Módszeres programozás: hatékonyság

E füzetben a hatékonyság témakörét tekintjük át. Programtervezési módszercsaládot szeretnénk adni jó programok készítéséhez.

Először megvilágítjuk, mit is nevezünk tulajdonképpen hatékonyságnak.

Másodjára a végrehajtási idő csökkentését vizsgáljuk meg (ezt a ciklusok végrehajtási számának vagy egyszeri végrehajtási idejének csökkentésével érhetjük el),

Harmadiknak a helyfoglalás csökkentésével foglalkozunk (ami az adatok mennyiségének - sorozatok elemszáma és elemmérete, illetve a programkód méretének csökkentését jelenti),

Negyedikként a bonyolultság csökkentése következik (amin belül az algoritmus és az adatszerkezet bonyolultsága szerepel).

Végül három részletesen kidolgozott példát nézünk meg, egyet a végrehajtási idő, egyet a helyfoglalás csökkentése, egyet pedig az adatreprezentáció megválasztása témaköréből.

Zsakó László

Módszeres programozás:

Hatékonyság

µlógia 6

ELTE Informatikai Kar

5., bővített kiadás

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

Tartalomjegyzék

Előszó

A programozási feladatok nagy osztályokba sorolhatók: pl. keresési, számlálási, rendezési, szétválogatási stb. feladatok. Ezen általános feladatosztályoknak megadható az általános megoldó algoritmusuk, s bebizonyítható, hogy az általános algoritmus helyes megoldása az általános feladatosztálynak. Emiatt hívhatjuk ezeket programozási tételeknek is.

Ezek az általános megoldások azonban legtöbbször nem veszik (nem vehetik) figyelembe a feladatok specialitásait, így nem lehetnek azok optimális megoldásai. Ezek hatékonyabbra átírása emiatt szükséges, elvégzendő feladat.

Fel lehet azonban fedezni a feladatokban a **hatékonyabbra írás szempontjából is rokon vonásokat**, azaz kialakítható **a hatékony algoritmusok tervezésének módszertana.**

A programozási tételek kialakulásánál azt vehetjük figyelembe, hogy az egyes konkrét feladatokban mi a közös, tehát az osztályba sorolást a feladat alapján végezhetjük. A hatékonysági szempontokból vett osztályok létrehozásánál nem a feladatból, hanem a feladatban használt adatszerkezetből, a megoldás szerkezetéből indulunk ki.

Ezek az osztályok ezért sokkal kevésbé lesznek formalizálhatók, de mégis megadható mindegyikhez az a közös gondolat, amelynek segítségével megadható a hatékonyabb megoldás. A dolgozatban elsősorban tehát programtervezési módszercsaládot szeretnék adni, áttekintve egyben a hatékonyságvizsgálat teljes témakörét.

A dolgozat algoritmusok, programok hatékonyságával foglalkozik.

A hatékonyságvizsgálatot a programozási munka szemszögéből vizsgáljuk, de nem tekintjük csupán -sőt elsősorban nem a kóddal való- játszogatásnak. Nem is abból a szempontból vizsgáljuk, mint az algoritmuselméleti szakemberek, akik egy feladatra az eredetitől lényegében eltérő, de valamilyen szempontból jobb algoritmust keresnek. Szemléletünk szerint **a hatékonyabbra írás egy programtervezési módszert, módszercsaládot jelent.**

Ebben a könyvben csak struktúrált programok vizsgálatával foglalkozunk.

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

Ha egy kész programot átírtunk hatékonyabbra, elképzelhető, hogy a javításokkal hibákat is helyeztünk el benne (legtöbbször sajnos éppen ez a helyzet áll fönn). Alapvető

követelmény, hogy az átírt program ugyanazokat az eredményeket adja, mint a korábbi változat. Erről meggyőződni csak úgy lehet, ha lefuttatjuk az összes korábbi tesztadattal. Ilyenkor (is) jó az, ha a tesztelés módját és eredményeit leírtuk, mert így az eredményeket könnyen összehasonlíthatjuk a korábbiakkal.

Egyes módszerek a füzetben több helyen is szerepelnek. A módszer lényege mindenütt ugyanaz, csupán a célja különbözik (pl.: a végrehajtási idő csökkentése, a bonyolultság csökkentése). Előfordul az is, hogy az ok különbözik, ami miatt az adott módszerre szükség van.

Hatékonyságról kétféle szempont szerint fogunk beszélni:

Globális hatékonyság: Itt az algoritmus (sőt néha a specifikáció) egésze vizsgálandó, s ennek megértése alapján kell elkészíteni a hatékony megoldást! Ez a tevékenység tehát szorosan a programtervezés része kell legyen!

Lokális hatékonyság: Itt a kód kiragadott részlete vizsgálandó, a program többi részétől függetlenül, a működés megértése nélkül, majd a megfelelő szerkezetek, formák felismerése után a kód transzformálható. Ez a tevékenység nagyrészt mechanikus, automatizálható munka, részben meg is valósítják bizonyos kódoptimalizáló programok.

Ez a füzet a **globális hatékonyság** vizsgálatával, javításával foglalkozik.

I. A hatékonyság fogalma

1. Példa: ciklikus léptetés

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

Gyakran kell ilyen jellegű részfeladatot megoldani, s a megoldás jelentősen befolyásolhatja programunk hatékonyságát.

Például amikor a feladat egy titkosírás megfejtése, amely egy kódtáblázat alapján készül. A táblázat egy eleme megmutatja, hogy egy betűt milyen másik betűvel kell helyettesíteni, a táblázat I. eleme az ABC I. betűje helyére írt betűt tartalmazza. Ez a titkosírás hosszú szövegek esetén viszonylag könnyen megfejthető az egyes betűk előfordulásának gyakorisága alapján. Tudjuk például, hogy a magyar nyelvben az „e" a leggyakoribb betű! Ha a titkosírást úgy módosítjuk, hogy minden sora (vagy akár minden betűje) más kódtáblázat alapján készüljön, akkor a megfejtést sokkal nehezebbé tesszük. Ezt például úgy végezhetjük, hogy minden egyes sor (betű) kódolása után átalakítjuk a kódtáblázatot. A táblázat soronkénti transzformációját a táblázat elemeinek ciklikus eltolásával (léptetésével) oldjuk meg.

Értelmezés: egy táblázat elemeinek eggyel balra való ciklikus léptetése azt jelenti, hogy minden elem eggyel előbbre kerül, az elsőt pedig a végére tesszük.

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

Legyen N a szöveg hossza, SZOVEG() pedig a K-val balraléptetendő szöveg!

```
Léptetés(SZÖVEG,N,K):                    (1. változat)
   Ciklus J=1-től K-ig
      BETŰ:=SZÖVEG(1)
      Ciklus I=2-től N-ig
         SZÖVEG(I-1):=SZÖVEG(I)
      Ciklus vége
      SZÖVEG(N):=BETŰ
   Ciklus vége
Eljárás vége.
```

Próbálkozzunk egy másik megoldással: ha megjegyezzük az első K db. elemet, akkor a K+1.-et rögtön a helyére, az első helyre tehetjük, a K+2.-et a másodikra, ..., s végül a

megjegyzett K db. elemet az előremásoltak mögé tesszük. A megjegyzéshez szükségünk van egy vektorra BETUK(K).

```
Léptetés(SZÖVEG,N,K):                    (2. változat)
  Ciklus I=1-től K-ig
     BETŰK(I):=SZÖVEG(I)
  Ciklus vége
  Ciklus I=K+1-től N-ig
     SZÖVEG(I-K):=SZÖVEG(I)
  Ciklus vége
  Ciklus I=1-től K-ig
     SZÖVEG(N-K+I):=BETŰK(I)
  Ciklus vége
Eljárás vége.
```

Próbálkozzunk egy harmadik megoldással is! Jegyezzük meg az első elemet! Tegyük a helyére a K-val később levőt, ekkor ez a hely felszabadul! Tegyük most a K-val később levő helyére a nála K-val később levőt (ez természetesen mod N értendő), s ha N-1 ilyen előremásolást elvégeztünk, akkor az utolsónak maradt üres helyre tegyük az először megjegyzettet!

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

Vajon jó-e ez a módszer? Próbáljuk ki papíron! N=6, K=1 esetben a megoldást jónak találjuk, azonban az N=6, K=3 esetben ez a program az első és a negyedik elemet cserélgeti! Gondoljuk meg, mi okozza, hogy a megoldás nem mindig jó!

Azon K értékek meghatározásához, amelyekre az algoritmus helyesen működik, némi matematikai ismeretre van szükségünk.

Állítás: Ha N és K olyan számok, hogy csak egyetlen közös pozitív osztójuk van, az 1 (N és K relatív prímek), akkor igaz: I*K (I=0, 1, 2, ..., N-1) N-nel való osztásának maradékai mind különbözőek és így kiadják az összes számot 0 és N-1 között.

Ezekhez az értékekhez egyet hozzáadva megkapjuk az összes 1 és N közötti számot és mindegyiket pontosan egyszer. Vegyük észre, hogy ha ezeket a számokat indexnek használjuk, akkor a sorozat első helyén levő indexű betű helyére pontosan a második helyen levő indexű betűt kell tenni, mivel a második index pontosan K hellyel mutat későbbre a táblázatban. Ez az összefüggés az összes egymást követő számpárra érvényes.

Mivel a fenti számsorozatban az összes 1 és N közötti index szerepel, ezért minden elem léptetését elvégeztük. Mivel mindegyik pontosan egyszer szerepel, ezért mindegyiket egyszer hoztuk előre. Mivel két elem közül a második (amit előrehoztunk) mindig K-val volt hátrább, ezért mindegyiket K-val hoztuk előre. Így a megoldás, ha feltesszük, hogy N és K relatív prímek:

```
Léptetés(SZOVEG,N,K):        (3. változat)
    HOVA:=1: BETŰ:=SZÖVEG(HOVA)          [első üres hely]
    Ciklus I=1-től N-1-ig
        HONNAN:=HOVA+K; Ha HONNAN>N akkor HONNAN:=HONNAN-N
        SZÖVEG(HOVA):=SZÖVEG(HONNAN)
        HOVA:=HONNAN                     [következő üres hely]
    Ciklus vége
    SZÖVEG(HOVA):=BETŰ
Eljárás vége.
```

Ha N és K nem relatív prímek, akkor a megoldás kicsit bonyolultabb:

```
Léptetés(SZOVEG,N,K):              (4. változat)
    Ciklus J=1-től lnko(N,K)-ig
        HOVA:=J: BETU:=SZOVEG(HOVA)
        Ciklus I=1-től N/lnko(N,K)-1-ig
            HONNAN:=HOVA+K; Ha HONNAN>N akkor HONNAN:=HONNAN-N
            SZOVEG(HOVA):=SZOVEG(HONNAN)
            HOVA:=HONNAN
        Ciklus vége
        SZOVEG(HOVA):=BETU
    Ciklus vége
Eljárás vége.
```

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

Vegyünk példánkkal kapcsolatban három követelményt:

- a megoldás legyen gyors,
- a megoldás kevés helyet használjon,
- a megoldás legyen egyszerű (könnyen és gyorsan elkészíthető, megérthető)!

Vizsgáljuk meg megoldásainkat a követelményeink szempontjából!

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

1. megoldás: **S=K*(N+1)**

2. megoldás: **S=N+K**

3. megoldás: ha az aritmetikai értékadásokat nem vesszük figyelembe:

S=N+lnko(N,K)

Megállapíthatjuk, hogy a sebesség tekintetében az első megoldás kirívóan rossz.

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

1. megoldás: H=N+1

2. megoldás: H=N+K

3. megoldás: H=N+1

Ebből a szempontból tehát a második megoldás rosszabb a többieknél.

Egyszerűség: A logikai egyszerűséghez pontos mérőszámot nem tudunk adni, így csak benyomásainkat írjuk le.

Az első és a második megoldást nagyon könnyen elkészítettük, a harmadikhoz azonban nagyobb elmélyülésre, komolyabb „fegyverekre" (matematikai ismeretekre) volt szükség. Ennek meggondolása, elkészítése bizony hosszabb időt vett igénybe, több munkánkba került.

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

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

Gyakori programkészítási stratégia az, hogy elkészítünk egy helyesen működő, de nagyon egyszerű megoldást, például olyat, ami közvetlenül levezethető a programozási tételekből. Ha ennek elkészülte után van még időnk, illetve valamilyen szempont miatt szükség van rá, akkor kezdünk töprengeni azon, hogyan lehetne a feladatra egy jobb megoldást adni.

2. Mi a hatékonyság

Az előző fejezet példája alapján megállapíthatjuk, hogy hatékonyságon háromféle dolgot érthetünk, háromfélét vizsgálhatunk: a **végrehajtási időt**, a **helyfoglalást**, illetve a **bonyolultságot**.

Amit tudunk: a végrehajtási idő gépfüggő jellemző, a helyfoglalás lényegesen függ az adat- és programábrázolástól, így ez nyelv-, sőt implementációfüggő (ez a végrehajtási időre is igaz), míg a logikai bonyolultság elsősorban a programozó felkészültségétől függ.

Az azonban biztos, hogy ha például egy program műveleteinek számát felére csökkentjük, akkor a végrehajtási idő is -mindentől függetlenül- a felére fog csökkenni. De ha például 10 összeadást helyettesítünk 2 szorzással, akkor már figyelembe kell venni, hogy ez az adott gépen, nyelven megéri-e vagy sem! Mi ebben a füzetben elsősorban az előbbi esettel foglalkozunk.

Mit is jelent a program végrehajtási ideje? Vannak olyan programok, amelyeknél a végrehajtási idő nem függ a bemenő adatoktól, másoknál viszont lényegesen függ:

```
Szorzás(A,B,C):
   C:=A*B
Eljárás vége.
```

```
Szorzás(A,B,C):
  C:=0
  Ciklus X=1-től B-ig
    C:=C+A
  Ciklus vége
Eljárás vége.
```

Az első algoritmus végrehajtási ideje A és B értékétől független (ez természetesen csak akkor igaz, ha a konkrét számítógépes megvalósítás olyan, hogy a szorzás végrehajtása nem függ a számok nagyságától adott szóhossz mellett), míg a másodikról ez nem mondható el, sőt a végrehajtási időt a szorzandók sorrendje is befolyásolja. Igy beszélhetünk egy program **minimális, maximális** és **átlagos végrehajtási idejéről.**

A helyfoglalást kétféle értelemben is vizsgálhatjuk. Egyrészt beszélhetünk a **programkód méretéről**, másrészt pedig a **változók helyfoglalásáról.** A megkülönböztetés értelme az, hogy más-más módszereket fogunk találni az egyik, illetve a másik csökkentésére.

Megjegyzés: A „programkód mérete" kifejezés sokszor félreértésekre ad lehetőséget. Ez ugyanis semmiképpen sem jelentheti azt, hogy például a programba kevesebb szóközt írjunk vagy az azonosítók hosszát minél rövidebbre válasszuk stb. Sokkal inkább jelenti azt, hogy az algoritmikus elemek (elágazások, ciklusok) számát csökkentsük, a többször leírt részeket csak egyszer írjuk le stb.

A helyfoglalás vizsgálatát más szempont is vezérelheti: **helyfoglalás a memóriában, helyfoglalás a háttértáron.**

Sokan állítják, hogy helyfoglalással ma már nem érdemes foglalkozni, mert a számítógépek memóriája, háttértár-kapacitása ezeket a vizsgálatokat feleslegessé teszi. Az ellentábor azt hangoztatja, hogy mindennapi munkájában gyakran tapasztalja, hogy mik-

roszámítógépe (IBM PC) memóriája, háttértára kicsinek bizonyul. Ez az eset komoly, nagyszámítógépes rendszereknél is előfordul.

A bonyolultságról a legnehezebb bármit is mondani, mivel ez az a jellemző, amelyet a legszubjektívabban lehet mérni. Fel lehet fogni úgy is, hogy mennyi munkába (időbe, tudásba) kerül megérteni, valamint úgy is, hogy mennyire van szükség a program megírásához.

A bonyolultság jelenthet **logikai bonyolultságot** (ezt a megoldáshoz szükséges ismeretekkel, a megoldás egyes részeiben szereplő tervezési döntések számával próbáljuk mérni) és jelenthet **szerkezeti bonyolultságot** is (ez utóbbit könnyebb mérni: pl. a programban szereplő döntéscsomópontok számával, az algoritmikus szerkezetek egymásbaágyazásának mélységével stb.).

II. A végrehajtási idő csökkentése

A program mely részeit vizsgáljuk a végrehajtási idő csökkentése érdekében? Azok a részek, amelyeket csak egyszer, kétszer hajtunk végre, biztosan nem befolyásolják lényegesen a futási időt, pontosabban ezek gyorsításával csak kisebb mértékben javul a program. Azokat kell megnéznünk, amelyekre sokszor kerül a vezérlés, azaz ciklusok belsejében vannak vagy rekurzív eljáráshívás miatt kell megismételni őket. Mivel ez utóbbi eset rendkívül hasonló a ciklusokhoz, ezért külön nem foglalkozunk vele.

Egy ciklus átlagos futási idejét a következő képelt határozza meg:

futási idő = lépésszám * egyszeri lefutás ideje.

A gyorsabbra írást tehát kétféleképpen tudjuk elérni, vagy a ciklusok végrehajtási számát csökkentjük, vagy pedig a ciklusmagok egyszeri végrehajtási idejét.

1. Ciklusok végrehajtási számának csökkentése

1.1. Sorozat elemszámának csökkentése

Ebben a fejezetben olyan feladatokkal foglalkozunk, amikor **egy sorozathoz kell rendelni egy értéket vagy egy újabb sorozatot.** Közös tulajdonsága ezeknek a feladatoknak, hogy **mindig van a sorozatnak olyan része, amelyben a keresett elem** -bármelyik is az- **garantáltan nem lehet. Ezt a részt** nem az adatszerkezet, hanem **a konkrét feladat alapján tudjuk meghatározni.** A ciklus lépésszámát ezért itt úgy csökkenthetjük, hogy ezt a részt a keresés megkezdése előtt elhagyjuk. A lényeg tehát **a specifikáció szigorítása.**

Szoros kapcsolat lehet e fejezet, illetve a helyfoglalás csökkentésével foglalkozó egyik fejezet (1.4) feladatai között, hiszen a hézagosan kitöltött struktúrák feldolgozása -éppen a kevesebb tárolt elem miatt- kevesebb lépést igényel.

Feladat: Döntsük el egy számról, hogy prímszám-e!

Megoldás: Az első, lehető legegyszerűbb megoldás:

```
Prím(N):
  I:=2
  Ciklus amíg I<N és I nem osztója N-nek
    I:=I+1
  Ciklus vége
  Prím:=(I=N)
Eljárás vége.
```

Milyen matematikai ismeretet használhatunk fel a végrehajtási szám csökkentéséhez? Egy számot akkor nevezünk prímszámnak, ha az 1-en és önmagán kívül nincs más osztója. Így ránézésre is igaznak tűnik: **az osztó biztosan nem lehet a szám és a szám fele között.**

```
Prím(N):
  I:=2
  Ciklus amíg I≤N/2 és I nem osztója N-nek
    I:=I+1
  Ciklus vége
  Prím:=(I>N/2)
Függvény vége.
```

Sőt, **ha van osztója, akkor kell lenni a szám négyzetgyökénél kisebb (vagy egyenlő) osztónak** is. (Gondoljunk a szám felbontásában szereplő tényezők a szám négyzetgyökére való szimmetriájára!) Ezért a fenti algoritmust úgy módosíthatjuk, hogy a végrehajtási számot lényegesen kisebbre vesszük. S ezzel elhagytunk nem is kevés felesleges(!) hasonlítást. Vegyük észre a számottevő nyereséget: mekkora a különbség 10000 és négyzetgyök 10000 között!)

```
Prím(N):
  I:=2
  Ciklus amíg I*I≤N és I nem osztója N-nek
    I:=I+1
  Ciklus vége
  Prím:=(I*I>N)
Függvény vége.
```

Megjegyzés: Az I*I≤N a négyzetgyökvonás esetleges pontatlansága miatt célszerűbb az I≤gyök(N)-nél. Jelent ez egy másik végrehajtási idő csökkenést is, mivel az időigényesebb gyök() függvény helyett csak egy szorzást kell elvégezni. (Más kérdés, hogy gyök(N)-t kiszámíthatnánk a ciklus végrehajtása előtt is, ez azonban már a ciklusmag egyszeri végrehajtási idejének csökkentése kérdéskörbe tartozik.)

Feladat: Határozzuk meg egy adott egész szám legkisebb osztóját, ami nem az 1!

Megoldás: Egy lehetséges megoldás a következő lehet:

```
Legkisebb osztó(N,I):
  I:=2
  Ciklus amíg I nem osztója N-nek
    I:=I+1
  Ciklus vége
Eljárás vége.
```

Vegyük észre, hogy ha a szám 2-vel nem osztható, akkor N/2-vel sem lesz, ha 3-mal, akkor N/3-mal sem, ... általában: ha K-val nem osztható, akkor N/K-val sem. Így tehát e K-val addig kell haladnunk, amíg K≤N/K teljesül, vagyis K≤gyök(N). Így az oszthatósági vizsgálatot gyök(N)-nél nagyobbakra felesleges elvégezni. Időben még úgyis nyerünk, ha a ciklusfeltételt egy újabb -nem túl bonyolultan kiértékelhető- tényezővel, a határ vizsgálatával kibővítjük. Persze így prím N-ek esetén a ciklusban nem találunk osztót, ezért ezt külön kell vizsgálnunk!

```
Legkisebb osztó(N,O):
  I:=2
  gyök_N:=SQR(N)
  Ciklus amíg I≤gyök_N és I nem osztója N-nek
    I:=I+1
  Ciklus vége
  Ha I≤gyök_N akkor O:=I különben O:=N
Eljárás vége.
```

Eddig a sorozat elemszámát úgy csökkentettük, hogy a végén levő elemeket hagytuk el. Most más utat választunk.

Jusson eszünkbe az az elemi tény, hogy az **egész számok** (legalább) kétfélék: **párosak vagy páratlanok**, vagyis 2-vel oszthatók vagy 2-vel nem oszthatók, akkor az első oszthatósági vizsgálatunkat felhasználhatjuk arra, hogy a potenciális osztók számát felére csökkentsük. Hisz a páratlan N-hez páros osztót keresni eleve felesleges.

```
Legkisebb osztó(N,O):
  I:=2
  gyök_N:=SQR(N)
  Ha I nem osztója N-nek akkor
       I:=3
       Ciklus amíg I≤gyök_N és I nem osztója N-nek
         I:=I+2
       Ciklus vége
  Elágazás vége
  Ha I≤gyök_N akkor O:=I különben O:=N
Eljárás vége.
```

Csábító „általánosítási lehetőséget" sejthetünk eme gondolat továbbvitelében. Tudni illik, ha megállapítjuk azt is, hogy 3-mal sem osztható, akkor talán még nagyobb lépések-

ben haladhatunk a potenciális osztók keresésénél, esetleg 3-mal növelhetjük az I-t minden lépésben. Sajnos mindjárt az elején pórul járnánk az optimalizáló kísérletünkben, mivel így pl. az 5-tel való oszthatóságot nem vizsgálnánk, sőt általában az ikerprímek (azaz kettő különbségű prímszomszédok) létezése miatt további hibákat is elkövetnénk. Most az döntötte romba nagystílű programötletünket, hogy a prímek túl szabálytalan elhelyezkedésűek. **Ha létezne egy analitikus** (=képlettel leírható) **függvény, amely felsorolná a prímeket**, akkor persze igen hatásos programhoz juthatnánk:

```
Legkisebb osztó(N,O):
  PI:=1; P:=2                [az 1. prímmel (=2-vel) való
                              oszthatóságot fogjuk vizsgálni]
  gyök_N:=SQR(N)
  Ciklus amíg P≤gyök_N és P nem osztója N-nek
    PI:=PI+1; P:=prím(PI)    [P a PI. prímszám]
  Ciklus vége
  Ha P≤gyök_N akkor O:=P különben O:=N
Eljárás vége.
```

A fenti megoldás hibája persze mindenki előtt világos, hogy ti. egy fenti **prím()** függvény megvalósítása munkaigényesebb volna, mint maga az alapfeladat. E program kapcsán támadhat egy újabb gyorsító javaslatunk: ha a fenti függvény nem is létezik, adjuk meg a prímeket előre egy vektorban! Valóban, ezzel a keresésre fordított idő igen hatékonyan lerövidíthető. Ezzel kapcsolatban rögtön az a kérdés merül föl, hogy az időért cserébe „feláldozott" memória mekkora. Erre egy nem feltétlenül elemi ismerettel lehet válaszolni: a prímek „számára N-ig" az N/log N becslést szokták adni, így ha megelégszünk azzal, hogy kb. 2^{14} nagyságrendbe eső számokkal foglalkozunk csupán (aminél többre nem is telik a személyi számítógépek egész típusú változóinak ábrázolásánál), akkor a program többlet memóriaigénye a gyök(2^{14}) számig levő prímek száma, azaz kb. $2^7/\log(2^7)$, ami néhányszor tízes nagyságrendű. A prímek 2^7-ig 41-en vannak.

Nem állítjuk, hogy ezzel a legoptimálisabb megoldást kaptuk, csupán azt, hogy e módszert alkalmazva, az eredetinél lényegesen jobbat kaptunk. Folytassuk az optimalizálást egy, a fenti feladathoz igen hasonlóval!

Feladat: Határozzuk meg egy adott szám legnagyobb valódi osztóját (ami különbözik magától a számtól és nem az 1)!

Megoldás: Térjünk vissza a korábbi feladat első megoldásához, mint alapmegoldáshoz, s azt úgy módosítsuk, hogy most az N-től kezdve visszafelé keressük az első osztót!

```
Legnagyobb osztó(N,O):
  I:=N-1
  Ciklus amíg I nem osztója N-nek
    I:=I-1
  Ciklus vége
  Ha I>1 akkor O:=I
Eljárás vége.
```

E megoldással kapcsolatban is felvethetők azok a gyorsító gondolatok, amelyeket az eredeti feladatnál megbeszéltünk. Ezért most egy másik oldalról közelítjük meg az optimalizálás kérdését. Vegyük szemügyre a fenti algoritmus működését egy konkrét számra! Legyen ez a szám pl. 51! Az I-t elindítva, 50-től visszafelé keressük az első olyan számot, amely osztója az 51-nek: 50, 49, ... 25, 24, ... , 17 és így tovább. Összefoglalva: amíg a lehetséges osztók N-ig sűrűn követik egymást, addig az N-nél nagyobbak egyre ritkábban. Így meglehetősen munkaigényes a legnagyobb osztó keresése, a természetesnek mondható „felülről lefelé" módszerrel.

Az elmondottakból nyilvánvaló a megvalósítás ötlete: keressük meg a legkisebb osztót, amely épp „kiegészítője" lesz a keresett legnagyobbnak. Gondoljunk vissza az alapfeladathoz fűzött első észrevételünkre (most persze egy másik érdekes aspektusból)! Ha 2-vel osztható, akkor N/2-vel is, ha 3-mal, akkor N/3-mal is. Jól látszik, hogy felülről az első lehetséges osztó, az N/2, és a második, az N/3 közötti „távolság" igen nagy lehet, a második N/3 és a harmadik N/4 „távolsága" bár már kisebb, de még mindig nagy, ...

Olyan megoldást kellene találnunk, amely először az N/2-t vizsgálja (hogy osztója-e N-nek), majd azonnal az N/3-at, utána az N/4-et, ... Ez egyenértékű azzal, ha a 2-vel, 3-mal, ... való oszthatóságot nézzük, hiszen **N=legkisebb osztó*legnagyobb osztó**. így tehát a megoldás szó szerint megegyezik a legkisebb osztó meghatározásával, csupán most nem a megtalált P lesz a keresett szám, hanem az N/P.

```
Legnagyobb osztó(N,O):
  PI:=1; P:=2          [az 1. prímmel (=2-vel) való
                        oszthatóságot fogjuk vizsgálni]
  gyök_N:=SQR(N)
  Ciklus amíg P≤gyök_N és P nem osztója N-nek
    PI:=PI+1; P:=prím(PI)   [P a PI. prímszám]
  Ciklus vége
  Ha P≤gyök_N akkor O:=N/P
Eljárás vége.
```

Nézzünk egy más jellegű feladatot, amelyben azonban szintén ugyanilyen jellemzőt kell figyelembe vennünk.

Feladat: Ismerjük egy ország városainak távolságait közúton. Adjuk meg azt a két várost, amelyek a legközelebb vannak egymáshoz!

Megoldás: Tároljuk minden várospárra a távolságukat az A(N,N) mátrixban, ahol A(I,J) az I. és a J. város távolsága, illetve 0, ha a két város között nincs út. Meg kell határozni a mátrix azon elemét, amely a nem nullák közül a legkisebb!

```
Legközelebbiek(N,A(,),V1,V2):
  TAV:=+∞
  Ciklus I=1-től N-ig
    Ciklus J=1-től N-ig
      Ha A(I,J)>0 és A(I,J)<TAV
                  akkor TAV:=A(I,J); V1:=I; V2:=J
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Az A(I,J)=0 kivétel kezelése miatt van egy vizsgálat a ciklusok belsejében. 1.2.2. alapján a kivételes esetet kiküszöbölhetjük, A(I,J):=+∞ utasítással, s így az A(I,J)>0 vizsgálat a ciklusmagból elhagyható.

Vegyük figyelembe, hogy A(I,J)=A(J,I), azaz a mátrix szimmetrikus, valamint, hogy A(I,I) biztosan 0! Ezzel a megvizsgálandó elemek száma a korábbinak kevesebb, mint felére csökken.

```
Legközelebbiek(N,A(,),V1,V2):
  TAV:=+∞
  Ciklus I=1-től N-1-ig
    Ciklus J=I+1-től N-ig
      Ha A(I,J)<TAV akkor TAV:=A(I,J); V1:=I; V2:=J
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Klasszikus feladat mátrixokkal a **Gauss elimináció**. Ha az egyenletrendszer mátrixa azonban tridiagonális mátrix, akkor lehetőség van sokkal gyorsabb megoldás készítésére is.

Feladat: Adjuk meg N elem K-adosztályú kombinációi közül az aktuális kombinációt követő kombinációt!

Megoldás: Keressük meg az előző kombinációban hátulról az első növelhető elemet, a mögötte levőket pedig állítsuk ennél 1-1-gyel nagyobbra:

```
Következő kombináció(KOMB(),N,K):
  H:=1
  Ciklus amíg KOMB(K+1-H)≥N+1-H
    H:=H+1
  Ciklus vége
  KOMB(K+1-H):=KOMB(K+1-H)+1
  Ciklus J=K+2-H-tól K-ig
    KOMB(J):=KOMB(J-1)+1
  Ciklus vége
Eljárás vége.
```

Ha megjegyeznénk minden egyes kombinációnál, hogy utoljára hol volt változás és az az elem mennyire változott, akkor az aktuálisban vagy ezen a helyen kell változtatni, vagy pedig eggyel előtte:

```
Következő kombináció(KOMB(),N,K):
  Ha M<N-H akkor H:=0
  H:=H+1; M:=KOMB(K+1-H)
  Ciklus J=1-től H-ig
    KOMB(K+J-H):=M+J
  Ciklus vége
Eljárás vége.
```

Feladat: Adott egy F file és egy MINTA(MH) szöveg. Adjuk meg a file azon sorait, amelyekben ez a szöveg szerepel!

Ez egy kiválogatási feladat, így a megoldása:

```
Szövegminta keresés(MINTA(),MH,F):
  SS:=0
  Ciklus amíg nem vége(F)
   Olvas(F,SOR,SH); SS:=SS+1
   Ha Eleme(MINTA(),MH,SOR(),SH) akkor Ki: SS,SOR
  Ciklus vége
Eljárás vége.
```

Azt, hogy egy szöveg tartalmaz-e egy rész-szöveget vagy sem, eldöntéssel oldhatjuk meg:

```
Eleme(MINTA(),MH,SOR(),SH):
   I:=1
   Ciklus amíg I≤SH-MH+1 és nem Azonos(MINTA(),MH,SOR(),I)
   I:=I+1
   Ciklus vége
   Eleme:=(I≤SH-MH+1)
Függvény vége.
```

Az azonosságvizsgálat egy újabb eldöntés:

```
Azonos(MINTA(),MH,SOR(),I):
  J:=1
  Ciklus amíg J≤MH és MINTA(J)=SOR(I+J-1)
    J:=J+1
  Ciklus vége
  Azonos:=J>MH
Eljárás vége.
```

Sokkal gyorsabb megoldást is írhatnánk ennél, nem kell ugyanis a mintaszöveget a szövegnek minden adott hosszúságú részsorozatával összehasonlítani, hanem a hasonlításból szerzett információ alapján többeket kihagyhatunk - tehát a **sorozat elemszámát csökkenthetjük.** Először vizsgáljunk meg egy példát!

MINTA: AAABA

SOR: ZXCAAZAA

Ha a MINTA és a SOR összehasonlításában itt tartunk, akkor a MINTA 3. A-betűjénél vesszük észre, hogy a szövegben itt nem a mintaszöveg kezdődik. Ekkor azonban felesleges megnézni az eggyel jobbratolt mintával való egyezést, sőt a kettővel jobbratoltat sem érdemes vizsgálni. Ezt felhasználva a minta minden betűjéhez megadhatunk egy számot, amennyivel később kell kezdeni az összehasonlítást, ha annál a betűnél eltérést tapasztaltunk. A fenti példában ez így néz ki:

AAABA

12315

Tároljuk ezeket a számokat a LEPES() vektorban! Ekkor a megoldás második szintje igy alakul:

```
Eleme(MINTA(),MH,SOR(),SH):
  I:=1
  Ciklus amíg I≤SH-MH+1 és nem Azonos(MINTA(),MH,SOR(),I,J)
    I:=I+LEPES(J)
  Ciklus vége
  Eleme:=(I≤SH-MH+1)
Függvény vége.
```

Ebben az esetben az azonosságvizsgálatnak meg kell adnia azt a J értéket, ahol a minta eltért a vizsgált szöveg megfelelő betűjétől!

Kicsit ügyesebb megoldást kaphatunk, ha a hasonlítást a minta legutolsó betűjével kezdjük, s visszafelé haladunk. Ekkor a fenti példában a következő lépésszámokat kapjuk:

AAABA

43221

vagy egy másik példát vizsgálva:

CBAABAAA

87444111

A lépésszámot a következőképpen kell meghatározni:

1. Ha a betű a mögötte levők között nem fordul elő, akkor annyi, amennyi a sorszáma hátulról.

2. Ha a mögötte levők között előfordul, akkor legalább annyi, mint az őt követő betű lépésszáma. Ekkor lépjünk tovább ennyit, majd még annyit adjunk hozzá a lépésszámhoz, ahánnyal később találunk vele azonosat. Ha ettől a ponttól nincs vele azonos, akkor alkalmazzuk az első szabályt!

Ekkor a megoldás 2. és 3. szintjét kell változtatni.

```
Eleme(MINTA(),MH,SOR(),SH):
  I:=hossz(MINTA)
  Ciklus amíg I≤SH és nem Azonos(MINTA(),MH,SOR(),I,J)
    I:=I+LEPES(J)
  Ciklus vége
  Eleme:= I≤SH
Függvény vége.

Azonos(MINTA(),MH,SOR(),I,J):
  J:=MH; K:=I
  Ciklus amíg J≥1 és MINTA(J)=SOR(K)
    J:=J-1; K:=K-1
  Ciklus vége
  Azonos:= J<1
Függvény vége.
```

1.2. Sorozat részekre osztása

Ebben a fejezetben olyan feladatokat vizsgálunk, amelyeknél **egy sorozathoz egy értéket vagy egy sorozatot kell rendelni.**

Sokszor nagy segítséget jelenthet a végrehajtási idő csökkentéséhez, ha kihasználjuk a rendelkezésünkre álló adatok valamilyen **speciális tulajdonságát.** Mi lehet ez a tulajdonság:

- **Tudjuk, hogy a feldolgozás során keresett elem a sorozat mely részében lehet** (azon belül esetleg ugyanezt újra tudjuk), illetve ezt könnyen eldönthetjük.
- **A sorozatot úgy átrendezhetjük, hogy a feldolgozás során vizsgálandó részek körülbelül azonos elemszámúak legyenek, s a feldolgozást a részekre kelljen elvégezni** (pl. a **Quicksort** algoritmus).

Feladat: Egy **rendezett** sorozatból kell kiválasztani egy konkrét elemet (azt tudjuk, hogy **biztosan benne van** a sorozatban -egyszer-, s arra vagyunk kíváncsiak, hogy hányadik).

Megoldás: A kézenfekvő megoldás a **lineáris kiválasztás** (X kiválasztása az A(N) vektorból, K lesz a sorszáma):

```
Kiválasztás(K,X):
  K:=1
  Ciklus amíg A(K)≠X
    K:=K+1
  Ciklus vége
Eljárás vége.
```

A **maximális hasonlításszám N**, a **minimális 1** lesz, az **átlagos** pedig **N/2** körüli (ha a különböző elemeket egyenlő gyakran kell keresni). Bár ebben a példában a végrehajtási idő már a bemenő adatoktól is függ, most egyelőre még ne törődjünk ezzel. A következő átalakítással azt a tulajdonságot használjuk ki, hogy a számsorozat **rendezett**. Az előző megoldásban meg tudtuk határozni azokat az elemeket, amelyeket már biztosan nem kell végignézni. Most lépésenként rész-szakaszokat fogunk elhagyni a vektorból, így a még megvizsgálandó elemek száma lépésenként nem eggyel fog csökkeni, hanem kb. feleződni fog. (A módszert az összehasonlítások számát leíró függvény miatt **logaritmikus keresésnek** is nevezik.)

```
Kiválasztás(K,X):
  A:=1; F:=N
  Ciklus
    K:=INT((A+F)/2)
    Ha A(K)<X akkor A:=K+1
    Ha A(K)>X akkor F:=K-1
  amíg A(K)≠X
  Ciklus vége
Eljárás vége.
```

A lényeg tehát: válasszuk ki a középső elemet, s ennek vizsgálata alapján döntsük el, hogy az előtte vagy a mögötte levők között kell-e tovább keresni!

Ez az elv nagyon sok feladatnál használható, ahol egy sorozathoz kell egyetlen értéket rendelni. Hasonló elv alapján adható meg egyszerre egy sorozat minimális és maximális értékű eleme is.

Nézzünk ugyanerre egy kissé más jellegű problémát, amelyből kiderül, hogy igazából nem is a rendezettség a fontos összekötője ezen feladatoknak, hanem valami, ami a rendezettségből is következik.

Feladat: Adott egy folytonos függvény, amely egy intervallum két végpontjában különböző előjelű (F(A)*F(B)<0). Keressünk az [A,B] intervallumon egy olyan helyet, amely a függvény egy gyökhelyétől E-nél kevesebbel tér el! Az első, egyszerű megoldás:

```
Gyökkeresés(A,B,K):
    K:=A+E
    Ciklus amíg F(K)*F(K-E)>0
        K:=K+E
    Ciklus vége
Eljárás vége.
```

Felezzük most is minden egyes lépésben a vizsgált intervallumot! Az előző feladathoz hasonlóan itt is meg tudjuk állapítani, hogy **melyik részintervallumban kell tovább keresni** (tehát ez volt a fő szempont, nem pedig a rendezettség, azaz nem szükséges feltétel a függvény monotonitása).

```
Gyökkeresés(A,B,K):
  Ciklus
    K:=(A+B)/2
    Elágazás
      F(A)*F(K)>0 esetén A:=K
      F(B)*F(K)>0 esetén B:=K
    Elágazás vége
  amíg B-A>E és F(K)≠0
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Vegyük észre az utóbbi algoritmuspárok szembetűnő formai hasonlatosságát, és hogy a feladatok különbözősége ellenére mennyire *egyfajta* gondolkozással készültek!

Nem csak a felezés lehet jó módszer, pl. nemlineáris egyenletek húrmódszerrel történő megoldásakor is részekre bontjuk az intervallumot, de nem felezzük.

Az újabb feladatban kicsit nehéz a felezendő intervallum meghatározása.

Feladat: Egy szekvenciális file-ban 1 és 1000000 közötti egész számok találhatók. 1000000 szám nem fér be a számítógép központi memóriájába. Adjunk meg egy olyan számot, ami nem szerepel ebben a szekvenciális állományban!

Megoldás: A legegyszerűbb megoldásban legfeljeb 1000000-szor végigolvassuk az állományt, s az első olyan számnál, amit nem találtunk az állományban, megállunk. Azt kell észrevenni, hogy a felezendő intervallumot nem az állományban levő számok határozzák meg, hanem az az intervallum, ahol a keresett szám lehet. Így a megoldásban számoljuk le, hogy hány szám van 1 és 500000 között, illetve 500001 és 1000000 között, s amelyik intervallumban kevesebbet találtunk, abban biztosan van még fel nem használt szám, tehát arra az intervallumra alkalmazzuk ugyanezt az eljárást!

Polinomok szorzására nézzünk egy érdekes módszert! A polinomokat bontsuk két részre:

$$p(x) = p_1(x) + x^{N/2}p_2(x) q(x) = q_1(x) + x^{N/2}q_2(x)$$

Számítsuk ki a következő három részszorzatot (N/2-edfokú polinomokat kell szorozni):

$$r_1(x) = p_1(x) * q_1(x)$$
$$r_2(x) = (p_1(x)+p_2(x)) * (q_1(x)+q_2(x))$$
$$r_3(x) = p_2(x) * q_2(x)$$

Ekkor a szorzatpolinom a következő összegként áll elő:

$$r(x) = r_1(x) + (r_2(x)-r_1(x)-r_3(x)) * x^{N/2} + r_3(x) * x^N$$

Mivel N-edfokú polinomok szorzása N^2 lépést igényel, ezért a fenti képletet továbbvíve N*log N lépésszámmal egy gyorsabb szorzást kapunk.

A részekre osztás elvét **nem csak két részre osztás**nál alkalmazhatjuk: ABC-sorrendbe rendezett szövegek keresésekor célszerű lehet például minden egyes betűhöz megadni az azzal kezdődő szavak közül a legelsőnek a sorszámát (címét), s ezt falhasználni a kereséskor.

Megjegyzés: Ezt a módszert a szakirodalom gyakran hívja „Oszd meg és uralkodj (Divide et impera)" módszernek is.

1.3. Sorozatok párhuzamos feldolgozása

Itt nem egyetlen sorozatot kell vizsgálni, hanem kettőt (vagy többet). A két sorozat elemeit valamilyen szempont alapján egymáshoz rendeljük, s a két sorozat ezen szempont

szerint rendezett. (Ez az eset fordul elő, ha két rendezett sorozatként ábrázolt halmazra kell valamilyen halmazműveletet elvégeznünk - egyesítést, metszetet, stb.)

A megoldás ebben az esetben mindig az lesz, hogy haladjunk a két (vagy több) sorozatban egyszerre, mindig abban tovább- lépve, amelyben szükséges.

Feladat: Adott két, rendezett sorozatként ábrázolt halmaz, adjuk meg az egyesítésüket!

Megoldás: Az alapmegoldást az **egyesítés** (unió) **tételének** alkalmazásával kapjuk (A() N elemű, B() M elemű, C() N+M elemű):

```
Egyesítés(N,A(),M,B(),C()):
  C():=A(); K:=N
  Ciklus J=1-től M-ig
    I:=1
    Ciklus amíg I≤N és A(I)≠B(J)
      I:=I+1
    Ciklus vége
    Ha I>N akkor K:=K+1; C(K):=B(J)
  Ciklus vége
Eljárás vége.
```

Ez a megoldás tehát azon alapul, hogy az egyesítésben az A() halmaz elemeit, valamint a B() halmaz A()-ban nem szereplő elemeit kell elhelyezni.

A hasonlítások száma itt **legfeljebb M*N** (ha egyáltalán nincs közös elem), illetve **legalább M*(M+1)/2** (N≥M esetén) vagy **N*(N+1)/2+(M-N)*N** (N<M esetén) (ez akkor fordul elő, ha B() minden eleme szerepel A()-ban, illetve fordítva).

Hogyan használhatnánk ki a rendezettséget? Induljunk ki a C() halmaz elemeiből! Első eleme mindenképpen meg fog egyezni vagy az A(), vagy a B() halmaz első elemével. A megoldás erre fog épülni: az első elemek közül a kisebbet tegyük bele az eredményhalmazba, majd vegyük hozzá a maradékhalmazok egyesítését! Ez tulajdonképpen az **összefuttatás tétele** lesz. Helyezzünk A() és B() végére is egy-egy végelemet, amelyek majd az összefuttatás leállását biztosítják!

Megjegyzés: Most és a továbbiakban +∞ jelöli tetszőleges adattípus értékkészletének legnagyobb elemét, -∞ pedig a legkisebbet.

```
Egyesítés(N,A(),M,B(),C()):
  A(N+1):=+∞; B(M+1):=+∞; I:=1; J:=1; K:=0
  Ciklus amíg I<N+1 vagy J<M+1
    K:=K+1
```

```
    Elágazás
      A(I)<B(J) esetén C(K):=A(I); I:=I+1
      A(I)=B(J) esetén C(K):=A(I); I:=I+1; J:=J+1
      A(I)>B(J) esetén C(K):=B(J); J:=J+1
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

A hasonlítások száma (az elágazás feltétele kiértékelését 1 egységnek véve) itt **legfeljebb** N+M, illetve **legalább maximum(N,M)**.

Állítás: A hasonlítások számát -a 3-irányú elágazás megvalósításától eltekintve- nem lehet tovább csökkenteni! (Ez ekvivalens azzal, hogy ha az adatokat szekvenciális file-okból olvassuk, akkor az olvasások száma tovább nem csökkenthető.)

Az állítás bizonyítása nagyon egyszerű, ugyanis a két adatállomány beolvasásához N+M+2 műveletre mindenképpen szükség van.

Feladat: Adott két, rendezett sorozatként ábrázolt halmaz, adjuk meg a metszetüket! (Ez szintén legyen rendezett!)

Megoldás: A megoldás ugyanarra az ötletre épül, mint az előző feladat megoldása, a **metszet tétel** (A() N elemű, B() M elemű, C() min(N,M) elemű):

```
Metszet(N,A(),M,B(),C()):
  K:=0
  Ciklus I=1-től N-ig
    J:=1
    Ciklus amíg J≤M és A(I)≠B(J)
      J:=J+1
    Ciklus vége
    Ha J≤M akkor K:=K+1; C(K):=A(I)
  Ciklus vége
Eljárás vége.
```

helyett itt is a rendezettségre építünk:

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

Ugyanezt az elvet alkalmazzák az **összefésülő rendezések**, illetve szekvenciális file-ok **időszerűsítő** algoritmusai.

1.4. Gyakoriság szerinti elrendezés

A ciklusmag végrehajtási száma az eddigi példákban nem függött a bemenő adatoktól (legalábbis nem használtuk ki), most egy olyan esetet vizsgálunk, amikor már függ, azaz van értelme beszélni **minimális, maximális**, illetve **átlagos végrehajtási időről**. Ebben a módszercsaládban tehát az **adatsorozatokat a feldolgozási igények, gyakoriságok alapján megfelelő sorrendbe rendezzük.** Gyakran használhatjuk ezt a módszert keresési feladatoknál.

Feladat: Egy táblázatban összetartozó értékpárokat tárolunk. A táblázat első oszlopában szereplő értékek valamelyikéhez kell megkeresnünk a hozzá tartozó értéket! (Tudjuk, hogy a keresett érték a táblázatban megtalálható.)

Megjegyzés: Ilyen feladat például egy telefonkönyv kezelése, - amikor névhez kell keresni telefonszámot; vagy egy gépkocsinyilvántartás, ahol rendszámhoz kell keresni tulajdonost stb.

Megoldás: Ha a következő egyszerű keresési eljárást tekintjük, hogyan használhatjuk ki azt a tényt -mert ebben fejeződik ki a futási idő adatfüggése-, hogy a keresés során egyes elemek megkeresésére gyakrabban lesz szükség, mint másokra?

```
Keresés(X):
  K:=1
  Ciklus amíg K≤N és A(K,1)≠X
    K:=K+1
  Ciklus vége
  Ha K≤N akkor Ki: A(K,2)
Eljárás vége.
```

Ebben az esetben, ha az elemeket **egyenlő gyakran** kellene keresni (azaz sok keresés átlagában ugyanannyiszor az elsőt, mint pl. az utolsót, a másodikat, mint az utolsóelőttit, ..., s mindegyik egyenlő S-sel), akkor az átlagos keresési idő, amit számunkra most a szükséges hasonlítások átlagos száma fog jelenteni:

$$T = \sum_{i=1}^{N} \frac{i * S}{N * S} = \frac{1}{N} * \sum_{i=1}^{N} i = \frac{N+1}{2}$$

Ha azonban **nem egyenlő gyakorisággal** kell keresni, akkor a gyakrabban szükségeseket előre kell vennünk, s így általában hamarabb megtalálhatók, míg a kevésbé gyakran

szükségeseket hátra. Ezek megtalálásához ugyan több idő szükséges, de mivel ritkán van rájuk szükség, ez az idő nem nagy veszteség a nyereséghez képest.

Annak belátása, hogy a gyakoriság ismeretében ezt az „előrendezést" elvégezve valóban nyereséghez jutunk, igen egyszerű dolog:

A számoláshoz jelöljük S_1, ..., S_n-nel az 1., ..., N. elem keresésének gyakoriságát! Vagyis, hogy az összes (S_1+ ... +S_n) keresésből hány ízben volt az 1., ..., N. kiválasztására szükség. így az összes összehasonlítások száma $1*S_1+2*S_2+ ...+N*S_n$ (feltételeztük, hogy a mátrixban meg nem található elemeket eleve nem keresünk). Ebből az átlagos keresési számra kapjuk:

$$T = \sum_{i=1}^{N} \frac{i * S_i}{S'}$$, ahol S'= Σ S_i az összes keresések száma.

Erről kell bebizonyítanunk, hogy valóban minden fentitől (lényegileg) eltérő sorrend esetén a T nagyobb lesz! Ez egyszerűen következik abból, hogy akárcsak két (különböző gyakoriságú) elemet megcserélünk, T növekedni fog. Legyen e két felcserélendő elem az **i.**, illetve a **j.**, úgy, hogy $S_i>S_j$ és **i<j** (éppen ez a feltételezett jó sorrend a két elemre nézve). A T-t megadó összegből minket csak a két tagot tartalmazó részletösszeg érdekel, azaz az $i*S_i+j*S_j$, illetve a $j*S_i+i*S_j$. Ezeket egymásból kivonva, összevonás után adódik, hogy $i*(S_i-S_j)+j*(S_j-S_i)<0$ (figyelembe véve az S_i, S_j és az i, j egymáshoz való viszonyát). Vagyis az eredeti részletösszeg kisebb, mint a fordított sorozatnál!

Hogy ezt számokkal is illusztráljuk, tételezzük föl az alábbi ismereteket egy konkrét esetről! Az 1. elemre N, a 2.-ra N-1, a 3.-ra N-2, ... alkalommal van szükségünk. Ekkor a hasonlítások átlagos száma a következőképpen alakul (a keresések száma: **N+(N-1)+(N-2)+...+1 = N*(N+1)/2**):

$$T = \frac{2}{N*(N+1)} * \sum_{i=1}^{N}(N-i+1)*i = \frac{2}{-N*(N+1)} * \left(\sum_{i=1}^{N} N*i - \sum_{i=1}^{N} i^2 + \sum_{i=1}^{N} i\right) =$$

$$= \frac{2}{N*(N+1)} * \left(N * \frac{N*(N+1)}{2} - \frac{N*(N+1)*(2*n+1)}{6} + \frac{N*(N+1)}{2}\right) =$$

$$= \frac{2}{N*(N+1)} * \frac{N*(N+1)}{2} * \left(N - \frac{2*N+1}{3} + 1\right) = N+1-\left(\frac{2*N+1}{3}\right) = \frac{N+2}{3}$$

E megoldásnál gyakori probléma, hogy sokszor a keresendő elemek gyakorisága nem állandó, hanem folytonosan változik. Így szükségességük gyakoriságát mindig újra meg kellene adni, ami igen sok munkát jelentene. Ha tárolnánk minden egyes elemhez, hogy eddig hányszor volt rá szükség, akkor e szempont szerint sorba tudnánk rendezni őket, de

ez munkaigényessége miatt nem az igazi megoldás. A következő program a keresés közben automatikusan módosítja a sorrendet egy egyszerű elv alapján: minden egyes megtalált elemet megcserél a sorban eggyel előtte állóval (ha nem a legelső volt). így azok, amelyekre gyakran van szükség, a vektor elején csoportosulnak, míg a ritkán előfordulók hátrafelé mozognak, s a végén találhatók meg.

```
Keresés(X,K,VAN):
  K:=1
  Ciklus amíg K≤N és A(K,1)≠X
    K:=K+1
  Ciklus vége
  VAN:=(K≤N)
  Ha K>1 akkor Csere(A(K),A(K-1))
Eljárás vége.
```

Megjegyzés: A **Csere** eljárás a mátrix K. és K-1. sorát cseréli fel! (az A(K) a teljes sort jelöli!)

Ilyen probléma felmerülhet háttértáron (lemezen) való keresésnél. Ha lemezről olvasunk, akkor a végrehajtási idő szempontjából nem mindegy, hogy az olvasófejnek kell-e mozognia, s ha igen, mennyit. A cél az lehet, hogy a leggyakrabban keresett elemekhez a fej viszonylag keveset mozogjon. Ebben az esetben az a célszerű elrendezés, hogy a leggyakoribb elemet a lemezen az adatsor közepére helyezzük el, s a továbbiakat tőle jobbra, illetve balra, a keresési gyakoriság szerint csökkenő sorrendben. Ezt a sorrendet hívják **orgonasíp elrendezésnek**.

Minden folytatólagos keresésénél is ugyanez az elrendezés a leghasznosabb.

A gyakoriság szerinti elrendezést másképpen is megvalósíthatjuk. Sokszor egy keresési feladatban célszerű a leggyakrabban szükséges elemeket **külön táblázatban** tárolni, s rájuk egy egészen **más keresési algoritmust** használni!

Speciális információt használnak fel a MI területén a heurisztikus keresési módszerek is.

Bináris fában keresésnél is felhasználhatjuk a fa gyakoriság szerinti felépítését a következőképpen:

Minden egyes részfára legyen igaz, hogy a gyökérelem hivatkozási gyakorisága nagyobb vagy egyenlő, mint a leszármazottai hivatkozási gyakorisága, továbbá a gyökérelemtől balra levő elemek mindegyike legyen kisebb a jobbra levő elemek mindegyikénél! A fa egy eleme négy adatot tartalmazzon: egy értéket, a jobboldali részfa legkisebb elemét, valamint a bal-, illetve a jobboldali részfára mutató értéket!

Ekkor a keresőalgoritmus így nézhet ki:

```
Keresés(FA,KER,CIM):
  Ha          FA^.ÉRTÉK=KER      akkor CIM:=FA
  különben Ha FA^.JOBBRA≤KER akkor Keresés(FA^.JOBB,KER,CIM)
  különben Keresés(FA^.BAL,KER,CIM)
Eljárás vége.
```

1.5. Sorozat elemeinek csoportos feldolgozása

Nem csak a rendezettség lehet olyan specialitás, aminek felhasználásával csökkenthető a ciklus végrehajtási száma. A következő kiválogatási feladatban a specialitás az lesz, hogy a megfelelő elemekről tudjuk, hogy biztosan egymás mellett vannak. (Az adatok valamilyen szempont szerint csoportosítva vannak.)

A **keresésnél** akkor használhatjuk a **csoportosítás elvét**, ha a keresési feltételnek meg nem felelő elemeket tudjuk olyan csoportokba foglalni, amelyeket egyetlen elem megvizsgálása után a keresés átléphet. Ugyanez igaz **eldöntési, kiválasztási, megszámlálási, kiválogatási** feladatokra is. **Rendezési** feladatoknál akár a csoportokon belüli rendezettség, akár a csoportok egymáshoz képesti rendezettsége kihasználható.

Feladat: Egy bányászati vállalat egy földterületen kutatófúrásokat végzett az ásványvagyon megállapítása céljából. Minden fúrásnál X méterenként vettek mintát és feljegyezték K fontos ásvány mennyiségét. Az adatok fúrólyukakon belül a mélység szerint sorba vannak rendezve, de az egyes fúrólyukak adatai bármilyen sorrendben lehetnek. Adjuk meg az S sorszámú fúrólyuk adatait!

Megoldás: Legyen összesen N mérésünk, ahol minden mérés K+2 adatot tartalmaz: fúrólyuk sorszáma, mélység, mennyiségek. A feladat egy kiválogatás:

```
Kiválogatás(N,K,A(,)):
  Ciklus I=1-től N-ig
    Ha A(I,1)=S akkor Ki: A(I,)
  Ciklus vége
Eljárás vége.
```

Használjuk ki az adatok speciális tulajdonságát! Ilyen sorszámú fúrólyuk biztosan van, s ennek adatai egymást követik. Így egy kiválasztás után még egy keresést kell alkalmaznunk, s a keresés közben az átlépett elemeket kiírni!

```
Kiválogatás(N,K,A(,)):
    I:=1
    Ciklus amíg A(I,1)≠S
        I:=I+1
    Ciklus vége
    Ciklus amíg I≤N és A(I,1)=S
        Ki: A(I,)
        I:=I+1
    Ciklus vége
Eljárás vége.
```

Megjegyzés: Ha ezt az adathalmazt rendezni szeretnénk, akkor is figyelembe vehetnénk ezt a speciális tulajdonságot, s csak a rendezett darabokat kellene egymás mögé tenni, illetve ugyanezt alkalmazhatnánk megszámolásra is!

Tehát a rendezésnél a **csoportosítás** elvét egyrészt akkor használhatjuk, ha a rendezendő adatok **rendezett csoportokban** állnak rendelkezésre. A másik gyakori esetben a csoportok lehetnek rendezetlenek, de a csoportok sorrendje már biztosan jó. Ekkor a rendezést csak a csoportokon belül külön-külön kell elvégezni, ami a rendezések lépésszáma és a rendezendő elemek száma közötti nemlineáris kapcsolat miatt nyilvánvaló nyereség a teljes sorozat rendezéséhez képest.

Egy egészen más feladat megoldása során is a csoportos feldolgozás elvét használhatjuk.

Feladat: Adjunk össze sorozatként tárolt sokszámjegyű számokat!

Megoldás: Ha a számokat vektorokban tároljuk számjegyenként, akkor két N számjegyből álló szám összeadásához N műveletre, illetve még egy átvitelképzésre van szükség.

Csoportosítsuk a számok számjegyeit négyes csoportokba! Ezek a számjegycsoportok még ábrázolhatók egy egész típusú változóban, sőt még össze is adhatók. Ezzel az összeadást N/4 művelettel elvégezhetjük, mint egy 10000-es számrendszerben való összeadást.

Ebbe a feladattípusba tartozik a **visszalépéses keresés** algoritmusa is (backtrack), amelyet a következő példán keresztül szemléltetünk:

Feladat: Helyezzünk el egy 4x4-es sakktáblán 4 vezért úgy, hogy egyik se üsse a másikat!

Megoldás: Nyilvánvaló, hogy a sakktábla minden oszlopába egy vezért kell elhelyezni, ezért a feladat szerint elég azt megmondani, hogy az I. vezért az I. oszlop melyik sorába tegyük.

Ez egy keresési feladat: számnégyesek közül kell kiválasztani egy adott tulajdonságút. A lehetséges számnégyesek:

1111, 1112, 1113, 1114, 1121, 1122, 1123, 1124,

...

2411, 2412, 2413, 2414, 2421, 2422, 2423, 2424,

...

4431, 4432, 4433, 4434, 4441, 4442, 4443, 4444.

Ezekből azonban 4^4 (256) db van, amit elég sokáig tarthat megvizsgálni. Felfedezhetünk azonban a számnégyesekben jól meghatározható csoportokat:

Mindegyik biztosan rossz:

11xx alakúak, 12xx alakúak, 131x alakúak, 132x alakúak,

...

2411, 2412, ...

A feladat egy lehetséges megoldása: 2413.

A probléma ezen csoportokkal az, hogy nem azonos méretűek, sőt előre általában nem is határozhatók meg. Ha azonban a lehetséges megoldás számait balról jobbra egyesével határozzuk meg, akkor minden esetben el tudjuk dönteni, hogy ezzel elérkeztünk-e egy csoporthoz vagy sem.

Hasonlítsuk tehát össze a kétféle kiválasztást (tudjuk, hogy biztosan van megoldás):

```
Lineáris keresés:
  SZ:=az első számnégyes
  Ciklus amíg SZ nem megoldás
   SZ:=a következő számnégyes
  Ciklus vége
  Ki: SZ
Eljárás vége.
```

```
Visszalépéses keresés:
  I:=1
  Ciklus amíg SZ nem kész
    Következő(SZ,JEGY,I,VAN)
    Ha VAN akkor SZ(I):=JEGY: I:=I+1
        különben SZ(I):=0: I:=I-1
  Ciklus vége
  Ki: SZ
Eljárás vége.
```

A **Következő** eljárás a VAN változót igazra állítja, ha a számnégyesnek meg tudta határozni a következő jegyét, hamis, ha nem. Egy a fentiekben definiált csoport akkor áll tehát elő, amikor a VAN változó hamis értékű lesz (ekkor a csoport tagjai -következő számok- előállításával nem kell tovább foglalkozni), illetve ha megtaláltuk a megoldást.

1.6. Ciklustranszformálás - indexelés

Ezeknél a feladatoknál mindig az lesz a lényeg, hogy egy **keresési, eldöntési feladatot indexeléssé alakítunk**, azaz egyetlen képlettel meghatározhatjuk a keresett elem helyét a

sorozatban. Kétféle esetet vizsgálunk meg: az egyikben a képlet **pontosan megadja** a keresett elem helyét, a másikban csak **„körülbelül"**.

Feladat: Adjuk meg N-ig az ikerprímeket! (Ikerprím az a két prímszám, amelyek különbsége 2.)

Megoldás: A legegyszerűbb megoldásban sorra vesszük a számokat 2-től, megvizsgáljuk, hogy az adott szám prím-e, valamint a nála 2-vel nagyobb szám prím-e. Az ilyen számokat kiírjuk.

```
Ikerprímek:
  Be: N              [N>3, egész]
  Ciklus I=2-től N-ig
    Ha prímszám(I) és prímszám(I+2) akkor Ki: I,I+2
  Ciklus vége
Program vége.

Prímszám(I):
  J:=2
  Ciklus amíg J*J≤I és J nem osztója I-nek
    J:=J+1
  Ciklus vége
  Prímszám:=(J*J>I)
Függvény vége.
```

A megoldás első szintjét -matematikai ismereteink felhasználásával- gyorsabbra írhatjuk. (Az ikerprímek csak páratlanok lehetnek, ha I prímszám és I+2 nem az, akkor a következő lehetőség az I+4.)

```
Ikerprímek:
  Be: N              [N>3, egész]
  I:=3
  Ciklus amíg I≤N
    Ha prímszám(I) akkor
            Ha prímszám(I+2) akkor Ki: I,I+2: I:=I+2
                           különben I:=I+4
            Elágazás vége
       különben I:=I+2
    Elágazás vége
  Ciklus vége
Program vége.
```

Ebben a megoldásban annak eldöntése, hogy egy szám prím-e, egy ciklusban történik. Ezt a ciklust meg lehet szüntetni, ha jobb adatszerkezetet választunk. Ha feltesszük, hogy N≤1000, akkor tároljuk a prímeket 1001-ig a PRIM(K) vektorban! Ekkor annak eldöntése, hogy egy szám prím-e, annyiból áll, hogy megnézzük, benne van-e a PRIM() vektor-

ban. Még egyszerűbb megoldást is találhatunk, ha eleve csak a PRIM() vektor elemeire vizsgáljuk meg, hogy két egymás utáni különbsége 2-e.

```
Ikerprímek:
  Be: N          [N>3, egész]
  PRIM(K) feltöltése: I:=1
  Ciklus amíg PRIM(I)≤N
    Ha PRIM(I+1)-PRIM(I)=2 akkor Ki: PRIM(I),PRIM(I+1)
  Ciklus vége
Program vége.
```

Megjegyzés: Vegyük észre, hogy egy korábbi feladatban szintén ezt (prímek tárolása) használtuk ki a gyorsításra. Mindkét esetben a **kereső, eldöntő ciklust transzformáltuk indexeléssé**. Ezt azonban nem mindig tehetjük meg. Több esetben segíthet azonban a **ciklusok felcserélése**.

Feladat: Utasszámlálást végeztünk autóbuszokon. Feljegyeztük, hogy milyen sorszámú buszon hány utas utazott. Ugyanolyan sorszámú buszt többször is megfigyeltünk. Adjuk meg az egyes buszok átlagos utasszámát!

Megoldás: Legyen N a megfigyelések száma, BS a buszok száma, MEGF(N,2) a konkrét megfigyelések (buszsorszám, utasszám)! Helyezzük el MFDB(BS)-ben a buszok megfigyeléseinek számait, UTDB(BS)-ben pedig, hogy összesen hányan utaztak az egyes buszokon!

A megoldás 1. változatában nézzük végig az egyes buszjáratokat! **Válogassuk ki** azokat a megfigyeléseket, amelyekben az adott buszt figyeltük meg, s **számoljunk** a megfelelő változókban!

```
Buszjáratok(N,MEGF(,),BS,MFDB(),UTDB()):
  Ciklus I=1-től BS-ig
    Ciklus J=1-től N-ig
      Ha MEGF(J,1)=I akkor MFDB(I):=MFDB(I)+1
                           UTDB(I):=UTDB(I)+MEGF(J,2)
      Elágazás vége
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Ebben a megoldásban a **kiválogatás** az, amely sok időt elvisz. Ezt a kiválogatást azonban nem helyettesíthetjük egyetlen indexeléssel (hiszen többször is megfigyelhettünk egy adott sorszámú buszt).

A megoldás 2. változatában **cseréljük fel a két ciklust**!

```
Buszjáratok(N,MEGF(,),BS,MFDB(),UTDB()):
  Ciklus J=1-től N-ig
    Ciklus I=1-től BS-ig
      Ha MEGF(J,1)=I akkor MFDB(I):=MFDB(I)+1
                           UTDB(I):=UTDB(I)+MEGF(J,2)
      Elágazás vége
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Tehát most megfigyeléshez keresünk buszjáratot. Ebből viszont garantáltan 1 van, sőt tudjuk is, hogy melyik az.

Így a 3. változatban a belső ciklust egyetlen indexeléssel helyettesítjük.

```
Buszjáratok(N,MEGF(,),BS,MFDB(),UTDB()):
  Ciklus J=1-től N-ig
    I:=MEGF(J,1)
    MFDB(I):=MFDB(I)+1
    UTDB(I):=UTDB(I)+MEGF(J,2)
  Ciklus vége
Eljárás vége.
```

Nézzünk meg röviden egy összetettebb példát is a számítógépes szimuláció témaköréből: a diffúziót síkban!

Egy nagyon egyszerű gázmodell szerint a molekulákat egy táblázatban ábrázoljuk, pl. A(I,J)=1, ha az (I,J) helyen van molekula, és 0, ha nincs. A modellben válasszunk ki egy véletlen helyet, s az itt levő molekulát mozgassuk. Ennek az elképzelésnek az a hibája, hogy a véletlenszerű választással gyakran üres helyet választunk, s így lassú lesz a szimuláció. Másik elképzelés szerint ne a helyekről tároljunk adatokat, hanem a molekulákról! Adjuk meg minden molekulához a koordinátáit! Ebben a modellben viszont sokáig tart eldönteni azt, hogy a molekula ütközik-e egy másik molekulával. Bármelyik megoldást is választjuk, mindenképpen lesz egy olyan ciklus, amely feleslegesen nagy lépésszámú. Az igazi megoldás: **tároljuk mindkét információt**, mert ezzel a molekulaválasztó és az ütközésvizsgáló ciklus lépésszámát is 1-re csökkenthetjük!

Ugyanezt az elvet használja egy egyszerű rendezési módszer: a rekeszes rendezés. Ez a beillesztéses rendezés egy variációja, ahol a beillesztő ciklust egy indexelés helyettesíti. A rendezendő adatok legyenek 1 és M közötti számok!

```
Rendezés(N,A(),M,B()):
  B():=0
  Ciklus I=1-től N-ig
    B(A(I)):=B(A(I))+1
  Ciklus vége
Eljárás vége.
```

Az eljárás végére érve a B() vektor tartalmazza, hogy az 1 és M közötti számok hányszor fordultak elő A() elemei között, így a rendezés B() nullázásától eltekintve (ami kezdőértékadásként akár fordítási időben is elvégezhető) N lépés alatt megoldható.

A másik feladatcsoportban nem tudunk pontos indexet mondani a keresett elemhez, de **közelítő indexet** igen!

Ha például az összes magyar állampolgár adatait kellene tárolni, s személyi szám szerint keresni közöttük, akkor a logaritmikus keresés is igen nagy futási időt jelentene, nem is beszélve a rendezéshez szükséges időről. A leggyorsabb megoldás persze az lenne, ha felvehetnénk egy akkora tömböt, amelyet a személyi számmal indexelhetnénk, ez azonban több nagyságrenddel több helyet igényel, mint ahány magyar ember van. Olyan függvényt valószínűleg nem találunk, amely a lehetséges személyi számokhoz egyértelműen egy sorszámot rendel, de valami hasonlót tehetünk.

Rendeljünk minden személyi számhoz ($A_1A_2...A_{11}$) a következőképpen egy értéket:

(A_1-1)*10000000 + ([A_2...A_7] * [A_8...A_{10}]) 7 középső jegye!

Ez az érték egy 0 és 20000000 közötti egész szám, a lehetséges indexeknek csupán kétszeresét foglalja magába. Semmi sem biztosítja azonban, hogy két különböző személyi számhoz így különböző értéket kapunk.

Az adatok elhelyezését úgy végezzük, hogy az adatot a kiszámított címre helyezzük el, ha az még nem foglalt! Ha az már foglalt lenne, akkor pedig ettől a helytől lineárisan keressünk egy szabad helyet, s oda tegyük! Ha ez a transzformáció elég „jó", azaz az értékeket az indextartományban jól szétszórja, akkor 2-3 hellyel később valószínűleg mindig lesz üres hely.

A keresést ugyanígy végezzük! Számítsuk ki a személyi szám alapján az adat helyét, s ha ott nem a keresett adat van, akkor vegyük sorra a mögötte levőket, amíg meg nem találjuk a keresettet!

A módszer lényege tehát egy kulcstranszformáció, amely a kulcsot egy nem túlságosan bő indextartományba transzformálja. Ez megadja a keresett érték helyét, illetve ehhez a helyhez közeli helyet, a mögötte levőket sorba kell nézni. Minden esetben meggondolandó, hogy az indextartomány mennyivel legyen bővebb a lehetséges értékek szá-

mánál! Ha sokkal nagyobb lenne, az helypazarlás, ha pedig csak kicsivel nagyobb, akkor nagyon gyakori lehet a kiszámított indexek átfedése, s emiatt az egyes elemek messzebb kerülhetnek ettől a helytől, ami a keresés idejét növeli. A kulcstranszformáláshoz sok esetben használnak indextáblázatot, indexfile-t is.

Érdekes gondolat: használjuk ezt az adatelhelyezési módszert adatok előrendezésére egy monoton kulcstranszformációs függvénnyel! Ezután ugyanis már olyan rendező módszert választhatunk, amely akkor optimális, ha az adatok majdnem rendezve vannak.

Egészen más feladatban hasonló közelítő értéket határoz meg két sokszámjegyű szám osztásakor a hányados becslésére.

1.7. Az iterált típus megfelelő finomítása

Amikor egy feladat során a használt adatstruktúra egy **sokaság**, akkor a megoldás hatékonyságát erőteljesen befolyásolhatja ennek további finomítása. Ebbe a típusosztályba négyféle alosztály tartozhat:

- **halmaz** (az elemek között nincs sorrendi kapcsolat),
- **sorozat** (minden elemet pontosan egy követ, illetve egy előz meg kivéve a két szélső elemet),
- **hierarchikus szerkezet** (minden elemet egy előz meg, de több is követhet, kivéve a legelsőt),
- **hálós szerkezet** (az egyes elemeket megelőzőkből és a következőkből is több lehet).

Feladattól függően megfelelő finomítást választva hatékony megoldásokat kaphatunk.

Feladat: Olvassuk be számjegyek egy halmazát, majd mondjuk meg, hogy egy adott számjegy szerepel-e ebben a halmazban!

Megoldás: A megoldás első változatában egy vektort fogunk használni. Ebben felsoroljuk a halmazhoz tartozó számjegyeket. Ekkor szükségünk van még egy változóra, amely a halmaz elemszámát tartalmazza. A beolvasás akkor ér véget, ha a bemenetről nem 0-9 közötti szám érkezik. Természetesen meg kell akadályozni, hogy egy számjegyet kétszer is felvegyünk a halmazba.

```
Halmazvizsgálat:
  DB:=0; Be: ELEM
  Ciklus amíg ELEM≥0 és ELEM≤9
    I:=1
    Ciklus amíg I≤DB és HALMAZ(I)≠ELEM
      I:=I+1
    Ciklus vége
    Ha I>DB akkor DB:=DB+1; HALMAZ(DB):=ELEM
    Be: ELEM
  Ciklus vége
  Be: ELEM                [a vizsgálandó elem]
  I:=1
  Ciklus amíg I≤DB és HALMAZ(I)≠ELEM
    I:=I+1
  Ciklus vége
  Ha I≤DB akkor Ki: ELEM," eleme a halmaznak."
Program vége.
```

Ebben a megoldásban tehát a halmazt sorozatként ábrázoltuk. Jobb megoldást kapunk, ha halmazként (a halmazt logikai vektorral megadva) ábrázoljuk.

A megoldás második változatában kihasználjuk azt, hogy a halmaznak kevés eleme van, s ezek is sorbarendezhetők. Ekkor HALMAZ(I) legyen IGAZ értékű, ha I eleme a halmaznak, s HAMIS egyébként.

```
Halmazvizsgálat:
  Ciklus I=0-tól 9-ig
    HALMAZ(I):=HAMIS
  Ciklus vége
  Be: ELEM
  Ciklus amíg ELEM≥0 és ELEM≤9
    HALMAZ(ELEM):=IGAZ
    Be: ELEM
  Ciklus vége
  Be: ELEM                [a vizsgálandó elem]
  Ha HALMAZ(ELEM) akkor Ki: ELEM," eleme a halmaznak."
Program vége.
```

Ez az ábrázolás akkor is jobb, ha halmazok metszetét, illetve unióját kell meghatározni. A programozási tételek között szereplő unió és metszet tételek szerint, ha a halmazt sorozatként ábrázoljuk, akkor ezek meghatározásához N, illetve M elemű halmazok esetén maximum N*M hasonlításra van szükség. A fenti ábrázolssal ez csupán annyi logikai műveletet jelent, ahány eleme lehet egy halmaznak. A halmaz típus jobb implementálása esetén (bitvektor) ez még gyorsabb lehet.

Nem minden esetben jobb azonban ez az ábrázolás. Nézzük meg például a halmazkiírás műveletét!

Az első ábrázolás esetén ez DB lépésben történhet meg, ami a halmaz konkrét elemszáma:

```
Halmazkiírás:
  Ciklus I=1-től DB-ig
    Ki: HALMAZ(I)
  Ciklus vége
Eljárás vége.
```

A második ábrázolásnál annyi lépés lesz, amennyi eleme egy halmaznak összesen lehet (most 10):

```
Halmazkiírás:
  Ciklus I=0-tól 9-ig
    Ha HALMAZ(I) akkor Ki:I
  Ciklus vége
Eljárás vége.
```

Ez merül fel akkor is, ha nem a feladat szempontjából szükséges adatok állnak rendelkezésre.

Feladat: Legyen adott egy vasútvonal, az egyes állomások közötti távolságokkal. Adjuk meg az I. és J. állomás távolságát!

Megoldás: Az A(N-1) vektor tartalmazza a távolságokat: A(I) az I. és az I+1. állomás távolsága.

```
Távolság(I,J):
  T:=0
  Ciklus K=I-től J-1-ig
    T:=T+A(I)
  Ciklus vége
Eljárás vége.
```

Ha előfeldolgozással elkészítjük a T(N-1) vektort, ahol T(I) az I. állomás távolsága az 1.-től, akkor a távolság meghatározása jóval egyszerűbb lesz:

```
Távolság(I,J):
  T:=T(J)-T(I)
Eljárás vége.
```

Más esetben akkor érünk el gyorsabb vagy lassúbb megoldást, ha a **sorozat**, illetve a **hierarchikus szerkezet** között választunk. Ha bináris fát használunk az elemek rendezésére azzal a tulajdonsággal, hogy a gyökérelemnél kisebb elemek a gyökértől balra, a nagyobbak pedig jobbra találhatóak, s a fát szekvenciálisan ábrázoljuk, akkor a kereséshez

ezt is felhasználhatjuk. A bináris fa felhasználása tehát a **sorozat részekre osztása** elvének alkalmazását jelenti.

Egy bináris fában akkor optimális a keresés (illetve a rendezés), ha bármely csomópontjától balra és jobbra kb. ugyanannyi elem van. Ha a fa nem ilyen szerkezetű, akkor kell foglalkozni a **kiegyensúlyozásával**.

A bináris fa szekvenciális ábrázolásánál az I. elem két szomszédja a 2*I. és a 2*I+1. helyen található, s azt tudjuk róluk, hogy A(I)≥A(2*I) és A(I)≤A(2*I+1). Egy erre alapozott keresés:

```
Keresés(N,A(),X,K,VAN):
  K:=1
  Ciklus amíg K≤N és A(K)≠X
    Ha A(K)>X akkor K:=2*K különben K:=2*K+1
  Ciklus vége
  VAN:=K≤N
Eljárás vége.
```

Ezt az adatszerkezetet nevezik kupacnak (vagy halomnak). Jól használható általában keresési feladatok megoldására, például egy Morze-dekódoló program a visszaalakítandó betűket ilyen struktúrában tárolhatja (rövid Morze-jelnél a fában balra, hosszúnál pedig jobbra lépve).

Nem mindig előnyös azonban a hierarchikus szerkezet használata sem. Sokszor alkalmazunk ilyeneket akkor is, ha a tényleges elemek a fa levelein helyezkednek el, s a többi csomópont a keresést megkönnyítő információt tartalmaz. Ekkor, ha az összes elemet ki kell írni, ki kell választani egy tetszőleges elemet, vagy egy elemet véletlenszerűen kell választani, akkor ez sokkal gyorsabban megtehető, ha az elemeket sorozatként ábrázoljuk.

Sorozat típus esetén további problémákat okozhat a **sorozat ábrázolása:**

- szekvenciális ábrázolás,
- láncolt ábrázolás.

Közismert, hogy a szekvenciális ábrázolás esetén rendkívül **könnyű elérni a sorozat tetszőleges elemét**, viszont nagyon **sok időbe kerülhet egy-egy elem kihagyása, illetve beillesztése. Láncolt ábrázolásnál fordított a helyzet**: ott az I. elem elérése lassú, az aktuális törlése vagy beillesztés ez elé pedig gyors.

Felmerül ez a probléma egy szövegszerkesztő esetén. Ha van benne sor törlés vagy sor beszúrás funkció, akkor célszerű a sorokat láncolva tárolni, ha viszont egy GOTO-szerű

parancsa is van (pozícionálás az I. sorra), akkor ez az ábrázolás máris sokkal kevésbé szerencsés.

Ugyancsak ábrázolási kérdés merül fel gráfok esetén: egy N szögpontú gráfot ábrázolhatunk a **csúcsmátrixával**, az **illeszkedési mátrixával**, illetve **élmátrixával** is.

1.8. Sorozat elemeinek rekurzív előállítása

Sokszor fordul elő, hogy a feladat egy sorozat előállítása. A sorozat egyes elemeit valamilyen Σ-t, Π-ot stb.-t tartalmazó képlettel adhatjuk meg. Ez alapján a megoldás egy ciklus, ami annyiszor fut le, ahány elemű a sorozat, s a belsejében egy másik ciklus kiszámítja a megfelelő elem értékét.

Ha azonban az **elemek egyetlen képlet alkalmazásával egymásból is előállíthatók**, akkor ennek felhasználásával egy sokkal hatékonyabb algoritmust kapunk.

Feladat: Adott egy számsorozat, határozzuk meg a sorozat K hosszúságú részsorozatainak átlagait (mozgóátlagok)!

Megoldás: Jelöljük a kiinduló sorozat elemeit A_i-vel, az eredményt pedig B_i-vel! Ekkor B_j a következőképpen számítható:

$$B_i = \sum_{j=0}^{k-1} A_{i+j}$$

Az ebből készített algoritmus:

```
Mozgóátlagok(N,K,A(),B()):
  Ciklus I=1-től N-K+1-ig
    S:=0
    Ciklus J=0-tól K-1-ig
      S:=S+A(I+J)
    Ciklus vége
    B(I):=S/K
  Ciklus vége
Eljárás vége.
```

Vegyük észre, hogy B_i és B_{i+1} számításában majdnem ugyanazok a tagok vesznek részt, így egymásból is kifejezhetők:

$$B_{i+1} = B_i - \frac{A_i}{k} + \frac{A_{i+k}}{k} = B_i + \frac{A_{i+k} - A_i}{k}$$

Így a lényegesen gyorsabb algoritmus:

```
Mozgóátlagok(N,K,A(),B()):
  S:=0
  Ciklus J=1-től K-ig
    S:=S+A(J)
  Ciklus vége
  B(1):=S/K
  Ciklus I=1-től N-K-ig
    B(I+1):=B(I)+(A(I+K)-A(I))/K
  Ciklus vége
Eljárás vége.
```

Ugyanezt a módszert alkalmazhatjuk egy másik feladatnál is, ahol a sorozat előző tagjából szorzással állítható elő az újabb.

Feladat: Számoljuk ki a Pascal háromszög N. sorát!

Megoldás: A következő számokat kell meghatároznunk:

$$\binom{n}{k} = \frac{n*(n-1)*\ldots*(k+1)}{(n-k)*(n-k-1)*\ldots*1}$$

Ez alapján a megoldó algoritmus:

```
Pascal_háromszög(N):
  Ciklus K=0-tól N-ig
    FNK:=1
    Ciklus I=1-től N-K-ig
      FNK:=FNK*(K+I)/I
    Ciklus vége
    P(K):=FNK
  Ciklus vége
Eljárás vége.
```

Az „n alatt a k" definíciójából könnyen ellenőrizhető átalakítással kapjuk:

$$\binom{n}{k} = \prod_{i=1}^{n-k} \frac{n-i+1}{i} = \binom{n}{k-1} * \frac{n-k+1}{k}$$

amely látszólagos „bonyolultabbsága" ellenére algoritmikus előnnyel rendelkezik (az eredeti definícióval szemben), ti. ha egy ilyen szám már ismert, akkor igen egyszerű műveletekkel származtatható belőle a következő! Az így elkészíthető megoldás:

```
Pascal_háromszög(N):
   P(0):=1
   Ciklus K=1-től N-ig
      P(K):=P(K-1)*(N-K+1)/K
   Ciklus vége
Eljárás vége.
```

Feladat: Adott egy N elemû sorozat, adjuk meg azon H hosszúságú részsorozatait, amelyek összege legalább K!

Ez a feladat tulajdonképpen az elsõ megismétlése, hiszen itt is mozgóösszegeket kell számolni, s ezek közül azokat megadni, amelyek egy adott határ fölé esnek. (A megoldásban az RSK() vektor tartalmazza a kiválogatott részsorozatok kezdõindexeit.)

```
Mozgóösszegek(N,K,H,A(),RSK()):
  DB:=0; S:=0
  Ciklus J=1-tól K-ig
    S:=S+A(J)
  Ciklus vége
  Ha S>H akkor DB:=1; RSK(DB):=1
  Ciklus I=1-től N-K-ig
    S:=S+(A(I+K)-A(I))
    Ha S>H akkor DB:=DB+1; RSK(DB):=I+1
  Ciklus vége
Eljárás vége.
```

Az eddigi feladatok közös jellemzõje volt, hogy **ciklusban kellett az összegzés tételét** alkalmazni.

Ugyanezt az elvet lehet alkalmazni akkor is, ha **ciklusban kell alkalmazni az eldöntés vagy a megszámlálás tételét**. Például: adjuk meg egy sorozat összes monoton növekvõ részsorozatát, adjuk meg egy sorozat összes legalább K db T tulajdonságú elemet tartalmazó H hosszúságú részsorozatát, ...

1.9. Dinamikus programozás

A fejezetet egy inspiráló példával kezdjük (bár nem ehhez a témához tartozik). A binomiális együtthatók kiszámolására ismert az alábbi képlet:

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1} \text{ és } \binom{n}{0}=\binom{n}{n}=1$$

Például n=4 és k=2 esetére a fenti képlet alkalmazásával az ábrán besötétített elemeket kell kiszámolni:

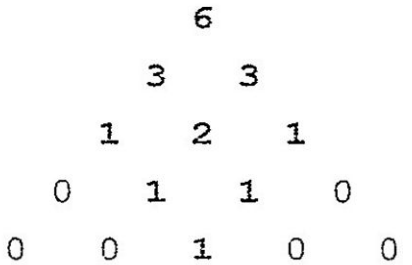
Ha egy rekurzív megoldást készítünk a képlet alapján, akkor azonban egyes elemeket sokszor ki kell számolni.

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

1. ábra

A 2. ábrán láthatjuk, hogy a rekurzív kiszámítás az első ábrán levő értékek közül melyiket hányszor számítja ki:

Jól látszik, hogy a számok az első ábrán látható számok tükörképei, azaz visszafelé haladva az egyes elemeket egyre többször számoljuk ki.



2. ábra

Ha azonban jó sorrendben számítatnánk ki az 1. ábrán látható elemeket, s a később még szükségeseket egy táblázatban tárolnánk, akkor minden szükséges elemet pontosan egyszer kellene kiszámolni.

Sok olyan optimalizálási feladat található, amelyre a visszalépéses keresés, illetve a szélességi vagy mélységi gráfkeresés megoldást ad, de az optimális megoldás előállításának lépésszáma az elemek számának exponenciális függvénye.

Bizonyos, elég gyakran előforduló esetekben azonban ennél lényegesen gyorsabb megoldást is kaphatunk. Ehhez két feltételnek kell teljesülnie.

1. A feladat megoldása N db egymásutáni részfeladat megoldásából áll, és az optimális megoldásra igaz lesz az, hogy minden részmegoldása is optimális a megfelelő részfeladatra..
2. Az egyes feladatokat tetszőleges ponton két részfeladatra bontva, a kapott részfeladatok között lesznek azonosak. (Ezek többszöri meghatározása felesleges - erre utalt a fejezet bevezető példája.)

A módszerre először egy klasszikus példát vizsgálunk:

Feladat: Adott n mátrix (M_1, M_2, ..., M_n), rendre $r_{i-1}*r_i$ méretűek, melyek ilyen sorrendben vett szorzatát kell kiszámítani. A mátrixszorzás asszociativitására építve keressünk egy olyan összeszorzási sorrendet, amelyben a műveletszám minimális!

Megoldás: Jelölje $m_{i,j}$ az $M_i*...*M_j$ mátrixszorzat kiszámításának költségét! Ekkor igazak a következő állítások:

(1) $m_{i,i+1}=r_{i-1}*r_i*r_{i+1}$ $\forall i\ (1\le i<n)$

(2) $m_{i,j}= \min (m_{i,k}+m_{k+1,r}+r_{i-1}*r_k*r_j)$ $\forall i\ (1\le i<n \text{ és } i+1<j)$

Az első szabály alapján az i. és az i+1. mátrix összeszorzásának optimális sorrendje egyértelmű, költsége pontosan kiszámítható. A második szabály szerint a minimális költségű $M_i*...*M_j$ szorzat kiszámítása az $M_i*...*M_k$ és a $M_{k+1}*...*M_j$ optimális rész-

szorzatok kiszámítása után ezek eredményének összeszorzásából áll, s meg kell keresnünk azt a k-t, amelyre ez a képlet a lehető legkisebb értéket adja. Az $m_{i,j}$ érték mellett a $K_{i,j}$ tartalmazza a minimális műveletszámhoz tartozó k értékét!

Be kellene látnunk, hogy a feladat ilyen megoldása megfelel a dinamikus programozás feltételeinek. Ehhez először vizsgáljuk meg, hogy ha találtunk egy optimális megoldást, akkor igaz-e, hogy ennek részmegoldásai optimálisak az ehhez tartozó részproblémák megoldására. Ez nagyon egyszerű, ugyanis, ha egy optimális megoldásban valamely K értéknél választottuk ketté a szorzatot, akkor az I-től K-ig terjedő részre ez biztosan optimális megoldást ad, hiszan ha lenne ennél jobb, az egyben az egész problémára is jobb megoldást adna. Az is nyilvánvaló, hogy ugyanannak a rész-szorzatnak a kiszámítási költségére többször is szükséf lehet, tehát el kell kerülnünk a többszöri kiszámítását.

```
Optimumkiválasztás:
  Ciklus I=1-től N-1-ig
    M(I,I+1):=R(I-1)*R(I)*R(I+1)
    M(I,I):=0
  Ciklus vége
  Ciklus L=1-től N-1-ig
    Ciklus I=1-től N-L-ig
      J:=I+L
      MIN:=I; MINÉRT:=M(I,I)+M(I+1,J)+R(I-1)*R(I)*R(J)
      Ciklus K=I+1-től J-1-ig
        ÉRT:=M(I,K)+M(K+1,J)+R(I-1)*R(K)*R(J)
        Ha ÉRT<MINÉRT akkor MIN:=K; ÉRT:=MINÉRT
      Ciklus vége
      K(I,J):=MIN; M(I,J):=MINÉRT
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Nincs más hátra, mint a tényleges kiszámítási sorrend kiírása. Erre használjuk a fenti megoldásban kiszámított K tömböt. Az I-től J-ig terjedő mátrixszorzat kiszámítása ugyanis úgy optimális, ha először kiszámoljuk az I-től a K(I,J)-ig, majd a K(I,J)+1-től a J-ig terjedő szorzatot, s utána a kapott két mátrixot összeszorozzuk.

```
Sorrend_kiírás(I,J):
  Ha J>I akkor Sorrend_kiírás(I,K(I,J))
               Sorrend_kiírás(K(I,J)+1,J)
               Ki: I,K(I,J),K(I,J)+1,J
Eljárás vége.
```

A feladat rekurzív megoldása is elkészíthető, ekkor azonban a többszöri kiszámolás elekrülése érdekében tárolni kell azokat az értékeket, amiket egyszer már kiszámoltunk, s

a továbbiakban még szükség lehet rájuk. Ezt a megoldásváltozatot hívják a dinamikus programozás rekurzív, **feljegyzéses módszerének.**

```
Előkészítés:
  Ciklus I=1-től N-ig
    Ciklus J=1-től N-ig
      M(I,I):=+∞
    Ciklus vége
  Ciklus vége
Eljárás vége.

Rek(I,J):
  Ha M(I,J)= +∞ akkor
     Ha i=j akkor M(I,J):=0
          különben Ciklus K=I-től J-1-ig
                     Q:=Rek(I,K)+Rek(K+1,J)+P(I-1)*P(K)*P(J)
                     Ha Q<M(I,J) akkor M(I,J):=Q
                   Ciklus vége
     Elágazás vége
  Elágazás vége
  Rek:=M(I,J)
Függvény vége.
```

Nézzünk egy másik példát erre a módszerre! Különböző méretű és tömegű építőkockáink vannak. Úgy lehet belőlük stabil tornyot építeni, hogy kisebb kockára nem lehet nagyobbat, illetve könnyebb kockára nem lehet nehezebbet tenni.

Feladat: N kocka alapján adjuk meg a belőlük építhető legmagasabb tornyot!

Meg kell gondolnunk, hogy itt hogyan alkalmazható a dinamikus programozás módszere. Első lépésként rendezzük sorba a kockákat méret szerint csökkenő sorrendbe. Ezután már csak azt kell megmondani, hogy az így kapott sorrendben mely kockákat lehet felhasználni, s melyeket nem, hogy a megoldás a tömeg szerinti sorrendnek is megfeleljen. A megoldás egyszerűsítése miatt tegyünk a sorozat végére egy 0 méretű és tömegű kockát!

A feladat megoldásához számítsuk ki, hogy mi az első I darab kockából építhető legmagasabb torony magassága akkor, ha legfelül az I. kocka található.

(1) m_1=magasság$_1$

(2) m_i= magasság$_i$+feltmax(m_1,m_2,...m_{i-1}) $\forall i\ (1 \le i \le n)$,

ahol a feltmax egyes paraméterei akkor használhatók, ha a megfelelő legfelső kocka tömege nem kisebb az I. kocka tömegénél. Jelölje ekkor a_i az I. kocka alatt levő kocka sorszámát!

Vizsgáljuk meg, teljesülnek-e a dinamikus programozás feltételei. Az N darab lehetséges kockából álló legmagasabb torony építésének megoldása egy kockasorozat. Ezt bárhol kettévágva nyilván igaznak kell lenni, hogy az addigi legfelsőig használható kockákat alkalmazva, s őt legfelülre téve az a legmagasabb torony, amit az optimális megoldásban megadtunk. Ha lenne magasabb, akkor nyilván az optimális megoldásban is az szerepelne. Az is látszik a fenti képletből, hogy ugyanarra a részmegoldásra többször is szükségünk lehet, hiszen m_i kiszámításához az összes korábbit használjuk.

```
Torony:
  M(1):=MAGASSÁG(1);
  Ciklus I=2-től N-ig
    MAX:=0; A(I):=0
    Ciklus J=1-től I-1-ig
      Ha TÖMEG(J)≥TÖMEG(I) akkor
        Ha MAX<M(J) akkor MAX:=M(J); A(I):=J
    Ciklus vége
    M(I):=MAGASSÁG(I)+MAX
  Ciklus vége
  MAX:=M(1); FELSŐ:=1
  Ciklus I=2-től N-ig
    Ha M(I)>MAX akkor MAX:=M(I); FELSŐ:=I
  Ciklus vége
Eljárás vége.
```

1.10. Mohó algoritmus

Egyes esetekben még a dinamikus programozásnál is hatékonyabb algoritmust használhatunk. A dinamikus programozás ugyanis egy probléma optimális megoldásához kiszámolta minden szükséges részprobléma optimális megoldását, s ezekből határozta meg a végleges megoldást. Ha tudnánk, hogy ezek közül melyek szükségesek, illetve melyek feleslegesek, akkor még gyorsabb megoldást kaphatnánk.

Erre szolgál a **mohó stratégia**, melynek alkalmazásához a következő feltételek teljesülésére van szükség:

1. A feladat megoldása N db egymásutáni részfeladat megoldásából áll, és az optimális megoldásra igaz lesz az, hogy minden részmegoldása is optimális a megfelelő részfeladatra..
2. A feladat optimális megoldása elérhető úgy is, hogy sorra vesszük a részfeladatokat, s meghatározzuk azok optimális megoldását, majd ezek egymásutánja adja a feladat optimális megoldását.

Összehasonlítva a dinamikus programozás feltételeivel, látható, hogy ez a módszer speciálisabb esetekben alkalmazható.

Válasszunk egy példát (*Cormen-Leiserson-Rivest: Algoritmusok* című könyvére alapozva)!

Feladat: Adott balról zárt, jobbról nyílt intervallumok egy halmaza. Ki kell választani közülük a lehető legtöbbet úgy, hogy páronként diszjunktak legyenek.

A feladat megoldásához rendezzük sorba az intervallumokat a végük szerint növekvő sorrendbe. Ekkor a feladat megoldására alkalmazható a mohó stratégia, ugyanis, ha az elsőnek véget érő intervallumot választjuk, akkor a többiekből biztosan legalább annyit tudunk még választani, mintha nem őt választanánk elsőnek. Ezután hagyjuk el azokat, akik az első vége előtt kezdődnek. Így a problémát visszavezettük egy egyszerűbb probléma megoldására. Jelölje K_i az intervallumok kezdetét, V_i pedig a végét, a vége szerint növekvően rendezve!

```
Legtöbb_intervallum:
  DB:=1; MEGOLDÁS(1):=1; UTOLSÓ:=1
  Ciklus I=2-től N-ig
    Ha K(I)≥V(UTOLSÓ) akkor DB:=DB+1; MEGOLDÁS(DB):=I
                             UTOLSÓ:=I
  Ciklus vége
Eljárás vége.
```

2. A ciklusmag végrehajtási idejének csökkentése

Ebben a fejezetben olyan példákkal foglalkozunk, amelyekben a ciklus lépésszámát nem tudjuk csökkenteni, viszont a ciklusmag végrehajtási idejét igen. Mivel a ciklusmagot sokszor hajtjuk végre, ez szintén jelentős futási idő csökkenést eredményezhet.

2.1. Elágazás transzformálása

Ez a feladatkör hasonlít az 1.6. fejezet feladataira. Ott egy ciklust **helyettesítettünk egyetlen indexeléssel.** Most ugyanezt tesszük egy **elágazással.** Ennek persze akkor van lényeges hatása, ha az elágazás egy ciklus belsejében van, s így a ciklus egyszeri lefutási idejét csökkenthetjük.

Feladat: Kerekítsünk egy valós számot!

Megoldás: Nézzük meg, hogy a számból kivonva az egész részét, .5-nél kisebb számot kapunk-e vagy sem!

```
Kerekít(X):
  Ha X-egészrész(X)<.5 akkor Kerekít:=egészrész(X)
                       különben Kerekít:=egészrész(X+1)
Eljárás vége.
```

A probléma itt az egészrész függvény többszöri kiszámítása, illetve maga a feltételvizsgálat. Ezt a feladatot meg lehet oldani egy egyszerű képlet alkalmazásával is.

```
Kerekít(X):
  Kerekít:=egészrész(X+.5)
Eljárás vége.
```

Ezen egyszerű eseten túllépve, az elágazás megszüntetését legtöbbször indexeléssel lehet elérni. (Ld. az 1.1.6. fejezetet!) Most a feltételek egymásutáni végigvizsgálása helyett egyetlen képlettel döntjük el, hogy az elágazás melyik ágát kell végrehajtani, sőt legtöbbször még ezeket az ágakat is összevonhatjuk.

Sok esetben ugyanazt kell tenni különböző adatokkal, s a tevékenység paraméterezhető az elágazás feltételeivel. Ilyenek lesznek a következő **megszámlálási, kiválogatási** feladatok.

Feladat: Szimuláljunk 100 kockadobást, s számoljuk meg, hogy ebből hány 1-est, 2-est stb. dobtunk!

Megoldás: A legegyszerűbb változatban egy hatirányú elágazást alkalmazunk. A DB(6) vektorban számoljuk az egyes dobások számát.

```
Kockadobás(DB()):
  DB():=0                [a vektor minden elemét nullázzuk]
  Ciklus I=1-től 100-ig
    X:=RND(6)            [1 és 6 közötti véletlen egész szám]
    Ha X=1 akkor DB(1):=DB(1)+1
    Ha X=2 akkor DB(2):=DB(2)+1
    Ha X=3 akkor DB(3):=DB(3)+1
    Ha X=4 akkor DB(4):=DB(4)+1
    Ha X=5 akkor DB(5):=DB(5)+1
    Ha X=6 akkor DB(6):=DB(6)+1
  Ciklus vége
Eljárás vége.
```

Ebben a megoldásban 600 hasonlítást és 100 értékadást kell elvégezni. Ha az értékadások után nem végeznénk el a további hasonlításokat (amelyek egyébként is feleslegesek), akkor kb. 300 hasonlítás marad.

```
Kockadobás(DB()):
  DB():=0                [a vektor minden elemét nullázzuk]
  Ciklus I=1-től 100-ig
    X:=RND(6)            [1 és 6 közötti véletlen egész szám]
    Ha X=1 akkor DB(1):=DB(1)+1 különben
    Ha X=2 akkor DB(2):=DB(2)+1 különben
    Ha X=3 akkor DB(3):=DB(3)+1 különben
    Ha X=4 akkor DB(4):=DB(4)+1 különben
    Ha X=5 akkor DB(5):=DB(5)+1 különben
    Ha X=6 akkor DB(6):=DB(6)+1
  Ciklus vége
Eljárás vége.
```

Természetesen nem ez jelenti a lényeges időcsökkentést. Használjuk magát a véletlenszámot indexelésre, s ezzel az indexeléssel helyettesítsük a teljes elágazást!

```
Kockadobás(DB()):
  DB():=0                [a vektor minden elemét nullázzuk]
  Ciklus I=1-től 100-ig
    X:=RND(6)            [1 és 6 közötti véletlen egész szám]
    DB(X):=DB(X)+1
  Ciklus vége
Eljárás vége.
```

Ugyanezt az elvet használhatjuk a következő feladatban is.

Feladat: Számoljuk meg, hogy egy szövegben az angol ABC betűi hányszor fordulnak elő!

Megoldás: Itt azonnal a transzformált megoldást írjuk fel, ugyanis szerintünk senkinek nem jutna eszébe egy 26-ágú elágazást írni a feladat megoldására.

```
Betűszámolás(SZOVEG,DB(26)):
  DB:=0              [a vektor minden elemét nullázzuk]
  Ciklus I=1-től hossz(SZOVEG)-ig
    DB(SZOVEG(I)):=DB(SZOVEG(I))+1
  Ciklus vége
Eljárás vége.
```

Ez a megoldás tehát abban az esetben végezhető el, amikor az elágazás ágai egymást kizáró feltételeket tartalmaznak. Ezek a feltételek valamilyen kifejezés bizonyos értékeire vonatkoznak. Az értékeknek meg lehet feleltetni indexértékeket.

Sokszor nem egy-egy hozzárendelés végezhető el az érték és az index között, pl. számoljuk meg, hogy egy szövegben hány nagybetű, kisbetű, számjegy, illetve egyéb jel van! Négy számlálóra van szükség, adjunk hozzá egy címfüggvényt!

Használjunk egy CIM() vektort, ahol a megfelelő helyeken 1, 2, 3 vagy 4 található!

Nézzünk egy hasonló feladatot!

Feladat: Ismerjük valahány ember személyi számát, adjuk meg, hogy hány még iskolába nem járó, általános iskolás korú, középiskolás korú, egyetemista korú van köztük!

Megoldás: A következő korcsoportokat vegyük figyelembe:

0 - 6
6 - 14
14 - 18
18 - 24

Ekkor a legegyszerűbb felépítésű program, amely a személyi számból csak a születési évet (EV) és az aktuális évet vizsgálja:

```
Számlálás(N,EV(),AKT,O,A,K,E):
  O:=0; A:=0; K:=0; E:=0
  Ciklus I=1-től N-ig
    X:=AKT-EV(I)
```

```
    Elágazás
      0≤X és X<6  esetén O:=O+1
      6≤X és X<14 esetén A:=A+1
     14≤X és X<18 esetén K:=K+1
     18≤X és X<24 esetén E:=E+1
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

Ha feltesszük, hogy mindenki ebben az évszázadban született, akkor az évszámot kettővel osztva 0 és 50 közötti számot kapunk. Ez az általunk vizsgált korcsoportokra (ha csak a hányadost nézzük):

0 - 6: 0, 1, 2
6 - 14: 3, 4, 5, 6
14 - 18: 7, 8
18 - 24: 9, 10, 11

Vegyünk fel egy 50 elemű tömböt (DB), s abban számoljunk:

```
Számlálás(N,EV(),AKT,O,A,K,E):
  Ciklus I=1-től N-ig
    X:=(AKT-EV(I))/2
    DB(X):=DB(X)+1
  Ciklus vége
  O:=DB(0)+DB(1)+DB(2); A:=DB(3)+DB(4)+DB(5)+DB(6)
  K:=DB(7)+DB(8); E:=DB(9)+DB(10)+DB(11)
Eljárás vége.
```

Itt tehát egy kisebb méretű tömb alkalmazására volt szükség a ciklusmag gyorsításához. Ugyanezt tapasztaljuk a következő feladatban is. Ha +1 helyett különböző növelés kellene, akkor a helyére +NOV(X)-et írjunk!

Feladat: Egy logikai hálózat működését szimuláljuk számítógépen, amelyben a következő áramköri elemek vannak: AND, OR, NAND, NOR, EXOR kapu, inverter. Készítsük el az egyes kapuk működését szimuláló eljárásokat!

Megoldás: Például az AND kapu akkor ad 1-et a kimenetén, ha mindkét bemenetén 1-et kap. A megoldó eljárás (amit persze az áramkör működését szimuláló ciklus belsejében használunk):

```
AND-kapu(BE1,BE2,KI):
  Ha BE1=1 és BE2=1 akkor KI:=1 különben KI:=0
Eljárás vége.
```

Ebben az eljárásban csak egyetlen elágazás van, de annak bonyolult a feltétele. Helyettesítsük az elágazást egy tömbindexeléssel (TAND(,) mátrix, mindkét indexe 0 vagy 1 lehet)!

```
AND-kapu(BE1,BE2,KI):
  KI:=TAND(BE1,BE2)
Eljárás vége.
```

Itt már csak az a kérdés, hogy nyertünk-e ezzel valamit, vagy nem. Mi kerül kevesebb időbe: egy összetett logikai feltétel kiértékelése vagy egy mátrix indexelése? Erre már csak gépfüggő választ adhatunk.

2.2. A kivételes eset kiküszöbölése

Sokszor **az teszi lassúvá a ciklusmag végrehajtását, hogy minden lépésben meg kell vizsgálni, hogy nem következett-e be egy egyetlen egyszer előforduló, kivételes eset**. Ekkor az algoritmust (és esetleg az adatszerkezetet) úgy kell átalakítani, hogy ez a kivétel szűnjön meg! Sok esetben a **kivételes eset a ciklus leállási feltételében fordul elő. Keresési, eldöntési** feladatokban akkor, ha vagy nincs a sorozatban a keresett elem vagy esetleg már be sem kell lépni a ciklusba!

Feladat: Döntsük el, hogy egy számsorozatban van-e 0 elem!

Megoldás: Eddigi ismereteink alapján az eldöntés tételt alkalmazhatjuk:

```
Eldöntés(N,A(),V):
  I:=1
  Ciklus amíg I≤N és A(I)≠0
    I:=I+1
  Ciklus vége
  V:=(I≤N)
Eljárás vége.
```

Ebben a megoldásban a **kivételes eset** az, amikor **nincs az A vektorban 0 értékű elem**, emiatt kellett a ciklusfeltételt ilyen bonyolultan felírni. A kivételes esetet megszüntethetjük úgy, hogy a **vektor utolsó eleme mögé** (N+1.-nek) **elhelyezünk egy fiktív**, 0 értékű **elemet**. Így biztosan találunk 0-t, s a megtalálás helye adja meg a választ eredeti kérdésünkre.

```
Eldöntés(N,A(),V):
   I:=1; A(N+1):=0
   Ciklus amíg A(I)≠0
      I:=I+1
   Ciklus vége
   V:=(I≤N)
Eljárás vége.
```

Egy ilyen javítással a futási idő 20-30 százalékkal is csökkenhet.

Ugyanezt az elvet alkalmazhatjuk szinte szó szerint az **unió**, a **metszet**, illetve a **szétválogatás helyben** programozási tételeknél is. Nézzük meg például a metszet tételt:

```
Metszet(A(),N,B(),M,C(),K):
  K:=0
  Ciklus I=1-től N-ig
    J:=1
    Ciklus amíg J≤M és A(I)≠B(J)
      J:=J+1
    Ciklus vége
    Ha J>M akkor K:=K+1; C(K):=B(J)
  Ciklus vége
Eljárás vége.
```

Helyezzük el a B() vektor végére minden egyes lépésben az A() vektor éppen keresett elemét:

```
Metszet(A(),N,B(),M,C(),K):
  K:=0
  Ciklus I=1-től N-ig
    J:=1; B(M+1):=A(I)
    Ciklus amíg A(I)≠B(J)
      J:=J+1
    Ciklus vége
    Ha J>M akkor K:=K+1; C(K):=B(J)
  Ciklus vége
Eljárás vége.
```

Második fajta példánk a kivételes eset kiküszöbölésére nem a ciklusfeltételt módosítja, hanem a ciklusmagból való utasításkihozással csökkenti a program végrehajtási idejét. Itt tehát **a kivételes eset a belső elágazásban fordul elő.**

Feladat: Adott egy N elemű számsorozat (A(N)), a B(N) számsorozatot a következőképpen definiáljuk:

Megoldás: A következő egyszerű programot fogjuk módosítani:

```
Számítás(N,A(),B()):
  Ciklus I=1-től N-ig
    Ha I=1 akkor B(I):=A(I)
       különben B(I):=(A(I)+A(I-1))/2
  Ciklus vége
Eljárás vége.
```

A **kivételes eset** itt nyilván az **I=1 értékre alkalmazandó más számítási módszer.** Ha ezt kiemeljük a ciklusból, akkor a ciklusmag egyetlen utasítását mindenféle feltétel vizsgálatától függetlenül végezhetjük el.

```
Számítás(N,A(),B()):
  B(1):=A(1)
  Ciklus I=2-től N-ig
    B(I):=(A(I)+A(I-1))/2
  Ciklus vége
Eljárás vége.
```

(Sőt, mint látható ezzel nemcsak idő-, hanem némi helynyereséghez is jutottunk.)

Feladat: Egy logikai hálózat működését szimuláljuk számítógépen, amelyben a következő áramköri elemek vannak: AND, OR, NAND, NOR, EXOR kapu, inverter. Készítsük el az egyes kapuk működését szimuláló programrészt!

Megoldás: A megoldásban használjuk fel azt, hogy az egyes kapuk működését egy táblázattal leírhatjuk, ahogyan azt az előző fejezetben tettük! Rendeljünk az egyes elemekhez egy-egy típusazonosítót (1 és 6 közötti egész számot)!

```
  Ciklus
    ...
    Elágazás
      TIPUS=1 esetén KI:=TAND(BE1,BE2)
      TIPUS=2 esetén KI:=TOR(BE1,BE2)
      TIPUS=3 esetén KI:=TNAND(BE1,BE2)
      TIPUS=4 esetén KI:=TNOR(BE1,BE2)
```

```
      TIPUS=5 esetén KI:=TEXOR(BE1,BE2)
      TIPUS=6 esetén KI:=TINV(BE)
    Elágazás vége
    ...
  Ciklus vége
```

Az első 5 eset vizsgálatát megszüntethetnénk, ha a típust bevezetnénk harmadik indexnek, az inverter azonban kivétel: neki csak egy bemenete van! **Szüntessük meg a kivételt**: vegyünk fel az inverternek is egy második -fiktív- bemenetet, amelynek értékétől nem függ az inverter működése! Ekkor már alkalmazható az **elágazás indexeléssé transzformálása**, s így egyetlen táblázat kezelését kell megoldanunk, amelynek 3 indexe van.

```
  Ciklus
    ...
    KI:=TABLAZAT(TIPUS,BE1,BE2)
    ...
  Ciklus vége
```

2.3. Ciklusok szétválasztása

Ez a fejezet tulajdonképpen egy előző fejezetbeli példa folytatását tartalmazza. Azt a problémát vizsgáljuk, amikor nem egyetlen elem számítása más, hanem többé. Itt tulajdonképpen olyan feladatok fordulnak elő, az adatsorozatra egy **szétválogatást** kellene alkalmazni, majd az egyes részeket külön-külön feldolgozni, a szétválogatást azonban meg akarjuk takarítani. Ekkor a feldolgozó ciklus belsejében szerepel a szétválogatási feltétel is. **Ha a szétválogatás az elemek értékétől függetlenül is elvégezhető, akkor érdemes a ciklust kettéosztani, s a feldolgozást mindkét részre önállóan elvégezni.**

Feladat: Adott egy N elemű számsorozat (A(N)), a B(N) számsorozatot a következőképpen definiáljuk:

Megoldás:

```
Számítás(N,A(),B(),K):
  Ciklus I=1-től N-ig
    Ha I≤K akkor B(I):=X-A(I) különben B(I):=X+A(I)
  Ciklus vége
Eljárás vége.
```

Ezt számítsuk inkább két egymás utáni ciklusban!

```
Számítás(N,A(),B(),K):
  Ciklus I=1-től K-ig
    B(I):=X-A(I)
  Ciklus vége
  Ciklus I=K+1-től N-ig
    B(I):=X+A(I)
  Ciklus vége
Eljárás vége.
```

Ez az átalakítás nem csak akkor végezhető el, amikor a sorozatot összefüggő részsorozatokra tudjuk bontani, hanem bármely esetben, ha az egyes **részsorozatok egyszerűen bejárhatók.** (Pl. páros-páratlan indexű elemek, 2-nél több részsorozat esete.)

Bármelyik esetről is van szó, a megoldás mindig némi kódismétléssel jár.

Nézzünk még egy példát ciklusok szétválasztására!

Feladat: Egy könyvtári nyilvántartást vezetünk, amelyben N könyvről tartjuk számon a szerzőjét, címét, kiadóját, megjelenési dátumát. Szeretnénk gyors hozzáférést biztosítani a szerző és a cím szerint sorbarendezett adatsor kiírásához, valamint a GONDOLAT kiadó által kiadott könyvek cím szerint sorbarendezett kiírásához.

Megoldás:

Jelölések:
- ADAT(N,4) - szerző, cím, kiadó, dátum az egyes könyvekről
- MUTATO(N,3) - a fenti 3 szempont szerinti lista
- MUTATO(0,K) - a K. lista első elemére mutat
- KULCS(3) - az egyes listák kulcsmezői az ADAT tömbben

Így a listák elkészítésére a következő eljárást kapjuk:

```
Listák elkészítése:
  Ciklus K=1-től 3-ig
    MUTATO(0,K):=0
    Ciklus I=1-től N-ig
      Ha K<3 vagy (K=3 és ADAT(I,3)="GONDOLAT")
                          akkor Beillesztés(I,K)
    Ciklus vége
  Ciklus vége
Eljárás vége.

Beillesztés(I,K):
   P:=0; J:=MUTATO(P,K)
   Ciklus amíg J>0 és ADAT(J,KULCS(K))<ADAT(I,KULCS(K))
      P:=J; J:=MUTATO(P,K)
  Ciklus vége
  MUTATO(I,K):=J; MUTATO(P,K):=I
Eljárás vége.
```

Itt a főeljárás belső ciklusában levő elágazás feltétele bonyolult, a külső ciklus első 2 lefutásakor a K<3 feltétel lesz biztosan igaz, utolsó lefutásakor pedig a K=3 mellett kell megvizsgálni, hogy a GONDOLAT kiadóról van-e szó. Válasszuk ketté a K<3 és a K=3 esetet!

```
Listák elkészítése:
  Ciklus K=1-től 2-ig
    MUTATO(0,K):=0
    Ciklus I=1-től N-ig
      Beillesztés(I,K)
    Ciklus vége
  Ciklus vége
  K:=3; MUTATO(0,K):=0
  Ciklus I=1-től N-ig
    Ha ADAT(I,3)="GONDOLAT" akkor Beillesztés(I,K)
  Ciklus vége
Eljárás vége.
```

Sok esetben a szétválasztás után célszerű egy újabb összevonást végezni, mint a következő feladatban is:

Feladat: Számítsuk ki a következő összeget (n≥5 és páratlan):

Megoldás: A legprimitívebb megoldásban a ciklus belsejében egy elágazás található:

```
Összegzés:
  S:=0
  Ciklus I=1-től N-ig
    Elágazás
      I=1 vagy I=N esetén S:=S+A(I)
      I páros      esetén S:=S+4*A(I)
      egyéb      esetben S:=S+2*A(I)
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

A ciklust szétválaszthatjuk az elágazás feltételei szerint három esetre:

```
Összegzés:
  S:=A(1)+A(N)
  Ciklus I=2-től N-1-ig 2-esével
    S:=S+4*A(I)
  Ciklus vége
  Ciklus I=3-tól N-2-ig 2-esével
    S:=S+2*A(I)
  Ciklus vége
Eljárás vége.
```

Sőt, mint a lokális hatékonyság vizsgálata alapján majd láthatjuk, a konstanssal szorzást kiemelhetjük a ciklusok mögé.

Ezután azonban célszerű a két ciklust -egy index eltolásával- újra összevonni (**sorozatok párhuzamos feldolgozása**):

```
Összegzés:
  S:=A(1)+4*A(2)+A(N)
  Ciklus I=3-tól N-2-ig 2-esével
    S:=S+2*A(I)+4*A(I+1)
  Ciklus vége
Eljárás vége.
```

2.4. Feltételek elhagyása

Itt a program által felhasznált adatokban rejlik a hatékonyabbra írás lehetősége, s nem látszik olyan szembetűnően a javítás módja, mint az eddigiekben. Lényegük, hogy **feltételek, vagy azok egyes részei elhagyhatók, ha a feltételtől függő utasítás az elhagyott esetekben nem változtat az állapottéren, illetve a változás egyszerűen helyreállítható.**

A következő algoritmussal, mint tipikussal, nagyon sok szimulációs program betétjeként találkozhatunk.

Feladat: Egy vektor (A(N)) elemeiként megjelenö "valamiknek" (atomoknak, molekuláknak, rókáknak stb.) az állapotát változtatja egy bizonyosra (0-ra) egy adott P valószínűséggel.

Megoldás: Feltételezzük, hogy a program futása során több ízben is rákerül a vezérlés, s kezdetben az A() vektor elemei mind 0-tól különbözőek.

```
Szimulációs lépés(N,P,A()):
  Ciklus I=1-től N-ig
    Ha A(I)≠0 és Véletlenszám<P akkor A(I):=0
  Ciklus vége
Eljárás vége.
```

Azt állítjuk, hogy az alábbi, működését tekintve egyenértékű megvalósítás hatékonyabb, ugyanis kihasználja azt a **specialitást, hogy 0 értékű változónak minden feltételtől függetlenül lehet 0 értéket adni**:

```
Szimulációs lépés(N,P,A()):
  Ciklus I=1-től N-ig
    Ha Véletlenszám<P akkor A(I):=0
  Ciklus vége
Eljárás vége.
```

Meg kell gondolnunk, hogy a hasonlítások számát felére csökkentve, cserében viszont az értékadások számát valamilyen mértékben növelve, a hatékonyság "mérlegének" nyelve a pozitív irányba billen! (Az értékadások növekedésének az az oka, hogy akkor is elvégezzük az A(I):=0 értékadást, amikor az A(I)=0 eleve teljesül.) A számoláshoz -szokás szerint- a hasonlítások és az értékadások számát vesszük alapul. Határozzuk meg, mennyi kell belőlük összesen, ha k-szor hajtjuk végre az első, illetve a második algoritmust!

Az első esetben **k*2*N a hasonlítások** és (átlagosan) **$P*N(1+(1-P)+(1-P)^2+...+(1-P)^k)$ az értékadások száma.**

Figyelembe véve, hogy ez utóbbi éppen egy mértani sorozat, a végösszeg az első algoritmushoz: $2*k*N+N-N*(1-P)^{k+1}$.

A második algoritmusban a **hasonlítások száma** csak **k*N**, míg az **értékadások** (átlagos) **száma P*N*k**. így a teljes összegre a **k*N+ P*k*N** képletet kapjuk.

E két formulát összehasonlítva azt kapjuk, hogy az első algoritmus műveletigénye a nagyobb, mégpedig annál inkább, minél kisebb a P. Vagyis kis P értékek esetén lesz különösen számottevő a hatékonyságnövekedés.

A következő részfeladat is szimulációs programok gyakori része.

Feladat: Adott egy mátrix (A(N,M)), amely 0 és 1 értékű elemeket tartalmaz, valamint a mátrix egy elemének indexe (I,J). Adjuk meg az adott hely 1-es szomszédjainak számát!

Megoldás: Az alapmegoldásban körbenézzük a szomszédokat, s vigyázva a szélső elemekre, számoljuk az 1-esek számát.

```
Szomszédszám(N,M,A(,),I,J):
  S:=0
  Ciklus K=I-1-től I+1-ig
    Ciklus L=J-1-től J+1-ig
      Ha K≥1 és K≤N és L≥1 és L≤M és (K≠I vagy L≠J)
                    akkor Ha A(K,L)=1 akkor S:=S+1
    Ciklus vége
  Ciklus vége
  Szomszédszám:=S
Eljárás vége.
```

Könnyen észrevehető az a **specialitás**, hogy **a mátrix csak 1-es és 0-ás elemeket tartalmaz**. 0-t pedig bármihez hozzáadhatunk, attól az nem változik meg. Így a 2. feltételvizsgálat elhagyható.

```
Szomszédszám(N,M,A(,),I,J):
  S:=0
  Ciklus K=I-1-től I+1-ig
    Ciklus L=J-1-től J+1-ig
      Ha K≥1 és K≤N és L≥1 és L≤M és (K≠I vagy L≠J)
                          akkor S:=S+A(K,L)
    Ciklus vége
  Ciklus vége
  Szomszédszám:=S
Eljárás vége.
```

Újabb **specialitás**, hogy a **fenti ciklusok olyanok, hogy nem csak a szomszédokat vizsgálnák meg, hanem magát az adott elemet is**. Ezt a megoldásban ki kellett zárnunk. Tudjuk azonban azt, hogy ha hozzáadnánk, akkor az összeg vagy nem változna, vagy pedig 1-gyel lenne nagyobb, mint szükséges. Vonjuk le ezt az értéket az összegből a ciklusson kívül, a ciklusmagban pedig ne ellenőrizzük!

```
Szomszédszám(N,M,A(,),I,J):
  S:=-A(I,J)
  Ciklus K=I-1-től I+1-ig
    Ciklus L=J-1-től J+1-ig
      Ha K≥1 és K≤N és L≥1 és L≤M akkor S:=S+A(K,L)
    Ciklus vége
  Ciklus vége
  Szomszédszám:=S
Eljárás vége.
```

Alkalmazhatjuk még a **kivételes eset kiküszöbölését**. Itt a kivételt a mátrix széleinek kezelése jelenti. Vegyük körbe a mátrixot egy-egy sorral, illetve oszloppal, amelyek 0-elemeket tartalmaznak!

```
0 0 0 0 0 0 0 0 0
0 X X X X X X X 0
0 X X X X X X X 0
0 X X X X X X X 0
0 X X X X X X X 0
0 0 0 0 0 0 0 0 0
```

Ezt a specialitást fogjuk kihasználni a megoldásban. A 0-elemek miatt a ciklusok belsejében mindenfajta vizsgálat nélkül összegezhetünk.

```
Szomszédszám(N,M,A(,),I,J):
  S:=-A(I,J)
  Ciklus K=I-1-től I+1-ig
    Ciklus L=J-1-től J+1-ig
      S:=S+A(K,L)
    Ciklus vége
  Ciklus vége
  Szomszédszám:=S
Eljárás vége.
```

2.5. Az adatok előfeldolgozása

A hatékonysági vizsgálatokat egy rendezési feladat kapcsán folytatjuk. Mondanivalónkkal egyben arra is szeretnénk például szolgálni, hogy a feladat megoldásának folyamatába hogyan illeszkedik az optimalizálási szándék, s hogyan alakul programunk egy ismert algoritmusból a feladat konkrét tulajdonságait figyelembe vevő hatékony programmá.

Ebben a fejezetben a gyorsítás lényege az lesz, hogy ha a szükséges részeredményeket **nem akkor számítjuk ki, amikor szükség van rájuk, hanem esetleg jóval korábban**, akkor ezzel milyen sebességnövekedést lehet elérni. Ez természetesen csak akkor vezet célhoz, ha egy ilyen részeredményre többször is szükség van (gyakran fordul elő, hogy egymásbaágyazott ciklusok belsejéből a részeredmény kiszámítását kivihetjük egy teljesen önálló ciklusba).

Ez azt jelenti, hogy itt **rendezési** feladatokkal, **egyesítéssel, metszettel, összefuttatással** fogunk foglalkozni.

Feladat: Adott egy mátrix, rendezzük át a sorait sorösszeg szerint növekvő sorrendbe.

Megoldás: Jelölje A(N,M) a mátrixot, N a sorai, M az oszlopai számát! Válasszunk ki egy ismert elemi rendezési módszert, nevezetesen a minimumkiválasztásos módszert. Ennek "absztrakt" algoritmusa a következő (jelölje H(I) az N elemű, rendezendő sorozat I. elemét):

```
Rendezés minimumkiválasztással:
  Ciklus I=1-től N-1-ig
    [keressük meg az I. legkisebbet]
    L:=I    [L az eddigi legkisebb elem indexe]
    Ciklus J=I+1-től N-ig
      Ha H(L)>H(J) akkor L:=J
    Ciklus vége
    Csere(L. elem,I. elem)    [az I. legkisebb helyretétele]
  Ciklus vége
Eljárás vége.
```

Írjuk át az algoritmust a konkrét feladatra! Itt a H() sorozat szerepét az A(,) mátrix sorai játszák. A rendezési relációt pedig a sorösszegek természetes sorrendje határozza meg.

```
Rendezés minimumkiválasztással(N,A(,)):
  Ciklus I=1-től N-1-ig
    L:=I
    Ciklus J=I+1-től N-ig
      S1:= az L. sor sorösszege
      S2:= a J. sor sorösszege
      Ha S1>S2 akkor L:=J
    Ciklus vége
    Csere(I. sor, L. sor)
  Ciklus vége
Eljárás vége.
```

Első észrevételünk az lehet, hogy **az L. sor sorösszegét felesleges kiszámolni mindig újra**, mivel a korábbi legkisebb sorösszeg mindig rendelkezésünkre áll:

```
Rendezés minimumkiválasztással(N,A(,)):
  Ciklus I=1-től N-1-ig
    L:=I: S1:= az L. sor sorösszege
    Ciklus J=I+1-től N-ig
      S2:= a J. sor sorösszege
      Ha S1>S2 akkor L:=J: S1:=S2
    Ciklus vége
    Csere(I. sor, L. sor)
  Ciklus vége
Eljárás vége.
```

A következő tény alapján még gyorsabb megoldást készíthetünk: **a mátrixnak összesen N sora van, így összesen N-szer kell kiszámolni a sorösszegeket!**

```
Rendezés minimumkiválasztással(N,A(,)):
  Ciklus I=1-től N-ig
    S(I):= az I. sor sorösszege
  Ciklus vége
```

```
  Ciklus I=1-től N-1-ig
    L:=I
    Ciklus J=I+1-től N-ig
      Ha S(L)>S(J) akkor L:=J
    Ciklus vége
    Csere(I. sor, L. sor); Csere(S(I),S(L))
  Ciklus vége
Eljárás vége.
```

Mint látható, az időnyereségért memóriaigény-növekedéssel fizettünk. Ez egyrészt a segédvektor helyigényéből, másrészt az eredeti algoritmust kiegészítő "adminisztrációs" betétekből (az S() feltöltése, S()-k cseréje) tevődik össze.

Ugyanilyen lehetőség még sok esetben előfordul olyan programozási tételekben, amelyeknél ciklus belsejében egy feltételt többször ki kell értékelni: **metszet, unió, összefuttatás**. Az **előfeldolgozás** itt akkor hasznos, ha nem egyszerűen az elemek alapján kell elvégezni a feldolgozást, hanem az elemeken értelmezett valamilyen függvény alapján. Nézzünk erre egy példát:

Feladat: Egy kereskedelmi vállalat a forgalmát hetenként, azon belül naponként tartja nyilván. A heti adatheteseket összegük szerint növekvő sorrendbe rendezték az első, illetve a második félévben. Adjuk meg hasonló rendezettségben az egész évi adatokat!

Megoldás: Egy adathetes a következőket tartalmazza: a hét sorszáma, a hét napjai forgalma naponként (HET1(N), HET2(N) vektorokban vannak az adatok, minden elemük egy-egy, a fentieket tartalmazó rekord)! Ekkor az összefuttatás tétel belsejében a következő elágazást találhatjuk:

```
  Elágazás
    heti összeg(HET1(I))<heti összeg(HET2(J)) esetén ...
    heti összeg(HET1(I))=heti összeg(HET2(J)) esetén ...
    heti összeg(HET1(I))>heti összeg(HET2(J)) esetén ...
Elágazás vége
```

Ha a heti összegeket előre kiszámítjuk, azzal pontosan N+M összegszámítást kell elvégezni, míg ebben a megoldásban esetleg sokkal többet.

Hasonló feladat előfordul a számítógépi grafikában is, geometrikus képek transzformációjakor. Egy pont transzformációja a pont koordinátáinak a transzformációs mátrix-szal való szorzását jelenti, egy transzformációsorozat esetén pedig mindegyikük mátrixával kell szorozni. Sok pont esetén ez felesleges szorzásokat eredményez. Mivel a mátrixszorzás asszociatív művelet, ezért előfeldolgozásként elkészíthetjük a transzformációs mátri-

xok szorzatát, s ezzel az eredménymátrix-szal szorozhatjuk most már az egyes pontok koordinátavektorát.

2.6. Az adatmozgatások számának minimalizálása

Általában sok időt vesz el az adatoknak a memóriában (háttértáron) való mozgatása. Ebben a fejezetben azzal foglalkozunk, hogy **mit lehet a mozgatás helyett tenni.** Ilyen feladat a **rendezés**, a **kiválogatás**, valamint a **szétválogatás**. A háttértárról memóriába mozgatás ezeken kívül még a **keresési, eldöntési, számlálási** feladatoknál is előfordul.

Vegyük az előző fejezet feladatát!

Feladat: Adott egy mátrix, rendezzük át a sorait sorösszeg szerint növekvő sorrendbe.

Megoldás: Most már csak az okoz gondot, hogy a mátrix sorainak cseréje sok időt vesz igénybe (különösen nagy M-ek esetén). Ha magukat a sorokat fizikailag nem cserélnénk fel, hanem valahogyan a csere tényét jegyezzük fel csupán, akkor ezt a problémát is megoldanánk. Ennek megvalósítása -ahogy sejthetjük- csak többlet tárfelhasználással lehetséges, például a következő gondolattal: mindegyikhez **hozzárendelünk egy mutatót**, amely megadja, hogy **az adott sor hányadik helyen szerepel**. Ekkor a sorok cseréje egy indexpár cseréjére redukálódik.

```
Rendezés minimumkiválasztással(N,A(,)H(I)):
  Ciklus I=1-től N-ig
    S(I):= az I. sor sorösszege; H(I):=I
  Ciklus vége
  Ciklus I=1-től N-1-ig
    L:=I
    Ciklus J=I+1-től N-ig
      Ha S(H(L))>S(H(J)) akkor L:=J
    Ciklus vége
    Csere(H(I),H(L))
  Ciklus vége
Eljárás vége.
```

A lényeg tehát, hogy adatmozgatás helyett csupán az adatokra mutató értékeket kell cserélgetni, ami lényegesen kevesebb időbe kerül.

Nézzünk meg egy másik módszert is ugyanennek a lehetőségnek a kihasználására!

Feladat: Adott egy mátrix, rendezzük át a sorait sorösszeg szerint növekvő sorrendbe.

Megoldás: Használjunk a rendezendő A(N,M) mátrix mellett egy MUT(N) mutatóvektort! MUT(0) mutasson a sorrendben 1. A()-beli sorra, MUT(I) pedig az I. utánira! Itt a **beillesztéses rendezést** fogjuk alkalmazni.

```
Rendezés beillesztéssel(N,A(,),MUT()):
  Ciklus I=1-től N-ig
    S(I):= az I. sor sorösszege
  Ciklus vége
  MUT(0):=1; MUT(1):=0
  Ciklus I=2-től N-ig
    MR:=0; MU:=MUT(MR)
    Ciklus amíg MU>0 és S(MU)<S(I)
      MR:=MU; MU:=MUT(MR)
    Ciklus vége
    MUT(I):=MU; MUT(MR):=I
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Ugyanezt a módszert használtuk a füzet 1. példájában (ciklikus léptetés), amikor az első változatból elkészítettük a másodikat, illetve a harmadikat. A másodiknál többletmemóriát vettünk igénybe azért, hogy minden adatot csak egyszer mozgassunk, a harmadiknál pedig a mozgatások sorrendjét változtattuk meg ugyanez miatt.

Ha **háttértáron kell keresni**, akkor sok időt visz el a megvizsgálandó rekordok behozása a memóriába (ezek a rekordok igen hosszúak is lehetnek). Sokat segítene, ha a rekordnak csak a hasonlításhoz szükséges mezőjét (mezőit) hoznánk be, s a teljes rekordot csak akkor, ha megtaláltuk a keresettet. Ehhez a kereséshez szükséges mezőket valahogyan el kell különíteni a többi mezőtől. Ezt úgy lehet megtenni, ha a file rekordja mellett **létrehozunk egy indextáblát**, amely tartalmazza a rekordok kereséséhez szükséges mezőket (kulcsmezők), valamint egy mutatóértéket a rekord többi mezőjére.

2.7. Felesleges műveletek kiküszöbölése

Sok esetben az **adatmozgatásokat** nem helyettesítjük mással, hanem **elhagyjuk**, mert az algoritmus átalakításával feleslegessé válnak.

A módszer tipikus példáit a rendezések körében találhatjuk. Az elemi rendezések javításait vizsgáljuk meg.

Legkisebb elemet helyére tevő rendezések:

A cserés rendezés lényege: az I. elemet hasonlítsuk össze az összes mögötte levővel, s amelyik kisebb nála, azzal cseréljük meg!

```
Cserés rendezés(N,A()):
  Ciklus I=1-től N-1-ig
    Ciklus J=I+1-től N-ig
      Ha A(I)>A(J) akkor Csere(A(I),A(J))
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

A külső ciklus egy lefutása alatt az eredmény egyetlen eleme (az I.) kerül biztosan a helyére, amelynek eléréséhez általában egynél sokkal több cserére van szükség. A következő, minimumkiválasztásos rendezés külső ciklusának hatása ugyanez, de a belső ciklusban nem cserélget.

```
Minimumkiválasztásos rendezés(N,A()):
  Ciklus I=1-től N-1-ig
    MIN:=I
    Ciklus J=I+1-től N-ig
      Ha A(MIN)>A(J) akkor MIN:=J
    Ciklus vége
    Csere(A(MIN),A(I))
  Ciklus vége
Eljárás vége.
```

Beillesztéses rendezés:

A beillesztéses rendezések lényege: külső ciklusuk egy lefutása alatt egy rendezett részsorozatba beillesztenek egy újabb elemet. Az alapmódszer:

```
Beillesztéses rendezés(N,A()):
  Ciklus I=2-től N-ig
    J:=I-1
    Ciklus amíg J>0 és A(J)>A(J+1)
      Csere(A(J),A(J+1)); J:=J-1
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Ebben a rendezésben egy elem helyre tevéséhez általában sok cserére van szükség. A hátrafelé csúszó elemek mindegyike egyszer mozdul el, a beillesztendőt azonban nagyon sokszor mozgatjuk. A javításban a **beillesztendő felesleges mozgatását szüntetjük meg.**

```
Beillesztéses rendezés(N,A()):
  Ciklus I=2-től N-ig
    J:=I-1: X:=A(J+1)
    Ciklus amíg J>0 és A(J)>X
      A(J+1):=A(J): J:=J-1
    Ciklus vége
    A(J):=X
  Ciklus vége
Eljárás vége.
```

Ide tartoznak az olyan feladatok is, amikor egy **kiválogatás vagy szétválogatás után kell az eredményre valamilyen programozási tételt alkalmazni** és a kiválogatás eredményére máshol nincs szükség. A megoldás ekkor: a kiválogatás vagy szétválogatás az elemeket ne gyűjtse ki (**felesleges adatmozgatás**), hanem azonnal dolgozzuk fel őket.

3. Alapelvében más megoldás keresése

Ebben a fejezetben az eddigiektől eltérően nem javítási módszereket közlünk, hanem olyan módszerekre, valamint ismeretekre hívjuk föl a figyelmet, amelyek egy feladat megoldására valami egészen más eljárást adnak. Ez a terület inkább egy algoritmuselméleti könyvbe tartozik, mint egy programozás módszertaniba, mi csupán a hatékonyság témakörének teljessége miatt foglalkozunk vele. (Részletességben, teljességben azonban meg sem közelítjük a megfelelő témájú algoritmuselméleti könyveket.)

3.1. Matematikai ismeretek kihasználása

Ez a fejezet más jellegű alkalmazás lesz: a matematikai ismeretek itt lehetőséget adnak az eredetitől eltérő, egészen más algoritmus alkalmazására. Itt a lényeg tehát a **specifikáció átalakítása.**

Feladat: Számoljuk ki a Pascal háromszög N. sorát!

Megoldás: Ez, mint ismert, az alábbi számok felsorolását jelenti:

$$\binom{n}{k} = \frac{n!}{k!*(n-k)!}, \text{ k=0,1,...,n}$$

```
Pascal_háromszög(N):
  Ciklus K=0-tól N-ig
    FN:=1; FK:=1; FNK:=1
    Ciklus I=1-től N-ig
      FN:=FN*I
    Ciklus vége
    Ciklus I=1-től K-ig
      FK:=FK*I
    Ciklus vége
    Ciklus I=1-től N-K-ig
      FNK:=FNK*I
    Ciklus vége
    P(K):=FN/(FK*FNK)
  Ciklus vége
Eljárás vége.
```

Első észrevételünk az lehet, hogy ezt a képletet **egyszerűsíthetjük**:

$$\binom{n}{k} = \frac{n*(n-1)*...*(k+1)}{(n-k)*(n-k-1)*...*1}$$

Ez alapján a következő algoritmus írható:

```
Pascal_háromszög(N):
  Ciklus K=0-tól N-ig
    FNK:=1
    Ciklus I=1-től N-K-ig
      FNK:=FNK*(K+I)/I
    Ciklus vége
    P(K):=FNK
  Ciklus vége
Eljárás vége.
```

Korábban láttuk, hogy az "n alatt a k" definíciójából könnyen ellenőrizhető átalakítással kapjuk:

$$\binom{n}{k} = \prod_{i=1}^{n-k} \frac{n-i+1}{i} = \binom{n}{k-1} * \frac{n-k+1}{k},$$

amely látszólagos "bonyolultabbsága" ellenére algoritmikus előnnyel rendelkezik (az credeti definícióval szemben), ti. ha egy ilyen szám már ismert, akkor igen egyszerű műveletekkel származtatható belőle a következő! Így az első elkészíthető megoldás:

```
Pascal_háromszög(N):
  P(0):=1
  Ciklus K=1-től N-ig
    P(K):=P(K-1)*(N-K+1)/K
  Ciklus vége
Eljárás vége.
```

Matematikai ismereteink segítségével további "időnyereséghez" lehet jutni. A ciklusmag végrehajtási számát csökkenthetjük, ha felhasználjuk e fogalom szimmetriáját:

$$\binom{n}{k} = \binom{n}{n-k}$$

A már kiszámolt tagok "szimmetrikus párjait" nem kalkuláljuk újra:

```
Pascal_háromszög(N):
  P(0):=1; P(N):=1
  Ciklus K=1-től N/2-ig
    P(K):=P(K-1)*(N-K+1)/K; P(N-K):=P(K)
  Ciklus vége
Eljárás vége.
```

A sikereken felbuzdulva újabb javítási lehetőségek után kutatunk. Hamar fölfedezhetjük az iménti módosításunk egy bosszantó mellékhatását: páros N-ekre a középső elemet kétszer állítjuk be. Hogyan küszöbölhetjük ki ezt az *időpazarlást*? Nyilván a páros esetben a középső elem külön -cikluson kívüli- beállításával:

```
Pascal_háromszög(N):
  P(0):=1; P(N):=1
  Ciklus K=1-től (N-1)/2-ig
    P(K):=P(K-1)*(N-K+1)/K; P(N-K):=P(K)
  Ciklus vége
  Ha N/2 egész akkor P(N/2):=P(N/2-1)*(N/2+1)*2/N
Eljárás vége.
```

Sajnos optimalizáló szándékunk most balul ütött ki, miért? (Számoljunk: az egy esetleges fölösleges értékadás érdekében mi mindenre kényszerültünk!)

Feladat: Számítsuk ki a Pascal háromszög adott elemét egy konkrét N, K értékre!

Megoldás: Használjuk az elõzõ feladat megoldásának egy részletét!

```
N_alatt_a_K(N,K,NK):
  P(0):=1; P(N):=1
  Ciklus I=1-től min(K,N-K)-ig
    P(I):=P(I-1)*(N-I+1)/I; P(N-I):=P(I)
  Ciklus vége
  NK:=P(K)
Eljárás vége.
```

Matematikai ismereteink újabb lehetőséget kínálnak:

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1} \text{ és } \binom{n}{0}=\binom{n}{n}=1$$

A megoldás ezek után nagyon egyszerű lehet, egy rekurzív függvényt kell készíteni:

```
N_a_K(N,K):
   Ha K=N vagy K=0 akkor N_a_K:=1
                   különben N_a_K:=N_a_K(N-1,K)+N_a_K(N-1,K-1)
Eljárás vége.
```

Ha a rekurziót el akarjuk kerülni, akkor nekünk kell megoldani az egyes elemek külön tárolását.

```
N_alatt_a_K(N,K,NK):
  Ciklus I=1-től N-K-ig
    B(I,0):=1
  Ciklus vége
  Ciklus I=1-től K-ig
    B(I,I):=1
  Ciklus vége
  Ciklus I=2-től N-ig
    Ciklus J=max(1,I-N+K)-tól min(I-1,K)-ig
      B(I,J):=B(I-1,J-1)+B(I-1,J)
    Ciklus vége
  Ciklus vége
  NK:=B(N,K)
Eljárás vége.
```

A megoldás kétségtelen hátránya, hogy nagyon sok helyet foglal egy (N+1)x(N+1)-es mátrix, ahol a kiszámolt elemeket tárolni kell. Ügyesebb megoldást kaphatunk, ha kicsit megvizsgáljuk a szükséges elemeket:

Legyen például N=5, K=3:

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

Ekkor tulajdonképpen elég lenne egy (K+1)x(N-K+1)-es mátrix, ha az elemeket másképpen indexelnénk. Sőt még ezen is lehet javítani, ugyanis egyszerre ennek a mátrixnak is csak egy sorára van szükség.

```
N_alatt_a_K(N,K,NK):
  Ciklus I=0-tól K-ig
    P(I):=1
  Ciklus vége
  Ciklus I=1-től N-K-ig
    Ciklus J=1-től K-ig
      P(J):=P(J)+P(J-1)
    Ciklus vége
  Ciklus vége
  NK:=P(K)
Eljárás vége.
```

Vajon melyik megoldás a jobb? Számoljunk!

Az 1. algoritmusban a szorzások száma lesz érdekes. Ez legrosszabb esetben (N/2)*2 szorzás lehet, ezen kívül (N/2)*4 összeadás, valamint (N/2)*2+2 értékadás és (N/2)*3 tömbindexelés.

A 2. algoritmusban az összeadások száma legfeljebb $2*(N/2)^2$, a tömbindexelések száma $3*(N/2)^2$, az értékadások száma pedig $(N/2)^2$. Ezen kívül kell még N+1 tömbindexelés és értékadás. (A ciklusszervezés idejétől mindkét esetben eltekintettünk.)

Tegyük fel, hogy egy szorzás ideje = 20 összeadás idejével, az összeadás, tömbindexelés, értékadás pedig ugyanannyi időbe kerül. Ekkor a két algoritmus mérőszáma:

1. algoritmus: $20*N+2*N+N+2+3*N/2 = \mathbf{24.5*N+2}$

2. algoritmus: $2*N^2/4+3*N^2/4+N^2/4+2*N+2 = \mathbf{1.5*N^2+2*N+2}$

A 2. algoritmus tehát akkor **jobb**, ha

$$1.5*N^2+2*N+2 < 24.5*N+2$$

azaz

$1.5*N^2 < 22.5*N$, tehát **N < 15** .

Nagyon sok ilyen feladatot találhatunk még. Közös lényegük, hogy a matematikai ismeretek felhasználásával az eredetitől gyökeresen eltérő algoritmust is kaphatunk.

Nézzünk egy megdöbbentő példát!

Feladat: Adott egy összefüggő gráf, döntsük el, hogy fa-e!

Megoldás: A gráfot csúcsmátrixával ábrázoljuk. A megoldáshoz matematikai ismereteket használunk.

1. változat: Definíció: **Egy összefüggő gráf akkor és csak akkor fa, ha bármely két csúcsa között pontosan 1 út van.**

A megoldás erre épít: kiválasztunk két csúcsot minden lehetséges módon, meghatározzuk a közöttük levő összes utat, majd eldöntjük, hogy ezek száma 1 vagy nem 1.

2. változat: Állítás: Egy összefüggő gráf akkor és csak akkor fa, ha egy tetszőleges csúcsából kiindulva, mindegyik csúcsba egyetlen út vezet.

Míg az előző megoldásban N*(N-1)/2 csúcspárról kellett valamit eldöntenünk, ebben csupán N-1 csúcspárról kell.

3. változat: Állítás: Egy összefüggő gráf akkor és csak akkor fa, ha egy csúcsából elindulva, párhuzamosan haladva minden lehetséges irányba, az utak nem találkoznak.

Ezt az állítást felhasználva, a már bejárt útrészleteket nem kell újra bejárni, ami újabb jelentős lépésszám-csökkenést eredményez.

4. változat: Állítás: Egy összefüggő gráf akkor és csak akkor fa, ha az élei száma = a csúcsai száma - 1.

A megoldásban tehát le kell számolni, hogy hány él van, s eldönteni hogy ez eggyel kisebb-e a csúcsok számánál. Ez utóbbi változat lényegesen egyszerűbb és gyorsabb programot eredményez.

III. A helyfoglalás csökkentése

Helyfoglaláson a memóriában, illetve a háttértáron elfoglalt helyet értjük.

A helyfoglalás csökkenését két úton érhetjük el: vagy a program változóinak helyigényét csökkentjük, vagy a programkód helyfoglalását.

1. Az adatok mennyiségének csökkentése

Ahogyan a **végrehajtási idő** csökkentésénél a **ciklusokkal** volt célszerű foglalkozni, ugyanúgy itt a **sorozatok** fognak középpontban állni. Ez legtöbbször vektor, illetve mátrix lesz, de előfordulhat sor, verem, file stb. is. A **ciklusoknál** kétféleképpen gondolkodtunk: vagy a **végrehajtások számát** próbáltuk csökkenteni, vagy **egy végrehajtás idejét**. Analóg módon járhatunk el itt is: vagy **a sorozat elemszámát csökkentjük** (sőt teljesen megszüntetjük), vagy pedig **egy elemének helyfoglalását**.

1.1. Az indexes változók kiküszöbölése

A tárolt adatok mennyiségének csökkentésére az egyik legegyszerűbb módszer az **indexes változók helyettesítése egyetlen változóval**. Erre természetesen akkor van mód, ha a bennük tárolt adatokra a programnak nincs tartósan szüksége.

Először olyan feladatokat vizsgálunk, ahol **egy sorozatot állítunk elő, de végül csak az utolsóként kiszámolt elemre van szükségünk**. Mindegyik feladat **memóriakorlátos** lesz, azaz megadható előre, hogy a sorozat egyes elemeinek meghatározásához a korábbiakból maximum hányra van szükség.

Nézzünk erre példákat! (A példák nagyon egyszerűek, úgy is mondhatnánk primitívek lesznek, célunk velük a módszer megértetése.)

Feladat: Számítsuk ki adott N számra N! értékét!

Megoldás: Tudjuk, hogy **i!=i*(i-1)!**. Helyezzük el F(i)-ben az i! értékét!

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

A ciklusmag egyetlen utasítása jól mutatja, hogy az addig kiszámolt értékek közül mindig **csak a legutolsóra van szükség**, így elég csupán azt tárolni:

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

Feladat: Határozzuk meg az N. Fibonacci számot!

Megoldás: Tudjuk, hogy a Fibonacciról elnevezett számsorozat i. tagját az **F(i)=F(i-1)+F(i-2)** formulával határozhatjuk meg, 1-nek állítva be a 0., valamint az 1. tagot. Az algoritmikus leírásban az i. Fibonacci számot F(i)-vel jelöljük, ami egyben egy F() vektor i. elemét is jelenti.

```
Fibonacci-szám(N,F()):
  F(0):=1; F(1):=1
  Ciklus I=2-től N-ig
    F(I):=F(I-1)+F(I-2)
  Ciklus vége
  Fibonacci:=F(N)
Függvény vége.
```

Most az új érték kiszámításához **az utolsó két kiszámított értékre van szükség**, így tehát célszerű csak azokat tárolni:

```
Fibonacci-szám(N,F):
  F0:=1: F1:=1
  Ciklus I=2-től N-ig
    F:=F0+F1; F0:=F1; F1:=F
  Ciklus vége
  Fibonacci:=F
Függvény vége.
```

Itt némi nehézséget okozott, hogy F0 mindig az utolsó előtti kiszámított értéket, F1 pedig az utoljára kiszámítottat tartalmazta, s ennek megőrzése érdekében a futási időben engedményeket kellett tennünk. (Gondoskodni kellett az F0←F1←F "léptetésről".) Ezen a problémán is lehet azonban segíteni. Vezessünk be F0 és F1 helyett egy vektort, amelynek csak a 0 és az 1 lehet az indexe, s jelölje K azt az elemét, amelyikben az utolsó előttinek kiszámított érték van:

```
Fibonacci-szám(N,F()):
  F(0):=1; F(1):=1; K:=0
  Ciklus I=2-től N-ig
    F(K):=F(K)+F(1-K); K:=1-K
  Ciklus vége
  Fibonacci:=F(1-K)
Függvény vége.
```

Lehetne az **1-K** helyett a **K+1 mod 2** kifejezés is, ebből látszik ugyanis, hogy ha nem az utolsó kettő, hanem az utolsó M szám összegeként kellene kiszámítani a következő értéket, akkor a módosított megoldás könnyen átalakítható lenne.

Nézzünk egy kicsit összetettebb feladatot, ahol nem ennyire nyilvánvaló az indexes változó elhagyásának lehetősége.

Feladat: Egy nyúlpopulációról az egyes korcsoportba eső egyedek számát tároljuk. Tudjuk, hogy egy E éves nyúlnak átlagosan S(E) utódja születik, illetve, hogy E éves korában H(E) valószínűséggel pusztul el. Kövessük nyomon a nyúlpopuláció korcsoportváltozását! K korcsoport van, az induló létszámok az X() vektorban találhatóak.

Megoldás: Csak 1 év változását írjuk le. Ez a változás persze egy olyan ciklus belsejében van, amely annyi évig fut, ameddig a korcsoporteloszlást vizsgálni kívánjuk. Vegyünk fel egy vektort az év végi eloszlás tárolására!

```
Éves változás(N,X()):
  Y(1):=0
  Ciklus I=1-től N-1-ig
    Y(I+1):=(1-H(I))*X(I); Y(1):=Y(1)+S(I)*X(I)
  Ciklus vége
  Y(1):=Y(1)+S(N)*X(N)
  X():=Y()
Eljárás vége.
```

Nézzük meg, miért is van szükségünk az Y() vektorra, miért nem lehet azonnal az X() vektorba tenni az eredményt? Ezt részben indokolja Y(1) egészen más kiszámítási szabálya (kivételes eset). A másik probléma: X(2) új értékéhez szükség van X(1) régi értékére, X(3)-hoz viszont X(2) szükséges, tehát annak kiszámítása előtt nem célszerű elrontani. Fordítsuk meg a kiszámítási sorrendet az 1. elemet pedig számítsuk külön!

```
Éves változás(N,X()):
  Y:=S(N)*X(N)
  Ciklus I=N-1-től 1-ig -1-esével
    X(I+1):=(1-H(I))*X(I); Y:=Y+S(I)*X(I)
  Ciklus vége
  X(1):=Y
Eljárás vége.
```

Ugyanezt a módszert használhatjuk minden olyan feladat megoldásánál, ahol a következő változtatásokat kell elvégezni:

$$X_{új}(I+1) := f(X_{régi}(I)),$$

illetve általában valahány korábbi értékből kell kiszámítani az újat.

Más esetekben **egy sorozatból egy másik sorozatot kell készíteni úgy, hogy a régire a feldolgozás végén már nincs szükség.** Ekkor olyan algoritmusokat használjunk,

amelyek a megfelelő feldolgozásokat **helyben elvégzik** (rendezés, kiválogatás, szétválogatás)! Nézzünk erre néhány egyszerű feladatot!

Feladat: Adott egy számsorozat, adjuk meg a nem 0 elemeit!

Megoldás: Ahelyett, hogy egy újabb vektort vennénk fel a kiválogatás miatt, hozzuk előre a szükséges elemeket a bemenő vektorban!

```
Tömörítés(N,A()):
  I:=1
  Ciklus amíg I<N
    Ha A(I)=0 akkor A(I):=A(N); N:=N-1 különben I:=I+1
  Ciklus vége
  Ha A(N)=0 akkor N:=N-1
Eljárás vége.
```

Nagyon hasonló algoritmus a kiválogatás helyben:

```
Kiválogatás(N,A(),M):
  M:=0
  Ciklus I=1-től N-ig
    Ha A(I) T tulajdonságú akkor M:=M+1; A(M):=A(I)
  Ciklus vége
Eljárás vége.
```

Programozási tételként mondtuk ki a hasonló szétválogatási tételt: a **szétválogatás helyben** algoritmusát. Érdekességként említjük meg, hogy az összefuttatást is lehet helyben végezni: az A(2*N) vektor első N, illetve azt követő N elemét fésüljük össze (a közös elemek az eredményben kétszer fognak szerepelni).

```
Összefuttatás(N,A()):
  Ciklus I=1-től N-ig
    Ha A(I)>A(N+1) akkor Csere(A(I),A(N+1)); J:=1
                         Ciklus amíg A(N+J)>A(N+J+1)
                           Csere(A(I+J),A(N+J+1)); J:=J+1
                         Ciklus vége
    Elágazás vége
    Ha A(2*N-I+1)<A(N) akkor Csere(A(2*N-I+1),A(N)); J:=1
                             Ciklus amíg A(N-J)>A(N-J+1)
                               Csere(A(2*N-I-J+1),A(N-J))
                               J:=J+1
                             Ciklus vége
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

1.2. Ciklusok összevonása

Ha az indexes változóra azért van szükség, hogy **információt adjunk át egyik ciklusból a másikba**, akkor alkalmazható ez a módszer. Az **indexes változó megszünteté-**

sét most úgy érhetjük el, hogy összevonjuk azokat a ciklusokat, amik közötti információcsere miatt volt szükség rá.

Feladat: Adjuk meg egy mátrix maximális sorösszegű sorát!

Megoldás: Kiszámítjuk a sorösszegeket (ez N db **összegzés**), majd meghatározzuk közülük a legnagyobbat (ez egy **maximumkiválasztás**). Az A(N,M) mátrixot használjuk, MA lesz a maximális sorösszegű sor sorszáma.

```
Maximális_sorösszegű_sor(N,M,A(,),MA):
  Ciklus I=1-től N-ig
    S:=0
    Ciklus J=1-től M-ig
      S:=S+A(I,J)
    Ciklus vége
    S(I):=S
  Ciklus vége
  MA:=1
  Ciklus I=2-től N-ig
    Ha S(MA)<S(I) akkor MA:=I
  Ciklus vége
Eljárás vége.
```

A maximum meghatározásához elvileg két értékre van szükség: a már megvizsgáltak közül a legnagyobbra, valamint a következő vizsgálandóra. Ha ezeket az értékeket csak akkor határoznánk meg, amikor éppen szükség van rájuk, akkor nem kellene használni az S() vektort. Tehát a megoldás itt a **ciklusok összevonása**.

Megjegyzés: Az előző rész 2.5. fejezetében éppen az adatok előfeldolgozása jelentette a futási idő csökkenést. Itt az előfeldolgozás megszüntetése okozza a helyigény csökkenését.

```
Maximális_sorösszegű_sor(N,M,A(,),MA):
  SM:=0; MA:=1
  Ciklus J=1-től M-ig
    SM:=SM+A(1,J)
  Ciklus vége
  Ciklus I=2-től N-ig
    S:=0
    Ciklus J=1-től M-ig
      S:=S+A(I,J)
    Ciklus vége
    Ha S>SM akkor SM:=S; MA:=I
  Ciklus vége
Eljárás vége.
```

Megjegyzés: A feladattal egy későbbi fejezetben újra foglalkozunk, ott még a programszöveg méretét is csökkenthetjük.

Adatfeldolgozási feladatoknál gyakori, hogy különböző részletösszegeket kell számolni, majd megjeleníteni. Ezek tárolása is jelentős helyet követel, ha nem végezzük el azonnal a szükséges feldolgozást. Nézzünk erre is egy példát!

Feladat: N termékről ismerjük, hogy mennyit gyártottak belőlük, és milyen egységáron. Egy terméket többször is gyárthatnak, s ekkor az egységár is lehet más. Készítsünk kimutatást az egyes gyártásokkor a termékek áráról, összegezzük ezt termékenként, illetve adjunk teljes összeget! (Az azonos termékekről készült rekordok egymás mellett vannak.)

Megoldás: A legegyszerűbben a feladat három részfeladatra bontható:

```
Kimutatás készítés:
  Árszámítás
  Termékenkénti összesítés
  Teljes érték összesítés
Eljárás vége.
```

Ezzel a programot felbontottuk három, körülbelül egyforma nehézségű részre, amelyek önálló megoldása nyilván könnyebb feladat.

```
Árszámítás(AN,ADAT(AN)):
  Nyit(F,FNEV); AN:=0
  Ciklus amíg nem vége(F)
    AN:=AN+1; Olvas(F,ADAT(AN)); Ki: ADAT(AN)
  Ciklus vége
  Zár(F)
Eljárás vége.

Termékenkénti összesítés(AN,ADAT(AN),TN,TAR(TN)):
  TKOD:=ADAT(1).KOD; TAR(1):=ADAT(1).AR; TN:=1
  Ciklus I=2-től AN-ig
    Ha TKOD≠ADAT(I).KOD akkor Ki: TKOD,TAR(TN)
                              TN:=TN+1; TAR(TN):=0
                              TKOD:=ADAT(I).KOD
    Elágazás vége
    TAR(TN):=TAR(TN)+ADAT(I).AR
  Ciklus vége
Eljárás vége.

Teljes érték összesítés(TN,TAR(TN)):
  OSSZ:=0
  Ciklus I=1-től TN-ig
    OSSZ:=OSSZ+TAR(I)
  Ciklus vége
  Ki: OSSZ
Eljárás vége.
```

A probléma, hogy mindegyik rész ad tovább valamilyen információt a következőnek, ami mindkét esetben egy hosszú adatsor. Ha a három részben szereplő ciklusokat összevonjuk, akkor ezekre a vektorokra nincs szükség. Olvassuk az F file-ból az adatokat!

```
Kimutatás készítés(F):
  Olvas(F,ADAT); Ki: ADAT
  OSSZ:=0; TKOD:=ADAT.KOD; TAR:=ADAT.AR
  Ciklus amíg nem vége(F)
    Olvas(F,ADAT)
    Ha TKOD≠ADAT.KOD akkor OSSZ:=OSSZ+TAR; Ki: TKOD,TAR
                           TAR:=0; TKOD:=ADAT.KOD
    Elágazás vége
    Ki: ADAT; TAR:=TAR+ADAT.AR
  Ciklus vége
  Ki: TKOD,TAR; Ki: OSSZ+TAR
Eljárás vége.
```

1.3. Hézagosan kitöltött struktúrák

Ez a módszer a **változók szerkezetének átalakításán** alapul. Mindegyik példában vektorokat vagy mátrixokat fogunk vizsgálni, s arra keressük a választ, hogy mely elemeik tárolására nincs szükségünk.

Mit lehet tenni olyan mátrixokkal, amelyek elemei közül csak nagyon kevés lesz a nem 0 értékű (ún. **ritka mátrixok**)?

Első példánkban egy olyan mátrixot vizsgálunk, amelyről tudjuk, hogy **minden sorában legfeljebb adott darabszámú nem 0 elem található**. Az alapváltozatban tartalmazza az A(N,N) mátrix egy szénhidrogén szénatomjainak kötéseit!

$$A(I,J)=\begin{cases}1 & ha\ van kötés\ az I.\ és\ a J.\ atom\ között\\ 0 & ha\ nincs\end{cases}$$

Mivel a helynyerés céljából a mátrixnak csak az "információt hordozó" elemeit tartjuk meg, ezért az elemekre történő hivatkozás változni fog. Bár számunkra lényegtelen az, hogy a mátrix mi módon vesz részt a feldolgozásban, a konkrétság kedvéért az algoritmusbeli változást a kötésben álló elemek kiírásával szemléltetjük.

```
Kötések(N,A(,)):
  Ciklus I=1-től N-1-ig
    Ciklus J=I+1-től N-ig
      Ha A(I,J)=1 akkor Ki: I,J
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Vegyük észre, hogy a mátrix szimmetrikus (a kötési kapcsolat kölcsönös volta miatt), így elegendő a kapcsolatban álló szénatomok "egyik oldali társán" végighaladni, vagyis a mátrix főátló feletti vagy alatti háromszögén (J=I+1-től N-ig).

Ha ehelyett minden sorról azt tárolnánk, hogy benne mely helyen van 1-es érték (ilyen a szénatom esetén legfeljebb 4 lehet), s a maradék helyen 0 lenne benne, akkor a helyfoglalást N*N-ről 4*N-re csökkenthetnénk. Ezzel a módosítással ráadásul a fenti

algoritmus mit sem bonyolódik. Tartalmazza a B(N,4) mátrix az egyes sorok 1-es elemeinek indexeit, azaz azt, hogy az I. atomnak mely más atomokkal van kötése.

```
Kötések(N,B(,)):
  Ciklus I=1-től N-1-ig
    Ciklus J=1-től 4-ig
      Ha B(I,J)>I akkor Ki: I,B(I,J)
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Újabb módosításunkban magukat a kötéseket (és nem az atomokra vonatkozó információkat) tároljuk. A C(K,2) mátrix soraiban egymással kötésben levő atomok sorszámai találhatóak. így a felhasznált hely **2*(N-1)** és **4*N** közötti érték lesz, a kötések számától függően (pontosan **2*kötésszám**). (A fenti intervallum a következőképpen adódik: ahhoz, hogy az atomok egy vegyületet alkothassanak, N-1 kötés kell, s minden szénatom legfeljebb 4 kötésben szerepelhet.) A fenti feladat megoldása még egyszerűbbé is válik:

```
Kötések(N,B(,)):
  Ciklus I=1-től K-ig
    Ki: C(I,1),C(I,2)
  Ciklus vége
Eljárás vége.
```

A legjobb tárolással persze legtöbbször a végrehajtási időben veszítünk, s nem nyerünk. Nézzük ehhez a következő feladatot!

Feladat: Döntsük el, hogy az I. és a J. atom között van-e kötés!

Megoldás: Az első változat a legegyszerűbb:

```
Van_kötés(I,J):
  Van_kötés:=(A(I,J)=1)
Eljárás vége.
```

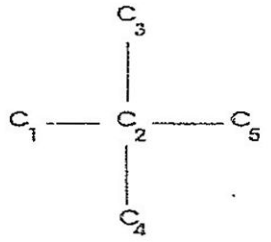
A második változat már csak egy ciklussal oldható meg:

```
Van_kötés(I,J):
  K:=1
  Ciklus amíg K≤4 és B(I,K)≠J
    K:=K+1
  Ciklus vége
  Van_kötés:=(K≤4)
Eljárás vége.
```

A harmadik változat is egy ciklus, de míg az előző esetben a ciklus lépésszáma maximum 4 volt, addig itt jóval több lehet:

```
Van_kötés(I,J):
  L:=1
  Ciklus amíg L≤K és (C(L,1)≠I vagy C(L,2)≠J)
                és (C(L,1)≠J vagy C(L,2)≠I)
    L:=L+1
  Ciklus vége
  Van_kötés:= (L≤K)
Eljárás vége.
```

Nézzük meg egy példavegyületen a háromféle tárolási módot!



$$A:\begin{pmatrix}0&1&0&0&0\\1&0&1&1&1\\0&1&0&0&0\\0&1&0&0&0\\0&1&0&0&0\end{pmatrix}\quad B:\begin{pmatrix}2&0&0&0\\1&3&4&5\\2&0&0&0\\2&0&0&0\\2&0&0&0\end{pmatrix}\quad C:\begin{pmatrix}1&2\\2&3\\2&4\\2&5\end{pmatrix}$$

25 hely **20 hely** **8 hely**

Az általánosabb helyzet: a mátrix "**ritka**", de az elemek teljesen szétszórtan helyezkednek el és tetszőleges értéket vehetnek föl. Ekkor nem követhetjük az előbb körvonalazott módszert. A következő információkat kell tárolnunk:

1. az elem mátrixbeli koordinátáit, és

2. magát az értéket.

Nyilvánvaló, hogy csak abban az esetben "éri meg" az áttérés, ha a "kitöltöttség" 3/N/N-nél kisebb. A példa kedvéért az A(N,N) mátrixnak **K db nem 0** eleme van (3*K<N*N). Az A(,) helyett a tárolást a D(K,3) mátrix-szal valósítjuk meg (D(,1)=az elem sor-, D(,2)=az elem oszlopindexe, D(,3)=az elem értéke), feltéve, hogy a mátrix elemei is egész típusúak.

Feladat: Írjuk meg azt az eljárást, amely képezi az így tárolt mátrix ún. transzponáltját (vagyis a főátlóra tükrözi a mátrixot)!

Megoldás: A megoldás nagyon egyszerű: a sor-, illetve az oszlopindexek cseréjéből áll.

```
Transzponálás(N,D(,),K):
  Ciklus I=1-től K-ig
    X:=D(I,1); D(I,1):=D(I,2); D(I,2):=X
  Ciklus vége
Eljárás vége.
```

Feladat: Az így tárolt mátrix elemei közül ki kell választani soronként a legkisebbet! A sor legkisebb értékét tartalmazza a MIN() vektor, a megfelelő indexet az IND().

Megoldás:

```
Sorminimumok(N,D(,),MIN(),IND()):
  MIN():=+∞; IND():=0 [1]
  Ciklus I=1-től K-ig
    Ha MIN(D(I,1))>D(I,3) akkor MIN(D(I,1)):=D(I,3)
                               IND(D(I,1)):=I
    Elágazás vége
  Ciklus vége
  Ciklus I=1-től N-ig
    Ha IND(I)=0 akkor MIN(I):=0 [csupa 0-t tartalmazó sor]
  Ciklus vége
Eljárás vége.
```

Az alábbi példában látható lesz, hogy a "ritkaság" tulajdonság figyelembe vétele mennyire hatékonyan működő algoritmust is létrehozhat. (Megjegyezzük: nem mindig ilyen szerencsés a helyzet, általában a helybeni optimalizálás az időbeli rovására történik.)

Feladat: Adott a D(,) mátrix a fenti értelmezéssel, valamint egy B(N) vektor. Számítsuk ki a mátrix és a vektor szorzatát!

```
Szorzás(K,D(,),N,B(),C()):
  C():=0
  Ciklus I=1-től K-ig
    C(D(I,1)):=C(D(I,1))+D(I,3)*B(D(I,2))
  Ciklus vége
Eljárás vége.
```

Ha mellétesszük az eredeti ábrázolás alapján működő algoritmust, akkor válik nyílvánvalóvá az ábrázolás különösen szerencsés volta. Most jelölje a mátrixot az A(N,N).

```
Szorzás(N,A(,),B(),C()):
  C():=0
  Ciklus I=1-től N-ig
    Ciklus J=1-től N-ig
      C(I):=C(I)+A(I,J)*B(J)
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Nemcsak mátrixok, hanem **vektorok** esetén is érdemes meggondolni a **hézagosan kitöltött struktúra** speciális ábrázolását.

Polinomok szokásos ábrázolása például a következő:

a $P(x)= a_0+a_1x+a_2x^2+...+a_nx^n$ polinomot az együtthatóival ábrázoljuk az A(N) vektor N+1 elemében.

[1] A vektorok egy "nem létező" kezdeti értékkel való beállítása.

Vegyük azonban a következő polinomot: $Q(x)= 1+x^{1000}+2*x^{5000}$! Ebben az esetben nyilvánvalóan helypazarlás mintegy 5000 db 0 értékű elem tárolása, sokkal jobb az értékes együtthatót és a hozzá tartozó kitevőt tárolni.

Sok hasonló probléma merül fel a számítógépi grafikában **raszterképek ábrázolásánál**. Az ilyen képeken vonalakat többféleképpen megadhatunk, és különböző képfajtáknál nem ugyanaz lesz a leghelytakarékosabb megoldás:

1. A kép minden egyes pontjáról megmondjuk, hogy világít vagy nem. Ehhez akkora mátrixra van szükség, amekkora a kép.
2. Sorpásztázás (minden sorban megadjuk, hogy abban az egyes pontok mettől meddig világosak, illetve sötétek, ...).
3. Vonalankénti megadás (egyenes szakaszok kezdő és végpontjai koordinátáit adjuk meg).

1.4. Speciális szerkezetű sorozatok

Eddig a **ritkaság** volt a legfőbb tulajdonság, amit figyelembe vehettünk, most **a speciális szerkezet** lesz, amelyre a gazdaságosabb tárolást építhetjük.

Feladat: Készítsük el az A kezdőértékű, X növekményű számtani sorozat első N tagját!

Megoldás: Egy sorozatról mindig az jut eszünkbe, hogy vektorban (file-ban) tároljuk az elemeit, mert így könnyen hivatkozhatunk rá. Ebben az esetben ez teljesen felesleges, hiszen minden elemet számolhatunk a kezdőértékből, a növekményből, valamint a sorszámból. Ezért, ha egyetlen elemre van szükségünk, akkor az `A + (S-1)*X` képlettel kaphatjuk meg az értékét, ha a sorozat összes tagját fel kell sorolnunk, akkor pedig a

```
B:=A
Ciklus I=1-től N-ig
  Ki: B
  B:=B+X
Ciklus vége
```

programrészletet használhatjuk.

Ugyanezt az elvet követhetjük, ha nem számtani, hanem mértani sorozat kezeléséről van szó.

A matematikai gyakorlat nagyon sokféle **különleges szerkezetű mátrix**ot ismer és használ. (Némelyiknek még nevet is adtak.)

A helytakarékosabb tárolás ötlete mindnél **a mátrix helyettesítése egy, csak az informatív elemeket tartalmazó vektorral.**

Nézzünk meg néhányat az ismertebbek közül!

1. **Diagonális mátrix**nak nevezik azt, amelynek 0-tól különböző eleme csak az ún. **főátlóban** van. Például:

$$\begin{pmatrix} a & 0 & 0 & \dots \\ 0 & b & 0 & 0 & \dots \\ 0 & 0 & c & 0 & \dots \\ & & & & \\ \dots & & 0 & 0 & 0 \\ \dots & & 0 & x & 0 \\ \dots & & 0 & 0 & y \end{pmatrix}$$

A megoldás nyilván egy D(N), a mátrix diagonálisában levő elemeket tartalmazó vektor lesz, a mátrix (I,J) elemére történő hivatkozás egyetlen vizsgálattal elintézhető. A címfüggvény (ha D(0) tartalmazza a 0 értéket) egy tömbindexet ad meg:

```
Cím(I,J):
  Ha I=J akkor Cím:=I különben Cím:=0
Eljárás vége.
```

A mátrix használata: **D(Cím(I,J))**

2. A **Toeplitz**, illetve **Hankel** nevet kapta az ún. **szalag-mátrix** pár, amelynek a fő-, illetve a mellékátlóval párhuzamosan levő elemei szalagszerűen megegyeznek. Például:

Toeplitz-mátrix

$$\begin{pmatrix} a & b & d & \dots \\ c & a & b & d & \dots \\ e & c & a & b & \dots \\ & & & & \\ \dots & & a & b & d \\ \dots & & c & a & b \\ \dots & & e & c & a \end{pmatrix}$$

Hankel-mátrix

$$\begin{pmatrix} \dots & & d & b & a \\ \dots & & d & b & a & c \\ \dots & & a & c & e \\ & & & & \\ d & b & a & \dots \\ b & a & c & \dots \\ a & c & e & \dots \end{pmatrix}$$

A **Toeplitz-mátrix** esetén T(I) legyen egyenlő a mátrix főátlóval párhuzamos I. szalagjában levő valamely elemmel; az 1 sorszámot rendeljük az (N,1)-hez, a 2-t az (N-1,1)-hez, ..., az N-t az (1,1)-hez,..., a 2*N-1-et az (1,N)-hez. (Vegyük észre a szabályszerűséget a főátlóval párhuzamos elemek indexeiben!)

Ekkor az (I,J) elemhez tartozó címfüggvény (ami most is egy tömbindex):

```
Cím(I,J):
  Cím:=N-(I-J)
Eljárás vége.
```

A mátrix használata: **T(Cím(I,J))**

A **Hankel-mátrix** esetén H(I) egyenlő a mátrix a mellékátlóval párhuzamos I. szalag valamely elemével; az (1,1) elemhez az 1 H()-beli indexet rendeljük, a (2,1)-t tartalmazó szalag elemeihez 2-t, ..., az (N,1)-hez N-t, ..., az (N,N)-hez a 2*N-1-t. Most is egy szabályszerűség vezet nyomra bennünket az (I,J) elem címének (indexének) megállapításánál:

```
Cím(I,J):
  Cím:=I+J-1
Eljárás vége.
```

A mátrix használata: **H(Cím(I,J))**

3. Egy biológiai probléma az alábbi ún. **Leslie-mátrix**ra vezet. Maga a probléma is igen érdekes és könnyen meg is érthető, érdemes megismerkedni vele. Vizsgáljunk, mondjuk, egy egérpopulációt! Csoportosítsuk őket kor szerint néhány korosztályba! Ha tudjuk, hogy az egyes korcsoportokba tartozó egerek milyen szaporák (azaz hány egérutóddal járulnak a népességhez), és milyen eséllyel léphetnek át a következő korcsoportba, akkor a korcsoportok létszámváltozását egy alábbi alakú mátrix-szal való szorzással számíthatjuk ki:

A mátrixban s_1, s_2, s_3, ..., s_n az egyes korcsoportokba tartozók utódjainak átlagos száma; $á_2$, $á_3$, ... $á_n$ az egyes korcsoportokba való átlépés valószínűsége.

$$\begin{pmatrix} s_1 & s_2 & s_3 & . & . & s_n \\ á_2 & 0 & 0 & . & . & 0 \\ 0 & á_3 & . & . & . & 0 \\ . & . & . & . & . & . \\ 0 & 0 & 0 & . & á_n & 0 \end{pmatrix}$$

Példa: Legyen három korcsoport 100-100-100 egérrel!

A mátrix: $\begin{pmatrix} 0 & 10 & 5 \\ .8 & 0 & 0 \\ 0 & .4 & 0 \end{pmatrix}$.

A következő időegységre (a korcsoportok "hossza" azonos) adódik a mátrix-szorzás után: $\begin{pmatrix} 0 & 10 & 5 \\ .8 & 0 & 0 \\ 0 & .4 & 0 \end{pmatrix} * \begin{pmatrix} 100 \\ 100 \\ 100 \end{pmatrix} = \begin{pmatrix} 1500 \\ 80 \\ 40 \end{pmatrix}$

Adjuk meg a **Leslie-mátrix** szerkezetét figyelembe vevő szorzó eljárást! A mátrix helyett 2 vektorban tároljuk a megfelelő nem 0 elemeket (A(N), S(N)), az egerek korcsoportbeli számát az E() vektor tartalmazza, a generációváltás utánit pedig az U().

```
Generációváltás(N,E(),U(),S(),A()):
  U(1):=0
  Ciklus I=1-től N-ig
    U(1):=U(1)+S(I)*E(I)        [az egérbébik száma]
  Ciklus vége
  Ciklus I=2-től N-ig
    U(I):=A(I)*E(I-1)           [I. korcsoportba belépők]
  Ciklus vége
Eljárás vége.
```

Az E() és U() vektorokra azért van szükség, mert U(I) kiszámításához szükség van E(I-1)-re. Az U() vektort **megszüntethetjük**, ha az E() vektor elemeit hátulról visszafelé számoljuk ki (ekkor az eredmény az E() vektorban keletkezik). Így a jobb megoldás:

```
Generációváltás(N,E(),S(),A()):
  E1:=S(1)*E(1)
  Ciklus I=N-től 2-ig -1-esével
    E1:=E1+S(I)*E(I); E(I):=A(I)*E(I-1)
  Ciklus vége
  E(1):=E1
Eljárás vége.
```

4. **Szimmetrikus mátrix.** A mátrix "felesleges" egyik felét kell elhagyni a tárolásnál (és például sorfolytonosan helyezzük el az elemeket egy vektorban).

```
Cím(I,J):
  Ha I≤J akkor Cím:=I*(I-1)/2+J-1
       különben Cím:=J*(J-1)/2+I-1
Eljárás vége.
```

A mátrix használata: **S(Cím(I,J))**

5. **Háromszögmátrix.** A 0-k elhagyása után a 4.-hez hasonló tárolással valósítható meg. Itt persze a kihagyott elemek értéke nem a vektor valamely elemével, hanem 0-val egyezik meg. Ehhez célszerű felvenni egy ezt tároló elemet.

```
Cím(I,J):
  Ha I≤J akkor Cím:=I*(I-1)/2+J-1 különben Cím:=0
Eljárás vége.
```

A mátrix használata: **H(Cím(I,J))**

6. **Vandermonde mátrix.** A lineáris algebra egyik gyakori mátrixa, a következő szerkezetű:

$$\begin{pmatrix} 1 & 1 & 1 & . & 1 \\ a_1 & a_2 & a_3 & . & a_n \\ a_1^2 & a_2^2 & a_3^2 & . & a_n^2 \\ . & . & . & . & . \\ a_1^{n-1} & a_2^{n-1} & a_3^{n-1} & . & a_n^{n-1} \end{pmatrix}$$

Itt nyilván a mátrix 2. sorának elemeit érdemes tárolni egy N elemű vektorban, s a többi elemet ezekből számítani.

1.5. Adatterületek megosztása

Sok olyan feladat van, ahol **két adatszerkezetet kell használni párhuzamosan** (ilyen például a szétválogatás két részre vagy két verem kezelése). Itt általában nem tudjuk előre, hogy melyikhez mennyi memóriára van szükség, de a kettő összegéről tudjuk. Tipikus megoldás ekkor **a két szerkezet egy helyen tárolása**, például helyezzük el a két vermet egyetlen vektorban, szembefordítva egymással! Megoldandó természetesen, hogy mindegyik csak a saját érvényes elemeit használja!

A veremkezelő eljáráscsomag a következőképpen nézhet ki (N elemű vektort használva):

```
Vereminicializálás:
  VMUT1:=0; VMUT2:=N+1
Eljárás vége.

Verembe(S,ELEM):
  Ha           S=1 akkor VMUT1:=VMUT1+1; V(VMUT1):=ELEM
  különben ha  S=2 akkor VMUT2:=VMUT2-1; V(VMUT2):=ELEM
Eljárás vége.

Veremből(S,ELEM):
  Ha           S=1 akkor ELEM:=V(VMUT1); VMUT1:=VMUT1-1
  különben ha  S=2 akkor ELEM:=V(VMUT2); VMUT2:=VMUT2+1
Eljárás vége.
```

```
Üres(S):
  Üres:=(S=1 és VMUT1=0) vagy (S=2 és VMUT2=N+1)
Eljárás vége.
```

Ha kettőnél több verem, sor, ... tárolására lenne szükség, akkor is megoldható a tárolásuk egy közös adatterületen, ekkor azonban az egyes struktúrák láncolására van szükség.

Hasonló feladat, amikor egy alsó- és egy felső-háromszögmátrixot tárolunk egy közös mátrixban (ha valamelyik diagonálisa üres). (Ekkor elképzelhető persze e két speciális szerkezetű mátrix tárolása vektorokban, a megfelelő címfüggvények alkalmazásával, de kényelmesebb lehet őket egyetlen mátrixban, közösen ábrázolni.)

Ha az A() alsó-, illetve az F() felső-háromszögmátrixot a közös M() mátrixban tároljuk (mindkettő diagonálisa is üres), akkor hozzájuk a következő eljárásokat használhatjuk:

```
A_érték(I,J):
  Ha I>J akkor A_érték:=M(I,J) különben A_érték:=0
Eljárás vége.

A_módosít(I,J,ÉRTÉK):
  Ha I>J akkor M(I,J):=ÉRTÉK
Eljárás vége.
```

illetve

```
F_érték(I,J):
  Ha I<J akkor F_érték:=M(I,J) különben F_érték:=0
Eljárás vége.

F_módosít(I,J,ÉRTÉK):
  Ha I<J akkor M(I,J):=ÉRTÉK
Eljárás vége.
```

Ha **különböző hosszúságú adatokat** kell tárolnunk egy adatszerkezetben (ilyen lehet például egy program sorainak tárolása), akkor egyik lehetőségünk, hogy a **hosszakat maximáljuk és minden adatot kiegészítünk a maximális hosszúságra.** Ekkor azonban sok felesleges helyet foglalhatunk le, különösen abban az esetben, amikor az adatok általában rövidek, s csak egy nagyon hosszú fordul elő közöttük. Ebben az esetben az adatterületet **láncolással** célszerű megosztani közöttük.

Szövegek ábrázolására szoktak egy másik megoldást alkalmazni: szöveg típusú változók, szöveg típusú vektorok egyes elemei csak **egy kezdőcímet és egy hosszinformációt** tartalmaznak (ez minden esetben 3 vagy 4 byte), s a szövegeket egy közös adatterületen helyezik el, aminek a kezelését külön kell megoldani.

Ennél az ábrázolásnál komoly problémát jelent **a különböző hosszúságú szabad helyek keletkezése**, amit bonyolult hulladékgyűjtő, illetve tárkiosztó eljárásokkal lehet ha-

tékonyan kezelhetővé tenni. Az üres helyek kezelését gyorsíthatja, ha a szövegeknek nem pontosan annyi karaktert foglalunk le, amennyire szükség van a tárolásukhoz, hanem a **szövegek tárolására szolgáló memóriát fix méretű blokkokra osztjuk**, s az egyes szövegek ábrázolásához ezeket a blokkokat láncoljuk.

1.6. Az adatelemek számítása

Most vizsgáljuk meg, hogyan csökkenthetjük egy sorozat egyes elemeinek helyfoglalását! A lényeg: válasszunk olyan adatreprezentációt, amelyben egy adatelem a lehető legkisebb helyet foglalja el! A problémát az okozza, hogy sokszor egy ilyen, tömören ábrázolt adatot is szeretnénk "normálisan" felhasználni (pl. ki szeretnénk írni a képernyőre).

A leghatásosabb módszer, amikor egy **adat tárolását megszüntethetjük**, mert mások értékeiből meghatározható.

Feladat: Egy személyi nyilvántartásban a következő adatokat tároljuk:

név, személyi szám, születési idő (év, hónap, nap).

Olvassuk be az adatokat egy file-ból és írjuk ki a képernyőre!

Megoldás: Az adatokat egy rekordszerkezetben adhatjuk meg, amely 3 mezőt tartalmaz (amelyek további részekből állhatnak):

Rekord: (név, személyi szám, születési idő: (év, hónap, nap))

Ha a név maximum 40 karakteres lehet, a személyi szám pontosan 11 számjegy, a születési év, hónap, nap pedig 3 egész szám, akkor (feltéve hogy egy számjegyet 1 byte-on, 1 egész számot 2 byte-on ábrázolunk, a rekord helyfoglalása: **57 byte**.

Ekkor az f file-t kiíró programrész:

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.SZEMELYI_SZAM
    Ki: ADAT.SZULETESI_IDO.EV,ADAT.SZULETESI_IDO.HONAP,
        ADAT.SZULETESI_IDO.NAP
   Ciklus vége
Eljárás vége.
```

Ebben az esetben a születési időre vonatkozó információt két helyen is tároljuk, s ez foglal felesleges byte-okat.

Ha **egy mező értéke egy másikéból közvetlenül számolható**, akkor azt számoljuk, és ne tartsunk fenn neki helyet a file-ban! Eszerint a rekordszerkezet:

Rekord: (név, személyi szám)

Ekkor egy rekord tárolásához már csak **51 byte** kell. Az adatokat kiíró eljárás:

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.SZEMELYI_SZAM
    EV:=1900+ADAT.SZEMELYI_SZAM(2)*10+ADAT.SZEMELYI_SZAM(3)
    HONAP:=ADAT.SZEMELYI_SZAM(4)*10+ADAT.SZEMELYI_SZAM(5)
    NAP:=ADAT.SZEMELYI_SZAM(6)*10+ADAT.SZEMELYI_SZAM(7)
    Ki: EV,HONAP,NAP
  Ciklus vége
Eljárás vége.
```

Természetesen most, és a további példákban is, a nyereségért fizetnünk kell: a program végrehajtási ideje nő.

Sok esetben egy mező értéke nem határozható meg egy másikéból közvetlenül. Előfordulhat azonban az, hogy többből igen. A következő példában **más mezőkből számolható mezők tárolását hagyjuk el.**

Feladat: Egy személyi nyilvántartásban a következő adatokat tároljuk:

név, alapbér, bérkiegészítés, nyugdíjjárulék, OTP-átutalás,
levonások összesen, kifizetett bér összesen.

Olvassuk be az adatokat egy file-ból és írjuk ki a képernyőre!

Megoldás: A név mellett most **6** számjellegű adatot kell tárolni minden rekordban. A kiíró eljárás:

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.ALAPBER,ADAT.KIEG
    Ki: ADAT.NYUGDIJ,ADAT.OTP,ADAT.LEVONASOK
    Ki: ADAT.OSSZESEN
  Ciklus vége
Eljárás vége.
```

Ebben az esetben 2 adat számolható a többiekből, pontosabban az egyik kiszámításához fel kell használni a másikat is. Ezzel a név mellett már csak 4 számjellegű adat marad.

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.ALAPBER,ADAT.KIEG
    LEVONASOK:=ADAT.NYUGDIJ+ADAT.OTP
    Ki: ADAT.NYUGDIJ,ADAT.OTP,LEVONASOK
```

```
    OSSZESEN:=ADAT.ALAPBER+ADAT.KIEG-LEVONASOK
    Ki: OSSZESEN
  Ciklus vége
Eljárás vége.
```

1.7. Az adatelemek kódolása

Ezek után rátérünk arra az esetre, amikor egy mező értékét okosabban lehet tárolni, de ezzel együtt meg kell oldani a **kódolás, dekódolás** problémáját.

Feladat: Egy személyi nyilvántartásban a következő adatokat tároljuk:

név, születési idő (év, hónapnév, nap).

Írjunk programot, amely ilyen adatokat tud beolvasni, elhelyezni egy file-ban, illetve a file tartalmát kiírni!

Megoldás: Az év és a nap egy egész szám, a hónapnév pedig szöveg.

```
Létrehozás(f):
  Ciklus amíg van adat
    Be: ADAT.NEV,
        ADAT.IDO.EV,
        ADAT.IDO.HONAPNEV,
        ADAT.IDO.NAP
    Ír(f,ADAT)
  Ciklus vége
Eljárás vége.
```

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.IDO.EV
    Ki: ADAT.IDO.HONAPNEV
    Ki: ADAT.IDO.NAP
  Ciklus vége
Eljárás vége.
```

A hónapok neve sok helyet foglal, a leghosszabb SZEPTEMBER 10 karakteres. Ezért a születési idő tárolásához **14 byte**-ra van szükség. Ha a hónapot a sorszámával tárolnánk, akkor ez csak **6 byte** lenne. Ekkor, ha a felhasználóval mégis hónapneveket akarunk közölni, akkor egy kódtáblázatban tárolni kell ezeket, s meg kell oldani a kódolást, dekódolást!

A módosított adatszerkezet:

név, születési idő (év, hónapsorszám, nap).

```
Létrehozás(f):
  Ciklus amíg van adat
    Be: ADAT.NEV,ADAT.IDO.EV,
        HNEV,ADAT.IDO.NAP
    I:=1
    Ciklus amíg HO(I)≠HNEV
      I:=I+1
    Ciklus vége
    ADAT.IDO.HONAP:=I
    Ír(f,ADAT)
  Ciklus vége
Eljárás vége.
```

```
Kiírás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,ADAT)
    Ki: ADAT.NEV
    Ki: ADAT.IDO.EV
    HNEV:=HO(ADAT.HONAP)
    Ki: HNEV
    Ki: ADAT.IDO.NAP
  Ciklus vége
Eljárás vége.
```

Itt az adatok kódolása vitt el több időt (meg kellett keresni a kódot a hónapnevek táblázatában), a dekódolás pedig kevesebbet (egy indexeléssel kiválasztottuk a hónapnevet).

Feladat: Egy személyi nyilvántartásban a következő adatokat tároljuk:

név, születési hely.

Írjunk programot, amely ilyen adatokat tud beolvasni, elhelyezni egy file-ban, illetve a file tartalmát kiírni!

Megoldás: Itt azt használhatjuk ki, hogy Magyarországon kb. 2000 helység van, azaz a helységek azonosítására egy egész típusú változó bőven elegendő. Ahogyan a hónapneveknek sorszámot feleltettünk meg, ugyanúgy megtehetjük ezt a helységek neveivel is. Ezután szó szerint ugyanazt a megoldást használhatjuk, mint az előző feladatban. Míg azonban az előző esetben a kódolást akár a felhasználóra is bízhattuk volna, ezt itt semmiképpen nem tehetjük meg.

Itt felmerül egy másik probléma is: hol tároljuk ezt a rengeteg helységnevet? Két lehetőség lenne: vagy maga a program tartalmazza a helységnevek táblázatát, vagy pedig egy másik file-ban tároljuk őket. Ezek közül az utóbbi lehetőség az ideális.

Harmadik probléma: mikor érdemes ezt a kódolást használni? Ha csak 10-20 ember személyi adatait kell nyilvántartani, akkor biztosan nem. Ha azonban a nyilvántartott személyek száma lényegesen nagyobb a helységek számánál (azaz a helységnevek többségét többször is kellene használni), akkor mindenképpen megéri a tömör tárolás.

A relációs adatbáziskezelés elmélete definiálja az **1. és a 2. normálformát**, ahol az utóbbit a következőképpen kaphatjuk meg:

Ha egy vagy több mező értékét más mezők egyértelműen meghatározzák, akkor felesleges ezeket annyi példányban tárolni, ahányszor ez a mezőkombináció előfordul, ehelyett ezeket vegyük ki egy külön táblázatba (relációba)!

A kódolás egy nevezetes példája karakterek különböző bithosszúságú kódolása, a Huffman-kód. Ennek lényege: a gyakran használt karaktereknek feleltessünk meg rövid kódot, a kevésbé gyakoriaknak pedig hosszabbat. A szövegben előforduló karakterek gyakorisága alapján meg lehet találni egy optimális kódolást, amivel a legrövidebben ábrázolhatjuk a szöveget.

Hasonló, az optimálishoz gyakran közel álló kódolás a Morze-ABC is.

Sok esetben a **sorozatnak nem egyetlen elemét kódolhatjuk egyértelműen, hanem egy részsorozatát.** Ez főleg akkor fordul elő, amikor az elemek maguk nagyon egyszerű szerkezetűek. Ilyen feladat például a szövegek tárolása. Itt a fő probléma, hogy gyakran sok szóköz szerepel benne egymás után. A szokásos megoldás ún. TAB-pozíciók kijelö-

lése. Ekkor egy TAB karakter kiírása azt jelenti, hogy a kiírást a következő TAB-pozíción kell folytatni.

Feladat: Egy F szövegfile-ban levő TAB karaktereket helyettesítsük megfelelő számú szóköz karakterrel!

Megoldás: A megoldásban figyelni kell, hogy a kiírásban melyik oszlopnál tartunk. Ha TAB karakter következik, akkor a következő tabulációs pozícióig szóközöket kell írni, különben pedig az aktuális karaktert. A megoldásban a bemenet és a kimenet megfeleltetése a következő:

Bemenet: jelek sorozata — Jel: betű ; TAB

Kimenet: csoportok sorozata — Csoport: betű ; szóközök TAB-pozícióig

```
Eljárás(f,g):
  Íráskezdés
  Ciklus amíg nem vége(f)
    Olvas(f,C); Csoportírás(g,C)
  Ciklus vége
Eljárás vége.
```

Az **Íráskezdés** feladata az első oszlop meghatározása.

```
Íráskezdés:
  OSZLOP:=1
Eljárás vége.
```

A kimenet szerkezetének megfelelően TAB karakter helyett szóközöket kell írni, a többi karaktert pedig változtatás nélkül kell átmásolni.

```
Csoportírás(g,C):
  Ha C=TAB akkor Szóközök_írása(g,OSZLOP)
         különben Betűírás(g,C,OSZLOP)
Eljárás vége.
```

A szóközök írása annyiból áll, hogy mindaddig írunk, amíg TAB-pozícióra nem érünk, s közben növeljük az aktuális oszlop sorszámát.

```
Szóközök_írása(g,OSZLOP):
  Ciklus
    Ír(g," "); OSZLOP:=OSZLOP+1
  amíg nem TAB-pozíció(OSZLOP)
  Ciklus vége
Eljárás vége.
```

A betűírás után vagy 1-gyel nő az aktuális oszlop sorszáma, vagy pedig az 1. oszlopra állunk (sorvég=NEWLINE karakter után).

```
Betűírás(g,C,OSZLOP):
  Ír(g,C)
  Ha C=NEWLINE akkor OSZLOP:=1 különben OSZLOP:=OSZLOP+1
Eljárás vége.
```

2. A programkód méretének csökkentése

A programkód méretét akkor tudjuk csökkenteni, ha bizonyos részek a programban többször szerepelnek. Ez tehát nem a program szövegének tömör írását jelenti, hanem olyan átalakítást, amellyel **ismétlődő kódrészeket csak egyszer, vagy egyszer sem írunk le.** Ezzel az átalakítással nyerhetjük általában a legkevesebb helyet, így csak akkor foglalkozzunk vele, ha már másképpen nem tudjuk csökkenteni a program méretét!

2.1. Az azonos funkciók közös eljárásba foglalása

Ebben az esetben az azonos programrészt több helyen is használtuk, s most, a **többszöri leírás helyett, egy eljárásba foglaljuk** és csak az eljáráshívást ismételjük többször. (**Az eljárások egyik fontos szerepe az absztrakció támogatása mellett a kódrövidítés.**)

Gyakori feladat több képernyős eredményt adó programoknál a lapozás. Ez a program különböző helyein válhat szükségessé. Egy jó lapozó eljárás a programkódot rövidebbé, áttekinthetőbbé teheti.

```
Lapozás:
  Kurzor állítás a képernyő legalsó sorába
  Ki: "Nyomjon le egy billentyűt!"
  Billentyűlenyomásra várakozás
  Képernyő törlés
Eljárás vége.
```

A lapozó eljárás esetleg tartalmazhatja valamilyen fix információ kiírását a képernyőre (pl. cím, idő stb.)

Ebben a példában az eljárásnak nem volt paramétere, a következőkben a paraméterezés is segíthet.

Vizsgáljuk tovább 2.1.2. fejezet feladatát!

Feladat: Adjuk meg egy mátrix maximális sorösszegű sorát!

Megoldás: A fenti fejezetben a következő programváltozatot kaptuk:

```
Maximális_sorösszegű_sor(N,M,A(,),MA):
  SM:=0; MA:=1
  Ciklus J=1-től M-ig
    SM:=SM+A(1,J)
  Ciklus vége
```

```
  Ciklus I=2-től N-ig
    S:=0
    Ciklus J=1-től M-ig
      S:=S+A(I,J)
    Ciklus vége
    Ha S>SM akkor SM:=S; MA:=I
  Ciklus vége
Eljárás vége.
```

Jól látható, hogy a programban szerepel olyan rész (a sorösszeg kiszámítása), amire két helyen van szükség. Ezt eddig kétszer írtuk le, most egyetlen eljárásra cseréljük.

```
Maximális_sorösszegű_sor(N,M,A(,),MA):
  MA:=1; Sorösszeg(N,M,A(,),1,SM)   [SM:=1. sor összege]
  Ciklus I=2-től N-ig
    Sorösszeg(N,M,A(,),I,S)          [S:=I. sor összege]
    Ha S>SM akkor SM:=S; MA:=I
  Ciklus vége
Eljárás vége.

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

Sok esetben nem tudunk olyan változókat adni, amelyek használatával különböző programszöveg-részletek azonossá tehetők, s így egyetlen eljárásba foglalhatók, adható viszont egy paraméter-eljárás vagy -függvény. Ilyen **függvénnyel paraméterezett eljárás** például a függvénytabellázás:

```
Függvénytáblázat(KEZDET,VÉG,L,F,TÁBLA()):
  I:=0
  Ciklus X=KEZDET-től VÉG-ig L-esével
    I:=I+1; TÁBLA(I).X:=X; TÁBLA(I).Y:=F(X)
  Ciklus vége
Eljárás vége.
```

Más esetekben még közös eljárás sem adható, ugyanis a különbség a feldolgozott adatok típusában van. Gyakori például a verem adattípus használata, de a verem elemei egyszer egész számok lehetnek, máskor valósak vagy szövegek ... A veremkezelő eljárások lényegében ugyanazok, csupán a verem elemtípusa, illetve a verembe kerülő és onnan kilépő elemek típusa más-más. Ekkor lehet hasznos az, ha **lehet eljárást típussal paraméterezni.** Ilyen paraméterezési lehetőségekkel foglalkozik a *Módszeres programozás* sorozat programozási bevezető ismeretekkel kapcsolatos kötete (µlógia 18).

2.2. Az adatok előfeldolgozása

Itt is egy többször ismétlődő rész megszüntetésével foglalkozunk, de még egy eljáráshívást sem teszünk a helyére. A lényeg: **mielőtt a tényleges feldolgozást elkezdenénk**, állítsuk elő az adatok valamilyen félig feldolgozott formáját, **végezzünk előfeldolgozást!**

Feladat: Készítsük el egy BASIC értelmező lexikális, szintaktikai elemző részét! (A bemenő szövegben bárhol tetszőleges számú szóköz lehet, s ezek a feldolgozást nem befolyásolják.)

Megjegyzés: Ilyen jellegű (bár jóval kisebb méretű) részfeladat gyakran fordul elő olyan programoknál, amelyek bemenetét kicsit szabadabb formában adhatjuk meg, mintegy egyszerű nyelvet használva.

Megoldás: A feladat lényegében szövegek beolvasásából, azok elemzéséből, majd valamilyen tevékenység elvégzéséből áll. Az 1. változatban a szóközök szűrését mindig a megfelelő programrész végzi. Egy sor hossza legyen N, a sort tároljuk a SOR() vektorban! Az f file feldolgozása:

```
Feldolgozás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,SOR)
    I:=1; N:=hossz(SOR)
    Ciklus amíg I≤N
      Felismer(SOR,I,N,X)          [X a tevékenység sorszáma,
                                    I-t változtatja]
      X. tevékenység(SOR,I,N)      [előrehaladhat SOR-ban]
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

Mind a felismerő eljárás, mind az egyes tevékenységek úgy kell legyenek megírva, hogy a közbeeső szóközöket ki tudják hagyni! Ez sokszori kódismétlést eredményez. Ezen segíthetünk azzal, hogy a szóközök szűrését előre elvégezzük, s a fenti kódrészleteket minden egyes helyről elhagyjuk. Ekkor természetesen a megoldás felső szintje alig változik:

```
Feldolgozás(f):
  Ciklus amíg nem vége(f)
    Olvas(f,SOR)
    I:=1; N:=hossz(SOR); Szóköz_szűrés(SOR,N)
    Ciklus amíg I≤N
      Felismer(SOR,I,N,X)          [X a tevékenység sorszáma,
                                    I-t változtatja]
      X. tevékenység(SOR,I,N)      [előrehaladhat SOR-ban]
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

```
Szóköz_szűrés(SOR,N):
  J:=0
  Ciklus I=1-től N-ig
    Ha SOR(I)≠" " akkor J:=J+1; SOR(J):=SOR(I)
  Ciklus vége
  N:=J
Eljárás vége.
```

Megjegyzés: Ezt a módszert alkalmaztuk már az előző rész 2.5. fejezetében is, ott azonban a ciklusmag végrehajtási idejének csökkentésére használtuk. Akkor valamit egy helyen, de sokszor számoltunk ki, s ezt emeltük ki előre. Most **valamit sok helyen végzünk el**, de a futás során mindegyiket csak egyszer, ezért a kiemelés a **programszöveg méretét** és egyben a **bonyolultságot** csökkenti.

Hasonló problémák fordulnak elő felhasználói adatok beolvasásakor.

Gyakori feladat például olyan válasz beolvasása, amely kérdésre a felhasználó IGEN-nel vagy NEM-mel válaszolhat. A legegyszerűbb esetekben a program az (I,i,N,n) válaszokat fogadja el helyeseknek, máskor esetleg még az (IGEN, igen, NEM, nem, ...) hosszabb szavakat is. Ha a program egynél több helyen használja a felhasználó válaszát, akkor mindenképpen érdemes már a beolvasás után megfeleltetni a válasznak egy logikai változót, s a későbbiekben már csak azt használni.

Egy másik példa a dátumok beolvasása. Itt a felhasználó esetleg hónap nevet ad a program kérdésére, ezt azonban az egyszerűbb feldolgozás miatt érdemes azonnal hónap sorszámmá alakítani. Általában a felhasználótól nem várhatjuk el kódok ismeretét, ezért szöveges válaszait minden esetben a programnak kell kódolnia egyszerűen feldolgozható kódokká.

2.3. Ciklusok összevonása

Ezt a módszert használtuk már az indexes változók megszüntetésére. Erre akkor volt lehetőség, ha két ciklus közötti információcserére használtuk az indexes változót. Ebben a fejezetben **ciklusokat akkor vonunk össze, ha ugyanazon adatsor egyes részeinek feldolgozását** -természetesen más-más algoritmussal- **külön ciklusokban végeztük.** Ez olyan feladatokban fordul elő, amelyekben **többféle szempont szerinti kiválogatott adatokat kell valamilyen módszerrel tovább feldolgozni. Ekkor célszerű lehet a kiválogatási ciklusokat összevonni egyszerű szétválogatássá.**

Feladat: Adjuk meg egy számsorozat pozitív, illetve negatív elemei négyzetösszegét! (A 0 pozitív számnak számít.)

Megoldás: Először válogassuk ki a pozitív, majd a negatív elemeket és mindegyikre számítsuk ki az összeget!

```
Összegzés(N,A(N),POZOSSZ,NEGOSSZ):
  POZOSSZ:=0
  Ciklus I=1-től N-ig
    Ha A(I)≥0 akkor POZOSSZ:=POZOSSZ+A(I)*A(I)
  Ciklus vége
  NEGOSSZ:=0
  Ciklus I=1-től N-ig
    Ha A(I)<0 akkor NEGOSSZ:=NEGOSSZ+A(I)*A(I)
  Ciklus vége
Eljárás vége.
```

Ebben az esetben a ciklusok belsejében levő elágazások pontosan olyanok, hogy feltételeik közül minden egyes A()-beli elemre az egyik teljesül (a két feltétel partícionálja azt a halmazt, amiből az A() sorozat elemei származnak). Ezért a **két ciklus összevonható** (sőt az elágazás ágairól a közös rész kiemelhető az elágazás elé).

```
Összegzés(N,A(N),POZOSSZ,NEGOSSZ):
  POZOSSZ:=0; NEGOSSZ:=0
  Ciklus I=1-től N-ig
    X:=A(I)*A(I)
    Ha A(I)≥0 akkor POZOSSZ:=POZOSSZ+X
           különben NEGOSSZ:=NEGOSSZ+X
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Az összevonás annál jobb eredményt ad, minél több volt a közös rész a különálló ciklusokban.

2.4. Programkód adattá transzformálása

Ebben a fejezetben az **egyetlenegyszer előforduló kódrészek megszüntetésének** (háttértárra helyezésének) lehetőségeivel foglalkozunk.

Programok gyakori része futásuk elején egy tájékoztató szöveg kiírása. Ez a programszövegben általában sok kiíró utasítást jelent. Ugyanígy előfordul, hogy a futás során a program képes valamilyen segítő információt kiírni a képernyőre - ez szintén kiíró utasítások sorozata. **A felesleges helyfoglalást az okozza, hogy ezzel a felhasználónak szánt szöveg is bekerül a program kódjába.** Egy 25x80-as képernyőt feltételezve így egy lapnyi szöveg is kb. 2000 byte-ot foglalhat a memóriában.

Sokkal jobb megoldás, ha ezeket a **szövegeket háttértáron tároljuk**, s amikor szükséges, akkor soronként (karakterenként) beolvassuk, majd kiírjuk a képernyőre. Ehhez elég a következő egyszerű eljárás:

```
Információközlés:
  Nyit(f,Információs file); Képernyőtörlés
  Ciklus amíg nem vége(f)
    Sorolvasás(f,SOR); Ki: SOR
  Ciklus vége
  Zár(f)
Eljárás vége.
```

A file-ból beolvasás ugyan lassúbb, mint a közvetlen kiírás, de az információadás akkor történik, amikor a felhasználó a gép előtt ül, s szeretné elolvasni a számára szükséges tudnivalókat. A felhasználó ekkor azonban nagyságrendekkel több időt tölt olvasással, mint a file-kezelés ideje, ezért ez a lassúbb megjelenítés nem okozhat problémát.

Van olyan eset, amikor a megjelenítendő szöveg **nem csak konstans információt tartalmaz**, ilyen a típuslevelek esete. Ekkor a szöveg nagy része állandó, de egyes részeit mindig konkrétan ki kell tölteni. Ekkor is érdemes az állandó részt háttértáron tárolni, jelölve benne a kitöltendő részeket.

A következő eljárás egy ilyen típuslevelet képes nyomtatni. A kitöltendő részeket lemezen %-jelek jelölik, két %-jel között van a hiányzó adatra vonatkozó kérdés. A levél nézzen ki a következőképpen:

> Kedves %címzett%!
>
> Osztálytalálkozót tartunk %dátum%-kor a budapesti %étterem% étteremben. Találkozás %időpont%-kor. A részletekkel kapcsolatban fordulj hozzám!
>
> Budapest, 1999.11.11.
>
> aláírás

A program ezt a levelet beolvassa, a kitöltetlen részekre rákérdez, s az egészet nyomtatóra nyomtatja:

```
Típuslevél:
  Nyit(f,Levél file); Képernyőtörlés
  Ciklus amíg nem vége(f)
    Olvas(f,CH)
    Ha CH≠'%' akkor Ki: CH; Nyomtat(CH)
            különben Százalékjelig_olvas(f,KÉRDÉS)
                     Ki: KÉRDÉS
                     Be: VÁLASZ
                     Nyomtat(VÁLASZ)
    Elágazás vége
  Ciklus vége
  Zár(f)
Eljárás vége.
```

Ugyanez a helyzet akkor, ha a programban **konstans ábrák** szerepelnek. Ez az ábra általában kezdőképernyő, illetve egy grafikus kép kezdeti állapota (pl. függvényábrázolásnál a koordinátatengelyek, skálabeosztás, tengelyfeliratok, ...)

Sokszor kell kódismétlést alkalmazni hasonló tevékenységek esetén. Ha ilyenkor csupán az adatok mások, amelyekre el kell végezni a közös tevékenységet, akkor célszerű a **kódismétlés helyett egy tömböt** alkalmazni, a tömbben tárolva a változó adatokat, majd a tevékenységet ciklusban a megfelelő számszor elvégezni.

Bonyolult grafikus ábrák megrajzolása általában sok utasítással történik. Ekkor programszöveg csökkentésre érdemes használni egy **grafikai leíró nyelvet**, az ábrát ezen a nyelven megadni, s készíteni hozzá egy egyszerű grafikus interpretert, amely a rajzolást elvégzi.

IV. A bonyolultság csökkentése

Ebben a fejezetben **logikai bonyolultsággal** foglalkozunk. A logikai bonyolultságot is a program szerkezete alapján lehet definiálni, pontos mérőszámot azonban nem tudunk adni. Eszerint egy program akkor egyszerűbb egy másiknál, ha benne **kevesebb elágazás és ciklus van, egyszerűbbek az elágazások, ciklusok feltételei, kisebb a struktúrák egymásba ágyazásának mélysége, valamint absztrakció segítségével a megértést megkönnyítő fogalmakat alkottunk.**

A bonyolultságvizsgálat matematikai alapjaival nem foglalkozunk, érdeklődő Olvasóinknak ajánljuk *Varga László: A programozási módszertan elmélete* című jegyzetét.

Nem foglalkozunk azzal az esettel, amikor egy bonyolult algoritmus helyett egy egyszerűbbet választunk, hanem inkább azt vesszük szemügyre, hogy egyes algoritmusok milyen elvek alapján tehetők egyszerűbbé.

Megjegyzés: A ciklusmag végrehajtási idejének csökkentése, a programszöveg méretének csökkentése legtöbbször a bonyolultságot is csökkenti.

A bonyolultság a legkevésbé precízen definiálható jellemző, de azért itt is találhatunk néhány módszert. Ezek általában a szerkezeti bonyolultság mérésére szolgálnak, ebből próbálunk közelíteni a logikai bonyolultsághoz. Mi lehet a bonyolultság mérésének célja? Talán a következők (de mindegyikhez hozzátehetnénk, hogy valamelyik más jellemző javulása érdekében megéri-e a bonyolultság növekedése):

- meg kell határozni egy elkészült programtermék komplexitását,
- két algoritmus közül melyik a bonyolultabb, nehezebben megérthető, megvalósítható,

Megjegyzés: Ebben a fejezetben (sőt a teljes könyvben) csak struktúrált programokkal foglalkozunk, így értelemszerűen nem szólunk struktúrált és nem struktúrált programok bonyolultságának összehasonlításáról sem.

Szerkezeti bonyolultság címen kétféle dologról beszélhetünk: az **algoritmus bonyolultságáról**, valamint az **adatszerkezet bonyolultságáról**. Mindkettő újabb részekből áll össze, amelyek s a fentiek is sokszor csak egymás kárára csökkenthetők. Nem álla-

pítható meg pontosan az sem, hogy a logikai bonyolultságban (a megértés, illetve az elkészítés nehézségében) ezek közül melyik milyen súllyal szerepel.

Emiatt azt állapíthatjuk meg, hogy a programok logikai bonyolultsága sok részjellemző valamilyen -gyakorlatilag ismeretlen- függvényeként írható le:

$$B=f(B_1,B_2,...,B_n)$$

1. Az algoritmus bonyolultsága

Az algoritmus bonyolultsága újabb két tényezőből áll össze: a **szerkezet bonyolultságából** és a **kifejezés bonyolultságából**.

A szerkezet bonyolultságának egyik lehetséges mérőszáma gráfelméleti vizsgálatokból származik.

Definíció: Egy programgráf **ciklikus bonyolultsága** az élei számából kivonva a csúcsai számát.

Könnyű belátni, hogy az így definiált mérőszám pontosan 1-gyel nagyobb a programban szereplő döntés csomópontok számánál, ami pedig struktúrált programra az elágazások és a ciklusok együttes számával egyenlő.

Ez a mérőszám azonban nagyon *furcsa* értéket ad összetett feltételű struktúrákra.

Példa: algoritmus — ciklikus bonyolultsága

`Ha p akkor f`	2
`Ha p és q akkor f`	2
`Ha p akkor Ha q akkor f`	3

A példából érezhető, hogy a második eset bonyolultsága valahol a két szélső között van (2.5?). Erre a problémára érzett rá Myers, s adta meg a ciklikus bonyolultsági szám módosítását:

Definíció: Egy programgráf **módosított ciklikus bonyolultsága** egy számpár: a ciklikus bonyolultsági szám, valamint a ciklikus bonyolultág megnövelve a feltételekben szereplő diszjunkciók és konjunkciók (vagy-műveletek és és-műveletek) számával.

Példa: algoritmus — ciklikus bonyolultsága

`Ha p akkor f`	2,2

```
Ha p és q akkor f          2,3
Ha p akkor Ha q akkor f    3,3
```

Sajnos ennek a mértéknek is lehet logikailag ellentmondó példát találni:

Példa: algoritmus ciklikus bonyolultsága

```
Ha p és q akkor f          2,3
r:=p és q; Ha r akkor f    2,2
```

Azt állapíthatjuk meg e példából, hogy az első algoritmusrészlet átalakításával a **szerkezet bonyolultsága csökkent**, de ezzel szemben a **kifejezés bonyolultsága nőtt.** (A példa egyben arra is utal, hogy miféle kapcsolat lehet e kétféle bonyolultsági jellemző között.)

Érezhető, hogy a logikai bonyolultságot lényegesen növelik az egymásba ágyazott struktúrák. Ezt a jellemzőt próbálja meghatározni a következő mérőszám.

Definíció: Egy programgráf **mélységi bonyolultságát** a következőképpen számíthatjuk ki: Vegyük a programgráf elemi struktúráit (elágazások, ciklusok), rendeljük hozzájuk azt a kitevőjű kettőhatványt, ahány magasabbrendű struktúra belsejében vannak, majd adjuk össze ezeket a számokat.

Megjegyzés: Az egymásbaágyazott struktúrák belsejében levő algoritmikus szerkezetek súlyának -a kettőhatványok helyett- más súlyfüggvényt is választhatnánk, a fentit nem elméleti megfontolások, hanem tanítási tapasztalatok támasztják alá.

Példa: A IV.1.4. fejezet első példájának három változatára a ciklikus, a módosított ciklikus és a mélységi bonyolultság a következőképpen alakul:

Ciklikus:	6	5	3
Módosított ciklikus:	6,6	5,5	3,4
Mélységi:	11	9	4

Az absztrakció a logikai bonyolultságot csökkenti, így érdemes egy olyan mérőszámot is bevezetni, amely ezt is figyelembe veszi.

Definíció: Egy programgráf **absztrakciós bonyolultságát** a következőképpen számíthatjuk ki: Vegyük az eljárások számát + az egyes eljárások mélységi bonyolultságát!

Példa: A IV.1.5. fejezet második példáján a fenti négy bonyolultságdefiníció:

Ciklikus:	4	4

Módosított ciklikus:	4,5	4,5
Mélységi:	8	8
Absztrakciós:	8	6

Megjegyzés: A ciklikus bonyolultság Myers-féle módosításának megfelelőit elvégezhetjük a mélységi, illetve az absztrakciós bonyolultságon is, ez azonban a módosított ciklikus bonyolultsághoz képest nem ad a logikai bonyolultságról újabb információt.

A **kifejezés bonyolultságát** a programfüggvény, mint kifejezés bonyolultsága, illetve a program szövegében szereplő kifejezések bonyolultsága adhatja meg. A kifejezések bonyolultságát legegyszerűbb esetben a bennük szereplő műveletek számával adhatjuk meg. Műveletnek lehet értelmezni programok esetén:

- az aritmetikai és logikai műveleteket,
- a függvényhívásokat,
- az értékmozgatásokat (értékadás, beolvasás, kiírás) és
- az eljáráshívásokat.

Ebben a részben nem szánunk külön fejezeteket az egyes bonyolultsági jellemzők javításának, mint az tettük a végrehajtási idő vagy a helyfoglalás esetében. Itt ugyanis az egyes jellemzők között szoros összefüggés van, s az egyik mérték csökkentése szinte mindig mások csökkentésével jár együtt.

1.1. A kivételes eset kiküszöbölése

Egy feladat megoldása legtöbbször azért bonyolult, mert a megoldás során sokféle speciális eset kezelését kell elvégezni. Ezt a bonyolultságot csökkenthetjük, ha ezeket a speciális, kivételes eseteket megszüntetjük.

Ezzel a módosított ciklikus bonyolultság mindenképpen csökken, de egyes esetekben a ciklikus bonyolultság is.

A kivétel itt is -mint az I. rész 2.2. fejezetében- a ciklusfeltételben és a ciklusmagban is lehet, illetve itt a cikluson kívüli részekkel is érdemes foglalkozni (ami a végrehajtási idő szempontjából általában mellékes).

Feladat: Egy szövegben határozzuk meg a szavak számát!

Megoldás: A szavak száma megegyezik a szavak kezdeteinek számával. Egy szó úgy kezdődik, hogy szóköz után betű következik, kivéve az első szót. Ha a szöveg első karaktere szóköz, akkor a fenti állítás az első szóra is igaz, ha betű, akkor nem (**ez a kivételes eset**).

```
Szavak_száma(MONDAT,SDB):
  SDB:=0; N:=hossz(MONDAT)
  Ciklus I=1-től N-1-ig
    Ha MONDAT(I)=" " és MONDAT(I+1)≠" " vagy
       I=1 és MONDAT(I)≠" " akkor SDB:=SDB+1
  Ciklus vége
Eljárás vége.
```

Helyezzünk el ekkor a szöveg első karaktere elé egy szóközt, így a kivételt megszüntethetjük, s a megoldó programot rövidebbé, egyszerűbbé tehetjük.

```
Szavak_száma(MONDAT,SDB):
  MONDAT:=elejére(" ",MONDAT)
  SDB:=0; N:=hossz(MONDAT)
  Ciklus I=1-től N-1-ig
    Ha MONDAT(I)=" " és MONDAT(I+1)≠" " akkor SDB:=SDB+1
  Ciklus vége
Eljárás vége.
```

Itt a ciklikus bonyolultság változatlan maradt, a módosított ciklikus bonyolultság azonban jelentősen csökkent. (Ezzel együtt nőtt a kifejezés bonyolultsága.) Figyeljük meg, hogy a bonyolultság csökkentésében szerepet játszik az adatabsztrakció: a szövegtípus egy művelete (elejére) teszi egyszerűbbé az algoritmust.

Itt különböző implementációs lehetőségeket kell számbavennünk. Elképzelhető, hogy a MONDAT egy vektor, amit 1-től indexelünk. Ennek 0. elemébe helyezzük el a bevezető szóközt. Ha a MONDAT-ot egy file-ból olvassuk be, s egy szöveg típusú változóba tesszük, akkor a beolvasás megkezdése előtt elhelyezhetjük benne (esetleg az input pufferban) ezt a szóközt, hiszen beolvasáskor úgyis kell alkalmazni a konkatenáció műveletét!

Olyan eset is előfordulhat, amikor a bonyolultság csökkentése (a kivétel megszüntetése) a végrehajtási időt növeli. Nézzük meg erre az összefuttatás klasszikus változatait!

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

Ebben az algoritmusban azért van szükség a két utolsó ciklusra, mivel nem garantálható, hogy a két sorozatból egyszerre fogyjanak el az elemek. Ha ezt megtesszük két **fiktív elem** elhelyezésével, akkor ezek a ciklusok feleslegessé válnak.

Ebben a példában a ciklikus bonyolultság csökkenése minden más mérőszám (még a kifejezés bonyolultsága) csökkenését is okozza.

```
Összefuttatás(A(),N.B(),M,C(),K):
  I:=1; J:=1; K:=0; A(N+1):=+∞; B(M+1):=+∞
  Ciklus amíg I<N+1 vagy J<M+1
    K:=K+1
    Elágazás
     A(I)<B(J) esetén C(K):=A(I); I:=I+1
     A(I)=B(J) esetén C(K):=A(I); I:=I+1; J:=J+1
     A(I)>B(J) esetén C(K):=B(J); J:=J+1
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

1.2. Funkciók elhagyása

A feladatok megoldására szolgáló programozási tételek alapján sokszor nem a legegyszerűbb megoldást kapjuk. Ennek oka az, hogy az általános algoritmusok sohasem használhatják ki az egyes feladatok specialitásait. A bonyolultságot (a végrehajtási időt és a helyfoglalást is) csökkentheti ilyenkor, ha **elhagyjuk a felesleges funkciókat**.

Ez minden esetben csökkenti a ciklikus bonyolultságot, ami a módosított ciklikus és a mélységi bonyolultság csökkenésével is jár.

Kiválogatási, szétválogatási feladatok esetén gyakori, hogy az eredményül kapott sorozatokat ki kell írni, s más teendő nincs velük. Ekkor célszerű a megfelelő elemek **kigyűjtését elhagyni**, s helyette azonnal -illetve amikor lehetséges- kiírni őket. Nézzünk példaként egy szétválogatási feladatot!

Feladat: Egy osztálynévsor alapján adjuk meg külön-külön az osztály lány-, illetve fiútanulóinak névsorát!

Megoldás: A klasszikus megoldás egy szétválogatás egy új vektorba, majd pedig két kiíró ciklus. (Most ne foglalkozzunk azzal, hogy hogyan lehet valakiről megállapítani egy névsor alapján, hogy fiú-e vagy lány!)

```
Névsorok(OSZT(),N):
  LDB:=0; FDB:=0
  Ciklus I=1-től N-ig
    Ha lány(OSZT(I)) akkor LDB:=LDB+1; SZ(LDB):=OSZT(I)
                     különben FDB:=FDB+1; SZ(N+1-FDB):=OSZT(I)
  Ciklus vége
  Ki: "Lányok névsora"
  Ciklus I=1-től LDB-ig
    Kiírás(SZ(I))
  Ciklus vége
  Ki: "Fiúk névsora"
  Ciklus I=N-től N+1-FDB-ig -1-esével
    Kiírás(SZ(I))
  Ciklus vége
Eljárás vége.
```

Ebben a feladatban a megoldást egy azonnali kiírássá és egy kiválogatássá egyszerűsíthetjük.

```
Névsorok(OSZT(),N):
  FDB:=0; Ki: "Lányok névsora"
  Ciklus I=1-től N-ig
    Ha lány(OSZT(I)) akkor Ki: OSZT(I)
                     különben FDB:=FDB+1; SZ(FDB):=OSZT(I)
  Ciklus vége
  Ki: "Fiúk névsora"
  Ciklus I=1-től FDB-ig
    Kiírás(SZ(I))
  Ciklus vége
Eljárás vége.
```

Ebben a fejezetben ezután az összefuttatás algoritmusát fogjuk használni e módszer illusztrálására.

```
Összefuttatás(N,A(),M,B(),C()):
  A(N+1):=+∞; B(M+1):=+∞; I:=1; J:=1; K:=0
  Ciklus amíg I<N+1 vagy J<M+1
    K:=K+1
    Elágazás
     A(I)<B(J) esetén C(K):=A(I); I:=I+1
     A(I)=B(J) esetén C(K):=A(I); I:=I+1; J:=J+1
     A(I)>B(J) esetén C(K):=B(J); J:=J+1
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

Megjegyzés: Logikailag nem azonos a háromirányú elágazás és a neki megfelelő egymásba ágyazott kétfelé ágazások bonyolultsága, a bonyolultsági mértékben azonban ezt nem vesszük figyelembe.

Feladat: Adott két rendezett sorozat, futtassuk össze őket egy rendezett sorozattá! A két sorozatban nincs azonos elem!

Megoldás: Itt a feladat specialitása, hogy A()-és B()-beli elem között sohasem fordulhat elő egyenlőség. Ez a megoldást jelentősen egyszerűsíti:

```
Összefuttatás(N,A(),M,B(),C()):
  A(N+1):=+∞; B(M+1):=+∞; I:=1; J:=1; K:=0
  Ciklus amíg I<N+1 vagy J<M+1
    K:=K+1
    Ha A(I)<B(J) akkor C(K):=A(I); I:=I+1
                 különben C(K):=B(J); J:=J+1
  Ciklus vége
Eljárás vége.
```

Itt tehát a belső elágazás egymásbaágyazottsági mértéke csökkent, ami a mélységi bonyolultságot csökkenti.

Feladat: Adott egy árukészletet tartalmazó file (név, mennyiség) és egy eladásokat tartalmazó file (név, mennyiség). Időszerűsítsük az utóbbi segítségével az árukészletet! (Minden áru csak egyszer szerepel bennük, s árunév szerint rendezettek.)

Megoldás: Itt a **specialitás** az, hogy eladni csak abból lehet, ami van. Tehát a 2. file-ban nem lehet olyan rekord, ami az 1.-ben nem volt.

Megjegyzés: File-ok helyett itt is vektorokat használunk, hogy a megoldást a fejezet első algoritmusához hasonlíthassuk.

Tároljuk az adatokat a következő vektorokban:

NÉV(N) - az áruk nevei
KÉSZLET(N) - a készlet
ELADÁSNÉV(M) - az eladott áruk nevei
ELADÁS(M) - az eladott mennyiség
ÚJKÉSZLET(N) - az új készlet

```
Időszerűsítés:
  ELADÁSNÉV(M+1):=+∞; I:=1; J:=1
  Ciklus amíg I≤N
    Ha NÉV(I)<ELADÁSNÉV(J)
          akkor ÚJKÉSZLET(I):=KÉSZLET(I)
       különben ÚJKÉSZLET(I):=KÉSZLET(I)-ELADÁS(J); J:=J+1
    Elágazás vége
    I:=I+1
  Ciklus vége
Eljárás vége.
```

1.3. Funkciók szétválasztása

A következő módszer a bonyolultság csökkentésére a **funkciók szétválasztása**. Ha olyan programrészünk van, amely kétféle dolgot csinál, s ezeket lehetőe egymás után is elvégezni, akkor ezt tegyük is úgy (hacsak a sebesség vagy a helyfoglalás problémája nem akadályozza ezt meg)!

Feladat: Adjuk meg egy számsorozat összes maximális értékű elemének a sorszámát! A „klasszikus" megoldás:

```
Maximumok(N,A(),DB,S()):
  MX:=A(1); DB:=1; S(DB):=1
  Ciklus I=2-től N-ig
    Ha A(I)>MX akkor DB:=0; MX:=A(I)
    Ha A(I)=MX akkor DB:=DB+1; S(DB):=I
  Ciklus vége
Eljárás vége.
```

Ebben az esetben a maximális érték kiválasztása, illetve az ezzel az értékkel rendelkezők kiválogatása egymás után is elvégezhető.

```
Maximumok(N,A(),DB,S()):
  MX:=A(1); DB:=0
  Ciklus I=2-től N-ig
    Ha A(I)>MX akkor MX:=A(I)
  Ciklus vége
```

```
  Ciklus I=1-től N-ig
    Ha A(I)=MX akkor DB:=DB+1; S(DB):=I
  Ciklus vége
Eljárás vége.
```

A módosítás után a hasonlítások száma ugyan 1-gyel több lett (sőt a „plusz" ciklus megszervezése is elvisz valamennyi időt), viszont a ciklusban végrehajtandó értékadások száma adott esetben lényegesen csökkenhet (azáltal, hogy az addigi maximális értékű elemeket esetleg feleslegesen nem gyűjtögetjük). Így -általában mondhatjuk- még gyorsabb programot is írunk, mint az előző esetben.

Megjegyzés: Ez érdekes példa, ugyanis a **logikai bonyolultság** úgy **csökken**, hogy közben a **szerkezeti bonyolultság nő**. Ennek oka, hogy az első megoldásnál a szerkezetből nem látszik, hogy a két elágazás nem független egymástól. (A kifejezés bonyolultsága viszont csökken.)

Hasonló feladat az, amikor egy sorozat maximumát és minimumát egyszerre kell meghatározni. Ezt is megírhatjuk egyetlen ciklusban, illetve szétválasztva külön maximum- és külön minimumkiválasztásra.

Általában azt mondhatjuk, hogy egyszerűbb megoldást kapunk, ha a programozási tételeket **nem egymás belsejében, hanem egymás után** alkalmazzuk egy feladat megoldásában.

Kétféle egyszerű **alaptípust** különböztethetünk meg, az egyikben az

$$Y := f(g(X))$$

egymásbaágyazott függvénykiszámítás helyett a

$$Z := g(X): Y := f(Z)$$

kiszámítási sorrendű algoritmus -bár hosszabb, de- egyszerűbb. A másikban az

$$Y, Z := f(X), g(X)$$

párhuzamos kiszámítás helyett pedig az

$$Y := f(X): Z := g(X)$$

egymásutáni kiszámítás.

1.4. A fiktív kezdőértékadás

Egy másik egyszerű módszer a **fiktív kezdőértékadás**. Sok algoritmus ugyanis attól válik bonyolulttá, hogy egy változó kezdőértékét az algoritmus belsejében határozzuk

meg, s annak vizsgálata, hogy meg kell-e ezt tenni, bonyolítja a megoldást (általában a struktúrák egymásba ágyazásának mélységét, illetve a feltételek összetettségét növeli).

Feladat: Adott egy sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Adjuk meg a sorozat legnagyobb T tulajdonságú elemét!

Megoldás: A feladatot két lépésben oldhatjuk meg: alkalmazzuk a **kiválogatás** és, ha van T tulajdonságú, a **maximumkiválasztás** tételeket!

```
Maximális(N,A(),VAN,MAX):
  DB:=0
  Ciklus I=1-től N-ig
    Ha A(I) T tulajdonságú akkor DB:=DB+1; B(DB):=I
  Ciklus vége
  VAN:=DB>0
  Ha VAN akkor MAX:=B(1)
                Ciklus I=2-től DB-ig
                  Ha A(MAX)<A(B(I)) akkor MAX:=B(I)
                Ciklus vége
  Elágazás vége
Eljárás vége.
```

A megoldás a normál maximumkiválasztásnál lényegesen bonyolultabb lett. Ennek oka, hogy nem tudtuk garantálni, hogy az első elem T tulajdonságú, hiszen akkor vehettük volna ezt kezdőértéknek. Próbáljuk **megkeresni** az első T tulajdonságút, s attól kezdeni a **maximumkiválasztást**!

```
Maximális(N,A(),VAN,MAX):
  I:=1
  Ciklus amíg I≤N és A(I) nem T tulajdonságú
    I:=I+1
  Ciklus vége
  VAN:=I≤N
  Ha VAN akkor
       MAX:=I
       Ciklus J=I+1-től N-ig
         Ha A(J) T tulajdonságú és A(MAX)<A(J) akkor MAX:=J
       Ciklus vége
  Elágazás vége
Eljárás vége.
```

A megoldás csak kicsit lett egyszerűbb: nincs szükség a B() vektor használatára. A ciklikus bonyolultság csökkent (egy elágazással kevesebb van a megoldásban), a módosított bonyolultságok azonban növekedtek (a legbelső elágazás feltétele összetett lett).

Ha a MAX változónak tudnánk olyan kezdőértéket adni, amely nem valóságos elemsorszám az A() vektorban, de az első T tulajdonságú elem biztosan nagyobb lesz nála,

akkor az első ciklusra nem lenne szükség. Adjunk a MAX változónak egy fiktív kezdőértéket, pl. 0-t, s tegyünk az A() vektor 0. elemébe is egy fiktív értéket!

```
Maximális(N,A(),VAN,MAX):
  MAX:=0; A(0):=-∞
  Ciklus J=1-től N-ig
    Ha A(J) T tulajdonságú és A(MAX)<A(J) akkor MAX:=J
  Ciklus vége
  VAN:=(MAX>0)
Eljárás vége.
```

Feladat: Írjuk ki egy mátrix azon sorainak sorszámát, amelyek sorösszege nagyobb, mint a szomszédos sorok sorösszege!

Megoldás:

```
Lokális_maximumok(N,M,A(,)):
  Ciklus I=1-től N-ig
    S:=0
    Ciklus J=1-től M-ig
      S:=S+A(I,J)
    Ciklus vége
    Elágazás
     I=1 esetén F:=S
     I=2 esetén E:=S
     I>2 esetén Ha F<E és E>S akkor Ki: I-1
                                    F:=E; E:=S
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

Itt az egyszerűsítést úgy lehet elérni, hogy az F és E változóknak megfelelő kezdőértéket adunk (úgy hogy a feltétel ne legyen igaz I=1 és I=2 esetén). Legyen ez a kezdőérték a számítógépben ábrázolható legnagyobb valós szám! Jelölje ezt az értéket +∞!

A megoldás:

```
Lokális_maximumok(N,M,A(,)):
  F:=+∞; E:=+∞
  Ciklus I=1-től N-ig
    S:=0
    Ciklus J=1-től M-ig
      S:=S+A(I,J)
    Ciklus vége
    Ha F<E és E>S akkor Ki: I-1
    F:=E; E:=S
  Ciklus vége
Eljárás vége.
```

Ekkor tehát az algoritmus belsejéből eltűnt a kezdőértékadás, s a szükségességének vizsgálata is.

1.5. Adatabsztrakció

Ebben a fejezetben foglalkozunk a legerőteljesebb, hatásában a többieknél sokkal lényegesebb módszerrel, az adatabsztrakcióval. E témával *Módszeres programozás* sorozatunk több másik tagja részletesebben is foglalkozott, itt csupán hivatkozunk azokra.

Típusok, típusműveletek, típuskonstansok alkalmas használata nagyban egyszerűsítheti az algoritmusokat.

Egy jellemző, negatív példát találhatunk Sedgewick -egyébként magas színvonalú- művében, a Quicksort eljárásban:

```
Quick:
  L:=1; R:=N; P:=2
  Ciklus
    Ha R>L akkor Szétválogat(L,R,I)
                 Ha I-L>R-I akkor S(P):=L; S(P+1):=I-1; L:=I+1
                           különben S(P):=I+1; S(P+1):=R; R:=I-1
                 Elágazás vége
                 P:=P+2
          különben P:=P-2; L:=S(P); R:=S(P+1)
    Elágazás vége
  amíg P≠0
  Ciklus vége
Eljárás vége.
```

Ha ugyanebben az algoritmusban a dőltbetűs részeket kicseréljük a megfelelő veremkezelő eljárásokra, sokkal egyszerűbb, áttekinthetőbb programot kapunk:

```
Quick:
  L:=1; R:=N; Vereminicializálás
  Ciklus
    Ha R>L akkor Szétválogat(L,R,I)
                 Ha I-L>R-I akkor Verembe(L,I-1); L:=I+1
                           különben Verembe(I+1,R); R:=I-1
                 Elágazás vége
          különben Veremből(L,R)
    Elágazás vége
  amíg a verem nem üres
  Ciklus vége
Eljárás vége.
```

Az új típus bevezetésével az algoritmus áttekinthetőbbé, egyszerűbbé vált, a különböző célú értékadások egymással nem keverednek. Itt most ugyan a kifejezés bonyolult-

sága csökkent, de ha egy helyen kellett volna verembe tenni, akkor még ezt sem mondhatnánk el a módosított megoldásról. Az egyszerűsítés alapja bizonyos utasításcsoport elnevezésében keresendő. Ez az utasításcsoport éppen egy típus egy műveletét jelenti.

Egyszerűbb példán nézzük meg új típuskonstansok bevezetésének hatását az algoritmusra, egy mátrix elemeinek nullázásán:

```
Ciklus I=1-től N-ig
  Ciklus J=1-től N-ig
    A(I,J):=0                    → A:=nullmátrix
  Ciklus vége
Ciklus vége
```

A fentihez hasonló egyszerű típusműveletek bevezetésével is egyszerűsíthetjük programunkat. A programozási nyelvekben általában csak elemi típusú adatokat lehet beolvasni, kiírni. Ha megírnánk összetett típusokra a beolvasó, kiíró műveleteket, akkor az algoritmusaink egyszerűbbek lehetnének. Például a mátrixbeolvasó eljárás:

```
Mátrixbeolvasás(A(),N):
  Ki: "Kérem a mátrix elemeit!"
  Ciklus I=1-től N-ig
    Ki: I,". sor elemei:"
    Ciklus J=1-től N-ig
      Ki: J,". elem:"; Be: A(I,J)
    Ciklus vége
  Ciklus vége
Eljárás vége.
```

A beolvasó eljárás megírása után az összes helyen, ahol mátrixbeolvasás volt, az algoritmust sokkal egyszerűbbé tehetjük, a ciklusok helyére egy eljáráshívást teszünk.

A következő feladatban a szövegtípust általánosítjuk (mint a μlógia sorozat 14. kötetében tettük), s az új típusműveletek használata hozza meg az absztrakciós bonyolultság csökkenését.

Feladat: Egy szöveg szavait kell kiírni egymás után, a szavakat vessző választja el egymástól, s a szóközöket nem szabad figyelembe venni. Az utolsó szót NEWLINE zárja. Adjuk meg a szavak számát is!

Megoldás:

```
Szavakra_tagolás:
  SZÓ:=""; DB:=0
  Ciklus
     Be: BETŰ
     Ha BETŰ="," vagy BETŰ=NEWLINE
                  akkor Ki: SZÓ; SZÓ:=""; DB:=DB+1
               különben Ha BETŰ≠" " akkor SZÓ:=SZÓ+BETŰ
  amíg BETŰ≠NEWLINE
  Ciklus vége
  Ki: DB
Eljárás vége.
```

Hogy az algoritmus megfogalmazása körül valami nincs rendjén, már az a tény is jelzi, hogy a SZÓ:="" kezdőértékadást két helyen is el kellett helyeznünk. Sőt a bonyolódás miatt az sem látszik világosan, hogy ezek egyáltalán jó helyen vannak-e, vagy sem. A bonyodalmakat (értsd: az algoritmus bonyolultságát) itt az okozza, hogy egyetlen sorba foglalunk össze 4 funkciót is: a szószámlálást, a szó kezdőértékadást, a szógyűjtést, a szóköz kihagyást. Az egyszerűbb, bár -kétségkívül- hosszabb megoldásban ezeket a funkciókat szintekre tagolva, könnyen érthetően fejezzük ki.

1. szint: a szöveg szavakból áll, az utolsó után NEWLINE áll.

Szöveg=Sorozat(Szó): a szöveg szavak sorozata.

```
Szavakra_tagolás:
  DB:=0
  Ciklus
    Szóolvasás(SZÓ,ELVÁLASZTÓ); DB:=DB+1; Ki: SZÓ
  amíg ELVÁLASZTÓ≠NEWLINE
  Ciklus vége
  Ki: DB
Eljárás vége.
```

2. szint: a szó karakterekből áll, amit elválasztójel (vessző, NEWLINE) zár le.

Szó=(Sorozat(Karakter),Elválasztó): a szó karakterek sorozata, amit egyetlen elválasztójel követ.

```
Szóolvasás(SZO,ELVALASZTO):
  SZÓ:=""; Olvasás(ELVÁLASZTÓ)
  Ciklus amíg ELVÁLASZTÓ≠"," és ELVÁLASZTÓ≠NEWLINE
    SZÓ:=SZÓ+ELVÁLASZTÓ; Olvasás(ELVÁLASZTÓ)
  Ciklus vége
Eljárás vége.
```

3. szint: a beolvasott karakterek közül ki kell hagyni a szóközöket.

Karakter: olyan jel, amit tetszőleges számú szóköz előzhet meg.

```
Olvasás(KARAKTER):
  Ciklus
    Be: KARAKTER
  amíg KARAKTER=" "
  Ciklus vége
Eljárás vége.
```

Itt a szerkezeti bonyolultság nem változott, ugyanakkor a logikai bonyolultság lényegesen csökkent. Ennek oka az, hogy a második megoldásban **egy-egy programegységhez (eljáráshoz)** mindig **egy döntés tartozik**, míg ez a három döntés az első megoldás egyetlen eljárásában volt elrejtve.

Vizsgáljuk meg e két megoldásra a fejezet elején felsorolt bonyolultsági mértékeket!

Ciklikus:	3	3
Módosított ciklikus:	3,4	3,4
Mélységi:	7	7
Absztrakciós:	7	3
Mód. absztrakciós:	7,9	3,4

Megállapíthatjuk tehát, hogy az absztrakciós bonyolultság lényegesen csökkent, miközben a többi mérték nem változott.

Az ilyen megoldásnak nagy előnye van módosítások esetén is, módosítsuk tehát a feladatot!

Feladat: Adjuk meg a leghosszabb szót!

Megoldás: Csak a megoldás első szintjén kell módosítani, s a többi rész változatlan.

```
Szavakra_tagolás:
  LEGHOSSZABB:=""
  Ciklus
    Szóolvasás(SZO,ELVALASZTO)
    Ha hossz(SZO)>hossz(LEGHOSSZABB) akkor LEGHOSSZABB:=SZO
  amíg ELVALASZTO≠NEWLINE
  Ciklus vége
  Ki: LEGHOSSZABB
Eljárás vége.
```

Megjegyzés: Vegyük észre, hogy itt használtuk a fiktív kezdőértékadás elvét is! A LEGHOSSZABB:="" értékadás olyan értéket ad a leghosszabb szót tartalmazó változónak, amelynél bármely szó hosszabb lesz. Így nincs szükség a ciklus előtt az első szó meghatározására.

2. Az adatszerkezet bonyolultsága

Az adatszerkezet, az adattípusok bonyolultságát is kétféle szempont szerint vizsgálhatjuk. egyrészt nézhetjük a **struktúra bonyolultságát**, másrészt pedig **a típus műveleteinek bonyolultságát.**

Struktúra szerint nyilván a legegyszerűbbek a struktúrálatlan, skaláradatok. Következő szint lehet az ezekből közvetlenül felépülő -egyeten típuskonstrukciós eszközt alkalmazó- összetett típusok, ...

Definíció: Egy adattípus **struktúrális bonyolultságának** nevezzük a típus definiálásában szereplő típuskonstrukciós eszközök számát.

Példa:

Egész	- 0
Rekord (x,y: Valós)	- 1
Tömb (1..N,Egész)	- 1
Rekord(db: Egész, Tömb(1..N,Karakter))	- 2
Test=Sorozat(Lap)	- 4
Lap=Sorozat(Szakasz)	
Szakasz=Tömb(1..2,Pont)	
Pont=Rekord(x,y: Valós)	
Test=Sorozat(Sorozat(Tömb(1..2,Rekord(x,y: Valós))))	- 4

Figyeljük meg a **típus struktúrális bonyolultsága** és az **algoritmus ciklikus bonyolultsága** közötti hasonlóságot: mindkettőt a bennük szereplő struktúrák számával mérhetjük.

Itt is felfedezhetjük azt a jellemzőt, hogy a típusok egymásbaágyazásának mélységével a bonyolultság nemlineárisan nő.

Az utolsó két példa ugyanazt a típust tartalmazza, mégsem tűnnek azonos bonyolultságúnak. Az első viszonylag könnyen megérthető, a második viszont jóval nehezebben. Ez arra utal, hogy a részstruktúrák elnevezése (az absztrakció alkalmazása) könnyebbé teszi a típus megértését.

Ezek alapján a típusokra is definiálhatjuk a mélységi és az absztrakciós bonyolultságot.

Definíció: Egy adattípus **mélységi bonyolultságát** a következőképpen számíthatjuk ki: Vegyük az adattípus definiálásoz felhasznált típuskonstrukciós eszközöket, rendeljük hozzájuk azt a kitevőjű kettőhatványt, ahány magasabbrendű struktúra belsejében vannak, majd adjuk össze ezeket a számokat.

Definíció: Egy adattípus **absztrakciós bonyolultságát** a következőképpen számíthatjuk ki: Vegyük a résztípusai számát + az egyes résztípusok mélységi bonyolultságát!

Példa: (típus bonyolultsági mértékei)

Test=Sorozat(Lap)
Lap=Sorozat(Szakasz)
Szakasz=Tömb(1..2,Pont)
Pont=Rekord(x,y: Valós)
mélységi - 15
absztrakciós - 7

Eddig a struktúrát a típus definiálója szempontjából vizsgáltuk, most térjünk át a típus felhasználója szerinti vizsgálatokra.

A felhasználó számára a típus annál bonyolultabbnak tűnik, minél mélyebbre kényszerül belelátni a belsejébe.

Definíció: Egy adattípus **hivatkozási bonyolultsága** a leghosszabb hivatkozási mélység, melyet a felhasználónak használnia kell.

Példa: (skalárok és összetett típusok hivatkozási bonyolultsága)

X - 1
A[I] - 2
B.X -2
C[I,J] - 3
D[I].Y - 3

A hivatkozási bonyolultság a típusabsztrakció megfelelő alkalmazásával csökkenthető.

Példa:

Test=Sorozat(Lap)
Lap=Sorozat(Szakasz)
Szakasz=Tömb(1..2,Pont)
Pont=Rekord(x,y: Valós)
Test=Sorozat(Sorozat(Tömb(1..2,Rekord(x,y: Valós))))

Itt mindkét típus **hivatkozási bonyolultsága** maximum 5 lehet, de ez az első esetben -megfelelő típusműveletek definiálásával- 2-re csökkenthető.

Térjünk rá ezek után a **típus műveleteinek bonyolultságára.** Itt is megvizsgálhatjuk a típus definiálója szerinti bonyolultságot, valamint a használója szerintit.

A típus készítője számára a műveletek algoritmusokként jelentkeznek, így ennek bonyolultsága megegyezik a szükséges algoritmusok bonyolultságával.

A típus használója számára a típusműveletek megértése annál nehezebb, minél több, egymással ki nem fejezhető van belőlük.

Definíció: Egy **típus műveleti bonyolultsága** legyen a típus független, egymással ki nem fejezhető műveletei száma.

Ez a mérték mintha arra utalna, hogy minden típushoz definiáljunk egyetlen -sokparaméterű- univerzális műveletet. Ez nyilvánvalóan nem igaz. A típus műveleteit úgy kell definiálni, hogy különböző feladatokra különböző műveletei legyenek, lehetőleg minimális paraméterszámmal.

Példa: (a BASIC programnyelv szövegtípusa műveletei, s a minimális műveletkészlet)

Az összes művelet:

+(X$,Y$), LEN(X$), LEFT$(X$,DB), MID$(X$,A,DB), MID$(X$,A), RIGHT$(X$,DB).

A többivel kifejezhető műveletek:

LEFT$(X$,DB) = MID$(X$,1,DB)
MID$(X$,A) = MID$(X$,A,LEN(X$)-A+1)
RIGHT$(X$,DB) = MID$(X$,LEN(X$)-DB+1,DB)

A minimális műveletkészlet:

+(X$,Y$), LEN(X$), MID$(X$,A,DB).

A műveletek számának növelése viszont egyszerűbbé teszi a típust felhasználó algoritmusokat, így mindenképpen előnyös a típus használóinak.

Vizsgáljuk meg ezzel szemben a LOGO programnyelv megfelelő függvényeinek minimális készletét!

Példa:

EMPTY? sorozat, FIRST sorozat, BUTFIRST sorozat, FPUT elem sorozat

2.1. Típuskonstrukciós eszközök összevonása

Nem egyforma bonyolultságú a sorozaton (tömbön, vermen, soron, ...) belüli rekordok alkalmazása és a rekordon belüli sorozatok (tömbök, ...) alkalmazása. A bonyolultságot minden esetben csökkentheti, ha egy rekord különböző mezői azonos típusú sorozatok, amelyet helyettesíthetünk rekordok sorozatával.

Példa:

Személyi adatok=Rekord(Nevek, Címek, Telefonszámok)
Nevek=Tömb(1..N,Név)
Címek=Tömb(1..N,Cím)

Telefonszámok=Tömb(1..N,Telefonszám)
Struktúrális bonyolultsága: 4.
Mélységi bonyolultsága : 7.
Személyi adatok=Tömb(1..N,Személy)
Személy=Rekord(Név, Cím, Telefonszám)
Struktúrális bonyolultsága: 2.
Mélységi bonyolultsága : 3.

2.2. Adatabsztrakció

A típusabsztrakció hasznosságára itt is csupán egy példával utalunk: a 2. fejezet elején említett *Test* típus kirajzoló eljárását írjuk meg kétféleképpen. (Ez egyben az algoritmikus absztrakcióra is példa lesz.)

Test=Sorozat(Lap)
Lap=Sorozat(Szakasz)
Szakasz=Tömb(1..2,Pont)
Pont=Rekord(x,y: Valós)

A típus ilyen definiálása alapján a típusabsztrakciós bonyolultságát a lehető legkisebb lesz. Az algoritmus absztrakciós bonyolultságát és a típus hivatkozási bonyolultságát hasonlítjuk össze.

```
Rajzol(T: Test):
  Ciklus I=1-től elemszám(T)-ig
    Ha látszik(T(I)) akkor Ciklus J=1-től elemszám(T(I))-ig
                             Transzformál(T(I)(J)(1).x,P1.x)
                             Transzformál(T(I)(J)(1).y,P1.y)
                             Transzformál(T(I)(J)(2).x,P2.x)
                             Transzformál(T(I)(J)(2).y,P2.y)
                             Rajzol(P1.x,P1.y,P2.x,P2.y)
                           Ciklus vége
    Elágazás vége
  Ciklus vége
Eljárás vége.
```

Algoritmusbonyolultság	Adatbonyolultság
Absztrakciós: 7	Hivatkozási: 5

```
Rajzol(T: Test):
  Ciklus I=1-től elemszám(T)-ig
    Ha látszik(T(I)) akkor Laprajzolás(T(I)))
  Ciklus vége
Eljárás vége.
```

```
Laprajzolás(L:Lap):
  Ciklus J=1-től elemszám(L)-ig
    Szakaszrajzolás(L(J))
  Ciklus vége
Eljárás vége.

Szakaszrajzolás(S: Szakasz):
  Átalakít(S(1),P1); Átalakít(S(2),P2)
  Rajzol(P1.x,P1.y,P2.x,P2.y)
Eljárás vége.

Átalakít(R: Pont,P: Pont):
  Transzformál(R.x,P.x); Transzformál(R.y,P.y)
Eljárás vége.
```

Algoritmusbonyolultság	Adatbonyolultság
Absztrakciós: 4	Hivatkozási: 2

V. Hatékonysági esettanulmányok

Háromféle feladatot fogunk vizsgálni ebben a fejezetben. Kiindulunk egy semmilyen szempontból sem hatékony alapmegoldásból, majd ezt nagyon sok lépésen keresztül (egyszerre csak egy javítást végezve) egyre jobbá alakítjuk.

1. A végrehajtási idő csökkentése

Tudjuk, hogy minden 1-nél nagyobb természetes szám felírható prímszámok szorzataként vagy másképpen fogalmazva, különböző prímszámok hatványainak szorzataként. Olvassunk be egy természetes számot, majd adjuk meg ezt a felbontást!

Alapmegoldás: Két vektort fogunk használni, a B()-ben tároljuk a lehetséges prímosztókat (2-től N-ig), az A()-ban pedig a prímosztók hatványkitevőit (amelyik szám nem prím vagy nem osztó, ott ez az érték 0).

```
Prímfelbontás(N):
  Ciklus I=1-től N-1-ig
    B(I):=I+1; A(I):=0
  Ciklus vége
  Ciklus I=1-től N-1-ig
    Ha B(I) osztója N-nek és Prím(B(I))
       akkor Kitevőszámolás(I,N)
  Ciklus vége
  Ciklus I=1-től N-1-ig
    Ha A(I)≠0 akkor Ki: B(I),A(I)
  Ciklus vége
Eljárás vége
```

```
Prím(x):
  Prím:=IGAZ
  Ciklus J=2-től X-1-ig
    Ha J osztója X-nek akkor Prím:=HAMIS
  Ciklus vége
Eljárás vége.
```

```
Kitevőszámolás(I,N):
  Ciklus J=1-től B(I) alapú logaritmus(N)-ig
    Ha B(I)^J osztója N-nek akkor A(I):=J
  Ciklus vége
Eljárás vége.
```

Foglalkozzunk először a **Prím** eljárással!

1. lépés: A ciklus lépésszámát csökkenthetjük, ha tudjuk, hogy ha egy szám nem prím, akkor van a négyzetgyökénél nem nagyobb osztója is.

```
Prím(X):
  Prím:=IGAZ
  Ciklus J=2-től négyzetgyök(X)-ig
    Ha J osztója X-nek akkor Prím:=HAMIS
  Ciklus vége
Eljárás vége.
```

2. lépés: A ciklus lépésszámát tovább csökkenthetjük, ha figyelembe vesszük, hogy ha egy számhoz találtunk egy osztót, akkor már felesleges tovább folytatni a vizsgálatot.

```
Prím(X):
  J:=2
  Ciklus amíg J≤négyzetgyök(X) és J nem osztója X-nek
    J:=J+1
  Ciklus vége
  Prím:=(J>négyzetgyök(X))
Eljárás vége.
```

3. lépés: A ciklusmag egyszeri végrehajtási idejét csökkenthetjük, ha a ciklustól független kifejezést a cikluson kívül, egyszer számítjuk ki.

```
Prím(X):
  J:=2; GYOK:=négyzetgyök(X)
  Ciklus amíg J≤GYOK és J nem osztója X-nek
    J:=J+1
  Ciklus vége
  Prím:=(J>GYOK)
Eljárás vége.
```

Ezután nézzük a **Kitevőszámolás** eljárást!

4. lépés: A ciklus lépésszámát eggyel csökkenthetjük, hiszen J=1-re már tudjuk, hogy oszthatóságot kapunk. Ezzel a programszöveg kicsit hosszabb lesz, hiszen az A(I):=1 kezdőértékadást el kell végeznünk.

```
Kitevőszámolás(I,N):
  A(I):=1
  Ciklus J=2-től B(I) alapú logaritmus(N)-ig
    Ha B(I)^J osztója N-nek akkor A(I):=J
  Ciklus vége
Eljárás vége.
```

5. lépés: A ciklusmag egyszeri futási idejét csökkenti, ha a ciklustól független részkifejezéseket a cikluson kívül számítjuk ki. Az indexelés is időbe kerül, ezért A(I)-t és B(I)-t csak a ciklusmagon kívül használjuk!

```
Kitevőszámolás(I,N):
    AI:=1; BI:=B(I)
    Ciklus J=2-től BI alapú logaritmus(N)-ig
      Ha BI^J osztója N-nek akkor AI:=J
    Ciklus vége
    A(I):=AI
Eljárás vége.
```

6. lépés: A ciklus lépésszámát jelentősen tovább csökkenthetjük. Ha ugyanis találtunk egy kitevőt, amelyre az oszthatóság már nem áll fenn, akkor az eggyel kisebb kitevő az éppen megfelelő.

```
Kitevőszámolás(I,N):
  BI:=B(I); J:=2
  Ciklus amíg J≤BI alapú logaritmus(N) és BI^J osztója N-nek
    J:=J+1
  Ciklus vége
  A(I):=J-1
Eljárás vége.
```

7. lépés: A ciklus egyszeri végrehajtásának idejét csökkenthetjük, ha gyorsabban elvégezhető műveleteket használunk (logaritmus helyett hatványozást).

```
Kitevőszámolás(I,N):
  BI:=B(I); J:=2
  Ciklus amíg BI^J≤N és BI^J osztója N-nek
    J:=J+1
  Ciklus vége
  A(I):=J-1
Eljárás vége.
```

8.-lépés: A ciklus egyszeri végrehajtásának idejét tovább csökkenthetjük, ha egy keresési feladatot át tudunk alakítani kiválasztássá. Itt még csak nem is kell alkalmazni a kivételes eset kiküszöbölését, ugyanis előbb-utóbb biztosan találunk olyan kitevőt, amelyre az oszthatóság nem áll fenn.

```
Kitevőszámolás(I,N):
  BI:=B(I); J:=2
  Ciklus amíg BI^J osztója N-nek
    J:=J+1
  Ciklus vége
  A(I):=J-1
Eljárás vége.
```

Most foglalkozzunk a **főprogrammal**!

9. lépés: Az első és a második ciklus összevonható, hiszen azonosak a ciklushatárok, és a második ciklus adott I-re azt az A(I)-t és B(I)-t használja, amit az első ciklus erre az I-re állít elő. Ugyanezt a harmadik ciklussal is megtehetjük.

```
Prímfelbontás(N):
  Ciklus I=1-től N-1-ig
    B(I):=I+1; A(I):=0
    Ha B(I) osztója N-nek és Prím(B(I))
       akkor Kitevőszámolás(I,N)
    Ha A(I)≠0 akkor Ki: B(I),A(I)
  Ciklus vége
Eljárás vége.
```

10. lépés: A helyfoglalást lényegesen csökkenthetjük az indexes változók képletté transzformálásával. B(I)=I+1 a program minden pontján, ezért helyettesíthetjük vele. Az indexszámítás elmaradása még a futási időt is csökkenti.

```
Prímfelbontás(N):
  Ciklus I=1-től N-1-ig
    A(I):=0
    Ha I+1 osztója N-nek és Prím(I+1)
       akkor Kitevőszámolás(I,N)
    Ha A(I)≠0 akkor Ki: I+1,A(I)
  Ciklus vége
Eljárás vége.

Kitevőszámolás(I,N):
  BI:=I+1; J:=2
  Ciklus amíg BI^J osztója N-nek
    J:=J+1
  Ciklus vége
  A(I):=J-1
Eljárás vége.
```

11. lépés: A futási időt csökkenti, ha a ciklustól független kifejezéseket cikluson kívül számoljuk ki (I+1). Helyettesíthetjük I+1-et I-vel, ha módosítjuk a ciklushatárokat. Ez feleslegessé teszi a **Kitevőszámolás** BI változóját is, ugyanakkor szükségessé A(N) létezését.

```
Prímfelbontás(N):
  Ciklus I=2-től N-ig
    A(I):=0
    Ha I osztója N-nek és Prím(I) akkor Kitevőszámolás(I,N)
    Ha A(I)≠0 akkor Ki: I,A(I)
  Ciklus vége
Eljárás vége.

Kitevőszámolás(I,N):
  J:=2
  Ciklus amíg I^J osztója N-nek
    J:=J+1
  Ciklus vége
  A(I):=J-1
Eljárás vége.
```

12. lépés: A ciklusmag egyszeri lefutásának idejét csökkenti, ha egy összetett feltételes utasítást kétszeri vizsgálattal oldunk meg, természetesen úgy, hogy az egyszerűbbet vizsgáljuk előbb.

```
Prímfelbontás(N):
  Ciklus I=2-től N-ig
    A(I):=0
    Ha I osztója N-nek akkor
       Ha Prím(I) akkor Kitevőszámolás(I,N)
    Ha A(I)≠0 akkor Ki: I,A(I)
  Ciklus vége
Eljárás vége.
```

13. lépés: A(I) csak a **Kitevőszámolás**ban kaphat 0-tól különböző értéket (és kap is, ha a program eljut erre az ágra), ezért a két feltételes utasítás összevonható.

```
Prímfelbontás(N):
  Ciklus I=2-től N-ig
    A(I):=0
    Ha I osztója N-nek akkor
       Ha Prím(I) akkor Kitevőszámolás(I,N); Ki: I,A(I)
  Ciklus vége
Eljárás vége.
```

14. lépés: Egy indexes változó megszüntethető, ha a ciklusmag adott lefutásakor mindig csak egy elemére van szükség, s azt a későbbiek során sem akarjuk felhasználni. Tehát nem kell az A() vektor sem, helyette egy A változóra van szükségünk.

```
Prímfelbontás(N):
  Ciklus I=2-től N-ig
    Ha I osztója N-nek akkor
       Ha Prím(I) akkor Kitevőszámolás(I,N); Ki: I,A
  Ciklus vége
Eljárás vége.

Kitevőszámolás(I,N):
  J:=2
  Ciklus amíg I^J osztója N-nek
    J:=J+1
  Ciklus vége
  A:=J-1
Eljárás vége.
```

15. lépés: A páros számok a 2 kivételével nem prímek, ezért felesleges őket vizsgálni. Így a ciklus lépésszáma kb. felére csökkenthető.

```
Prímfelbontás(N):
  Ha 2 osztója N-nek akkor Kitevőszámolás(2,N); Ki: 2,A
  Ciklus I=3-tól N-ig 2-esével
    Ha I osztója N-nek akkor
       Ha Prím(I) akkor Kitevőszámolás(I,N); Ki: I,A
  Ciklus vége
Eljárás vége.
```

16. lépés: Ha már megtaláltuk az összes prímosztót, akkor nem kellene tovább vizsgálni a szóba jöhető osztókat. Osszuk el N értékét a megtalált prímosztókkal! Így N előbb-utóbb 1 lesz, s ez jelenti, hogy elfogytak a prímosztók. Ehhez két helyen kell javítanunk.

```
Prímfelbontás(N):
  Ha 2 osztója N-nek akkor Kitevőszámolás(2,N); Ki: 2,A
  I:=3
  Ciklus amíg N≥1
    Ha I osztója N-nek akkor
       Ha Prím(I) akkor Kitevőszámolás(I,N); Ki: I,A
    I:=I+2
  Ciklus vége
Eljárás vége.

Kitevőszámolás(I,N):
  J:=2
  Ciklus amíg I^J osztója N-nek
    J:=J+1
  Ciklus vége
  A:=J-1; N:=N/(I^A)
Eljárás vége.
```

17. lépés: Ha a **Kitevőszámolás**ban a ciklusmagban osztanánk N-et, akkor a ciklusfeltételben szereplő hatványozásra nem lenne szükség.

```
Kitevőszámolás(I,N):
  J:=2; N:=N/I
  Ciklus amíg I osztója N-nek
    J:=J+1; N:=N/I
  Ciklus vége
  A:=J-1
Eljárás vége.
```

18. lépés: A fenti eljárást tovább egyszerűsíthetjük J megszüntetésével, valamint a ciklusON kívüli N:=N/I beolvasztásával a ciklusba.

```
Kitevőszámolás(I,N):
  A:=0
  Ciklus
    A:=A+1; N:=N/I
  amíg I osztója N-nek
  Ciklus vége
Eljárás vége.
```

19. lépés: Újabb matematikai tétel ismeretében megszüntethetünk a **főprogram**ban egy feltételvizsgálatot. Egy szám legkisebb osztója ugyanis prím. Ha osztjuk vele annyiszor, ahányszor csak lehet, akkor az így kapott szám első osztója ugyancsak prím lesz. Ez azt jelenti, hogy egy osztó prímszám voltának vizsgálatára nincs szükség.

```
Prímfelbontás(N):
  Ha 2 osztója N-nek akkor Kitevőszámolás(2,N); Ki: 2,A
  I:=3
  Ciklus amíg N≥1
    Ha I osztója N-nek akkor Kitevőszámolás(I,N); Ki: I,A
    I:=I+2
  Ciklus vége
Eljárás vége.
```

20. (utolsó) lépés: A futási idő csökken, ha olyan típust használunk, amelyen gyorsabban végezhetők el a műveletek, a helyfoglalás pedig akkor csökken, ha kisebb tárigényű típust használunk. Ha tetszőleges (valós) számok helyett egészeket használunk, akkor mindkettőt elérjük. így a program végső formája, amelyben minden változó egész típusú:

```
Prímfelbontás(N):
  Ha 2 osztója N-nek akkor Kitevőszámolás(2,N); Ki: 2,A
  I:=3
  Ciklus amíg N≥1
    Ha I osztója N-nek akkor Kitevőszámolás(I,N); Ki: I,A
    I:=I+2
  Ciklus vége
Eljárás vége.

Kitevőszámolás(I,N):
  A:=0
  Ciklus
    A:=A+1; N:=N/I
  amíg I osztója N-nek
  Ciklus vége
Eljárás vége.
```

2. A helyfoglalás csökkentése

Alapfeladatunk egy könyvtár-nyilvántartás / visszakeresés. (Valójában bármely feladat, amelyben szöveges információk tárolása, illetve a velük kapcsolatos műveletek dominálnak, éppúgy példánk lehetne.)

Tároljuk a könyvekről a következő adatokat:

szerző könyvcím kiadás éve jellemzők

(ez a könyvtári információ „egysége", nevezzük **dokumentum**nak)!

Megjegyzés: a többszerzős könyveket minden szerzőjének nevével, „több példányban" is elraktározzuk. Jellemzők alatt a következőket értjük: néhány (előre rögzített számú) „szabványosított" tulajdonságjegyből a rá illőket soroljuk föl, pl.: „fizika, ismeretterjesztő" vagy „programozás, gépi kód, fizika". Ezek alapján valamely témában, valamely szinten megírt könyveket anélkül is visszakereshetünk, hogy konkrét könyvekről (szerző + cím) tudnánk.

Alapmegoldás: Számoljunk! Egy szerző nevéhez maximum 40 betű (byte) kell, egy könyvcímhez maximum 100, a jellemzőkhöz darabonként 20-20 betű. Ha csak 5 tulajdonsággal jellemzünk minden művet, a szöveges adatok tárolása akkor is 40+100+5*20=240 byte-ot emészt föl, ami néhány száz kötetes, könyvtárnak még nem igen nevezhető „könyvgyűjtemény" esetén is iszonyatos helyigényt jelent.

1. lépés: A könyvek legtöbbje esetén nem használjuk ki ezt a hatalmas helyet, sőt általában csak egy nagyon kis részére van szükségünk. Ezért célszerű olyan ábrázolást választani, ahol a **szövegek különböző hosszúak lehetnek**, s mindegyik tartalmazza a saját hosszát is.

Számoljunk újra! A szerzők nevéhez átlagban 15 byte, a mű címéhez 25, a jellemzőkhöz darabonként 10-10 byte szükséges. A szöveges adatok tárolásához így már átlagosan csak 15+25+5*10=90 byte kell.

Példa: (egy könyvtári dokumentumra)

szerző és	konkrét tartalma	hossza
szerző	Kovács Mihály	13
műcím	Számítógép a fizikatanításban	29
kiadás éve	1985	2
jellemzők	fizika	6
	számítógép	10
	gépi kód	8
	mérés/kiértékelés	17
	összesen:	85

2. lépés: Kézenfekvőnek látszik a választható tulajdonságok körét előre rögzíteni, mintegy a könyvek „elemi osztályozását" megadni. Ezen **tulajdonságjegyeket külön vektorban gyűjtve**, az egyes könyveknél a megfelelő **jellemzők helyett** csak a vektorbeli **indexét** kell felsorolni. Tehát a jellemző fogalmat egy hosszú karaktersorozat helyett egy számmal „rövidítjük". A nyereség így kb. 45 byte. (Elhanyagolhatónak tekintettük a tulajdonságjegyek vektorát, ami nagyszámú könyvtári dokumentum esetén megengedhető.)

A jellemzők szótárát felépítve ugyanez:

szerző és	konkrét tartalma	hossza
szerző	Kovács Mihály	13
műcím	Számítógép a fizikatanításban	29
kiadás éve	1985	2
jellemzők	%1 (a szótárbeli index)	1
	%2	1
	%3	1
	%4	1
	összesen:	48
A jellemzők szótára	%1: fizika	6
	%2: számítógép	10
	%3: gépi kód	8
	%4: mérés/kiértékelés	17
	összesen:	41

(Bevezettük az indirekciók szemléltetésére a %... jelölést, %1 tulajdonképpen az 1. szótári elem memóriacímét jelöli.)

3. lépés: A példát látva az az ötletünk támad, hogy próbáljuk meg **a szótárban szereplő szavakat** -ha lehet- **a cím rövidítésére** is felhasználni. Valahogy így:

műcím %2 a %1tanításban 15 byte-ra rövidült....

Gondoskodnunk kell az így tömörített szöveg egyértelműségéről, vagyis meg kell tudnunk különböztetni a szótárba való **hivatkozás**okat a betűkódoktól. (A 65 lehet az A betű kódja, de lehet a szótár 65. szava is.) Két lehetőségünk van.

1. Mivel a „normál" szövegben a betűk (kisbetűkkel kibővített) ASCII kódja kisebb, mint 128, ezért a byte első bitje betű kód esetén garantáltan 0. Ezt állítsuk be 1-re a szótárba hivatkozó index esetén! Nyilvánvaló következménye ennek, hogy legfeljebb 128 szóból állhat a szótár.

2. Az előbbi megoldás hátrányát úgy küszöbölhetjük ki, hogy az indirekciókat (a szótárba utalásokat) 1 helyett 2 byte-on kódoljuk, az 1. byte 1. bitjét az előbbi megfontolásunk szerint állítjuk be, a további 7+8=15 bit pedig a szótárbeli mutató.

szerző és	konkrét tartalma	hossza
szerző	Kovács Mihály	13
műcím	%2 a %1tanításban	17
kiadás éve	1985	2
jellemzők	%1 (a szótárbeli index)	2
	%2	2
	%3	2
	%4	2
	összesen:	40

4. lépés: **A névelők** a magyar mondatok gyakori szereplői. Az őket megelőző és követő szóközzel együtt 3-4 byte-ot jelent minden előfordulásuk. Adjunk nekik speciális azonosítót (ezzel persze csökkentjük a szótár lehetséges méretét): %A legyen az A névelő a két határoló szótaggal együtt, az AZ névelőt pedig %B jelölje (ez a két speciális jel 1-1 byte-ot foglal el)!

szerző és	konkrét tartalma	hossza
szerző	Kovács Mihály	13
műcím	%2%a%1tanításban	15
kiadás éve	1985	2
jellemzők	%1 (a szótárbeli index)	2
	%2	2
	%3	2
	%4	2
	összesen:	38

5. lépés: Kovács Mihály szerzőnek számos könyve van, így nyilvántartásunkat tovább zsugoríthatjuk azzal, hogy a **nevet is** minden dokumentumban **kódoltan vesszük föl** (miután persze a szótárban is följegyeztük). Tehát

szerző és	konkrét tartalma	hossza
szerző	%5	2
műcím	%2%a%1tanításban	15
kiadás éve	1985	2
jellemzők	%1 (a szótárbeli index)	2
	%2	2
	%3	2
	%4	2
	összesen:	27

A jellemzők szótára		
	%1: fizika	6
	%2: számítógép	10
	%3: gépi kód	8
	%4: mérés/kiértékelés	17
	%5: Kovács Mihály	13
	összesen:	54

6. lépés: A névkódolásnál „vérszemet kaptunk" és gondolatainkat így göngyölítjük tovább: Valószínűleg Kovács vezetéknevű szerzőtől több mű is található könyvtárunkban, sőt a Mihály keresztnév is több helyen felhasználható. Vegyük föl a szótárba mindkettőt önálló szóként, de ahelyett, hogy a dokumentum névmezőjében két indirekcióként (%5 %6) hivatkoznánk a névre, a **szótárba** magát a nevet összeállító **hivatkozásláncot** is beveve, egyetlen kétszeres indirekciót tartalmazó hivatkozást írunk (%7).

szerző és	konkrét tartalma	hossza
szerző	%5	2
műcím	%2%a%1tanításban	15
kiadás éve	1985	2
jellemzők	%1 (a szótárbeli index)	2
	%2	2
	%3	2
	%4	2
	összesen:	27

A jellemzők szótára		
	%1: fizika	6
	%2: számítógép	10
	%3: gépi kód	8
	%4: mérés/kiértékelés	17
	%5: Kovács	6
	%6: Mihály	6
	%7: %5 %6	5
	összesen:	60

Ezzel nagy lépést tettünk a szótár általánosítása és a tömörítés legjobb megvalósítása felé.

Foglaljuk tehát össze, amit ezen ábrázolásról tudnunk kell! A szöveges információk kétféle alapelemből építhetők föl: **betűk**ből és szótárbeli **hivatkozások**ból (indirekciókból). A szótárbeli tartalom éppúgy, mint a könyvdokumentumok a fenti módon **többszörös hivatkozású szóláncok**at is magukba foglalhatnak. Ennyit az „elméletről", nézzük meg, hogy ez az ábrázolás milyen feladatokat ró ránk, programozókra!

Konkrét feladatként oldjuk meg a következőt: egy, a fentiek szellemében felépített könyvtárból keressük ki az összes adott tulajdonságú dokumentumot! A tulajdonság legyen a szerző neve (SZERZO) és egy adott jellemző (JELLEMZO).

A megoldás algoritmusa (DOKU(N).NEV, DOKU(N).CIM, DOKU(N).DATUM, DOKU(N).TUL(5), SZOTAR(K) jelölik rendre a dokumentumok egyes mezőit: név, cím, dátum, tulajdonságok, illetve a szótárt; N kötetből áll a nyilvántartás, K a szótár szavainak, szóláncainak száma):

```
Visszakeresés:
  Be: SZERZO,JELLEMZO
  Ciklus I=1-től N-ig
    DNEV:=Szövegelőállítás(DOKU(I).NEV) [név]
    Ha DNEV=SZERZO akkor
       J:=1
       Ciklus
         DJEL:=Szövegelőállítás(DOKU(I).TUL(J))
         J:=J+1
       amíg J≤5 és DJEL≠JELLEMZO
       Ciklus vége
       Ha J≤5 akkor DCIM:=Szövegelőállítás(DOKU(I).CIM)
                    Ki: SZERZO,DCIM
       Elágazás vége
    Elágazás vége
  Ciklus vége
Program vége.

Szövegelőállítás(SZOVEG):
  EREDMENY:="" [a felgöngyölített szöveg]
  Ciklus amíg hossz(SZOVEG)>0 [amíg van kifejtetlen]
    BETU:=első(SZOVEG)
    Ha BETU normál jel akkor
                 EREDMENY:=végére(BETU,EREDMENY)
                 SZOVEG:=elsőutániak(SZOVEG)
                     különben
                 B:=index(BETU,SZOVEG(1..2))
                 SZOVEG:=elsőutániak(elsőutániak(SZOVEG))
                 SZOVEG:=elejére(SZOTAR(B),SZOVEG)
    Elágazás vége
  Ciklus vége
  Szövegelőállítás:=EREDMENY
Eljárás vége.
```

3. Az adatreprezentáció megválasztása

Ebben a mintapéldában az elemek kódolását változtatjuk meg. BASIC programnyelvi specialitásokat veszünk figyelembe.

Egy, a rádioaktív bomlást szimuláló algoritmust vizsgálunk. Legyen kezdetben N db. atom! Az A-atom időegységenként P valószínűséggel bomlik B-re. Az atomokat betűjeleikkel azonosítjuk.

Alapmegoldás: Helyezzük el az egyes atomok betűjeleit az A(N) vektorban!

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

A BASIC programra gondolva, egy szöveg típusú változó (tömbelem) tárolásához annyi byte-ra van szükségünk, ahány betűt elhelyezünk benne. Ez most 1. Ezen kívül általában 3 (pl. HT-1080Z) vagy több (6 - ABC80) byte kell a BASIC értelmezőnek.

Memóriaigény: 3*N byte.

Megjegyzés: A memóriaigénybe nem számítjuk be a segédváltozó kat (pl. ciklusváltozó), illetve az algoritmusok közös változóit (ilyen a P változó).

Egy egész típusú változónak nem szükséges ennyi hely, így a következő módosítást végezzük el:

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

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

Memóriaigény: 2*N byte.

Megállapíthatjuk, hogy ettől a módosítástól a programban elhelyezett kiírás bonyolultabb lett.

A szövegek ábrázolásából adódik a következő módosítás gondolata: ha egy változóba eggyel több betűt helyezünk el, az a felhasznált helyet egy byte-tal növeli.

2. lépés: Tároljuk az egyes atomokat az A$ változóban!

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

Memóriaigény: N+3 byte.

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

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

3. lépés: Tároljuk az egyes atomok számát A-ban és B-ben!

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

Memóriaigény: 0 byte.

Irodalomjegyzék

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

2. J.L. Bentley: A programozás gyöngyszemei.
   Műszaki Könyvkiadó, Budapest, 1988.

3. T.H. Cormen - C.E. Leiserson - R.L. Rivest: Algoritmsok.
   Műszaki Könyvkiadó, Budapest, 1997.

4. Demetrovics J. - I. Denev - R. Pavlov: A számítástudomány matematikai alapjai.
   Tankönyvkiadó, Budapest, 1985.

5. Gács P. - Lovász L.: Algoritmusok.
   Tankönyvkiadó, Budapest, 1978.

6. D.E. Knuth: A számítógépprogramozás művészete I-III.
   Műszaki Könyvkiadó, Budapest, 1986-1988.

7. Programozási feadatok I-II.
   Kossuth Kiadó, Budapest, 1997.

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

9. Számítástechnika középfokon.
   OMIKK, Budapest, 1987.

10. Varga L.: Rendszerprogramok elmélete és gyakorlata.
    Műszaki Könyvkiadó, Budapest, 1978.

10. Varga L.: Programok analízise és szintézise.
    Műszaki Könyvkiadó, Budapest, 1981.

A *µlógia* sorozat eddig megjelent tagjai

1. Horváth László - Szlávi Péter - Zsakó László: Modellezés és szimuláció
2. *Szlávi Péter - Zsakó László: Programozási forgácsok - KÖMAL feladatmegoldás*[2]
3. *Turcsányiné Szabó Márta: A Commodore-64 Terrapin LOGO leírása*
4. Szlávi Péter - Zsakó László: Módszeres programozás: Rekurzió
5. Szlávi Péter: A számítógépről népszerűsítő stílusban
6. Zsakó László: Módszeres programozás: Hatékonyság
7. Makány György: Programozási nyelvek: PROLOGIKA
8. *Szlávi Péter: A programkészítés technológiája*
9. Szlávi Péter - Zsakó László: Szimulációs modellek a populációbiológiában
10. *Pintér László: Programozási tételek rekurzív megvalósítása*
11. *Temesvári Tibor: Alkalmazói rendszerek: QUATTRO PRO 3.0*
12. Szlávi Péter - Zsakó László: Módszeres programozás: Adatfeldolgozás
13. Krammer Gergely: Turbo Grafika
14. Pap Gáborné - Szlávi Péter - Zsakó László: Módszeres programozás: Szövegfeldolgozás
15. Pintácsi Imréné - Siegler Gábor - Zsakó László: Szimulációs modellek a kémiában
16. Horváth László - Szabadhegyi Csaba - Szlávi Péter - Zsakó László: Függvényábrázolás
17. Szabadhegyi Csaba - Szlávi Péter - Zsakó László: Szimulációs modellek a fizikában
18. Szlávi Péter - Zsakó László: Módszeres programozás: Programozási bevezető
19. Szlávi Péter - Zsakó László: Módszeres programozás: Programozási tételek
20. *Hack Frigyes: Számítógéppel támogatott problémamegoldás*
21. Szlávi Péter - Temesvári Tibor - Zsakó László: Módszeres programozás: A programkészítés technológiája
22. Szlávi Péter – Temesvári Tibor – Zsakó László: Programozási nyelvek: Alapfogalmak
23. Illés Zoltán: Programozási nyelvek: C++
24. *Szlávi Péter - Temesvári Tibor - Zsakó László: Programozási nyelvek: ELAN*
25. *Ökrös László: Programozási nyelvek: Az LCN LOGO leírása*
26. *Hack Frigyes: Informatika*
27. Pap Gáborné - Szlávi Péter - Zsakó László: Módszeres programozás: Rekurzív típusok
28. *Hack Frigyes: Programozási nyelvek: BASIC*
29. *Kincses Zoltán: Gólyakalauz az Internet használatához*
30. Zsakó László: Az informatika ismeretkörei
31. Hack Frigyes: Informatikai ismeretek

---

[2] A *µlógia*-sorozat dőlt betűkkel szedett tagjai már nem kaphatók.

32. Hack Frigyes: DERIVE

33. Pozsár Erika: Internet és társadalmi viszonyok

34. Pap Gáborné - Szlávi Péter - Zsakó László: Módszeres programozás: Adattípusok

35. Köves Gabriella: TEX lépések

36. Nagy István: Szövegszerkesztés a Word 97-tel

37. Gábor Béla: Programozási nyelvek: A Delphi programozása

38. Szlávi Péter - Zsakó László: Módszeres programozás: Gráfok, gráfalgoritmusok

39. Fejezetek a számítástechnika történetéből I.

40. Fejezetek a számítástechnika történetéből II.

41. Vágvölgyi István: Prezentáció és grafika oktatása

42. Nagy Sándorné: Adatbázis-kezelés Access 97-tel

43. Hack Frigyes: 3D-grafika geometriai alapjai

44. Zsakó László (szerk.): Fejezetek a számítógépi grafikából

45. Dávid András – Pap Gáborné: Adatszerkezetek példatár

46. Zahuczkiné Bischof Annamária – Gergó Lajos: Numerikus módszerek

47. Szamper Aranka – Gergó Lajos: Ismerkedés a MAPLE V rendszerrel

Szilánkok részsorozat[3]

1. Szlávi Péter: A típusfogalom fejlődése az algoritmikus nyelvekben.

2. Temesvári Tibor: ELAN1 programozási nyelv leírása

3. Szlávi Péter: Előadás a szövegtípusokról

4. Szlávi Péter: Előadás a rekurzióról

5. Szlávi Péter: Előadás a gráftípusról

6. Szlávi Péter: Adatok, adattípusok

7. Szlávi Péter: Előadás a sorozattípusokról

8. Szlávi Péter: Előadás a file-típusokról és a táblázattípusról

9. Szlávi Péter: Előadás a táblázattípusról

10. Gálos György - Pap Gáborné: Adatszerkezetek példatár

11. Szlávi Péter: Gondolatok a típus-specifikációk kompatibilitásának vizsgálatáról

---

[3]A Mikrológia sorozat előzeteseként megjelent könyvek, melyek később beépülnek a sorozatba. (A dőlt betűkkel szedettek ilyenek, emiatt már nem kaphatók.)