

Eszterházy Károly Főiskola  
Matematikai és Informatikai Intézet

# Adatszerkezetek és algoritmusok

Geda Gábor

Eger, 2012

Készült a TÁMOP-4.1.2-08/1/A-2009-0038 támogatásával.



A projekt az Európai Unió  
támogatásával valósul meg.

# Tartalomjegyzék

<b>1. Előszó</b>	<b>4</b>
<b>2. Bevezetés</b>	<b>6</b>
2.1. Adatok . . . . .	10
2.2. Típusok . . . . .	11
2.3. Az adatszerkezetek osztályozása . . . . .	12
2.4. A vezérlés lehetőségei . . . . .	14
2.5. Fogalmak, jelölések . . . . .	17
2.5.1. Hatékonyság . . . . .	17
2.5.2. Folyamatábra . . . . .	18
2.5.3. Leírónyelv . . . . .	19
2.5.4. Vezérlési szerkezetek . . . . .	19
2.5.5. Szekvencia . . . . .	19
2.5.6. Szelekció . . . . .	19
2.5.7. Iteráció . . . . .	21
2.6. Feladatok . . . . .	24
<b>3. Alapalgoritmusok</b>	<b>31</b>
3.1. Sorozatszámítás . . . . .	31
3.1.1. Összegzés . . . . .	31
3.1.2. Megszámlálás . . . . .	32
3.1.3. Kiválogatás . . . . .	32
3.1.4. Minimum- és maximumkiválasztás . . . . .	33
3.2. Feladatok . . . . .	35
<b>4. Keresés, rendezés</b>	<b>45</b>
4.1. Sorozat . . . . .	45
4.2. Keresés sorozatokban . . . . .	49
4.2.1. Eldöntés . . . . .	49
4.2.2. Lineáris keresés . . . . .	50
4.2.3. Kiválasztás . . . . .	50
4.2.4. Strázsás keresés . . . . .	51

4.2.5.	Lineáris keresés rendezett sorozatban . . . . .	52
4.2.6.	Bináris keresés . . . . .	53
4.3.	Rendezés . . . . .	56
4.3.1.	Beszűrő rendezés . . . . .	67
4.3.2.	Shell rendezés . . . . .	70
4.3.3.	Összefuttatás . . . . .	74
4.4.	Feladatok . . . . .	76
<b>5.</b>	<b>Halmaz</b>	<b>80</b>
5.1.	Feladat . . . . .	82
<b>6.</b>	<b>A verem és a sor</b>	<b>83</b>
6.1.	A verem alkalmazásai . . . . .	83
6.1.1.	Posztfixes kifejezés kiértékelése . . . . .	83
6.1.2.	Infixes kifejezés ellenőrzése . . . . .	86
6.2.	A verem megvalósítása . . . . .	88
6.3.	Sor . . . . .	92
6.3.1.	Egyszerű sor . . . . .	92
6.3.2.	Léptető sor . . . . .	95
6.3.3.	Ciklikus sor . . . . .	97
6.4.	Feladatok . . . . .	100
<b>7.</b>	<b>Rekurzió</b>	<b>103</b>
7.1.	Rekurzív definíció – rekurzív algoritmus . . . . .	104
7.2.	Hanoi tornyai . . . . .	110
7.3.	Feladat . . . . .	114
<b>8.</b>	<b>Dinamikus adatszerkezetek</b>	<b>120</b>
8.1.	Lista . . . . .	122
8.2.	Feladat . . . . .	140
8.3.	Fa . . . . .	143
8.4.	Feladat . . . . .	145

<b>9. Visszalépéses keresés</b>	<b>147</b>
9.1. Feladat . . . . .	158
<b>10.Melléklet</b>	<b>160</b>
10.1. Közismert azonosítók és képzési szabályaik . . . . .	160
10.1.1. Személyi azonosítók képzése . . . . .	163
10.1.2. Adózó polgár adóazonosító jelének képzése . . . . .	164
10.1.3. Társadalombiztosítási azonosító jel képzése . . . . .	165
10.1.4. Vényazonosító . . . . .	165
10.1.5. Az ISBN (International Standard Book Number) . . .	167
10.1.6. Az ISSN (International Standard Serial Number) . . .	169
10.1.7. Bankkártyaszám . . . . .	170
10.1.8. EAN-13, EAN-8 (European Article Numbering) . . . .	171
<b>Irodalomjegyzék</b>	<b>173</b>

# 1. Előszó

Az informatika tudományának ezt, a többihez képest viszonylag állandónak mondható területét sokan, sokféle módon közelítették már meg. Számtalan igen értékes munka született. Az újabbak – különösen a hasonló, bevezető jellegűek – írását elsősorban nem is az ismeretanyag tartalmi változása teszi szükségessé, sokkal inkább a „célközönség” igényel más, újabb megközelítést.

Maga az algoritmus kifejezés a bagdadi arab tudós, al-Hvárizmi (Abu Dzsafar Muhammad bin Músza al-Hvárizmi, élt kb. 780-tól kb. 845-ig, Al-Khvorizmi, Al-Khorizmi stb.) nevének eltorzított, rosszul latinra fordított változatából ered. A időszámítás után 700-1200 között eltelt időszak az arab birodalmakban a kultúra, a tudomány virágzásának ideje volt. Ennek részben a mongol, részben a keresztény hódítások vetettek véget. Az arabok legnagyobbbrészt a hinduktól, Európa pedig al-Hvárizmitól és utódaitól vette át nemcsak a helyiértékes, tízes rendszerű számírást<sup>1</sup>, hanem az alapfokú algebrai és trigonometriai ismereteket is (szöveges egyenletek felírása, megoldása).

Az akkori idők egyik legnagyobb hatású műve a térségben, talán rögtön a Korán után, minden bizonnyal az al-Hvárizmi által írt Algebra (Al-kitab al-muktaszár fi-hiszáb al-dzsabr val-mukabala = Rövid könyv a helyrerakásról (al-dzsabr) és az összevonásról) volt. Az al-dzsabr szóból ered mai „algebra” szavunk. De al-Hvárizmi írt egy aritmetikai jellegű, a hindu tízes számrendszert ismertető könyvet is, ez csak latin fordításban maradt meg, címe így kezdődik: Dixit Algorithmi (Ezt mondja al-Hvárizmi:). Innen eredt a latin szó, ami aztán szétterjedt a többi európai nyelvben is. A 820 körül írt könyv eredetije eltűnt, a cím teljes latin fordítása a következő: „Liber Algorithmi de numero Indorum” (azaz „Algorithmus könyve az indiai számokról”). A hindu számírást ismertető könyvét az Al-Mamún kalifa (Harun ar-Rasid fia, lásd: Ezeregyéjszaka...) által épített bagdadi „Bölcsesség Házá”-ban írta. A könyvet Adelard bathi angol szerzetes fordította a XII. században, ebből a fordításból és egyéb arab eredetű forrásból ismerte meg Európa az új számírást. Az arab források miatt terjedt el az „arab számok” kifejezés, amely

<sup>1</sup> addig római számokkal illetve abakusszal, az ókor számológépével számoltak.

elfedi a hindu eredetet.

Az algoritmus a matematika és az informatika fontos fogalma. Az elméleti informatika egyes részterületei foglalkoznak velük, így az algoritmuselmélet, a bonyolultságelmélet és a kiszámíthatóságelmélet. Számítógépes programok is így vezérlik a számítógépeket.

Egy probléma megoldására irányuló, adott szinten elemi lépések sorozata akkor tekinthető algoritmusnak, ha van egy vele ekvivalens Turing-gép, ami minden megoldható bemenetre megáll.

Készült a TÁMOP-4.1.2-08/1/A-2009-0038 támogatásával..

## 2. Bevezetés

Az élet különböző területein felmerülő feladatok megoldására már a számítógépek megjelenése előtt is meg tudták adni olyan lépések sorozatát, amely elvezet az adott probléma megoldásához. Gondoljunk csak bele, hogy az Euklideszi algoritmus „utasításait” követve meg tudjuk határozni két egész szám legnagyobb közös osztóját, vagy már a kisiskolások is ismerik, hogyan tudják papíron összeadni, kivonni, szorozni vagy osztani egymással azokat a számokat, amelyekkel ezeket a műveleteket fejben nem képesek elvégezni. Geometria órán szintén megtanulják, hogy milyen lépések sorozata vezet el egy szakasz felező-merőlegeséhez vagy egy szög szögfelezőjéhez. Ilyen és ehhez hasonló tevékenységsorozatok elsajátíttatása hosszú idő óta célja az oktatásnak. Tehát az oktatás egyik fő célja a problémamegoldásra való föl-készítés. Összetettebb problémák megoldására is ezek révén a minták révén vagyunk képesek.

Tehát akkor, amikor a számítógépet a problémamegoldásban hívjuk segítségül, akkor „csupán” két dogot kell tenni.

1. Az adott feladat jellemzőit, a probléma leírását meg kell jelenítenünk a számítógépben.
2. A feladat megoldásához vezető lépéseket szintén a számítógéppel kell végrehajtatnunk.

Lényegében ezt nevezzük programkészítésnek. Ezt a folyamatot és annak bizonyos összefüggéseit szemlélteti a [2.1](#) ábra. Az ábrán jól elkülönítve látható a folyamat – tárgyalásunk szempontjából fontos – öt szakasza és azok közötti kapcsolatrendszer.

1. Specifikáció:

A feladat célkitűzéseit fogalmazza meg. Itt határozzuk meg pontosan, hogy milyen adatok állnak rendelkezésre és azokkal kapcsolatban milyen elvárásaink lehetnek (előfeltétel). Itt kell rögzíteni azt is, hogy az adatok feldolgozásától mit várunk, azaz milyen tulajdonságokkal ren-



delkező (utófeltétel) kimenetet szeretnénk<sup>2</sup>, azaz leírjuk a bemenet és a kimenet közötti összefüggést.

## 2. Tervezés:

Ennek a fázisnak a sikeréhez elengedhetetlen a pontos specifikáció. Itt határozzuk meg az algoritmusok tervezésével a bemenettől a kimenetig vezető „utat” és az adatszerkezetek kialakításával azokat az eszközöket, amelyekkel ezen az úton végig kívánunk menni. Adatszerkezetek és az algoritmusok tervezésének kölcsönös kapcsolatát jelzi a 2.1 ábrán a közöttük lévő nyíl, ugyanis az adatszerkezet megválasztása meghatározza a vele végezhető műveleteket és viszont, ha eldöntjük, hogy milyen algoritmust szeretnénk fölhasználni, gyakorlatilag leszűkítettük az adatszerkezetek körét is, ahonnan választhatunk az adataink ábrázolására. Természetesen a feladat jellege már jelentősen befolyásolja azt is, hogy milyen programozási nyelvet választhatunk. Ha azonban döntöttünk az algoritmusok és az adatszerkezetek vonatkozásában, akkor ezzel tovább szűkülhet a válaszható programozási nyelvek köre.

Az előzőekben szeretnénk volna érzékelteni, hogy a tervezési szakaszban hozott döntéseink mennyire meghatározóak lesznek a program és nem utolsó sorban munkavégzés minősége szempontjából is. A problémához rosszul illeszkedő adatstruktúra nagyon megbonyolíthatja az algoritmusainkat, de a valóban jól fölépített adatmodell lényegesen leegyszerűsítheti azt, ezzel hatékonyabbá téve a fejlesztők munkáját, de a későbbi program működését is. Tehát a tervezés fázisában az adatszerkezetek, a velük végezhető műveletek által meghatározott algoritmusok és az előző kettő implementálására szolgáló programozási nyelv összehangolt megválasztása döntő fontosságú a további munka sikere szempontjából.

## 3. Kódolás:

Ebben a szakaszban valósítjuk meg a korábbiakban megtervezett al-

<sup>2</sup> Természetesen fontos kérdés az is, hogy az rendelkezésre álló adatok birtokában, azok feldolgozásával egyáltalán a kívánt eredmény elérhető-e?

goritmust és adatszerkezetet a választott programozási nyelven. (Ha az előzőekben elég körültekintően jártunk el, akkor ez a nagyon fontos munka szinte mechanikusan is történhet.)

#### 4. Tesztelés, hibajavítás:

A munkának ebben a szakaszában ellenőrizzük, hogy a program megfelel-e a specifikációban foglaltaknak. A program összetettségétől függően több-kevesebb hibára mindig kell számítanunk, de kellően alapos tervezéssel elkerülhetők azok a súlyos hibák, amelyek a tervezési szakaszban gyökereznek és javításukhoz például az adatmodell módosítása, vagy az algoritmusok újra gondolása szükséges.

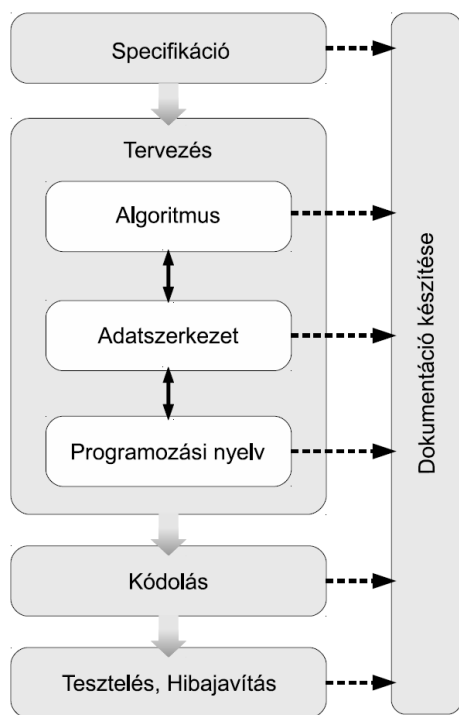
#### 5. Dokumentálás:

Ideális esetben ez a tevékenység végig kíséri a fejlesztés teljes folyamatát. Bár sok esetben, a munka során nem látjuk át a fontosságát, különösen a program utóélete vonatkozásában fontos lelkiismeretesen végeznünk.

A korábbiakban két nagyon fontos, jellegükben is eltérő kategóriáról volt szó. Mindkettővel kapcsolatban lehetnek elképzeléseink akár a hétköznapi tapasztalataink alapján is, hiszen az adatok – már a szóhasználat révén is – szinte hétköznapi fogalomnak számítanak, az algoritmusoknak pedig az a leegyszerűsített megfogalmazása, miszerint valamely probléma megoldására irányuló elemi lépések sorozataként adható meg, szintén közérthető, ennél fogva szintén megfelelő alapokat biztosít a további tárgyaláshoz.

Mindezek megvalósítására az évek során számtalan általánosan használható, vagy éppen speciális eszközt hoztak létre.

A mindennapi élet, vagy akár a tudomány különböző területein gyakran pótoljuk az „eredeti” valami „hasonmással”. Leegyszerűsítve és általánosítva a kettő között csupán az a különbség, hogy nem minden paraméterük azonos. Pontosabban, a „hasonmás” csak a számunkra fontos paramétereiben egyezik meg az „eredeti” paramétereivel (vagy csak közelíti meg azt), így csak bizonyos szempontból alkalmas az „eredeti” helyettesítésére. Például általában a kirakatban sem élő emberekre aggatják a ruhákat, holott nekik



2.1. ábra. A programkészítés lépései.

szeretnék eladni. Mindenesetre a kirakati bábok méretüket és alakjukat tekintve hasonlóak az emberhez, így a célnak megfelelnek. Bevett gyakorlat volt, hogy az embereknek szánt gyógyszereket állatkísérletekkel tesztelték mielőtt a gyógyászat gyakorlatába bevezették volna az alkalmazásukat. Ebben az esetben természetesen a formai különbözőség az elhanyagolható, és sokkal fontosabb a két szervezet (emberi és állati) működésbeli hasonlósága. Korábban, a különböző járművek tervezése során, például autók, hajók, repülőgépek áramlástani vizsgálataihoz elkészítették azok általában kicsinyített változatát. Ezek csak formájukat tekintve egyeztek meg az eredetivel, de méretbeli különbségen túl általában funkcionálisan is különböznek attól (például nincsen bennük motor). Ilyen esetekben azt mondjuk, hogy modelleket készítünk és alkalmazunk. Általában véve modellnek nevezzük tehát azokat a dolgokat, amelyek csak a számunkra fontos paramétereikben egyeznek meg a modellezni kívánt objektummal, jelenséggel.

A minket körülvevő világ legkülönbözőbb dolgairól jegyzünk meg illetve rögzítünk különféle számunkra fontos információkat, ugyanakkor másokat, amelyek nem szükségesek a számunkra – holott az adott dolog pontos jellemzéséhez azok is szükségesek – egyszerűen elhanyagolunk. Az ilyen módon összetartozó adatok összességét, amely adatok közötti logikai kapcsolatot pontosan az jelenti, hogy ugyanazt a dolgot jellemzik, adatmodellnek nevezzük. Az adott dolog adatmodellje tehát nem fogja tartalmazni az adott feladat szempontjából lényegtelen jellemzőkre vonatkozó adatokat.

A modell tehát a valós világ illetve annak egy részének absztrakciója révén jön létre. Az ilyen adatmodell megalkotása nagyon hasonlít a szöveges matematikai feladatok<sup>3</sup> esetében a megoldás első lépéseéhez, amikor az adatok rögzítése, jelölések bevezetése, és az egyes adatok közötti összefüggések keresése történik. Lényegében tehát itt is adatmodellt építünk és az adatmodellünkkel különböző műveleteket végzünk, hogy eljussunk a megoldáshoz. Az elvonatkoztatások során begyakorolt sablonok alkalmazása a későbbiekben segít minket olyan feladatok megoldásában, amilyenekkel korábban nem is találkoztunk. Valójában az algoritmizálás során is hasonló „szöveges feladatokat” kell megoldanunk, de a helyzet annyival bonyolultabb, hogy produkálnunk kell még az adatmodell és a megoldás menetének egy viszonylag kötött eszközrendszer segítségével történő leírását is a számítógéphez közeli formában.

## 2.1. Adatok

A legkülönbözőbb dolgok és jelenségek rendszerként való megközelítése egy a tudományterületektől független szemléletmód, amelynek kialakítására az informatika oktatása – ezen belül az algoritmizálás és a programozás tanítása – során különösen jó lehetőségek kínálkoznak. Ez a látásmód a legkülönbözőbb területeken kamatoztatható a problémamegoldás során.

Egy rendszernek tekintjük a valós dolgok azon részét, amely fontos a vizsgált probléma szempontjából, az egyes részek közötti kapcsolatok miatt.

<sup>3</sup>Természetesen ez igaz más tudományterületek számítási feladataira is, hiszen egy számítási feladat mondjuk a kémia területéről fölfogható úgy, mint egy szöveges matematikafeladat (pl.: keverési feladatok).

Bár a rendszer behatárolásával a valóság jelentős részét eleve kizárjuk, a rendelkezésünkre álló erőforrások végeessége miatt általában még az ezeket jellemző adatokat és összefüggéseket sem tudjuk maradéktalanul leírni. A modellalkotás az az eszköz, amely a valós rendszer absztrakciója révén a lényegtelen jellemzőket elhagyva a rendelkezésre álló ismereteket kezelhetővé teszi. A valós rendszer fölépítését a részei közötti kapcsolatok alapján adhatjuk meg. Ezeknek a kapcsolatoknak kell visszatükröződniük a rendszer adatmodelljében is.

## 2.2. Típusok

Az adatmodell hatékonyabb implementálása érdekében az egyes programozási nyelvek a tapasztalatok alapján többé-kevésbé hasonló adattípusok használatát teszik lehetővé.

Bizonyos adatok esetében nincs értelme valamely részüket elkülöníteni és azzal műveletet végezni. Mivel az ilyen adatoknak nincsenek önálló jelentéssel bíró részei, így nincs értelme beszélni a részek kapcsolatáról sem, tehát a szerkezetüktől sem. Az ilyen adatokra azt mondjuk, hogy elemi típusúak. Az algoritmusok tervezése vonatkozásában ezek elsősorban elvi kategóriát képviselnek, és a jobb átláthatóság érdekében célszerűnek tűnik viszonylag kevés számú elemi típus bevezetése, természetesen annak szem előtt tartásával, hogy a fejlesztés későbbi fázisát jelentő kódolás során ez ne jelentsen problémát.

A fentieknek megfelelően a későbbiekben az adatok jellege miatt a következőket tekintjük elemi típusoknak:

- egész,
- valós,
- logikai,
- szöveg.

Az elemi típusok közé szokták sorolni a karaktereket is. Bár a szöveg valóban karakterekből áll és valóban lehet értelme vizsgálni az egyes karaktereket is

– csak úgy, ahogyan egy egész szám egyes helyiértékein álló számjegyeket (például oszthatóság szempontjából, vagy az egy bájtón ábrázolt egész leg-  
alacsonyabb helyiértékű bitjét hasonló céllal). Az esetek többségében azon-  
ban az egyes karakterek a feladat szempontjából nem értelmezhetők, vagy  
az algoritmizálás során nincs szükség arra. A túlságosan szigorú besorolás a  
későbbiek során már csak azért is elveszíti a jelentőségét, mert az adatmodell  
tervezésekor még nem a memóriában való tárolás mikéntje, sokkal inkább az  
dominál, hogy milyen értékeket vehet föl az adat, és vele milyen műveleteket  
végezhetünk.

### 2.3. Az adatszerkezetek osztályozása

Az összetettebb problémák adatmodelljében a rendelkezésre álló jellemzők  
adatok formájában történő tárolásán túl a közöttük lévő logikai kapcsola-  
tokat is meg kell tudnunk jeleníteni. Ennek megvalósítására olyan összetett  
adatszerkezeteket kell kialakítanunk, amelyekkel a későbbiek folyamán ha-  
tékonyan tudunk majd műveleteket végezni, amibe az is beletartozik, hogy  
a választott programozási nyelv kellően támogatja azt.

A összetett adatszerkezeteket az alábbi szempontok szerint csoportosít-  
hatjuk:

1. Az elemek típusa szerint az adatszerkezet lehet

**homogén.** ha minden eleme azonos típusú, vagy

**heterogén.** ha az elemek nem azonos típusúak.

2. Az adatszerkezeten értelmezett művelet szerint

**struktúra nélküli.** adatszerkezet esetében az adatelemek között sem-  
miféle kapcsolat nincs (pl.: halmaz)

**asszociatív.** adatszerkezetek elemei között nincs lényegi kapcsolat, az  
elemei egyedileg címezhetők (pl.: tömb).

**szekvenciális.** adatszerkezetben minden elem – két kitüntetett elem  
kivételével (első és utolsó) – pontosan egy másik elemtől érhető  
el, és minden elemtől pontosan egy elem érhető el. Ez alól csak a

két kitüntetett elem kivétel, mert az adatszerkezetnek nincs olyan eleme, amelyből az első elem elérhető volna, és nincs olyan eleme sem, amely az utolsóból volna elérhető (pl.: egyszerű lista).

**hierarchikus.** adatszerkezet esetében egy olyan kitüntetett elem van (a gyökérelem), amelynek megadása egyenértékű az adatszerkezet megadásával. A gyökérelem kivételével minden elem pontosan egy másik elemből érhető el, de egy elemből tetszőleges (véges) számú további elemet lehet elérni. Ez utóbbi alól a csak az adatszerkezet végpontjai kivételek (pl.: bináris fa).

**hálós.** adatszerkezet elemei több elemből is elérhetőek és egy adott elemből is több további elemet tudunk elérni (pl.: gráf).

### 3. Az elemek száma szerint

**statikus.** adatszerkezet elemszáma állandó. Az általa elfoglalt tárterület nagysága nem változik a műveletek során.

**dinamikus.** adatszerkezetek elemszáma is véges, hiszen a rendelkezésre álló tár nagysága behatárolja a bővítés lehetőségét. Ugyanakkor az elemszáma a műveletek során változhat. Értelmezve van az adatszerkezet üres állapota, illetve amikor elfogyott az erre a célra fenntartott tárterület, akkor azt mondjuk, hogy megtelt az adatszerkezet. Az ilyen adatszerkezetek sajátossága, hogy új adatelemmel történő bővítése során az elem számára le kell foglalni a memória egy megfelelő területét, illetve akkor, amikor egy elem feleslegessé válik, akkor gondoskodni kell arról, hogy az így felszabaduló tárterület később ismét lefoglalható legyen.

### 4. Az adatelemek tárolási helye szerint

**folytonos.** reprezentáció esetén az az egybefüggő, legszűkebb tárterület, amely tartalmazza az adatszerkezet valamennyi elemét, csak ennek az adatszerkezetnek az elemeit tartalmazza.

**szétszórt.** reprezentációjú adatszerkezet elemei elvileg a tár tetszőleges részén lehetnek, az egyes elemek eléréséhez szükséges infor-

mációt a többi elem tárolja.

## 2.4. A vezérlés lehetőségei

A Neumann-elvek szerint működő gépek az utasításokat időben egymás után hajtják végre. Ez megfelel a legtöbb, korábban – a matematika területéről, a számítógépek megjelenése előtt – ismert algoritmus feldolgozásának. Ilyen algoritmusra vezet például a másodfokú egyenletek megoldására alkalmas

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

megoldóképlet is, hiszen először az egyenletet

$$ax^2 + bx + c = 0$$

alakra hozzuk, azután az együtthatók ismeretében kiszámítjuk a diszkrimináns értékét. Ezt követően pedig, annak értékétől függően meghatározhatjuk az egyenlet gyökeit.

Egy másik, szintén a matematika területéről ismert példa az  $a$  és  $b$  egész számok  $(a, b \in \mathbb{Z} \setminus \{0\})$  legnagyobb közös osztójának meghatározására alkalmas euklideszi algoritmus. Az algoritmus első lépésében kiszámítjuk az  $r_1$  osztási maradékot úgy, hogy az osztó nem lehet a nagyobb szám<sup>4</sup>:

$$a = b \cdot q_1 + r_1.$$

(Ahol  $q_1$  az  $a$  és  $b$  egészosztásának hányadosa.)

Ha a maradék nem nulla ( $r_1 \neq 0$ ), akkor újabb osztásra van szükség, de most az előző osztás osztóját kell osztanunk a maradékkal:

$$b = r_1 \cdot q_2 + r_2.$$

Természetesen most is előfordulhat, hogy a maradék nullától különböző. Ebben az esetben ismét a maradékot kell számolnunk a fentebb vázolt szabályok

<sup>4</sup> Könnyen belátható, hogy ha nem így teszünk, az algoritmus akkor is helyes eredményt szolgáltat, csak eggyel többször kell maradékot számítanunk.



szerint:

$$\begin{aligned} r_1 &= r_2 \cdot q_3 + r_3 \\ r_2 &= r_3 \cdot q_4 + r_4 \\ &\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ r_{n-1} &= r_n \cdot q_{n+1} + r_{n+1}. \end{aligned}$$

Abból, hogy az  $r_i$  maradékok értéke  $i$  növekedésével csökken – azaz a következő osztás maradéka egy pozitív egész értékkel mindig csökken az előző maradékhoz képest –, az következik, hogy a fenti osztások sorozatát ismételve véges számú művelet után be fog következni, hogy a maradék nullává válik, azaz van olyan  $n$ , hogy az

$$r_{n+1} = 0$$

teljesül. (Tehát  $n + 1$  osztást kell végeznünk.)

Mindkét jól ismert példa lehetőséget add annak szemléltetésére, hogy a sorrendi vezérlés nem azonos azzal, hogy az algoritmusaink lineárisak volnának. Láthattuk, hogy a másodfokú egyenlet megoldása során a gyök alatti kifejezés értéke alapján tudjuk eldönteni azt, hogyan is számoljunk tovább. (Ezért is szükséges azt előbb kiszámítani.) Tehát a diszkrimináns értéke meghatározza a további lehetőségeket, lépéseket. Annak értékéről csupán csak az egyenlet ismeretében nem tudunk semmit sem mondani a legtöbb esetben, tehát kiszámítása része az algoritmusnak.

Az euklideszi algoritmus esetében – bár szintén a korábbi számítások eredménye határozza meg a további lépéseket – mégis más a helyzet. Ha az első osztás maradéka nem nulla, akkor – triviális esetektől eltekintve – nem tudhatjuk, hogy hány további maradékra lesz még szükség<sup>5</sup>. Csak az adott osztás maradékának ismeretében tudjuk megmondani, hogy van-e még szükség továbbiakra, de azt nem, hogy vajon még hányra? Ez természetesen azt jelenti, hogy ez az algoritmus sem lesz lineáris, azaz a konkrét eset ( $a$  és  $b$  értéke) határozza meg a megoldáshoz vezető lépések sorozatát.

<sup>5</sup> Szemben a másodfokú egyenlet megoldásával, ahol a diszkrimináns kiszámítása után, annak ismeretében már meg tudjuk mondani a további lépéseket.

Az algoritmusok leírására használt eszközök és a különböző programozási nyelvek természetesen tartalmaznak olyan elemeket, amelyekkel a hasonló esetek megfelelő módon leírhatók a számítógép számára. Gyakran előfordul, hogy az algoritmizálással, programozással ismerkedők számára gondot okoz a két eset megkülönböztetése, bár a gyakorlatban képesek alkalmazni a megoldóképletet, illetve meg tudják határozni a két szám legnagyobb közös osztóját az ismertetett módon. Ennek általában az lehet az oka, hogy a lépések alkalmazása mechanikusan történik és nem föltétlen tudatosult azok lépésenkénti jelentősége. Az algoritmizálás oktatásakor azt kell elérnünk, hogy kialakulhasson a probléma megoldásához vezető tevékenység – adott eszközre jellemző – elemi lépésekre való bontásának képessége.

Az algoritmus szekvencialitásának megváltoztatására az elágazások szolgálnak, amelyek lehetővé teszik, hogy a korábbi műveletek eredményétől függően más-más lépések hajtódjanak végre. Például, ha az euklideszi algoritmus leírásában az szerepel, hogy az  $a$  és a  $b$  szám osztási maradékát kell számítani, ugyanakkor a nagyobb számmal nem oszthatunk, akkor gondoskodnunk kell arról, hogy a maradék számításakor  $a \geq b$  teljesüljön. Ez azt jelenti, hogy az  $a$  és a  $b$  szimbólumok értékét föl kell cserélnünk.

```
1 //...
2 HA b>a akkor
3     x ← a
4     a ← b
5     b ← x
6 HVége
7 //...
```

Elágazás

## 2.1. algoritmus

A további problémákat az ismételt osztások leírása okozhatja. Fontos annak fölismerése, hogy nem célszerű – adott esetben nem is lehetséges – az egyes lépéseket annyiszor leírni, ahányszor végre kell majd hajtani. Ha

azonban következetesen ragaszkodunk ahhoz, hogy az  $a$  az osztandó a  $b$  pedig az osztó, és elfogadjuk, hogy egy korábbi osztás osztandójára a későbbiek folyamán már nincs szükségünk a további számításokhoz, akkor belátható, hogy az  $a$  és a  $b$  értéket minden osztás után aktualizálnunk kell a fentebb említett *funkciójuknak* megfelelően.

```
1 //...
2   Ciklus
3     r ← a Div b;
4     a ← b;
5     b ← r;
6   CVége_Amikor (r=0);
7   Ki: a;
8 //...
```

Ismételt végrehajtás

## 2.2. algoritmus

### 2.5. Fogalmak, jelölések

#### 2.5.1. Hatékonyság

Induljunk ki abból a két nyilvánvaló dologból, hogy a számítógép számára minden művelet elvégzéséhez időre van szükség, valamint minden tárolt adat helyet foglal. Kézenfekvő, hogy ha egy feladat megoldásához különböző műveletsorozatok végrehajtásával is eljuthatunk, akkor azt az utat célszerű választani, amely a probléma megoldása szempontjából kedvezőbb. Például hamarabb szolgáltatja ugyanazt az eredményt, vagy kevesebb tárhelyet igényel a megoldás során. Természetes az is, hogy ha lehetőség van egy adat tárolása során annak tárigényét csökkentenünk az információtartalom csökkenése nélkül, akkor azt meg is kell tennünk.

Végezzünk most egy nagyon pontatlan, de talán mégis tanulságos szá-

mítást az alábbi kifejezések kiértékeléséhez szükséges idővel kapcsolatban<sup>6</sup>.

$$\prod_{i=1}^n a^i$$

$$a^{\sum_{i=1}^n i}$$

$$a^{\frac{n(n+1)}{2}}$$

Tételezzük föl, hogy számítógépünk esetében az összeadás, a szorzás és a hatványozás műveletének elvégzése rendre  $t_1$ ,  $t_2$  és  $t_3$  időt igényel és ezek között az alábbi összefüggés áll fenn:

$$2t_1 = t_2, \quad 2t_2 = t_3.$$

Ez azt jelenti, hogy egy szorzat kiszámításához szükséges idő alatt gépünk két összeadást, egy hatvány előállításához szükséges idő alatt pedig két szorzást képes elvégezni.

$$nt_2 + nt_3 = n(t_2 + t_3) = 6nt_1$$

$$nt_1 + t_3 = (n + 4)t_1$$

$$t_1 + 2t_2 + t_3 = 9t_1$$

### 2.5.2. Folyamatábra

A folyamatábra lényegében az algoritmust szemléltető irányított gráf. Csomópontjainak utasításokat, éleinek adatokat feleltethetünk meg. Van egy kitüntetett csomópontja, amelyből elindulva bármely másik csomópontba el lehet jutni – ez a kiinduló-csomópont, továbbá egy másik speciális csomópont, a vég-csomópont, amelyből nem vezet út más csomópontokhoz, de ebbe a csomópontba bármely más csomópontból el lehet jutni.

<sup>6</sup> A kifejezések közötti egyenlőség teljesül a matematikából jól ismert azonos alapú hatványok szorzására és az első  $n$  természetes szám összegére vonatkozó összefüggések miatt.

### 2.5.3. Leírónyelv

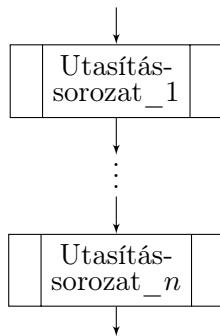
Ennek az eszköznek a segítségével programozási nyelvektől függetlenül, mégis azokhoz igen hasonló formában adhatjuk meg algoritmusainkat a konkrét programozási nyelvek kötöttségei nélkül.

### 2.5.4. Vezérlési szerkezetek

A vezérlési szerkezetek segítségével oldható meg, hogy az algoritmus utasításainak végrehajtási sorrendjét a kívánt módon tudjuk szervezni.

### 2.5.5. Szekvencia

Ez a vezérlési szerkezet a benne megadott utasítások egyszeri, minden feltételtől független, a megadás sorrendjében történő végrehajtását biztosítja.

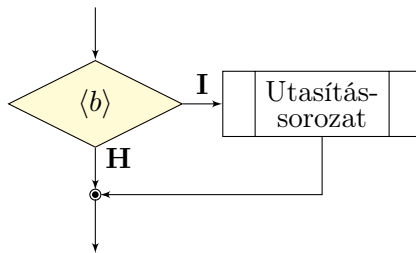


```
.  
.   
.   
Utasítássorozat_1;  
.   
.   
.   
Utasítássorozat_n;  
.   
.   
. 
```

### 2.5.6. Szelekció

#### Egyágú elágazás

Ha az elágazásban megadott  $\langle b \rangle$  logikai kifejezés értéke igaz, akkor az Utasítássorozat utasításait végre kell hajtani, majd az elágazást követő első utasításra kerül a vezérlés. Ellenkező esetben – a  $\langle b \rangle$  hamis értéke esetén – az algoritmus végrehajtását az elágazást követő utasítással kell folytatni az Utasítássorozat végrehajtása nélkül.



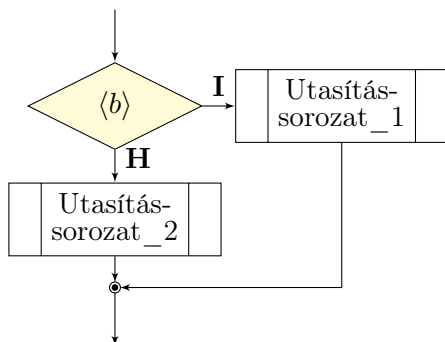
```

.
.
.
Ha <b> akkor
    Utasítássorozat;
HVége;
.
.
.

```

### Kétágú elágazás

Az itt megadott Utasítássorozat\_1 utasításai pontosan akkor hajtódnak végre, ha a  $\langle b \rangle$  logikai kifejezés értéke igaz, míg annak hamis értéke esetén az Utasítássorozat\_2 hajtódik végre. Ezt követően a vezérlés az elágazást követő első utasításra kerül.



```

.
.
.
Ha <b> akkor
    Utasítássorozat_1;
Különben
    Utasítássorozat_2;
HVége;
.
.
.
  
```

### 2.5.7. Iteráció

Az algoritmizálás során, már nem túl bonyolult feladatok esetében is gyakran van arra szükség, hogy valamely vezérlési szerkezet utasításainak kiértékelését egy végrehajtás után esetleg újból végrehajtsa a rendszer. Erre a problémára az iteráció kínál lehetőséget. Az iterációs vezérlési szerkezeteket alapvetően két két nagy csoportba sorolhatjuk. A vezérlési szerkezet első utasításának végrehajtása előtt

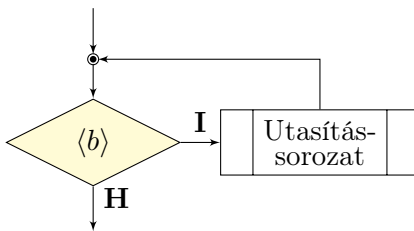
- még nem ismerjük
- már tudjuk

a szükséges ismétlések számát. Ennek megfelelően a vezérlési szerkezet megvalósítására általában az alábbi utasítások állnak rendelkezésre.

#### Elől tesztelő ciklus

A ciklusmag utasításai mindaddig végrehajthatódnak, míg a  $\langle b \rangle$  logikai kifejezés értéke **igaz**. Azaz az algoritmus végrehajtása a **H** ágon csak akkor folytatódik, ha a  $\langle b \rangle$  logikai kifejezés értéke hamissá válik. Ebből az is következik, hogy

- ha  $\langle b \rangle$  hamis közvetlenül a ciklusba való belépés előtt, akkor a ciklusmag utasításai egyszer sem hajtódnak végre.
- ha pedig végrehajtódnak a ciklusmag utasításai, de sohasem válik a  $\langle b \rangle$  logikai kifejezés értéke hamissá, akkor nem fog kilépni a ciklusból, azaz nem kerülhet a vezérlés a ciklust követő utasításra, tehát  $\langle b \rangle$ -nek függenie kell a ciklusmag utasításaitól.



```

.
.
.
Ciklus_Míg <b>
    Utasítássorozat;
CVége;
.
.
.
  
```

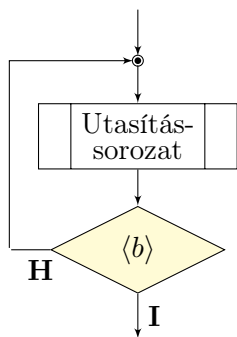
### Hátul tesztelő ciklus

A ciklusmag utasításai mindaddig végrehajtódnak, míg a  $\langle b \rangle$  logikai kifejezés értéke **hamis**. Azaz az algoritmus végrehajtása az **I** ágon csak akkor folytatódik, ha a  $\langle b \rangle$  logikai kifejezés értéke igazgá válik. Ebből az is következik, hogy

- a  $\langle b \rangle$  belépéskori értékétől függetlenül a ciklusmag utasításai egyszer biztosan végre fognak hajtódni.
- mivel a ciklusból való kilépés csak akkor következik be, ha a ciklus végén lévő feltételvizsgálatkor a  $\langle b \rangle$  logikai értéke igaz, a ciklusmag



utasításaitól  $\langle b \rangle$  függenie kell<sup>8</sup>.



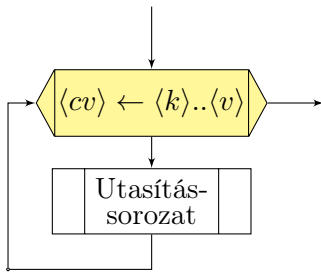
```
.  
.   
.   
Ciklus  
    Utasítássorozat;  
CVége_Amikor <b>;  
.   
.   
. 
```

### Előírt lépésszámú ciklus <sup>9</sup>

Az előírt lépésszámú ciklus alkalmazására akkor van lehetőségünk, ha a ciklusmag utasításainak végrehajtása előtt már ismerjük a szükséges ismétlések számát. Bár ez az utasítás feltételes ciklus alkalmazásával kiváltható volna, az algoritmus jobb olvashatósága, könnyebb megértése miatt használata feltétlen indokolt.

<sup>8</sup> Abban az esetben, ha ciklusba való belépéskor  $\langle b \rangle$  igaz és nem függ a ciklusmag utasításaitól, a ciklusból való kilépés természetesen bekövetkezik az utasítások egyszeri végrehajtása után, de ha ezt kívánja meg a probléma, akkor nem indokolt a hátul tesztelő ciklus alkalmazása.

<sup>9</sup> Az itt bemutatott csomópontnak önmagában nincsen értelme, csak az itt megadott-hoz hasonló speciális szerkezetű részgráfokban. Maga a csomópont lényegében egy elágazás és egy gyűjtőcsomópont összevonásával jött létre, de tartalmaz olyan információkat is, amelyek a  $\langle cv \rangle$  inicializálására és változására utalnak.



```

.
.
.
Ciklus <cv>←<k>..<v>
    Utasítás-sorozat;
CVége;
.
.
.

```

## 2.6. Feladatok

- Adott koordinátaival a sík egy  $P(x, y)$  pontja és az origó középpontú  $r$  sugarú kör. Adjuk meg azt a logikai kifejezést, amely akkor igaz, ha a  $P$  pont
  - belső pontja a körnek.
  - a körön kívül helyezkedik el.
  - illeszkedik a körvonalra.
- Adjuk meg azokat a logikai kifejezéseket, melyeknek értéke pontosan akkor igaz, amikor a fenti feltételek nem teljesülnek.
- Írjunk algoritmust, amely beolvassa egy  $P(x, y)$  pont koordinátáit és eldönti, hogy az hogyan helyezkedik el az origó középpontú  $r$  sugarú körhöz képest.
- Adott koordinátaival a sík egy  $P(x, y)$  pontja és az  $ABCD$ -négyzet  $(A(1; 1), B(1; -1), C(-1; -1), D(-1; 1))$ . Adjuk meg azt a logikai kifejezést, amely akkor igaz, ha a  $P$  pont
  - belső pontja a négyzetnek.

- b) a négyzeten kívül helyezkedik el.
- c) illeszkedik a négyzet valamely oldalára.
5. Adjuk meg azokat a logikai kifejezéseket, melyeknek értéke pontosan akkor igaz, amikor a fenti feltételek nem teljesülnek.
6. Írjunk algoritmust, amely beolvassa egy  $P(x, y)$  pont koordinátáit és eldönti, hogy az hogyan helyezkedik el az  $ABCD$ -négyzethez képest.
7. Adott koordinátaival a sík egy  $P(x, y)$  pontja és az  $ABCD$ -négyzet  $(A(0; 1), B(1; 0), C(0; -1), D(-1; 0))$ . Adjuk meg azt a logikai kifejezést, amely akkor igaz, ha a  $P$  pont
- a) belső pontja a négyzetnek.
- b) a négyzeten kívül helyezkedik el.
- c) illeszkedik a négyzet valamely oldalára.
8. Adjuk meg azokat a logikai kifejezéseket, melyeknek értéke pontosan akkor igaz, amikor a fenti feltételek nem teljesülnek.
9. Írjunk algoritmust, amely beolvassa egy  $P(x, y)$  pont koordinátáit és eldönti, hogy az hogyan helyezkedik el az  $ABCD$ -négyzethez képest.
10. Legyenek  $a, b$  és  $c$  pozitív valós értékek. Adjuk meg azt a logikai kifejezést, melynek értéke pontosan akkor igaz, ha
- a)  $a, b$  és  $c$  lehetnek egy háromszög oldalhosszai.
- b)  $a, b$  és  $c$  hosszúságú szakaszokkal
- α) derékszögű háromszög szerkeszthető.
- β) egyenlő szárú háromszög szerkeszthető.
- γ) szabályos háromszög szerkeszthető.
11. Adjuk meg azokat a logikai kifejezéseket, melyeknek értéke pontosan akkor igaz, amikor a fenti feltételek nem teljesülnek.

12. Írjunk olyan algoritmust, amely beolvassa  $a, b$  és  $c$  pozitív értékeket és kijelzi, hogy azokkal, mint oldalhosszakkal szerkeszthető-e háromszög, és ha igen akkor milyen.
13. Írjunk olyan algoritmust, amely beolvassa  $a, b$  és  $c$  pozitív értékeket és az alábbi tartalmú szövegek egyikét olyan módon írja ki, hogy a benne található szavakat csak egyszer használgatjuk föl.

- *A megadott adatokkal nem szerkeszthető háromszög.*
- *A megadott adatokkal derékszögű háromszög szerkeszthető.*
- *A megadott adatokkal szabályos háromszög szerkeszthető.*
- *A megadott adatokkal egyenlő szárú háromszög szerkeszthető.*
- *A megadott adatokkal egyenlő szárú derékszögű háromszög szerkeszthető.*

(Természetesen a szavak sorrendjének nem föltétlen kell megegyeznie a fentiekkel, de a többször előforduló szövegrészek –háromszög, derékszögű, egyenlő szárú, stb.– csak egyszer szerepelhetnek kiíratásban.)

14. Írjunk algoritmust, amely három valós érték  $(a, b, c)$  bekérése után kijelzi, hogy az  $ax^2 + bx + c = 0$  másodfokú egyenletnek hány valós gyöke van.
15. Írjunk algoritmust, amely három valós érték  $(a, b, c)$  bekérése után számítja az  $ax^2 + bx + c = 0$  másodfokú egyenlet a valós gyökeit.
16. Írjunk algoritmust, amely mindaddig beolvas egy  $(a, b, c)$  valós számhármast, amíg az  $ax^2 + bx + c = 0$  másodfokú egyenletnek
- legalább 1 valós gyöke van.
  - pontosan 2 valós gyöke van.
  - nincs valós gyöke.
17. Írjunk algoritmust, amely az  $a$  és  $b$  pozitív egészek legnagyobb közös osztóját számítja ki az euklideszi algoritmus alapján. Az algoritmus

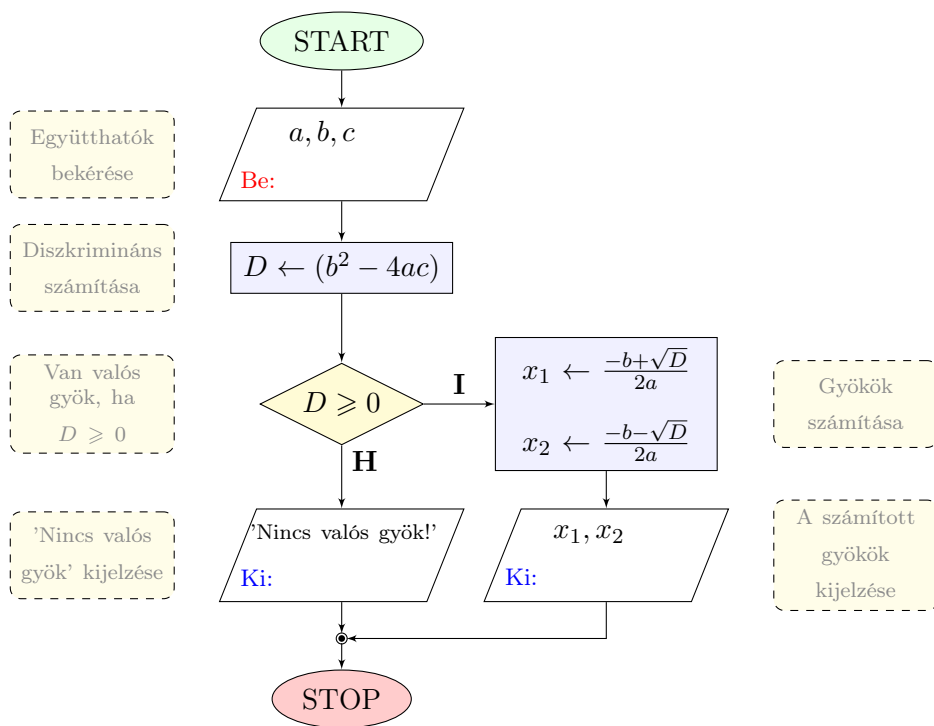
a legnagyobb közös osztó mellett jelezze ki azt is, hogyha a számok relatív prímek.

18. Az  $A$  négyzetes mátrix esetében tudjuk, hogy főátlóján kívül csak 0 szerepel<sup>10</sup>. Tehát ha a mátrixnak  $n$  sora van, akkor csupán  $n$  elem tárolása „hasznos”, hiszen a többről biztosan tudjuk, hogy értékük 0. Tároljuk a teljes mátrix helyett annak csupán a főátlóbeli elemeit egy  $n$  elemű  $V$  vektorban. Írjunk függvényt, amelynek paraméterébe beírva a  $V$  vektort és a kívánt elem  $A$  mátrixbeli indexeit, visszaadja a megfelelő mátrixbeli elem értékét.
19. Az  $A$  négyzetes mátrix esetében tudjuk, hogy főátlója alatti elemek értéke rendre 0<sup>11</sup>. Tehát ha a mátrixnak  $n$  sora van, akkor hány „hasznos”, 0-tól különböző eleme van a mátrixnak? Tároljuk a teljes mátrix helyett annak csupán azon elemeit, amelyekről nem tudjuk biztosan, hogy értékük 0 egy  $V$  vektorban sorfolytonosan. Írjunk függvényt, amelynek paraméterébe beírva a  $V$  vektort és a kívánt elem  $A$  mátrixbeli indexeit, visszaadja a megfelelő mátrixbeli elem értékét.
20. Az  $A$  négyzetes mátrix esetében tudjuk, hogy főátlója feletti elemek értéke rendre 0<sup>12</sup>. Tehát ha a mátrixnak  $n$  sora van, akkor hány „hasznos”, 0-tól különböző eleme van a mátrixnak? Tároljuk a teljes mátrix helyett annak csupán azon elemeit, amelyekről nem tudjuk biztosan, hogy értékük 0 egy  $V$  vektorban sorfolytonosan. Írjunk függvényt, amelynek paraméterébe beírva a  $V$  vektort és a kívánt elem  $A$  mátrixbeli indexeit, visszaadja a megfelelő mátrixbeli elem értékét.

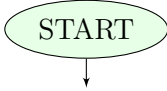
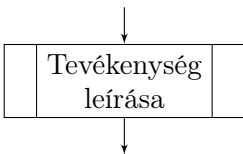
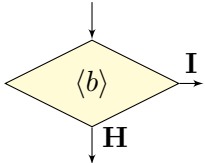


<sup>10</sup> Diagonális mátrix

<sup>11</sup> Felső-háromszög mátrix

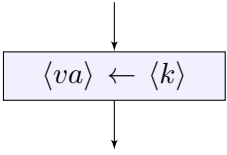
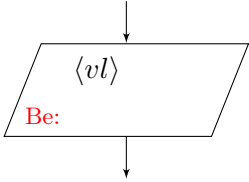
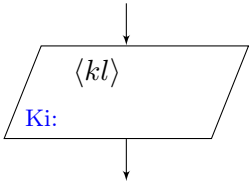
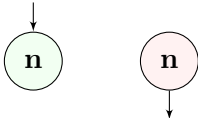
<sup>12</sup> Alsó-háromszög mátrix



2.2. ábra. A másodfokú egyenlet valós megoldásait szolgáltató algoritmus folyamatábrája

Csomópont	Utasítás	Be- futó élek száma	Ki- futó élek száma
	<b>START:</b> az algoritmus kezdetét jelöli. A valódi program grájában csak egyszer szerepelhet.	-	1
	<b>Tevékenység:</b> a program-gráf tetszőleges olyan rész-gráfa jelölhető vele, amelyhez pontosan egy bemenő és egy kimenő él tartozik és praktikusán önálló funkcióval bír.	1	1
	<b>Döntés:</b> az algoritmus végrehajtása az utasításban megadott $\langle b \rangle$ logikai kifejezés értékétől függően ( <b>I</b> gaz/ <b>H</b> amis), az <b>I</b> -vel vagy a <b>H</b> -val jelzett él mentén elérhető csomóponton folytatódik.	1	2
	<b>Gyűjtő csomópont:</b> akkor alkalmazzuk, amikor kettő vagy több különböző programrészlet azonos részprogram végrehajtásával folytatódik <sup>7</sup> .	2	1
	<b>STOP:</b> az algoritmus végét jelöli. A valódi program grájában csak egyszer szerepelhet.	1	-

2.1. táblázat  
Folyamatábra jelölései.

Csomópont	Utasítás	Be- futó élek száma	Ki- futó élek száma
	<b>Értékkadás:</b> hatására a baloldalon lévő változóhoz tartozó memóriaterület megváltozik a jobboldalon található kifejezés értékének megfelelően.	1	1
	<b>Bemenet:</b> az utasításban megadott $\langle vl \rangle$ véges számú változó értékének megváltoztatására van lehetőség, valamely arra alkalmas perifériáról beolvasott értékek alapján.	1	1
	<b>Kimenet:</b> az utasításban megadott $\langle kl \rangle$ véges számú kifejezés értéke jeleníthető meg, valamely arra alkalmas periférián.	1	1
	<b>Kapcsolódási pont:</b> nem jelenti valódi csomópontját a programgráfnak, csupán technikai szerepe van. Akkor alkalmazzuk, ha a folyamatábra írásának fizikai akadályai vannak.	1   -	-   1

2.2. táblázat  
Folyamatábra további jelölései.



## 3. Alapalgoritmusok

### 3.1. Sorozatszámítás

#### 3.1.1. Összegzés

Adott az  $n$  elemű  $a$  sorozat és a sorozat elemein értelmezett kétoperandusos művelet. Ennek operátorát jelölje  $\diamond$ . Az algoritmus a sorozathoz az

$$a_1 \diamond a_2 \diamond \dots \diamond a_n$$

értéket rendeli.

```
1 Függvény Összegzés(A:sorozat):Elemtip;
2 Változó
3   S: Elemtip;           // S-ben keletkezik majd az összeg
4   I: egész;             // ciklusváltozó
5 Algoritmus
6   S ← 0;                // S kezdőértéke 0
7   Ciklus I ← (1..N)
8     S ← S + A[I];       // s = a1 + a2 + ... + ai
9   CVége;
10  Összegzés ← S;        // A  $\sum_{i=1}^n a_i$ 
11                          // visszatérési érték beállítása.
12 FVége;
```

Összegzés

#### 3.1. algoritmus

A 3.1. algoritmus az összegzés egy lehetséges megvalósítását mutatja be, amikor is a művelet az aritmetikai összeadás, és az algoritmusban fölhasználjuk, a műveletnek zérus-eleme a 0.

### 3.1.2. Megszámlálás

Adott az  $n$  elemű  $a$  sorozat és a sorozat elemein értelmezett  $T$ -tulajdonság<sup>13</sup>. A 3.2. algoritmus a sorozathoz hozzárendeli a  $T$ -tulajdonságú elemeinek a számát, amely 0-tól  $n$ -ig terjedő egész értéket jelenhet. Könnyű látni, hogy a függvény visszatérési értéke pontosan akkor 0 ha a sorozatnak nincsen egyetlen  $T$ -tulajdonságú eleme sem, illetve  $n$ , ha minden elem  $T$ -tulajdonságú.

```
1 Függvény Megszámlálás( $\underline{A}$ :sorozat):egész;  
2 Változó  
3   I:  egész;  
4   DB: egész;           // DB a részsorozat elemszáma  
5 Algoritmus  
6   DB  $\leftarrow$  0;  
7   Ciklus I  $\leftarrow$  (1..N)  
8     Ha  $T(\underline{A}[I])$  Akkor  
9       DB  $\leftarrow$  DB + 1;  
10    HVége;  
11    CVége;  
12    Megszámlálás  $\leftarrow$  DB;  
13 CVége;
```

Megszámlálás

### 3.2. algoritmus

#### 3.1.3. Kiválogatás

Ezt az algoritmust akkor alkalmazhatjuk, ha az  $n$  elemű  $a$  sorozathoz  $T$ -tulajdonságú elemeinek részsorozatát szeretnénk hozzárendelni. és a sorozat elemein értelmezett  $T$ -tulajdonság.

<sup>13</sup> Az, hogy a sorozat elemein értelmezve van a  $T$ -tulajdonság, azt jelenti, hogy annak a halmaznak amelyből a sorozat elemét „válogatjuk” vannak ilyen elemei, de az korántsem teljesül, hogy magának a sorozatnak is lesznek ilyen elemei

```

1  Függvény Kiválogat(A,B:sorozat):egész;
2  Változó:
3    I:  egész;
4    DB: egész;          // DB a részsorozat elemszáma
5  Algoritmus
6    DB ← 0;
7    Ciklus I ← (1..N)
8      Ha T(A[I]) akkor
9        DB ← DB + 1;
10       B[DB] ← A[I];
11     HVége;
12   CVége;
13   Kiválogat ← DB;
14  FVége;

```

Kiválogatás

### 3.3. algoritmus

Összevetve a kiválogatás 3.3. algoritmusát a megszámlálás 3.2. algoritmusával, könnyen látható a közöttük lévő szerkezeti és funkcionális hasonlóság.

#### 3.1.4. Minimum- és maximumkiválasztás

Legyen adott az  $n$  elemű  $a$  sorozat, melynek értékkészlete rendezett halmaz. Ilyen esetekben lehet feladat a sorozat minimális/maximális értékű elemének meghatározása. Előfordul, hogy csupán az adott elem értékének meghatározása a cél, de szükségünk lehet esetenként maximális illetve minimális értékű elem első vagy utolsó előfordulási helyének meghatározására is.

```

1 Függvény MinimumÉ(A:sorozat):Elemtip;
2 Változó
3   MinÉ: Elemtip;           // a legkisebb elem értéke
4   I: egész;                // ciklusváltozó
5 Algoritmus
6   MinÉ ← A[1];             // MinÉ kezdőértéke
7   Ciklus I ← (2...N)       // Az 1. elemet a MinÉ kezdőértékével
8                             // vettük figyelembe.
9       Ha A[I] < MinÉ akkor
10
11           // Ha az aktuális elem kisebb,
12           MinÉ ← A[I];     // annak az értékét jegyezzük meg.
13       HVége;
14       CVége;
15   MinimumÉ ← MinÉ;        // visszatérési érték beállítása.
16 FVége;

```

Minimális érték kiválasztása

### 3.4. algoritmus

A 3.4. algoritmus a sorozat legkisebb elemének értékét határozza meg, de értelmezése után könnyen megírható a maximális értéket meghatározó változat is az algoritmus 9. sorának megfelelő módosításával.

Bizonyos esetekben nem elegendő ismernünk csupán a sorozatban előforduló maximális illetve minimális értéket, fontos információt hordozhat annak előfordulási helye is. A 3.4. algoritmust módosíthatjuk úgy, hogy az egyes elemek értékét a sorszámukon keresztül érjük el. Ennek érdekében a 3.5. algoritmus 6. sorában az első elem sorszámát és nem annak értékét tároljuk el MinI-ben. A ciklusmag elágazásában ennek segítségével érjük el az eddig minimálisnak talált elem értékét és hasonlítjuk össze a sorozat aktuális elemének értékével az algoritmus 9. sorában. Amennyiben az aktuális elem értéke kisebb az eddigi legkisebbnél, akkor annak sorszámát tároljuk el a MinI változóban. Mivel MinI értéke az algoritmus 11. sorában csak akkor

módosul, ha az eddigi legkisebbnél kisebb értéket találtunk, biztosítva van, hogy ciklusból kilépve MinI a minimális értékű elem első előfordulásának sorszámát tartalmazza.

```
1 Függvény MinimumH(A:sorozat):Egész;  
2 Változó  
3   MinI: egész;           // a legkisebb elem indexe  
4   I: egész;              // ciklusváltozó  
5 Algoritmus  
6   MinI ← 1;              // MinI kezdőértéke legyen 1  
7   Ciklus I ← (2...N)     // Az 1. elemet a MinI kezdőértékével  
8                           // vettük figyelembe.  
9       Ha A[I] < A[MinI] akkor  
10                           // Ha az aktuális elem kisebb,  
11           MinI ← I;      // annak a sorszámát jegyezzük meg.  
12       HVége;  
13   CVége;  
14   MinimumH ← MinI;      // visszatérési érték beállítása.  
15 FVége;
```

Minimális értékű elem első előfordulási helyének  
meghatározása

### 3.5. algoritmus

A 3.5. algoritmus értelmezése után az Olvasóra bízunk a minimális értékű elem utolsó, a maximális értékű elem első és utolsó előfordulási helyét produkáló algoritmus megírását.

## 3.2. Feladatok

1. Adjuk meg a fejezet leírónyelvű algoritmusait blokkdiagram formájában.
2. Összegezzük egy négyzetes mátrix

- a) főátlójának elemeit.
- b) mellékátlójának elemeit.
- c) főátló feletti elemeit
  - $\alpha$ ) oszlop-folytonosan.
  - $\beta$ ) sor-folytonosan.
- d) főátló alatti elemeit
  - $\alpha$ ) sor-folytonosan.
  - $\beta$ ) oszlop-folytonosan.
- e) mellékátló feletti elemeit
  - $\alpha$ ) oszlop-folytonosan.
  - $\beta$ ) sor-folytonosan.
- f) mellékátló alatti elemeit
  - $\alpha$ ) sor-folytonosan.
  - $\beta$ ) oszlop-folytonosan.

3. Az összegzés tételének alkalmazásával számítsuk az alábbi kifejezések értékét, feltételezve, hogy elegendő csak véges sok tagot figyelembe vennünk:

a)

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

b)

$$4 \sum_{k=0}^{\infty} (-1)^k \frac{1}{2k+1}$$

c)

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots$$

d)

$$\sqrt{6 \sum_{k=0}^{\infty} \frac{1}{k^2}}$$

e)

$$\frac{\pi^4}{90} = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \dots$$

f)

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \dots$$

g)

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdot \dots$$

h)

$$\pi = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1}$$

i)

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^{-k}}{2k+1}$$

j)

$$\pi = 3 + \frac{4}{2 \cdot 3 \cdot 4} - \frac{4}{4 \cdot 5 \cdot 6} + \frac{4}{6 \cdot 7 \cdot 8} - \frac{4}{8 \cdot 9 \cdot 10} + \dots$$

k)

$$\pi = \sum_{k=0}^{\infty} \left( \frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \left( \frac{1}{16} \right)^k$$

l)

$$\pi = \frac{\prod_{k=0}^{\infty} \left( 1 + \frac{1}{4k^2-1} \right)}{\sum_{k=0}^{\infty} \frac{1}{4k^2-1}}$$

m) Gauss–Legendre iterációs algoritmus

$$a_0 := 1;$$

$$b_0 := \frac{1}{\sqrt{2}};$$

$$t_0 := \frac{1}{4};$$

$$p_0 := 1;$$

$$a_{n+1} := \frac{a_n + b_n}{2}$$

$$b_{n+1} := \sqrt{a_n b_n}$$

$$t_{n+1}:=t_n-p_n(a_n-a_{n+1})^2$$

$$p_{n+1}:=2p_n$$

$$\pi \approx \frac{(a_n+b_n)^2}{4t_n}$$

$n)$

$$e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots$$

$o)$

$$e=1+\frac{1}{1}\left(1+\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(1+\frac{1}{5}(1+\ldots)\right)\right)\right)\right)$$

$p)$

$$\frac{1}{e}=1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\ldots$$

$q)$

$$\ln 2=1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\ldots$$

$r)$

$$2=1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots$$

$s)$

$$\frac{1}{4}=\frac{1}{1\cdot 2\cdot 3}+\frac{1}{2\cdot 3\cdot 4}+\frac{1}{3\cdot 4\cdot 5}+\cdots$$

$t)$

$$\sum_{k=0}^\infty \frac{1}{(k-1)k(k+1)}$$

$u)$

$$\frac{1}{(l-1)(l-1)!}=\frac{1}{1\cdot 2\cdot \ldots \cdot l}+\frac{1}{2\cdot 3\cdot \ldots \cdot (l+1)}+\frac{1}{3\cdot 4\cdot \ldots \cdot (l+2)}+\cdots$$

$v)$

$$L(x):=\sum_{j=0}^k\left(y_j\prod_{0\leqslant m\leqslant k\atop m\neq j}\frac{x_i-x_m}{x_j-x_m}\right)$$



$$\begin{aligned}
L(x) = & y_0 \frac{(x-x_1)\dots(x-x_k)}{(x_0-x_1)\dots(x_0-x_k)} + y_1 \frac{(x-x_0)(x-x_2)\dots(x-x_k)}{(x_1-x_0)(x_1-x_2)\dots(x_1-x_k)} + \dots \\
& \dots + y_j \frac{(x-x_0)\dots(x-x_{j-1})(x-x_{j+1})\dots(x-x_k)}{(x_j-x_0)\dots(x_j-x_{j-1})(x_j-x_{j+1})\dots(x_j-x_k)} + \dots \\
& \dots + y_k \frac{(x-x_0)\dots(x-x_{k-1})}{(x_k-x_0)\dots(x_k-x_{k-1})}
\end{aligned}$$

4. Adott az  $a$  páratlan elemszámú ( $n = 2k + 1$ , ahol  $k \in N$ ) sorozat és biztosan tudjuk, hogy a sorozatnak legalább 3 eleme van. Képezzük az alábbi összeget:

$$S := a_1 + 2a_2 + 2a_3 + 2a_4 + \dots + 2a_{n-1} + a_n$$

5. Adott az  $a$  páratlan elemszámú ( $n = 2k + 1$ , ahol  $k \in N$ ) sorozat és biztosan tudjuk, hogy a sorozatnak legalább 5 eleme van. Képezzük az alábbi összeget:

$$S := a_1 + 4a_2 + 2a_3 + 4a_4 + 2a_5 + \dots + 4a_{n-1} + a_n$$

6.

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

7.

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

8.

$$\cos x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

9. Koordinátaival adott egy  $n$  elemű pontsorozat a síkban. Határozzuk meg, hány olyan eleme van, amely kívül esik egy adott  $r$  sugarú  $O(x_0, y_0)$  középpontú körön.

10. Koordinátaival megadott  $n$  elemű, síkbeli pontsorozat elemei jelentsék

egy  $n$ -oldalú sokszög csúcpontjait. (A megadás sorrendje megfelel körüljárási iránynak.) Határozzuk meg, hogy a sokszögnek hány adott  $d$ -nél hosszabb oldala van.

11. Koordinátaival adott egy  $n$  elemű pontsorozat a síkban. Határozzuk meg annak az  $O(x_0, y_0)$  középpontú körnek a sugarát, amelyen kívül nem esik egyetlen pont sem.
12. Koordinátaival adott egy  $n$  elemű pontsorozat a síkban. Határozzuk meg azt az elemét, amely egy adott  $O(x_0, y_0)$  ponttól a legtávolabb van.
13. Koordinátaival megadott  $n$  elemű, síkbeli pontsorozat elemei jelentsék egy  $n$ -oldalú sokszög csúcpontjait. (A megadás sorrendje megfelel a körüljárási iránynak.) Határozzuk meg, hogy a sokszögnek hány adott  $d$ -nél hosszabb oldala van.
14. Koordinátaival megadott  $n$  elemű, síkbeli pontsorozat elemei jelentsék egy  $n$ -oldalú sokszög csúcpontjait. (A megadás sorrendje megfelel a körüljárási iránynak.) Határozzuk meg, hogy a sokszögnek hány adott  $d$ -nél hosszabb oldala van.
15. Határozzuk meg egy sorozat második legnagyobb elemének első előfordulási helyét.
16. A 3.6, a 3.7, a 3.8 és a 3.9 algoritmusok bankkártyaszám<sup>14</sup> ellenőrzését végzik. Elemezzük a fent említett algoritmusok hatékonyságát végrehajtási idő, tárigény és bonyolultság szempontjából is.

<sup>14</sup> A bankkártyaszám felépítéséről és ellenőrzésének mechanizmusáról további információk a 10.1.7 alfejezetben találhatók.

```

1 Függvény Luhn1(A: kód):Logikai;
2 Változó
3   i: egész;
4   SUM: elemtip;
5 Algoritmus
6   SUM ← 0;
7   Ciklus (i ← 1 .. A.elemszám)
8       HA (i MOD 2)=0 akkor
9           HA 2*A[i] > 9 akkor
10              SUM ← SUM + 2*A[i] - 9;
11           Különbén
12              SUM ← SUM + 2*A[i];
13           HVége;
14       Különbén
15              SUM ← SUM + A[i];
16           HVége;
17   CVége;
18   Luhn1 ← (SUM MOD 10)=0;
19 FVége;

```

Luhn-algoritmus egy lehetséges megvalósítása

### 3.6. algoritmus

```

1 Függvény Luhn2( A: kód ): Logikai;
2 Változó
3   i: egész;
4   SUM: elemtip;
5 Algoritmus
6   SUM ← 0;
7   Ciklus (i ← 1 .. A.elemszám)
8       HA (i MOD 2)=0 akkor
9           HA A[i] > 4 akkor
10              SUM ← SUM + 2*A[i] - 9;
11           Különbén
12              SUM ← SUM + 2*A[i];
13           HVége;
14       Különbén
15          SUM ← SUM + A[i];
16       HVége;
17   CVége;
18   Luhn2 ← (SUM MOD 10)=0;
19 FVége;

```

Luhn-algoritmus egy lehetséges megvalósítása

### 3.7. algoritmus

```

1 Függvény Luhn3( A: kód ): Logikai;
2 Változó
3   i: egész;
4   SUM: elemtip;
5 Algoritmus
6   SUM  $\leftarrow$  0;
7   i  $\leftarrow$  1;
8   Ciklus_Míg (i  $\leq$  A.elemszám)
9       HA A[i] > 4 akkor
10          SUM  $\leftarrow$  SUM + 2*A[i] - 9;
11       Különbén
12          SUM  $\leftarrow$  SUM + 2*A[i];
13       HVége;
14       i  $\leftarrow$  i + 2;
15   CVége;
16   i  $\leftarrow$  2;
17   Ciklus_Míg (i  $\leq$  A.elemszám)
18       SUM  $\leftarrow$  SUM + A[i];
19       i  $\leftarrow$  i + 2;
20   CVége;
21   Luhn3  $\leftarrow$  (SUM MOD 10)=0;
22 FVége;

```

Luhn-algoritmus egy lehetséges megvalósítása

### 3.8. algoritmus

```

1 Függvény Luhn4( A: kód ): Logikai;
2 Változó
3   i: egész;
4   SUM: elemtip;
5 Algoritmus
6   SUM ← 0;
7   i ← 1;
8   Ciklus_Míg (i ≤ A.elemszám)
9       HA A[i] > 4 akkor
10          SUM ← SUM + 2*A[i] - 9 + A[i + 1];
11       Különben
12          SUM ← SUM + 2*A[i] + A[i + 1];
13       HVége;
14       i ← i + 2;
15   CVége;
16   Luhn4 ← (SUM MOD 10)=0;
17   FVége;

```

Luhn-algoritmus egy lehetséges megvalósítása

### 3.9. algoritmus

## 4. Keresés, rendezés

Az alábbiakon kívül természetesen számtalan kapcsolat van a matematika és az informatika tudománya között, hiszen az informatika minden részterületének matematikailag megalapozottnak kell lennie. A továbbiakban néhány, elsősorban az algoritmizálás szempontjából fontos matematikai fogalmat, ismeretet tárgyalunk az informatika szempontjából praktikus megközelítésben.

### 4.1. Sorozat

A logikailag összetartozó, hasonló jellegű, általában azonos jelentéssel bíró adatok összességét, mikor az egyes elemek értékén kívül azok sokaságon belüli helye is fontos, a matematika eszközszerében sorozatként tudjuk megadni. Az elemek egymáshoz képest elfoglalt helyének megadása céljából az elemeket megsorszámozzuk. Matematikailag ez azt jelenti, hogy a természetes számokhoz  $(1, 2, \dots, n)$  hozzárendeljük egy másik  $\mathbb{H}$  halmaz elemeit. Ezt a matematikában az alábbi módon szokták jelölni:

$$a : \mathbb{N} \rightarrow \mathbb{H}$$

Ez lényegében egy olyan függvénykapcsolatot jelent, melyben létrejött rendezett párok első eleme természetes szám (a sorozat elemeinek indexe), a második elem pedig a sorozat adott elem<sup>15</sup>. A mi szempontunkból a véges sorozatok a legfontosabbak, különösen, ha azok elemeit tárolnunk is kell, illetve az elemekkel műveleteket is kell végeznünk.

Sorozatokat alkotnak például valamely időszakban a naponkénti deviza-

<sup>15</sup> A következő számsorozat az Euró árfolyamának naponkénti alakulását adja meg egy időszakban: 293,44; 291,29; 292,96; 291,21; 290,96; 291,05.

A fentieknek megfelelően ez a sorozat a következő számpárokkal is megadható, hiszen függvényről van szó: (1;293,44) (2;291,29) (3;292,96) (4;291,21) (5;290,96) (6;291,05).

Ezzel a megadással egyenértékű az alábbi is:  $a_1 = 293,44$ ;  $a_2 = 291,29$ ;  $a_3 = 292,96$ ;  $a_4 = 291,21$ ;  $a_5 = 290,96$ ;  $a_6 = 291,05$ .

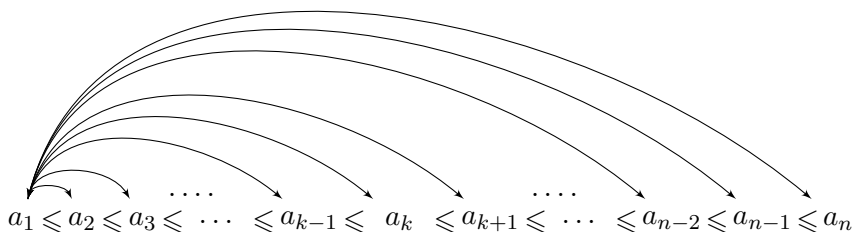
Általában azonban az sem okoz félreértést, ha a sorozat elemeit egyszerűen felsoroljuk.

árfolyamok. Logikailag összetartozó értékekről beszélhetünk, mivel egy sorozat elemei egy adott deviza árfolyamának változását írja le, így az egyes elemek jellege is azonos. Az egyes elemeket nem cserélhetjük föl, hiszen az adott napra jellemző az értékük. Hasonló képen beszélhetünk egy adott év napi középhőmérsékleteinek sorozatáról is.

Legyenek  $a_1, a_2, \dots, a_n$  valós számok, amelyek rendre bizonyos dolgok valamely mérhető tulajdonságát jelentik. Rendezzük el ezeket az értékeket úgy, hogy az egyes értékekre teljesüljön, hogy ne legyenek nagyobbak az őket követőnél. (Természetesen ezt nem értelmezhetjük a sorozat utolsó  $a_n$  elemére, hiszen annak nincsen rákövetkezője.)

Kézenfekvőnek tűnik, hogy nevezzük ilyenkor az  $a_i$ -k sorozatát növekvőnek. Matematikai jelöléssel:

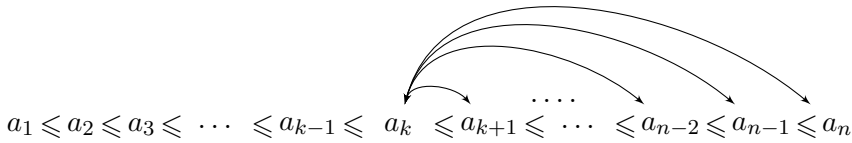
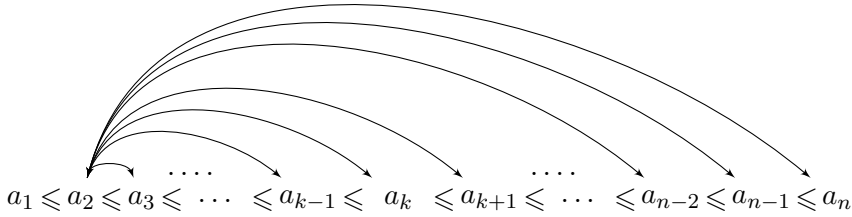
Monoton növekvőnek (nem csökkenőnek) nevezzük az  $(a_n)$  sorozatot, ha  $\forall i, j \in \mathbb{N}$ , amikor  $(1 \leq i < n)$  és  $(i < j)$  esetén  $a_i \leq a_j$  teljesül.



$i = 1$  esetén ez azt jelenti, hogy az  $a_1$  kisebb vagy egyenlő az öt követő összes  $n - i$ , azaz  $n - 1$  elemtől.

Ha pedig  $i = 2$ , akkor a kiválasztott elem az összes öt követő  $n - i$ , azaz  $n - 2$  elemtől nem lehet nagyobb.

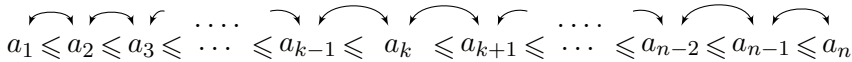




$i = k$  esetén az  $a_k$ -nak már csak az öt követő  $n - k$  elemmel szemben van ilyen „kötelezettsége”. Az utolsó előtti  $a_{n-1}$ -nek természetesen csak az öt követő  $a_n$ -tól kell kisebbnek vagy egyenlőnek lennie.

Ez tehát azt jelenti, hogy a növekvően rendezett sorozatban egyetlen elem sem lehet nagyobb az öt követőtől. Úgy is megfogalmazhatnánk, hogy nagyobb sorszámú elem értéke nem lehet kisebb egy kisebb sorszámú elem értékénél.

Ezzel ekvivalens az a megfogalmazás is, hogy az ilyen sorozatokban bármely elem értéke kisebb vagy egyenlő az öt követő elem értékénél, azaz  $\forall i \in \mathbb{N}, \quad \text{amikor } (1 \leq i < n), \quad a_i \leq a_{i+1}$  teljesül.



Bár az egyik megfogalmazásból következik a másik, és fordítva, azaz matematikai értelemben ekvivalensek. Ennek megfelelően, ha egy sorozat eleget tesz az egyiknek, akkor a másiknak is megfelel. Ennek ellenére mégis érdemes elgondolkodni azon, hogy az algoritmizálás során melyiket célszerű alkalmaznunk.

Legyen most feladatunk annak eldöntése, hogy egy sorozat vajon növekvően rendezett-e?

1. Az első esetben ennek eldöntéséhez a sorozat minden elemét (az utolsó kivételével) össze kell hasonlítani a nála nagyobb sorszámúakkal. Ez összesen

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

összehasonlítást jelent, hiszen összehasonlítások számának megadásához a természetes számokat kell összegeznünk 1-től  $(n-1)$ -ig.

2. A második esetben csupán azt kell megvizsgálnunk, hogy az egyes elemek az utánuk következővel milyen „viszonyban” vannak. Mivel az utolsónak nincs rákövetkezője, ez csak

$$n-1$$

összehasonlítást jelent.

Feltételezhetjük, hogy a számítógép számára az egyes összehasonlításokat jelentő logikai kifejezések kiértékeléséhez  $t$  időre van szüksége. A fentiekből könnyen látható, hogy az első esetben a teljes kiértékeléshez szükséges idő,

$$\frac{n(n-1)}{2}t,$$

ami  $n^2$ -tel arányos, míg a másik esetben ez az idő csupán

$$(n-1)t,$$

ami lineáris függvénye a sorozat elemszámának.<sup>16</sup>

Szigorúan monoton növekvőnek akkor nevezzük egy sorozatot, amikor az elemek között éles egyenlőtlenség teljesül ( $a_i < a_{i+1}$  illetve  $a_i < a_j$ ).

<sup>16</sup> Természetesen ez az idő lehet lényegesen kevesebb is abban az esetben, ha a sorozat nem rendezett, hiszen ha például már az első két elem sorrendje nem megfelelő, akkor nincs is szükség további összehasonlításokra.

## 4.2. Keresés sorozatokban

Látni fogjuk majd, hogy ha bizonyos dolgokat valamely mérhető tulajdonságuk szerint sorba rakunk, akkor a velük való bizonyos művelet egyszerűbben végezhetőek el. Máskor azonban ezt nem tehetjük meg, mert az elemek sorrendjének megváltoztatásával értékes információkat veszítenénk. A fentiek miatt fontos foglalkoznunk a rendezett és a rendezetlen sorozatok műveleteivel is.

### 4.2.1. Eldöntés

Adott az  $n$  elemű  $a$  sorozat és a sorozat elemein értelmezett  $T$  tulajdonság. Az algoritmus aszerint rendel a sorozathoz Igaz vagy Hamis logikai értéket, hogy a sorozatnak van vagy nincs  $T$  tulajdonságú eleme.

```
1 Függvény Eldöntés( $\underline{A}$ :sorozat):logikai;  
2 Változó  
3   I: egész;  
4 Algoritmus  
5   I  $\leftarrow$  1;  
6   Ciklus_Míg (I  $\leq$  N)  $\wedge$  (Nem T( $\underline{A}$ [I]))  
7     I  $\leftarrow$  I + 1;  
8   CVége;  
9   Eldöntés  $\leftarrow$  (I  $\leq$  N);  
10 FVége;
```

Eldöntés

### 4.1. algoritmus

A 4.1. algoritmus mindaddig vizsgálja a sorozat elemeit az elsővel kezdve, míg a sorozat elemei el nem fogynak, vagy nem talál egy  $T$ -tulajdonságút. Mivel a 6. sorban egy elől tesztelő feltételes ciklus található, a ciklusmag mindaddig végrehajtódik, amíg a ciklusfeltétel értéke Igaz. A kifejezés elemei közötti  $\wedge$  művelet miatt ez akkor következhet be, ha már túlhaladtunk

a sorozat végén ( $I > N$ ), vagy ha az aktuális pozíción  $T$ -tulajdonságú elemet találtunk. Ha tehát a ciklusból kilépve  $I > N$  teljesül, akkor az csak azt jelentheti, hogy a sorozatnak nincs  $T$ -tulajdonságú eleme, mert különben már hamarabb kilépett volna a ciklusból.

#### 4.2.2. Lineáris keresés

A 4.1. algoritmus értelmezésével könnyen belátható, hogy nem használtunk ki minden információt, amely az algoritmus végrehajtása során keletkezett. Gondoljuk csak el, hogy ha a ciklusból azért léptünk ki mert  $T$ -tulajdonságú elemet találtunk, az  $I$  értéke őrzi annak előfordulási helyét. Ennek értékét adja vissza a lineáris keresés algoritmus (a 4.2).

```
1 Függvény LinKer(A:sorozat, Hely:egész):logikai;  
2 Változó  
3   I: egész;  
4 Algoritmus  
5   I ← 1;  
6   Ciklus_Míg (I ≤ N)  ∧  (Nem T(A[I]))  
7     I ← I + 1;  
8   CVége;  
9   Hely ← I;  
10  LinKer ← (I ≤ N);  
11 FVége;
```

Lineáris keresés

4.2. algoritmus

#### 4.2.3. Kiválasztás

Adott az  $n$  elemű  $a$  sorozat és a sorozat elemein értelmezett  $T$  tulajdonság. Biztosan tudjuk, hogy a sorozatnak van  $T$  tulajdonságú eleme. Az algoritmus

kimenete a T tulajdonságú elem helye<sup>17</sup>.

```
1 Eljárás Kiválasztás(A:sorozat , Hely:egész);
2 Változó
3   I: egész;
4 Algoritmus
5   I ← 1;
6   Ciklus_Míg (Nem T(A[I]))
7     I ← I + 1;
8   CVége;
9   Hely ← I;
10 EVége;
```

Kiválasztás

### 4.3. algoritmus

Látható hasonlóságokon túl a 4.3 algoritmus jelentős eltéréssel is rendelkezik a 4.1 és a 4.2 algoritmusokhoz képest. Mivel a sorozatnak van T-tulajdonságú eleme, így biztosan találunk egyet, mielőtt az  $I > N$  teljesülne, ezért annak vizsgálata, hogy I értékével sorozatbeli elemre hivatkozunk-e, szükségtelen.

#### 4.2.4. Strázsás keresés

A kiválasztás tételének (4.3. algoritmus) tapasztalatait fölhasználva a lineáris keresést (4.2. algoritmus) hatékonyabbá tehetjük. Ha feltételezzük, hogy lineáris keresés ciklusfeltételének kiértékeléséhez 2, a ciklusmag utasításának végrehajtásához 1 időegységre van szükség, akkor belátható, hogy a kiválasztás tételének végrehajtási ideje kevesebb a lineáris keresésénél. Ez azt jelenti, hogy ha a sorozatról biztosan tudnánk, hogy tartalmaz T-tulajdonságú elemet, akkor azt gyorsabban lennénk képesek megtalálni azt. Ilyen információval általában nem rendelkezünk, de az adatszerkezet megfelelő kiala-

<sup>17</sup> Amennyiben a sorozatban több T tulajdonságú elem is található, akkor ezek közül az első előfordulási helyet adja vissza.

kításával általában lehetőségünk van a sorozat elemein kívül még egy elem számára helyet biztosítani. Tároljunk az adatszerkezet  $(n + 1)$ . pozícióján egy T-tulajdonságú elemet (4.4. algoritmus 6. sor). Így ha az eredeti sorozat nem is, de az úgynevezett strázsaelemmel kibővített sorozat biztosan tartalmazza a keresett elemet. Ami a strázzás keresés hatékonyságát illeti, a tapasztalat valóban azt mutatja, hogy a futási ideje megközelítőleg  $\frac{2}{3}$ -a a lineáris keresésének.

```

1 Függvény StrázzásKer(A:sorozat, Hely:egész): logikai;
2 Változó
3   I: Egész;
4 Algoritmus
5   I ← 1;
6   A[N + 1] ← KeresettTulajdonságúElem;
7   Ciklus_Míg (Nem T(A[I]))
8     I ← I + 1;
9   CVége;
10  StrázzásKer ← (I ≤ N);
11  Hely ← I;
12 FVége;
```

Strázzás keresés

#### 4.4. algoritmus

##### 4.2.5. Lineáris keresés rendezett sorozatban

A kereső algoritmusok leállási feltételét úgy is megfogalmazhatnánk, hogy nincs értelme tovább keresni, ha megtaláltuk a sorozat keresett elemét, illetve biztosan tudjuk azt mondani az eddig megvizsgált elemek alapján, hogy a sorozat nem tartalmazza azt. Ezt az utóbbi kijelentést az előző algoritmusainkban csak akkor tehetjük, a sorozat minden elemét megvizsgáltunk.

Más a helyzet rendezett sorozat esetében (ha a keresés kulcsa megegyezik a rendezés kulcsával). Tételizzük föl, hogy egy növekvően rendezett sorozat

nem tartalmazza a keresett elemet, de van a keresettnél kisebb és nagyobb eleme is. Ha sorozat elemeit az elsőtől kezdve vizsgáljuk, előbb-utóbb megtaláljuk a keresett elemnél nagyobb elemek közül a legkisebbet. Itt a keresést be is fejezhetjük, hiszen az ezt követő elemek az aktuálisnál még nagyobbak. A rendezett sorozaton való lineáris keresés 4.5. algoritmusáé tehát akkor fog megállni, ha az alábbiak közül egyik teljesül:

1. megtalálta a keresett elemet,
2. a keresett elemnél már nagyobb az aktuális elem,
3. „elfogytak” a sorozat elemei, azaz  $I > N$  teljesül.

(Jegyezzük meg, hogy  $I > N$  csak akkor következhet be, ha keresett elemet a sorozat nem tartalmazza és az nagyobb a sorozat utolsó eleménél is.)

```

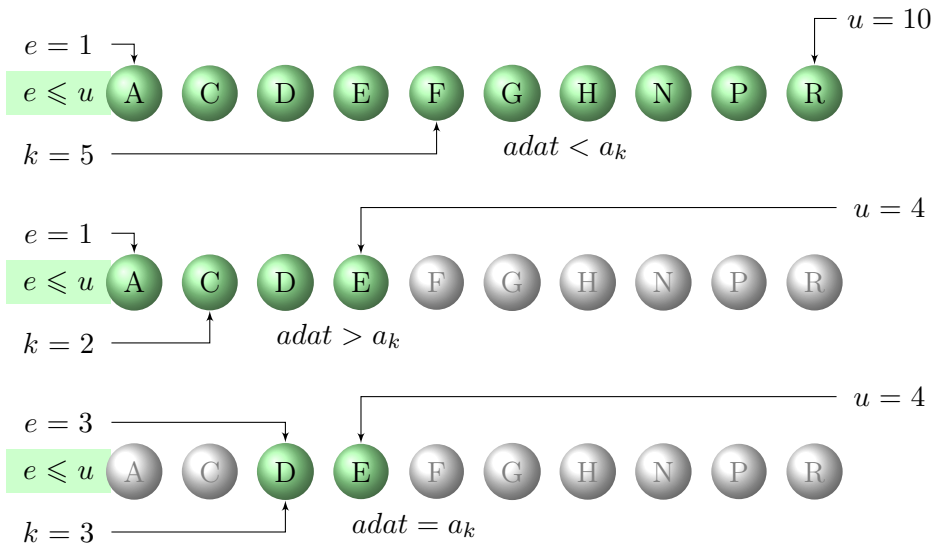
1  Függvény LinKerRend(A:sorozat , Adat:elemtip , Hely:
    egész):logikai;
2  Változó
3    I: egész;
4  Algoritmus
5    I ← 1;
6    Ciklus_Míg ((I ≤ N) ∧ (Adat > A[I]))
7      I ← I + 1;
8    CVége;
9    Hely ← I;
10   LinKerRend ← ((I ≤ N) ∧ (Adat = A[I]));
11  FVége;
```

Lineáris keresés rendezett sorozatban

#### 4.5. algoritmus

##### 4.2.6. Bináris keresés

A következő kereső algoritmusnak szintén a sorozat rendezettsége az előfeltétele. Működését talán azzal a gyakorlati példával lehetne szemléltetni, aho-



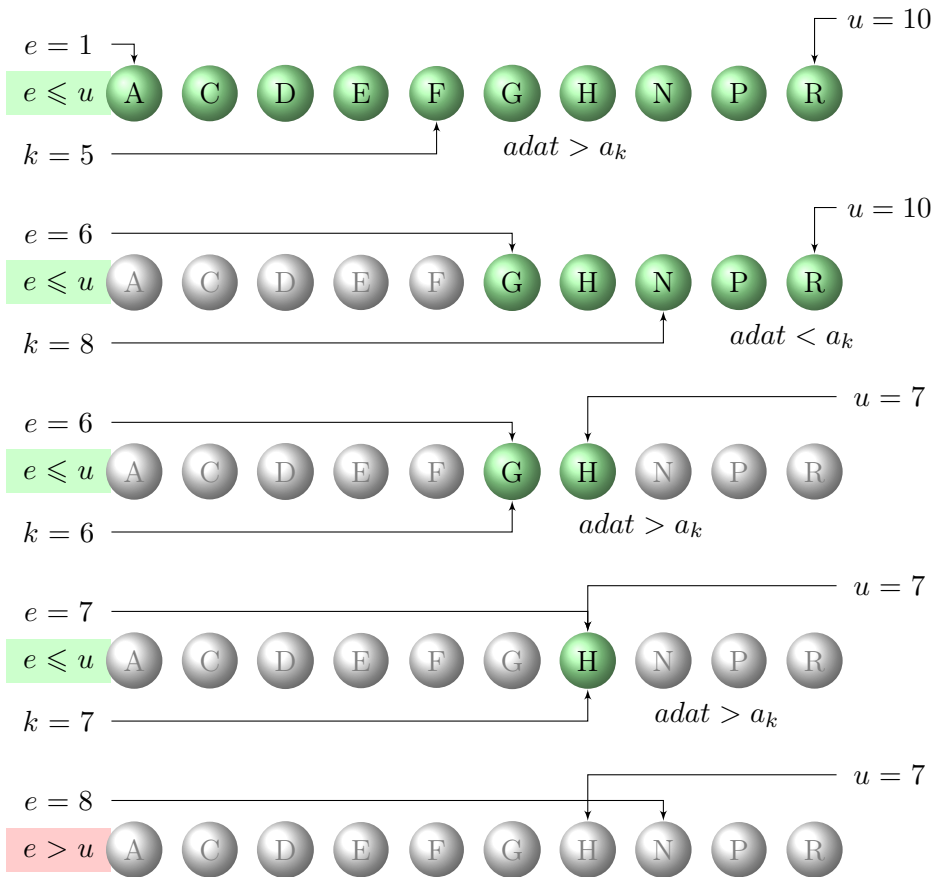
4.1. ábra. A keresett  $\text{adat} = \text{„D”}$

gyan egy lexikonban megkeresünk egy szócikket. A lexikonban való keresés nem lenne túlságosan hatékony, ha lineáris keresés algoritmusát követnénk. Helyette általában inkább úgy járunk el, hogy ha az aktuális szócikk nem egyezik meg a keresettel, akkor megpróbáljuk megbecsülni, hogy a keresett mennyivel lehet az aktuális előtt vagy után.

Tételezzük föl tehát, hogy a sorozat növekvően rendezett. Válasszuk ki a sorozat középső elemét. Amennyiben az nem a keresett elem, akkor összehasonlítva a keresett elemmel, el tudjuk dönteni, hogy a keresést a középső elemet követő, vagy az azt megelőző elemek részsorozatán van értelme tovább folytatni, ahogyan ezt a 4.1. ábra is szemlélteti.

Az első vizsgálatot követően vagy megtaláltuk az elemet, vagy teljes biztonsággal kizárhatjuk a további keresésből megközelítőleg az elemek felét. Hasonló módon eljárva a maradék, még ki nem zárt elemek részsorozatával, ha nem találtuk meg a keresett elemet az adott rész közepén, most az eredeti sorozat megközelítőleg negyedét zárhatjuk ki a további keresésből. A 4.1. és a 4.2. ábrákon jól látható, hogy a további keresést mindig egyre csökkenő – megközelítőleg feleződő – elemszámú sorozaton folytatjuk tovább





4.2. ábra. A keresett  $adat = „J”$

mindaddig, amíg vagy meg nem találjuk a keresett elemet, vagy a vizsgálatra kijelölt részsorozat elemszáma nullává nem válik. Ezt az  $E$  és  $U$  változók  $E > U$  relációja jelzi.

```

1  Függvény BinKer(A:sorozat, Adat:elemtip, Hely: egész):
    logikai;
2  Változó
3    E, U, K: egész;
4  Algoritmus
5    E ← 1;
6    U ← N;
7    K ← (E + U) div 2;
8    Ciklus_Míg (E ≤ U ∧ A[K] ≠ Adat)
9      Ha Adat < A[K] akkor
10        U ← K - 1;
11      Különbén
12        E ← K + 1;
13      HVége;
14      K ← (E + U) div 2;
15    CVége;
16    Hely ← K;
17    BinKer ← (E ≤ U);
18  FVége;

```

Bináris keresés

#### 4.6. algoritmus

### 4.3. Rendezés

Hasonlítsuk most össze egy sorozat első elemét rendre az összes rákövetkezőjével, és ha az első elemhez képest az azt követők között kisebbet találunk, akkor cseréljük meg a két összehasonlított elemet. Ez a műveletsor biztosítja azt, hogy a sorozat legkisebb eleme az első helyre kerüljön (4.7. algoritmus).

```

1  Változó
2  I: egész;      // ciklusváltozó
3  J: egész;      // j. helyre kerül majd a legkisebb elem
4  C: elemtip;    // segédváltozó két elem cseréjéhez
5  Algoritmus
6  J ← 1;        // j kezdő értéke legyen 1.
7                // Innen számítva
8                // dolgozzuk föl a sorozatot.
9  Ciklus I ← (J + 1..N) // Vesszük a rákövetkező elemeket...
10     Ha A[I] < A[J] akkor // HA az i. elem kisebb...
11         C ← A[I];        // ... a sorozat aktuális (i.)
12         A[I] ← A[J];      // és j. elemének cseréje,
13         A[J] ← C;        // de j = 1.
14     HVége;
15     CVége;
16         // A sorozat legkisebb eleme
17         // biztosan az 1. (azaz a j.) helyre került.

```

A legkisebb elem az első helyre kerül (közvetlen kiválasztás)

#### 4.7. algoritmus

Miután a legkisebb elem az első helyre került, a második elemmel kezdődő résszel ugyanezt végrehajtva a második helyre kerül majd a sorozat második legkisebb eleme is (4.8. algoritmus).

```

1  Változó
2  I: egész;      // ciklusváltozó
3  J: egész;      // j. helyre kerül majd a második legkisebb
4  C: elemtip;    // segédváltozó két elem cseréjéhez
5  Algoritmus
6  J ← 2;        // j kezdő értéke legyen 2.
7                // Innen számított részét dolgozzuk föl
8                // a sorozatnak, tehát az első elemet már nem.
9  Ciklus I ← (J + 1..N) // Vesszük a rákövetkező elemeket...
10     Ha A[I] < A[J] akkor // HA az i. elem kisebb...
11         C ← A[I];        // ...a sorozat aktuális (i.)
12         A[I] ← A[J];     // és j. elemének cseréje,
13         A[J] ← C;        // de j = 2.
14     HVége;
15     CVége;
16         // A részsorozat második legkisebb eleme
17         // biztosan a 2. (azaz a j.) helyre került.

```

A második legkisebb elem a második helyre kerül  
(közvetlen kiválasztás)

#### 4.8. algoritmus

Most már a sorozat első két eleme a végleges helyére került, hiszen az aktuális részsorozat (kezdetben a teljes sorozat azt követően pedig végleges helyükre került elemek figyelmen kívül hagyásával nyert részsorozat) legkisebb elemét az első helyre juttattuk. Hasonló megfontolások miatt a sorozat  $j$ -edik eleme úgy kerülhet a végleges helyére, ha előtte már  $j - 1$  részsorozatra a fenti műveletet elvégeztük.

```

1 Eljárás RendKözv(A: sorozat);
2 Változó
3   I: egész;      // ciklusváltozó
4   J: egész;      // j. helyre kerül majd a legkisebb elem
5   C: elemtip;    // segédváltozó két elem cseréjéhez
6 Algoritmus
7   Ciklus J ← (1..N-1)      // a j. a részsorozat 1. eleme
8       // Innen számítva dolgozzuk föl a sorozatot.
9   Ciklus I ← (J + 1..N)    // Vegyük a rákövetkező
10      Ha A[I] < A[J] akkor // elemeit a j. részsorozatnak...
11          C ← A[I];        // HA az i. elem kisebb...
12          A[I] ← A[J];     // ... a sorozat aktuális (i.)
13          A[J] ← C;        // és j. elemének cseréje,
14      HVége;
15      CVége;
16          // A j. részsorozat legkisebb eleme
17          // biztosan a j. helyre került.
18  CVége;
19 EVége;

```

Sorozat rendezése közvetlen kiválasztással

#### 4.9. algoritmus

Könnyen belátható, hogy utoljára az  $a_{n-1}$  és az  $a_n$  által alkotott kételemű részsorozat esetében értelmezhetjük a fenti műveletet.

A 4.9. algoritmusban tehát a J változó aktuális értéke mutatja meg, hogy a sorozat hányadik pozíciójára kerül a belső ciklus által kiválasztott elem, egy újabb elemmel bővítve a sorozat rendezett részét.

Az algoritmus hatékonyságát az alábbiak szerint összegezzük. A rendezés során  $n - 1$  elemet kell a végleges helyére juttatni az elsőtől az utolsó előttiig úgy, hogy közben az aktuális rész első elemét összehasonlítsuk az összes nála nagyobb sorszámúval, és ha szükséges, akkor megcseréljük őket.

Amikor a legkisebb elemet juttatjuk az első helyre ( $J=1$ ), akkor az  $a_1$ -et össze kell hasonlítani az össze rákövetkező  $n - 1$  darab elemmel. A  $J$  utolsó értéke  $n - 1$  lesz, ami azt jelenti, hogy ahhoz, hogy a második legnagyobb elem a helyére kerüljön, csak 1 összehasonlítást kell végeznünk. Az összes összehasonlítások számát tehát

$$\sum_{c=1}^{n-i} c$$

alakban adhatjuk meg.

Arra való tekintettel, hogy a nem minden összehasonlítás eredményezi az összehasonlított elemek cseréjét és a két elem cseréje három értékadással oldható meg, az értékadások száma legfeljebb

$$3 \sum_{c=1}^{n-i} c.$$

Eleve rendezett sorozat esetén természetesen nem történik egyetlen értékadás sem.

A rendezés során a sorozat elemein tárolásán kívül szükség van még egy elem típusú tárhelyre, tehát az algoritmus tárigénye

$$n + 1.$$

Könnyű látni, hogy ha a 4.9. algoritmus a sorozat  $j$ . pozíciójára eljuttat egy elemet, ha a későbbiekben ennél kisebbet talál, akkor azt ezzel ismét megcseréli. Ez azt jelenti, mire a megfelelő elemet sikerül a helyére juttatni, addig esetlegesen több fölösleges cserét is végre fogunk hajtani. Mindez kiküszöbölhető volna úgy, ha – megőrizve a rendező algoritmus alapkonceptióját – először kiválasztanánk az aktuális részsorozat legkisebb elemét és ezt követően már legfeljebb csak egy cserét kellene végezni. Így tehát a 4.10. minimumkiválasztással történő rendezésre tekinthetünk a 4.9. algoritmus hatékonyabb változatának is.

```

1 Eljárás RendMin(A: sorozat);
2 Változó
3   MinI: egész; // a mindenkori legkisebb elem sorszáma
4   I: egész;    // ciklusváltozó a minimum kiválasztásához
5   J: egész;    // j. helyre kerül majd a legkisebb elem
6   C: elemtip;  // segédváltozó két elem cseréjéhez
7 Algoritmus
8   Ciklus J ← (1..N-1) // A j. a részsorozat 1. eleme
9       // A j. elemtől a sorozat végéig terjedő
10      // részre valósul meg a minimum kiválasztása.
11   MinI ← J;
12   Ciklus I ← (J + 1..N) // Vegyük a rákövetkezőket...
13       Ha A[I] < A[MinI] akkor
14           MinI ← I;
15           // HA az eddigi legkisebbnél kisebbet találtunk,
16           // annak a sorszámát jegyezzük meg.
17       HVége;
18   CVége;
19       // A j. részsorozat legkisebb elemének
20       // sorszáma MinI-ben van...
21   Ha MinI ≠ J akkor
22       C ← A[MinI]; // Ha mégsem a j. elem a legkisebb,
23       A[MinI] ← A[J]; // akkor meg kell cserélni.
24       A[J] ← C;
25   HVége;
26       // A j. részsorozat legkisebb eleme
27       // biztosan a j. helyre került.
28   CVége;
29   EVége;

```

Rendezés minimum kiválasztásával

4.10. algoritmus

A minimumkiválasztással történő rendezés során az alapkoncepció miatt – ez jól látható a 4.9. és éfalg:rendmin. algoritmus szerkezetének összehasonlításakor is – ugyanannyi összehasonlítást kell végezni, mint a közvetlen kiválasztással történő rendezés során.

Az adatmozgatások tekintetében már lényegesen kedvezőbb lehet a helyzet, hiszen a 4.10. algoritmus belső ciklusa – a módszer nevéhez híven – lényegében egy minimumkiválasztás. Valójában tehát az algoritmus meghatározza minden egyes részsorozat legkisebb elemének helyét, és azt megcseréli az adott részsorozat első, azaz a  $j$ . részsorozat esetében a teljes sorozat  $j$ . elemével. Ez azonban azt jelenti, hogy az értékadások maximális száma

$$3(n - 1)$$

lehet.

A tárigény tekintetében szintén nincs különbség az előzőhöz képest, az elemek tárolásán túl, a csere elvégzéséhez szintén szükséges egy további elem típusú segédváltozó. Tehát az algoritmus tárigénye

$$n + 1.$$

Korábban láttuk, hogy a sorozatok rendezettsége ellenőrizhető a szomszédos elemek összehasonlításával. Nevezetesen, ha bármely két szomszédos elem sorrendje megfelelő, akkor a sorozat is rendezett. Erre épít a buborékrendezés 4.11. algoritmus.

Az algoritmus először a sorozat elejétől kezdve minden elemet összehasonlít a rákövetkezőjével, és ha a sorrendjük nem megfelelő, akkor meg is cseréli azokat. Ez a művelet sor  $n - 1$  összehasonlítás és legfeljebb ugyanennyi csere árán biztosítja, hogy a sorozat legnagyobb eleme a sorozat végére kerüljön. Ezt követően már csak – az utolsó elem figyelmen kívül hagyásával – az első  $n - 1$  elemmel kell megismételnünk a műveletsort, miközben  $n - 2$  összehasonlítást és legfeljebb ugyanennyi cserét végzünk el. Ekkorra a sorozat második legnagyobb eleme került a végleges helyére.

Hasonló módon a sorozat többi eleme is végleges helyére „buborékolat-



ható”. Utolsóként a sorozat első két eleméből álló részsorozatban végezve a műveletsort a második és az első elem is helyére kerül.

```
1 Eljárás RendBuborék(A: sorozat);
2 Változó
3   I: egész;      // ciklusváltozó
4   J: egész;      // a feldolgozandó részsorozat sorszáma
5   C: elemtip;    // segédváltozó két elem cseréjéhez
6 Algoritmus
7   Ciklus J ← (1..N-1)      // a j. részsorozat feldolgozása
8       // Az elejétől az  $n - j$ . járjuk be a sorozatot.
9   Ciklus I ← (1..N-J)
10      Ha  $\underline{A}[I] > \underline{A}[I + 1]$  akkor
11          C ←  $\underline{A}[I]$ ;      // Ha a szomszédok sorrendje...
12           $\underline{A}[I] \leftarrow \underline{A}[I + 1]$ ;    // ...nem megfelelő, az elemek
13           $\underline{A}[I + 1] \leftarrow C$ ;      // ( $i$ . és  $i + 1$ .) cseréje szükséges
14      HVége;
15  CVége;
16      // A j. részsorozat legnagyobb eleme
17      // biztosan az  $n - j + 1$ . helyre,
18      // a részsorozat végére került.
19  CVége;
20 EVége;
```

Buborékredezés

#### 4.11. algoritmus

A fentiek alapján és a 4.11. algoritmus értelmezése alapján az olvasóra bízunk annak belátását, hogy a buborékredezés hatékonyságát tekintve megegyezik a közvetlen kiválasztással történő rendezés (4.9.) algoritmus hatékonyságával.

```

1 Eljárás RendBuborék_J1(A: sorozat);
2 Változó
3   I: egész;           // ciklusváltozó
4   J: egész;           // a feldolgozandó részsorozat sorszáma
5   C: elemtípus;       // segédváltozó két elem cseréjéhez
6   VoltCsere: logikai;
7                       //
8 Algoritmus
9   J ← 1;
10  VoltCsere ← Igaz;
11  Ciklus_Míg (J ≤ N-1) ∧ VoltCsere
12    VoltCsere ← Hamis;
13    // a j. részsorozat feldolgozása
14    // Az elejétől az n - j. járjuk be a sorozatot.
15    Ciklus I ← (1..N-J)
16      Ha A[I] > A[I + 1] akkor
17        C ← A[I];           // Ha a szomszédok sorrendje...
18        A[I] ← A[I + 1];     // ...nem megfelelő, az elemek
19        A[I + 1] ← C;       // (i. és i + 1.) cseréje szükséges
20        VoltCsere ← Igaz;  // Folytatni kell a rendezést
21      HVége;
22    CVége;
23    // A j. részsorozat legnagyobb eleme
24    // biztosan az n - j + 1. helyre,
25    // a részsorozat végére került.
26  J ← J + 1;
27  CVége;
28 EVége;

```

A buborékrendezés hatékonyabbá tehető, ha az algoritmus „fölismeri”, hogy a sorozat rendezetlen része is rendezetté vált már

#### 4.12. algoritmus

Ugyanakkor vegyük észre, hogy a buborékrendezés során azon túl, hogy a nagyobb értékű elemek sorozat vége felé gyorsan vándorolnak, a kisebbek a sorozat eleje felé mozdulnak el egy hellyel. Ez azt jelenti, hogy miközben a sorozat végén a rendezett rész elemszáma eggyel nőtt, addig az ettől jobbra lévő elemek rendezettsége is megváltozik, esetleg rendezetté is válhat. A 4.11. módosításával elérhető, hogy az algoritmus „felismerje” azt, ha a rendezettség a kisebb indexű elemek körében is bekövetkezett. Ekkor ugyanis a teljes sorozat rendezetté vált. Ezt a módosítást tartalmazza a 4.12. algoritmus.

A 4.12. javított buborék-rendezés „csak” azt képes fölismerni, hogy a sorozat baloldali, rendezésre váró része teljesen rendezetté vált. Ezt az jelzi, hogy belső ciklus lefutása során nem volt szükség cserére. Ugyanakkor előfordulhat, hogy belső ciklusban jóval a ciklusváltozó végértékének elérése előtt történt az utolsó csere, azaz a sorozat az utolsó csere helyétől már rendezett. Ezt képes fölismerni a buborékrendezés 2. javított 4.13. algoritmus.

```

1 Eljárás RendBuborék_J2(A: sorozat);
2 Változó
3   I: egész;      // ciklusváltozó
4   J: egész;      // a feldolgozandó részsorozat sorszáma
5   C: elemtip;    // segédváltozó két elem cseréjéhez
6   UtolsóCsereH: egész;
7                   // az utolsó csere helyét jelzi majd
8 Algoritmus
9   J ← 1;
10  Ciklus_Mig (J ≤ N-1)
11    UtolsóCsereH ← 0;
12    // a j. részsorozat feldolgozása
13    // Az elejétől az n - j. járjuk be a sorozatot.
14    Ciklus I ← (1..N-J)
15      Ha A[I] > A[I + 1] akkor
16        C ← A[I];      // Ha a szomszédok sorrendje...
17        A[I] ← A[I + 1]; // ...nem megfelelő, az elemek
18        A[I + 1] ← C;   // (i. és i + 1.) cseréje szükséges
19        UtolsóCsereH ← I; // Folytatni kell a rendezést
20      HVége;
21    CVége;
22    // A j. részsorozat legnagyobb eleme
23    // biztosan az n - j + 1. helyre,
24    // a részsorozat végére került.
25    J ← N - UtolsóCsereH;
26  CVége;
27 EVége;

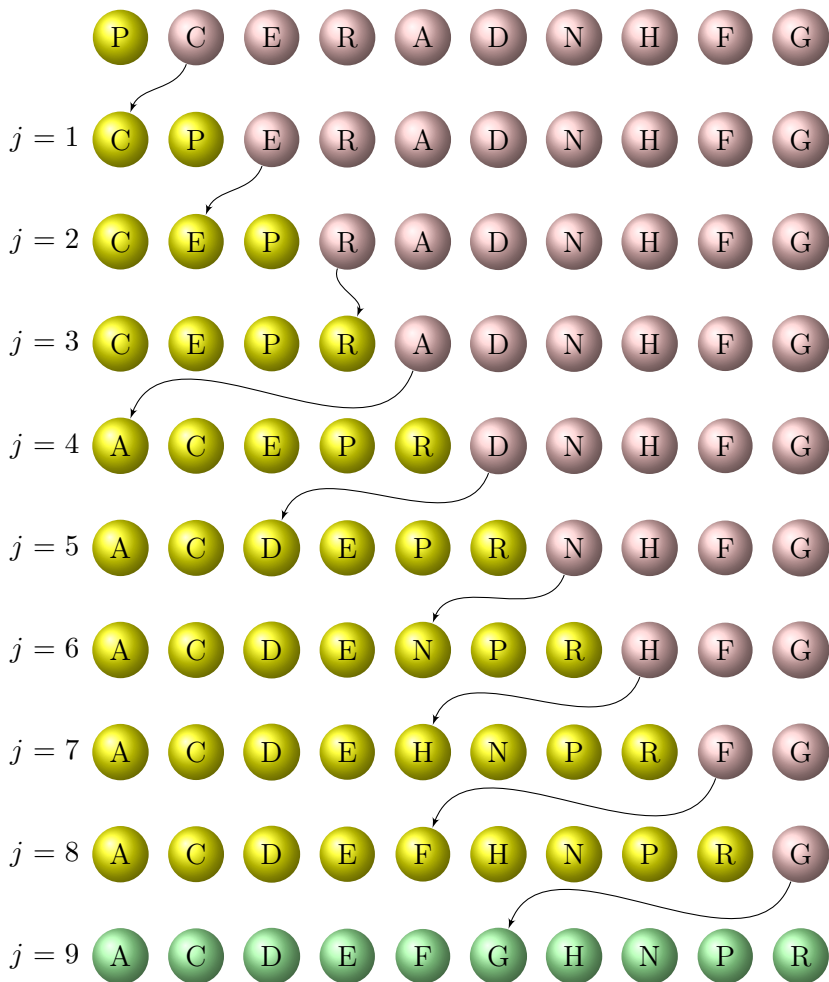
```

A buborékrendezés algoritmusában lehetőség van az utolsó csere helyének tárolására, így egynél több elem is csatolható a rendezett részsorozathoz

#### 4.13. algoritmus

### 4.3.1. Beszűrő rendezés

A beszűrő rendezés elve szerint mindig a következő elem helyét kell megkeresnünk egy már rendezett részsorozatban. Ezt egy  $n$  elemű sorozat esetében  $n - 1$ -szer kell megtennünk, hiszen az  $a_1$ -et, mint egy egy elemű részsorozatot rendezettnek tekinthetjük, és az  $a_2$ -től a sorozat végéig összesen ennyi elem van, amelyeket be kell illeszteni a rendezés során az egyre bővülő rendezett részbe.



4.3. ábra. Egy 10 elemű sorozat rendezése beszűrő rendezéssel

Ezt szemlélteti a 4.3. ábra egy 10 elemű sorozat példáján. Piros színnel a rendezésre váró elemeket, zölddel a teljesen rendezett sorozatot jelöltük. Sárga a sorozat egyre bővülő rendezett részét jelöli, amelybe ha bele is kerül egy elem, koránt sem biztos, hogy az lesz a végleges helye. Nyíllal jelöltük, hogy a rendezetlen rész első elemét a rendezettek közé hová is illesztettük be. Az ábrán megfigyelhető, hogy a beillesztetett elemtől balra található sárgával jelzett elemek mindegyike az előző sorban elfoglalt helyükhöz képest egy pozícióval balra helyezkednek el. Így történhet meg például az, hogy az eredetileg a 4. helyen található „R” – bár viszonylag gyorsan a rendezett részbe kerül – a rendezés bejezésekor már a sorozat végére vándorolt.

A  $j$ -edik lépésben tehát a sorozat  $j+1$ -edik elemét szeretnénk a rendezett részsorozatban elhelyezni. Hogy ezt megtehessek, valójában két problémát kell megoldanunk.

1. Meg kell keresnünk azt a helyet, ahová „beillik” az  $a_{j+1}$ . Itt jegyezzük meg, hogy valószínűleg célszerű lesz majd figyelembe vennünk, hogy ezt a keresést egy rendezett sorozatban kell végeznünk, és tudjuk, hogy erre vannak hatékonyabb algoritmusaink.
2. Ugyanakkor problémát jelent, hogy nem elég megtalálnunk azt a helyet, ahol a beillesztendő elemnek lennie kellene, hiszen az a hely már egészen biztosan „foglalt”, hanem helyet is kell biztosítanunk a számára anélkül, hogy a sorozat elemei közül bármelyiket elveszítenénk.

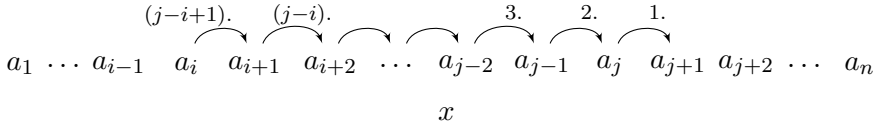
Tételezzük fel, hogy ismerjük annak az elemnek a sorszámát, amely elé be kell illesztenünk az  $a_{j+1}$ -et. Ha most kivennénk (pontosabban másolatot készítünk) a sorozatnak az  $a_{j+1}$ , azaz a beillesztendő eleméről (4.14 algoritmus 9. sora),

$$a_1 \quad \dots \quad a_{i-1} \quad a_i \quad a_{i+1} \quad a_{i+2} \quad \dots \quad a_{j-2} \quad a_{j-1} \quad a_j \quad a_{j+1} \quad a_{j+2} \quad \dots \quad a_n$$

$x \quad \longleftarrow$

akkor ezt követően az öt megelőző ( $a_j$ ) elemmel kezdve, a sorozat eleje felé haladva, az  $a_i$ -vel bezárólag az elemeket egy hellyel jobbra kell mozgatni

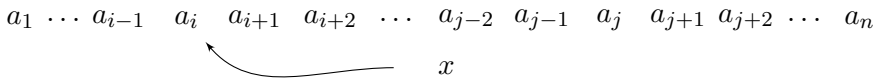
(4.14 algoritmus 12. sor). Ha az  $a_j$  az első ilyen elem az  $a_i$  pedig az utolsó, akkor ez összesen  $(j - i + 1)$  adatmozgatással valósítható meg.



A két szélső esetet figyelembe véve:

1. ha  $a_j \leq a_{j+1}$ , akkor nem kell az  $a_j$ -t föntebb „csúsztatni” mert a megfelelő helyen van.
2. ha  $a_{j+1} < a_1$ , akkor pedig  $j$  számú elem mindegyikét egy-egy hellyel jobbra kell léptetni.

Ilyen módon fölszabadul a hely a beillesztendő elem számára a megfelelő helyen, és ekkor helyére kerülhet az  $x$ -ben átmenetileg tárolt elem (4.14 algoritmus 15. sor).



```

1 Eljárás RendBeszűrő(A: sorozat);
2 Változó
3   I: egész;      // ciklusváltozó
4   J: egész;      // a feldolgozandó részsorozat sorszáma
5   X: elemtip;    // segédváltozó elem átmeneti tárolására
6 Algoritmus
7   Ciklus J ← (1..N-1) // n-1 elemet kell beilleszteni
8     I ← J;
9     X ← A[J + 1]; // a j+1-edik elem helyét keressük
10           // a j-ediktől visszafelé a rendezett részben
11   Ciklus_Míg (I ≥ 1) ∧ (A[I] > X)
12     A[I + 1] ← A[I]; // az i-edik mozgatása egygyel föntebb
13     I ← I - 1;      // a sorozat eleje felé haladva keresünk
14   CVége;
15   A[I + 1] ← X; // X beillesztése a rendezett részbe
16   EVége;
17 EVége;

```

Beszűrő rendezés

#### 4.14. algoritmus

#### 4.3.2. Shell rendezés

Tekintsük az alábbi rendezetlen sorozatot.

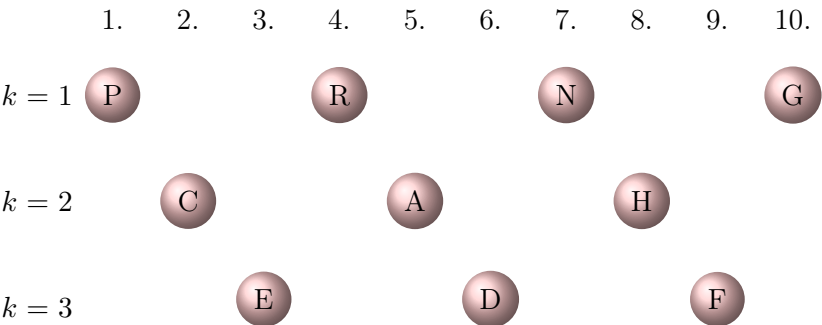


Válasszunk a sorozat elemszámától függő  $d$  egész értéket, és bontsuk föl a sorozatot  $d$  számú részsorozatokra az alábbi szempontok szerint:

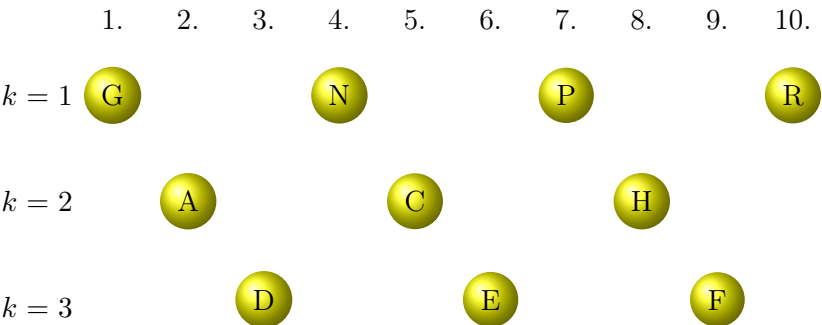
- egy sorozatba az eredeti sorozat egymástól  $d$  távolságra lévő elemei tartozzanak.
- az eredeti sorozat  $k$ -adik elemei (ahol  $1 \leq k \leq d$ ) rendre legyenek a  $k$ -adik sorozat első elemei.



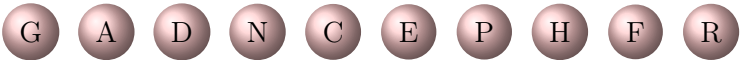
A fenti két feltétel biztosítja, hogy az eredeti sorozat minden eleme pontosan egy részsorozatába tartozhat. Példánkban kezdetben legyen  $d = 3$ .



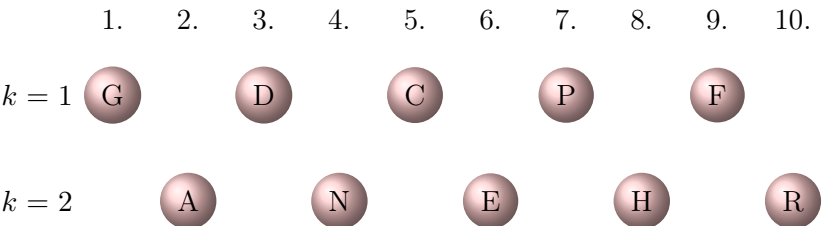
Végezzük el az így kapott részsorozatok rendezését valamely rendező algoritmussal külön-külön.



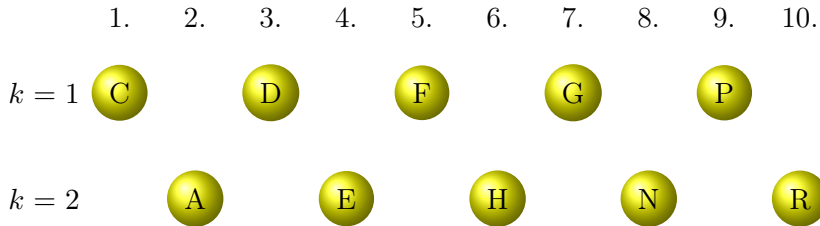
Ezt követően ha egy sorozatként tekintjük az elemeket, nem föltétlen kapunk rendezett sorozatot.



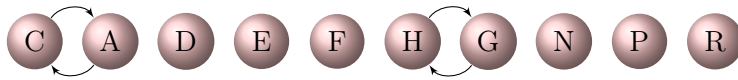
Csökkentsük most a  $d$  értékét és legyen  $d = 2$ .



Végezzük el az így nyert részsorozatok rendezését is az előzőekhez hasonló módon.



Ezt követően ha ismét egy sorozatként tekintünk az elemekre, azt figyelhetjük meg, hogy, az egyes elemek meglehetősen közel kerültek a végleges helyükhöz. Esetünkben ez azt jelenti, hogy csupán néhány szomszédos elem cseréje is elegendő volna a rendezettség eléréséhez.



Végezzük tehát el a rendezést  $d = 1$  esetére is.

Az alapgondolat szerint tehát kezdetben egymástól távolabbi elemeket hasonlítunk és mozgatunk, ami általában azt eredményezi, hogy az elemek gyorsabban közelítenek a végleges helyükhöz.

```

1 Eljárás RendShellMin(A: sorozat);
2 Változó
3   I, J, MinH: egész; // változók az alap rendezéshez
4   D: egész; // elemek közötti távolság
5   K: egész; // az aktuális, rendezendő sorozat sorszáma
6   C: elemtip; // segédváltozó elem átmeneti tárolására
7 Algoritmus
8   D ← N;
9   Ciklus
10      D ← D div 3 + 1;
11      Ciklus K ← 1..D
12          J ← K; 'az_"alap"-algoritmus_KEZDETE'
13          Ciklus_Míg J ≤ N-D
14              MinH ← J; I ← J + D;
15              Ciklus_Míg I ≤ N
16                  Ha A[I] < A[MinH] akkor
17                      MinH ← I;
18                      HVége;
19                      I ← I + D;
20                      CVége;
21                      Ha J ≠ MinH akkor
22                          C ← A[J]; A[J] ← A[MinH]; A[MinH] ← C;
23                          HVége;
24                          J ← J + D;
25                      CVége; 'VÉGE_az_"alap"-algoritmusnak'
26      CVége;
27      CVége_Amikor D=1;
28 EVége;

```

Shell

#### 4.15. algoritmus

### 4.3.3. Összefuttatás

A rendezett sorozatok feldolgozása során szükségessé válhat két (azonos módon) rendezett sorozat elemeinek egy rendezett sorozatban való egyesítése. Kézenfekvő megoldásnak tűnne a sorozat elemeit egy adatszerkezetben egyszerűen egymás után másolni, majd az így kapott, most már rendezetlen sorozatot az egyik korábban megismert rendező algoritmussal rendezni.

Az összefuttatás tétele egy ennél jóval hatékonyabb megoldást kínál. Tételizzük fel, hogy mindkét sorozat növekvően rendezett, és nem föltétlen azonos elemszámúak. Az ilyen elrendezésből az következik, hogy a sorozatok első elemei ( $a_1$  és  $b_1$ ) egyben a sorozaton belül a legkisebbek is. Az is nyilvánvaló, ha az egyesített sorozatnak szintén növekvően rendezettnak kell lennie, akkor annak első eleme csak a  $\text{Min}(a_1, b_1)$  lehet. A következő lépésben már az egyik sorozat első és másik sorozat második elemének összehasonlításával dönthetjük el, hogy melyik elem kerüljön az új sorozatban.

Általánosan megfogalmazva: vezessük be mindkét sorozat esetében az aktuális elem fogalmát. Kezdetben a sorozatok első elemei legyenek az aktuálisak.

- Végezzük el a két aktuális elem összehasonlítását, és a kisebbiket írjuk bele az egyesített sorozat elemeit tartalmazó adatszerkezet következő pozíciójába.
- Ezután abban a sorozatban, amelynek elemét az új sorozatba írtuk, a következő elemet tekintjük aktuálisnak.

A fenti lépések sorozatát végezzük mindaddig, amíg az egyik sorozat végére nem érünk. Ezt követően a másik sorozat maradék elemeit az egyesítés végére írhatjuk, hiszen azok az eddig feldolgozott összes elemnél nagyobbak. A fenti elvek alapján végzi két rendezett sorozat egyesítését a [4.16.](#) algoritmus.

```

1 Eljárás Összefuttatás(A, B, C: sorozat);
2 Változó AI, BI, CI: egész;
3 Algoritmus
4   AI ← 1; BI ← 1; CI ← 0;
5   Ciklus_Míg (AI ≤ A.elemszám) ∧ (BI ≤ B.elemszám)
6     CI ← CI + 1;
7     Ha A[AI] < B[BI] Akkor
8       C[CI] ← A[AI]; AI ← AI + 1;
9     Különbén
10      Ha A[AI] > B[BI] Akkor
11        C[CI] ← B[BI]; BI ← BI + 1;
12      Különbén
13        C[CI] ← A[AI]; AI ← AI + 1;
14        CI ← CI + 1;
15        C[CI] ← B[BI]; BI ← BI + 1;
16      HVége;
17    HVége;
18  CVége;
19  // A vagy B maradék elemeit C végére írjuk
20  Ciklus_Míg (AI ≤ A.elemszám)
21    CI ← CI + 1; C[CI] ← A[AI];
22    AI ← AI + 1;
23  CVége;
24  Ciklus_Míg (BI ≤ B.elemszám)
25    CI ← CI + 1; C[CI] ← B[BI];
26    BI ← BI + 1;
27  CVége;
28 EVége;

```

Összefuttatás

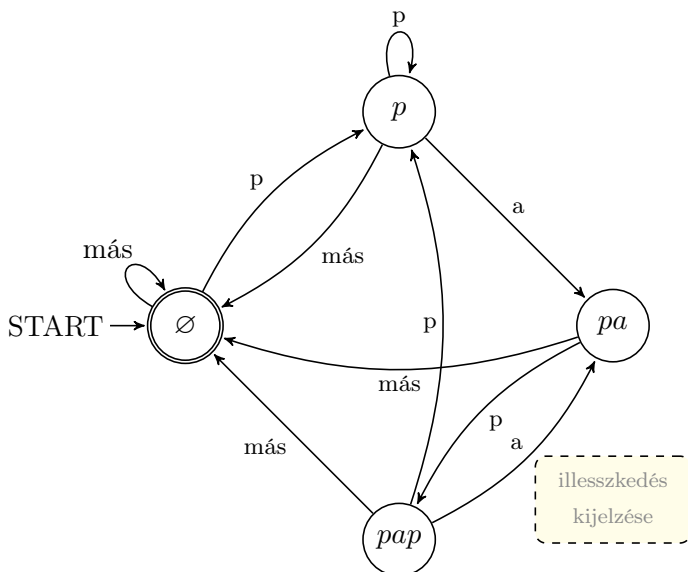
4.16. algoritmus

#### 4.4. Feladatok

1. Adjuk meg a fejezet leírónyelvű algoritmusait blokkdiagram formájában.
2. Döntsük el a sorozatról, hogy
  - a) növekvően,
  - b) csökkenőenrendezett-e?
3. Írjunk algoritmust, amely *Igaz* illetve *Hamis* értéket rendel az  $a_n$  sorozathoz attól függően, hogy az növekvően rendezett vagy sem.
4. Koordinátaival adott egy  $n$  elemű pontsorozat a síkban. Határozzuk meg, van-e olyan eleme, amely kívül esik egy adott  $r$  sugarú  $O(x_0, y_0)$  középpontú körön.
5. Koordinátaival megadott  $n$  elemű, síkbeli pontsorozat elemei jelentsek egy  $n$ -oldalú sokszög csúcspontjait. (A megadás sorrendje megfelel körüljárási iránynak.) Határozzuk meg, hogy van-e a sokszögnek egy adott  $d$ -nél hosszabb oldala.
6. Módosítsa a lineáris keresés algoritmusát úgy, hogy a keresést a sorozat utolsó eleménél kezdjük.
7. Módosítsa a strázsás keresés algoritmusát úgy, hogy a keresést a sorozat utolsó eleménél kezdjük.
8. Módosítsa a rendezett sorozatra értelmezett lineáris keresés algoritmusát úgy, hogy a keresést a sorozat utolsó eleménél kezdjük.
9. Módosítsa a rendezett sorozaton való lineáris keresés algoritmusát úgy, hogy ha tudjuk, hogy a sorozat monoton csökkenő.
10. Módosítsa a bináris keresés algoritmusát úgy, hogy ha tudjuk, hogy a sorozat monoton csökkenő.

11. Egészítse ki a tanult rendező algoritmusok programkódját a megfelelő utasításokkal, hogy az alkalmas legyen az aktuális rendezendő sorozat esetében a feltételvizsgálatok és az adatmozgatások számának meghatározására.
12. A megismert rendező algoritmusokat módosítsa úgy, hogy a sorozatok rendezettsége nem növekvő legyen.
13. A minimumkiválasztással való rendezés algoritmus alapján készítse el azt az algoritmust amely a maximumkiválasztásra épül és a rendezés végrehajtása után sorozat elemei
  - monoton növekvő
  - monoton csökkenősorozatot alkotnak.
14. A fentebb bemutatott buborékrendezés algoritmusának (4.11) nagy hátránya, hogy a sorozat végéhez közel található kis értékű elemek (teknősök) nagyon lassan „találják meg” helyüket a sorozat elején, míg a sorozat elején lévő nagy értékű „nyulak” jóval gyorsabban a végleges helyükre kerülnek. A 4.11 algoritmus alapján írjuk meg a „Koktél rendezés” algoritmusát, amely fölváltva buborékoltatja a nagy értékű elemeket a sorozat végére, majd a kicsiket az elejére.
15. A „Koktél rendezés” elvi alapja tehát fentebb bemutatott buborékrendezés (4.11). Ebből az is következik, hogy ennek az algoritmusnak is elvégezhető a „javítása”, ahogyan a buborékrendezésé.
  - A 4.12 algoritmus elve alapján módosítsa a „Koktél rendezés” algoritmusát úgy, hogy álljon le az algoritmus, ha a sorozat középső, „rendezetlen” része is rendezetté vált.
  - A 4.13 algoritmus elve alapján módosítsa a „Koktél rendezés” algoritmusát úgy, hogy ha lehetséges, akkor esetleg több elemet is a rendezett részhez csatol azáltal, hogy figyeljük benne az utolsó csere helyét.

16. Írjunk algoritmust és programot, amely egy hosszú szövegben megtalálja a „papa” szó előfordulásait.
17. Írjunk algoritmust és programot, amely a 4.4 ábrán látható gráf-modell alapján működik és egy hosszú szövegben megtalálja a „papa” szó előfordulásait.



4.4. ábra. Egy szövegben a *papa* szó előfordulásait kereső „állapot-gép” gráfmodellje

18. Könnyű látni, hogy egy növekvően rendezett sorozatban az  $i$ -edik elemet  $(i - 1)$  nála kisebb elem előz meg. Ha tehát egy rendezetlen sorozat valamely eleméről tudjuk, hogy a sorozatnak van  $(i - 1)$  eleme, amelyek nála kisebbek, akkor azt is tudjuk, hogy a rendezett sorozatban ennek az elemnek az  $i$ -edik helyre kell kerülnie.
  - Írjunk eljárást a fenti elvek alapján, amely az  $n$  elemű  $a$  rendezetlen sorozat elemeit egy vektor paraméterben kapja meg és egy másik vektorban rendezetten adja vissza.
  - Végezzük el az algoritmus hatékonyságának vizsgálatát.



19. A 4.3 ábra egy 10 elemű sorozat állapotait szemlélteti annak beszűrő rendezéssel történő rendezése során. Határozza meg, hogy a rendezés során hány összehasonlítást és értékadást kellett elvégezni, ha az ábrán megadott sorozatból indultunk ki.
20. Adott az  $n_a$  és az  $n_b$  elemű  $a$  és  $b$  növekvően rendezett sorozat. Növekvően rendezett sorozatokról lévén szó teljesül, hogy az egyes sorozatok utolsó elemei a legnagyobbak. Az elemek tárolására szolgáló adatszerkezetek lehetővé teszik azok egy-egy elemmel való bővítését az  $n_a + 1$ -edik és az  $n_b + 1$ -edik pozíción. Tároljunk ezeken a pozíciókon  $\max(a_{n_a}, b_{n_b})$ -nél nagyobb tetszőleges elemet.

## 5. Halmaz

A halmaz a matematika egyik alapfogalma, melyet leginkább úgy tudunk körülírni, mint bizonyos egymástól különböző dolgok „összességét”. Ez a fogalom számtalan probléma matematikai modellezését teszi egyszerűbbé, és a megoldást jelentő hatékony algoritmus előállításához föltétlen szükséges egy alkalmas modell. Ilyen esetekben segíthetik munkánkat egy jól felépített adatszerkezet és a hozzá köthető műveletek algoritmusai. Természetesen az alábbiakban bemutatni kívánt adatszerkezet csak véges,  $n$  elemű halmazok reprezentálására alkalmas, lévén a számítógép memóriája is véges. Ugyanakkor más szempontból törekedni fogunk a szokásos halmazelméletben ismert műveletek, fogalmak megvalósítására.

halmaz	
komplementere:	$\overline{A} := \{x   x \notin A\}$
két halmaz	
uniója:	$A \cup B := \{x   x \in A \vee x \in B\}$
két halmaz	
metszete:	$A \cap B := \{x   x \in A \wedge x \in B\}$
két halmaz	
különbsége:	$A \setminus B := \{x   x \in A \wedge x \notin B\}$
részhalmaz:	$A \subset B, \text{ ha } \forall x \in A \text{ esetén } x \in B$
halmaz	
elemszáma:	$ A $
üres halmaz:	$\emptyset, \quad  \emptyset  = 0 \text{ (üres halmaz elemszáma nulla)}$
diszjunkt	
halmazok:	$A \cap B = \emptyset \quad (\text{metszetük az üres halmaz}).$

Jelölje  $U$  a probléma szempontjából fontos elemek összességét. Azt a halmazt, amely tartalmaz minden elemet, amely a feladat kapcsán egyáltalán szóba jöhet, univerzumnak is szokás nevezni. Bár a halmaz jellegéből adó-

dóan annak elemei között semmiféle kapcsolat nincs, tekintsük most az  $U$  univerzum elemeinek egy rögzített sorrendjét. Ez természetesen egy  $n$  elemet tartalmazó sorozatot jelent, hiszen az  $U$ -nak is  $n$  eleme van.

$$U := \{ \quad e_1 \quad e_2 \quad e_3 \quad \dots \quad e_i \quad \dots \quad e_n \quad \}$$

Mivel  $U$  az univerzum, így egy tetszőleges  $A$  halmaz ( $A \subset U$ ) minden eleme is megtalálható  $U$ -ban. Ezért ahelyett, hogy az  $A$  elemeit újból eltárolnánk, reprezentáljuk a halmazt egy olyan  $(a_n)$  logikai sorozattal, amelynek  $i$ -edik eleme pontosan akkor igaz, ha  $e_i \in A$  (ahol  $e_i$  az univerzum elemeiből képzett sorozat  $i$ -edik eleme).

Ebben a konstrukcióban meglehetősen egyszerűen és gyorsan valósíthatók meg a halmazműveletek, ráadásul anélkül, hogy az egyes halmazok elemeivel ténylegesen dolgozni kellene. Elegendő csak az őket reprezentáló logikai sorozatokkal műveletet végeznünk. Ez annyit jelent, hogy például az unió művelete a két halmazt reprezentáló logikai sorozathoz egy olyan logikai sorozatot rendel, amely a két halmaz unióját reprezentálja. Lényegében tehát a halmazműveleteket logikai műveletekre vezetjük vissza.

A továbbiakban legyenek adottak azok a logikai sorozatok, amely az  $A$  és a  $B$  halmazt reprezentálják. Jelölje  $a_i$  és  $b_i$  egy-egy általános elemét az említett logikai sorozatoknak.

$\overline{A}$  : A fentiek alapján nyilvánvaló, hogy az  $A$  halmaz  $U$ -ra vonatkoztatott komplementerét reprezentáló logikai sorozat az  $A$ -t reprezentáló logikai sorozat elemeinek negálásával állítható elő  $(\overline{a_i})$ .

$A \cup B$  : A két halmaz unióját reprezentáló logikai sorozat elemeit úgy kaphatjuk meg, hogy az  $A$  és  $B$  halmazokat reprezentáló logikai sorozatok elemei között logikai vagy műveletet végzünk  $(a_i \vee b_i)$ .

$A \cap B$  : A két halmaz metszetét reprezentáló logikai sorozat elemeit úgy kaphatjuk meg, hogy az  $A$  és  $B$  halmazokat reprezentáló logikai sorozatok elemei között logikai és műveletet végzünk  $(a_i \wedge b_i)$ .

$A \setminus B$  : A két halmaz különbségét reprezentáló logikai sorozat elemeit úgy kaphatjuk meg, hogy az  $A$  halmazt reprezentáló logikai sorozat elemei

között és  $B$  halmazokat reprezentáló logikai sorozat elemeinek negáltja között logikai és műveletet végzünk  $(a_i \wedge \overline{b_i})$ .

$|A|$ . : Az  $A$ -t reprezentáló logikai sorozat *Igaz* értékű elemeinek a száma.

$A \subset B$ . :  $a_i \wedge \overline{b_i} = \textit{Hamis}$  ( $\forall i$  esetén). Tehát ha létezik olyan  $i$ , hogy  $a_i \wedge \overline{b_i} = \textit{Igaz}$ , az azt jelenti, hogy  $e_i \in A$  és  $e_i \notin B$  egyszerre teljesül (ahol  $e_i$  az univerzum elemeiből álló sorozat  $i$ . eleme).

$A \cap B = \emptyset$ . :  $a_i \wedge b_i = \textit{Hamis}$  ( $\forall i$  esetén). Tehát ha létezik olyan  $i$ , hogy  $a_i \wedge b_i = \textit{Igaz}$ , az azt jelenti, hogy  $e_i \in A$  és  $e_i \in B$  egyszerre teljesül (ahol  $e_i$  az univerzum elemeiből álló sorozat  $i$ . eleme).

## 5.1. Feladat

1. A fejezetben tárgyaltakra építve deklaráljunk halmaz adatszerkezetet és valósítsuk meg az adatszerkezet kezeléséhez szükséges eljárásokat, függvényeket.
2. Deklaráljunk halmaz adatszerkezetet és adjuk meg az adatszerkezet kezeléséhez szükséges szükséges eljárásokat, függvényeket, ha az egyes halmazok elemeit egy-egy rendezetlen sorozatként adjuk meg. (A szükséges eljárások és függvények megírásához fontoljuk meg a korábban megismert algoritmusok használatát.)
3. Deklaráljunk halmaz adatszerkezetet és adjuk meg az adatszerkezet kezeléséhez szükséges szükséges eljárásokat, függvényeket, ha az egyes halmazok elemeit egy-egy rendezett sorozatként adjuk meg. (A szükséges eljárások és függvények megírásához fontoljuk meg a korábban megismert algoritmusok használatát.)

## 6. A verem és a sor

A különböző informatikai rendszerek esetében gyakran az jelent megoldást, hogy az eltárolt homogén elemek sorozatához csak a sorozat végén legyen lehetőségünk hozzáférni különböző műveletek elvégzése céljából.

A verem olyan homogén, szekvenciális adatszerkezet, amely esetében az adatszerkezetben tárolt elemek sorozatához csak az egyik végénél férhetünk hozzá. Bővíthetjük a vermet egy újabb elemmel, ekkor azt mondjuk, hogy adatot helyezünk a verem tetejére, vagy kiolvashatjuk a verem tetején lévő elemet. Ez tehát azt jelenti, hogy leghamarabb az adatszerkezetben legutoljára elhelyezett elemhez férhetünk hozzá, illetve a benne legelőször elhelyezett adatot olvashatjuk ki legutoljára<sup>18</sup>.

A sor szintén homogén és szekvenciális, de esetében a bővítés az elemek sorozatának egyik, míg az olvasás a másik végén lehetséges. Az adatszerkezetbe elhelyezhetünk új elemet az egyik végén, de elem kiolvasására csak az adatszerkezet másik végén van lehetőség. Ez azt eredményezi, hogy az adatszerkezetbe legelőször elhelyezett elem olvasható ki leghamarabb, és a legutoljára beírt adathoz férhetünk hozzá legutoljára<sup>19</sup>.

### 6.1. A verem alkalmazásai

A veremből tehát a tárolás sorrendjével ellentétes sorrendben tudjuk a verem tetején elhelyezett adatokat kiolvasni. Ezt a tulajdonságát meglehetősen sok probléma megoldása során tudjuk hasznosítani.

#### 6.1.1. Posztfixes kifejezés kiértékelése

A matematikai kifejezések írásakor általában az infixes jelölést szoktuk alkalmazni. Ez annyit tesz, hogy a kétoperandusos műveletek esetében az operátort közrefogja a két operandus, továbbá nyitó- és végzárójeleket alkalmazunk, amikor módosítani szeretnénk a műveletek elvégzési sorrendjét.

<sup>18</sup> LIFO: Last In First Out

<sup>19</sup> FIFO: First In First Out

Például a

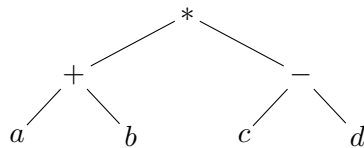
$$(a + b) * (c - d)$$

kifejezés kiértékelése során az  $a$  és a  $b$  összegét kell megszoroznunk  $c$  és  $d$  különbségével, míg ha zárójelek alkalmazása nélkül írjuk a kifejezést

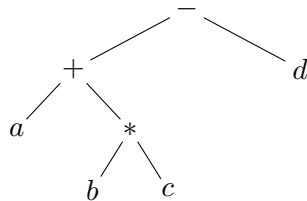
$$a + b * c - d$$

alakban, akkor  $a$ -hoz hozzá kell adnunk  $b$  és  $c$  szorzatát<sup>20</sup>, amiből kivonjuk  $d$  értékét.

Nagyon jól szemlélteti a két kifejezés közötti különbséget azok kifejezés-fája.



Az elsőnél például az összeadás két operandusa az  $a$  és a  $b$ ,



míg a másodiktól leolvasható, hogy az  $a$ -hoz a  $(b * c)$  szorzatot kell hozzáadni.

$$(a + b) * (c - d) \equiv a \ b + \ c \ d - *$$

$$a + b * c - d \equiv a \ b \ c * + \ d -$$

#### 6.1. táblázat. infixes — postfixes

<sup>20</sup> Természetesen elsőként a  $b * c$  szorzat értékét kell kiszámítanunk

A posztfixes jelölés esetén nincs szükség zárójelek alkalmazására (6.1. táblázat), és egy verem segítségével megadhatunk olyan egyszerű algoritmust, amellyel a kifejezés kiértékelése elvégezhető. Az algoritmus az alábbi szabályok alapján működik:

1. A kifejezést balról jobbra haladva olvassuk elemenként<sup>21</sup> a kifejezés végéig.
2. Amikor operandust olvasunk, akkor azt eltároljuk egy veremben.
3. Amikor pedig operátor következik, akkor kiolvasunk a verem tetejéről két operandust és levégezzük közöttük az operátor által kijelölt műveletet, amelynek az első operandusa a másodjára kiolvasott érték lesz, a második operandus pedig az, amit elsőként olvastunk a veremből.

Amennyiben a kiértékeléshez helyesen adtuk meg a kifejezést, az értéke a verem tetejéről olvasható ki, miután a kifejezés utolsó elemét is feldolgoztuk.

Az egyszerűség kedvéért tételezzük fel, hogy a verem a földolgozás kezdetén üres volt, tehát a kiértékelés végeztével a veremben csak a kifejezés értéke lehet. Könnyen belátható, hogyha a kifejezés a szükségesnél kevesebb operátort tartalmaz, akkor kiértékelés végeztével a veremben nem csak egy érték marad. Ez azért van így, mert az operátor hiánya miatt elmaradt a hozzá tartozó operandusok kiolvasása.

Ugyanakkor, ha az operátorok száma több a kifejezésben megadott operandusokhoz képest, a feldolgozás során be fog következni az az állapot, hogy operandust szeretnénk olvasni a veremből, de az üres.

Itt szeretnénk megjegyezni, ha a kiértékelést formálisan végezzük, azaz a verem tetejéről leemelt operandusok közé illesztjük az aktuális operátort és ezt a kifejezést – ami természetesen pontosan ezért infixes kifejezés – helyezzük el a verem tetején, akkor lényegében a posztfixes kifejezés infixessé való konvertálását végezzük.

<sup>21</sup> A kifejezés csak operandusokat és operátorokat tartalmaz.

### 6.1.2. Infixes kifejezés ellenőrzése

Fentebb láttuk, hogy egy infixes kifejezésben zárójelek alkalmazásával tudjuk befolyásolni a megadott műveleti jelek figyelembevételének sorrendjét. Ebből az következik, hogy a helyes zárójelezés kulcsfontosságú az infixes kifejezés szempontjából.

Egy zárójeleket tartalmazó kifejezés esetében megfigyelhetjük, hogyha mondjuk balról jobbra haladva olvassuk az elemeket, a nyitózárójelek sorrendje ellentétes a végzárójel-párjaikhoz képest. Azt is mondhatnánk, hogy azt a zárójelet kell leghamarabb „bezárni”, amelyiket legutoljára nyitottunk. Ez a kifejezés bizonyos részeinek egyféle egymásba ágyazottságot eredményezi.

Megjegyezzük továbbá, hogy hasonló szerkezet jellemzi például a programozási nyelvek forráskódját is, ami tovább növeli a következő algoritmus fontosságát. Nem beszélve arról, néhány programnyelv<sup>22</sup> esetében a különböző egymásba ágyazott egységek határát különböző zárójelek jelzik. Például a ?? ábrán látható C# -forráskódrészletben „{ }” zárójelpár foglalja magába a blokkokat, „[” és „]” fogja közre a tömbindexeket és kerek zárójeleket használunk a matematikai kifejezések írásakor.

Az algoritmus végrehajtása során a kifejezés karaktereit balról jobbra olvassuk (a forráskódot természetesen soronként – kezdve az első sorral – balról jobbra).

1. Ha nyitó zárójelet olvassunk, akkor azt eltároljuk a veremben,
2. Végzárójel olvasása esetén megvizsgáljuk, hogy a verem tetejéről kiolvasott nyitózárójelnek ez lehet-e a párja.

<sup>22</sup> Például C, C++, C#, Java, stb.



```

1
2     static int Szamolas(int[,] t, int r)
3     {
4         int db = 0;
5         for (int i = 0; i < 2; i++)
6         {
7             for (int j = 0; j < 10; j++)
8             {
9                 if (Math.Sqrt((t[i, j]) * (t[i, j])) > r)
10                {
11                    db++;
12                }
13            }
14        }
15        return db;
16    }
17    static bool Eldontes(int[,] t, int r)
18    {
19        for (int i = 0; i < 2; i++)
20        {
21            for (int j = 0; j < 10; j++)
22            {
23                if (Math.Sqrt((t[i, j]) * (t[i, j])) > r)
24                {
25                    return (Math.Sqrt(t[i, j]) * (t[i, j]) > r);
26                }
27            }
28        }
29        return false;
30    }

```

6.1. program. A C#-prgramkódban jól megfigyelhetők a különböző zárójelpárok egymáshoz képest elfoglalt helye.

A matematikában általában a kifejezések jobb olvashatósága érdekében többszintű zárójelezést alkalmazunk. Egy ilyen kifejezés ellenőrzése során tehát valóban azt kell ellenőrizni, hogy a kifejezés aktuális végzárójele megfelelő-e a verem tetejéről kiolvasott nyitózárójelnek. Mivel a ellenőrzés során nem foglalkozunk a kifejezés többi elemével, ezért maga az algoritmus meglehetősen széleskörűen alkalmazható. Például ellenőrizhetjük vele a 6.1. kódrészletben láthatóhoz hasonló forráskódok helyességét zárójelezés szempontjából.

## 6.2. A verem megvalósítása

Az adatszerkezetek megvalósításához szükség van egy homogén adatszerkezetre, amelyben majd az elemeket tároljuk. Erre acélra egy vektort fogunk használni, és az elemekhez való hozzáférés „korlátozását” – a fenti elveknek megfelelően – az adatszerkezet kezelését végző függvényekkel oldjuk majd meg.

A verem esetében az elemek tárolására szolgáló vektorhoz egy verem-mutatót (SP<sup>23</sup>) rendelünk, amelynek mindenkor értéke megmutatja, hogy a vektor mely elemét olvashatjuk ki leghamarabb. Ugyanakkor az SP értékét úgy is értelmezhetjük, hogy megmutatja annak az elemnek a helyet, amely után következő pozíció szabad a következő, az adatszerkezetben tárolni kívánt elem számára. (Az adatszerkezethez szükséges deklarációkat a 6.1. algoritmus mutatja.)

```
1 // ...
2 Konstans
3   MaxElem= elemek_maximális száma;
4 Típus
5   ElemTip: tárolandó_elemek_típusa;
6   VeremTip: rekord
7     Elem: tömb[1..MaxElem] ElemTip;
8     SP: egész;
9   RVége;
10 // ...
```

A verem adatszerkezet deklarációja

### 6.1. algoritmus

Természetesnek tűnő dolog azt mondani, hogy mielőtt bármilyen műveletet is végeztünk volna a veremmel, az állapota legyen üres. Mivel annak az elemnek a helyét a sokaságon belül, „amelyre” rá tehetjük majd a következő

<sup>23</sup> Stack Pointer

elemet az SP mező értéke határozza meg, ezért az SP kezdő értékét 0-nak kell választanunk. Erről gondoskodik a 6.2. algoritmus.

```
1 Eljárás VeremKezd(Verem: VeremTip);  
2 Algoritmus  
3   Verem.SP  $\leftarrow$  0;  
4 EVége;
```

A verem adatszerkezet inicializációja

### 6.2. algoritmus

Az előző példákból kitűnt az is, hogy fontos tisztában lennünk azzal, hogy üres-e az adatszerkezet, mert – ahogyan ezt fõntebb is láttuk – ez azt jelezte, hogy a kifejezés hibásan van megadva. Legalább ennyire fontos az is, hogy az adatszerkezetünk alkalmas-e további elemek befogadására. Ez indokolja azt, külön függvények szolgálnak a verem eme két fontos állapotának lekérdezésére.

A VeremÜres függvény (6.3. algoritmus) az adatszerkezet inicializálási állapotát jelzi és pontosan akkor tér vissza Igaz logikai értékkel, ha nem tudunk adatot kiolvasni belõle.

```
1 Függvény VeremÜres(Verem: VeremTip):logikai;  
2 Algoritmus  
3   VeremÜres  $\leftarrow$  (Verem.SP < 1);  
4 FVége;
```

A verem adatszerkezet üres állapotának lekérdezése

### 6.3. algoritmus

A VeremTeli szintén egy logikai típusú függvény, melynek Igaz visszatérési értéke egyszerűen „csak” azt jelzi, hogy az adatszerkezet számára lefoglalt tárterület megtelt.

Mindkettõnek csupán egy VeremTip típusú paramétert kell megadnunk a függvény hívásakor. Természetesen a visszatérési érték a paraméterként

megadott adatszerkezet állapotát jellemzi.

```
1 Függvény VeremTeli(Verem: VeremTip):logikai;  
2 Algoritmus  
3   VeremTeli  $\leftarrow$  (Verem.SP=MaxElem);  
4 FVége;
```

A verem adatszerkezet teli állapotának lekérdezése

#### 6.4. algoritmus

Ezekhez a funkciókhoz nem föltétlen kellene külön függvényeket rendelnünk, hiszen látható módon semmi mást nem tartalmaznak, mint csupán annak a logikai értéknek az előállítását, amely a visszatérési értéküként szolgál majd. Arról nem is beszélve, hogy így a hívással rontjuk az algoritmusunk hatékonyságát. Deklarációjukat és a későbbiekben való alkalmazásukat csupán a következő alprogramok jobb olvashatósága, és az a szándékunk indokolja, hangsúlyozni szeretnénk az adatszerkezet fenti két állapotának fontosságát.

Azért, hogy hatékonyság fontosságát mégis hangsúlyozzuk, a fenti két állapotlekérdező függvénynél is változóparaméterként adjuk át a vizsgálat tárgyát képező adatszerkezetet, hiszen így tárhely és végrehajtási idő tekintetében is jobb lesz a program. Természetesen más a helyzet a verem kezdőállapotra hozását végző VeremKezd 6.2. függvény esetében, hiszen itt a verem változó paraméterként való megadása biztosítja azt, hogy a függvényben, a verem-paraméteren végrehajtott változtatások (nevezetesen az SP értékének beállítása) a hívás helyén, a függvényhívásból való visszatérés után is „érzékelhetőek” legyenek.

A következő két logikai típusú függvény segítségével tudunk a paraméterként megadott verem tetejére adatot elhelyezni, illetve a verem tetején lévő adatot kiolvasni. A művelet sikerét a függvények Igaz illetve Hamis visszatérési értéke jelzi a hívás helyén.

A második formális paraméter (Adat) a VeremBe függvény esetében bemenetként szolgál, míg a VeremBől függvénynek kimenete lesz. Ez magya-

rázza, hogy míg az első esetben az Adat értékparaméter, addig a másodikban már változóparaméterként van deklarálva.

```
1 Függvény Verembe(Verem: VeremTip; Adat: ElemTip):  
    logikai;  
2 Algoritmus  
3 Ha nem VeremTeli(Verem) akkor  
4     Verem.SP ← Verem.SP + 1;  
5     Verem.Elem[Verem.SP] ← Adat;  
6     Verembe ← Igaz;  
7 Különben  
8     Verembe ← Hamis;  
9 HVége;  
10 FVége;
```

Írás verem adatszerkezetbe

#### 6.5. algoritmus

```
1 Függvény VeremBől(Verem: VeremTip; Adat: ElemTip):  
    logikai;  
2 Algoritmus  
3 Ha nem VeremÜres(Verem) akkor  
4     Adat ← Verem.Elem[Verem.SP];  
5     Verem.SP ← Verem.SP - 1;  
6     VeremBől ← Igaz;  
7 Különben  
8     VeremBől ← Hamis;  
9 HVége;  
10 FVége;
```

Olvasás a verem adatszerkezetből

#### 6.6. algoritmus

### 6.3. Sor

A sor adatszerkezet esetében – bár a benne eltárolt elemeket a veremhez képest másként adja vissza – a vereméhez hasonló funkciók megvalósítására lesz szükség.

Az adatszerkezet deklarációjából kitűnik, hogy az elemek tárolására szintén egy tömb szolgál, míg a bemeneti és kimeneti műveletek megvalósítását az adatszerkezet Első és Utolsó mezője segítségével tudjuk majd megvalósítani (6.8. algoritmus). Funkciójukat tekintve az Első mező annak az elemnek az adatszerkezeten belül elfoglalt helyét mutatja, ahonnan leghamarabb olvashatunk ki, míg az Utolsó mező értéke azt helyet őrzi, ahova legutoljára írtunk be adatot.

#### 6.3.1. Egyszerű sor

```
1 // ...
2 Konstans
3   MaxElem= elemek_maximális száma;
4 Típus
5   ElemTip: tárolandó_elemek_típusa;
6   SorTip: rekord
7     Elem: tömb[1..MaxElem] ElemTip;
8     Első, Utolsó: egész;
9   RVége;
10 // ...
```

A sor adatszerkezet deklarációja

#### 6.7. algoritmus

Használóan természetes, hogy üresnek kell tekintenünk ezt az adatszerkezetet is mielőtt bármilyen műveletet végeztünk volna vele. Ennek az állapotnak a beállításért a 6.8. algoritmus a felelős. Itt adjuk meg az Első és az Utolsó mezők értékeit úgy, amellyel az adatszerkezet üres állapotát jelezzük.

Egyben azt is biztosítjuk, hogy az első adat tárolása a vektor első pozícióján történjen majd meg, hiszen mivel az Utolsó értéke – funkciója szerint – a legutoljára eltárolt elem helyét őrzi, a következő szabad hely a benne tárolt értéknél eggyel nagyobb pozíción van. Az inicializáláskor megadott 0 kezdőérték biztosítja, hogy az első elem az vektor első elemébe kerüljön, illetve az Első mező inicializálási értéke – 6.8. algoritmus 3. sora – pedig valóban az elsőként kiolvasható elemre fog mutatni.

```
1 Eljárás SorKezd(Sor: SorTip);  
2 Algoritmus  
3   Sor.Első  $\leftarrow$  1;  
4   Sor.Utolsó  $\leftarrow$  0;  
5 EVége;
```

A sor adatszerkezet inicializációja

## 6.8. algoritmus

A vereméhez hasonló megfontolásokból deklaráljuk itt is az adatszerkezet állapotait lekérdező SorÜres (6.9. algoritmus) és SorTeli (6.10. algoritmus) függvényeket.

Mivel az Első a leghamarabb elérhető, míg az Utolsó a legutoljára eltárolt elem hejét jelzi, könnyen belátható, hogy az adatszerkezetbe történő íráskor és a belőle történő olvasáskor is mindkettő értékét növelnünk kell. Ugyanakkor az is természetes, hogy az adatszerkezet üres legyen abban az esetben, ha a benne korábban eltárolt adatot ki is olvastuk, azaz pontosan annyiszor olvastunk belőle, ahányszor írtunk bele. Így hát természetes, hogy az inicializáláskor az Első és a Utolsó mezők között teljesülő összefüggés ( $\text{Első} > \text{Utolsó}$ ) mindenkor azt jelzi, hogy az adatszerkezet üres.

```

1 Függvény SorÜres(Sor: SorTip):logikai;
2 Algoritmus
3   SorÜres  $\leftarrow$  (Sor.Első > Sor.Utolsó);
4 FVége;

```

A sor adatszerkezet üres állapotának lekérdezése

### 6.9. algoritmus

Mivel az Utolsó mező az utoljára az adatszerkezetben elhelyezett elem helyét mutatja, természetesen nem képes további elemet befogadni, ha értéke elérte az elemek tárolására szolgáló Elem vektor felső indexhatárát (6.10. algoritmus).

```

1 Függvény SorTeli(Sor: SorTip):logikai;
2 Algoritmus
3   SorTeli  $\leftarrow$  (Sor.Utolsó=MaxElem);
4 FVége;

```

A sor adatszerkezet teli állapotának lekérdezése

### 6.10. algoritmus

A fenti elvek figyelembevételével valósítja meg az elemek sorba történő írását a 6.11. és azok sorból történő olvasását a 6.12. algoritmus.



```

1 Függvény SorBa(Sor: SorTip; Adat: ElemTip):logikai;
2 Algoritmus
3   Ha nem SorTeli(Sor) akkor
4       Sor.Utolsó ← Sor.Utolsó + 1;
5       Sor.Elem[Sor.Utolsó] ← Adat;
6       SorBa ← Igaz;
7   Különbén
8       SorBa ← Hamis;
9   HVége;
10 FVége;

```

Írás sor adatszerkezet végére

#### 6.11. algoritmus

```

1 Függvény SorBól(Sor: SorTip; Adat: ElemTip):logikai;
2 Algoritmus
3   Ha nem SorÜres(Sor) akkor
4       Adat ← Sor.Elem[Sor.Első];
5       Sor.Első ← Sor.Első + 1;
6       SorBól ← Igaz;
7   Különbén
8       SorBól ← Hamis;
9   HVége;
10 FVége;

```

Olvasás sor adatszerkezet elejéről

#### 6.12. algoritmus

### 6.3.2. Léptető sor

Könnyen belátható, hogyha a fentebb bemutatott egyszerű sor esetében elvégezzük MaxElem számú adat tárolását az adatszerkezetben, akkor méltán

„látja” úgy a SorTeli függvény, hogy az adatszerkezet nem képes további elemek befogadására. Ezt követően ha ugyanennyi adatot ki is olvasunk, természetesen üres lesz az adatszerkezet. Vegyük figyelembe, hogy a kiolvasások során az Utolsó mutató értéke – amely az utoljára tárolt elem helyét mutatja – nem változott és értéke Maxelem. Könnyen belátható, hogy így bekövetkezett az Első és az Utolsó mezőknek egy olyan ellentmondásos állapota, amelyek alapján az állapotlekérdező függvényeink egyszerre „látják” az egyszerű sort üresnek és telinek.

Ennek az ellentmondásnak a feloldására adunk egy lehetséges megoldást a léptető sor műveleteinek értelmezésével.

```

1  Függvény LéptetSorBól(Sor: SorTip; Adat: ElemTip):
    logikai;
2  Változó
3  I: Egész;
4  Algoritmus
5  Ha nem SorÜres(Sor) akkor
6      Adat ← Sor.Elem[Sor.Első];
7      Ciklus I ← (Sor.Első..Sor.Utolsó-1)
8          Sor.Elem[I] ← Sor.Elem[I + 1];
9      CVége;
10     Sor.Utolsó ← Sor.Utolsó-1;
11     LéptetSorBól ← Igaz;
12  Különben
13     LéptetSorBól ← Hamis;
14  HVége;
15  FVége;

```

Adat olvasása a léptető sorból

### 6.13. algoritmus

Valójában csak arra kell ügyelnünk, hogy az elemek tárolására szolgáló vektor elemeit, amelyek fölszabadulnak egy-egy elem kiolvasása után, ismét

alkalmassá váljanak a sorban eltárolt elemek tárolására. Ennek egy lehetséges módja lehet az, ha egy elemnek a tárhely elejéről történő kiolvasása után az őt követő elemek mindegyikét egyel előrébb léptetjük. Így lényegében az elsőként kiolvashat elem mindig a tárhely első pozícióján lesz elérhető. Ezt valósítja meg a 6.13. algoritmus.

### 6.3.3. Ciklikus sor

Bár a léptető sor segítségével megoldottuk azt a problémát, hogy az adatszerkezet csak akkor nem képes újabb elemet befogadni, ha az elemek tárolására szolgáló tárterület valóban megtelt, ezt olyan áron értük el, hogy a minden kiolvasáskor annyi adatmozgatást végeztünk, ahány elemet tárolt az adatszerkezet. Ez természetesen jelentősen rontja az adatszerkezet időbeli hatékonyságát.

A ciklikus sort úgy is elképzelhetjük, hogy az egyszerű sor esetében a tárterület elején felszabaduló elemeket kezdjük feltölteni, ha már a tárterület végén elfogytak a szabad helyek.

Ha így járunk el, akkor az utolsó elem kiolvasásakor és az utolsó szabad hely feltöltésekor is az Első és az Utolsó mezők ugyanaz az egymáshoz viszonyított állapota valósulhat meg ( $\text{Első} > \text{Utolsó}$ ). Ez ellentmondásos helyzet, hiszen az egyik esetben az adatszerkezet üres, a másikban pedig megtelt. Ez az állapot tehát nem alkalmas arra, hogy a későbbiek során majd el tudjuk dönteni, hogy milyen a sor állapota.

Ennek feloldására újra értelmezzük a sor állapotlekérdezéseit végző függvényeket (6.14., 6.15. algoritmusok) és sorba történő írás (6.16. algoritmus) valamint a sorból történő olvasás (6.17. algoritmus) műveleteit.

```

1 Függvény CiklSorÜres(Sor: SorTip): Logikai;
2 Algoritmus
3   CiklSorÜres  $\leftarrow$  (Sor.Utolsó=0);
4 FVége;

```

A ciklikus sor üres állapotának lekérdezése

#### 6.14. algoritmus

```

1 Függvény CiklSorTeli(Sor: SorTip): Logikai;
2 Algoritmus
3   CiklSorTeli  $\leftarrow$  ((Sor.Első=1  $\wedge$  Sor.Utolsó=MaxElem)  $\vee$  (
      Sor.Utolsó>0  $\wedge$  Sor.Utolsó=Sor.Első-1));
4 FVége;

```

A ciklikus sor teli állapotának lekérdezése

#### 6.15. algoritmus

```

1 Függvény CiklSorBa(Sor:SorTip; Adat:ElemTip): Logikai;
2 Algoritmus
3   Ha NEM CiklSorTeli(Sor) Akkor
4       Ha Sor.Utolsó<MaxElem Akkor
5           Sor.Utolsó←Sor.Utolsó + 1
6       Különbén
7           Sor.Utolsó←1;
8       HVége;
9       Sor.Elem[Sor.Utolsó]←Adat;
10      CiklSorba←Igaz;
11  Különbén
12      CiklSorba←Hamis;
13  HVége;
14 FVége;

```

Írás ciklikus sorba

6.16. algoritmus

```

1  Függvény CiklSorBól(Sor: SorTip; Adat: ElemTip):
    Logikai;
2  Algoritmus
3  Ha NEM CiklSorÜres(Sor) Akkor
4      Adat ← Sor.Elem[Sor.Első]
5      Ha Sor.Első=Sor.Utolsó Akkor
6          SorKezd(Sor);
7      Különbén
8          Ha Sor.Első<MaxElem Akkor
9              Sor.Első ← Sor.Első + 1;
10         Különbén
11             Sor.Első ← 1;
12         HVége;
13     HVége;
14     CiklSorBól ← Igaz;
15     Különbén
16         CiklSorBól ← Hamis;
17     HVége;
18 FVége;

```

Olvasás ciklikus sorból

## 6.17. algoritmus

### 6.4. Feladatok

1. Oldjuk meg két verem (V1, V2) egyidejű kezelését abban az esetben, ha szeretnénk a vektorként rendelkezésre álló tárterület maximális kihasználtságát biztosítani, ugyanakkor tudjuk, hogy amikor a V1 verem sok elemet tárol, akkor a V2 keveset, és fordítva<sup>24</sup>.

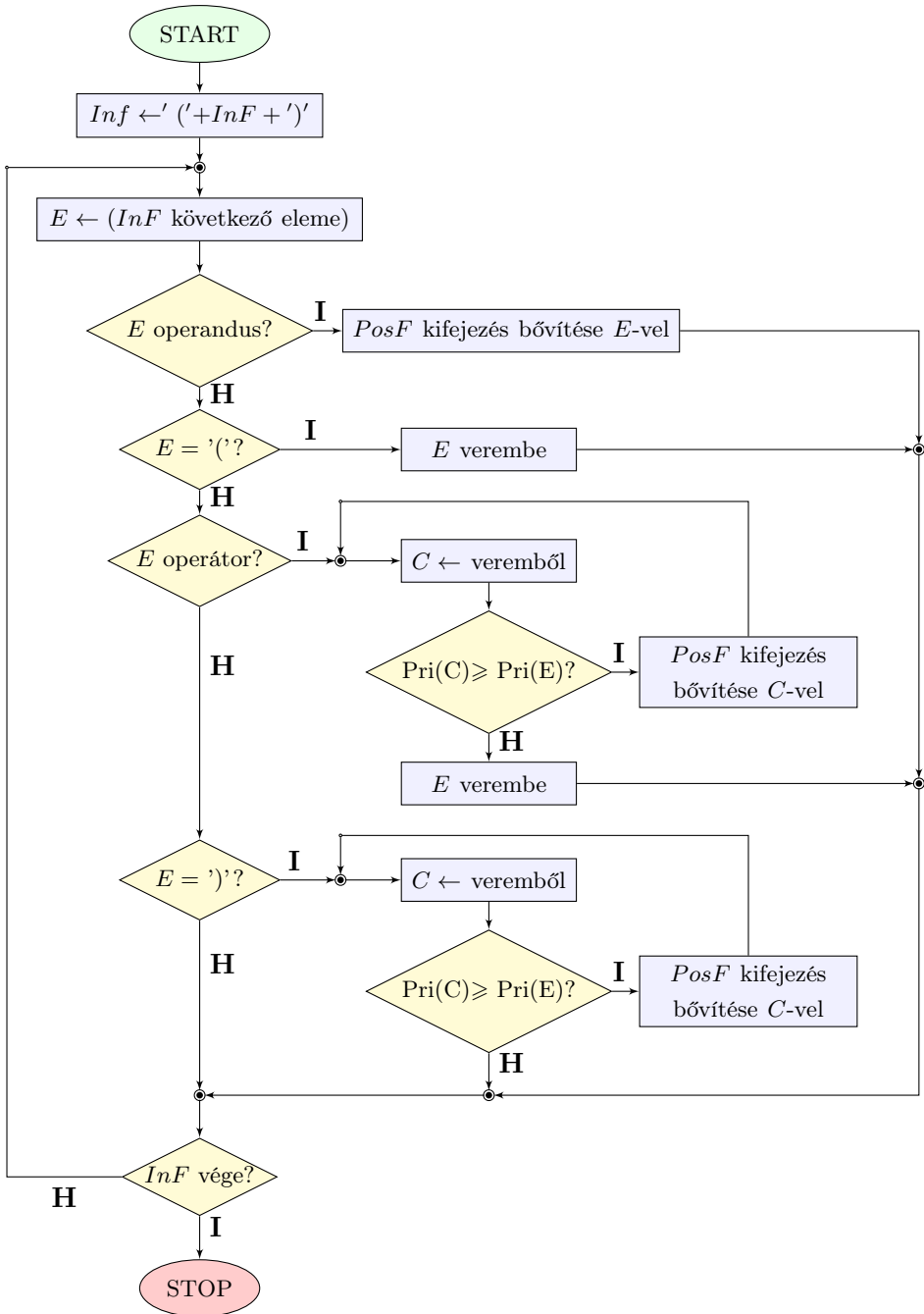
Feltételezve, hogy a vektort 1-től MaxElem-ig indexelhetjük, inicializálási állapotban a V1 verem SP1 veremmutatójának értéke 0, míg a

<sup>24</sup> Ez a megoldás azért lehet hasznos, mert két verem osztozik ugyanazon a tárterületen.

V2 verem SP2 mutatója  $\text{MaxElem}+1$  értéket vesz föl.

Írjuk meg mindkét verem kezeléséhez szükséges eljárásokat, függvényeket.

2. A 6.1. ábra az infixes kifejezés posztfixessé való konvertálását írja le folyamatábra segítségével. Adjuk meg a művelet algoritmusát leíró nyelven.
3. A léptető sor hatékonysága javítható volna úgy, hogy a sorban tárolt elemek mozgatását csak abban az „indokolt” esetben végeznénk el, ha tárterület végére már valóban nem tudunk újabb elemet elhelyezni. Értelmezzük ennek az adatszerkezetnek a megvalósításához szükséges alprogramokat.



6.1. ábra. Postfixes kifejezés átalakítása Infixessé



## 7. Rekurzio

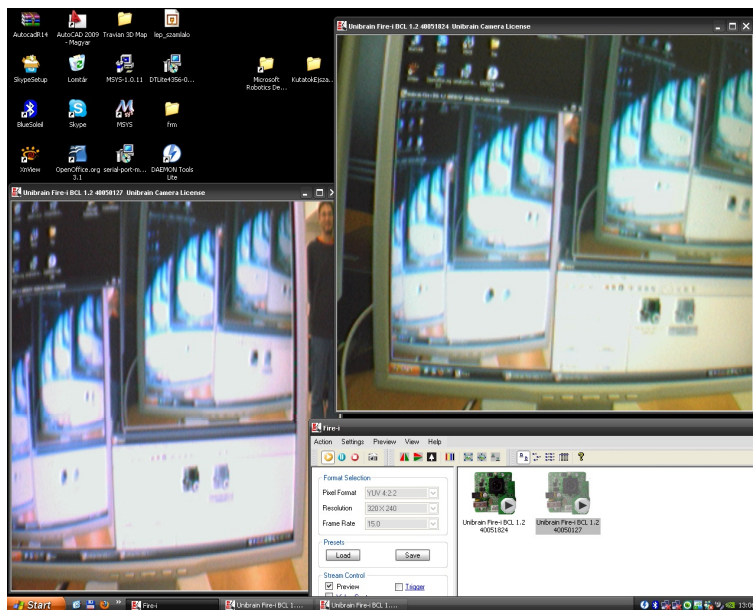
A rekurzio fogalmával a legkülönbözőbb területeken találkozhatunk. Teljesen megszokott a matematikában, hiszen nagyon sok fogalmat rekurzív módon definiálunk (7.1. egyenlet). Itt például előfordul sorozatok jelzője ként is (7.2. egyenlet). Ha tudjuk, hogy a matematika többek között a természet törvényszerűségeinek leírására is alkalmas, akkor talán nem meglepő az sem, hogy a természetben találhatók olyan dolgok, amelyek a legtalálóbban a rekurzióval jellemezhetők (7.1. ábra). Ugyanakkor a nyelvészet is használja ezt a kifejezést bizonyos szerkezetek leírására. Legyen szó a tudomány bármely területéről is, a közös az bennük az, hogy a szó maga valamiféle ismétlődés leírására szolgál. Az informatika – elsősorban a programozás – területén feltehetően annak matematikai gyökerei miatt honosodott meg. Találkozhatunk rekurzív programokkal és adatszerkezetekkel egyaránt. Nem törekszünk a fogalom definiálására, inkább szeretnénk szemléletessé tenni azt korábbi ismeretekre építve, megpróbáljuk kialakítani azt a gondolkodásmódok, amely szükséges rekurzív algoritmusok értelmezéséhez és megalkotásához.



7.1. ábra.  
Önhasonló alakzat a természetben

Mi magunk is könnyen előidézői lehetünk a rekurzív jelenségnek, ha egy vagy több kamerát arra a monitorra irányítunk, amely megjeleníti a kamerák által előállított képet. Így készült a 7.2. ábrán látható kép is. Ehhez

hasonló jelenséget jóval egyszerűbben, két egymással szembe fordított tükör segítségével is elő lehet idézni. Ez a látvány jól érzékelteti azt, amit az önhasonlóság fogalma alatt értünk, és bizonyos esetekben a fraktál fogalmával jellemzünk.

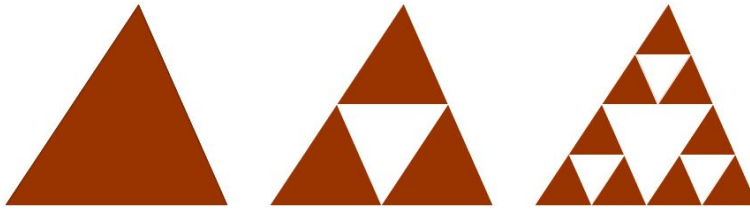


7.2. ábra. Két kamerás

Az önhasonlóság fogalmának megértése segít abban, hogy a rekurzív tevékenységet is megérthessük. Képzeljünk csak el egy egyszerű háromszöget. A háromszöget középvonalai négy egybevágó, az eredetivel hasonló háromszögre osztja. Távolítsuk el a középső háromszöget, és a maradék hárommal ismételjük meg a műveletet, majd az így nyert háromszögekkel hajtsuk végre ezt a műveletet újból és újból. Ennek a műveletsornak az első néhány lépését szemlélteti a 7.3. ábra.

## 7.1. Rekurzív definíció – rekurzív algoritmus

Bizonyos esetekben a probléma megfogalmazása eleve rekurzív, ahogyan ez sok helyen, főként matematikai definíciókban is fölfedezhető. Ilyenek például a téma tárgyalásakor „kötelezően” említendő  $k!$  (7.1) és a Fibonacci-számok



7.3. ábra. Sierpinski-háromszög.

(7.2) definíciója.

$$k! := \begin{cases} 1 & \text{ha } k = 0, \\ k(k-1)! & \text{ha } k > 0. \end{cases} \quad (7.1)$$

Ha összevetjük a  $k!$  7.1. definícióját és a  $k!$  értékének számítását végző 7.1. rekurzív algoritmust, könnyen belátható, hogy az lényegében nem más, mint a definíció leírónyelven való megfogalmazása.

```

1 Függvény Fakt(K: egész): egész;
2 Algoritmus
3   Ha K>0 akkor
4     Fakt ← K * Fakt(K-1);
5   Különbén
6     Fakt ← 1;
7   HVége;
8 EVége;
```

$k!$  értékének számítása rekurzív függvénnyel

### 7.1. algoritmus

Alkalmazzuk most a faktoriális definícióját  $k!$  értékének számítása során. A definíció szerint

$$k! = k(k-1)!$$

értékével ( $k > 0$  esetén). A definíció értelmében azonban  $(k-1)!$  helyébe  $(k-1)(k-2)!$ -t írhatunk, ha  $(k-1) > 0$  teljesül. Könnyen látható, hogy a

definíciót alkalmazva a

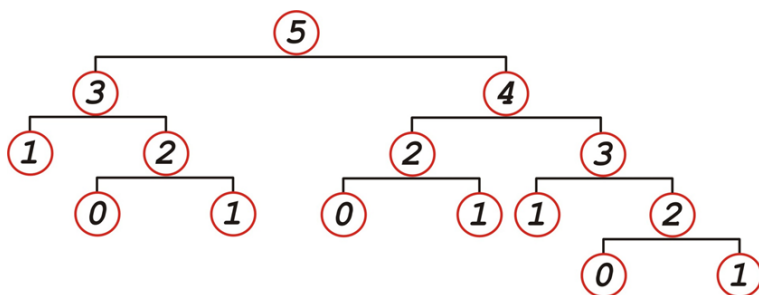
$$k(k-1)(k-2)(k-3)\dots 2\cdot 1$$

szorzatot kapjuk. Ezt a szorzatot azonban a korábban megismert összegzés tételének 3.1. algoritmusával is tudjuk produkálni.

A Fibonacci-számok alkotják az egyik legismertebb másodrendben rekurzív sorozat elemeit. Definíciójuk (7.2) értelmében a sorozat elemei a 2. elemtől kezdődően előállíthatók az előző két elem összegeként. Ahhoz azonban, hogy az  $F_2$ -t számítani tudjuk, meg kell adnunk,  $F_0$  és  $F_1$  értékét.

$$F_n := \begin{cases} 0 & \text{ha } n = 0, \\ 1 & \text{ha } n = 1, \\ F_{n-1} + F_{n-2} & \text{ha } n > 1. \end{cases} \quad (7.2)$$

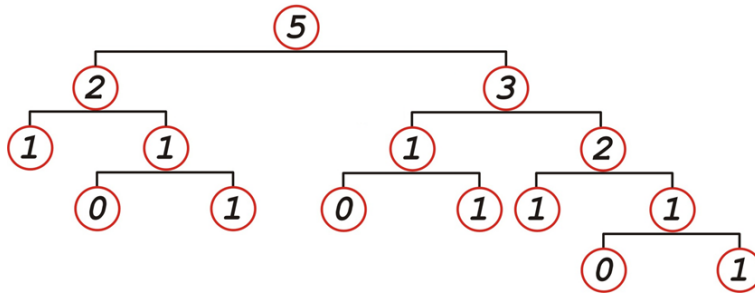
Ezt szemlélteti a 7.4. ábra is. Ennek értelmében az  $F_5$  csak  $F_3$  és  $F_4$  ismeretében számítható. Ugyanakkor  $F_3$  kiszámításához ismernünk kell  $F_1$ -et és  $F_2$ -t, illetve  $F_4$  is csak  $F_2$  és  $F_3$  értékének ismeretében számítható. (Fontos megjegyeznünk, hogy a 7.4. ábrán a sorozat elemeinek sorszámát és nem azok értékeit tüntettük föl.)



7.4. ábra. Fibonacci-számok előállítása. A sorozat egyes elemei közötti kapcsolat. (Az ábrán a Fibonacci-sorozat elemeinek sorszámát tüntettük föl.)

A jobb érthetőség kedvéért tekintsük a 7.5. ábrát, amelyről leolvasható, hogy a csomópontokban feltüntetett számok az alattuk lévő két szomszédjuk összegeként számíthatók. Természetesen ez alól azok az elemek képeznek

kivételt, amelyeknek nincs alsó szomszédjuk, de ezekről a 7.2. definíció az értékük megadásával rendelkezik.



7.5. ábra. Fibonacci-számok előállítás. (Az ábrán a Fibonacci-sorozat elemeinek értékeit tüntettük föl.)

A fentieknek pontosan megfelel az  $n$ . Fibonacci-szám előállítását végző rekurzív 7.2. algoritmus is, amelyben szintén fölfedezhetők a 7.2 definíció elemei, azaz ha az előállítandó elemre teljesül  $n > 1$  (azaz nem  $F_1$  vagy  $F_0$  értékét kell számítanunk), akkor ahhoz az előző két elem ismerete szükséges (7.2. algoritmus 4. sor).

```

1 Függvény Fibo(N: egész): egész;
2 Algoritmus
3   Ha N>1 akkor
4     Fibo ← Fibo(N-1) + Fibo(N-2);
5   Különbén
6     Fibo ← N;
7   HVége;
8 EVége;

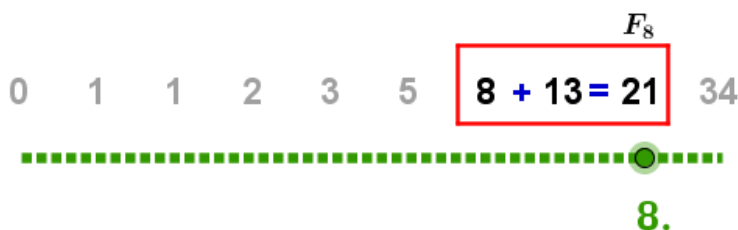
```

Az  $n$ . Fibonacci-féle szám előállítására rekurzív  
algoritmussal

## 7.2. algoritmus

A 7.2. definíciót úgy is értelmezhetjük, hogy az első két elem –  $F_0 = 0$  és  $F_1 = 1$  – megadása után a további elemeket az előző kettő összegeként

állíthatjuk elő (7.6. ábra). Azaz elegendő volna a 7.5. ábra elemeit az ábra jobb alsó sarkából kiindulva, balra fölfelé haladva bejárni. Ez a megközelítés az  $F_n$  előállításának iteratív algoritmusához vezet, amelynek kétségtelenül van olyan előnye, hogy az nem fogja előállítani a 7.5. ábrán feltüntetett összes számot, hanem a sorozat minden elemét csupán csak egyszer. (A 7.4. ábrán jól látható, hogy abban  $F_3$  értéke kétszer, míg  $F_2$  háromszor is szerepel.)



7.6. ábra. Fibonacci-számok előállítása. A sorozat valamely elemének értéke  $F_2$ -től kezdődően az öt megelőző két elem összegeként számítható (7.2).

Az előzőekben arra láthattunk két példát, hogy a fogalmak rekurzív definíciója alapján hogyan írhatunk rekurzív algoritmust. Ugyanakkor utaltunk az iteratív megvalósítás lehetőségére is. A fenti két esetben nem nehéz belátni, hogy az iteratív változatok végrehajtási ideje kevesebb lesz a rekurzív megfelelőjükhöz képest.

Most nézzük meg, hogy egy jól ismert iteratív algoritmus, a bináris keresés (4.6. algoritmus) rekurzív megfelelője hogyan készíthető el.

```

1  Függvény  RekBinKer (A:sorozat ,Adat:elemtip ,Hely,E,U:
    egész):logikai;
2  Változó
3      K:  egész;
4  Algoritmus
5      Ha E>U  akkor
6          RekBinKer ← Hamis;
7      Különbén
8          K ← (E + U) div 2;
9          Ha (A[K]>Adat)  akkor
10             RekBinKer ← RekBinKer (A,Adat ,Hely,E,K-1);
11             Különbén
12                 Ha (A[K]<Adat)  akkor
13                     RekBinKer ← RekBinKer (A,Adat ,Hely,K + 1,U);
14                     Különbén
15                         Hely ← K;
16                         RekBinKer ← Igaz;
17                     HVége;
18             HVége;
19     HVége;
20 FVége;

```

Bináris keresés rekurzív algoritmussal

### 7.3. algoritmus

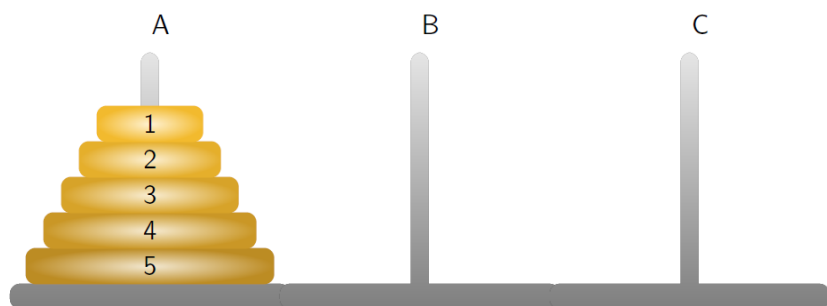
Ahogy az előző két példában is láttuk, az ismételt, önmagára irányuló rekurzív hívásokat egy elágazással tudjuk vezérelni. Az ebben alkalmazott logikai kifejezést báziskritériumnak nevezzük. Megfigyelhetjük, hogy mindkét algoritmusban rekurzív hívások paraméterei a bemenő paraméterekhez képest egyszerűbbek, míg végül olyan értékeket adunk meg paraméterekként, amelyekről a definíció már explicit módon rendelkezik ( $0! = 1$ ,  $F_0 = 0$ ,  $F_1 = 1$ ).

Ismerve a bináris keresés algoritmusának elvét tudjuk, hogy nem folytatjuk tovább a keresést abban az esetben, ha már nulla eleművé „zsugorodott” a sorozatnak azon része, amely tartalmazhatja a keresett elemet, vagy már megtaláltuk azt.

## 7.2. Hanoi tornyai

A rekurzió tárgyalásánál szintén klasszikusnak számító, jól értelmezhető a Hanoi tornyai-probléma. Megfelelő szemléltetés mellett sokat segíthet a rekurzió elvének megértésében, és a hozzá kapcsolódó szemléletmód elsajátításában.

### Hanoi tornyai – 5 korongos rendszer



7.7. ábra. Hanoi tornyai 5 koronggal (kezdő állapot)

A matematikai feladvány<sup>25</sup> – amelyet számtalan módon dolgoztak már föl, többek között számítógépes játékként is – szerint a játékban rendelkezésünkre áll három függőleges rúd (ezeket jelölje  $A$ ,  $B$  és  $C$ ) és  $n$  darab különböző átmérőjű korong<sup>26</sup>, amelyek az  $A$  jelű rúdra vannak fölfűzve, lent-

<sup>25</sup> Ezt a matematikai játékot Edouard Lucas francia matematikus nevéhez kötjük, aki 1883-ban találta föl. Alapjául az a legenda szolgált, amely szerint a világ teremtésétől kezdve egy 64 korongból álló feladvánnyal kezdtek „játszani” Brahma szerzetesei. (A szabályok megegyeznek az leírtakkal.) A legenda szerint, amikor a szerzetesek végeznek a feladvány megoldásával és a szabályoknak megfelelően átrakják a 64 korongot, akkor lesz vége a világnak.

<sup>26</sup> Az egyszerűbb hivatkozás érdekében a korongokat fentről lefelé megsorszámozzuk.



ről fölfelé haladva átmérő szerint csökkenő sorrendben. A feladat szerint a korongokat át kell juttatni az  $A$  rúdról a  $C$ -re úgy, hogy ott ugyanígy helyezkedjenek el és közben be kell tartanunk a következő szabályokat:

1. egyszerre csak egy korongot mozgathatunk,
2. egy korongra csak nála kisebbet tehetünk,
3. korongok átmeneti tárolására fölhasználhatjuk a  $B$  rudat.

A probléma megoldásához szükséges rekurzív gondolkodásmódot általános estre a 7.1 táblázat, 4 korong esetére pedig a 7.2 táblázat szemlélteti. Az alapgondolat szerint, ha a legalsó korong kivételével át tudnánk helyezni a nála kisebbeket a  $B$  rúdra, akkor már a szabályoknak megfelelően a legnagyobbat át is tehetnénk a  $C$ -re. Ezt követően már „csak” a  $B$ -n lévő korongokat kellene a  $C$ -re átpakolni. A táblázatok 0-val jelzett oszlopa az alap feladatot fogalmazza meg, míg az 1 jelű ennek felbontását a fentebb leírt három – összetett, elemi és összetett – lépésre. Lényegében az összetett lépések felbontásával egy  $n$ -korongos problémát egy  $(n - 1)$ -korongos, egy 1-korongos és egy újabb  $(n - 1)$ -korongos problémára bonthatjuk föl. Az újabb összetett problémák további fölbonthatásával végül már csak 1-korongos mozgatások sorozatává egyszerűsödik a feladat (7.2 táblázat utolsó oszlopa), lévén a korongok száma véges.

A táblázatokban az alábbi jelöléseket alkalmaztuk:

$A \xrightarrow{(1, \dots, n-1)} B$  : Az  $A$  rúdról  $B$  rúdra több korong áthelyezése.

Végrehajtásakor minden korongot át kell helyezni az 1-es sorszámútól kezdődően az  $(n - 1)$ -sel bezárólag, tehát összesen  $n - 1$  darabot.

$A \xrightarrow{(n-2)} C$  : Az  $A$  rúdról  $C$  rúdra egyetlen korong – melynek sorszáma  $(n - 2)$  – áthelyezése, a szabályoknak megfelelő módon.

A legkisebb korong sorszáma tehát 1 lesz, a legnagyobbé pedig megegyezik a korongok számával.

$k$	0	1	2	3	...
1				$A \xrightarrow{(1,\dots,n-3)} B$	...
2			$A \xrightarrow{(1,\dots,n-2)} C$	$A \xrightarrow{(n-2)} C$	...
3				$B \xrightarrow{(1,\dots,n-3)} C$	...
4		$A \xrightarrow{(1,\dots,n-1)} B$	$A \xrightarrow{(n-1)} B$	$A \xrightarrow{(n-1)} B$	...
5				$C \xrightarrow{(1,\dots,n-3)} A$	...
6			$C \xrightarrow{(1,\dots,n-2)} B$	$C \xrightarrow{(n-2)} B$	...
7				$A \xrightarrow{(1,\dots,n-3)} B$	...
8	$A \xrightarrow{(1,\dots,n)} C$	$A \xrightarrow{(n)} C$	$A \xrightarrow{(n)} C$	$A \xrightarrow{(n)} C$	...
9				$B \xrightarrow{(1,\dots,n-3)} C$	...
10			$B \xrightarrow{(1,\dots,n-2)} A$	$B \xrightarrow{(n-2)} A$	...
11				$C \xrightarrow{(1,\dots,n-3)} A$	...
12		$B \xrightarrow{(1,\dots,n-1)} C$	$B \xrightarrow{(n-1)} C$	$B \xrightarrow{(n-1)} C$	...
13				$A \xrightarrow{(1,\dots,n-3)} B$	...
14			$A \xrightarrow{(1,\dots,n-2)} C$	$A \xrightarrow{(n-2)} C$	...
15				$B \xrightarrow{(1,\dots,n-3)} C$	...

7.1. táblázat

A korongok átmozgatása általános esetben.

Vegyük észre, hogy a fentebb vázolt három lépést értelmezve a 4-korongos problémára, az alábbi lépéseket jelenti

$$A \xrightarrow{(1,2,3)} B, A \xrightarrow{(4)} C \text{ és } B \xrightarrow{(1,2,3)} C,$$

amelyekből a középső „elemi” mozgatás, az első és a harmadik pedig lényegében megfelel a 3-korongos problémának (7.2 táblázat 1. oszlop). A 7.2 táblázat 2. oszlopában az is megfigyelhető, hogy az előző oszlopban jelzett „összetett” mozgatásokat visszavezethetjük 2-korongos problémák és „elemi” mozgatások leírására, míg végül az 3. oszlopban már csak „elemi” mozgatá-

sokat láthatunk. Ezeket a műveleteket fentről lefelé végrehajtva a 4-korongos feladványt oldhatjuk meg.

Természetesen tetszőleges véges  $n$ -korongos feladvány esetében is elkészíthető olyan táblázat, amelynek utolsó oszlopában már csak a feladat megoldását jelentő „elemi” mozgatások sorozata található.

$k$	0	1	2	3
1				$A \xrightarrow{(1)} B$
2			$A \xrightarrow{(1,2)} C$	$A \xrightarrow{(2)} C$
3				$B \xrightarrow{(1)} C$
4		$A \xrightarrow{(1,2,3)} B$	$A \xrightarrow{(3)} B$	$A \xrightarrow{(3)} B$
5				$C \xrightarrow{(1)} A$
6			$C \xrightarrow{(1,2)} B$	$C \xrightarrow{(2)} B$
7				$A \xrightarrow{(1)} B$
8	$A \xrightarrow{(1,2,3,4)} C$	$A \xrightarrow{(4)} C$	$A \xrightarrow{(4)} C$	$A \xrightarrow{(4)} C$
9				$B \xrightarrow{(1)} C$
10			$B \xrightarrow{(1,2)} A$	$B \xrightarrow{(2)} A$
11				$C \xrightarrow{(1)} A$
12		$B \xrightarrow{(1,2,3)} C$	$B \xrightarrow{(3)} C$	$B \xrightarrow{(3)} C$
13				$A \xrightarrow{(1)} B$
14			$A \xrightarrow{(1,2)} C$	$A \xrightarrow{(2)} C$
15				$B \xrightarrow{(1)} C$

7.2. táblázat

4 db korong átmozgatása. (A 4 korongos feladat a táblázat utolsó oszlopában található elemi lépések végrehajtásával megoldható.)

Az algoritmusunknak is a fentebbi megállapításokat kell tükröznie. A 7.4. algoritmusban paraméterként elsőként adjuk át a korongok  $n$  számát, majd a következő paraméter jelentse azt a rudat ahonnan az  $n$  szám korongot át kell mozgatni. A következő paraméter a cél-rúdnak azonosítója, míg az utolsó

paraméterben megadott rudat segédként használhatjuk probléma megoldása során. Tehát a formális paraméterek jelölésével a feladatban  $n$  korongot kell át mozgatni az R1-rúdról az R3-ra, melynek során fölhasználhatjuk R2-t.

Megfigyelhetjük, hogy a 7.4. algoritmus 4. sora egy  $(n-1)$ -korongos probléma megoldását jeleni, ahol az eredeti „forrás” R1 rúdról  $(n-1)$  korongot az R2 „segéd”-rúdra kell átmozgatni. Az 5. sor utasítása kijelzi, hogy az  $n$ . korongot – megengedett módon – áthelyezzük az R3 „cél”-rúdra, ezt követően pedig a 6. sor utasítása az R2 „segéd”-rúdról az ott korábban elhelyezett  $(n-1)$  korongot teszi a formális paraméterek között „cél”-ként megjelölt R3-ra.

```

1 Eljárás Hanoi(N:egész, R1,R3,R2: karakter);
2 Algoritmus
3   Ha DB>0 akkor
4     Hanoi(N-1,R1,R2,R3);
5     Ki: N, '._korong_átrakása:', R1, '->', R3;
6     Hanoi(N-1,R2,R3,R1);
7   HVége;
8 EVége;
```

A Hanoi tornyai probléma megoldása rekurzív  
algoritmussal

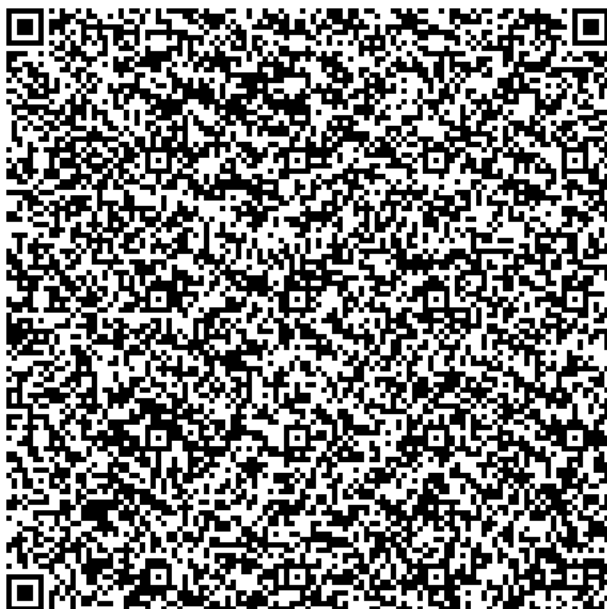
#### 7.4. algoritmus

### 7.3. Feladat

1. A definíció értelmezése alapján írjuk meg a  $k!$  értékét iterációval számító függvény algoritmusát.
2. Írjunk iteratív algoritmust az  $n$ . Fibonacci-féle szám előállítására.
3. Írjon rekurzív algoritmust, amelyben  $a_0 = 0$ ,  $a_1 = 1$ , valamint a 2. elemtől kezdődően minden elem előáll az öt megelőző 2 elem négyzetinek különbségként ( $a_i = a_{i-1}^2 - a_{i-2}^2$ ).

4. Írjon rekurzív algoritmust, amely előállítja egy sorozat elemeit, ahol  $a_0 = 0$ ,  $a_1 = 1$ ,  $a_2 = 2$ , valamint a 3. elemtől kezdődően minden elem előáll az öt megelőző 3 elemből  $a_i = a_{i-1} - a_{i-2} + a_{i-3}$  alakban.
5. Írjon rekurzív algoritmust, amely előállítja egy sorozat elemeit, amelyben  $a_0 = 0$ ,  $a_1 = 1$ ,  $a_2 = 2$ , valamint a 3. elemtől kezdődően minden elem az öt megelőző 3 elem alapján  $a_i = (a_{i-1} - a_{i-3})^2$  alakban adható meg.
6. Az *ABCDEFGHJKLMNOP* sorozatot megfelezzük. A baloldali rész elemeit balról, a jobboldali elemeit jobbról véve az elemeket rendre megcseréljük. Ezt követően az egyes részsorozatokkal ugyanezt végezzük, amíg 2 elemű részeket nem kapunk. Írja föl az egyes lépések után nyert sorozatokat. Írjon leíró nyelvű algoritmust.
7. A 7.8. ábra egy QR-kódot<sup>27</sup> mutat be, melynek tárolása egy olyan  $n \cdot n$  típusú mátrixban is történhet, amelyben az  $i$ -edik sor  $j$ -edik eleme tárolja a kép  $i$ -edik sorának  $j$ -edik képpontját 0 vagy 1 formájában, attól függően, hogy az adott négyzet fehér vagy fekete.  
Írjunk olyan rekurzív algoritmust, amely sorozattá alakít egy ilyen mátrixot az alábbiak szerint. Az oszlopok és a sorok számának felezésével osszuk fel a mátrixot négy almátrixra, majd balról jobbra és fentről lefelé haladva ezekkel az almátrixokkal tegyük ugyanezt mindaddig, míg egy elemet nem kapunk. Az így elért elem legyen az előállítandó sorozat következő eleme. (Az egyszerűség kedvéért tételezzük fel, hogy  $n = 2^k$  alakú, ahol  $k \in \mathbb{N}$ .)
8. Adott egy olyan sorozat, amelyet a fentebb leírt módon állítottunk elő. A sorozat felhasználásával adjuk meg a QR-kódnak megfelelő négyzetes mátrixot.
9. A 7.1. és a 7.1. táblázat alapján határozzuk meg általánosan, hogy az  $n$ -korongos feladat hány elemi mozgatóval oldható meg.

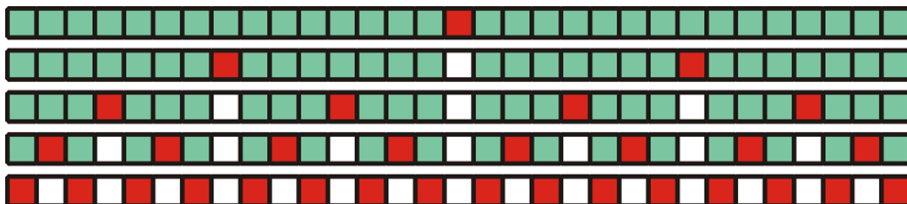
<sup>27</sup> Az egyre népszerűbb kétdimenziós kódról tartalmaz néhány alapvető információt a 7.8. ábra.



7.8. ábra. QR-kód

10. Írjunk rekurzív rendező algoritmust a következő elgondolásra építve. Tekintsük az  $n$ -elemű sorozat elemeit, mint önálló egyelemű sorozatokat, amelyek így rendezettek is egyben. Ezek alapján két-két szomszédos elemet egyesíthetünk egy kételemű rendezett sorozattá, majd a szomszédos kételemű sorozatokat négyelemű rendezett sorozatokká egyesíthetjük az összefuttatás algoritmus (4.16. algoritmus) felhasználásával, míg végül két olyan rendezett sorozat összefuttatását kell elvégeznünk, amelyek elemszáma azonos, vagy legfeljebb eggyel tér el az elemszámuk.
11. Írjunk rekurzív algoritmus, amely előállítja egy  $n$  elemű sorozat összes permutációit, a következő elgondolás alapján. Válasszunk ki a sorozat első helyére elemeket az összes lehetséges módon. Ezek rögzítése után képezzük a maradék  $(n - 1)$  elem összes permutációit.
12. Írjuk meg egy  $n$ -elemű sorozat elemeit binárisan bejáró algoritmust. Elsőként válasszuk ki a sorozat középső elemét – amely a sorozatot

két részre osztja – és jelezzük ki azt. Ezt követően a középső elemet megelőző és az azt követő elemek részsorozatával is tegyük ugyanezt.



7.9. ábra. Bináris bejárás.

Ezt szemlélteti a 7.9. ábra, ahol piros szín jelzi a rekurzió adott szintjén feldolgozásra kerülő elemeket, zöld a még feldolgozásra várókat és fehér a korábban már feldolgozottakat. Figyeljük meg, hogy a rekurzió utolsó szintjét jelképező utolsó sorban csak piros és fehér színnel jelzett elemek találhatók. Magyarázzuk meg ennek a jelentését.

13. Értelmezzük a 7.1. programot.

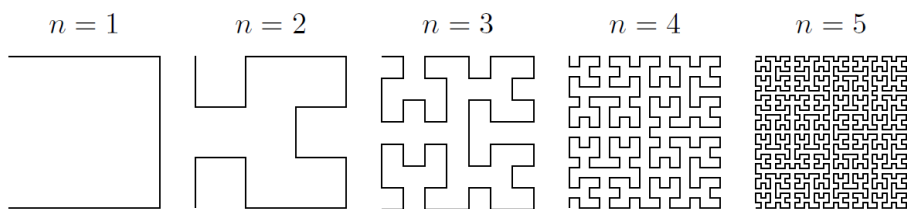
```

1      public void S(Point A, Point B, Point C)
2      {
3          if (D(A, B) >= H)
4          {
5              Point Fab = new Point();
6              Point Fac = new Point();
7              Point Fbc = new Point();
8              Fab.X = (A.X + B.X) / 2;
9              Fab.Y = (A.Y + B.Y) / 2;
10             Fac.X = (A.X + C.X) / 2;
11             Fac.Y = (A.Y + C.Y) / 2;
12             Fbc.X = (B.X + C.X) / 2;
13             Fbc.Y = (B.Y + C.Y) / 2;
14             S(A, Fab, Fac);
15             S(Fab, B, Fbc);
16             S(Fac, Fbc, C);
17         }
18         else
19         {
20             Vonal(A, B);
21             Vonal(A, C);
22             Vonal(B, C);
23         }
24     }
25     public double D(Point A, Point B)
26     {
27         return Math.Sqrt(Math.Pow(A.X-B.X,2)+Math.Pow(A.Y-B.Y,2));
28     }
29     public void Vonal(Point A, Point B)
30     {
31         Update();
32         Graphics g = CreateGraphics();
33         Pen pen = new Pen(Color.Black);
34         g.DrawLine(pen, A, B);
35     }

```

### 7.1. program. Rekurzív algoritmus alapján működő C#-prgramkód





7.10. ábra. Hilbert-görbe

## 8. Dinamikus adatszerkezetek

Tudjuk azt, hogy a memóriában tárolt adatok „kinyeréséhez” szükségünk van bizonyos információkra. Nem elegendő ismernünk a tárterület kezdetét, amely tartalmazza a kérdéses adatot 0 és 1 értékek sorozatát jelentő fizikai jelek formájában, ismernünk kell ennek a kívánt részsorozatnak a hosszát is. Sajnos még ez sem elegendő, mert tudnunk kell azt is, hogy hogyan értelmezzük azt<sup>28</sup>. Mindezen információkat például a változó deklarácójakor a változó azonosítójához rendeljük. Pontosabban azért, hogy a programozó válláról levegyük ezt terhet, mindezt maga a rendszer végzi el, nekünk a későbbiekben elegendő csak az azonosítóval hivatkoznunk az adatra, a rendszer „tudni” fogja, hogy hol keresse és azt is, hogyan kell értelmeznie a szintén az azonosító által kódolt hosszúságú bitsorozatot.

Leegyszerűsítve ez úgy történik, hogy a rendszerünk a deklarációban megadottak alapján létrehoz egy táblázatot, amely tartalmazza ezeket az információkat (azonosító, kezdőcím, típus) és valóban hozzá is rendeli a szükséges (a típus által igényelt) tárrészeket az így létrehozott táblázat egyes soraihoz, ügyelve arra, hogy ezek a tárterületek részben ne fedhessék át egymást, azaz részben ne tartozhasson egyszerre két vagy több különböző azonosítóhoz is<sup>29</sup>.

Bár ez nagyon kényelmes „szolgáltatás”, vizsgáljuk meg mégis, jelenthet-e előnyt, ha mi magunk döntünk arról, hogy a memória mely területe tartozzon az adott szimbólumhoz – vagy éppen tartozzon-e hozzá egyáltalán. Lényegében arról volna szó, hogy az egyes adatokhoz tartozó címinformáció megváltoztatását magában a programban el lehessen végezni. Ennek lehetőségét biztosítják egyes programozási nyelvek a pointer (mutató) típus segítségével. Az ilyen típusú adat jelentése tehát egy tárcím, de társítunk mellé egy adattípust is. Így a segítségével meg tudjuk mondani, hogy hol található

<sup>28</sup> A változó típusa meghatározza az adat tárolásához szükséges terület nagyságát és azt, hogy hogyan kell értelmezni az ott található értéket.

<sup>29</sup> Ez alól a cím szerinti paraméterátadás kivétel, hiszen ott pontosan az a cél, hogy az aktuális paraméterhez tartozó memóriaterülethez rendeljük a formális paramétert is, ezzel biztosítva azt, hogy amikor megváltoztatjuk az aktuális paraméterként deklarált változó értékét, akkor lényegében az aktuális paraméter értéke változzon meg.

az adat, de azt is, hogy a memória adott területén található adattal milyen műveletek végezhetők. Közvetlen módon az adat címét, közvetve azonban magát az adatot, az adott címen tárolt értéket is el tudjuk érni.

Hasonló a helyzet a tömbök használata során is. Gondoljunk csak a sorozat elemei közül a legkisebbet kiválasztó 3.4. algoritmusra, amely a megfelelő elem értékét adja vissza, és a minimális értékű elem sorszámát meghatározó 3.5. algoritmusra, amely a legkisebb elem első előfordulási helyét határozza meg. Természetesen a sorszám ismeretében – közvetve – a legkisebb elem értékét is meg tudjuk határozni, hiszen tudjuk, hogy hol található a sokaságon belül. Így van ez mutatók használata esetében is. Különbséget kell tennünk a memóriában tárolt adat címe (a mutató értéke) és a mutató által beazonosítható tárterület tartalma között.<sup>30</sup>

Milyen előnyökkel járhat, ha a programozó maga határozhatja meg – természetesen ésszerű keretek között – egy adat tárolási helyét a memóriában? Kétségtől hatékonyabban tudjuk két változó „tartalmát” megcserélni, ha ténylegesen nem cseréljük föl az adatok tárolására szolgáló két megfelelő tárterület tartalmát, hanem csupáncsak a változókhoz rendelt tárcímeket.

Már önmagában ez is előny lehetne. Vezetjük be a mutatótípus egy olyan speciális értékét<sup>31</sup>, amelynek az a jelentése, hogy nem tartozik az adott mutatóhoz memóriaterület – egyszerűen azt is szoktuk mondani, hogy az a mutató nem mutat sehova, amelyben ezt a speciális értéket tároltuk. Ez lehetővé teszi, hogy a program futása során csak akkor foglaljon helyet egy változó a memóriából, amikor az valóban szükséges. Természetesen ezt a gondolatot adatszerkezetekre is alkalmazhatjuk, sőt tovább is gondolhatjuk. Például egy mutató felhasználásával rendelkezhetünk arról, hogy tartozzon memória az adatszerkezethez vagy sem. A gyakorlatban azonban lehet igény az adatszerkezethez rendelt memóriaterület nagyságának ennél „finomabb”

<sup>30</sup> Kezdő programozók esetében kinek-kinek hosszabb-rövidebb ideig tart, míg valóban különbséget tud tenni a tömbelemek indexe (I) és azon elemek értéke (A[I]) között, amelyekre az adott index(ek) segítségével hivatkozni tudunk. Hasonló nehézséggel találkozhatunk a mutató típus használata során is, amikor különbséget kell tenni a mutató értéke, és a mutató segítségével indirekt módon elérhető memóriatartalom között.

<sup>31</sup> nil, végjel, stb.

változtatására<sup>32</sup>. Ha azonban az adatszerkezet elemeit úgy határozzuk meg, hogy az tartalmazzon további egy vagy több mutatót, akkor az adatszerkezet minden eleméhez további elemet illetve elemeket kapcsolhatunk. Így lehetővé válik az adatszerkezethez tartozó tárterület nagyságának adatelemenkénti változtatása.

Elkülönítünk egy tárterületet kimondottan a dinamikusan kezelt adatok számára. Minden itt tárolt adat címét tárolnunk kell valahol. Ez a fent példából is kitűnik. A legegyszerűbb esetben elegendő egy mutató az adatszerkezet egy eleméhez tartozó tárterület azonosítására, de ez további információkat tartalmazhat, további elemek helyére vonatkozóan. Az ilyen adatszerkezetek egyik csoportját a különböző láncolt listák képezik.

## 8.1. Lista

A legegyszerűbb lista esetében minden listaelem – a tárolandó adaton kívül – egy mutatóval van kiegészítve. Ennek az értékéből tudhatjuk meg, hogy van-e egyáltalán további eleme az adatszerkezetnek, vagy ha van, akkor a memória mely részében található.

Ebből az következik, hogy ennek a homogén adatstruktúrának az egyes elemei a memóriában szétszórtan helyezkednek el, elérésük csak szekvenciálisan lehetséges, de az adatszerkezet tárigénye dinamikusan változhat.

A továbbiakban szeretnénk modellezni magát a dinamikus tárkezelést azzal együtt, hogy konkrét adatszerkezetek műveleteit is bemutatnánk. A modellhez alapul szolgál a mutató – memória és a tömbindex – tömb között vont párhuzam. Modellünkben a dinamikus adatok számára felhasználható tárterületet egy vektor fogja jelenteni, a mutatók szerepét pedig az indexek töltik be.

<sup>32</sup> Például olyankor, amikor nem ismerjük előre a feldogozásra váró adatok számát, vagy a feladat megoldása során az adatok száma dinamikusan változik.

```

1  // ...
2  Típus
3      MutatóTip = Egész;
4      ÉrtékTip = tárolandó_értékek_típusa;
5      ListaElemTip = Rekord
6          Érték: ÉrtékTip;
7          Köv: MutatóTip;
8      RVége;
9  Konstans
10     MaxElem = maximális_elemszám;
11     Végjel = 0;
12  Változó
13     Elem: Tömb [1..MaxElem] ListaElemTip;
14  // ...

```

A lista modellezéséhez szükséges deklarációk

## 8.1. algoritmus

```

1  Változó
2    Szabad : MutatóTip;
3    //...
4  Eljárás SzabadListaKezd;
5  Változó
6    I : Egész;
7  Algoritmus
8    Szabad  $\leftarrow$  1;
9    Ciklus I  $\leftarrow$  1..MaxElem-1
10     Elem[I].Köv  $\leftarrow$  I + 1;
11  CVége;
12  Elem[MaxElem].Köv  $\leftarrow$  Végjel;
13  Evége;

```

A  
 lista modellezéséhez szükséges tárterület inicializációja,  
 a szabad lista fölépítése

## 8.2. algoritmus

```

1 Függvény Lefoglal(Hely: MutatóTip): Logikai;
2 Algoritmus
3   Ha Szabad  $\neq$  Végjel Akkor
4     Hely  $\leftarrow$  Szabad;
5     Szabad  $\leftarrow$  Elem[Szabad].Köv;
6     Lefoglal  $\leftarrow$  Igaz;
7   Különben
8     Lefoglal  $\leftarrow$  Hamis;
9   Hvége;
10 FVége;

```

Tárterület lefoglalása egy elem számára a szabad lista elejéről

### 8.3. algoritmus

```

1 Eljárás Felszabadít(Hely: MutatóTip);
2 Algoritmus
3   Ha Hely  $\neq$  Végjel Akkor
4     Elem[Hely].Köv  $\leftarrow$  Szabad;
5     Szabad  $\leftarrow$  Hely;
6   HVége;
7 EVége;

```

A feleslegessé vált elem visszaláncolása a szabad lista elejére

### 8.4. algoritmus

```

1 Eljárás ListaKezd (Fej: MutatóTip);
2 Algoritmus
3   Fej ← Végjel;
4 EVége;

```

A lista kezdetben nem tartalmaz egyetlen elemet sem

### 8.5. algoritmus

```

1 Eljárás ListaFeldolg (Fej: MutatóTip);
2 Változó
3   P : MutatóTip;
4 Algoritmus
5   P ← Fej;
6   Ciklus míg P ≠ Végjel
7     Feldolgozó_eljárás(Elem[P].Érték);
8     P ← Elem[P].Köv ;
9   CVége;
10 EVége;

```

A lista elemeinek szekvenciális elérése

### 8.6. algoritmus



```

1 Függvény ListaKeresés (Fej,Hely: MutatóTip; Adat:
   ÉrtékTip): Logikai;
2 Változó
3   P : MutatóTip;
4 Algoritmus
5   P  $\leftarrow$  Fej;
6   Ciklus míg (P  $\neq$  Végjel) És (Elem[P].Érték  $\neq$  Adat)
7     P  $\leftarrow$  Elem[P].Köv ;
8   CVége;
9   Hely  $\leftarrow$  P;
10  ListaKeresés  $\leftarrow$  (P  $\neq$  Végjel);
11 EVége;

```

A listában csak szekvenciálisan kereshetünk<sup>33</sup>.

## 8.7. algoritmus

```

1  Függvény ListaBaElöl(Fej: MutatóTip; Adat: ÉrtékTip):
    Logikai;
2  Változó
3    P : MutatóTip;
4  Algoritmus
5    Ha Lefoglal(P) Akkor
6      Elem[P].Érték ← Adat;
7      Elem[P].Köv ← Fej;
8      Fej ← P;
9      ListaBaElöl ← Igaz;
10   Különben
11     ListaBaElöl ← Hamis;
12   HVége;
13  FVége;

```

A lista bővítése az elején

8.8. algoritmus

```

1 Függvény ListaTörölElöl(Fej: MutatóTip): Logikai;
2 Változó
3   P : MutatóTip;
4 Algoritmus
5   Ha Fej  $\neq$  Végjel Akkor
6       P  $\leftarrow$  Fej;
7       Fej  $\leftarrow$  Elem[Fej].Köv;
8       Felszabadít(P);
9       ListaTörölElöl  $\leftarrow$  Igaz;
10  Különbén
11      ListaTörölElöl  $\leftarrow$  Hamis;
12  HVége;
13  FVége;

```

A lista első elemének törlése

8.9. algoritmus

```

1  Függvény ListaBaElemUtán(EzUtán: MutatóTip; Adat:
    ÉrtékTip): Logikai;
2  Változó
3    P : MutatóTip;
4  Algoritmus
5    Ha ( EzUtán  $\neq$  Végjel ) És Lefoglal( P ) Akkor
6      Elem[ P ].Érték  $\leftarrow$  Adat;
7      Elem[ P ].Köv  $\leftarrow$  Elem[ EzUtán ].Köv ;
8      Elem[ EzUtán ].Köv  $\leftarrow$  P;
9      ListaBaElemUtán  $\leftarrow$  Igaz;
10   Különbén
11     ListaBaElemUtán  $\leftarrow$  Hamis;
12   HVége;
13   FVége;

```

Új elem beillesztése a lista egy adott eleme után

#### 8.10. algoritmus

```

1  Függvény ListaTörölElemUtán(EzUtán: MutatóTip):
    Logikai;
2  Változó
3    P : MutatóTip;
4  Algoritmus
5    Ha ( EzUtán  $\neq$  Végjel )  $\wedge$  ( Elem[ EzUtán ].Köv  $\neq$ 
        Végjel ) Akkor
6      P  $\leftarrow$  Elem[ EzUtán ].Köv;
7      Elem[ EzUtán ].Köv  $\leftarrow$  Elem[ P ].Köv;
8      Felszabadít ( P );
9      ListaTörölElemUtán  $\leftarrow$  Igaz;
10   Különben
11     ListaTörölElemUtán  $\leftarrow$  Hamis;
12   HVége;
13   FVége;

```

A lista adott elemét követő elemének törlése

8.11. algoritmus

```

1  Függvény ListaBaVég(Fej: MutatóTip; Adat: ÉrtékTip):
    Logikai;
2  Változó
3    P : MutatóTip;
4  Algoritmus
5    Ha Fej = Végjel Akkor
6      ListaBaVég  $\leftarrow$  ListaBaElől( Fej, Adat )
7    Különbén
8      P  $\leftarrow$  Fej;
9      Ciklus_Míg Elem[P].Köv  $\neq$  Végjel
10       P  $\leftarrow$  Elem[P].Köv;
11       Cvege;
12       ListaBaVég  $\leftarrow$  ListaBaElemUtán(P, Adat);
13    HVége;
14    FVége;

```

A lista bővítése az adatszekezet végén

8.12. algoritmus

```

1 Függvény ListaTörölvég (Fej: MutatóTip): Logikai;
2 Változó
3   P, PE : MutatóTip;
4 Algoritmus
5   Ha Fej ≠ Végjel Akkor
6       Ha Elem[ Fej ].Köv ← Végjel Akkor
7           ListaTörölvég ← ListaTörölEli ( Fej )
8       Különbén
9           PE ← Fej; P ← Elem [ Fej ].Köv;
10          Ciklus_míg Elem[ P ].Köv ≠ Végjel
11              PE ← P;
12              P ← Elem[ P ].Köv;
13          CVége;
14          ListaTörölvég ← ListaTörölElemUtán ( PE );
15      HVége;
16  Különbén
17      ListaTörölvég ← Hamis;
18  HVége;
19  FVége;

```

A lista utolsó elemének a törlése

### 8.13. algoritmus

```

1  Függvény ListaBaRend(Fej: MutatóTip; Adat: ÉrtékTip ):
    Logikai;
2  Változó
3    PE, P : MutatóTip;
4  Algoritmus
5    Ha (Fej = Végjel)  $\vee$  (Adat < Elem[Fej].Érték) Akkor
6      ListaBaRend  $\leftarrow$  ListaBaElöl(Fej, Adat );
7    Különben
8      PE  $\leftarrow$  Fej; P  $\leftarrow$  Elem[Fej].Köv;
9      Ciklus_míg (P  $\neq$  Végjel)  $\wedge$  (Elem[P].Érték < Adat)
10         PE  $\leftarrow$  P;
11         P  $\leftarrow$  Elem[P].Köv;
12      CVége;
13      ListaBaRend  $\leftarrow$  ListaBaElemUtán( PE, Adat );
14      HVége;
15  FVége;

```

Rendezett lista bővítése

8.14. algoritmus



```

1  Függvény ListaTörölRend(Fej:MutatóTip; Adat:ÉrtékTip):
    Logikai;
2  Változó
3    PE, P: MutatóTip;
4  Algoritmus
5    Ha (Fej ≠ Végjel) Akkor
6      Ha Elem[Fej].Érték = Adat Akkor
7        ListaTörölRend ← ListaTörölElöl(Fej );
8      Különbén
9        PE ← Fej; P ← Elem[Fej].Köv;
10     Ciklus_Míg (P ≠ Végjel) ∧ (Elem[P].Érték < Adat)
11       PE ← P;
12       P ← Elem[P].Köv;
13     CVége;
14     Ha (P ≠ Végjel) ∧ (Elem[P].Érték = Adat) Akkor
15       ListaTörölRend ← ListaTörölElemUtán(PE);
16     Különbén
17       ListaTörölRend ← Hamis;
18     HVége;
19   HVége;
20   Különbén
21     ListaTörölRend ← Hamis;
22   HVége;
23   FVége;

```

Rendezett lista egy elemének a törlése

8.15. algoritmus

```

1 Függvény SListaKezd(Fej, Vég: MutatóTip): Logikai;
2 Algoritmus
3   Ha Lefoglal( Fej ) Akkor
4       Vég  $\leftarrow$  Fej;
5       SListaKezd  $\leftarrow$  Igaz;
6   Különben
7       SListaKezd  $\leftarrow$  Hamis;
8   HVége;
9 FVége;

```

A strázsás lista inicializációja

#### 8.16. algoritmus

```

1 Eljárás SListaFeldolg(Fej, Vég: MutatóTip);
2 Változó P: MutatóTip;
3 Algoritmus
4   P  $\leftarrow$  Fej;
5   Ciklus míg P  $\neq$  Vég
6       Feldolgozó_eljárás(Elem[P].Érték);
7       P  $\leftarrow$  Elem[P].Köv ;
8   CVége;
9 EVége;

```

A strázsás lista feldolgozása

#### 8.17. algoritmus

```

1  Függvény SListaKeres(Fej, Vég, Hely: MutatóTip; Adat:
    ÉrtékTip): Logikai;
2  Változó
3    P : MutatóTip;
4  Algoritmus
5    Elem[Vég].Érték  $\leftarrow$  Adat;
6    P  $\leftarrow$  Fej;
7    Ciklus míg Elem[P].Érték  $\neq$  Adat
8      P  $\leftarrow$  Elem[P].Köv ;
9    CVége;
10   Hely  $\leftarrow$  P;
11   SListaKeres  $\leftarrow$  (P  $\neq$  Vég);
12  EVége;

```

Keresés strázsás listában

8.18. algoritmus

```

1  Függvény SListaFűzElemElé(EzElé, Vég: MutatóTípus;
    Adat: ÉrtékTípus): Logikai;
2  Változó
3    P: MutatóTípus;
4  Algoritmus
5    Ha Lefoglal(P) Akkor
6      Elem[P] ← Elem[EzElé];
7      Elem[EzElé].Érték ← Adat;
8      Elem[EzElé].Köv ← P ;
9      Ha Vég = EzElé Akkor
10       Vég ← P;
11     HVége;
12     SListaFűzElemElé ← Igaz;
13   Különben
14     SListaFűzElemElé ← Hamis;
15   HVége;
16 FVége;

```

A strázsás lista bővítése, új elem fölvétele adott elem elé

## 8.19. algoritmus

```

1 Függvény SListaTörölElem(Ezt, Vég: MutatóTípus):
    Logikai;
2 Változó
3   P: MutatóTípus;
4 Algoritmus
5   Ha Ezt  $\neq$  Vég Akkor
6       P  $\leftarrow$  Elem[ Ezt ].Köv;
7       Elem[Ezt]  $\leftarrow$  Elem[P];
8       Ha Vég = P Akkor
9           Vég  $\leftarrow$  Ezt;
10      HVége;
11      Felszabadít ( P );
12      SListaTörölElem  $\leftarrow$  Igaz;
13 Különben
14      ListaTörölElem  $\leftarrow$  Hamis;
15      HVége;
16      FVége;

```

A strázsás lista adott elemének a törlése

## 8.20. algoritmus

További láncolt listák:

- rendezett láncolt lista: nem utólag rendezzük a listát, hanem az új elem felvitele a rendezettség megtartása mellett történik,
- ciklikusan láncolt lista: az utolsó elem mutatója az első elemre mutat,
- többszörösen láncolt lista: egy elemben több mutató is található, így több láncolat mentén is bejárható, azaz több szempont szerint is lehet rendezett,
- két irányban láncolt lista: egy listaelemben két mutató található. Az egyik a következő, a másik az adott elemet követő elem címét tárolja. Lényegében a többszörös láncolás egy speciális esete.

## 8.2. Feladat

1. Adott két kétirányban láncolt lista F1, V1 és F2, V2 fej- és végmutatókkal. Írjon leírónyelvű algoritmust, amely a F1 fejű lista végére fűzi a másik lista elemeit. (Az elemek elérési sorrendje nem változhat. A F1-es lista utolsó eleme után a másik lista első eleme következzen.)
2. Adott két strázsás lista F1, V1 és F2, V2 fej- és végmutatókkal. Írjon leírónyelvű algoritmust, amely az adatszerkezet sajátosságait fölhasználva (ciklus használata nélkül) az F1 fejű lista végére fűzi a másik lista elemeit. (Az elemek elérési sorrendje nem változhat. A F1-es lista utolsó eleme után a másik lista első eleme következzen.)
3. Adott két ciklikusan láncolt lista C1 és C2 feje. Írjon leírónyelvű algoritmust, amely a C1 fejű lista végére fűzi a másik lista elemeit. (Az elemek elérési sorrendje nem változhat. A C1-es lista utolsó eleme után a másik lista első eleme következzen. Az algoritmus legyen alkalmas minden eset kezelésére:
  - a) az 1. és a 2. lista is üres,
  - b) az 1. nem, de a 2. üres,
  - c) stb.)
4. Írjon leírónyelvű algoritmust, amely elvégzi a megadott kétirányban láncolt lista
  - a) első,
  - b) utolsóelemének törlését.
5. Írjon leírónyelvű algoritmust, amely elvégzi a megadott ciklikusan láncolt lista
  - a) első,
  - b) utolsó

elemének törlését.

6. Írjon leírónyelvű algoritmust, amely elvégzi a megadott ciklikusan láncolt lista bővítését a lista

*a)* elején,

*b)* végén.

7. Írjon leírónyelvű algoritmust, amely elvégzi a megadott kétirányban láncolt lista bővítését a lista

*a)* elején,

*b)* végén.

8. Egy listában vegyesen találhatók lányok és fiúk adatai. A fiúk udvariasak, ezért szeretnénk olyan listát kapni, amely előbb az összes lány majd az összes fiú adatait tartalmazza.

9. Egy két irányban láncolt lista elemein Hanyag Elek programozó csak az egyik irányban való láncolást végezte el (ami így egyszeresen láncolt lista). Fejezzük be Elek munkáját.

10. Írjon algoritmust, amely megfordítja egy lista elemeinek sorrendjét.

11. A gyorsabb elérés érdekében homogén elemek sorozatát nem egy listában tároljuk, hanem  $n$  számú, praktikusán közel azonos elemszámú listában és a listafejeket egy  $n$  elemű vektorba szervezzük. Az aktuális adathoz a megfelelő listát az adathoz tartozó *kulcs* segítségével a *kulcs* MOD  $n$  összefüggés alapján rendeljük. Írja meg az adatszerkezet bővítését és az adatszerkezetben való keresés algoritmusát.

12. A dinamikus tárkezelést egy olyan vektor segítségével modellezzük, melynek elemei rekordok. A rekordok egyik mezője a tárolandó adatot, másik pedig az adott elemet követő elem tömbben elfoglalt helyét mutatja meg. Az alábbi táblázat ennek a tömbnek az elemeit tartalmazza. Az első sor az egyes elemek sorszámát, a második a tárolandó

adatot, míg a harmadik az adott elemet követő elem vektorban elfoglalt helyét mutatja meg.

- a) Hány végjeles lista elemeit tárolja a tömb? (Állítását indokolja.)
- b) Milyen sorrendben követik egymást az elemek abban a listában melynek feje F1 és F1=5?
- c) A táblázat ezeken kívül tartalmaz olyan elemeket is, amelyek nincsenek ebben a fenti listában. Mi lehet az értéke az SZ szabad lista fejnek és az F2 listafejnek?

1	2	3	4	5	6	7	8	9	10
Sanyi	Laci	Juli	Évi	Benő	Saci	Pityu	Oszi	Luca	Lipót
3	6	0	2	8	10	9	4	1	0



### 8.3. Fa

A továbbiakban a bináris fa adatszerkezetként történő megvalósítására szorítkozunk, hiszen a segítségével minden más fa megvalósítható.

```
1 Eljárás Sor2Fa(a: sorozat, E,U: egész, P: mutató);
2 Változó
3   K: egész;
4 Algoritmus
5   Ha  $E \leq U$  akkor
6      $K \leftarrow (E + U) \text{ div } 2$ ;
7     Ha Lefoglal(P) akkor
8       Elem[P].érték  $\leftarrow A[K]$ ;
9       Sor2Fa(a, E, K-1, Elem[P].Bal);
10      Sor2Fa(a, K+1, U, Elem[P].Jobb);
11     HVége;
12 Különbén
13      $P \leftarrow \text{VégJel}$ ;
14     HVége;
15 EVége;
```

Vektorban tárolt sorozat elemeinek bináris fában való  
tárolása

8.21. algoritmus

```

1 Eljárás INorder(P: mutató);
2 Algoritmus
3   Ha  $P \neq \text{VégJel}$  akkor
4     INorder(Elem[P].Bal);
5     Feldolgoz(Elem[P].Érték);
6     INorder(Elem[P].Jobb);
7   HVége;
8 EVége;

```

Bináris fa inorder bejárása

8.22. algoritmus

```

1 Eljárás POSTorder(P: mutató);
2 Algoritmus
3   Ha  $P \neq \text{VégJel}$  akkor
4     POSTorder(Elem[P].Bal);
5     POSTorder(Elem[P].Jobb);
6     Feldolgoz(Elem[P].Érték);
7   HVége;
8 EVége;

```

Bináris fa postorder bejárása

8.23. algoritmus

```

1 Eljárás PREorder(P: mutató);
2 Algoritmus
3   Ha  $P \neq \text{VégJel}$  akkor
4     Feldolgoz(Elem[P].Érték);
5     PREorder(Elem[P].Bal);
6     PREorder(Elem[P].Jobb);
7   HVége;
8 EVége;

```

Bináris fa preorder bejárása

## 8.24. algoritmus

### 8.4. Feladat

1. Írjon leírónyelvű algoritmust, amely paraméterül kapja két fa gyökérmutatóját ( $P_1, P_2$ ), valamint egy ADAT értéktípust. A függvény állítsa elő azt a fát, amelynek gyökéreleme az ADAT-ot tartalmazza, baloldali részfája a  $P_1$  által, jobboldali részfája pedig a  $P_2$  által megadott fa, gyökérmutatóját pedig  $P_1$ -ben kapjuk vissza. ( $P_2$ -vel megadott fa a visszatérés után legyen üres.)
2. Írjon leírónyelvű algoritmust (Favágó), amely paraméterül kapja egy fa gyökérmutatóját ( $P$ ). Ha a fa nem „üres”, akkor gyökérelemből kiinduló bal és jobboldali részfákat „levágja”, azok gyökérelemeinek a címét a B és J mutató típusú paraméterekben visszaadja és elvégzi az eredeti fa gyökérelemének törlését.
3. A bináris fa
  - a) intorder,
  - b) preorder,
  - c) postorder

bejáró algoritmusa alapján írjon olyan algoritmust, amelyben az aktuális elem feldolgozása az elem Bal és Jobb mutatójában tárolt ér-

tékek cseréjét jelenti. Rajzolja föl azt a keresőfát, amelynek elemiben az angol ABC első 7 karakterét tároljuk (A...G). Rajzolja föl, hogyan változtatja meg az algoritmus ezt a fát?

4. Rajzolja le a bináris keresőfa szerkezetét, ha következő elemeket a megadott sorrendben szűrjük be egy kezdetben üres keresőfába.

(14; 4; 3; 23; 12; 1; 11; 7; 13; 5)

5. Egy bináris fa elemeit vektorban tároljuk. A gyökérelemet a vektor első elemében. Teljesül továbbá, hogy bármely  $i$ . elem baloldali gyermekét a  $(2i)$ ., jobboldalit pedig a  $(2i + 1)$ . elem tárolja. Ha a fa nem folytatódik egy adott irányban, akkor ott egy speciális értéket tárolunk.

a) Írja meg rekurzívan az inorder, preorder és postorder bejáró algoritmusokat erre a tárolási módra.

b) A keresőfában tárolt *GFC AEBD* elemeket helyezze el az alábbi táblázatban úgy, hogy az a fentieknek megfeleljen. (A táblázat celláiban feltüntetett sorszámok a megfelelő tömbindexeket jelölik.)

1	2	3	4	5	6	7	8	9	10	11	12	13

6. Egy fa elemeit vektorban tároljuk. A gyökérelemet a vektor első elemében. Teljesül továbbá, hogy bármely  $i$ . elem baloldali gyermekét a  $(3i - 1)$ ., jobboldalit pedig a  $(3i + 1)$ . elem tárolja. Harmadik, középső gyermek a  $(3i)$ . elembe van tárolva.

a) Egyértelmű-e a tárolás ezen módja? (Válaszát indokolja.)

b) A fentebb vázolt elv alkalmas-e bináris fa elemeinek tárolására?  
Ha igen, milyen feltételekkel?

c) Írjon rekurzív algoritmust, amely bejárja az adatszerkezetet.

## 9. Visszalépéses keresés

A visszalépéses keresés algoritmusai sokrétű, általános alkalmazhatósága és a hozzá kapcsolódó adatszerkezetek miatt is külön említést érdemel. Bár már jóval előbb ismert volt, gyakorlati alkalmazása csak a számítógépek megjelenésével és elterjedésével vált lehetővé. Elsősorban olyan problémák megoldásánál célravezető az alkalmazása, ahol analitikus megoldást nehezen találhatunk, így szisztematikus próbálgatásokat végzünk.

A témakör tárgyalásánál hagyományosan említjük meg a szintén ezzel a módszerrel megoldható úgynevezett nyolckirálynő-problémát. Itt föltétlen hangsúlyozni szeretnénk, hogy bár ez a feladvány megoldható a visszalépéses keresés módszerével, semmiképpen sem azonos vele. A visszalépéses keresés egy sok területen eredményesen alkalmazható, jóval általánosabb algoritmus. Ennek megfelelően, bár kiindulópontnak ezt a – keletkezését tekintve a XIX. század közepére tehető – feladványt tekintjük, szeretnénk láttatni annak jóval szélesebb körű alkalmazási lehetőségeit.

A probléma megfogalmazása végtelenül egyszerű: helyezzünk el 8 királynőt egy szabványos sakktáblán úgy, hogy a játék szabályai szerint egyikük se legyen ütésben. A Max Bezzel által 1848-ban felvetett problémát 1850-ben Carl Friedrich Gauss és Georg Cantor általánosítva  $n$ -králynő-problémaként fogalmazta meg. Ez a megfogalmazás némileg előrébb visz bennünket ahhoz, hogy a visszalépéses keresést, mint általánosabb megoldási módszert tárgyaljuk.

Ésszerű elgondolásnak tűnik minden királynőt a sakktábla egy-egy sorához rendelni, hiszen – tekintve, hogy a szabályok szerint a királynő vízszintesen, függőlegesen és átlósan léphet – egy figura a vele egy sorban lévő másikat biztosan ütné. Tehát az egyes figurákat csak a hozzájuk tartozó soron belül mozgathatjuk el. Ennek rögzítése után már csak a függőleges és az átlós irányú ütésre kell majd figyelniünk.

Helyezzük el az 1. királynőt a hozzá tartozó (első) sor első pozícióján<sup>34</sup>.

<sup>34</sup> Természetesen ez az elhelyezés pillanatnyilag megfelelő, és a későbbiek során is arra fogunk törekedni, hogy az ütésmentes állapot az újabb figura elhelyezése után is fennmaradjon.

Ezt követően a 2. királynő számára keressünk egy ütésmentes helyet a hozzá tartozó (második) sorban. Folytassuk a megoldást az újabb és újabb királynők elhelyezésével hasonló módon, ameddig ez lehetséges.

Természetesen bekövetkezhet az, hogy az újabb,  $i$ . királynő számára nem találunk megfelelő helyet az  $i$ . sorban. Ekkor alapelvünknek megfelelően – az ütésmentes állapotot fenntartjuk a megoldás során, hiszen ez biztosítja azt, hogy az utolsó királynő elhelyezésekor sem kerül egyetlen figura sem ütésbe – az ezt megelőzően elhelyezett  $(i - 1)$ . királynő számára keresünk új, megfelelő helyet az  $(i - 1)$ . sorban (az  $i$ . királynő pedig lekerül a tábláról). Ha ez sem lehetséges, akkor ugyanezt a műveletet végezzük el az  $(i - 2)$ . királynővel is. Ezeket a visszalépéseket mindaddig végezhetjük, amíg vissza nem érünk az 1. királynőhöz. Ekkor ezt a figurát kell olyan pozícióra helyezni, ahol még korábban nem volt, ami praktikusán a következő mezőt jelenti.

Az algoritmusnak értelemszerűen akkor van vége, ha az utolsó királynő számára is találtunk megfelelő helyet az ő sorában. Ha azonban feltételezzük, hogy a problémának több megoldása is van, akkor az utolsó királynő elhelyezése után – megoldás detektálása mellett – próbáljunk a számára újabb megfelelő helyet keresni. Ha ez nem sikerül, akkor a korábban ismertetett módszernek megfelelően – ezt a figurát levéve a tábláról – az öt megelőző királynő számára keressünk helyet a saját sorában, és inntől járjunk el az előzőeknek megfelelően – ha egy királynőt megfelelően el tudtunk helyezni, akkor a következő számára keresünk helyet, ha pedig nem, akkor az azt megelőző számára keresünk új, megfelelő helyet. Ekkor természetesen bekövetkezhet, hogy az 1. királynő számára már minden rendelkezésére álló helyet kipróbáltunk, ami azt jelenti, hogy nincsenek további megoldások.

Könnyű látni, hogy minden királynő számára – az  $n$ -királynő-probléma esetén –  $n$  darab, egy sorba tartozó pozíció van „fenntartva”. Ezeket formálisan az  $1, \dots, n$  sorozattal tudjuk leírni. Tehát valójában  $n$  darab ilyen sorozatot kell megadnunk a feladat formális leírásához. A feladat megoldásához lényegében minden sorozatnak egy elemét kell kiválasztanunk a szabályoknak megfelelően, hiszen egy figura egyszerre csak egy pozíción lehet, ugyanakkor egy elemét ki kell választanunk, hiszen a figurát el kell helyez-

nünk a táblán. Ezek a kiválasztott elemek matematikai értelemben szintén sorozatot alkotnak. Ezt szemlélteti a 9.1 táblázat. Az  $i$ -vel jelzett oszlop a királynők sorszámát tartalmazza, a „db.” oszlop  $i$ . sora az  $i$ . sorozat elemszámát jelenti, a sor következő nyolc eleme az adott királynőhöz tartozó választható mezők értékeinek sorozata, az  $x_i$ -vel jelzett oszlopból kiolvasható, hogy abból a sorból a sorozat hányadik elemét választottuk, az utolsó oszlop  $i$ . eleme pedig az  $i$ . sorozat kiválasztott elemének értékét jelenti. Tehát a táblázat utolsó oszlopából kiolvasható a feladat egy megoldása, ami szintén egy sorozat formájában adható meg.

$i$	db.	1.	2.	3.	4.	5.	6.	7.	8.	$x_i$	
1.	8	1	2	3	4	5	6	7	8	6.	6
2.	8	1	2	3	4	5	6	7	8	3.	3
3.	8	1	2	3	4	5	6	7	8	7.	7
4.	8	1	2	3	4	5	6	7	8	2.	2
5.	8	1	2	3	4	5	6	7	8	4.	4
6.	8	1	2	3	4	5	6	7	8	8.	8
7.	8	1	2	3	4	5	6	7	8	1.	1
8.	8	1	2	3	4	5	6	7	8	5.	5

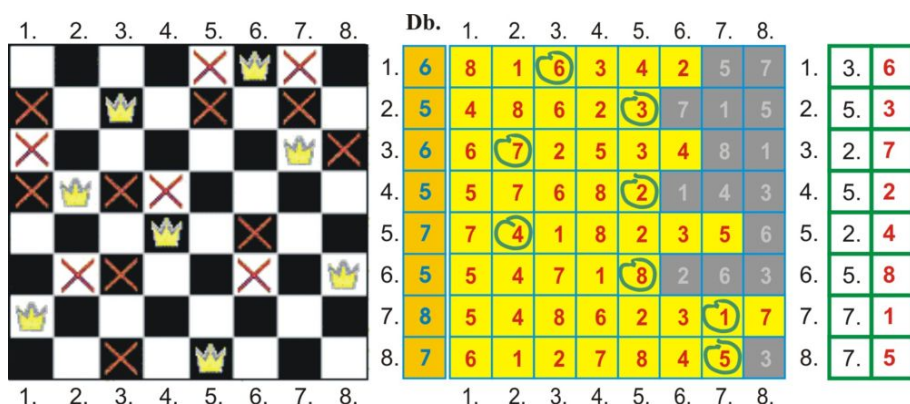
9.1. táblázat

A 8-királynő-probléma adatainak és egy lehetséges megoldásának ábrázolása

A feladat formális leírásából kitűnnek annak specialitásai:

1. a sorozatok elemszáma – a lehetséges pozíciók száma – azonos és elemenként megegyeznek,
2. a sorozatok elemeinek értéke megegyezik a sorszámukkal,
3. a sorozatok rendezettek,
4. a sorozatok száma és elemeinek a száma azonos.

A fentiekből következik az is, hogy ebben a speciális esetben az egyes sorozatok tárolásától eltekinthetünk, hiszen számlálással az elemek előállít-



9.1. ábra

A 8-királynő-probléma egy lehetséges általánosítása

hatók. Ha azonban bevezetnénk olyan megszorításokat, hogy a tábla bizonyos – nem szisztematikusan kiválasztott – pozícióira nem léphetünk, akkor minden királynő esetében szükségessé válik az elhelyezéséhez figyelembe vehető mezők tárolása<sup>35</sup>. Ezt tekinthetjük a 8-királynő-probléma további általánosításának is. Erre láthatunk egy példát a 9.1 ábrán. Az ábrán látható sakktábla bizonyos pozícióit véletlenszerűen letiltottuk és csak a fennmaradóakra kísértük meg a királynő elhelyezését. A táblán jeleztük a letiltott mezőket és a feladat egy lehetséges megoldását.

Az alábbiakban közöljük a fentebb általánosított problémát megoldó algoritmust, melynek értelmezését az Olvasóra bízunk.

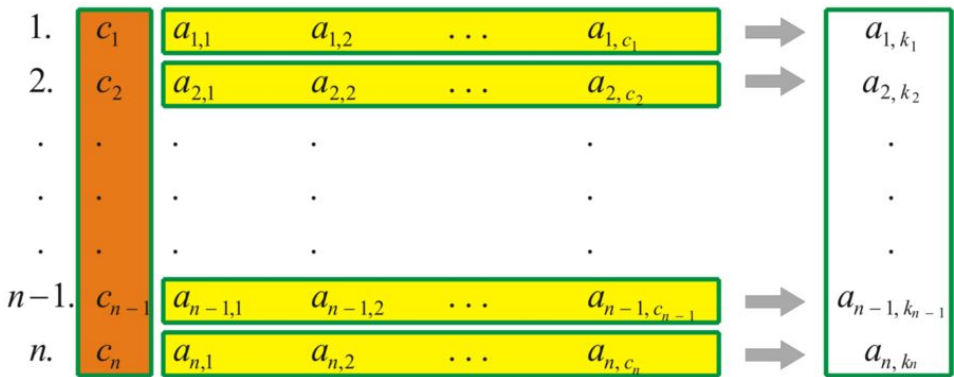
Visszalépéses kereséssel tehát azok a feladatok oldhatók meg, ahol adott  $n$  darab véges, de nem üres sorozathoz kell hozzárendelnünk egy  $n$ -elemű sorozatot úgy, hogy ennek a sorozatnak az  $i$ . elemét az  $i$ . adott sorozat elemei közül választhatjuk, de az elem megválasztását befolyásolja az, hogy a többi sorozatból korábban mely elemeket választottuk.

A 9.2 ábrán sárga mezőben jelöljük az adott  $n$  darab sorozatot. Ezek elem-

<sup>35</sup> A 9.1 ábrán látható sorozatokat úgy állíthatjuk elő, hogy a 9.1 táblázat sorozatainak előállítjuk egy-egy véletlen sorrendjét, és az így nyert – most már rendezetlen, de még azonos elemeket más-más sorrendben tartalmazó – sorozatok utolsó néhány, véletlenszerűen megadott számú elemét elhagytuk. Az letiltott mezők sorszáma szürke színnel van jelezve.



száma rendre  $c_1, \dots, c_n$ . A megoldást jelentő sorozatot az ábra jobb szélén fehér mezőben jelöltük. Ezt a sorozatot egyes sorozatok (sárga mezőben)  $k_1, \dots, k_n$  sorszámú elemei alkotják. Az ilyen feladatok megoldásánál tehát elsődleges fontosságú az adott sorozatok és az elemek kiválasztását befolyásoló szabályok meghatározása.



9.2. ábra  
A visszalépéses kereséssel megoldható feladatok egy lehetséges  
adatrepresentációja.

```

1 Algoritmus BackTr;
2 Konstans
3     ndb= 8;
4     m= 8;
5     t= 25;
6 Típus
7     sorozat_tip= Rekord
8     no: egész;
9     s: tömb [1..m] egész;
10    RVége;
11    stip= tömb [1..ndb]: sorozat_tip;
12 Változó
13    v: stip;
14    x: tömb [1..ndb] egész;
15    k, N: egész;
16 Algoritmus
17    feltolt(v); kiir1;
18    k ← 1; x[k] ← 0;
19    HA VisszalepKeres(k) akkor
20        Ki:('Van_megoldás!');
21        Ciklus k ← 1 .. ndb
22            Ki:(x[k], '_');
23        CVége;
24        kiir2;
25    Különbén
26        Ki: ('Nincs_megoldás!');
27    HVége;
28 AVége.

```

Visszalépéses keresés

9.1. algoritmus

```

1 Függvény VanÜtés( i : egész): logikai;
2 Változó
3   j: egész;
4 Algoritmus
5   j ← 1;
6   Ciklus_Mig (j<i) ∧ (v[i].s[x[i]] ≠ v[j].s[x[j]]) ∧
      (|v[i].s[x[i]]-v[j].s[x[j]]| ≠ (i-j))
7     j←j+1;
8   CVége;
9   VanÜtés ← (j<i);
10 FVége;

```

Van ütés

## 9.2. algoritmus

```

1 Függvény Talál( i :integer): logikai;
2 Algoritmus
3   x[i]←x[i]+1;
4   Ciklus_Mig (x[i]≤v[i].no) ∧ (VanÜtés(i))
5     x[i]←x[i]+1;
6   CVége;
7   Talál←(x[i]≤v[i].no);
8 FVége;

```

Talál helyet

## 9.3. algoritmus

```

1 Függvény VisszalépKeres( i : egész): logikai;
2 Algoritmus
3   Ciklus_Míg (i>1) ∧ (i ≤ ndb)
4     Ha Talál(i) akkor
5       i ← (i + 1);
6       Ha i ≤ ndb akkor
7         x[i] ← 0;
8       HVége
9     Különbén
10      i ← (i - 1);
11    HVége;
12  CVége;
13  VisszalépKeres ← (i>ndb);
14  FVége;

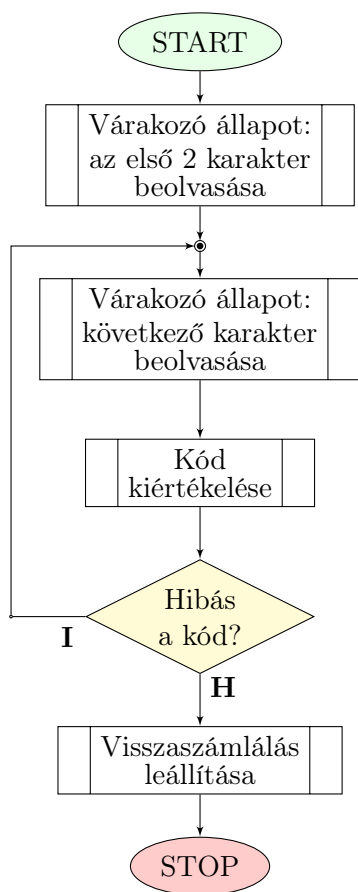
```

BackTr

#### 9.4. algoritmus

Szuper Hősünkre ismét a Világ megmentésének felelősségteljes feladata hárul. A mindent elpusztító robbanást csak az tudja megakadályozni, aki a terminálon begépel a folyamatot leállító négyjegyű kódot. Hős-ünknek sajnos halvány elképzelése sincs a kódról, csupán azt tudja, hogy a rendszer új kódként értelmezi a már begépelte sorozat utolsó 3 elemét az újonnan begépelttel. Ha ez helyes, akkor a visszaszámlálás leáll a szokásos hatalmas vörös kijelzőn, ha pedig nem, akkor egy újabb leütéssel kiegészítve a korábbi elemek utolsó 3 tagját új kóddal próbálkozhatunk. Tehát az első kódot akkor értelmezi a rendszer, amikor a negyedik leütés történik, és innentől minden egyes további leütés lényegében egy újabb kód megadását jelenti (9.3. ábra).

Könnyen belátható, hogy a kódok száma véges. Ha kód alatt négyjegyű, tízes számrendszerbeli számokat értünk, akkor összesen 10 000 ilyen kód le-



9.3. ábra. Kód beolvasó és kiértékelő algoritmus  
(a kód hossza 3)

hetséges<sup>36</sup>. Ezek begépelése a legrosszabb esetben 40 000 leütést jelentene, ha 0000-tól 9999-ig minden számot be akarunk gépelni. Ebben az esetben azonban nem használnánk ki a fentebb említett szabályt, és az feltehetően több időt igényelne. A szabály fölhasználásával vajon mennyivel rövidíthető le ez az idő? Ebben az esetben hány leütésre lenne szükség? Természetesen szeretnénk elkerülni a kódok ismétlődését is. Vajon lehetséges-e ez is?

A feladat szövegében rögzítettük a kód hosszát és bár ezzel kapcsolatban nem tartalmazott információt, feltehetően mindenki úgy gondolja, hogy a kód jegyei tíz félék lehetnek. A probléma általánosítását jelenti, ha azt mondjuk, hogy a kód egy  $k$  jegyű  $n$ -alapú számrendszerbeli szám.

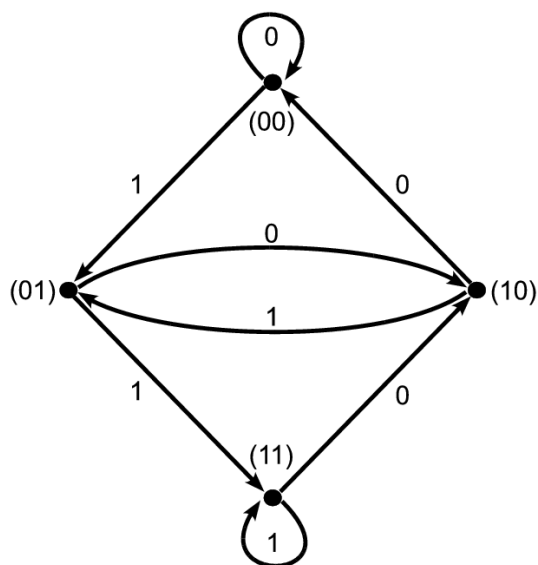
A jobb átláthatóság reményében tekintsük most azt az egyszerűbbnek tűnő esetet, amelyben  $k = 3$  és  $n = 2$ . Ekkor a megoldás egy háromjegyű, kettes számrendszerbeli szám. Tudjuk, hogy három biten 0-tól 7-ig ábrázolhatjuk a számokat, ami 8 különböző kódot jelent.

A rendszer által, a következő leütéskor értelmezendő kód értéke két dologtól függ:

1. Mi volt a két utolsó leütés?
2. Mi lesz a következő leütés?

Az előbbi tekinthetjük a rendszer állapotának és szemléltessük a gráf csomópontjaival, az utóbbit pedig annak az átmenetnek, amelynek hatására új állapotba kerül és ezeket jelöljük a gráf élei. A 9.4 ábán jól látható, hogy ( $k = 3$  és  $n = 2$  esetén) az egyes állapotok kétjegyű számokkal írhatók le, ezért a gráfnak 4 csomópontja lehet. A gráf éleihez írt számjegyek azokat az átmeneteket – leütést – jelölik, amelyek hatására egy másik állapotba kerül a rendszer, és közben értelmezett kód úgy áll elő, hogy az él kiindulópontjához írt számot jobbról kiegészítjük az él jelzésével. Például amikor a rendszer a (00)-állapotban van, és közben 1-et adunk meg, akkor a (01)-állapotba kerül és közben 001 kód áll elő. Ugyanakkor az is leolvasható, hogy az új állapot az aktuális kódból – példánkban 001-ből – úgy származtatható, hogy annak első, legmagasabb helyiértékű jegyét elhagyjuk.

<sup>36</sup> Természetesen az 1 000-nél kisebb számok esetében bevezető nullákat írunk.



9.4. ábra

A „Szuper Hős”-probléma gráfmodellje  
 $k = 3$  és  $n = 2$  esetén.

A kérdés tehát az, hogy mely csomópontból kell kiindulnunk, és mely éleket kell bejárnunk, hogy közben az összes háromjegyű kód előálljon? Természetesen arra is törekednünk kell, hogy a lehető legkevesebb leütés történjen, azaz a lehető legkevesebb élet vegyük igénybe. Az ábráról látható, hogy nincs értelme egy élen többször is „végig menni”, hiszen akkor olyan kódot adunk meg, amit már korábban megadtunk, ugyanakkor minden élt figyelembe kell vennünk egyszer, mert különben lesz olyan kód, ami nem áll elő. Tehát lényegében ebben az esetben is azt kell eldönteni, mint a korábbi két példában. A különbség csupán annyi, hogy jelen esetben irányított gráf a modell. Az ábra alapján az is könnyen belátható, hogy csak akkor tudunk kijelölni az irányított gráfon olyan zárt útvonalat, amely minden élet pontosan egyszer tartalmaz, ha minden csomópontra teljesül, hogy a hozzá tartozó befutó és kimenő élek száma azonos.

A fenti példából levonhatjuk azt a következtetést, hogy  $k$  értéke a csomópontok számát, míg az  $n$  az egy csomópontból kiinduló és az oda befutó

élek számát fogja meghatározni.

## 9.1. Feladat

1. Visszalépéses kereséssel oldjuk meg a korábban ismertetett „Szuper Hős”-problémát általános  $k$  és  $n$  esetén. Mik alkotják az adott sorozatokat, és mik az elemek választásának szabályai?
2. Érdekes optikai jelenség, hogy ha a fénysugát két átlátszó anyag határfelületéhez ér, akkor egy része visszaverődik, egy része pedig átlép a határfelületen és a másik közegben halad tovább. A 9.5 ábra azt szemlélteti, hogy a négy egymással párhuzamos határfelület esetében hogyan viselkedik a legfelső határfelületen (balra fönt) belépő fénysugár. Látható módon egy része visszaverődik a második felületről, egy része tovább halad, amelynek egy része szintén visszaverődik, egy része tovább halad a harmadik határfelülethez és így tovább<sup>37</sup>.

Az ábrán látható, hogy az első visszaverődés 3 különböző módon valósulhat meg. A második visszaverődés már 6 különböző módon jöhet létre, hiszen például a legalsó határfelületről elsőként visszaverődő fénysugár egy-egy része a felső három réteg mindegyikéről részben visszaverődik.

Három határfelület esetén hány különböző módon valósulhat meg 1, 2, 3, 4, 5 visszaverődés?



9.5. ábra  
Fénysugár viselkedése határfelületen.

<sup>37</sup> A fény egy része eljut a legalsó határfelülethez, és egy része természetesen innen is visszaverődik, egyrésze pedig kilép a rendszerünkől. A fénysugárnak ezt a részét nem követjük tovább, ahogyan a felső határfelületen keresztül távozó részét sem.



3. Adjuk meg azt a visszalépéses keresés elvén működő algoritmust, amely az előző feladatot általánosan megoldja, azaz  $n$  határfelület esetén megadja az 1, 2, 3, 4, 5 visszaverődéssel járó utak számát.

## 10. Melléklet

### 10.1. Közismert azonosítók és képzési szabályaik

Az utóbbi mintegy 15 évre visszatekintve megfigyelhető a számítógépek, később a lokális hálózatok mind több területen való megjelenése. Már e két dolog együttese is sok előnnyel járt az egyes cégek, hivatalok ügyviteli munkájában, adatfeldolgozásában. Napjaink és a közeljövő problémája az egyedi számítógépek és a lokális hálózatok világméretű megbízható kommunikációjának biztosítása.

Ha ennek a kihívásnak is sikerül megfelelnünk (és ennek sikeréről az embereket is meg tudjuk győzni), akkor szélesebb körben elterjedhet a (teljesen) elektronikus ügyintézés és az elektronikus kereskedelem. Például egy olyan ügyel kapcsolatban, amelyben több hivatal is érdekelt, a szükséges, különböző okmányok beszerzése az ügyfél feladata (ami általában csak igen sok kilincselés árán lehetséges), holott sok esetben az közvetlenül nem is az ő érdeke. Szerencsére már napjainkban is léteznek jól működő rendszerek. Ilyennel találkozhatunk például cégek alapításakor. Itt a cégbíróság elektronikus úton tartja a kapcsolatot a megfelelő hivatalokkal (APEH, TB, kamara, KSH). Ez egyfelől gyorsabbá teszi az eljárást, kíméli az ügyfél idejét, másrészt megnehezíti a fantomcégek létrehozását.

Sok esetben a továbbblépés gátja elsősorban nem a technikai hiányosság, hanem az, hogy a megfelelő jogi szabályozás is várat magára. Az eddigi sikertelen törvényi beavatkozások kudarcai azzal magyarázhatók, hogy megpróbálták szabályozni a technikai megvalósítást is, holott az általában rövid időn belül elavulttá válik. Napjainkban az ügyvitel gyakorlatára általában mégis az jellemző, hogy a feladó az általa elektronikusan létrehozott dokumentumot hagyományos módon továbbítja, amit aztán a címzett általában szintén elektronikusan rögzít. Közben a faxkészülékek és a nyomtatók ontják a papírra nyomtatott dokumentumok tömegét. Ennek az ellentmondásos állapotnak a feloldása jelentős idő, energia és nyersanyag megtakarítását eredményezhetné. (Csupa olyan dolog, amelyből napjainkban egyre kevesebb van.)

Az Európai Unió országaiban több területen is (szociális ellátás: nyugellátás, egészségügyi, családi ellátás, valamint a baleseti sérültek, fogyatékosok és a munkanélküliek ellátása; munkaügy; statisztikai adatok gyűjtése stb.) indítottak projekteket, amelyekben az EDI (Electronic Data Interchange: elektronikus adatcsere) technikáját sikerrel alkalmazzák. Ezzel összevetve azonban megfigyelhető, hogy a technológia alkalmazásában a Távol-Kelet, az Egyesült államok és Ausztrália jóval előrébb jár. A továbbiakban néhány fontos, gyakran használt, a feladatok megoldásához szükséges azonosító felépítését ismertetjük. Mint az a korábbiak alapján nyilvánvalóvá vált, különböző egyed-előfordulásokhoz az azonosítóknak különböző, egyedi értéke tartozik, melyeknek jelentését különböző szintű megállapodások rögzítene. Ezért nagyon fontos a felépítésük pontos definíciója. Szerkesztésük sokféle módon történhet. Ismeretük azért is fontos, mert az alkalmazott kódszámrendszerek minősége, kidolgozottsága meghatározhatja a teljes rendszer működésének hatékonyságát. A helyesen kialakított kódszámrendszer sokrétű osztályozást, csoportosítást tesz lehetővé, kevés jelet használ, bővíthető, egyszerű felépítésénél, tömörségénél fogva gyors keresési lehetőséget biztosít. Mint látni fogjuk, bizonyos karakterek ill. karakterek csoportja különböző információt kódolhat, míg mások „csak” az azonosítók egyediségét hivatottak biztosítani. Azonosítóként használhatunk egyszerű sorszámot (pl.: TAJ-szám), ami a hozzá tartozó egyed tulajdonságairól semmiféle információt nem hordoz. Alkalmazhatunk betűrövidítéseket (pl.: gépjárművek nemzetközi jelzése, kémiai elemek vegyjele). Gyakran hierarchikus (csoportképző) kódszámrendszereket alkalmazunk (pl.: telefonszám). Ekkor az azonosító bizonyos részei a hozzá tartozó egyed különböző csoportokba való besorolását teszi lehetővé.

Azonosító	A hierarchia szintjei
TAJ	sorszám
Személyi szám	személy neve (állampolgársága) születési idő (év, hó, nap) sorszám
ISBN	ország kiadó kiadvány (sorszám)
Vényazonosító	az orvos azonosítója sorszám

Az így képzett azonosítók néhány pozíción sorszámot is tartalmaznak azért, hogy az azonos csoportba tartozó egyedeket is meg tudjuk különböztetni.

Az azonosító gyakran hosszabb a szükségesnél <sup>38</sup>. Ennek több féle oka is lehet. Későbbi bővítés lehetőségét úgy kívánják biztosítani, hogy bizonyos csoportképzők számára több karaktert foglalnak le az aktuálisan szükségesnél. Néhány azonosítót úgy terveznek meg, hogy azok „beépítve” tartalmazzák a helyességük ellenőrzésének (esetleg hibás olvasás ill. bevitel esetén a hiba javításának) lehetőségét. Ez mind az adatvédelem, mind pedig az adatbázis integritásának megőrzése szempontjából nagyon praktikus kialakítása az azonosítóknak (2.4.4, 2.4.5). Ha a program tartalmazza a megfelelő algoritmust, minimálisra csökkenthető a szándékos vagy véletlen „elírásból” származó hibás adatfelvitel. Ezért a felépítésük bemutatásával együtt néhány esetben az ellenőrzés algoritmusát is megadjuk.

Tételezzük föl, hogy az ellenőrző algoritmusokhoz az azonosítót egy A sorozat elemeiként adtuk meg. (Az egyszerűbb érthetőség kedvéért eltekintünk az elemek esetenként szükséges konverziójától. A rájuk való hivatkozáskor a „[]” jelek között megadott érték a karakter sorozatban elfoglalt helyét jelenti, amit egyszerű balról jobbra történő sorszámozással nyerünk 1-től a sorozat elemszámaig). A TESZT logikai típusú változó értéke pedig attól függően lesz Igaz vagy Hamis, hogy az azonosító helyes volt vagy sem.

<sup>38</sup> Azaz tartalmazhat olyan részeket, amelyek nem az azonosító egyediségét biztosítják, ill. nem az egyedek osztályozását teszik lehetővé.

A számítógépek széles körű elterjedése megteremtette az automatikus azonosítás feltételeit is <sup>39</sup>. Jegyzetünkben nem térhetünk ki az egyes lehetőségek (biometrikus, optikai, mágneses, félvezetős módszerekkel történő azonosítás) technikai megvalósításaira, csupán néhány esetben a kódolt adatokból nyerhető információkat ismertetjük. Napjainkra a vonalkód-technika gyors és biztonságos adatbeviteli lehetőséget kínál, ezért szükségesnek tartjuk, hogy néhány gondolat erejéig említést tegyünk róla. A legelterjedtebb típusok esetében ismertetjük a kód egyes pozícióin található karakterek jelentését és az ellenőrzés lehetőségét. Bár az alapgondolat már jóval korábban megszületett, és a matematikai háttér is régen rendelkezésre áll, de tömeges elterjedésüket az igazán megbízható, olcsó, kisméretű vonalkód-olvasók hiánya sokáig késleltette. Napjainkra azonban ez az akadály elhárult, és alkalmazásuk szinte mindennappossá vált.

#### 10.1.1. Személyi azonosítók képzése

1. A személyi azonosító 11 jegyű.
2. A személyi azonosítók képzése
  - a) az 1. számjegy a személy nemét, születésének évszázadát, az 1997.01.01. előtt születettek esetében állampolgárságukat is kódolja (10.1)
  - b) a 2-7. számjegyet a születési év utolsó két jegye, a hónap és a nap kétjegyű sorszáma adja.
  - c) a 8-10. az azonos napon születettek sorszámát (lajstromszám) jelentő számjegyek.
  - d) a 11. számjegy az ellenőrző kód.
3. A 11. számjegy képzése az előzőekből úgy történik, hogy a számjegyeket megszorozzuk a sorszámaikkal és a szorzatokat összegezzük. A 11.

<sup>39</sup> Bár 1932-ben még egyáltalán nem volt jellemző a „számítógépek széles körű elterjedése”, Wallace Flint (Harvard Egyetem) megálmodta talán az első ilyen rendszert. A lyukkártyaolvasó által vezérelt berendezés a kártyán lévő kódnak megfelelő árut automatikusan továbbította a raktárból a pénztárhoz, elkészítette a számlát és módosította a készletnyilvántartást.



1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
8									
a				b			c		d

10.3. táblázat  
Az adóazonosító jel szerkezete.

- b) 2-6. a számjegyek a személy születési időpontja és a 1867.01.01. dátum között eltelt napok száma.
- c) 7-9. a számjegyek az azonos napon születettek között véletlenszerűen kiosztott sorszámot tartalmazzák.
- d) a 10. számjegy az ellenőrző szám.

3. A 10. számjegy képzése az előzőekből úgy történik, hogy a számjegyeket megszorozzuk a sorszámmal és ezeket a szorzatokat összegezzük. A 10. számjegy a szorzat 11-gyel való osztásának maradéka. (Azok a 2. c szerinti sorszámmak nem adhatók ki, amelyekre a maradék 10.)

### 10.1.3. Társadalombiztosítási azonosító jel képzése

1. A TAJ-szám 9 jegyű.
2. Képzési szabályok:
  - a) 1-8. egy folyamatosan kiadott sorszámmal.
  - b) A 9. számjegy az ellenőrző ún. CDV kód. Képzésekor a páratlan helyen álló számjegyeket hárommal, a páros helyen állókat héttel kell megszorozni és a szorzatokat összegezni. A CDV kód az összeg tízzel való osztásának maradéka.

### 10.1.4. Vényazonosító

Ez a kód a gyógyszerek és a gyógyászati segédeszközök eladási tételeinek azonosítását teszi lehetővé. Segítségével a vényt kiállító orvos személye is meghatározható, mert tartalmazza az ő azonosítóját is. Mivel a vényen vonalkód

1.	2.	3.	4.	5.	6.	7.	8.	9.
3	7	3	7	3	7	3	7	
a							b	

10.4. táblázat  
A TAJ-szám fölépítése.

formájában is megtalálható, így egyben példa az EAN-13 zárt rendszerben történő alkalmazása.

1. A vényazonosító 13 jegyű.
2. Információk a vényazonosítóképzésével kapcsolatban
  - a) Nem dokumentáljuk
  - b) az orvos egyedi azonosítójának tekinthető ún. „pecsétszám”.
  - c) Nem dokumentáljuk
  - d) folyamatos ötjegyű sorszám.
  - e) ellenőrzőkód, melynek értékét az EAN-13 típusú vonalkód ismeretetésénél leírtaknak megfelelően számíthatjuk ki.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
a	b					c	d				e	

10.5. táblázat  
A vényazonosító részei.

Megjegyzés: A gyógyszertárban az eladott gyógyszerhez tételenként akkor is rendelnek egy 13 jegyből álló azonosítót, ha az nem „vényköteles”.

- a) a vény hiányának oka (91, 94, 95, 97, 98, 99)



b) 11 jegyű folyamatos sorszám

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
a			b									

### 10.1.5. Az ISBN (International Standard Book Number)

A könyveknek ezt a nemzetközileg elfogadott azonosítóját 1974. január 1. óta Magyarországon is használják. Alkalmas minden önálló mű (de nem példány és általában nem kiadás)<sup>40</sup> egyértelmű azonosítására. Az ISBN hazai koordinációját az Országos Széchényi Könyvtár végzi. Az azonosító négy fő szerkezeti egységre bontható. Az első 9 jegy felbontásában lehetnek eltérések, de az utolsó karakter mindig az ellenőrző kód funkcióját tölti be. Az alábbiakban egy lehetséges felosztás alapján ismertetjük az ISBN képzésére vonatkozó szabályokat.

1. Az ISBN 10 jegyből áll
2. A pozíciók sorszámozását jobbról balra végezzük.
  - a) a 10-8. számjegyek az ország kódja  
(pl.: Magyarország esetében 963).
  - b) a 7-5. számjegyek a kiadó kódja.
  - c) a 4-2. számjegyek a kiadványt azonosítják.
  - d) az 1. számjegy az ellenőrző kód
3. Az ellenőrző kód képzése az előzőekből úgy történik, hogy mindegyiket megszorozzuk a sorszámával és a szorzatokat összegezzük. Az 1. számjegy úgy számítható, hogy az összeg 11-gyel való osztásának maradékát kivonjuk 11-ből, ha az 1-nél nagyobb. Ha azonban a maradék

<sup>40</sup> Bizonyos esetekben előfordulhat, hogy egyazon mű különböző kiadásainak más az ISBN-je, de ezt általában a kiadó kódjának változása indokolja. Ekkor a korábbi kiadás(ok) azonosítóit is feltüntetik. Többkötetes művek esetén, az egyes kötetek rendelkeznek egyedi és összefoglaló azonosítóval is.

10	9	8	7	6	5	4	3	2	1
a			b			c			d

10.6. táblázat  
Az ISBN fölépítése.

0, akkor az ellenőrző kód is 0 lesz, ha pedig 1, akkor a kód helyébe „x”-et írunk, mert ebben a két utóbbi esetben a fenti különbség nem volna megadható egy pozíción.

Pl.: 963 184 210 x, 071 050 523 x, 058 251 690 0

A vonalkód-technikát fölhasználták az ISBN számok megjelenítésére is. Erre az EAN-13 típus bizonyult a legalkalmasabbnak.

- a) az első három pozíción minden esetben 978 kombináció található. Ez azt jelzi, hogy az EAN-13 típusú vonalkód könyvet azonosít.
- b) a 4-12. pozíción található kilenc karakter az eredeti ISBN számjegyei, az ellenőrző kód nélkül.
- c) a 13. karakter ellenőrző funkciót tölt be az EAN-13 ismertetésekor leírtaknak megfelelően. (Ez tehát szükségtelenné teszi az ISBN utolsó pozícióján található ellenőrző kódjának szerepeltesét, ami egyben lehetetlen is, ha az „x”, mert ez a vonalkódtípus csak számok ábrázolását teszi lehetővé.)

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
9	7	8										
a			b								c	

10.7. táblázat  
Az ISBN megjelenítése EAN-13 vonalkóddal.

Pl.: A 963 184 210 x, ISBN számnak megfelelő vonalkód a következő számsorozatot kódolja

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.		
9	7	8		9	6	3	1	8	4	2	1	0		4
a				b							c			

#### 10.1.6. Az ISSN (International Standard Serial Number)

Az ISSN folyóiratok, hírlapok, évkönyvek, időszakosan megjelenő jelentések, közlemények, különböző adattárak, időszakosan megrendezett konferenciák kiadványainak azonosítására használatos kód. Nemzetközi központja 1972 óta működik Párizsban. A nyolc karakterből álló azonosítót két négyelemű karaktercsoportra bontva, egymástól kötőjellel elválasztva tüntetik fel. (Az előzőekben ismertetett ISBN-től eltérően egyes elemei semmiféle jelentést nem hordoznak.) Az utolsó, nyolcadik pozíción az ellenőrző kód található. Helyességének ellenőrzése az ISBN-éhez hasonló algoritmus alapján történhet. Egyre több folyóiraton találhatunk az azonosításukat segítő vonalkódokat, melyeknek alapja szintén az EAN-13. Ezekből természetesen kiolvasható az ISSN.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.		
9	7	7									0	0		
a				b						c		d		

10.8. táblázat

Az ISSN megjelenítése EAN-13 vonalkóddal.

- a) az első három pozíción minden esetben 977 áll. Ez azt jelzi, hogy az EAN-13 típusú vonalkód ISSN számot kódol.
- b) a 4-10. pozíción található hét karakter az eredeti ISSN számjegyei, az ellenőrzőkód nélkül.

c) a következő két pozíción mindig 00 áll.

d) a 13. karakter ellenőrző funkciót tölt be az EAN-13 ismertetésekor leírtaknak megfelelően,

Pl.: A ISSN 0864 9421-nek vonalkódos formában a következő számso-  
rozat felel meg:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
9	7	7	0	8	6	4	9	4	2	0	0	6
a				b						c		d

### 10.1.7. Bankkártyaszám

1. A bankkártyaszám (Magyarországon) 16 jegyű.

a) az első négyelemű számcsoport a bankot azonosítja

Pl.: Budapest Bank: 5892

OTP Bank: 4909

b) A bankkártyaszám valódiságának vizsgálata a Luhn<sup>41</sup>-algoritmus felhasználásával történik. Az ellenőrzés során balról jobbra haladva a páratlan pozíción álló számjegyeket 2-vel megszorozzuk. Ha a szorzat értéke 9-nél nagyobb, a szorzatból kivonunk 9-et. Az így kapott számok összegéhez hozzáadjuk a páros sorszámú számokat. Ha ez az összeg 0-ra végződik, az azonosító helyes.

Pl.: Az 1234 5678 9012 3452 bankkártya-szám esetén:

2. A bankkártyaszám valódiságának vizsgálata a Luhn -algoritmus felhasználásával történik. Az ellenőrzés során balról jobbra haladva a páratlan pozíción álló számjegyeket 2-vel megszorozzuk. Ha a szorzat értéke 9-nél nagyobb, a szorzatból kivonunk 9-et. Az így kapott számok összegéhez hozzáadjuk a páros sorszámú számokat. Ha ez az összeg 0-ra végződik, az azonosító helyes.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1

	a	
--	---	--

10.9. táblázat  
A 16 jegyű bankkártyaszám fölépítése.

Pl.: Az 1234 5678 9012 3452 bankkártya-szám esetén:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.															
2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1															
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	2															
2	+	2	+	6	+	4	+	1	+	6	+	5	+	8	+	9	+	0	+	2	+	2	+	6	+	4	+	1	+	2

#### 10.1.8. EAN-13, EAN-8 (European Article Numbering)

Az európai kiskereskedelemben talán ezt a kódrendszert alkalmazzák legelterjedtebben az áruk azonosítására, de akár szélesebb körben elterjedt kód-szabványnak is tekinthető.

1. az EAN-13 vonalkód 13 jegyű karaktersorozat, csak numerikus értékek ábrázolására alkalmas.
2. információk az azonosító képzésével kapcsolatban
  - a) 1-2. vagy 1-3. a számjegyek a termék származási helyét, pontosabban annak a szervezetnek az azonosítóját adják meg, amely a gyártó kódját kiadta. (Magyarország azonosítója 599, Olaszországé 80-83. A 20 és 29 közötti értékek belső használatra vannak fenntartva. Mint korábban láttuk, a 977 és a 978 le van foglalva az ISSN és az ISBN számára.)

<sup>41</sup> H. Peter Luhn az IBM kutatója nyomán.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	
1	3	1	3	1	3	1	3	1	3	1	3		
		a		└─┘		b				c		d	

10.10. táblázat  
Az EAN-13 vonalkód fölépítése.

- b) A következő négy vagy öt számjegy a termék gyártóját azonosítja.  
(Ezek kiosztását a Magyar Gazdasági Kamara Csomagolási és Anyagmozgatási Országos Szövetség ETK/EAN Irodája végzi.)
- c) A további karakterek az utolsó kivételével a terméket azonosítják.  
Ezek megfelelő módon történő megválasztása a gyártó felelősége.
- d) a 13. számjegy az ellenőrző kód.
3. A 13. jegy képzése úgy történik, hogy az első 12 jegyet paritásának megfelelően eggyel, ill. hárommal megszorozzuk, és a szorzatokat összegezzük. A 13. pozícióra az a számjegy (0-9) kerül, amellyel az összeg tízzel oszthatóvá tehető.

## Irodalomjegyzék

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Új algoritmusok, SCOLAR INFORMATIKA, 2003.
- [2] Járdán Tamás, Pomaházi Sándor: Adatszerkezetek és Algoritmusok, Líceum Kiadó, Eger 1996.
- [3] Iványi Antal: Informatikai algoritmusok 1-2., ELTE Eötvös Kiadó, 2004.
- [4] <http://algo-rythmics.ms.sapientia.ro/>