INFORMATIKA

Király Sándor

A programozás logikája II.

20

KÖZÉPISKOLAI SEGÉDKÖNYV

INFORMATIKA 20.

KIRÁLY SÁNDOR

A programozás logikája II.

KÖZÉPISKOLAI SEGÉDKÖNYV

SEGÉDKÖNYVI ENGEDÉLYSZÁM: OM 34.588/8/98.XIV.

SOROZATSZERKESZTŐ:

DOMBOVÁRI MÁTYÁS

LEKTOR:

DR. NÉMET ISTVÁN
ELTE KÍSÉRLETI GYAKORLÓ GIMNÁZIUM
ÉS SZAKKÖZÉPISKOLA

FELELŐS SZERKESZTŐ:

SZALAY ÁGNES

ISBN 963 7309 19 5
GO-0020

A KIADÁSÉRT FELELŐS: A GRADUATION BT ÜGYVEZETŐJE.
TÖRDELŐ SZERKESZTŐ: DRIMMER PÉTER
MŰSZAKI SZERKESZTŐ: BELEZNAINÉ S. ANNAMÁRIA
NYOMDA: R+V TERCIA KFT. - B.EST. NYOMDA, BUDAPEST

1999.

KÉSZÜLT A VILÁGBANKI PROGRAM KERETÉBEN,
A MŰVELŐDÉSI ÉS KÖZOKTATÁSI MINISZTÉRIUM
ÉS A KÖZISMERETI INFORMATIKA CSOPORT
TARTALMI GONDOZÁSÁBAN

A SEGÉDKÖNYVVEL KAPCSOLATOS ÉSZREVÉTELEKET
SZÍVESEN FOGADJUK.

GRADUATION BT. 2045 TÖRÖKBÁLINT, PF. 35.

Tartalomjegyzék

Tartalomjegyzék ... 4
Bevezetés ... 7
I. Adattípusok, adatszerkezetek ... 8
Összetett adattípusok, adatszerkezetek (Egyéb típuskonstrukciók) ... 8
Rekord ... 8
Alternatív rekord ... 9
(Hatvány)Halmaztípus-konstrukció ... 10
Fájlok ... 11
Bináris fák ... 15
II. A rekurzió ... 17
Gyorsrendezés (Quicksort) ... 24
III. A visszalépéses keresés tétele (backtrack) ... 29
Feladatok ... 42
IV. Gráftípusok ... 45
Bevezetés ... 45
Tárolási módok ... 46
Adjacencia-mátrix (csúcs-, szomszédsági mátrix) ... 46
Adjacencia-lista (csúcs-, szomszédsági lista) ... 47
Incidencia-mátrix ("pont-él" mátrix) ... 47
Mutatós ábrázolás ... 48
A gráfbejárás algoritmusai ... 49
Szélességi bejárás ... 49
Mélységi bejárás ... 51
Útkeresési feladatatok ... 53
Hamilton-kör és -út keresése ... 54
Labirintusproblémák ... 58
Feladatok ... 63
V. Numerikus és statisztikai módszerek ... 65
Lineáris egyenletrendszerek megoldása ... 65
Gauss-féle kiküszöbölési eljárás ... 67
a) Gauss elimináció ... 67
b) A Gauss-Jordan algoritmus ... 70
A determináns ... 70
Mátrixinvertálás ... 75
Interpoláció ... 76
Lineáris interpoláció ... 77
Egyismeretlenes nemlineáris egyenlet ... 78
Gyökbehatárolás intervallum felezéssel ... 79
Húr módszer ... 80
Numerikus integrálás ... 82

A téglalap módszer ... 83
Trapéz módszer ... 84
Korreláció- és regressziószámítás ... 86
Idősorok vizsgálata ... 89
Lineáris trend ... 90
Exponenciális trend ... 91
Feladatok ... 95
VI. ELEMI GRAFIKA ... 97
Szakasz rajzolása ... 97
Görbék ... 103
Interpoláló görbék ... 104
Lagrange féle interpoláció ... 104
Közelítő görbék ... 106
Bezier-görbék ... 106
B-spline-ok ... 106
Kör rajzolása ... 107
Forgatás ... 108
Feladatok ... 110
IRODALOMJEGYZÉK ... 111

Bevezetés

Ez a könyv a *Programozás logikája I* című könyv folytatásának tekinthető. Feltételezi az első rész tartalmának ismeretét, hiszen olyan problémák és megoldásuk találhatóak ebben a könyvben, melyek egy programozási ismereteket most kezdő számára bizony egyáltalán nem érthetőek. Az algoritmusok megadása verbális leírással történik, ami nagyban segíti a gépi reprezentációt. A könyv azon tanulók számára készült, akik magasabb szinten szeretnének foglalkozni a programozással, nem elégednek meg az alapokkal.

Az első fejezet néhány összetett adatszerkezet jellemzőit tartalmazza, csak azokat, amelyek nem kerültek ismertetésre az első részben.

A második fejezet az alapoktól kezdve ismerteti a rekurziót, fogalmát, használhatóságát, jellemzőit, és bemutat néhány alkalmazást.

A harmadik fejezet szerves folytatása a visszalépéses algoritmusok bemutatása (negyedik fejezet), hiszen egy részük a rekurzióra épül. A klasszikus problémák bemutatásán keresztül kerül ismertetésre a téma, mellyel igen sok feladat oldható meg.

A negyedik fejezet témája a gráfok problémaköre, amelyben gráfbejárási, útkeresési és labirintussal kapcsolatos feladatok kerülnek megoldásra.

Az ötödik fejezetben néhány algebrai és numerikus matematikai probléma algoritmusok segítségével történő megoldása található

A hatodik fejezet grafikával foglalkozik, ismerteti az elemi grafikai feladatokat, a szakaszrajzolástól a függvényábrázolásig.

Minden fejezet végén feladatok találhatóak, amelyek megoldása a tanuló számára ajánlott az anyag elsajátítása érdekében.

I. Adattípusok, adatszerkezetek

Összetett adattípusok, adatszerkezetek (Egyéb típuskonstrukciók)

Rekord

Lényege, hogy az összetett változónak (rekord) adunk egy nevet, s az egyes, rendszerint különböző típusú részeinek (mezőinek) szintén. Így lehetővé válik a rekord elemeinek önálló, külön kezelése és a részek együttes feldolgozása is.

A rekordnak nincs adatszerkezet megfelelője, strukturált típus, ami mögött nincs adatszerkezet. A rekord a nyelvekben egy heterogén és általában dinamikus, strukturált adattípus. A heterogén annyit jelent, hogy a rekord különböző típusú adatelemekből épülhet fel. A rekordnak mezői vannak - rekord mezőkről beszélünk, - és a mezők tetszőleges típusúak lehetnek. Nagyon sok nyelvben a rekord mérete változhat futás közben. A PASCAL is ilyen.[1] Más-más mező tartozhat a rekordhoz.

A rekordnak van fix része és lehet változó része. Ha csak fix rész van, akkor az annyit jelent, hogy az adott rekordtípus mindig ugyanolyan szerkezetű.

Ha deklarálunk egy rekord típusú változót, akkor a teljes rekordra, az összes mezőre együtt hivatkozhatunk a rekord nevével, abban a sorrendben, ahogy a mezőket felírtuk. Tudunk hivatkozni a mezőkre is külön-külön - pontozott jelöléssel (ez a minősített név):

rekordnév.mezőnév (a típusban deklarált mezőnév)

Amikor megmondom, hogy melyik rekord melyik mezőjéről van szó, ezt minősítésnek hívják a számítástechnikában - a név pedig a minősített név. A rekord a mezők típusától függően nem biztos, hogy folytonos tárterületet foglal le - lehet, hogy lyukak vannak a foglalt tárterületben.

Strukturálás: Rekordtípus = *Rekord*
1. mezőszelektor : értéktípus,
2. mezőszelektor : értéktípus,
.
.
.

Értékhalmaz: a mezők értéktípusai által meghatározott alaphalmazok direkt szorzata

Műveletek: szelekciós függvény (". mezőszelektor" nevű) konstrukciós függvény (Rekordtípus nevű), (elképzelhetők) transzfor-

---

[1] Valójában a leghosszabbnak megfelelő helyet foglalja le a rekord számára.

mációs függvények, amelyek a teljes rekordstruktúrát érintik

Relációk: = (mezőnkénti egyezés), ≠ [2]

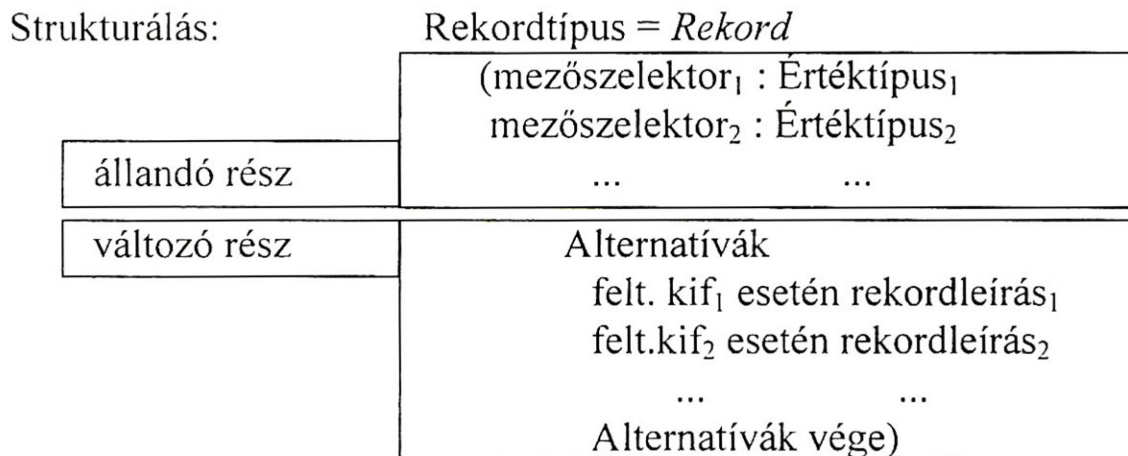
Pl.: **Típus** Dátum = *Rekord*
(év: 0.. 2000,
hó : 1..12,
nap: 1..31)

Konstans Béke : *Dátum*= (év:1945, hó: 5, nap: 9)

Alternatív rekord

Abban különbözik a rekordtól, hogy az összetételében változhatnak a részek is, a részek tartalmától függően (azaz a rekord nemcsak fixrészből áll).

Személyi adatnyilvántartás esetében természetes, hogy mindenféle adatot nem kell mindenkiről tárolni. Így szükségünk lehet arra, hogy egy összetett adatszerkezet (rekord) többféle, alternatív szerkezetben valósítsunk meg. Ez a többféleség persze a rekord valamennyi mezőinek az értékeitől függ. (Pl. leánykori neve csak nőknek van). Ha a nyelv nem adja meg ennek a szerkezetnek a támogatását, akkor mindenkiről minden adatot nyilván kell tartani.



(Itt a felt.kif feltételes kifejezést, a rekordleírás$_j$ a rekord törzsét jelöli)

Értékhalmaz: az egyes alternatív összeállítású mezők értéktípusainak direktszorzat - uniója

Műveletek: szelekciós függvény (".mezőszelektor" nevű), konstrukciós függvény (Rekordtípus nevű), elképzelhetők transzformációs függvények is, amelyek a teljes rekordstruktúrát érintik.

Relációk: = (mezőnkénti), ≠

[2] A mezők maguk is lehetnek összetettek, ilyenkor értelemszerűen "lebontva az egyszerű típusú részig" értendő! A TP nem ismeri.

Ábrázolás: az egyes mezőknek (komponenseknek) szükséges mennyiségű tárterület folytonos tartománya, a leghosszabbhoz igazítva.

Pl.:

Típus Adat=*Rekord*
Szemigszam:*Szöveg*[8]
Neve:*Szöveg*[35]
Kora: *Egész*
Neme:*Karakter*
Alternatívák
Ha Neme="F": Katonaigszam:*Szöveg*[8]
Ha Neme="N": Leánykorinev:*Szöveg*[30]
Alternatívák vége

(Hatvány)Halmaztípus-konstrukció

Olyan adatszerkezet, melyet szintén a használati módjával definiálunk. Matematikai halmaz, mint adatszerkezet. A halmaznak vannak elemei. Egy adatelem csak egyszer fordulhat elő a halmazban (azonos értékű adatelemek nem lehetnek a halmazban). Más adatszerkezetre ez nem igaz. Tudjuk azt vizsgálni, hogy egy adott érték eleme-e a halmaznak vagy nem, tudjuk a szokásos halmazműveleteket végezni (unió, metszet, különbség, részhalmaz stb.).

Strukturálás: Halmaztípus = *Halmaz*(Elemtípus)[3]
Értékhalmaz: az alaphalmaz (Elemtípus által meghatározva) iteráltja ("mely elemek vannak benne a halmazban")
Műveletek: * (metszet), + (egyesítés), - (különbség), Üres (Üreshalmaz létrehozás: eljárás), vagy Üres'Halmaztípus előre definiált konstans, Üres? (logikai értékű függvény)
Relációk: $=, \neq, <, \leq, >, \geq$ (parciális rendezés: a tartalmazás alapján)
Ábrázolás: ahány elem, annyi biten, logikai értelmezéssel: eleme (1=**Igaz**) vagy nem (0=**Hamis**): vagy az elemeinek felsorolásával (egyfajta sorozatként reprezentálva)

Pl.:

Típus Nap = 1..31
Foglalt = *Halmaz*(nap)
Változó ma, holnap : Foglalt
Konstans munkanap :Foglalt= (7..11,14..18)

[3] Mivel a "halmaz-ság" tulajdonság nem teszi szükségessé elemeinek közvetlen kiválaszthatóságát (emiatt halmaz, s nem vektor!), ezért nincs is komponenseire vonatkozó szelekciós függvény, e típuskategóriához, s így az értelmezési tartomány megadásától a definícióban akár el is lehet tekinteni.

ma:=[12]
holnap:=[13]
Ha ma+holnap **Nem** Eleme(munkanap) **akkor Ki**: "Hétvége"

Fájlok

Funkció szerint az alábbi típusú fájlok léteznek:

Input funkciójú fájl: olyan fájl, amely létezik, ebből olvasunk és a fájl változatlanul továbbra is létezni fog (az olvasást és írást mindig a processzor felől nézzük).

Output: Olyan fájl, amely nem létezett, most hozzuk létre, ebbe csak írunk (lényeges változás következik be).

Update: Olyan fájl, amelybe eddig is létezett és ezután is létezni fog, azonban megváltozik a tartalma. Olvasni is tudunk belőle és írni is tudunk bele.

Ez a programban vagy a programozásban a tevékenységben betöltött szerepüket jelenti a fájloknak.

Fájlokkal kapcsolatos műveletek:

1. *Létrehozás*: a következő alműveletek kapcsolódnak hozzá.

a) *Összekapcsolás*: logikai és fizikai fájl összekapcsolása. Amikor a programban leírt logikai fájlhoz megmondjuk, hogy majd melyik fizikai fájl fog hozzátartozni, amivel majd dolgozni akarunk. A létrehozás folyamán értelemszerűen csak output funkciójú fájl jöhet szóba, hiszen most hozzuk létre. Pl. erre az összekapcsolásra a PASCAL-ban az ASSIGN (ejtsdösszájn) nevű eljárás szolgál. Semmi mást nem csinál az összekapcsolás mint művelet, csak azt mondja, ehhez a logikai állományhoz majd egy ilyen nevű fizikai állomány fog tartozni.

A PASCAL-ban ez a nyelvnek a része, más nyelveknek nincs ilyen része, a nyelven kívül kell megcsinálni az összekapcsolást. Ez az összekapcsolás még csak a lehetőségét teremti meg arra, hogy a logikai és fizikai fájl valamilyen módon kapcsolatba kerüljön.

Ez egy előkészítés és nem egy tényleges kapcsolat.

b) *Fájl megnyitása* (megnyitás, mint fogalom): bármilyen fájlt, amivel dolgozni akarunk, meg kell nyitni. Minden nyelvben van ilyen eszköz, amivel a fájl megnyitását el lehet végezni. PASCAL-ban több ilyen eszköz is van.

Lényegében a megnyitás az, ami a tényleges kapcsolatot megteremti a logikai és fizikai fájl között. Pl. létrehozásnál egy csomó adminisztrációt elvégez a megnyitás. Egy fizikai állománynak nagyon sok jellemzője van. Ezeket a jellemzőket le kell adminisztrálni a lemezen. A lemez tartalomjegyzékébe bekerül az adminisztráció (név, területfoglalás stb.), előkészíti a lemezt arra, hogy fogadni tudja a fizikai fájlt. Csak megnyitás után tudunk dolgozni.

Input típusú fájloknál pedig megtörténik annak ellenőrzése, hogy létezik-e a megnyitni kívánt file.

Állomány létrehozásánál lényeges, hogy milyen szervezési módot alkalmazunk, milyen szervezésű állománnyal akarunk dolgozni? A szervezési mód nagyon lényeges kérdés. A létrehozáskor kell eldönteni, hogy milyen szervezésű állományt hozunk létre.

A PASCAL-ban nincs állománykezelő. Többféle osztályozása van a szervezési módoknak. Az egyik osztályozásnál beszélünk:

- *egyszerű* és
- *összetett* állományszerkezetről vagy szervezésű állományokról.

Egyszerű állományszerkezetben csak a tényleges adatok vannak benne, azok az adatok, amelyekkel dolgozni akarunk majd.

Az összetett állományszerkezetben a fenti adatokon túlmenően ún. szerkezethordozó adatok is vannak. Ezek olyan információk, amelyek a fizikai állomány adott periférián való elhelyezkedésére vonatkozó adatokat tartalmaznak (tehát magára az állomány szerkezetére vonatkozó információk is).

Egyszerű állományszerkezet

Az egyszerű állományszerkezetek további osztályozásánál két szempontot szokás figyelembe venni:

1. Az állományban elhelyezkedő rekordok azonosítói között van-e valamilyen kapcsolat?
2. A rekordok azonosítói és a rekordnak az adott periférián elfoglalt (fizikai) helye között van-e kapcsolat?

Ennek megfelelően négyféle egyszerű állományszerkezetről szokás beszélni:

	1. kérdés	2. kérdés	Létrehozható
1. szeriális	nincs	nincs	szalagon és lemezen
2. szekvenciális	van	nincs	szalagon és lemezen
3. direkt	van	van	lemezen
4. random	nincs	van	lemezen

Összetett állományszerkezet is csak lemezen hozható létre.

Szeriális:

- szerkezet nélküli állomány
- az egyetlen olyan állomány, mely nem használja ki, hogy van rekordazonosító - ez annyit jelent, hogy a rekordok tetszőleges sorrendben vannak egymás után (ott, ahol vannak) - nem tudjuk megmondani, hogy hol van az egyik, vagy másik rekord. Ugyanaz az állomány marad, ha a rekordokat felcseréljük, összekeverjük.

Szekvenciális:

- szekvenciális állomány úgy keletkezik szeriálisból, hogy azonosító szerint berendezzük

- a szekvenciális állomány tehát azonosító szerint rendezett állomány
- a rekordazonosítók közötti kapcsolatot az jelenti, hogy ha van n db rekord, akkor annak a rendezettség szempontjából egyetlen sorrendje van
- az a legáltalánosabb állományszerkezet.

Szekvenciális bemeneti állományból csak olvasni lehet, kimenetibe csak írni lehet (mágnesszalagnál).

Strukturálás:	File-típus: *InputSzekvenciálisFile* (Elemtípus)
	File-típus: *OutputSzekvenciálisFile* (Elemtípus)
Értékhalmaz:	az alaphalmaz iteráltja (repezentáció-függő mennyiségű elem sokasága)
Műveletek:	**Megnyit** : output vagy input file megnyitása
	Olvas = az input file következő elemének olvasása.
	Ír = a kimeneti állomány végére új rekord írása.
	Vége? = a bemeneti állomány végén vagyunk-e.
Relációk :	nem szokásos
Ábrázolás :	pufferelési technikával

Pl.:

Típus Elem = ?
Változó F: InputSzekvenciálisFile (Elem)
e: elem
Megnyit(f)
Ciklus amíg nem vége?(f)
Olvas(f,e): **Ki**: e
Ciklus vége

(Az elem tetszőleges típus lehet, gyakran rekord)

Nyilván a megnyitáshoz még további információkra van szükség (pl. a file nevére), ezek azonban az algoritmizálás szempontjából közömbösek, vagy egyéb körülményekből kiderülnek.

A műveletek a Sor szerkezet műveleteire hasonlítanak: a bemeneti állomány olyan sor, amelyből csak kivehetünk elemet, a kimeneti állomány pedig olyan, amelybe csak betehetünk.

Direkt állomány

- a direkt állomány úgy képzelhető el, hogy rekordhelyek vannak 1-től valameddig, bármely rekord egyértelműen megmondható, hogy hol van - tehát van valamilyen rendezettség (a rekordazonosítók között az a kapcsolat, hogy ezt a sorrendet adják leképezés után)
- innentől kezdve, ha megmondtuk az azonosítót, akkor egyből hozzá tudunk nyúlni ahhoz a rekordhoz (szekvenciálisnál ezt nem tudtuk megcsinálni)

- a klasszikus direkt állománynál ez úgy néz ki, hogy a rekordhelyek száma fix - tehát amikor az állományt létrehozzuk, rögzítjük, hogy hány rekordból fog állni - sorszámozott rekordhelyek vannak, és majd oda fogjuk bepakolni a rekordokat
- későbbi rendszerek, programnyelvek direkt állomány helyett ún. relatív állományt kezelnek; ez annyi változást jelent, hogy kiegészíthetjük az állományt, de csak a végét újabb rekordhelyekkel - az eddigi rekordazonosítóknál nagyobb azonosító rekord is megjelenhet.

Random állomány

- a random annyiban különbözik a direkttől, hogy nem létezik kölcsönösen egyértelmű leképezés a rekordazonosító és egy ilyen számsorozat között - tehát nem létezik egy sorszámra való kölcsönösen egyértelmű leképezés
- csak egy egyértelmű leképezés létezik. Bármely azonosítóhoz hozzátudunk rendelni egyértelműen egy sorszámot, de más-más azonosítókhoz ugyanazt a sorszámot fogjuk hozzárendelni. Az állomány tényleges szerkezete továbbra is a sorszámon alapszik (tehát vannak rekordhelyek, melyek a sorszámon alapulnak), de több olyan rekord lesz, amit ugyanarra a sorszámra képeztük le - ezeket hívja a random túlcsorduló rekordoknak
- a random állomány szerkezet a legbonyolultabb egyszerű állományszerkezet
- az egy helyre leképezett rekordokat valahogyan le kell rendezni és azokat meg is kell találni, amikor elő akarjuk venni - ez a random állománykezelés kulcskérdése.

Minden nyelv tud kezelni szeriális, szekvenciális állományokat. Általában tudnak kezelni direktet és nagyon kevés, ami randomot is tud kezelni.

Pl. a standard PASCAL csak egy fix rekordformátumú, szekvenciális állomány kezelésére alkalmas. A TURBO ezt annyival fejeli meg, hogy van egy ún. rekordmutató, ami sorszám jellegű. Nullától kezdve vannak a rekordok (n-1)-ig - n db rekord. Nem az azonosítóból képzi a sorszámot, hanem ahogy fölvittük. Ez tulajdonképpen egy kvázi direkt állománykezelés. A felvitel sorrendjében tudunk dolgozni közvetlenül. A rekordmutató értékét tudjuk állítani 0-tól (n-1)-ig.

Összetett állományszerkezet

Összetett állományszerkezetek mindegyike azért van, hogy majd a feldolgozás nevű műveletet segítse. Tehát nem a létrehozást segíti, mert azt csak bonyolítja. Egy létező állomány rekordjaihoz hozzá akarunk nyúlni - ennek a segítésére vannak az összetett állomány szerkezetek.

Alapvetően kétféle technika létezik összetett állományszerkezet létrehozására:

1. *Láncolás*: A rekordokhoz kapcsoljuk hozzá a plusz információkat. Az állományban, a rekordok mellett jelennek meg a plusz információk - beleír az állományba.

2. *Indexelés*: Az állományon kívül jelennek meg az információk.

Láncolás: legegyszerűbb esetben azonos szerkezetű rekordok vannak, és minden rekord mellett megjelenik egy mutató mező. A mutató mező mindig azt mondja meg, hogy az adott rekordnak melyik a rákövetkezője. A mutató mező a fizikaitól lényegesen eltérő logikai sorrendet mutathat. Ezután az összes mezőt felfűzzük egy láncra, vagy létrehozhatunk részláncot, stb. (A nagyobb állományokat kezelő rendszerek ezt a módszert alkalmazzák.)

Indexelés esetén minden esetben van egy egyszerű szerkezetű alapállományunk és erre épül rá az indexszerkezet. Minden esetben az alapállomány mellé felépül egy indexállomány vagy indextábla.

Az indextáblában 2 oszlop van és legegyszerűbb esetben annyi sor, ahány rekord van az alapállományban. Az első oszlop tartalmazza az alapállomány rekordjainak az azonosítóit növekvő sorrendben. A második oszlop tartalmaz egy olyan információt, hogy az adott azonosítójú rekord hol helyezkedik el. Ez lehet egy tényleges lemezcím, vagy egy másfajta információ, amiből ki lehet számolni ténylegesen a helyet. Ezt indexnek hívjuk.

Ez a rendszer azért jó, mert ha el akarjuk érni valamelyik rekordot, nem kell "belemászni" az állományba (ami akármekkora lehet, míg az indextábla jóval kisebb). Az indextáblában rögtön láthatjuk, hogy pl. ilyen azonosítójú rekord nincs is benne. Tehát csak az indextáblát vizsgálni sokkal egyszerűbb. Egyáltalán a keresést egyszerűen végig tudjuk csinálni - használhatom a bináris keresést.

Ezt a szerkezetet egyszintű, azonosítóra épülő (elsődleges kulcsra épülő) teljes indexelésnek vagy indextáblának nevezik. Az indextáblát felépíthetjük úgy is, hogy nem minden rekordnak az azonosítóját soroljuk fel, hanem csak bizonyos rekordoknak az azonosítóját. Pl. minden ötödiket, vagy minden olyan rekordot, amely egy adott sávon több rekordot helyez el. Ez a nem teljes indextábla. Az indextábla nem más mint egy szekvenciális adatállomány, tehát az indextáblát indexelhetem - ez az ún. többszintű indexelés.

Pl. van egy lexikon. Az elsődleges indextáblában benne van az összes szóról, hogy melyik oldal, melyik oszlopának, hányadik sorában szerepel. A második szinten meg csak annyi van, hogy egy adott oldalon ez az utolsó szó.

A DBASE egyszintű indexelést tartalmaz, lehet elsődleges és másodlagos kulcsra indexelni (ez utóbbi esetben invertált állományok vannak mögötte). Relációs adatbáziskezelés mögött (pl. DBASE) invertált állományok tömege van.

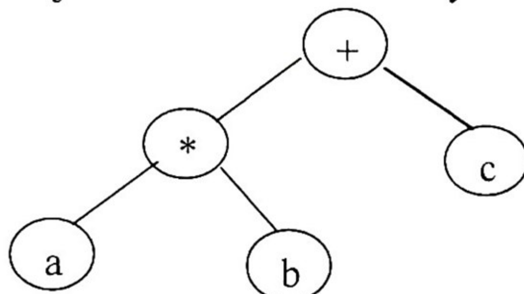
Adatbáziskezelés mögött mindig összetett adatszerkezet van, vagy az egyik, vagy a másik, vagy mind a két típus.

Bináris fák

A fákat a gráfokból tudjuk levezetni. Legyen adott egy V halmaz, amelynek elemeit nevezzük pontoknak. Legyen továbbá adott egy E halmaz, amelynek elemeit úgy kapjuk, hogy vesszük a V halmaz elemeiből képzett ketteseket. Ezeket végül is

hívhatjuk éleknek is. A gráf tehát nem más, mint pontok és az őket összekötő (vagy nem kötő) élek halmaza. Abban az esetben nevezzük ezt a gráfot bináris fának, ha minden pontból legfeljebb 2 él indul ki és bármely pontból bármely pontba csak egyféleképpen lehet eljutni.

a*b+c aritmetikai kifejezés kiszámítási szabályát bináris fával ábrázolhatjuk:



A bináris fát láncolt ábrázolással tárolhatjuk. Minden elemet (ADAT(N)) két mutatóval tárolhatunk (BAL(N)), JOBB(N), amelyek megadják az elemtől balra, illetve jobbra elhelyezkedő elem címét. A GYÖKÉR nevű változó fogja tartalmazni a gyökérelem címét. Ha egy mutató értéke 0, az jelentse azt, hogy a fa arra nem folytatódik

Bejárás:
A bináris fa bejárásakor három stratégiát alkalmazhatunk:

- Középső, baloldal, jobboldal. (Preorder bejárás)
- Baloldal, középső, jobboldal. (inorder bejárás)
- Baloldal, jobboldal, középső (postorder bejárás)

Nézzük meg a preorder bejárást! (A GYÖKÉR címmel kell használni!)

KBJ nem rekurzívan:

```
Eljárás KBJ
    Be(-1); Mut:=Gyökér
    Ciklus amíg Mut>=0
        Ciklus amíg Mut>0
            Ki: Adat(mut)
            Ha Jobb(Mut)<>0 akkor
                Be(JOBB(Mut))
            Elágazás vége
            Mut:=BAL(Mut)
        Ciklus vége
        Ki(Mut)        **veremműveletek**
    Ciklus vége
Eljárás vége
```

Feladat: Írjuk meg a többi bejárást is nem rekurzívan.
A rekurziv eljárások a könyv következő fejezetében találhatók!

II. A REKURZIÓ

Akkor mondjuk valamiről, hogy rekurzív, ha saját magát is tartalmazza, vagy önmagával van definiálva. Rekurzióval nemcsak matematikában, hanem a mindennapi életben is találkozhatunk. A rekurzió különösen hasznos eszköze a matematikai definícióknak. Bizonyára ismeretes a természetes számok, a fastruktúrák, vagy bizonyos függvények példája:

1. természetes számok:
a) 1 természetes szám,
b) egy természetes szám "rákövetkezője" természetes szám.

2. Az n! (n faktoriális) függvény (nemnegatív egészekre):
a) 0!=1
b) **Ha** n>0 **akkor** n!:=n*(n-1)!

A rekurzió ereje nyilvánvalóan az, hogy véges sok állítással végtelen sok dolog definiálására nyílik lehetőség. Ugyanígy végtelen sok számítási lépést ki lehet fejezni egyetlen, véges rekurzív programmal anélkül, hogy benne explicit (közvetlenül) ciklust szerepeltetnénk. Rekurzív algoritmusok használata elsősorban akkor indokolt, ha a megoldandó probléma, a kiszámítandó függvény, vagy a feldolgozásra váró adatok struktúrájának definíciója eleve rekurzív. Általánosan egy P rekurzív programot (P-t nem tartalmazó) S_j alaputasításoknak és magának P-nek valamilyen összetételével lehet kifejezni:

$$P \equiv P[S_j, P]$$

Programok rekurzív kifejezhetőségének szükséges és elégséges eszköze az eljárás, amely egy utasítás nevet kap, és egyben meghívható. Ha egy P eljárás explicit hivatkozást tartalmaz saját magára, akkor közvetlenül rekurzívnak mondjuk. Ha P hivatkozik egy Q eljárásra, amely (közvetlen vagy közvetett) hivatkozást tartalmaz P-re, akkor P-t közvetetten rekurzívnak nevezzük. A rekurzió használata tehát a programszöveg alapján nem mindig látszik azonnal.

Egy eljáráshoz gyakran tartoznak olyan helyi változók, konstansok, típusok és eljárások, amelyek az eljáráson belül vannak definiálva és azon kívül nincs jelentésük, nem is léteznek. Ha egy ilyen eljárást rekurzív módon használunk, minden híváskor a helyi változók egy új halmaza keletkezik!! (Ez sajnos sok memóriába fog kerülni!!) Annak ellenére, hogy e változók neve megegyezik az eljárás korábbi példányaiban szereplő megfelelő helyi változók nevével, értékükben természetesen különbözhetnek. A nevek azonosságából semmilyen konfliktus sem származik, ezt biztosítják az azonosítók hatáskörére vonatkozó szabályok. Ezek megszabják, hogy az azonosítók mindig a legújabban keletkezett változóhalmazra vonatkozzanak. Ugyanilyen

szabályok vonatkoznak az eljárás paramétereire is, amelyek definíció szerint szintén az eljáráshoz tartoznak.

A ciklusutasításokhoz hasonlóan a rekurzív eljárásoknál is előfordulhat soha véget nem érő számítás, tehát itt is foglalkozunk kell a befejeződés kérdésével. Természetesen feltehetjük, hogy a P eljárás rekurzív hívása egy F feltételtől függ, amely egy idő múlva már nem teljesül. A rekurzív függvény sémája tehát a következőképpen írható fel:

Eljárás Rekurzív
Ha F **akkor** Rekurzív **Egyébként Vége**
Eljárás vége

Addig, amíg az F feltétel teljesül, az eljárás újra és újra meghívja önmagát. Ha a feltétel már nem teljesül, az eljárás befejeződik.

A gyakorlati felhasználásokban a rekurzió mélységének (hányszor hívja meg az eljárás önmagát) nemcsak a végességét kell igazolni, hanem lényeges azt is ellenőrizni, hogy a fellépő legnagyobb mélység nem lesz-e túl nagy. Ez azért fontos, mert a P eljárás minden rekurzív hívásakor a helyi változók újabb tárhelyet foglalnak el. A helyi változókhoz járul még a számítás pillanatnyi állapotának adminisztrálása is. Erre azért van szükség, hogy az új P példány befejeződését követően folytatni lehessen a korábbi példány végrehajtását.

Amikor ugyanis a P meghívja önmagát, akkor a P eljárás még *nem* fejeződött be(!!), hiszen a P eljárásban csak egy eljárás hívása történt és még nem értük el a P eljárás végét. A rekurzió lényege ott van, hogy nem akármilyen eljárás került meghívásra, hanem P. Ezt talán úgy lehet a legkönnyebben megérteni, ha úgy képzeljük el, mintha egy teljesen más eljárás lett volna meghívva, csak a név ugyanaz. Ha ez az eljárás ismét nem fejeződik be, mivel a belsejében egy hivatkozás történik egy eljárásra, - történetesen megint egy P nevűre -, akkor immár két olyan eljárás van, amelyik nem fejeződött be. Ez a " mese" addig folytatódik, amíg P (hogy hányadik P, az megint egy fontos dolog) befejeződik. (Ez akkor történhet meg, ha F nem igaz). Természetesen a végrehajtásnak ott kell folytatódnia, ahol "megszakadt", tehát a P hívását követően. Így a függőben maradt P eljárásokat (a többes szám itt nem egészen igaz) is tovább kell folytatni.

Mikor ne használjunk rekurziót?

A megoldás nagyon egyszerű. Akkor, ha a rekurzió mélysége túl nagy, ami a program lelassulását, sőt veremtúlcsordulás-hibára történő rendellenes leállását okozhatja. Természetesen nem érdemes a problémát rekurzióval megoldani akkor sem, ha az iterációs megoldás a kézenfekvő és a rekurzió használata csak elbonyolítja az algoritmust. A rekurzió jobb megértéséhez nézzünk meg néhány példát szembesítve némelyiküket az iteratív alakjukkal.

Jól ismert a faktoriális számok példája. $Fakt_i$:=i! (pl.: ha i=4, i! = 1*2*3*4)
i = 0, 1, 2, 3, 4, 5 ...
$Fakt_i$=1, 1, 2, 6, 24, 120

Ez a formula kínálja a rekurzív algoritmus használatát. Legyen I és F változók a rekurzió i-edik szintjén az i és F_i értékei jelölésére. Ekkor a sorozat következő elemének megadásához az alábbi számítási lépések szükségesek:

i:=i+1 F:=i*F

A függvény a következő lesz:

```
Függvény Fakt(i) ** i egész **
Ha i>0 akkor
   F:=i*Fakt(i-1)
Egyébként
   F:=1
Elágazás vége.
Függvény (Visszaad:F)
```

Hívjuk meg a Fakt függvényt mondjuk 3-mal. Nézzük mi történik:

i=3, tehát az F értéket kapna, csakhogy a függvény meghív egy Fakt nevű függvényt és átad neki 2-őt. Nagyon fontos azonban azt látni, hogy itt a Fakt végrehajtása abbamaradt és F -nek nincs még értéke.

i=2. A Fakt másodszor került meghívásra és F-nek (ez már nem az előbbi F, ez a másodszor meghívásra került Fakt F változója és nem az először meghívotté, ami nagyon fontos tény!!) megint nem tud értéket adni, mert ismét egy Fakt függvényre történik hivatkozás.

i=1. Fakt immár harmadszor kerül meghívásra, de ennek az F-nek sem sikerül értéket adni, ezért felfüggesztődik a végrehajtás, mivel újra egy Fakt-ot kell hívni, de most egy nagy 0-t fog megkapni paraméterként.

i=0 Micsoda változás! Az i>0 feltétel nem teljesül, így ez a szerencsés F értéket kap, mégpedig 1-et és ez a Fakt eljárás befejezi (pálya) futását.

De hol is folytatódik az algoritmus? Ott, ahol legutóbb a végrehajtás "abbamaradt". De hiszen ez megint egy Fakt! És éppen egy értékadásnál, amit nem sikerült végrehajtani, mivel negyedszer kellett meghívni egy Fakt függvényt, ami visszaadta 1-et. Így már az értékadás könnyen elvégezhető. F:=1*1. Látni kell, hogy amikor még az értékadást nem tudta elvégezni, i értéke 1 volt, így 1-gyel kell elvégezni a szorzást.

Így sikerült befejeznie a futását a "második" Fakt-nak is, és ő is 1-et ad vissza. De hol lesz a folytatás. Persze ott, ahol legutóbb abbamaradt. Ezt pedig az a Fakt, amelyik másodszor került meghívásra és szintén nem sikerült F-nek értéket adni, mivel egy Fakt-ot kellett meghívni. Most, hogy ez a Fakt 1-et adott vissza, így az értékadás elvégezhető. F:=2*1, ugyanis itt i értéke 2 volt, így ez a Fakt 2-t ad vissza.

(Visszafelé haladok!) Ez a Fakt is befejezte tehát működését, így folytatódhat a végrehajtás pontosan ott, ahol még nem történt meg az értékadás. Ebben a Fakt-ban az i értéke 3, azaz F:=3*2. Ez a Fakt tehát 6-ot ad vissza, ami éppen 3! és nincs több futását "felfüggesztett" Fakt!

Teljesen világos, hogy ebben az esetben a rekurzió egyszerű iterációval helyettesíthető:

i:=0 ; F:=1
Ciklus amíg i<n
 i:=i+1; F:=i*F
Ciklus vége

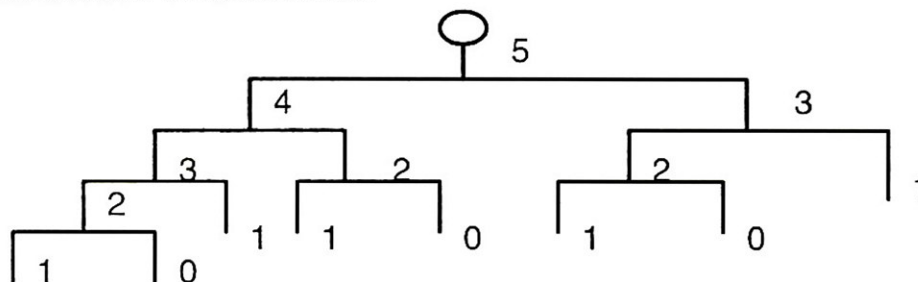
Itt az algoritmus n!-t számol.

A Fibonacci-számokat a következőképpen szokták definiálni:
$F_0=1$, $F_1=1$ majd $F_i=F_{i-1}+F_{i-2}$ (i>1 és egész)
A első 7 Fibonacci-szám tehát: 1,1,2,3,5,8,13 (minden F. szám - az első kettőt kivéve - az őt közvetlenül megelőző két szám összege.)

Határozzuk meg az i-edik Fibonacci számot!

Függvény Fib(i)
Ha i=0 **akkor** F:=0
Egyébként
 Ha i= 1 **akkor**
  F:=1
 Egyébként
  F:=Fib(i-1)+Fib(i-2)
 Feltétel vége
Feltétel vége
Függvény vége (visszaad: F)

A Fib kiszámítása a Fib(i) eljárással e függvény rekurzív hívását jelenti. De milyen gyakorisággal? Minden i>1 paraméterrel történő további hívás további két hívást jelent, tehát a hívások száma exponenciálisan növekszik. (lásd az ábrát!) Az ilyen program természetesen célszerűtlen.



Biztosan nem kerülhető el a memória elfogyasztása és szinte biztos, hogy elég nagy i-nél a program futási hibával leáll. Nézzük, hogyan is működik egy apró módosítás után:

```
Függvény Fib(i)
Ha i= 0 akkor
    F:=0                ***
Egyébként
    Ha i = 1 akkor      **
        F:= 1
    Egyébként
        F1 := Fib(i- 1)  *
        F2 := Fib(i- 2)  ****
        F:= F1 + F2      *****
    Feltétel vége
Feltétel vége
Függvény vége (Visszaad F)
```

Nézzük meg, miben tér az el a korábban megadott algoritmustól:

Legyen i kezdeti értéke 5 !

i=5. F1 értéke ismeretlen, mert Fib meghívásra kerül 4-gyel (*-nál)

i=4. F1 értéke ismeretlen, mert Fib meghívásra kerül 3 -mal (*-nál)

i=3. F1 értéke ismeretlen, mert Fib meghívásra kerül 2-vel (*-nál)

i=2. F1 értéke ismeretlen, mert Fib meghívásra kerül 1-gyel (*-nál)

i=1. F értéke 1 lesz. (**-nál) és az eljárás befejeződik, visszaadja F-et, azaz 1-et, és ott folytatódik a végrehajtás, ahol az utolsó hívás megtörtént, azaz *-nál és i értéke 2 (!)

i=2. F1 értéke ekkor már nem ismeretlen, hiszen a Fib(i-1) értéke 1. F2 nem kap értéket, hiszen hivatkozás történik Fib-re. Fib tehát meghívásra kerül 0-val, így F elnyeri jutalmát és 0-t kap(***-nál), de ezzel az eljárás befejezi a működését (itt volt i értéke 0) és F-et, azaz 0-t ad vissza. Így F2 értéke 0 lesz (****-nál). F értéke viszont rögtön utána F1+F2, azaz 1. Az a Fib tehát, ahol i értéke 2, befejezi a működést és visszaadja F értékét, azaz 1-et annak a Fib-nek, ahol i=3 és *-nál abbamaradt a futása.

i=3. F1 most felveheti a Fib(i-1)-et, azaz 1-et, de F2 nem ilyen szerencsés, hiszen nem kap értéket ****-nál, mivel Fib(i-2)-nek még nincs értéke. A dolog simán megy, hiszen a meghívott Fib-ben F értéke **-nál 1 lesz, így 1-et fog visszaadni.

Ezért F2 most már felveheti az 1-et, így F értéke (*****-nál) 2. Ez a Fib is befejezi működését és visszaadja a 2-t annak a Fibnek, amelyik meghívta, azaz annak, ahol i=4

i=4. F1 most felveheti a Fib(i-1)-et, azaz 2-t, de F2 nem ilyen szerencsés, hiszen nem kap értéket ****-nál, mivel Fib(i-2)-nek még nincs értéke. A dolog nem megy simán, hiszen a meghívott Fib-ben i értéke 2 lesz (4-2).

i=2. Ekkor *-nál Fib meghívásra kerül 1-gyel, F értéke 1 lesz, amit vissza is ad a függvény, így *-nál F1 értéke 1 lesz. F2 sem kap simán értéket, hanem először meg kell hívnia egy Fib-et 0-val. Ebben a Fib-ben - mivel i=0 - F értéke 0 lesz így ez a Fib 0-t ad vissz..

Ezért F2 most már felveszi ezt a 0-t, így F értéke 1 lesz (*****-nál) Ez a Fib is befejezi működését és visszaadja a 1-et annak a Fibnek, amelyik meghívta, azaz annak, ahol i=4.

i=4. F2 most felveheti jól megérdemelt jutalmát, azaz 1-est kap a Fib(i-2) által visszadottat. Így F értéke 3 lesz (2+1). Ez a Fib tehát 3-at ad vissza, annak a Fib-nek, ahol i=5.

i=5. F1 értéke tehát 3 lesz, de F2 még mindig nem kap értéket, hiszen Fib-et hív 3-mal.

i=3. Itt F1 nem kap értéket, hiszen Fib-et hív 2-vel

i=2. Ebben a Fib-ben F1 értéke 1 lesz, de csak azután, miután a meghívott Fib (1-gyel) visszaad 1-et. F2 értéke úgy lesz 0, hogy a 0-val hivott Fib ezt adja vissza. Így tehát F értéke 1 lesz (1+0) és visszatér a végrehajtás oda, ahol i=3

i=3. F1 most már felveszi a kapott 1-et, és F2 is felveszi az 1-et, miután az 1-gyel meghívott Fib visszaadja ezt az 1-et. F=2 tehát az a Fib, ahol i=5, ott folytatódik a végrehajtás.

i=5. F2 most már felveszi a kapott 2-t, így végre befejezheti az olvasó a Fib-ek olvasását, hiszen F értéke 5 lesz (3+2) és valamennyi Fib-et sikerült kivégezni.

Természetesen a Fibonacci-számokat iteratív sémával is kiszámíthatjuk. Az új $x=Fib_{i+1}$ és $y=Fib_i$ segédváltozók bevezetésével elkerülhető ugyanazon értékek többszöri kiszámítása:

Eljárás Fib
i:=1; x:=1; y:=0
Ciklus amíg i< n
    z:=x ; i:=i+1
    x:=x+y; y=z
Ciklus vége[4]

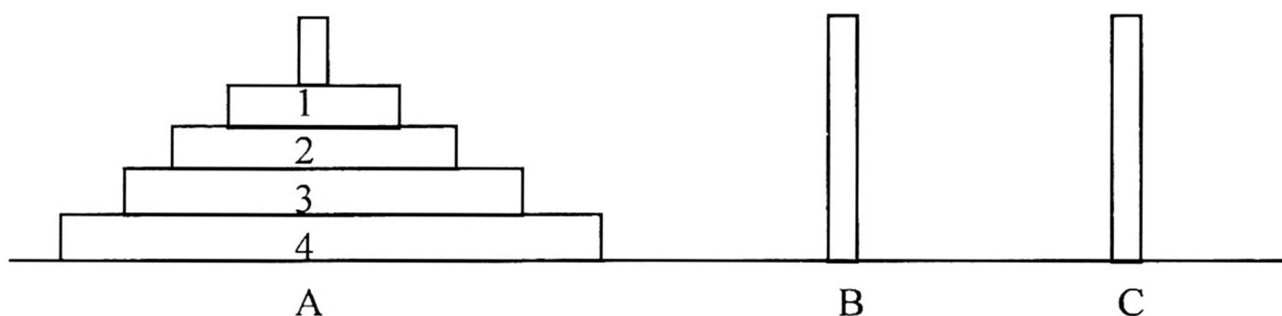
Aki megismerte a rekurziót, könnyen abba a hibába eshet, hogy kényelmessége és eleganciája miatt **akkor** is használja, amikor nincs is rá szüksége, mert a feladat egyszerűbben és hatékonyabban oldható meg ciklusok használatával. A rekurziót mindig helyettesíthetjük ciklusokkal.

Nézzünk még egy példát a rekurzióra!

Adott három rúd: A,B,C. Az A rúdon induláskor N darab lyukas korong van, nagyság szerint felfelé csökkenő sorrendben, a korongok mind különböző átmérőjűek.

Át kell rakni a korongokat a C rúdra a következő szabályok szerint:

- a B rudat kisegítő, közbenső tárolásra használhatjuk;
- minden lépésben egy korongot mozgathatunk;
- minden korong csak nála nagyobb korongon lehet.



N bemenő adat.

Olyan megoldást akarunk, amely lépésről lépésre megadja, melyik korongot honnan hová kell tenni.

A feladat onnan kapta a nevét, hogy állítólag Hanoi közelében egy kolostorban egy 64 aranykorongból álló tornyot raknak át az ott élő szerzetesek az előbbi szabályok szerint, minden nap egyetlen korongot mozgatva. A legenda szerint a világvége

[4] Az x,y,z változókra vonatkozó értékadás kifejezhető két értékadással is, szükségtelenné téve a z segédváltozót: x:=x+y: y:=x-y. Az eredmény az x-ben keletkezik.

akkor fog bekövetkezni, ha befejeződik a torony átrakása. (Nem ebben az évezredben lesz!)

A megoldáshoz az adja az ötletet, hogy 2-es korongot úgy kell átrakni, hogy az 1-es korongot áttesszük B-re, ekkor a 2-es korong áttehető C-re majd a B-re félretett korong áttehető C-re.

Ha n>0 **akkor**
    n-1 magas torony átrakása A-ról B-re
    az n-dik korong átrakása A-ról C-re
    n -1 magas torony átrakása B- ről C- re
Egyébként semmit sem kell csinálni
Elágazás vége

A feladatot eggyel alacsonyabb torony kétszeri átrakására és egy korong mozgatására vezettük vissza. A legegyszerűbb az az eset, amikor 0 magasságú tornyot kell mozgatni, ekkor ugyanis semmit sem kell csinálni.

Eljárás Hanoi(n,honnan,mivel,hova) ***n egész, a többi karakteres***
Ha n>0 **akkor**
    hanoi(n-1,honnan,hova,mivel)
    Ki: n " korongot tedd", honnan, "rúdról", hova, "rúdra"
    hanoi(n-1,mivel,honnan,hova)
Elágazás vége
Eljárás vége

Az algoritmus elemzése hasonlít a teljes indukcióra. Egy magasság esetén jól működik, hiszen kétszer hívja meg 0 magasságú toronyra, amiről tudjuk, hogy helyes. Ha n-1-et helyesen, akkor n-et is helyesen kezeli.

A tanulság tehát: kerüljük a rekurzió használatát, ha nyilvánvaló az iterációval való megoldás. Ez azonban semmiképpen sem jelentheti a rekurzió elvetését. Igen sok alkalmazása van, amint azt az ezt követő fejezetek is igazolni fogják. Maga az a tény, hogy a rekurzív eljárások végül is nem rekurzív gépeken kerülnek megvalósításra, azt bizonyítja, hogy gyakorlatilag minden rekurzív program átalakítható tisztán iteratív programmá. Ez azonban maga után vonja egy rekurziós veremtár (stack) explicit kezelését, amelynek műveletei oly mértékben elhomályosíthatják a program lényegét, hogy nehezen követhetővé válik. Azokat az algoritmusokat tehát, amelyek természetüknél fogva inkább rekurzívak, mint iteratívak, rekurzív eljárások formájában kell megfogalmazni, figyelembe véve azt a tényt, hogy a rekurzió mélysége csak akkora lehet, amelyet elvisel a számítógép stack-je.

GYORSRENDEZÉS (QUICKSORT)

Az alapváltozatot, amely csere elven alapszik, 1962-ben C.A.R. Hoare hozta nyilvánosságra.

A módszer lényege:

Először ki kell keresnünk egy ún. *pivot* elemet és a tömböt két résztömbre kell osztanunk. Az eljárás a következő: balról indulva addig vizsgálja a tömböt, amíg a pivot elemnél nagyobb vagy egyenlő elemet nem talál, majd jobbról indulva keres a tömb elemei között addig, amíg a pivot elemnél kisebb vagy egyenlő elemet nem talál. Ekkor a két elemet felcseréli (ha azok nem azonos féltömbben voltak) és újrakezdi a vizsgálatot. A tömb vizsgálata és felosztása abbamarad, ha a kétoldali vizsgálat során valahol találkozik. Eddig a tömböt két félre osztotta, ahol a baloldali elemek kisebbek/egyenlőek, a jobb oldaliak pedig nagyobbak/egyenlőek, mint a pivot elem. Ezután az eljárás rekurzív módon a résztömbökhöz fordul addig, amíg a résztömbök már csak egy elemet tartalmaznak, amely nyilván rendezettnek tekinthető.

Nézzük végig egy példán az algoritmus működését (10 elemű a vektor, ezért quicksort (1,10) módon hívjuk meg).

```
Eljárás quicksort(alsó, felső)
 i:=alsó;j:=felső; pivot:=a[(alsó+felső) div 2]  **div legyen egész osztás**
Ciklus
   Ciklus amíg a[i]<pivot
      i:=i+1
   Ciklus vége
   Ciklus amíg pivot<a[j]
      j:=j-1
   Ciklus vége
   Ha  i<=j  akkor          ** a megtalált elemek cseréje **
      y:=a[i]; a[i]:=a[j]; a[j]:=y
      i:=i+1; j:=j-1
   Elágazás vége
amíg  i>j
Ha  alsó<j akkor
   Quicksort (alsó,j)
Elágazás vége
Ha i<felső akkor
   Quicksor(i,felső)
Elágazás vége
Eljárás vége      (Hívása quicksort(alsó_index, felső_index)
```

Példa:
Kezdő sorrend: 7 16 1 9 12 2 7 14 0 4

1. felosztás: i=1, j=10
pivot 12 ← pivot elem

7 16 1 9 **12** 2 7 14 0 4

A balról megtalált elem 16, jobbról 4 ezért megcseréli őket:

7 4 1 9 **12** 2 7 14 0 16

i=5, j=9
A balról megtalált elem 12 (éppen a pivot elem), jobbról 0, csere:

7 4 1 9 0 2 7 14 **12** 16

2. felosztás : i=1, j=7
pivot 9

7 4 1 **9** 0 2 7 14 12 16

A balról megtalált elem 9, jobbról 7, ezért megcseréli őket:

7 4 1 7 0 2 **9** 14 12 16

3. felosztás: i=1, j=6
pivot 1

7 4 **1** 7 0 2 9 14 12 16

A balról megtalált elem 7, jobbról 0 ⇒megcseréli őket:

0 4 **1** 7 7 2 9 14 12 16

i=2, j=4
A balról megtalált elem 4, jobbról 1⇒megcseréli őket:

0 1 **4** 7 7 2 9 14 12 16

4. felosztás: i=1, j=2
pivot 0

0 1 4 7 7 2 9 14 12 16

Nincs változás
5. felosztás i=3, j=6
pivot 7

0 1 4 **7** 7 2 9 14 12 16

A balról megtalált elem 7, jobbról 2⇒megcseréli őket

0 1 4 2 7 **7** 9 14 12 16

6. felosztás: i=3, j=4
pivot 4

0 1 **4** 2 7 7 9 14 12 16

A balról megtalált elem 4, jobbról 2⇒megcseréli őket

0 1 2 **4** 7 7 9 14 12 16

7. felosztás i=8, j=10
pivot 12

0 1 2 4 7 7 9 14 **12** 16

A balról megtalált elem 14, jobbról 12⇒megcseréli őket

0 1 2 4 7 7 9 **12** 14 16

8. felosztás i=9, j=10
pivot 14

0 1 2 4 7 7 9 12 **14** 16

Nincs változás

Megfigyelhető, hogy a pivot elem megválasztása a futási idő tekintetében jelentős szerepet játszik. Különösen kerülni kell a feleslegesen sok rekurzív hívást. Ez akkor adódik, ha mindig a legkisebb elemet választjuk pivot elemként. A pivot elem választása akkor a legkedvezőbb, ha ha a tömböt két közel egyforma résztömbre osztja fel.

Hatékonysága: n elem esetén a rekurzió miatt 2*n méretű vermet igényel. Az összehasonlítások száma n*(n-1)/2-1, véletlenszerű előrendezettség esetén csak n*lg n. A mozgatások száma monoton növekvő előrendezettség esetén 2*(n-1), monoton csökkenő esetben 2*n +(n-4)/2. Véletlenszerű előrendezettség mellett pedig az összehasonlítások számának körülbelül a fele.

Eddig csak az úgynevezett *közvetlen* rekurzióról volt szó. Ismert azonban a közvetett rekurzió fogalma is. Azaz amikor az f1 eljárás belsejében egy hivatkozás van az f2 eljárásra, az f2-ben pedig egy hivatkozás f1-re. Nézzük meg egy példán, hogyan is néz ez ki a gyakorlatban:

Vege: egy eljárás, amely a kapott szöveg első karakterét levágja.
Eleje: egy eljárás, amely a kapott szöveg utolsó karakterét vágja le.
Elso: a kapott szöveg első karakterét visszaadó függvény.
Utolso: a kapott szöveg utolsó karakterét levágó függvény.
Hossz: a szöveg hosszát visszaadó függvény

Eljárás Elj1(S) ***S szöveg típusú***
Ha Hossz(S)>0 **akkor**
  Ki: Utolsó(S)
  Eleje(S)
  Elj2(S) ***az Elj2 hívása. Ezért az Elj1 még nem fejeződik be ***
Elágazás vége
Eljárás vége

Eljárás Elj2(S) *** S szöveg típusú ***
Ha Hossz(S)>0 **akkor**
  Ki: Elso(S)
  Vege(S)
  Elj1(S) *** az Elj1 hívása, az Elj2 nem fejeződik be ***
Elágazás vége
Eljárás vége

Ha meghívjuk az Elj2-t és S értéke ÁKOS, akkor:

1) kiíródik az Á karakter, majd a Vege(s] miatt S értéke KOS lesz és az Elj1 meghívásra kerül
2) S értéke KOS, ezért kiíródik az S karakter, majd az Eleje(S) miatt S-ben KO lesz és Elj2 lesz kerül meghívásra
3) kiíródik K, S-ben O lesz és meghívja Elj1-et.
4) kiíródik O, S értéke üres és meghívásra kerül Elj2
5) S hossza 0, hiszen üres így az eljárás befejeződik. Ezek után megtörténik az eljárások befejezése a meghívás ellenkező sorrendjében.

Vagyis sikerült egy szöveget kiíratni elejéről és a végéről felváltva, befelé adva a karaktereket.

III. A VISSZALÉPÉSES KERESÉS TÉTELE (BACKTRACK)

Ezzel a tétellel a problémamegoldás igen széles területén alkalmazható algoritmus kerül ismertetésre, melynek lényege a feladat megoldásának megközelítése *rendszeres* próbálgatásokkal. Sok esetben ennél jobb módszert nem is követhetünk.

Általános feladat

Adott n darab sorozat, melynek rendre M[1]) M[2], ... elemszámúak. (az M vektor első eleme adja az első sorozat elemszámát, ...) Ki kell választani mindegyikből egy-egy elemet úgy, hogy az egyes sorozatokból való választások más választásokat befolyásolnak (pl. nem lehet két sorozatból azonos számú elemet választani). Tulajdonképpen ez egy bonyolult keresési feladat: egy adott tulajdonsággal rendelkező szám N-est kell keresni. A megoldást úgy készítjük el, hogy ne kelljen az összes lehetőséget végignézni.

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

A megoldás - mint a korábbi tételeknél láttuk - egyszerűbbé válik, ha tudjuk, hogy megfelelő tulajdonságú az elemsorozat. Most azonban kövessük az általános esetmegoldást!

Algoritmus:

```
Eljárás
 i:=1; x[1]:=0
Ciklus amíg i>=1 És i<= N
   Ha van_jó_eset(i) akkor
      i:=i+1;x[i]:=0 **vigyázni kell az index túllépésre, pl. x[] n+1 elemű**
   Egyébként
      i:=i-1
   Elágazás vége
Ciklus vége
Van:=i>n
Eljárás vége **megoldás az x[]-ben keletkezik**
```

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

FüggvényEljárás Van_Jó_Eset

Ciklus

x[i]:=x[i]+1

amíg x[i]>M[i] **Vagy Nem** Rossz_eset(i,x[i])

Ciklus vége

Eljárás vége (visszaad: x[i]<=M[i])

Rossz választásnak nevezzük azt, amelyet a korábbi választások közül valamelyik megakadályoz.

FüggvényEljárás Van_Rossz_Eset(i,X[i])

j=1

Ciklus amíg j<i **És** (j,X[j]) nem zárja ki (i,X[i]) - t

j:=j+1

Ciklus vége

Eljárás vége (visszaad: j<i)

Feladat:

Készítsünk algoritmust, amely elhelyez egy sakktáblán nyolc vezért úgy, hogy egyik sem üti a másikat!

Mivel 64 mezőn a 8 vezért 4 426 163 368-féleképpen helyezhetünk el, igen sok vizsgálatra lenne szükség. (Különösen akkor, ha a feladatnak nincs megoldása!)

Ennél jobb matematikai modell a következő. A 8x8-as sakktábla mezőit a szokásostól eltérően egy-egy számpárral adhatjuk meg: A két szélső mező a8 = (0,0) és h1=(7,7). Mivel a vezér az elfoglalt mező egész sorát üti, tehát a nyolc vezér mindegyikét másik sorba kell helyezni. Eszerint egy ilyen elhelyezést egy 8-as számrendszerben felírt 8-jegyű számmal adhatunk meg. Például a 04752613 számnyolcas az ábrán látható állást reprezentálja.

	0	1	2	3	4	5	6	7
0	O							
1					O			
2								O
3						O		
4			O					
5							O	
6		O						
7				O				

A lineáris keresési algoritmust alkalmazva tehát végigmegyünk a [00000000 .. 77777777] intervallumon, és ellenőriznünk kell, hogy az állás megfelelő-e. A keresési intervallum hossza 8^8 =16777216, ami lényegesen, mintegy 260-szor kisebb, mint az összes elhelyezések száma. Ha nem tudnánk, hogy e feladatnak van megoldása, akkor merészség lenne egy ilyen programot elindítani, hiszen egyetlen feltételvizsgálat ebben az esetben nagyon összetett: 8*7/2=28 pár kölcsönös helyzetét kell vizsgálni.

Az is nyilvánvaló, hogy még így is vannak olyan állások, amelyeket ki kell hagyni a vizsgálatból. Ilyenek például az egyenlő számjegyeket tartalmazó helyzetek, elég tehát a különböző oszlopokon elhelyezhető vezérek 8!=40320 számú különböző állását vizsgálni. Hasonló okoskodással csökkenthetnénk tovább az esetek számát, de ez nem minden feladatban végezhető el triviálisan és nem biztos, hogy elegendően beszűkül az átfésülendő tartomány. Ezért célszerűbb a következő módszert alkalmazni: hogy ne kelljen a teljes állást vizsgálni, redukáljuk a feladatot k<=8 vezér elhelyezésére. Ha k számú vezért nem tudunk úgy elhelyezni, hogy egymást ne üssék, akkor a következő k+1-edik elhelyezésével már nem is érdemes kísérletezni. De ha az első k vezér állása megfelelő, akkor kereshetünk helyet a következőnek.

Nyilvánvaló, hogy ilyenkor is kerülhetünk abba a helyzetbe, hogy újabb vezér már nem helyezhető el a táblán. Ekkor az első k vezér elhelyezése nem megfelelő a folytatáshoz. Felfüggesztjük tehát a k+1-edik elhelyezésére vonatkozó próbálgatást és visszalépünk egy szinttel: keresünk egy újabb elrendezést az első k vezér számára. Rá kell mutatni az ötlet még egy előnyére, nevezetesen amikor egy újabb vezérnek keresünk helyet, akkor elegendő ennek a többihez való viszonyát vizsgálni, hiszen amazok egymás közötti elhelyezése már megfelelő.

```
Eljárás Nyolc_Vezér
Ciklus i:=0-tól 7-ig
    v[i]:=-1            ** kezdő helyzet**
Ciklus vége
k:=0
Ciklus amíg  k>=0 És  k<=7
    v[k]:=v[k]+1
    Ha v[k]>7 akkor
        v[k]:=-1
        k:=k-1
    Egyébként
        Ha jo_az_állás akkor k:=k+1
    Elágazás vége
Ciklus vége
Ki: V[]                 ** a vektor elemeinek kiíratása **
Eljárás vége
```

```
Függvény jo_az_állás
i:=k-1
joe:=Igaz
Ciklus amíg i>=0 És joe                                  Abszolútérték függvény
   joe:=v[i]<>v[k] És ABS(v[i]-v[k]))<> ABS(i-k)   **ne legyenek azonos
                                                      sorban ill. oszlopban
   és                                                 átlóban**
   i:=i-1
Ciklus vége
Függvény vége (Visszaad:joe)
```

Ha a feladatot az alapfeladatot szigorúan követve próbáljuk megoldani, akkor az előzőhöz hasonló megoldást kaphatunk.

Megfeleltetés:

```
N                                  - 8
Sorozatok                          - az egyes vezérek elhelyezései
M[1]=M[2]=...N
 (j,X[j]) nem zárja ki (i,X[i])    - t : X[i] <>X[j] És ABS(X[i]-X[j])<>i-j (azaz a
 j vezér nincs egy sorban vagy átlóban az i vezérrel)
```

Algoritmus:

```
Eljárás Nyolc_Vezér_Elhelyezése
i:=1
Ciklus amíg i>=1 És i<=N
   Ha van jó eset(i) akkor
      i:=i+1: X[i]:=0
   Egyébként
      i:=i-1
   Elágazás vége
Ciklus vége
Van:=i>N
Eljárás vége

FüggvényEljárás Van_Jó_Eset(i)
Ciklus
   X[i]:=X[i]+1
amíg X[i]>N Vagy  Nem Rossz_Eset(i,X[i])
Eljárás vége (Visszaad: X[i]<=N)

Függvény Rossz_Eset(i,X[i])
j:=1
Ciklus amíg j<i És X[i]≠X[j] És ABS(X[i]-X[j])≠i-j
   j:=j+1
```

```
Ciklus vége
Függvény vége (Visszaad: j<i)
```

A probléma megoldható rekurzív algoritmus segítségével is.

```
Eljárás próbál (i)            **i egész**
az i-edik vezér elhelyezésének előkészítése
Ciklus
    következő jó választás
    Ha jó akkor
        a vezér elhelyezése
        Ha i<8 akkor
            próbál(i+1)       **itt a rekurzió**
            Ha sikertelen akkor vezért levenni
        Elágazás vége
    Elágazás vége
amíg sikeres Vagy nincs több hely
```

Ez egy elég általános algoritmus, a továbblépéshez az adatok ábrázolását kell meghatározni. Legyen

X[i] a vezér pozíciója az i-edik oszlopban
A[j] Igaz, ha nincs vezér a j-edik sorban
B[k] Igaz, ha nincs vezér a k-adik ⁄ átlón
C[k] Igaz, ha nincs vezér a k-adik \ átlón

Az egyik átló esetében az i+j koordináta összege állandó, a másik átlóra pedig i- állandó.

A vezér elhelyezése a következőt jelenti:
X[i]:=j: A[j]:= **Hamis**: B[i+j]:=**Hamis**: C[i-j]:=**Hamis**

A vezért levenni parancs finomítása:
A[j]:= **Igaz**: B[i+j]:= **Igaz**: C[i-j]:=**Igaz**

A jó elágazás akkor teljesül, ha a kiszemelt i,j mező szabad vonalakra (azaz igazra állított sorra és átlókra) esik. Ezt a következő logikai kifejezés ábrázolja:

A[j] És B[i+j] És C[i-j]

Az algoritmus kialakítását ezzel befejeztük.

```
Program Vezérek  ** csak egy megoldást keres**
Ciklus i:=1-től 8-ig
    a[i]:=Igaz
Ciklus vége
```

```
Ciklus i:=2-től 16-ig
    b[i]:=Igaz
Ciklus vége
Ciklus i:=-7-től 7-ig
    c[i]:=Igaz
Ciklus vége
Próbál(1,q)                    **q logikai**
Ha q=Igaz akkor
    Ciklus i:=1-től 8-ig
    i: X[i]
    Ciklus vége
Elágazás vége
Vége (Vezérek)

Eljárás próbál (i,q)                ** i egész, q logikai és q-nál cím szerinti a
                                       paraméterátadás**
j:=0
Ciklus
    j:=j+1;q:=Hamis
    Ha a[j] És b[i+j] És c[i-j] akkor                ** szabad a mező **
        x[i]:=j
        a[j]:=Hamis;b[i+j]:=Hamis;c[i-j]:=Hamis **elhelyezése a j. vezérnek**
        Ha i<8 akkor                                 ** ha van még**
            próbál(i+1,q)
            Ha q=Hamis akkor                         **ha sikertelen volt**
                a[j]:=Igaz; b[i+j]:=Igaz; c[i-j]:=Igaz  ** levesszük a j. vezért**
            Elágazás vége
        Egyébként
            q:=Igaz
        Elágazás vége
    Elágazás vége
amíg q Vagy j=8
Eljárás vége (próbál)
```

A feladatot módosítsuk úgy, hogy az összes lehetséges megoldást irassuk ki. Ehhez először a Próbál eljárást kell finomítani.

```
Eljárás Próbál(i)              ** i egész**
Ciklus k:=1-től m-ig
    a k-adik jelölt kiválasztása
    Ha megfelelő akkor
        feljegyezzük
        Ha i<n akkor
```

```
            próbál(i+1)
        Egyébként
            Ki: megoldás
        Elágazás vége
        a feljegyzés törlése
    Elágazás vége
Ciklus vége
Eljárás vége
```

A teljes algoritmus:

```
Program Vezér    ** az összes megoldás**
Ciklus i:=1-től 8-ig
    a[i]:=Igaz
Ciklus vége
Ciklus i:=2-től 16-ig
    b[i]:=Igaz
Ciklus vége
Ciklus i:=-7-től 7-ig
    c[i]:=Igaz
Ciklus vége
Próbál(1)
Program vége

Eljárás Póbál(i)            **egész**
Ciklus j:=1-től 8-ig
    Ha a[j] És b[i+j] És c[i-j] akkor
        x[i]:=j
        a[j]:=Hamis; b[i+j]:=Hamis; c[i-j]:=Hamis
        Ha i<8 akkor
            próbál(i+1)
        Egyébként
            Ki: X           ** a megoldást tartalmazó X kiírása**
        Elágazás vége
        a[j]:=Igaz; b[i+j]:=Igaz; [i-j]:=Igaz
    Elágazás vége
Ciklus vége
Eljárás vége
```

Valójában összesen 12 lényegesen különböző eset van, az algoritmus azonban nem érzékeli a szimmetriát.

Maradjunk továbbra is a sakktáblánál és nézzünk meg egy olyan visszalépéses algoritmust, amely a rekurziót használja.

Feladat

A huszár útja a táblán. Vegyünk egy n^2 mezőre osztott n*n-es táblát. Egy huszárt, amely a sakk szabályai szerint mozog - helyezzünk a tábla valamelyik (x0, y0) koordinátájú mezőjére. A feladat egy olyan út kijelölése, amely lefedi a táblát, más szóval egy olyan n^2-1 lépésből álló útvonalat kell kialakítani, amelyen végighaladva a huszár pontosan egyszer érinti a tábla minden mezőjét.

Az n^2 mező lefedésének problémáját kézenfekvő a következő kérdésre egyszerűsíteni: hogyan hajtson végre a huszár egy következő lépést, ill. hogyan dönthető el, hogy nincs további lépése. Definiálni kell tehát egy olyan algoritmust, amely egy soron következő lépést keresi:

Eljárás próbál
Készítsük elő a jelöltek kiválasztását
Ciklus
    válasszuk a következőt
    Ha megfelelő **akkor**
        feljegyezzük
        Ha a megoldás nem teljes **akkor**
            próbáljunk egy következő lépést
            Ha sikertelen **akkor**
                töröljük a bejegyzést
            Elágazás vége
        Elágazás vége
    Elágazás vége
amíg a lépés sikeres **Vagy** nincs több jelölt
Eljárás vége

Ha pontosabban kívánjuk leírni az algoritmust, döntenünk kell az adatok ábrázolásáról. Nyilvánvaló, hogy a táblát egy mátrixszal fogjuk ábrázolni, neve legyen h.

h[x,y] = 0 : az (x,y) mező szabad,
h[x,y] = i : az (x,y) mezőre esik az i-edik lépés ($1 <= i <= n^2$)

Dönteni kell a megfelelő paraméterek megválasztásáról is. Ezek adják majd meg a következő lépés kiindulási feltételeit, ill. jegyzik fel a lépés sikeres voltát. Az első feladatra elég a kiinduló mező (x,y) koordinátáinak és (adminisztrációs célból) a már megtett i lépések számának a megadása. A második feladatra egy q logikai változót használhatunk. q = Igaz, ha a lépés sikeres, q = Hamis, ha a lépés sikertelen.

Egy (x,y) kiindulási koordinátapárhoz elvileg nyolc lehetséges (u,v) koordinátájú lépés tartozik (H a huszár).

	3		2	
4				1
		H		
5				8
	6		7	

Az x,y értékeiből az u,v értékeket egyszerűen a megfelelő koordinátakülönbségek hozzáadásával kaphatjuk meg. Ezeket az értékeket a különbségpárok tömbjében vagy a két koordinátához tartozó különbségek egy-egy tömbjében tárolhatjuk. Az s halmazban a lehetséges indexhatárok szerepeljenek, azaz 1,2,.., n-ig az egész számok.

```
Eljárás próbál (i,x,y,q)     ** i,x,y egészek és q logikai és q-nál cím szerinti a
k:=0                            paraméterátadás **
Ciklus
  k:=k+1: q1:= Hamis
  u:= x + a[k] : v:= y + b[k]       ** a következő lépés **
  Ha u benne van s-ben És v benne van s-ben akkor
    Ha h[u,v] = 0 akkor
      h[u,v]:= i
      Ha i< nnégyzet akkor      ** ha még van hely a táblán**
        próbál (i+1,u,v,q1)
        Ha q1=Hamis akkor h[u,v]:=0
      Egyébként
        q1:=Igaz
      Elágazás vége
    Elágazás vége
  Elágazás vége
amíg q1 Vagy k=8
q:=q1
Eljárás vége

Program Huszár
n:=8; négyzet:=64             ** így nem kell mindig a programban kiszámol-
ni**
s:= [1,2,3,4,5,6,7,8]         ** az s halmaz elemei**
a,b vektorok := lehetséges lépésektől való eltérés
h[1,1]:= 1                    ** az első pozíció **
próbál(2,1,1,q)
Ha q akkor Ki: h mátrix Egyébként Ki: "nincs megoldás"
Program vége
```

Ezek után a visszalépéses algoritmusok általános formáját is megadhatjuk rekurzív módon:

Eljárás *próbál*
k:=0
Ciklus
    k:=k+1
    Ha *megfelel* ***akkor***
        feljegyezzük
        Ha i<n **akkor**
            próbál(i+1)
            Ha sikertelen **akkor** törljük a bejegyzést
        Különben
            Sikeres:=**Igaz**
        Elágazás vége
    Elágazás vége
amíg sikeres **Vagy** k=m
Eljárás vége

Térjünk vissza a fejezet elején alkalmazott visszalépéses alapalgoritmushoz és ennek felhasználásával oldjuk meg a következő feladatot:

Feladat:

Lefedhető-e egy adott szakasz egyszeresen $h_1,h_2,...,h_n$ hosszúságú kisebb szakaszokkal?

a) Keressük meg az összes megoldást,

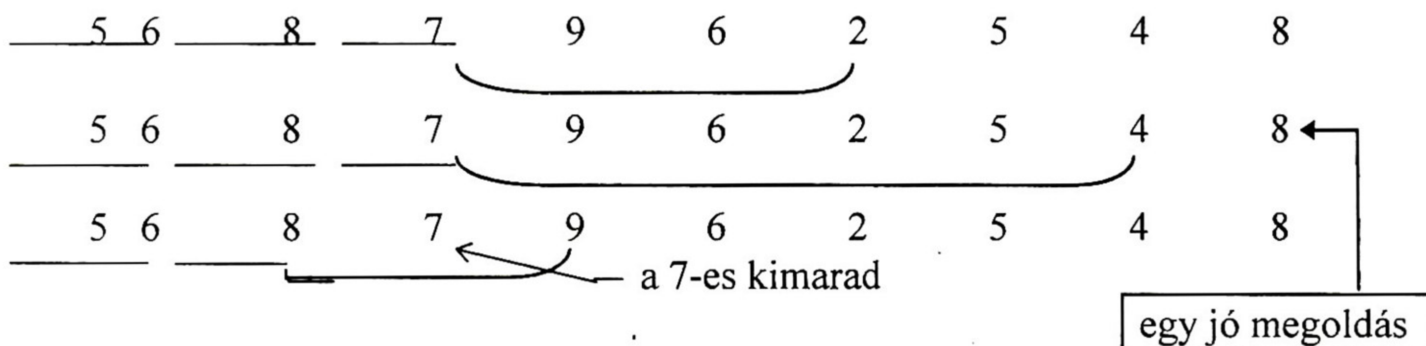
b) majd az összes, legkevesebb szakasz felhasználásával elérhető megoldást.

A feladat annyiban tér el az alapfeladattól, hogy mindegyik sorozat csak egy elemet tartalmaz, és nem kötelező minden sorozatból elemet választani. Ha valamelyikből nem tudok választani, akkor venni kell a következő szakaszt. Visszalépéskor azonban figyelni kell arra, hogy melyik sorozatból illetve szakaszból nem választottam, és nem szabad ide lépni.

Nézzük meg a következő példát. Legyenek a $h_1,h_2,...,h_{10}$ szakaszok hossza 5,6,8,7,9,6,2,5,4,8 és a H szakasz hossza 30.

Ebben az esetben vehetem sorba az 5,6,8,7 szakaszokat(összegük 26), de a 9 és a 6 hosszú már nem jó, hiszen az összhossz 30. A következő jó szakasz tehát a 2 hosszú, de az összhossz még így is csak 28. A maradék szakaszokat is megpróbáljuk kiválasztani, de ezek túl hosszúak, így vissza kell lépni, de nem a 6, hanem az utolsó "jó", azaz a 7 hosszúságú szakaszra. Ezután már nem is próbálkozhatom a 9 és a 6 hosszú szakaszokkal, hiszen már az előbb megállapítottam, hogy ezek túl hosszúak, ezért választottam a 2 hosszú szakaszt, ami szintén nem vezetett megoldáshoz, azért a 4 hosszút kell választani, miután megállapítottuk, hogy az 5-ös túl hosszú. Egy megoldást sikerült megtalálni. Mivel a jelenlegi szakaszokkal nem lehet továbblépni, ezért

vissza kell lépni a 8-as szakaszig és választani a következőt, azaz a 9 hosszú szakaszt, ahogyan azt a következő ábra szemlélteti:



Az eddigiekből már látható a megoldás menete és az is, hogy célszerű egy láncolt listát kialakítani, hogy meghatározható legyen a következő választható szakasz és lehessen tudni, hogy hová lehet visszalépni. Ezért célszerű egy n elemű vektort használni, amelyek elemei rekord típusúak a következő mezőkkel :

hossz : a szakasz hossza
előre: az őt követő szakasz indexe
hátra : az őt megelőző szakasz indexe (az elsőé 0)

Ha egy szakasz nincs benne a megoldásban(nincs előző vagy őt követő szakasz), akkor önmagára mutat.

Ezek után nézzük meg az algoritmust, amely az előbbi példához készült:

```
Eljárás Kiir        ** a láncolt listán visszafelé haladva kiírja a megoldást**
cikl1:=cikl
Ki: rud[j].hossz
Ciklus amíg rud[cikl1].hatra <> 0
    Ki: rud[cikl1].hossz
    cikl1:=rud[cikl1].hatra
Ciklus vége
Ki: rud[cikl1].hossz
Eljárás vége

FüggvényEljárás Jo_e       ** megkeresi a következő jó szakaszt**
j:=rud[cikl].elore+1       ** itt van jelentősége a láncolásnak**
talal:=Hamis
Ciklus amíg j<=n És Nem talal
    Ha ossz+rud[j].hossz <= osszhossz
        talal:=Igaz
        Ha ossz+rud[j].hossz = osszhossz akkor Kiir  **lefedhető**
    Egyébként j:=j+1        ** egyesével lépked előre **
    Elágazás vége
```

```
Ciklus vége
Eljárás vége (Visszad: talal)

Program Szakaszok
ossz:=0 ;n:=10
** Alapbeállítás **
rud[1].hossz:=5: rud[2].hossz:=6:rud[3].hossz:=8:rud[4].hossz:=7:
rud[5].hossz:=9:rud[6].hossz:=6:
rud[7].hossz:=2:rud[8].hossz:=5:rud[9].hossz:=4:rud[10].hossz:=8:
Ciklus cikl:=1 -től n-ig       ** most n értéke 10 **
   rud[cikl].elore:=cikl       ** az elején a mutatók "magukra" mutatnak**
   rud[cikl].hatra:=cikl
Ciklus vége
rud[1].hatra:=0                ** az első rekord hátra mutatója 0 lesz**
cikl:=1
ossz:=ossz+rud[cikl].hossz  **bekerül az összbe az első szakasz hossza**
Ciklus amíg cikl>=1 És cikl<=n
   Ha Jo_e akkor
      rud[cikl].elore:=j     ** láncolás előre **
      rud[j].hatra:=cikl     ** láncolás hátra **
      ossz:=ossz+rud[j].hossz **a kiválasztott szakasz hossza kerül az össz-be**
      cikl:=j
   Egyébként
      ossz:=ossz-rud[cikl].hossz
      rud[cikl].elore:=cikl           ** az előre láncolás megszűnik **
      Ha rud[cikl].hatra=0 akkor  ** ha elérte az első szakaszt**
         cikl:=cikl+1          **a "következő első" szakasz választása**
         rud[cikl].hatra:=0
         ossz:=rud[cikl].hossz
      Egyébként
         cikl:=rud[cikl].hatra        ** visszalépés!!!!**
      Elágazás vége
   Elágazás vége
Ciklus vége
Program vége
```

A feladat b) részének megoldását az olvasóra bízom.
Nézzük meg az első fejezetben tanult binárisfa-bejárások rekurzív változatait:

```
Eljárás KBJ(mut)
   Ha mut<>0 akkor
      Ki: Adat(mut)
      KBJ(BAL(mut))
```

KBJ(JOBB(mut))
Elágazás vége
Eljárás vége

Eljárás BKJ(mut)
Ha mut<>0 **akkor**
BKJ(Adat(Mut))
Ki: Adat(mut)
BKJ(JOBB(mut))
Elágazás vége
Eljárás vége

Eljárás BJK(mut)
Ha mut<>0 **akkor**
BJK(BAL(mut))
BJK(JOBB(mut))
KI: Adat(Mut)
Elágazás vége
Eljárás vége

FELADATOK

1) Fedjünk le egyszeresen egy négyzetet különböző méretű kisebb négyzetekkel.

2) Készítsünk latin négyzetet. (Az n*n latin négyzet minden sorában és minden oszlopában 1-től n-ig kell felsorolni a számokat, azaz minden szám csak egyszer szerepelhet ugyanabban a sorban ill. oszlopban). Készítsük el az összes ilyen négyzetet.

3) Egy N tagú társaságban tudjuk, hogy kik ismerik egymást. Válasszuk ki a legtöbb embert úgy, hogy :
   a) közülük mindenki ismeri a másikat
   b) közülük senki sem ismeri a másikat.

4) Hosszabb turistaútra indulunk. Hátizsákunkban maximálisan N kilogramm kenyeret tudunk cipelni. Adott a magunkkal viendő tárgyak tömege (természetesen együtt jóval nehezebbek N-nél) és használati értéke (például 10 pont a feltétlenül szükséges, 1 pont az "ha van hely magammal viszem"). Állítsuk össze a hátizsákot, hogy a magunkkal vitt tárgyak együttes használati értéke a legnagyobb legyen.

5) Egy házasságközvetítő irodában a számítógépet hívják segítségül M nő és M férfi összeházasításához. Mindegyik nő sorba teszi az összes férfit és mindegyik férfi aszerint, hogy milyen szívesen házasodna össze velük. Ezek alapján az számítógép párosítja össze őket. A házasítást akkor tekinthetjük jónak, ha nincs olyan nő és férfi, akik egymást jobban kedveli (előbbre állnak a választási sorrendben), mint kijelölt házastársukat.
   a) Keressünk egy lehetséges jó megoldást!
   b) Keressük meg az összes jó megoldást!

6) Adott négyzetekkel, melyeknek összterülete megegyezik egy téglalap területével, lefedhető-e a téglalap:
   a) Ha a négyzetek között nincs egybevágó?
   b) Ha a négyzetek között vannak egybevágóak?

7) Készítsünk olyan 0 és 1 elemből álló N*M-es mátrixot, amelynek minden sorában I darab, minden oszlopában j darab egyes van, ezenkívül bármely két sorában pontosan k darab egyes áll ugyanazon a helyen!

8) Egy táblán az - ábrán látható módon - 32 mozgatható bábot helyeztünk el, a középső helyet üresen hagyva. A bábukkal úgy lépünk, hogy a közvetlen vízszintes vagy függőleges szomszédon át egy üres helyre ugrunk. Az átugrott bábut le-

vesszük. Készítsünk olyan programot, melynek segítségével az összes bábut levesszük, és az egyetlen fennmaradó a középső (eredetileg üres) helyre kerüljön!

```
    x x x
    x x x
x x x x x x x
x x x 0 x x x
x x x x x x x
    x x x
    x x x
```

9) Egy nyelv fordítójának feladat, hogy egy beírt mondatot az ún. start szimbólumból (S) kiindulva felismerjen; azaz az adott helyettesítési szabályok alapján rekonstruálja a mondathoz vezető lépéseket. Jelölje A,B... azokat a szavakat (nem írásjeleket), amelyeket helyettesítenünk kell, x,y... azokat a szavakat (írásjeleket), amelyekből a mondat áll. A mondat csak írásjelekből állhat! A helyettesítési szabály jelölésére példa:
S::= A/x
értelmezése: S helyébe A-t (tovább helyettesítendő szó) vagy x-et helyettesítünk a mondatba.

Egy nyelvben a helyettesítési szabályok az alábbiak:
S::=A/B A::=xA/Y B::=xB/z
Készítsünk programot, amely eldönti, hogy az alábbi mondatok értelmesek-e a nyelvben.

xxxz,	zx
xyz,	xxyzxyz
xxA,	tetszőleges x,y,z -ből álló sorozat
z,	tetszőleges sorozat

10) Négy színes kockát kell elhelyezni egymás mellett úgy, hogy az azonos síkban levők lapjainak színe különböző legyen (a kockákat szorosan egymás mellé teszszük le)! P=piros, K=kék, Z=zöld, F=fehér
A kockák színezése az alábbi:

```
P F K   Z F P    K     P
  F       F      Z     F
  Z       Z      P     P
  P       K     FZK   PKZ
```

11) Tekintsünk egy olyan sakktáblát, amelynek nyolc sora van, de az oszlopok száma 2 és 8 között változik. Oldjuk meg ezen a sakktáblán a nyolcvezér-problémát.

12) Adott egy N x M-es mátrix, amely tartalmazza egy terület domborzati térképét (elemei pozitív egész számokat tartalmaz, amelyek a megfelelő pontok magasságát jelölik). A terület (mátrix) egy adott pontjára labdát teszünk, amely mindig

lefelé gurul. A lehetséges irányok közül (4 ilyen van) véletlenszerűen választ. Írjunk programot, amely eldönti, hogy egy adott pontra helyezve a labdát az kigurul-e!

13) Adott N db élelmiszerbolt, amelyek különböző áron értékesítenek kenyeret és M pékség, ahol azonos fajtájú kenyeret állítanak elő, de különböző árakon. Minden bolt csak egy pékségtől rendelhet árut, de egy pékség több boltot is elláthat kenyérrel. Ismerjük ezen kívül a pékségek kapacitását, távolságukat az egyes boltoktól, valamint a boltok által rendelt mennyiséget. Írjunk olyan programot, amely képes különböző stratégiák esetén kiírni, hogy melyik bolt honnan kapja a kenyeret, mennyi az egyes rendelések szállítási távolsága, mennyi az egyes boltok és pékségek haszna az ismert árak esetén. A stratégiák a következők:
    a) minden bolt tudjon rendelni az eladási árnál olcsóbb kenyeret;
    b) a szállítások össztávolsága minimális legyen, de egyik távolság se legyen egy bizonyos távolságnál (Y) nagyobb;
    c) a boltok összköltsége a lehető legkisebb legyen.

14) A felhasználó megadja, hogy egy sakktáblára hány vezért és hány futót szeretne helyezni. Ha létezik olyan lerakása a bábuknak a táblára, hogy ne üssék egymást, jelenítse meg ezt a program!

IV. GRÁFTÍPUSOK

BEVEZETÉS

A gráf pontok és az őket összekötő élek együttese (ez utóbbit pontokon értelmezett relációként is felfoghatjuk, pontosabb definíciót az adatszerkezet c. fejezetben olvashattál). Igen sok alkalmazás vezethető vissza gráfelméleti problémákra. Csak néhányat a közismertek közül:

- párosítás
- szállítási feladatok
- térképszínezés (a szomszédos országok színe más legyen, és a lehető legkevesebb színnel)[5]
- labirintusproblémák

Fogalmak:

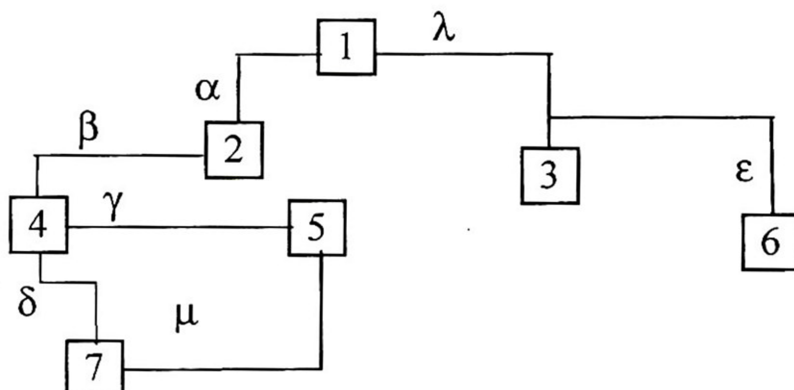
- *Teljes gráf*: az n pontú gráf, amelynek bármely két pontja össze van kötve éllel.
- *Irányított és irányítatlan gráf* : az irányított gráfban az éleknek van iránya, tehát az élek a pontok között nem szimmetrikus kapcsolatot teremtenek.
- *Út*: olyan egymáshoz csatlakozó élsorozat, amely egyetlen ponton sem megy át kétszer
- *Kör*: Olyan út, amelynek kezdő -és végpontja megegyezik
- *Izolált pont*: amelyre nem illeszkedik él.
- *Fa*: összefüggő körmentes gráf
- *Euler- vonal, Hamilton-kör*: olyan élsorozat, amely a gráf egyetlen egy élén sem megy át egynél többször, és a gráf minden élét tartalmazza, illetve (Hamilton körnél) a gráf minden pontját tartalmazza
- *Hurokél*: olyan él, amelynek kezdő - és végpontja ugyanaz a csúcs.

Legyen n - pontok száma, é - élek száma.

[5] Nem is olyan régen bizonyították be, hogy négy szín elegendő

TÁROLÁSI MÓDOK

Tároljuk valamilyen formában a következő gráfot!



Adjacencia-mátrix (csúcs-, szomszédsági mátrix)

A[i,j] **Igaz**, ha van a gráfban (i,j) él (az i. csúcsból a j. csúcsba vezető, vagy fordítva, j-ből i-be).

A[i,j] **Hamis**, ha nincs a gráfban (i,j).

Azaz:

	1	2	3	4	5	6	7
1		I	I				
2	I			I			
3	I					I	
4		I			I		I
5				I			I
6			I				
7				I	I		

Néhány észrevétel:

1. Ha a gráf nem irányított, akkor az A szimmetrikus.
2. Az A egyes soraiban levő Igaz értékek száma megegyezik a sorhoz tartozó pontból kimenő élek számával (p(i) "kimenő" fokszámával), az oszlopbeli Igazak száma pedig a pontba bejövő élek számával ($p^*(i)$ "bemenő" fokszámával). Speciálisan a nem irányított gráfokra e kettő megegyezik ($p(i) = p^*(i)$).
3. A hurokéleket a mátrix főátlója írja le.
4. Ha é<<[6]n, akkor az A nagyon rosszul "kitöltött", azaz az Igaz értékek relatív száma <= 2*é/(n*n) kicsi. (A két szorzó az irányítatlan gráfoknál értelmes. Ha nincs hurokél, akkor természetesen lehet egyenlőség.)

[6] Nagyságrendekkel kisebb

5. Ha párhuzamos élek is vannak, akkor ez az ábrázolás ezt nem tudja jelezni. Ekkor az ábrázolást pl. a következőképpen lehetne módosítani:

$$A[i,j] \begin{cases} >0, \text{ ha n párhuzamos él megy (i,j) között} \\ = 0, \text{ ha nincs a gráfban (i,j) él.} \end{cases}$$

A rossz kitöltöttség okozta memóriapazarláson segíteni lehet a következő ábrázolási módszerrel:

Adjacencia-lista (csúcs-, szomszédsági lista)

A[i]=az i-ből kiinduló élek végpontjainak a listája .

1	2	3	
2	1	4	
3	1	6	
4	2	5	7
5	4	7	
6	3		
7	4	5	

Incidencia-mátrix ("pont-él" mátrix)

	α	β	γ	δ	ε	λ	μ
1	I					I	
2	I	I					
3					I	I	
4		I	I	I			
5			I				I
6					I		
7				I			I

$$I[p,e]= \begin{cases} \text{Igaz, ha az e él a p ponton megy át} \\ \text{Hamis, ha nem.} \end{cases}$$

Nyilvánvaló, hogy

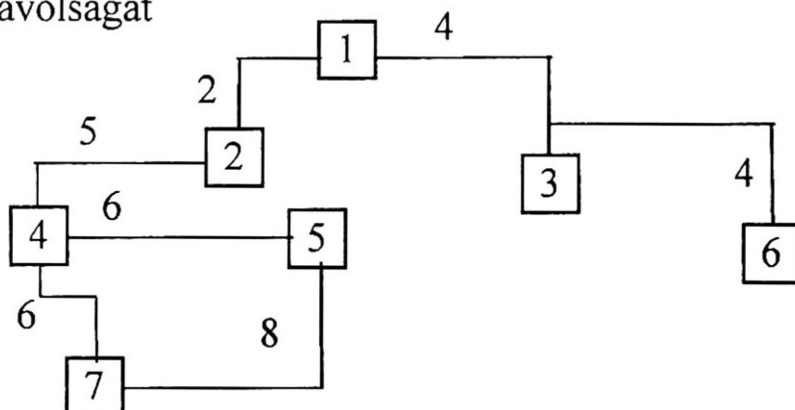
1. I minden oszlopában 1 vagy 2 Igaz érték lehet; 1 is csak akkor, ha hurokélhez tartozik.
2. Kitöltöttsége ennek sem túl jó. $\leq 2*é/(n*n)=2/n$.
3. A sorokbeli Igaz értékek száma meghatározza a pont fokszámát.
4. A fenti definíció nem irányított gráfokra vonatkozik.

Számítástechnikailag sokkal kézenfekvőbb ábrázolás: "kössük össze" azokat pontelemeket, amelyek között (irányított) él vezet. Az összeköttetést a "szokványos" memóriamutatók biztosíthatják.

Mutatós ábrázolás

Egy-egy eleme a gráfszerkezetnek tartalmazza a pont leírását magát és a vele relációban levő pontelemek "címét" (a többszörös éleket nem tároljuk). Ezzel gyakorlatilag mutatókkal láncolt listát tudunk megvalósítani, viszont nem összefüggő gráfok esetén a mutatók nem fogják összetartani a szerkezetet, azaz nem lehet csak általuk bejárni az összes elemet, vagyis a láncolt lista valójában több láncolt listarészből tevődik össze, amelyek között a kapcsolatot biztosítani kell.

Az eddigiekben az egyes pontokat "egységnyi" élek kötötték össze. Gyakori azonban, hogy az élekhez hosszúságot kell rendelni, azaz meg kell mondani a pontok egymástól vett távolságát



Ekkor pl. az incidencia-mátrix a következő lehet:

$$I[p,q] = \begin{cases} t, \text{ ha } |p-q|=t \\ \infty, \text{ ha nincs a gráfban } (p,q) \text{ él.} \end{cases}$$

	1	2	3	4	5	6	7
1	∞	2	4	∞	∞	∞	∞
2	2	∞	∞	5	∞	∞	∞
3	4	∞	∞	∞	∞	4	∞
4	∞	5	∞	∞	6	∞	6
5	∞	∞	∞	6	∞	∞	8
6	∞	∞	4	∞	∞	∞	∞
7	∞	∞	∞	6	8	∞	∞

Ezzel egy távolság-mátrixot sikerült definiálni.

Természetesen további ábrázolási módok léteznek még, és találhatók ki, ezek végiggondolása azonban az olvasó feladata marad.

A GRÁFBEJÁRÁS ALGORITMUSAI

Olyan alapvető algoritmusokról lesz szó, amelyekre a legtöbb gráfra vonatkozó tevékenység algoritmusát építeni lehet. Sok esetben közömbös az is, hogy milyen ábrázolással valósítjuk meg a gráfot. Ahol ez lényeges, ott külön említésre kerül ez a tény. (Ha több komponensből áll a gráf, az eljárásokat hívó környezetnek kell gondoskodnia arról, hogy a bejárás részgráfonként megtörténjen!)

Szélességi bejárás

Az egyik legáltalánosabban elterjedt algoritmust fogjuk megismerni, amelyet igen sok helyen hasznosítanak. Leggyakrabban például a nyomtatott áramköröket tervező programok használják fel arra, hogy két pont összekötését megvalósítsák, és alkalmazzák például egyes fordítóprogramok a nyelvi elemzéshez, a szintaktikai fa bejárásához.

A kínai hadseregnek a RIZS hadművelet szigorúan titkos haditervét kell átjuttatnia egy óriási hegységrendszeren. A katonák nem tudnak hegyet mászni, így csak az ösvényeket használhatják (az ellenség természetesen a hegyet mássza). Idegen terepen vannak, térkép nélkül, és csak annyit tudnak, hogy van biztosan átjáró. A parancsnok a következő tervet dolgozta ki:

A hegység lábánál felsorakoztatja a hadsereget, és elindítja azt. A menetelő katonák nyomot hagynak maguk után, hogy később visszataláljanak oda, ahonnan elindultak. Ha a hadsereg útelágazáshoz érkezik, annyi részre válnak szét, ahány elágazás van. Ha megint ilyen kereszteződéshez érnek, megint tovább osztják a csoportokat, s így tovább. (Elég sokan vannak hozzá!) Ha valamelyik csoport osztódott, a kisebb egységek is megkapják a haditerv egy-egy másolatát. Ha egy csoport olyan nyomokkal találkozik, amelyet ott járó társaik hagytak, visszafordulnak és a lerakott jelzések segítségével visszatérnek a kiindulóponthoz. Mivel minden lehetséges útra jut katona,

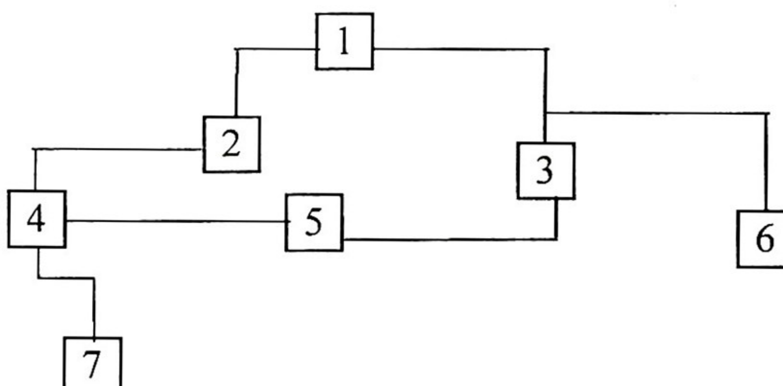
így lesz olyan, amelyik pontosan a létező legrövidebb utat járja be, tehát a haditerv eljut a hegység túloldalára.

Próbáljunk meg egy ilyen algoritmust írni!

Alapelve a következő: egy tetszőleges kezdőpontból egy impulzust indítunk, amely végighullámzik a gráfon mindaddig, amíg el nem éri a gráf "szélét". Ennek megvalósításához egy sorra van szükség. Ebbe a sorba az impulzus által elért, s még fel nem dolgozott pontok kerülnek. Annak elkerülésére, hogy egy már "bejárt" vagy "bejárásra éppen kijelölt" (azaz a sorban már benne lévő) pontot újból felvegyünk a sorban, egy halmazt használunk. A halmaz őrizze meg azokat a pontokat, amelyekbe már eljutott az algoritmus. Az ún. szélességi bejárás algoritmusának vázlata a következő:

```
Eljárás Szélességi_bejárás ( x ) **a bejárás kezdőpontja**
    pont:=x
    Sor, Halmaz nullázása
    Sorba(Segédpont)  ┌── **betesszük a pontot a Sorba és a Halmazba**
    Halmazba(pont)    └──
    Ciklus amíg Nem üres a Sor **Ha a sor üres, akkor nincs több pont**
        Sorból(pont)   ( **Kivesszük és kiírjuk, feldolgozzuk a pontot**
        Ki: pont       (
        Ciklus i:=1 -től KövetkezőPontokSzáma(pont) -ig
                           └── **a pontból hány pontba vezet él - függvény**
            segédpont:= KövetkezőPont(pont,i) **vesszük ezek közül az i-ediket
                                                  - függvény**
            Ha Nem eleme a halmaznak a segédpont akkor
                Sorba(segédpont),
                Halmazba(segédpont) **még nem jártunk itt**
            Elágazás vége
        Ciklus vége
    Ciklus vége
Eljárás vége
```

Kövessük nyomon az algoritmust az alábbi példa segítségével. Legyen a gráf:



A bejárás sorrendje: 1, majd egy fázisban 2,3, aztán együtt 4,5, egyedül a 6, és végül 7.

Nézzük meg, hogyan alakul a sor és a halmaz az algoritmus során:

Sor	Halmaz
0	0
1	1
2,3	1,2,3
3,4	1,2,3,4
4,5,6	1,2,3,4,5,6
5,6,7	1,2,3,4,5,6,7
6,7	1,2,3,4,5,6,7
7	1,2,3,4,5,6,7

Mélységi bejárás

Persze a kínai hadseregnek könnyű a dolga, hiszen a a sok katona egyszerre, párhuzamosan tud keresni.

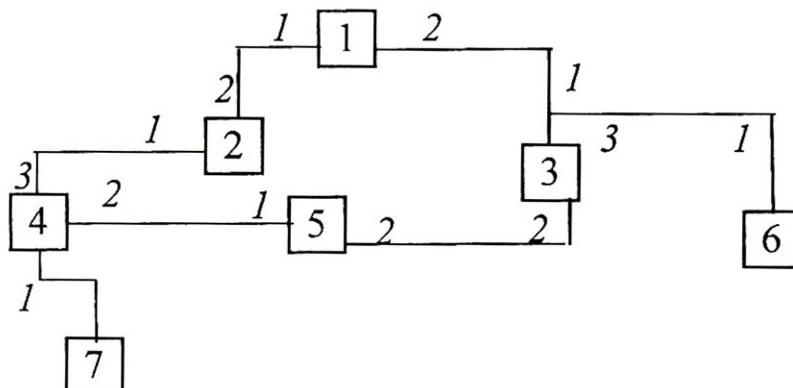
A bergengóc hadseregnek csak kevés katonája van, és azok között is csak egy akad, akinek képesítése van a haditerv szállításra. Taktikát kell tehát váltani. Tegyük fel, hogy a képesítéssel rendelkező futni is tud, és ráadásul elég sokat, különösen ösvényeken. Feladata nem csak az lesz, hogy elvigye a haditervet, hanem a megtalált utat is fel kell jegyeznie. A taktika a következő lesz: elindul a kezdőpontból és ha abból több út indul, akkor kiválaszt egyet és azon fut tovább, amíg egy újabb csomóponthoz nem ér, és ott megint egy utat választ... és így tovább. Azokat az utakat, amelyeken már járt (futott), megjelöli, és mindig megjegyzi azt is, hogy mely csomópontokat érintett. Két lehetőség van: a futár elérte célját vagy egy olyan csomópontba jutott, ahonnan már nem lehet továbbmenni (mert a csomópontból nem visz ki út, vagy már valamennyi, a csomópontból kiinduló úton járt). Ha a futár egy csomópontban elakadt, visszalép arra a csomópontra, ahol előzőleg már járt, s onnan most választ egy másik úton próbál továbbhaladni. Ha innen sincs merre mennie, akkor az ezt megelőző pontra lép vissza és onnan keres utat.

Az algoritmus ennek megfelelően a következő lesz: induljunk a gráf egy tetszőleges pontjából, majd vegyünk egy belőle közvetlenül elérhető pontot. E szomszédos pontból járjuk be ugyanezzel a módszerrel a gráfot. Természetesen a bejárt részeket feljegyezzük, hogy egy másik úton ugyanoda jutva nehogy még egyszer újra bejárjuk. Ahhoz, hogy vissza tudjunk lépni egy elágazási pontra, ha az egyik úton már nem tudunk tovább haladni, vermet használunk (visszalépéses algoritmus). Ez azonban nem elég, ugyanis nem csak azt kell tudnunk, hogy mely pontnál jártunk már, hanem azt is, hogy az elágazási pont mely élén indultunk el korábban, ezért az elágazási pontokból kiinduló éleket sorszámozzuk.

```
Eljárás Mélységi_bejárás (x)          **egy tetszőleges kezdőpont**
pont:=x; elsz:=1                      **a pont 1. élénél tartunk**
Verem, Halmaz nullázása
VerembeBe(pont, 1)                    **a verembe az induló él sorszáma is bekerül**
Ciklus amíg Nem üres a Verem
    Ha Nem eleme a pont a Halmaznak akkor
        Ki: pont
        HalmazbaBe(pont)
    Elágazás vége
                                       **a pontból kiinduló élek közül
    élsz:=KövetkezőJóÉl(pont, élsz)     a következő olyannak a sorszáma,
                                       amely nem Halmazbeli pontba vezet**
    Ha élsz<= KiindulóÉlekSzáma(pont) akkor       **ha ez még jó**
        Verembe(pont, élsz)           **ebből az élből halad tovább**
        pont:=KövetkezőPont(pont,élsz) ; élsz:=1
    Különben
        Veremből Ki(pont,élsz); élsz:=élsz+1    **egy újabb éllel próbálko-
        zunk**
    Elágazás vége
Ciklus vége
Eljárás vége

FüggvényEljárás KövetkezőJóÉl(pont, élsz)
    Ciklus amíg élsz<=KiindulóÉlekSzáma(pont) És Eleme a
                        KövetkezőPont(pont,élsz) a Halmaznak
        élsz:=élsz+1
    Ciklus vége
Eljárás vége (Visszaad: élsz)
```

Kövessük végig ezt az algoritmust is az előbbi példagráf segítségével (az egyes csúcsokból kiinduló éleket *számokkal* azonosítjuk):



visszalépések

A bejárás sorrendje: 1,2,4,7,(4), 5,3,6 (3,5,4,2,1)

pont/él	verem	halmaz
	0	0
1/1	1:1	1
2/1	1:1,1:1	1,2
4/1	1:1,1:1,2:1	1,2,4
7/1	1:1,1:1,2:1,4:1	1,2,4,7
4/2	1:1,1:1,2:1	1,2,4,7
5/1	1:1,1:1,2:1,4:2	1,2,4,7,5
3/1	1:1,1:1,2:1,4:2,5:2	1,2,4,7,5,3
6/2	1:1,1:1,2:1,4:2,5:2,3:3	1,2,4,7,5,3,6
3/4	1:1,1:1,2:1,4:2,5:2	1,2,4,7,5,3,6
5/3→3	1:1,1:1,2:1,4:2	1,2,4,7,5,3,6
4/3→4	1:1,1:1,2:1	1,2,4,7,5,3,6
2/2→3	1:1,1:1	1,2,4,7,5,3,6
1/2→3	1:1	1,2,4,7,5,3,6
	0	

ÚTKERESÉSI FELADATATOK

Az útkeresési feladatok alapja út keresése két tetszőleges pont között a gráfban. Ennek a problémának a megoldására az előbbiekben említett algoritmusokat kell úgy átírni, hogy az algoritmus akkor álljon le, ha vagy sikerült a gráf bejárása vagy eljutottam a keresett végpontba.

Nehezebb feladat annak az eldöntése, hogy két pont között melyik a legrövidebb út egy gráfban. Ezt elvileg megadhatjuk, ha az adjacencia-mátrixokat szorozgatjuk, de a mátrixok szorzása nagyon időigényes feladat. Gyorsabb megoldáshoz jutunk, ha a szélességi bejárással haladunk a pontokon, miközben hozzájuk rendeljük azt a lépésszámot, amelyet odáig kellett tennünk. Azokhoz a pontokhoz, amelyekhez többféleképpen is el lehet jutni, nyilván ez a lépésszám már az első eléréskor ki lesz töltve, azaz éppen a legrövidebb út hosszával van megcímkézve illetve megjelölve. Ezért szükség van egy lépésszám-vektorra, ami tartalmazza a címkéket. A korábbi algoritmusban (szélességi bejárás) egy halmazt használtunk arra, hogy a többszörös elérést észrevegyük. Most ezt a szerepet eljátszhatja a lépésszámcímke is, ha kezdőértékként valamilyen nonszensz értékkel vannak megjelölve a pontok (pl. egy nagy számmal). Össze kell tehát rendelni a korábbi halmazműveletet a lépésszámvektorművelettel.

Üreshalmaz → Lépésszám:=$+\infty$ vektor

Halmazba(pont) → Lépésszámpont:= a pont távolsága

Eleme? a segédpont → Lépésszám(segédpont) $\neq +\infty$

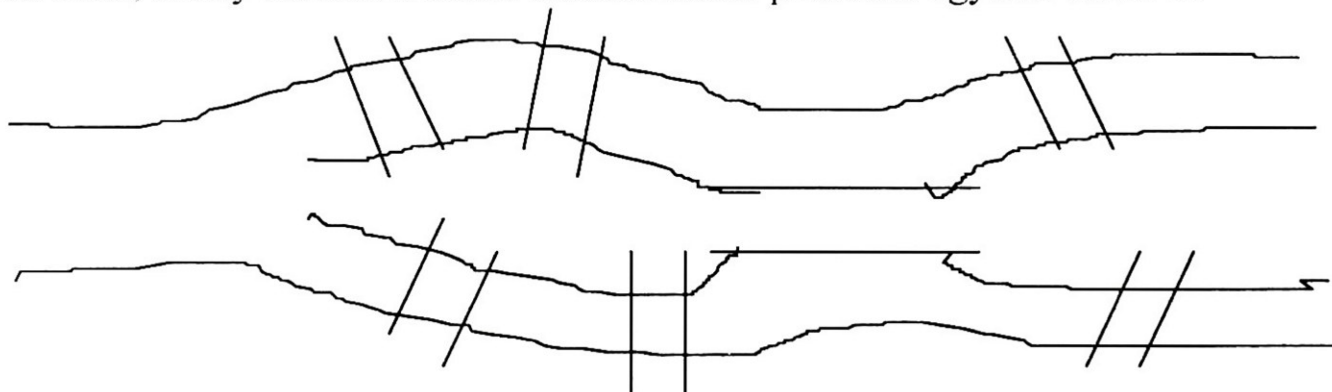
Eljárás LrútHossza(x , y, táv) ****x,y kezdő és végpont(cím szerint paraméterátadás)****

pont:=x; táv:=0
Üressor:Lépésszám:=+∞ -vektor
Sorba(pont);Lépésszám(pont):=táv
Ciklus amíg Nem Üressor? ***még van bejáratlan pont***
  És Lépésszám(y)=+∞ ***még nem jutottunk y-hoz***
  Sorból(pont)
  Táv:=Lépésszám(pont)+1
  Ciklus i:=1-**től** KövetkezőPontokSzáma(pont)-**ig**
    segédpont:=KövetkezőPont(pont,i)
    Ha Lépésszám(segédpont)=+∞ **akkor**
      Sorba(segédpont)
      Lépésszám(segédpont):=Táv
    Elágazás vége
  Ciklus vége
Ciklus vége
Táv:=Lépésszám(y)
Eljárás vége

Ennek az algoritmusnak az átalakításával könnyen megoldható két pont közötti egy legrövidebb út meghatározása, illetve két pont közötti legrövidebb utak száma.

HAMILTON-KÖR ÉS -ÚT KERESÉSE

Rejtvényújságokat gyakran használók körében biztosan ismert a következő feladat: "Húzzunk az ábra A és B pontja között vonalat a toll felemelése nélkül úgy, hogy minden élen pontosan egyszer haladjunk végig!" A klasszikus példa nem más, mint a königsbergi (a várost a Preyer folyó szeli át és hét hidat építettek az ábra szerint) hidak problémája. A város polgárai azt a kérdést vetették fel, hogy lehet-e olyan sétát tenni, amely közben a sétáló minden hídon pontosan egyszer halad át.



Ez gyakorlatilag egy gráfelméleti probléma, amelyhez hasonló (valamivel kevésbé szigorú megkötéssel) a következő:

Egy országjáró kirándulást szeretnénk csinálni a kiválasztott városokban úgy, hogy a városok között vasúton utazunk, s minden városban pontosan egyszer akarunk megfordulni. Azt szeretnénk, hogy utazásunk végeztével hazaérkezzünk, vagyis kirándulásunk induló-és célpontja ugyanaz a város legyen. Ezt olyan gráffal írhatjuk le, ahol a városok felelnek meg a gráf pontjainak, s két pontot akkor kötünk össze éllel, ha a megfelelő városok között létezik közvetlen vasútvonal.

A kérdés az, hogy adott gráfon lehet-e ilyen utat találni. Az a) ábrán olyan gráfot láthatunk, amely megfelel igényeinknek (a kirándulás útvonalát a vastagított élek jelzik), a b) ábrán olyat, amelyet nem tudunk az elképzelés szerint bejárni.

A Hamilton-kör és -út definíciói alapján teljesen világos, hogy ha G gráfnak létezik Hamilton-köre, akkor Hamilton-útja is. Értelemszerűen hasonló módon definiálhatjuk G irányított Hamilton-körét és irányított Hamilton - útját is. A feladat tehát az, hogy egy gráfról el tudjuk dönteni, létezik-e Hamilton-köre.

Tétel: Legyen G egy egyszerű gráf. Ha G minden pontjának foka nagyobb vagy egyenlő mint k, ahol $k>1$ és G pontjainak száma nem több, mint 2k, akkor G-nek van Hamilton köre.

Ha ezt a tételt felhasználva szeretnénk egy algoritmust adni, amely eldönti egy gráfról, hogy létezik Hamilton-köre, s egyúttal még meg is adja azt, akkor számolnunk kell azzal, hogy az algoritmus csak "kis" gráfok esetén lesz igazán használható. Sebességnövekedést érhetünk el, ha a körök és az utak kutatása közben a p! esetből sokat kizárunk, mert észrevesszük, hogy egy adott pontból továbbhaladva már biztosan nem találhatunk Hamilton-kört vagy utat.

Próbáljuk meg felépíteni az algoritmust, amely nem csinál mást, mint egy lehetséges módon megpróbálja felépíteni a Hamilton-utat. Tegyük fel, hogy p pontja van a gráfunknak. Állítsuk elő a p pont összes permutációját[7]:

$p_{i1}, p_{i2}, \ldots, p_{ip}$, ahol $i_1, i_2, \ldots, i_p$ az $1,2,\ldots,p$ számok egy permutációja. Tekintsük egyenként a permutációkat, és vizsgáljuk meg az alábbi élek létezését:

$(p_{i1},p_{i2}),(p_{i2},p_{i3}),\ldots,(p_{i,p-1},p_{ip})$.

[7] Azaz vesszük a pontok összes lehetséges sorrendjét. Pl. az 1 2 3 számok permutációi: 1 3 2, 2 1 3, 2 3 1, 3 2 1, 3 1 2 és természetesen az 1 2 3.

Ha az összes ilyen létezik, akkor ezek a fenti sorrendben a gráf egy Hamilton-körét adják. Hogyan gyorsíthatnánk ezen az algoritmuson? Teljesen nyilvánvaló, hogy az élek vizsgálatát csak addig kell folytatni, amíg nem létező élt találunk, hiszen ez már eleve kizárja a Hamilton-út létezését. Tegyük fel, hogy a $(p_{i1},p_{i2}),\ldots,(p_{i,k-1},p_{ik})$ éleket létezőnek találtuk, a $(p_{i,k},p_{ik+1})$ éleket pedig nem. Ekkor azokat a permutációkat, amelyek első k+1 elemei $(p_{i1},p_{i2}),\ldots,(p_{i,k},p_{ik+1})$, tovább már nem kell vizsgálnunk, hiszen a $(p_{i,k},p_{ik+1})$ él nem létezik, tehát ezek a permutációk nem írhatnak le Hamilton-kört. Azaz, ha például a pontok: 1 2 3 4 5 6 és a 3-ból nem vezet él a 4-be, akkor nincs értelme vizsgálni sem az 1 2 3 4 5 6 sem az 1 2 3 4 6 5 sorozatot sem.

Két részprobléma maradt még, amelyeket meg kell oldani:

1. Ellenőriznünk kell az adott él létezését. Ezt könnyen megtehetjük: hozzuk létre a gráf csúcsmátrixát, ebből közvetlenül elvégezhetjük az ellenőrzést.
2. Elő kell állítani az 1,2,...,p számok összes permutációját, sőt meg kell oldani az egyes permutációk szelektív átugrását is. A fenti számokat úgy is lehet tekinteni, mint egy p jegyű, p számrendszerbeli számot. Ezeket a számokat nagyság szerint rendezve, egyúttal a permutációk közül is megadhatunk egy sorrendet. (Egy permutációt nagyobbnak nevezünk egy másiknál, ha a neki megfelelő szám nagyobb.)

A most következő algoritmussal növekvő sorrendben fogjuk előállítani a permutációkat. Bármely permutáció "legyártásához" csak az előzőre lesz szükség:

Vegyük az előző permutációt, ezt egy számsor írja le. Keressük meg a számsor végéről indulva és visszafelé haladva azt az első számot, amely mögött nála nagyobb szám áll. Ha nincs ilyen, az azt jelenti, hogy a p szám pontosan fordított sorrendben van leírva, vagyis az eddigieken túl nincs több lehetséges permutáció. Ha van ilyen, jegyezzük meg a helyét, legyen ez az i-edik szám. Most keressük meg a mögötte állók között a nála nagyobb számok közül a legkisebbet. Cseréljük ezt fel az i-edik számmal, majd rendezzük az i-edik hely mögött álló számokat növekvő sorrendbe. Ezzel megkaptuk a következő permutációt.

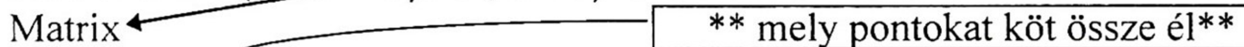
A permutációkat p számrendszerbeli számoknak tekintve könnyű ellenőrizni, hogy ez az algoritmus helyesen működik.

Tegyük fel, hogy a k-adik jegytől kezdve szelektív ugrást akarunk a permutációban. Ezt úgy valósíthatjuk meg, hogy a k-adik szám mögött állókat csökkenő sorrendbe rendezzük, majd az így kapott permutációból (amely könnyen belátható, hogy nagyobb, mint az eredeti) előállítjuk a következőt.

Lássuk ezek után az algoritmust, amelynek "gyorsítása" az olvasó feladata marad:

mátrix:n*n-es tömb logikai elemekkel, (adjacencia-mátrix)
tömb: n elemű tömb egész elemekkel, a pontok sorszámát tartalmazza

```
Program Hamilton
Be: szam (egesz és <n) **a pontok száma**
Ciklus cikl:=1-től szam-ig
    tömb[cikl]:=cikl **a pontok sorszámát tartalmazza**
Ciklus vége
```



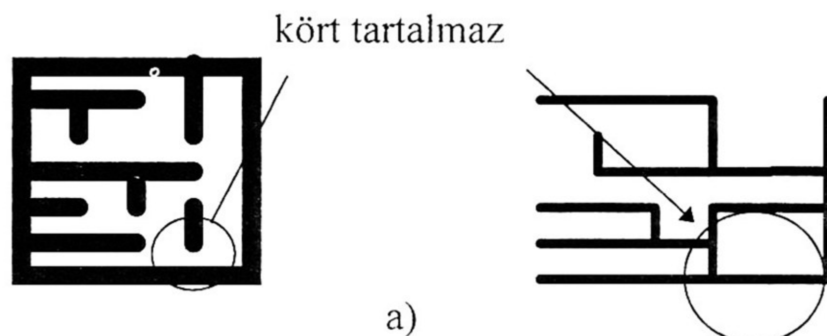
```
Rendeztömböt(1) **eljárás, amely feltölti a Mátrix elemeit Hamis értékkel**
Matrix                                   ** mely pontokat köt össze él**
Be: pont1,pont2 (egészek, különbözőek és benne vannak [0,n] -ben)
Ciklus amíg pont1<>0 **0 a végjel**
    mátrix[pont1,pont2]:=Igaz
    mátrix[pont2,pont1]:=Igaz
    Be: pont1,pont2 (egészek, és benne vannak [0,szám]-ban)
Ciklus vége
cikl:=szám-1
Ciklus amíg cikl<>1 **a permutáció első eleme 1 kell, hogy legyen(ebből
    cikl:=szám-1: Voltcsere=Hamis                              indulunk)**
    Ciklus amíg cikl<> 1 És Nem Voltcsere
        Ha tömb[cikl]<tömb[cikl+1] akkor
            index:=Keres(cikl+1)
            Csere(tömb[cikl],tömb[index])
            Voltcsere:=Igaz
            Rendeztömböt(cikl+1)
            Ha Vizsgál akkor
                Ki: tömb **az így kapott permutáció jó, ezért ki kell iratni a tömb
            Elágazás vége                                           elemeit**
            Elágazás vége
        Cikl:=Cikl-1
    Ciklus vége
Ciklus vége
Program vége
FüggvényEljárás Vizsgál **a kapott permutáció Hamilton-kör?**
jó:=Igaz
i:=1
Ciklus amíg i<>szám És jó
    Ha mátrix[tömb[i],tömb[i+1]]=Hamis akkor **a pontok között van-e él?*
        jó:=Hamis
    Elágazás vége
    i:=i+1
Ciklus vége                           **vissza is tudok-e jutni?**
Ha Jo És mátrix[tömb[szám],tömb[1]] akkor
    jo:=Igaz
Egyébként
```

jo:=**Hamis**
Elágazás vége
Eljárás vége (Visszaad:jó)

A Csere eljárás a kapott két paramétert cseréli fel a tömbben, a Keres eljárás pedig megkeresi a kapott paraméternek megfelelő indexű elemtől nagyobbak közül a lehető legkisebbet a tömbben, a [paraméter, szám] intervallumban. A Rendeztömböt eljárás a kapott paraméterindextől kezdődően a tömböt növekvő sorrendbe rendezi. Ezeknek az eljárásoknak a megvalósítását valamint az algoritmus gyorsabbá tételét (esetleg egy újabb algoritmus elkészítését) az olvasóra bízom.

LABIRINTUSPROBLÉMÁK

Mi a labirintus? [8]Tulajdonképpen egy gráf, hiszen a folyosókból (élek) és elágazásokból (csomópontok) áll. Ez azonban így még nem korrekt definíció, ugyanis egy "tisztességes" labirintustól elvárjuk, hogy bármely pontjából bármelyikbe eljuthassunk, s ne legyen olyan zárt terület, amelyből nem tudunk kijutni (ezt már börtönnek hívják), illetve oda nem tudunk bemenni.



Azt mondhatjuk tehát, hogy egy labirintus tulajdonképpen egy összefüggő (irányítottan összefüggő) gráf. A továbbiakban nem irányított gráfokkal foglalkozunk, mivel az "irányított útvesztők" kezelési technikája teljesen hasonló ezekéhez. Az ábrán egy labirintust és a hozzá tartozó gráfot mutatja. A labirintust több szempontból is osztályozhatjuk. Az egyik ilyen fontos szempont lehet az, hogy a labirintus hány dimenzióban (síkban, ill. térben) írható le. Az útkeresések esetében ez teljesen mindegy, de a tervezési eljárásoknál ennek ismerete már igen fontos. A másik ilyen szempont az, hogy a labirintust leíró gráf egyszerű-e vagy sem.

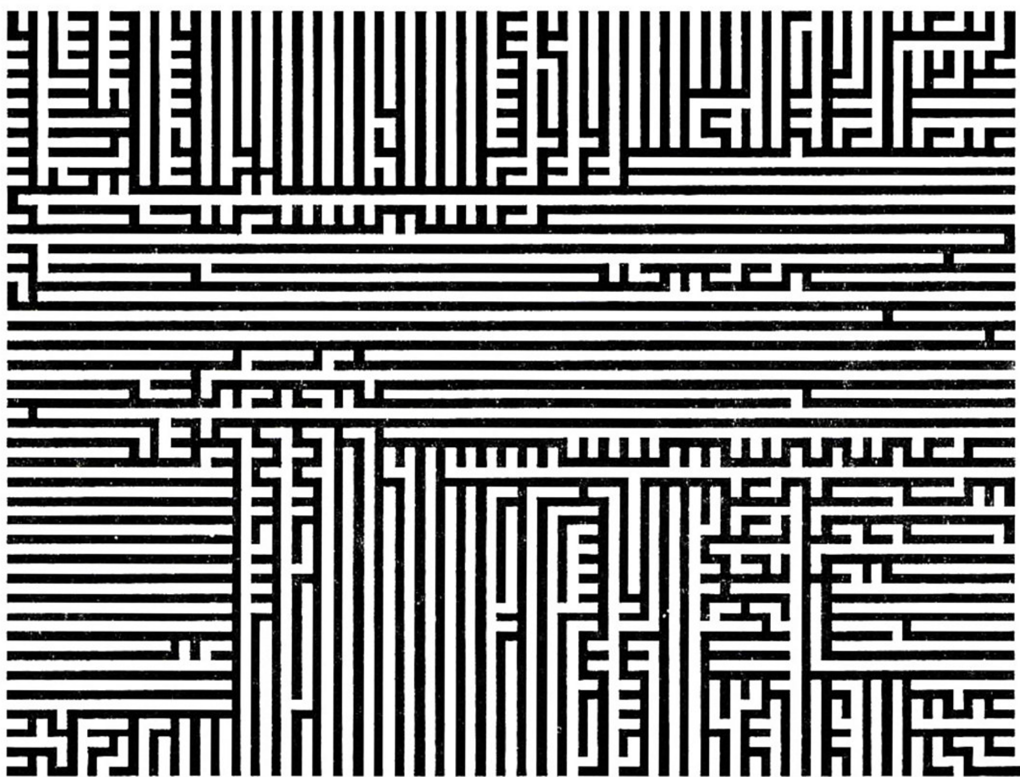
Egy harmadik lehetőség annak vizsgálata, hogy a labirintusnak megfeleltetett összefüggő gráf fa-e, vagy pedig tartalmaz köröket. Más szempontokat is találhatnánk, de számunkra ezek lesznek fontosak.

[8] Talán ismert az olvasó számára a Minotaurusz által őrzött labirintus, amelyből a királyfi Ariadné fonala segítségével szabadította ki a

Az útvesztő megtervezése tulajdonképpen egyszerű feladat: egy összefüggő gráfot kell előállítanunk. Erre egy lehetséges mód a következő:

A gráf pontjait két halmazba soroljuk, az egyik halmazba a "beépített", a másik halmazba a "kimaradt" pontokat. Kezdetben a gráfunk nem tartalmaz éleket csak egy "beépített" pontot. Ezután a gráfot élekkel bővítjük: vagy két beépített pont között húzunk élt, vagy pedig egy éllel újabb, eddig "kimaradt" pontot kapcsolunk a beépített pontokhoz. Az élek bővítését csak akkor állíthatjuk le, ha már nincs "kimaradt" pont. Az így kapott gráf biztosan összefüggő lesz, hiszen bármely két pont között létezik út.

Ahhoz, hogy egy labirintust papírra lerajzolhassunk, annak síkbelinek kell lennie, s ahhoz, hogy az a) ábrához hasonlóan egy "négyzethálót" feleltethessünk meg a gráfnak, annak - az összefüggőségen túl - egyszerű gráfnak kell lennie, s az egyes pontok fokszáma legfeljebb négy lehet. Egy ilyen gráfhoz rendelhető labirintust láthatunk a következő ábrán:



b)

Ez a labirintus számítógép segítségével készült, csupán gráfelméleti tételek felhasználásával (fehér színű a fal). Nézzük meg, hogyan is készült!

Az útvesztő tárolásához egy kétdimenziós karaktertömböt használunk fel. A karaktertömbben természetesen különböző karakterek jelzik a labirintus falát és az utat. A tervezés menete a következő:

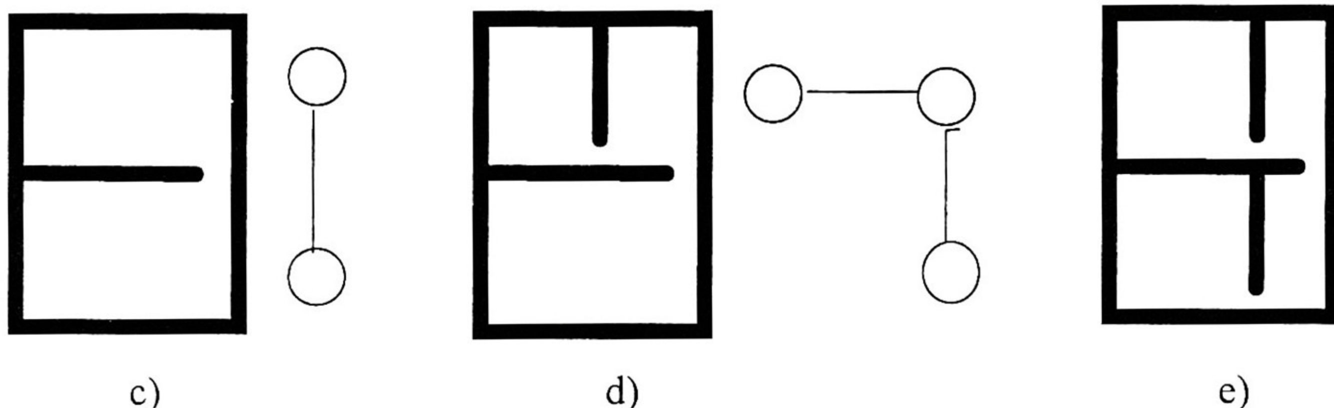
1. A tömb minden eleme legyen út. A keretet határoló elemek helyére írjunk fal elemet.

Négy irányból fogunk falakat építeni, mindig a labirintus megfelelő széléről indulunk. Ezeket a falakat csak páratlan koordinátájú sorokba, illetve oszlopokba tehetjük, hogy ne zárhassuk el egy területet a labirintus többi részétől.

2. Válasszunk ki véletlenszerűen egy irányt és az adott irányban egy szabad sort ill. oszlopot, s építsünk oda falat. A fal építése úgy történik, hogy addig haladunk előre az adott sorban (oszlopban), amíg már csak a szemben levő határoló falat látjuk, s ekkor "lerakjuk az első téglát". A falat egészen addig húzzuk, amíg újabb falba nem ütközünk, de vigyázzunk arra, hogy a két fal között maradjon még út. Ha a labirintus szemben lévő falát elértük, megjegyezzük, hogy az adott sor, ill. oszlop már nem szabad (vagyis már beépítettük).

A c) ábrán láthatunk egy behúzott falat. A d) ábrán már egy másik fal is berajzolásra került, s ez egészen addig ér, amíg az előzőleg berajzolt fal nem "keresztezi". Az itt kiválasztott oszlopot még nem építettük meg teljesen.

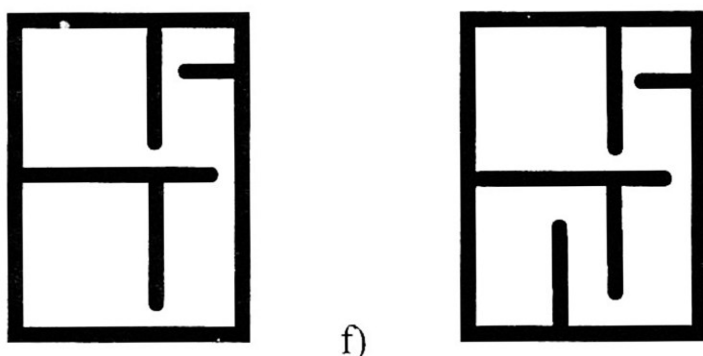
Az e) ábrán választásunk ismét arra az oszlopra esett, mint a d) ábrán. Most a 2. lépének megfelelően folytatjuk ennek építését, egészen a labirintus széléig, s megjegyezzük, hogy ennek az oszlopnak az építését már befejeztük.



c) d) e)

3. Ha van még szabad sor, vagy oszlop, a 2. lépésre ugrunk.
4. Készen vagyunk, a karaktertömb a labirintust tartalmazza.

Az f. ábrán az építés következő két fázisát látjuk.



f)

Könnyen belátható, hogy az ilyen módon épített labirintus bármely pontjából bármelyikbe eljuthatunk. Kezdetben (c) ábra) egyetlen pontból álló gráffal írtuk le az útvesztőt. Az első fal behúzása két részre bontja a labirintust, amelyek egy folyosóval

vannak összekötve. Az ezt leíró gráf két pontból és az őket összekötő élből áll. (c) ábra) Nyilvánvaló, hogy minden újabb fal építése egy újabb pont csatlakoztatása a gráfhoz egy újabb éllel, s így az elkészült labirintust egy fa írja le, (hiszen ennek az egyszerű gráfnak eggyel több pontja van, mint éle), s így annak bármely két pontja között létezik út. Az algoritmus ismeretében most már hozzá is lehetne kezdeni a program írásához:

Ha a labirintus elemeit karaktertömbként tároljuk, nagyon könnyű megoldanunk egy fal építését, hiszen a tömbelemek lekérdezésével könnyen meg tudjuk valósítani a fenti algoritmus 2. lépését. Nyilván kell tartanunk a szabad sorokat és az oszlopokat is. Ezt megtehetjük úgy, hogy minden sorhoz, illetve oszlophoz egy jelzőt rendelünk, amely azt mutatja, hogy a sor (oszlop) szabad-e. Ez egy kicsit nehézkes megoldani, ugyanis véletlenszerűen választunk sort, illetve oszlopot, s ha az nem szabad, másikat kell választanunk. Ha már kevés a szabadon beépíthető hely, könnyen előfordulhat, hogy a véletlenszám-generátor csak nehezen "találja meg" valamelyik szabad helyet.

A szabad sorok és oszlopok nyilvántartására a labirintusgeneráló eljárás elején a szabad helyek koordinátáit felírhatjuk egy-egy "kártyalapra", s megjegyezzük, hogy hány lapunk van. Ha szabad helyre van szükségünk, húzunk egy kártyát, megnézzük az általa kijelölt helyet, majd kidobjuk a lapot a pakliból. Megjegyezzük, hogy már eggyel kevesebb lapunk van, s ezt figyelembe vesszük a következő húzáskor. Ez a módszer igen egyszerű és könnyen programozható.

A kártyapaklit írja le egy tömb, amelybe a lapok vannak felsorolva, s egy index, amely a legutolsó lapra mutat. Azt, hogy egy lap szabad sort, vagy oszlopot jelent, a lapok "előjelével" különböztethető meg.

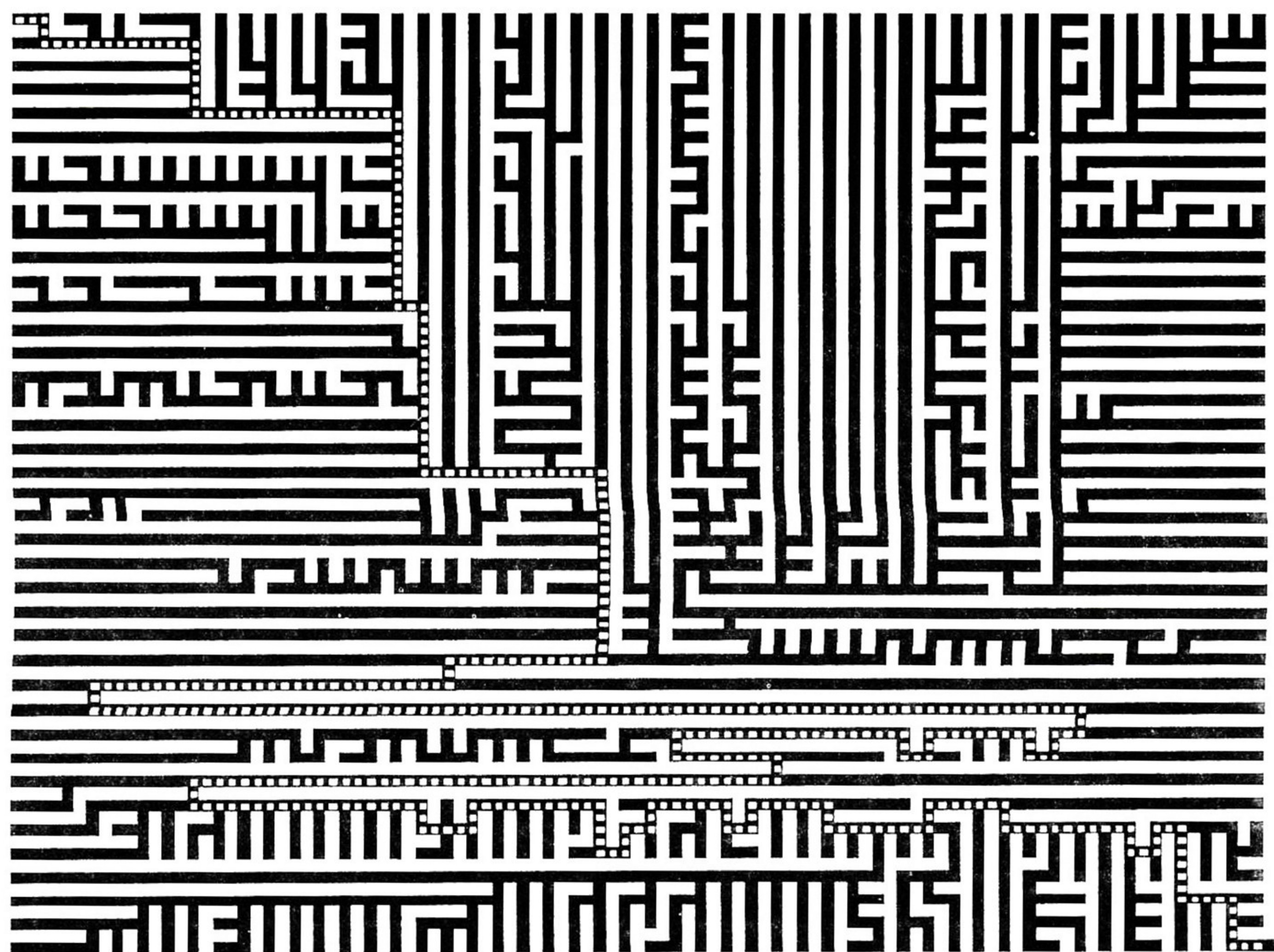
Lehetőségek:

1. Ha szeretnénk, hogy a programunk ne csak fa-labirintust tudjon építeni, hanem olyat is, amelyik tartalmaz köröket, akkor ezt megoldhatjuk, ha a falak építésekor az adott sorban ill. oszlopban lévő minden páratlan koordinátájú "téglát" csak bizonyos (kívülről megadható) valószínűséggel építhetünk be.
2. Az első néhány fal igen hosszú lesz: teljes hosszában betölt egy sort vagy egy oszlopot, s ez azt eredményezi, hogy egyszerű labirintust kapunk sok, hosszú folyosóval. Ha azonban egy sort, ill. oszlopot nem építünk be teljesen, hanem az építés közben véletlenszerűen leállunk, akkor sok egymást keresztező, rövid folyosót kapunk. Természetesen a leállás valószínűségét itt is a hívó programból kell tudnunk megadni.

Már csak a labirintust kell bejárnunk. Az útkeresések közül már két módszer bemutatásra került, ezek bármelyike használható. (Ha a labirintus nem tartalmaz kört, akkor persze nincs szükség annak vizsgálatára, hogy egy adott pontban már jártam-e) Most azonban, mivel a gráfot egészen speciálisan tároljuk (nem kell átalakítani a ka-

raktertömböt pl. adjacencia-mátrixszá), ezért egy rekurziót használó bejárást alkalmazunk, amelynek a lényege a következő:

Beteszünk a labirintusba egy egeret, amely arra van idomítva, hogy keressen sajtot, de úgy, hogy bajusza balra néző szálai mindig érintsék a falat (természetesen nem akarjuk, hogy ne jusson ki a labirintusból ezért a labirintus [gráf] körmentes, és nem olyan helyre tesszük az egeret, ahol bajusza egy „sziget" körvonalát érinti). Ha valamelyik irány zsákutcába vezet, akkor visszajut az elágazáshoz, (egy irányban mindig szabad az út), és sorrendben a következő irányban fog tovább haladni, de közben bajsza (a bal) továbbra is érinti a falat. Ha kíváncsiak vagyunk a megoldásra, csak be kell kenni az egér talpát ehető festékkel, (amit persze az egér felnyal, hiszen éhes) így csak azon utakon marad meg a festék, ahol csak egyszer járt. Ez látható a következő ábrán is:



g)

Készítsük el a leírtak figyelembevételével az algoritmust, amely körmentes gráfot generál, és ezt a bejárásnál ki is használja:

FELADATOK

1) Generáljunk egy olyan labirintust, amelyben kétfajta fal is van. Az egyiken Jerry sem tud áthaladni, a másikon csak Tom nem. A labirintusban egy véletlen helyre "rakjunk le" egy sajtot. Tom üldözze Jerry-t (ha több irányba is haladhat egy "csomópontban", akkor Jerry felé menjen!), Jerry viszont a sajt irányába haladjon, kivéve, ha meglátta Tom-ot, ekkor ugyanis menekülnie kell. Jerry használja ki a "speciális" falat. Tom kétszer olyan gyorsan mozogjon, mint Jerry. A feladatot oldjuk meg úgy is, hogy a labirintusnak van Tom számára nem elérhető pontja és úgy is, hogy nincs ilyen.

2) Egy megyében bankrablás történt az A városban. A rendőrség tudja, hogy a rablók a B városba akarják a zsákmányt szállítani. Ismerjük a megye úthálózatát. A rendőrség úttorlaszokat szeretne állítani (csak utakra teheti meg, kereszteződésekben nem!), úgy , hogy biztosan elkapják a tetteseket, de a lehető legkevesebb úttorlaszt kelljen állítani. Írjuk meg az algoritmust.

3) Adjacencia és incidencia mátrix segítségével tárolt gráf esetében is adjuk meg, hogy egy gráf valamennyi csúcsának a fokszáma páros vagy két páratlan fokszámú van, a többi páros.

4) Adjacencia és incidencia mátrix segítségével tárolt gráf esetében is döntsük el, hogy a gráf összefüggő-e.

5) Írjunk algoritmust, amely egy csúcsmátrixot élmátrixszá alakít vagy fordítva.

6) Döntsük el két gráfról, hogy azonos számú csúcsuk van-e, és a megfelelő (azonos) csúcsok között van-e mindkét gráfban él!

7) Egy munkafolyamat több részből áll (mindegyikük különböző időt igényel) és ismerjük, hogy egyes részeket mely részek elvégzése után lehet megkezdeni. Számítsuk ki, hogy mennyi idő alatt lehet elvégezni a munkát.

8) Adott a síkon N darab piros és ugyanannyi kék pont. Párosítsuk össze a piros és a kék pontokat egymással úgy, hogy a párokban lévő pontok távolságainak az összege minimális legyen.

9) Adott N személyről az A[N,N] mátrix. A[i,j]=1 ha i és j ismerik egymást egyébként 0. Döntsük el, hogy összeállítható-e a társaságban egy olyan négy főből álló társaság, amelyiknek tagjai nem ismerik egymást.

10) Egy éhes egérnek egy labirintusban elhelyeznek egy darab sajtot. Írjunk programot, amely segít az egérnek megkeresni a sajthoz vezető utat.

11) Egy éhes egérnek egy labirintusban elhelyeznek egy darab sajtot. Írjunk programot, amely segít az egérnek megkeresni a sajthoz vezető legrövidebb utat.

V. NUMERIKUS ÉS STATISZTIKAI MÓDSZEREK

LINEÁRIS EGYENLETRENDSZEREK MEGOLDÁSA

Számos probléma megoldása visszavezethető lineáris egyenletrendszerek megoldására. Direkt és iteratív módszereket alkalmazhatunk. A következőkben a lineáris egyenletrendszerek megoldásának legfontosabb módszereit ismerjük meg.

Egy üzemben 4 gép 4 fajta terméket állít elő egy adott idő alatt. Az első gép az első termékből x_1-et, a másodikból x_2-t, a harmadikból 2-szer többet mint a második gép, a negyedikből x_4 darabot állít elő, összesen 300 db-ot. A második gép az első termékből 2-szer a másodikból 3-szor a negyedikből fele annyit állít elő mint az első gép, összesen 430-at. A 3. gép az első termékből a második gép által előállított mennyiség 2-szeresét, a 2. termékből 4 -szer annyit, a 3-ból 3-szor annyit, mint az első gép, a negyedikből pedig 4-szer annyit, mint az első gép, összesen 520-at. A 4. gép adatai: 1. termékből a harmadik gép 1,5-szeresét, a 2.termékből az első gép által előállított termék 3-szorosát, a 3. termékből a második gép 4 szeresét, a negyedikből a 2. gép 8-szorosát, összesen 520-at. Mennyit állítottak elő az egyes gépek az egyes termékekből?

A feladatot természetesen le lehet írni a matematika jelölésrendszerével is, és ez talán most érthetőbb is, mint a hétköznapi szöveges leírás. Jelöljük x_3-mal a második gép által előállított mennyiséget. Ekkor

az első gép által előállított mennyiséget az egyes termékekből a következő egyenlet írja le:

$$x_1 + x_2 + 2x_3 + x_4 = 300$$

A második, harmadik és a negyedik gép esetében:

$$2x_1 + 3x_2 + x_3 + x_4/2 = 430$$
$$4x_1 + 4x_2 + 6x_3 + 4x_4 = 560$$
$$6x_1 + 3x_2 + 4x_3 + 4x_4 = 520$$

Így egy 4 ismeretlenből és 4 egyenletből álló elsőfokú egyenletrendszert kapunk, amelyet meg kellene oldani. Ez nem is olyan nehéz, de olyan módszerrel kellene megoldani, amelyet akkor is könnyen lehet használni, ha pl. 20 ismeretlen és 20 egyenlet van vagy mondjuk n ismeretlen és n egyenlet.

A lineáris egyenletrendszer általános felírásban a következő:

$$a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n = y_1$$
$$a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n = y_2$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$a_{n1} x_1 + a_{n2} x_2 + \ldots + a_{nn} x_n = y_n$$

, ahol az a-val jelölt számok az együtthatók. Ugyanez mátrix írásmódban:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots a_{1n} \\ a_{21} & a_{22} & \ldots a_{2n} \\ \ldots & & \\ & & \\ \ldots & & \\ a_{n1} & a_{n2} & \ldots a_{nn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \\ \\ \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \\ \\ \\ y_n \end{bmatrix}$$

,azaz

$$Ax = y,$$

ahol A n*n-es mátrix, x és y n elemű vektorok.

A feladat az x vektor meghatározása. Feltételezzük, hogy minden ismeretlennek legalább az egyik együtthatója nem nulla és az egyenlet determinánsa (lásd. később) nem nulla. Ilyenkor az egyenletrendszernek egyetlen megoldása van:

$x_1 \ldots x_n$

Kivételesen az egyenletrendszernek a determinánsa nulla is lehet, ilyenkor gyakrabban nincs megoldása, ritkábban pedig végtelen sok megoldása van. Ebben az esetben egy vagy több ismeretlen értéke tetszőlegesen választható, a többi ismeretlen pedig ezek lineáris függvénye.

A lineáris egyenletrendszerek megoldása különböző módszerekkel történhet:

- kiküszöbölési eljárás,
- fokozatos közelítés módszere

Az y jobboldali vektort helyezzük el az A mátrix n+1-edik oszlopába. Ha az A mátrixot egy oszloppal kiterjesztjük, könnyebb az algoritmus kidolgozása

$a_{i,n+1} = y_i$

Visszaadja az egyenletet a

$$\sum_{j=1}^{n} a_{i,j} x_j = a_{i,n+1} \quad (i= 1,2, \ldots ,n)$$

Gauss-féle kiküszöbölési eljárás

Oldjuk meg az előbbi feladatunkat az egyik legelterjedtebb módszer, a Gauss elimináció segítségével. A lineáris egyenletrendszer sorozatos - ekvivalens - átalakításokkal felső háromszög mátrixú egyenletrendszerré alakítjuk, melyből sorozatos visszahelyettesítéssel a megoldásvektor elemeit közvetlenül megkaphatjuk:

$$\begin{array}{llll}
a'_{11} x_1 + & a'_{12} x_2 + & \ldots + a'_{1n} x_n & = a'_{1,n+1} \\
0 & a'_{22} x_2 + & \ldots + a'_{2n} x_n & = a'_{2,n+1} \\
0 & 0 & \ldots & \\
0 & 0 & + a'_{nn} x_n & = a'_{1,n+1} \ldots
\end{array}$$

Ha sikerül ilyen formára transzformálni, akkor az alsó egyenlet egyismeretlenesre redukálódik, sorozatos visszahelyettesítéssel megoldható az egyenletrendszer. A jobboldali vektor is transzformálódik. Ha egy egyenletrendszert több jobb oldallal kell megoldani, akkor mindent újra kell számolni.

a) Gauss elimináció

Az A n-edrendű szimmetrikus mátrixot n-1 iterációs lépésnek vetjük alá. Legyen k az iterációs index, i és j a mátrix elem indexe. Az iteráció képlete:

$$a^{(k)}_{ij} = a^{(k-1)}_{ij} - (a^{(k-1)}_{ik} / a^{(k-1)}_{kk}) \, a^{(k-1)}_{kj}$$

k=1,2, ... ,n-1
i=k+1, ... , n
j=k, ... , n+1

Nézzük meg az egyenletrendszerünkön az iteráció első lépését a képlet szerint:

$x_1 + x_2 + 2x_3 + x_4 = 300$
$2x_1 + 3x_2 + x_3 + x_4/2 = 430$
$4x_1 + 4x_2 + 6x_3 + 4x_4 = 560$
$6x_1 + 3x_2 + 4x_3 + 4x_4 = 520$

az eredeti egyenlet

$x_1 + x_2 + 2x_3 + x_4 = 300$
$0x_1 + 1x_2 - 3x_3 - 3/2x_4 = -170$ $a_{21}=2-(2/1)*1$, $a_{22}=3-(2/1)*1$, $a_{23}=1-(2/1)*2$, $a_{24}=1/2-(2/1)*1$, $430-600$
$0x_1 - 0x_2 - 2x_3 + 0x_4 = -640$ $a_{31}=4-(4/1)*1$, $a_{32}=4-(4/1)*1$, $a_{33}=6-(4/1)*2$, $a_{34}=4-(4/1)*1$, $560-1200$
$0x_1 - 3x_2 - 8x_3 - 2x_4 = -1280$ $a_{41}=6-(6/1)*1$, $a_{42}=3-(6/1)*1$, $a_{43}=4-(6/1)*2$, $a_{44}=4-(6/1)*1$, $520-1800$

az első lépés után.

A 2.lépés után:

$x_1 + x_2 + 2x_3 + x_4 = 300$
$0x_1 + 1x_2 - 3x_3 - 3/2x_4 = -170$
$0x_1 - 0x_2 - 2x_3 + 0x_4 = -640$
$0x_1 - 0x_2 - 17x_3 - 6.5x_4 = -1790$

majd a 3. lépés után:

$x_1 + x_2 + 2x_3 + x_4 = 300$
$0x_1 + 1x_2 - 3x_3 - 3/2x_4 = -170$
$0x_1 - 0x_2 - 2x_3 + 0x_4 = -640$
$0x_1 - 0x_2 - 0x_3 - 6.5x_4 = 3650$

Látható, hogy csak az első egyenletben maradt x_1-es tag. Általában a 2. iteráció során csak a második és az első egyenletben lesz x_2-es tag, a harmadik iteráció során csak a harmadikban, a másodikban és az elsőben lesz x_3-as tag és így tovább, de most már a 2. lépésben kiesett egy x_2-es tag és egy x_4-es tag is.

Most már csak vissza kell helyettesítenünk, így kapjuk megoldásként:

$x_4=3650/(-6.5)=-561.53$
$x_3=-640/2=320$
$x_2=-170+3*320-1.5*561.53=-52.3$
$x_1=300+52.3-320*2+561.53=-273.83$

,azaz az x vektor elemeit.

Általános esetben az iteráció n-1 lépése után megkapjuk a háromszög mátrixot

$$
\begin{aligned}
a_{11} x_1 + a_{12} x_2 + \ldots \quad & + a_{1n} x_n = a_{1,n+1} \\
a^{(1)}_{22} x_2 + \ldots \quad & + a^{(1)}_{2n} x_n = a^{(1)}_{2,n+1} \\
\ldots & \\
\ldots & \\
& a^{(n-1)}_{nn} x_n = a^{(n-1)}_{n,n+1}
\end{aligned}
$$

Az átrendezés után a főátló alatti elemek nullák lesznek. Visszahelyettesítéssel az utolsó egyenletből kell kiindulni. Baj van akkor, ha a főátlóbeli elem 0 vagy nagyon kicsi szám.

A gyökök meghatározását az n-edik gyök kiszámításával kezdjük

$$x_n = a^{(n-1)}_{n,n+1} \,/\, a^{(n-1)}_{nn}$$

Majd az ismert gyökök visszahelyettesítésével kapjuk meg az egyenletrendszer összes gyökét:

$$x_{n-1} = 1/\ a^{(n-2)}_{n-1,n-1} \left[a^{(n-2)}_{n-1,n-2} - a^{(n-2)}_{n-1,n} \right]$$

Ezek után nézzük meg az algoritmust, amely a példabeli számokra készült:

```
Program Gauss
n:=4
a[1,1]:= 1 ; a[2,1]:= 2 ; a[3,1]:= 4 ; a[4,1]:= 6
a[1,2]:= 1 ; a[2,2]:= 3 ; a[3,2]:= 4 ; a[4,2]:= 3
a[1,3]:= 2 ; a[2,3]:= 1 ; a[3,3]:= 6 ; a[4,3]:= 4
a[1,4]:= 1 ; a[2,4]:= 1/2; a[3,4]:=4 ; a[4,4]:= 4
a[1,5]:= 300 ;a[2,5]:=430 ;a[3,5]:=560; a[4,5]:=520 **az a[i,n+1]-ben tárol-
                                                        juk az y_i-t**
Ciklus k:=1-től n-1-ig
   Ciklus i:=k+1-től n -ig
      Szorzo:=a[i,k]/a[k,k] **itt lehetne esetleg 0-val való osztás**
      Ciklus j:=k-től n+1-ig
         a[i,j]:=a[i,j]-a[k,j]*Szorzo
      Ciklus vége
   Ciklus vége
Ciklus vége
ered[n]:=a[n,n+1]/a[n,n] **először az x_n-et számolja ki**
Ciklus i:=n-1-től 1-ig -1 esével
   Szorzo:=0
   Ciklus j:=n-től i+1-ig -1 esével
```

```
        Szorzo:=szorzo+ered[j]*a[i,j]
      Ciklus vége
      ered[i]:=(a[i,n+1]-szorzo)/a[i,i]
   Ciklus vége
   Ciklus i:=n-től 1-ig -1 esével
      Ki: ('x', i, ': ',ered[i])
   Ciklus vége
   Program vége
```

b) A Gauss-Jordan algoritmus

A Gauss-elimináció végeredménye egy felső háromszög alakú rendszer. Ha ezek után a kapott a' mátrixban a második sor a'_{12} / a'_{22}-szorosát kivonjuk az első sorból, akkor az (1,2) pozícióban nullát kapunk - míg a főátló alatti nullákat nem érintjük. Ezután a harmadik sor segítségével az (1,3) és (2,3) pozíciókban állítunk elő nullákat és így tovább (azaz a Gauss eliminációt folytatjuk tovább, de ezúttal a felső háromszögben is nullákat szeretnénk előállítani). Ez a Gauss-Jordan algoritmus. Végeredménye egy diagonális mátrix a bal oldalon (a főátlóban vannak csak 0-tól különböző elemek), egy vektor a jobboldalon, azaz a mátrix n+1. oszlopában. A megoldást a vektor elemeinek és a diagonálisban lévő számok hányadosa adja, azaz nincs visszahelyettesítés.

Feladat: Készítsük el az algoritmust!

A DETERMINÁNS

Mielőtt „nekiesnénk" egy négyzetes mátrixú egyenletrendszernek, célszerű volna megvizsgálni, hogy létezik-e egyáltalán megoldás. Egy tétel szerint az ilyen típusú egyenletrendszer egyértelmű megoldásának szükséges és elégséges feltétele, hogy mátrixának a determinánsa ne legyen 0.

Általában valamilyen a_{11}, a_{12}, a_{21}, a_{22} elemekből alkotott

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

alakú kifejezést másodrendű determinánsnak nevezzük, amelyeknek értéke $a_{11}a_{22} - a_{12}a_{21}$,amit úgy kapunk meg, hogy a főátló két végén álló elemek szorzatából

kivonjuk a mellékátló két végén álló elemek szorzatát. Az elsőfokú kétismeretlenes egyenletrendszer megoldása a másodrendű determinánssal így írható fel:

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}}; \quad x_2 = \frac{\begin{vmatrix} a_{11} & b_1 \\ a_{21} & b_2 \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}}$$

feltéve, hogy a közös nevező :

$$D = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \neq 0$$

A D determinánst, amely az egyenletrendszer bal oldalán álló együtthatókat természetes elrendezésben tartalmazza, az egyenletrendszer determinánsának szokás nevezni.

Vegyük észre, hogy a számlálókban álló determinánsok úgy származtathatók az egyenletrendszer determinánsából, hogy annak az ismeretlennek az együtthatói helyébe, amelyet éppen ki akarunk számítani, az egyenlet jobb oldalán álló számokat helyettesítjük. Jelölje az x_1 számlálójában álló determinánst D_1 (ennek első oszlopában vannak az egyenletek jobb oldalán álló számok), az x_2 számlálójában álló determinánst pedig D_2 (ennek második oszlopában vannak az egyenletrendszer jobb oldalán álló számok). Ha $D \neq 0$, akkor az egyenletrendszer megoldása:

$$x_1 = D_1/D; \; x_2 = D_2/D.$$

Az elsőfokú egyenletrendszerek megoldásának ezt a módját Cramer-szabálynak nevezzük. Természetesen léteznek harmad-, negyed-,... n-edrendű determinánsok is, értékük meghatározása azonban nem olyan egyszerű, mint a másodrendű determinánsoké. Harmadrendű determináns kiszámítására létezik az ún. Sarrus-szabály.

Harmadrendű determinánst nagyon egyszerűen számolhatunk a Sarrus-szabály segítségével, amely visszavezeti a számítást másodrendű determinánsok kiszámítására:

A

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

harmadrendű determináns mellé leírjuk még egyszer az első és a második oszlopot:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{matrix}$$

A determináns értékét úgy kaphatjuk meg, hogy a főátló irányában összeköthető ellenhármasok szorzatösszegéből kivonjuk a mellékátló irányában összeköthető ellenhármasok szorzatösszegét. Ekkor ugyanis

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} =$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} -$$
$$-(a_{13}a_{22}a_{31} + a_{11}a_{23}a_{32} + a_{12}a_{21}a_{33},$$

és ez a sorrendtől eltekintve megegyezik a kifejtéssel kapott eredménnyel.

Magasabbrendű determinánsok értékének kiszámítására már nem létezik ilyen egyszerű kiszámítási módszer.

N -edrendű determinánsnak nevezzük az n^2 elemből álló, n sort és n oszlopot tartalmazó, következő alakú táblázatot:

$$\begin{vmatrix} a_{11} & a_{12} & \dots\ a_{1n} \\ a_{21} & a_{22} & \dots\ a_{2n} \\ \dots & & \\ \dots & & \\ a_{n1} & a_{n2} & \dots\ a_{nn} \end{vmatrix}$$

Az n-edrendű determinánsnak a következő értéket tulajdonítjuk:

$$\begin{vmatrix} a_{11} & a_{12} & \dots\ a_{1n} \\ a_{21} & a_{22} & \dots\ a_{2n} \\ \dots & & \\ \dots & & \\ a_{n1} & a_{n2} & \dots\ a_{nn} \end{vmatrix} =a_{11}A_{11}+a_{12}A_{12}+\dots+a_{1n}A_{1n}=\sum_{i=1}^{n} a_{1i}A_{1i}$$

ahol A_{1i} az a_{1i} elemhez tartozó (n-1)-edrendü aldeterminánst jelenti, amelyet úgy kapunk meg, hogy az első sor és az i-edik oszlop elhagyásával adódó (n-1)-edrendű

determinánst $(-1)^{1+i}$-vel megszorozzuk. A determináns értékének ez a meghatározási módja a determináns első sora szerinti kifejtése. (kifejthető a determináns teljesen hasonló módon bármely sora vagy bármely oszlopa szerint is). Ha a fenti kifejezésben szereplő A_{11}, A_{12},..., A_{1n} (n-1)-edrendű aldeterminánsokat a közölt módon (n-2)-edrendű aldeterminánsaikkal, ezeket (n-3)-adrendű aldeterminánsaikkal fejezzük ki, és ezt az eljárást tovább folytatjuk, végül másodrendű aldeterminánsokhoz jutunk. Értéküket kiszámítva, mindegyik másodrendű determinánsból két-két szorzatot kapunk, így az n-edrendű determináns kifejtése során kapott n-tényezős szorzatok száma n!

Ezek az n-tényezős szorzatok közvetlenül is felírhatók, és ez a felírás adja az n-edrendű determináns - előbbivel egyenértékű - másik definícióját:

$$\sum_{S}(-1)^K a_{1k1}\, a_{2k2},\ldots\, a_{nkN}$$

ahol $k1$, $k2$,..., kN az 1,2,...,n oszlopindexek egy sorrendjét jelenti; K e sorrend inverzióinak a számát jelöli; az összegzés pedig az 1,2,..,n elemek minden lehetséges sorrendjének terjesztendő ki. A tagok felírására azonban N>3 esetében nincs a Sarrus-szabályhoz hasonló egyszerű módszer.

Az előző példában szereplő mátrix determinánsa:

$$\begin{vmatrix} 1 & 1 & 2 & 1 \\ 2 & 3 & 1 & 1/2 \\ 4 & 4 & 6 & 1/2 \\ 6 & 3 & 4 & 4 \end{vmatrix}$$

Ez kifejtve az első sora szerint:

$$1* \begin{vmatrix} 3 & 1 & 1/2 \\ 4 & 6 & 1/2 \\ 3 & 4 & 4 \end{vmatrix} - 1* \begin{vmatrix} 2 & 1 & 1/2 \\ 4 & 6 & 1/2 \\ 6 & 4 & 4 \end{vmatrix} +2* \begin{vmatrix} 2 & 3 & 1/2 \\ 4 & 4 & 1/2 \\ 6 & 3 & 4 \end{vmatrix} -1* \begin{vmatrix} 2 & 3 & 1 \\ 4 & 4 & 6 \\ 6 & 3 & 4 \end{vmatrix}$$

Természetesen a 3-adrendű determinánsokat is ki lehetne fejteni, de a Sarus-szabállyal egyszerűbb az értékek meghatározása:

$1*19 - 1*14 + 2*26 - 1*44 = 13$

Mindkét definícióból látszik, hogy egy n-edrendű determináns kiszámítása már viszonylag kicsi n esetén is elég hosszadalmas és fárasztó feladat. Az alábbiakban egy n-edrendű determináns kiszámítására használható algoritmust kerül megadásra, amely

a kifejtés szerint egészen a harmadrendű aldeterminánsokig "megy le" és ezekre alkalmazza a Sarrus-szabályt.

FüggvényEljárás Számít(m,n) ***m mátrix, n egész típusú***

Ha n=3 **akkor** ***kiszámítja a determinánst a Sarrus szabály segítségével***

Szám:=m[1,1]*m[2,2]*m[3,3]+m[2,1]*m[3,2]*m[1,3]+

m[3,1]*m[1,2]*m[2,3] m[3,1]*m[2,2]*m[1,3] -

m[2,1]*m[1,2]*m[3,3] - m[1,1]*m[3,2]*m[2,3]

Egyébként

Érték:=0

Ciklus i:=1-**től** n-**ig**

Ciklus j:=1-**től** i-1 -**ig**

Ciklus k:=2-**től** n-**ig**

Uj[j,k-1]:=m[j,k] ***az Uj-ba kerül a k-1-ed aldetermináns***

Ciklus vége

Ciklus vége

Ciklus j:=i+1 -**től** n-**ig**

Ciklus k:=2-**től** n-**ig**

Uj[j-1,k-1]:=m[j,k]

Ciklus vége

Ciklus vége ***maradékképzés***

Ha i mod 2=0 **akkor** Értek:=Ertek-m[i,1]*Számít(Uj,n-1)**Egyébként**

Értek:=Értek+m[i,1]*Számít(Uj,n-1)

Ciklus vége

Szám:=Ertek

Elágazás vége

Eljárás vége (Visszaad:Szám)

Program Sarrus

Be: n (>2, egész) ***a mátrix nagysága***

Be: M ***az m mátrix beolvasása***

Ki:Számít(M,n) ***Mátrix típus és a méret átadása***

Program vége

MÁTRIXINVERTÁLÁS

Ha az A mátrixnak B mátrix az inverze, akkor a

AB=E

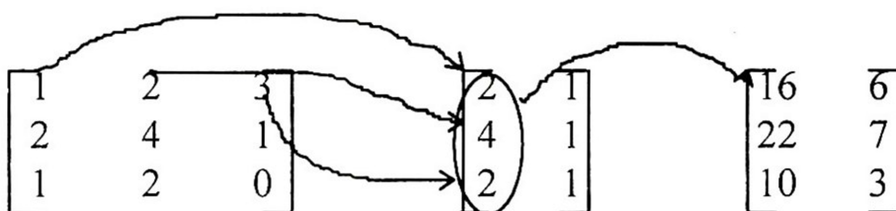
szorzat egységmátrixot kell, hogy adjon, ahol az E egységmátrix

$$E = \begin{bmatrix} 1\ 0 \ldots 0\ 0 \\ 0\ 1 \ldots 0\ 0 \\ . \\ 0\ 0 \ldots 1\ 0 \\ 0\ 0 \ldots 0\ 1 \end{bmatrix}$$

Mátrixot a következőképpen tudunk szorozni, ha (Kompozíciós szorzás) adott az A_{nm} és a B_{mr} mátrixok, elemeik a_{ij}, b_{kl} (i=1,2,...,n ; j=1,2,..m ; k=1,2,..m ; l=1,2,.. r), akkor a

$$C_{n,r} = A*B = \sum_{i=1}^{n} \sum_{l=1}^{r} \sum_{j=1}^{m} a_{ij} * b_{jl}$$

a két mátrix kompozíciós szorzata. Pl.: *az összeg*



Feladat:

A képlet segítségével írjuk meg két mátrix szorzását végrehajtó algoritmust.

Egy mátrix inverzét a következőképpen lehet meghatározni. Írjuk a mátrix mellé a megfelelő egységmátrixot. Hajtsuk végre a mátrixon a Gauss-Jordan algoritmust, de a műveleteket végezzük el az egységmátrixon is. Amikor már csak az eredeti mátrix diagonálisában szerepelnek 0-tól különböző értékek (a Gauss-Jordan algoritmus befejeződik), akkor már csak a mátrix minden sorát el kell osztani a diagonálisában szereplő számmal, így az eredeti mátrix helyén az egységmátrot kapjuk. Az algoritmus eredményeképpen viszont az eredeti mátrix mellé írt egységmátrix helyén megkapjuk a keresett inverz mátrixot.

A példában szereplő mátrix esetében:

1	1	2	1	1	0	0	0
2	3	1	1/2	0	1	0	0
4	4	6	1/2	0	0	1	0
6	3	4	4	0	0	0	1

az iteráció első lépése:

1	1	2	1	1	0	0	0
0	1	- 3	3/2	- 2	1	0	0
0	0	2	0	- 4	0	1	0
0	- 3	- 8	- 2	- 6	0	0	1

itt keletkezik az inverz

és természetesen folytatva az algoritmust megkaphatjuk az invertált mátrixot.

Feladat: Írjuk meg a mátrixinvertáló algoritmust és ellenőriztessük le az eredmény!

INTERPOLÁCIÓ

A numerikus számítások során gyakori, hogy valamely f(x) függvényről csupán korlátozott mennyiségű információnk van. Ismerjük a függvényt több, de véges számú független változó értékére vonatkozóan, s ebből - legalább közelítőleg - elő kell állítanunk egyes további x-ekhez tartozó függvényértékeket.

Az ismert függvénypontok közötti függvényértékek közelítő meghatározását nevezzük interpolációnak.

A függvények grafikus ábrázolása olyan terület, ahol lépten- nyomon találkozhatunk az interpoláció szükségességével. Ha néhány pontjával megadott függvényt kell folytonos görbével ábrázolnunk, a függvény ismeretlen pontjait mindig valamilyen interpolációs módszerrel határozzuk meg.

Feltételezzük, hogy az f(x) függvény értékét csupán az

$$x=a_0, a_1, \dots , a_n$$

pontokban az ún. interpolációs alappontokban ismerjük. Az f(x) függvényt az $[a_0;a_n]$ intervallumban az y(x) interpolációs függvénnyel helyettesítjük. Arra törekszünk, hogy a tényleges és a közelítő függvény különbsége

$$c(c) = f(x) - y(x) ,$$

vagyis az $\varepsilon(x)$ hiba az interpolációs alappontokban eltűnjön, közöttük pedig előírtan kicsi maradjon.

Lineáris interpoláció

Ismert az x(n) abszcissza és y(n) ordináta. Lineáris interpolációval keressük az x_1-hez tartozó y_1 értékét.

A lineáris interpoláció menete:

- megkeressük az intervallumot, amely közé esik az x_1,
- az intervallum két pontján átfektetünk egy egyenest

$$y_1 = y_1 + (y_{i+1} - y_1) / (x_{i+1} - x_i) \left[x_1 - x_i\right]$$

Az eljárás hibajelzést adjon, ha az x_1 a tartományon kívül esik.

Lineáris interpoláció:

```
FüggvényEljárás lin(x,x1,y1,x2,y2) **valósak**
Függvény vége (visszaad: y1+(y2-y1)/(x2-x1)*(x-x1)

Eljárás beolvas (n,x,y)       **n egész, x,y két tömb**

Be: n  (n<=50, egész)         **interpolációs alappontok bekérése**
Ciklus i:=1-től n-ig
   Be: x[i],y[i]  **nem figyeli, ha ugyanazon x-hez különböző y-t adnak
                                                              meg*
Ciklus vége
Rendez(n,x,y)          **lerendezi x és y n elemű tömböket növekvő sorrendbe**
Eljárás vége

Függvény gyök (x,y,n,xk) **x,y vektorok, n egész, xk valós**
jo:=Hamis
Ciklus i:=1-től n-1-ig
   Ha x[i]<=xk és x[i+1]>=xk akkor
      jo:=Igaz
      hely:=i
   Elágazás vége
Ciklus vége
Ha jo akkor
   gyok:=lin(xk,x[hely],y[hely],x[hely+1],y[hely+1])
Egyébként
   Ki: "az alappontokkal nem interpolálható"
   gyok:=0
Elágazás vége
Eljárás vége (Visszaad: gyok)
```

```
Program Interpolál
Beolvas(n,x,y)
Be: xk,xv,dx
Ha xk<x[1] akkor xk:=x[1]
Ha xv>x[n] akkor xv:=x[n]
u:=xk
Ki: "Lineárisan interpolált értékek:"
Ciklus
    yk:=gyok(x,y,n,u)
    Ki: u,yk
    u:=u+dx
 amíg u>xv
 Program vége
```

EGYISMERETLENES NEMLINEÁRIS EGYENLET

Tegyük fel, hogy az

f(x)

függvénynek egy adott tartományban egy vagy több zérushelye van. A zérushely meghatározására szolgáló algoritmusok indításához rendszerint ismerni kell - a módszertől függően - egy vagy több, a zérushely (gyök) környezetében levő x értékeket. A következőkben általában valós gyökök meghatározásával foglalkozunk.

A módszerek alkalmazásához ismerni kell a keresendő gyök intervallumát ill. tartományát. Ilyen egyetlen gyököt tartalmazó intervallum vagy tartomány meghatározását a gyök elkülönítésének nevezzük.

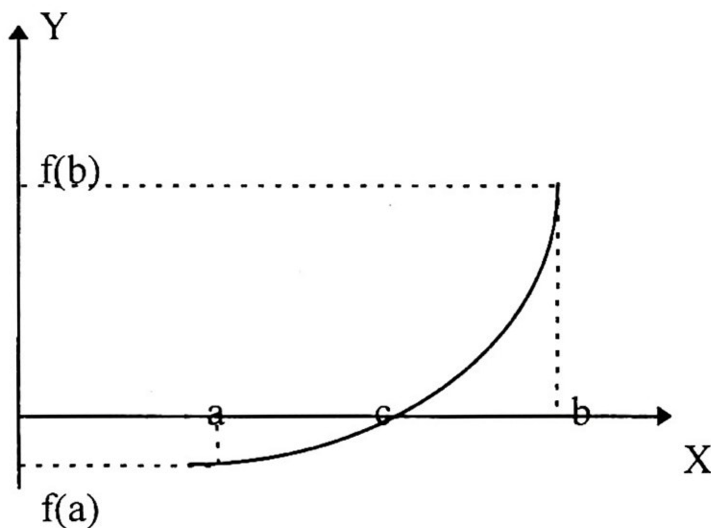
Végezhetjük grafikusan is! Az egyenlet valós gyökeinek számáról és elhelyezkedéséről tájékozódhatunk, ha az f(x) függvény görbéjét felvázoljuk. Az x tengellyel való metszéspontok és érintkezési pontok szolgáltatják az egyenlet közelítő gyökeit.

Gyakran az y=f(x) függvény nehezen ábrázolható, de f(x) felbontható f(x)=u(x)-v(x) alakban, és y_1 = u(x), valamint y_2 = v(x) egyszerűbben ábrázolható. Ilyenkor a gyökök a görbék metszéspontjai.

Az egyismeretlenes nemlineáris egyenlet megoldásának módszerei:

- behatárolás intervallumfelezéssel
- húr módszer
- fokozatos közelítés (szukcesszív approximáció)

Gyökbehatárolás intervallum felezéssel



Az intervallumfelezéses módszer lényege: ha tudjuk, hogy egy függvény egy [a;b] intervallumban folytonos és az intervallum két végpontjában a függvényértékek ellentétes előjelűek, akkor a megoldást az intervallum közepén keressük, azaz a következő tippünk:

x:= (a+b) / 2

Ha itt f(x)<0, *akkor* x *lesz az intervallum új felső határa, ha* f(x)>0 akkor pedig x az új alsó határ. A határok ilyen módosítását, azaz az intervallum szűkítését addig kell folytatni, amíg az [a;b] intervallum hossza kisebbé nem válik, mint a megengedett hibahatár.

Mivel minden lépés az előző hossz felére csökkenti az intervallum hosszát, k lépés után az intervallum h hossza a kezdeti b-a értékről h= |b-a| / $(2)^k$-adikon értékre csökken. Kérdés: mi van akkor, ha több gyök is van? (sajnos erre a módszer abszolút érzéketlen). Ha az intervallumot felezzük, és

f[(a+b) /2] = 0 akkor x = (a+b) /2

az egyenlet gyöke.

Ha f[(a+b) / 2] $\neq$ 0 akkor c = (a+b) /2

Ha sgn[9](f(c)) megegyezik az sgn f(b) előjelével, akkor b:=c különbı ı a:=c.

[9] Sgn() = előjelfüggvény

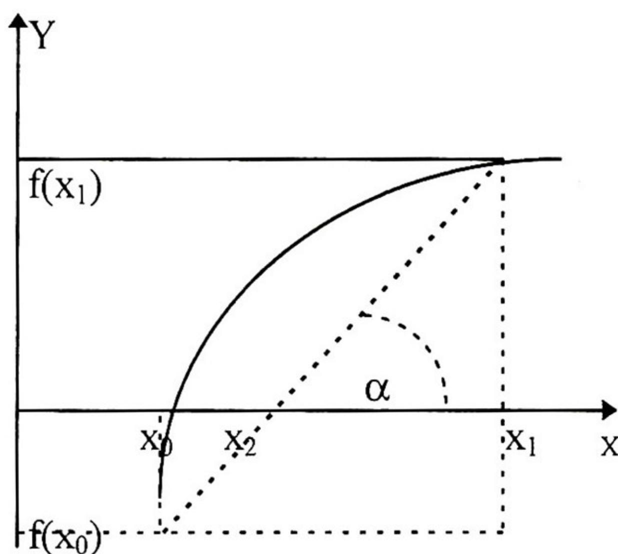
A leállási feltétel

$|a-b| < \varepsilon$

A konvergencia nagyon rossz! A gyök közelítő behatárolása jó, utána érdemes pl. Newton - Raphson módszert alkalmazni.

Húr módszer

Előnyösebbek azok az algoritmusok, melyeknek konvergenciája kellőképpen gyors, azonban számolási igényük kevesebb. A legjellegzetesebb ilyen interpolációs módszer a húr módszer, amely szintén két alappontra támaszkodik, és az alappontok között a függvényt egyenessel helyettesítjük.



$f(x_0)f(x_1) < 0$

Írjuk fel a húr meredekségét kétfajta módon:

$\text{tg}\,\alpha = (f(x_1)-f(x_0)) / (x_1-x_0)$
$\text{tg}\,\alpha = f(x_1) / (x_1-x_2)$

Egyenlővé téve a két egyenletet

$$\frac{f(x_1)-f(x_0)}{x_1-x_0} = \frac{f(x_1)}{x_1-x_2}$$

Legyen $y_0 = f(x_0)$, $y_1 = f(x_1)$

akkor x_2 közelítő gyök értéke

$x_2=x_1 - y_1 ((x_1-x_0) / (y_1-y_0))$

Leállási feltétel

$|x_2-x_1| < \varepsilon$

Az intervallummódosítás $f(x_2)$ előjele szerint történik, ha

sgn $f(x_2) > 0$ $x_1:=x_2$ $y_1:=y_2$

különben

sgn $f(x_2) < 0$. $x_0:=x_2$ $y_0:=y_2$

Egyismeretlenes nemlineáris egyenlet megoldása az ismertetésre került módszerekkel

```
FüggvényEljárás Fv(x:) **x valós, ennek a függvénynek a gyökét keressük*
f:=x*x*x+x-0.7
Eljárás vége (Visszaad: f)

Függvény Közel(a,b,hibahatár,r) **a,b,hibahatár valósak, r logikai**
Ha fv(a)*fv(b)<0 akkor
    q:=a
    w:=b
    Ciklus
        xe:=(q+w)/2
        Ha fv(xe)*fv(q)>0 akkor
            q:=xe
        Egyébként
            w:=xe
        Elágazás vége
    amíg ABS(fv(xe))<hibahatár
    r:=Igaz
    kozel:=xe
Egyébként
    r:=Hamis
    kozel:=0
Elágazás vége
Eljárás vége (Visszaad:kozel)

FüggvényEljárás Húr(a,b,hibahatár,r) **a,b,hibahatár valósak, r logikai**
x0:=a
x1:=b
```

Ha fv(x1)*fv(x0)<0 **akkor**
  Ciklus
    y0:=fv(x0)
    y1:=fv(x1)
    x2:=x1-y1*(x1-x0)/(y1-y0)
    Ha fv(x2)*y1>0 **akkor**
      x1:=x2
    Egyébként
      x0:=x2
    Elágazás vége
  amíg ABS(fv(x2))<hibahatár
  hur:=x2
  r:=Igaz
Egyébként
  r:=**Hamis**
  hur:=0
Elágazás vége
Eljárás vége (Visszaad: hur)

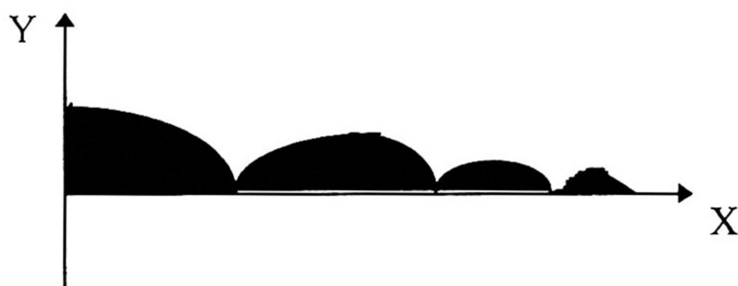
Program Egyenletmegoldás
Ki: 'Alsó-, felsőhatár és a hibahatár?'
Be: a,b,hibahatár ***az alsó- és a felsőhatár, valamint a hibahatár bekérése***
Menükészítés
A módszernek megfelelő függvény meghívása, a,b,hibahatár,r paraméterekkel, az utolsó parm-nél cím szerinti paraméterátadás történik.
Ha r=Igaz akkor Ki: a valós gyök
Program vége

NUMERIKUS INTEGRÁLÁS

Számítsuk ki 0.1% pontossággal az

$$f(x)=\left|\sin(x)/x\right|$$

függvény és az x tengely által bezárt terület nagyságát.



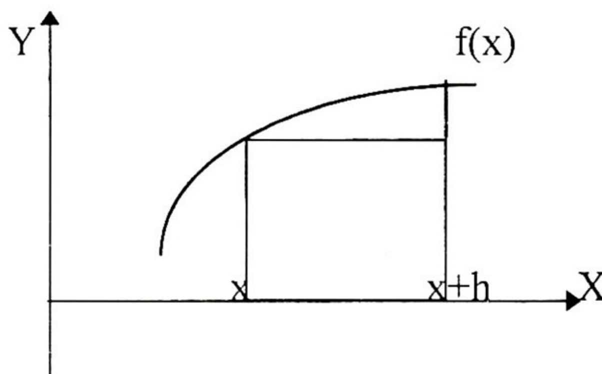
Ezt a függvényt alatti területet közelítőleg úgy tudjuk meghatározni, hogy az integrálás [a;b] tartományát felosztjuk n darab

h= (b-a)/n

hosszúságú szakaszra, egy-egy ilyen kis szakaszon pedig az f(x) függvény alatti területet valamilyen egyszerűbb mértani alakzat területével helyettesítjük. Ez a helyettesítő alakzat lehet téglalap, trapéz, illetve olyan is, amelynek a teteje másodfokú parabola, az oldalai és az x tengellyel érintkező oldala pedig egyenes. Az egyes numerikus integrálási módszereket tehát az határozza meg, hogy milyen alakzattal közelíthetjük az integrálandó függvényt az egyes h hosszúságú szeletekben. Azt is sejthetjük, hogy annál pontosabb lesz a közelítés, minél jobban illeszkedik az alakzat a függvényhez, illetve minél nagyobb n értéket választunk.

A téglalap módszer

Ez a legegyszerűbb, és persze a legpontatlanabb módszer:



Egyetlen n hosszúságú tartományban a közelítő téglalap területét a

t=f(x)*h

kifejezés adja, ha pedig a teljes [a;b] tartományra összeadjuk ezeket a t értékeket, akkor a teljes T területet h kiemelése után így fejezhetjük ki:

T:=h*[f(a) + f(a+h) + ... + f(a+(n-1)h)]

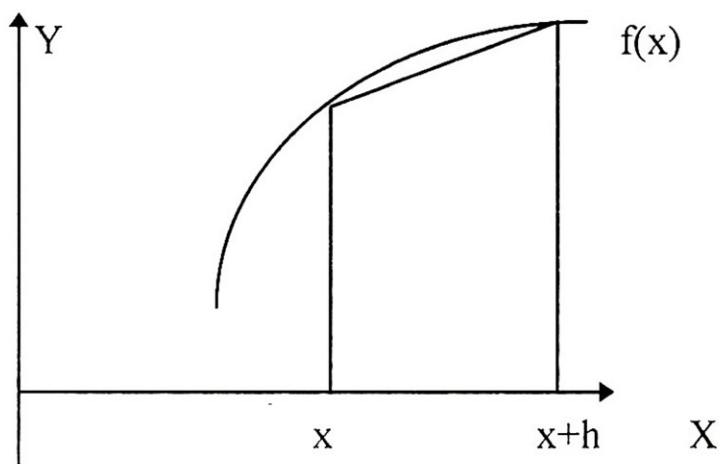
Itt éppen n értékeket kell összegezni: az integrálandó függvény értékét az a, a+2h, a+3h, ... a+(n-1)*h helyeken, majd az így kapott eredményt meg kell szorozni h-val. Így az algoritmus:

Eljárás Téglalap
Be: n
a:=0.1; b:=10
h:=(b-a)/n
t:=0 ; x:=a
k:=0

```
Ciklus
    t:=t+sin(x)/x
    x:=x+h
    k:=k+1
amíg k=n
t:=t*h
Ki: t
Eljárás vége
```

Trapéz módszer

Minden egyes h hosszúságú szakaszon az integrálandó függvény alatti területet olyan trapéz területével közelítjük, amely a szakasz végein illeszkedik a függvényhez:



Egy ilyen trapéz területe:

t= (f(x) + f(x+h)) * h / 2

A teljes T közelítő értéke:

T= h[f(a)/2 + f(a+h)+ f(a+2h) + ... + f(a+(n-1)h) + f(b)/2)]

azaz a két szélső pontban a függvényértékek felét, minden belső pontban pedig a függvényértékeket kell összegezni, majd ezt szorozni h-val.

Az algoritmus:

```
Eljárás trapéz
Be: n              **pontszám beolvasása**
A:=0.1: b:=10      **integrálási határok**
H:=(b-a)/n         **lépéshossz**
```

```
T:=(Sin(a)/a+Sin(b)/b)/2          **kezdeti érték**
X:=A+H
K:=1                  **Ciklusszámláló**
Ciklus
   t:=t+sin(x)/x      **összegzés**
   x:=x+h
   k:=k+1
amíg k=n
Ki: Sin(x)/x integrálja a,b között t*h
Eljárás vége
```

Célszerű lenne, ha az eredmény pontosságának a vizsgálatát a program végezné. Ezt úgy oldhatjuk meg, hogy az algoritmus egy további, az egész közelítést magába foglaló ciklusban duplázza a pontszámot és minden ciklus végén megvizsgálja, mennyit változott az integrálközelítő összeg. Ha a különbség elég kicsi, nem kell tovább folytatni a pontszám növelését.

A program:

```
relatívhiba:=0.001
N:=20                 **pontszám kezdeti értéke**
A:=0.1: b:=10         **integrálási határok**
t:=0
Ciklus
   h:=(b-a)/n         **lépéshossz**
   t:=(Sin(a)/a+Sin(b)/b)/2       **kezdeti érték**
   x:=A+h
   k:=1               **Ciklusszámláló**
   Ciklus
      t:=t+sin(x)/x          **összegzés**
      x:=x+h
      k:=k+1
   amíg k=n
   t:=t*h
   n:=n+n
Amíg ABS((t-et)/t)> 3* relatívhiba
Ki: "Sin(x)/x integrálja a,b között", t*h
```

KORRELÁCIÓ- ÉS REGRESSZIÓSZÁMÍTÁS

Arra fogjuk keresni a választ, hogy két egymással sztochasztikus[10] kapcsolatban lévő események között - amelyeket adatokkal jellemzünk - milyen szoros kapcsolat van (korrelációszámítás), majd megpróbáljuk az egymásra gyakorolt hatásukat számszerűsíteni, e hatások irányát és mértékét meghatározni. (regressziószámítás)

Kiindulóadataink egy táblázatba foglalva a következők:

A vándorlási különbözet é az épített lakások száma dunántúli városokban (1960-1962)

Sorszám	*Város*	*Vándorlási különbözet (fő)* X	*Az épített lakások száma* Y
1.	Győr	3855	1993
2.	Kaposvár	3528	1147
3.	Szekszárd	1534	588
4.	Székesfehérvár	4903	2018
5.	Szombathely	2622	1285
6.	Tatabánya	4040	1458
7.	Veszprém	3078	858
8.	Zalaegerszeg	2363	1142
9.	Ajka	1875	864
10.	Dunaújváros	7220	1867
11.	Esztergom	1537	547
12..	Keszthely	724	205
13.	Komárom	461	385
14.	Komló	925	479
15.	Kőszeg	286	78
16.	Mohács	-73	237
17.	Mosonmagyaróv.	1139	509
18	Nagykanizsa	566	819
19.	Oroszlány	3802	658
20.	Pápa	-320	365
21.	Sopron	2117	430
22.	Tata	635	371
23.	Várpalota	2257	420
	Összesen:	49047	18273
	Átlag	2134	814

A táblázat a 23 dunántúli városban megfigyelt vándorlási különbözetet (X) , továbbá a felépült lakások számát tartalmazza a megadott időszakban. A két mennyiségi ismérv közötti kapcsolat szorosságát mérhetjük a kovarienciával (C), amely a korrelációszámítás egyik mérőszáma és a következőképpen lehet kiszámolni:

[10] statisztikai valószínűségen alapuló

$$C=\frac{\sum d_x d_y}{n}=\frac{\sum xy}{n}-\bar{x}\bar{y},$$

ahol $d_x=(x_i-\bar{x})$ $d_y=(y_i-\bar{y})$, azaz a megfelelő átlagtól vett eltéréseket jelenti. (x_i, y_i az értékpárokat jelenti, i=1,2,...,n)

Ha C=0, akkor az X és Y korrelálatlanok, egyébként a C előjele a kapcsolat irányát mutatja. Felső korlát nincs, de ha C közel van 0-hoz, akkor az laza kapcsolatra mutat.

Az algoritmus, amely a két ismérv közötti kovarienciát kiszámítja és az adatokat egy szöveges file-ból olvassa, a következő:

```
Program Cov
Összrendel(f,"ADAT.INP")
Nyit(f)
i:=0
atlagy:=0;atlagy:=0;gyujt:=0
Ciklus amíg Nem Vége?(f)
    i:=i+1
    Be: x[i],y[i] f-ből
    gyujt:=gyujt+x[i]*y[i]
    atlagx:=atlagx+x[i]
    atlagy:=atlagy+y[i]
Ciklus vége
Ha i<>0 akkor
    Ki:"A kovariencia értéke:",gyujt/i -(atlagx / i)*(atlagy / i)
Egyébként
    Ki: "Nincs adat!"
Elágazás vége
Zár(f) **a file zárása**
Program vége
```

Példánkban egyébként C értéke 844 047, ami szoros kapcsolatra enged következtetni.

A kovariencia sajnos csak a lineáris típusú kapcsolatok szorosságát képes jellemezni, ezért szükségünk van általános jellegű mérőszámra, amely bármely fajtájú korrelációs kapcsolat szorosságának mérésére alkalmas.

Erre szolgál a korrelációs hányados, amelyet a következőképpen lehet kiszámolni:

$K^2=1-SB(y)/S(y)$,

ahol $SB(y) = (\overline{y_{ij}} - y_i)^2$ $S(y)= (y_{ij} - \bar{y})^2$, y_{ij} az épített lakások számát jelenti városonként, $\bar{y}$ ezek számtani, y_i a csoportok számtani átlagát jelenti.

Ugyanis a számításhoz csoportosítjuk a megfigyelt értékeket a tényezőváltozó osztályközei szerint, és ezután számítjuk ki a részátlagokat a következő táblázat szerint:

Vándorlási különbözet	*A város neve*	*A városban épített lakások száma(y_{ij})*	$\bar{y}_i$	$(y_{ij}-\bar{y}_i)^2$	$(y_{ij}-\bar{y})^2$
	Pápa	365		139	201601
	Mohács	237		19544	332929
-600	Kőszeg	78	376,8	89281	541696
	Komárom	385		67	184041
	Nagykanizsa	819		195541	25
	Összesen	1884		304572	-
	Tata	*371*		*19044*	*196249*
	Komló	479		900,0	112225
	Mosonm.r	509		0	93025
601-2000	Szekszárd	588	509,0	6241	51076
	Esztegom	547		1444	71289
	Ajka	864		126025	2500
	Keszthely	205		92416	370881
	Összesen	3563		246070	-
	Sopron	*430*		*315395*	*147456*
	Várpalota	420		326727	155236
	Zalaegerszeg	1142		22620	107584
	Szombathely	1285		86084	221841
2001-4000	Veszprém	858	991,6	17849	1936
	Kaposvár	1147		24149	110889
	Győr	1993		1002802	1390041
	Oroszlány	658		111289	24336
	Összesen	7933		1906915	-
	Székesfehérv.	2018		56169	1449616
4001-	Dunaújváros	1867		7396	1108809
	Tatabánya	1458	1781,0	104329	414736
	Összesen	5343		167894	-
Együtt:		18723	814,0	**2625451**	**7290017**

A táblázat segítségével könnyen kiszámítható a korrelációs hányados értéke(K):

K^2= 1- 2 625 451/7 290 017= 0,6399 amiből K=0,799.

Minél közelebb van K^2 az 1-hez, annál szorosabb a kapcsolat az ismérvek között, így lényegében az előző példában kiszámított korrelációs együtthatóhoz közel álló eredményhez jutottunk, ami nem meglepő, hiszen a szóban forgó kapcsolat közel áll a lineáris típushoz.

Feladat:
Egy szekvenciális file-ban a városok neve és a városban épített lakások száma található. A csoportok végét egy csillag jelzi a file-ban. Készítsük el a K értékét meghatározó algoritmust.

IDŐSOROK VIZSGÁLATA

Valamely jelenség fejlődését, időbeli alakulását számokkal fogjuk jellemezni, azaz egy időintervallum bizonyos pontjaihoz számokat fogunk hozzárendelni. Például megmérjük egy újszülött gyermek testsúlyát minden hét első napján 1 éven át. Ekkor különböző dátumokhoz egy számot rendelünk, így vizsgáljuk a gyermek testsúlyának gyarapodását, amiből következtetéseket vonhatunk le. Valószínűleg a gyermek az év végén nehezebb lesz, mint az év elején (a *trend* növekedést mutat), de lesznek időnként visszaesések (pl. egy gyomorrontás miatt - *véletlen ingadozás*), esetleg télen gyorsabban nő a testsúlya (többet eszik, kevesebbet mozog - *periodikus ingadozás*).

A fejlődés törvényszerűségeinek tanulmányozásakor az idősorok statisztikai elemzésének problémája az egyes komponensek elkülönítése. Az elemzés szempontjából három komponenst különböztetünk meg: 1. alapirányzat vagy trend (az idősorban tartósan érvényesülő tendencia), 2. periodikus ingadozás (rendszeresen ismétlődő idényszerű vagy szezonális ingadozás, pl. nyáron több üdítőital fogy mint máskor), 3. véletlen ingadozás(véletlenek igen sok, egyenként nem jelentős, egymás hatását elősegítő vagy keresztező tényezők végső eredménye, pl. leég egy palackozó üzem, ezért kevés az üdítőital).

Kérdés, hogy az összetevők milyen módon kapcsolódnak egymáshoz. Ha additív módon, akkor használhatjuk a következő képletet:

$y_t = yt_t + S_t + v_t$. (y_t a tényleges adat, yt_t a trend S_t a periodikus adat, v_t a véletlen a t. időpontban.)

Ha az idősor periodicitásra is tekintettel vagyunk, akkor az idősor komponenseinek additív kapcsolatát tükröző alapképlet:

$$y_{ij} = yt_{ij} + S_j + v_{ij},$$

ahol y_{ij} az i-edik periódus (pl. év) j-edik szakaszának (pl. hónap) adata, yt_{ij} az alapirányzat, S_j a periodikus összetevő bármely periódus j-edik szakaszában ugyanazt az értéket adja, v_{ij} pedig a véletlen komponensnek egy megvalósult értéke.

Ha multiplikatív módon kapcsolódnak, akkor a képlet úgy módosul, hogy a komponenseket össze kell szorozni és nem összeadni.

Az elkövetkező két fejezetben a komponensek meghatározásának a módjával foglalkozunk.

Lineáris trend

A lineáris trendfüggvényt a következőképpen lehet meghatározni:

$$yt = b_0 + b_1 t$$

Az ismeretlen b_0 és b_1 kiszámítására ismert a következő két egyenlet:

$$b_0 = \frac{\sum_{t=1}^{n} y_t}{n} \quad \text{és} \quad b_1 = \frac{\sum_{t=1}^{n} t y_t}{\sum_{t=1}^{n} t * t}$$

Ez azonban csak akkor igaz, ha t értékét úgy választjuk meg, hogy $\sum t = 0$. Ehhez csak azt kell tennünk, hogy az idősor közepéhez (t=) 0-t rendelünk, majd t értéke az idősor eleje felé haladva eggyel nő, a vége felé haladva eggyel csökken a következő táblázat szerint:

A csokoládétermelés alakulása

Év	Termelés ezer tonna (y_t)	t	t^2	ty_t
1963	1139	-8	64	-9112
1964	1109	-7	49	-7763
1965	1200	-6	36	-7200
1966	1208	-5	25	-6040
1967	1211	-4	16	-4849
1968	1352	-3	9	-4056
1969	1302	-2	4	-2604
1970	1357	-1	1	-1357
1971	1566	0	0	0
1972	1639	1	1	1639
1973	1563	2	2	3126
1974	1722	3	4	5166
1975	1892	4	8	7568
1976	1777	5	16	8885
1977	1950	6	32	11700
1978	2004	7	64	14028
1979	2051	8		16408
Összesen:	26042	0.	408	15544

A példában b_0=26042/17 = 1531,88, b_1=25544/408 = 62,61, így a trendegyenlet:

$$yt=1531,88 + 62,61t \quad (\sum t = 0)$$

Az alapirányzat szerint a csokoládétermelés évenként 62,61 ezer tonnával növekszik. (ekkora az évenkénti átlagos változás). b_0 az 1971-re eső trendérték.

Ezek után nézzük az algoritmust, amely meghatározza az adatokból a trendegyenletet.

```
Program idősor1
Összerendel(f,"ADAT.INP")
Nyit(f)
n:=0;ossz:=0
Ciklus amíg Nem Vége?(f)
   n:=n+1
   Be: adat[n]-be f-ből
   ossz:=ossz+adat[n]
Ciklus vége
t:=(n DIV 2 ) * (-1)
tnegyzet:=0;tszery:=0
Ciklus cikl:=1 -től n-ig
   tnegyzet:=tnegyzet+t*t
   tszery:=tszery+t*a[cikl]
   t:=t-1
Ciklus vége
Ki:: "yt=",ossz, "+", tszery/tnegyzet, "t"
Zár(f)
Program vége
```

Exponenciális trend

A trendfüggvény a következő:

$Yt=ab^t$, logaritmizált változatban $\log Yt = \log a + t \log b = b_0 + b_1 t$.

b_0 és b_1 kiszámítása a lineáris trendnél megadottak szerint történik, csak yt helyett log yt szerepel a képletben.

Feladat: Írjuk meg az exponenciális trendet megadó algoritmust, de nem tudjuk, hogy páros vagy páratlan számú adattal rendelkezünk.

Szezonális eltérések

Feladatunk most az, hogy az idősor adatainak felhasználásával adjuk meg a komponensek értékét. A trendet számíthatjuk mozgóátlagolással is, ami a következőt jelenti: először el kell dönteni, hogy hány tagú mozgóátlagot kívánunk számítani, vagyis hány adatot használunk fel egy-egy átlag kiszámításához. Legyen ez most k. A számítás menete a következő: kiszámítjuk az idősor első k adatának egyszerű számtani átlagát. Ez az első trendérték, amelyet az érintett időszak közepéhez - vagyis (k+1)/2-edik időszakhoz rendelünk. Ezután elhagyjuk az első adatot, és ehelyett vesszük a következő, (k+1)-ediket. Ismét átlagot számítva nyerjük a következő mozgóátlagot, vagyis trendértéket, amelyet a megfelelő időszakhoz rendelünk. Így haladunk, amíg az utolsó adatot is felhasználjuk. Az eredményül kapott trendértékek sorozata a kiegyenlített idősor. Vegyük észre, hogy k tagú mozgóátlag használatánál (k-1) adattal rövidül a sor, ha k páratlan, és k adattal, ha k páros szám.

A számítás menete látható a következő táblázatból, ahol k=4.

Az almalé kiskereskedelmi forgalma [11]

Év(i)	Negyedév(j)	Forgalom millió liter y_{ij}	Trend yt_{ij}	Eltérés $y_{ij}-yt_{ij}$	Véletlen v_{ij} $(y_{ij}-(yt_{ij}+S_j))$
1969	I.	108,0	-	-	-
	II.	165,9	-	-	-
	III.	171,4	→138,7	32,7	-4,4
	IV.	104,8	141,0	-36,2	-3,1
1970	I.	117,2	144,9	-27,7	3,1
	II.	175,4	150,3	25,1	-1,7
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.
1977	III.	213,4	213,4	36,4	-0,7
	IV.	176,5	215,6	-39,1	-6,0
1978	I.	186,3	219,8	-33,5	-2,7
	II.	255,6	225,0	30,6	3,8
	III	271,6	-	-	-
	IV.	196,6	-	-	-

A táblázat már megmutatja a többi komponens kiszámításának módját is a következő táblázat segítségével, amely a szezonális eltérések kiszámítására készült:

[11] A megértést nem nehezíti valamennyi adatat feltüntetése, ezért ezek kimaradtak a táblázatból.

Szezonális eltérések kiszámítása

Év	Eltérés a trend től az			
	I.	II.	III.	IV.
	negyedévben			
1969	-	-	32,7	-36,2
1970	-27,7	25,1	40,4	-27,3
1971	-37,2	24,1	37,5	-25,2
.				
.				
.				
1977	-30,4	31,9	36,4	-39,1
1978	-33,5	30,6	-	-
Összeg:	**-278,2**	**240,0**	**33,2**	**-299,7**
Átlag	-30,91	26,67	36,89	-33,3
Korrigált:	-30,8	26,8	371,	-33,1

A negyedévenként ható szezonális eltéréseket tehát úgy számíthatjuk ki, hogy az azonos negyedévi eltéréseket összeadjuk és eloszjuk az adatot tartalmazó negyedévek számával. A korrekciótényező számítása:

(-30,91+26,67+36,89-33,3)/ 4 = -0,16

A *Szezonális eltérések kiszámítása* táblázat utolsó sorában a korrekció is el lett végezve. Pl.

S_1=-30,91+0,16=-30,75.

A véletlen értékét úgy lehet kiszámolni, az adott értékből (y_{ij}) kivonjuk a trend (yt_{ij}) és a megfelelő (korrigált) szezonhatás(S_j) összegét ($yt_{ij}+S_j$)
Ezek után nézzük meg az algoritmust, amely az adatokat egy szekvenciális fileból olvassa és meghatározza a komponensek értékét.

```
Program idősor2
n:=Beolvas
k:=4:b:=(k+1) div 2: m=4            **4 negyedév van**
Ciklus c:=1-től m-ig                **vektorok nullázása**
    szezon[c]:=0
    darab[c]:=0
Ciklus vége
Ciklus c:=1-től n-k-ig              **mozgó átlagolás**
```

```
    ossz:=0
    Ciklus cikl:=1-től k-ig
        ossz:=a[c+cikl-1,1) + ossz
    Ciklus vége
    a[c+b,2]:=ossz   **a mátrix második oszlopában lesznek a trendada-
    tok**
    c:=c+1
Ciklus vége
Ciklus c:=1 + (k-2) -től n- (k-2)-ig   **a szezonális eltérések kiszámítása**
    melyik:=c MOD k
    szezon[melyik]:=szezon[melyik] +a[melyik,1]-a[melyik,2]
    darab[melyik]:=darab[melyik]+1 **a darabba kerül, hogy hány adat van*
Ciklus vége
ossz:=0
Ciklus c:=1-től m-ig   **az átlagok kiszámítása**
    szezon[c]:=szezon[c]/darab[c]
    ossz:=ossz+szezon[c]
Ciklus vége
ossz:=ossz/4
Ciklus c:=1-től m-ig   **a korrekció**
    szezon[c]:=szezon[c]-ossz
Ciklus vége
Ciklus c:=1 + (k-2) -től n-(k-2) -ig   **a véletlen kiszámítása**
    melyik:= c MOD k
    a[c,3]:=a[c,1]-a[c,2]+szezon[melyik]
Ciklus vége
Ki:: c[],szezon[]   **az eredmények kiírása**
Program vége
FüggvényEljárás Beolvas
Összerendel(f,"ADATI.INP")
Nyit(f)
n:=0
Ciklus amíg Nem Vége?(f)
    n:=n+1
    Be: a[n,1] -be f-ből
Ciklus vége
Zár(f)
Eljárás vége (Visszaad: n)
```

FELADATOK

1) Írjuk olyan algoritmust, amely a Cramer-szabály segítségével old meg egy n ismeretlenes négyzetes mátrixú lineáris egyenletrendszert! (Természetesen $x_i=D_i/D$, ahol D_i-t úgy kapjuk, hogy az A mátrix i. oszlopába az y vektort írjuk.

2) Határozzuk meg az A(N,N) mátrix K. hatványát!

3) Határozzuk meg az A(N,N) mátrixból a B(N,N) mátrixot a következő képlet alapján: B=1/2 * (A+T), ahol T az A(N,N) mátrix transzponáltja!

4) Adott az A(N,M) mátrix és a B(N) vektor (B(N) az 1-N számok egy permutációját tartalmazza). Helyezzük át a mátrix sorait úgy, hogy az i. sor a B(i). sorba kerüljön!

5) Határozzuk meg A(N,N) mátrix alsó és felső háromszöge azon elemeinek összegét, amelyek olyan sorban vannak, aminek első eleme pozitív!

6) Generáljuk a 0,1,2 számjegyekből olyan hosszúságú sorozatot, amelyben nincsenek szomszédos megegyező részsorozatok (például a 0101 sorozat nem jó, mert a 01 ismétlődik egymás mellett)!

7) Táblázatos formában írjuk ki az első egyre végződő háromjegyű, négyjegyű, ... n jegyű prímszámot!

8) Négyzetrácsot fektetünk a Fertő tóra. 0-t írunk oda, ahol víz van a rácspontban, egyet oda, ahol szárazföld van. Határozzuk meg a tó legdélibb, legkeletibb, legészakibb és legnyugatibb pontját!

9) Írjunk algoritmust a binomiális együtthatók kiszámítására a faktoriálisok és a Pascal-háromszög alapján egyaránt!

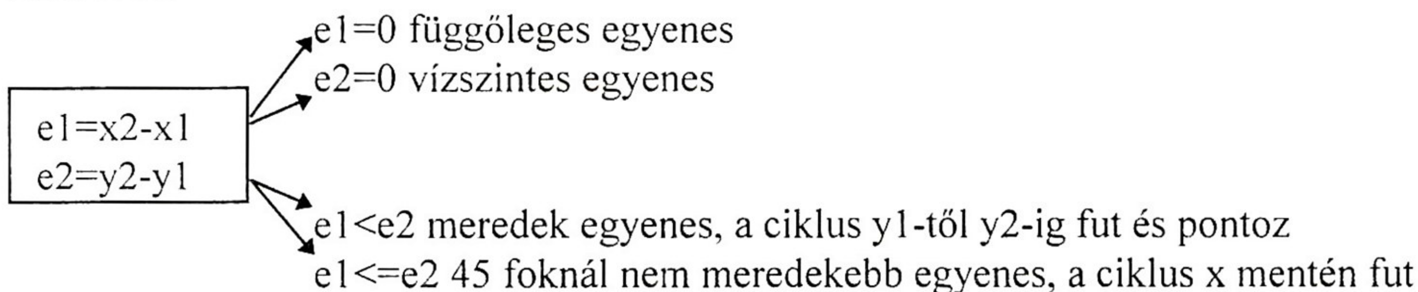
10) Készítsük el egy legfeljebb 10 elemű halmaz összes ciklikus permutációját!

VI. Elemi grafika

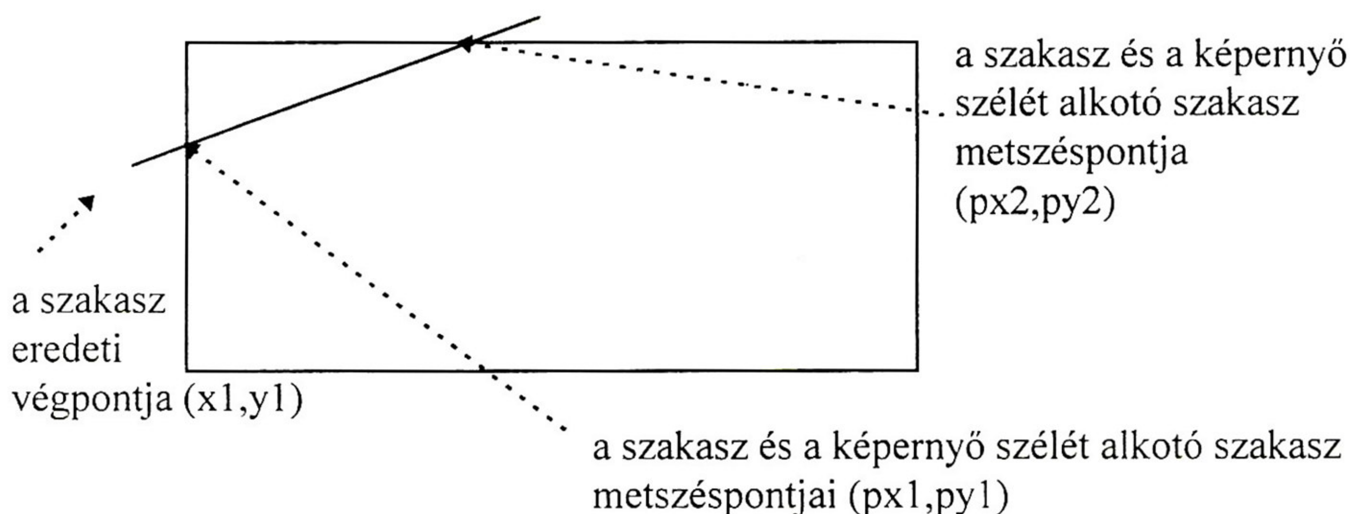
A programnyelvek mai implementációi már tartalmazzák azokat a függvényeket, eljárásokat, amelyek segítségével az egyszerűbb alakzatokat meg lehet rajzolni, ezért az alábbi eljárásoknak olyan nagy gyakorlati haszna nincs, hiszen egy részüket már "adják" a programhoz, másik részüket viszont meg kell írni. Építsünk fel tehát egy grafikus rendszert, amely feltételezi, hogy gépünk tud pontot megjelölni a képernyőn a megadott koordinátájú pontban, (Pont(x,y,i)) és a koordinátarendszer origója a bal felső sarok, az X tengely a vízszintes és az Y a függőleges, i színkód.

Szakasz rajzolása

A szakasz rajzolása rendkívül alapos munkát igényel, hiszen végső soron minden egyéb rajzolás erre vezethető vissza. Először tegyük fel, hogy az K(x1,y1) és a V(x2,y2) pontok a képernyőre esnek. Az egyenes helyzetétől függően elágazást vezetünk be:



Kívánatos, hogy szakaszrajzolónk akkor is működjön, ha a végpontok valamelyike nem esik a képernyőre. Ekkor meg kell határozni az egyenes (a szakasz) és az egyenest metsző "képernyő szélek" metszéspontját illetve metszéspontjait, és ezek lesznek a szakasz "belépő" és "kilépő" pontjai.



Két pont ismeretében egy egyenes egyenlete megadható, két egyenes egyenletéből pedig nem túl bonyolult meghatározni a metszéspontokat (nem kell hozzá még Gauss elimináció sem).

A "Vagas" függvény visszaadja a px1,py1,px2,py2 változókon keresztül a szakasz új végpontjait. A függvény igazat ad vissza, ha a szakasznak van olyan pontja, amely a képernyőn megjeleníthető, egyébként hamisat.

```
Függvény Vagas(x1,y1,x2,y2, px1,py1,px2,py2) **egészek, az utolsó
                                  négynél cím szerinti paraméterátadás**[12]
                                  **a vágást meghatározó téglalap koordinátái**
   Eljárás Csere
      Seged:=px1; px1:=px2; px2:=Seged
      Seged:=py1; py1:=py2; py2:=Seged
   Eljárás vége
Ha py1>py2 akkor Csere **Fenti**
Ha py1<y1 akkor
   Ha py2>=y1 akkor
      px1:=Kerekit((px1)+(px1-px2)*(py1-y1)/(py2-py1))
      py1:=y1
   Elágazás vége
Elágazás vége
Ha py2>y2 akkor  Lenti}
   Ha py1<=y2 akkor
      px2:=Kerekít((px1)+(px1-px2)*(py1-y2)/(py2-py1))
      py2:=y2
   Elágazás vége
Elágazás vége
Ha  px1>px2 akkor Csere **Bal**
Ha px1<x1 Akkor
   Ha px2>=x1 Akkor
      py1:=Kerekít((py1)+(py1-py2)*(px1-x1)/(px2-px1))
      px1:=x1
   Elágazás vége
Elágazás vége
Ha px2>x2 Akkor **Jobb**
   Ha px1<=x2 Akkor
      py2:=Kerekít((py1)+(py1-py2)*(px1-x2)/(px2-px1))
      px2:=x2
   Elágazás vége
Elágazás vége
```

[12] Az eljárás a "Vagas" függvény meghívása után megrajzolja a szakaszt.

```
Vágás:= ((px1>=x1) És (px1<=x2) És (py1>=y1) És (py1<=y2)) Vagy
                  ((px2>=x1) És (px2<=x2) És (py2>=y1) És (py2<=y2))
FüggvényEljárás vége (Visszaad Vágás)

**OszlopMax = a maximális oszlopkoordináta**
**SorMax = a maximális sorkoordináta**

Eljárás Vonal(x1,y1,x2,y2)          **egészek**
Ha Vagas(0,0,OszlopMax,SorMaxY ,x1,y1,x2,y2) Akkor
XElt:=x2-x1
YElt:=y1-y2
  Ha XElt=0 akkor
    Ha YElt=0 akkor Pont(x1,y1,1)
    Egyébként
      Ha y1<y2 akkor
        Ciklus i:=y1-től y2-ig  Pont(x1,i,1)
      Egyébként
        Ciklus i:=y2-től y1-ig Pont(x1,i,1)
      Elágazás vége
    Elágazás vége
  Egyébként
    Ha YElt=0 akkor
      Ha x1<x2 akkor
        Ciklus i:=x1-től x2-ig Pont(i,y1,1)
      Egyébként
        Ciklus i:=x2-től x1-ig Pont(i,y1,1)
      Elágazás vége
    Egyébként
    Ha Abs(xElt)>Abs(yElt) Akkor
      Ha x2<x1 Akkor
        Seged:=x1; x1:=x2; x2:=Seged
        Seged:=y1; y1:=y2; y2:=Seged
      Elágazás vége
      AktX:=x1; a:=y1; b:=(y2-y1)/(x2-x1)
        Ciklus i:=1-től x2-x1+1-ig
          Pont(AktX,Kerekít(a),1)
          Inc(AktX): a:=a+b              **AktX növelése**
        Ciklus vége
    Egyébként
      Ha y2<y1 Akkor
        Seged:=x1; x1:=x2; x2:=Seged
        Seged:=y1; y1:=y2; y2:=Seged
```

```
            Elágazás vége
            AktY:=y1; a:=x1; b:=(x2-x1)/(y2-y1)
              Ciklus i:=1-től y2-y1+1-ig
                Pont(Kerekít(a),AktY,1)
                AktY:=AktY+1; a:=a+b
              Ciklus vége
        Elágazás vége
      Elágazás vége
    Eljárás vége
```

A következőkben ismertetendő gyorsabb algoritmus alapgondolata Dan Cohentől és Iván Sutherlandtól származik, és egyenes vágására alkalmas.

Rendeljünk a szakasz végpontjaihoz egy-egy négybites kódot az alábbi módon:

1001	1000	1010
0001	0000	0010
0101	0100	0110

itt az ablak

A kód egyes bitjei a következőket jelentik:

1. bit: a pont a képernyőtől balra esik,
2. bit: a pont a képernyőtől jobbra van,
3. bit: a képernyő alatt van,
4. bit : a képernyő felett van.

Ha tehát a szakasz mindkét végpontjához a 0000 kód tartozik, akkor az teljes egészében a képernyőre esik, egyébként vágni kell a szakaszt.

Nem látható közvetlenül, de ellenőrizhetjük, hogy ha a két végpont kódjain bitenként logikai és műveletet végzünk és az eredmény nem csupa nulla, akkor a szakasz teljesen a képernyőn kívülre esik, tehát a rajzolásból kihagyható. Ha az és művelet eredménye nulla, de legalább az egyik végpont a képernyőn kívülre esik, akkor a szakaszt fel kell szabdalni. A képernyőt határoló egyenesek és a szakasz metszéspontjai könnyen meghatározhatók, amelyekre a konjukciót (és műveletet) elvégezve végül a szakasz látható részét megkaphatjuk. Az alábbi algoritmus éppen azt valósítja meg.

Megfeleltetések:

Típus:Él = (bal,jobb,alsó, felső)
Típus: kinnkod= Halmaz(Él) ***elemei Él típusúak***

Eljárás Vagas(x1,y1,x2,y2) ***mind valós. Cím szerint történik a paraméterátadás***
Eljárás Kod (x,y,c) ***x,y valósak, c kinnkod típusú és cím szerint tört. a paraméterátadás***
c:=[ ] ***a halmaz nullázása***

```
        Ha x<xbal akkor
            c:=[bal]
        Egyébként
            Ha x>xjobb akkor c:=[jobb]
        Elágazás vége
        Ha y<yalso akkor
            c:=c+[also]
        Egyébként
            Ha y>yfelso akkor c:=c+[felso]
        Elágazás vége
    Eljárás vége **Kod**
kod(x1,y1,c1):kod(x2,y2,c2)
Ha c1*c2=[ ] akkor
    Ciklus amíg (c1<>[ ]) Vagy (c2<>[ ])
        c:=c1
        Ha c=[ ] akkor c:=c2
        Ha bal eleme a c -nek akkor
            y:=y1+ (y2-y1) * (xbal-x1)/(x2-x1)
            x:=xbal
        Egyébként
            Ha jobb eleme c-nek akkor
                y:=y1+(y2-y1)*(xjobb-x1)/(x2-x1)
                x:=xjobb
            Egyébként
                Ha also eleme c-nek akkor
                    x:=x1+(x2-x1)*(yalso-y1)/y2-y1)
                Egyébként
                    Ha felso eleme c-nek akkor
                        x:=x1+(x2-x1)*(yfelso-y1)/(y2-y1)
                        y:=yfelso
                    Elágazás vége
                Elágazás vége
            Elágazás vége
        Elágazás vége
        Ha c=c1 akkor
            x1:=x;y1:=y;kod(x,y,c1)
        Egyébként
            x2:=x;y2:=y; kod(x,y,c2)
        Elágazás vége
    Ciklus vége
    **Itt már (x1,x2) és (y1,y2) pontok olyan szakasz végpontjai, amelyek
      teljes gészében láthatók az ablakban**
```

Elágazás vége
Eljárás vége

Adott [a,b] intervallumon folytonos függvényeket a szakaszrajzoló eljárást felhasználva most már könnyen megrajzolhatunk. Ha a megadott intervallum nagy, akkor gyorsítani kell a rajzolást. Ezt úgy tehetjük meg (a pontosság rovására) , hogy az [a,b] intervallumra előírt pontok számát megadjuk.

```
Program Fvgorbe
Ki: "Az intervallum végpontjai:"
Be: a,b
Ki: "A pontok száma:"
Be: n
dx:=(b-a) / n
x1:=a; y1:=f(a) **f a függvény, amit ábrázolunk**
x2:=x1+dx
Ciklus
    y2:=f(x2)
    **x1,x2,y1,y2-t itt egészre kellene kerekíteni**
    szakasz(x1,y1,x2,y2)
    x1:=x2; y1:=y2
    x2:=x1+dx
Amíg x2>b
Ha n*dx <> (b-a) akkor
    x2:=b;y2:=F(x2)
    **itt is kerekíteni kell egészre**
    szakasz (x1,y1,x2,y2)
Elágazás vége
Program vége
```

Ez így elég egyszerűnek tűnik, de nem nagyon használható ebben a formában:

- hiányzik a függvénypontok megfeleltetése a képernyőpontoknak, ugyanis máshol van az origó mint a valóságban. Gondoljunk csak arra, hogy ha a (-100 -10) pontot kellene láttatnunk. Így, ha a (kx0,ky0) pontot tekintjük origónak a képernyőn, akkor az (x,y) pont a képernyőn (kx0+x,ky0-y) koordinátájú pont lesz.
- elképzelhető, hogy a függvény nem analitikus formában van megadva, hanem például egy állományban vannak az ábrázolandó pontok koordinátái. Ebben az esetben tudnunk kell a szélső függvényértékeknek megfelelő koordinátákat.
- lehetséges, hogy a függvényértékek "nem férnek el" a képernyőn. Pl. az értékkészlet elemei rendre a [-1200,-1400] intervallumba esnek. Természete-

sen használható a vágás, de nem fogunk látni semmit, hiszen nem esnek a képernyőre a függvényértékek. Ekkor transzformálni kell a függvényértékeket (pl. ha túl nagyok, akkor egy egynél kisebb számmal lehet szorozni a függvényértékeket). Gonduljunk például a sin(x) függvényre, amelynek értékkészlete a [-1,1] intervallumbeli valós számok. Ha ezt szeretnénk ábrázolni, nyilván például 100-zal szorozni kell a sin(x) által visszaadott való számokat.

Feladat: a leírtak figyelembevételével készítsük el a függvényábrázoló algoritmust.

GÖRBÉK

A görbék megadásának egyik elterjedt módja a paraméteres előállítás. Ebben az esetben a görbe pontjainak mindkét koordinátáját egy paraméter függvényében adjuk meg:

$x=f(t)$
$y=g(t) \quad t\in[tmin,tmax]$,

ahol (x,y) a görbén lesz. Az ábrázolást úgy végezzük, hogy a [tmin,tmax] intervallumon osztópontokat veszünk fel:

$tmin=t_0<t_1<...<t_{k-1}<t_k<...<t_n=tmax$

és megrajzoljuk az

$\overline{(f(t_{k-1}),g(t_{k-1})),((f(t_k),g(t_k))} \quad k=1,2,..,n$

szakaszokat, vagyis a görbe húrjait. Persze vigyázni kell, hogy a húrok elég rövidek legyenek - különben a kapott ábra nem folytonos görbe illúzióját kelti.

Példa:
A **Lissajoux görbe** egyenlete:
$x=Sin(at)$
$y=Cos(t+b) t\in[a,b]\in R$ és a,b az [a,b] intervallum kezdete és vége.

Feladat: Készítsünk Lissajoux görbéket a következő paraméterek szerint:

$a=3/2$ $a=5/4$
$b=\pi/2$ $b=\pi/2$

Célszerű a lépésközt $\pi/100$ körülire venni, és nem szabad megfeledkezni, hogy a kapott (x,y) értékek 0..1 közé essenek, tehát ***szorozni kell őket egy konstanssal*** (például 100-zal).

Még egy görbetípus egyenlete:
x=[c+d*Exp(Sint)]*Sin(a*t)
y=[c+d*Exp(Sint)]*Cos(a*t) $t \in \Re$

Példa a paraméterekre:

Paraméter	Eset				
	1.	2.	3.	4.	5.
a	2/3	1/2	2/3	4/3	1/3
b	1/2	2/3	4/3	5/3	
c	30 mindenhol				
d	3 mindenhol				

Ez a módszer működik felületekre is. Felületek analitikai ábrázolása ezzel megegyezően a

z=f(x,y) segítségével vagy

x=f(t,u)
y=g(t,u)
z=h(t,u) paraméteres módszerrel történik.

A gömb paraméteres egyenlete:

x=r*sint*cost
y=r*sint*sinu
z=r*cost

ahol $t \in [0,360]$ $u \in [0,180]$

Interpoláló görbék

Az interpoláló görbe olyan függvény, amely előre megadott támpontok mennyiségén a helyes sorrendben átvonul.

Lagrange féle interpoláció

Ismertek az $x_1,x_2,\dots,x_n$ alappontok és a hozzájuk tartozó $y_1,y_2,\dots,y_n$ függvényértékek. Létezik pontosan 1 db, legfeljebb n-1-edfokú p polinom, amelyre $p(x_i)=yi$ $\forall i=1,2,..,n$. A p polinomot nevezzük Lagrange-féle interpolációs polinomnak.

Legyen $l_i(x)$ a Lagrange-féle interpolációs **alappolinom!**

$l_i(x)=((x-x_1)*(x-x_2)* \ldots *x-x_{i-1})*(x-x_{i+1})*..*(x-x_n)) \,/\, ((x_i-x_1)*(x_i-x_2)*\ldots*(x_i-x_{i-1})*(x_i-x_{i+1})*\ldots*(x_i-x_n)$

azaz

$$l_i(x) = \prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{x-x_i}{x_j-x_i}$$

A Lagrange-féle polinom: $p(x)=y_1*l_1(x)+\ldots+y_n*l_n(x)$.
Könnyű belátni, hogy $p(x_i)=y_i$

Példa: Legyen ismertek egy függvény értékei a következő alappontokban:

$x_1=100$ $y_1=10$
$x_2=121$ $y_2=11$
$x_3=144$ $y_3=12$

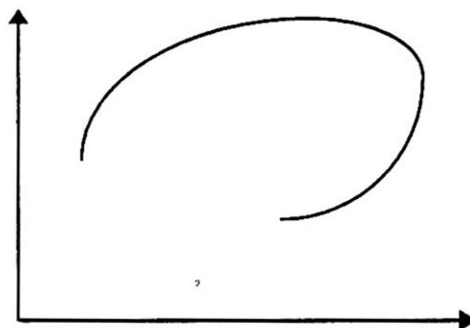
Kérdés: Mit vesz fel a függvény az x=110 pontban?

Megoldás: $p(110)=(10*(110-121)*(110-144)) \,/\, ((100-121)*(100-144)) + ((11*(110-100)*(110-144)) \,/\, ((121-100)*(121-144)) + (12*(110-100)*(110-121)) \,/\, ((144-100)(144-121)) = 10{,}4865$

Ezek után a görberajzolás úgy történik, kijelölünk és megrajzoljuk az alappontokat, majd kiszámítjuk a p polinom értékét minden pontban x_1 és x_n között. Ha a fenti példát nézzük, akkor ki kell számolni p értékét 101, 102, ..., 143 pontokban és megjeleníteni a koordinátákat.

Hátránya az interpolációnak - többek között -, hogy x értékekben rendezett támpontokat igényel, ezért csak balról jobb haladó görbéket tesz lehetővé. Nincs lehetőség például visszafelé haladó görbék rajzolására.

Példa:



A fenti görbét interpolációval nem igazán kehet megrajzolni.

KÖZELÍTŐ GÖRBÉK

Különböző alkalmazásoknál elegendő, ha az előállított görbe csak megfelelően formálható, de nem szükséges, hogy azokon a pontokon át is haladjon, melyek meghatározzák, azaz a meghatározó támpontok hatásának csak megközelítőleg kell pontosnak lennie.

Bezier-görbék

Bezier francia matematikus, a Renault gyárban karosszériatervezéssel foglalkozott. Egy Bezier görbe ((B(t)) a következőképpen definiálható:

$$B(t)=\sum_{i=0}^{n} p_i * B_{i,n}(t), \qquad t\in[0,1]$$

ahol

$$B_{i,n}(t)=\binom{n}{i} * t^i * (1-t)^{n-i}$$ *"n alatt az i" van a zárójelben*

A B(t) a t paraméter által előállított görbe; a Bezier-görbe tehát paraméteresen ábrázolható és ezért tetszőleges síkgörbéknél használható. Minden $t\in[0,1]$ értéknek a görbe egyik pontja felel meg.

A p_i-k a támpontok vektorai, tehát

$$\vec{p_i}=\begin{pmatrix} x_i \\ y_i \end{pmatrix}$$ *p_i egy vektor*

A megvalósítás menete:

Megjeleníteni a támpontokat: p_i-ket.

2) Fussa be t az intervallumot például 0,05 lépésközzel. Azaz t=0,05 majd 0,1, stb. Minden t -re számoljuk ki B(t)-t. Nyilván egy vektort kapunk, azaz a (x_i,y_i) pontot. Ezeket kell ábrázolni, illetve (TP-ben) LineTo-val összekötni.

B-spline-ok

Elvileg ugyanolyan felépítésűek, mint a Bezier-görbék, csak a $B_{i,n}(t)$ helyett más függvényeket használnak.

A B-spline-ok formája:

$$B(t)=\sum_{i=0}^{n} p_i * N_{i,k}(t),$$

ahol

$$N_{i,1}(t) = \begin{cases} 1 \text{ ha } t_i <= t <= t_{i+1} \\ 0 \text{ egyébként} \end{cases}$$

$$N_{i,k}(t) = \frac{(t-t_i) * N_{i,k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+1}-t)\, N_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}$$

ha $\in [0,n-k+2]$

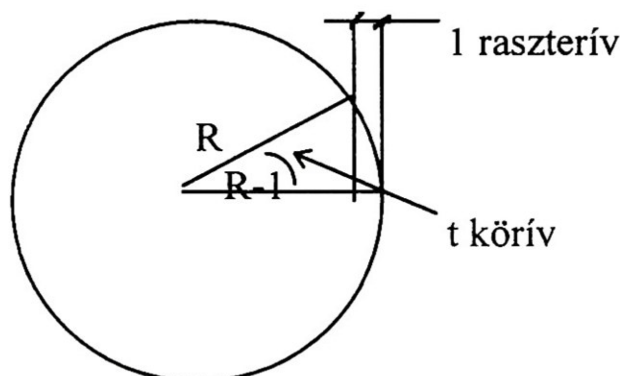
$$t_i = \begin{cases} 0 & \text{ha } i<k \\ i-k+1 & \text{ha } k<=i<=n \\ n-k+2 & \text{ha } i>n \end{cases}$$

KÖR RAJZOLÁSA

Kört és körívet nem a szokásos

x=r cost
y=r sint

egyenletrendszerrel rajzolunk, mert akkor sokszor kell trigonometrikus függvényt számoltatnunk. Azt nézzük meg, hogy egy r sugarú kört adott raszterfelbontás esetén hány részre kellene felosztani ahhoz, hogy húrokkal tudjuk közelíteni.



Az ábra alapján:

R cost = R-1
cost = 1-1/R
R sint = R2-(R-1)2 = 2R+1
sint = 2R+1/R

A P(x,y) pont origó körüli t szöggel való elforgatását az

x'=xcost - ysint
y'=xsint + ycost

egyenletrendszer írja le, ami jól használható lesz a forgatás során. Kört azonban máshogyan is lehet rajzolni (a kör egyenletét felhasználva):

```
**(x,y) a kör középpontja , r a sugár**
 Eljárás Kör(x,y,r) **egészek**
 yy:=r; xx:=0; rNegyzet:=r*r
 Ciklus
     xx:=xx+1
     Ha  xx*xx+yy*yy>rNegyzet akkor  yy:=yy-1
     Ha xx*xx+yy*yy>rNegyzet akkor xx:=xx-1
     Pont(x+xx,y+yy): Pont(x-xx,y+yy)
     Pont(x+xx,y-yy): Pont(x-xx,y-yy)
 Amíg  yy<=0
 Eljárás vége
```

FORGATÁS

Ha a síkban a P(x,y) pontot szeretnénk elforgatni alfa fokkal az (x0,y0) pont körül, akkor a következő képlettel[13] számolhatunk, hogy a P2(x2,y2) pontot kapjuk:

x2=x0+(x-x0)*Cos(alfa)-(y0-y)*Sin(alfa)
y2=y0+(x-x0)*Sin(alfa)+(y0-y)*Cos(alfa)

Alfa-t fokokban kell megadni (xo,yo) a forgatási középpont

[13] Az addíciós tételből levezethető a képlet.

Eljárás Forgat(x,y,xo,yo,Alfa) ***Alfa valós, a többi egész, x,y cím szerint***
Alfa:=-Alfa/180*Pi ***átszámítás radiánba***
xx:=Kerekít(xo+(x-xo)*Cos(Alfa)-(y-yo)*Sin(Alfa))
y:=Kerekít(yo+(x-xo)*Sin(Alfa)+(y-yo)*Cos(Alfa)); x:=xx
Eljárás vége ***x és y tartalmazza a pont új koordinátáit***

Az eljárás az (x,y) koordináták által meghatározott pontot forgatja el (x0,y0) körül. A pont új koordinátája az x és az y változókba kerül. Ennek az eljárásnak a segítségével több pont által meghatározott alakzatot is el tudunk forgatni, hiszen csak egymás után meg kell hívni ezt az eljárást a megfelelő pontokra.

FELADATOK

1) Forgassunk el egy szakaszt egy pont körül 1 fokkal többször is!

2) Forgassunk el egy síkidomot kétszeresen, többször egymás után! Egyrészt a középpontja, másrészt egy külső pont körül forgassunk, de más-más szöggel!

3) Ábrázoljuk koordinátarendszerben a -200*X*X-500 függvényt (a [-200,200] intervallumon)!

4) Ábrázoljuk a Sin(x)/x függvényt és fessük be a függvény és az X koordináta közötti területet!

5) Ábrázoljuk az

   x(t) = r*sin(t)

   y(t) = r*cos(t)

   paraméteres alakban adott kört a 0<t<=7.28 intervallumban!

6) Rajzoljunk n darab szakaszt a képernyőn. Ha lehetséges, rajzoljuk meg azt a szakaszt, amelynek van legalább egy közös pontja az összes többi szakasszal!

7) Rajzoljunk a szakaszrajzoló segítségével egy kockát!

8) Írjunk algoritmust, amely elforgat egy sokszöget!

9) Készítsünk programot, amely elsőfokú-,másodfokú -és szögfüggvényeket és a transzformációkat ábrázolja, illetve szemlélteti!

10) Egy szöveges file miden sorában számpárok találhatók, ahol az első szám az értelmezési tartomány egy eleme, a másik a hozzá tartozó függvényérték. Ábrázoljuk koordinátarendszerben a függvényt!

11) Írjunk algoritmust, amely egy hasábot az X tengelyre tükröz!

IRODALOMJEGYZÉK

Szlávi Péter: Adatok, adattípusok

Szlávi Péter: Előadás a gráftípusról

Számítástechnikai feladatok 2000-ig I-II.: Szerkesztette:Dr. Hetényi Pálné

Számítástechnika középfokon: Szerkesztette: Dr. Hetényi Pálné

Niklaus Wirth : Algoritmusok+Adatstruktúrák = Programok

Angster Erzsébet - Kertész László : Turbo Pascal 6.0

Szlávi Péter: Adatok, adattípusok

Benkő Péterné, Benkő László, Tóth Bertalan: Programozzunk C nyelven!

GO-0020

A sorozat moduljai

1. Frank Pálné	Bevezetés az informatikába
2. Faránki Gyula	Informatikai eszközök
3. Busi Lajos	Számítógépes szoftverek
4. Bánhegyesi Zoltán	Számítógép-hálózatok
5. T. Várkonyi Attila	A természet informatikája
6. Dr. Koncz József	Irodai alkalmazások: szövegszerkesztés
7. Módos Gábor	Word for Windows 2.0 szövegszerkesztő
8. Bánhegyesi Zoltán	Adatfeldolgozás alapjai
9. Bánhegyesi Zoltán	dBASE III Plus kezelése
10. Módos Gábor	Works for Windows 3.0 alapjai
11. Nemcsik János	dBASE IV kezelésének alapjai
12. Faragó István-Piross László	A szövegszerkesztés alapjai
13. Nemcsik János	A táblázatkezelés logikája
14. Módos Gábor	Excel for Windows 4.0 táblázatkezelő
15. Dombovári Mátyás	Informatika a technikában
16. Busi Lajos	Szövegszerkesztés egyszerűen: WinWord 6.0
17. Király Sándor	A programozás logikája I.
18. Módos Gábor	Programozás Turbo Pascal nyelven
19. Bánhegyesi Zoltán	Kapcsolat a külvilággal: Internet
20. Király Sándor	A programozás logikája II.
21. Miklósi Viktor	Windows '95 kezdő felhasználóknak
22. Busi Lajos	MS-Office for Windows '97. I/1. (Word-Excel)
23. Tóth Márton László	Multimédia
24. Sándor Miklós	CAD/CAM alapjai
25. Bánhegyesi Zoltán	Adatfeldolgozás Access 2.0 segítségével
26. Busi Lajos	Középiskolai feladatgyűjtemény
27. Bánhegyesi Zoltán	A számítógépes szimuláció alapjai
28. Bánhegyesi Zoltán	MS-Office for Windows '97. II. (Access)
29. Busi Lajos	MS-Office for Windows '97. I/2. (Word-Excel)
30. Király Sándor	C++ programozás alapjai
31. Király Sándor	Novell Netware 4.1 felhasználói ismeretek
41. Frank-Faránki-Busi-Bánhegyesi	Informatika I. (1-4 modulok)
42. Nemcsik-Busi-Bánhegyesi	Informatika II. (13. 16. 25. modulok)
43. Király Sándor	Informatika III. (17. és 20. modulok)
44. Busi Lajos-Bánhegyesi Zoltán	Informatika IV. (22. 28. 29. modulok)