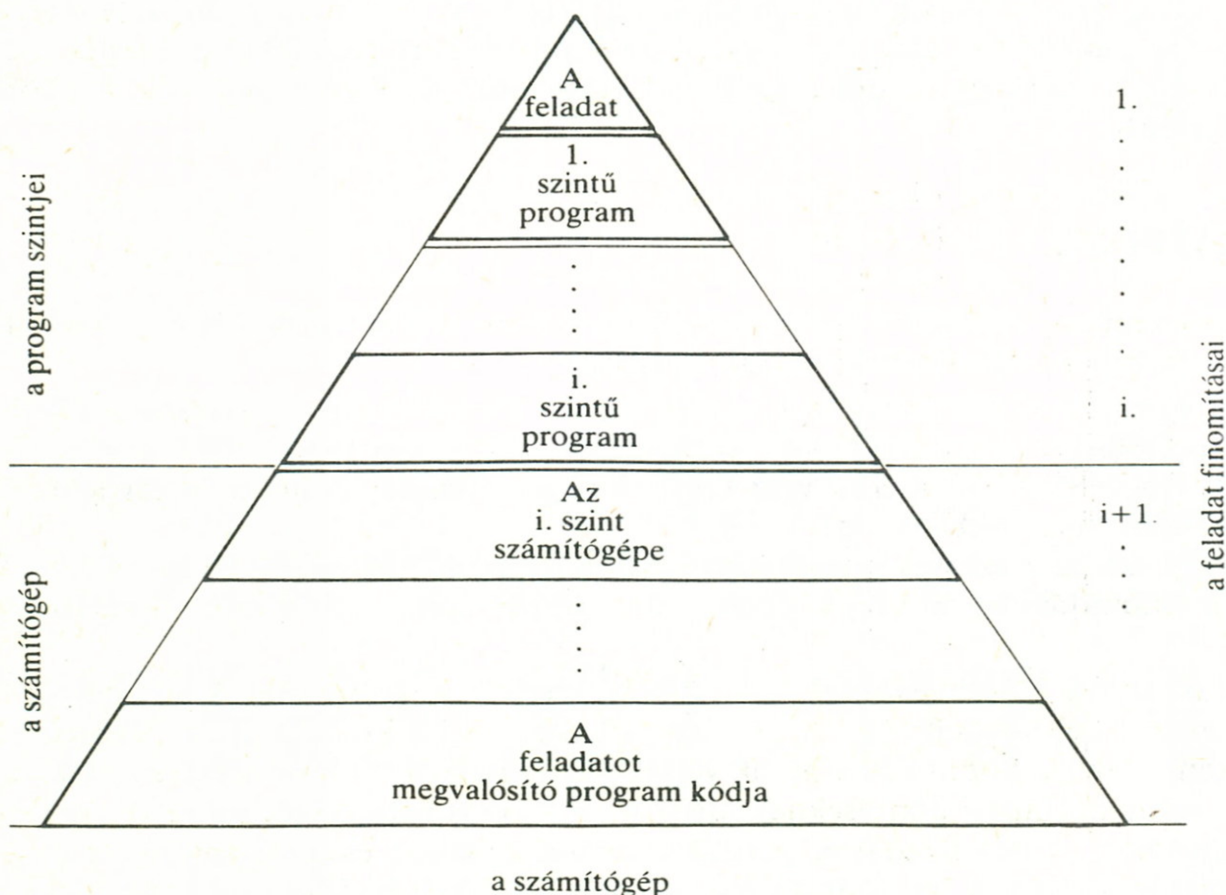
b) A felhasználó még ha tudja is, hogy mit és hogyan tegyen, néha akkor is másként tesz, ami egy nem eléggé felkészült program számára tragikus lehet.

c) A felhasználó pánikba eshet, ha nem tudja nyomon követni a program futását, ami kihathat az eredményre is.

A PROGRAMOZÁSI ELVEK

Egyik legfontosabb, sokféleképpen alkalmazható elvünk az ókori latin kultúrából ránk maradt *Oszd meg és uralkodj* elve alapján fogalmazható meg: *Oszd részekre*, majd a részek független megoldásával az egész feladatot könnyebben oldhatod meg. Így programodat könnyen kézben tarthatod, vagyis *uralkodhatsz* felette.

Ezt a stratégiai elvet tartjuk szem előtt akkor, amikor gondolkodásmódunkat helyes mederbe igyekszünk terelni. *Lépésenkénti finomításnak* nevezik ezt az elvet a feladatmegoldás filozófiájában. A feladat megoldását először átfogóan végezzük el, nem törődve a részletekkel, amelyekről érdemben amúgy sem tudnánk helyesen dönteni az adott pillanatban sok, még pontosan előre nem látható probléma miatt. Tehát a feladatot néhány (nem túl sok!) részfeladatra bontjuk. Úgy is mondhatnánk: a feladatot megoldjuk a legfelső szinten. Ha volna olyan gép, amelyen léteznének azok az utasítások, amiket mi részfeladatokként megadunk, akkor máris futtatható lenne a program. Ezt az eljárást fogjuk követni az egyes részfeladatok megoldásakor is (*a részek feletti uralkodás* érdekében), mindaddig, amíg olyan utasítások szintjéig nem érünk, amelyeket gépünk (kódolás után) már végre tud hajtani. (*Piramis elv* – l. a 2. ábrát.)



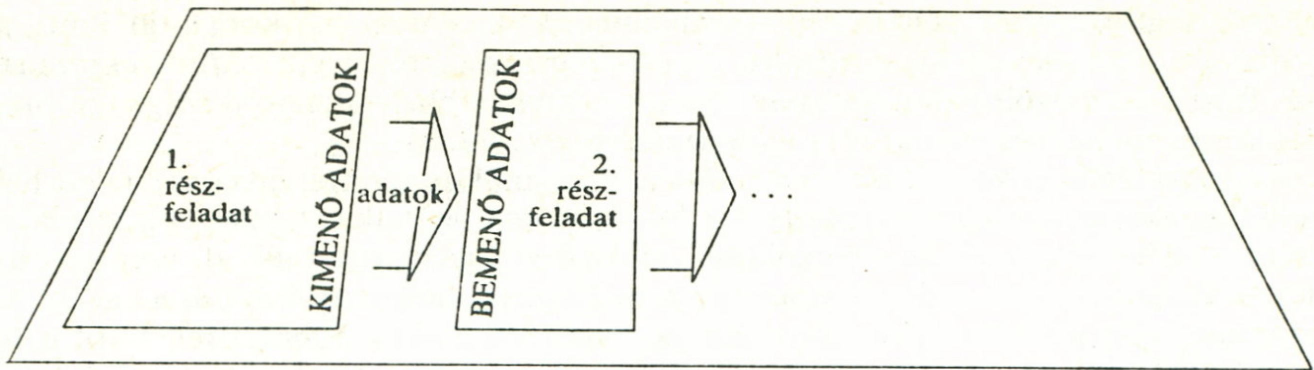
2. ábra

Mindez így elmondva rettentő egyszerűnek tűnik, holott az ilyen fegyelmezett gondolkodás nem könnyű.

A részfeladatokra bontás a következőket foglalja magában: pontosan ki kell jelölni, hogy az adott részművelet milyen adatokat kezel, milyeneket állít elő, és ezeket miként kell egymáshoz rendelni; vagyis ha *ez* az adat jött a részművelethez, akkor *azt* az adatot kell kapjuk eredményként. A *hogyanról* most nem elmélkedünk.

Nyilvánvaló, hogy két azonos szinten definiált részfeladat között biztosítani kell a harmóniát úgy, hogy a végrehajtásban előbb következő az utána következő adatait szolgáltassa (3. ábra). A részfeladatokra bontáskor persze felmerülhetnek illesztési problémák.

A feladat finomításának egy szintje . . .



. . . a piramis egy szintjén

3. ábra

Az elmondottakból már körvonalazódik a *lépésenként* elnevezés értelme. Óvatosan, megfontoltan haladunk az ismeretlenbe; vigyázunk, hogy minden lépésnél megtartsuk uralmunkat az éppen megondandó részfeladat felett. A finomítás pedig a lépésenkénti megközelítés mikéntjére utal. A lépés során kirajzolódott elemibb, egymástól már jól elhatárolható, s ezért egymástól függetlenül kezelhető* tevékenységek még elemibbekre bontását jelenti.

TAKTIKAI ELVEK

Milyen taktikai elveket – vagy kissé szerényebben szólva, jó tanácsokat – adhatunk a lépésenkénti finomítás elvének megvalósításához?

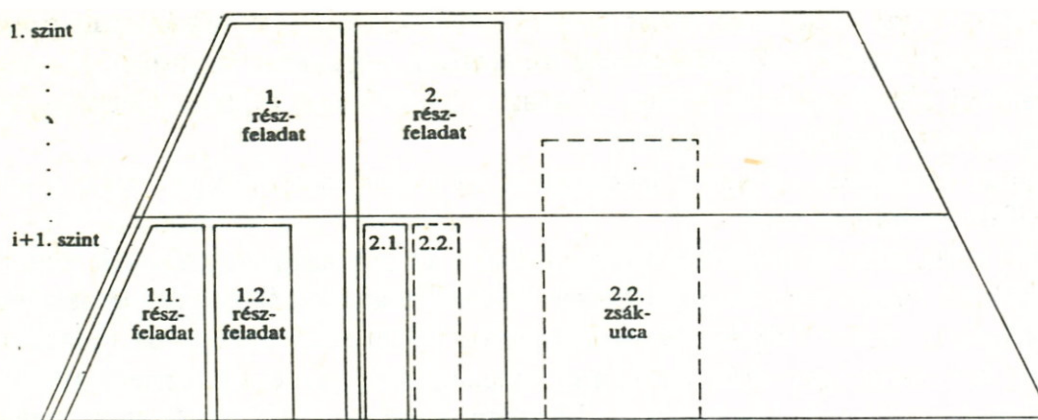
1. A *párhuzamos finomítás* elve, amely szerint a szint összes részfeladatára kell elvégezni a finomítást. Nem szabad előre sietni valamelyik könnyebbnek vélt ágon, mert előfordulhat, hogy munkánk kárba vész (egy esetleges visszalépés miatt, s kár lenne ilyen lélektani terhet nyakunkba venni).

Egy-egy szinthez szervesen hozzátartoznak a részfeladatok adatai is. Így világos, hogy az adatok finomítása során sem szaladhatunk előbbre, mint amit a szint eljárásai megkívánnak.

2. A *döntések elhalasztásának* elve, amely szerint egyszerre csak kevés dologról, de következetesen kell rendelkezni. A problémát kevés, de jól körülhatárolt részproblémára kell bontani. Ám óvakodni kell a másik véglettől is, hiszen a túl kevés részre bontás általában nem vezet optimális döntésekhez, mert nem azonos nehézségű, súlyú részproblémákat eredményez. A részfeladatok pontos körvonalazásához eleinte még nincs elegendő ismeret birtokunkban, s ha döntésünket később felül kell bíráljuk, még visszalépések is adódhatnak. Az a jó döntés, amely a későbbiek során a legkevésbé köti meg kezünket. Célszerű az adatok és eljárások finomításánál minél későbbre halasztani azokat a döntéseket, amelyek kihasználják a gép, ill. a programozási nyelv konkrét sajátosságait. Annál jobb a programunk, minél többet tartalmaz a feladat lényegi felépítéséből és minél kevesebbet a gép, ill. a nyelv kötöttségeiből.

3. *Vissza az ősökhöz* elv, amire akkor van szükségünk, amikor körültekintő megfontolásaink ellenére zsákutcába kerülünk. Ekkor vissza kell lépni az előző szinthez, és újra végig kell gondolni a részfeladatokra bontást, a keserű tapasztalatok figyelembevételével (4. ábra).

*Ezek a tevékenységek függetlenül kezelhetők, hiszen egymáshoz való viszonyukat az illesztéseik megadásakor már pontosan figyelembe vettük.



ha a 2.2-nél zsákutcába jutottunk, akkor újra kell gondolni a 2. részfeladatot, s ha ennek (az i. szinten adott) specifikációja változtatás nélkül nem tartható, akkor a teljes i. szint újra feldolgozandó!

4. ábra

Bármennyire is csábító lenne a zsákutcából a 2. részfeladat újra átgondolásával, kijutni, elkerülhetetlen minden részfeladatot újra megfontolni, hogy a beépített új elem hézagmentesen illeszkedjen.

4. A *nyílt rendszerű felépítés* elve, miszerint nemcsak a feladatra, hanem a feladatot is tartalmazó feladatkörre alkalmazható programot érdemes definiálni. Ezzel elkerülhetjük a későbbi kényszerű feladat- vagy programidomítási gondokat, azaz nem egy *nyárra* hozunk létre programot. (Szokás ezt még a *feladatáltalánosítás* elveként is emlegetni.) Ennek az elvnek kifejezett alkalmazásával találkozunk majd a programcsaládokról szóló fejezetben.

5. A *döntések el nem mulasztásának* elve: a ki nem mondott, de hallgatólagosan meghozott döntések rugalmatlanná és álnokká teszik a programot. Néha a *döntések kinyilvánításának elveként* is hivatkoznak rá. Szorosan összefügg azzal, hogy a fejlesztői dokumentációt a program tervezésével, írásával párhuzamosan kell készíteni. (Ebben úgy írjuk le a programot, hogy működését más is megérthesse, szükség esetén módosíthassa.)

6. Az *adatok elszigetelésének* elve azt mondja ki, hogy a programot biztos mederben csak úgy lehet tartani, ha az egyes programegységek a megtervezett illesztéseken, kapcsolatokon kívül egymással nem érintkezhetnek. Ezért tehát a programegységekhez tartozó adatokat ki kell jelölni és el kell szigetelni más programegységektől. Az adatok kijelölését legcélszerűbb a programegységben betöltött szerepük alapján csoportosítani: *közös* (pontosabban fogalmazva globális), ezen belül *bemeneti (input)* és *kimeneti (output)* adatok. A programegységhez és csak hozzá tartozó ún. *saját* (pontosabban fogalmazva *lokális*) adatokat, amelyektől – mint később látni fogjuk – a jó helykihasználás miatt hasznos elválasztani az ún. *munkaadatokat* (ezeket szokták a programozók *sajátnak* nevezni). Ez utóbbiak a programegység csak egyszeri működéséhez, időlegesen felhasznált segédadatok.*

TECHNOLÓGIAI ELVEK

A programírás már ecsetelt didaktikai elveitől a technológiai elvek vezetnek el bennünket a technikai elvek birodalmához.

1. Kevés, de egyértelmű szabályt kell kialakítani az algoritmusok leírására. Ez legyen számunkra kellően kényelmes gondolataink szárnyalásához, de a kényelem nem mehet a precizitás, az egyértelműség rovására! Ebben a legfontosabb algoritmikus szerkezeteknek helyet kell kapniuk.

Melyek is ezek a nevezetes szerkezetek?

*Programírás közben érdemes táblázatot készíteni. Ebben összegyűjtjük a megfelelő szintszámokhoz tartozó változóneveket, eljárásneveket és hatáskörüket. Erre később példát is mutatunk.

(A programozással még csak most ismerkedők számára az itt leírtak minden bizonnyal nehezen emészthetők és esetlegesnek hathatnak, egyelőre fogadják el magyarázkodás nélkül. A későbbi alkalmazások során – azaz a következő fejezetekben – ezek értelme, értelmezése, célszerűsége kiderül majd.)

Az adatokat *beolvasó* és *kiíró* utasítások az *ablak* szerepét játsszák a külvilág felől, ill. a program felhasználója felé.

A program változóinak értékkel való ellátását az *értékadó* utasítások végzik.

A feltételektől függő végrehajtást teszik lehetővé az ún. *feltételes* utasítások.

A számítógépre szánt feladatok mindegyike feltételezi bizonyos részfeladatok ismételt elvégzését. A számítógép erősségét, a gyorsaságot éppen ezek a mechanikus ismétlések használják ki a legjobban! Ezek megvalósítása a *ciklusutasítások* segítségével történik.

A program adott szintjén elemi utasításként felhasznált, meghatározott, de nem finomított részprogramok beépítését (az ún. eljáráshívást) is meg kell oldanunk nyelvünkben.

Természetesen a felhasznált és még hiányzó eljárások finomítása (másként szólva: az eljárás kifejtése) sem hiányozhat.

Nyelvünk részletes és összefoglaló leírása az 1. Függelékben található. (Egy hasonló, bár precízebb nyelv szerepel az [5] 3. fejezetében.)

E sajátos nyelvjárás, ill. leíró nyelv mellett szólnak a következő érvek. Az így készült program mechanikusan átírható bármelyik gépre, ill. nyelvre. Csupán a kódolási szabályokat kell megadni. A kódolási szabályok megadása persze nem mindig problémamentes, de ugyanezek a problémák felmerülnek a program újbóli, közvetlen megírásánál is! Az e módszerrel szemben gyakran felhozott érv, hogy nem vezet optimális programra, mert nem számol a nyelv, ill. a gép sajátosságaival, általában nem igaz, ui. a kódolási szabályok megalkotásánál ezeket figyelembe vehetjük. A program viszont e lokális optimalitáson túl rendelkezik a feladat logikáját mélyen figyelembe vevő algoritmikus optimalitással.

2. Az algoritmikus nyelv kialakításánál jó betartani a könnyű olvashatóság érdekében a *bekezdéses leírás* és az *összetett utasítások zárójelezése* elveket. Ahogy az író is főbb gondolatait, a történés főbb eseményeit fejezetekbe tömöríti, ahogy az egyes epizódok külön-külön bekezdéseket alkotnak a fejezeten belül, valahogy úgy kell algoritmikus gondolatainkat, az algoritmus főbb eseményeit, epizódjait is jól láthatóan elkülöníteni a programban. A program teljes levezetése (finomítása) után a program szerkezetének vissza kell tükröznie a szintekre tagozódást: egy szint elemi utasításai a bekezdések azonos szintjeit alkossák! Ezeket az elveket is szem előtt tartottuk nyelvünk megalkotásakor.

3. A *beszédes azonosítók* elve. A változóknak olyan nevet érdemes adni, ami utal arra, hogy mire használjuk. Ez kizárja az azonosítók keveredését, hiszen a név sugallja a funkciót, az algoritmusban betöltött szerepet. Nagy segítséget nyújt a kódoláskor is, pl. lehetővé teszi, hogy minimális számú változót rendeljünk az adatokhoz, hiszen a munkaváltozókhoz azonos neveket is rendelhetünk.

Az elvek további magyarázata helyett egy példával illusztráljuk az előbbieket. Legyen a feladat az, hogy a szövegben szereplő JUNIUS nevet minden előfordulásánál helyettesítsük a JULIUS névvel. Az algoritmus legfelsőbb szintje ilyesmi lehet:

```
Allj a SZöVEG elejére
Ciklus amig nincs vége a SZöVEGnek
   Ha  következö szava a SZöVEGnek "JUNIUS"
       akkor helyettesiteni "JULIUS"-sal
       különben helyben hagyni
   Elágazás vége.
Ciklus vége
```

Mi itt a szembetűnő? Valamit addig kell ismételten elvégezni, amíg a szöveg végére nem érünk. Ez a valami jól láthatóan elválik az őt mintegy keretbe foglaló ciklusutasítás *kezdő* és *befejező* zárójelétől:

```
Ciklus feltétel
    valami
Ciklus vége
```

Hasonlóan könnyen és főleg tévedésmentesen értelmezhető a ciklus magját (belsejét) alkotó elágazás. A *bekezdéses leírás* és az *összetett utasítások zárójelezése* elvek következetes alkalmazásával bonyolutabb program is világosan érthető szerkezettel írható meg. A program kissé formálisabbra így alakítható át:

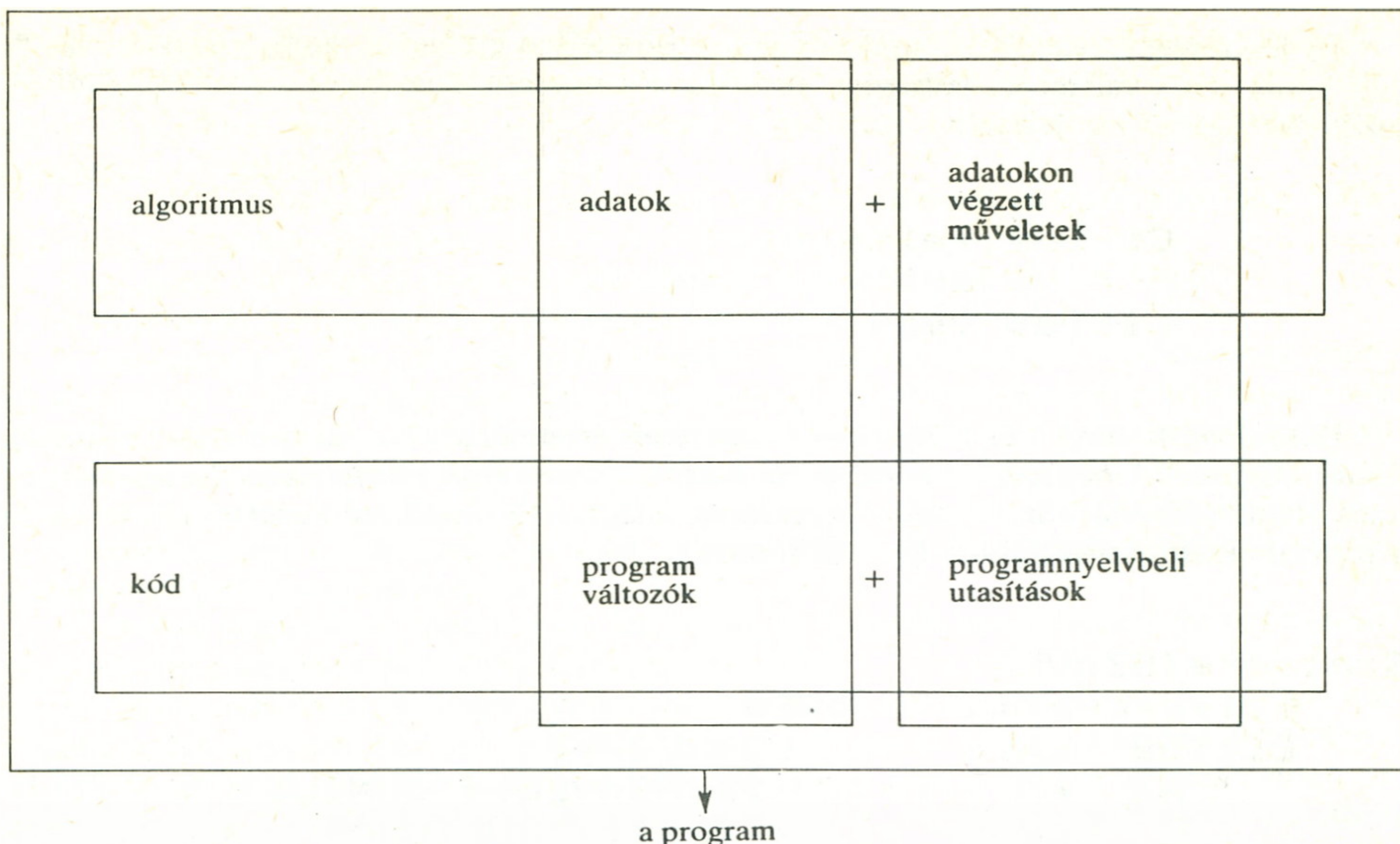
```
Szócsere(SZöVEG):
    elejére-állás(SZöVEG)
    Ciklus  amig  nem SZöVEG-vég
        Ha  következö-szó(SZöVEG)="JUNIUS"
            akkor helyettesit("JULIUS")
            különben helybenhagy
        Elágazás vége
    Ciklus vége
Eljárás vége.
```

Az eljárások paramétereit (azokat az adatokat, amelyeket kívülről kap, ill. kifelé ad) az eljárás neve mögötti zárójelpárok közé írtuk. A megoldásban elemi utasításként használjuk a *következő szó*, a *helyettesít* és a *helybenhagy* eljárásokat. Ezeket most nem finomítjuk!

A *nyílt rendszerű felépítés* elvét követve még általánosíthatjuk úgy a megoldást, hogy a JÚNIUS és JÚLIUS szövegeket is paraméterként soroljuk föl az eljárásban:

```
Szócsere(SZöVEG,MIT,MIRE):
    elejére-állás(SZöVEG)
    Ciklus  amig  nem SZöVEG-vég
        Ha  következö-szó(SZöVEG)=MIT
            akkor helyettesit(MIRE)
            különben helybenhagy
        Elágazás vége
    Ciklus vége
Eljárás vége.
```

Az előző három elvből már körvonalazódik a *programozás folyamata*. Látható, hogy két jól elválasztható tevékenységre bomlik: az *algoritmus készítésére és a kódolásra* (5. ábra).



a programozás folyamata **5. ábra**

A feladat teljes – számítógépes – megoldása persze jóval többet jelent az eddig boncolgatott programozásnál. Ez a tevékenység (ti. a feladat megoldása) tartalmazza a programozásnál semmivel sem kisebb jelentőségű feladatmegfogalmazást – a feladatmegoldás első lépéseként –, az elkészült program tesztelését, sőt optimalizálását, s birtokbavételét, azaz használatát is. Könyvünkben később részletesen taglaljuk ezeket. Jelentőségük aláhúzására azonban már itt is ejtünk róluk néhány szót.

A *feladatmeghatározást* (specifikálást) szokták a leginkább félvállról venni és ebből származik később a legtöbb gond, mivel az elvárásokat nem kellően gondoljuk át. Magától értetődőnek véve elmarad a pontos megfogalmazás és az eredményeknek, ill. a rendelkezésre álló adatoknak az összehangolása. Nagyon fontos tudni, hogy a feladatmeghatározás jósága meghatározó szerepet játszik a programkészítésre fordított idő mennyisége és a készülő program jósága szempontjából.

A természetes emberi tényezők miatt a legnagyobb igyekezet és odafigyelés ellenére is előfordulhatnak kódolási, ill. algoritmikus hibák. Ezek feltárására a *tesztelés* hivatott. Teszteléskor a már kész programot megfelelően választott adatkombinációkkal módszeresen futtatjuk.

Bár a program helyessége szempontjából nem vettünk figyelembe *hatékonysági* kérdéseket, gyakran ezek a program használhatóságát súlyosan érintik. Legkésőbb teszteléskor derülhet fény a program legegyengébb (pl. lassú) pontjaira. Itt is kihasználható (sőt ki kell használni) a módszeres program azon jó tulajdonságát, hogy ez a leggyengébb láncszeme is – mint bármely másik – pontosan meghatározott, s így a programból kiemelhető és esetleg teljesen új alapgondolatokra építve újraírható. Ezután már veszély nélkül ültethető vissza az új, hatékonyabban működő programrész. A hatékonyságról lesz még szó a 4. példában, s utána.

A továbbiakban technikai jellegű ismérveket sorolunk fel, amelyek a program kódjával kapcsolatosak. Inkább úgy mondhatjuk, hogy az előzőek a program megírásához szükségesek, ez utóbbiak pedig a program használhatóságához elengedhetetlenek. Ilyen értelemben beszélhetünk a *csak helyes* programról, amely a feladat logikája szempontjából tökéletes, és a *jó* programról, amely ezen túl elő is segíti saját felhasználását.

1. Az *udvarias program* bemutatkozással kezdi ténykedését, s ezzel tudatja a felhasználójával képességeit, szolgáltatásait, használatának mikéntjét. Az udvariasság másik fontos megnyilvánulása, hogy a program futása során megjelenő kérdések bárki számára – azaz a nem számítástechnikus szakemberek számára is – érthetőek és a válaszok a lehető legegyszerűbben megadhatók legyenek. Így pl. nem rabolja feleslegesen a felhasználó türelmét azzal, hogy a már megadott adatokból kiszámítható, származtatható adatokat kér.

2. A *bolondbiztos (biztonságos) program,* amit a kísérletezni vágyó, vagy éppen balszerencsés felhasználó sem képes ellenőrizetlen vágányokra terelni azáltal, hogy nem a megfelelő módon vagy nem a megfelelő pillanatban válaszol a feltett kérdésre. Ennek érdekében a program kritikus pontjait, azaz ahol a felhasználó közvetlenül avatkozik be a program további menetébe, nagy odafigyeléssel kell megírni. Az esetleges hibalehetőségekre fel kell készíteni a programot úgy, hogy a felhasználónak lehetősége legyen a helyesbítésre is. Nem támaszkodhatunk a számítógép, ill. az értelmezőprogram eleve meglevő hibajelzéseire. Ezek ui. arra valók, hogy segítségükkel felderíthessük és kijavíthassuk az esetleges programhibákat, tehát a program írója, nem pedig a használója számára készültek. A felhasználónak végeredményben semmi köze sincs ahhoz, hogy a program nem közvetlenül, hanem csak a BASIC közvetítésével mozgatja a számítógépet.

A *bolondbiztos* tulajdonság egyébként nemcsak e technikai elv, hanem a piramis elv közvetlen folyománya is, mert ahogy megadtuk az egyes eljárások közötti elvárásainkat (ti. egy eljárás végső feltételei nem lehetnek bővebbek, mint az őt követő eljárás bemeneti feltételei), ugyanúgy rögzíteni kell a külvilág és a beolvasó eljárás kapcsolatára vonatkozó kívánalmakat is. A külvilág azonban a programnak nem része, s így nem is programozható, ezért a bemeneti eljárások kezdő feltételeinek körét megfelelően ki kell szélesíteni. Ez azt jelenti, hogy a bemenő adatok értéktartományát, összeférhetőségét ellenőrizni kell. Ha ezek az eredeti feltételeket nem teljesítik, akkor valamilyen jelzés kíséretében új adatbevitelt kell kezdeményezni.

3. A *jól olvasható program.* A program módosításakor, továbbfejlesztésekor óriási előnyt jelent, ha nem kell programunk minden mellékes vonását újra feltérképezni a megértéshez, hanem lényeges tulajdonságai a program megfelelő helyén könnyen kiolvasható formában megtalálhatók, s így a sebész magabiztos mozdulatával nyúlhatunk bele a program legérzékenyebb részeibe is. Már két idevágó elvet is említettünk, a *bekezdéses leírás* és az *összetett utasítások zárójelezése* elveket. Ezt egészíthetjük ki a kódoláskor különösen nagy jelentőségűvé váló *jó magyarázatok (kommentek)* elvével. A programozási nyelvre való áttéréskor ui. – a programozási nyelv kötöttségei miatt – sok, az algoritmust nagyban jellemző tulajdonság elveszne, ha ezeket az információkat nem őriznénk meg egy-egy jól megfogalmazott megjegyzés formájában.

4. A (jól) *dokumentált program.* Sokszor nincs lehetőség – a program méretére rótt korlátozások miatt – arra, hogy az előző elvet maradéktalanul megvalósíthassuk. Ekkor le kell írni a program fontos vonásait: az algoritmusát, a változóit és ezek szerepét és a kódolásnál követett szabályokat. Ezeket a dokumentációban is rögzíteni kell, amelyben ezenkívül még foglalkozni kell a használat mikéntjével és az esetleges, előrelátható fejlesztési lehetőségekkel is.

A következőkben a már ismertetett *udvariasságra* és *bolondbiztosságra* vonatkozó elv *finomításairól* lesz szó. Ezek a használatot befolyásoló tényezőkre hívják föl a figyelmet. Legfőbb mondanivalójuk, hogy nagy gondot kell fordítani a program által megjelenített információk külalakjára. Ide nem csak az eredmény jellegű kiirandók tartoznak, hanem pl. a tájékoztató, a felhasználóval való párbeszéd módja is.

1. A *lapkezelési technika.* A kiirandó szövegek, adatok logikai egységekre bontva, jól különüljenek el, egyszerre csak annyi és olyan ütemezésben, amennyit s ahogy a felhasználó be tud fogadni. A képernyőre vonatkozó információk esetén ez a lapkezelési technika.

Mit is jelent a lapkezelés? Egyszerre egy képernyőlapnyi információt jelenítünk meg, s a felhasználónak lehetősége van lapozásra, pl. egy adott billentyű lenyomásával jelzi a gépnek *Elolvastam! Lapozz!* (Többek között ennek megvalósítására használható a *Várj, amíg szükséges* utasítás.)

Mit kell még figyelembe venni egy lap összeállításánál? Ügyelni kell pl. a *képernyőlap arányos* kitöltésére, és jó, ha az egy lapon belül szereplő, logikailag szorosan össze nem tartozó információk egymástól elkülönülnek.

Hatásos, ha a mondanivaló legfontosabb elemeit – a gép adta lehetőségek figyelembevételével – kiemeljük (inverz betűkkel vagy villogtatva, vagy más színű háttérrel, ill. betűkkel stb.).

2. A *menütechnika* a lapkezeléssel szorosan összefüggő módszer, amely a felhasználóval való párbeszéd elegáns megszervezésére alkalmas. Általában bonyolult szolgáltatásokkal rendelkező programoknál használatos, amelyből a felhasználó – akár egy menüből – kiválaszthatja a számára szükséges lehetőséget. Minden egyes válaszával (válaszcsoporttal) a kérdések egy nagy hányadát kizárja, ezeket a számítógépnek fel sem kell tennie, megkímélve a felhasználót a fölösleges válaszadásoktól.

3. A kérdéseknél nagyon sokszor épp az okoz bizonytalanságot, hogy a felhasználónak fogalma sincs arról, hogy az adatot milyen mértékegységben kell megadni. Ezért a kérdés szövege mellett célszerű közölni az *adat mértékegységét,* sőt – ha nem magától értetődő, akkor – még az *értéktartományt* is.

Így elkerülhető, hogy pl. a program egy szöget radiánban vár, a gyanútlan felhasználó pedig a legnagyobb természetességgel fokban adja meg az értéket. Az ilyesmiből származó hibát nyilván nem kell ecsetelnünk.

4. A *fontos adatok kiemelésének* nemcsak az információk könnyebb megértése szempontjából van jelentősége – amint ezt már a lapkezelési technikánál taglaltuk –, hanem hasznos a program állapotának, meghatározó paramétereinek azonnali visszajelzésekor is.

Például amikor a számítógép egy hosszadalmas számítást végez, vagy bármilyen időigényes tevékenységbe fog – a hangsúly a hosszadalmasságon, időigényességen van! –, akkor ne maradjon el időnként egy-egy kiírás, ami értesíti a felhasználót, hogy mely tevékenységet végzi éppen a program, és, hogy még kis türelmet kér.

5. A legelemibb elvárás a képernyőn megjelenő szövegekkel szemben, hogy a sorok/szavak *tördelése* a helyesírás szabályainak megfeleljen. Ne sajnálja a programozó a fáradságot mondanivalójának gördülékeny megfogalmazására, szép elhelyezésére, hiszen csak ily módon kaphat mindenki számára kellemes programot!

6. *Következetes beolvasási szokások.* Tartsunk mértékletességet a beolvasási módszerek változatosságában. Nem díjazzák a felhasználók kiterjedt programozási ismereteinket, ha a választ hol <RETURN>-nel lezárva, hol anélkül várja a program.

7. *A hiba megfontolt kijelzése.* A hibák kézben tartásának szükségességéről már volt szó, de a hibák jelzésének mikéntje is jellemzi a programot. Igyekezni kell a hibajelzés

legmegfelelőbb módjának kiválasztására. Ehhez a következő szempontokat érdemes megfontolni:

– A hibajelzés időpontja. Hibát akkor kell jelezni, amikor bekövetkezett, nem pedig valamely következményekor!

– A hibajelzés *képi környezete.* Ha a kezelői hiba javítása után folytatható a végrehatjás, akkor az eredeti képernyőlátványt vissza kell-e, ill. vissza lehet-e állítani?

– A hibajelzés időtartama. Mindig a felhasználó dönthessen a továbbhaladásról!

– A hibajelzés mozgósító ereje. Biztosan észlelhető legyen, érthető legyen, azaz ne legyen túl rövid – csak a program írója ismeri a megfejtést, esetleg külön búvárkodást igényel –, túl hosszú – az ember az ilyenkor természetes türelmetlensége miatt csak hevenyészve képes végigszaladni a leírt *regényen* –, várható felhasználói számára érthető szakkifejezéseket tartalmazzon.

A továbbiakban már megtehetjük első lépéseinket az elmélettől a gyakorlat felé. Vagyis arra a kérdésre keressük a választ néhány tanulságos példán keresztül, hogy hogyan készül a program.

ELSŐ PÉLDA A MÓDSZERES PROGRAMOZÁSRA
(a kitűzött feladat módszeres megoldása)

Térjünk vissza ahhoz a feladathoz, amelyre már láttunk egy *vadon nőtt* megoldást, s az elvek ismeretében gyürkőzzünk neki a megfontolt megoldás elkészítésének. A következőkben az Olvasó ne restellje a lap szélére saját megjegyzéseit is lejegyezni:

– a feladatmegoldás mely fázisában tartunk,
– milyen döntést hoztunk,
– mire emlékeztet ez a lépés,
– egyéb megoldási lehetőségek stb.

Ezzel az *írásos vandalizmussal* senki sem szentségteleníti meg a könyvet, hanem a megoldás felé haladó úton valódi és aktív társsá lép elő. A visszalapozástól megkímélve az Olvasót, megismételjük a feladatot.

Feladat: Adott egy csillagtérkép, ahol a csillagokat a koordinátáikkal adjuk meg (háromdimenziós térkép). Csillagsűrűsödéseknek nevezzük azokat a lehető legnagyobb csoportokat, amelyekben legalább S0 darab csillag van és minden ott levő csillagra igaz, hogy el lehet jutni tőle a csoport bármely tagjához olyan úton, amely a csoport tagjain át vezet és a lépések hossza nem nagyobb S1 fényévnél (elég sűrűn vannak a csillagok). Határozzuk meg a térképen a csillagsűrűsödések számát!

Pontosítsuk, mit is tudunk a feladatról! Adottak N db pont (csillag) koordinátái egy tetszőleges koordináta-rendszerben, a sűrűsödést meghatározó csillagszám (S0) és a sűrűsödésbe tartozás távolságkritériuma (S1). Helyezzük el a koordinátákat az X( ), Y( ), Z( ) vektorokban (X(i), Y(i), Z(i) az i. csillag koordinátái! Az eredmény a sűrűsödések száma (SSZAM) lesz. A lépésenkénti finomítás elvét használjuk, a programot felülről lefelé fejtjük ki, így a legfelső szint a következő tevékenységekből fog felépülni:

```
Program:                                          (1. szint)
    Az adatok beolvasása
        [ N a csillagok száma, X(I),Y(I),Z(I) I=1,
          ...N a koordináták ]
    SSZAM := 0
    A sürüsödések számának meghatározása
        [ SSZAM  a sürüsödések száma ]
    Ki: SSZAM
Program vége.
```

Emlékeztetünk a programkészítés technológiai elveire: beszédes azonosítók, bekezdéses leírás stb. Mit kell tudni a nyelvről, amelyen programjainkat írjuk? A példából látható, hogy – a formalizmuson túlmenően – nagyon hasonlít a magyar nyelvre. A programot egyszerűen mondatok sorozatának tekintjük, amelyek valamilyen tevékenységek egymás utáni elvégzésére szólítanak fel. Két további kifejtést nem igénylő utasítást találunk a programban. Az egyik segítségével egy változónak értéket adunk. A másik utasítás arra szólít fel, hogy egy változó értékét írjuk ki a képernyőre. Ez lesz a program eredménye. A programkészítés elvei szerint megjegyzéseinket, elvárásainkat az egyes eljárásokkal szemben szögletes zárójelek között tüntetjük fel. Ezek az egyes eljárások kifejtésénél szükségesek, mintegy azok feladatának meghatározását jelentik. Az első tevékenység könnyen megoldható, most csupán a leíró nyelvvel való ismerkedés kedvéért foglalkozunk vele. Három utasításfajtát tartalmaz: az első az adatokat beolvasó utasítás, a második az

indexes változók helyfoglalását adja meg, a harmadik pedig bizonyos utasítás(ok) többszöri (ismételt) elvégzésére szólít fel úgy, hogy megadja az elvégzések számát.

```
Az adatok beolvasása:                          (2.1. szint)
     Be: N,S0,S1
     Tömb X(N),Y(N),Z(N)
     [ az X(.),Y(.),Z(.) N elemü vektorok a
       koordinátáknak ]
     Ciklus I=1-töl N-ig
          Be: X(I),Y(I),Z(I)
     Ciklus vége
Eljárás vége.
```

Fordítsuk figyelmünket a második tevékenységre, a megoldás legbonyolultabb részére. Első gondolatunk – figyelembe véve a *döntések elhalasztásának* elvét – az lehet, hogy a csillagokat egyenként vizsgáljuk meg, s döntsük el, hogy sűrűsödésbe tartoznak-e:

```
                              (2.2. szint, 1. változat)
A sürüsödések számának meghatározása:
     Ciklus I=1-töl N-ig
          Az I. csillagot tartalmazò sürüsödés vizs-
                    gàlata
     Ciklus vége
Eljárás vége.
```

A következő feladatunk azon csoport meghatározása, amelybe egy adott csillag tartozik. Ha ennek elemszáma legalább S0, akkor a sűrűsödések számát eggyel növelhetjük. Az aktuális csoport elemszámát jelöljük CSSZAM-mal!

```
                                              (3. szint)
Az I. csillagot tartalmazò sürüsödés vizsgálata:
     CSSZAM:=0
     Az I. csillagot tartalmazò csoport vizsgálata
        [ CSSZAM= a csoport elemeinek száma ]
     Ha CSSZAM >= S0 akkor  SSZAM:=SSZAM+1
Eljárás vége.
```

Figyeljük meg, hogy az algoritmus kifejtésében újra egyetlen döntést hoztunk és minden továbbit igyekeztünk elhalasztani. A döntés: csillagonként az őt tartalmazó csoport vizsgálatára vezetjük vissza a problémát!

A feladat ilyen szintű megoldásában egy hibát vehetünk észre: ha az összes csillag egy sűrűsödésbe tartozik, akkor itt a sűrűsödések számaként a csillagok számát kapjuk. Ez nyilvánvaló hiba! Ezek szerint tehát nem szabad az összes csillagra elvégezni az előbbi tevékenységet, hanem csak azokra, amelyeket még nem soroltunk be egyetlen sűrűsödésbe, egyetlen csoportba sem. Olyan utasításra lesz szükségünk, amelynek segítségével

egy tevékenységet bizonyos feltételtől függően hajthatunk végre. (Figyelem! Itt a program működésére vonatkozó lényeges döntést hoztunk!)

Lépjünk vissza az előző szintre! Ezen többletinformáció tárolására használjunk egy V( ) vektort, V(I) legyen IGAZ, ha az I. csillagot még sehova sem soroltuk be! Kezdetben a V( ) vektor összes eleme IGAZ értékű, majd a vizsgálatok során a már megvizsgáltakat HAMIS értékűre állítjuk. A V( ) vektor elemeit feltételes utasítás feltételeként használhatjuk.

```
                                    (2.2. szint, 2. változat)
A sürüsödések számának meghatározása:
     Tömb V(N)
     Ciklus I=1-töl N-ig
          V(I):=IGAZ
     Ciklus vége            [ még egyet sem vizsgáltunk]
     Ciklus I=1-töl N-ig
               [ SSZAM a sürüsödések száma az I. csil-
                 lag vizsgálata elött ]
          Ha V(I) akkor    Az I. csillagot tartalmazò
                                  sürüsödés vizsgálata
               [ SSZAM a sürüsödések száma az I. csil-
                 lag vizsgálata után és az  I.-kel egy
                 egy csoportba tartozòkat már nem kell
                 vizsgálni ]
     Ciklus vége
Eljárás vége.
```

Megjegyzés: A BASIC-ben jártas Olvasó nyilván észrevette, hogy ilyen típusú V( ) vektor nem létezik a BASIC-ben. Mégis bátran definiáljuk ilyennek, s nem törődünk azzal, hogy e logikai típusú változónak milyen BASIC változót fogunk megfeleltetni, s felhasználásánál hogyan tudjuk megvalósítani kezelését. Ez is példázza, hogy sohasem kell algoritmuskészítés során figyelembe venni a programozási nyelv kötöttségeit, irányadó mindig a probléma logikája legyen!

Ez a módosítás a 3. szint algoritmusát szerencsénkre nem befolyásolja, így az az előbb megírt formában elfogadható.

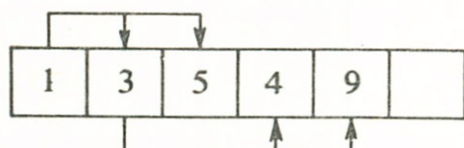
Egyetlen kifejtetlen *láncszem* maradt a programban, nevezetesen az I. csillagot tartalmazó csoport vizsgálata.

Kérjük Olvasónkat, most ne sajnálja az időt és alaposan nézze újra végig, s gondolja át az előttünk álló programot! Próbálkozzék ezen a szinten belátni a program helyességét, feltételezve a még kifejtetlen eljárások helyes működését! Vannak formális módszerek is a helyesség belátására, mi ezektől egyelőre eltekintünk.

Elérkeztünk egy olyan ponthoz, ahol már komolyan figyelembe kell vennünk a *sűrűsödés* definícióját (s már tovább nem tudjuk halogatni e súlyponti fogalom felhasználását). Eddig a pillanatig egy egyszerű megszámlálási feladatot kellett megoldanunk. A nehézséget az okozza, hogy rendkívül bonyolult az egyes csoportok, valamint csillagszámuk meghatározása. Milyen csillagok tartoznak egy csoportba? Az elsőnek választott csillag, a hozzá *közel* levő csillagok, az azokhoz közel levő csillagok és így tovább. Ezért a kezdő csillagból kiindulva addig kell közeli csillagokat keresni, amíg az összes addig megtalált csillagot meg nem vizsgáltuk. Ehhez szükségünk van az aktuális csoportba tartozó csillagokra. Ezek sorszámait a CS(N) SOR-ban tároljuk.

A SOR egy olyan sorozat, amelyből elvenni mindig csak az elejéről lehet. (Nekünk ui. az éppen összeállítás alatt levő csoport következő, még nem vizsgált eleméhez *közeli* csillagokat kell felvenni a csoportba.) A SOR-ba felvenni új számokat pedig csak a végére lehet. (Egy új csillag felvételekor teszünk ilyet.)

Például a 6. ábrán levő sorba az 1. miatt vettük fel a 3.-at és az 5.-et, a 3. miatt pedig a 4.-et és a 9.-et.



6. ábra

A SOR-t egy vektorban tároljuk. Az elvétel miatt tudnunk kell, hogy a vektor melyik elemét tekintjük a SOR első elemének, ezt egy mutató jelzi. Azt, hogy hol van a SOR utolsó eleme, szerencsére nem kell külön kibogarásznunk, mert a már használt CSSZAM változó tartalmazza. Vezessünk be egy új változót (K), amely a SOR első elemére mutat, ez éppen a vizsgált csillag csoportbeli sorszámát adja meg! A CS(N) vektor az eljárás saját változója, így – figyelembe véve az adatok elszigetelésének elvét – az eljárásnál magasabb szinten nem használható! Itt egy új típusú ciklust alkalmazunk, hiszen most előre nem tudjuk, hogy a ciklus utasításait – a ciklusmagot – hányszor kell végrehajtani. Ismerjük azonban azt a feltételt, amelynek teljesülése esetén szükség van a ciklusmag újabb végrehajtására.

```
                                        (4. szint)
Az I. csillagot tartalmazò csoport vizsgàlata:
    Tömb CS(N)
    Csillag felvétele a csoportba(I)
        [ CSSZAM=1 , a csoportnak az I. csillag az
          elsö eleme, amelyet màr nem kell vizs-
          gàlni ]
    K:=1
    Ciklus amig K <= CSSZAM
            [ K=a csoport ennyiedik tagjàt vizsgàl-
              juk , CSSZAM= a csoportnak a K. csil-
              lag vizsgàlata elötti tagjai szàma ]
          A CS(K). csillaghoz közeli csillagok vizs-
                        gàlata
            [ CSSZAM= a csoport K. csillagàhoz kö-
              zeli csillagokkal együtt a csoportnak
              ennyi tagja van és màr nem kell a
              CS(K).-hoz közelieket vizsgàlni ]
        K:=K+1
    Ciklus vége
Eljàràs vége.
```

Ezen a szinten két kifejtetlen tevékenységet találunk. A csillagfelvételt egyszerűen el tudjuk intézni: jelezni kell, hogy ezt a csillagot a továbbiakban ne vizsgáljuk, az aktuális csoportba tartozó csillagok számát eggyel növelni kell, majd a CS( ) vektorba be kell jegyezni ezt a csillagot (egy új elem felvétele a SOR-ba).

```
                                              (5.1. szint)
Csillag felvétele a csoportba(I):
     V(I):=HAMIS
     CSSZAM:=CSSZAM+1
     CS(CSSZAM):=I
Eljárás vége.
```

Meg kell még keresni az adott csillaghoz közeli csillagokat. Először határozzuk meg, hogy mely csillagok között keressük ezeket! Biztosan nincs szükségünk a most vizsgáltat megelőzőekre, a későbbiek közül pedig csak azok érdekesek, amelyeket eddig még nem vizsgáltunk.

```
                                              (5.2. szint)
A CS(K). csillaghoz közeli csillagok vizsgálata:
    Ciklus J=I+1-től N-ig
           [ CSSZAM   a csoport tagjai száma  a J.
             csillag vizsgálata elött ]
         Ha V(J) akkor  A J. csillag vizsgálata
           [ CSSZAM   a csoport tagjai száma  a J.
             csillag vizsgálata után ]
    Ciklus vége
Eljárás vége.
```

Már csak a J. csillag *közeliségét* kell eldönteni: ez egy távolságmeghatározásból áll, s ha a távolság kisebb S1-nél, akkor fel kell venni a csillagot az aktuális csoportba!

```
A J. csillag vizsgálata:                      (6. szint)
    T:= A J. és a CS(K). csillag távolsága
    Ha T <= S1 akkor  Csillag felvétele a
                                  csoportba(J)
Eljárás vége.
```

A feladat megoldásának első, tagadhatatlanul a nehezebb, de mindenképpen szebb részével végeztünk, következik a program kódolása, ami mechanikusan végezhető, de éppen ezért nagy veszélyforrás, s így lankadatlan figyelemre van szükség.

A kódolást kezdjük a megoldás felső szintjével! Itt az első, a második és a negyedik tevékenységet közvetlenül átírhatjuk BASIC nyelvre. Természetesen figyelembe kell venni, hogyan kell jó tulajdonságú programot készíteni. Ezért a program először egy tájékoztatót írjon a feladatról (technikai elvek)!

Megjegyzés: Néhány HT–1080Z specialitásra hívjuk fel a figyelmet:

CLS – a képernyő törlése;
PRINT@ – pozicionálás a képen;
INKEY – a billentyű lenyomásának figyelése.

```
10 CLS : PRINT "CSILLAGSURUSODESEK SZAMA"
20 PRINT
30 PRINT "  A PROGRAM EGY TETSZOLEGES CSILLAGTERKE-"
40 PRINT "PEN  A  CSILLAGSURUSODESEK  SZAMANAK MEG-"
50 PRINT "HATAROZASARA KEPES. CSILLAGSURUSODESEKNEK"
60 PRINT "NEVEZZUK  AZOKAT  A  LEGNAGYOBB CSOPORTO-"
70 PRINT "KAT , AMELYEKBEN  LEGALABB 'SO' DB. CSIL-"
80 PRINT "LAG  VAN , ES  MINDEN  OTT LEVO CSILLAGRA"
90 PRINT "IGAZ , HOGY  BARMELYIK CSOPORTTARSAHOZ EL"
100 PRINT "LEHET JUTNI OLYAN UTON , AMELY A CSOPORT"
110 PRINT "TAGJAIN KERESZTUL HALAD  ES  EGYIK LEPES"
120 PRINT "SEM HOSSZABB 'S1' FENYEVNEL."
130 PRINT £15*64,"HA ELOLVASTA, NYOMJA LE A";
         "<NEW LINE>-T!";
140 IF INKEY$="" THEN 140
```

A tájékoztató kiírása után az adatok beolvasása a feladat. Elöljáróban meg kell jegyezni, hogy olyan rendszerekben, ahol a program eredménye televízió képernyőjére kerül, a képernyőt nem célszerű sorok végtelen sorozatának tekinteni. (Bár lehetőség van arra, hogy az utolsó sor kiírása után a képet automatikusan feltoljuk, ezt azonban most nem célszerű használni.) Az automatikus egymás után írás helyett a képeket tervezzük meg, a képek váltásáról pedig magunk gondoskodjunk!

Minden beolvasás előtt írjuk ki, hogy mit szeretnénk beolvasni, ellenőrizzük a kapott adatokat, s ha hibásak, akkor kérdezzünk újra (biztonságosság)! Vannak olyan BASIC rendszerek, amelyekben a DIM utasítást nem érdemes többször végrehajtani, így érdemes kiemelni a program elejére (a HT–1080Z esetén javasoljuk).

```
150 CLS
160 INPUT "HANY CSILLAG VAN A TERKEPEN";N
170 IF N<1 THEN 150
180 DIM X(N),Y(N),Z(N),V(N),C(N)
190 INPUT "CSILLAGSZAM";SO
200 IF SO<1 OR SO>N THEN 190
210 INPUT "TAVOLSAG (FENYEV)";S1
220 IF S1<0 THEN 210
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250    PRINT "AZ";I;". CSILLAG KOORDINATAI";
260    INPUT X(I),Y(I),Z(I)
270 NEXT I
```

Következik az SSZAM kezdőértékének beállítása, a sűrűsödések számának meghatározása és kiírása. Tekintettel kell lennünk arra, hogy egyes BASIC nyelvjárásokban nem lehet akármilyen hosszú változóneveket használni. Ezért az SSZAM változónevet is kódolni kell, ezért az SSZAM név helyett a programban mindenhol S-et fogunk használni. A hosszú nevek használata ellen és mellette is több érvet hozhatnánk, de a legfontosabb: e könyv azoknak is szól, akik csak a szigorúbb BASIC változatokkal dolgoznak. Vegyük észre, hogy az előbb a DIM utasításban már a CS( ) vektor nevét is C-re kódoltuk!

A változók *kódolásakor* valamilyen rendet kell tartani, hiszen vigyázni kell arra, nehogy különböző változóknak a BASIC programban azonos név feleljen meg. Például készíthetünk a változókról egy táblázatot, amelyben szerepel az algoritmusleírásban használt név, a BASIC programban kódolt név, esetleg az, hogy a program mely részében, részeiben használjuk őket stb. Itt emlékeztetünk a technikai elveknél az *adatok elszigeteléséről* mondottakra.

```
280 CLS : PRINT "DOLGOZOM!"
290 S=0
300 GOSUB 400 : REM A SURUSODESEK SZAMANAK MEGHAT.
310 CLS
320 PRINT "A CSILLAGSURUSODESEK SZAMA:";S
330 STOP
```

A kódolásból hátramaradt a sűrűsödések számát meghatározó szubrutin megírása. Ezzel egy döntéshez jutottunk: hogyan tudunk egy vektorban IGAZ logikai értéket tárolni? Válasszuk a következő megoldást (nem ez az egyetlen!): az IGAZ értéket jelentse az, ha a V vektor megfelelő elemében 0 van, a HAMIS értéket pedig, hogy 1!

```
400 REM A SURUSODESEK SZAMANAK (S) MEGHATAROZASA
410 FOR I=1 TO N
420     V(I)=0 : REM  EZT MEG NEM VIZSGALTUK
430 NEXT I
440 FOR I=1 TO N
450     IF V(I)=0 THEN GOSUB 500 : REM AZ I. CSIL-
        LAGOT TARTALMAZO SURUSODES VIZSGALATA
460 NEXT I
470 RETURN
```

Figyeljük meg, hogy ha egy szubrutint írunk, akkor az elején mindig leírjuk röviden a feladatát!

Az algoritmus kifejtésének következő részéből az 500-as soron kezdődő szubrutin elkészítése következhet. A kódoláson egyszerűsíthetünk egy kicsit. Egy újabb szubrutint kellene írni, de azt csak egyetlen helyen használjuk, így a szubrutinhívás helyére egyszerűen behelyettesíthetjük magát a szubrutin szövegét. Itt kell felhívnunk a figyelmet a programozó *szabadságára:* ugyanazt az algoritmust többféleképpen is lehet kódolni. A kódolási szabályok nagyvonalúan használhatók. Ebben a programban lesz egy olyan szubrutin, amelyet több helyen, különböző paraméterekkel szeretnénk használni, és így meg kell szervezni a paraméterátadást. A BASIC a szubrutinparaméterezéshez nem ad eszközt, ezért be kell – külön e célra – vezetnünk egy ún. paraméterváltozót (L), amelynek a szubrutinhívás előtt a megfelelő kifejezés értékét adjuk (520-as programsor). Természetesen ezt a változót kell használnunk a szubrutinban is. Újabb változó kódolását kell elvégeznünk, a CSSZAM nevű változóból BASIC programban A lesz. Könnyen ellenőrizhető a változókról készült táblázat alapján, hogy e kódolás lehetséges, mivel ezt a nevet még nem kódoltuk, s ilyen BASIC azonosítót sem használtunk. A C( ) vektorra vonatkozó DIM utasítást már a programban elhelyeztük, így itt nyugodtan felhasználhatjuk.

```
500 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
510 A=0
520 L=I : GOSUB 600 : REM AZ L. CSILLAG FELVETELE
530 K=1 : REM ELOSZOR A KEZDO CSILLAGHOZ KOZELIE-
         KET FOGJUK VIZSGALNI
540 IF K<=A THEN 550 ELSE 570
550   GOSUB 700 : REM A C(K). CSILLAGHOZ KOZELIEK
         VIZSGALATA
560   K=K+1 : GOTO 540
570 REM A ITT A CSOPORT CSILLAGJAINAK SZAMA
580 IF A>=S0 THEN S=S+1
590 RETURN
```

A csillag felvételét elvégző szubrutinban az L paramétert kell használnunk, nem pedig az algoritmusleírásban megadott I-t:

```
600 REM AZ L. CSILLAG FELVETELE A CSOPORTBA
610 V(L)=1
620 A=A+1 : C(A)=L
630 RETURN
```

Következik a C(K). csillaghoz közeli csillagok vizsgálatát végző szubrutin. Itt is behelyettesítjük az algoritmusban található *A J. csillag vizsgálata* nevű eljárást:

```
700 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
710 IF I=N THEN RETURN : REM UTOLSO CSILLAG
720 FOR J=I+1 TO N
730     IF V(J)=0 THEN 740 ELSE 760
740     T=SQR((X(J)-X(C(K)))ü2+(Y(J)-Y(C(K)))ü2+
         (Z(J)-Z(C(K)))ü2)
750     IF T<=S1 THEN L=J : GOSUB 600 : REM A J.
         CSILLAG FELVETELE
760 NEXT J
770 RETURN
```

A programmal elkészülve foglaljuk össze, hogyan kódoltuk a változókat! Ez azért lényeges tevékenység, mert a BASIC programozási nyelv nem ismer *saját* hatáskörű változókat, s így a kódolásnál ügyelni kell a névhozzárendelés egyértelműségére. Az egyértelműségtől – mint láttuk – csak egy esetben célszerű eltérni: az egyes eljárások saját változói esetén. Így a program kódolt változói, ill. az algoritmusleírásban adott névvel használt változók:

N	–	N	közös
X(N)	–	X(N)	közös
Y(N)	–	Y(N)	közös
Z(N)	–	Z(N)	közös
S0	–	S0	közös

S1	–	S1		közös
SSZAM	–	S		közös
V(N)	–	V(N)	2.2	saját
I	–	I	2.1	munka
I	–	I	2.2	saját
CSSZAM	–	A	3.	saját
CS(N)	–	C(N)	4.	saját
	–	L	4.,6.	saját
K	–	K	4.	saját
J	–	J	5.2	munka
T	–	T	6.	munka

A táblázatban nem szerepel a változók egyéb jellemzőinek leírása.

Ezzel elkészült a program a *feladathoz!* Azt gondolhatnánk, hogy készen vagyunk a munkával, ez azonban – mint a későbbiekben látni fogják – koránt sincs így.

MÁSODIK PÉLDA A MÓDSZERES PROGRAMOZÁSRA
(hasonló problémák, hasonló programok)

Nézzünk egy másik feladatot!

Feladat: Adott M ember között N db rokoni kapcsolat (*A rokona B-nek* formában). Adjuk meg azon legnagyobb embercsoportok számát, amelyeken belül mindenki rokona a csoport bármely más tagjának. A rokonság szimmetrikus és tranzitív reláció: ha A rokona B-nek és B rokona C-nek, akkor A is rokona C-nek.
Pontosítsuk, mit is tudunk a feladatról: az embereket sorszámaikkal azonosítjuk, adott N db számpár (a rokoni kapcsolatok). (Pl.: Az (1,2), (2,3), (2,4), (5,6), (6,7) kapcsolatokból azt kaphatjuk, hogy két csoport van, az elsőbe tartozik az 1., 2., 3., 4. ember, a másodikba pedig az 5., 6. és 7.)
Ezeket a számpárokat az X(N,2) mátrixban tároljuk. A program eredménye a rokonsági csoportok száma (RCSSZAM) lesz. A program felépítése:

```
Program:                                    (1. szint)
    Az adatok beolvasása
      [ X(N,2) tartalmazza a rokonsági kapcsolato-
        kat ]
    RCSSZAM := 0
    A rokonsági csoportok számának meghatározása
      [ RCSSZAM a rokonsági csoportok száma ]
    Ki: RCSSZAM
Program vége.
```

Az algoritmus nagyon hasonlít az előző feladat megoldásának legfelső szintjére. Ebben persze nincs semmi meglepő, a programok egy jelentős csoportja ilyen szerkezetű. Foglalkozzunk a második tevékenységgel! Vizsgáljuk meg az egyes rokoni kapcsolatokra azt, hogy belőlük milyen csoportot lehet kiépíteni! Most is találhatunk az előző feladatéhoz hasonló megoldást.

Emlékezzünk vissza: ott csak azokat a csillagokat kellett megvizsgálni, amelyeket még nem soroltunk be csoportba. Ebben az esetben azokat a kapcsolatokat kell megvizsgálni, amelyeket rokoni csoport felépítésére még nem használtunk. Igy itt is pontosan úgy használjuk a V( ) vektort, mint az előző feladat megoldásában.

```
                                            (2. szint)
A rokonsági csoportok számának meghatározása:
    Ciklus I=1-től N-ig
        V(I):=IGAZ
    Ciklus vége
    Ciklus I=1-től N-ig
            [ RCSSZAM  a rokonsági csoportok száma
              az  I. kapcsolatból kiinduló csoport
              vizsgálata előtt ]
        Ha V(I) akkor  Az I. kapcsolatból kiinduló
                             csoport vizsgálata
            [ RCSSZAM  a rokonsági csoportok száma
              az  I. kapcsolatból kiinduló csoport
              vizsgálata után ]
```

Az eddigi hasonlóság gyümölcsözőnek bizonyult. Próbáljuk meg továbbra is az előző feladat megoldása alapján készíteni a megoldást! Az aktuális csoport elemszámát most CSSZAM-mal jelöljük, a számlálást pedig az elemszámtól függetlenül elvégezzük.

```
                                                   (3. szint)
Az I. kapcsolatból kiinduló csoport vizsgálata:
    CSSZAM:=0
    Az I. kapcsolatból kiinduló csoport meghatáro-
                        zása
        [ CSSZAM  a csoport  elemeinek száma, a cso-
          port  elemeit  a  továbbiakban  nem  kell
          vizsgálni ]
    RCSSZAM:=RCSSZAM+1
Eljárás vége.
```

A csoport meghatározását úgy oldhatjuk meg, hogy felvesszük az aktuális kapcsolatban szereplő két embert, majd a csoport minden egyes tagjának megkeressük az összes rokonát és azokat is felvesszük a csoportba. Ezt a tevékenységet egészen addig végezzük, amíg találunk rokoni kapcsolatokat. Használjuk az előző feladat CS(N) vektorát, CSSZAM a csoport tagjainak száma.

```
                                                   (4. szint)
Az I. kapcsolatból kiinduló csoport meghatározása:
    V(I):=HAMIS
    Az X(I,1). ember felvétele a csoportba
    Az X(I,2). ember felvétele a csoportba
        [ CSSZAM=2 , a csoportnak tagja az  I. kap-
          csolatban  szereplö  két  ember  , ezeket a
          továbbiakban nem kell vizsgálni ]
    K:=1
    Ciklus amig K <= CSSZAM
          [ CSSZAM a csoport tagjai száma a csoport
            K. tagja rokonai vizsgálata elött ]
        A CS(K). ember rokonainak keresése
          [ CSSZAM a csoport tagjai száma a csoport
            K. tagja rokonai vizsgálata után ]
        K:=K+1
    Ciklus vége
Eljárás vége.
```

A két definiálatlan tevékenységből egy ember felvétele a csoportba hasonló egy csillag felvételéhez. A felvétel feltétele bonyolult, mert most nem tudjuk szavatolni, hogy az éppen vizsgált ember nem kerülhetett bele korábban a csoportba.

```
                                          (5.1. szint)
Az L. ember felvétele a csoportba:
    Ha CS(P)<>L  [P=1,..CSSZAM] akkor
        CSSZAM:=CSSZAM+1
        CS(CSSZAM):=L
    Elágazás vége
Eljárás vége.
```

A CS(K). ember rokonainak vizsgálata abból áll, hogy megvizsgáljuk a még figyelembe nem vett rokoni kapcsolatokat (ahogyan korábban a még figyelembe nem vett csillagokat vizsgáltuk):

```
                                          (5.2. szint)
A CS(K). ember rokonainak vizsgálata:
    Ciklus J=I+1-től N-ig
        [ CSSZAM a csoport tagjai száma a J. kap-
          csolat vizsgálata előtt ]
      Ha V(J) akkor  A J. rokoni kapcsolat vizs-
                                        gálata
        [ CSSZAM a csoport tagjai száma a J. kap-
          csolat vizsgálata után ]
    Ciklus vége
Eljárás vége.
```

Azt kell még eldönteni, hogy a J. rokoni kapcsolat növeli-e a csoportot, azaz szerepel-e benne a CS(K). ember, s ha igen, akkor a rokonát fel kell venni a csoportba.

```
                                          (6. szint)
A J. rokoni kapcsolat vizsgálata:
    Ha X(J,1)=CS(K) akkor
        V(J):=HAMIS
        Az X(J,2). ember felvétele a csoportba
    különben  Ha X(J,2)=CS(K) akkor
                  V(J):=HAMIS
                  Az X(J,1). ember felvétele a csoportba
              Elágazás vége
    Elágazás vége
Eljárás vége.
```

Ezzel algoritmusunk elkészült, a program kódolásával most nem foglalkozunk.

Nagyon fontos tanulságot vonhatunk le az előbbi gondolatsorból: két jelentősen eltérő szövegű feladatnak szinte azonos megoldását kaptuk. A feladatokban ezek szerint volt valami közös: a szövegük mögötti szerkezet! Ha ezeket a szerkezeteket egy konkrét feladat megoldása során felismerjük, akkor a megoldás nagyon egyszerű, világos lesz. Ez azt sugallja, hogy léteznek valamilyen programsémák (elvek), amelyeket az egyes konkrét feladatok megoldásánál alkalmazni lehet, s ha lehet, akkor kell is. Ez így is van, mint ahogyan a következő fejezetben látható.

PROGRAMOZÁSI TÉTELEK

Hogyan fogjunk hozzá egy feladat – módszeres – megoldásához! A példákból kiderült, hogy a felülről lefelé való kifejtés során háromféle szerkezetet használtunk: utasítások *egymás utáni végrehajtását*, feltételtől függő *elágazást*, valamint utasítások ismételt végrehajtását *(ciklus)*; utasításcsoportjainkat pedig *eljárásokra* tagoltuk. Mit lehet tenni azonban, ha nem tudjuk, hogy melyik kifejtést is alkalmazzuk? Honnan várhatunk segítséget? A segítség magukban a feladatokban található, csak fel kell ismerni!

Ismerkedjünk meg néhány tipikus programsémával! Néhány *típusalgoritmust* adunk meg, amelyek felhasználhatók konkrét feladatok megoldása során. *Programozási tételeket* fogunk kimondani (kicsit szerényebben programozási recepteknek is nevezhetnénk őket). Elnevezésük a matematikai tételekkel kapcsolatos hasonlóságból származik. Ezek a tételek azt mondják ki, hogy a megadott programok *helyes megoldásai* a hozzájuk tartozó feladatoknak. A tételek bizonyítását a közölt programok helyességének bizonyítása jelenti. Egy későbbi fejezetben programok helyességének bizonyításával is foglalkozunk, amelynek segítségével az itt megadott tételek is bizonyíthatók (bár bizonyításuk menetét nem közöljük).

Ezek után a legtöbb feladat megoldása során nincs is más dolgunk, mint a kifejtés adott szintjén meghatározni, hogy milyen feladatról van szó, s utána alkalmazni a megfelelő algoritmust. Csak néha-néha van szükség új tételek kimondására, alkalmazására. Az ilyen típusú programkészítés nyilvánvalóan biztosítja, hogy ha a feladatot jól értettük meg, akkor helyes algoritmust írunk megoldására. Megjegyzendő azonban, hogy a tételek gépies alkalmazásával nem mindig a lehető legkedvezőbb programot kapjuk. A helyes megoldás hatékonyabbra való átírása azonban már külön tevékenység, s általában erősen feladatfüggő.

Feladataink jól meghatározott osztályokba sorolhatók kiindulási adataik és az általuk szolgáltatott eredmény alapján. A legegyszerűbbek egy adatból egy adatot számolnak ki. Kicsit bonyolultabbak azok, amelyek egy adatból egy sorozatot adnak. (Például adjuk meg az X első elemű, L növekményű számtani sorozat első N tagját!)

Számítástechnikai szempontból bonyolultabbak az olyan feladatok, amelyekben egy sorozathoz kell rendelni egy értéket, vagy pedig egy újabb sorozatot. Például annak a vizsgálata, hogy egy adott szám (N) prím-e, azt jelenti, hogy el kell döntenünk, hogy a 2 és N/2 közötti számok közül osztója-e valamelyik az N-nek. Megjegyezzük, hogy egy sorozatot sokszor egyetlen bemeneti paraméter értékével meg tudunk adni, amiből arra a megtévesztő következtetésre juthatnánk, hogy valami nagyon egyszerű feladatról van szó.

Ezt a bonyolultabb csoportot tovább bontjuk. Az egyszerű feladatok esetén a sorozathoz egyetlen értéket kell hozzárendelni, bonyolultabb esetben pedig egy sorozatot.

Az első osztályba tartoznak az olyan feladatok, amelyekben egyszerre csak a sorozat egyetlen elemének vizsgálatára van szükség, ill. olyanok is, amelyeknél egyszerre több elemet is figyelembe kell venni. A második csoportban lehetséges, hogy a sorozat elemeiből új értékeket kell kiszámolni (darabszámuk lehet az eredeti sorozat elemszámától különböző is), valamint az is, hogy az eredeti sorozat valamilyen permutációját kell előállítani.

Az egyes típusalgoritmusoknál mindig megadjuk az általános feladatot, az azt megoldó algoritmust, egy konkrét feladatot, a feladatot megoldó algoritmust, a feladatban szereplő adatok megfeleltetését az általános feladat adataival.

ÉRTÉK-HOZZÁRENDELÉS SOROZATOKHOZ
(elemenkénti feldolgozás)

Összegzés

A feladatban adott egy N elemű számsorozat. Számoljuk ki az elemek összegét! A sorozatot (számhalmazt) – most és a továbbiakban is – az N elemű A(N) vektorban tároljuk. (Bár sok egyszerű esetben a sorozat egyes elemeit a sorszámukból is meghatározhatjuk).

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

Példa: adjuk össze az első N természetes számot!

Megfeleltetés:

sorozat – 1,2...,N.

```
Eljárás:
         S:=0
         Ciklus I=1-től N-ig
                S:=S+I
         Ciklus vége
Eljárás vége.
```

Rögtön ez az első példa mutatja, hogy a feladat megfogalmazása, a felhasznált adatok megadása, az elképzelés rögzítése mennyire befolyásolja a megoldást. A feladatra ui. egy sokkal egyszerűbb megoldást is kaphatunk, ha a következőképpen értelmezzük: számoljuk ki az N→N*(N+1)/2 függvény értékét valamilyen N természetes számra. Így a megoldás:

```
Eljárás:
         S:=N*(N+1)/2
Eljárás vége.
```

Megjegyzés: Ekkor az összegzési tétel alkalmazása elhibázott lépés lenne.

Eldöntés

Ebben a feladatban rendelkezésre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Az algoritmus eredménye: van-e a sorozatban legalább egy T tulajdonsággal rendelkező elem?

```
Eljárás:
     I:=1
     Ciklus amig I<=N és A(I) nem T tulajdonságu
          I:=I+1
     Ciklus vége
     VAN_E:= I<=N
Eljárás vége.
```

Példa: készítsünk algoritmust, amely egy névről (X$) eldönti, hogy egy hónap neve-e!
Megfeleltetés:

N – 12;
sorozat – január, február, ... (A$(N));
tulajdonság – A$(I)= X (1<=I<=12).

```
HÓNAP_E(X$):
     I:=1
     Ciklus amig I<=12 és A$(I)<>X$
          I:=I+1
     Ciklus vége
     HÓNAP_E:= I<=12
Eljárás vége.
```

Hasonló feladat a következő: rendelkezésre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Az algoritmus eredménye: igaz-e, hogy a sorozat minden eleme T tulajdonságú?

```
Eljárás:
     I:=1
     Ciklus amig I<=N és A(I) T tulajdonságu
          I:=I+1
     Ciklus vége
     IGAZ_E:= I>N
Eljárás vége.
```

Felhívjuk Olvasónk figyelmét a megoldás és az előző algoritmus nagymértékű hasonlóságára. Ez tulajdonképpen várható is volt, hiszen a *mindegyik T tulajdonságú* ugyanazt jelenti, mint a *nem létezik nem T tulajdonságú.*

Kiválasztás

Rendelkezésre áll egy N elemű sorozat, egy, a sorozat elemein értelmezett T tulajdonság, valamint azt is tudjuk, hogy a sorozatban van legalább egy T tulajdonságú elem. A feladat ezen elem sorszámának a meghatározása.

A megoldás az előző alapján nagyon egyszerű. Csupán annyi az eltérés, hogy azt a többletismeretet, hogy biztosan van T tulajdonságú elem, az eljárásban levő ciklusfeltétel egyszerűsítésére használjuk.

```
Eljárás:
     I:=1
     Ciklus amig A(I) nem T tulajdonságu
          I:=I+1
     Ciklus vége
     SORSZ:=I
Eljárás vége.
```

Példa. Készítsünk algoritmust egy természetes szám 1-től különböző pozitív osztójának meghatározására! (Ilyen van, ha nem korlátozzuk a feladatot valódi osztó megkeresésére.)

Megfeleltetés (a sorozat elemei számíthatók):

sorozat – 2, 3, ..., N;
tulajdonság – I osztója N-nek.

```
OSZTO(N):
     I:=2
     Ciklus amig I nem osztója N-nek
          I:=I+1
     Ciklus vége
     OSZTO:=I
Eljárás vége.
```

Keresés

Rendelkezésre áll egy N elemű sorozat, egy, a sorozat elemein értelmezett T tulajdonság. Olyan algoritmust kell írni, amely eldönti, hogy van-e T tulajdonságú elem a sorozatban, és ha van, akkor megadja a sorszámát. (Ez a feladat az előző kettő összefoglalása.)

```
Eljárás:
     I:=1
     Ciklus amig I<=N és A(I) nem T tulajdonságu
          I:=I+1
     Ciklus vége
     VAN_E:= I<=N
     Ha VAN_E akkor SORSZ:=I
Eljárás vége.
```

Példa. Készítsünk algoritmust, amely egy autóbusz-menetrend alapján megállapítja, hogy van-e A városból B-be közvetlen járat, s ha van, akkor megad egy ilyet. (A menetrendben indulási és végállomások szerepelnek.)

Megfeleltetés:

N – a buszjáratok száma;
sorozat – (ind. áll., végáll.), ... (A$(N,2));
tulajdonság – A$(I,1)=„A" és A$(I,2)=„B".

```
Járat keresés:
     I:=1
     Ciklus amig I<=N és
                     (A$(I,1)<>"A" vagy A$(I,2)<>"B")
          I:=I+1
     Ciklus vége
     VAN_JáRAT:= I<=N
     Ha VAN_JáRAT akkor SORSZ:=I
Eljárás vége.
```

Megszámolás

Rendelkezésre áll egy N elemű sorozat, egy, a sorozat elemein értelmezett T tulajdonság. Most a T tulajdonsággal rendelkező elemek megszámolása a feladat.

```
Eljárás:
     S:=0
     Ciklus I=1-töl N-ig
          Ha A(I) T tulajdonságu akkor S:=S+1
     Ciklus vége
Eljárás vége.
```

Példa. Készítsünk algoritmust, amely N ember fizetése alapján megadja a 3000 Ft-nál kevesebbet keresők számát!

Megfeleltetés:

sorozat – fizetés1, fizetés2,... (A(N));
tulajdonság – $A(I) < 3000$.

```
Kis fizetésüek száma:
     S:=0
     Ciklus I=1-töl N-ig
          Ha A(I)<3000 akkor S:=S+1
     Ciklus vége
Eljárás vége.
```

Megjegyzés: Érdemes elgondolkodni az e tételben szereplő, valamint az összegzési tételben szereplő algoritmusok formai hasonlóságán. Rádöbbenhetünk, hogy ez az összegzési tétel egy változata. A különbség oka: a sorozat minden tagjának 1 az értéke, viszont az egyes tagokat nehezebb megadni:

$$\sum_{i=1}^{n} A(i) \quad \leftrightarrow \quad \sum_{\substack{i:\,A(i)\\ T\ \text{tul.}}} 1$$

Keresés rendezett sorozatban

Adott egy N elemű rendezett sorozat és egy keresett elem (X). Olyan algoritmus készítése a feladat, amely megadja, hogy szerepel-e a keresett elem a sorozatban, s ha igen, akkor megadja a sorszámát. Ez hasonló a korábbi keresési feladathoz. Most kihasználjuk, hogy a vizsgált sorozat rendezett. Ennek alapján bármely elemről el tudjuk dönteni, hogy

a keresett elem előtte vagy utána van-e vagy esetleg éppen megtaláltuk. Ez a logaritmikus keresés (ui. az összehasonlítások száma az elemszám logaritmusával arányos, ellentétben az előző kereséssel, ahol az összehasonlítások száma lineárisan függ az elemszámtól), Az eljárás A és F változói mindig annak a részintervallumnak az alsó és felső végpontjai, amelyben a keresett elem benne lehet.

```
Eljárás:
    A:=1 : F:=N
    Ciklus
        K:=INT((A+F)/2)
        Ha  A(K)<X  akkor A:=K+1
        Ha  A(K)>X  akkor F:=K-1
    amig A<=F és A(K)<>X
    Ciklus vége
    VAN_E:= A(K)=X
    Ha VAN_E akkor SORSZ:=K
Eljárás vége.
```

Megjegyzések:

– Általában igaz az, hogy a kiinduló sorozat tulajdonságait felhasználva az egyszerű (lineáris) keresésnél sokkal hatékonyabb algoritmus készíthető.

– Ha rendezett sorozaton mégis az elemi keresési módszert alkalmazzuk, akkor a kiválasztott elem speciális tulajdonsággal is rendelkezni fog (ez lesz a legkisebb, ill. a legnagyobb adott tulajdonságú elem).

Példa. Készítsünk algoritmust, amely ismerőseink lakóhelye (városa) alapján megad egy Sopronban lakót (lakóhely szerint sorbarendezett listánk van)!

Megfeleltetés:

N – az ismerőseink száma;
sorozat – (név, város),...(A$(N,2));
tulajdonság – A$(I,2)=„SOPRON”.

```
Ismerös cime:
    A:=1 : F:=N
    Ciklus
        K:=INT((A+F)/2)
        Ha  A$(K,2)<"SOPRON"  akkor A:=K+1
        Ha  A$(K,2)>"SOPRON"  akkor F:=K-1
    amig A<=F és A$(K,2)<>"SOPRON"
    Ciklus vége
    VAN_E:= A$(K,2)="SOPRON"
    Ha VAN_E akkor SORSZ:=K
Eljárás vége.
```

ÉRTÉK HOZZÁRENDELÉSE SOROZATOKHOZ

Az eddigi tételeknél (típusfeladatoknál) egyszerre a kiinduló sorozat egyetlen elemét kellett vizsgálni. Most olyanok következnek, ahol egyszerre több elemre is szükségünk lehet. Ezek közül sok átfogalmazható a korábbi feladatoknak megfelelőre.

Feladat: Döntsük el, hogy egy sorozat monoton növekvő-e!
1. variáció: Döntsük el, hogy egy sorozat minden eleme nagyobb vagy egyenlő-e, mint az azt megelőző! (Itt egyszerre mindig két elemet kell vizsgálni.)
2. variáció: Végezzük el a következő megfeleltetést:
elemszám
sorozat – (1.elem,2.elem),(2.elem,3.elem),...;
tulajdonság – az I. elempár 1. tagja nem nagyobb a 2.-nál.

Így már egy egyszerű eldöntési feladatot kaptunk.

Ebből a csoportból egyetlen feladattípussal ismerkedünk meg, amely nem vezethető vissza az előzőekre: ez a legnagyobb, ill. a legkisebb elem kiválasztásának feladata.

Maximumkiválasztás

Ebben a feladatban egy sorozat legnagyobb elemét kell megtalálni. Lényeges nehézséget okoz a korábbiakkal szemben az, hogy most egyszerre nem elég a sorozat egyetlen elemét vizsgálni, hiszen a legnagyobb az, amely mindegyiknél nagyobb, azaz mindegyikkel össze kell hasonlítani. A feladat megoldását úgy kapjuk, hogy visszavezetjük elemenkénti feldolgozásra. Ha ismerjük K elem közül a legnagyobbat és veszünk hozzá egy új elemet, akkor a maximum vagy az eddigi vagy pedig az új elem lesz. Induláskor az első elemet maximumnak tekintjük.

```
Eljárás:
    MAX:=1
    Ciklus I=2-töl N-ig
        Ha A(MAX)<A(I) akkor MAX:=I
    Ciklus vége
Eljárás vége.
```

Megjegyzés: Minimumkiválasztásnál csak a feltételes utasítás feltétele fordul meg (<helyett>lesz).

Ha a kérdést úgy tesszük fel, hogy mennyi a legnagyobb érték, akkor egy másik megoldást is kaphatunk:

```
Eljárás:
    ERTEK:=A(1)
    Ciklus I=2-töl N-ig
        Ha ERTEK<A(I) akkor ERTEK:=A(I)
    Ciklus vége
Eljárás vége.
```

Megtehetjük azt is, hogy az előbbi két eljárást egybeépítjük. Ez főleg akkor célszerű, ha egy érték kiszámítása sok időbe telik, bonyolultan végezhető el.
Példa. Készítsünk algoritmust, amely N ember magasságának ismeretében megadja a legnagyobb magasságot és a legmagasabb sorszámát.

Megfeleltetés:

sorozat – magasság1,magasság2,... (A(N)).

```
Maximum:
     MAX:=1 : ERTEK:=A(1)
     Ciklus I=2-töl N-ig
          Ha ERTEK<A(I) akkor MAX:=I : ERTEK:=A(I)
     Ciklus vége
Eljárás vége.
```

SOROZAT HOZZÁRENDELÉSE SOROZATHOZ

A következőkben áttérünk arra az esetre, amikor az eredmény is egy sorozat. Nem foglalkozunk olyan egyszerű példákkal, amikor a sorozat minden elemén (egymástól függetlenül) egy függvény értékét kell kiszámítani (pl. adjuk meg a sin(x) függvény értékét a [0,5] zárt intervallumban, 0.1 lépésközzel). Ide tartozik az az eset is, amikor a sorozat egyes elemeire más-más függvényt kell akalmazni.
Például $a_1, a_2, \ldots, a_n \rightarrow 1*a_1, 2*a_2, \ldots, n*a_n$

Kiválogatás.

Ebben a feladatban egy N elemű sorozat összes T tulajdonsággal rendelkező elemét kell meghatározni. Gyűjtsük a kiválogatott elemek sorszámait a B( ) vektorban!

```
Eljárás:
     J:=0
     Ciklus I=1-töl N-ig
          Ha A(I) T tulajdonságu akkor J:=J+1 : B(J):=I
     Ciklus vége
Eljárás vége.
```

Példa. Készítsünk algoritmust, amely N ember magasságának ismeretében megadja az X magasságúakat!

Megfeleltetés:

sorozat – magasság1,magasság2,... (A(N))
tulajdonság – A(I)=X

```
X-magasságuak(X):
     J:=0
     Ciklus I=1-töl N-ig
          Ha A(I)=X akkor J:=J+1 : B(J):=I
     Ciklus vége
Eljárás vége.
```

Megjegyzés: Ha az X érték meghatározását az előző típusalgoritmussal végezzük, akkor az így kapott program éppen a legmagasabb embereket válogatja ki.

Ezután olyan algoritmusokkal foglalkozunk, amelyek a sorozat elemeinek valamilyen permutálását végzik. Ilyen permutáció készítése a feladata pl. a rendezéseknek. Lényegük: a kapott adatokat növekvő vagy csökkenő sorrendbe kell rendezni.

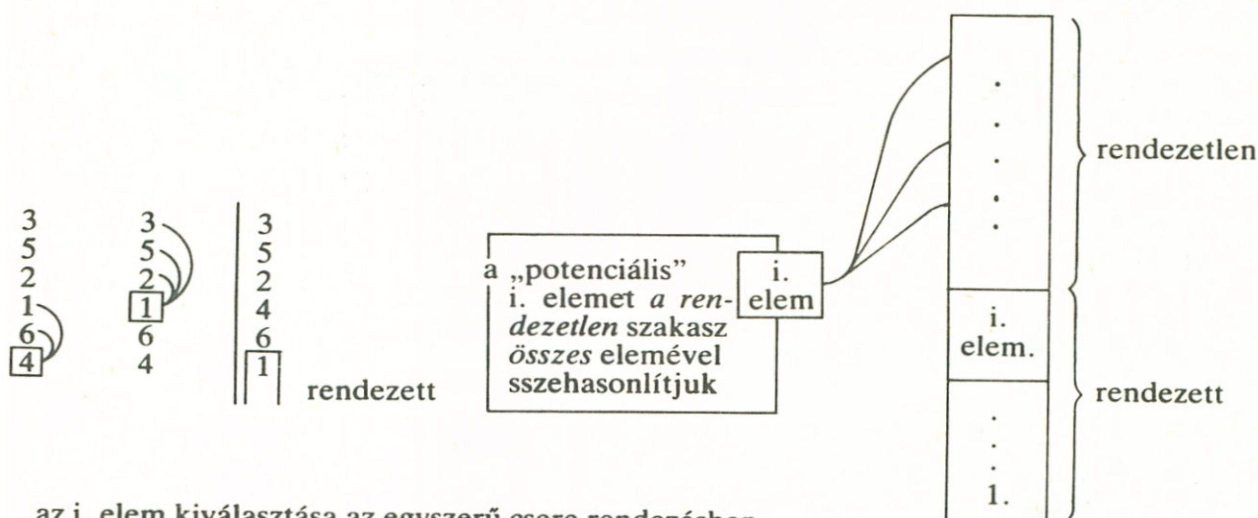
Rendezés

A rendezési feladatok megoldásához mindig két elvet használunk: egy adott tulajdonságú permutáció előállítását, valamint az indukációt. Többféle rendezési módszert ismertetünk:

– minimumkiválasztásos rendezés;
– buborékos rendezés;
– beillesztéses rendezés.

1. rendezés: Állítsunk elő olyan sorozatot, amely az eredeti sorozat elemeiből áll, de a legkisebb elem a legelső helyre került. Az indukciós lépés: a sorozat I.–N. elemeit permutáljuk úgy, hogy közülük a legkisebb kerüljön az I. helyre. Így az első lépésben a helyére kerül a legkisebb, az I. lépésben az I., s ha I értéke N–1 lesz, akkor a feladatot megoldottuk (7. ábra).

```
Minimumkiválasztásos rendezés(N,A(N)):
    Ciklus I=1-töl N-1-ig
          [ az elsö I-1 szám rendezett]
       Az I. számot az I. helyre
          [ az elsö I szám rendezett]
    Ciklus vége
Eljárás vége.
```



az i. elem kiválasztása az egyszerű csere rendezésben, 7. ábra

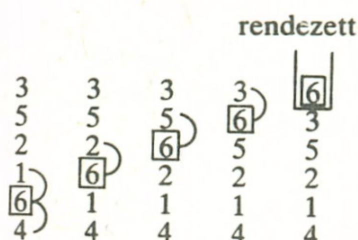
A permutáció előállítását minimumkiválasztással végezzük.

```
Az I. számot az I. helyre:
    L:=I
    Ciklus J=I+1-töl N-ig
        Ha A(L)>A(J) akkor L:=J
    Ciklus vége
    Csere (A(L),A(I))
Eljárás vége.
```

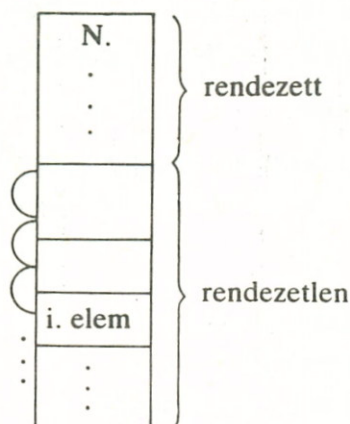
2. rendezés: Állítsunk elő olyan sorozatot, amely az eredeti sorozat elemeiből áll, de minden szomszédos párt cseréljünk fel az 1. és az N. között, ha eredetileg fordított sorrendben voltak! Igy a legnagyobb elem az utolsó helyre kerül. Az indukciós lépés: a sorozat 1.–I. elemeit cseréljük fel az előbbi módszer szerint. Tehát először az N. kerül a helyére, az I. vizsgálatánál az I., s legvégül a 2. (az első ekkor már automatikusan a helyén lesz). Előfordulhat, hogy ez a módszer egy lépésben nem csak a legnagyobbat teszi a helyére, hanem még néhányat a megelőzők közül is. Az utolsó csere helye határozza meg azt, hogy ezek az elemek hányan vannak (8. ábra).

```
Buborékos rendezés(N,A(N)):
    I:=N
    Ciklus amig I>1
              [ az I+1.-töl rendezett a sorozat]
         Az elsö I elem permutálása
              [ a CS+1.-töl rendezett a sorozat]
         I:=CS
    Ciklus vége
Eljárás vége.

Az elsö I elem permutálása:
    CS:=0
    Ciklus J=1-töl I-1-ig
         Ha A(J)>A(J+1) akkor Csere (A(J),A(J+1)) : CS:=J
    Ciklus vége
Eljárás vége.
```



a nagyság szerinti i. elem helyet cserél aktuális *szomszédjával, ha* az a rendezés szerint *hátrább való*

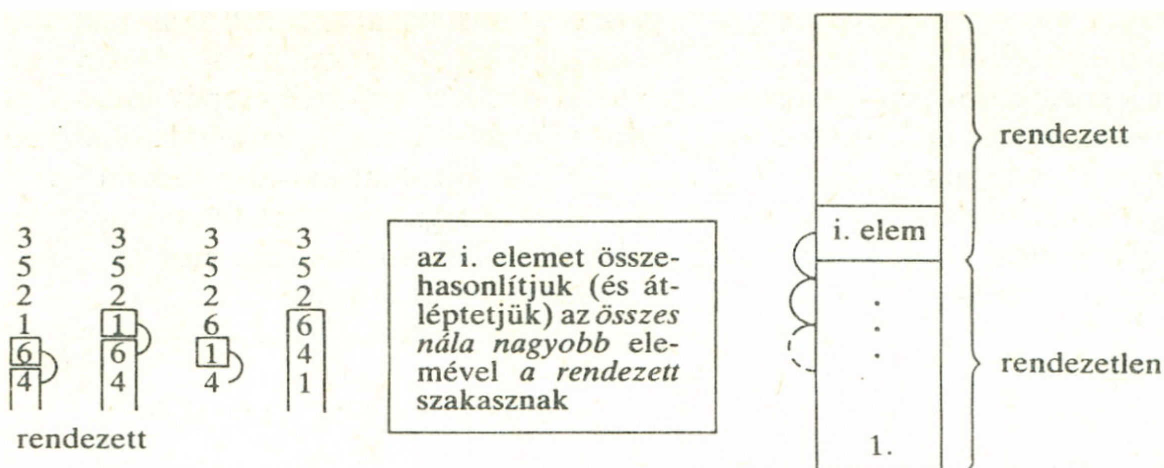


az i. elem „felszállása" a „buborékos" rendezésben

8. ábra

3. rendezés: Olyan sorozatot állítsunk elő, amely az eredeti sorozat elemeiből áll, de az első két elem sorrendje helyes. Az indukciós lépés: a sorozat elemeit úgy cseréljük fel, hogy az első I elem sorrendje legyen helyes. Igy I értékét változtatva 2-től N-ig, a teljes sorozatot rendezzük (9. ábra).

```
Beillesztés:
    Ciklus I=2-töl N-ig
              [ az elsö I-1 sorrendje jó]
         Az I. beillesztése
              [ az elsö I sorrendje jó]
    Ciklus vége
Eljárás vége.
```



az i. elem beillesztése a 3. rendezés szerint

9. ábra

A beillesztés azt jelenti, hogy az I. elemet addig kell előre hozni az első I–1 között, amíg a helyére nem kerül.

```
Az I. beillesztése:
          J:=I-1 : X:=A(I)
          Ciklus amig J>0 és A(J)>X
                    A(J+1):=A(J)
                    J:=J-1
          Ciklus vége
          A(J+1):=X
Eljárás vége.
```

Újabb feladatunk nem rendezi sorba a sorozat elemeit, csupán egy részfeladatot végez el, mégis ez az alapja az egyik leggyorsabb rendezési módszernek.

Szétválogatás

Rendelkezésre áll egy sorozat, valamint egy kijelölt eleme (pl. X=A(1)). Cseréljük fel úgy a sorozat elemeit, hogy az X-nél kisebbek X előtt legyenek, a nála nagyobbak pedig utána.

```
Eljárás(A(N)):
          X:=A(1)
          I:=1 : J:=N
          Ciklus amig I<J
                    Ciklus amig I<J és A(J)>=X
                              J:=J-1
                    Ciklus vége
                    Ha I<J akkor A(I):=A(J) : I:=I+1
                    Ciklus amig I<J és A(I)<=X
                              I:=I+1
                    Ciklus vége
                    Ha I<J akkor A(J):=A(I) : J:=J-1
          Ciklus vége
          A(I):=X
Eljárás vége.
```

Gondoljuk meg, hogy e szétválogatási eljárás alapján hogyan írhatnánk egy rendező algoritmust! Ebből arra is fény derül, hogy miért oly hatékony e rendezési módszer.

SOROZATOKHOZ HOZZÁRENDELT SOROZATOK

Egy olyan feladatosztályra térünk át, ahol több sorozatból kell készíteni egy újabb sorozatot. Ezek általában visszavezethetők korábbi feladatokra, néhánnyal mégis érdemes külön is foglalkozni. A legegyszerűbbek közé tartoznak pl. olyanok, amelyek két sorozatból indulnak ki, ezeket halmazoknak tekintik, s valamilyen halmazműveletet végeznek rajtuk (egyesítés, metszet, ...). Ezek általában kiválogatási, eldöntési feladatokra vezethetők vissza.

Bizonyos esetekben kicsit bonyolultabb a helyzet, pl. ha az elemek sorrendje nem lehet tetszőleges.

Összefuttatás

Adottak két rendezett sorozat elemei. Állítsunk elő belőlük egy sorozatot úgy, hogy az eredeti sorozatok minden eleme szerepeljen benne, de amelyek mindkettőben benn voltak, azok csak egyszer, s ez a sorozat is rendezett legyen! (Például növekvő sorrendű sorozatokat vizsgálunk.)
A két sorozat: A(N),B(M). Az eredmény C(N+M).

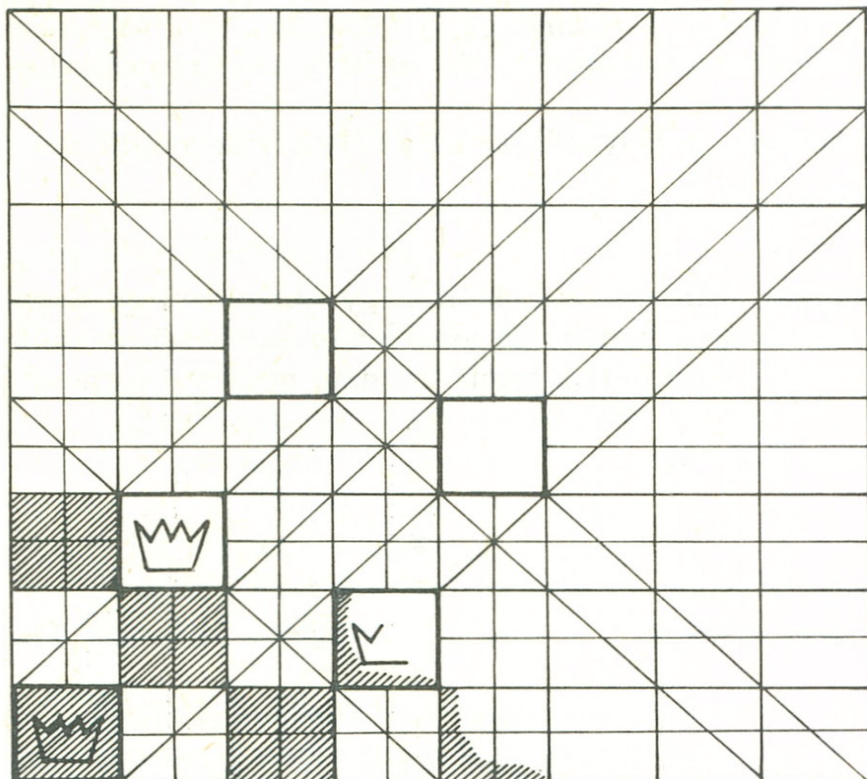
```
Eljárás:
     I:=1 : J:=1 : K:=0
     Ciklus amig I<=N és J<=M
          K:=K+1
          Elágazás
               A(I)<B(J) esetén C(K):=A(I) : I:=I+1
               A(I)=B(J) esetén C(K):=A(I) : I:=I+1 :
                                                  J:=J+1
               A(I)>B(J) esetén C(K):=B(J) : J:=J+1
          Elágazás vége
     Ciklus vége
     Ciklus amig I<=N
          K:=K+1
          C(K):=A(I) : I:=I+1
     Ciklus vége
     Ciklus amig J<=M
          K:=K+1
          C(K):=B(J) : J:=J+1
     Ciklus vége
Eljárás vége.
```

Visszalépéses keresés (backtrack)

Az utolsó feladattal a problémamegoldás egy igen széles területén alkalmazható algoritmust mutatunk be, amelynek lényege a feladat megoldásának megközelítése *rendszeres próbálgatásokkal*. Sok esetben ennél jobb módszert nem is követhetünk! A következőkben az általános algoritmusát adjuk meg. Célszerű azonban megértéséhez egy konkrét

feladattal párhuzamba állítva is meggondolni. Legyen pl. ez a klasszikus 8 vezér probléma! A 10. ábrán egy *zsákutcába* jutott vezérlehelyezési kísérlet látható.

Adott N sorozat, amelyek rendre M(1), M(2), ... elemszámúak. Ki kell választani mindegyikből egy-egy elemet úgy, hogy egyes sorozatokból való választások másokat befolyásolnak (pl. nem lehet két sorozatból azonos sorszámú elemet választani). Tulajdonképpen ez egy bonyolult keresési feladat: egy adott tulajdonsággal rendelkező szám N-est kell keresni. A megoldást úgy készítjük el, hogy ne kelljen az összes lehetőséget végignézni!



10. ábra

Először megpróbálunk az első sorozatból választani egy elemet, ezután a következőből, s ezt mindaddig csináljuk, amíg a választás lehetséges. Amikor áttérünk a következő sorozatra, akkor jeleznünk kell, hogy ebből még nem próbáltunk elemet választani. X(I) jelöli az I. sorozat kiválasztott eleme sorszámát, ha még nem választottunk, akkor értéke 0 lesz. Ha nincs jó választás, akkor visszalépünk az előző sorozathoz, s megpróbálunk abból egy másik elemet választani. Ez az egész eljárás vagy úgy ér véget, hogy minden sorozatból sikerült választani (ekkor megkaptuk a megoldást), vagy pedig úgy, hogy a visszalépések sokasága után már az első sorozatból sem lehet újabb elemet választani (ekkor a feladatnak nincs megoldása).

A megoldás – mint a korábbi tételeknél láttuk – egyszerűbbé válik, ha tudjuk, hogy van megfelelő tulajdonságú elemsorozat. Most azonban kövessük az általános eset megoldását!

```
Eljárás:
    I:=1
    Ciklus amig I>=1 és I<=N
        Ha VAN_JÓ_ESET(I) akkor I:=I+1 : X(I):=0
                          különben I:=I-1
    Ciklus vége
    VAN_E:= I>N
Eljárás vége.
```

Az I. sorozatból úgy választunk elemet, hogy próbálunk mindaddig új elemet venni, amíg egyáltalán van további elem, és az éppen vizsgáltat nem lehet választani. Ha a keresgélés közben a sorozat elemei nem fogytak el, akkor az előző szintnek válaszolhatjuk azt, hogy sikeres volt a választás. Ha pedig az utolsó sem felelt meg, akkor azt, hogy vissza kell lépni az előző sorozathoz.

```
VAN_JÓ_ESET(I):
    Ciklus
        X(I):=X(I)+1
    amig X(I)<=M(I) és ROSSZ_ESET(I,X(I))
    Ciklus vége
    VAN_JÓ_ESET:= X(I)<=M(I)
Eljárás vége.
```

Rossz választásnak nevezzük azt, amelyet a korábbi választások közül valamelyik megakadályoz. (Például a következő 8 vezér feladatban egy vezért nem tehetünk olyan helyre, ahol a korábban elhelyezett vezérek ütnék.)

```
ROSSZ_ESET(I,X(I)):
    J:=1
    Ciklus amig J<I és (J,X(J)) nem zárja ki (I,X(I))-t
        J:=J+1
    Ciklus vége
    ROSSZ_ESET:= J<I
Eljárás vége.
```

Megjegyzés: A backtrack algoritmusra alapozva a keresési feladatokhoz hasonlóan el lehet készíteni eldöntési, kiválasztási, megszámolási, kiválogatási feladatok megoldását, sőt még a valamilyen szempontból legjobb megoldás elkészítését is.
Példa. Készítsünk algoritmust, amely elhelyez egy sakktáblán 8 vezért úgy, hogy egyik sem üti a másikat!

Megfeleltetés:

N – 8;
sorozatok – az egyes vezérek lehetséges elhelyezései;
M(1)=M(2)=...=N
(J,X(J)) nem zárja ki (I,X(I))-t:
X(I) <> X(J) és abs(X(I)–X(J))<>I–J
(azaz a J. vezér nincs egy sorban vagy átlóban az I. vezérrel)

A megoldás:

```
8 vezér elhelyezése:
    I:=1
    Ciklus amig I>=1 és I<=N
        Ha VAN_Jò_ESET(I)  akkor I:=I+1 : X(I):=0
                           különben I:=I-1
    Ciklus vége
    VAN_E:= I>N
Eljàràs vége.

VAN_Jò_ESET(I):
    Ciklus
        X(I):=X(I)+1
    amig X(I)<=N és ROSSZ_ESET(I,X(I))
    Ciklus vége
    VAN_Jò_ESET:= X(I)<=N
Eljàràs vége.

ROSSZ_ESET(I,X(I)):
    J:=1
    Ciklus amig J<I és X(I)<>X(J) és ABS(X(I)-X(J))<>I-J
        J:=J+1
    Ciklus vége
Eljàràs vége.
```

HARMADIK PÉLDA A MÓDSZERES PROGRAMOZÁSRA
(a tételek alkalmazhatósága)

Feladat: Írjuk ki a 2 és N közötti összes primszámot!

A feladat megoldására próbáljuk alkalmazni az előző fejezetben megismert programozási tételeket! A legfelső szint egy *kiválogatási* feladat, hiszen egy számhalmaz valamilyen tulajdonságú elemeit kell megadnunk. A számhalmaz elemei közvetlenül meghatározhatók, így csupán a legnagyobb elem értékére van szükségünk.

Így a megoldás legfelső szintje a következő lesz:

```
Program:
    Be: N            [N>2 és egész]
    Ciklus I=2-töl N-ig
        Ha I primszám akkor Ki: I
    Ciklus vége
Program vége.
```

A megoldandó részfeladat: Egy számról el kell döntenünk, hogy prím-e! Fogalmazzuk át a feladatot: el kell döntenünk, hogy az I számnak 2 és I/2 között van-e osztója! Ez most egy *eldöntési* feladat, így megoldása:

```
I primszám:
    J:=2
    Ciklus amig J<=I/2 és J nem osztója I-nek
        J:=J+1
    Ciklus vége
    I primszám := J=I
Eljárás vége.
```

Már készen is vagyunk a feladat megoldásával. El kell ismernünk, hogy ez bizony nem a lehető legoptimálisabb megoldás. Ahhoz, hogy hatékonyabb legyen a megoldás, a ciklusfeltételben szereplő I/2-t – matematikai ismereteink alapján – ki kell cserélni $\sqrt{I}$-re. A hatékonysággal egyébként még egy későbbi fejezetben foglalkozunk.

Próbáljunk egy másik megoldást készíteni! Bár a programozási tételek sokat segítenek a feladatok megoldásában, ne feledkezzünk meg a józan észről sem! Most is a feladat átfogalmazásával láthatjuk meg, hogyan lehet újabb megoldást találni.

Vegyük a 2 és N közötti összes számot! Hagyjuk el közülük azokat, amelyek nem primek, majd írjuk ki a maradékot! Olyan halmazt kell ábrázolnunk, amelynek elemeit nem tudjuk egyszerűen meghatározni (a legvégén a prímszámok maradnak benne). Így tárolásához egy vektort használunk, ennek nem 0 értékű elemei tartoznak a halmazhoz.

```
Program:
    Be: N            [N>2 és egész]
    Ciklus I=2-töl N-ig
        A(I):=I            [a lehetséges primek feltöltése]
    Ciklus vége
    Nem primek lehagyása
    Ciklus I=2-töl N-ig
        Ha A(I)>0 akkor Ki: I
    Ciklus vége
Program vége.
```

A nem prímszámok halmazát osszuk fel részekre: ilyenek a 2-vel osztható számok a 2 kivételével, a 3-mal oszthatók a 3 kivételével ... A megoldás ezt a felosztást fogja követni: először a 2-vel oszthatóakat hagyjuk el ...

```
Nem primek elhagyása:
    Ciklus J=2-töl N/2-ig
        A J-vel oszthatòak elhagyása
    Ciklus vége
Eljárás vége.
```

A J-vel oszthatók elhagyása egy *kiválogatási* feladat. Gyorsabb megoldás azonban, ha kiszámoljuk, hogy melyik szám az első J-vel osztható, s mindegyik után megadjuk a következőt is:

```
A J-vel oszthatòak elhagyása:
    Ha A(J)>0 akkor
        Ciklus K=2*J-töl N-ig J-esével
            A(K):=0
        Ciklus vége
    Elágazás vége
Eljárás vége.
```

Ez volt az Eratosztenészi szita módszere.

Ezután egy feladaton keresztül bemutatjuk új tételek létrehozását.

Feladat: Adott egy N elemű sorozat, adjuk meg a különböző elemeit!

A tételünk azt mondja ki, hogy a *különbözők kiválogatásának* a következő algoritmus egy helyes megoldása.

Ez tulajdonképpen egy *kiválogatási* feladat, egy sorozat adott tulajdonságú elemeit kell meghatároznunk.

A tulajdonság megállapítása pedig *eldöntési* feladat: akkor rendelkezik egy elem az előbbi tulajdonsággal, ha nem szerepel a már kiválogatottak között.

Tartalmazza az A( ) vektor az eredeti sorozat elemeit, a B( ) vektor a kiválogatottak sorszámait, J pedig legyen a kiválogatottak száma.

```
Eljárás:
    J:=0
    Ciklus I=1-töl N-ig
        K:=1
        Ciklus amig K<=J és A(B(K))<>A(I)
            K:=K+1
        Ciklus vége
        Ha K>J akkor J:=J+1 : B(J):=I
    Ciklus vége
Eljárás vége.
```

A következő feladat azt szemlélteti, hogy meglevő típusalgoritmusok alkalmas módosításával hogyan tudunk egy bonyolultnak látszó feladatot egyszerűen megoldani.

Feladat: Adott N ember és N állás. Mindegyik ember megadja, hogy mely állásokat tudná betölteni. Készítsünk programot, amely eldönti, hogy betölthető-e velük minden állás, s ha igen, akkor megad egy ember–állás hozzárendelést.

A feladat figyelmes végigolvasása után kitalálható, hogy itt a visszalépéses keresés tipikus alkalmazásáról lesz szó. Ehhez a következő adatmegfeleltetés lenne kényelmes:

M(I) – az I. ember által betölthető állások száma;
A(I,J) – az I. ember által betölthető J. állás sorszáma;
X(I) – az I. ember által választott munka sorszáma (ez az eredmény).

Ekkor a „(J,X(J)) nem zárja ki (I,X(I))-t" a következőképpen értelmezzük:

A(I,X(I))<>A(J,X(J)),

vagyis kétszer ugyanaz az állás nem tölthető be.

Más szempontból azonban ez sokszor nem szerencsés adatmegadás. Sokkal természetesebb a következő:

A(I,J)= 1, ha az I. ember a J. munkát vállalja,
0, egyébként.

Ebben az esetben sajnos nem tudjuk egyszerűen alkalmazni a visszalépéses keresést, némi módosításra van szükség. Egészítsük ki az M(1),...M(N) elemű sorozatokat N elemű-ekre! Mindegyik sorozatban benne lesz az összes állás, majd külön jelöljük, hogy az illető melyeket tudná betölteni. A mátrix egyes elemei 0 vagy 1 értékűek. Az algoritmus csak azokat az állásokat veszi figyelembe minden egyes embernél, amelyeket az illető be tud tölteni. A megoldás 2. és 3. szintje fog változni:

```
VAN_JÓ_ESET(I):
         Ciklus
                  X(I):=X(I)+1
         amig X(I)<N és
                        (A(I,X(I)=0 vagy ROSSZ_ESET(I,X(I)) )
         Ciklus vége
         VAN_JÓ_ESET:= X(I)<=N
Eljárás vége.
```

Egy munkaválasztás akkor rossz, ha azt a munkát valaki már vállalta:

```
ROSSZ_ESET(I,X(I)):
         J:=1
         Ciklus amig J<I és X(J)<>X(I)
                  J:=J+1
         Ciklus vége
         ROSSZ_ESET:= J<I
Eljárás vége.
```

A PROGRAM HELYESSÉGE

A programírás körüli bonyodalmakkal összemérhető nagyságrendű az elkészült program helyességének vizsgálata. Sőt ezen a területen olykor nagy a zűrzavar, sokan hajlamosak ezeket a kérdéseket félvállról venni.

Egy program helyességéről csak akkor beszélhetünk, ha valamilyen módszerrel meggyőződtünk róla.

A program helyességének belátásához a következő módszerek használatosak:

1. Programigazolás (vagy tesztelés): próbafeladatok lefuttatásával a program megbízhatóságát vizsgáljuk. Itt a programhiba felfedezése a cél.
2. Programérvényesítés: a feladatmeghatározás alapján a program logikai helyességének igazolása valódi környezetben. Tehát gyakorlati feladatokra használjuk a programot. A cél annak belátása, hogy megoldásukra a program alkalmas-e.
3. Programhitelesítés: a program helyességének és a feladatmeghatározás előírásainak jóváhagyása a hosszú idő alatt összegyűlt futtatási tapasztalatok alapján.
4. Hibakeresés: az előző módszerek egyikével megtalált hiba helyének, okának a megtalálása.
5. Programhelyesség-bizonyítás: a program logikai helyességének kimutatása axiómák, következtetési szabályok segítségével, a feladatmeghatározás alapján.

Az előbbiekből három módszerrel részletesen foglalkozunk, a másik kettőre csak néhány gondolattal utalunk. Éspedig:

A programérvényesítés a programnak egy vagy néhány gyakorlati feladatra való alkalmazását jelenti. Ennek során a feladat kitűzője meggyőződhet arról, hogy a program teljesíti-e elvárásait. Ebbe beletartoznak a helyesség, a hatékonyság, a formai szempontok és egyéb követelmények.

A programhitelesítés lényege a program végleges formában való rögzítése. A használat során szerzett tapasztalatok alapján pontosítható a megoldott feladatok köre, a kezelés, a program által nyújtott szolgáltatások, az alkalmazás környezeti feltételei stb.

TESZTELÉS

A tesztelés célja, hogy minél több hibát megtaláljunk a programban. Ahhoz, hogy az összes hibát megtaláljuk, nem kell próbát tenni az összes lehetséges bemenő adattal. Nézzünk egy egyszerű példát:

69 lista

```
10 INPUT I%,J%
20 PRINT I%,J%
```

Például a HT–1080Z-n 2^{16} különböző értékű egész számot tudunk tárolni, ezért az összes lehetőség kipróbálásához 2^{32} számot kellene behelyettesíteni. Ez nyilvánvalóan fölösleges. A másik végletet a következő – két szám osztására készült – programrészleten mutatjuk be:

70 lista

```
10 INPUT A,B
20 PRINT A*B : REM A/B KIIRASA
```

Amennyiben ezt a programot olyan bemenő adatokkal próbáljuk ki, amelyben A=0 vagy B=1, akkor a program helyesen működik, a hibát nem tudjuk felfedezni. Természetesen az a célunk, hogy a tesztelést olyan módszerrel hajtsuk végre, amellyel a próbák száma erősen lecsökkenthető. Tesztesetnek a be- és kimenő adatok és feltételek együttes megadását nevezzük. Fogalmazzuk meg a tesztelés alapelveit:

1. A jó teszteset az, ami várhatóan egy még felfedezetlen hibát mutat ki a programban. Például két szám legnagyobb közös osztóját számoló programot az [5,5] adatpár után a [6,6]-tal teljesen felesleges kipróbálni.
2. A teszteset nemcsak bemenő adatokból, hanem a hozzájuk tartozó eredményekből is áll. Egyébként nem tudnánk a kapott eredmény helyes vagy hibás voltáról beszélni. A későbbi felhasználás alatt célszerű a teszteseteket is leírni.
3. A meg nem ismételhető tesztesetek kerülendők, feleslegesen megnövelik a programtesztelés költségeit, idejét. Nem is beszélve arról a bosszúságról, amikor a programunk egy hibás futását nem tudjuk megismételni, s így a hiba is felfedetlen marad.
4. Teszteseteket mind az érvénytelen, mind az érvényes adatokra kell készíteni.
5. Minden tesztesetből a lehető legtöbbet kell hasznosítani, azaz minden teszteset eredményét alaposan végig kell vizsgálni. Ezzel jelentősen csökkenthető a szükséges próbák száma.
6. Egy próba eredményeinek vizsgálata során egyaránt fontos megállapítani, hogy miért nem valósít meg a program valamilyen funkciót, amit elvárunk tőle, ill., hogy miért végez olyan tevékenységeket is, amelyeket nem feltételeztünk róla.
7. A program tesztelését csak a program írójától különböző személy tudja hatékonyan ellátni. A tesztelés ui. nem egy jóindulatú tevékenység. Saját munkáját pedig mindenki jónak feltételezi.

A programtesztelés módszereit két csoportba oszthatjuk aszerint, hogy a tesztelés során végrehajtjuk-e a programot vagy nem. Ha csak a program kódját vizsgáljuk, akkor statikus, ha a programot végre is hajtjuk a tesztelés során, akkor *dinamikus* tesztelésről beszélünk.

STATIKUS TESZTELÉSI MÓDSZEREK

1. Kódellenőrzés

A kódellenőrzés a program szövegének megvizsgálását jelenti. Az algoritmus logikáját kell ekkor a programban végigkövetni, és megfigyelni, hogy a kettő egyező-e. Csupán a kód alapján is viszonylag könnyen tud hibákat felfedezni egy kívülálló. Sokszor a programozó maga veszi észre a hibákat, miközben valakinek részletesen magyarázza.

2. Formai ellenőrzés, keresztivatkozási táblázatok készítése

Egy programban az előforduló hibákat két csoportra oszthatjuk: formai (szintaktikai), ill. tartalmi (szemantikai) jellegű hibákra. Például az ABC80-as számítógép a formai ellenőrzést a sor beírásakor elvégzi, s ha hibát talál, akkor ezt jelzi, és a beírt sort nem fogadja el. A HT–1080Z számítógép esetén nem beíráskor történik meg ez az ellenőrzés, hanem csak a program végrehajtása közben. Ekkor a formai ellenőrzést megfelelően választott tesztadatokkal való kipróbálással lehet elérni. A program minden utasítását végre kell hajtani legalább egyszer. Igen sok információt ad egy programról, ha különböző kereszthivatkozási táblázatot készítünk róla (kereszreferencia). Ennek egyik típusa a változókról készült táblázat, amelynek felépítése pl. a következő lehet:

Változónév	Típus	Sorszámlista

A sorszámlistában azok a sorok szerepelnek, amelyekben az adott változó előfordul. A sorszám előtt speciális jellel jelölhetjük, hogy a változó értéket kapott az adott sorban.

Változónév	Típus	Sorszámlista
ALFA	VALÓS	*100,110,520
G	EGÉSZ	190,*530

Természetesen kézzel készíteni ilyen táblázatot nagyon munkaigényes feladat, sok esetben azonban a számítógép biztosítja automatikusan ezt a lehetőséget.

3. Tartalmi ellenőrzés, ellentmondás-keresés

A formai hibáknál sokkal nehezebb felfedni a programban a tartalmi hibákat. Ilyen hiba lehet az, hogy egy változónak értéket adunk, de ezután nem használjuk semmire, vagy közvetlenül utána még egyszer értéket kap, pl.:

```
100 I=1
110 I=5
```

Ellentmondás kereséssel a program vezérlési folyamában is fedezhetünk fel hibákat. Például egy utasításhoz a program soha nem jut el:

```
100 GOTO 200
110 I=1
```

és nincs a programban vezérlésátadás a 110-es utasításra. Bonyolultabb dolog a következő végtelen ciklust felfedezni.

```
100 GOTO 300
.
200 GOTO 100
.
300 GOTO 200
```

DINAMIKUS TESZTELÉSI MÓDSZEREK

Az előző részben a statikus tesztelési módszereket vizsgáltuk, ahol a tesztelést a program végrehajtása nélkül, a program szövegének vizsgálatával végeztük. A dinamikus tesztelési módszerek alapelve éppen az, hogy a programot működés közben vizsgáljuk. Teszteseteket kétféle módon tudunk választani. Egy lehetőség az ún. *fekete doboz módszer,* más néven adatvezérelt tesztelés. E módszer alkalmazásakor a tesztelő nem veszi figyelembe a program belső szerkezetét, pontosabban nem azt tekinti elsődleges szempontnak, hanem a teszteseteket a feladatmeghatározás alapján választja meg.

A cél természetesen a lehető leghatékonyabb tesztelés elvégzése, azaz az összes hiba megtalálása a programban. Ez ugyan elvileg lehetséges, ha a programot az összes lehetséges bemenő adatra kipróbáljuk (kimerítő bemenet tesztelés). Ezzel a módszerrel azonban, mint korábban láttuk, mennyiségi akadályok merülnek fel.

Egy másik lehetőség a *fehér doboz módszer* (logikavezérelt tesztelés). Ebben a módszerben a tesztesetek megválasztásánál lehetőség van a program belső szerkezetének figyelembevételére is.

A cél a program minél alaposabb tesztelése. Erre jó módszer a programban az összes lehetséges utat, azaz tesztesetet végigjárni, amennyiben ez megtehető (kimerítő út tesztelés). Ám még viszonylag kis programok esetén is igen nagy lehet a tesztelési utak száma. Gondoljunk a ciklusokra! Sőt ezzel a módszerrel a hiányzó utakat nem lehet felderíteni.

Mivel sem a fehér doboz módszerrel, sem a fekete doboz módszerrel nem lehetséges a kimerítő tesztelés, el kell fogadnunk, hogy nem tudjuk egyetlen program hibamentességét sem szavatolni. A további cél ezek után az összes lehetséges teszteset halmazából a lehető leghatékonyabb tesztesetcsoport kiválasztása lehet.

Fekete doboz módszerek

1. Ekvivalenciaosztályok keresése

A tesztelés alapelveinek ismertetésénél jó tesztesetnek neveztük azt, amelyre minél nagyobb valószínűséggel áll, hogy hibát találunk vele a programban. Másrészt megállapítottuk, hogy a kimerítő bemeneti tesztelés gyakorlatilag megvalósíthatatlan, így meg kell elégednünk a bemenő adatok egy szűk részhalmazával való teszteléssel. Ezek után azért, hogy ez a részhalmaz minél hatásosabb legyen, a benne szereplő tesztesetekre teljesüljenek a következők:

– Minden tesztesetnek annyi bemeneti feltételt (a bemenő adatokat leíró állítást) kell kielégítenie, amennyit csak lehetséges, hogy ezzel a szükséges tesztesetek számát csökkentsük. Például a csillagsűrűsödéses feladatban egyszerre próbáljunk a csillagszámra, a távolságra, a csoportlétszámra, valamint a csoportok térbeli elhelyezkedésére vonatkozó tesztadatokat adni.

– A bemeneti tartományt valamilyen módon részekre kell osztani, és ezekre a részekre jellemző teszteseteket kell választani. Ezekre a részekre (osztályokra) legyen igaz

a következő: ha egy ilyen osztályból választunk egy tesztesetet, és ezzel hibát találunk a programban, akkor az osztály más elemét választva is nagy valószínűséggel hibát találnánk a programban. Hasonlóan: ha a kiválasztott tesztesetre a program jól működik, az osztály másik elemét választva is nagy valószínűséggel helyes eredményt adna.

Ezeket az elveket veszi figyelembe az ekvivalenciaosztályok módszere.

Megjegyzés: Ekvivalenciaosztályokat nem csak az érvényes, hanem az érvénytelen adatokhoz is létre kell hozni, és a programot azokkal is kipróbálni.

Néhány jó tanács az ekvivalenciaosztályok megtalálásához:

– Ha a bemeneti feltétel értéktartományt definiál, az érvényes ekvivalenciaosztály legyen a megengedett bemeneti értékek halmaza, az érvénytelen ekvivalenciaosztályok pedig az alsó és a felső határoló tartomány.

Például ha az adatok osztályzatok (értékük 1 és 5 között van), akkor ezek az ekvivalenciaosztályok rendre: 1 <= i <= 5 és i < 1 és i > 5.

– Ha a bemeneti feltétel értékek számát határozza meg, akkor az előzőhöz hasonlóan járjunk el. Például ha be kell olvasnunk legfeljebb 6 karaktert, akkor az érvényes ekvivalenciaosztály 0–6 karakter beolvasása, az érvénytelen ekvivalenciaosztály 6-nál több karakter beolvasása. (0-nál kevesebb nem fordulhat elő.)

– Ha a bemenet feltétele azt mondja ki, hogy a bemenő adatnak valamilyen meghatározott jellemzővel kell rendelkezni, akkor két ekvivalenciaosztályt kell felvenni: egy érvényeset és egy érvénytelent.

– Ha feltételezhetjük, hogy a program valamelyik ekvivalenciaosztályba eső elemeket különféleképpen kezel, akkor a feltételezésnek megfelelően bontsuk az ekvivalenciaosztályt további osztályokra.

Ha már rendelkezésünkre állnak az ekvivalenciaosztályok, akkor a teszteseteket a következő két elv alapján határozhatjuk meg:

– Amíg az érvényes ekvivalenciaosztályokat le nem fedtük, addig készítsünk olyan teszteseteket, amelyek minél több érvényes ekvivalenciaosztályt lefednek!

Több hiba esetén előfordulhat, hogy a hibás adatok lefedik egymást, a második hiba kijelzésére az első hibajelzés miatt már nem kerül sor.

– Minden érvénytelen ekvivalenciaosztályra írjunk egy-egy, az osztályt lefedő tesztesetet.

2. Határeset-elemzés

A határeset-elemzés két dologban különbözik az ekvivalenciaosztályok keresésének módszerétől.

– Az ekvivalenciaosztály kiválasztott elemének a határon levő elemeket választja.

– Nem csak a bemeneti, hanem a kimeneti ekvivalenciaosztályt is figyelembe veszi.

Felsorolunk néhány szempontot a határeset-elemzéshez:

– Ha a bemeneti feltétel egy értéktartományt jelöl meg, írjunk teszteseteket az érvényes tartomány alsó és felső határára és az érvénytelen tartománynak a határ közelébe eső elemére! Például ha a bemeneti tartomány a (0,1) nyílt intervallum, akkor a 0, 1, 0.01, 0.99 értékekre érdemes kipróbálni a programot.

– Ha egy bemeneti feltétel értékek számosságát adja meg, akkor hasonlóan járjunk el, mint az előző esetben. Például ha rendeznünk kell 1...128 nevet, akkor célszerű a programot kipróbálni 0, 1, 128, 129 névvel.

– Használjuk az előbbi feltételeket az összes kimeneti osztályra is.

Fehér doboz módszerek

1. Utasítások egyszeri lefedésének elve

A módszer lényege olyan tesztesetek kiválasztása, amelyek alapján minden utasítást legalább egyszer végrehajthatunk a programban. Bár ez sokszor jó módszer, de nem tökéletes, nézzük meg ui. a következő egyszerű példát:

Ha X > 0 akkor ki: X

Ebben a példában egyetlen próbával elérhetjük az összes utasítás végrehajtását (pl. X=1), de ezzel a próbával nem derülne ki az, ha az X>0 feltétel helyett az X>=0 szerepelne, azaz a program hibás lenne.

2. Döntéslefedés elve

Itt az előzőnél egy kicsit erősebb követelményt alkalmazunk. A programban minden egyes elágazás igaz, ill. hamis ágát legalább egyszer be kell járni a tesztelés során. A döntéslefedés elvét figyelembe véve eleget teszünk az utasításlefedés követelményének is. Itt is maradnak azonban problémák. Nézzünk egy példát:

Ha X>0 vagy Y>0 akkor ki: X*Y

Ebben az esetben az (X=1, Y=1) és az (X=–1, Y=–1) tesztesetek lefedik a döntéseket, de nem vennénk észre velük azt, ha a második feltételt (Y>0) rosszul írtuk (vagy lehagytuk) volna.

3. A feltétellefedés elve

Ebben az esetben olyan teszteseteket kell készíteni, amelyhez a döntésekben szereplő minden feltételt legalább egyszer hamis, ill. igaz eredménnyel értékelünk ki. Ez a módszer általában hatásosabb az előzőnél, de nem mindig. Például:

Ha X>0 és Y>0 akkor ki: X*Y

Itt az (X=1, Y=–1) és az (X=–1, Y=1) tesztadatok elégségesek ezen elv megvalósításához, de az elágazás igaz ágát egyiknél sem hajtjuk végre, s így ez az előző elv követelményét nyilvánvalóan nem teljesíti.

4. A döntés- vagy feltétellefedés elve

Az előző pontban levő példából látható, hogy a feltétellefedés követelményét kielégítő tesztesetek nem feltétlenül elégítik ki a döntéslefedés követelményét. Ezért az előző két elvet egyesítő módszert kell készíteni úgy, hogy mindkét elv érvényesüljön.

Következtetés: Az ismertetett módszerek nem zárják ki egymást, önmagában egyik sem célravezető, így általában együttes használatuk szükséges.

Van a tesztelésnek egy különleges fajtája, amelyet stressz-tesztnek hívnak. Olyan programok esetén alkalmazunk ilyet, amelyek vagy nagy adatmennyiséggel dolgoznak, vagy fontos, hogy feladatukat adott időn belül elvégezzék. (Gondoljunk arra, hogy az adatmennyiség növelésével új minőség állhat elő.) Sokszor előfordul, hogy programok rejtett hibáit csak a valós tesztadatokkal való kipróbálás hozza elő, ezek közül is csak az, amely a gyakorlatban is ritkán fordulhat elő. Egyszerű példa: egy 100 adat kezelésére felkészült program:

```
200 DIM A(100)
210 INPUT N : IF N>100 THEN 210
220 FOR I=1 TO N
230 A(I+1)=I*N
240 NEXT I
```

A program mindaddig jól működik, amíg egyszer az N=100 tesztadattal nem próbáljuk ki. Sőt, ha a 200-as sort kihagyjuk, de a tesztadataink között nincs N=9-nél nagyobb eset, akkor sem vesszük észre a hibát.

Tehát a stressz-teszt lényege: próbáljuk ki a programot nagy adatmennyiséggel, ha mód van rá, akkor a felhasználó minél gyakoribb beavatkozásával.

PÉLDA A TESZTADATOK VÁLASZTÁSÁRA

Teszteljük a csillagsűrűsödések számát meghatározó programot (algoritmust)!

Először állítsunk elő tesztadatokat fehér doboz módszerekkel! Első esetben használjuk az utasításlefedés elvét, azaz minden utasítást legalább egyszer végre kell hajtani. Nézzük, hogyan tudjuk ezt elérni. A 2.2-ben levő feltétel először garantáltan igaz értékű, így a mögötte levő utasítást biztosan végrehajtjuk. A 4.-ben található ciklus magját is végrehajtjuk legalább egyszer, hiszen a kiindulócsillag mindenképpen tagja lesz a csoportnak. A 3.-ban levő darabszámnövelő utasítást akkor lehet végrehajtani, ha a térképünkön van sűrűsödés, ebből pedig következik, hogy 6.-ban elő fog fordulni, hogy az aktuális távolság kisebb volt S1-nél, s így az ettől függő feltételes utasítást is végrehajtottuk. Az 5.2. programrészlet ciklusmagját abban az esetben mindenképpen végrehajtjuk, ha van legalább 2 csillag a térképen (amit viszont a bemenő adatokról feltételeztünk).

Így azt kaptuk, hogy az algoritmus minden utasítását végrehajtjuk legalább egyszer, ha a térképünkön van sűrűsödés, azaz elég egyetlen jól megválasztott tesztadatsor, pl.:

N=3, S0=3, S1=100; koordináták: (1,1,1), (1,1,2), (1,2,2).

Eredmény: 1.

A tesztadatok kiválasztásának második esete a döntéslefedés elve volt. Itt úgy kell tesztadatokat választani, hogy minden feltétel lehessen igaz és hamis értékű is. Nézzük végig az algoritmusban szereplő feltételeket: 2.2.-ben a feltétel először biztosan teljesül, s ha van legalább kéttagú csoport, akkor lesz hamis értékű is. A 3. algoritmusrészlet igényli, hogy legyen sűrűsödés (igaz érték), ill. legyen sűrűsödésbe nem tartozó csillag is (hamis érték). A 4. ciklusmagját – mint láttuk – egyszer legalább végrehajtjuk, azaz a ciklusfeltétel lesz igaz és hamis értékű is. Az 5.2. eljárás azt követeli meg, hogy egy csillag vizsgálatakor legyen már olyan, nála nagyobb sorszámú csillag, amelyet besoroltunk egy korábbi csoportba, 6. pedig újra a legalább kéttagú csoport létezését követeli, valamint azt, hogy legyen olyan csillagpár is, amelyek a kritériumnál messzebb vannak egymástól.

Ezeket a feltételeket szintén ki lehet elégíteni egyetlen adatsorozattal:

N=3, S0=2, S1=10; koordináták: (1,1,1), (100,1,1), (1,1,2).

Ez esetben is 1 lesz az eredmény.

Nézzük meg most azt, hogy milyen tesztadatválasztást igényelnek a fekete doboz módszerek. Ezek szerint a programot ki kell próbálni érvényes és érvénytelen adatokkal is, valamint a két adattartomány határán. Így N, S0 és S1 értékeire kell meghatároznunk az érvényes és érvénytelen tartomány határait:

N: Feltételezésünk szerint értéke legalább 2, így kipróbálandó <2,2 és >2 értékekkel.

S0: A sűrűsödésbe tartozás csillagszámfeltétele legalább 2 és legfeljebb N érték lehetett, így tesztadatnak 2-nél kisebbet, N-nél nagyobbat, 2-t, N-et, valamint 2 és N közöttit érdemes választani.

S1: Válasszuk úgy, hogy lehessen a csillagok közötti minimális távolságnál kisebb, a maximálisnál nagyobb, ill. a kettő közötti érték is.

A sűrűsödés alakzata (formai tulajdonságai) szerint:

- Szakasz mentén elhelyezkedő csillagok.
  N=4, S0=2, S1=2; koordináták: (1,1,1), (1,1,2), (1,1,3), (1,1,4).
  Eredmény: 1.
  Megjegyzés: Kipróbálandó esetleg az iránytól való függés is.
- Sík mentén elhelyezkedő csillagok.
  N=4, S0=2, S1=2; koordináták: (1,1,1), (1,1,2), (1,2,1), (1,2,2).
  Eredmény: 1.
- Háromdimenziós tartományt elfoglaló csillagok.

A határeset-analízis alapján vehetünk további tesztadatokat:

- N=S0, N=S0–1.
- S0 = a maximális csoportlétszám;
  S0 = a maximális csoportlétszám + 1.
- S1 = két csillag pontos távolsága (ettől függjön a csoportlétszám).

Amennyiben ekvivalenciaosztályokat akarunk választani, akkor a sűrűsödések és a csillagok kapcsolatából indulhatunk ki, így választhatunk olyan adatsorozatot, amelyben:

Nincs sűrűsödés.
N=2, S0=2, S1=1; koordináták: (1,1,1), (100,1,1).
Eredmény: 0.

1 sűrűsödés van.

- Minden csillag beletartozik.
  N=2, S0=2, S1=10; koordináták: (1,1,1), (1,1,2).
  Eredmény: 1.
- Nem minden csillag tartozik bele.
  N=3, S0=2, S1=10; koordináták: (1,1,1), (1,1,2), (100,1,1).
  Eredmény: 1.

Több sűrűsödés van.

- Minden csillag sűrűsödésbe tartozik.
  N=4, S0=2, S1=3; koordináták: (1,1,1), (1,1,2), (5,5,5), (5,6,5).
  Eredmény: 2.
- Nem minden csillag tartozik sűrűsödésbe.
  N=5, S0=2, S1=3; koordináták: (1,1,1), (1,1,2), (5,5,5), (5,6,5), (9,9,9).
  Eredmény: 2.

PROGRAMHIBA-KERESÉS

Az előző fejezetekben a hibafelderítéssel foglalkoztunk, ennek a fejezetnek a tárgya a hiba behatárolása, a hiba kijavítása. E két résztevékenységre vonatkozóan a tesztelés alapelveihez hasonlóan megfogalmazhatunk néhány alapelvet.

A hibakeresés alapelvei:

1. A hibakeresési eszközök használata előtt célszerű igen alaposan végigvizsgálni a programot, és a program logikája alapján megkeresni a hiba okát. A hibakereső eszközöket csak a program alapos vizsgálata után vegyük igénybe!

2. Amíg a hiba helyét és okát pontosan nem találtuk meg, addig ne kezdjünk bele a javításba! A programban végzett meggondolatlan javítások újabb hibákat okozhatnak.

A hibajavítás alapelvei:

1. Ha a programban hibát találunk, akkor ennek a program más részeire is lehet hatása, azaz elképzelhető, hogy újabb hibákat is fogunk találni.

2. A hibát kell kijavítani és nem csak a tüneteit megszüntetni! Ha nem vizsgáljuk meg elég körültekintően a hiba okát, akkor csak részben tudjuk javítani.

3. A hibajavítás után a programot alapos tesztnek kell alávetni! Javítás közben új hibák keletkezhetnek!

4. Annak a valószínűsége, hogy egy hibát jól kijavítottunk, a program méretével arányosan csökken.

5. A hibák száma és súlyosságuk általában a program méreténél sokkal gyorsabban növekszik.

6. A hibajavítás visszanyúlhat a program tervezési fázisába is. Nagy programok esetén a programhibák többsége a tervezés során keletkezik.

PROGRAMHIBA-KERESÉSI MÓDSZEREK

1. Indukciós módszer

(Indukció (Új Magyar Lexikon): Abból a tényből, hogy nagyszámú tárgynak meghatározott tulajdonsága van és közös nemhez tartozik, arra következtetünk, hogy az adott nemhez tartozó összes tárgynak megvan ez az ismertető jegye.)

A hibakeresést a következőképpen végezzük: kiindulunk a rendelkezésre álló teszteseteredményekből, majd megpróbáljuk őket rendezni. Azokat a teszteseteket is célszerű megvizsgálni, amelyek nem idézik elő az adott hibát. A rendezett adatokból megpróbálunk valamilyen feltevést tenni a hiba okára vonatkozóan. Ha ezt a feltevést igazolni tudjuk, akkor következhet a hiba kijavítása, egyébként a folyamatot elölről kell kezdeni.

Például: X=0, 1, 5, 100 – a program jól működik;
X=–1, –7, –50 – rosszul működik.

Feltevés: a program a negatív számokra működik hibásan.

2. Dedukciós módszer

(Dedukció (Új Magyar Lexikon): Abból az ítéletből, hogy az adott nemhez tartozó összes tárgy meghatározott ismertetőjeggyel rendelkezik, arra következtetünk, hogy bizonyos, az adott nemhez tartozó tárgyak szintén rendelkeznek a szóban forgó ismertetőjeggyel.)

A módszer lényege az, hogy egyre szűkíti a hiba lehetséges okainak körét. A meglevő teszteseteredményekből adódó mindenféle lehetséges okot fel kell tételezni az első lépés-

ben, majd ezek közül ki kell küszöbölni azokat, amelyek a részletesebb vizsgálat során nem állják meg a helyüket. Ha élünk egy feltevéssel, ugyanúgy igazolnunk kell, mint az előző módszer esetén. Ha nem sikerül, akkor újabb információkat kell gyűjtenünk a hibakereséshez a hibajelenségről.

Például: Feltevés: a program mindig rosszul működik.

Teszt: X=3, 5, 7 – rossz eredmény;
X=12, 20 – jó eredmény.

Feltevések:

– X = primszám → rossz;
– X = páratlan → rossz;
– X < 10 → rossz;
– 4 nem osztója X-nek → rossz.

Amelyik feltevés igaz, az az által kijelölt halmaz (→ ekvivalenciaosztály) minden egyes tagjára rosszul fog működni a program.

Ez a két módszer elsősorban az algoritmusbeli hibák keresésére jó, a következő pedig a BASIC program vizsgálatához nyújt segítséget.

3. Visszalépéses technika

A legismertebb hibakeresési módszer, amelyet úgy végzünk el, hogy kiindulunk a hiba előfordulásának helyéről és a programot visszafelé hajtjuk mindaddig, amíg a végrehajtás eredményét hibásnak találjuk. Ha elérkeztünk egy olyan ponthoz a programban, ahol a hibás eredmények után helyes eredményeket kapunk, akkor valószínűleg megtaláltuk a hiba forrását.

4. Teszteléssel segített hibakeresés

A teszteseteket megkülönböztethetjük aszerint, hogy hibát akarunk felfedezni, vagy pedig egy ismert hibát akarunk előidézni a programban (a hiba okát keresve). Az utóbbi típusú tesztesetek szolgálnak hibakeresésre (emlékeztetünk a tesztesetek megismételhetőségére). Ezeknek a teszteseteknek jellegzetessége, hogy csak egyetlen feltételt fednek le.

Ezt a módszert általában nem önállóan, hanem az előző három módszer segítésére használjuk.

HIBAKERESÉSI ESZKÖZÖK

A legegyszerűbb eszköz – amely minden személyi számítógépen rendelkezésre áll – a programban elhelyezett alkalmas kiíró utasítások, amelyek segítségével nyomon követhetjük programunk működését. Más hasznos eszközök már sajnos nem minden esetben találhatók meg az egyes géptípusokon.

Nyomkövetés

A nyomkövetés lehetővé teszi, hogy megfigyeljük, programunk milyen utasításokat hajtott végre. A BASIC-ek általában azt a lehetőséget adják, hogy a végrehajtott utasítások sorszámait kiírathatjuk a képernyőre. Erre a célra a TRACE (TRON) parancs szolgál, a nyomkövetés kikapcsolását pedig a NOTRACE (UNTRACE, TROFF) paranccsal érhetjük el. Ezek az alapszavak magába a programba is elhelyezhetők, így a program egy részét is nyomon lehet követni. Fejlettebb rendszerekben megadható, hogy csak bizonyos típusú utasítások (pl. vezérlésátadás) végrehajtását jelezze ki a számítógép, vagy pedig az egyes utasítások eredményét is közölje valamiképpen.

Nyomkövetés a hibától visszafelé

A BASIC nyelvjárások túlnyomó többségében sajnos nincs meg ez a lehetőség, ami lehetővé teszi, hogy hiba esetén megkapjuk az utoljára végrehajtott néhány utasítás sorszámát és egyéb jellemzőit.

Lépésenkénti végrehajtás

Ez inkább a gépi kódú programozáshoz használható segédeszköz, egy-egy utasítás végrehajtása után mindig a felhasználó dönthet a következő feladat kiválasztásáról.

Töréspontok elhelyezése

Néhány BASIC változatban lehetőség van BREAK utasítások elhelyezésére a programban. Ezeknél a végrehajtás megszakad, a változók értékei kiírathatók, megváltoztathatók, majd a CONT paranccsal folytatható a program. Ugyanezt a hatást a STOP utasítás elhelyezésével is el lehet érni. Sajnos vannak olyan géptípusok, ahol a megszakított program nem folytatható, tehát ez a lehetőség nem használható. Fejlettebb rendszerekben megadható a törésponton való áthaladások száma, ami után megszakad a program végrehajtása.

PÉLDÁK A HIBAKERESÉSRE

Először egy egyszerű programban keressünk hibát, amely két szám legnagyobb közös osztójának meghatározására szolgálna, ha helyesen működne. A hibakeresési módszerek bemutatására szándékosan választottunk ilyen rövid programot, durva hibákkal tűzdelve:

```
10      INPUT A,B
20      X=A : Y=B
30      IF X=Y THEN 60
40      IF X<Y THEN X=X-Y : GOTO 20
50      IF Y<X THEN Y=X-Y
60      GOTO 30
70      PRINT X
80      STOP
```

Bár ránézésre is felfedezhető néhány hiba, most kövessük hibakereső módszereinket! Válasszunk a programhoz tesztadatokat az ekvivalenciaosztályok módszerével! Ezek a következők lesznek (A és B természetes szám lehet):

				Eredmény
1.	A = B:	(1,1)		1
		(8,8)		8
2.	A < B:	A osztója B-nek:	(3,6)	3
		van közös osztó:	(4,6)	2
		nincs közös osztó:	(1,6)	1
3.	A > B:	az előzőek megismételhetők fordított sorrendben (elképzelhető, hogy az előző csoporthoz képest más lesz az eredmény).		
4.	Érvénytelen adatok: (0,1), (1,–4), (1,0,5) stb.			

Első tapasztalatunk, hogy bármelyik tesztadatot is használjuk, a program nem ad semmilyen eredményt. A programot megvizsgálva azt tapasztaljuk, hogy kiíró utasítás és STOP is van benne. Ezek szerint a program ezekre a sorokra (70,80) soha nem jut el, azaz van benne – legalább egy – végtelen ciklus. Az ötletszerűen megírt program hibája, hogy nem látszik világosan, hol is van ez a ciklus. Ellentmondás-kereséssel találhatunk egy érdekességet: elképzelhető, hogy a 30-as és a 60-as sorokban a program végtelen sokáig jár felváltva. Kézenfekvő javítás:

```
30      IF X=Y THEN 70
```

Ezután a próbákat elölről kezdve az első két esetben megkapjuk a helyes eredményt. A (3,6) adatpárra azonban újra nincs semmilyen válasz sem. Ugyanezt tapasztaljuk az összes A<B, ill. A>B típusnál is. Most a program elvileg eljuthatna a kiíró utasításhoz, azaz kicsit bonyolultabb a feladatunk. Még mindig a programszöveg elemzésénél maradva, újabb ellentmondásra figyelhetünk föl: ha kezdetben A<B teljesült, akkor a program 20-as és a 40-es sorok között keringhet végtelen ciklusban. Figyelmesen megszemlélve a megoldást, a következő javítást kell elvégeznünk (az újbóli felesleges X=A, Y=B értékadások elkerülésére):

```
40      IF X<Y THEN X=X-Y : GOTO 30
```

Újabb próbát végezve, az A=B párok továbbra is jó eredményt szolgáltatnak (szerencsére nem rontottunk el semmit!), a többiek azonban most sem adnak eredményt. A programban most már vezérlési rendellenességet nem találunk, így a logikáját kell figyelembe vennünk. Mivel a hiba helyéről semmi támpontunk nincs, így valamilyen hibakeresési eszköz segítségét kell igénybe vennünk. Az egyik legegyszerűbb megoldást választjuk: elhelyezünk a program 30-as és 40-es sora közé egy kiírást (hiszen az ismeri jól a programját, aki tudja, hogy alkalmas pontokon a változói milyen értéket vehetnek fel):

```
35        PRINT X,Y
```

Újra megvizsgáljuk a teszteseteket. Most az A<B típusú pároknál azt tapasztaljuk, hogy X értéke folyamatosan csökken, s rögtön az első lépésben negatív lesz. A program szövegéből kitűnik, hogy az eredmény meghatározására azt a módszert használja, hogy A és B értékét egyre kisebbre transzformálja úgy, hogy a legnagyobb közös osztó megmaradjon. Nyilvánvaló, hogy X és Y nem lehet negatív! A kiírás szerint azonban negatív lesz, ugyanakkor Y értéke nem változik. Az A>B típusú pároknál is érdekes eredményt kapunk: X értéke változatlan marad, Y pedig felváltva vagy a kezdő B érték, vagy pedig A–B. Tehát mindkét esetben a nagyobb érték marad változatlan, ami arra utal, hogy nem a megfelelő változó értékét változtatjuk meg a programban. Reméljük észrevette az Olvasó, hogy most az indukciós módszert használjuk!

Első, ám meggondolatlan javítás a következő lehet:

```
40        IF X<Y THEN Y=X-Y : GOTO 30
50        IF Y<X THEN X=X-Y
```

Ez a javítás tipikus példa a tüneti kezelésre. Ezzel ui. nem segítünk azon, hogy az egyik változó értéke lehet negatív is! Van itt még más hiba is: olyankor végzünk el egy kivonást, amikor nem lenne szabad! Valóban, a 40-es sorban a kisebb számból vonjuk ki a nagyobbat, ott a kivonást meg kell fordítani:

```
40        IF X<Y THEN Y=Y-X : GOTO 30
```

Újra kipróbálva a programot az összes érvényes adattal, most már helyes eredményt kapunk. Az érvénytelen adatokkal való próba során azonban még mindig dolgozik, és hamis eredményt ad, ha ad egyáltalán eredményt. A bemenő adatok nincsenek ellenőrizve, tehát azt még pótolni kell.

A hibák mellett még az is zavaró, hogy a programból nem olvasható ki, hogy mit csinál, mik készítőjének szándékai.

Elkészült a helyesre javított program (bár még a módszeres programtól várt minden jó tulajdonsággal nem rendelkezik):

```
10 INPUT A,B
15 IF A<1 OR B<1 OR A<>INT(A) OR B<>INT(B)
        THEN 10
20 X=A : Y=B
30 IF X=Y THEN 70
40 IF X<Y THEN Y=Y-X : GOTO 30
50 IF Y<X THEN X=X-Y
60 GOTO 30
70 PRINT X
80 STOP
```

Figyeljük meg, hogy a módszeresen megírt, nagyságrendekkel nagyobb és bonyolultabb programban sem lesz nehezebb hibát keresni!

Helyezzünk el a csillagsűrűsödéses programban két hibás sort (képzeljük el, hogy a kódolás során hibát vétettünk ebben a két sorban):

```
610     V(I)=1
750     IF T<S1 THEN L=J : GOSUB 600
```

Próbáljuk ki a programot a következő adatsorral (egy sűrűsödés van, de nem minden csillag tartozik bele): N=4, S0=2, S1=3, a csillagok koordinátái: (1,0,1), (1,2,1), (1,4,1), (9,1,1)!

A programunk válasza: a 620-as sorban hiba van. Írjuk ki az A változó értékét! A kapott érték 5. Ezután a CS vektor tartalmát kell kiírnunk, s meglepve tapasztalhatjuk, hogy a 2. csillag kétszer szerepel benne. Mi a programot úgy terveztük meg, hogy egy csillagot kétszer ne vehessünk fel. Csináljunk most egy olyan módosítást, amely kizárja, hogy a csillagot még egyszer felvegyük. Ez nem a végleges javítás lesz, hiszen most csak egy tünetet szüntetünk meg, s tesszük ezt annak érdekében, hogy könnyebben megtalálhassuk a hibát. A javítás:

```
601     FOR B=1 TO A
602     IF CS(B)=L THEN RETURN
603     NEXT B
```

Újra kipróbáljuk a programot, a válasza: 2 sűrűsödés van, azaz nem jól működik. Most használjuk a dedukciós módszert hibakeresésre. A hiba lehetséges okai:

- Olyan csillagnál is számol sűrűsödést, ami nincs sűrűsödésben.
- Egy sűrűsödés több tagjánál is számol.
- Minden csillagot sűrűsödésnek vesz.

Vegyük észre, hogy a harmadik feltevés az első kettő együttes teljesülését jelentené, egyetlen próbálkozásunk azonban ezt cáfolja. Ezek szerint meg kellene vizsgálni, hogy az első kettőből melyik az igaz. A hibakeresést segítsük teszteléssel! A következő eseteket vizsgáljuk meg:

- Nincs sűrűsödés;
  N=2, S0=2, S1=1; koordináták: (1,1,1), (9,9,9).
- Minden csillag egy sűrűsödésben van;
  N=3, S0=2, S1=3; koordináták: (1,0,1), (1,2,1), (1,4,1).

Azt tapasztalhatjuk, hogy a második feltevés volt az igaz.

Most újabb részfeltevéseket mondunk ki:

- Minden sűrűsödésbe tartozó csillagnál számolunk egyet.
- Legalább kettőnél, de nem mindegyiknél számolunk.
- Van olyan csillag, amelyiknél nem egyet számolunk.
- A számláló értékét illegális helyen is változtatjuk.

Eddigi próbáink alapján az első feltevés nem teljesül. A harmadik úgy lehetne igaz, ha egy csillag többször előfordulhatna a sűrűsödés induló csillagai között, vagy ha lehetséges lenne, hogy nem eggyel növeljük a számlálót. A programlista gyors átnézésével meggyőződhetünk arról, hogy egyik eset sem áll fönn, sőt a negyedik feltevés sem. Maradt a második. Próbáljuk meghatározni, hogy a sűrűsödés hány tagjánál számolunk! Olyan tesztadatokkal próbáljuk ki a programot, amelyben minden csillag egy sűrűsödésbe tartozik, 2 csillag már sűrűsödés, s a csillagszám rendre 2, 3, 4, ezután pedig 3 csillag kell a sűrűsödéshez, s a csillagszám 3, majd 4.

Így a sűrűsödések számára kapott érték: CSSZAM–SO+1! Vajon melyek ezek? Megtalálásukhoz helyezzünk el a programban egy kiíró utasítást ott, ahol a sűrűsödések számát növeljük:

```
581      PRINT I,S,A
```

Azt tapasztalhatjuk, hogy az utolsó SO–1 tagnál nem számolunk, a többinél viszont igen, holott a másodiktól kezdődően nem is hajthatnánk végre ezt a kiíró utasítást. Mivel a feltételvizsgálat szerepel a 450-es sorban, most már nyilvánvaló, hogy az egyes csillagok vizsgálatának letiltását nem végezzük el, a javítás:

```
610      V(L)=1
```

A korábbi adatokkal kipróbálva programunkat, most már helyes eredményt kapunk.
Itt valami borzasztó dolog történt! A másik hibát nem vettük észre! A probléma oka: nem próbáltuk ki elég következetesen a programot. Hiányoznak még pl. a határeset-analízis során megadott tesztadatokkal való próbák. Ezek segítségével felfedezhetjük a másik hibás sort is.

Megjegyzés: Ez utóbbi hiba nem is annyira súlyos, gondoljunk csak a gyakorlati feladatra! A mért értékeink (csillagkoordináták) ui. pontatlanok, így ehhez képest az egyenlőség megengedése vagy kizárása lényegtelen. Ez jelentős különbség a matematikai, ill. a valós méréseket feldolgozó programok között!

HELYESSÉGBIZONYÍTÁS

A teszteléssel csak annyit tudunk kimutatni, hogy a program hibás, avagy, hogy a hiba nem került felszínre. Megnyugtatóan a programhelyesség-bizonyítással lehet a program jóságáról meggyőződni.

Egy feladat felől fogjuk e problémát megközelíteni, amelyhez megadjuk a programot, s bebizonyítjuk, hogy ez helyes megoldása a feladatnak. Legyen a feladat egy N elemű A( ) vektor maximális és minimális elemének (helyének, indexének) a megkeresése. Ennek egy megoldása lehet a következő program:

```
Eljárás max-min(A(),N):
     MAX:=A(1) : MIN:=A(1) : MAXH:=1 : MINH:=1
     Ciklus I=2-töl  N-ig
       A:=A(I)
       Elágazás
         MAX<A  esetén  MAX:=A : MAXH:=I
         MIN>A  esetén  MIN:=A : MINH:=I
       Elágazás vége
     Ciklus vége
     Ki: MAXH;MAX,MINH;MIN         [ MAX=A() maximális
      értéke és A(MAXH)=MAX ;  MIN=A() minimális
      értéke és A(MINH)=MIN ]
Eljárás vége.
```

Lássuk be a program helyességét! Egyáltalán hogyan fogjunk neki? Próbáljunk elvárásainknak megfelelő állításokat beilleszteni a programba, majd ezek teljesülését ellenőrizni. Ha ezen a kétségkívül nehéz, sok megérzést igénylő munkán túl vagyunk, akkor már csak ezen állítások egymásba való átalakulását kell kimutatni. Az átalakulást maguk az állítások közötti programrészek biztosítják.

Hogy milyen legyen az állítás? Hogyan találhatók az egyes szerkezethez legjobban simuló formulák? Mik ezek jellegzetes, egyedi – de mégis eléggé általános – sajátosságai? Ezekre a kérdésekre keressük a választ az előbbi példaprogram elemzése során.

```
   Eljárás max-min(A(),N):
----->    [ E ]
          MAX:=A(1) : MIN:=A(1) : MAXH:=1 : MINH:=1
----->    [ p1 ]
          Ciklus I=2-töl  N-ig
----->      [ p2 ]
            A:=A(I)
            Elágazás
              MAX<A  esetén  MAX:=A : MAXH:=I
              MIN>A  esetén  MIN:=A : MINH:=I
            Elágazás vége
----->      [ p3 ]
          Ciklus vége
----->    [ p4 ]
          Ki: MAXH;MAX,MINH;MIN         [ MAX=A() maximális
           értéke és A(MAXH)=MAX ;  MIN=A() minimális
           értéke és A(MINH)=MIN ]
----->    [ U ]
   Eljárás vége.
```

Az eljárást megtűzdeljük állításokkal. Egyelőre csak helyük rögzített. Hogy miért éppen ezeket a helyeket jelöljük ki, miért éppen ezeket tekintjük kritikus helyeknek, erre később még visszatérünk, addig is érdemes ezen eltöprengeni.

Az állítások társas kapcsolatai

Kezdjük azzal az *elfajult* esettel, hogy egyetlen elem *közül* kell a legkisebbet és a legnagyobbat kiválasztani.* Ekkor a ciklus végrehajtására nincs szükség, így
a 'p1 és N=1'-ből következzen a 'p4'.
A ciklusba való belépéskor elvárjuk – a *folytonosság* miatt –, hogy
a 'p1 és N>1 és I=2'-ből következzen a 'p2'.
A ciklus többszöri lefutásához elengedhetetlenül szükséges *folytonosság* biztosítását kívánja meg a ciklusmagot alkotó utasításoktól a következő állításazonosság:
'p3(i)' = 'p2(i+1)' (i=2,...,N–1)
(az egyszerűbb írásmód kedvéért bevezettük és később is alkalmazzuk az állítások függvényszerű írását; pl. a p3(i) jelentése: a p3 a ciklusváltozó i értéke mellett).
Másként ezt így is megfogalmazhatjuk:
a 'p3 és I<N'-ből következzen a 'p2'.
Végül a ciklusból kilépve is igaznak kell lennie a p4-ben megfogalmazott állításnak. Formálisan:
a 'p3 és I=N'-ből következzen a 'p4'.
Bizonyításunk logikai vázát megadtuk. Most már csak pontos kivitelezése marad hátra és beláthatjuk (sőt bebizonyíthatjuk) a program helyességét vagy helytelenségét.

A bizonyítás

Induljunk ki a programtól elvárt tulajdonságból (az eljárás utófeltételéből):
U = MAX>=A(j) (j=1...N) és A(MAXH)=MAX és
MIN<=A(j) (j=1...N) és A(MINH)=MIN.

A kiírás során a program állapotterében** nem történik változás, így

p4 = U.

A ciklusban p3-t úgy kell megfogalmazni, hogy az változatlanul igaz *(invariáns)* maradjon a ciklus akárhányszori lefutása után is. Hiszen csak így remélhetjük, hogy a ciklus egyszeri végiggondolásával is következtetni tudjunk a hatására bekövetkezett megváltozásra. Tehát pl. igaz legyen a ciklusba való belépés pillanatában, ill. onnan kilépve is. Egy ilyen mindig igaz állítás (legalábbis elvárásaink szerint) maga a p3:
p3 = MAX>=A(j) (j=1...I) és A(MAXH)=MAX és
MIN<=A(j) (j=1...I) és A(MINH)=MIN,

azaz az éppen érvényes I-ig rendelkezünk a legnagyobb és legkisebb értékkel (MAX,MIN), s ezek helyével (MAXH,MINH).

Ekkor nyilvánvaló, hogy ha a ciklus *lejár*, vagyis, ha I=N, akkor p3 éppen a kívánt p4 állításba megy át:

p3 és I=N= p4.

*Bár a feladat így leegyszerűsítve önmagában valószínűtlen, hogy valaha is előforduljon – erre ui. senki sem használ számítógépet –, egy nagy feladat részeként azonban már nem elképzelhetetlen.

** Az állapottér egy képzeletbeli sokdimenziós tér, amelyben a program „mozog" a végrehajtott utasításai segítségével.
Az állapottér egy-egy dimenzióját alkotja a program egy-egy változója. Így a változók pillanatnyi értékei határozzák meg, hogy a program az állapottér éppen melyik pontjában tartózkodik.

A ciklusból kilépve az állítástranszformáció tehát *folytonos*. Kérdés, hogy az algoritmus e folytonosságot biztosítja-e a ciklus egyszeri lefutása során is? Ez jelentené az állítás állandóságát, invariáns voltát. E kérdés megválaszolásához p2-ről kell többet tudnunk!

A p2 és p3 között egy elágazás van, tehát e tényt kell p2 felépítéséhez megfognunk. Írjuk föl az elágazásbeli három (!) esetet úgy, hogy p3 következzék belőle! Ha a végrehajtás során a 'MIN:=A : MINH:=I. utasításokat végrehajtva jutottunk a p3-hoz rögzített ponthoz, akkor p4 csak úgy maradhat igaz, ha 'A<A(j) (j=1...I–1)'. Vagyis p3 igazsága fenntartható, ha az utasításokat a 'MIN>A' feltétel teljesüléséhez kötjük. Ezt tettük az elágazás megfelelő feltételével! Hasonló megfontolásokkal látható be az elágazás másik ága. Ha 'A' a jelenlegi MAX és MIN értékek közé esik (ez a 3. eset), akkor az elágazás p3-t nem módosítja, így állandóságát (invarianciáját) nem *bántja*.

Nos, az elágazással az 'A(1),...,A(I–1),A' halmaz minimumát, ill. maximumát a 'MIN', 'MAX', 'MAXH' változókban állítottuk elő. Mivel az 'A:=A(I)' utasítással bővítettük ezt a halmazt, ezért az elágazásbeli utasítások a valóban vizsgált 'A(1),...,A(I–1),A(I)' halmazból választják ki a maximumot és a minimumot. Így p2-re a ciklusváltozó eggyel való megnövekedése miatt az adódik, hogy az 'A(1),...,A(I–1)' halmaz minimumát, maximumát tartalmazzák az előbbi változók. Más szóval

p2(I)=p3(I–1).

Az elágazásra vonatkozó következtetéseinket foglalja össze a következő három állítás:

p3(I–1) és MIN>A(I) => p3(I),
és p3(I–1) és MAX<A(I) => p3(I),
és p3(I–1) és MIN<A(I) <=MAX =>p3(I).

Megnyugtató eredmény:

p3(I–1) => p3(I).

Tehát azt beláttuk, hogy az utófeltétel teljesülni fog, feltéve, hogy a ciklusból egyszer kilépünk. (A ciklusból való kilépést a szakmai nyelv a ciklus *terminálásaként* említi.) E probléma persze az ilyen *számlálásos* ciklusoknál normális esetben nemigen merülhet föl* mégis érdemes meggondolni, hogy miként látható be – általában is – a terminálás. A kilépés feltételéhez való határozott közeledést kell ügyesen detektálnunk! Például, ha találunk egy, a ciklusra jellemző adattól függő függvényt, amely értéke a ciklus minden egyes lefutása során csökken, és véges lépés során szükségképpen eléri a 0-t, akkor máris célhoz értünk. A konkrét számlálásos ciklusra a c(I):=N–I függvény ilyen.

Terminálás: c(I)=c(I–1)–1 és c(I)>=0.

Ahhoz, hogy a ciklus a p3-t ne változtassa, p3-nak belépéskor is igaznak kell lennie. Vizsgáljuk meg, milyen p1-ből következik p3! Vegyük p3-hoz az 'I=1' feltételt! Ezt sugallja, hogy a ciklust I=2-től indítjuk.

p1 = p3 és I=1.

Ezt a korábbi jelölésünkkel így írhatjuk:

p1=p3(1),

p3 kifejtése után

p1 = MAX>=A(j) (j=1..1) és A(MAXH)=MAX és
MIN<=A(j) (j=1..1) és A(MINH)=MIN,

*Nem normálisnak tekintjük a véges pontosságból és az aritmetikai műveletek pontatlanságából származó kisiklásokat, amik nem is olyan ritkán fordulnak elő. Ám ez nem az algoritmus logikai hibája.

egyszerűbben írva

p1 = MAX=A(1) és A(MAXH)=MAX és
MIN=A(1) és A(MINH)=MIN.

Az E-re visszakövetkeztetni már nem jelent nehézséget, hiszen épp a p1 – előző kívánalmak szerinti – beállítását végzi az első két utasítás. Így E-re megkötésünk nincs, vagyis

E = igaz.

Ha szigorúan vesszük a bizonyítást, az E azononsan igaz volta helyett a bemenő adatok tulajdonságaira vonatkozó állítást kellett volna kapjuk. (E=az N természetes szám, a vizsgált A(1),...,A(N) vektorelemek valós számok.) Mivel a bizonyítás során nem tekintettük ezek teljesülését kétségesnek, ezért nem bukkant föl egyik állításunkban sem.

Képzeljük el ezt a módszert komolyabb méretű programok esetén! Sajnos gyakorlatilag járhatatlan az út és valljuk be, túlságosan mesterkéltnek, természetellenesnek is tűnik! Mégsem lehet lemondani a program helyességének belátásáról. Megoldásként egyetlen lehetőség marad – egyszer már segített, hátha most is hasznos lesz –: követni azt az elvet, amely már programozási vezérfonalunk is volt: *a lépésenkénti finomítást.* Nem elképzelhetetlen, hogy eredményre vezet, hiszen úgyis pontosan megadtuk az eljárásokat, azaz az általuk megvalósított transzformációkat, vagyis az elő- (E) és utófeltételt (U).

Bizonyítandó tehát csak a következő állítás lesz:

E Program U,

(azaz az E-nek eleget tevő bemeneti paraméterek esetén a program kimeneti paraméterei U-beliek lesznek és befejeződik),
ha

Program : Program_1 [p_1] ... Program_n [p_n],

(vagyis a programot elemibb eljárásokból állítottuk össze).
Meggondolandó részállítások:

E Program_1 p_1
... ...
p_{n-1} Program_{n-1} p_n
p_n => U.

E részállítások bizonyítására a megfelelő (rész)program további kifejtése után nyílik lehetőség. Mindenesetre az elkészült program helyessége máris bizonyítható e részállítások helyességéből kiindulva, sőt ***bizonyítani is kell.*** Egy fontos szemléletről van szó: a program bizonyítását már az egyes szintek megírásával *párhuzamosan* kell elvégezni.

A 'p' feltételek nemcsak a program egyedi transzformációt írják le, hanem az összes korábbi eredményt is tartalmazzák.

Nem mindig kell egyébként a részállítások bizonyításához az előfeltétel szigorú alakját felhasználni, néha elegendő annak egy *gyengített* változata is. Ezzel többnyire egyszerűbb alakú feltételhez jutunk, vagy a bizonyításhoz alkalmasabb formulát kapunk.

Megjegyzések:

– Nem egy elterjedt módszert tárgyaltunk, hanem többnek a *hangulatát* érzékeltettük.

– Nem törekedtünk a bizonyításnál a teljesen formális leírásra, csak annyira, amennyire a megértéshez szükséges volt. A vázolt módszer csak vázlata a valóságos helyességbizonyításnak, így pontosabban helyességbizonygatásnak nevezhetnénk, de azért arra alkalmas, hogy ürügyén

1. a program írója újra átgondolja a programját (pontosabban egy szintjét), s ezzel a hibákat felszínre hozhassa;

2. a vállalkozó szellemű Olvasók a formális helyességbizonyítás felé nagy lépést tehessenek.

– A helyességbizonyítás gondolatsorát folytatva, a formalizálás után eljuthatunk az *automatikus programgenerálás* gondolatához, megvalósításához*.

*Az automatikus programgenerálás lényege, hogy a feladat megoldásának csak a formális leírását (specifikációját) kell megadni, amiből a gép maga állítja elő a programot. Leegyszerűsítve azt mondhatjuk, hogy nem marad más teendő, mint a mit megfogalmazása, mert a hogyan ebből automatikusan származtatható.

NEGYEDIK PÉLDA A MÓDSZERES PROGRAMOZÁSRA
(programok összehasonlíthatóságáról)

Gyakran kell olyan jellegű feladatot megoldani, ahol a megoldás jelentősen befolyásolhatja a program hatékonyságát.

Feladat: Adott egy N elemű táblázat, amelynek elemeit K hellyel kell ciklikusan balra léptetni (K<N).

Ezt a részfeladatot tartalmazó feladat egy titkosírás megfejtése, amelyet egy kódtáblázat alapján készítünk. A táblázat egy eleme megmutatja, hogy egy betűt milyen másik betűvel kell helyettesíteni, a táblázat I. eleme az ábécé I. betűje helyére írt betűt tartalmazza. Ez a titkosírás hosszú szövegek esetén viszonylag könnyen megfejthető az egyes betűk előfordulásának gyakorisága alapján. Tudjuk pl., hogy a magyar nyelvben az „e" a leggyakoribb betű. Ha úgy módosítjuk a titkosírást, hogy minden sor vagy akár minden betű más kódtáblázat alapján készül, akkor a megfejtés sokkal nehezebb. A táblázat soronkénti transzformációját a táblázat elemeinek ciklikus eltolásával oldjuk meg.

Egy táblázat elemeinek eggyel balra való ciklikus léptetése jelentse azt, hogy minden elem eggyel előbbre kerül, az elsőt pedig a végére tesszük.

A balra eggyel léptetésnél megjegyezzük az első elemet, az első helyére tesszük a másodikat, a második helyére a harmadikat, ..., s végül az utolsó helyére a korábban megjegyzett elsőt. A megjegyzéshez szükségünk van egy új változóra. Ha mindezt K-szor végezzük el, akkor megtörtént a K-val balra léptetés.

```
Léptetés:                                  (1. változat)
    Ciklus J=1-től K-ig
        X$:=T$(1)
        Ciklus I=2-től N-ig
            T$(I-1):=T$(I)
        Ciklus vége
        T$(N):=X$
    Ciklus vége
Eljárás vége.
```

Próbálkozzunk egy másik megoldással. Ha megjegyezzük az első K db elemet, akkor a K+1-diket rögtön a helyére, az első helyre tehetjük, a K+2-diket a másodikra, ..., s végül a megjegyzett K db elemet az előremásoltak mögé tesszük. A megjegyzéshez szükségünk van egy vektorra – X$(K).

```
Léptetés:                                  (2. változat)
    Ciklus I=1-től K-ig
        X$(I):=T$(I)
    Ciklus vége
    Ciklus I=K+1-től N-ig
        T$(I-K):=T$(I)
    Ciklus vége
    Ciklus I=1-től K-ig
        T$(N-K+I):=X$(I)
    Ciklus vége
Eljárás vége.
```

Próbálkozzunk egy harmadik megoldással is! Jegyezzük meg az első elemet! Tegyük a helyére a K-val később levőt, a K-val később levő helyére a nála K-val később levőt, ..., s ha N–1 ilyen előremásolást elvégeztünk, akkor az utolsónak maradt üres helyre tegyük az először megjegyzettet!

Megjegyzés: Módszerünk jól működik, ha szavatolni tudjuk, hogy minden egyes elemet előrehoztunk, egy lépésben K hellyel, és mindegyiket csak egyszer hoztuk előre.

```
Léptetés:                               (3. változat)
    L1:=1
    X$:=T$(L1)                          ; első üres hely
    Ciklus I=1-től N-1-ig
        L2:=L1+K
        Ha L2>N akkor L2:=L2-N          ; a maradék
        T$(L1):=T$(L2)
        L1:=L2                          ; köv. üres hely
    Ciklus vége
    T$(L1):=X$
Eljárás vége.
```

Vajon jó-e ez a módszer? Próbáljuk ki! N=6, K=1 esetben a megoldás jó, azonban az N=6, K=3 esetben a program csak az első és a negyedik elemet cserélgeti! Keressük meg, hogy milyen K értékekre működik az algoritmus helyesen. Tegyük fel, hogy N és K olyan számok, hogy csak egyetlen közös pozitív osztójuk van, az 1! Ekkor igaz: I∗K (I=0,1,2,...,N–1) N-nel való osztásának maradékai mind különbözőek, és így kiadják az összes számot 0 és N–1 között. Ezekhez az értékekhez egyet hozzáadva megkapjuk az összes 1 és N közötti számot és mindegyiket pontosan egyszer. Vegyük észre, hogy ha ezeket a számokat indexnek használjuk, akkor a sorozat első helyén levő indexű betű helyére pontosan a második helyen levő indexű betűt kell tenni, mivel a második index pontosan K hellyel mutat későbbre a táblázatban. Ez az összefüggés az összes egymást követő számpárra érvényes.

Mivel az előbbi számsorozatban az összes 1 és N közötti index szerepel, ezért minden elem léptetését elvégeztük. Mivel mindegyik pontosan egyszer szerepel, ezért mindegyiket egyszer hoztuk előre. Mivel két elem közül a második (amit előrehoztunk) mindig K-val volt hátrább, ezért mindegyiket K-val hoztuk balra.

Matematikai érdeklődésű Olvasóinkra bízzuk a feladat megoldását abban az esetben, amikor K-nak és N-nek van az 1-től különböző pozitív osztója.

Térjünk rá a fejezet fő kérdésére: van három megoldásunk, vajon melyik a *jobb?* Válaszunk: attól függ, mi a célunk! Ez nagyon fontos megállapítás, egy feladat két megoldása általában sohasem hasonlítható össze önmagában. Az összehasonlításhoz mindig kell valamilyen követelményrendszer, amit a feladat kitűzésekor kell megállapítani.

Vegyünk példánkkal kapcsolatban három követelményt: a megoldás

– legyen gyors;
– kevés helyet használjon;
– legyen egyszerű (könnyen és gyorsan elkészíthető)!

Vizsgáljuk meg megoldásainkat a követelményeink szempontjából!

Sebesség (S): Egy megoldás sebességét mérjük a végrehajtott értékadó utasítások számával!

1. megoldás: S=K∗(N+1).
2. megoldás: S=N+K;
3. megoldás: S=4∗N–1;
ha az aritmetikai értékadásokat nem vesszük figyelembe: S=N+1.

Tehát a sebesség szempontjából az első megoldás kirívóan rossz.

Helyfoglalás (H): Mérjük a helyfoglalást a szöveg típusú változók, tömbelemek számával!

1. megoldás: H=N+1;
2. megoldás: H=N+K;
3. megoldás: H=N+1.

Ebből a szempontból tehát a második megoldás rosszabb a többieknél.

Egyszerűség: Az első és a második megoldást nagyon könnyen elkészítettük, a harmadikhoz azonban némi matematikai ismeretre volt szükségünk. Ennek az elkészítése bizony hosszabb időt vett igénybe, több munkánkba került.

Minden szempontból kielégítő megoldást nem kaptunk, ahogyan teljesen rosszat sem. Tehát a követelményeket rangsorolni kell, s azt a megoldást fogadjuk el legjobbnak, amelyik a legfontosabb követelmény(ek) szerint jó, a kevésbé fontos(ak) szerint pedig nem kell annyira jónak lennie.

Azt tapasztaltuk, hogy egy feladat megoldásai közül a feladat kitűzésekor megadott követelményrendszer alapján tudunk választani. Természetesen minden feladat megoldásakor nem készíthetjük el az összes megoldást. Mi ilyenkor a teendő? Ha a lépésenkénti finomítás során eljutunk egy, a megoldást befolyásoló döntésig, akkor ott fel kell vázolni a megoldási lehetőségeket, és ezen lehetőségek teljes kifejtése nélkül – korábban szerzett programozási tapasztalatokra hagyatkozva – kell kiválasztani azt, amit a legjobbnak tartunk.

PROGRAMOK HATÉKONYSÁGA

A hatékonyságvizsgálat, a hatékonyabbra átírás előfeltétele a program helyessége. Annak ui. semmi értelme sincs, hogy egy hibásan működő program gyorsabban működjön.

A programozásban hatékonyságon általában a helyfoglalás és a futási idő lehető legkisebb értékét értjük. Ez a két szempont legtöbbször ellentmondó követelményeket támaszt a programmal szemben, így ilyenkor engedményeket kell tenni. A hatékonyság vizsgálatánál mindig meg kell különböztetni a hatékonyság két fajtáját: az *algoritmus* és a *kód* hatékonyságát.

A fontos jellemzők:

- maximális végrehajtási idő;
- átlagos végrehajtási idő (a végrehajtási idő várható értéke);
- indexes és szöveg típusú változók helyfoglalása;
- a programkód hossza.

AZ ALGORITMUS HATÉKONYSÁGA

Legtöbbször a program hatékonyságát elsősorban az algoritmus befolyásolja, s a kódolásnak, a BASIC nyelv jellemzői figyelembevételének lényegesen kisebb a hatása. Elsőként ott érdemes a hatékonyság vizsgálatával foglalkozni, ahol a program sokszor hajt végre egy műveletet, azaz a ciklusokat kell vizsgálni. Két lehetőség van: csökkentsük a ciklusok végrehajtásainak számát vagy a ciklusok egyszeri végrehajtási idejét. Az előbbire akkor van lehetőségünk, ha ki tudjuk használni az adatok valamilyen speciális tulajdonságát vagy matematikai ismereteinket.

Nézzünk először egy keresési feladatot az algoritmus hatékonysága bemutatására!

Feladat: Adott egy N elemű rendezett számsorozat az A(N) vektorban és egy keresett elem (X), amely biztosan megtalálható az elemek között. Feladat a keresett elem sorszámának meghatározása.

(Emlékeztetünk a programozási tételek két keresésére.)

```
1. keresés:
          I:=1
          Ciklus amig A(I)<X
                      I:=I+1
          Ciklus vége
Keresés vége.

2. keresés:
          A:=1 : F:=N
          Ciklus
                      K:=INT((A+F)/2)
                      Ha A(K)<X akkor A:=K+1
                      Ha A(K)>X akkor F:=K-1
          amig A(K)<>X
          Ciklus vége
Keresés vége.
```

Az első algoritmus esetén egy szám megtalálásához a vizsgálatok átlagos száma kb. N/2, a második esetben pedig N 2-es alapú logaritmusával arányos. Tehát a hatékonyabb algoritmus egyben bonyolultabb is, így több helyet foglal és elkészítéséhez nagyobb programozási tapasztalat szükséges.

Második példánkban próbáljuk minimalizálni a ciklusmag végrehajtásainak számát.

Feladat: Döntsük el egy számról, hogy primszám-e!
A szám legyen az A változóban!

```
1. algoritmus:
   Ciklus I=2-töl N-1-ig
      Ha I osztója A-nak akkor Ki: "Nem prim" :
                                Eljárás vége
   Ciklus vége
   Ki: "Prim"
Eljárás vége.
```

Csökkentsük a ciklusmag végrehajtásainak számát! Egyrészt tudjuk, hogy a legnagyobb osztó nem lehet N/2-nél nagyobb, így az N–1-es határt máris N/2-re csökkenthetjük. Másrészt azt is tudjuk, hogy ha van osztó, akkor közülük a legkisebb még az adott szám négyzetgyökénél sem lehet nagyobb, így a felső határt tovább csökkenthetjük: 92. lista

```
2. algoritmus:
   Ciklus I=2-töl N-ig
      Ha I osztója A-nak akkor Ki: "Nem prim" :
                                Eljárás vége
   Ciklus vége
   Ki: "Prim"
Eljárás vége.
```

Gyakori módszer a hatékonyabbra írásra, hogy a program már kiszámolt értékeit indexelésre használjuk, amivel sok feltételes utasítást takaríthatunk meg. Vizsgáljuk meg a következő két algoritmust:

1. algoritmus:

```
        Ciklus I=1-töl 100-ig
                X:=RND(6)
                Ha X=1 akkor S1:=S1+1
                Ha X=2 akkor S2:=S2+1
                Ha X=3 akkor S3:=S3+1
                Ha X=4 akkor S4:=S4+1
                Ha X=5 akkor S5:=S5+1
                Ha X=6 akkor S6:=S6+1
        Ciklus vége
        Ki: S1,S2,S3,S4,S5,S6
Eljárás vége.
```

2. algoritmus:

```
        Ciklus I=1-töl 100-ig
                X:=RND(6)
                S(X):=S(X)+1
        Ciklus vége
        Ki: S(1),S(2),S(3),S(4),S(5),S(6)
Eljárás vége.
```

Ebben az esetben, tehát az algoritmus (és így a program) szövege is rövidebb lett és az átlagos végrehajtási időt is jelentősen csökkentettük.

Most egy *formális módszert* ismertetünk a végrehajtási idő várható értékének kiszámítására:

1. Határozzuk meg a bemenő adatok eloszlását!
2. Határozzuk meg a program részeredményeinek eloszlását!
3. Az előbbiek segítségével határozzuk meg a programon átvezető utak végrehajtásának valószínűségét! $P(u_i)$
4. Határozzuk meg az egyes utak végrehajtási idejét! $t(u_i)$
5. Ezek felhasználásával számoljuk ki a várható értéket!

$t_á = \Sigma\, t(u_i) * P(u_i)$

Ez a módszer kis programoknál tökéletesen, ill. nagy programoknál a 3. és a 4. pont kivételével elvégezhető. Nagyobb és bonyolultabb programoknál azonban a kézi módszerek nem megfelelőek, túlságosan sokáig tartanak.

Bonyolultabb esetekben a program speciális tesztesetekkel való lefuttatására van szükség, és a ténylegesen mért végrehajtási idők segítségével kell becsülni az átlagos végrehajtási időt. Itt a teszteseteknek a megválasztása lényeges, amihez szintén ismerni kell a bemenő adatok várható eloszlását.

Tesztadatválasztásnál megismertük az ekvivalenciaosztályok módszerét, amelyben a lehetséges bemenő adatokat osztályokba soroltuk, majd minden osztályból egy adattal (adatcsoporttal) próbáltuk ki a programot. Most is osztályokra kell bontani a lehetséges adatokat, de úgy, hogy az egyes osztályokba tartozó adatokra nagyjából azonos végrehajtási időt kapjunk. Eddig a módszer megegyezik a formális módszerrel, az újdonság ezután az, hogy egy-egy ilyen adattal a programot valóban kipróbáljuk, s a mért értéket használjuk fel mérőszámnak.

A csillagsűrűsödéses feladatban a vizsgált csillagok számára az ekvivalenciaosztályok lehetnek a következők:
– 2...50 csillag – 50 %-os gyakorisággal;
– 51...100 csillag – 35 %-os gyakorisággal;
– 101... csillag – 15 %-os gyakorisággal.

Ahhoz, hogy felfedezzük programunkban a kevésbé hatékony részeket, szükségünk van a program részeinek végrehajtásához szükséges idő becslésére. (Tapasztalatok szerint a program futási idejének 90 %-áért a programszöveg 3–5 %-a a felelős!) Ehhez a programban időmérési kezdő- és végpontokat kell kijelölni és elvégezni az egyes tartományok mérését. Ezt a legegyszerűbb esetben valamilyen figyelemfelhívó utasítással (kiírás a képre, hangjelzés) lehet a felhasználóra bízni, olyan gépeken pedig, ahol van belső óra, ott automatikus időmérést végezhetünk.

Példa a helyfoglalás optimalizálására

A következő példában egy, a radioaktív bomlást szimuláló algoritmust vizsgálunk. Legyen kezdetben N db atom! Az A atom időegységenként P valószínűséggel bomlik B-re. Az atomokat betűjeleikkel azonosítjuk.
1. változat: Helyezzük el az egyes atomok betűjeleit az A$(N) vektorban!

```
Eljárás:
  Ciklus I=1-töl N-ig
    Ha RND<P és A$(I)="A" akkor A$(I):="B" : B=:B+1
    Ki: A$(I)
  Ciklus vége
  Ki: N-B,B
Eljárás vége.
```

A BASIC programra gondolva, egy szöveg típusú változó (tömbelem) tárolásához annyi byte-ra van szükségünk, ahány betűt elhelyezünk benne. Ez most 1. Ezenkívül általában 4 (pl. HT–1080Z) vagy több (6 – ABC80) byte kell a BASIC értelmezőnek. Egy egész típusú változónak nem szükséges ennyi hely, így a következő módosítást végezzük el:

2. változat: Helyezzük el az egyes atomokat az A%(N) vektorban, 1 jelöli az A betűt, 2 pedig a B-t!

```
Eljárás:
    Ciklus I=1-töl N-ig
        Ha RND<P és A%(I)=1 akkor A%(I):=2 : B:=B+1
        Ha A%(I)=1 akkor Ki: "A" különben Ki: "B"
    Ciklus vége
    Ki: N-B,B
Eljárás vége.
```

Megállapíthatjuk, hogy ettől a módosítástól a programban elhelyezett kiírás lett bonyolultabb.
A szövegek ábrázolásából adódik a következő módosítás gondolata: ha egy változóba eggyel több betűt helyezünk el, az a felhasznált helyet egy byte-tal növeli.

3. változat: Tároljuk az egyes atomokat az A$ változóban!

```
Eljárás:
    Ciklus I=1-től N-ig
        Ha RND<P és A$ I. betüje="A" akkor
                            Cseréljük "B"-re: B:=B+1
        Ki: A$ I. betüje
    Ciklus vége
    Ki: N-B,B
Eljárás vége.
```

Ezzel a módosítással egy olyan algoritmust kaptunk, amelyet sokkal nehezebb megírni (a ZX–81, ZX Spectrum kivételével), továbbá a futási ideje is hosszabb lesz.

Az utolsó változatban lemondunk az egyes atomok egyedi tárolásáról, így információt vesztünk, viszont cserébe rengeteg helyet kapunk és a végrehajtási idő is sokkal kisebb lesz.

4. változat: Tároljuk az egyes atomok számát A-ban és B-ben!

```
Eljárás:
    Ciklus I=1-től A-ig
        Ha RND<P akkor B:=B+1
    Ciklus vége
    A:=N-B
    Ki: A,B
Eljárás vége.
```

A bonyolultság, ill. a kód hosszának csökkentésére két egyszerű elvet mondunk ki. Az első *a kivételes eset kiküszöbölésének elve,* a második pedig *a fiktív kezdőértékadás elve.*

A kivételes eset kiküszöbölése.

Ha a feladat pl. egy szövegben a szavak számának meghatározása, akkor a következőképpen járhatunk el. A szavak száma megegyezik a szavak kezdeteinek számával. Egy szó úgy kezdődik, hogy szóköz után betű következik, kivéve az első szót. Ha a szöveg első karaktere szóköz, akkor az előbbi állítás az első szóra is igaz, ha betű, akkor nem (ez a kivételes eset). Helyezzünk el ekkor a szöveg első karaktere elé egy szóközt, így a kivételt megszüntethetjük, a megoldó programot rövidebbé, egyszerűbbé tehetjük.

Megjegyzés: Hasonlóan, egy 1-től indexelt vektor 0. elemébe helyezünk el egy megfelelő értéket.

Ugyanezt az elvet szemléltetjük a kiválasztási algoritmus átalakításával. Korábban az algoritmus ciklusfeltételében két dolgot kellett vizsgálni: azt, hogy van-e még vizsgálandó elem, s hogy megtaláltuk-e a keresettet. Helyezzük az N+1. elem helyére a keresettet, így biztosan meg tudjuk találni, s ha a legvégén találtuk meg, akkor nem volt az eredeti sorozatban. Az új algoritmus (van-e X elem az A(N) sorozatban?):

```
Eljárás:
    I:=1 : A(N+1):=X
    Ciklus amíg A(I)<>X
        I:=I+1
    Ciklus vége
    VAN_E:= I<=N
Eljárás vége.
```

A fiktív kezdőértékadás.

Az elv bemutatására a maximumkiválasztás programját módosítjuk, ahol az N elemű A( ) vektor maximális értékű elemének az értékét kell megadni:

```
Eljárás:
     ERTEK:= A() alsò korlátja
     Ciklus I=1-töl N-ig
          X:=A(I)
          Ha ERTEK<X akkor ERTEK:=X
     Ciklus vége
Eljárás vége.
```

A megoldás lényege: ha A( ) egy elemének bonyolult a meghatározása, akkor azt csak egyszer végezzük el, s kiinduló értéknek egy olyan fiktív értéket választunk, amelyet az algoritmus biztosan módosít.

A PROGRAMKÓD HATÉKONYSÁGA

A kód hatékonyságának javításával csak ciklusok belsejében érdemes foglalkozni, akkor, amikor egy utasítást sokszor végre kell hajtani. Eléréséhez adunk néhány jó tanácsot:

1. Használjunk a programban gyorsabb számolást és kisebb helyfoglalást lehetővé tevő adattípusokat!
Például valós számok helyett egészek, szöveg típusú változóban tárolt számok helyett szám típusú változók stb.

2. Használjunk *takarékos* műveleteket!
Például A^2 helyett A∗A, 2∗A helyett A+A.

3. A feltételek egyszerűbb alakra hozása.
Például
– (nem A > B) *helyett* A <=B;
– I=1 és J=1 vagy I=N és J=1 vagy I=1 és J=M vagy I=N és J=M *helyett* (I=1 vagy I=N) és (J=1 vagy J=M).

4. Kerüljük – ha lehet – a különböző típusú adatokkal való műveletvégzést!

5. Gyakran a függvények többszöri kiszámítása lassítja a programot, így törekedjünk csökkentésükre.
Például két program a következő sorozat kiszámítására (hasonló a Fibonacci-számokhoz):

$a_n = \sin a_{n-1} + \sin a_{n-2}$
$a_0 = 1 \quad a_1 = 1$

```
100 A=1 : E=1
110 U=SIN(A)+SIN(E)
120 PRINT U;
130 E=A : A=U
140 GOTO 110
```

```
100 A=SIN(1) : E=SIN(1)
110 U=A+E
120 PRINT, U;
130 E=A : A=SIN(U)
140 GOTO 110
```

Megjegyzés: Az első változat kb. kétszer annyiszor számítja ki a szinusz függvény értékét, mint a második.

6. Egyszerűsítsük az aritmetikai kifejezéseket!
Például exp(A) * exp(B) => exp(A+B)
sin(X) / cos(X) => tan(X).

Bevezetünk néhány *programtranszformációt*, amelyek segítségével helyes programunkat hatékonyabbá tehetjük. A következő algoritmusokban a feltételeket F-fel, a ciklusfeltételeket CF-fel, az utasításokat U-val jelöljük. A példákban az eredeti és a hatékonyabbra átírt programokat formailag másként helyezzük el.

```
1.      Ha F akkor U1 : U2  ---->  U1
            különben U1 : U3        Ha F akkor U1
                                        különben U3

        Pl.:    100 IF A>B THEN X=A+B : Y=A-B
                           ELSE X=A+B : Y=0

        100 X=A+B
        110 IF A>B THEN Y=A-B ELSE Y=0

2.      Ha F akkor U1 : U3   ---->  Ha F akkor U1
            különben U2 : U3                különben U2
                                        U3

        Pl.:    100 IF A<B THEN A=B : PRINT A
                           ELSE B=A : PRINT A

        100 IF A<B THEN A=B ELSE B=A
        110 PRINT A

3.      Ha F1 és F2 akkor U   ----> Ha F1 akkor
                                      Ha F2 akkor U

        Pl.:    100 X=RND(90)
                110 IF X=90 AND MID$(A$,1,1)=" "
                       THEN A$=MID$(A$,2)

        100 X=RND(90)
        110 IF X=90 THEN IF MID$(A$,1,1)=" "
                THEN A$=MID$(A$,2)
```

```
4.          Ha F1 akkor U1 különben --> Ha F1 akkor U1
              Ha F2 akkor U2                  különben U2



            Pl.:     100 X=RND(0)
                     110 IF X<0.5 THEN F=F+1 ELSE
                              IF X>=0.5 THEN I=I+1

            100 X=RND(0)
            110 IF X<0.5 THEN F=F+1 ELSE I=I+1

5.          Ha F akkor U1     ---->    Ha F akkor U1 : U2
            Ha F akkor U2



            Pl.:     100 IF A(I,J)>0 THEN S=S+1
                     110 IF A(I,J)>0 THEN PRINT I;J

            100 IF A(I,J)>0 THEN S=S+1 : PRINT I;J

6.          Ciklus amig CF        --->       Ciklus amig CF
                     U1 : U3                           U1 : U2
            Ciklus vége                                U3
            Ciklus amig CF                   Ciklus vége
                     U2 : U3
            Ciklus vége

            Pl.:     100 MIN=A(1)
                     110 FOR I=2 TO N
                     120   IF MIN>A(I) THEN MIN=A(I)
                     130 NEXT I
                     140 MAX=A(1)
                     150 FOR I=2 TO N
                     160     IF MAX<A(I) THEN MAX=A(I)
                     170 NEXT I

            100 MIN=A(1) : MAX=A(1)
            110 FOR I=2 TO N
            120   IF MIN>A(I) THEN MIN=A(I)
            130   IF MAX<A(I) THEN MAX=A(I)
            140 NEXT I
```

```
7.      U                       --->    Ciklus
        Ciklus amig CF                          U
                U                       amig CF
        Ciklus vége                     Ciklus vége

        Pl.:    10 S=0
                20 INPUT A
                30 IF A<0 OR A>5 THEN 20
                40 IF A=0 THEN 90
                50   S=S+A
                60   INPUT A
                70   IF A<0 OR A>5 THEN 60
                80 GOTO 40
                90 ...

        10 S=0
        20    INPUT A
        30    IF A<0 OR A>5 THEN 20
        40    S=S+A : REM A=0 NEM ZAVAR
        50 IF A<>0 THEN 20
        60 ...

8.      Ciklus amig CF          ---->   Ha F akkor
                Ha F akkor U              Ciklus amig CF
        Ciklus vége                                U
                                          Ciklus vége
                                        Elágazás vége


        Pl.:    100 S=0
                110 FOR I=1 TO N
                120   IF A(1)>0 THEN S=S+A(I)
                130 NEXT I

        100 IF A(1)<=0 THEN 150
        110 S=0
        120 FOR I=1 TO N
        130   S=S+A(I)
        140 NEXT I
        150 ...

9.      Ciklus amig CF        ---->    U1
                U1                     Ciklus amig CF
                U2                             U2
        Ciklus vége                    Ciklus vége
```

Pl.:

```
100 FOR H=HO TO 0 STEP -1
110   EH=M*G*H
120   EM=M*G*HO-M*G*H
130   PRINT H,EH,EM
140 NEXT H
```

```
100 X=M*G : Y=M*G*HO
110 FOR H=HO TO 0 STEP -1
120   EH=X*H : EM=Y-EH
130   PRINT H,EH,EM
140 NEXT H
```

Megjegyzés: Az átírt változatban Y=X*HO is lehetne, de ez a hatékonyságot nem befolyásolja lényegesen, a közölt megoldás pedig szemléletesebb.

Ha a programot átírtuk hatékonyabbra, elképzelhető, hogy a javításokkal hibákat is helyeztünk el benne (legtöbbször sajnos éppen ez a helyzet áll fönn). Alapvető követelmény, hogy az átírt program ugyanazokat az eredményeket adja, mint a korábbi változat. Erről meggyőződni csak úgy lehet, ha lefuttatjuk az összes korábbi tesztadattal. Ilyenkor (is) jó az, ha a tesztelés módját és eredményeit leírtuk, mert akkor az eredményeket könnyen összehasonlíthatjuk a korábbiakkal.

Ha a program a használat során helyesnek bizonyul, akkor is szükség lehet módosításra. Ez akkor fordul elő, ha a felhasználójának újabb igényei születnek, egyes részfeladatokat másképp szeretne megoldani, hasonló feladat megoldására szeretné a programot alkalmazni. Ekkor egy meglevő, helyes programból egy újabbat kell előállítani, amely megfelel a módosított igényeknek.

Két megközelítésmódot követhetünk. Az elsőben a két programot egy közös ős leszármazottjának tekintjük. A két feladat (az eredeti és a módosított) valamilyen szintig azonos megoldást igényel, s ettől a szinttől kezdve érvényesülnek az eltérések. Célszerű ilyenkor a közös őst külön is elkészíteni, majd a közös alatti szinteken a korábbi tervezés során hozott egyes döntéseket felülbírálni, az új részeket teljesen kidolgozni.

A másik megközelítésben – ahol a feladatok között sokkal nagyobb rokonság áll fönn – megpróbáljuk a módosításokat a korábbi megoldás néhány, de nem feltétlenül a legalsó szintjére korlátozni. Mindkét módszer lényeges eleme, hogy a módosításokat nem a program szövegében kezdjük elvégezni. Azzal ui. súlyos hibákat okozhatunk, amelyek kijavítása tovább tarthat, mint az egyes részek újratervezése. A másik tény, ami a közvetlen programszövegbeli javítás ellen szól: a sok meggondolatlan belejavítástól a program szövege egyre áttekinthetetlenebb lesz.

Ha eleve számítunk ilyen programcsaládok létrehozására, akkor célszerű az első módszert követni. Tehát a feladatot a várható *feltételeltérések* alapján feladatosztályra szélesítjük, majd hozzáfogunk a feladat (családot)ot megoldó program felső szintjeinek meghatározásához. Igyekszünk az egyes szinteknél a lehető legáltalánosabban fogalmazni (*nyílt rendszerű felépítés* elve), s ameddig csak lehet, halogatni az egyes feladatok felé kanyarodó döntéseket (*a döntések elhalasztásának* elve). Így jutunk el a feladatcsaládhoz tartozó közös ős programhoz.

Ha pedig utólagos – és egyedi – igény merült fel, akkor a kisebb ráfordítást követelő második módszert használjuk.

Olyan feladatokat fogunk kitűzni, amelyek közeli vagy távoli rokonságban vannak korábbi feladatainkkal.

Módosítsuk a csillagsűrűsödéses feladatot!

Feladat: Adott egy csillagtérkép, ahol a csillagokat a koordinátáikkal adjuk meg (háromdimenziós térkép). Csillagsűrűsödésnek nevezzük azokat a lehető legnagyobb csoportokat, amelyekben legalább S0 db csillag van és minden ott levő csillagra igaz, hogy el lehet jutni tőle a csoport bármely tagjához olyan úton, amely a csoport tagjain át vezet és a lépések hossza nem nagyobb S1 fényévnél (elég sűrűn vannak a csillagok). Határozzuk meg a térképen a csillagsűrűsödések számát és a csillagsűrűsödésbe tartozó csillagokat!

Mi az, ami változik a korábbi feladathoz képest? Ott, ahol növeltük eggyel a sűrűsödések számát, most még ki kell írni a sűrűsödésbe tartozó csillagok sorszámait is. Ezeket a korábbi megoldásban is meghatároztuk. Így csak a megoldás 3. szintje fog változni:

```
                                          (3. szint)
Az I. csillagot tartalmazó sürüsödés vizsgálata:
   CSSZAM:=0
   Az I. csillagot tartalmazó csoport vizsgálata
   Ha CSSZAM>=SO akkor SSZAM:=SSZAM+1  :
                          Csillagok kiirása
Eljárás vége.
```

Elkészítünk egy új eljárást, amely a csoportba tartozó csillagok darabszámát és sorszámait írja ki:

```
                                           (4.2. szint)
Csillagok kiirása:
     Ki: SSZAM
     Ciklus X=1-töl CSSZAM-ig
          Ki: CS(X)
     Ciklus vége
Eljárás vége.
```

Látható, hogy most egy nagyon közeli rokon megoldást készítettünk. Módosítsuk tovább a feladatot!

Feladat: Most az eredmény a kettőscsillagok száma legyen, minden más feltételt hagyjunk változatlanul!
Ebben az esetben is a 3. szinten kell változtatni:

```
                                           (3. szint)
Az I. csillagot tartalmazò sürüsödés vizsgálata:
     CSSZAM:=0
     Az I. csillagot tartalmazò csoport vizsgálata
     Ha CSSZAM=2 akkor SSZAM:=SSZAM+1
Eljárás vége.
```

Újabb hasonló feladatot fogunk vizsgálni.

Feladat: Az eredmény legyen a sűrűsödésbe nem tartozó csillagok száma.
Még mindig a 3. szintet módosítjuk.

```
                                           (3. szint)
Az I. csillagot tartalmazò sürüsödés vizsgálata:
     CSSZAM:=0
     Az I. csillagot tartalmazò csoport vizsgálata
     Ha CSSZAM<SO akkor SSZAM:=SSZAM+CSSZAM
Eljárás vége.
```

Ezután kicsit távolabbi rokonokkal foglalkozunk.

Feladat: Írjuk ki a legnagyobb sűrűsödés csillagszámát!
A megoldás 3. szintje egy maximumkiválasztással bővül, a maximális értéket tartalmazó változó kezdőértéke legyen 0! Ezt az értékadást még az 1. szinten el kell végezni!

```
Program:                                   (1. szint)
     Az adatok beolvasása
     MAX:=0
     A maximális sürüsödésszám meghatározása
     Ki: MAX
Program vége.
                                           (3. szint)
Az I. csillagot tartalmazò sürüsödés vizsgálata:
     CSSZAM:=0
     Az I. csillagot tartalmazò csoport vizsgálata
     Ha CSSZAM>=SO akkor
                  Ha CSSZAM>MAX akkor MAX:=CSSZAM
Eljárás vége.
```

Olvasónkra bízzuk azt a módosítást, amikor a legnagyobb sűrűsödés csillagait is ki kell írni!

A következő feladatokhoz felépítünk egy programcsaládot.

Ismerünk N darab szülő–gyerek kapcsolatot.

1. **feladat**: Keressük meg egy adott ember gyerekeit!
2. **feladat**: Keressük meg egy adott ember szüleit!
3. **feladat**. Keressük meg egy adott ember testvéreit!
4. **feladat**: Keressük meg egy adott ember őseit!
5. **feladat**: Keressük meg egy adott ember leszármazottait!
6. **feladat**: Keressük meg egy adott ember összes rokonát!

A megoldások közös részéhez tartozik az adatábrázolás megválasztása. Legyen N a szülő–gyerek kapcsolatok száma, a kapcsolatokat tároljuk az X(N,2) mátrixban! Itt emberek sorszámait fogjuk elhelyezni, az első oszlop minden egyes kapcsolatban a szülőt, a második pedig a gyereket azonosítja. Tároljuk az egyes feladatoknak megfelelő emberek sorszámait az S(2*N) vektorban! D jelezze a sorra vett emberek számát, a vektor első eleme pedig annak a sorszáma legyen, akinek valamilyen rokonait kell megadni (beolvasáskor K-val jelöljük)!

A megoldás legfelső szintje azonos az összes feladatnál:

```
Program:                                          (1. szint)
    Az adatok beolvasása [N, X(N,2), K]
    D:=0
    A megfelelö tulajdonságu emberek  kiválogatása
        [ az S() vektor elsö D eleme tartalmazza a
          sorszámokat ]
    Az eredmény kiirása
Program vége.
```

Az adatok beolvasását és az eredmény kiírását nem részletezzük (ezek ráadásul az algoritmus szintjén azonosak is), csak a kiválogatással foglalkozunk.

Az első két feladatban olyan sorszámokat kell keresnünk, amelyek párja egy adott érték a mátrix valamelyik sorában. A 4., 5., 6. feladatban az így kapott sorszámú emberekhez újra kell keresni ilyen párokat. A legbonyolultabb a 3. eset. Itt ui. először meg kell határozni a keresett ember szüleit, majd pedig a szülők összes gyerekét. Így a 2. szintből három változatot készítünk:

```
                                          (2.(1,2) szint)
A megfelelö tulajdonságu emberek kiválogatása:
    A K. ember felvétele S()-be
    Az S(1). ember megfelelö rokonai felvétele
Eljárás vége.
```

Az összes megfelelő rokon kiválogatását a korábbi feladatokban megismert SOR nevű adatszerkezet felhasználásával oldjuk meg (első, második példa a módszeres programozásra).

```
                                        (2.(4,5,6) szint)
A megfelelö tulajdonságu emberek kiválogatása:
    A K. ember felvétele S()-be
    A:=1
    Ciklus amig A<=D
        Az S(A). ember megfelelö rokonai felvétele
        A:=A+1
    Ciklus vége
Eljárás vége.
```

A harmadik változatban a megtalált szülőket átmásoljuk az S( ) második felébe, majd megkeressük a gyerekeiket.

```
                                      (2.(3) szint)
A megfelelö tulajdonságu emberek kiválogatása:
   A K. ember felvétele S()-be
   Az S(1). ember szülei felvétele
   D1:=D
   Ciklus I=2-töl D-ig            ; az elsö szülöt az N+1.
      S(I+N-1):=S(I)              ; helyre tesszük
   Ciklus vége
   D:=1                           ; az elsö ember marad
   Ciklus I=N+1-töl N+D-ig
      Az S(I). ember gyerekei felvétele
   Ciklus vége
Eljárás vége.
```

A 6. feladat kétféle típusú rokonságot enged meg, míg a többiek most már csak egyfélét. Így a harmadik szintet csak ennél a feladatnál dolgozzuk ki.

```
                                      (3. (6) szint)
Az S(A). ember megfelelö rokonai felvétele:
   Az S(A). ember gyerekei felvétele
   Az S(A). ember szülei felvétele
Eljárás vége.
```

A többi feladatnál a megfelelő rokonok megegyeznek az adott ember gyerekeivel, ill. szüleivel, így ezt ezen a szinten nem részletezzük.

A negyedik szinten újra hasonló feladatokat kell megoldani, itt aszerint botjuk szét a megoldásokat, hogy szülőket vagy gyerekeket kell keresni. Alkalmazzuk a kiválogatásra megismert tételt, megoldásunk a következő lesz:

```
                              (4.1 (2,3,4,6) szint)
Az X. ember szülei felvétele:
   Ciklus L=1-töl N-ig
      Ha X(L,2)=X akkor Az X(L,1). felvétele
   Ciklus vége
Eljárás vége.
```

```
                              (4.2 (1,3,5,6) szint)
Az X. ember gyerekei felvétele:
   Ciklus L=1-töl N-ig
      Ha X(L,1)=X akkor Az X(L,2). felvétele
   Ciklus vége
Eljárás vége.
```

Maradt a legutolsó, 5. szint elkészítése, amely mindegyik feladatnál azonos. Egy adott sorszámú embert akkor kell felvenni, ha még nem szerepel az S( ) vektorban.

```
                                          (5. szint)
Az X. ember felvétele:
    Ha   S(J)<>X [J=1,2,...D]   akkor
                    D:=D+1  :  S(D):=X
Eljárás vége.
```

Ezzel elkészült a teljes programcsalád. Természetesen a kódolás során is érdemes követni az előbbi módszert, így azt a munkát is megkönnyíthetjük, s biztonságossá tehetjük. Célszerű a programok beírását a közös részek elkészítésével és felvételével kezdeni, majd ehhez egyenként beírhatók az egyes megoldások egyedi részei, így sok gépelési munkától is mentesülünk.

Javasoljuk Olvasónknak, hogy az egyes feladatokhoz elkészült algoritmusokat külön-külön is írja le! Így még szembetűnőbb lesz hasonlóságuk és egyszerűségük.

Megjegyzések:

– Bár a 2. szinten az egyes megoldások jelentősen eltérnek egymástól, mégis azt tapasztaltuk, hogy a 4. szint új eljárásai már szinte azonosak, az 5. szinten pedig teljes azonosság áll fenn. Ez a jelenség nem csak ebben a példában fordul elő. Vannak olyan feladatosztályok, amelyek megoldásait közös elemi utasításkészletből tudjuk megoldani. Tapasztalt programozók ilyenkor az *alulról felfelé tervezés* módszerét is követhetik: az adott nyelv elemi utasításaiból felépíthetik a programcsalád valamely szintjének elemi utasításait és csak ezután foglalkoznak az egyes családtagok elkészítésével. Természetesen e módszer alkalmazásához nagyfokú előrelátás szükséges, hogy valóban azokat a tevékenységeket határozzuk meg, amelyekre a megoldás során szükségünk lesz.

– Programcsaládok készítése a tesztelési/hibakeresési munkát is jelentősen megkönnyíti, hiszen a közös részek kipróbálását/kijavítását elég a család egyetlen tagjával elvégezni, s a többieknél elég már csak az egyedi részt vizsgálni.

NÉHÁNY SZÓ A FELADATMEGHATÁROZÁSRÓL

Ha a megoldás algoritmusát elkészítettük, majd a programot kódoltuk, leteszteltük, ezenkívül hatékonysága is jó, akkor vajon mondhatjuk-e, hogy a feladatot jól megoldottuk? Sajnos nem mindig! Előfordulhat, hogy nem értettük meg pontosan a feladatot, és így mást csináltunk, mint amit kellett volna.

A feladatmeghatározás hiánya szokta a legtöbb vitát okozni a programozó és a feladat kitűzője között. Érdekes, hogy az ellentmondás akkor is fellép, ha e két ember egy és ugyanaz. Ennek többnyire az az oka, hogy az elvárások nincsenek átgondoltan megfogalmazva, a rendelkezésre álló adatok nincsenek összhangban, az algoritmizálhatóság szempontjából a fogalmak nincsenek tisztázva. A sikeres feladatmeghatározás tehát erősen befolyásolja a programkészítésre fordított időt, és a kész program alkalmazhatóságát.

Mitől rosszak a feladatmeghatározások? A legtöbb esetben maga a feladat nem teljesen tisztázott, a használt fogalmak nem egyértelműek, a szöveges leírás félreérthető. Sokszor a feladat kitűzője nem ért a számítástechnikához, ezért nem tudja megítélni, hogy mit, s mennyit kell leírnia.

Miből áll a feladatmeghatározás? Szerepeljen benne mindaz, ami a feladat pontos megoldásához szükséges. Adjuk meg:

– a bemenő adatokat, s összefüggéseiket;
– az eredményeket, s kiszámítási szabályukat;
– a megoldással szembeni követelményeket (forma, sebesség, ...);
– a megoldást korlátozó tényezőket!

Milyen legyen a feladatmeghatározás? Mindenképpen olyannak kell lennie, amelyből a program készítője pontosan azt a feladatot és pontosan úgy oldja meg, ahogyan a feladat kitűzője várta. Három, egymástól eltérő követelményeket állító szempontrendszer adható meg. Az első szerint a feladatmeghatározás legyen egyértelmű, pontos, teljes (ne legyen definiálatlan fogalom). A második szerint legyen rövid, tömör (ha lehetséges, akkor formalizált). A harmadik szemléletes, érthető, tagolt formát követel. A baj abból származik, hogy általában az egyértelmű, pontos feladatmeghatározás hosszú, s így nehezen áttekinthető. A tömör, formális feladatmeghatározás ugyan pontos lehet, de sajnos, nem szemléletes. Ami pedig szemléletes, az legtöbbször se nem pontos, se nem egyértelmű. Mindezekből nyilvánvaló, hogy valamilyen kompromisszumot kell találni.
Például (nem törekszünk a teljes formalizálásra):

Feladat: Ismerjük adott számú ember magasságát! Határozzuk meg a legmagasabb sorszámát és magasságát!

Jelölések: N – az emberek száma;
X(N) – a magasságok;
LM – a legmagasabb magassága;
LS – a legmagasabb sorszáma.

A program N és X(N) ismeretében kiszámítja LM és LS értékét.
Bemenet: N természetes szám és X(I)>0 valós számok [I=1,2,...,N].
Eredmény: LM >= X(I) [I=1,2,...,N] és
1 <= LS <= N és LM = X(LS).
Egyéb feltételek: N, X(N) változatlan marad. Ha több egyforma magasságú is van, akkor elég egyet megadni.

PÉLDA A FELADATMEGHATÁROZÁS BONYODALMAIRA

Vegyük a már jól ismert csillagsűrűsödéses feladatot! Ezt eddig így fogalmaztuk meg: *Adott egy csillagtérkép, ahol a csillagokat koordinátáikkal adjuk meg (háromdimenziós térkép). Csillagsűrűsödésnek nevezzük azokat a lehető legnagyobb csoportokat, amelyekben legalább S0 db csillag van és minden ott levő csillagra igaz, hogy el lehet jutni tőle a csoport bármely tagjához olyan úton, amely a csoport tagjain át vezet, és a lépések hossza nem nagyobb S1 fényévnél (elég sűrűn vannak a csillagok). Határozzuk meg a térképen a csillagsűrűsödések számát!*

Itt a fogalom, amit definiálni kell, a csillagsűrűsödés.

Megjegyzés: E definíció alapján egy csillag sem sorolható be több sűrűsödésbe! A sűrűsödések nem fedik át egymást.

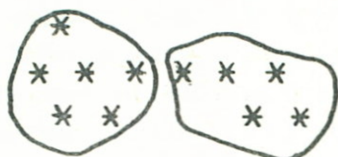
A feladat meghatározása szerint megfelelő távolságot és a 3 csillagszámot választva a következő térképen 2 csillagsűrűsödés van (11. ábra).

11. ábra

A csillagsűrűsödésbe tartozás feltételét megadhatjuk másként is, de a darabszámos feltételt változatlanul hagyjuk:

Most azokat a lehető legnagyobb csoportokat hívjuk sűrűsödésnek, amelyek bármely két tagja S1 fényévnél közelebb van egymáshoz. (Mondhatnánk úgy is, hogy befoglalhatjuk őket egy S1 fényév átmérőjű gömbbe.) A sűrűsödések nem átfedőek.

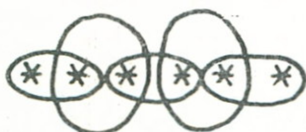
Ebben az esetben az előbbi példában alkalmas távolságot és az 5 csillagszámot választva már csak egy sűrűsödést kapunk, ellenben a következő ábrán nem egyet, hanem kettőt (12. ábra).



12. ábra

Sőt előfordulhat az a furcsa eset is, hogy egyes csillagok több halmazba is besorolhatók, s hogy végül is melyikbe tartoznak, azt a vizsgálat sorrendje dönti el.

Ha a sűrűsödések átfedhetik egymást, akkor az eredetileg egy csillagsűrűsödés helyett már akár 5 sűrűsödéstkapunk (13. ábra).

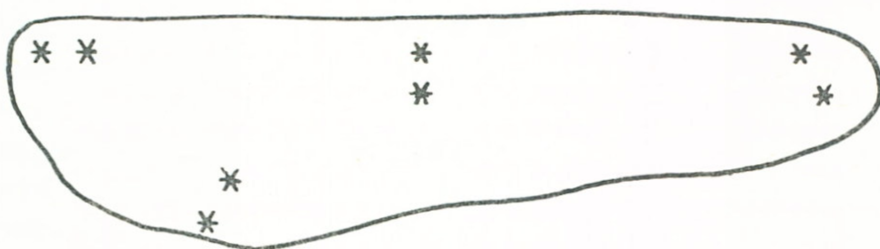


13. ábra

Újabb definíciónk még furcsább eredményeket fog adni, most is változatlan darabszámos feltétellel dolgozunk:

Azokat a lehető legnagyobb csoportokat hívjuk sűrűsödésnek, amelyekben bármely csillaghoz van S1 fényévnél közelebbi csillag.

Ez a feltétel első ránézésre nagyon hasonlít a legelső definíciókra, mégis mennyire mást jelent! Bármilyen meglepő is, de ezzel a feltétellel a következő csillagtérképen egyetlen sűrűsödést találunk, amelybe az összes csillag beletartozik (14. ábra).

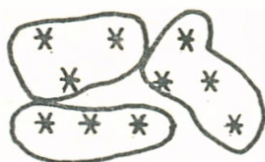


14. ábra

Nézzünk most egy másik részfeltételt. Amennyiben elhagyjuk a lehető legnagyobbra vonatkozó megszorítást, úgy a feladatnak egy újabb variációját kapjuk:

Azokat a csoportokat hívjuk sűrűsödésnek, amelyek legalább S0 csillagból állnak és bármely tagjáról bármely tagjára el lehet jutni legfeljebb S1 fényéves lépésekkel a csoport tagjain keresztül. A sűrűsödések nem fedhetik át egymást.

Most egy újabb térképünkön megfelelő távolságot és a 3 csillagszámot választva már 3 sűrűsödést találhatunk (15. ábra).



15. ábra

Utolsó variációnkban a *legalább* szót módosítjuk és ezzel egy újabb feladathoz jutunk. *Azokat a lehető legnagyobb csoportokat hívjuk sűrűsödésnek, amelyekbe pontosan S0 db csillag tartozik (a legelső távolságkritériumot figyelembe véve).*

Ha csillagszámnak a 2-t választjuk, akkor a megfelelő távolság megadásával megkaphatjuk a térképen található kettőscsillagok számát (3) (16. ábra).

16. ábra

Mindezekből látható, hogy a megoldás helyessége már a feladat megértésének pillanatában eldőlhet, hiszen ha valamit félreértelmezünk, akkor egészen más programot készíthetünk, mint amire szükségünk van.

NÉHÁNY SZÓ A DOKUMENTÁCIÓRÓL

Hátra van még a programdokumentálás. Miért van egyáltalán szükség rá? Nos, két okból is: A programot
1. használni,
2. néha módosítani, továbbfejleszteni kell.

Talán azt kérdezi az Olvasó, hogy: „Miért kell életünket újabb papírhegyekkel nehezíteni? Hiszen az elveket betartva a felhasználót maga a program eligazítja. Minek kell a használat mikéntjét még papírra is rögzíteni? ..." E dohogás érthető, de a programdokumentálás mégis elkerülhetetlen. Megnyugtatásul sietünk előre jelezni, hogy nem újabb többletmunkára szeretnénk rábírni az Olvasót, hanem arra, hogy amit a feladat megoldása közben kitalál, amiről dönt, azt rögtön írja is le a dokumentációba. Tehát a dokumentáció a feladatmegoldás kezdetétől a program átadásáig elkíséri a program íróját. Térjünk rá a részletekre!

A használatot támogató programleírást *felhasználói dokumentációnak*, a továbbfejlesztés érdekében készítettet *fejlesztői dokumentációnak* hívják.

A felhasználói leírás

Nézzük meg, milyen feladatokat ró a leírásra a használhatóság!

Abból az alapelvből kell kiindulni, hogy a felhasználónak több hasonló feladatot megoldó program közül van lehetősége válogatni. Tehát mindent el kell követni, hogy választása az elvárásainak leginkább megfelelő programra essék. Ezért közölni kell a program *feladatának* pontos leírását.

Semmiképpen nem maradhat ki annak a *környezetnek* a leírása, amely szükséges a program zavartalan működéséhez. Így pl. a *tárigény*, vagy a hardver kellékek (magnó, televízió, botkormány), vagy a programozási segédeszközök (mint a HT–1080Z esetében a *bővítő* rutin stb.). Továbbá sokszor jó tudni, hogy a program milyen *módszert* követ a feladat megoldásában. Nevezetes módszereknél elegendő csak névre utalni, máskülönben megéri néhány mondatban összefoglalni a tulajdonságait. A program *nyelve* különösen lényeges a betöltésnél. Például a HT–1080Z esetén más parancs szolgál a BASIC nyelvű és más, a gépi kódú program betöltésére. Bár a módszerből is, a program nyelvéből is következtetni lehet a gyorsaságra, mégis egy átlagos *futási időt* meg kell adni a dokumentációban.

A következő fontos tanács, hogy a felhasználó tárgyhoz tartozó ismereteiről minél kevesebbet tételezzünk fel. Ez nem lebecsülés, hanem segítség.

A *betöltés módját* teljes alapossággal kell leírni (milyen paranccsal, s hogyan?)!

Példák:

HT–1080Z esetére:

```
>SYSTEM
*?Z80SUB
>CLOAD "A"
RUN
```

A Z80SUB nevű gépi program betöltése, a főprogram.

ZX Spectrum esetére:

```
LOAD "Z80SUB" CODE

LOAD "BETüK" CODE


LOAD "A PROG"
RUN
```

A Z80SUB nevű gépi program betöltése, a BETŰK nevű saját jelkészlet beolvasása, maga a főprogram.

Megjegyzés: Sokszor meglehetősen bonyolult lehet egy ilyen program összeállítása. Ezért nemcsak nagyvonalúság, udvariasság részünkről, hanem célszerű is egy ún. *betöltő* programot írni, ami automatikusan elvégzi e barátságtalan feladatot, s a számítástechnikában járatlan felhasználót nem terheli feleslegesen.

Példa: Az előbbi ZX Spectrum program betöltését az „A PROG" (a főprogram) következő kiegészítésével lehet automatikus betöltődésre és elindulásra bírni:

```
2 LOAD "Z80SUB" CODE : LOAD "BETüK" CODE
```

Ha mindezen kívül még készítünk egy képernyőtervet is, amelynek adatait elsőként olvastatjuk be, hogy a betöltés hosszú perceiben ne unatkozzék a felhasználó, akkor elmondhatjuk, hogy igazán barátságos programot készítettünk

```
1 LOAD "REKLáM" CODE SCREEN$
```

Sajnos nem minden BASIC-ben valósítható meg ez ilyen elegánsan, pl. a HT–1080Z-nél sem. Ezeknél jól bevált ötlet, hogy maga a BASIC program tartalmazza a gépi utasítások kódjait. Például az első sorban elhelyezett REM utasítás mögött.

E példa egy másik, nagyon fontos és gyakran elfelejtett tényt is figyelembe vesz: a program használata már a betöltéskor (sőt még korábban) elkezdődik, így a korábban felsorolt elvek betartására már itt is gondot kell fordítani. Esetenként kell eldönteni, hogy a biztonságosság érdekében egy darabban vagy pedig *betöltő* programmal segítve végezzük-e el e kényes tevékenységet. Másrészt a barátságosság jegyében nyújtunk-e némi felüdülést a felhasználónak a *nyitó* kép segítségével vagy sem.

A *futó program használatának* leírása azt jelenti, hogy megadjuk, milyen kérdéseket tesz fel a program, és mit felelhet rá a felhasználó. Figyelem! Ez nem teszi fölöslegessé a programba beépített segítséget, mivel nem várható el a felhasználótól, hogy állandóan a dokumentációból, mint egy *partitúrából* lesse a teendőket! A leírásnak tartalmaznia kell a program figyelmeztető jelzéseit, a hiba okát és kiküszöbölésének módját. Végül néhány, akár hibás felhasználói válaszokat is tartalmazó példafuttatást is mutassunk be!

A fejlesztői leírás

A fejlesztői leírás tartalmáról e könyvben már sok szó esett, így itt csak összefoglaljuk a korábban elhangzottakat. A fejlesztői leírás olyanoknak készül, akik a programon módosítani szeretnének, a programot más feladatra alkalmazni, vagy kiterjeszteni akarják. Ezért minden ehhez szükséges tudnivalót tartalmaznia kell.

A fejlesztői leírás készítése lényegesen egyszerűbb feladat, mint a felhasználói megírása, mivel a programkészítés végére érve szinte minden anyag elkészül hozzá. Nézzük, melyek is ezek:

– A *feladat* pontos meghatározása. Ezt még a feladat kitűzője adja, vagy a vele való megbeszélés során módosul.

– A megoldás *algoritmusa,* a program *kódja,* a *változók táblázata.* Ezek az algoritmuskészítés, a programkódolás szakaszában keletkeznek, a dokumentáláskor esetleg némi magyarázattal kell ellátni őket.

– A *tesztelés* leírása, azaz milyen bemenő adatokra, milyen eredménnyel *válaszol* a program. Arról is szóltunk már, hogy a tesztadatokat miért célszerű leírni – tesztelés megismétlése javítás, gyorsítás után, hibakereséskor. Ezek a szempontok a program módosításakor is fontosak, így a dokumentációban ezekre is szükség van.

– Szinte egyetlen, újonnan elkészítendő anyag a program *fejlesztési lehetőségeinek,* ill. korlátainak leírása. Ebből hamar kiderülhet, hogy egyáltalán érdemes-e hozzáfogni a módosításhoz.

– Ha a megoldás során valamilyen sajátos tervezési vagy kódolási *szabályokat* alkalmaztunk, akkor érdemes ezeket is tudatni a program esetleges továbbfejlesztőjével.

A dokumentáció formai követelményei

A dokumentáció elsődleges célja a program leendő felhasználóinak, továbbfejlesztőinek való segítségnyújtás. Ezért olyannak kell lennie, hogy minden számukra szükséges tudnivalóhoz könnyen hozzájuthassanak. Ehhez elsődleges szempont természetesen, hogy a dokumentáció mindezeket tartalmazza, de a használatát egyéb követelmények betartásával jelentősen megkönnyíthetjük. Ezek a következők:

– A dokumentáció ne legyen túl hosszú, hiszen egy program használatához senki sem akar egy *regényt* elolvasni.

– A dokumentáció ne legyen túl rövid, mert akkor tömörsége miatt érthetetlen lesz, s így használhatatlan.

– A dokumentáció legyen világosan tagolt, s a tagolás segítse elő az egyes tudnivalók gyors keresését.

– A dokumentáció legyen tömör, Olvasója ne vesszen el a részletekben.

– A dokumentáció legyen olvasható; a túlságos formalizálás az érthetőség rovására megy.

UTÓSZÓ

A könyv végére érve tekintsük át még egyszer a programkészítés lépéseit!

1. Feladatmeghatározás (mit kell megoldani?).
2. Algoritmuskészítés (hogyan kell megoldani a feladatot?).
3. Kódolás (eredménye a programozási nyelven megírt program).
4. Helyességbelátás (eredménye a helyesen működő program).
5. Minőség- és hatékonyságvizsgálat, javítás (eredménye a minden igényt kielégítő, jó program).
6. Dokumentálás (eredménye a mások által is használható program).

Felsorolásunk bizonyos sorrendiséget is jelöl, de felhívjuk a figyelmet arra, hogy az egyes tevékenységek időben átfedik egymást, olyannyira, hogy a dokumentálást már a feladatkitűzés pillanatában el kell kezdeni.

Mindazon elvek és módszerek, amelyekről e könyvben szó esett, a professzionális programozási kultúrának is részei. Mi igyekeztünk úgy bemutatni és kiegészíteni őket, hogy ez nem számítástechnikai szakemberek számára is világos és hasznos legyen. Természetesen a programozó matematikusok még nagyszámú, újabb elvet és módszert is alkalmaznak, amelyek elsajátításához több éves tanulás szükséges. Ezek ismeretére és alkalmazására csak nagyméretű feladatoknál van szükség, amelyeket már jobb, ha rájuk hagyunk.

1. FÜGGELÉK

A LEÍRÓ NYELV SZABÁLYAI

A megoldás algoritmusának, adatszerkezeteinek leírására egy, a magyar nyelvhez közel álló – nem túl szigorú – nyelvet használunk. Nézzük meg ezt a nyelvet egy kicsit részletesebben! Felsoroljuk utasításait. Egy utasítás neve mindig nagy betűvel kezdődik (itt, és csak itt idézőjelek közé írjuk őket). A kis betűkkel írt fogalmakat részletesebben kifejtjük, a nagy betűkkel írtakat nem. Egy utasítás általában egy mondat.

Program:
  'Program':
    utasítássorok
  'Program vége'.
Utasítássor:
  utasítás: utasítássor
 vagy utasítás
Értékadó utasítás:
  azonosító:=kifejezés
  Jelentése: a megadott változó felveszi a kifejezés értékét.
Azonosító: változónév vagy tömbelem vagy tömbnév.
Kifejezés: Tetszőleges, a matematikában megszokott kifejezés.
Olvasó utasítás:
  'Be': azonosítók
  Jelentése: beolvas az írógéptől tetszőleges adatokat a megadott változókba, amelyeket vesszővel kell elválasztani egymástól.
Író utasítás:
  'Ki': kifejezések [tetszőleges szöveg]
  Jelentése: kiírja a megadott kifejezéseket a képernyőre. Ha az egyes kifejezések között pontosvessző van, akkor közvetlenül egymás mögé; ha vessző, akkor a következő, ún. tabulátorpozícióra. Megadható egy speciális kifejezés, amely azt jelzi, hogy a következő kiírás hova kerüljön a képernyőn (sor, oszlop). A szögletes zárójelben megadott szöveg tetszőleges, a kiírás formátumára vonatkozó megjegyzés lehet.
Megjegyzés: [,] jelek között tetszőleges szöveg lehet. Ezt használjuk föl arra is, hogy állításokat helyezzünk el a programban elvárásainkról, amelyeket bizonyos utasítások környezetében a program kódolásakor végrehajtható utasításokká is átalakíthatunk.
Feltételes utasítás:
  'Ha' logikai kifejezés 'akkor' utasítássor
vagy
  'Ha' logikai kifejezés 'akkor' utasítássorok
  'Elágazás vége'
vagy
  'Ha' logikai kifejezés 'akkor' utasítássor
  'különben' utasítássor
vagy
  'Ha' logikai kifejezés 'akkor' utasítássorok
  'különben' utasítássorok
  'Elágazás vége'

  Jelentése: A logikai kifejezés igazságértékétől függően a megfelelő utasítássorok hajtódnak végre, s ezek után a feltételes utasítás utáni első utasítás. Ha a feltételes utasításbeli utasítássorokban további feltételes utasítás van, akkor az 'Elágazás vége' alapszó kiírásával egyértelművé kell tenni, hogy melyik, hol fejeződik be.

Várakozás:

'Várj, amíg kell'

Jelentése: vár, amíg a felhasználó nem avatkozik közbe (nem nyom le egy billentyűt az írógépen).

Ciklus:

'Ciklus' ciklusváltozó = kezdőérték'-től'
végérték'-ig'
utasítássorok
'Ciklus vége'

Jelentése: a közbezárt utasítássorokat (ciklusmag) végrehajtjuk a ciklusváltozó lehetséges értékeivel, amelyeket a kezdőértéktől a végértékig egyesével felvesz.

'Ciklus'ciklusváltozó=kezdőérték'-től'
végérték'-ig'
lépésköz'-ösével'
utasítássorok
'Ciklus vége'

Jelentése: a közbezárt utasítássorokat végrehajtjuk a ciklusváltozó lehetséges értékeivel, amelyeket a kezdőértéktől a végértékig a megadott lépésközzel felvesz.

Megjegyzés: A kétféle ciklusutasítás esetén előfordulhat, hogy a ciklus magját egyszer sem hajtjuk végre.

'Ciklus amíg' logikai kifejezés
utasítássorok
'Ciklus vége'

Jelentése: a ciklusmag végrehajtása a logikai kifejezés értéke alapján történik. A ciklusmagot addig kell végrehajtani, amíg a logikai kifejezés igaz. Ezt az első végrehajtás előtt is megvizsgáljuk. Amikor hamis lesz, akkor a 'Ciklus vége' utáni utasítással folytatjuk a program végrehajtását.

'Ciklus'
utasítássorok
'amíg' logikai kifejezés
'Ciklus vége'

Jelentése: Az előzőtől abban különbözik, hogy a ciklusmagot egyszer mindenképpen végrehajtjuk és a végén vizsgáljuk meg, hogy kell-e még.

'Ciklus amíg szükséges'
utasítássorok
'Ciklus vége'

Jelentése: a közbezárt utasítássorokat a felhasználó engedélyével hajtja végre. Az utasítások egyszeri végrehajtásának engedélyezésére tetszőleges billentyű lenyomása felel meg, kivéve az 'N' betűt. Az utóbbi esetén a 'Ciklus vége' utáni utasításon folytatódik a program végrehajtása. Az utasítást értelmezhetjük úgy is, hogy csupán a ciklus befejezése igényel felhasználói beavatkozást, egyébként a végrehajtás automatikusan folytatódik. A kódolási lehetőségektől függ, hogy melyiket választjuk. Például az ABC80-on várakozással célszerű megoldani, a HT–1080Z-n pedig úgy, hogy a folytatáshoz ne kelljen beavatkozni.

Eljáráshívás:

Tetszőleges magyar nyelvű mondat, amelyet később kifejtünk. Mögötte zárójelben megadhatók az eljárás paraméterei.

Jelentése: Végezzük el az eljárásban leírt tevékenységet, majd folytassuk a programot az eljáráshívás utáni utasítással.

Megjegyzés: Eljárások, ill. a később definiálandó függvények esetén külön problémát okoz ezek saját változóinak, ill. paramétereinek kezelése, ezt minden esetben az adott feladatnak megfelelően kell elvégezni.

Eljárásdefiníció:

ELJÁRÁS NÉV:
utasítássorok
'Eljárás vége'

Jelentése: Ha a programban az ELJÁRÁS NÉV szöveget látjuk, akkor a helyére behelyettesítendők a megadott utasítássorok.

Adatleírás:

'Tömb' tömbnév (max. index[ek])

Jelentése: Megállapítjuk, hogy a továbbiakban az adott néven legfeljebb max. indexű vektort, ill. mátrixot fogunk használni.

Függvényhivatkozás:

Tetszőleges magyar nyelvű mondat, amelyet zárójelek között követnek a függvény paraméterei (kifejezések).

Jelentése: (csak kifejezésben használható) számoljuk ki a függvény értékét a zárójelben felsorolt értékekkel, s ezzel az értékkel végezhetjük a további műveleteket.

Függvénydefiníció:

FÜGGVÉNY NÉV (azonosítók):
A függvény értékének kiszámítása
FÜGGVÉNY NÉV:= a kiszámított érték
'Függvény definiálás vége'.

Jelentése: így kell kiszámítani a függvény értékét, amelyet a FÜGGVÉNY NÉV nevű változóba kell elhelyezni.

FÜGGVÉNY NÉV (azonosítók): = kifejezés

Jelentése: a függvényt ezzel a kifejezéssel kell kiszámítani.

Sokirányú elágazás:

'Elágazás'
feltétel$_1$ 'esetén' utasítássorok$_1$
feltétel$_2$ 'esetén' utasítássorok$_2$
... ...
feltétel$_n$ 'esetén' utasítássorok$_n$
'Elágazás vége'

Jelentése: Azokat az utasítássorokat kell végrehajtani, amelyek feltétele a leírás sorrendjében először teljesül. Utána, vagy ha egyik sem teljesül, akkor az elágazás utáni első utasítással kell folytatni a program végrehajtását.

vagy

'Elágazás'
feltétel$_1$ 'esetén' utasítássorok$_1$
feltétel$_2$ 'esetén' utasítássorok$_2$
... ...
'egyéb esetben' utasítássorok$_n$
'Elágazás vége'

Jelentése: Az előzőtől csupán annyiban tér el, hogy az n. feltételt 'minden egyéb'-bé tágítottuk ki.

2. FÜGGELÉK

BASIC KÓDOLÁSI SZABÁLYOK

Nézzük meg ezután, mit kell tudnunk a BASIC programozási nyelvről! Tulajdonképpen más nyelvek esetén sem kellene sokkal több információt megjegyezni, de e könyvben csak a BASIC-kel foglalkozunk.

Az algoritmusleírást az előző függelékben leírtak szerint készítjük és mögé írjuk az egyes utasítások BASIC megfelelőjét. A BASIC program minden sora elé sorszámot írunk. A változó- és függvényneveket is kódolni kell a BASIC programban. Az egyes személyi számítógépek a BASIC-nek sajnos más-más nyelvjárásait értik, így a kódolási szabályokat is több gépre készítettük el. Ezeket a szövegben egy-egy betűvel fogjuk jelölni:

A – ABC80;
C – Commodore 64;
H – HT–1080Z;
S – ZX Spectrum;
Z – ZX–81.

Megjegyzések:

– Bár mechanikusan elvégezhető *kódolási szabályokat* mondunk ki, mégse felejtsük, hogy ezek nagyvonalú használata az igazán célszerű!

– A kódolási szabályok kialakításánál az egyszerűséget fontosabb szempontnak tartottuk a hatékonyságnál.

– A ZX–81-en az utasítássor csak egyetlen utasítást jelent.

– A Commodore 64-en a [CLR] a képernyőtörlést előidéző vezérlőkarakter.

```
1.      Program
                 utasitások
        Program vége .

   A:    S1   PRINT CHR$(12);'Tájékoztató'
         ..   ...
         S2   STOP

   C:    S1   PRINT "[CLR]Tájékoztató"
         ..   ...
         S2   STOP

   H,S:  S1   CLS : PRINT "Tájékoztató"
         ..   ...
         S2   STOP

   Z:    S1   CLS
         S2   PRINT "Tájékoztató"
         ..   ...
         S3   STOP
```

2. azonositò := kifejezés

```
A,C,H:        S1  azonositò = kifejezés

S,Z:          S1  LET azonositò = kifejezés
```

3. **Be:** azonositòk

```
A,C:          S1  PRINT "kérdés";
              S2  INPUT azonositòk

H,S:          S1  INPUT "kérdés";azonositòk

Z:            S1  PRINT "kérdés";
              S2  INPUT azonositò
```

4. **Ki:** kifejezések
   [a kiirás formájára vonatkozò szöveg]

```
A,C,H,S,Z:          S1  PRINT kifejezések
                  [ az utasitás elöbbi leirásnak
                      megfelelö alakja ]
```

5. [magyarázò szöveg]

```
A,C,H,S,Z:          S1  REM magyarázò szöveg
```

6. **Ha** logikai kifejezés **akkor** utasitássor

```
A,C,H,S,Z:          S1  IF logikai kifejezés THEN ...
```

7. **Ha** logikai kifejezés **akkor**
   utasitássorok
   Elágazás vége

```
A,H:          S1  IF logikai kifejezés THEN S2 ELSE S3
              S2  .....
              S3  itt lesz a köv. utasitás

C,S,Z:        S1  IF NOT (logikai kifejezés) THEN GOTO S3
              S2  ...
              S3  itt lesz a köv. utasitás
```

8. **Ha logikai kifejezés akkor** utasitássor
különben utasitássor

```
A,H:        S1  IF logikai kifejezés THEN ... ELSE ...

C,S,Z:      S1  IF logikai kifejezés THEN GOTO S4
            S2  a 'hamis-ág' utasitásai
            S3  GOTO S5
            S4  az 'igaz-ág' utasitásai
            S5  itt lesz a köv. utasitás
```

9. **Ha logikai kifejezés akkor**
utasitássorok
különben
utasitássorok
Elágazás vége

```
A,H:        S1  IF logikai kifejezés THEN S2 ELSE S4
            S2  ...
            S3  GO TO S5
            S4  ...
            S5  itt lesz a köv. utasitás

C,S,Z:      azonos a 8. kòdolásával
```

10. **Várj amig kell**

```
A:        S1  GET X$

C:        S1  GET X$ : IF X$="" THEN S1

H:        S1  IF INKEY$="" THEN S1

S:        S1  PAUSE 0

Z:        S1  PAUSE 4E4
```

11. **Ciklus c=k-tòl v-ig**
utasitássorok
Ciklus vége

```
A,C,H,S,Z:  S1  FOR c=k TO v
            ..  ...
            S2  NEXT c
```

Megjegyzés: A Commodore 64 és a HT–1080Z esetében ügyelni kell arra, hogy a ciklus *legalább egyszer lefutó.* Ezért, ha előfordulhat az az eset, hogy már a kezdőérték is túlhaladta a végértéket, akkor a ciklus egyszeri végrehajtását elkerülni csak az S0 IF k>v THEN S3 utasítással tudjuk. Az S3 a ciklust követő utasítás. Ez a megjegyzés magától értetődően érvényes a *számlálásos ciklus* mindegyikére. Értelemszerűen ott is meg kell gondolni! Mi külön említést már nem teszünk erről.

```
12. Ciklus c=k-tól v-ig 1-esével
                  utasitássorok
    Ciklus vége

    A,C,H,S,Z:    S1   FOR c=k TO v STEP 1
                  ..   ...
                  S2   NEXT c

13. Ciklus amig logikai kifejezés
                  utasitássorok
    Ciklus vége

    A,H:          S1   IF logikai kifejezés THEN S2 ELSE S4
                  S2   ...
                  S3   GO TO S1
                  S4   itt lesz a köv. utasitás

14. Ciklus
              utasitássorok
    amig logikai kifejezés
    Ciklus vége

    A,C,H,S,Z:    S1   ciklusmag 1. utasitás
                  ..   ...
                  S2   IF logikai kifejezés THEN GOTO S1

    C,S,Z:        S1   IF NOT (logikai kifejezés) THEN GOTO S4
                  S2   ...
                  S3   GO TO S1
                  S4   itt lesz a köv. utasitás

15. Ciklus amig szükséges
                  utasitássorok
    Ciklus vége

    A,C:      S1   GET X$ : IF X$="N" THEN S3
              ..   ...
              S2   GO TO S1
              S3   itt lesz a köv. utasitás

    H,S,Z:    S1   IF INKEY$="N" THEN GOTO S3
              ..   ...
              S2   GO TO S1
              S3   itt lesz a köv. utasitás
```

```
16. Eljáráshivás(kifejezések)

    A,C,H:   S1   paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S3 : REM ELJARAS NEV

    S:       S1   LET paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S3 : REM ELJARAS NEV

    Z:       S0   REM eljárás hivás : ELJARAS NEV
             S1   LET paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S3
```

Megjegyzés: S3 a hívott szubrutin első sorának sorszáma.

```
17. ELJARAS NEV:
                   utasitássorok
    Eljárás vége

    A,C,H,S,Z:   S1   REM ELJARAS NEV
                 ..   ...
                 S2   RETURN

18. Tömb tömbnév (max.indexek)

    A,C,H,S,Z:   S1   DIM tömbnév (max.indexek)
```

Megjegyzés: A ZX Spectrum és a ZX–81 esetén DIM utasításban egyetlen tömbnév lehet.

```
19. azonositò:= FUGGVENY NEV (kifejezések)

    A,C,H:   S1   paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S4 : REM FUGGVENY NEV
             S3   azonositò = FUGGVENY NEV

    S:       S1   LET paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S4 : REM FUGGVENY NEV
             S3   LET azonositò = FUGGVENY NEV

    Z:       S0   REM függvény hivás : FUGGVENY NEV
             S1   LET paraméter = kifejezés
             ..   ... (annyi, ahány paraméter van)
             S2   GOSUB S4
             S3   LET azonositò = FUGGVENY NEV
```

Megjegyzések:

– S4 a kiszámítandó függvény első sorának sorszáma.

– Ha a függvény bonyolultabb kifejezésben szerepel, akkor az előbbihez hasonlóan (értelemszerűen) kell elvégezni a kódolását.

Más kódolási lehetőség (ahol van DEF FN utasítás és a függvényt így definiáltuk):

```
A,S: S1  azonositò=FN függvény név(kifejezések)

C:   S1  azonositò=FN függvény név(kifejezés)
```

```
20. FUGGVENY NEV(azonositòk):
                 utasitássorok
    Függvény definiálás vége

    A,C,H:  S1  REM FUGGVENY NEV(azonositòk)
            ..  ...
            S2  FUGGVENY NEV= kifejezés
            S3  RETURN

    S,Z:    S1  REM FUGGVENY NEV(azonositòk)
            ..  ...
            S2  LET FUGGVENY NEV= kifejezés
            S3  RETURN
```

Más kódolás (ahol van DEF FN utasítás és a definíció egyetlen képlet):

```
A,S: S1  DEFFN FUGGVENY NEV(azonositòk) =
                                     kifejezés

C:   S1  DEFFN FUGGVENY NEV(azonositò) =
                                     kifejezés
```

```
21. Elágazás
          f1 esetén utasitás1
          f2 esetén utasitás2
          ...
          fn esetén utasitásn
    Elágazás vége

     A,C,H,S,Z: S1        IF f1 THEN GO TO S10
                S2        IF f2 THEN GO TO S20
                ..        ...
                Sn        IF fn THEN GO TO Sn0
                Sm        GO TO S0
                S10        ...
                ..         ...
                S19        GO TO S0
                S20        ...
                ...        ...
                Sn0        ...
                ..         ...
                S0         REM itt az elágazás vége

    A,C,H:   S0  ON kifejezés GOTO S1,S2,...Sn
```

Megjegyzés: Néha az elágazás feltételei könnyen átalakíthatók 1 és n közötti természetes számokká, ekkor kedvezőbb kódolást biztosít:

A,C,H: S0 ON kifejezés GOTO S1, S2,...Sn.

```
22. Elágazás
           f1     esetén  utasitás1
           f2     esetén  utasitás2
           ...
          egyéb esetben  utasitásn
    Elágazás vége

     A,C,H,S,Z: S1         IF f1 THEN GO TO S10
                S2         IF f2 THEN GO TO S20
                ..         ...
                Sn         IF fn THEN GO TO Sn0
                Sm         ... az 'egyéb esetben' vég-
                                         rehajtandók
                Sm9        GO TO S0
                S10        ...
                ..         ...
                S19        GO TO S0
                S20        ...
                ...        ...
                Sn0        ...
                ..         ...
                S0         REM itt az elágazás vége
```

3. FÜGGELÉK

A CSILLAGSŰRŰSÖDÉSEK SZÁMÁT MEGHATÁROZÓ PROGRAM KÓDOLÁSA

A programot a különböző gépekre úgy kódoltuk, hogy a kód a lehető legnagyobb hasonlóságot mutassa a HT–1080Z BASIC változattal. Megfigyelhető, hogy a legtöbb eltérés az egyes gépek sajátosságai miatt van. (Kép letörlés, sorok száma a képen, a felhasználó beavatkozására való várakozás stb.)

Kezdjük az ABC80-as változattal!

```
10 PRINT CHR$(12);"CSILLAGSURUSODESEK SZAMA"
20 PRINT
30 PRINT "  A PROGRAM EGY TETSZOLEGES CSILLAGTERKE-"
40 PRINT "PEN  A  CSILLAGSURUSODESEK  SZAMANAK MEG-"
50 PRINT "HATAROZASARA KEPES. CSILLAGSURUSODESEKNEK"
60 PRINT "NEVEZZUK  AZOKAT  A  LEGNAGYOBB CSOPORTO-"
70 PRINT "KAT , AMELYEKBEN  LEGALABB 'SO' DB. CSIL-"
80 PRINT "LAG  VAN , ES  MINDEN  OTT LEVO CSILLAGRA"
90 PRINT "IGAZ , HOGY  BARMELYIK CSOPORTTARSAHOZ EL"
100 PRINT "LEHET JUTNI  OLYAN UTON , AMELY A CSOPORT"
110 PRINT "TAGJAIN KERESZTUL  HALAD  ES  EGYIK LEPES"
120 PRINT "SEM HOSSZABB 'S1' FENYEVNEL."
130 PRINT CUR(22,0);'HA ELOLVASTA, NYOMJA LE A';
' <RETURN>-T!';
140 GET W9$
150 PRINT CHR$(12);'HANY CSILLAG VAN A TERKEPEN';
155 ONERRORGOTO 150
160 INPUT N
170 IF N<1 THEN 150
180 DIM X(N),Y(N),Z(N),V(N),C(N)
190 PRINT 'CSILLAGSZAM';
195 ONERRORGOTO 190
200 INPUT SO : IF SO<1 OR SO>N THEN 190
210 PRINT 'TAVOLSAG (FENYEVBEN)';
215 ONERRORGOTO 210
220 INPUT S1 : IF S1<0 THEN 210
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250 PRINT 'AZ';I;'. CSILLAG KOORDINATAI';
255 ONERRORGOTO 250
260 INPUT X(I),Y(I),Z(I)
270 NEXT I
280 PRINT CHR$(12);'DOLGOZOM!'
290 S=0
300 GOSUB 400 : REM A SURUSODESEK SZAMANAK MEGHAT.
310 PRINT CHR$(12)
320 PRINT 'A CSILLAGSURUSODESEK SZAMA:';S
330 STOP
```

```
400 REM A SURUSODESEK SZAMANAK (S) MEGHATAROZASA
410 FOR I=1 TO N
420 V(I)=0 : REM EZT MEG NEM VIZSGALTUK
430 NEXT I
440 FOR I=1 TO N
450 IF V(I)=0 THEN GOSUB 500 : REM AZ I. CSILLAGOT
TARTALMAZO SURUSODES VIZSGALATA
460 NEXT I
470 RETURN
500 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
510 A=0
520 L=I : GOSUB 600 : REM AZ L. CSILLAG FELVETELE
530 K=1 : REM ELOSZOR A KEZDO CSILLAGHOZ KOZELIE-
KET FOGJUK VIZSGALNI
540 IF K<=A THEN 550 ELSE 570
550 GOSUB 700 : REM A C(K). CSILLAGHOZ KOZELIEK
VIZSGALATA
560 K=K+1 : GOTO 540
570 REM A= A CSOPORT CSILLAGJAINAK SZAMA
580 IF A>=S0 THEN S=S+1
590 RETURN
600 REM AZ L. CSILLAG FELVETELE A CSOPORTBA
610 V(L)=1
620 A=A+1 : C(A)=L
630 RETURN
700 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
710 DEF FNT(A,B)=SQR((X(A)-X(B))*(X(A)-X(B))+
(Y(A)-Y(B))*(Y(A)-Y(B))+(Z(A)-Z(B))*(Z(A)-Z(B))) :
REM TAVOLSAG-FUGGVENY
720 FOR J=I+1 TO N
730 IF V(J)=0 THEN 740 ELSE 760
740 T=FNT(J,C(K))
750 IF T<=S1 THEN L=J : GOSUB 600 : REM A J.
CSILLAG FELVETELE A CSOPORTBA
760 NEXT J
770 RETURN
```

Most nézzük a Commodore 64 kódolást!

```
10 PRINT "[CLR]CSILLAGSURUSODESEK SZAMANAK MEGHAT."
20 PRINT
30 PRINT "  A PROGRAM EGY TETSZOLEGES CSILLAGTERKE-"
40 PRINT "PEN  A  CSILLAGSURUSODESEK  SZAMANAK MEG-"
50 PRINT "HATAROZASARA KEPES. CSILLAGSURUSODESEKNEK"
60 PRINT "NEVEZZUK  AZOKAT  A  LEGNAGYOBB CSOPORTO-"
70 PRINT "KAT , AMELYEKBEN  LEGALABB 'S0' DB. CSIL-"
80 PRINT "LAG  VAN , ES  MINDEN  OTT LEVO CSILLAGRA"
```

```
90 PRINT "IGAZ , HOGY  BARMELYIK CSOPORTTARSAHOZ EL"
100 PRINT "LEHET JUTNI OLYAN UTON , AMELY A CSOPORT"
110 PRINT "TAGJAIN KERESZTUL HALAD  ES  EGYIK LEPES"
120 PRINT "SEM HOSSZABB 'S1' FENYEVNEL."
130 PRINT "HA ELOLVASTA, NYOMJA LE A <RETURN>-T!";
140 GET W9$ : IF W9$="" THEN 140
150 PRINT "[CLR]HANY CSILLAG VAN A TERKEPEN";
160 INPUT N
170 IF N<1 THEN 150
180 DIM X(N),Y(N),Z(N),V(N),C(N)
190 PRINT "CSILLAGSZAM";
200 INPUT SO : IF SO<1 OR SO>N THEN 190
210 PRINT "TAVOLSAG (FENYEVBEN)";
220 INPUT S1 : IF S1<O THEN 210
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250 PRINT "AZ";I;". CSILLAG KOORDINATAI";
260 INPUT X(I),Y(I),Z(I)
270 NEXT I
```

```
10 PRINT "[CLR]CSILLAGSURUSODESEK SZAMANAK MEGHAT."
20 PRINT
30 PRINT "  A PROGRAM EGY TETSZOLEGES CSILLAGTERKE-"
40 PRINT "PEN  A  CSILLAGSURUSODESEK  SZAMANAK MEG-"
50 PRINT "HATAROZASARA KEPES. CSILLAGSURUSODESEKNEK"
60 PRINT "NEVEZZUK  AZOKAT  A  LEGNAGYOBB CSOPORTO-"
70 PRINT "KAT , AMELYEKBEN  LEGALABB 'SO' DB. CSIL-"
80 PRINT "LAG  VAN , ES  MINDEN  OTT LEVO CSILLAGRA"
```


```
90 PRINT "IGAZ , HOGY  BARMELYIK CSOPORTTARSAHOZ EL"
100 PRINT "LEHET JUTNI OLYAN UTON , AMELY A CSOPORT"
110 PRINT "TAGJAIN KERESZTUL HALAD  ES  EGYIK LEPES"
120 PRINT "SEM HOSSZABB 'S1' FENYEVNEL."
130 PRINT "HA ELOLVASTA, NYOMJA LE A <RETURN>-T!";
140 GET W9$ : IF W9$="" THEN 140
150 PRINT "[CLR]HANY CSILLAG VAN A TERKEPEN";
160 INPUT N
170 IF N<1 THEN 150
180 DIM X(N),Y(N),Z(N),V(N),C(N)
190 PRINT "CSILLAGSZAM";
200 INPUT SO : IF SO<1 OR SO>N THEN 190
210 PRINT "TAVOLSAG (FENYEVBEN)";
220 INPUT S1 : IF S1<O THEN 210
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250 PRINT "AZ";I;". CSILLAG KOORDINATAI";
260 INPUT X(I),Y(I),Z(I)
270 NEXT I
```

```
280 PRINT "[CLR]DOLGOZOM!"
290 S=0
300 GOSUB 400 : REM A SURUSODESEK SZAMANAK MEGHAT.
310 PRINT "[CLR]"
320 PRINT "A CSILLAGSURUSODESEK SZAMA:";S
330 STOP
400 REM A SURUSODESEK SZAMANAK (S) MEGHATAROZASA
410 FOR I=1 TO N
420 V(I)=0 : REM EZT MEG NEM VIZSGALTUK
430 NEXT I
440 FOR I=1 TO N
450 IF V(I)=0 THEN GOSUB 500 : REM AZ I. CSIL-
LAGOT TARTALMAZO SURUSODES VIZSGALATA
460 NEXT I
470 RETURN
500 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
510 A=0
520 L=I : GOSUB 600 : REM AZ L. CSILLAG FELVETELE
530 K=1 : REM ELOSZOR A KEZDO CSILLAGHOZ KOZELIE-
KET FOGJUK VIZSGALNI
540 IF K>A THEN 570
550 GOSUB 700 : REM A C(K). CSILLAGHOZ KOZELIEK
VIZSGALATA
560 K=K+1 : GOTO 540
570 REM A= A CSOPORT CSILLAGJAINAK SZAMA
580 IF A>=S0 THEN S=S+1
590 RETURN
600 REM AZ L. CSILLAG FELVETELE A CSOPORTBA
610 V(L)=1
620 A=A+1 : C(A)=L
630 RETURN
700 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
710 IF I=N THEN RETURN
720 FOR J=I+1 TO N
730 IF V(J)<>0 THEN 760
740 T=SQR((X(J)-X(C(K)))ü2+(Y(J)-Y(C(K)))ü2+
(Z(J)-Z(C(K)))ü2)
750 IF T<=S1 THEN L=J : GOSUB 600 : REM A J.
CSILLAG FELVETELE A CSOPORTBA
760 NEXT J
770 RETURN
```

Ezután lássuk a ZX Spectrum kódolást!

```
10 CLS
15 PRINT "CSILLAGSURUSODESEK SZAMA"
20 PRINT
30 PRINT "  A PROGRAM EGY TETSZOLEGES CSIL-"
40 PRINT "LAGTERKEPEN  A CSILLAGSURUSODESEK"
50 PRINT "SZAMANAK  MEGHATAROZASARA  KEPES."
60 PRINT "  CSILLAGSURUSODESEKNEK  NEVEZZUK"
70 PRINT "AZOKAT  A LEGNAGYOBB CSOPORTOKAT,"
80 PRINT "AZOKAT  LEGALABB 'SO' DB. CSILLAG"
90 PRINT "VAN  ES MINDEN OTT LEVO CSILLAGRA"
100 PRINT "IGAZ , HOGY BARMELYIK CSOPORTTAR-"
101 PRINT "SAHOZ  EL LEHET JUTNI OLYAN UTON,"
102 PRINT "TON , AMELY A CSOPORT TAGJAIN KE-"
110 PRINT "RESZTUL HALAD  ES EGYIK LEPES SEM"
120 PRINT "HOSSZABB 'S1' FENYEVNEL."
130 PRINT "NYOMJA LE A <RETURN>-T!";
140 PAUSE 0
150 CLS
160 INPUT "HANY CSILLAG VAN A TERKEPEN";N
170 IF N<1 THEN GOTO 150
180 DIM X(N) : DIM Y(N) : DIM Z(N)
181 DIM V(N) : DIM C(N)
190 INPUT "CSILLAGSZAM";SO
200 IF SO<1 OR SO>N THEN GOTO 190
210 INPUT "TAVOLSAG (FENYEVBEN)";S1
220 IF S1<0 THEN GOTO 210
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250 INPUT ("AZ";I;". CSILLAG KOORDINATAI?");
X(I),Y(I),Z(I)
270 NEXT I
280 CLS : PRINT "DOLGOZOM!"
290 LET S=0
300 GOSUB 400 : REM A SURUSODESEK SZAMANAK MEGHAT.
310 CLS
320 PRINT "A CSILLAGSURUSODESEK SZAMA:";S
330 STOP
400 REM A SURUSODESEK SZAMANAK (S) MEGHATAROZASA
410 FOR I=1 TO N
420 LET V(I)=0 : REM EZT MEG NEM VIZSGALTUK
430 NEXT I
440 FOR I=1 TO N
450 IF V(I)=0 THEN GOSUB 500 : REM AZ I. CSIL-
LAGOT TARTALMAZO SURUSODES VIZSGALATA
460 NEXT I
```

```
470 RETURN
500 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
510 LET A=0
520 LET L=I : GOSUB 600 : REM AZ L. CSILLAG FELVETELE
530 LET K=1 : REM ELOSZOR A KEZDO CSILLAGHOZ KOZELIE-
KET FOGJUK VIZSGALNI
540 IF K>A THEN GOTO 570
550 GOSUB 700 : REM A C(K). CSILLAGHOZ KOZELIEK
VIZSGALATA
560 LET K=K+1 : GOTO 540
570 REM A= A CSOPORT CSILLAGJAINAK SZAMA
580 IF A>=S0 THEN LET S=S+1
590 RETURN
600 REM AZ L. CSILLAG FELVETELE A CSOPORTBA
610 LET V(L)=1
620 LET A=A+1 : LET C(A)=L
630 RETURN
700 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
710 IF I=N THEN RETURN
720 FOR J=I+1 TO N
730 IF V(J)<>0 THEN GOTO 760
740 LET T=SQR((X(J)-X(C(K)))ü2+(Y(J)-Y(C(K)))ü2+
(Z(J)-Z(C(K)))ü2)
750 IF T<=S1 THEN LET L=J :  GOSUB 600 : REM A J.
CSILLAG FELVETELE
760 NEXT J
770 RETURN
```

Végül nézzük a ZX–81 kódolást!

```
 10 CLS
 15 PRINT "CSILLAGSURUSODESEK SZAMA"
 20 PRINT
 30 PRINT "  A PROGRAM EGY TETSZOLEGES CSIL-"
 40 PRINT "LAGTERKEPEN  A CSILLAGSURUSODESEK"
 50 PRINT "SZAMANAK  MEGHATAROZASARA  KEPES."
 60 PRINT "  CSILLAGSURUSODESEKNEK  NEVEZZUK"
 70 PRINT "AZOKAT  A LEGNAGYOBB CSOPORTOKAT,"
 80 PRINT "AZOKAT  LEGALABB 'SO' DB. CSILLAG"
 90 PRINT "VAN  ES MINDEN OTT LEVO CSILLAGRA"
100 PRINT "IGAZ , HOGY BARMELYIK CSOPORTTAR-"
101 PRINT "SAHOZ  EL LEHET JUTNI OLYAN UTON,"
102 PRINT "TON , AMELY A CSOPORT TAGJAIN KE-"
110 PRINT "RESZTUL  HALAD ES EGYIK LEPES SEM"
120 PRINT "HOSSZABB 'S1' FENYEVNEL."
130 PRINT "NYOMJA LE A <RETURN>-T!";
140 PAUSE 4E4
```

```
150 CLS
151 PRINT "HANY CSILLAG VAN A TERKEPEN?";
160 INPUT N
170 IF N<1 THEN GOTO 150
171 PRINT N
180 DIM X(N)
181 DIM Y(N)
182 DIM Z(N)
183 DIM V(N)
184 DIM C(N)
190 PRINT "CSILLAGSZAM?";
200 INPUT SO
201 IF SO<1 OR SO>N THEN GOTO 190
202 PRINT SO
210 PRINT "TAVOLSAG(FENYEVBEN)?";
220 INPUT S1
221 IF S1<0 THEN GOTO 210
222 PRINT S1
230 REM A KOORDINATAK BEOLVASASA
240 FOR I=1 TO N
250 PRINT "AZ";I;". CSILLAG KOORDINATAI?";
260 INPUT X(I)
261 INPUT Y(I)
262 INPUT Z(I)
263 PRINT X(I);" ";Y(I);" ";Z(I)
270 NEXT I
280 CLS
281 PRINT "DOLGOZOM!"
290 LET S=0
291 REM A SURUSODESEK SZAMANAK MEGHATAROZASA
300 GOSUB 400
310 CLS
320 PRINT "A CSILLAGSURUSODESEK SZAMA:";S
330 STOP
400 REM A SURUSODESEK SZAMANAK (S) MEGHATAROZASA
410 FOR I=1 TO N
419 REM  EZT A CSILLAGOT MEG NEM VIZSGALTUK
420 LET V(I)=0
430 NEXT I
440 FOR I=1 TO N
449 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
450 IF V(I)=0 THEN GOSUB 500
460 NEXT I
470 RETURN
500 REM AZ I. CSILLAGOT TART. SURUSODES VIZSGALATA
510 LET A=0
520 REM AZ L. CSILLAG FELVETELE
```

```
521 LET L=I
522 GOSUB 600
530 REM ELOSZOR A KEZDO CSILLAGHOZ KOZELIEKET
FOGJUK VIZSGALNI
531 LET K=1
540 IF K>A THEN GOTO 570
550 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
551 GOSUB 700
560 LET K=K+1
561 GOTO 540
570 REM A= A CSOPORT CSILLAGJAINAK SZAMA
580 IF A>=S0 THEN LET S=S+1
590 RETURN
600 REM AZ L. CSILLAG FELVETELE A CSOPORTBA
610 LET V(L)=1
620 LET A=A+1
621 LET C(A)=L
630 RETURN
700 REM A C(K). CSILLAGHOZ KOZELIEK VIZSGALATA
710 IF I=N THEN RETURN
720 FOR J=I+1 TO N
730 IF V(J)<>0 THEN GOTO 760
740 LET T=SQR((X(J)-X(C(K)))**2+(Y(J)-Y(C(K)))**2+
(Z(J)-Z(C(K)))**2)
750 IF T>S1 THEN GOTO 760
751 REM A J. CSILLAG FELVETELE
752 LET L=J
753 GOSUB 600
760 NEXT J
770 RETURN
```

[1] *Dahl, O. J.–Dijkstra, E. W.–Hoare, C. A. R.:* Strukturált programozás. Budapest, Műszaki Könyvkiadó, 1978.
[2] *Széplaki Á.:* Programhiba keresés, tesztelés, minőségvizsgálat. Budapest, KSH SZÁMOK, 1979.
[3] *Várkonyi Zs.:* Bevezetés a modern programtesztelésbe. Budapest, Műszaki Könyvkiadó, 1979.
[4] *Aszalós J.–Erki I.:* Bevezetés a strukturált programozásba. Budapest, KSH SZÁMOK, 1980.
[5] *Varga L.:* Programok analízise és szintézise. Budapest, Akadémiai Kiadó, 1981.
[6] *Wirth, N.:* Algoritmusok + adatstruktúrák = programok. Budapest, Műszaki Könyvkiadó, 1982.
[7] *Alcock, D.:* Ismerd meg a BASIC nyelvet! Budapest, Műszaki Könyvkiadó, 1983.
[8] *Fóthi Á.:* Bevezetés a programozáshoz. Jegyzet. Budapest, Tankönyvkiadó, 1983.
[9] *Kőhegyi J. (szerk)–Harmathy Z.–Zsakó L.–Helfenbein H.:* Ismerd meg a BASIC nyelvjárásait! (HT–1080Z, ABC80, ZX–81). Budapest, Műszaki Könyvkiadó, 1984.
[10] *Szlávi P.–Zsakó L.:* A programozás ABC-je = Az ABC' programozása. Budapest, ELTE TTK Számítástechnikai Tanszék, ABC-s füzetek, 1984.
[11] *Appel Gy.–Kőhegyi J.–Zsakó L.:* Számítógépes feladatok (példatár személyi számítógépekre). Budapest, INTERPRESS Kiadó és Nyomda Vállalat, 1985.
[12] *Kőhegyi J.(szerk.)–Szlávi P.–Seprődi L.–Pomózi I.:* Ismerd meg a BASIC nyelvjárásait! (ZX Spectrum, TI99/4A, PROPER–16). Budapest, Műszaki Könyvkiadó, 1985.

50 Ft

SZLÁVI PÉTER–ZSAKÓ LÁSZLÓ

MÓDSZERES PROGRAMOZÁS

(kísérlet a módszeres programozás elterjesztésére, személyi számítógépet használók számára)

- A programozás körüli problémák
- Kiút: a programozási elvek
  vagy másként fogalmazva: Nem elég a nyelv ismerete, sőt nem a nyelv ismerete a fontos
- Programkészítés a gyakorlatban
- Szellemi kapaszkodóink: a programozási tételek
- Az elkészült program helyességének vizsgálata
  azaz
  tesztadatok a program kipróbálásához,
  és
  a hibák megkeresésének módszerei,
  továbbá
  egy formális helyességbizonyító módszer
- A program hatékonyságának vizsgálata
- A programok családi kapcsolatai
- Amit a feladatmeghatározásról tudni illik
- Dokumentáció készítése a programhoz
- Egy egyszerű leíró nyelv

A leggyakoribb BASIC nyelvjárások
kódolási szabályok ABC80-ra, HT–1080Z-re, ZX–81-re, ZX Spectrumra és Commodore 64-re:
A mintaprogram átírása más gépekre

Műszaki Könyvkiadó