Pentelényi Pál

AZ ALGORITMIKUS SZEMLÉLETMÓD KIALAKÍTÁSA ÉS FEJLESZTÉSE A TANÍTÁSI-TANULÁSI FOLYAMATBAN

Budapest
1999

Bánki Donát Műszaki Főiskola
Tanárképző Tanszék

Lektorálta

Tóth Béláné
Varga Lajos

ISBN 963 85138 8 8

A kiadvány TEMPUS Phare támogatással készült

LIGATURA LTD - ÁFÉSZ PRESS, VÁC

Előszó

Az algoritmikus szemlélet fejlesztése méltatlanul mellőzött terület a hazai oktatásban. Szerepe lehet ebben annak, hogy úgy tűnik, mintha az 'algoritmikus gondolkodás' elnevezés valamiféle mechanikus tevékenységet sugallna, amit eleve ellentétesnek érzünk a kreativitással. Jelen írásom egyik fontos célja, hogy az említett hiányt pótolja és a feltételezett ellentmondást feloldja.

Az algoritmikus szemlélet oktatási folyamatra gyakorolt hatásait és módszertani vonzatait vizsgálva azt tapasztaljuk, hogy ezek a rendszerszemlélet szükségszerű és automatikus érvényesülésében összegezhetők. Az algoritmikus szemléletmód természetéből fakadóan nyújt hathatós segítséget a tananyag (különböző szintű egységeinek és egészének) világos rendszerezéséhez éppúgy, mint az oktatási gyakorlat legkülönbözőbb területein felmerülő feladatok áttekinthető, logikus megoldásmenetének tervezéséhez. Ugyanakkor, az algoritmikus gondolkodásra nevelés, az algoritmikus szemléletmód kialakítására való törekvés amellett, hogy növeli az adott tantárgy oktatásának hatékonyságát, a számítástechnikában tanított programozást is segíti.

Remélhetőleg e tanulmány segítséget nyújthat ahhoz is, hogy a szakközépiskolai alapozó tantárgyak újabb kiadásra kerülő tankönyveit, feladattárait fokozatosan a vázolt algoritmikus szemléletmód jegyében lehessen kiegészíteni. Meggyőződésünk, hogy a műszaki pedagógusképzés keretében a leendő középiskolai tanárokat is célszerű erre a szemléletre, illetve e szemlélet átadásának igényére nevelni, felkészíteni; ugyanakkor a tanártovábbképzés révén megvalósítható, hogy a már gyakorló tanárok elsajátítsák az esetleg hiányzó programozási (elsősorban algoritmizálási) alapismereteket, az algoritmikus gondolkodást.

Ugyanebben a témakörben írt The Possibilities and Methods of the Formation and Development of Algorithmic Thinking in Technical Training c. PhD dolgozatomat 1999-ben védtem meg a Budapesti Műszaki Egyetemen. A Szakképzéspedagógiai PhD-program keretében 1995-től 1998-ig folytatott tanulmányaim, ill. ezekkel összefüggésben végzett kutatómunkám révén vált lehetővé, hogy a Bánki Donát Műszaki Főiskolán 1984 óta folytatott oktatási kísérletek eredményeit és tapasztalatait dolgozatomban összegezzem. Ezúton fejezem ki köszönetemet a Budapesti Műszaki Egyetem Műszaki Pedagógia Tanszékének, hogy a 3 éves szakképzéspedagógiai PhD-program során tanulmányaimat segítették. Külön köszönet illeti Tóth Bélánét, aki témavezetőként segítette és irányította munkámat. Köszönettel tartozom továbbá Fekete István és Gyaraki F. Frigyes opponenseknek véleményük kialakításával kapcsolatos körültekintő figyelmükért és összes fáradozásukért; észrevételeiket és kiegészítéseiket, értékes javaslataikat jelen magyar nyelvű munkámban is hasznosítani igyekeztem.

Tartalomjegyzék

1. Ember és technika

1.1. A fizikai és a szellemi munkák mechanizálása és gépesítése

Legutóbbi évtizedeinket gyakran az automatizálás korának nevezik. Ez az elnevezés azoknak a gépi berendezéseknek az elterjedésére utal, amelyek egyre több munkát képesek egyvégtében, emberi beavatkozás nélkül elvégezni.

A fejlődés rugója kétségkívül az a tény, hogy az ember igyekszik a fárasztó munkáktól menekülni: mind a fizikai munkáktól, mind pedig a szellemi rutinmunkáktól.

Történetileg két alapvető korszakot különböztetünk meg: mechanizálást és gépesítést.

A fizikai munkák területén kétségkívül a szerszámok használata az, amely a mechanizálást jellemzi. A szerszám arra szolgált - és szolgál mind a mai napig -, hogy az ember saját testi erejével végzett munkáját megkönnyítse, annak hatásfokát javítsa. Ettől még azonban az energiaforrás továbbra is az élő test marad; akár a saját emberi, akár a segítségül hívott állatok teste. Tehát mindenképpen biológiailag is átalakított energiákat használ fel. A döntőnek vagy forradalminak nevezhető változást a gépesítés jelenti, amikor is az történik, hogy az élő test az energiaáramból kiléphet és a természet energiái gépeken keresztül hasznosulnak.

A szellemi rutinmunkák automatizálása is régi törekvése az emberiségnek. A közismert régi számológép, az abacus például az emlékezést segíti: amikor három golyóhoz hozzáadunk kettőt, nem kell mind a két számot egyszerre emlékezetünkben tartanunk.

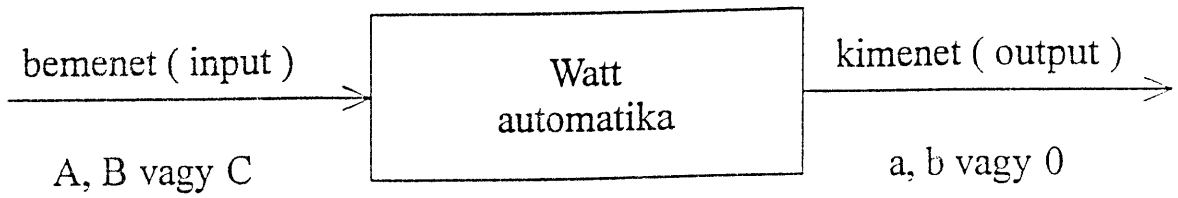
Rutinmunka a gyakran ismétlődő, egyszerű számolási műveletek végzése is - például az összeadásé. A pénztárgép tetszőlegesen sok számot képes összeadni, sőt a részösszegek "gyűjtését" (összegzését) is elvégzi - tehát egy korlátozott emlékezettel, memóriával rendelkező, mechanikus, számtani műveletet végző gép.

Szellemi rutinmunka az a tevékenység is, amit "a váratlan körülményektől lényegében független, - többnyire ciklikusan ismétlődő - vezérlésnek" nevezhetünk. Ilyen feladat volt például az első gőzgépek szelepeinek nyitása-zárása. A munka tehát "egy adott jelenség észlelése" - "beavatkozás" unalomig ismétlődő egymásutánja, ciklusa volt. A fiatal Watt kitalált egy szerkezetet, amely az észlelést és a szükséges beavatkozást - vezérlést - automatikusan elvégzi: lényegében ezzel kezdődött az ipari forradalom.

Ismeretes, hogy már a múlt században meglepően bonyolult mintájú szöveteket szőttek automatikus vezérlésű szövőgépeken, vagy például századunk elején az automatikus - lyukszalaggal vezérelt - gépzongorák nyolc-tízperces műveket is játszottak. A lyukszalagon lévő lyukak helyzete döntötte el, hogy a pneumatikus játszóberendezés mikor melyik billentyűt mozdítsa meg, vagyis a lyukak "vezérelték" a zongorát, és a lyukak összessége - az utasítások egymásutánja - jelentette az elvégzendő műveleteket és azok sorrendjét: a programot.

A vezérlés fogalomkörébe tartozó szellemi rutinmunka az egyszerű vizsgálat és döntés is. A döntés akkor egyszerű, ha csupán néhány lehetőségre kell felkészülnünk, és bármi lesz is a lehetőségek vizsgálatának eredménye, mindig egyértelmű, hogy mit kell tennünk. Mechanikus "döntéshozó" eszköz például egy almaosztályozó: a rácsok nagyság szerint szétválasztják az almát, és a különböző méretűeket más-más ládába továbbítják.

Példáinkban olyan kifejezéseket használtunk, amelyek a számítástechnikának is alapfogalmai: memória, számtani (aritmetikai) műveletek, vezérlés, vizsgálat (összehasonlítás), döntés, program. A példákban olyan tevékenységeket vázoltunk, melyek minden fázisában egyértelműen eldönthető, hogy a következő fázisban mit kell tenni. Ha a gőzgép dugattyúja az egyik holtponton van (A esemény), nyisd ki az a szelepet. Ha a másik holtponton van (B esemény), nyisd ki a b szelepet. Ha nincs egyik holtponton sem (C esemény), ne csinálj semmit: 0 tevékenység. Világos, hogy a három esemény közül valamelyik mindig bekövetkezik ("igaz"), és mindig csak az egyik következik be. A Watt-féle automatika mindig egyértelműen el tudja dönteni, mit tegyen (1.ábra):



(és más semmilyen körülmények között nem lehet!)

1.ábra

Az automatika tehát észlel bizonyos tényeket (bejövő, "input" adatokat), ezeket mint A, B vagy C eseményeket azonosítja, majd elvégzi a "ha A, akkor a, ha B, akkor b, ha C, akkor 0" logikai hozzárendelést, és vagy önmaga elvégzi az a, b, 0 tevékenységet, vagy utasítást ad annak elvégzésére.

A mai számítógép több lényeges szempontból különbözik az említett automatáktól. Képes arra, hogy az input adatokkal logikai és aritmetikai műveleteket végezzen, vagyis az adatokat - emberi beavatkozás nélkül, igen bonyolult utasítások alapján - más adatokká alakítsa. Működése igen gyors, mivel elektromos impulzusokkal dolgozik. Mindig van kisebb-nagyobb memóriája, ahol az adatokat és a programokat tárolja. A program tárolt voltából adódik, hogy a program megváltoztatása esetén ugyanaz a gép egészen más jellegű feladatok elvégzésére képes. Az elektronikus számítógép sok különböző célra felhasználható, univerzális, flexibilis (hajlékony, rugalmas) eszköz. Működtetéséhez azonban szükség van arra, hogy a megoldandó feladatot és a megoldás menetét a matematika, a logika nyelvén fogalmazzuk meg.

1.2. Analóg és digitális technika

Mint köztudott, építőelemek szempontjából, valamint az információ-kezelés technikája szerint kétféle gépet különböztetünk meg: analóg és digitális számítógépeket. E megkülönböztetés azonban sokkal általánosabb, minthogy csak a számítógépekre lenne alkalmazható.

Az analóg és digitális technika, sőt ugyanazon eszköz vagy jelenség analóg és digitális szemlélete más területeken is megkülönböztethető: a karóránk is lehet analóg vagy digitális, vagy egy voltmérő műszer is lehet ugyanaz, sőt egy matematikai függvény megadása is lehetséges diagrammal vagy értéktáblázattal.

Az analóg számítógépek olyan gépelemekből épülnek, amelyek valamilyen folytonosan változó fizikai mennyiséggel jelképezik, jelenítik meg a "számítás" eredményét, és - elvileg - a bemenő adatok legkisebb változására is képesek reagálni.

A digitális számítógépek kapcsoló jellegű üzemben működnek. A számokat itt a számjegyeknek megfelelő kapcsolóelemek ábrázolják. Nem okvetlenül, sőt elsősorban nem mechanikus és kézzel mozgatható kapcsolókra kell gondolnunk, hanem ún. kapcsoló üzemre. Kapcsoló üzem bármely áramkörben lehetséges, ahol nem azt nézzük, hogy mekkora áram folyik, hanem azt, hogy egyáltalán folyik-e áram vagy sem; vagy két adott pont között a

feszültség pl. 0V vagy 6V. Átvitt értelemben kétállású kapcsolónak tekinthető az is, hogy egy mágnesgyűrű az egyik vagy a másik irányban van mágnesezve; vagy akár az, hogy egy darab papírnak néhány négyzetmilliméternyi felülete ki van lyukasztva vagy nincs kilyukasztva. Tehát kétállású kapcsoló elképzelhető mechanikusan is, elektronikusan is, mágnesesen is. A digitális eszköz lényegében olyan jelekkel dolgozik, amelyek egymástól élesen és jól elkülöníthetők; mivel a gépelemek száma véges, ezért a lehetséges különböző állapotok száma is véges, következésképpen a számolási pontosság véges. (Egy digitális eszköz az egymáshoz nagyon közeli értékeket általában nem tudja megkülönböztetni.)

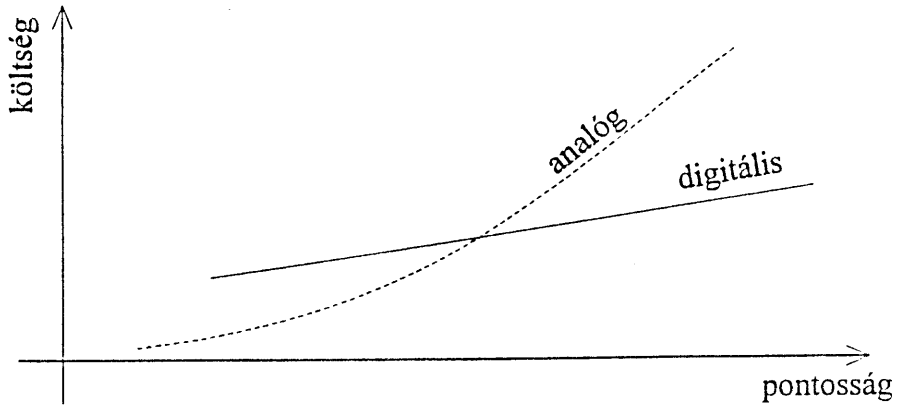
Első pillanatban szinte érthetetlen a digitális eszközök térhódítása. Az analóg eszközök elvileg pontosak, a digitális eszközök nem: a $\sqrt{2}$-t analóg eszközzel (körzővel) elvileg "végtelen pontosan" meg tudjuk szerkeszteni, ugyanakkor egy digitális eszközben csak véges pontossággal ábrázolhatjuk (hiszen az ábrázolt számjegyek száma szükségképpen véges). A mennyiségek, amelyekkel dolgozunk (például egy szabályozási folyamatban a hőmérséklet, az elektromos áram), rendszerint analóg mennyiségek: az analóg mennyiségeket - megint csak szükségszerű pontosságvesztés árán - digitális jelekké kell alakítanunk, ún. analóg-digitális konverterekkel. Tipikus AD-konverter például az autó kilométer-számlálója. A megtett út analóg (folytonos változású) mennyiség, azonban a digitális mérőeszközünkön például az 1-es szám nem jelzi, hogy vajon 1000 m-t, 1638 vagy éppen 1999 m-t tettünk-e meg. Végül az analóg eszközök "tálcán kínálnak" bizonyos bonyolult matematikai műveleteket. Ilyen például az integrálás, a differenciálás. Tudjuk, hogy a kondenzátor feszültsége arányos a rajta átáramló töltéssel, ami nem más, mint az áram idő szerinti integrálja:

$$u(t) = \frac{1}{C} \int i(t)\, dt$$

Az egyszerű kondenzátor tehát elvileg "pontosan" tud integrálni, ugyanezt egy digitális eszköz bonyolult, közelítő módszeren alapuló programmal végzi, végső soron "pontatlanul".

Tekintve, hogy az analóg gépeknél folytonos működésű műszerekről van szó, a pontosságot csak úgy fokozhatjuk, ha egyre finomabb kidolgozású, következésképpen egyre drágább gépelemeket használunk. Ahol viszont kapcsolók vannak, ott a kapcsolóállások egymástól - viszonylag olcsó gépelemek esetén is - jól elkülöníthetők és a pontosság fokozására pedig egyszerűen az az eszköz áll rendelkezésünkre, hogy kapcsolókból többet használunk. Nem csak egész számok, hanem törtrészek is modellezhetők

kapcsolóhelyzetekkel, tehát - mint ahogy a számtanban több számjegy leírásával -, úgy a digitális gépben: több kapcsoló használatával lehet elérni a nagyobb pontosságot. Ha tehát egy diagramot készítünk, ahol a pontosság függvényében a gépépítési költségeket szeretnénk ábrázolni, akkor a digitális gépek pontosságának a függvényében a rájuk fordított költség növekedése közel lineárisnak tekinthető, mert hiszen kétszeres pontosság eléréséhez kétszerannyi kapcsolót kell használni. A pontosság fokozása az analóg gépek esetében pedig meredek költségnövekedéssel jár, mihelyt egy bizonyos határon túljutunk (2.ábra). Többek között ennek az egyszerű felismerésnek köszönhető, hogy az utóbbi négy évtizedben a digitális gépek sokkal nagyobb mértékben elterjedtek, mint az analóg gépek.



2.ábra

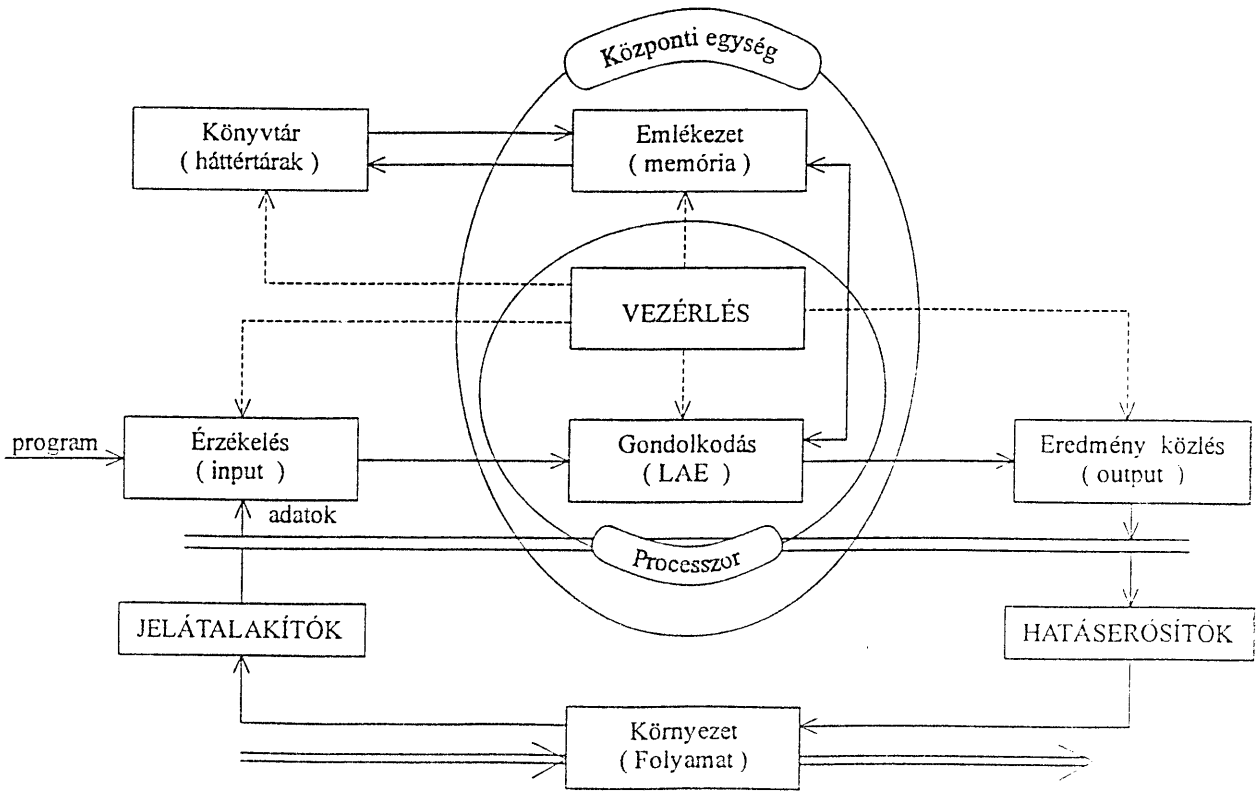
1.3. Párhuzam az ember és a számítógép között

A ma ismert legáltalánosabb számítóeszköz, az ún. Turing-gép elvi leírását Turing angol matematikus adta meg 1936-ban. Munkássága nyomán élénk kutatás indult meg a számítások automatizálása terén. Megszülettek az első, relékkel és jelfogókkal működő gépek, amelyek az ember számára szinte beláthatatlan mennyiségű számítást végeztek el igen rövid idő alatt. A második világháború meggyorsította a fejlesztést: az angol haditengerészet például ilyen géppel fejtette meg (dekódolta) a német tengeralattjárók rejtjelezett (kódolt)

rádióüzeneteit. Ezek a gépek azonban még mindig inkább a bevezetőben említett automaták rokonai. Fix programmal működtek, a "program" lényegében maga a gép szerkezete volt. Ahhoz, hogy más algoritmus szerint működjenek, valamiképp módosítani kellett a gép szerkezetét.

Neumann János 1948-ban zseniális felismerésre jutott: egy igazán általános célú számítóeszköznél a program éppúgy "adat", mint maguk az "igazi adatok" - tehát a programot is a memóriában lehet - sőt, kell! - tárolni, akárcsak az adatokat. Az adatok és a program fizikai megjelenítése is legyen egyforma, a gépet pedig úgy kell megépíteni, hogy a memóriát olvasva mindig tudja, vajon egy "adat" programlépést, műveletet jelent, vagy operandust, amivel a műveletet végre kell hajtani.

Az alábbiakban a számítógép legteljesebb kiépítésű változatát vázoljuk, amikor is feltételezzük, hogy a számítógép - bizonyos tekintetben - már ugyanúgy dolgozik, mint az ember. Érzékeli a környezetében lejátszódó folyamatokat, azokból mérési adatokat vesz fel, ezeket feldolgozza, a feldolgozás eredményeképpen döntéseket hoz, ítéleteket alkot és - döntései nyomán - be is avatkozik a folyamatba (3.ábra).



3.ábra

Automatizálási értelemben ez egy zárt hatáslánc. Az ábra alján a folyamatot ábrázoltuk és feltételezzük, hogy a folyamatból valamilyen megfigyelés vagy mérés útján - vagy a közvetlen érzékszerveinkkel, vagy műszerek útján (jelátalakítók) - eljutnak bizonyos adatok az érzékszerveinkhez (érzékelés). Az érzékelés nyomán az adatok az ember agyába jutnak és az ember gondolkodóképességének lesznek alávetve (gondolkodás). Az emberi agynak azonban nem a gondolkodás az egyetlen szerepköre, hanem rendelkezik még emlékezőtehetséggel is (emlékezés). A gondolkodás az emlékezéssel kétirányú kapcsolatban van: tehát a gondolatainkban megforduló adatokat képesek vagyunk az emlékezetünkben eltárolni, illetve - szükség szerint - onnan előszedni. A gondolkodás eredményeképpen születő döntéseink, ítéleteink a közlőszerveken keresztül mehetnek vissza a környezetünkbe; a közlőszerveinkből kimenő információk pedig - bizonyos hatáserősítőkön keresztül - visszajuthatnak a folyamathoz és azt valamiféleképpen befolyásolhatják. Ez valójában már majdnem a teljes emberi tevékenység, de a leglényegesebb dologról még nem emlékeztünk meg. Az emberi agy ugyanis nem csak gondolkodik és emlékezik, hanem mindezeket a tevékenységeket önmaga felügyeli is, röviden: vezérli (vezérlés). Az ábrán szaggatott vonal jelzi azt, hogy a vezérlés mindegyik másik funkcióra hatással van: tehát vezéreljük az érzékszerveink működését, vezéreljük a gondolkodásunkat és természetesen vezérlőimpulzusokkal látjuk el az emlékezetünket is, nemkülönben a közlőszerveinket.

Ebben a legáltalánosabb sémában, ahol eddig csupán az ember funkcióiról esett szó, a Neumann-féle számítógépépítési alapelvek már benne vannak. Ezek a következők:

Szükség van memóriára: az emlékezés szerepét tehát a számítógép memóriája fogja betölteni. A memóriában azonos fizikai megjelenítésben, kódoltan vannak tárolva a környezetből beérkező adatok és ezen adatok feldolgozását irányító utasítások, vagyis a program. Neumann azt is bebizonyította, hogy az információ tömörsége és kezelhetősége szempontjából a legcélszerűbb a kettes számrendszeren alapuló kódolás.

Neumann első alapelvében az az újszerű, hogy a program is a tárban van, éspedig ugyanolyan fizikai megjelenítésben, mint bármely adat. Tehát ha mágnesekben rögzítve, akkor mágnesekben, ha félvezetős a memória, akkor a félvezetőkön bizonyos pontokon mérhető feszültségekben rögzítjük az adatokat és az utasításokat egyaránt - és ezeket együttesen nevezzük a számítógép rendelkezésére álló információknak.

A tárolt utasítások alapján képes a gép soklépéses feladatokat önállóan, kezelői beavatkozás nélkül megoldani.

Szükség van központi vezérlő egységre, amely különbséget tesz adatok és utasítások között, és az adatokat az utasításoknak megfelelően feldolgozza.

Szükség van a műveleteket ténylegesen végrehajtó ún. logikai-aritmetikai egységre (LAE), amely - mint neve is mutatja - egyszerű logikai és számtani műveletek elvégzésére képes.
Amiről eddig szó volt: a vezérlés, a gondolkodás és az emlékezés szerepét eljátszó gépegységek együttesen alkotják a számítógép ún. központi egységét. A központi egységen belül a vezérlőegységet és a logikai-aritmetikai egységet processzornak is szokták hívni. Tehát a processor a tényleges eljárást lefolytató gépegység (processus - eljárás).
Szükség van a számítógép környezetével kapcsolatot tartó perifériális egységekre: input és output egység (bemenő és kimenő egység).
A perifériák feladata kettős. A központi egység elektromos vagy mágneses jelekkel "dolgozik"; ezeket az ember az öt érzékszervével nyilván nem tudja érzékelni. Tehát a perifériák szerepe egyfelől az, hogy az ember által érzékelhető vagy leadható jelekből olyat csináljon, amelyet a központi egység érzékelni tud és fordítva (jelátalakítás).
A perifériák szerepe másfelől annak a sebességkülönbségnek az áthidalása, amely a gyors központi egység és a lassú emberi működés között fennáll. A központi egység, minthogy elektronikus alegységekből áll, nyilvánvalóan sokkal gyorsabban fog működni, mint az ember keze, amellyel adatokat gépel be esetleg, de gyorsabban működik az elektromechanikus eszközöknél is. (A leggyorsabban mindig az elektronikus egységek, a leglassabban a mechanikus egységek működnek.)
A perifériák közé szokás sorolni még az ún. háttértárakat is, amelyek a számítógép memóriájának a kapacitásbővítésére szolgálnak. A háttértárak természetesen szintén a központi vezérlőegység felügyelete alatt állnak (az ábrán szaggatott vonallal jelképezett vezérlőimpulzusok).
Megkülönböztetésül a háttértáraktól, a központi egységhez tartozó tár neve: operatív tár (operatív - működő, a működésbe folytonosan beavatkozó).
A Neumann-féle számítógépépítési alapelvek, e négy követelmény lényegében magában foglalja a mai számítógépek szerkesztési alapelveit is. Az adatfeldolgozás mennyiségi növekedése újabb gépegységek beiktatásához mint minőségi változáshoz vezetett, a korszerű építőelemek pedig sokkal gyorsabb működésű és nagyobb kapacitású gépek építését eredményezték; ez a fejlődés azonban a neumanni alapelvek lényegét nem érintette.

Megjegyezzük még, hogy az ábrán a szabályozott folyamattal állandó, emberi közvetítés nélkül kapcsolatban álló, ún. on-line számítógépet vázoltunk. Itt az ember közvetlen beavatkozására nem kerül sor, az ember csak szemlélője egy ilyen komplex módon automatizált folyamatnak. Ma a számítógépek többsége még nem ebben az automatizáltsági mértékben üzemel, hanem az

ábrán látható dupla vízszintes vonal feletti kiépítettségben, a folyamattól elválasztva (off-line).

A számítógép tehát on-line üzemel, ha a programjának működéséhez szükséges adatokat közvetlenül, emberi beavatkozás nélkül kapja meg (például a szabályozandó folyamat lényeges paramétereit a mérőkészülékek elektromos jelei formájában), és ha a program eredményeként kapott adatokat közvetlenül adja át a szabályozandó folyamatnak, vagyis emberi beavatkozás nélkül, közvetlenül szabályozza.

Ellenkező esetben a számítógép off-line üzemel: az adatbevitel az ember dolga, a számítógép meghatározza a teendőket, majd azokat ismét az ember végzi (végezteti) el. Tényleges gyakorlati problémák megoldásánál is rendszerint a feldolgozandó adatokat nem a mérőműszerekről közvetlenül vesszük, hanem a mért adatokat előbb kézi erővel valahol rendszerezzük és aztán úgy adjuk a számítógépnek. Nem is szólva a másik oldalról, a számítógép output oldaláról, amikor is a gép eredményeket közöl; például termelésirányítás esetén sokszor nem hagyjuk a számítógép döntéseit közvetlenül a termelési folyamatba beavatkozni, hanem összeül a termelési értekezlet és a számítógép által szolgáltatott feldolgozott adatokat vagy elfogadja, vagy elveti.

1.4. Informatika

1.4.1. Az informatika eredete

Az 'informatika' viszonylag új elnevezés. Francia (informatique) és német (Informatik) nyelvterületen használták először az 1970-es években. Magyarországon csak az utóbbi évtizedben kezdett tért hódítani.

1985-ben a Budapesti Műszaki Egyetem Gépészmérnöki Karán informatika laboratóriumot hoztak létre.

1987-ben a Villamosmérnöki Karon megkezdődött az informatika szakon az oktatás.

Ugyanebben az évben a Kandó Kálmán Villamosipari Műszaki Főiskolán is informatikus képzés indult.

Hasonló képzés - tartalmát illetően - természetesen már évekkel korábban is volt a tudományegyetemeken, műszaki egyetemeken és főiskolákon.

Az informatika az elektronika és a matematika határterületén jött létre. Erre utalnak az angol nyelvterületen korábban használt kifejezések: Computer Science (számítógép-tudomány) és Computing Science (számítástudomány). Az utóbbi évtizedekben azonban előbb az Informatics, majd az Information Technology kifejezések használata vált általánossá.

Korábban az informatikát csak tapasztalati tudománynak tekinthettük, hiszen alapvetően a különböző tudományterületekről és a gyakorlatból vett eljárások és fogások keveréke volt.

A 70-es évektől kezdve mégiscsak egyre önállóbb tudománnyá vált.

1.4.2. Az informatika részterületei

Az informatikának négy részterülete alakult ki: a műszaki informatika, az alkalmazott informatika, az alapinformatika és az elméleti informatika.

A műszaki informatika területe egy előre meghatározott tulajdonságú számítógép megépítése.

Az alkalmazott informatika területe a számítógépek használata, alkalmazása a legkülönbözőbb területeken. Az alkalmazott informatikus nyilván team-munkában dolgozik, együttműködve a megfelelő alkalmazási területek szakértőivel.

Az alapinformatika az előbbi két részterület között helyezkedik el, mintegy hidat képezve közöttük. Ebből adódóan kettős feladatot tölt be: az egyik oldalról a berendezéseket kell ellátnia olyan rendszerprogramokkal, amelyek a szükséges alkalmazói programok készítéséhez nyújthatnak megfelelő feltételeket, a másik oldalról viszont - ugyancsak a hatékony használhatóság érdekében - a berendezésekkel szemben támasztott követelményeket is fel kell tárnia, hogy közvetíteni tudja ezeket a műszaki informatikusoknak. Minthogy azonban a számítógépek tervezésénél a különböző célokra való használhatósággal szemben elsősorban inkább a műszaki megvalósítás lehetőségei dominálnak, az alapinformatikusnak gyakran utólag kell alkalmassá tennie a meglévő gépeket bizonyos feladatok elvégzésére. Jóllehet a fejlődés a műszaki informatika és az alapinformatika közötti határ elmosódását valószínűsíti, a számítógépek és az alkalmazások közötti távolság bizonyára még sokáig megmarad.

Az informatika tudományos alapjainak megteremtésével az elméleti informatika foglalkozik. Ennek a területnek köszönhető, hogy az informatika mint tudomány elfogadottá vált. A fejlődés bár óriási, a gyakorlatban alkalmazott módszerek az elméletek előtt járnak - az informatika még mai is inkább kísérleti jellegű tudomány.

1.5. A digitális számítógép használata

1.5.1. Közvetítés a gépi kód és a magas szintű nyelvek között

A ma üzemelő, korszerű, elektronikus elemekből épült számítógépek döntő többsége digitális.

Digitális elemekből épül: kétállású kapcsolók vannak benne; tehát ha a géppel valamit csináltatni akarunk, akkor ezekre a kapcsolókra kell hatást gyakorolnunk. A számítógép nem ember, tehát semmiféle nyelven nem ért, csakúgy, mint a villanykapcsoló, vagy egy motornak a ki-be kapcsolója. A digitális számítógép is ilyen ki-be kapcsolókkal rendelkezik, és ezeket a kapcsolókat kell billegtetnünk ahhoz, hogy a számítógép működjék. A digitális számítógépnek azonban ilyen kapcsolókból sokkal több "jutott", mint például egy szivattyút meghajtó aszinkronmotornak. Például 2 kapcsoló már 4-féle variációban állítható be $\left(2^2 = 4\right)$, 3 kapcsoló már 8-félében $\left(2^3 = 8\right)$ ... stb. ... 12 kapcsoló pedig $2^{12} = 4096$-féle variációban.

A számítógép memóriájában lévő kapcsolókat csoportosítjuk. A leggyakoribb csoportosítás: 8 kapcsoló = 1 egység (1 byte), ami $2^8 = 256$ variációt ad. A memóriát rekeszekre osztjuk, például úgy, hogy minden rekeszben 8 kapcsoló legyen, vagyis 1 byte-nyi információt tároljunk.

Kétállású kapcsolóink állhatnak "felfelé" vagy "lefelé". Szimbolikusan az egyik állapotot 1-gyel, a másikat 0-val jelöljük. Ekkor egy memóriarekesz tartalma (a benne lévő kapcsolók állása) ilyen jellegű kifejezésekkel írható le:

01101010, 11001101 ... stb. (256 variáció). A 0 és az 1 a kettes számrendszer számjegyei: bináris számjegyek. Ebből származik a "bit" elnevezés (binary digit = kettes számrendszerbeli számjegy). A bit az információ legkisebb egysége. Saját "közlendőnket" ilyen, kettes számrendszerbeli (bináris) jelsorozatokkal kell kifejeznünk. Neumann első alapelve szerint "közlendőnk" kétféle lehet: utasítás vagy "valódi adat". Utasításainkat és adatainkat tehát a gép számára érthető jelrendszerbe (gépi kódba) kell átírnunk. Minden adatot kettes számrendszerben kell ábrázolnunk. Még nagyobb baj, hogy a gépnek közvetlenül, gépi kódban kiadható utasítások száma igen csekély. A processzorok utasításkészlete (a különböző gépi utasítások száma) mindössze néhány tucat, ritkán haladja meg a százat, és általában egyetlen igazi aritmetikai utasítás van köztük: az összeadás. Minden más utasítás logikai jellegű. Természetesen ez nem azt jelenti, hogy a 27x33 szorzat értékét csak úgy kaphatjuk meg, ha 27-szer elvégeztetjük az "add össze" utasítást. Egy logikai művelettel (az eltolással) mindez leegyszerűsíthető. Azt azonban látjuk, hogy ahol már egyetlen szorzás is bonyodalmakat okoz, ott alaposan meg kell kínlódnunk a programkészítéssel.

Az előbbieket egy egyszerű feladattal illusztráljuk. Adjunk össze két egész számot, például 17-et a 22-vel. Tegyük fel, hogy a 17 már benn van a memória 113-ik rekeszében, a 22 a 114. rekeszben, és az eredményt a 115. rekeszbe akarjuk tenni.

Gépünknek egyetlen összeadás-utasítása van: ez egy speciális tárolóhelyen, az ún. akkumulátorban lévő számhoz hozzá tud adni egy, a memória meghatározott területén lévő számot, és az összeadás eredményét az akkumulátorban tárolja. Teendőink tehát a következők:

1. utasítás: "vidd be a 113-ik rekesz tartalmát az akkumulátorba"

2. utasítás: "add hozzá a 114. rekesz tartalmát az akkumulátor tartalmához"

3. utasítás: "vidd ki az akkumulátor tartalmát a 115. rekeszbe"

Legyen az	1. utasítás	műveleti bináris kódja:	10101101
	2. utasítás		01101101
	3. utasítás		10001101

Ezek az ún. "műveleti kódok", ezekről ismeri fel a processzor, mit kell tennie. A "mivel", vagyis "milyen adattal" azonban még nem tisztázott. A műveleti kód után meg kell adni, hol található az az adat, amelyet be kell vinni az akkumulátorba stb., vagyis meg kell adni az adat "címét": az adatot tároló memóriarekesz számát. 255 memóriarekesz esetén a címzés egy byte hosszúságú információval megoldható.

Példánkban a memóriarekeszek sorszáma: 113, 114, 115, azaz binárisan 01110001, 01110010, 01110011 .

Gépi programunk tehát:

memóriarekesz	1	2	3	4
	10101101	01110001	01101101	01110010
	műveleti kód	cím	műveleti kód	cím

5	6	113	114	115
10001101	01110011	... 00010001	00010110	00100111
műveleti kód	cím	bináris 17	bináris 22	bináris 39

(a program végrehajtása után)

Valamivel egyszerűbb leírást kapunk, ha a bit-variációkat négyes csoportokba foglalva egy-egy 16-os számrendszerbeli (hexadecimális) számjeggyel fejezzük ki:

1	2	3	4	5	6		113	114	115
AD	71	6D	72	8D	73	...	11	16	27

Gépi programunk elég "ijesztő", holott példánkban az összeadandók nagysága korlátozott (legfeljebb 0 és 255 közötti egész számokat adhatunk össze, összegük sem lehet 255-nél több).

Nem foglalkoztunk az adatok ki- és bevitelének nehéz problémájával, és a címzés (az adott memóriarekesz kijelölése) meglehetősen nehéz problémáit is

alaposan leegyszerűsítettük. Azt azonban már ennek alapján is elképzelhetjük, hány ilyen jellegű utasítást kellene leírnunk például az $\frac{x^n}{n!}$ függvény egyetlen helyettesítési értékének kiszámításához.

Programozhatunk tehát gépi kódban, de ez roppant fáradságos és temérdek hibalehetőséget rejt. Ha a számítógépek feltalálása óta még mindig csak így lehetne programozni, akkor néhány elszánt vállalkozón kívül alig akadna ember, aki számítógépet használ.

A számítógépek elterjedésének előfeltétele az volt, hogy olyan "nyelvek" szülessenek, amelyeken a felhasználók viszonylag könnyen meg tudják fogalmazni problémáik megoldásának algoritmusát, a gép pedig az ilyen nyelven megírt programokat át tudja alakítani megfelelő gépi utasítások sorozatává az ún. fordítóprogrammal. A fordítóprogram tehát egy speciális, gépi nyelven megírt program: bemenő adatai (inputja) egy mesterséges, de az emberi gondolkodáshoz közel álló nyelven megírt szöveg, az ún. forrásprogram sorai, kimenete (outputja) pedig maga is egy gépi nyelvű program, az ún. tárgyprogram, amely pontosan azokat a műveleteket végzi el, mint amit a forrásprogram előír. Természetesen a lefordított, gépi nyelvű program általában sokkal hosszabb, mint a forrásprogram.

A fordítóprogram kétféleképpen dolgozhat: vagy úgy, hogy először a teljes forrásprogramot lefordítja gépi kódra, és csak ezután indítható (futtatható) a program (az ilyen fordítóprogramot compiler-nek nevezzük), vagy úgy, hogy a forrásprogramot soronként fordítja ("értelmezi"), és az így kapott gépi utasításokat azonnal végrehatjtja (interpreter), vagyis a fordítás és a programfutás időben nem válik szét. A compiler munkája az írásban fordító emberéhez, az interpreteré a tolmácséhoz hasonlít.

A programozási nyelvek nagy része "feladatorientált", vagyis elsősorban egy bizonyos feladatcsoport megoldására alkalmas.
A FORTRAN például a műszaki számítások nyelve (FORmula TRANslator), míg a COBOL az üzleti életben való felhasználásra készült (Common Business Oriented Language). Ennek megfelelően egy szám szinuszának meghatározása FORTRAN-ban egyetlen utasítás, COBOL-ban alig lehet leírni, ugyanakkor egy formanyomtatványra kiírt kimutatás készítése a napi üzleti forgalomról a COBOL-ban egyszerű, FORTRAN-ban bonyolult feladat.

A programozási nyelveket teljesítőképességük alapján két nagy csoportra oszthatjuk: assembly nyelvekre és magas szintű nyelvekre.

Az assembly nyelveknél a forrásprogram egy utasítása általában egy gépi utasításnak felel meg (ennyi lesz belőle a fordítás után), vagyis a forrásprogram hossza összemérhető a gépi kódú program hosszával. A programozás mégis könnyebb, mint gépi kódban, mivel

- a műveleti kódok helyett meghatározott rövid szavakat, rövidítéseket írhatunk

- a tár egyes területeit "szimbolikus címekkel" jelölhetjük, pl. X1, X2, Z3. A műveletek megadásánál tehát nem kell konkrét tárterületre hivatkoznunk, megszabadulunk a címkiszámítástól, ami a programozás talán legnagyobb hibaforrása

- a bonyolult, de rutinszerű műveleteket (pl. input-output műveletek) előre megírt gépi részprogramok (subroutine-ok, makro-k) befűzésével egyszerűsíthetjük

- a programba megjegyzéseket írhatunk (magyarázó szövegeket), amelyek a program működését világosabbá teszik; ezeket a megjegyzéseket a gép fordítás közben figyelmen kívül hagyja.

A fenti gépi kódú program ASSEMBLER nyelven nagyjából a következőképpen nézne ki (feltéve, hogy az X1, X2, Z3 szimbolikus címek már konkrét tárterületet jelentenek, és az adatok benn vannak X1, X2-ben):

```
LDA X1    (LoaD to Accumulator, azaz töltsd be az akkumulátorba
           X1 tartalmát)

ADD X2    (ADD, azaz add hozzá az akkumulátor tartalmához
           X2 tartalmát; az eredmény az akkumulátorban lesz)

STA Z3    (STore Accumulator, azaz tárold az akkumulátor
           tartalmát Z3-ban)
```

Láthatóan az assembly szintű nyelv még mindig inkább a "gépi gondolkodáshoz" áll közelebb, de azért sokkal érthetőbb, áttekinthetőbb, mint a gépi kód.

1.5.2. A nyelvek szintaxisa és szemantikája

A magas szintű (általános célú vagy feladatorientált) programozási nyelvek olyan utasításokból állnak, amelyek többé-kevésbé hasonlítanak a matematikában, műszaki életben stb. megszokott jelrendszerhez. A fenti példának megfelelő utasítás szinte minden magas szintű programnyelven a következő:

$$Z3 = X1 + X2$$

Itt csak egyetlen szokatlan dolog van: ez az utasítás nem azt jelenti, hogy Z3 egyenlő X1 és X2 összegével (vagyis nem egyenlet, mint ahogyan a matematikában megszoktuk), hanem ún. értékadó utasítás: azt jelenti, hogy Z3 vegye fel X1 és X2 összegének az értékét. A gép kiszámolja, mi van az egyenlőségjel jobb oldalán (függetlenül attól, mi van éppen Z3-ban), és az eredményt beírja Z3-ba (bármi volt is ott).

A matematikában az

$$X = X + 1$$

kifejezés értelmetlen: nincs olyan véges X, amely önmagánál eggyel nagyobb lenne. A számítástechnikában viszont ez gyakori utasítás, hiszen azt jelenti: "vedd az X jelű rekesz tartalmát, adj hozzá egyet, és az eredményt tárold újra X-ben", vagyis növeld meg eggyel X-et.

Ha most fordított sorrendben nézzük a fenti három példát, látszik, milyen hosszú gépi program lesz egyetlen tömör, magas szintű nyelven kiadott utasításból. Belátható, milyen nehéz dolga van a fordítóprogramnak. Az Y=SIN(X) , azaz "számold ki X szinuszát, és az eredményt tárold Y-ban" magas szintű nyelven kiadott utasítás többszáz gépi műveletet igényel. De nehéz dolga van a fordítóprogramnak azért is, mert pontosan meg kell értenie, mit kívánunk.

Elérkeztünk a nyelvek szintaxisának és szemantikájának problémájához. (Szintaxis: a helyesírási szabályok összessége, vagyis az, hogy mit szabad leírni, és mit nem; szemantika: jelentéstan.) A helyesírásra a hétköznapi életben

is ügyelünk, holott egy apró tévedés többnyire nem okoz zavart. Ha azt írják: "számold ki X szimuszát", nagy valószínűséggel "tudni lehet", hogy a szinuszra gondoltak: segít az asszociációs képesség. A számítógép azonban nem asszociál: működése szigorúan determinisztikus és mindazt, amire "betű szerint" nem tanították meg, nem érti (nem is lenne célszerű, ha értené!).

A programnyelvek tehát szigorú helyesírási szabályokat írnak elő. Meghatározzák, milyen jelek (betűk, számok, speciális jelek) fordulhatnak elő a programban, az utasításokat milyen kulcsszavakkal kell jelölnünk, változóinkat (pl. X1, X2, Z3 stb.) milyen módon nevezhetjük el, hogyan választhatjuk el az egyes utasításokat stb. A szigorúság érthető: azt, amit leírunk, egy gépnek kell értelmeznie. Mindaddig, míg a forrásprogramban szintaktikai hiba van, a fordítóprogram általában nem hajlandó lefordítani. A számítógép egyik haszna, hogy pontosságra nevel.

Tegyük fel, hogy hibátlan algoritmus alapján szintaktikailag hibátlan programot sikerült írnunk. Lefordíttatjuk és elindítjuk. Az első futtatásoknál többnyire azt tapasztaljuk, hogy a program valamilyen hibajelzéssel leáll, vagy ha végigfut, az eredmény akkor sem az, amit várnánk. A hiba abból gyökerezik, hogy nem ügyeltünk a nyelv szemantikájára, jelentéstanára. Bármilyen nyelven programozunk, pontosan ismernünk kell annak szemantikáját, vagyis azt, hogy egy-egy utasításunk hatására a gép pontosan mit csinál. Ez elsősorban a nyelv strukturáján múlik, de függ a gép felépítésétől is. Ez a tény magyarázza, hogy miért nem lehet egyetlen magas szintű programnyelv ismertetésére korlátozni a számítástechnika oktatását: a bonyolultabb feladatoknál menthetetlenül zsákutcába jutnánk.

A fordítóprogram szemantikáján múlik, vajon az

$$X = Y/Z1 * Z2$$

utasítás hatására a gép $\frac{Y \cdot Z_2}{Z_1}$ - gyel vagy $\frac{Y}{Z_1 \cdot Z_2}$ - vel tölti fel az X változót.

A következő utasítások viszont - melyek az $\binom{n}{k}$ binomiális együttható értékét számolják ki - szemantikailag hibátlanok, a legtöbb gép mégis előbb-utóbb hibás eredményt ad:

A1 = 1*2*3*4* ... (n-ig leírjuk az egész számokat)
A2 = 1*2*3* ... (k-ig leírjuk az egész számokat)
A3 = 1*2*3* ... (n-k-ig leírjuk az egész számokat)
T = A1/(A2*A3)

Azt várnánk, hogy T egész szám lesz és így dolgoznánk vele tovább, pl. tömbök elemeit indexelhetnénk T-vel. A valóságban T nem mindig egész. Már viszonylag kis n-ek esetén is tört számokat kapunk eredményül, ami a továbbiakban komoly logikai hibát okozhat. Az ok egyszerű: a véges pontosságú számábrázolás és műveletvégzés. Elkerülni azonban csak úgy lehet, ha számítunk rá, ehhez viszont valamennyire ismernünk kell magát a gépet is.

A programozási nyelv tehát a megengedett utasítások (rendszerint angol nyelvű) kulcsszavainak összessége és a nyelv szintaxisa - együttesen. A szintaxis fogalomkörébe tartozik az, hogy

- milyen karakterek fordulhatnak elő a programban

- milyen szabályok szerint képezhetünk változóneveket, címkéket, saját függvény neveket stb.

- milyen sorrendben és milyen formátumban írhatjuk le az utasításokat, hogyan választhatjuk el őket egymástól

- milyen hosszú lehet egy-egy utasítás stb.

A nyelv szemantikája, jelentéstana: a nyelvben rögzített információk, mindenek előtt az utasítások értelmezési szabályainak összessége.

A programozási nyelveket általában gépektől függetlenül alakítják ki, és igyekeznek úgy megalkotni, hogy egy adott területen minél több feladatot meg lehessen oldani velük. Ezeket a - konkrét géptől elvonatkoztatott, "abstract gépre" elképzelt - nyelveket hivatkozási nyelveknek nevezzük. Közismert nyelvek például:

FORTRAN (FORmula TRANslator)
COBOL (Common Business Oriented Language)
ALGOL (ALGOrithmic Language)
PL/1 (Programming Language Nr.1.)
FOCAL (FOrmula CALculator)
BASIC (Beginners All purpose Symbolic Instruction Code)
PASCAL (a XVII. században élt francia matematikusról)

Amikor egy számítógépgyár valamelyik géptípusát alkalmassá akarja tenni egy magas szintű programnyelv fogadására - vagyis elkészíti a nyelvnek az adott gépre szóló fordítóprogramját - , gyakran dönt úgy, hogy a hivatkozási nyelv bizonyos utasításait elhagyja, mert saját gépén az annak megfelelő műveleteket nehéz, vagy éppen lehetetlen lenne megvalósítani. A hivatkozási nyelvnek az adott gépre való alkalmazását, adaptálását (ami tehát a lehetőségek valamelyes módosítását jelenti), implementálásnak nevezzük. Ebben az értelemben beszélünk olykor IBM-COBOL-ról, TPA-FOCAL-ról, COMMODORE-BASIC-ről stb. Adott programozási nyelv valamely gépi megvalósulását gépi reprezentánsnak is nevezik.

A fentiekből nyilvánvaló, hogy a fordítóprogram mindig "gépspecifikus", és a lefordított tárgyprogramok is azok (vagyis egy gépi kódú program rendszerint nem futtatható más típusú gépen, mint amilyenen fordították). A magas szintű nyelveken írt programok azonban elvileg gépfüggetlenek: ugyanazt a FORTRAN nyelvű forrásprogramot elvileg bármely gép FORTRAN fordítóprogramja le tudja fordítani, és futtatni tudja, vagyis a programokat forrás-állapotban cserélni lehet a gépek között. Ez azonban rendszerint csak elv: a különböző gépi reprezentánsok miatt a forrásprogramok cseréje rendszerint módosításokat kíván.

2. A technikai fejlődés hatása a programozás oktatására

2.1. A programozás tanításának történeti áttekintése

Tapasztalatok mutatják, hogy ha történeti elemzést adunk egy adott szakterületről, ez egyfajta motivációt adhat a hallgatónak, sőt a szóban forgó kérdéskör jobb megértését is szolgálja. A programozás tanításának történetét tanulmányozva az alábbiakban láthatjuk majd, hogy a problémamegoldás menetében fokról-fokra egyre erőteljesebben került előtérbe az algoritmus-szerkesztés szerepe, a konkrét programnyelvtől, ill. számítógéptípustól független algoritmizálás.

Ha valamely terület művelésére szánjuk el magunkat, legelőször is annak történeti áttekintésével kell kezdenünk.Az algoritmikus problémamegoldás tanításával foglalkozók számára a programozás oktatásának történeti áttekintése, fejlődésének, eredményeinek és hibáinak, történeti tanulságainak vizsgálata

elősegíti a jelen jobb megértését és a jövő igényeinek a lehető legpontosabb meghatározását. A történetiség elvének követelménye a szakoktatásra vitathatatlanul fennáll, de indokolt lehet ezt annak egy speciális részterületére, pl. a számítástechnika, az informatika, a programozás oktatására is adaptálni még akkor is, ha a szóban forgó tudományág viszonylag fiatal volta csak szerényebb méretű időívet enged vizsgálnunk. "A fejlettség igen gyakran nem a fejlődés természetes egymásutánjának, hanem egészen más okoknak az eredménye, aminek figyelmen kívül hagyása, vagy helytelen egyoldalú magyarázata téves következtetéseknek lehet forrása. A múlt megértése nélkül a fejlődés tanulságai a jelenre nézve érthetetlenek."[1] A programozás oktatástörténeti áttekintése magától értetődő természetességgel teszi majd helyére az algoritmus kitüntetett szerepét: a hangsúlyt az algoritmus és nem a program fogalmára kell helyeznünk.

Az 1950-es évekig a programozók néhány konkrét számítógép használatát tanulták meg. A gépek konstruktőrei tanították meg őket egy-egy adott számítógép használatára. Gyakorlatilag minden programozónak saját egyéni módszere volt arra, hogyan szerkesszen algoritmust az adott számítógépre. Kezdetben a programozás magukkal a számítógépekkel folytatott kísérletezést jelentette. Ebben az időben nem volt még szükség általánosan használható programozási módszerekre, már csak a korlátozott lehetőségek és a számítógépek szerényebb megbízhatósága miatt sem.

A 60-as éveket a programozási nyelvek koraként szoktuk emlegetni. A programozási módszerek egységesítésére vonatkozó törekvések magas szintű programozási nyelvek kidolgozásához vezettek. Több száz (többé-kevésbé gépfüggetlen) programozási nyelv született, így azután a programozás tanításának a kérdése is előtérbe került. Kezdetben ez a programozási nyelvek tanításában merült ki, az algoritmus-szerkesztés folyamatával az oktatásban nemigen foglalkoztak. Az oktatási céllal készült kidolgozott feladatok is inkább csak az adott programozási nyelv elemeinek tanítását célozták.

Az algoritmikus programozás kora akkor kezdődött, amikor a 70-es években fejlődésnek indult a programozás tanításának módszertana. A strukturált programozással kapcsolatos módszertani eredmények elsősorban E.W.Dijkstra, O.J.Dahl és C.A.R.Hoare nevéhez fűződnek.

Az előbbiek alapján elmondható tehát, hogy a programozás tanítása kísérleti jellegű tudományként indult, hiszen az kezdetben tudományokra és gyakorlati tapasztalatokra egyaránt támaszkodó eljárások és fogások együttese volt. Donald E. Knuth amerikai professzor 1968-ban megjelent könyvének címe: "The Art of Computer Programming" is arról árulkodik, hogy maga a

---

[1] Vígh, A.: *Az iparoktatás története*, Budapest, 1932.

szerző is inkább mesterségbeli ügyességről, a "beavatottak" művészi fokra fejlesztett gyakorlati tudásáról kíván szólni, semmint elméleti tudományról. A fejlődést jól mutatja E.W.Dijkstra 1976-ban megjelent művének címe: "A Discipline of Programming". Hasonlóképpen D.Gries 1981-ben napvilágot látott munkájának címe: "The Science of Programming", amely a programozástanítás történetútjának újabb mérföldkövét jelzi.

2.2. Algoritmus, folyamatábra és program Terminológia-történeti megfontolások

A programozás tanításának történetét áttekintve egyértelmű igazolást nyer koncepciónk, miszerint a hangsúlyt a program helyett az algoritmusra kell helyeznünk.

Nem lenne szerencsés, ha valamely problémamegoldó algoritmust mindjárt egy konkrét programnyelven kezdenénk írni, hiszen ez esetben a probléma helyett az adott programozási nyelv írná elő, hogy milyen részletekkel foglalkozzunk.

Valamennyire is összetettebb probléma esetén pedig, még az algoritmus, vagyis a megoldásra vezető képletes-szöveges utasításrendszer ismeretében is, közvetlenül számítógépes programot írni, - különösen kezdők számára - egyébként is veszélyes vállalkozás. A programhibák keresése és javítása sokkal több időnkbe és fáradságunkba kerülhet, mintha munkánkban végig áttekinthető rendet tartunk. A rendtartás egyik legjobb eszköze az algoritmus rajzos vázlata, a folyamatábra.

Folyamatábrát rajzolni lényegében annyit jelent, mint megfogadni a klasszikus tanácsot: a feladat megoldását "előre megtervezni", ógörögül "programozni", magyarul "elő-rajzolni". Ezért módszertani szempontból alapvetően célszerű valamely feladat számítógépi megoldása során a modell-algoritmus-folyamatábra-program utat végigjárni. Jóllehet valamely adott magas szintű programnyelvet szem előtt tartva rajzolhatunk ún. program-orientált folyamatábrát is, mégis a technikai fejlődés szülte szüntelen változások közepette még nagyobb jelentőséget kell tulajdonítanunk az általános folyamatábra készítésének, amely egyszersmind konkrét programnyelvtől, ill. számítógéptípustól független algoritmizálást is jelent.

Érdekes megfigyelni, hogy - amint ez az előbbiekből kitűnik - az "algoritmus" és a "program" szavak közül, eredeti jelentését tekintve a

"program" szó takarja azt a fogalmat, amit ma rajzos algoritmusnak, folyamatábrának mondunk. A program szót azonban ma az algoritmus kódolt változatára használjuk: amikor programozunk, valójában egy algoritmust írunk le valamilyen programozási nyelven.

Korunkban az 'algoritmus' szó a számítástechnika egyik alapvető szakkifejezéseként került be szótárunkba; ilyen értelemben csak a század derekán jelent meg először. A szó kialakulására vonatkozóan különféle elképzelésekkel találkozunk. E. Donald Knuth a "logaritmus" szó érdekes betűjátékát vélte felfedezni benne; néhány történész az I. században élt arab matematikus, Abu Dzsafar Muhammad Ibn Musza al Hvarizmi nevével hozza összefüggésbe. Ez utóbbit látszik alátámasztani az a tény, hogy angol lexikonok korábbi kiadásaiban a mai "algorithm" helyett az "algorism" szót találjuk meg, amely eredetileg arab számokkal végzett aritmetikai műveleteket jelentett. A szónak ebben az értelemben vett jelentése az idők folyamán feledésbe merült. Módosult alakjában már az új tudomány által használt fogalom nyert kifejezést: Az algoritmus olyan utasítások sorozata, amely azon műveletek sorrendjét határozza meg, amelyek adott feladat(csoport) bármely lehetséges kimenetelére nézve annak megoldását adják.

A definíciót átgondolva érezzük, hogy az algoritmus univerzális fogalom. A telefonfülkében elhelyezett "emeljük le a kagylót - várjuk meg a búgó hangot - dobjuk be az érmét - tárcsázzunk ... stb. ..." telefonálási útmutatótól kezdve, a két természetes szám legnagyobb közös osztójának meghatározására szolgáló (Euklidestől származó) mechanikus eljáráson keresztül egészen egy gépalkatrész gyártási folyamatának leírásáig valójában algoritmusokról van szó. A paprikás krumpli vagy a mákos beigli receptje is algoritmus, csakúgy, mint például a fecskék költözési ösztöne - még akkor is, ha ez utóbbi algoritmus lépéseit pontosan nem ismerjük. (Vajon maga a Világmindenség, amely minden részletében szigorúan tervezett törvényszerűségek szerinti felépítettséget és fejlődést mutat, nem - egy esetleg önmagát is befolyásoló - Algoritmus még ma is folytatódó "végrehajtása"?)

Amikor a tanárképzésben a tanárjelölteknek például a tanulás tanítását tanítjuk, voltaképpen a tanulás algoritmusát kell átgondolnunk, kidolgoznunk. A kezdő programozók tanításakor pedig magának a számítógépes feladatmegoldás menetének, "algoritmusának" (probléma - modell - algoritmus - program) vázolását használjuk az algoritmus fogalmának - talán legelső -

analóg megvilágítására. Korántsem véletlen ez utóbbinak szembeszökő hasonlósága Pólya György[2] problémamegoldásra ajánlott (itt csak vázlatosan közölt) algoritmusával:

1) a probléma megértése,
2) terv készítése,
3) a terv kivitelezése,
4) visszatekintés.

3. Az algoritmikus feladatmegoldás tanítása – Az algoritmikus gondolkodás jelentősége az oktatásban

3.1. A hagyományos és a számítógépes feladatmegoldás

Vizsgáljuk meg az algoritmus szerepét és helyét a problémamegoldásban. A feladatmegoldásnak bizonyos menetrendje természetesen már a számítógépek megjelenése előtt is kialakult. A következőkben a hagyományos és a számítógépes problémamegoldás munkafázisainak összehasonlító áttekintésével szeretnénk megvilágítani azt, hogy a tárolt programú számítógép használata mennyiben módosítja e menetrendet.

A hagyományos feladatmegoldás munkafázisai:

(1) a probléma felvetése és elemzése,
(2) a megoldás (legtöbbször matematikai) modelljének megszerkesztése vagy kiválasztása,
(3) megoldási terv készítése (a megoldás algoritmusának kidolgozása),
(4) a feladat tényleges megoldása az algoritmus útmutatása szerint,
(5) a kapott eredmények elemzése, felhasználása.

[2] Pólya, Gy.: *A gondolkodás iskolája*, Budapest, 1977.

A számítógépes feladatmegoldás munkafázisai:

(1) a probléma felvetése és elemzése,
(2) a megoldás (legtöbbször matematikai) modelljének megszerkesztése vagy kiválasztása,
(3) megoldási terv készítése (a megoldás algoritmusának kidolgozása),
(4) Az algoritmus számítógépi programba foglalása, majd a program számítógépi futtatása,
(5) a kapott eredmények elemzése, felhasználása.

Felületesen szemlélve azt gondolhatnánk, hogy a hagyományos és a számítógépes feladatmegoldás algoritmusa közötti egyetlen különbség, hogy a megoldási tervet az egyik esetben az ember, a másik esetben a számítógép hajtja végre. A két munkafolyamat közötti döntő különbség valójában persze az algoritmus előre kidolgozottságának mértékében és pontosságában rejlik. A hagyományos megoldásnál az algoritmus szervezése és végrehajtása akárhányszor egybefolyhat, mert a gondolkodó-számoló ember fejében a soron következő műveletek kiválasztásával legtöbbször a végrehajtás is együtt jár; ámbár terjedelmesebb feladat megoldásához mindig célszerű un. megoldási tervet készíteni, amely más szavakkal éppen az algoritmus többé-kevésbé pontos megszerkesztését jelenti.

3.2. Programnyelvtől, ill. számítógéptípustól független algoritmusok

3.2.1. A mondatszerű leírás vizualizálása (Pszeudokód - Folyamatábra - Struktogram)

Különböző algoritmusleíró eszközök közül választhatunk: pl. folyamatábra, struktogram, mondatszerű leírás. Mindegyiknek közös célja megoldási tervet készíteni egy adott feladattípusra, amely valójában a megoldás menetének géptől és konkrét programnyelvtől független, szemléletes, a logikai gondolatmenetet egyértelműen követő és a szerkezeti egységeket világosan tükröző leírása. Ezeket az algoritmusleíró eszközöket használhatjuk akkor is, ha nem okvetlen számítógéppel kell megoldanunk egy adott feladatot. A tanítási-tanulási folyamatban a problémamegoldásnak erre a kreativitást igénylő szakaszára, az algoritmus konstruálására kell koncentrálnunk, hiszen a végrehajtás csupán mechanikus tevékenység, akár az ember, akár a gép végzi azt.

3.2.2. A magasszintű programnyelvek alaputasításai

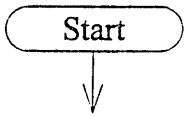
Alapvető folyamatábra szimbólumok

Az algoritmust - a jobb áttekinthetőség kedvéért - gyakran rajzos formában adjuk meg: ún. folyamatábrát, vagy más szóval blokkdiagramot készítünk.

A továbbiakban a (valamely) magas szintű nyelven programozó ember algoritmus-szerkesztő, folyamatábra-készítő tevékenységéhez szükséges legfontosabb ismereteket, a szokásosan használt szimbólumokat és ezek jelentését foglaljuk össze röviden.

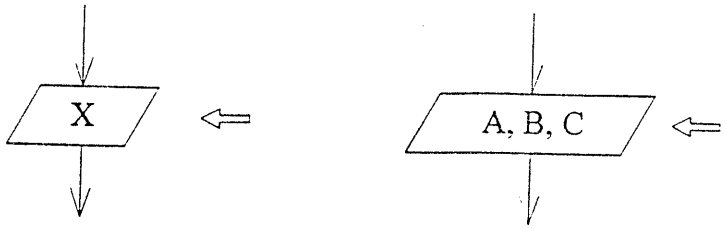
A folyamatábra szimbólumokkal kapcsolatban megjegyezzük, hogy az általunk használt jelölések nem kizárólagosak, az irodalomban számos más változattal is találkozhatunk.

A folyamatábra kezdetét a 4.ábrán látható szimbólummal jelöljük.



4.ábra

Az adatbeviteli utasítás révén a beviteli periférián keresztül adunk értéket valamely változó(k)nak (5.ábra).



a) b)

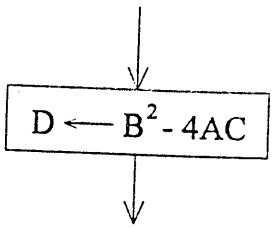
5.ábra

Az 5.a) ábrán az X nevű változó, az 5.b) ábrán az A, B és C nevű változók kapnak értéket adatbevitellel: a számítógép a beadott értékeket az X, ill. az A, B és C változónevekkel azonosított tárterületekre tárolja.

Az értékadó utasítás a bal oldalon álló (egyetlen) változónak ad értéket: a jobb oldalon álló aritmetikai kifejezés kiszámított értékét. 6.a) ábra.
Az értékadó utasítás végrehajtása két - időben egymás utáni - lépésből áll : a számítógép

- először megállapítja a jobb oldalon álló kifejezés értékét,
- majd ezt az értéket elrakja a bal oldalon álló változó nevével azonosított tárterületre.

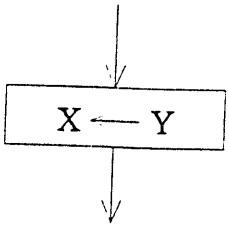
(A balra mutató nyíl szimbólum jól kifejezi az említett időbeli sorrendiséget.)



6.a) ábra

Tekintsük például a 6.a) ábra értékadó utasításának végrehajtását:
D „legyen egyenlô" $B^2 - 4AC$.

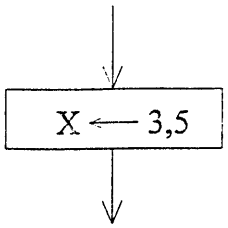
A számítógép „előveszi" a B, az A és a C változónevekkel azonosított tárterületekről az ott tárolt számértékeket, ezekkel az adatokkal kiszámítja a $B^2 - 4AC$ aritmetikai kifejezés értékét, majd elteszi ezt a D változónévvel azonosított tárterületre.



6.b) ábra

A 6.b) ábrán látható értékadó utasítás „X legyen egyenlő Y" hatására a számítógép „előveszi az Y nevű fiók tartalmát és ezt beteszi az X nevű fiókba". Mint minden hasonlat, a „fiókos hasonlat" is sántít: a számítógép különböző változónevekkel azonosított tárterületei egy kissé másként „működnek", mint az említett „ilyen vagy olyan nevű fiókok". Valójában,

amikor a számítógép előveszi az Y változónévvel azonosított tárterületről az ott tárolt számértéket, majd elrakja azt az X változónévvel azonosított tárterületre, akkor ezen tevékenység eredményeként mindkét tárterületen őrzi ugyanazt a számértéket. „Kivette az Y fiókból, majd elhelyezte az X fiókban." (De ezáltal az Y fiók nem ürült ki - miként a hasonlat sugallná -, hanem megmaradt az eredeti tartalma!)



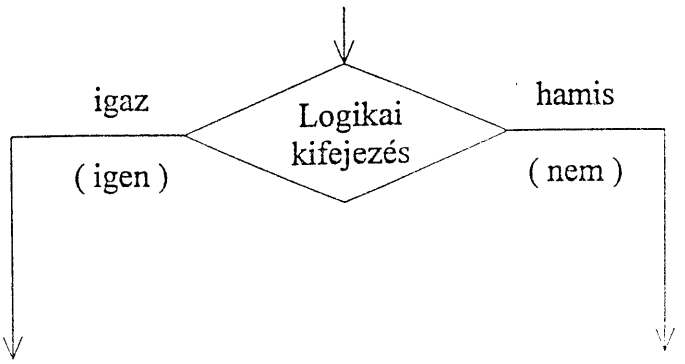
6.c) ábra

Végül a 6.c) ábrán is értékadó utasítást látunk: "X legyen egyenlő 3,5". A számítógép ez esetben is az utasítás jobb oldalával foglalkozik először: itt konstans számértéket talál, ezt tárolja el a bal oldalon álló változónévvel azonosított tárterületre. A jobb oldalon talált konstans számértéket, a 3,5-et "beteszi az X nevű fiókba".

Az értékadó utasítás jobb oldalán álló aritmetikai kifejezésben változók, konstansok és függvények lehetnek matematikai alapműveleti jelekkel összekapcsolva, ugyanakkor a zárójelezés is megengedett.

A függvények értékét a számítógép általában valamely előre megírt gépi rutin végrehajtásával számítja ki. Folyamatszervezéskor azokat a függvényeket használhatjuk az aritmetikai kifejezésekben, amelyeknek feldolgozó rutinját a választott programozási nyelv fordítóprogramja kezelni tudja.

A döntés révén a folyamatábra elágazik. A számítógépes program tényleges futása majd bizonyos feltétel(ek) teljesülésétől függően - vagylagosan - valamelyik utasítás(csoport) végrehajtásával folytatódik. A döntés tehát feltételtől függő vezérlésátadást valósít meg.



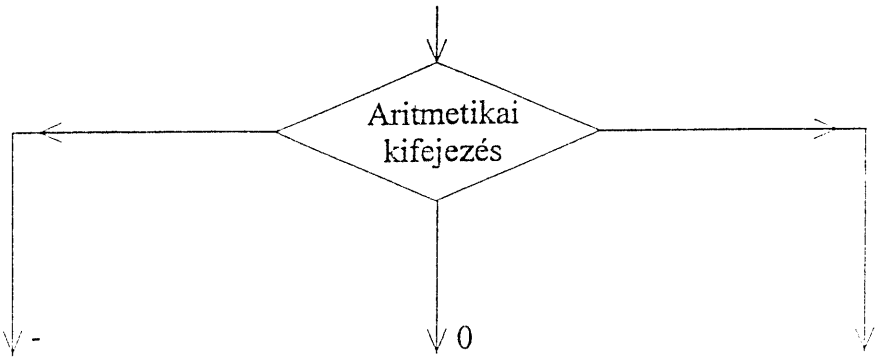
7.ábra

A logikai döntés kétfelé történő elágazást tesz lehetővé (7.ábra). A rombuszba feltételként logikai kifejezést írhatunk, amelyet kiértékelve logikai értéket: IGAZ-at vagy HAMIS-at (IGEN-t vagy NEM-et) kapunk eredményül. A logikai kifejezés legegyszerűbb formája egyetlen reláció. A matematikából ismeretesek a lehetséges relációjelek: $=$, $<$, $\leq$, $>$, $\geq$, $\neq$. Egyszerűbb logikai kifejezéseket logikai operátorokkal összekapcsolva tetszőlegesen bonyolult logikai kifejezést készíthetünk. Az alapvető Boole-operátorokat a 8.ábrán látható táblázatban foglaltuk össze.

Logikai művelet (Boole - operátor)		Általános Boole-algebrai jelölés és elnevezés	Halmazalgebrai analógia
VAGY (logikai összeadás)	OR	$\vee$ diszjunkció	$\cup$ halmazok egyesítése vagy uniója
ÉS (logikai szorzás)	AND	$\wedge$ konjunkció	$\cap$ halmazok közös része vagy metszete
Tagadás	NOT	$\neg$ negáció	$\overline{}$ kiegészítő vagy komplementer halmaz

8.ábra

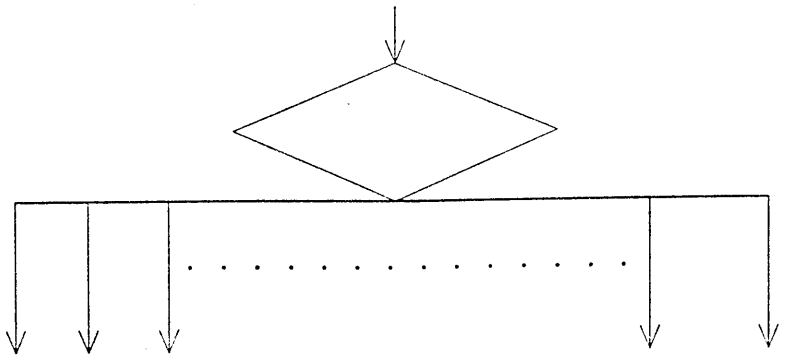
Az aritmetikai döntés folyamatábra szimbóluma a 9.ábrán látható. A rombuszba írt aritmetikai kifejezés kiszámított (aktuális) értékének előjelétől függően itt három irányba ágazhatunk el.



9.ábra

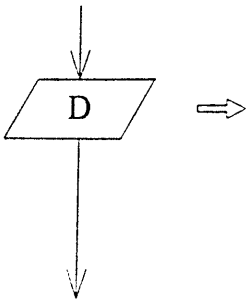
Természetesen a kétféle döntés bármelyike megvalósítható a másik segítségével is.

Háromnál többfelé történő elágazás folyamatábra szimbólumát adtuk meg a 10.ábrán.Ilyenkor például a rombuszba beírt aritmetikai kifejezés kiszámított értéke a különböző irányokba való elágaztatás feltétele lehet.



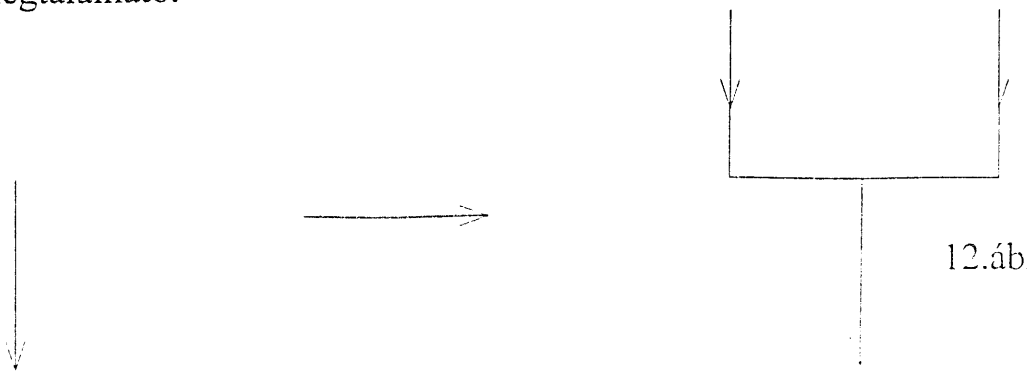
10.ábra

A 11.ábrán az eredmények kiíratására szolgáló adatkiviteli utasítás folyamatábra szimbólumát adtuk meg. Kézenfekvő, hogy ez hasonló az adatbevitel jelképéhez, az információ-áramlás ellentétes irányát a nyíl jelzi.



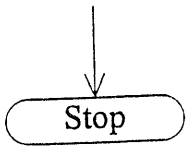
11.ábra

A folyamatábra egyes blokkjait folyamatvonallal kapcsoljuk össze, jelezvén a műveletek végrehajtásának sorrendjét (12.ábra). A programíráskor néha a folyamatvonalnak is utasítást feleltetünk meg. Ez a feltétel nélküli vezérlésátadást megvalósító ugró utasítás majdnem minden magas szintű nyelvben megtalálható.



12.ábra

A folyamatábra végét jelző szimbólumot a 13.ábrán láthatjuk.

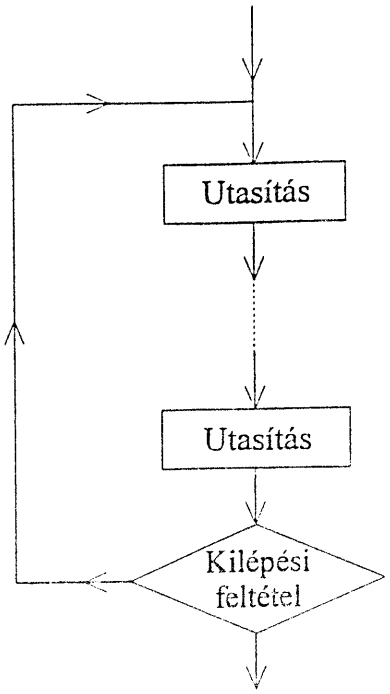


13.ábra

3.2.3. Ciklusok

A feladatok számítógépes megoldásában gyakran előfordul, hogy bizonyos utasításokat többször is végre kell hajtani. Ezt az utasítások ismételt leírása nélkül is megvalósíthatjuk.

A programozó munkája kevesebb - sőt adott esetben lényegesen kevesebb lehet -, ha egyes utasításokat csak egyszer-kétszer kell a programba írnia akkor is, ha azokat a gép bármilyen sokszor is hajtja majd végre. Ilyenkor a vezérlést a már végrehajtott utasításokra adjuk át feltételes (vagy feltétel nélküli) ugró utasítással. Az utasításokat összekötő folyamatvonalak hurkot, ún. ciklust alkotnak (14.ábra).

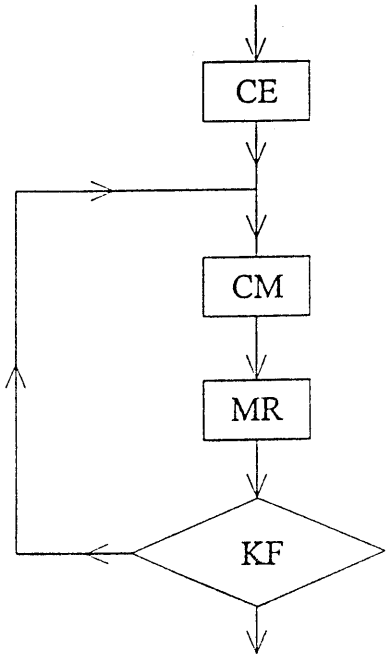


14.ábra

A ciklusban a többször végrehajtandó utasításokat a ciklus magjának nevezzük. A ciklusban feltételes vezérlésátadó utasításnak (döntésnek) is kell lennie, hogy a ciklusmag megfelelő számú végrehajtása után a programfutás tovább haladhasson, a ciklusból kiléphessen. A kilépés feltételének előbb-utóbb teljesülnie kell, és ezt úgy lehet elérni, hogy a feltételben szereplő változók értékét a ciklusban lévő utasítások ismételt végrehajtásával megváltoztatjuk.

3.2.3.1. Ciklusszervezés logikai döntéssel

A ciklus fő részei:



15.ábra

A 15.ábra rövidítéseinek jelentése:

CE: A ciklus előkészítése
A ciklus előkészítése többnyire kezdeti érték(ek) megadását jelenti.

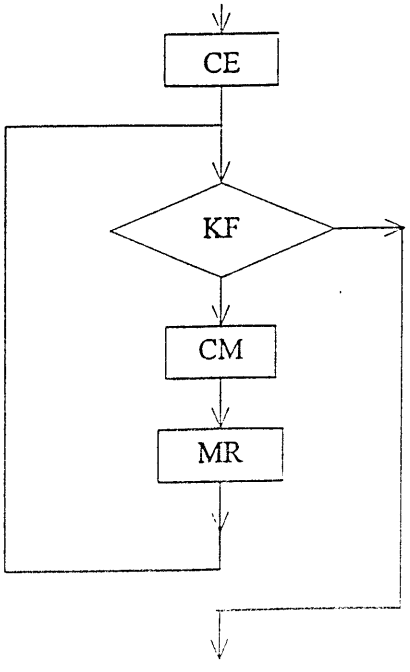
CM: Ciklusmag
A ciklusmag a többszöri végrehajtásra szánt utasítás(csoport). Állhat egyetlen utasításból, de lehet bármilyen bonyolult programrészlet is. Feltételes utasításokat vagy akár ciklusokat is tartalmazhat.

MR: Módosító rész
A módosító rész révén válik lehetővé, hogy a kilépési feltétel egyszer majd teljesül.

KF: Kilépési feltétel
A logikai döntés eredményeként vagy a ciklusmagot kell ismételten végrehajtani, vagy ki kell lépni a ciklusból.

A 15.ábrán ún. "hátul tesztelő ciklus"-t látunk: a ciklusmag után a módosító rész következik és a ciklus végén ("hátul") vizsgáljuk a kilépési feltételt. Jól látható, hogy ebben a ciklusszervezési sémában a ciklusmag legalább egyszer végrehajtásra kerül.

A hurokban szereplő blokkok sorrendje tetszés szerint választható meg. Így beszélhetünk például ún. "elől tesztelő ciklus"-ról is (16.ábra): itt mindjárt a kilépési feltétel vizsgálatával kezdődik a ciklus. Az ilyen fajta ciklusszervezésnél előfordulhat, hogy a ciklusmag utasításait a számítógép egyszer sem hajtja végre.



16.ábra

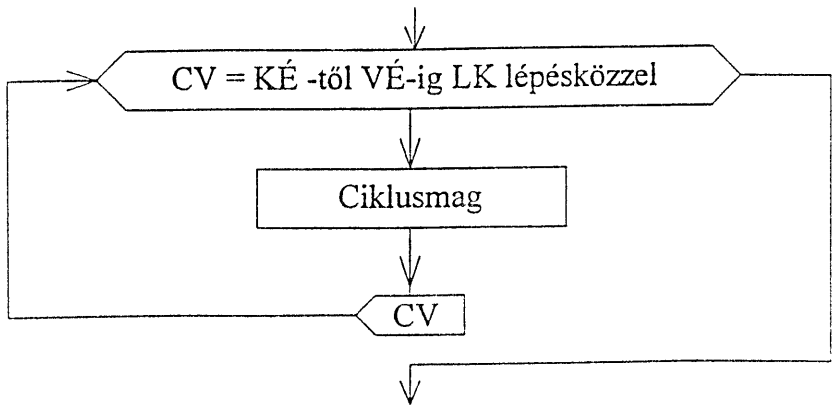
Vannak olyan természetű feladatok, amelyeknél már a ciklusba való belépés előtt eldől, hogy a ciklusmagot hányszor kell majd végrehajtani. Ilyenkor a végrehajtásokat számoljuk. A számlálóként használt változót ciklusváltozónak nevezzük. Azt, hogy a ciklus magja hányadszor kerül végrehajtásra, a ciklusváltozó mindenkori értéke mutatja, ezért e változó értékét - a ciklus minden egyes végrehajtásakor - egy értékadó utasításnak például eggyel növelnie kell (módosító rész). Az ilyen ún. számlálással vezérelt ciklusoknál tehát a ciklusmag végrehajtásának darabszámát előre rögzítjük.

3.2.3.2. Ciklusszervezés ciklusutasítással

A ciklusok használata a feladatok számítógépes megoldása során olyan gyakori, hogy a magas szintű programozási nyelvek ún. ciklusutasításokat is tartalmaznak. Ezek általában a számlálással működő ciklusokat vagy a 15. vagy a 16.ábra szerint valósítják meg anélkül, hogy a ciklust szervező

CIKLUSELŐKÉSZÍTÉS-t,
KILÉPÉSI FELTÉTEL-t és
MÓDOSÍTÓ RÉSZ-t

önálló utasítások formájában le kellene írnunk. Egy ilyen ciklusutasítás használata esetén már a folyamatábra készítésekor is lehetőségünk van a ciklus rövidített rajzolására. A ciklusutasítás folyamatábra szimbóluma a 17.ábrán látható.



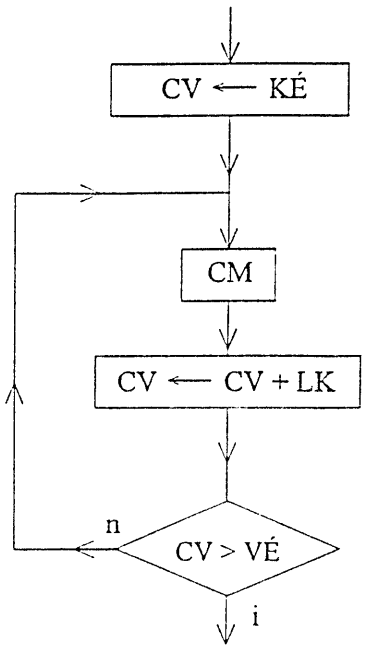
Az ábrán használt rövidítések:

CV: ciklusváltozó,
KÉ: kezdőérték,
VÉ: végérték,
LK: lépésköz.

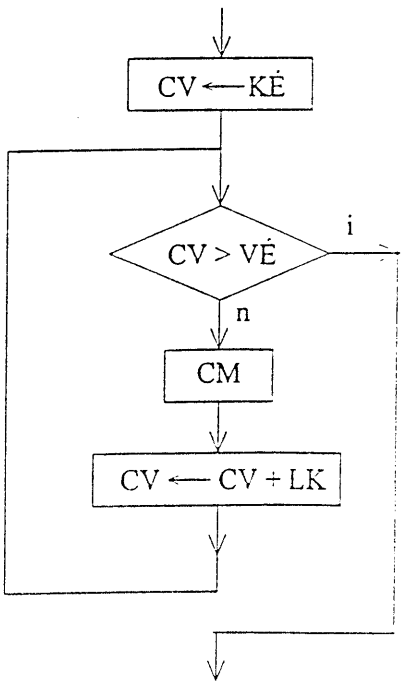
17.ábra

Ezek együtt azt jelentik, hogy a CV ciklusváltozó értéke a KÉ kezdőértéktől a VÉ végértékig az LK lépésköznyi értékkel változik és a ciklusmag utasításait a gép minden új érték felvétele után végrehajtja. Az újonnan használt hatszög alakú blokk neve: ciklusfej. Ez tehát "egy személyben ellátja" a CE ciklus-előkészítés, az MR módosító rész és a KF kilépési feltétel blokkjainak feladatát. A ciklusmag végét az ábrán látható ötszög alakú blokk jelzi, amelybe a ciklusváltozó nevét írjuk.

A 17.ábrán látható jelkép, a ciklusutasítás szimbóluma csupán annyit jelez, hogy (valamely) számlálással vezérelt ciklusról van szó. Magáról a ciklusutasításról tehát - a programnyelv ismerete nélkül - nem tudhatjuk, hogy a számlálással működő ciklusoknak a 18.ábra szerinti "hátul tesztelő", vagy a 19.ábra szerinti "elől tesztelő" változatát takarja-e.



18.ábra



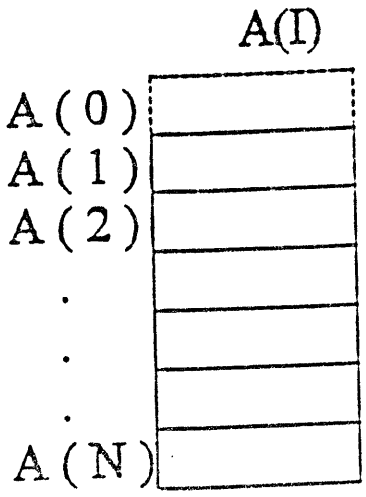
19.ábra

3.2.4. Indexes változók (Tömbök)

A magas szintű programozási nyelvek kivétel nélkül lehetővé teszik a tömbváltozók (indexes változók) használatát. A tömbváltozót a neve és az indexe együttesen azonosítja.

Az operatív tárat egy sokfiókos szekrényhez hasonlítva beszéltünk már A, B, C stb. nevű (index nélküli) változók tartalmáról mint az "A, B, C stb. nevű (egyes) fiókok" tartalmáról. Ezek az index nélküli változók "magányos fiókok" voltak az operatív tárban.

Tekintsünk egy valamilyen szempontból összetartozó, N elemű adathalmazt! Jogosan merül föl az igény, hogy ezeket az adatokat ne rendszertelenül tároljuk ("ne szórjuk szét a szekrény legkülönbözőbb elhelyezkedésű fiókjaiba"), hanem erre a célra jelöljük ki az operatív tár egy tömbjét. A 20.ábrán például az A(I) nevű egyméretű tömböt szemléltetjük, amely N db adat tárolására alkalmas, ha az index: I=1,2,...,N. A tömb egyes tárterületeit ezúttal csupán az index azonosítja: A(1), A(2), ... , A(N).

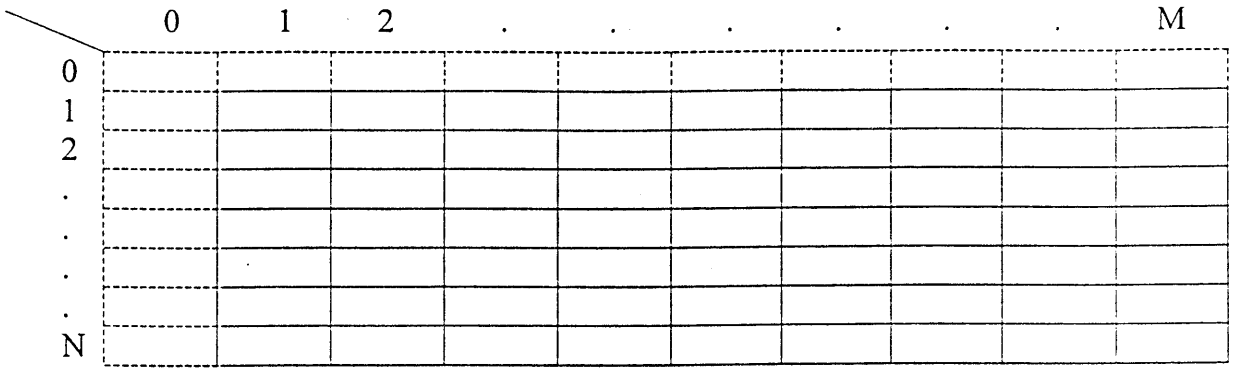


20.ábra

Az egyméretű tömb helyett nem véletlenül használatos a matematikából kölcsönvett vektor elnevezés is, amelynek elemeit egyindexes változóknak nevezik.

Ugyanúgy a kétméretű tömböt, amelynek elemei kétindexes változók, a matematikából kölcsönvett kifejezéssel mátrixnak nevezik.

Például egy NxM-es A mátrix elemeit célszerűen az A(I,J) nevű kétméretű tömbben tárolhatjuk (21.ábra). Az A(I,J) kétindexes változó a kétméretű tömb "I-edik sorában és J-edik oszlopában található fiókját" azonosítja.



A (I, J)

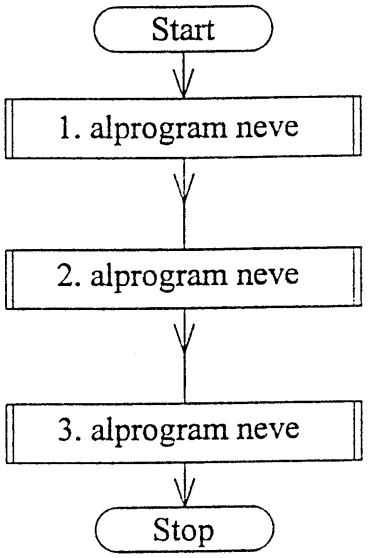
21.ábra

Természetesen lehetnek kettőnél nagyobb méretű tömbök is (kettőnél több indexes változók). Az indexek értéke mindig egész szám. A tömbváltozó indexébe konstanst, változót vagy aritmetikai kifejezést írhatunk.

Megjegyezzük, hogy az indexben szereplő kifejezés formájára és tartalmára nézve a programozási nyelvek különböző megszorításokat tartalmaznak.

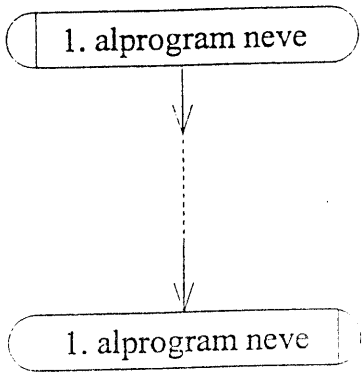
3.2.5. A folyamatábra tördelése

Bonyolultabb, nagyobb lélegzetű feladatok terjedelmesebb folyamatábrái gyakran nehezen áttekinthetők. Ilyen esetben célszerű először vázlatos, ún. fő folyamatábrát készíteni, amely rövidsége miatt könnyen áttekinthető és jól követhető (22.ábra).



22.ábra

A vázlatos fő folyamatábra programrészletekből, alprogramokból (más szóval: eljárásokból) állhat, amelyeket azután külön folyamatábrákon részletezhetünk (23.ábra).



23.ábra

Ennek a megoldásnak előnye az is, hogy a résztevékenységeket leíró folyamatábrák külön, akár más személy által is elkészíthetők és külön kipróbálhatók, lejátszhatók.

Az itt ismertetett módszer, a folyamatábrák tördelése nem csak az algoritmus (a folyamatábra) készítésekor nyújthat jelentős segítséget. Ha egy bizonyos részfeladatot a fő folyamatban több helyen is végre kell hajtani, úgy ezt az eljárást csak egyszer kell kidolgoznunk (legfeljebb a nevét kell többször leírnunk). Hogy ne csak a folyamatábrák, hanem a programok tördelésére (és az ismétlések elkerülésére) is lehetőség nyíljon, természetesen a magas szintű programozási nyelvekben is használhatók ilyen alprogramok (eljárások). Az eljárások tehát nem önálló programok, hanem a főprogramból hívható programrészek, alprogramok. A számítógép operatív tárjában általában csak egy példányban vannak jelen. Végrehajtásuk a főprogramból kezdeményezhető, de végrehajtásuk után a főprogram végrehajtása folytatódik az ott soron következő utasítással. Az eljárások egymásba ágyazhatók, vagyis az alprogramnak is lehet(nek) alprogramja(i). Az alprogramok szerkesztésekor a különböző programnyelvek eltérő szabályait kell figyelembe venni.

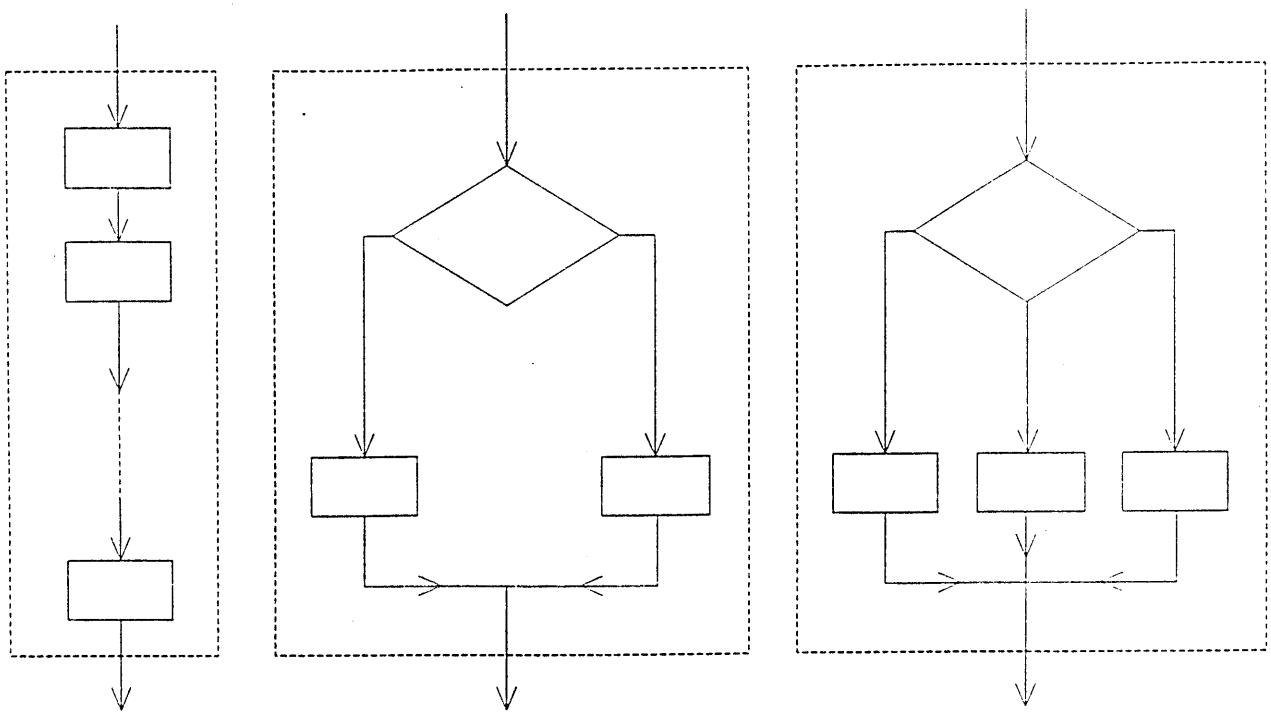
3.2.6. A strukturált programozásról

A vázlatos fő folyamatábra készítésének gondolata magában hordozza a modulrendszer érvényesítésének lehetőségét.

Az egyes programrészletekből, alprogramokból mint építőkockákból tetszőlegesen bonyolult építmények emelhetők. Maguk a bonyolult programok is - mint modulok - építőkövei lehetnek egy esetleg még magasabb struktúrának.

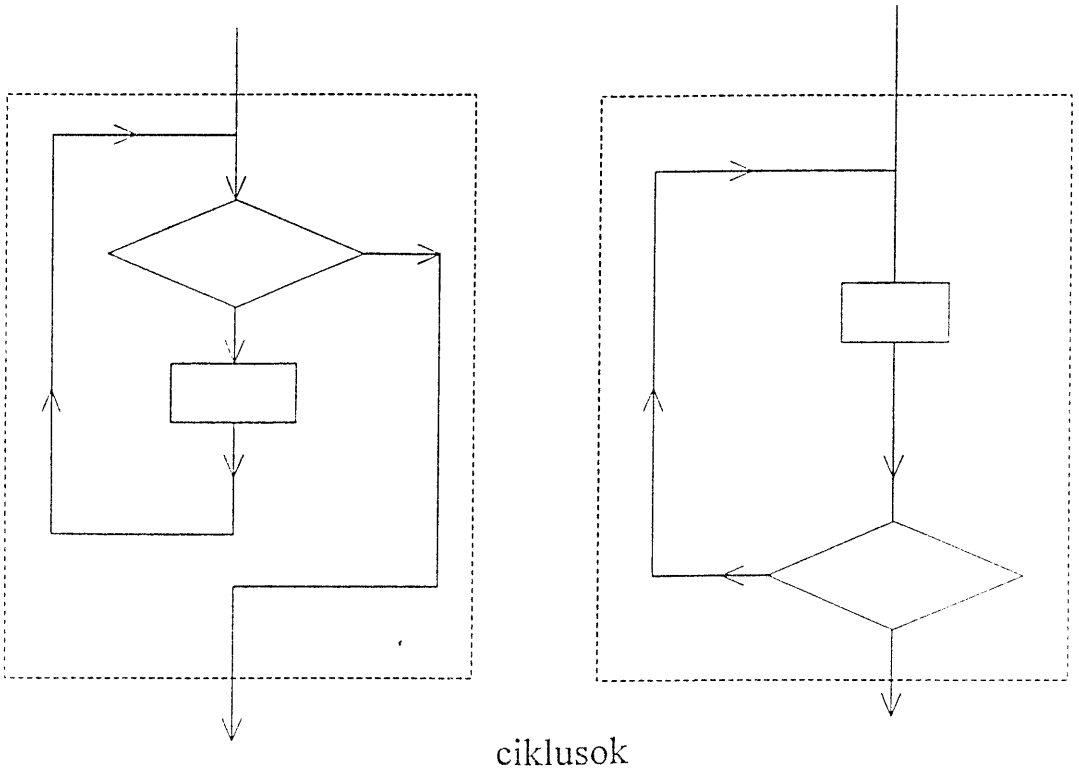
Ahhoz, hogy a leírt elvet érvényre juttathassuk, bizonyos értelemben szigorú rendet kell tartanunk az algoritmus-szerkesztés, a folyamatábra megrajzolása során. Csak olyan programrészleteket komponáljunk, amelyek valóban építőkocka módjára rendszerbe illeszthetők lesznek. Célszerűen egy ilyen egységnek (modulnak) egyetlen bemenete és egyetlen kimenete legyen. És megfordítva: olyan folyamatábrákat rajzoljunk, amelyeknek szinte bármely részlete "kiemelhető".

A továbbiakban bemutatjuk azokat a folyamatábra-struktúrákat (24.ábra), amelyeket a már megismert folyamatábra szimbólumokból építettünk össze, és amelyekből modulszerűen a legbonyolultabb algoritmusok is jól áttekinthetően felépíthetők. Az ábrán az egyes struktúrákat szaggatott vonallal határoltuk.



sorozat

elágazások



ciklusok

24.ábra

A hibakeresés megkönnyítésének igénye is azt szorgalmazza, hogy algoritmusaink megfogalmazásakor csakis ilyen folyamatábra-szerkezetekből építkezzünk, a strukturált programozás előbb tárgyalt elveit és szabályait szigorúan betartsuk.

4. Algoritmusokat vagy algoritmus-szerkesztést tanítsunk?

4.1. Az algoritmus-szerkesztés tanítása és a megoldási algoritmusok gép nélküli ellenőrzése

A teljes problémamegoldás úgy kezdődik, hogy a problémát elemezzük, utána ebből matematikai modellt alkotunk, a matematikai modellhez kidolgozunk egy algoritmust és végül az algoritmus alapján programot írunk: probléma - modell - algoritmus - program.

Tehát adva van valamilyen probléma és ehhez nekünk kell matematikai modellt felállítanunk. Olyan helyzetben vagyunk, mint a középiskolás diák, amikor szöveges egyenletet kell megoldania: a szövegben felvetett problémából valamilyen képletet, valamilyen modellt kell felírnia. Ilyenkor a szöveg lényegtelen részeit elhagyjuk és csak a lényegesekre koncentrálunk. Ez egy középiskolai példa szintjén fölöttébb egyszerű; az életben felvetődő problémákból azonban sokszor csak nehezen tudunk helyes modellt alkotni, mert sokan vitatják azt, hogy a problémában mi a lényeges és mi a lényegtelen. A kérdés tehát annak eldöntése, hogy a probléma valódi megoldásához ténylegesen milyen feladatot kell kitűzni. A matematikai modellben tehát a kitűzendő feladat körvonalai foglaltatnak.

A modell birtokában hozzáfoghatunk az algoritmus megszerkesztéséhez. Az algoritmus a megoldás egyértelmű szabályrendszere, amely véges sok utasítás meghatározott rendben történő előírása révén tetszőlegesen sok - egymástól csak adataikban különböző - feladat megoldására vezet. Amikor nyilvános helyről telefonálunk és a kifüggesztett "használati utasítás" szerint járunk el, akkor is algoritmust használunk: ahhoz, hogy kapcsolatot teremthessünk valamely másik állomással, meghatározott sorrendben végre kell majd hajtanunk az algoritmus utasításait - le kell vennünk a kagylót, meg kell várnunk a búgó hangot, be kell dobnunk az érmét stb. A különböző adatok itt a különböző hívószámok. Az algoritmus megfogalmazható szövegesen is, de könnyebben igazodunk el, ha rajzosan adjuk meg. Az algoritmus rajzos megfogalmazását blokk-diagramnak vagy folyamatábrának is szokás nevezni.

A számítógép eredményes használatának kulcskérdése, hogy megfelelő-e az algoritmus, amely alapján dolgozunk. Egy pontos, részletesen kidolgozott, szöveges vagy rajzos algoritmus birtokában a forrásprogram megírása szinte mechanikus tevékenység, ezért részletesen kell foglalkozunk az algoritmus készítésének alapelveivel.

Az előzőekben részben már hangsúlyoztuk, hogy a hétköznapi gondolkodás, a matematikai gondolkodás és a gép "gondolkodása" valamelyest eltér egymástól. Mivel algoritmusunkat adott esetben számítógép fogja végrehajtani, mindig ügyelnünk kell arra, hogy az egyes lépések mögött "érezzük az absztrakt gépet": tudnunk kell, mire képes és mire nem. (Rendszerint azt nem vesszük tudomásul, hogy a gép elemibb lépéseket végez, mint mi.) Érdemes tekintettel lenni arra is, hogy milyen megoldás mennyi memóriát köt le, és hány gépi utasítás végrehajtását igényli (vagyis mennyi futási időt jelent).

Tegyük fel, hogy van valamilyen X és Y változónk, és ezek tartalmát fel akarjuk cserélni. Egy matematikai levezetésben könnyű dolgunk van: az

$$\begin{matrix} X \rightarrow Y \\ \\ Y \rightarrow X \end{matrix} \qquad \text{vagy még inkább az} \qquad X \Leftrightarrow Y$$

jelölések bármelyike félreérthetetlenül kifejezi szándékunkat. Ám ha nem vagyunk elég körültekintőek, és a fenti hozzárendelést mechanikusan írjuk le egy algoritmusba, hibás eredményt kapunk:

$$X \leftarrow Y$$

$$Y \leftarrow X$$

Az első utasítással minden rendben van, hiszen azt jelenti: "vedd az Y változó jelenlegi (aktuális) tartalmát, és tedd az X-be". Ha ez az utasítás egy számítógép programjában szerepel, akkor X és Y egy-egy pontosan meghatározott (megcímzett) memóriarekeszt jelent. A számítógép lépésről lépésre halad, tehát az $X \leftarrow Y$ utasítás hatására az Y jelű rekesz tartalmát átmásolja X-be. Most rátér a következő utasításra: $Y \leftarrow X$. Mivel az X rekesz tartalma most már ugyanaz, mint az Y rekeszé, ezért a második utasítás végrehajtása után - a felcserélés helyett - mindkét rekeszben az eddigi Y értékét találjuk, és az X eredeti tartalma elveszett.

Az előbbiek (a helytelen algoritmus) belátását követési táblázat készítése segíti:

1. BE X, Y
2. KI X, Y ("eredeti")
3. X ← Y
4. Y ← X
5. KI X, Y ("átrendezett")

Legyen pl. a két érték 3 és 5 :

X	3	5	
Y	5	5	

Az eredmény tehát: eredeti: X = 3 Y = 5 átrendezett: X = 5 Y = 5

(helytelen!)

A fenti változócserét helyesen csak egy harmadik rekesz közbeiktatásával végezhetjük el, pl. így:

Z ← X
X ← Y
Y ← Z

Kezdők számára segítheti a probléma önálló megoldását, ha megkérdezzük, hogyan cserélnék ki például két (egy vörösborral és egy fehérborral töltött) hordó tartalmát. Az alapvető analógia akkor is segít, ha a hasonlat egyes részleteiben eltérések mutatkoznak. Sőt, ha rávilágítunk a megfelelő különbségekre, az eltérések maguk is segíthetik a jobb megértést.

Követési táblázat a helyes algoritmushoz:

1. BE X, Y
2. KI X, Y ("eredeti")
3. Z ← X
3. X ← Y
4. Y ← Z
5. KI X, Y ("átrendezett")

Az eredeti értékek legyenek ismét 3 és 5 :

X	~~3~~		5		
Y	~~5~~			3	
Z		3			

Az eredmény most: eredeti: X = 3 Y = 5 átrendezett: X = 5 Y = 3

(helyes)

4.2. A megoldási algoritmusok finomításának folyamata A "lépésről lépésre" módszer

A következő feladat megoldásával az algoritmus szerkesztés folyamatát szeretnénk érzékeltetni.

Legyen $x_1, x_2, x_3, ..., x_n$ mindegyike határozottan pozitív, vagyis $x_i > 0$ $\forall$i-re. Rendezzük ezeket a számokat nagyság szerint csökkenő sorozatba.

1. megoldás

Tegyük fel először, hogy n kicsi, legyen pl. n = 3 .

Hogyan rendeznénk mi három számot? Az igazat megvallva "ránézésre látjuk", melyik a legnagyobb, melyik a legkisebb: ha a három szám pl. 5, 18, 13, azonnal le tudjuk írni a helyes sorrendbe: 18, 13, 5. Nem is olyan könnyű megfogalmazni, mit csináltunk, hiszen "a megoldás olyan természetes".

Mindenesetre valószínűleg így gondolkodtunk:

1) egy darab papírra leírjuk a három számot
2) kikeressük a legnagyobbat
3) leírjuk a papír egy másik részére
4) valamilyen módon jelezzük, hogy ezzel a számmal már nem kell foglalkoznunk (pl. áthúzzuk)
5) a maradékból kikeressük a legnagyobb számot
6) leírjuk a papír más részére írt szám után
7) áthúzzuk
8) a megmaradt egyetlen számot leírjuk a papír másik részére, az előző kettő után.

Milyen számítógépes algoritmust sugall ez az eljárás? Mindenekelőtt kell lennie legalább 3 memóriarekesznek - pl. X1, X2, X3 -, amelyekbe beolvassuk a rendezni kívánt számokat. Ki kell jelölnünk másik három rekeszt ("a papír másik részét"), ahová immár helyes - nagyság szerint csökkenő - sorrendben átírjuk a számokat. Elvileg írhatnánk az X1, X2, X3 rekeszbe, ez azonban bonyodalmat okozna. Legyen tehát a rendezett számoknak kijelölt három rekesz Y1, Y2, Y3.

Hogyan keressük ki a három szám közül a legnagyobbat? A legkézenfekvőbb módszer a következő (a legnagyobb számot Y1-ben tároljuk majd!):

a) Y1-be beírjuk X1-et
b) összehasonlítjuk Y1-et X2-vel. Ha Y1<X2, akkor Y1-be beírjuk X2 tartalmát.
   (Ha Y1≥X2, akkor Y1-et nem változtatjuk)
c) összehasonlítjuk Y1-et X3-mal. Ha Y1<X3, akkor Y1-be beírjuk X3 tartalmát.
   (Ha Y1≥X3, akkor nem változtatjuk Y1-et).

Hogyan válasszuk ki a második legnagyobb számot? Közölnünk kellene a géppel, hogy "a legnagyobb számmal már ne foglalkozzon; az így megmaradtak közül keresse ki a legnagyobbat!" Melyik is a legnagyobb szám? Ebben a pillanatban nem tudjuk, csak azt, hogy mennyi! Ha azonban tudnánk, melyik, "áthúzhatnánk". Azért kellene áthúzni, hogy többé ne jelenhessen meg, mint "legnagyobb". Mivel számaink a feladat megfogalmazása szerint határozottan pozitívak, ezért a legnagyobb szám helyére nullát írva és ezután a számok között a legnagyobbat megkeresve a második legnagyobbat kapjuk. Ez azért lenne jó, mert így ismét az a), b), c) lépéseket végezhetnénk el, csak éppen Y1 helyébe Y2-t írva.

Melyik a legnagyobb szám? "Az, amelyiknek az értéke Y1. Ezt kell tehát kinullázni", feleljük, és máris írjuk a lépéseket:

d*) ha Y1 = X1, akkor X1 ← 0 (vagyis X1-et töröljük)
e*) ha Y1 = X2, akkor X2 ← 0
f*) ha Y1 = X3, akkor X3 ← 0

Ezzel algoritmusunkban elkövettük a létező legkellemetlenebb logikai hibát. Ha egy program soha nem működik jól, (vagyis az input adatokból nem a kívánt outputot állítja elő), az kellemetlen, de legalább azonnal észrevesszük a tesztelés (próbafuttatás, logikai vizsgálat) során, és több-kevesebb kínlódással megtalálhatjuk, kijavíthatjuk a hibát. A programozó igazi "rémálma" az a program, amelyik többnyire jól működik, néha azonban nem; pl. a tesztelés során húszszor jól lefut, ennek alapján hibátlannak nyilvánítjuk, majd az "éles" futásban hibás eredményt ad. Sajnos a fenti algoritmussal megírt programunk is ilyen lenne, ugyanis

ha az input	akkor az output
5, 18, 13	18, 13, 5 (jó),
142, 11, 53	142, 53, 11 (jó),
...	...

stb.

viszont ha az input	akkor az output
14, 2, 14	14, 2, 0 (rossz!)

Tehát ha több, egyenlő nagyságú szám van az input adatok között, akkor ezeket egy kivételével "elnyeli". Miért? Azért, mert amikor megtalálta a legnagyobb számot, a d*), e*), f*) utasításokban minden olyan tárolót kinullázott, amelyekben ez a szám szerepel, holott a feladat megfogalmazása nem zárja ki annak a lehetőségét, hogy akár az összes szám egyforma legyen!

Gondoskodnunk kell tehát arról, hogy a gép a d), e), f) lépésekben mindig csak egy tárolót nullázzon ki (tulajdonképpen a "legnagyobb szám" meghatározását kell módosítanunk: "a legnagyobb szám az első olyan, amelynek értéke Y1"). Az algoritmus folytatása tehát:

d) ha Y1 = X1, akkor X1 ←0, és ne hajtsd végre sem e)-t, sem f)-et!
e) ha Y1 = X2, akkor X2 ←0, és ne hajtsd végre f)-et!
f) ha Y1 = X3, akkor X3 ←0

Végre eljutottunk a második legnagyobb szám meghatározásához. Most már szinte mechanikusan írhatjuk:

g) Y2 ← X1
h) ha Y2 < X2, akkor Y2 ← X2 (ha nem, nincs teendőnk)
i) ha Y2 < X3, akkor Y2 ← X3 (ha nem, nincs teendőnk)
j) ha Y2 = X1, akkor X1 ← 0, és ne hajtsd végre sem k)-t, sem l)-t
k) ha Y2 = X2, akkor X2 ← 0, és ne hajtsd végre l)-t
l) ha Y2 = X3, akkor X3 ← 0.

A rendezés gyakorlatilag befejeződött, hiszen a két szám már a "helyén" van Y1, Y2-ben, és csak egyetlen X nem zérus.
Nyugodtan írhatjuk tehát végeljárásként:

m) ha X1 <> 0, akkor Y3 ← X1
n) ha X2 <> 0, akkor Y3 ← X2
o) ha X3 <> 0, akkor Y3 ← X3.

Nézzük meg pontosan, hogyan is alakulnak a megfelelő követési táblázatok!

Először a helytelen d*) e*) f*) ... és persze az ennek megfelelő ... j*) k*) l*) lépések esetén.

1*) Legyen X1, X2, X3 rendre 5, 18, 13 .

X1	5	
X2	~~18~~	~~0~~
X3	~~13~~	~~0~~
Y1	~~5~~	18
Y2	~~5~~	13
Y3	5	

Az eredmény Y1, Y2, Y3 - ra 18, 13, 5 (helyes).

2*) Legyen X1, X2, X3 rendre 142, 11, 53.

X1	~~142~~				~~0~~											
X2	11															
X3	~~53~~												~~0~~			
Y1		(142)														
Y2								~~0~~	~~11~~	(53)						
Y3															(11)	
		a)	b)	c)	d)*	e)*	f)*	g)	h)	i)	j)*	k)*	l)*	m)	n)	o)

Az eredmény Y1, Y2, Y3 - ra 142, 53, 11 (helyes).

3*) Legyen X1, X2, X3 rendre 14, 2, 14.

X1	14				~~0~~											
X2	2										~~0~~					
X3	14						~~0~~									
Y1		14														
Y2								~~0~~	2							
Y3																
		a)	b)	c)	d)*	e)*	f)*	g)	h)	i)	j)*	k)*	l)*	m)	n)	o)

" lenyelte " !

Az eredmény Y1, Y2, Y3 - ra 14, 2, 0 (rossz!).

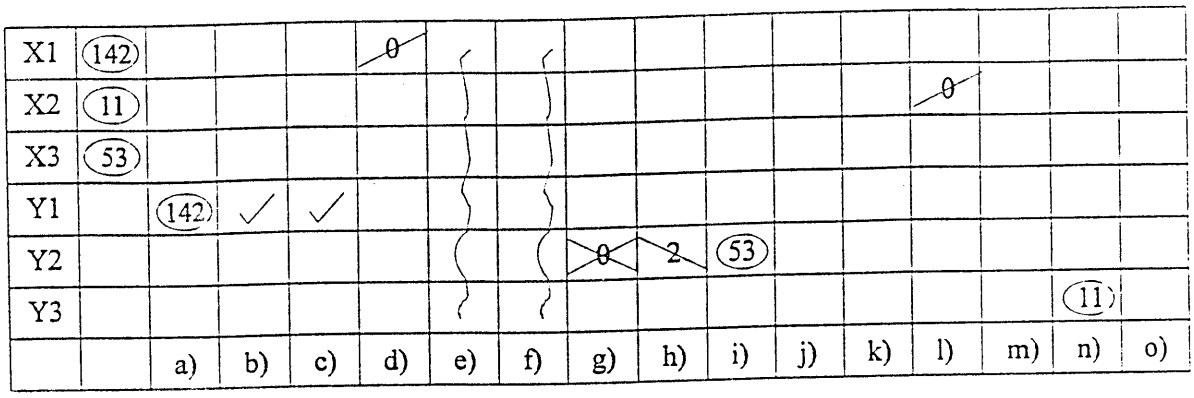
A helyes d) e) f) ... és persze az ennek megfelelő ... j) k) l) lépések esetén a követési táblázat az alábbiak szerint alakul.

1) Legyen X1, X2, X3 rendre 5, 18, 13 .

X1	5															
X2	~~18~~					~~0~~										
X3	~~13~~												~~0~~			
Y1		~~5~~	(18)	✓												
Y2								~~5~~	✓	(13)	✓	✓				
Y3														(5)		
		a)	b)	c)	d)	e)	f)	g)	h)	i)	j)	k)	l)	m)	n)	o)

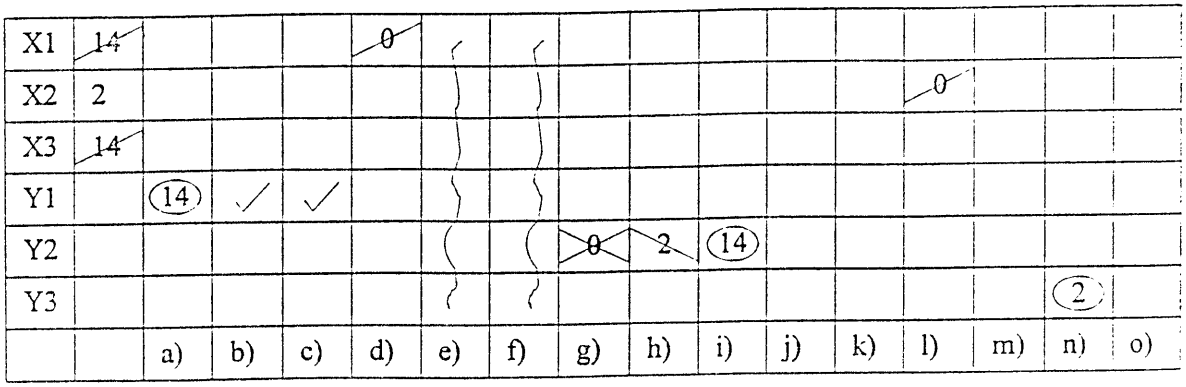
Az eredmény Y1, Y2, Y3 - ra 18, 13, 5 (helyes).

2) Legyen X1, X2, X3 rendre 142, 11, 53.



Az eredmény Y1, Y2, Y3 - ra 142, 53, 11 (helyes).

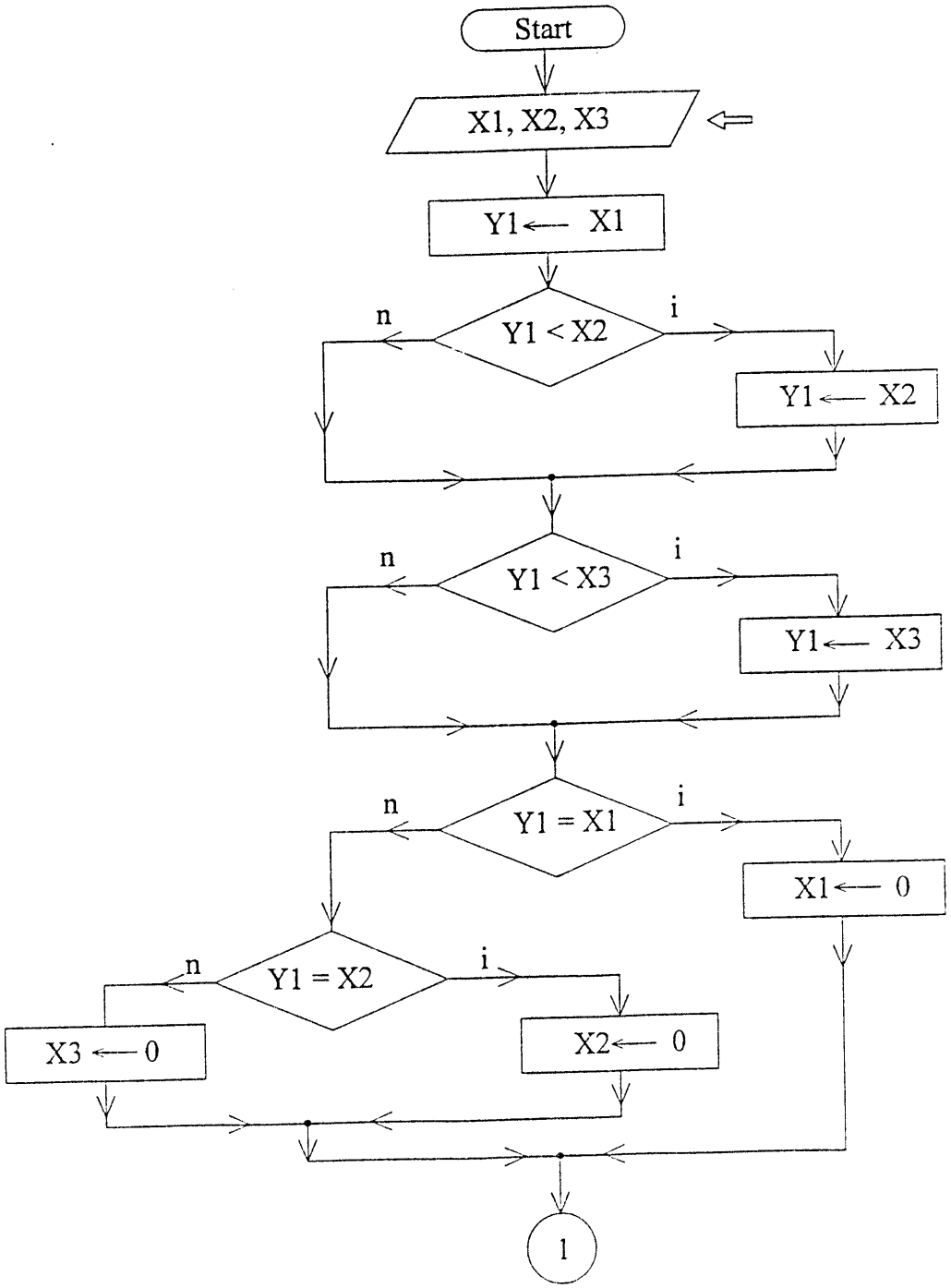
3) Legyen X1, X2, X3 rendre 14, 2, 14.
(KRITIKUS ESET)

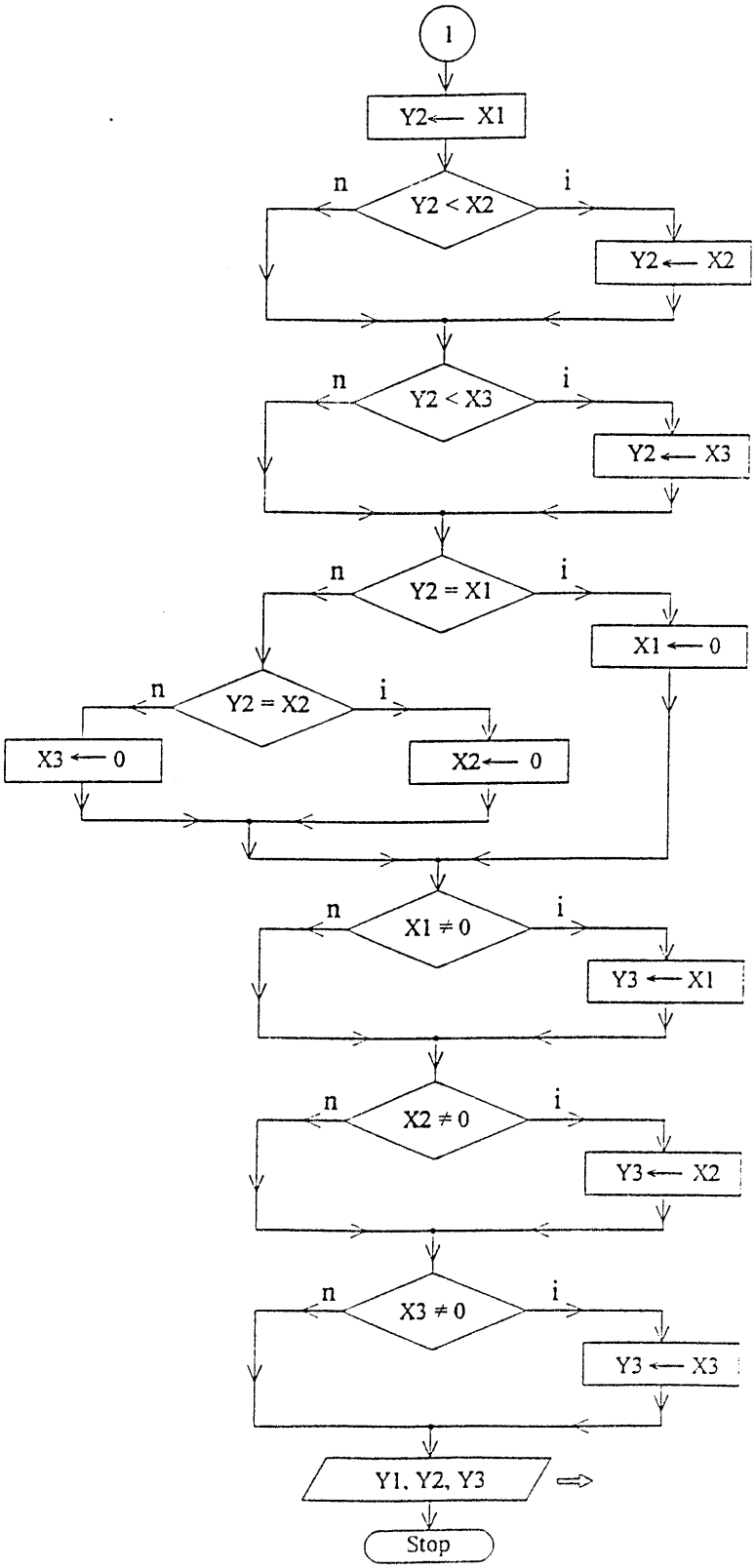


Az eredmény Y1, Y2, Y3 - ra 14, 14, 2 (kritikus esetben is jó).

Aki valóban érti, hogyan működik ez az algoritmus, bizonyára észreveszi, hogy az f) és az l) sorban a feltételvizsgálat felesleges, nyugodtan írhattuk volna így is: X3 ← 0 . Miért?

Szöveges algoritmusunknak az említett felesleges döntések nélküli változatát folyamatábrán is szemléltetjük:





Folyamatábra (A rendezési feladat 1.megoldása)

Fogalmazzuk meg első megoldásként kapott algoritmusunkat pszeudokódban is! Mielőtt azonban ezt megtennénk, gondoljuk át, hogy a pszeudokódokra vonatkozó megállapodásunk értelmében nem használhatjuk a strukturáltság ellen ható (a BASIC-ből talán ismerős) GOTO "ugró" utasítást. A d), e), f) és az ezzel analóg j), k), l) lépéseknél merülhetne föl ez a kérdés; a probléma azonban megoldható ugró utasítás használata nélkül is, ha az egyes logikai döntések "nem" ágát is kihasználva, egymásba ágyazott elágazásokkal dolgozunk.

```
algoritmus_a rendezési_probléma_első_megoldása
1.  BE X1, X2, X3
2.  Y1 ← X1
3.  HA   Y1 < X2        AKKOR
      3.1. Y1 ← X2
    HAVÉGE
4.  HA   Y1 < X3        AKKOR
      4.1. Y1 ← X3
    HAVÉGE
5.  HA    Y1 = X1       AKKOR
            5.1. X1 ← 0
                          KÜLÖNBEN
            5.2. HA     Y1 = X2      AKKOR
                          5.2.1.  X2 ← 0
                                      KÜLÖNBEN
                          5.2.2.  X3 ← 0
                 HAVÉGE
    HAVÉGE
6.  Y2 ← X1
7.  HA   Y2 < X2        AKKOR
            7.1.   Y2 ← X2
    HAVÉGE
8.  HA   Y2 < X3        AKKOR
            8.1.   Y2 ← X3
    HAVÉGE
```

```
9.  HA    Y2 = X1      AKKOR
              9.1. X1 ← 0
                              KÜLÖNBEN
              9.2. HA    Y2 = X2      AKKOR
                              9.2.1.  X2 ← 0
                                                KÜLÖNBEN
                              9.2.2.  X3 ← 0
                    HAVÉGE
        HAVÉGE
10. HA   X1 <> 0      AKKOR
              10.1.  Y3 ← X1
        HAVÉGE
11. HA   X2 <> 0      AKKOR
              11.1.  Y3 ← X2
        HAVÉGE
12. HA   X3 <> 0      AKKOR
              12.1.  Y3 ← X3
        HAVÉGE
13. KI  Y1, Y2, Y3
algoritmus_vége (a rendezési probléma 1. megoldása)
```

2. megoldás

Első megoldásunk - bár hibátlan - egyáltalán nem "szép". Három szám rendezése kedvéért viszonylag hosszú programot kellett írnunk: a ténylegesen leírt (lerajzolt) döntési utasítások száma

3 szám esetén $(3-1 + 3-1)*(3-1) + 3 = 11$,

(miért éppen ennyi?)

4 szám esetén $(4-1 + 4-1)*(4-1) + 4 = 22$,

5 szám esetén $(5-1 + 5-1)*(5-1) + 5 = 37$

... stb.

n szám esetén $2*(n-1)*(n-1) + n$ döntési utasítást kellene leírnunk.

A program nem "általánosítható" n szám rendezésének esetére (n tetszőleges). Az is bosszantó, hogy tulajdonképpen majdnem pontosan ugyanazokat az utasításokat ismételgetjük, csak egyszer Y1-gyel, egyszer Y2-vel stb. Az igazi elvi probléma az, hogy annyi utasítást kellett leírnunk, ahány lépés a feladat megoldásához szükséges.

Szinte kivétel nélkül mindig érvényes a következő szabály: az olyan algoritmus, amelyben minden utasítást legfeljebb egyszer hajtunk végre, nem való számítógépre! Azért nem, mert a programozás több időt igényel, mint amennyi alatt egy közönséges kalkulátorral megoldjuk a feladatot.

Minden magas szintű programnyelv kínál olyan lehetőséget, hogy a hasonló jellegű változókat ún. tömbökbe foglaljuk össze, közös névvel és egy indexszel hivatkozzunk rájuk, és ez az index maga is lehet változó. A jelölésmód is hasonlít a matematikából ismert $y_1, y_2, y_3, ..., y_n$ jelöléshez.

Amikor a matematikában az $y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7$ összeget

$x = \sum_{i=1}^{7} y_i$ - vel jelöljük, mindegyik y_i külön szám, egymáshoz semmi közük;

(igaz, közös tulajdonsággal rendelkeznek, t.i., hogy mindegyikük összeadandó).

A $\sum_{i=1}^{7} y_i$ összegképzés ciklikus műveletvégzést jelöl ki, a következő

algoritmussal:

1) legyen az i index értéke 1
2) legyen x értéke 0
3) legyen x értéke $x + y_i$
4) növeld meg i értékét 1-gyel
5) ha $i \leq 7$, térj vissza a 3. lépéshez
6) vége.

Példánkban az i index változó. A tömbök használatának és az indexek változóként való megadásának a számítástechnikában (is) az az előnye, hogy a program ciklikussá tehető: a gép többször is végrehajtja ugyanazokat az utasításokat (amelyek fizikailag is azonosak, a memóriában csak egyszer szerepelnek), csupán az index értékét változtatja. Ha tömbökkel és változóként megadott indexekkel dolgozunk, a programot könnyű "tágítani": minimális átalakítással ugyanaz a program 7 helyett száz, ezer stb. számot is összeadhat.

Szövegesen megadott összegző algoritmusunkat természetesen másként is megfogalmazhattuk volna, pl. így:

1) legyen az i index értéke 1
2) legyen x értéke 0
3) legyen x értéke $x + y_i$
4) növeld meg i értékét 1-gyel
5) ismételd meg a 3) és 4) utasításokat mindaddig, amíg i nem nagyobb 7-nél
6) vége.

Egy másik - az előzővel egyenértékű - lehetséges változat:

1) legyen x értéke 0

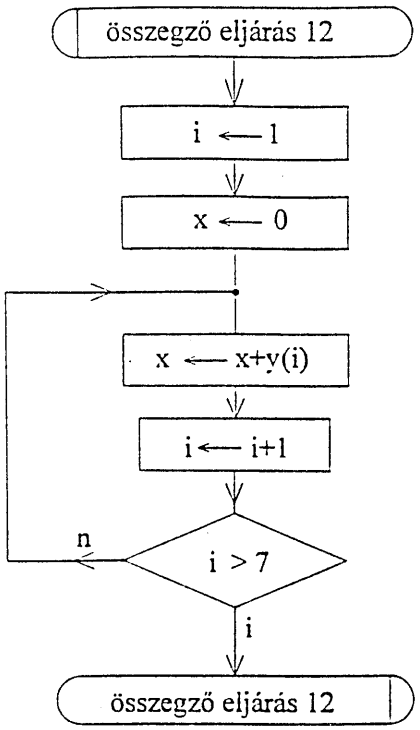
2) vedd az i értékeit 1-től 7-ig (egyesével) és az i minden értékénél hajtsd végre a következő utasítást:

legyen x értéke $x + y_i$

3) vége.

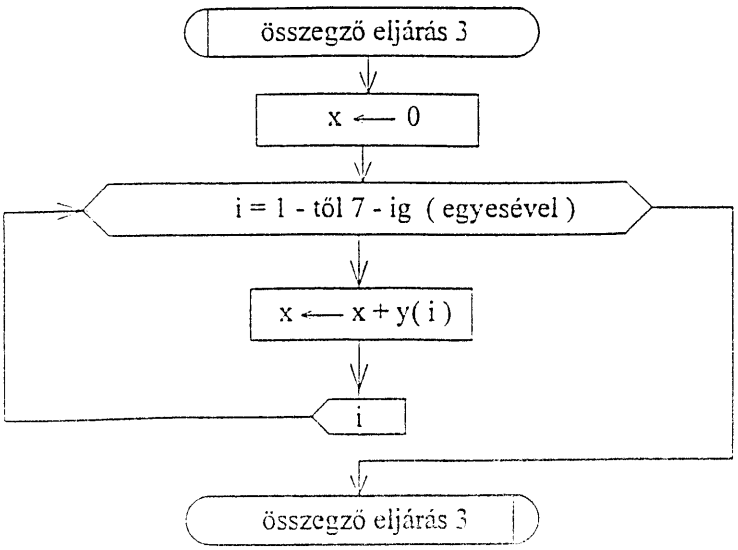
A szövegesen megadott algoritmusok különböző változataihoz nyilván különböző pszeudokódos programokat, ill. folyamatábrákat adhatunk meg. A három változatot összehasonlítva látható, hogy az első kettő lényegében azonos tömörségű (a második csupán annyiból szerencsésebb, hogy nem szerepel benne a nem kívánatos ugró utasítás), a harmadik változat azonban lényegesen tömörebb.

Az első és második változatot közös folyamatábrával szemléltethetjük; a megfelelő pszeudokódokat mindjárt az egyes folyamatábra szimbólumok mellé írjuk:



összegző_eljárás_1_2
1. i ← 1
2. x ← 0
3. CIKLUS
    3.1. x ← x + y(i)
    3.2. i ← i + 1
  AMEDDIG i > 7 NEM TELJESÜL
eljárás_vége.

A végrehajtás során a gép ciklikusan ismétli a 3.1. és 3.2. utasításokat (a ciklusmagot), mindaddig, amíg az indexváltozó (ciklusváltozó) meg nem halad egy előre meghatározott értéket (esetünkben a hetet). Példánkban magunk kezeljük az indexváltozót (mi növeljük és - ebben a megoldásban a ciklus végén - mi vizsgáljuk, elérte-e a felső határt), erre azonban itt nincs feltétlenül szükség. Minden magas szintű programnyelv biztosítja az ún. számlálással működő ciklusképzés lehetőségét, ez felel meg a harmadik változatnak:



összegző_eljárás_3
1. x ← 0
2. CIKLUS i = 1-től 7-ig (egyesével)
    2.1. x ← x + y(i)
  CIKLUSVÉG
eljárás_vége.

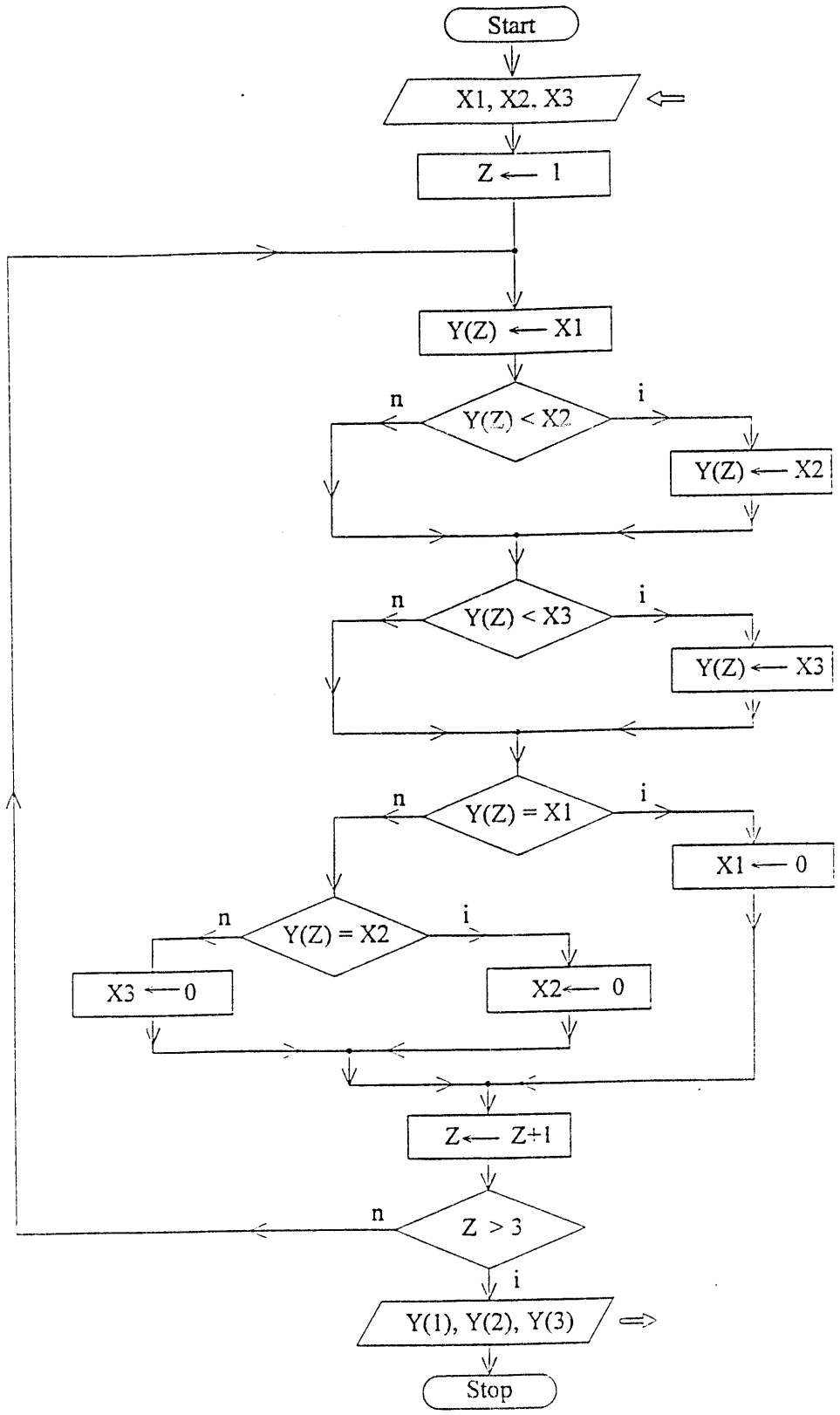
Amikor a gép először érkezik a 2. utasításhoz ("változtasd i értékét 1-től 7-ig"), beállítja i értékét 1-re, majd végrehajtja a "ciklusvég"-ig található utasítás(oka)t. Most eggyel megnöveli i értékét, majd megvizsgálja, nem haladtuk-e meg a "ciklusfej"-ben megadott felső határt ("7-ig"). Ha nem, akkor visszatér a 2.1. utasításra és ismét végrehajtja a ciklus magját (természetesen i = 2 - vel). Ha a ciklusvég elérésekor azt érzékeli, hogy i, a ciklusváltozó további növelés esetén meghaladná az előírt határt, akkor nem tér vissza, hanem áttér a ciklusvég utáni utasításra.

Térjünk vissza a rendezési feladathoz! Tekintsük az Y1, Y2, Y3 együttesét egyetlen tömbnek, melynek elemeit a magas szintű nyelvek helyesírási szabályainak megfelelően Y(1), Y(2), Y(3) jelöli.

Ismét hangsúlyozzuk: a nyereség az, hogy a tömb elemeire Y(Z) formában is hivatkozhatunk; hogy ez melyik Y-t jelenti, azon múlik, hogy az utasítás végrehajtásának pillanatában mennyi Z ("aktuális") értéke. Ennek kihasználásával a három szám rendezésére szolgáló szöveges algoritmusunk most a következő (X1, X2, X3 benn van a memóriában):

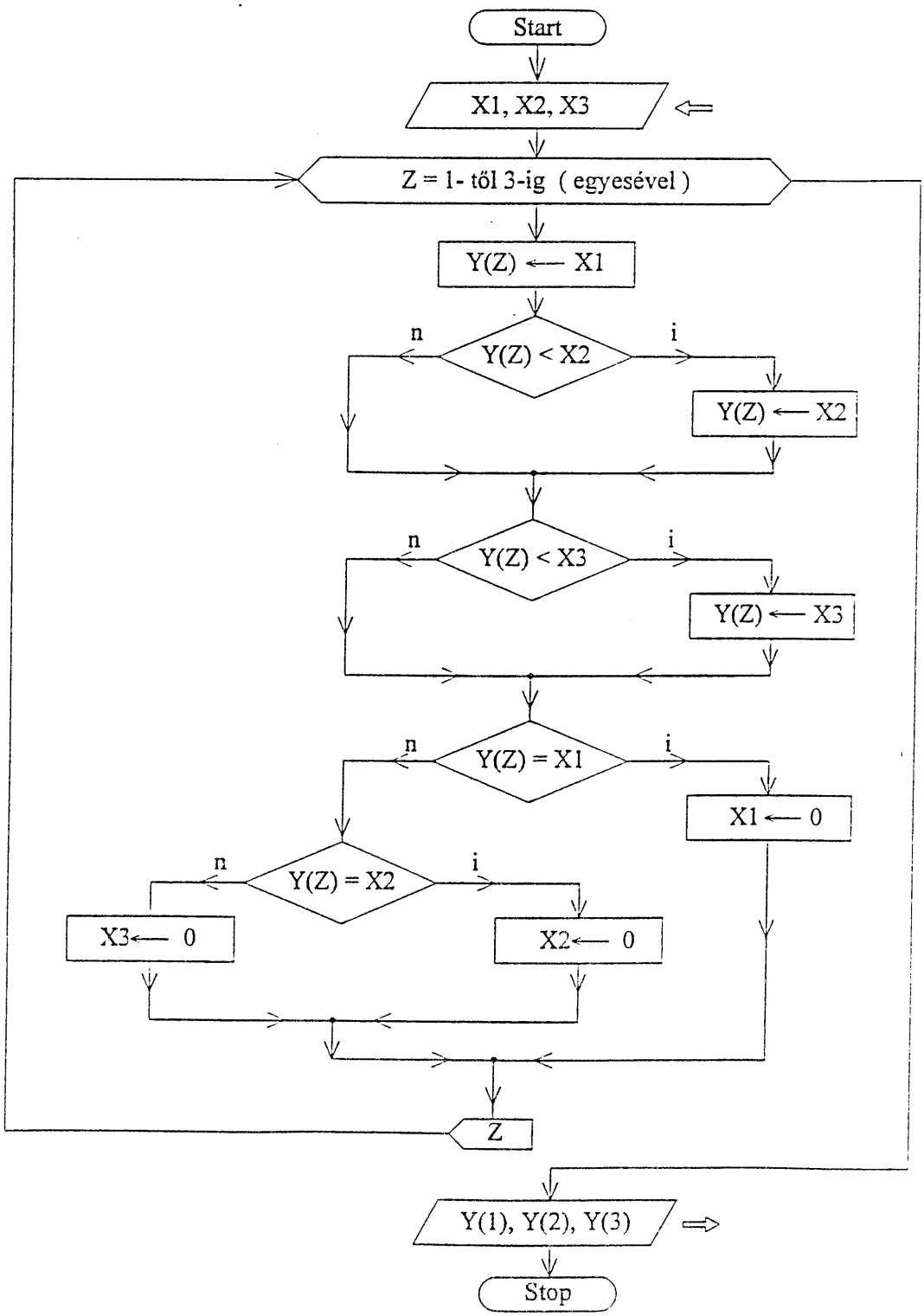
a) $Z \leftarrow 1$
b) $Y(Z) \leftarrow X1$
c) ha $Y(Z) < X2$, akkor $Y(Z) \leftarrow X2$
d) ha $Y(Z) < X3$, akkor $Y(Z) \leftarrow X3$
e) ha $Y(Z) = X1$, akkor $X1 \leftarrow 0$, és ne hajtsd végre sem f)-et, sem g)-t
f) ha $Y(Z) = X2$, akkor $X2 \leftarrow 0$, és ne hajtsd végre g)-t
g) $X3 \leftarrow 0$
h) Z értékét növeld meg eggyel
i) ha $Z \leq 3$, térj vissza b)-hez
j) vége.

A pszeudokódban írt programot rögtön a folyamatábra után írjuk. A ciklusszervezést illetően - a segédpéldaként tárgyalt összegző algoritmushoz hasonlóan - itt is két változatot közlünk.



Folyamatábra (A rendezési probléma 2.megoldása 1_2)

```
algoritmus_rendezési_probléma_1_2
1. BE X1, X2, X3
2. Z ← 1
3. CIKLUS
      3.1. Y(Z) ← X1
      3.2. HA  Y(Z) < X2  AKKOR
                        3.2.1. Y(Z) ← X2
           HAVÉGE
      3.3  HA  Y(Z) < X3  AKKOR
                        3.3.1. Y(Z) ← X3
           HAVÉGE
      3.4. HA  Y(Z) = X1  AKKOR
                        3.4.1. X1 ← 0
                             KÜLÖNBEN
                        3.4.2. HA Y(Z) = X2 AKKOR
                                     3.4.2.1. X2 ← 0
                                                KÜLÖNBEN
                                     3.4.2.2. X3 ← 0
                                HAVÉGE
           HAVÉGE
      3.5. Z ← Z + 1
   AMEDDIG Z > 3 NEM TELJESÜL
4. KI Y(1), Y(2), Y(3)
eljárás_vége.
```

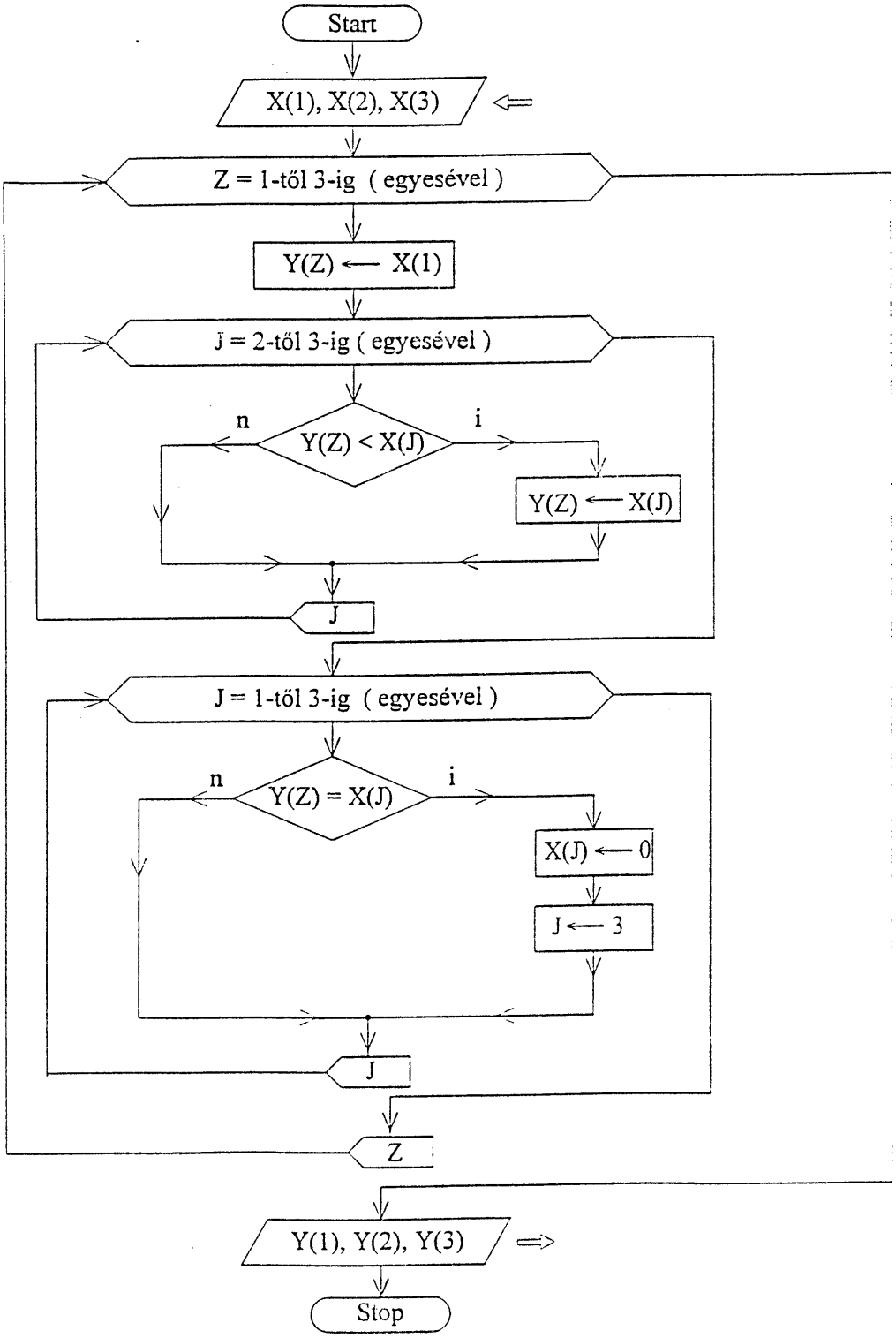


Folyamatábra (A rendezési probléma 2.megoldása_3)

```
algoritmus_rendezési_probléma_3
1. BE X1, X2, X3
2. CIKLUS Z = 1-től 3-ig (egyesével)
      2.1. Y(Z) ← X1
      2.2. HA Y(Z) < X2 AKKOR
                        2.2.1. Y(Z) ← X2
           HAVÉGE
      2.3  HA Y(Z) < X3 AKKOR
                        2.3.1. Y(Z) ← X3
           HAVÉGE
      2.4. HA Y(Z) = X1 AKKOR
                        2.4.1. X1 ← 0
                               KÜLÖNBEN
                        2.4.2. HA Y(Z) = X2 AKKOR
                                          2.4.2.1. X2 ← 0
                                                KÜLÖNBEN
                                          2.4.2.2. X3 ← 0
                                  HAVÉGE
           HAVÉGE
    CIKLUSVÉG
3. KI Y(1), Y(2), Y(3)
eljárás_vége.
```

Ez a program lényegesen rövidebb, mint amit az 1. megoldásban kaptunk. Ha "ki akarjuk terjeszteni", vagyis háromnál több számot akarunk rendezni, minden újabb szám (néhány apró módosításon kívül) csupán 2 újabb döntés leírását igényli.

Megoldásunk jóllehet egyszerűbb, mint az első, de valami még mindig zavaró benne. Miért ne lehetne X1, X2, X3 is egy tömb három eleme? Elvi akadály nincs, de azért óvatosan kell eljárnunk. Nagy hiba lenne, ha egyszerűen minden X1, X2, X3 helyére X(Z)-t írnánk. Z valóban 1-től 3-ig változik, nekünk azonban Z minden értékére külön-külön meg kell vizsgálni mind a három lehetséges X-et. Szükség lenne tehát két újabb ciklusra is (új ciklusváltozóval), melyek az első cikluson belül működnek, azaz rögzített Z-re Y(Z)-t minden X-szel összehasonlítják.



Folyamatábra (A rendezési probléma 2.megoldása_4)

A folyamatábra alapján a (pszeudo)kódolás szinte mechanikus tevékenység:

```
algoritmus_rendezési_probléma_24
1. BE X(1), X(2), X(3)
2. CIKLUS  Z = 1-től 3-ig (egyesével)
     2.1. Y(Z) ← X(1)
     2.2. CIKLUS  J = 2-től 3-ig (egyesével)
                2.2.1. HA  Y(Z) < X(J)  AKKOR
                             2.2.1.1.  Y(Z) ← X(J)
                           HAVÉGE
          CIKLUSVÉG
     2.3. CIKLUS  J = 1-től 3-ig (egyesével)
                2.3.1. HA  Y(Z) = X(J)  AKKOR
                             2.3.1.1.  X(J) ← 0
                             2.3.1.2.  J ← 3
                           HAVÉGE
          CIKLUSVÉG
   CIKLUSVÉG
3. KI  Y(1), Y(2), Y(3)
algoritmus_vége.
```

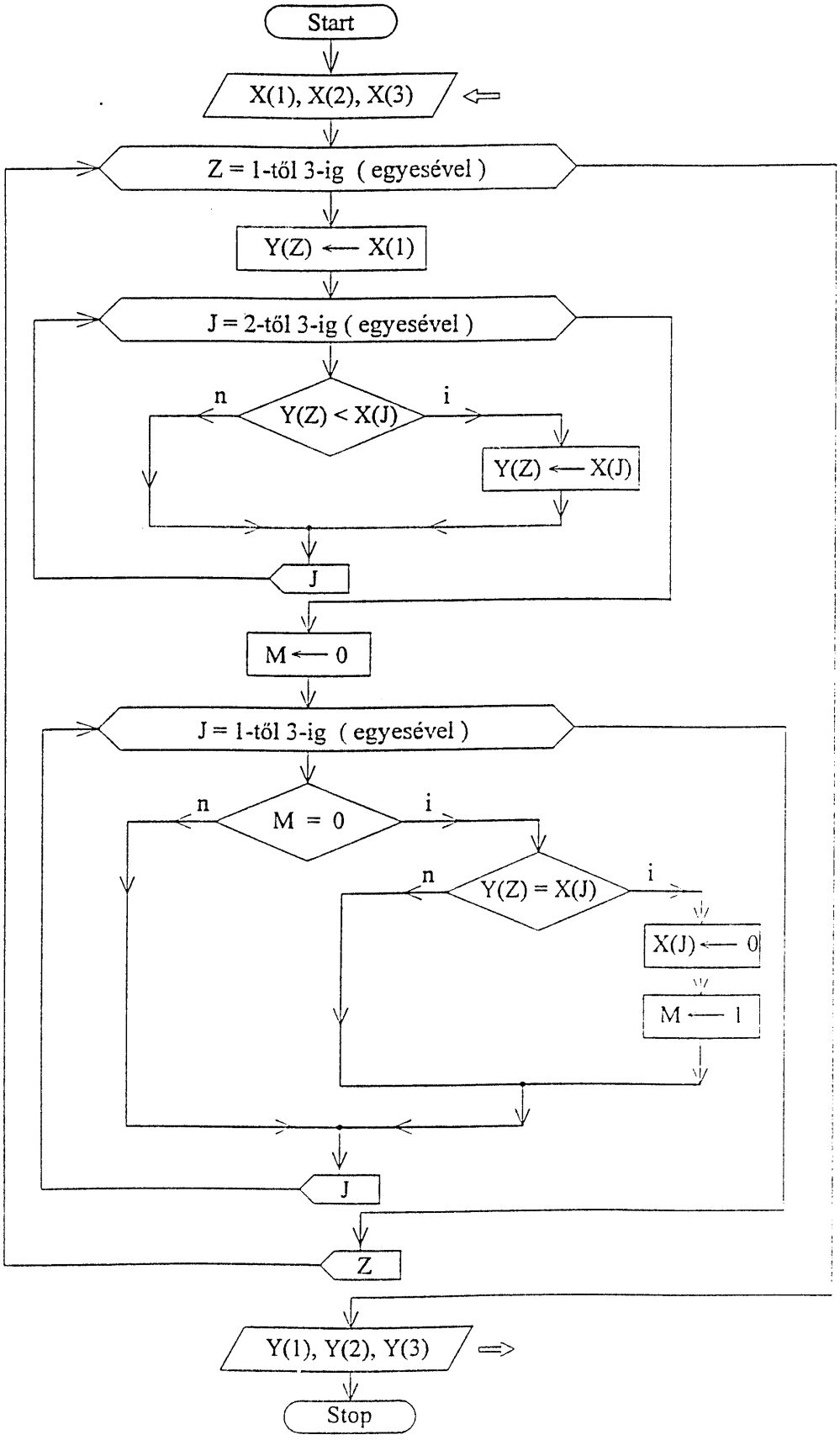
A program egymásba ágyazott ciklusokat tartalmaz. Az egyes ciklusokat a megfelelő mélységű - párba állított (CIKLUS - CIKLUSVÉG) - bekezdések jól olvashatóvá, áttekinthetővé teszik. Szigorú szabály minden magas szintű nyelvben, hogy két ciklus vagy teljesen diszjunkt legyen, vagy az egyik teljes egészében legyen benne a másikban (próbáljuk megindokolni, miért kell így lennie!). A 2.2. ciklus teljesen diszjunkt a 2.3. ciklustól, vagyis nincs közös utasításuk, viszont mindkettő valódi része a 2. ciklusnak, vagyis minden utasításuk a 2. cikluson belül van (és a 2. ciklusnak van még egyéb utasítása is). A diszjunkt ciklusoknak lehet azonos ciklusváltozójuk (J), arra azonban ügyelnünk kell, hogy az egymásba ágyazott ciklusok ciklusváltozója ne legyen azonos (Z és J). (Miért?)

A 2.3.1.2. utasítás "nem szép": a programozási tankönyvek nem javallják, sőt, olykor kifejezetten tiltják. Okkal: nehezen felderíthető hibák forrása lehet, ha egy cikluson belül mi magunk módosítjuk a ciklusváltozó értékét (vagyis a ciklusváltozó egy értékadó utasítás - esetünkben a J ← 3 - bal oldalán szerepel). A ciklusváltozót azért módosítottuk, mert a 2.3.1. logikai döntés Y(Z) = X(J) feltétele többször is teljesülhet, az X(J) ←0 értékadó utasítást

azonban csak egyszer szabad végrehajtani. Ha tehát az X(J) ← 0 első végrehajtásakor J értékét 3-ra állítjuk, a ciklusmag esetleges újbóli végrehajtása előtt aktuális feltételvizsgálat során a gép "azt hiszi", hogy a ciklust már háromszor végrehajtotta és ezért kilép a ciklusból. Jobb lenne azonban, ha valamilyen más módon adnánk a gép tudtára, hogy az X(J) ← 0 utasítást egyszer már végrehajtotta. Egy újabb változót, pl. M-et használhatjuk erre a célra.

M-nek azt kell jeleznie, hogy adott Z mellett az X(J) ← 0 nullázást a gép egyszer már végrehajtotta. (Ilyenkor természetesen már a 2.3.1.-beli Y(Z) = X(J) feltételvizsgálat is felesleges, tehát átugorhatjuk.) M - szerepét tekintve - speciális változó: ún. flag (zászló) azt mutatja, hogy valami megtörtént-e már, vagy sem. Természetes választás, hogy legyen M értéke 0, ha adott Z mellett az Y(Z) = X(J) feltétel még egyszer sem teljesült, és M értéke 1, ha a feltétel egyszer már teljesült. Programunk tehát így módosul:

```
algoritmus_rendezési_probléma_25
1. BE X(1), X(2), X(3)
2. CIKLUS  Z = 1-től 3-ig (egyesével)
      2.1.  Y(Z) ← X(1)
      2.2.  CIKLUS  J = 2-től 3-ig (egyesével)
                  2.2.1.  HA  Y(Z) < X(J)  AKKOR
                                    2.2.1.1.  Y(Z) ← X(J)
                              HAVÉGE
            CIKLUSVÉG
      2.3.  M ← 0
      2.4.  CIKLUS  J = 1-től 3-ig (egyesével)
                  2.4.1.  HA  M = 0  AKKOR
                                    2.4.1.1.  HA  Y(Z) = X(J)  AKKOR
                                                      2.4.1.1.1.  X(J) ← 0
                                                      2.4.1.1.2.  M ← 1
                                                HAVÉGE
                              HAVÉGE
            CIKLUSVÉG
   CIKLUSVÉG
3. KI  Y(1), Y(2), Y(3)
algoritmus_vége.
```



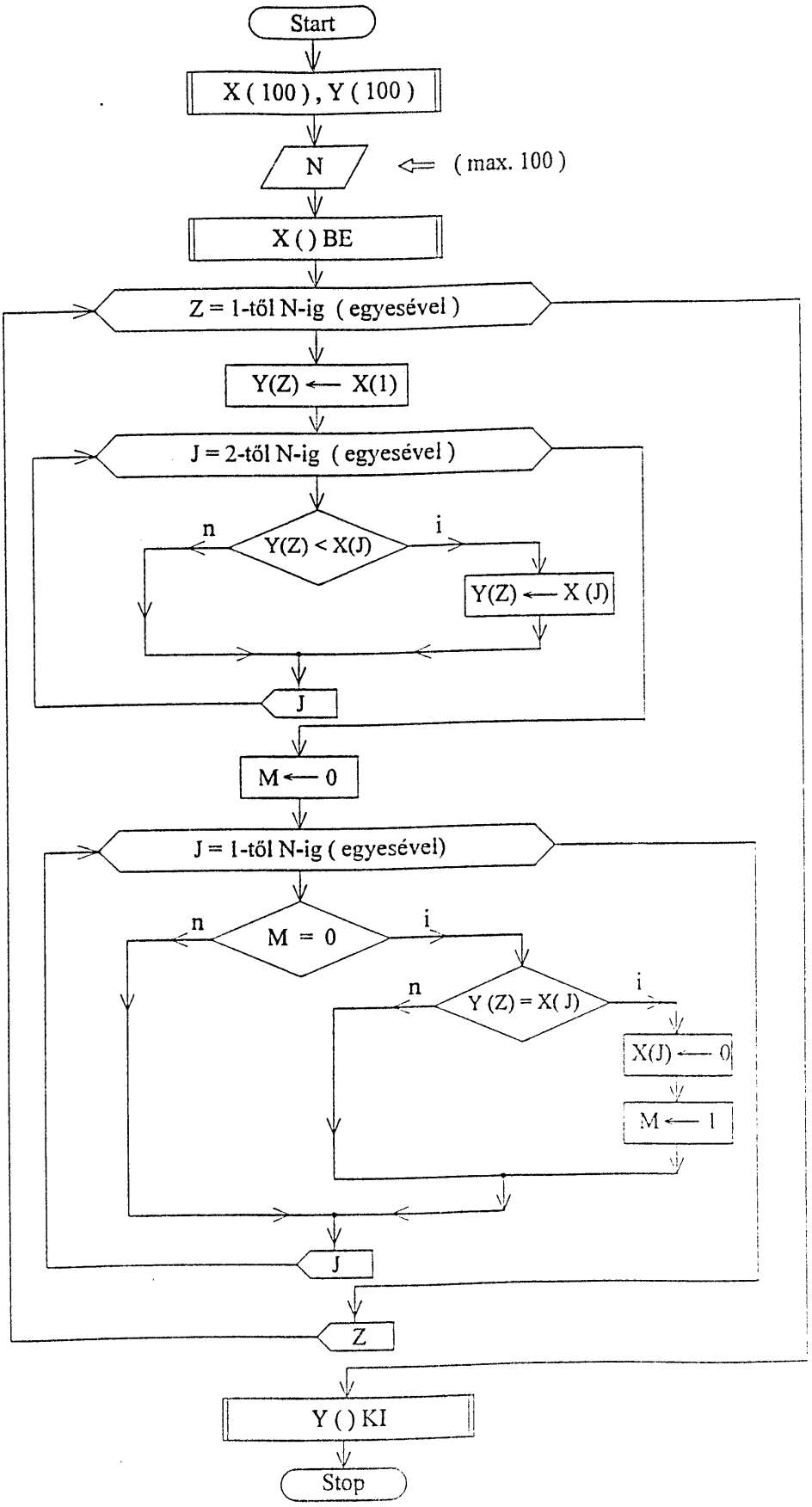
Folyamatábra (A rendezési probléma 2. megoldása_5)

A 2.3. miatt M minden újabb Z-nél kinullázódik, tehát az X(J) ← 0 utasítást a gép minden Z-re pontosan egyszer hajtja végre. Vegyük észre, hogy programunk már szinte teljesen általános: könnyen alkalmassá tehető N szám rendezésére, ahol N futásról futásra változhat - feltéve, hogy nem halad meg egy előre megadott felső határt.

Mindössze arra van szükség, hogy a programban

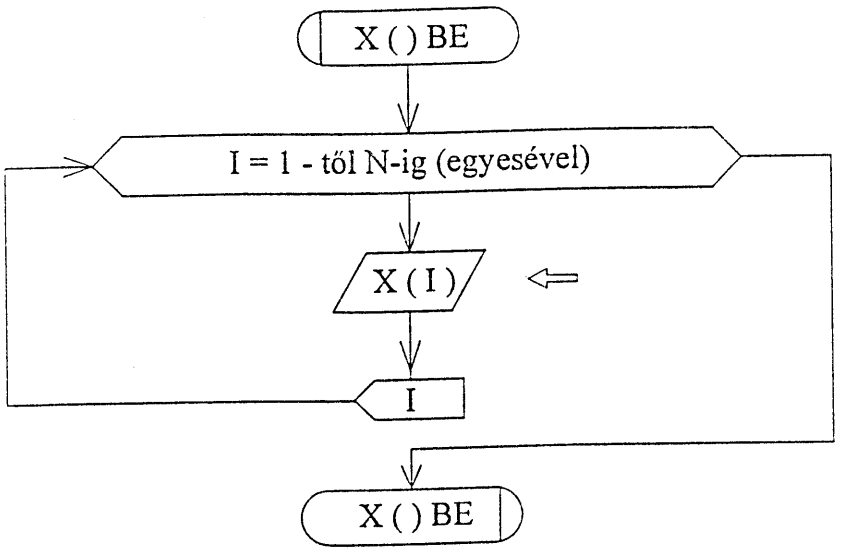
a) első lépésként kérdezzük meg, hogy az adott futásban hány számot kell rendezni
b) jelöljünk ki az N megengedett legnagyobb értékének megfelelő tárterületet az X( ) és Y( ) tömböknek
c) olvassunk be N számot az X( ) tömbbe
d) a program további része változatlan, csak arra kell ügyelnünk, hogy a ciklusokban a ciklusváltozók felső határa 3 helyett az aktuális N.

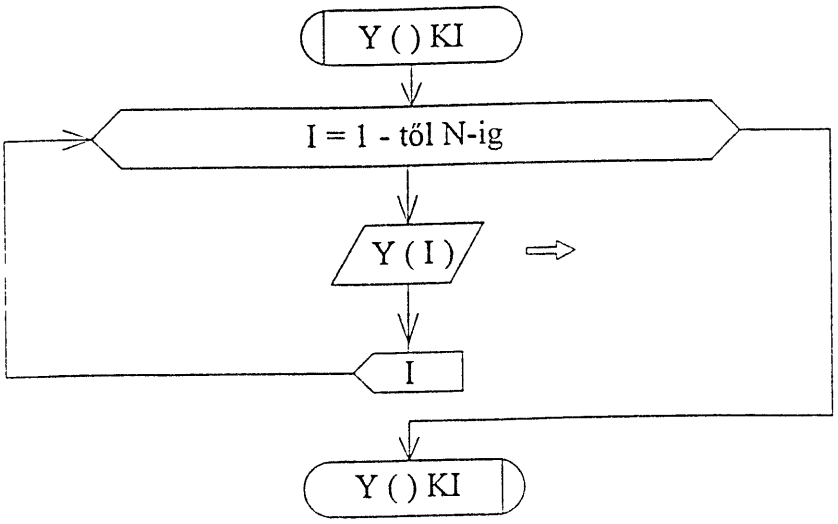
Készítsük el a feladat 2. megoldását jelentő algoritmust (legfeljebb 100 szám rendezésére) mindhárom algoritmus-leíró eszközzel!

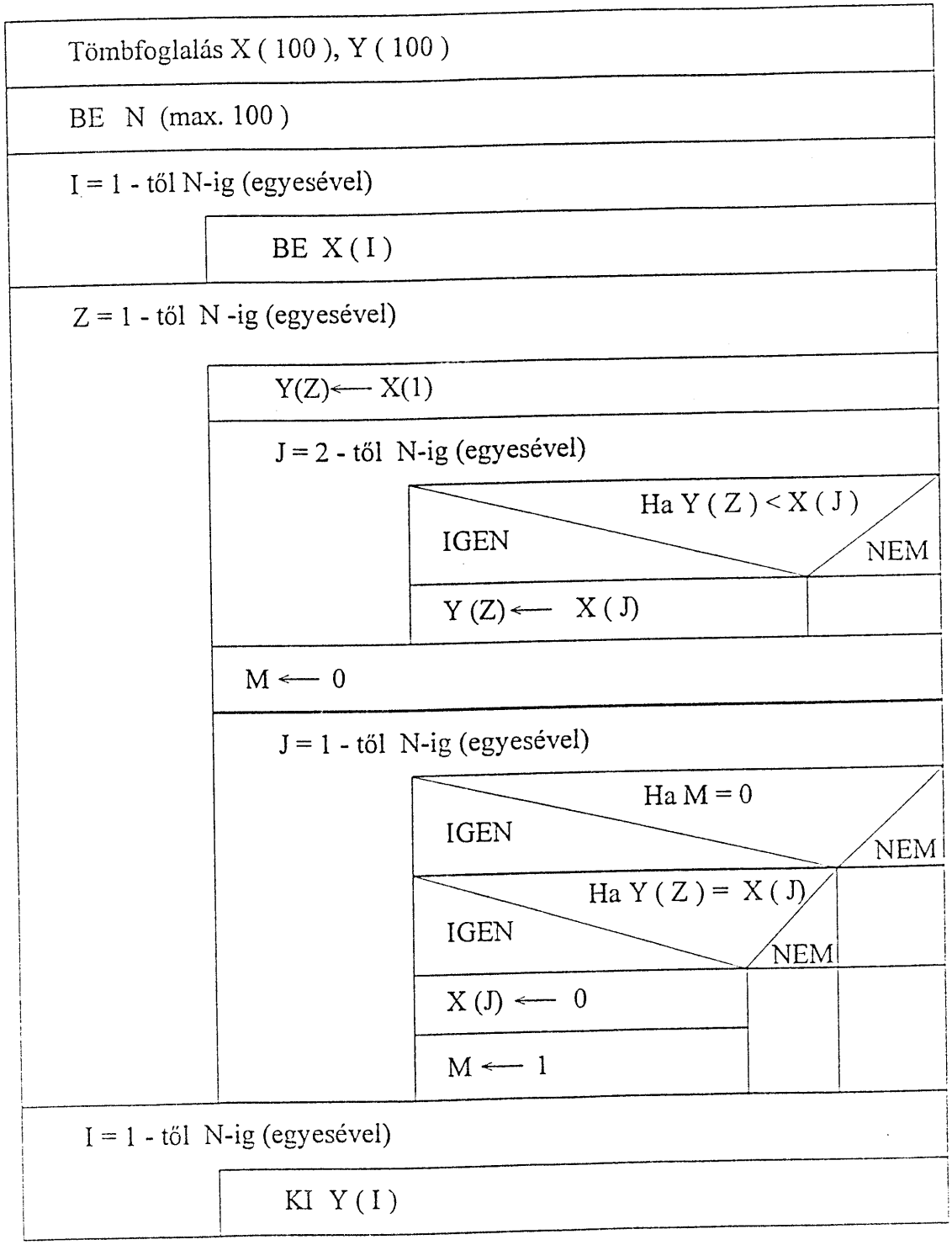


Folyamatábra (A rendezési probléma 2.megoldása)

Az adatbeviteli, ill. -kiíratási rutinokra vonatkozó folyamatábra részletek:







Struktogram (A rendezési probléma 2.megoldása)

```
algoritmus_a_rendezési_probléma_második_megoldása
0. TÖMBFOGLALÁS X(100), Y(100)
1. KI 'Hány számot kíván rendezni? (max. 100)'
2. BE N
3. CIKLUS I = 1-től N-ig (egyesével)
     3.1. BE X(I)
   CIKLUSVÉG
4. CIKLUS Z = 1-től N-ig (egyesével)
     4.1. Y(Z) ← X(1)
     4.2. CIKLUS J = 2-től N-ig (egyesével)
                4.2.1. HA Y(Z) < X(J) AKKOR
                              4.2.1.1. Y(Z) ← X(J)
                       HAVÉGE
          CIKLUSVÉG
     4.3. M ← 0
     4.4. CIKLUS J = 1-től N-ig (egyesével)
                4.4.1. HA M = 0 AKKOR
                              4.4.1.1. HA Y(Z) = X(J) AKKOR
                                            4.4.1.1.1. X(J) ← 0
                                            4.4.1.1.2. M ← 1
                                       HAVÉGE
                       HAVÉGE
          CIKLUSVÉG
   CIKLUSVÉG
5. CIKLUS I = 1-től N-ig (egyesével)
     5.1. KI Y(I)
   CIKLUSVÉG
algoritmus_vége.
```

(Mi történik, ha a program használója - a figyelmeztetés ellenére - az N változó részére 100-nál nagyobb számot ad meg? Hogyan egészíthetnénk ki a programot, hogy erre ne kerülhessen sor?)

N számot rendező programunk lényegében ugyanolyan hosszú, mint a 3 számot rendező 1. megoldás! Ezt a tömbök használatának és a ciklusképzésnek köszönhetjük. Ez már "igazi számítógépes program", a gép jóval több utasítást hajt végre, mint ahányat leírtunk: azonos műveleteket ismétel sokszor, változó adatokkal. Van azonban egy - gyakorlati szempontból lényeges - hátránya:

igen nagy tárterületet igényel. Minden szám szerepel az X és az Y tömbben is. N szám esetén legalább 2*N rekeszre van szükségünk. Ha sok számot kívánunk rendezni, könnyen előfordulhat, hogy nem férünk el a memóriában. Nyilván célszerű, hogy valamennyi rendezendő szám egyszerre, bent legyen a memóriában - N rekeszre tehát biztosan szükségünk van. Vajon nem lehetne egy-két további rekesszel megoldani a feladatot?

3. megoldás

Megmutatjuk, hogy N számot rendezni tudunk összesen N+1 rekeszben - sőt, új, "kényszer szülte" módszerünk további előnyökkel is jár. Mindenesetre gyökeresen más gondolatokra - más algoritmusra van szükségünk, mint amivel eddig dolgoztunk. Eddigi algoritmusaink lényege az volt, hogy mindig egy adott halmazban a legnagyobb számot kerestük, vagyis pontosan utánoztuk az emberi gondolkodást. Ez a gépi megvalósításban bonyodalmakat okozott.

Ha N szám nagyság szerint csökkenő sorrendben rendezve van, akkor $x_i \geq x_{i+1}$ $\forall$ i-re. Ha tehát valamilyen j-re azt tapasztaljuk, hogy $x_j < x_{j+1}$ és egyszerűen felcseréljük x_j -t x_{j+1} -gyel, a számok máris "kissé rendezettebbek", hiszen most már $x_j^* > x_{j+1}^*$. Hasonlítsuk össze most x_{j+1} -et x_{j+2} -vel, és ha $x_{j+1} < x_{j+2}$, akkor cseréljük fel x_{j+1} -et x_{j+2} -vel .

Próbálkozzunk tehát a következő algoritmussal:

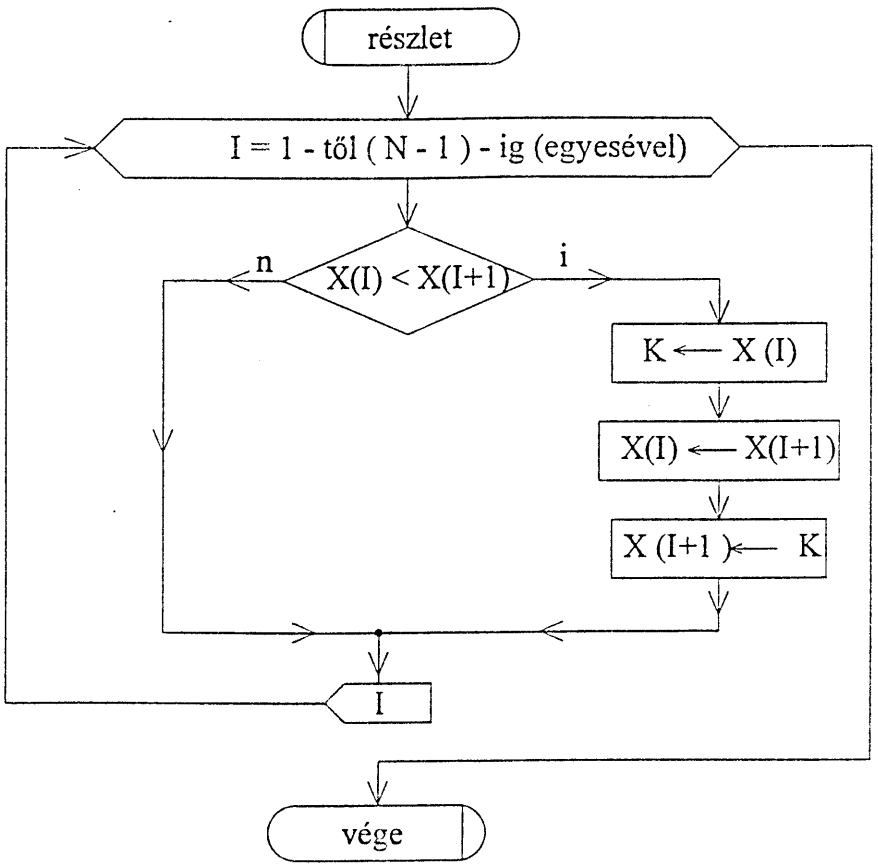
a) az i változó értéke legyen 1

b) ha $x_i < x_{i+1}$, akkor felcseréljük x_i -t x_{i+1} -gyel

c) növeljük i értékét 1-gyel

d) ha i < N, akkor visszatérünk b)-re.

Már tisztáztuk, hogy két tároló tartalmát mindig egy harmadik tároló közbeiktatásával cserélhetjük fel (ez az a bizonyos +1 tároló). Legyen ez az átmeneti tároló K. Máris felrajzolhatjuk a szöveges algoritmus-vázlatot részletező folyamatábrát, ill. a megfelelő pszeudokódos programrészletet:



1. CIKLUS I = 1-től N-1 -ig (egyesével)
    1.1. HA X(I) < X(I+1) AKKOR
        1.1.1. K ← X(I)
        1.1.2. X(I) ← X(I+1)
        1.1.3. X(I+1) ← K
    HAVÉGE
CIKLUSVÉG

A ciklusváltozó felső határa N-1, hiszen a ciklus N-edik végrehajtása során már x_n-t hasonlítanánk (a nem létező) x_{n+1}-gyel!

Készen van-e a rendezés, amikor a gép kilép a fenti ciklusból? Sajnos nem! Próbáljuk ki például öt számmal:

"rendezés előtti" sorrend	"rendezés utáni" sorrend
2, 7, 11, 4, 28	7, 11, 4, 28, 2

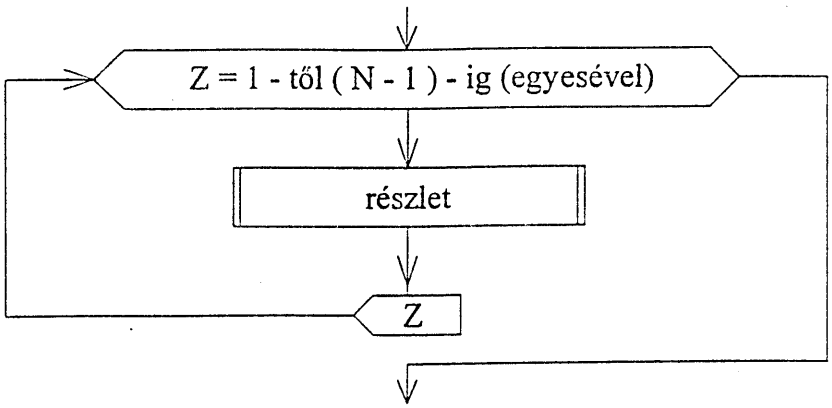
A követési tábla alakulása:

N	5				
X(1)	2 (7)				
X(2)	7 2	(11)			
X(3)	11	2	(4)		
X(4)	4		2	(28)	
X(5)	22			(2)	
i	1	2	3	4	5
K	2	2	2	2	

Egyetlen dologban lehetünk csak biztosak: ha nem a legnagyobb szám állt legelől, akkor a legnagyobb szám egy hellyel előbbre került. (A második legnagyobb számra ez már nem feltétlenül igaz! Konstruáljunk példát ennek bizonyítására!) További teendőink meghatározásában az az egyszerű gondolat segít, hogy programunknak "az elképzelhető legrosszabb esetben" is helyes

eredményt kell adnia. Mi az elképzelhető legrosszabb eset? Az, hogy a legnagyobb szám a program indulásakor a legutolsó helyen áll, vagyis N-1 hellyel kellene előbbre lépnie. Mivel az 1. ciklus teljes végrehajtása során a legnagyobb szám egy hellyel tud előrelépni, a teljes ciklust N-1 -szer meg kell ismételnünk.

A 1. ciklust tehát beleágyazzuk egy másik ciklusba, amit szintén N-1 -szer hajtunk végre. Legyen ennek ciklusváltozója Z:



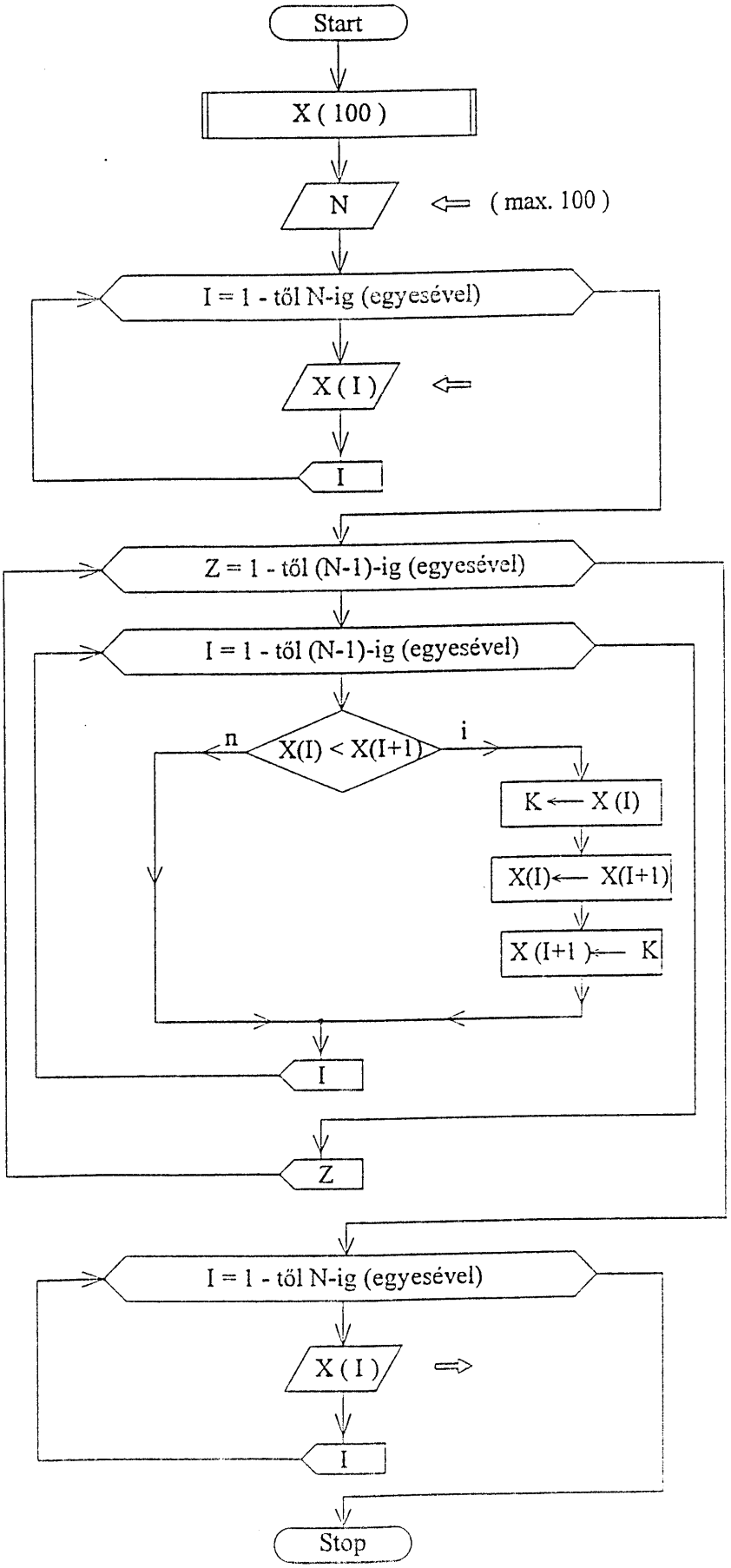
A megfelelő pszeudokódos változat:

```
4. CIKLUS  Z = 1-től N-1 -ig (egyesével)
     4.1. CIKLUS  I = 1-től N-1 -ig (egyesével)
                4.1.1. HA  X(I) < X(I+1)  AKKOR
                             4.1.1.1.  K ← X(I)
                             4.1.1.2.  X(I) ← X(I+1)
                             4.1.1.3.  X(I+1) ← K
                  HAVÉGE
          CIKLUSVÉG
   CIKLUSVÉG
```

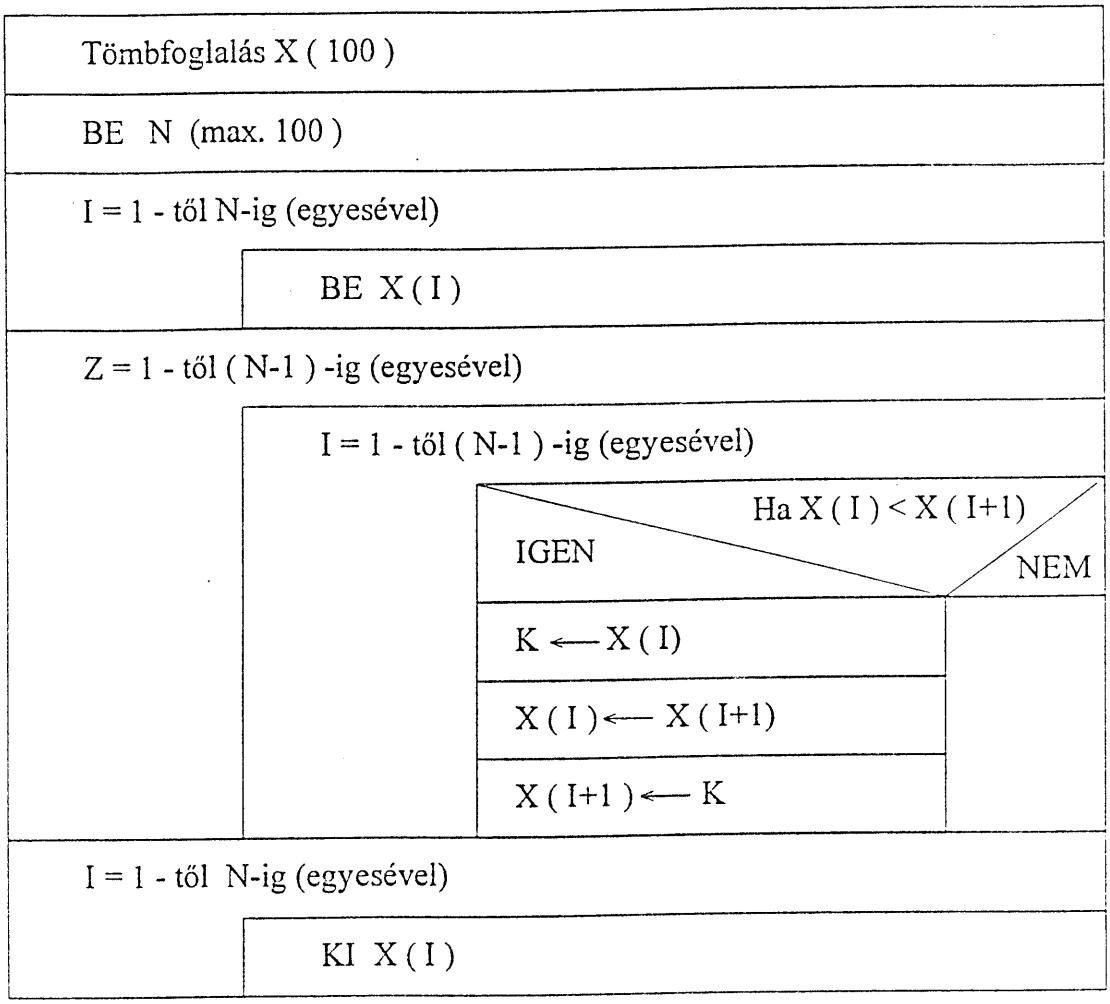
A Z ciklusváltozó értékét a cikluson belül sehol nem használjuk, egyszerűen csak arra szolgál, hogy a gép N-1 -szer megismételje a teljes 4.1. ciklust. Ez a módszer igen gyakori a számítástechnikában. A 4. ciklust a szó szoros értelmében egy "számlálással működő ciklusutasítás" valósítja meg.

Most már biztosak lehetünk abban, hogy a legnagyobb szám az első helyen áll. Ennél azonban több is igaz. Gyakorlatképpen bizonyítsuk be, hogy a 4. (külső) ciklusból kilépve a teljes X( ) tömb - csökkenőleg - rendezve van! (Pontosan milyen állítás igaz a második legnagyobb számra, a harmadik legnagyobb számra, stb., illetve a legkisebb számra?)

Harmadik megoldásként született (szintén legfeljebb 100 számot) rendező algoritmusunkat adjuk meg folyamatábrában, struktogramban és pszeudokódolt program formájában is!



Folyamatábra (A rendezési probléma 3.megoldása)



Struktogram (A rendezési probléma 3.megoldása)

```
algoritmus_a_rendezési_probléma_harmadik_megoldása
0. TÖMBFOGLALÁS X(100), Y(100)
1. KI 'Hány számot kíván rendezni? (max. 100)'
2. BE N
3. CIKLUS I = 1-től N-ig (egyesével)
     3.1. BE X(I)
   CIKLUSVÉG
4. CIKLUS Z = 1-től N-1 -ig (egyesével)
     4.1. CIKLUS I = 1-től N-1 -ig (egyesével)
                4.1.1. HA X(I) < X(I+1) AKKOR
                           4.1.1.1. K ← X(I)
                           4.1.1.2. X(I) ← X(I+1)
                           4.1.1.3. X(I+1) ← K
                  HAVÉGE
          CIKLUSVÉG
   CIKLUSVÉG
5. CIKLUS I = 1-től N-ig (egyesével)
     5.1. BE X(I)
   CIKLUSVÉG
algoritmus_vége.
```

A program kevesebb utasításból áll, mint a 2. megoldás, és 2*N rekesz helyett csupán N+1 rekeszt használ a rendezéshez (az elemek felcserélését mindig ugyanazon a K rekeszen keresztül végzi). Ráadásul gyorsabb is. A rendezéshez szükséges időt jellemezhetjük az elvégzett összehasonlítások számával ("HA X(I) < X(I+1)"). A harmadik megoldásban a teljes rendezéshez

$$(N-1)*(N-1) \approx N*N$$

összehasonlítás szükséges, míg a másodikban - a legrosszabb esetben -

$$N*(N-1 + 2*N) = N*(3*N-1) \approx 3*N*N$$ összehasonlítás,

a legjobb esetben pedig $\approx 3/2*N*N$ összehasonlítás.

(A futási időt természetesen a végrehajtott cserék száma is befolyásolja.)

A második megoldással írt program futási ideje tehát legrosszabb esetben kb. háromszorosa a harmadik programénak, és a legjobb esetben sem gyorsabb.

Még ennél is nagyobb előny azonban, hogy észrevétlenül túlléptünk a feladat követelményein. A harmadik megoldás már nem használja ki azt a feltételt, hogy a számok mind határozottan pozitívak. Az első és második megoldásnál szükségünk volt erre, hogy a számok kinullázásával jelezhessük: "ezzel már nem kell többé foglalkoznunk". A harmadik algoritmus - mivel nem a legnagyobb számot keresi, mindig csak két szomszédos számról állapítja meg, melyik nagyobb - tetszőleges számok rendezésére is alkalmas: az inputon megjelenhetnek a negatív számok és a 0 is. Végül: ha a 4.1.1. döntésben a < jel helyett > jelet írunk, minden további változtatás nélkül ugyanez a program csökkenő helyett növekvő sorrendbe rendezi az X( ) tömb elemeit.

4., ... stb. megoldás

Rendezési algoritmusunkat lépésről-lépésre javítottuk. Vajon eljutottunk a legjobb algoritmushoz? Létezik-e egyáltalán ilyen?

Harmadik algoritmusunk bizonyos értelemben valóban optimális. Nehezen képzelhető el, hogy N számot N+1 -nél kevesebb tárolórekesz használatával rendezzünk. Memórialekötés szempontjából tehát feltehetően elértük az optimumot. A program rövid, tömör, "számítógépszerű": a ciklusképzések miatt sok gépi utasítás végrehajtását indukálja, a 0. ("helyfoglaló") utasítás megváltoztatásával 200, 1000, 10000, ... stb. szám rendezésére is alkalmas - mindaddig "tágítható", amíg az adatok egyáltalán befférnek a rendelkezésre álló memóriába. Nem valószínű, hogy lényegesen kevesebb programsor leírásával is meg tudnánk oldani a feladatot, vagyis a program tömörsége szempontjából is az optimum körül járunk.

Mi a helyzet a futási idővel? Mennyi idő alatt végzi el adott mennyiségű szám rendezését? Láttuk, hogy 100 szám esetén kb. 10 000 összehasonlítást végez, 200 szám esetén 40 000 összehasonlítást stb.: a futási idő az adatok számának négyzetével nő. Ráadásul a legtöbb összehasonlítást "fölöslegesen" végzi el. Ezt tekintve, futási ideje ugyanannyi, ha (véletlenül) az input eleve teljesen rendezett, mint amikor az input "a lehető legrendezetlenebb". A futási idő szempontjából tehát biztosan nem optimális.

Léteznek más rendezési algoritmusok is. Kedvelt megoldás pl. a következő: a rendezni kívánt számokat egyetlen nagy X( ) tömb helyett kisebb - A( ), B( ), C( ), ... - tömbökbe olvassuk be, pl. az első hét számot az A( )-ba, a következő hetet B( )-be, stb. Az A( ), B( ), C( ), ... stb. tömböket önmagukban rendezzük, majd a tömbök elemeit nagyság szerint "összeválogatjuk" egy Y( ) tömbbe. Ha az A( ), B( ), C( ), ... - immár rendezett - tömbök valamelyikének első eleme átkerült Y( )-ba, akkor ezt az elemet töröljük a megfelelő tömbből, és a tömb összes többi elemét egy-egy hellyel "előregörgetjük". Így a rendezés bármely fázisában "a következő legnagyobb szám" csak valamelyik A( ), B( ), C( ), ... tömb első eleme lehet. Ez a módszer általában gyorsabb, mint a mi harmadik megoldásunk (futási ideje tehát rövidebb), viszont hosszabb programot és nagyobb memóriaterületet igényel. És ez természetes. Szeretnénk hangsúlyozni, hogy a számítástechnikában rendszerint nincs minden szempontból optimális megoldás!

4.3. Az induktív és deduktív megközelítés összehasonlító elemzése a 'programozási tételek' tanításában

A programozási tételek alapvető algoritmustípusok általános megfogalmazását jelentik. Jóllehet a 'tétel' elnevezés az állítás kimondását, majd az ezt követő precíz bizonyítást ebben a sorrendben megjelenítő deduktív megközelítést sejtet, a középfokú oktatásban a tanulók életkori sajátosságaiból következő szerényebb absztrakciós képesség miatt az induktív tárgyalásmódot célszerű előnyben részesítenünk. A felsőoktatásban érthető módon többnyire deduktív tárgyalásmóddal találkozunk, noha a programozási tételek bizonyítását még ott is gyakran mellőzik.

Hasonlítsuk össze például a legegyszerűbb programozási tétel, az összegzés tételének deduktív és induktív tárgyalásmódját!

A deduktív megközelítés egy változatát - a programozási tételeket bevezető magyarázat megfelelő részletével együtt - a "Számítástechnika középfokon"[3] c. könyvből idézzük.

"... Az egyes típusalgoritmusoknál mindig megadjuk

- az általános feladatot,
- az azt megoldó algoritmust,
- egy konkrét feladatot,
- a feladatban szereplő adatok megfeleltetését az általános feladat adataival,
- a feladatot megoldó algoritmust.

T1. Az összegzés tétele

Általános feladat:

Adott egy N elemű számsorozat. Számoljuk ki az elemek összegét! A sorozatot most és a továbbiakban is az N elemű A(N) vektorban tároljuk.

Megjegyzés:

A későbbi feladatok során előfordulhat, hogy a vizsgált sorozat szöveges típusú változókat tartalmaz, de ezeket a konkrét programnyelvtől való függésük miatt külön nem jelöljük.

Algoritmus:

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

[3] *Számítástechnika középfokon* (Dr. Hetényi Pálné szerk.), OMIKK, Budapest, 1987., 61-62. pp.

F7.1. feladat:

N napon keresztül, naponta egy alkalommal megmértük a hőmérsékletet. Adjuk még az N napos időszak átlaghőmérsékletét!

Megfeleltetés:

sorozat - A(N)

Algoritmus:

átlagszámítás (N, A( ), ÁTLAG):
S:=0
Ciklus I=1-től N-ig
S:=S+A(I)
Ciklus vége
ÁTLAG:=S/N
Eljárás vége.

Megjegyzés: az összegzés tételét alkalmazzuk akkor is, ha a feladat

A(1) * A(2) * . . . * A(N) vagy
A(1) ÉS A(2) ÉS . . . ÉS A(N) kiszámítása,

természetesen a kezdőérték-adás és a művelet módosításával. ..."

Az alábbiakban az összegzés tételének egy lehetséges induktív megközelítését mutatjuk be.

1. Feladat:

Egy osztályban N tanuló írt fizika dolgozatot.
Dolgozatjavítás után az eredmények ismeretében számítsuk ki az osztály-átlagot!

A probléma elemzése:

N db osztályzat átlagát kell kiszámolni.

Matematikai modell:

n db szám (x_1, x_2, ... , x_n) számtani közepe:

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Az algoritmus szerkesztése:

Az összegző rekesz tartalmát kiindulásként nullára állítjuk (az összeget a zérus összeadandó nem változtatja).
Az összegző utasítás: SUM ← SUM + X úgy működik, hogy az összeg addigi értékéhez hozzáadja a soron következő összeadandót, majd ezt az új összeget tárolja, a korábbit felülírva. Ezt az utasítást nyilván N-szer kell majd végrehajtani, amíg a teljes összeg kialakul. Ezt tehát számlálással vezérelt ciklusba szervezzük, csakúgy mint az N db összeadandó (osztályzat) adatbevitelére szolgáló BE X utasítást.
A ciklusból való kilépés után a kialakult végső összeget az összeadandók darabszámával osztva kapjuk a keresett átlagot.

Az algoritmusban használt jelölések:

Az adatok és a keresett mennyiség vonatkozásában

$n, x_i, \bar{x} \quad \Rightarrow \quad$ N, X, XA

Az összeg lépésről-lépésre történő összegyűjtésére használt ún. összegző rekesz neve SUM .

A számlálásra használt változó neve I .

Az algoritmus pszeudokódolt változata:

algoritmus_osztályátlag
1. KI 'Hány osztályzat átlagát számoljuk? (N=?)'
2. BE N
3. **SUM ← 0**
4. **CIKLUS I = 1-től N-ig** (egyesével)
   4.1. BE X
   4.2. **SUM ← SUM + X**
   CIKLUSVÉG
5. XA ← SUM/N
6. KI 'Az osztály-átlag: XA= ', XA

algoritmus_vége

Az algoritmus ellenőrzése ún. követési táblázattal:

Próbáljuk ki algoritmusunkat 3 osztályzat (4,5,3) átlagának kiszámítására!

N	3				
SUM	0	4	9	12	
I	1	2	3	4	
X	4	5	3		
XA					4

Az algoritmus helyesen működik, hiszen $\frac{4+5+3}{3}=4$, a várt eredményt kaptuk.

A követési táblázatot tanulmányozva látható, hogy a programfutás végén az egyes változóknak csak az utolsó értéke érhető el, az N, SUM, I, *X*, XA változók tartalma most rendre 3, 12, 4, *3*, 4. Az összeadandók közül tehát csak az utolsóként megadott "***hármas*** osztályzatra emlékszik". Ez az ára annak, hogy megoldásunkban index nélküli változóval dolgoztunk, az összes osztályzatot ugyanabba a változóba olvastuk be és felhasználás után rendre felülírtuk azokat a soron következővel. Kitűzött feladatunkat hibátlanul, sőt így memóriatakarékosan oldottuk meg, hiszen a ciklusból való kilépés után nem volt már szükségünk az egyes osztályzatokra. Más a helyzet a következő feladatnál.

2. Feladat:

Egy 35 fős osztályban N tanuló írt fizika dolgozatot.
Dolgozatjavítás után az eredmények ismeretében számítsuk ki az osztály-átlagot!
Határozzuk meg a szórást is!

1. Megoldás

A probléma elemzése:

N db osztályzat átlagát és szórását kell kiszámolni.

Matematikai modell:

n db szám (x_1, x_2, ... , x_n)

számtani közepe: $$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{\sum_{i=1}^{n} x_i}{n}$$

szórása: $$\sigma = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \ldots (x_n - \bar{x})^2}{n-1}} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

Az algoritmus szerkesztése:

Az előző példa algoritmus-szerkesztési megfontolásai természetesen itt is érvényesek. Tekintettel arra azonban, hogy a szóráshoz szükséges átlagtól való eltérések miatt (az átlag kiszámítása után!) újra szükségünk lesz az adatokra (az egyes osztályzatokra), ezúttal indexes változót (egyméretű tömböt = "vektort") használunk. Ha viszont már úgy is tömböt használunk, különítsük el az

algoritmus adatbeviteli és adatfeldolgozási blokkjait. (Most láthatjuk, hogy ezzel a szétválasztással biztosított jobb áttekinthetőséget az 1. feladat memória-takarékos megoldásában nem lehetett megvalósítani.)

Az algoritmusban használt jelölések:

Az adatok és a keresett mennyiség vonatkozásában

$$n, x_i, \bar{x}, \sigma \quad \Rightarrow \quad N, X(\,), XA, XS$$

Az egyes összegek lépésről-lépésre történő összegyűjtésére használt ún. összegző rekeszek neve

az osztályzatok esetében SUM ,
az átlagtól való eltérés-négyzetek esetében SQUM .

A számlálásra használt (és itt egyben index)változó neve I .

Az algoritmus pszeudokódolt változata:

```
algoritmus_osztályátlag_szórás
0. TÖMBFOGLALÁS  X(35)
1. KI  'Hány osztályzat átlagát  számoljuk? (N=?)'
2. BE  N
3. CIKLUS  I = 1-től N-ig  (egyesével)
     3.1. BE  X(I)
   CIKLUSVÉG
4. SUM ← 0
5. CIKLUS  I = 1-től N-ig  (egyesével)
     5.1. SUM ← SUM + X(I)
   CIKLUSVÉG
6. XA ← SUM/N
7. SQUM ← 0
8. CIKLUS  I = 1-től N-ig  (egyesével)
     8.1. SQUM ← SQUM + (X(I)-XA)↑2
   CIKLUSVÉG
9. XS ← NÉGYZETGYÖK(SQUM/(N-1))
```

```
10. KI 'Az osztály-átlag: XA=', XA
11. KI 'A szórás: XS=', XS
algoritmus_vége
```

2. Megoldás

A 2. Feladat megoldható index nélküli változókkal (tömb használata nélkül) is. Hogyan? Egyszersmind csökkenthető-e a ciklusok száma?
(Útbaigazítás: a matematikai modell átalakítása)

Ha először megoldunk legalább két konkrét példát, ezek algoritmusait elemezhetjük és **a közös részleteket** általánosan is értelmezhetjük. A **közös lényegi mag** lesz az, amelynek absztrakciója egy adott programozási tételben nyerhet megfogalmazást.

4.4 Példa komplex megoldási algoritmusok kidolgozására

A következőkben megszerkesztjük a Gauss(-Jordan) elimináció algoritmusának egy lehetséges változatát

1. lineáris egyenletrendszerek megoldására,
2. mátrixok multiplikatív inverzének előállítására,
3. determinánsok értékének kiszámítására

példát adva ezzel arra, miként oldhatunk meg komplex és többcélú problémákat.

1. Lineáris egyenletrendszerek lehetséges megoldásainak probléma-elemzése és matematikai modellje:

Lineáris egyenletrendszerek megoldásakor tekintsük a következő ekvivalens átalakításokat:

(i) két egyenlet sorrendjének felcserélése,
(ii) valamelyik egyenlet végigszorzása egy nem nulla valós számmal,
(iii) valamelyik egyenlet konstansszorosának hozzáadása egy másikhoz.

Ezek a műveletek az egyenletrendszer megoldáshalmazát nem változtatják meg, az átalakított egyenletrendszer az eredetivel ekvivalens, megoldáshalmaza az eredetiével azonos.

Munkánkat megkönnyíthetjük oly módon, hogy az egyenletrendszer egésze helyett csak az együtthatókat és konstansokat tartalmazó ún. kibővített mátrixot írjuk le.

1. Példa

$$x_1 - 4x_2 = 1$$
$$3x_1 + 2x_2 = 17$$

$$\left[\begin{array}{cc|c} \boxed{1} & -4 & 1 \\ \textcircled{3} & 2 & 17 \end{array}\right] \begin{array}{l} \\ \leftarrow (-3) \end{array}$$

$$\left[\begin{array}{cc|c} 1 & -4 & 1 \\ 0 & \textcircled{14} & 14 \end{array}\right] \begin{array}{l} \\ \cdot \frac{1}{14} \end{array}$$

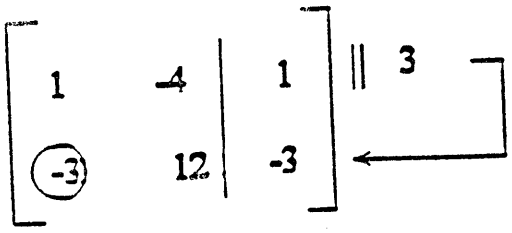
$$\left[\begin{array}{cc|c} 1 & \textcircled{-4} & 1 \\ 0 & \boxed{1} & 1 \end{array}\right] \begin{array}{l} \leftarrow \\ \cdot 4 \end{array}$$

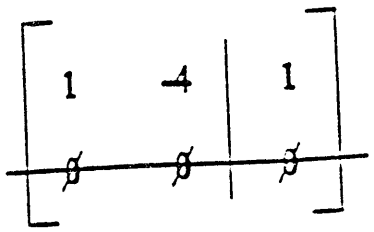
$$\left[\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & 1 \end{array}\right]$$

$$x_1 = 5$$
$$x_2 = 1$$

2. Példa

$$x_1 - 4x_2 = 1$$
$$-3x_1 + 12x_2 = -3$$

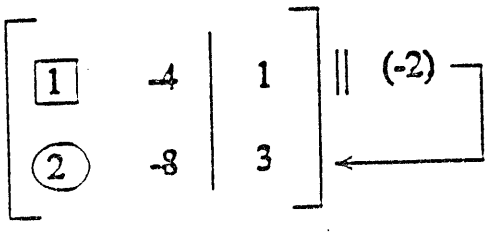


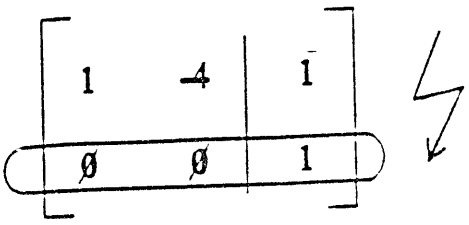


$$x_1 - 4 \cdot x_2 = 1$$
$$0 \cdot x_1 + 0 \cdot x_2 = 0$$

3. Példa

$$x_1 - 4x_2 = 1$$
$$2x_1 - 8x_2 = 3$$





$$x_1 - 4 \cdot x_2 = 1$$
$$0 \cdot x_1 + 0 \cdot x_2 = 1$$

Egy lineáris egyenletrendszer megoldása alapvetően háromféleképpen alakulhat.

Az egyenletrendszernek

(i) egyetlen megoldása van,
(ii) végtelen sok megoldása van,
(iii) nincs megoldása.

Ezt a három alapesetet illusztrálják az előbbiekben megoldott példák. A továbbiakban részletesen csak az első esettel foglalkozunk, az utóbbi kettőre adott esetben csak mint 'rendkívüli eset'-re hivatkozunk.

Az algoritmus szerkesztését előkészítő megfigyelések:

Egyfelől megfigyelhető, hogy 'rendkívüli eset'-et jelez az a tény, ha a Gauss elimináció (itt: 1-es főátlóbeli elemek, és alattuk csupa nulla) nem vihető végig. Másfelől világosan kell látnunk, hogy ha az egyenletrendszernek egyetlen megoldása van, úgy nem csak, hogy végigvihető a Gauss elimináció, de folytatható a csupa 1-es főátlóbeli elemek fölötti eliminálással egészen addig, amíg ún. teljesen redukált rendszert nem kapunk, ami maga a megoldás.

Egy újabb példát oldunk meg erre az esetre, lehetőséget teremtve ezáltal arra, hogy a lehetséges összes (esetlegesen) szükséges átalakítást megfigyeljük, az elimináció folyamatát analizáljuk.

Figyeljük meg, hogy a Gauss-Jordan módszer az ekvivalens átalakításoknak megfelelő eliminációs lépések révén az $[\,\underline{A}\,|\,\underline{b}\,]$ kibővített mátrixot (INPUT) az $[\,\underline{I}\,|\,\underline{x}\,]$ módosított mátrixba (OUTPUT) viszi át.

Példa

$$
\begin{aligned}
2x_1 - 4x_2 &= -6 \\
2x_1 - 4x_2 + 3x_3 &= 6 \\
-3x_2 + 4x_3 &= 10
\end{aligned}
$$

$$
\left[\begin{array}{ccc|c}
\textcircled{2} & -4 & 0 & -6 \\
2 & -4 & 3 & 6 \\
0 & -3 & 4 & 10
\end{array}\right] \quad \| \cdot \frac{1}{2}
$$

$$
\left[\begin{array}{ccc|c}
1 & -2 & 0 & -3 \\
0 & \textcircled{0} & 3 & 12 \\
0 & -3 & 4 & 10
\end{array}\right] \quad \leftarrow \; \leftarrow
$$

$$
\left[\begin{array}{ccc|c}
1 & -2 & 0 & -3 \\
0 & \boxed{1} & \frac{4}{3} & \frac{-10}{3} \\
0 & 0 & \textcircled{3} & 12
\end{array}\right] \quad \| \cdot \frac{1}{3}
$$

$$
\left[\begin{array}{ccc|c}
1 & \textcircled{-2} & 0 & -3 \\
0 & \boxed{1} & 0 & 2 \\
0 & 0 & 1 & 4
\end{array}\right] \quad \leftarrow \; \| \cdot 2
$$

$$
\left[\begin{array}{ccc|c}
\boxed{1} & -2 & 0 & -3 \\
\textcircled{2} & -4 & 3 & 6 \\
0 & -3 & 4 & 10
\end{array}\right] \quad \| \cdot (-2) \; \leftarrow
$$

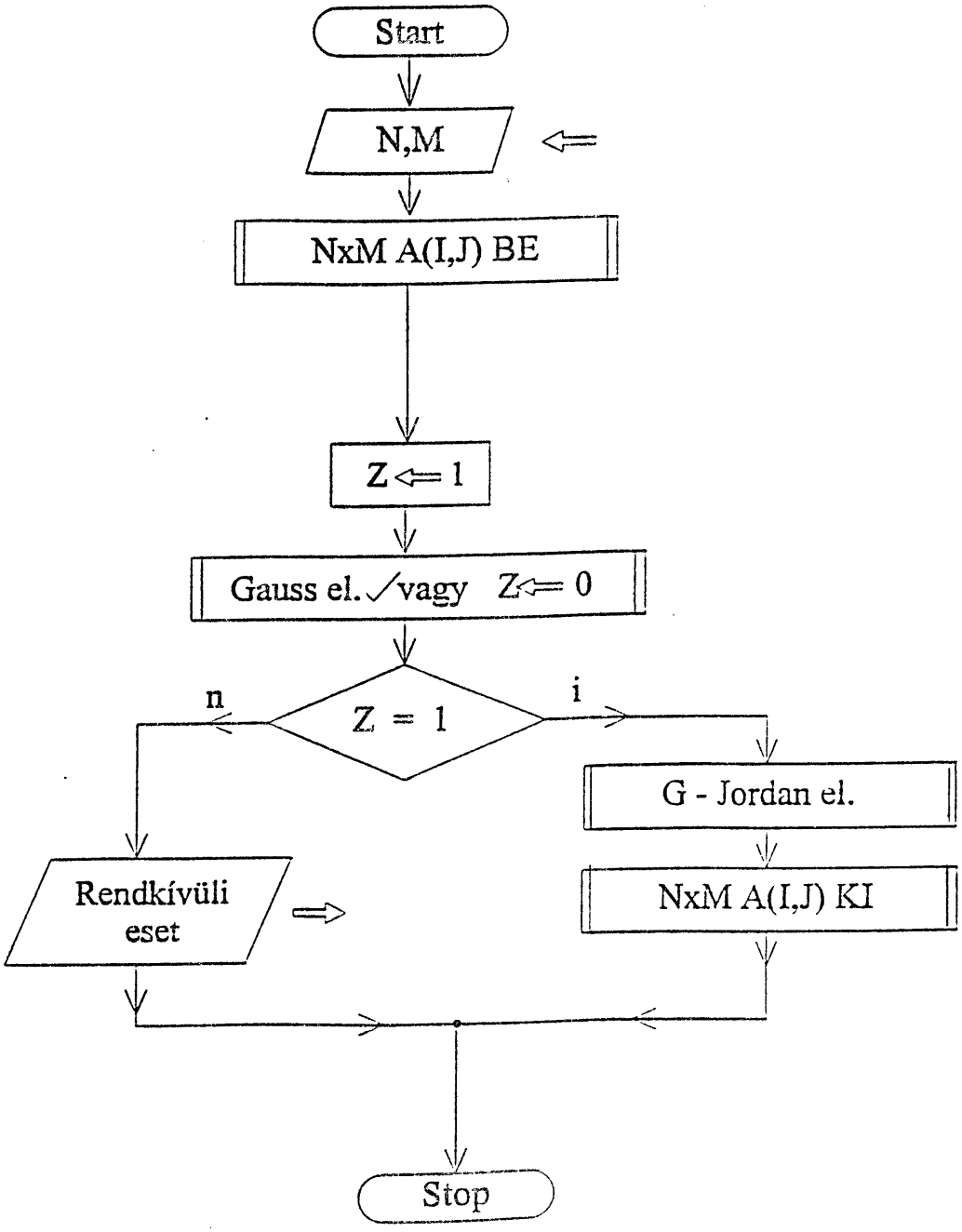
$$
\left[\begin{array}{ccc|c}
1 & -2 & 0 & -3 \\
0 & \textcircled{-3} & 4 & 10 \\
0 & 0 & 3 & 12
\end{array}\right] \quad \| \cdot \left(\frac{-1}{3}\right)
$$

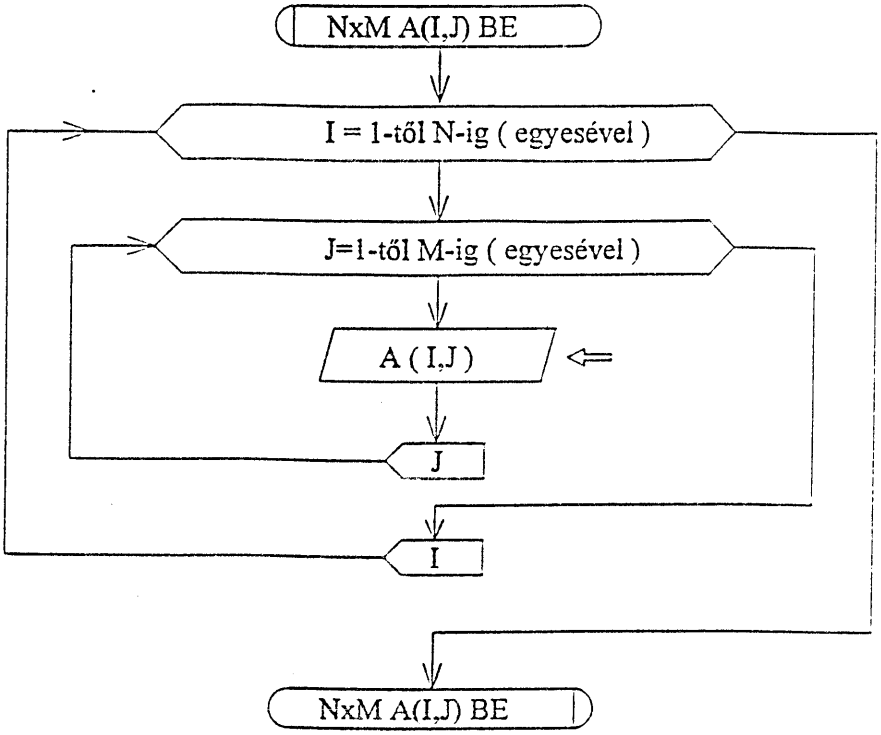
$$
\left[\begin{array}{ccc|c}
1 & -2 & 0 & -3 \\
0 & 1 & \textcircled{\frac{-4}{3}} & \frac{-10}{3} \\
0 & 0 & \boxed{1} & 4
\end{array}\right] \quad \leftarrow \; \| \cdot \frac{4}{3}
$$

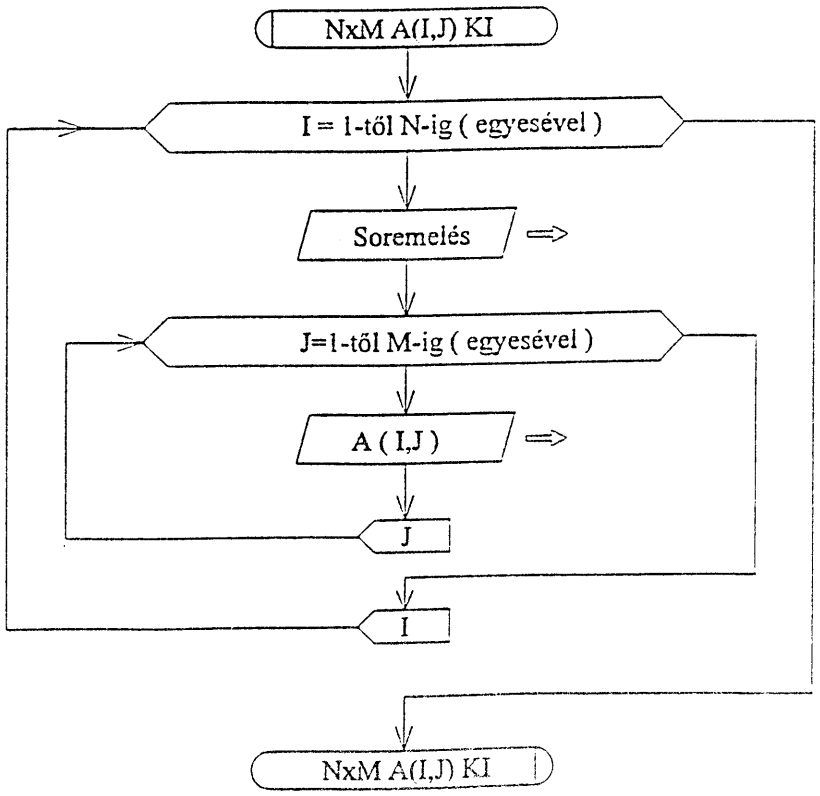
$$
\left[\begin{array}{ccc|c}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 2 \\
0 & 0 & 1 & 4
\end{array}\right]
$$

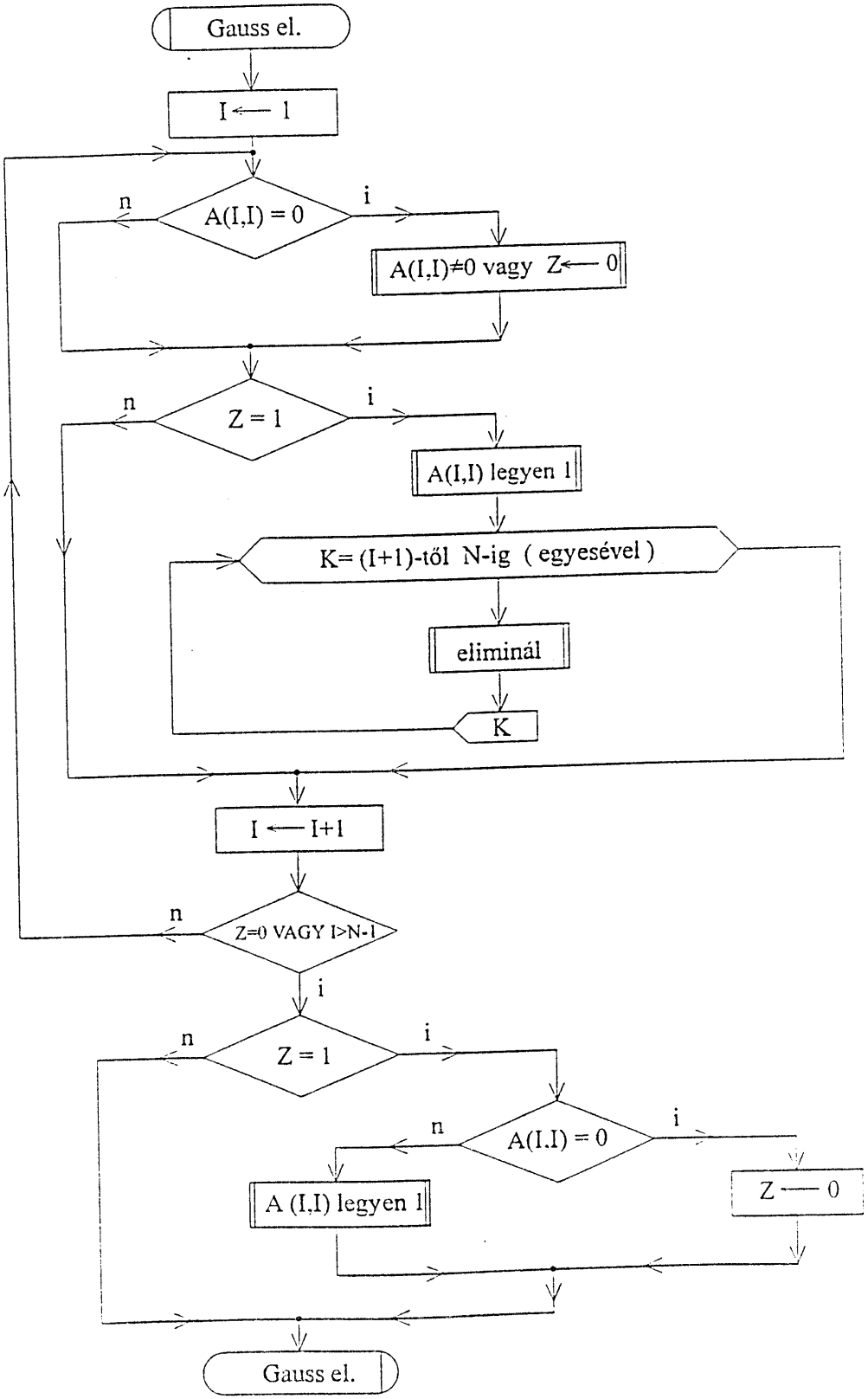
$$
\begin{aligned}
x_1 &= 1 \\
x_2 &= 2 \\
x_3 &= 4
\end{aligned}
$$

A fő folyamatábra az algoritmus vázlatát mutatja. Ez négy olyan alprogramot tartalmaz, amelyeket a továbbiakban külön folyamatábrák segítségével részletezünk majd. Zászlót (Z változó) használunk annak jelzésére, vajon a Gauss elimináció sikeres-e vagy sem.

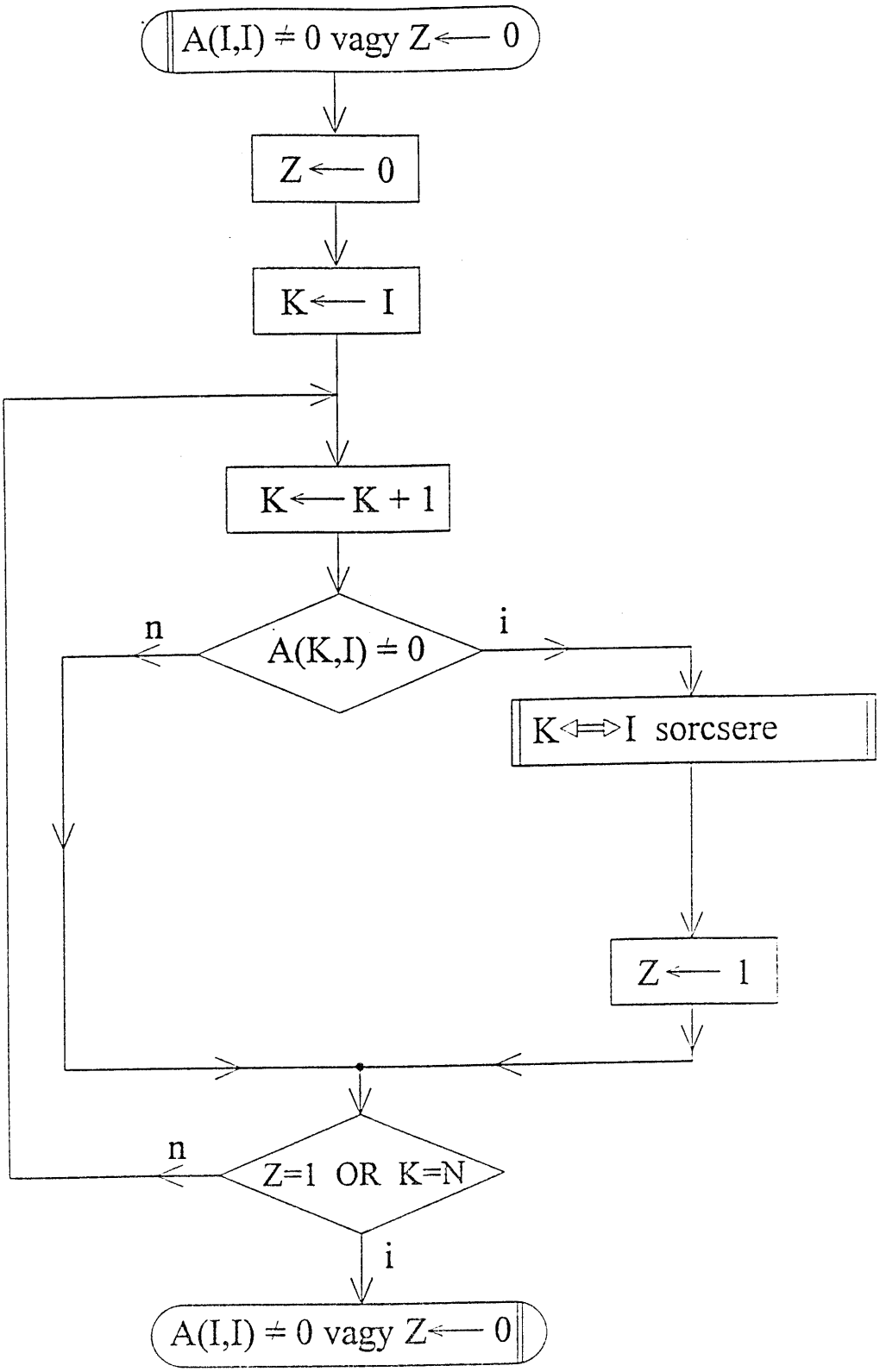



NxM A(I,J) BE
I = 1-től N-ig (egyesével)
J=1-től M-ig (egyesével)
A (I,J)
J
I
NxM A(I,J) BE

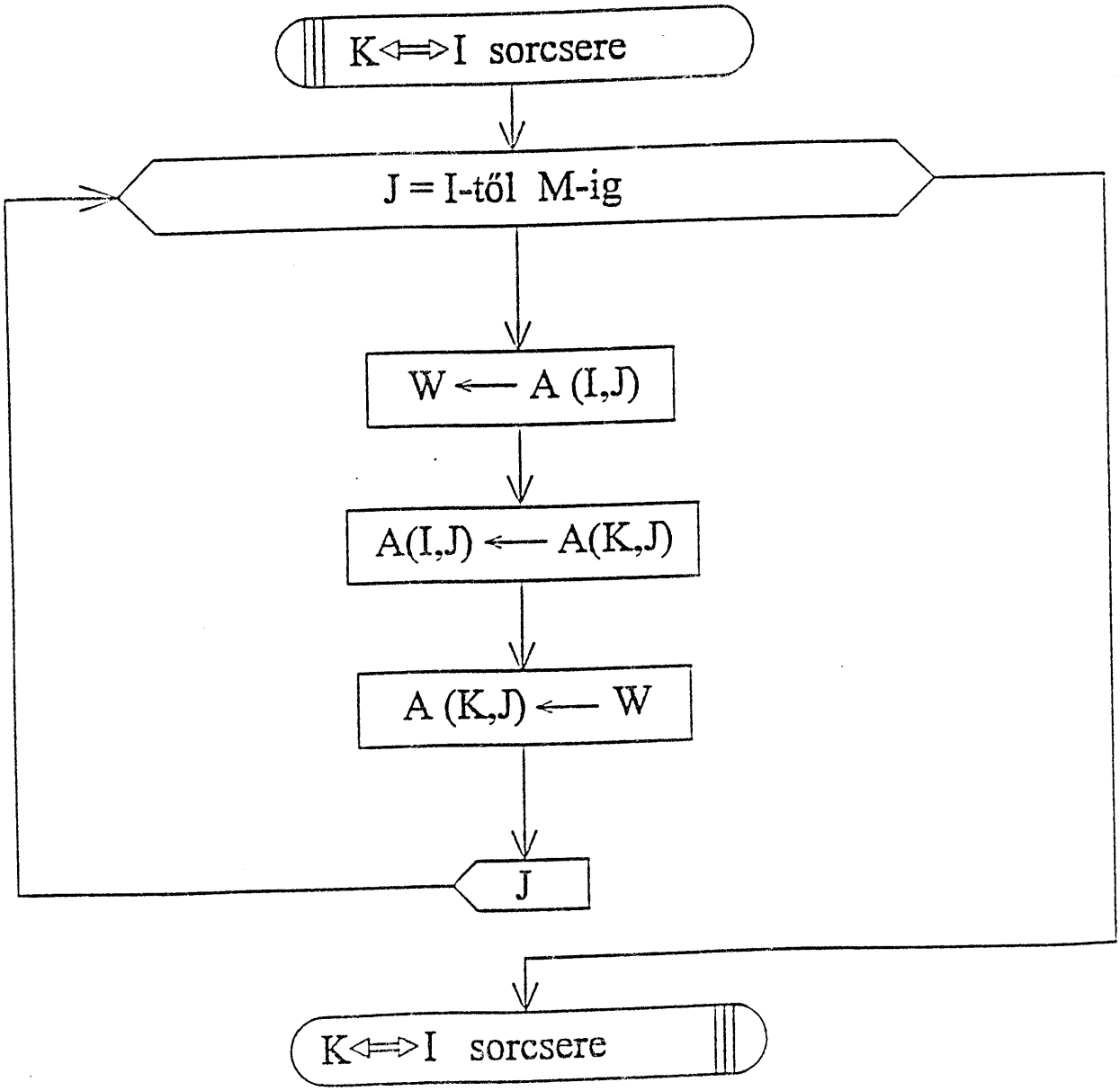

NxM A(I,J) KI
I = 1-től N-ig (egyesével)
Soremelés
J=1-től M-ig (egyesével)
A (I,J)
J
I
NxM A(I,J) KI

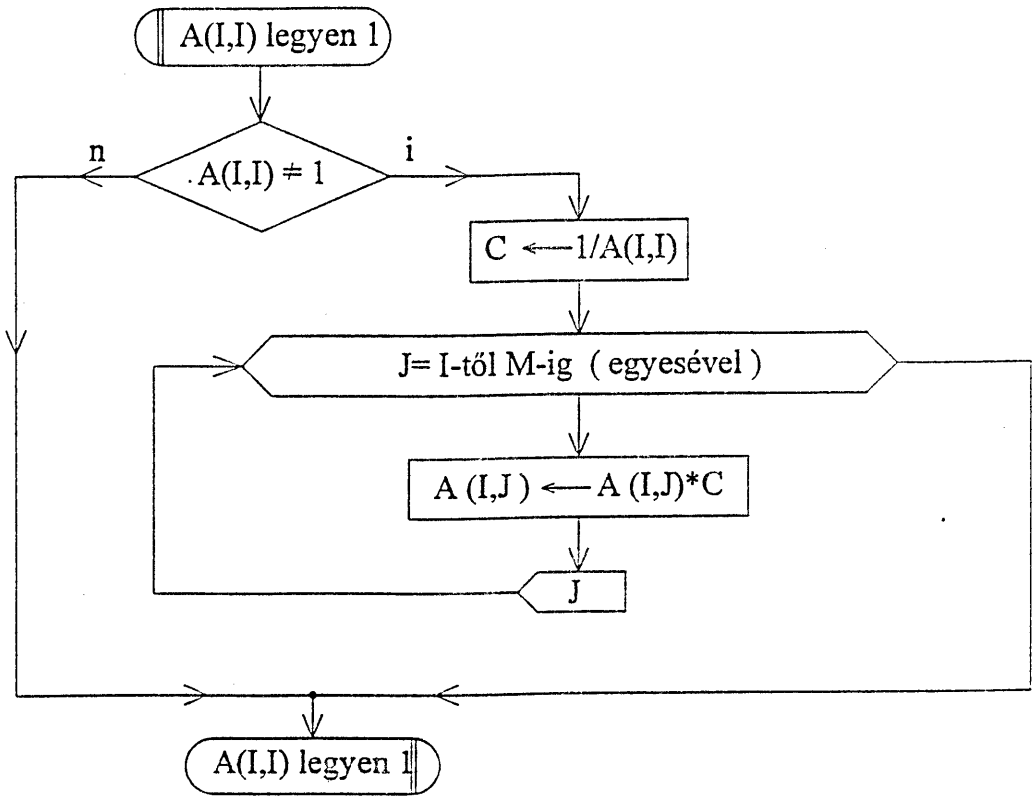

Gauss el.
I ⟵ 1
A(I,I) = 0
n
i
A(I,I)≠0 vagy Z⟵ 0
Z = 1
n
i
A(I,I) legyen 1
K= (I+1)-től N-ig (egyesével)
eliminál
K
I ⟵ I+1
Z=0 VAGY I>N-1
n
i
Z = 1
n
i
A(I,I) = 0
n
i
A (I,I) legyen 1
Z ⟵ 0
Gauss el.

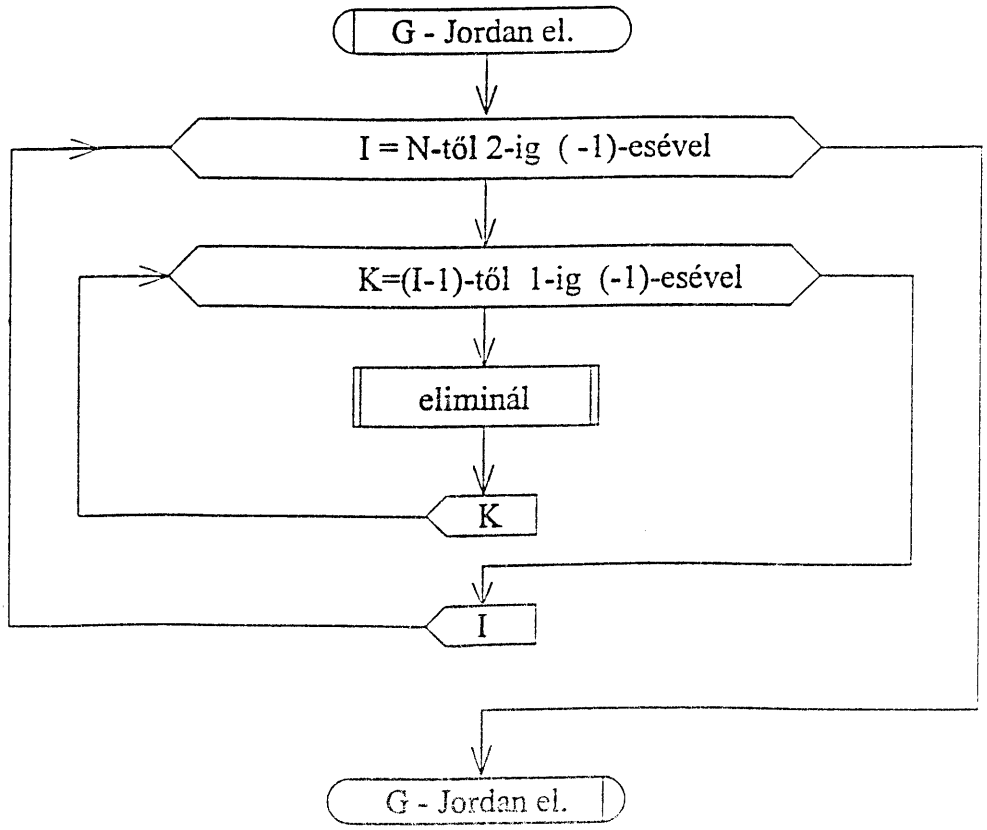
A 'Gauss elimináció' eljárás feladata, hogy a főátló alatti eliminálást elvégezze, amennyiben az lehetséges. A főátló mentén minden elemet meg kell vizsgálni. Ha valamelyikük nulla lenne, úgy alkalmas sorcserével meg kell próbálni elérni azt, hogy ne legyen nulla ('A(I,I)≠0 vagy Z←0' nevű eljárás). Ha ez nem sikerül, úgy a zászló (Z nevű változó) nullára állításával 'rendkívüli esetet' kell jelezni. Bármely nem nulla főátlóbeli elem redukálható 1-esre ('A(I,I) legyen 1' nevű alprogram feladata) és ezt célszerű meg is tenni, hiszen így könnyebb dolga lesz az 'eliminál' nevű eljárásnak. A ciklus szervezésével kapcsolatban itt két érdekes megfigyelést tehetünk. Az egyik az, hogy miután nem tudjuk előre, hányszor lesz majd a ciklusmag végrehajtva, nem használhatunk számlálással vezérelt FOR ciklust. A flexibilis ciklusok közül megoldásunkban egy hátultesztelő ciklust választottunk. A másik hangsúlyozásra érdemes momentum az, hogy az utolsó főátlóbeli elem vizsgálata és az ezzel kapcsolatos teendők a ciklusmagon kívül, a ciklusból való kilépés után kellett, hogy helyet kapjanak, hiszen eliminálási teendő itt már nincsen.
Ez az első hierarchiai szintű alfolyamatábra további 3 második hierarchiai szintű eljárást tartalmaz, amelyek külön folyamatábrák segítségével nyernek majd részletezést. Egyikük, az 'A(I,I) legyen 1' nevű a 'Gauss elimináció' eljárás két különböző helyéről is aktivizálható. (Később látni fogjuk, hogy az 'eliminál' eljárás pedig nem csak ebből, hanem a 'G-Jordan el.' nevű másik, ugyancsak első hierarchiai szintű alprogramból is hívható.)


A(I,I) ≠ 0 vagy Z ← 0
Z ← 0
K ← I
K ← K + 1
A(K,I) ≠ 0
n
i
K ⇐⇒ I sorcsere
Z ← 1
Z=1 OR K=N
n
i
A(I,I) ≠ 0 vagy Z ← 0

Ez a második hierarchiai szintű eljárás bizonyos esetekben aktivizálhatja a harmadik hierarchiai szinten megfogalmazott 'K↔I sorcsere' rutint, amelyet ugyancsak külön alfolyamatábrán részletezünk.




A(I,I) legyen 1
n
A(I,I) ≠ 1
i
C ⟵1/A(I,I)
J= I-től M-ig (egyesével)
A (I,J) ⟵ A (I,J)*C
J
A(I,I) legyen 1

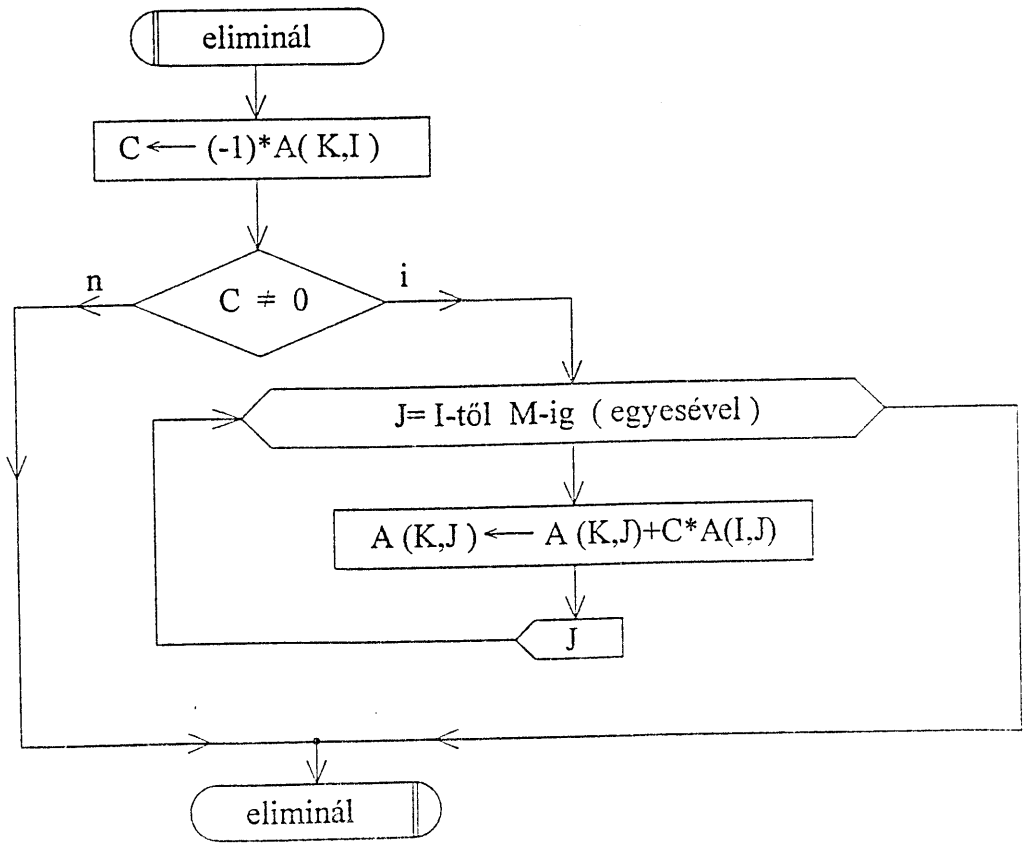

G - Jordan el.
I = N-től 2-ig (-1)-esével
K=(I-1)-től 1-ig (-1)-esével
eliminál
K
I
G - Jordan el.

A 'G-Jordan el.' eljárást tanulmányozva megállapíthatjuk, hogy az sokkal egyszerűbb, mint a 'Gauss elimináció', hiszen azon a ponton, amikor az meghívásra kerül, biztosak lehetünk abban, hogy

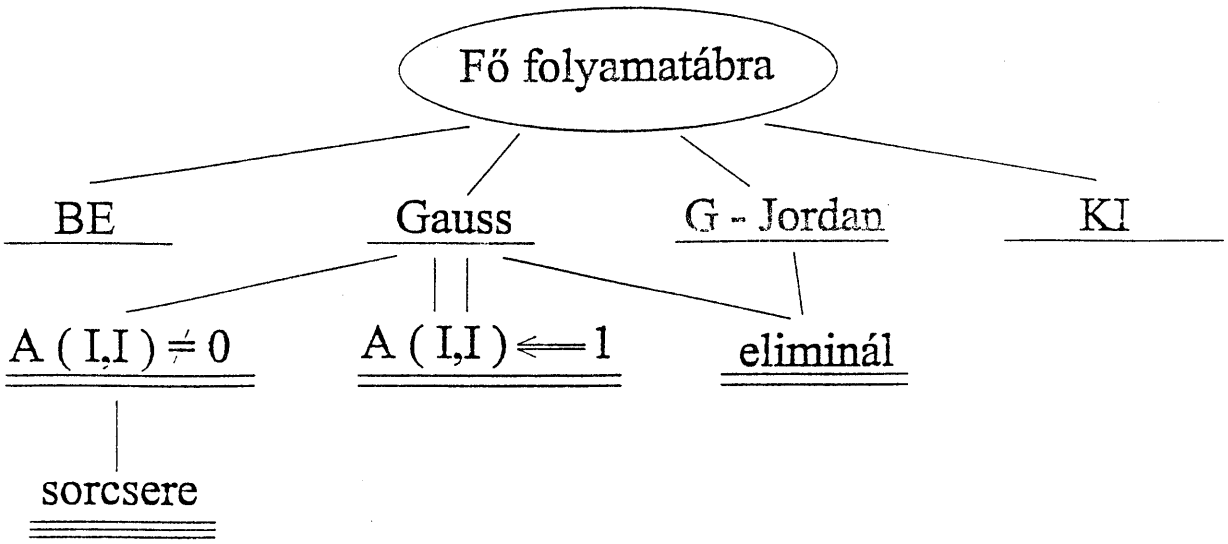
1) a főátlóban nem lehet nulla elem, sőt valamennyi főátlóbeli elem 1-es; és így az 'eliminálás' alprogram az egyetlen, amelyre ez esetben szükség van.

2) a G-Jordan eliminálási folyamat (főátló feletti nullázás) biztosan végigvihető, következésképpen számlálással vezérelt FOR ciklus használható.

A második hierarchiai szintű 'eliminál' alprogram aktivizálása természetesen mindkét eliminációs eljárásból (mind a 'Gauss elimináció'-ból, mind pedig a 'G-Jordan el.'-ból) kezdeményezhető.



Az alábbiakban egy hierarchia-ábra segítségével foglaljuk össze az algoritmus valamennyi részletét:

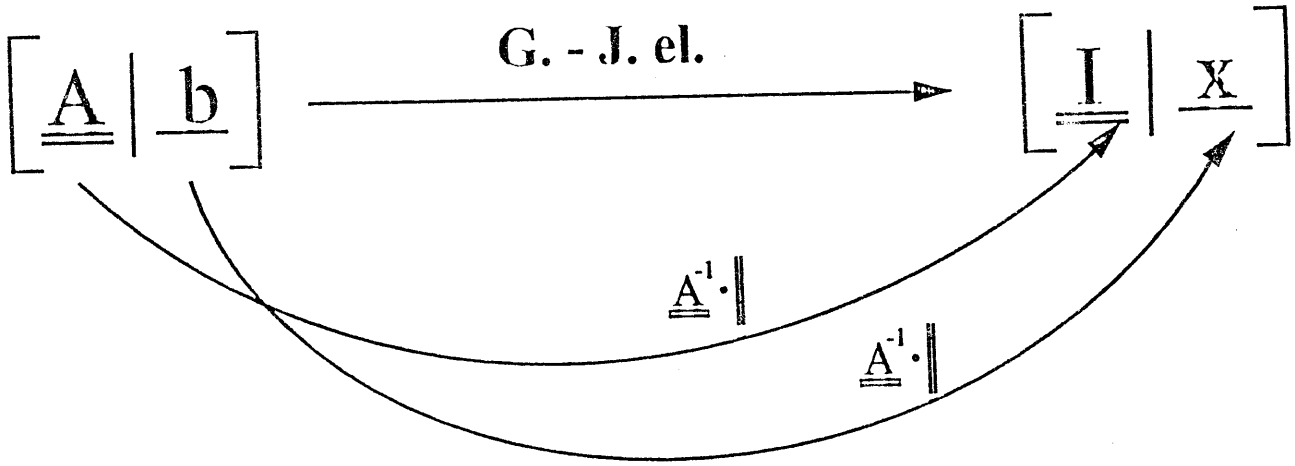


2. Probléma-elemzés és matematikai modell mátrix multiplikatív inverzének meghatározásához:

Az első részben a Gauss-Jordan eliminációs eljárást mint lineáris egyenletrendszerek megoldási módszerét tárgyaltuk. Ugyanakkor ismeretes, hogy egy lineáris egyenletrendszer tömören mátrixegyenletként is kezelhető. Ha tehát a mátrixegyenlet mindkét oldalát balról megszorozzuk az egyenletrendszer együtthatómátrixának inverzével, úgy mindjárt az egyenletrendszer megoldását kapjuk. Összehasonlítva egy lineáris egyenletrendszer (feltételezve, hogy pontosan egy megoldása van) előbbiekben említett két különböző megoldási módszerét - arra a felismerésre jutunk, hogy a Gauss-Jordan elimináció hatásában megfelel az együttható mátrix inverzével balról történő szorzásnak.

(1)

GAUSS - JORDAN ELIMINATIO LINEÁRIS EGYENLETRENDSZEREK MEGOLDÁSÁRA



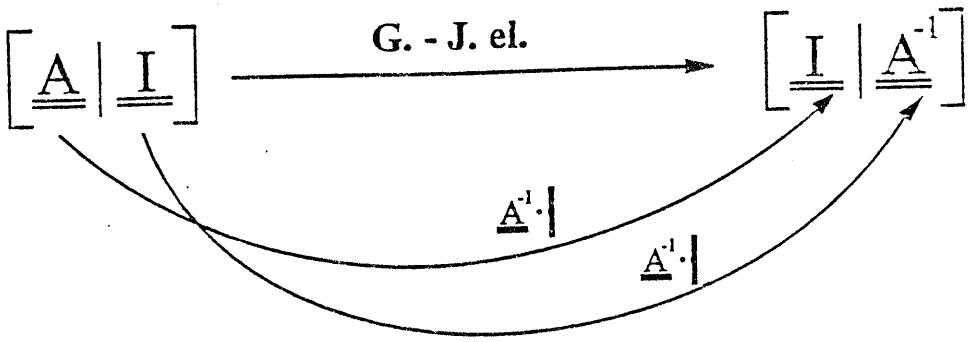
LINEÁRIS EGYENLETRENDSZER MEGOLDÁSA MÁTRIXEGYENLET RENDEZÉSÉVEL :

$$\underline{\underline{A}} \cdot \underline{x} = \underline{b} \xrightarrow[(\underline{\underline{A}}^{-1} \cdot \underline{\underline{A}} = \underline{\underline{I}})]{\underline{\underline{A}}^{-1} \cdot \|} \underline{x} = \underline{\underline{A}}^{-1} \cdot \underline{b}$$

Az algoritmus szerkesztését előkészítő megfigyelések:

Az előbbi összehasonlító elemzés mutatja, hogy ugyanaz az algoritmus (amit az első részben mint lineáris egyenletrendszerek megoldási algoritmusát dolgoztunk ki) mátrix invertálására is használható. Ilyenkor az algoritmus INPUT-ja az $[\,\underline{A} \mid \underline{I}\,]$ kombinált mátrix, a keresett inverz mátrixot pedig az OUTPUT-ként kapott $[\,\underline{I} \mid \underline{A}^{-1}\,]$ kombinált mátrix második tömbrészében találjuk. (Adott esetben a 'rendkívüli eset' itt azt jelzi, hogy az $\underline{A}$ mátrix nem invertálható.)

(2) **GAUSS - JORDAN ELIMINATIO MÁTRIX INVERTÁLÁSÁRA**



Példa

$$[\,\underline{A}|\underline{I}\,] = \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ -3 & 1 & -2 & 0 & 1 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \|\cdot(3) \\ \\ \| \end{array} \quad \|\cdot(-1)$$

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 7 & 7 & 3 & 1 & 0 \\ 0 & -3 & -2 & -1 & 0 & 1 \end{array}\right] \|\cdot\tfrac{1}{7}$$

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & \frac{3}{7} & \frac{1}{7} & 0 \\ 0 & -3 & -2 & -1 & 0 & 1 \end{array}\right] \|\cdot(3)$$

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & \frac{3}{7} & \frac{1}{7} & 0 \\ 0 & 0 & 1 & \frac{2}{7} & \frac{3}{7} & 1 \end{array}\right] \quad \| \cdot (-2)$$

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 1 & \frac{1}{7} & -\frac{2}{7} & 0 \\ 0 & 1 & 1 & \frac{3}{7} & \frac{1}{7} & 0 \\ 0 & 0 & 1 & \frac{2}{7} & \frac{3}{7} & 1 \end{array}\right] \quad \| \cdot (-1)$$

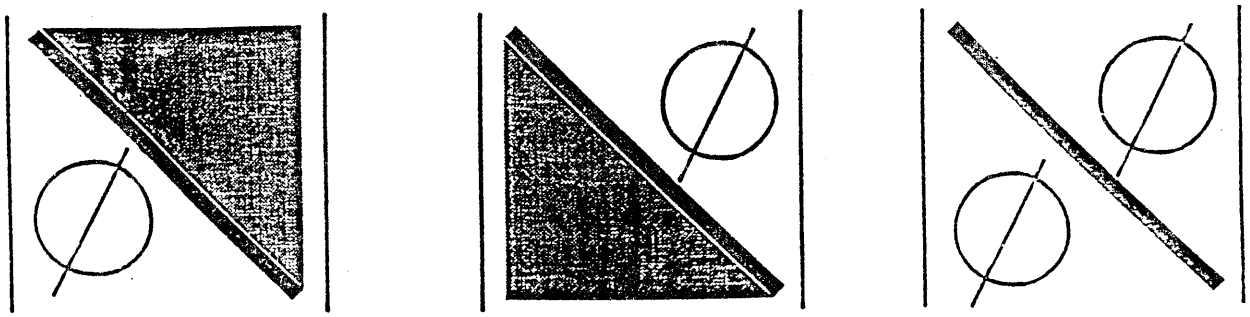
$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{1}{7} & -\frac{5}{7} & -1 \\ 0 & 1 & 0 & \frac{1}{7} & -\frac{2}{7} & -1 \\ 0 & 0 & 1 & \frac{2}{7} & \frac{3}{7} & 1 \end{array}\right] = \left[\underline{\underline{I}} \,|\, \underline{\underline{A}}^{-1}\right]$$

$$\underline{\underline{A}} \cdot \underline{\underline{A}}^{-1} = \underline{\underline{I}}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ -3 & 1 & -2 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} -\frac{1}{7} & -\frac{5}{7} & -1 \\ \frac{1}{7} & -\frac{2}{7} & -1 \\ \frac{2}{7} & \frac{3}{7} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. Probléma-elemzés és matematikai modell determináns kifejtéséhez:

Egy háromszögmátrix determinánsa kiszámítható a főátlóbeli elemeinek szorzataként. Ugyanez igaz nem csak felső vagy alsó háromszögmátrixoknál, de bármely diagonálmátrix esetén is.



Ilyen speciális mátrixokhoz juthatunk, ha egy kvadratikus mátrixot a Gauss(-Jordan) elimináció procedurájának vetünk alá. Most azonban, mátrixok helyett azok determinánsáról lévén szó, az eliminálás során figyelembe kell vennünk a megfelelő determináns-tulajdonságokat. Az egyes elemi operációk ugyanis módosíthatják a determináns értékét. Az alábbiakban összefoglaljuk, hogy a kvadratikus mátrixon végzett egyes elemi műveletek miként módosítják determinánsának értékét:

Ha az elemi művelet

(i) két sor cseréje;
(ii) egy sor végigszorzása $\lambda \in R$ valós számmal;
(iii) egy sor valahányszorosának hozzáadása egy másik sorhoz ;

akkor

(i) $\det(\underline{A}') = -\det(\underline{A})$
(ii) $\det(\underline{A}') = \lambda \cdot \det(\underline{A})$
(iii) $\det(\underline{A}') = \det(\underline{A})$

Az algoritmus-szerkesztést előkészítő megfigyelések:

Az alábbiakban megoldunk egy példát abból a célból, hogy megfigyelhessük mindazokat a lehetséges átalakításokat, amelyek révén egy adott kvadratikus mátrixot a kívánt speciális alakba redukálunk, párhuzamosan végrehajtva a determináns értékére vonatkozó szükséges korrekciókat.

Példa

$$\underline{\underline{A}} = \begin{bmatrix} 2 & 4 & -8 \\ 1 & 0 & -1 \\ -1 & 1 & 1 \end{bmatrix}$$

$$\det(\underline{\underline{A}}) = \begin{vmatrix} 2 & 4 & -8 \\ 1 & 0 & -1 \\ -1 & 1 & 1 \end{vmatrix}$$

két sor cseréjének hatására a determináns előjelet vált

$$\det(\underline{\underline{A}}) = - \begin{vmatrix} 1 & 0 & -1 \\ 2 & 4 & -8 \\ -1 & 1 & 1 \end{vmatrix}$$

a második sorból kiemelhető a 2-es közös osztó

$$\det(\underline{\underline{A}}) = (-2) \cdot \begin{vmatrix} 1 & 0 & -1 \\ 1 & 2 & -4 \\ -1 & 1 & 1 \end{vmatrix}$$

$\| \cdot (-1)$ $\quad$ $\| \cdot (1)$

az első sor (-1)-szeresének hozzáadása a másodikhoz és magának az első sornak a hozzáadása a harmadikhoz nem befolyásolják a determináns értékét

$$\det(\underline{\underline{A}}) = (-2) \cdot \begin{vmatrix} 1 & 0 & -1 \\ 0 & 2 & -3 \\ 0 & 1 & 0 \end{vmatrix}$$

a két sor cseréje ismét a determináns előjelváltását eredményezi

$$\det(\underline{\underline{A}}) = (+2)\cdot \begin{vmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 2 & -3 \end{vmatrix} \quad \| \cdot(-2)$$

a második sor (-2)-szeresének
hozzáadása a harmadikhoz
nem módosítja a determináns értékét

$$\det(\underline{\underline{A}}) = (+2)\cdot \begin{vmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & -3 \end{vmatrix} =$$

fejtsük ki a determinánst
első oszlopa szerint

$$= (+2)\cdot \left\{ 1\cdot \begin{vmatrix} 1 & 0 \\ 0 & -3 \end{vmatrix} - 0\cdot \begin{vmatrix} 0 & -1 \\ 0 & -3 \end{vmatrix} + 0\cdot \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix} \right\} =$$

$$= (+2)\cdot 1\cdot 1\cdot(-3) = \underline{-6 = \det(\underline{\underline{A}})}$$

Tehát egy háromszögmátrix determinánsa valóban megkapható a főátlóbeli elemeinek szorzataként.

A 'sor-redukció'-ra irányuló eliminálási procedúra már a háromszög-forma fázisánál befejezhető (miként e bemutató példánál tettük is); persze lehetne folytatni is, amíg a diagonál alakot el nem érjük. Noha a determináns kényelmes kiszámításához elegendő a háromszög-forma, most rendelkezésünkre áll a kész Gauss-Jordan algoritmus, amely az utóbbi - teljesen redukált - verziónak felel meg.

Az előző példa megoldási folyamatát tanulmányozva láthatjuk, hogy szinte ugyanaz az (első és második részben tárgyalt) algoritmus determináns kifejtésére is használható.

Eredeti algoritmusunkat milyen apró módosításokkal, kiegészítésekkel tehetnénk alkalmassá determináns-kifejtésre? (A választ az Olvasóra bízzuk.)

Érdemes megjegyezni, hogy ebben a harmadik alkalmazásban a 'rendkívüli eset' azt jelenti, hogy a determináns értéke zérus.

5. Algoritmikus szemléletű oktatás

Mivel a kérdéskör immár közel négy évtizedes múltra tekint vissza, néhány mondatot áldozhatunk keletkezése körülményeire. Az 1950-es években az USA-ban mint üstökös indult pályájára a programozott oktatás. Elméleti alapját a Skinner-i kis lépésekből álló információ →kérdés →válasz →megerősítés (illetve korrekció) tanuláspszichológiai "képlet" jelentette, melyeket az erre a célra szerkesztett speciális tankönyvek, illetve oktatógépek továbbítottak. A tanulók részére jelentős sikerélményt biztosító önálló tanulási módszer gyengeségei (nem teszi próbára az értelmi erőket: a gondolkodást, a problémamegoldást nem fejleszti a kívánt mértékben) igen hamar felszínre kerültek. A módszer vitatói, helyesebben továbbfejlesztői közül Crowder elágazó programjaiban a lépések már nagyobb terjedelműek, egy-egy gondolati egységet fognak át, és az előre megadott feleletek közül történő választástól függően a tanuló más-más úton halad tovább a program feldolgozásában.

Landa (L.N.Landa: Algoritmizálás az oktatásban; Tankönyvkiadó, Budapest 1969) algoritmusok alkalmazásával vélte az oktatás, tanulás hatékonyságát fokozni. Ezek elsajátításával a tanuló olyan megismerési, ismeretszerzési műveletek birtokába jut, amelyek elősegítik az eredményes, önálló tanulást, a logikus gondolkodás fejlődését. Két fő formája, a felismerési és a megoldási algoritmus vált ismertté.

Az algoritmusok további finomítását, flexibilisebbé tételét (humanizálását, optimalizálását) szolgálták azok az öt éven át tartó kísérletek, melyek eredményei egyértelművé tették, hogy "A humanizált megtanulandó algoritmusok megkönnyíthetik a megoldási (átalakítási) folyamatok elsajátítását azokban a tárgykörökben, melyek alkalmasak és célszerűek algoritmikus előírások készítésére; az algoritmikus előírások transzferhatása a tárgykört képező tantárgyon belül érvényes, más tárgyakra esetleges; a humanizált fogalom általánosabb értelmű és magában foglalja az algoritmusok felhasználásának módszerét is, mely a 'konstruktív' módszerrel analóg."[4]

Mind a felismerési, mind a megoldási algoritmusok számos, igen eltérő diszciplínához tartozó témakörben készültek. Matematikából: 19, nyelvtanból: 10, fizikából: 3, történelemből: 1, földrajzból: 3, kémiából: 7, számvitelből: 2, testnevelés (és sportból): 10, műszaki témákból: 16, kereskedelmi gyakorlatból: 1 és számítástechnikai ismeretekből: 2 oktatási algoritmust regisztrált egy 1974-ben készült felmérés. Azóta még inkább szélesedik ezek felhasználási területe (egészségügy, diagnosztika, rutin terápia; műszaki berendezések különböző szintű szervizelése stb.)[5]

5.1. Algoritmikus szemléletű tanítási-tanulási folyamat

"Modell és algoritmus együttoktatására a hallgatóság jelentős részének szüksége is van: mindazoknak, akik - bármely okból - a különböző elnevezésű tantárgyakban tanultakat saját munkájukkal integrálni nem tudják. A tananyagnak - a képzési cél szempontjából - túlzott tantárgyi szétaprózása a ma már nélkülözhetetlen rendszerszemlélet kialakítását akadályozza."[6]

---

[4] Dr. Gyaraki F. Frigyes: *Algoritmusok és algoritmikus előírások didaktikai felhasználásának és optimalizálásának lehetőségei,*
Kandidátusi értekezés tézisei, Budapest, 1976. 15.p.

[5] Dr.Gyaraki F. Frigyes - Dr.Biszterszky Elemér: *Didaktikai tanulmányok gyűjteménye;* Műegyetem Kiadó, Budapest, 1996.

[6] Dr.Fekete István: *Matematika és számítástechnika integrált oktatása,*
Kandidátusi értekezés tézisei, Budapest, 1986. 5.p.

Az algoritmikus szemléletmód kialakítása, az algoritmikus gondolkodásra nevelés fokozza általában az oktatás hatékonyságát, ugyanakkor az informatika tantárgycsoportban tanított programozást is segíti.

Nem a számítástechnika és az informatikával kapcsolatos tantárgyak jelentik az egyedüli lehetőséget, hogy a tanulókban kialakítsuk az algoritmikus szemléletmódot. Tapasztalatok mutatják, hogy a számítástechnika használata más tantárgyakban mindkét oldal lehetőségeit megsokszorozó, termékenyítő kölcsönhatást eredményez. Egy adott feladattípus általános megoldásához azt a 'probléma-modell-algoritmus' utat kell bejárnunk, amely a számítástechnikában a programozásnál nyilván nem kerülhető ki. A nem számítógépes megoldás során azonban sokszor sajnos nem így járunk el, jóllehet a feladatmegoldásnak ez a menetrendje biztosítja a komplex áttekintést.

5.2. Párhuzam a tanári és a programozói tevékenység között

Fejezetünk kiindulópontjául Hámori professzor állítását idézzük, miszerint "A számítástechnikai eszközök (számítógépek) tanítási-tanulási folyamatban betöltött funkcióinak (alkalmazásának) vizsgálata a tanulási folyamat elemzésén alapszik. A tanulás és tanítás komplex tevékenységek, melyek cselekvések, műveletek sorozatára bonthatók."[7] A következőkben a tanári és a programozói tevékenységet hasonlítjuk össze, előkészítve ezzel az algoritmikus szemlélet oktatási folyamatra gyakorolt hatásainak, módszertani vonzatainak vizsgálatát. Mindezek eredményeként válik majd világossá, hogy az oktatási gyakorlatban felmerülő bármely feladatnál algoritmikus szemléletű megoldást célszerű a tanulóknak bemutatni, illetve a tanulók önálló tevékenysége során elvárni - függetlenül attól, hogy számítógéppel vagy anélkül kell a feladatot megoldani. A tanárképzés vonatkozásában különösen is fontos, hogy tanárjelöltjeinkben kialakítsuk ezt a szemléletet, felkészítsük őket az algoritmikus feladatmegoldás tanítására. Megfigyelések bizonyítják, hogy a tanulók észjárásában akárhányszor tanáraik gondolkodásmódja is felfedezhető, az algoritmikus szemléletmód így nyilván továbbadható.

---

[7] Dr.Hámori Miklós: *Tanulás és tanítás számítógéppel*, Kandidátusi értekezés tézisei, Budapest, 1984., 5.p.

A leendő tanárok számára - a választott szaktól függetlenül - egyébként is alapvető fontosságú, hogy megfelelő programozási ismeretekkel és az algoritmikus gondolkodás képességével is rendelkezzenek. Gondoljunk csak arra, hogy amikor számítógéppel szeretnénk megoldani egy feladatot, valójában a számítógépes program megírása azt jelenti, hogy a programozó "megtanítja a számítógépet" az adott feladattípus megoldására. A megoldás algoritmusa ilyen értelemben "tanítási terv" eredményeképpen születik, amiből következik, hogy a programozó mindjárt egyfajta tanári tevékenységet is gyakorol, miközben persze következetesen elvárják tőle "a tanítvány" képességeinek - lehetőségeinek figyelembevételét. Érdekes lehet még a folyamat két fő fázisának analógiájára is rávilágítani:

általános fölkészülés - programnyelvtől független algoritmus-szerkesztés

adott osztályhoz való alkalmazkodás - konkrét programnyelvre adaptálás

A hasonlat megint nem csak, hogy jól mutatja, de vitathatatlanná teszi a két fázis elkülönülését és sorrendiségét. (Ámbár igaz, hogy a programozó számára az "alkalmazkodás" többé-kevésbé állandó, míg a tanár számára inkább változó feltételek figyelembevételét jelenti.) Mindenesetre a programozói tevékenység tanítással való kapcsolatát vizsgálva eszünkbe jut a "docendo discimus" elve, amit Joseph Joubert (1754 - 1824), a latin szellemet újjáéleszteni szándékozó francia klasszicista gondolkodó így fogalmaz meg: "Aki tanít, az kétszeresen tanul.". Valóban - nem csak, hogy tudnunk, de sokkal mélyebben elmerülve, elemezgetve-boncolgatva kell ismernünk azt, aminek a tanítására vállalkozunk. Így készülve mások tanítására, minden alkalommal magunkat is óhatatlanul tanítjuk, vagyis tanulunk.

5.3. A mintapéldák szerepe az algoritmizálás tanítási-tanulási folyamatában

Az algoritmizálás tanítási-tanulási folyamatát elemezve világossá válik, hogy elsősorban nem kész algoritmusokat kell tanítanunk, hanem sokkal inkább azt, hogyan születik egy algoritmus. "A megtanulandó algoritmusoknak főleg csak akkor van értelmük és helyük a tanítás-tanulás folyamatában, amennyiben ezek a legmagasabb szinten (lehetőleg az automatizmusok szintjén) jelennek

meg."[8] A "szorzótáblát" persze majd csukott szemmel is tudni kell: 3x8 = 24 (gondolkodás nélkül!); ez persze nem azt jelenti, hogy azelőtt valamikor ne mutatták (és értettük) volna meg, hogy a 3x8 azt jelenti, hogy 8+8+8 - és ez persze 24. A megoldandó feladat legyen életközeli. Az algoritmus szerkesztését minden esetben előzze meg a gyakorlati probléma elemzése és ennek eredményeképpen a modell-alkotás.

Tisztázni kell, milyen adatok állnak rendelkezésre (INPUT) és milyen eredményeket várunk (OUTPUT). Az algoritmus azt a pontos menetrendet (útmutatást) jelenti, amelyet követve eljutunk a célhoz, azaz a feladat megoldásához. Ahhoz, hogy ezt összeállítsuk, nekünk magunknak tudnunk kell megoldani az adott feladatot. Lényegében azt a magunknak feltett kérdést kell megválaszolnunk: "Én hogyan csinálnám?" . Ugyanakkor persze - számítógépes megoldás esetén - mindjárt gondolnunk kell arra is, hogy a gép sokkal elemibb műveleteket értelmez és végez, mint az ember - a teendőket ennek megfelelően kell lebontanunk számára.

Az algoritmus egyes lépéseinek feltárásába - folyamatosan megfelelő segítséget adva - a tanulókat is bevonhatjuk, ezzel biztosítva a fokozatosságot az önállóságra nevelésben. Nagy előny, hogy a közösen megoldott feladatot a tanuló így "sajátjának érzi", ugyanakkor ez veszélyt is jelenthet: a legtöbb tanuló úgy vélekedik, hogy ha valamit megértett, azt tudja is. Rá kell ébreszteni őket, hogy a megértés csak szükséges, de nem elégséges feltétele a tudásnak.

Vegyük egyelőre csak a reproduktív tudást. Ezt a legkönnyebb ellenőrizni: tudja-e üres papíron reprodukálni a korábban közösen már megalkotott (és megértett) megoldást? Ha sikerül rávenni őket erre az önellenőrzésre, azt fogják tapasztalni, hogy a megértés nem feltétlenül tudás. Mi a teendő ilyenkor? A helyes megoldás újbóli áttanulmányozása, amelynek során még az is kiderülhet, hogy bizonyos pontokon a megértés sem volt megalapozott. Ezeket tisztázva érezheti ismét úgy, hogy most már tudja, de ezt ismét ellenőrizni kell. Az imént vázolt "ciklusmagot" kell végrehajtani újra és újra, amíg a "kilépési feltétel" (hibátlan reprodukálás) nem teljesül. Hogyan is lehetne valaki sikeres egy ismeretlen feladat megoldásában, ha nehézségei vannak az ismert reprodukálásában? Sajnos mi tanárok is hibásak vagyunk abban, hogy alulértékelve a reproduktív tudást, engedjük kihagyni a tanítás-tanulás folyamatának ezt az alapvetően fontos láncszemét.

Nagy ellentmondás ez, hiszen ugyanakkor a definíciókat, tételeket precízen megköveteljük, tehát valójában nem tagadjuk a reproduktív tanulás szükségességét. Idegen nyelv tanításakor - tanulásakor a szavak, nyelv-

---

[8] Dr.Gyaraki F. Frigyes: Algoritmusok és algoritmikus előírások didaktikai felhasználásának és optimalizálásának lehetőségei
Kandidátusi értekezés. 1976. 20 p.

tani szabályok elsajátítását mindenki fontosnak tartja, de mi a helyzet a memoriterekkel? Ma mintha elfelejtkeznénk arról, hogy a kötött szövegek szó szerinti (ugyanakkor persze minden részletében akár elemekre szétbontva is tudatosan értett) megtanulásával nem egyszerűen csak a nyelvtudáshoz szükséges kellékekre teszünk szert, hanem annak komplex megjelenési formáira tárolunk el olyan mintákat, amelyeket részben vagy egészben, sőt éppen a komponálási minták vezérlésével már akár átalakítva, bizonyos értelemben újat alkotva hasznosíthatunk. A motiváció is biztosított, ha jól megválasztott, a mindennapi életben jól hasznosítható anyagokat választunk ki erre a célra. Bárki kipróbálhatja, hogy ha egy témakörben (tudatosan, részleteiben is értve) idegen nyelven megtanul 10-20 eredeti mondatot, akkor amire ezeket folyékonyan (automatikusan) tudja mondani egyszersmind képes lesz arra is, hogy ha nem is annyira folyékonyan, de szinte tetszés szerinti átalakított variációkban a saját megfogalmazásában is gyakorlatilag hibátlanul tudja interpretálni a kérdéses témát az adott idegen nyelven. Ez óriási dolog, hiszen ez implicite azt is jelenti, hogy az adott témakörben saját gondolatainak kifejtésére is képes. Esetünkben a jól megválasztott mintapéldák tölthetik be ilyen "memoriterek" szerepét. A sakkjátékban rejlő kimeríthetetlen szellemi lehetőségeket aligha vitatná bárki, mégis ki állítaná, hogy a sakk-lépések elsajátítása vagy az alap-stratégiák reproduktív megtanulása elhagyható, netán szégyellni való lenne?

5.4. Az algoritmikus feladatmegoldás tanításának néhány kognitív pszichológiai vonatkozása

5.4.1. Produktív és reproduktív gondolkodás

A Gestalt-pszichológusok (az alaklélektan képviselői) szerint a gondolkodó ember "átlátja" a probléma szerkezetét és a megoldás érdekében "újrastrukturálja" azokat.[9] Az elmélet főbb megállapításai:

- a problémamegoldó gondolkodás egyszerre produktív és reproduktív;
- a reproduktív problémamegoldás a korábbi tapasztalatok újrahasznosítása;
- a produktív problémamegoldás lényegében a probléma szerkezetének átlátása és újrastrukturálása;
  (az ún. funkcionális rögzítettség az átlátást hátráltató tényező lehet)
- a probléma szerkezetének átlátása rendszerint hirtelen következik be.

[9] Eysenck, M.W. & Keane, M.T.: Kognitív pszichológia
Nemzeti Tankönyvkiadó, Budapest, 1997. 603 p.

A kognitív teljesítménnyel kapcsolatban megállapítható, hogy mindenekelőtt a *tapasztalat* az, amelynek révén az ember egyre jobban, egyre nagyobb szakértelemmel tud dolgokat megcsinálni, válik szakértővé. A közelmúltban számos kísérlet irányult arra, hogy a problémamegoldás kognitív elméleteit (pl. problématér-elméletek) az emberi szakértelemre is kiterjesszék. A sakkozás, fizikai problémák megoldása és a számítógépes programozás területén végzett kutatások eredményeként számítógépes modellekkel támogatott elméletek is születtek. J.R.Anderson például a Gondolkodás Adaptív Vezérlése (GAV) kognitív architektúrájának részeként a készségek elsajátítására vonatkozóan dolgozott ki a szakértelem sokféle területén alkalmazható általános elméletet.
Ezeknek a szakértelemmel foglalkozó kutatásoknak jelentős része a többé-kevésbé ismerős problémák megoldásának mikéntjét vizsgálja, ahol valójában a sematikus tudás kerül közvetlen alkalmazásra.

5.4.2. Analogikus problémamegoldás és kreativitás

A kreativitással foglalkozó kutatások arra a kérdésre keresnek választ, hogyan vagyunk képesek - nyilván kreatív - megoldást produkálni olyan esetekben, amikor a problémával összefüggő ismerettel nem rendelkezünk?

Wallas klasszikusnak számító leírása szerint a kreatív folyamat fő fázisai:

- az *előkészület* fázisa:
  a probléma megfogalmazása, első próbálkozások a megoldásra;
- az *inkubáció* fázisa:
  a problémát félretesszük, más kérdésekkel foglalkozunk;
- az *illumináció* fázisa:
  váratlanul "beugrik" a megoldás;
- a *verifikáció* fázisa:
  a hirtelen belátott megoldás ellenőrzése.

Ez a séma sajnos nem sokat árul el a kreatív cselekvés mögöttes kognitív folyamatairól. Valószínűsíthető azonban, hogy az *analogikus gondolkodás* az a kulcs, amelynek révén a probléma megoldásával foglalkozó személy *indirekt* módon mégis csak korábbi tapasztalatokat használ fel. Koestler szerint a

kreativitás két nagyon is különböző elképzeléshalmaz analógiájából születik. Matematikus szemmel az analogikus gondolkodás "analógiája" egy ún. *analogikus leképezés*, amely az említett két elképzeléshalmaz között létesít hozzárendelést: egy adott gondolathalmaz (alaptartomány) fogalmi szerkezetét egy másikra (céltartományra) képezi le.

Az analogikus leképezés néhány jellemzője:

- az alaptartomány és a céltartomány fogalom-, ill. tulajdonság-elemeinek megfeleltetése;
- az alaptartomány bizonyos fogalom-, ill. tulajdonság-elemeit átvisszük a céltartományba és ezzel ott új fogalom-, ill. tulajdonság-szerkezeteket vezetünk be;
- általában komplex (koherens, integrált, nem töredékes) ismeretek kerülnek átvitelre (l. Gentner szisztematikusság-elvét);
- pragmatikus célszerűség nem komplex ismeret átvitelét is indokolhatja.

Rutherford a Naprendszer analógiáját használta az atom-szerkezet modellezésénél, de analógia az alapja Einstein fénysugár-meglovaglására vonatkozó gondolatkísérletének is. Az analogikus problémamegoldás kutatása lényegében azokat a kreatív tevékenységgel kapcsolatos körülményeket vizsgálja, amelyek között az emberek valójában analógiákat fedeznek fel valamely általuk már ismert, ill. a megoldásra váró ismeretlen probléma között.

A kreatív folyamat értelmezésében két fő irányzat ismeretes. Az egyik az újszerű produktum létrehozását lényegében véletlen eseménynek képzeli el, amelynek valószínűségét befolyásolja a tudásanyag nagysága (a rendelkezésre álló elemszám megszabja a lehetséges kombinációk mennyiségét és komplexitását), valamint az ötletáramlás sebessége. Az új kombinációk kialakítását szigorú értékelésnek kell követnie, amelynek során szembesülnek az ötletek a követelményekkel. A jelentős kreatív termék kiemelése eszerint szelektív működés eredménye. A másik irányzat szerint az az alkotás célirányos, amelyet kezdettől fogva korlátok közé szorít a kitűzött cél. Ez az értelmezés a kreativitást tulajdonképpen a problémamegoldással azonosítja.

Vajon egy algoritmus készítése melyik értelemben tekinthető kreatív folyamatnak? Véleményem szerint alapvetően az utóbbi értelemben, hiszen egy algoritmus megkomponálása természeténél fogva célirányos alkotásban valósul meg; azonban mégsem zárhatjuk ki az első, a véletlen eseménynek képzelt újszerű produktum létrehozását, amely ha adott esetben megszületik, a célirányos alkotásnak "csak" "melléktermé", még akkor is, ha alkalmasint ez

a "melléktermék" - objektív értékelés szerint (nem a célirányos alkotás szempontjából nézve) - értékesebbnek bizonyulhat, mint maga a célirányos alkotás. Ilyen esetben a véletlen eseményként született "melléktermék" mint igenis jelentős kreatív termék, pozitív szelekció tárgya lesz. Tehát egy algoritmus készítése elsősorban célirányos alkotás, amelyet kezdettől fogva korlátok közé szorít a kitűzött cél, akárhányszor előfordulhat azonban, hogy a megoldási algoritmus komponálásának folyamatában, a lépésről-lépésre finomítás során egy adott ponton a kitűzött céllal kimutatható kapcsolatban nem álló másik, - ún. véletlenszerű -, ám ugyanakkor újszerű produktum is születik.

A feladatmegoldás pszichológiai vonatkozásaival kapcsolatos vizsgálódásaink azt mutatják, hogy a problémamegoldás kognitív elméletei az algoritmikus feladatmegoldásra is kiterjeszthetők. Az algoritmikus szemléletű megoldás az analogikus problémamegoldás révén hozzájárul az algoritmikus szemlélet kiterjesztéséhez a tanítási-tanulási folyamat egészében.

Irodalom

Anderson, J.R.: The Adaptive Character of Thought
Hillsdale, Lawrence Erlbaum Associates
Cambridge, Harvard University Press, 1990.

Barthélemy, J.P., Cohen, G. & Lobstein, A.: Algorithmic Complexity
UCL Press, London, 1996. 256 p.

Benkő, A.P. - Bércesné Novák, Á. - Fialáné, Dér, Zs. - Hosszú, F. - Lukács, O. - Nagy Róbertné - Őri, I. - Pentelényi, P. - Szekér Istvánné : QuickBasic
(Szerk.: Lukács, O. & Pentelényi P.)
BDGMF jegyzet, 1990. 212 p.

Békési, A. - Brückner, H. - Dusza, Á. - Hámori, M. - Kovács, E. - Párizs, Gy. - Szűcs, E. - Varga, T.: Számítógéppel segített oktatás fejlesztési irányzatai
OMFB tanulmány, Budapest, 1982. 1-90 pp.

Bércesné Novák, Á. - Fialáné Dér, Zs. - Hosszú, F. - Lukács, O. - Nagyné Földvárszky, M. - Nagy Róbertné - Őri, I. - Pentelényi, P.: Matematika-számítástechnika szigorlat, Írásbeli feladatok gyűjteménye
(Szerk.: Hosszú, F. - Lukács, O.)
SZÁMALK, 1989. 162 p.

Biszterszky, E.: A programozott oktatás alkalmazási lehetőségei és kísérleti fejlesztése a műszaki felsőfokú intézményekben
Kandidátusi értekezés tézisei, 1981. 15 p.

Biszterszky, E. - Gyaraki, F. F.: Didaktikai tanulmányok gyűjteménye
Műegyetem Kiadó, Budapest, 1996.

Bódy, B. - Fekete, I. - Pallinger, F. - Schubert, T. - Zsigmond, A.: Számítógép-programozási útmutató és példatár
Műszaki Könyvkiadó, Budapest, 1981. 602 p.

Brückner, H. - Lukács, O. : A folyamatábráktól a programozásig
Tankönyvkiadó, Budapest, 1987. 356 p.

Buzan, T.: Use Your Head
BBC Books, 1996. 154 p.

Cotton, J.: The Theory of Learning Strategies
Kogan Page, London, 1995. 156 p.

Dahl, O.J. - Dijkstra, E.W. - Hoare, C.A.R.: Structured Programming
Academic Press, London & New York, 1972. 220 p.

Dahl, O.J. - Dijkstra, E.W. - Hoare, C.A.R.: Strukturált programozás
Műszaki Könyvkiadó, Budapest,1978. 203 p.

Dijkstra, E.W.: A Discipline of Programming
Prentice-Hall, New York, 1976.

Eysenck, M.W. & Keane, M.T.: Cognitive Psychology
Psychology Press, UK, 1997. 542 p.

Eysenck, M.W. & Keane, M.T.: Kognitív pszichológia
Nemzeti Tankönyvkiadó, Budapest, 1997. 603 p.

Fekete, I.: Számítástechnika
Műszaki Könyvkiadó, Budapest, 1979.

Fekete, I. : Matematika és számítástechnika I., II.
Műszaki Könyvkiadó, Budapest, 1986. I. 205 p., II. 300 p.

Fekete, I.: Matematika és számítástechnika integrált oktatása
Kandidátusi értekezés tézisei, 1986. 7 p.

Fercsik, J.: Programozott technikai berendezések alkalmazásának kísérleti vizsgálata a műszaki felsőoktatásban
Kandidátusi értekezés tézisei, 1973.

Fialáné Dér, Zs. - Hosszú, F. - Nagyné Földvárszki, M. - Nagy Róbertné - Pentelényi, P. - Rudas, I.:
Matematika és számítástechnika útmutató és példatár I., (Rudas, I. szerk.)
Műszaki Könyvkiadó, Budapest, 1987. 514 p.

Fialáné Dér, Zs. - Őri, I. - Pentelényi, P.: Mathematics in English I.
BDMF jegyzet, 1992. 127 p.

Gentner, D.: The Mechanisms of Analogical Reasoning
In Similarity and Analogical Reasoning, eds. Vosniadou, S. & Ortony, A.
Cambridge University Press, 1989.

Green, T.R.: Programming Languages as Information Structures
Academic Press Ltd., London, 1990.

Gries, D.: The Science of Programming
Springer-Verlag, New York, 1981. 366 p.

Griffiths, M. & Tagg, E.D.: The Role of Programming in Teaching Informatics
Elsevier Science Publishers B.V., Amsterdam, 1991. 212 p.

Gyaraki, F. F.: A megtanulandó algoritmusok szerepe matematikai korrepetáló programok készítésénél
Audio-vizuális Közlemények, 1967/5. 36-47. pp.

Gyaraki, F. F.: Algoritmusok a matematika tanításában
A matematika tanítása, 1969. január, XVI. évf. 1. szám, 13-18. pp.

Gyaraki, F. F.: Algoritmusok és algoritmikus előírások didaktikai felhasználásának és optimalizálásának lehetőségei
Kandidátusi értekezés, 1976. 20 p.

Hámori, M.: Tanulás és tanítás számítógéppel
Tankönyvkiadó, Budapest, 1983. 239 p.

Hámori, M.: Tanulás és tanítás számítógéppel
Kandidátusi értekezés tézisei, 1984. 13 p.

Harel, D.: Algorithmics the Spririt of Computing
Addison-Wesley, Harlow, England, 1996. 476 p.

Hetényi Pálné (szerk.): Számítástechnika középfokon
OMIKK, Budapest, 1987. 275 p.

Hoc, J.M., Green, T.R.G., Samurcay, R. & Gilmore, D.J.:
Psychology of Programming
Academic Press, London, 1990. 290 p.

Hoyles, C. & Sutherland, R.: Logo Mathematics in the Classroom
Routledge, London, 1989. 242 p.

Knuth, D.E.: The Art of Computer Programming
Addison-Wesley, Harlow, England, 1997. 650 p.

Koster, C.H.A.: Programozás felülnézetben
Műszaki Könyvkiadó, Budapest, 1988. 267 p.

Landa, L.N.: Algoritmizálás az oktatásban
Tankönyvkiadó, Budapest, 1969.

Lénárd, F.: A problémamegoldó gondolkodás
Akadémiai Kiadó, Budapest, 1964.

Lukács, O. - Pentelényi, P.: Ipari feladatok matematikai statisztikai megoldása személyi számítógéppel
Gépgyártástechnológia, XXVI. évf., december, 1986. 541.p.

Lukács, O. - Pentelényi, P.: Mathematics in English III.
BDMF jegyzet, 1993. 229 p.

McCormick, C.B. & Pressley, M.: Educational Psychology
Longman, Harlow, England, 1997. 522 p.

McIntyre, P.: Teaching Structured Programming in the Secondary Schools
Krieger Publishing Company, Malabar, Florida, 1991. 222 p.

Pentelényi, P.: Kiegészítés a Matematika és Számítástechnika c. tárgy előadásaihoz
Budapest, 1984. BDGMF. 136 p.

Pentelényi, P.: A számítástechnika oktatásával kapcsolatos néhány didaktikai kérdés
A műszaki főiskolák matematika, fizika és számítástechnika oktatóinak
X. országos tanácskozása
Budapest, 1986. BDGMF. 13.p.

Pentelényi, P.: Módszertani kísérletek a Matematika és Számítástechnika c. tárgy esti tagozaton történő oktatásában
Jubileumi tudományos ülésszak
Budapest, 1989. BDGMF. 117-119.p.

Pentelényi P.: Módszertani gondolatok az "Integrálási szabályok" tanításáról
Főiskolák matematika, fizika és számítástechnika oktatóinak XVI. országos konferenciája
Szombathely, 1992. Berzsenyi Dániel Tanárképző Főiskola. 11.p.

Pentelényi, P.: The Possibilities of Forming and Developing Algorithmic Thinking when Teaching Various Subjects
Engineering Education World Congress, Wien-Budapest, 1996.
in Education by Communication, ed. Melezinek, A. & Kiss, I. 525-528.p.

Pentelényi, P.: Segédanyag az Informatika Módszertan c. tantárgyhoz
BDMF elektronikus jegyzet, 1996. 84502 l., 1.09 MB.

Pentelényi, P.: Tanítható-e a problémamegoldás?
Gondolatok az algoritmikus feladatmegoldás tanításáról és az algoritmus-szemléltetésről
Szakoktatás, XLVII. évf. március, 1997. 26-27.p.

Pentelényi, P.: Algoritmikus szemléletű oktatás
Gép, IL. évf. szeptember, 1997. 53-55.p.

Pentelényi, P.: Some Historical and Professional Linguistic Aspects of Teaching
Algorithmic Problem Solving
BME, Szakképzéspedagógiai PhD-program
II. tudományos szemináriuma, 1997. 6 p.

Pentelényi, P.: Can Problem Solving be Taught?
Engineering Education '97, IGIP Symposium, Klagenfurt, 1997.
in Overinformed - Undereducated? , ed. Melezinek, A. 293-296.p.

Pentelényi, P.: Development of Algorithmic Thinking
Ligatura Ltd., Budapest, 1998. 253 p.

Pentelényi, P.: The Possibilities and Methods of the Formation and Development of Algorithmic Thinking in Technical Training
Theses of Doctoral Dissertation, Budapest, 1998., 10 p.

Pentelényi, P.: A mintapéldák szerepe az algoritmizálás tanítási-tanulási folyamatában
Szakképzési Szemle, XV.évf., 1999. 60-64.pp.

Pólya, G.: How to Solve it?
Princeton University Press, 1990. 253 p.

Ralston, A. & Neill, H.: Algorithms
Hodder Headline Plc., London, 1997. 186 p.

Robertson, L. A.: Simple Program Design
Nelson ITP, Cambridge, 1997. 203 p.

Shen, A.: Algorithms and Programming
Birkhauser, Basel, 1997. 217 p.

Skinner, B.F.: Science and Human Behavior
New York, Free Press, 1953.

Stone, R.G. & Cooke, D.J.: Program Construction
Cambridge University Press, Cambridge, 1990. 372 p.

Vígh, A. : Az iparoktatás története
Budapest, 1932.

Wallas, G.: The Art of Thought
New York, Harcourt Brace Jovanovich, 1926.

Wirth, N.: Algoritmusok + adatstruktúrák = programok
Műszaki Könyvkiadó, Budapest, 1982. 345 p.