Programozás kezdőknek

Dr. Bertók Csanád

Előszó

"Hazánkban informatikushiány van" – hallani mindenfelől. Ezt vezető politikusaink és különböző privát szervezetek igyekeznek egyre kreatívabb (és kétségbeesettebb) módokon orvosolni: gyakran lehet hallani olyan szoftverfejlesztő képzésekről, melyeket kimondottan kismamáknak, nyugdíjasoknak, sőt akár elítélteknek indítanak. Jelen könyvnek nem célja, hogy ezek minőségét, szükségességét, vagy úgy általában véve a hasznosságát vizsgálja, ehelyett sokkal inkább az érettségi után, felsőoktatás előtt álló fiatalságot igyekszik megcélozni. Sajnos az első mondatban említett informatikushiánynak van több negatív vonzata, melyek közül itt most egyet emelnék ki: sok szülő úgy véli, hogy gyermekének úgy tudja megalapozni a sikeres életet ha elküldi őt programozást tanulni. Teszik ezt úgy, hogy sem ők maguk, sem a fiatalok nem tudják igazán, hogy mit is takar ez a szakma, gyakran csupán a "mindig a számítógép előtt ülsz, biztos remek programozó leszel, azokból most úgyis hiány van" mentalitás áll a döntés hátterében. Az ilyen döntések az esetek túlnyomó többségében vezetnek ahhoz, hogy a diák a képzés során küszködik, egyre nehezebben veszi az akadályokat, majd egy idő után feladja és szakot, szakmát vált. Az egyetemek igen hamar felismerték ennek az ún. "lemorzsolódásnak" a jelentőségét és igyekeznek különböző felzárkóztató kurzusokkal segíteni azon diákokat akik előzetes programozói tudás nélkül kezdik meg tanulmányaikat. Ezen kurzusok főként a programozáshoz szükséges gondolkodásmód elsajátítását tűzik ki célul, általában hetente 2-4 órában, rendszerint egy féléven keresztül. A kezdeményezés remek, a rendelkezésre álló idő viszont igen szűkös, hisz egy félév alatt szinte lehetetlen utolérni azon diákokat akik középiskolás éveik túlnyomó többségében emelt szinten tanulták az informatikát és így alapszintű programozói tudásra tettek szert. Épp emiatt született meg ez a könyv.

A könyv célja, hogy teljesen az alapokról indulva, konyhanyelven elmagyarázza a programozáshoz szükséges alapismereteket minden olyan diáknak akinek "halvány lila gőze sincs" arról, hogy mi fán terem a programozás. Természetesen ennek elengedhetetlen vonzata az, hogy a könyv helyenként pontatlan terminológiát használ, az érthetőség oltárán pedig gyakran feláldozzuk a precizitást is. Így a programozásban jártasabb emberek számára jelen könyv sokkal inkább jelent humorforrást, mintsem szakirodalmat.

Bár a könyv fő fejezetei "nyelvfüggetlen" elven íródtak és csupán ún. pszeudokódot (az algoritmusok leírásának olyan eszköze ahol a szükséges utasítások bemutatás emberek által érthető módon történik) tartalmaznak az utolsó fejezetben bemutatásra kerülnek a korábban tanult technikák egy-egy konkrét programozási nyelv keretein belül.

A könyv felépítése törekszik arra, hogy a könnyű témaköröktől indulva, fokozatosan evezzen mélyebb vizekre. Ennek következtében, ha az olvasó úgy érzi, hogy a soron következő szekció számára nem tartalmaz újdonságot, úgy nyugodtan hagyja ki a neki nem releváns részeket és lapozzon olyan fejezethez ahol újdonságot talál.

Tartalomjegyzék

1. Mindenki programozó

Az előszóban egy félmondat erejéig kitértem arra, hogy hazánkban igyekeznek különböző embercsoportokat (kismamák, nyugdíjasok, stb.) megismertetni a programozással és szoftverfejlesztővé képezni. Én jelen könyv keretei között még ennél is egy lépéssel továbbmegyek és egymagam megoldom az országot (és lassan a világot) érintő informatikushiányt: **mindenki programozó**. Persze ha ilyen egyszerű volna, akkor nem lenne szükség arra, hogy ezeket a sorokat papírra vessem. Azonban a fejezet címe nem csupán egy figyelemfelkeltő kijelentés, hanem egy olyan mondat mely nem kevés igazságtartalommal rendelkezik: élete során már mindenki programozott...legfeljebb nem számítógépet. A programozás lényege, hogy egy kitűzött feladatot oly módon oldunk meg, hogy utasítások sorozatát közvetítjük egy olyan eszköznek mely ezeket képes végrehajtani. Ezek az eszközök persze a köztudatban leggyakrabban számítógépként, telefonként, vagy tabletként vannak jelen, ám a programozást nem lehet csupán erre a három tárgyra korlátozni. Gondoljunk bele az alábbi egyszerű példába:

1.1. Példa. *Ahhoz, hogy eljussunk a Rőtszakállú Herceg kastélyából az Üveghegyi Késdobálóba először 100 métert kell egyenesen haladnunk a 17-es úton, majd az első elágazásnál jobbra kell fordulni a 32-es útra és 500 métert vezetni. Végül egy éles bal kanyar után a 93-as útra letérünk és 270 méter autózás után épp az úticél előtt találjuk magunkat.*

A fenti leírás alapján bárki képes lenne eljutni a starttól a célállomásig, pusztán azáltal, hogy lépésről lépésre követi az utasításokat. Tehát az 1.1-es példa lényegében egy program, mely minket, embereket programoz be arra, hogy eljussunk a kastélyból a csehóba. Természetesen ezt a programot egy önvezető autó nem képes értelmezni, hisz rengeteg olyan dolog szerepel benne, mely csupán az emberek számára bír jelentéssel. Ám rövid gondolkodás után igen könnyen átírhatjuk a fenti sorokat úgy, hogy csak a szükséges utasításokat tartalmazza:

1. **Menj** <100 métert> : útszám 17.
2. **Fordulj** *jobbra*.
3. **Menj** <500 métert> : útszám 32.
4. **Fordulj** *balra*.
5. **Menj** <270 métert> : útszám 93.
6. **Állj.**

A fenti hat pontos leírás már sokkal inkább érthető egy gép számára, hiszen míg az embereknek készült példában igen sok szinonimát használtunk ugyanarra az utasításra (haladj, vezess, autózz), addig itt egyetlen parancsot (**menj**) adtunk ki. Hasonló a helyzet a **fordulj** szóval is. Látható, hogy ebben a gépiesített utasítássorozatban a hasonló dolgokat kifejező parancsok ugyanolyan szerkezettel rendelkeznek. Ha arra utasítom az autómat, hogy vezessen egy adott

úton valahány métert, akkor nekem azt "**Menj** <X métert> : útszám Y" módon kell megtennem, a hasonló mondatokat nagy valószínűséggel nem fogja tudni értelmezni.

Egy másik egyszerűen érthető példa egy sütemény recept alapján történő elkészítése. Mivel jelen könyv szerzője nem büszkélkedhet mestercukrászokat és nagymamákat megszégyenítő konyhai ismeretekkel, így az alábbi leírást a https://www.nosalty.hu/recept/brownie oldalról kölcsönöztem.

1.2. Példa (Brownie készítése)**.**
Hozzávalók:

1. *200g vaj*
2. *60 g tejcsokoládé*
3. *100 g étcsokoládé*
4. *2x120 g cukor*
5. *4 db tojás*
6. *130 g finomliszt*
7. *1 csipet só*

1. *A vajat és a csokoládékat vízgőz felett megolvasztjuk. Nem kell forrónak lennie, elég, ha folyékony és homogén masszát kapunk.*
2. *Hozzákeverjük az egyik adag cukrot a csokis-vajas krémhez, és félretesszük.*
3. *A tojásokat egy tálba ütjük, majd hozzáadjuk a másik adag cukrot, és habverővel elkeverjük benne. A tojásos massza felét hozzákeverjük a csokikrémhez. A másik felét elektromos habverővel 3-4 perc alatt jó habosra, könnyűre felverjük, és két részletben ezt is hozzáforgatjuk a csokis krémhez.*
4. *Beleszitáljuk a lisztet, hozzáadunk egy csipet sót, majd elkeverjük, hogy ne maradjanak benne csomók.*
5. *Sütőpapírral bélelt tepsibe tesszük, majd 180 fokos sütőbe 25 percre. Akkor jó, ha a teteje kemény, de belül még krémes, nem szabad túlsütni.*

A recept alapján még a konyhai kontárok is képesek több-kevesebb sikerrel elkészíteni ezt a süteményt. Az is könnyen látszik, hogy egy sütőrobot számára azonban a fenti sorok nem értelmezhetőek. Így ahhoz, hogy egy gép számára is megérthetővé és végrehajthatóvá váljon a receptünk, kénytelenek vagyunk az utasításokat egyszerűsíteni. Ennek sajnos velejárója, hogy a recept hossza viszont nőni fog. A leírás során az alábbi utasításokkal találkozunk:

1. **Megolvaszt**
2. **Belerak, beleszitál**
3. **Kever**
4. **Félretesz**
5. **Felver**
6. **Süt**

Ez a hat utasítás szükséges a sütemény elkészítésének automatizálásához, ám sajnos nem elegendő. Egy ember számára evidens, hogy miután kiadjuk a **Félretesz** utasítást, a következő munkafolyamatot egy új edényben kezdjük el, ám a gép számára ez korántsem egyértelmű.

Ahhoz, hogy a robotunk ezt megtegye be kell vezetnünk a "**Dolgozz új edényben**" parancsot is, mely magában foglalja a "Félretesz" és a "Vegyél elő új tálat" utasításokat. Hasonlóan, nekünk a "**Belerak** <4> tojás" azt jelenti, hogy feltörve tesszük bele a tojásokat, ám a robotnak ezt külön ki kell kötnünk. Végül a "**Belerak** sütőpapír" helyett bevezetjük a "**Kibélel**" parancsot. Így (a fenti hozzávalók betáplálását követően) egy mindenféle érzékelővel ellátott robottal az alábbiak szerint készíttethetjük el a süteményünket:

1. **Dolgozz új edényben** : edényszám 1, típus tál.
2. **Belerak** vaj : mennyiség 1 adag.
3. **Belerak** tejcsokoládé : mennyiség 1 adag.
4. **Belerak** étcsokoládé : mennyiség 1 adag.
5. **Megolvaszt** : módszer vízgőz, állapot homogén.
6. **Belerak** cukor : mennyiség 0.5 adag.
7. **Kever** : módszer kézi, időtartam 3 perc.
8. **Dolgozz új edényben** : edényszám 2, típus tál.
9. **Feltör** tojás : mennyiség 4.
10. **Belerak** tojás : mennyiség 4.
11. **Belerak** cukor : mennyiség 0.5 adag.
12. **Kever** : módszer habverő, időtartam 3 perc.
13. **Dolgozz új edényben** : edényszám 1, típus tál.
14. **Belerak** edény 2 : mennyiség 0.5 adag.
15. **Dolgozz új edényben** : edényszám 2, típus tál.
16. **Kever** : módszer habverő, időtartam 3 perc.
17. **Dolgozz új edényben** : edényszám 1, típus tál.
18. **Belerak** edény 2 : mennyiség 0.25 adag.
19. **Kever** : módszer kézi, időtartam 1 perc.
20. **Belerak** edény 2 : mennyiség 0.25 adag.
21. **Kever** : módszer kézi, időtartam 1 perc.

22. **Belerak** liszt : mennyiség 1 adag.

23. **Belerak** só : mennyiség 1 adag.

24. **Kever** : módszer kézi, időtartam 3 perc.

25. **Dolgozz új edényben** : edényszám 3, típus tepsi.

26. **Kibélel** : mivel sütőpapír.

27. **Belerak** edény 1 : mennyiség 1 adag.

28. **Süt** : hőfok 180, időtartam : 25 perc.

A fenti leírás látszólag tartalmazza a lépéseket melyeket a robotunknak meg kell tennie a sütemény elkészítéséhez. Ám valójában még ez a 28 pont sem elegendő, hisz a robot nem tudja hol vannak az új edények, hogy kapcsolja be a sütőt, mit tegyen a koszos eszközökkel, és még sorolhatnánk. Látható, hogy ezek alapján egy programozónak nem elegendő az emberek számára készült lépéseket egyszerűsíteni, kénytelen kitérnie a legapróbb részletekre is, ügyelve arra, hogy az általa megtervezett folyamat hibamentesen fusson le. A fenti leírás működik ha végtelen sok edényünk, tepsink, alapanyagunk van, ám hibát eredményez ha valamilyen szükséges hozzávalóból hiányt szenvedünk. A hibák kezelését a programozási nyelvek más-más módon oldják meg, az alapelvekre egy későbbi fejezetben térünk ki.

A fejezetben szereplő két példa segít annak megértésében, hogy miként gondoljunk a számítógépeinkre: célszerű úgy tekinteni rájuk, mint egy elképesztően okos és ügyes újszülöttre. Bármit meg tud csinálni amire utasítjuk, ám a világ dolgaihoz nem ért, hiányzik belőle a "józan paraszti ész". Ha azt mondjuk neki, hogy menjen át az út másik oldalára és hozzon nekünk egy doboz tejet a boltból, akkor ő anélkül sétál át az úttesten, hogy ellenőrizné, hogy biztonságos-e átkelni. Ha a bolt be van zárva, akkor megpróbál betörni, hogy teljesítse a kiosztott feladatot. Összefoglalva: **a számítógép pontosan azt csinálja amit mondunk neki, se többet, se kevesebbet.** Ez a kijelentés teljesen magától értetődő azoknak akik legalább egy-két éve foglalkoznak programozással, ám a területtel csak most ismerkedő diák számára ez a mondat az, melyet mindennél jobban az eszébe kell vésnie.

1.1. Feladatok

Írjuk át az alábbi "utasításokat" oly módon, hogy egy robot is megértse őket! Törekedjünk arra, hogy minél kevesebb parancs segítségével oldjuk ezt meg. **Tipp:** a parancsok számának növelése helyett inkább próbáljunk meg egy-egy meglévő utasítást bővíteni oly módon, hogy extra információt adunk meg neki. Például a "fordulj jobbra" és "fordulj balra" helyett használhatunk "**Irányváltás** : irány *jobb*" és "**Irányváltás** : irány *bal*" sorokat.

1. A célállomáshoz való eljutáshoz egyenesen kell haladni addig amíg el nem jutunk egy elágazáshoz. Itt balra kell fordulni, majd a rövid autózás után a következő elágazásnál jobbra. Ezt követően a körforgalom 2. kijáratánál kell távozni és fél km-t menni az úton.

2. Ahhoz, hogy megkapjuk a számolás eredményét, mentsük el külön betűvel a 6 oldal hosszát, majd adjuk őket össze. Ezt követően az eredményt osszuk el az első két oldal különbségével és szorozzuk meg a harmadik és negyedik szorzatával.

3. A sütemény elkészítéséhez addig adjunk vizet a tésztához 100 ml-es egységekben ameddig a színe sötétbarnáról aranybarnára változik. Minden egyes "víz-egység" hozzáadása után háromszor keverjük meg balra, kétszer pedig jobbra. Amint elértük a szükséges állapotot, bontsuk öt egyforma részre a tésztát és süssük ki 30 perc alatt 185 fokon.

4. Legyen X értéke 17, Y értéke pedig 25. Adjuk össze ezt a két számot, majd osszuk el 2-vel. Az eredményt mentsük el Z-ként. Cseréljük meg Z-t Y-al, majd Y-t X-el. Végül adjuk össze Y-t és X-et és mentsük el V-ként.

2. Adattípusok, nevezéktan

Bár a számítógépek világában mindent nullák és egyesek építenek fel, könnyen belátható, hogy csupán ezen két szám segítségével kimondottan nehéz gyorsan és eredményesen programokat fejleszteni. Így mind a hatékony programozáshoz, mind a könyv további fejezeteinek megértéséhez elengedhetetlen, hogy megismerkedjünk néhány alapvető adattípussal és az egyik legegyszerűbb, kezdők számára is könnyen érthető adatstruktúrával. Ezen típusok minden programozási nyelvben jelen vannak, ám a kezelésük nyelvtől függően különböző lehet. Erre még a későbbiekben visszatérünk, ám először álljon itt a típusok ismertetése a hivatalos, angol elnevezésükkel (és ahol elterjedt, rövidítésükkel) együtt:

- **Igaz/hamis érték** (*boolean*): talán a legegyszerűbb adattípus. Az ilyen típusú változók csupán az igaz, vagy a hamis értéket vehetik fel. Az igaz értéket szokás 1-el, míg a hamisat 0-val is jelölni, de a nyelvek jelentős részében a *true*, illetve *false* elnevezés a használatos.

- **Egész számok** (*integer, int*): ezen adattípus neve magáért beszél, így ahelyett, hogy részletesebb leírást adnánk róla, csupán egy hasznos érdekességre hívjuk fel a figyelmet. A matematikával ellentétben programozás során az egész számok természetesen nem $-\infty$-től $+\infty$-ig terjedhetnek, hanem két korlát közé vannak beszorítva. Ezen korlátok jelentős részével a programozók túlnyomó többségének manapság már nem kell foglalkoznia (régen amikor a számítógépekben még csak néhány kbyte memória állt rendelkezésre fontos volt, hogy egy szám $1, 2, 4, 8$ byte-ot foglal-e), ám ha valaki mikrokontrollerek, vagy egyéb hardvereszközök programozását tűzi ki célul annak még napjainkban is hasznos (sőt, elengedhetetlen) ha megismerkedik az *int* különböző fajtáival.

- *unsigned*: egy olyan előtag mely az *int* (és a következő pontban szereplő változatai) elé kerülhet (pl. *unsigned int, unsigned short*). A szerepe annyi, hogy az ilyen típusú változónak csupán nemnegatív értéke lehet. Alapesetben egy egész típusú változó $-$korlát-tól $+$korlát-ig terjedhet (egész pontosan $+$korlát-1-ig, hisz a számítástechnikában 2-hatványokkal dolgozunk és ha pl. -8-tól $+8$-ig terjedhetne egy szám úgy a 0 miatt ez már 17 érték lenne ami nem 2-hatvány. Ezért ilyen esetben -8-tól $+7$-ig vehet fel értékeket a szám, ami a 0-val együtt épp 16 darab különböző lehetőség), ám ha mi tudjuk, hogy negatív értékekre sosincs szükségünk (pl. mert darabszámra vagyunk kíváncsiak ami sosem mehet 0 alá) úgy az *unsigned* előtag használatával a felső korlátot kitolhatjuk nagyjából a kétszeresére. Így ha a zárójeles megjegyzésben látott -8-tól $+7$-ig tartó korláttal dolgozunk, úgy az *unsigned* jelző felhasználásával ez elcsúsztatható 0-tól $+15$-ig anélkül, hogy a memóriában több helyet foglalnánk.
- *byte (char), short, long*: szintén egész számok, ám az alsó-, illetve felső korlátjuk eltér a hagyományos *int*-étől. A *byte* (vagy több nyelvben *char*) változó -128-tól $+127$-ig terjed (unsigned esetén ez eltolódik 0-tól 255-ig). A *short* -2^{15}-tól $+2^{15}-1$ terjedelmű, míg a *long* -2^{32} alsó és $+2^{32}-1$ felső korláttal rendelkezik. Emiatt szokták a *byte*-ot 8, a *short*-ot 16, a "hagyományos" *int*-et 32, míg a *long*-ot 64 bites egészeknek is nevezni.
  - Fontos megjegyzés, hogy az *int* illetve *long* mérete architektúrától is függ, ám a mai modern számítógépek esetén rendszerint ezek az értékek szerepelnek.
  - Egy másik fontos megjegyzés, hogy az egyes nyelvekben ezek máshogy szerepelhetnek. Ami C-ben *char* and Java-ban *byte*. Hasonlóképp egyes nyelvekben a *long*-ot *long int*-ként írják, míg más nyelvekben egyszerűen csak *long*-ként szerepel. Így a fenti leírásban szereplő nevek javarészt tájékoztató jellegűek, a pontos nevezéktanért tekintsük át az éppen használandó programozási nyelv dokumentációját.

- **Lebegőpontos számok** (*float, double*): ezek a programozás "törtjei". A számítógépes reprezentációjuk pontos ismertetése nem része jelen könyvnek, azonban, hogy a *float* és *double* közötti különbséget megértsük elengedhetetlen ha legalább alapszinten bemutatjuk, hogyan is tárolódik egy lebegőpontos szám. Minden ilyen érték egy előjel bittel kezdődik (0 ha pozitív, 1 ha negatív), melyet a kitevő (exponens) bitek és végül az ún. mantissza bitek követnek. A *float* esetén az exponens 8 bitből, a mantissza pedig 23 bitből áll (így az előjellel együtt megkapjuk a 32 bites méretet), míg *double* esetén ugyanezek 11 és 52 bitesek (így végül 64 bitet kapunk. Innen is ered a *double*, mint kétszeres pontosság elnevezés, mely félrevezető, hisz az ábrázolható számok terjedelme nem kétszerese a *float*-nál tapasztalhatónak). A *float* esetén az alsó és felső korlát nagyságrendileg $\pm 10^{38}$ körül van, míg ugyanez *double* esetén $\pm 10^{308}$.
  - A különböző lebegőpontos számok mérete szintén architektúrától függ, ám a mai

modern számítógépeken a fentebb leírt értékekkel "számolhatunk".

  - Néhány programozási nyelv nem tartalmaz külön *float* és *double* típusú számokat hanem a tizedestörtek automatikusan dupla precízióval számolódnak. A Python nyelvben pl. a törteket *float*-nak nevezzük (és ezt a nevet használjuk típuskonverzió – ld. később – esetén is) annak ellenére, hogy azok valójában *double* értékek (Pythonban alapból nincs egyszeres precíziójú tizedestört).

- **Szöveg** (*string, str*): ahogy a neve is sugallja az ilyen típusú változók szövegek tárolására alkalmasak. Ezek lehetnek egy-egy szótól egészen több mondatig terjedő bekezdések is, az eltárolandó szöveg méretének főként a memória szab határt, hisz egyetlen karakter tárolása 1 byte-ot vesz igénybe, sok programozási nyelv pedig korlátozza a változók maximális méretét (de szükségtelen megijedni, ha nem szeretnénk egy változóban egy egész lexikont eltárolni úgy igen kicsi az esélye, hogy problémába ütköznénk). Felhasználásukkal kapcsolatban szükséges kiemelni néhány hasznos/fontos dolgot, illetve egy érdekességet is:

  - A *string*eket dupla macskaköröm közé szokás tenni, ezzel jelölve azt, hogy az adott szó, mondat, stb. **nem** egy változó, hanem egy érték. Tekintsük példaként a következő két kódrészletet:

    ```
    alma = 13                  alma = 13
    X = "alma"                 X = alma
    KIÍR(X)                    KIÍR(X)
    ```

    Látszik, hogy az egyetlen különbség a 2. sorban található: a bal oldali kódnál az alma szó dupla macskakörömben szerepel, míg jobb oldalon nem. A bal kódrészletnél így az X változó értékéhez az alma **szót** rendeljük hozzá, ezért a kiírás során ez jelenik meg a képernyőn. Ezzel szemben a jobb oldali kódrészletnél az X változó értéke az alma **változó** értékével lesz egyenlő, azaz az alma változó helyére behelyettesítődik a 13. Ezért a kiírásnál a képernyőn 13-at fogunk látni. A kezdő programozók esetén viszonylag gyakori hiba a macskakörmök elhagyása, ám szinte minden modern fejlesztői környezet jelzi ezt.

  - Ahogy az előző pontban megtanultuk a *string*eket általában dupla macskakörömmel jelezzük. Figyelni kell azonban, hogy az egyszeres macskakörmöt a korábban már megismert *char* változóknál szokás használni. A nyelvek többségében így nem ugyanazt jelenti a *'c'* és a *"c"*. Az első meglepő módon egy **egész szám**, mely tízes számrendszerben a 99-es értékkel rendelkezik (akit pontosabban is érdekel, annak érdemes rákeresnie az ASCII táblára, mely egy olyan táblázat ami a különböző karakterek egész számbeli értékeit tartalmazza), míg a második egy két karakter hosszúságú szöveg. A két karakter a *'c'* betű és az ún. stringeket lezáró ún. null ('$\backslash 0$') karakter. Ebből látszik, hogy pl. egy 5 betűből álló szó hossza valójában $5+1$ karakter, hiszen minden szöveg végét ez a null terminator jelzi. Ennek akkor lehet

szerepe ha egy feladat során egy bekért változó hossza limitálva van. Ha a feladat szövege szerint egy név pl. maximum 30 karakter hosszú lehet, akkor ennek szövegként való tárolásához egy 31 méretű tömbre van szükség.

  - Az utolsó "macskakörmökhöz kapcsolódó" megjegyzés inkább az érdekesség kategóriába tartozik: több programozási nyelv (konkrétan kiemelendő a Python, mint egyetemeken is gyakran oktatott nyelv) nem tesz különbséget a szimpla és a dupla macskakörmök között. Így pl. Python-ban az *"alma"* és az *'alma'*, vagy a *'c'* és *"c"* ugyanúgy *string*et jelölnek (sőt, a hivatalos dokumentációban a fejlesztők a *'... '*-t javasolják a szövegek kezelésére). Ez akkor lehet zavaró ha valaki már foglalkozott olyan nyelvvel melyben külön jelölést használnak a karakterekre és a szövegekre. Jelen könyv szerzője is inkább a dupla macskakörmöt használja Pythonban (is) annak ellenére, hogy a hivatalos dokumentáció nem ezt javasolja (természetesen ez csupán egy szubjektív preferencia és nem követendő példa).
  - A *stringek* a nyelvek jelentős részében ún. "immutable" (megváltoztathatatlan) adattípusok ami azt jelenti, hogy egy létrehozott szónak közvetlenül általában nem tudjuk kicserélni egy-egy betűjét, vagy módosítani a tartalmát (tehát pl. nem írhatjuk azt, hogy az X változó 5. betűje legyen mostantól "a"). Természetesen minden nyelvben rengeteg beépített függvény áll rendelkezésünkre a különböző műveletek elvégzésére, így ez nem korlátozza számottevően a munkánkat. Emiatt ez a megjegyzés csupán emlékeztetőként/figyelmeztetőként került jelen könyv lapjai közé (ha bármikor *string*-ekkel dolgozunk jusson eszünkbe ez és nézzük meg az adott nyelv dokumentációjában, hogy az általunk elvégzendő feladatra melyik függvényt/metódust szükséges használni).

- **Tömbök**: a programozás világában a tömbök olyan változók, melyek több adat egyidejű tárolására szolgálnak, így annak ellenére, hogy hivatalosan nem lehet őket az egészekkel, lebegőpontos számokkal, szöveggel,... egy szinten említeni (a tömbök ún. adatszerkezetek és nem adattípusok), a minél korábbi megismerésük elengedhetetlen a kezdő programozók számára. A nyelvek egy részénél (pl. C, C++, Java) egy tömb csupán egyetlen típusú adatot képes tárolni (így megkülönböztetünk *int, double, float, str(ing),...* tömböket), míg más nyelvekben nyugodtan keverhetőek a tömbben lévő adatok típusai (például Python esetén szerepelhet ugyanabban a tömbben a 15, "alma" és a 9.78 is). Amennyiben egy tömb képes "kevert típusú" adatokat is tárolni úgy *listának* nevezzük (így pl. a Python-al ismerkedő kezdő programozók általában a lista szóval találkoznak mikor erről az adatszerkezetről tanulnak), ám a kezelésük ugyanúgy történik, mint a tömböknek, így mi is ezt a nevet használjuk. Tömbök kezelésére nyelvtől függően több művelet áll rendelkezésünkre, ám jelen könyvben csupán az ún. indexelést mutatjuk be.

  - Amennyiben egy tömbben lévő elemre hivatkozni szeretnénk, úgy az elem értékének felhasználása helyett annak **helyével** (*indexével*) dolgozunk. Így ha például a töm-

bünk az $[5, 13, 2, 8, -6, 0, 21]$ elemekből áll (megj.: tömbök elemeit általában szögletes zárójelek közé tesszük. Néhány nyelvben elterjedt a kapcsos zárójelek közötti felsorolás is, sőt vannak olyan nyelvek ahol a létrehozás során kapcsos zárójeleket használunk, ám az indexeléshez szögleteset) úgy az egyes elemek indexe (helye) és értéke:

index	0.	1.	2.	3.	4.	5.	6.
érték	5	13	2	8	-6	0	21

Fontos kiemelni, hogy szinte mindegyik programozási nyelvben az **indexelés** 0**-tól kezdődik** mely kezdő programozók számára kicsit idegen és nehezen megszokható, ám kellő gyakorlással ez a számozás természetessé válik. (Megj.: ez nem csupán tömbökre igaz, pl. egy fájl kezdő sora is a 0. sor). Az alábbiakban mutatunk néhány példát a fentebb látott $[5, 13, 2, 8, -6, 0, 21]$ tömb elemeinek felhasználására:

```
tomb = [5,13,2,8,-6,0,21]
KIÍR(tomb[3]) -> 8
KIÍR(tomb[3]+tomb[0]+tomb[4]) -> 7
x = tomb[6]
y = tomb[1]*tomb[2]
KIÍR(x+y) -> ez 21+13*2 = 47
tomb[3] = 100
tomb[6] = 200
tomb[2] = 300
KIÍR(tomb) -> [5,13,300,100,-6,0,200]
```

Látható, hogy egy tömb elemeivel ugyanúgy tudunk dolgozni, mint egy-egy változóval. Az értékét felhasználhatjuk (matematikai) műveletekben, sőt az utolsó sorok alapján akár meg is változtathatjuk őket. A fentebb látott *tomb[*3*]*=100 lényegében ugyanolyan mintha egy már meglévő *x* változó értékét módosítanám 100-ra, a különbség csupán annyi, hogy itt az *x* szerepét a *tomb* nevű adatszerkezet 3. eleme veszi át. Így az a sor csak annyit mond, hogy "fogd a *tomb* változó 3. elemét és (akármi is volt az) változtasd meg 100-ra".

– Amennyiben összes eleme a fentebb megismert adattípusok valamelyike (akár keverve is ha a nyelv lehetővé teszi) úgy *1D (egy dimenziós)* tömbről beszélünk. Az egyszerű "tömb" elnevezés általában ezt jelenti, a gyakorlatban ritkán szokták hozzátenni az *1D* előtagot. Ezzel szemben ha egy tömb elemei maguk is tömbök úgy $2, 3, \ldots$ *dimenziós* tömbökről van szó. A 2 dimenziós tömböket (és "valamennyire" a 3 dimenziósakat is) szemléletesen is el lehet képzelni:

  * A 2 dimenziós tömbök olyan tömbök, melyek minden eleme egy-egy "sima" (1 dimenziós) tömb. Például a $tomb2d = [\,[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]\,]$

egy 2 dimenziós tömb. Elképzelni úgy lehet egyszerűen őket, mint egy táblázatot:

sor \ oszlop	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

Az indexelés ún. dupla-indexekkel zajlik ahol a sorindexet adjuk meg először, majd az oszlopindexet. Így ha hozzá szeretnénk férni például a 6-os elemhez úgy az $tomb2d[1][2]$-vel tehetjük meg. Hasonlóan a 10 nem más, mint $tomb2d[3][0]$. A legkönnyebben megérteni úgy lehet a dupla indexelést, hogy ha keresünk egy elemet úgy először megkeressük, hogy hanyadik "altömbben" van benne (például amikor a 6-ot keressük akkor az a $[4, 5, 6]$ altömbben van, mely az altömbök közül az 1-es indexű (0-tól indexelve). Hasonlóan a 10-es elem a $[10, 11, 12]$-es altömbben van, ami 0-tól indexelve a 3. altömb), majd megnézzük, hogy ennek az altömbnek hanyadik eleme (a 6-os elem a $[4, 5, 6]$-os altömb 2. eleme, míg a 10 a $[10, 11, 12]$-es altömb 0. eleme). Ezt a két kapott számot összerakva már meg is kaptuk a keresett indexelést (a 6-os elem az 1-es altömb 2. eleme, így $tomb2d[1][2]$-vel hivatkozunk rá, míg a 10-es elem a 3-as altömb 0. eleme, így $tomb2d[3][0]$ az "elnevezése").

* A fentiekhez hasonlóan a 3 dimenziós tömbök olyan adatszerkezetek, melyek minden eleme egy-egy 2 dimenziós tömb. Elképzelni úgy lehet őket, mintha az elemeket egy 3 dimenziós koordináta-rendszer x, y és z tengelyének egész pontjaira írnánk fel. Például ha azt mondjuk, hogy az *"alma"* a $(2, 3, 5)$-ös "helyen" van, az azt jelenti, hogy (0-tól indexelve) a 2. két dimenziós altömb 3. egy dimenziós altömbjének 5. eleme. Talán még egyszerűbb ha egy klasszikus, 10 emeletes lakóházat képzelünk el ahol emeletenként pl. 6 lakás van és minden lakásban pl. 3 szoba. Ha ebben a lakóházban keressük az embereket úgy egy adott lakás egy egyszerű egy dimenziós tömb, hiszen a keresett ember lehet a 0., 1. vagy 2. szobában. Egy emelet viszont már egy két dimenziós tömb, 3 elemű lakásokat tartalmaz. Míg végül az egész ház egy 3 dimenziós tömb, hisz 10 darab két dimenziós "emelet" van benne. Ennek alapján ha azt mondom, hogy Péter a $haz[3][5][2]$-ben található az azt jelenti, hogy Péter pillanatnyilag a 3. emeleten van az 5. lakásban (3. em., 5. ajtó) és ezen belül a 2. szobában (pl. nappaliban). Látható, hogy az indexelés ugyanúgy (és ugyanolyan elven) történik, mint az egy- és két dimenziós tömböknél, egyszerűen az indexeléshez használt szögletes zárójelek száma növekszik 3-ra.

A fentebb felsorolt adattípusok és adatszerkezet elegendőek ahhoz, hogy viszonylag komo-

lyabb feladatokat is megoldjunk a segítségükkel, így az utolsó rövid pont mellyel foglalkoznunk kell mielőtt rátérnénk a gyakoribb vezérlőszerkezetek bemutatására az nem más, mint az ún. "nevezéktan".

A nevezéktan, vagy elnevezési konvenciók (angolul *naming conventions*) a programozásnak az a "területe", mely a változók, függvények, osztályok, stb. neveinek írásmódjával foglalkozik. Elöljáróban leszögezhetjük, hogy az itt leírtak csupán a tipp/javaslat kategóriába tartoznak és nem kőbe vésett szabályok. Sőt, gyakran ezek a konvenciók cégenként is változnak így előfordulhat az, hogy egy munkahelyen teljesen a programozóra bízzák a változók, stb. nevének kitalálását, máshol viszont szigorú szabályokhoz kötik ezeket. Annyi azonban bizonyos, hogy a saját munkánkat is jelentősen megkönnyítjük azzal, ha már a programozás első percétől kezdve kialakítunk (elfogadunk) egy számunkra szimpatikus módszert és próbáljuk magunkat ahhoz tartani (többször előfordul egyetemistáknál, hogy amikor a programot elkészítik még tudják, hogy mit jelent az *x,y,z,x1,x2,...* változó, ám amint a programjukat hetekkel később meg kell védeniük már elfelejtették, hogy mi mit jelöl. Ez könnyen megelőzhető beszédesebb nevek használatával). Az alábbiakban bemutatunk néhány gyakran használt konvenciót, illetve néhány hasznos tippet is.

- Ha valaminek nevet adunk akkor célszerű olyan nevet választani mely beszédes és jól tükrözi az adott változó, függvény, osztály, stb. funkcióját. Amennyiben a megfelelő változók korrekt módon definiálva vannak úgy senki sem akadályoz meg bennünket abban, hogy olyat írjunk, hogy

  $$x = a \cdot \frac{y}{w}$$

  ám ebből senki sem fog rájönni, hogy ez a képlet (és a végén az x változó) valójában arra szolgál, hogy kiszámítsuk, hogy az autónk 1 km megtételekor hány forintnyi benzint használ el. Ha azonban ehelyett azt írjuk, hogy

  ```
  ar_per_km = benzinar_per_liter * (ut_kmben/hasznalt_benzin)
  ```

  úgy mindenki egyértelműen tudni fogja, hogy melyik változó mit jelöl. Sőt még tovább is lehetne cifrázni a változók nevét, hogy explicit módon megmondjuk, hogy a *benzinar_per_liter* és az *ar_per_km* forintban értendő, a *hasznalt_benzin* pedig literben, ám ezzel elérkeztünk a következő ponthoz.

- Ne essünk át a ló túloldalára! Kezdő programozók gyakran billegnek az "egybetűs" (*x,y,z*) és az extrém precíz (*a_csabai_kolbasz_kilonkenti_ara_forintra_kerekitve*) elnevezések között, ám az igazán megfelelő nevek valahol a kettő között vannak. Egy rövid és "közepesen beszédes" változó is rengeteget segít abban, hogy megértsük mit is látunk. És ha netán úgy érezzük, hogy a név nem elég pontos úgy bármikor lehetőségünk van egy-egy komment beszúrásával precízebben leírni, hogy mire gondoltunk:

  ```
  # az AR_KM valtozo megadja, hogy az autonk 1 km megtetelekor
  ```

```
# hany forintnyi benzint hasznal fel
# a BENZINAR a benzin literenkenti ara forintban
# az UTHOSSZ a megtett ut hossza kilometerben
# a MENNYIBENZIN a teljes felhasznalt benzinmennyiseg literben

ar_km = benzinar * (uthossz/mennyibenzin)
```

Látszik, hogy a változók nevei bár beszédesek és segítenek a megértésben, mégsem túl precízek. Az elején szereplő kommentek viszont egyértelművé teszik, hogy mire is gondoltunk. A kommenteket lehet még informatívabbá tenni például oly módon, hogy minden sor végén megadjuk az adott változó típusát (függvények esetén a paraméterek, visszatérési érték típusát), stb. Természetesen ezek csupán tippek ahhoz, hogy egy-egy nagyobb projektnél ne vesszünk el a jelölések tengerében, így nem feltétlenül kell követendő példaként gondolni rájuk.

- A változók, függvények,... nevének kitalálása az első lépés a könnyen olvasható kód irányába. A következő feladat mellyel meg kell birkóznunk nem más, mint, hogy hogyan írjuk le a több szóból álló változókat, függvényeket, osztályokat. Szerencsére az évek során többféle elfogadott elnevezési konvenció jelent meg a programozók között mely kimondottan ezt a problémát igyekszik kezeli. Néhány a legelterjedtebbek közül:
  - *camelCase*: az első szó kivételével minden szót nagybetűvel kezdünk. Pl. *benzinarPerLiter*. A nevét a tevék két púpjáról (mint kezdő nagybetűk) kapta (annak ellenére, hogy az első szó kezdőbetűje kicsi). Szokták *lower camelCase*-nek is nevezni, hogy jelezzék, hogy az első betű kicsi. Többek között például a Java nyelv használja ezt a technikát (érdekesség: az osztályok nevei PascalCase-ben íródnak).
  - *PascalCase:* a fentiekhez hasonló azzal a különbséggel, hogy itt minden szó kezdőbetűje nagybetű. Szokás *upper CamelCase*-nek is nevezni. Az egyik legnagyobb programozási nyelv ami ezt használja a C# (érdekesség: ha az első szó max. két betűs úgy a C# (lower) camelCase-t használ. Például: isPrime).
  - *snake_case:* minden szó csak kisbetűket tartalmaz, a szavak között pedig "alsóvonások" (angolul underscore) szerepelnek. Nevéhez hűen a Python programozási nyelv használja előszeretettel.
  - *dash-case*: minden szó közé kötőjelet (angolul dash) teszünk. Viszonylag ritka, csupán néhány speciális programozási nyelv (pl. COBOL) engedi meg, hisz könnyen összekeverhető az egyszerű kivonással.

A fentieknek természetesen léteznek variációi (pl. a *snake_case* keverhető *PascalCase*-el, hogy *Benzinar_Per_Liter* jellegű változókat kapjunk), így bátran ki lehet jelenteni, hogy

ahány nyelv annyiféle konvenció. Általában a nyelvek keverve használják a fenti technikákat (különbözőképp írják a változókat, függvényeket, osztályneveket, konstansokat, stb.), sőt gyakran a cégek is "felülírják" az egy-egy nyelvben bejáratott és elterjedt nevezéktant a saját maguk által használttal. Így kezdőként csupán annyit tűzzünk ki célul, hogy válasszunk ki egyet a fentiek közül (a legelterjedtebb a *camelCase* és a *PascalCase*) és próbáljuk azt használni.

- Csupán érdekességként említendő, hogy az elnevezési konvencióknak magyar vonatkozásuk is van. Simonyi Károly (manapság inkább Charles Simonyi) a MS Office megalkotója javasolta az azóta is széleskörűen használt "magyar jelölést" melynek lényege, hogy a változók első néhány betűje megadja azok típusát (ún. systems hungarian) vagy azok "funkcióját" (apps hungarian). Típus esetén például ha egy változó egy egész szám akkor a neve *"i"* betűvel kezdődik (iDiakokSzama), ha long integer akkor *"l"*-el (lCsillagokSzama), ha szöveg akkor *"str"*-el (strFelhasznaloID), míg a tömbök az *"arr"* kezdőszót tartalmazzák (így pl. az arriSzamok egy olyan tömböt tartalmaz melynek elemei egészek). A funkció terén kicsit lazábbak a megkötések, ott a cél az volt, hogy beszédesebbé tegyék a változók neveit úgy, hogy közben egységesítik a nevezéktant. Pl. colNumber, colIndex, colSum mind-mind egy táblázat oszlopairól (column) tárol információt.

Most, hogy megismerkedtünk néhány gyakran használt változótípussal (illetve egy adatszerkezettel is) és ejtettünk néhány szót az elterjedtebb elnevezési konvencióról eljött az idő, hogy rátérjünk arra, hogy milyen főbb vezérlőszerkezetek állnak rendelkezésre ahhoz, hogy elkészítsük programjainkat.

3. A programok vezérlése

Az előző fejezetekben szót ejtettünk arról, hogyan lehet emberi nyelven megfogalmazott leírásokat átfogalmazni oly módon, hogy egy gép is megértse őket. Azonban ez még csupán az első lépés ahhoz, hogy igazi programozókká váljunk. Ha visszaolvassuk az első fejezetben leírtakat, akkor feltűnhet, hogy az utasítások egyértelműek, lineárisan követik egymást. A legegyszerűbb programokhoz ez elegendő is, ám már egy kezdő programozó is igen hamar rákényszerül arra, hogy ennél valamivel bonyolultabb vezérlést építsen be a programkódjába. Amint korábban kiderült, a való élet és a számítógép programozása több hasonlóságot hordoz magában, mint azt elsőre hinnénk. A reakcióink gyakran nem előre kiszámíthatóak, hanem több különböző feltétel teljesülésétől, vagy épp hiányától függnek. A legegyszerűbb példa erre az, ha egy vasárnap délutáni bevásárlásra gondolunk: ha meg szeretnénk venni az újságban kinézett akciós terméket, ám az nincs a boltban, akkor több alternatíva közül választhatunk:

a) Kihagyjuk az árut a kosarunkból és nem helyettesítjük semmivel.

b) Szintén kihagyjuk, de hasonló termékkel helyettesítjük.

c) Elmegyünk egy ugyanolyan boltba, hátha ott még kapható.

A lehetőségek száma természetesen ennél akár sokkal magasabb is lehet, ám már ez a néhány opció is jól szemlélteti, hogy a feladataink elvégzése során elengedhetetlen bizonyos feltételek vizsgálata.

A fentiekkel analóg módon természetesen a programozás során is megjelennek a programunkat különböző módon vezérlő utasítások, melyek három fő csoportra oszlanak: **értékadó-**, **elágaztató-** és **ciklusszervező** utasítások.

3.1. Értékadó utasítások

Ezek azok az utasítások, melyekkel az előző fejezetben találkoztunk. A feladatuk csupán annyira korlátozódik, hogy a számítógép számára egy "megjegyzendő információt" közvetítsenek. A későbbiekben nagyon fontos szerepet fog játszani a következő kijelentés (melyre természetesen a megfelelő fejezetben még visszatérünk): **a számítógép csak azt az információt jegyzi meg amire külön felszólítjuk!** Kissé leegyszerűsítve ez annyit tesz, hogy ha egy információt (legyen az egy szám, szó, egy matematikai művelet eredménye, stb.) szeretnénk a későbbiekben is felhasználni, úgy mindig tároljuk el egy értékadó utasítás segítségével. A szerkezetük nagyon egyszerű, mindössze három részből állnak:

- A "tárolóként szolgáló" változó neve.
- Egy egyenlőségjel.
- Az eltárolandó kifejezés.

Az első pontban szereplő név tetszőleges lehet, ám célszerű betartani az előző fejezetben ismertetett ajánlásokat. Az alfejezetet lezárandó, álljon itt egy egyszerű példa, mely jól szemlélteti az utasítás használatát.

3.1. Példa. *Legyen X értéke 20, Y értéke 40. Adjuk össze ezt a két számot, az eredményt tároljuk Z-vel. Végül írjuk felül X értékét annak négyzetével, Y-t pedig szorozzuk meg mind X-szel, mind Z-vel.*

```
X=20
Y=40
Z=X+Y
X=X*X
Y=Y*X*Z
```

1. megjegyzés a példához: nagyon sok kezdő számára szokatlan lehet az utolsó két sor, hisz a középiskolai matematikaórán tanultak értelmében az egyenlőségjel bal- és jobb oldalán lehetőségünk nyílna egyszerűsítésre (azaz pl. az X=X∗X esetén ha X értéke nem 0, úgy a vele

való osztást követően 1=X adódna), ám ne felejtsük el, hogy jelen esetben az egyenlőség nem egy matematikai operátor, hanem egy "értékadó" jel. Így az X=X∗X-et az alábbiak szerint kell érteni:

1. Az X változó egy új értéket fog kapni.
2. Ez az új érték **egyenlő lesz** azzal, hogy…
3. Az X változó régi értékét megszorozzuk az X változó régi értékével (X∗X).

Ezekre alapozva, ha kezdőként gondunk adódik ezen utasítások megértésével, úgy a hagyományos "balról jobbra" történő olvasás helyett próbáljuk meg először az egyenlőség jobb oldalán szereplő műveleteket elvégezni, majd a kapott eredményt tároljuk el a bal oldalon szereplő változóban. A fenti három pont így az alábbiak szerint is értelmezhető:

1. Szorozzuk meg az X változót önmagával (X∗X).
2. A kapott eredményt tároljuk el X néven.

2. megjegyzés a példához: nagyon gyakori hiba (több egyetemi hallgatónál visszaköszön), hogy egy változó értékének megváltoztatása során csupán az egyenlőségjel jobb oldalát írják le. Így például a "Legyen K értéke 20, majd növeljük K-t a kétszeresére" utasítást **hibásan** az alábbiak szerint próbálják megoldani:

```
K=20
K*2
```

Bár elsőre logikusnak tűnhet ez a megoldás, ám ismételten kiemelem a fentebb már hangoztatott állítást, miszerint: **a számítógép csak azt az információt jegyzi meg amire külön felszólítjuk!** Mivel a második sorban nem jeleztük számára, hogy tárolja is el a számítás eredményét, így az utasítást csupán elvégzi, de a végeredményt nem jegyzi meg. Ahogy fentebb már szó volt róla, ennek a mondatnak a későbbiekben is fontos szerepe lesz.

3.2. Elágaztató utasítások

Ebben az alfejezetben az elágaztató utasítástípusról ejtünk néhány szót. Ezen vezérlési struktúra megjelenik a mindennapi életünkben is, így valamilyen formában szinte mindenki találkozott már vele. Gyakori megnevezés még a *feltételes elágazás*, vagy csak egyszerűen *if–else* utasítás is. A működés szemléltetésre egy igen egyszerű példa az, ha egy dolgozat sikerességét szeretnénk megállapítani: **ha az elért pontszám meghaladja az összesen megszerezhető pontok** 60%**-át, úgy a dolgozat sikeres. Amennyiben az elért százalék** 60**, vagy annál kevesebb, úgy sajnos megbuktunk.** Ehhez hasonló példák százait lehet találni az életünkben, ám sokkal hasznosabb az, ha ezek helyett inkább megvizsgáljuk az utasítás általános szerkezetét:

```
HA (feltétel teljesül) AKKOR:
    X cselekedetet végrehajt.
EGYÉBKÉNT:
    Y cselekedetet végrehajt.
```

A fenti sorok lefordíthatóak könnyen emberi nyelvre is: *amennyiben a megadott állítás igaz, úgy az X "dolgot" végzem el. Ha a megadott állításom hamis, úgy viszont az Y-t "csinálom meg".* A fejezet legfontosabb mondata így: **a feltételnek egy IGAZ, vagy HAMIS eredményt kell visszaadnia.** Ez alapján az utasítás általános szerkezete az alábbiak szerint is felírható:

```
HA igaz AKKOR:
    X cselekedetet végrehajt.
EGYÉBKÉNT (ha hamis):
    Y cselekedetet végrehajt.
```

3.2. Példa. *Egy diáktársunknak az alábbi módon tudjuk elmondani, hogy hogyan kell kiszámolni egy x szám abszolútértékét:*

```
HA (x negatív) AKKOR:
    Vedd le előle a mínusz jelet.
EGYÉBKÉNT:
    Hagyd úgy a számot, ahogy van.
```

3.3. Példa. *A fentebb említett "dolgozat sikerességét" vizsgáló feltétel az alábbiak szerint fogalmazható meg:*

```
HA (elért százalék nagyobb, mint 60%) AKKOR:
    Dolgozat sikeres.
EGYÉBKÉNT:
    Dolgozat sikertelen.
```

A programozás során előfordulhat olyan szituáció is, mikor nem érdekel minket az egyébként ág. A való életből kiragadott példa, mikor a diák magyarázkodni kezdene a tanárának, hogy miért nem készítette el a házi feladatot, ám az oktató kijelenti, hogy "nem érdekli, hogy mi az oka, csak az számít, hogy nem csinálta meg". Ilyen esetekben elegendő csak a HA ágat elkészíteni, a feltétel EGYÉBKÉNT ága elhagyható. A 3.2 így elkészíthető az alábbiak szerint is:

3.4. Példa. *Egy x szám abszolútértékének kiszámítása egyszerűbben:*

```
HA (x negatív) AKKOR:
    Vedd le előle a mínusz jelet.
```

Bár a példánál nincs kiírva, de valójában itt is megjelenik az EGYÉBKÉNT ág oly módon, hogy "EGYÉBKÉNT ne csinálj semmit". Tehát ha az x szám negatív, úgy "levesszük a mínusz jelet", míg ha nem negatív akkor semmit nem csinálunk vele.

A fentebbi példák során megfigyelhető, hogy kerültük az összehasonlító operátorok használatát. Ennek az egyetlen oka, hogy kényelmesebb és kezdők számára könnyebben tanulható ha összefoglalva, egyben szerepeltetjük őket. Egy elágazásban (főként) az alábbi összehasonlító operátorok jelenhetnek meg:

1. A egyenlő B-vel: $A == B$.
2. A nem egyenlő B-vel: $A\,! = B$.
3. A kisebb (nagyobb), mint B: $A < B$, vagy $A > B$.
4. A kisebb (nagyobb), vagy egyenlő, mint B: $A <= B$, vagy $A >= B$.

Az első kettőt leszámítva a fenti operátorok abszolút egyértelműek, a "józan paraszti ész" alapján működnek. Az egyenlőség dupla jele csupán arra szolgál, hogy megkülönböztesse egymástól a korábban már látott értékadó utasítást és a logikai "igaz-e, hogy egyenlő a két érték"-et. Szintén kezdő hiba az egyenlőségjelek számának összekeverése, mely jobb esetben (több programozási nyelv figyelmeztet erre) figyelmeztetést, vagy hibát eredményez, rosszabb esetben azonban hiba nélkül lefut a programunk, csak épp nem azt csinálja amit szeretnénk vele csináltatni. Nézzük meg elméleti szinten, hogy mi történik az alábbi két esetben:

1.
```
X = 10
HA (X == 15) AKKOR:
    KIÍR("Alma")
    KIÍR(X)
EGYÉBKÉNT:
    KIÍR("Körte")
    KIÍR(X)
```

2.
```
X = 10
HA (X = 15) AKKOR:
    KIÍR("Alma")
    KIÍR(X)
EGYÉBKÉNT:
    KIÍR("Körte")
    KIÍR(X)
```

Az első eset úgy működik, ahogy a fentiek alapján várnánk: első lépésben megnézi, hogy X értéke egyenlő-e 15-el. Ennek két lehetséges eredménye lehet: vagy **igaz**, vagy **hamis**. Mivel X nem 15, ezért a válasz **hamis**, így az EGYÉBKÉNT ág fog lefutni, azaz a végeredmény:

```
Körte
10
```

Ezzel szemben a második kódrészlet megértése valamivel bonyolultabb: mivel egyszeres egyenlőségjelet használtunk, ezért itt nem az egyenlőség vizsgálata zajlik, hanem egy értékadás. Tehát lényegében a program megváltoztatja X értékét 15-re. Felmerül a kérdés, hogy ennek, hogy lesz **igaz**, vagy **hamis** végeredménye? (Hiszen fentebb említésre került, hogy egy ilyen szerkezetben a program mindig ezen két érték valamelyikét keresi) Sajnos a válasz nyelvtől függően is többféle lehet. A két leggyakoribb:

- a program az értékadó művelet sikerességét vizsgálja. Amennyiben képes volt megváltoztatni X értékét 15-re, úgy a végeredmény **igaz**, ám ha valamilyen oknál fogva képtelen volt rá, akkor **hamis** lesz a válasz.
- a program az értékadó művelet eredményét vizsgálja. Amennyiben ez az eredmény nem 0, úgy **igaz**-nak tekinti, míg ha 0, úgy **hamis**-nak. (Ezzel találkozhatunk pl. a C nyelvben.)

Mivel X-et a legelején eleve számként definiáltuk, így semmi akadálya annak, hogy az értékét felülírjuk, így a megváltoztatása sikeres volt (az új érték pedig nem 0), a végeredmény **igaz**, ezért a képernyőn az alábbi két sor jelenik meg:

```
Alma
15
```

Természetesen ez egy félig-meddig elvont példa, hiszen korábban említettük, hogy több programozási nyelv figyelmeztet, vagy eleve nem is engedi az értékadó operátorok használatát elágazások feltételeiként, ám az $=$ és $==$ közötti különbséget remekül szemlélteti.

A fenti operátorok közül a "nem egyenlő" jelölése lehet még szokatlan a programozásban járatlan tanulók számára. Tekintettel arra, hogy a billentyűzeten nem szerepel $\neq$ jel, így szükség volt valamilyen alternatívára, így egyrészről mondhatnánk, hogy "tanuld meg, mert így van". Másrészt azonban a programozásban a !-nek önmagában is van jelentése: **tagadás** (negáció). Mivel az elágazásokban a feltétel **igaz**, vagy **hamis**, így egy tagadással könnyen megváltoztathatjuk azt az ellenkezőjére. A legegyszerűbb példa a már tárgyalt "nem egyenlő" ($! =$), ám egyéb esetekben is szükség lehet a ! használatára. Szemléltetésként egyetlen példát mutatunk be, ám a későbbi fejezetek során kitérünk még a használatára.

3.5. Példa. *Kázmér utálja a piros gyümölcsöket, ezért ír egy apró programot, mely eldönti egyről, hogy piros-e, avagy sem. Legyen ez a PIROS(gyümölcs), mely* ***igaz*** *értéket ad vissza, ha a beírt gyümölcs piros, egyébként pedig* ***hamisat****. A célunk, hogy olyan elágazást készítsünk, mely tudatja Kázmérral, hogy az általunk megadott gyümölcs ehető-e számára. A fentiek alapján több lehetséges megoldásunk van:*

1. *A legegyszerűbb megoldás, ha egy összehasonlító operátorral megvizsgáljuk, hogy a PIROS függvény végeredménye egyenlő-e a hamissal. Itt nincs szükségünk tagadásra.*

```
X="körte"
HA PIROS(X)==hamis AKKOR:
    KIÍR("Edd meg!")
```

Itt az alábbiak történnek:

a) A számítógép behelyettesíti X helyére a körtét.

*b) Kiértékeli a PIROS("körte") utasítást és eldönti, hogy az **hamis**.*

*c) Behelyettesíti ezt a **hamis** értéket az egyenlőség bal oldalára, így a szerkezet első sora "HA **hamis** == **hamis** AKKOR:" lesz.*

*d) A == kiértékeli, hogy a bal és jobb oldal megegyezik-e. Mivel a két oldal egyenlő, így a == eredménye **igaz**.*

*e) A számítógép behelyettesíti ezt az **igaz** eredményt a feltételbe, így az első sor "HA **igaz** AKKOR:" lesz.*

*f) Mivel a feltételünk **igaz**, így a gyümölcs ehető.*

Lássuk a fenti felsorolást lépésről lépésre:

```
X="körte"
HA PIROS(X)==hamis AKKOR:
    KIÍR("Edd meg!")
------------------------------
HA PIROS("körte")==hamis AKKOR:
    KIÍR("Edd meg!")
------------------------------
HA hamis==hamis AKKOR:
    KIÍR("Edd meg!")
------------------------------
HA igaz AKKOR:
    KIÍR("Edd meg!")
------------------------------
Eredmény: Edd meg!
```

2. *Az előző kódrészlet megoldható tagadással is:*

```
X="Körte"
HA PIROS(X)!=igaz AKKOR:
    KIÍR("Edd meg!")
```

Ekkor a fentihez hasonló dolgok történnek:

a) A számítógép behelyettesíti X helyére a körtét.

*b) Kiértékeli a PIROS("körte") utasítást és eldönti, hogy az **hamis**.*

*c) Behelyettesíti ezt a **hamis** értéket az egyenlőség bal oldalára, így a szerkezet első sora "HA **hamis** ! = **igaz** AKKOR:" lesz.*

*d) A ! = két lépést hajt végre. Először megvizsgálja, hogy a két oldal egyenlő-e, majd ennek az eredményét "megfordítja" (negálja). Most a bal és jobb oldal nem egyenlő egymással, így az egyenlőség **hamis** értékkel tér vissza. Ezt a **hamis** értéket a ! az ellenkezőjére változtatja, így a végeredmény **igaz** lesz.*

*e) A számítógép behelyettesíti ezt az **igaz** eredményt a feltételbe, így az első sor "HA **igaz** AKKOR:" lesz.*

*f) Mivel a feltételünk **igaz**, így a gyümölcs ehető.*

3. *Az előző esetekben megfigyelhetjük, hogy a végső döntést az e) pont alapján hoztuk meg. Így mivel tudjuk, hogy a PIROS(gyümölcs) eleve **igaz**, vagy **hamis** értékkel tér vissza, így megspórolhatjuk a ==, vagy ! = használatát és egyszerűsíthetjük a szerkezetet.*

```
X="Körte"
HA !PIROS(X) AKKOR:
    KIÍR("Ne edd meg!")
```

Nézzük meg itt mik történnek:

a) A számítógép behelyettesíti X helyére a körtét.

*b) Kiértékeli a PIROS("körte") utasítást és eldönti, hogy az **hamis**.*

*c) A számítógép behelyettesíti ezt a **hamis** eredményt a feltételbe, így az első sor "HA ! **hamis** AKKOR:" lesz.*

*d) A ! megfordítja a **hamis** eredményt **igaz**-ra, így az első sor "HA **igaz** AKKOR:" lesz.*

*d) Mivel a feltételünk **igaz**, így a gyümölcs ehető.*

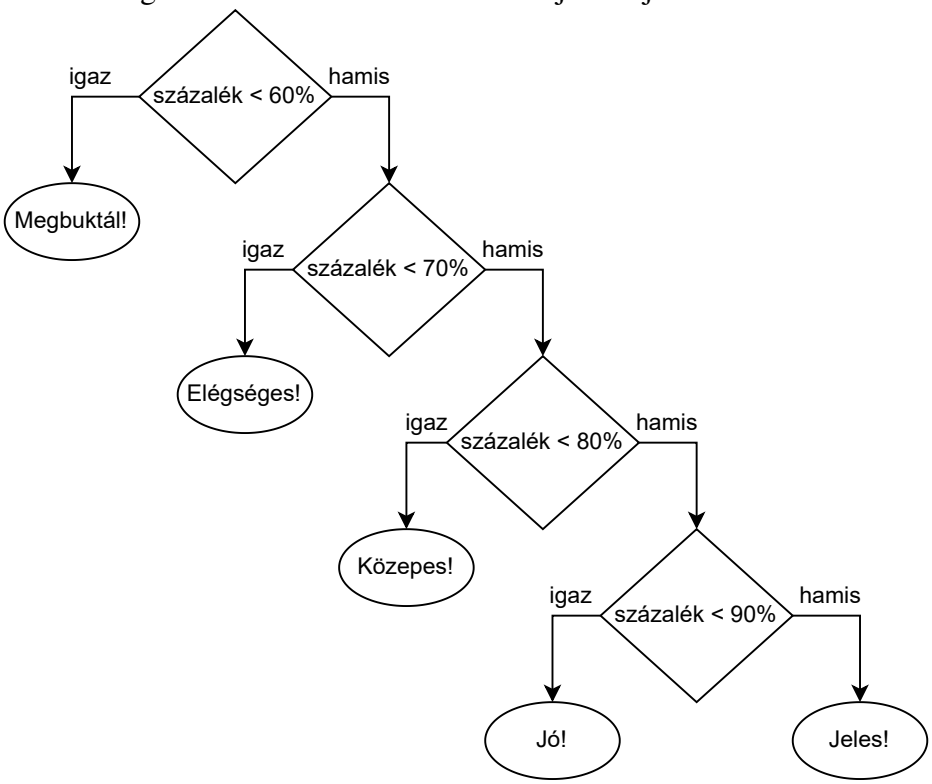
*Az első két esetben először a PIROS(gyümölcs) által visszaadott értéket hasonlítottuk össze a == és ! = operátorok segítségével az **igaz**, vagy **hamis** értékek valamelyikével, így ezen két operátor eredménye volt az mely a végén eldöntötte, hogy mi történik az elágazásban. Ezzel szemben a harmadik példánál felhasználtuk azt, hogy a PIROS(gyümölcs) már önmagában megadja a szükséges **igaz**, vagy **hamis** értéket, így elegendő csupán ezt kiértékelnünk, majd az eredményt felhasználni a döntés során.*

Fontos megjegyzés, hogy a fentebb bemutatott utasítás úgynevezett "egyszerű", vagy "bináris" (kétértékű) elágazás. Ez csupán annyit jelent, hogy egy adott feltétel esetén csupán annak teljesülését (vagy épp nem teljesülését) tudjuk vizsgálni. Így például egy dolgozat esetén csak

annyit tudunk mondani, hogy sikerült-e a teszt, ám az érdemjegyről nem tudunk további információt nyerni. Ha mégis szeretnénk egy ilyen jellegű problémát megoldani úgy az egyik lehetőség az, hogy az EGYÉBKÉNT ágba egy újabb elágazást teszünk az alábbiak szerint:

```
HA (elért százalék <  60%) AKKOR:
    KIÍR("Megbuktál!")
EGYÉBKÉNT:
    HA (elért százalék < 70%) AKKOR:
        KIÍR("Elégséges!")
    EGYÉBKÉNT:
        HA (elért százalék < 80%) AKKOR:
            KIÍR("Közepes!")
        EGYÉBKÉNT:
            HA (elért százalék < 90%) AKKOR:
                KIÍR("Jó!")
            EGYÉBKÉNT:
                KIÍR("Jeles!")
```

Ha egy diáknak 83%-ot sikerült elérnie, úgy mivel az nem kisebb, mint 60%, ezért az első esetben az EGYÉBKÉNT ág értékelődik ki. Ezen belül a program megnézi, hogy kisebb-e, mint 70%. Mivel ez sem teljesül, így ismét az EGYÉBKÉNT felé megyünk tovább. Itt a 80%-al vetjük össze és mivel még ennél sem kisebb, így még mindig az EGYÉBKÉNT irányába indulunk. Végül összevetjük 90%-al és mivel ennél már kisebb, így kiíródik a "Jó!" eredmény. A könnyebb érthetőség kedvéért a fenti sorokat le is rajzolhatjuk:



Az ábra alapján a kiértékelés már nem okoz nehézséget, hiszen egyszerűen csak "végig kell húzni az ujjunkat" a megfelelő nyilakon egészen addig míg el nem érünk egy végső állapotig.

Talán érezhető, hogy azon túl, hogy ez a megoldás igen körülményes, a leírása is túlzottan sok időt és helyet igényel. Így a programozási nyelvek (rendszerint) lehetővé teszik nekünk azt, hogy a fenti szerkezetet lényegesen egyszerűbb, kompaktabb formában írjuk le. Az ilyen utasítássorozatot "többértékű elágazásnak" nevezik és az alábbiak szerint használják:

```
HA (feltétel 1 teljesül) AKKOR:
    A cselekedetet végrehajt.
EGYÉBKÉNT HA (feltétel 2 teljesül) AKKOR:
    B cselekedetet végrehajt.
EGYÉBKÉNT HA (feltétel 3 teljesül) AKKOR:
    C cselekedetet végrehajt.
...
EGYÉBKÉNT:
    Z cselekedetet végrehajt.
```

A felépítése hasonló a korábban látott elágazáshoz, azt csupán néhány új sorral egészítjük ki. Az EGYÉBKÉNT HA sorok száma tetszőleges lehet. (Amennyiben ezekből 0 van, úgy szó szerint visszakapjuk a kétértékű elágazást.) A fejezetben eddig tárgyaltakkal összhangban az EGYÉBKÉNT ág is elhagyható, így egy elágaztató utasítás teljes általánosságban az alábbi három részből épül fel:

1. HA + feltétel - **pontosan egy darab**.
2. EGYÉBKÉNT HA + feltétel - **tetszőleges számú (akár 0 is).**
3. EGYÉBKÉNT - 0**, vagy** 1 **darab.**

A legegyszerűbb példa az ilyen jellegű elágazások felhasználására a – korábban is látott – dolgozatok érdemjegyének kiszámítása.

3.6. Példa. *Tegyük fel, hogy egy dolgozatnál az elégséges szint* 60%*-nál kezdődik, majd minden* 10% *egyel jobb jegyet eredményez. Az osztályzat megállapításához használhatjuk az alábbi elágazást:*

```
HA (elért százalék < 60%) AKKOR:
    KIÍR("Megbuktál!")
EGYÉBKÉNT HA (elért százalék < 70%) AKKOR:
    KIÍR("Elégséges az eredmény!")
EGYÉBKÉNT HA (elért százalék < 80%) AKKOR:
    KIÍR("Közepes az eredmény!")
EGYÉBKÉNT HA (elért százalék < 90%) AKKOR:
```

```
    KIÍR("Jó az eredmény!")
EGYÉBKÉNT:
    KIÍR("Jeles az eredmény!")
```

A fenti példával – és általában az elágazások használatával – kapcsolatban nagyon fontos kiemelni néhány dolgot. A legfontosabb az EGYÉBKÉNT HA ágak egymás utáni sorrendje. Ez a két szó a programozásban a következőt jelenti: "Ha egyik korábbi feltétel sem teljesül, de **EZ MOST IGEN**, akkor". Így nagyon fontos fejben tartani, hogy egy adott EGYÉBKÉNT HA ág **csakis akkor kerül kiértékelésre** ha egyetlen korábbi ág sem futott le. Így például a 76%-hoz tartozó (közepes) eredményhez az alábbiak szerint jutunk:

1. Az elért százalék legalább 60, tehát az első feltétel nem teljesül (hamis) **és**
2. Az elért százalék legalább 70, tehát a második feltétel sem teljesül (hamis) **és**
3. Az elért százalék kevesebb, mint 80, tehát a harmadik feltétel teljesül (igaz).

Ez a leírás egyben azt is jelenti, hogy amennyiben egy adott ág teljesül, úgy az alatta lévőek nem kerülnek kiértékelésre. Így hiába lenne igaz az is, hogy a 76% kevesebb, mint 90%, mivel egy korábbi feltétel már teljesül ($76\% < 80\%$), ezért a program a 90%-hoz tartozó sort már figyelmen kívül hagyja. Így ismét egy nagyon fontos mondat következik: **egy többértékű elágazásnak mindig nulla, vagy egy ága fut le**. Az alábbi esetek fordulhatnak elő:

- Ha az elágazás valamelyik feltétele igaz, úgy a **legelső** igaz feltételhez tartozó ág fog lefutni és semmi más nem.
- Ha az elágazás egyetlen feltétele sem igaz, úgy:
  - Ha van EGYÉBKÉNT sora, akkor az teljesül.
  - Ha nincs, akkor "semmi nem történik".

Bővebb magyarázatért tekintsük az alábbi esetet:

```
HA 10 < 0 AKKOR:
    V művelet.
EGYÉBKÉNT HA 10 < 5 AKKOR:
    W művelet.
EGYÉBKÉNT HA 10 < 15 AKKOR:
    X művelet.
EGYÉBKÉNT HA 10 < 20 AKKOR:
    Y művelet.
EGYÉBKÉNT:
    Z művelet.
```

Könnyen látszik, hogy az első két feltétel hamis, míg a harmadik és negyedik igaz. Így mivel most van igaz feltételünk (szám szerint kettő is), ezért azok közül az első teljesül, tehát az X művelet fog végrehajtódni, majd kilépünk az egész elágazásból annak ellenére, hogy még lenne igaz állításunk. Tehát lényegében a fenti utasításokból a számítógépünk csupán "ennyit értékel ki":

```
HA 10 < 0 AKKOR:
    V művelet.
EGYÉBKÉNT HA 10 < 5 AKKOR:
    W művelet.
EGYÉBKÉNT HA 10 < 15 AKKOR:
    X művelet.
```

Kezdő programozóknál gyakori hiba, hogy a fentiek helyett egyszerűen "sok HA" sort írnak egymás alá majd meglepődnek mikor nem a várt eredmény jelenik meg a képernyőn. Nézzünk erre egy konkrét példát, ahol bal oldalt a már ismert helyes megoldás szerepel (ld. 3.6-os feladat), míg jobb oldalon ugyanaz, azzal a különbséggel, hogy az "EGYÉBKÉNT HA" kezdetű sorokat egyszerűen "HA" kezdetűekre cseréltük:

```
HA (százalék < 60%) AKKOR:
    KIÍR("Megbuktál!")
EGYÉBKÉNT HA (százalék < 70%) AKKOR:
    KIÍR("Elégséges!")
EGYÉBKÉNT HA (százalék < 80%) AKKOR:
    KIÍR("Közepes!")
EGYÉBKÉNT HA (százalék < 90%) AKKOR:
    KIÍR("Jó!")
EGYÉBKÉNT:
    KIÍR("Jeles!")
```

```
HA (százalék < 60%) AKKOR:
    KIÍR("Megbuktál!")
HA (százalék < 70%) AKKOR:
    KIÍR("Elégséges!")
HA (százalék < 80%) AKKOR:
    KIÍR("Közepes!")
HA (százalék < 90%) AKKOR:
    KIÍR("Jó!")
EGYÉBKÉNT:
    KIÍR("Jeles!")
```

Mi történik ha a diák dolgozata 95% lett? Mivel 95 nagyobb mind 60-nál, 70-nél, 80-nál és 90-nél is, így a kódrészletek minden feltétele hamis, ezért mindkét esetben az EGYÉBKÉNT ág fut le, tehát a hallgató jelest érdemel. Azonban ha a tanuló 65%-ot ér el, akkor az eredmény már eltér a két oszlop esetén: a bal és jobb oldalon egyaránt kiértékelődik az első feltétel ($< 60\%$), mely hamisnak bizonyul, ezért eddig semmi nem történik. Teljesen hasonló módon a $< 70\%$ sor is lefut, mely igaz, ezért a képernyőn mindkét kódnál az "Elégséges!" szó jelenik meg. Mivel **a bal oldal egyetlen feltételrendszert alkot**, így az ottani kódrészlet itt meg is áll, hiszen talált egy igaz sort, ezért felesleges a maradékot megvizsgálnia. Ezzel szemben **a jobb oldalon egymástól független elágazások szerepelnek**, melyek mindegyikét külön-külön kiértékeli a program. Mind a $< 80\%$, mind a $< 90\%$-os sorok igazak, ezért a képernyőn a már ott szereplő "Elégséges!" szó alatt megjelenik a "Közepes!" és a "Jó!" is. Tehát a végeredmény az eredeti és módosított kódok esetén:

```
Elégséges!                          Elégséges!
                                    Közepes!
                                    Jó!
```

Ahogy már említettük egy többértékű elágazás mindig egy darab HA ágból, 0, vagy több EGYÉBKÉNT HA és 0, vagy 1 EGYÉBKÉNT ágból áll. Ezzel szemben egy kétértékű elágazás pontosan egy darab HA ágból és 0, vagy 1 darab EGYÉBKÉNT ágból tevődik össze. Végezetül szemléltetésképp bekeretezzük, hogy a fenti példa esetén mely egységek alkotnak egy elágazást (figyeljük meg, hogy a jobb oldalon az EGYÉBKÉNT csak az utolsó HA-val alkot egy egységet, így akár mind a 4 darab téglalapból kaphatunk választ).

```
HA (százalék < 60%) AKKOR:
    KIÍR("Megbuktál!")
EGYÉBKÉNT HA (százalék < 70%) AKKOR:
    KIÍR("Elégséges!")
EGYÉBKÉNT HA (százalék < 80%) AKKOR:
    KIÍR("Közepes!")
EGYÉBKÉNT HA (százalék < 90%) AKKOR:
    KIÍR("Jó!")
EGYÉBKÉNT:
    KIÍR("Jeles!")
```

```
HA (százalék < 60%) AKKOR:
    KIÍR("Megbuktál!")
```

```
HA (százalék < 70%) AKKOR:
    KIÍR("Elégséges!")
```

```
HA (százalék < 80%) AKKOR:
    KIÍR("Közepes!")
```

```
HA (százalék < 90%) AKKOR:
    KIÍR("Jó!")
EGYÉBKÉNT:
    KIÍR("Jeles!")
```

3.3. Ciklusszervező utasítások

Az utolsó utasítástípus mely elengedhetetlen programjaink elkészítéséhez az ún. ciklusszervező utasítás. A korábbi fejezetek során megtanultuk, hogy a számítógép egyszerre egyetlen utasítást képes végrehajtani (jelen könyv keretein belül nem fog szó esni a párhuzamosítható algoritmusokról, így fogadjuk el ezt az állítást igaznak annak ellenére, hogy nem teljesen fedi a valóságot). Így, ha a programunkban egy adott folyamatot egymás után több alkalommal meg kell ismételnünk, úgy vagy leírjuk ezeket a folyamatokat egymás alá, vagy "megmondjuk" a számítógépnek, hogy végezze el ő az ismétlést. A legegyszerűbb, való életből vett példa, ha a ruhavásárlásra gondolunk. Ha csak egyetlen ruhára van pénzünk, akkor addig próbálgatjuk ezeket, míg egy számunkra megfelelőt nem találunk. A számítógép nyelvére lefordítva ez valahogy így nézne ki (gondoljuk meg, hogy miért hagyhatjuk el a feltételekben az "== igaz" részeket):

```
HA MEGFELELŐ(ruha1) == igaz AKKOR:
```

```
    MEGVESZ(ruha1)
EGYÉBKÉNT HA MEGFELELŐ(ruha2) AKKOR:
    MEGVESZ(ruha2)
EGYÉBKÉNT HA MEGFELELŐ(ruha3) AKKOR:
    MEGVESZ(ruha3)
...
EGYÉBKÉNT HA MEGFELELŐ(ruha99999) AKKOR:
    MEGVESZ(ruha99999)
```

Ha egy hatalmas boltra gondolunk, ahol több ezer ruha áll rendelkezésünkre, úgy a fenti programkód is elképesztően hosszúra nyúlna. Így sokkal egyszerűbb valami ehhez hasonló algoritmust készíteni:

```
AMÍG MEGFELELŐ(talált ruha) != igaz ADDIG:
    Vedd elő a következő ruhát!
```

(**Megjegyzés**: a fenti feltételt az előző fejezet alapján írhattuk volna ! MEGFELELŐ(talált ruha) alakban is.)
Könnyen meggondolható, hogy ez a két sor lényegében (majdnem) ugyanazt valósítja meg, mint a fentebbiek. Felveszünk egy ruhát és amennyiben az nem megfelelő, úgy keresünk egy másikat. Ezt egészen addig ismételgetjük amíg találunk egy számunkra megfelelőt (bár a kétsoros kódunkban nincs benne szó szerint, hogy meg is vesszük a megfelelőt, de ettől az apróságtól most jótékonyan eltekintünk). Ez a példa, bár nem túl precíz, mégis jól szemlélteti, hogy mi a lényege ennek az utasítástípusnak: ugyanolyan utasítások sorozatának végrehajtása anélkül, hogy külön-külön kiírnánk az összeset. Három fő típusát különböztethetjük meg az ilyen vezérlőszerkezeteknek:

1. Elöltesztelő (*while*) ciklus.
2. Hátultesztelő (*do*... *while*) ciklus.
3. Számláló (*for*) ciklus.

Bár mindhárom arra való, hogy addig ismételje a benne szereplő utasításokat, míg egy megadott feltétel igaz, mégis számos eltérést tartalmaznak egymáshoz képest (ám a későbbiekben megmutatjuk, hogy viszonylag könnyen átalakíthatóak egymásba).

Elöltesztelő (*while*) ciklus: ezek azok az utasítások melyek a kezdők számára a legkönnyebben megérthetőek. Az általános szerkezetük:

```
AMÍG feltétel igaz ADDIG:
    X cselekedetet végrehajt.
```

A fenti sorok alapján könnyen látszik, hogy honnan kapta a nevét a ciklus: először megnézi a feltétel teljesülését (először tesztel), majd ha az teljesült akkor folytatja a belső utasításokkal. A

feltétel rész ugyanúgy működik, mint ahogy azt az előző fejezetben már láttuk: egy **igaz**, vagy **hamis** értéket vár. Igaz érték esetén "belépünk a ciklusba", míg hamis értéknél befejeződik a ciklus futása. Így lényegében az alábbi műveletsor fut le a számítógép "fejében":

1. Teljesül a feltétel? $\begin{cases} \textbf{IGEN} \rightarrow \text{2-es pont.} \\ \textbf{NEM} \rightarrow \text{3-as pont.} \end{cases}$
2. Hajtsd végre az X cselekedetet, majd ugorj az 1-es pontra.
3. Ciklus vége.

Megjegyzés: mielőtt továbbmennénk, fontos felhívni egy olyan hibára a figyelmet, mely a programozással ismerkedő diákok egy jelentős részét érinti. A fentiek alapján a ciklus akkor hajtja végre egyszer a benne szereplő utasítást ha feltételként **igaz** értéket talál. Több diák viszont hajlamos **hibásan** azt gondolni, hogy a ciklus addig ismétlődik ameddig a benne szereplő feltétel igazzá nem válik (tehát számukra a hamis feltétel a "jó" feltétel). Ezt a hibát könnyű elkerülni ha a fenti három pontot szem előtt tartva programozunk.

3.7. Példa. *Adjunk az X változónak kezdőértékként 10-et, majd ezt addig növeljük kettesével amíg X értéke 100 nem lesz. Írjuk ki az összes előforduló X értéket pontosan egyszer!*

```
X = 10
AMÍG X < 100 ADDIG:
    KIÍR(X)
    X = X + 2
KIÍR(X)
```

Figyeljük meg, hogy mivel minden lehetséges értékét ki szeretnénk írni, így a 10 kiírását akkor kell megtennünk mielőtt növelnénk X-et, tehát a KIÍR(X)-nek meg kell előznie az $X = X + 2$-t. Másrészt, a ciklus során ha X értéke 98 akkor még lefut a benne szereplő két sor, de amint azt megnöveltük 100-ra, már nem lépünk be a ciklusba (hiszen $100 \not< 100$), így amint vége az AMÍG utasításssorozatnak szükséges még egyszer kiírnunk X-et.

Egy másik lehetséges megoldás:

```
X = 10
KIÍR(X)
AMÍG X < 100 ADDIG:
    X = X + 2
    KIÍR(X)
```

Az elv teljesen ugyanaz, mint a fenti esetben, azonban most először növelünk és csak utána írunk ki. Ezért a 10-et még a ciklus előtt ki kell írnunk, ám a ciklus végén nem szükséges egy újabb kiíratást beszúrni, hisz amikor 98-al belépünk az utolsó részbe akkor először megnöveljük

100-*ra, majd utána egyből ki is írjuk.*

Egy kicsit bonyolultabb megoldáshoz jutunk ha kombináljuk az előző fejezetben tanultakat a ciklusokkal:

```
X = 10
AMÍG X < 100 ADDIG:
    KIÍR(X)
    X = X + 2
    HA X == 100 AKKOR:
        KIÍR(X)
```

Ebben a megoldásban a kezdeti értékadást leszámítva mindent az AMÍG ciklusban oldunk meg. Az első pár sor ugyanaz, mint az első megoldásban, és ez tökéletesen működik is a $10, 12, \ldots, 98$ *számok kiírásánál. Azonban, ahogy fentebb láttuk, a* 100*-at már nem írná ki. Így a cikluson belülre beillesztettünk egy elágazást ami minden egyes kör végén megnézi, hogy* X *értéke* 100 *lett-e. Ha nem, akkor semmit nem csinál, de ha igen, akkor megjeleníti a képernyőn azt.*

A helyes megoldáshoz azonban a feltétel apró módosítása is eljuttathat minket:

```
X = 10
AMÍG X < 101 ADDIG:
    KIÍR(X)
    X = X + 2
```

Látható, hogy ez a megoldás szinte teljesen megegyezik a legelsővel, a két különbséget csupán a feltételben szereplő 100 *megváltoztatása jelzi* 101*-re, illetve itt elhagytuk a végén szereplő kiírást. Mivel kettesével lépkedünk és* 10*-től indulunk ezért idővel elérünk a* 100*-hoz. Azonban, mivel teljesül, hogy* $100 < 101$*, ezért belépünk a ciklusba és kiírjuk a* 100*-at is, majd* X*-et* 102*-re növeljük. Azonban mivel* $102 \not< 101$*, ezért a következő iteráció már nem fut le, a ciklus megáll.*
Megjegyzés: *az utolsó megoldásnál figyelni kell arra, hogy míg az összes többi esetben* X *értéke a ciklus végén mindig* 100 *volt, most ez* 102*. Ez problémát okozhat ha elfelejtkezünk róla és később szeretnénk* X*-et felhasználni.*

Hátultesztelő (*do... while*) ciklus: az előző ciklushoz nagyon hasonló utasítástípus, mely a kezdő programozási feladatok során viszonylag ritkán kerül felhasználásra. Az eltérés az elöl- és hátultesztelő ciklusok között az, hogy a feltétel ellenőrzése itt nem az ismétlések előtt történik, hanem közvetlenül utánuk. Így az általános szerkezet az alábbiak szerint alakul:

```
ISMÉTELD:
    X cselekedetet végrehajt.
ADDIG AMÍG feltétel igaz
```

Ennek megfelelően a számítógép "fejében" az alábbiak zajlanak:

1. Hajtsd végre az X cselekedetet, majd ugorj a 2-es pontra.
2. Teljesül a feltétel? $\begin{cases} \textbf{IGEN} \rightarrow \text{1-es pont.} \\ \textbf{NEM} \rightarrow \text{3-as pont.} \end{cases}$
3. Ciklus vége.

Könnyen meggondolható, hogy ebben az esetben az X utasítás egyszer mindenképp végrehajtódik. Így egy igen fontos különbség az elöl- és hátultesztelő ciklusok között:

- Elöltesztelő ciklus esetén elképzelhető, hogy a ciklusban szereplő utasítás sosem fut le (ha a feltétel már az elején hamis, akkor egyáltalán nem lépünk be a ciklusba).
- Hátultesztelő ciklusnál a ciklusban szereplő utasítás legalább egyszer mindig lefut (a feltétel hamis voltára csak az első kör végén derül fény).

Ezen két pont alapján többféleképp is átalakíthatjuk a két ciklust egymásba. Példaként bemutatjuk a legegyszerűbb megoldást:

Elöltesztelő átalakítása hátultesztelővé:

```
X cselekedetet végrehajt.
AMÍG feltétel igaz ADDIG:
    X cselekedetet végrehajt.
```

Hátultesztelő átalakítása elöltesztelővé:

```
HA feltétel igaz AKKOR:
    ISMÉTELD:
        X cselekedetet végrehajt.
    ADDIG AMÍG feltétel igaz
```

A bal oldalon szereplő kódnál egyszerűen arra utasítjuk a számítógépet, hogy mielőtt eljutna a ciklushoz hajtsa végre egyszer a megadott műveletet (tehát lényegében a feltétel tesztelése a művelet első végrehajtását követően történik meg, így effektíve "hátul tesztelünk"). A jobb oldalon ezzel szemben először egy elágazásban megvizsgáljuk, hogy a feltétel igaz-e. Ha nem, úgy semmi nem történik, ám ha igen, akkor belépünk a ciklusba. Az elágazás miatt a ciklus csak akkor kezdődik el ha megbizonyosodtunk róla, hogy a feltételünk az elején teljesül, így lényegében még a ciklus első futása előtt "előre teszteltük" azt.

Számláló (*for*) ciklus: az utolsó tárgyalt ciklustípus magyarul a (le)számláló ciklus, ám sokkal elterjedtebb a for ciklus megnevezés. Az utasítás általános működését a következő, egyelőre kissé talán bonyolult példán keresztül mutatjuk be:

```
FOR i=0;i<100;i=i+2:
    X cselekedetet végrehajt.
```

A ciklus a fentiek alapján négy részre osztható:

1. *Inicializációs rész:* ennek szerepe, hogy beállítsa az ún. ciklusváltozó (a fenti példában i) kezdeti értékét. A példában ez $i = 0$.

2. *Feltétel:* a korábbi két ciklusnál is megtalálható igaz, vagy hamis eredménnyel rendelkező utasítás. A példában ez $i < 100$.

3. *Inkrementációs rész:* a ciklusváltozó módosításáért felelős szakasz. Minden egyes ismétlés után az ebben a részben leírt utasítás hajtódik végre. Jelen esetben minden "kör" végén i értéke automatikusan 2-vel növekszik.

Részletesebben a példában szereplő ciklusban az alábbiak történnek:

1. Létrejön egy (ideiglenes) i nevű változó és az értéke 0-ra állítódik. Ez a rész pontosan egyszer fut le.

2. Megvizsgáljuk, hogy teljesül-e a feltétel ($i < 100$). $\begin{cases} \textbf{IGEN} \rightarrow \text{3-as pont.} \\ \textbf{NEM} \rightarrow \text{STOP.} \end{cases}$

3. Végrehajtjuk pontosan egyszer az X cselekedetet, majd...

4. Az i változó értékéhez 2-t hozzáadunk és visszalépünk a 2)-es pontra.

A jobb megértés érdekében megvalósíthatjuk a számláló ciklust egy elöltesztelő ciklus segítségével is:

(Le)számláló ciklus:

```
FOR i=0;i<100;i=i+2:
    X cselekedetet végrehajt.
```

Elöltesztelő ciklus:

```
i = 0
AMÍG i < 100 ADDIG:
    X cselekedetet végrehajt.
    i = i + 2
```

Most, hogy ismertettük a (le)számláló ciklusok alapvető működését rátérhetünk arra, hogy bemutassuk, hogy milyen esetekben célszerű használni őket:

1. A legegyszerűbb eset az amikor előre tudjuk, hogy egy műveletet pontosan n-szer szeretnénk végrehajtani. Ekkor a ciklusváltozónak nincs szerepe a cikluson belül, hanem szimplán arra szolgál, hogy "számon tartsa", hogy épp hanyadik ismétlésnél járunk. Például ha szeretnénk pontosan 15-ször kiírni azt, hogy "Hello", akkor az alábbiak szerint járhatunk el:

   ```
   FOR i=0;i<15;i=i+1:
       KIÍR("Hello")
   ```

   Ekkor lényegében a program beállítja i értékét 0-ra, majd minden egyes "kör" elején megnézi, hogy az éppen aktuális érték kisebb-e, mint 15 és ha igen akkor kiírja pontosan egyszer, hogy "Hello". Végül 1-et hozzáad i-hez. Így i értéke az alábbiak szerint változik: $i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$, ám mivel $15 \not< 15$, ezért az utolsó esetben már nem történik kiírás.

2. A (le)számláló ciklusok "fő ereje" (és magának a névnek az eredete) azonban abban rejlik, hogy a ciklusváltozót, mint egy számlálót tudjuk használni az utasítás belsejében. Egy szemléletes példa erre az, amikor egy webáruházban minden termék kap egy cikkszámot (0-tól kezdve, egyesével növekedve). Ekkor ha arra vagyunk kíváncsiak, hogy melyik termékből hány darab áll rendelkezésre az alábbi ciklust készíthetjük el:

   ```
   FOR i=0;i<termékek száma;i=i+1:
       KIÍR(i-edik termék ára)
   ```

   Látszik, hogy itt a ciklusváltozót nem csupán a ciklus "fejlécében" használtuk fel, hanem a belsejében is. Így tudunk hivatkozni az éppen vizsgálandó termékre. Mivel i magától növekszik, nekünk külön nem kell ezzel foglalkoznunk, idővel kiírásra kerül a $28.$, $29.$, $30.$, ... termék ára is (feltéve, hogy van ennyi árucikkünk).

Az alfejezet lezárásaként álljon itt néhány hasznos/érdekes megjegyzés, melyek egy része túlmutat a teljesen kezdő programozói szinten:

- Viszonylag sok esetben lehet szükségünk arra, hogy egy ciklust még azelőtt megállítsunk mielőtt teljesülne a kilépési feltétel. Erre szinte minden programozási nyelvben ugyanaz a kulcsszó ad lehetőséget: **break**. Amint a program egy **cikluson belül** találkozik egy *break* utasítással, azonnal megszakítja a ciklus futását és az utána következő kódra ugrik. Fontos megjegyzés, hogy ez csupán ciklusokon belül működik, azokon kívül nem lehetséges használni ezt a kulcsszót.

  Egy egyszerű példa erre az ha egy "hagyományos" boltban üdítőt szeretnénk vásárolni. Ekkor a sor elejétől indulva keressük azt az italt mely megfelel az ízlésünknek (egy ciklussal a sor elejétől a végéig "megyünk"), ám ha megtaláltuk a kívánt italt úgy levesszük a polcról és ahelyett, hogy a többit is végignéznénk megyünk a listánkon felírt következő termék felé. Ilyenkor "megszakítottuk" a ciklust azelőtt mielőtt minden lehetőséget végignéztünk volna.

  Egy másik egyszerű példa az ha a számítógépünkön egy program addig várakozik míg egy másik program jelet nem küld neki (például egy internetes adatcsomagot). Ekkor mivel előre nem tudjuk, hogy ez a csomag mikor fog megérkezni így viszonylag gyakran bevett szokás az, hogy készítünk egy végtelen ciklust ami pl. másodpercenként ellenőrzi, hogy beérkezett-e a csomag. Ha nem akkor vár egy másodpercet és újra ellenőriz, ha viszont igen, úgy megállítja a ciklust, kilép belőle. Mivel ez egy végtelen ciklus így a kilépési feltétel hiányzik, ezért csak a cikluson belülről tudjuk leállítani.

  Ehhez a példához nagyon hasonló az a feladattípus mellyel szinte minden programozással barátkozó diák viszonylag hamar találkozik. A szövege valahogy így kezdődik: *"Készíts olyan programot mely soronként olvas be a billentyűzetről a "vége" szóig..."*. Ennek megoldására több lehetőség is adódik, ám egy viszonylag gyakori nagyjából így néz ki:

```
AMÍG igaz ADDIG:
    szo = BEKÉR("Adj meg egy szót!")
    HA szo == "vége" AKKOR:
        ÁLLJ (break)
    EGYÉBKÉNT:
        ...
```

Látszik, hogy mivel a feltétel *igaz* ezért ez egy végtelen ciklus, viszont a ciklusban található elágazás lehetőséget biztosít arra, hogy mégis kilépjünk belőle.

- Viszonylag ritkábban használt funkció a ciklus manuális előre léptetése. Erre a nyelvek túlnyomó többségében a **continue** kulcsszó ad lehetőséget. A *break*-el ellentétben ha a programunk egy *continue* utasítással találkozik úgy ahelyett, hogy megszakítaná a ciklust, egyszerűen a következő iterációra ugrik. Ezzel lehetőségünk nyílik arra, hogy bizonyos esetek felett átlépjünk, azokat ne vizsgáljuk meg. Természetesen a *break*-hez hasonlóan ennek is csupán egy cikluson belül van értelme, azon kívül nem lehetséges használni. Néhány példa mely szemlélteti, hogy milyen esetekben lehet hasznos a *continue* utasítás:

  Ha kíváncsiak vagyunk arra, hogy 2-től 999999999999999-ig mely számok prímek akkor egy lehetőség az, hogy egyesével megnézzük az összes pozitív egészet 2-től kezdődően és eldöntjük, hogy az éppen vizsgált szám prím-e. Azonban a prímség vizsgálata költséges, így megéri minél kevesebb ellenőrzést végrehajtani. Ennek az egyik módját az alábbi rövid kódrészlet mutatja be:

```
FOR i=2;i<=999999999999999;i=i+1:
    HA i osztható 2-vel AKKOR:
        continue
    EGYÉBKÉNT:
        PRIMSZAME(i)
```

  Látható, hogy mivel a 2 kivételével egyetlen páros prím sincs ezért azokat felesleges megvizsgálnunk, egyből átugorhatjuk őket. Természetesen ugyanezt megtehetnénk a 3-al, 5-el, stb. osztható számokkal is, sőt arra is van lehetőség, hogy i-t 1 helyett eleve 2-vel növeljük, így ha pl. 3-tól kezdünk (a 2-t pedig kézzel megnézzük az elején) úgy automatikusan átugrunk minden páros számot. Természetesen ez nem a leghatékonyabb kódrészlet a feladat végrehajtására de jól bemutatja a *continue* utasítás egy lehetséges felhasználását.

  Egy másik szemléletes példa az ha születendő gyermekünknek szeretnénk nevet választani az utónévkönyvből. Ekkor minden nevet egyesével elolvasunk és eldöntjük, hogy tetszik-e nekünk. Ám ha tudjuk, hogy a gyermekünk leány lesz úgy a fiúneveket automatikusan átugorhatjuk.

Hálózatbiztonsági szakemberek gyakran foglalkoznak egy adott hálózaton lebonyolított forgalom megfigyelésével. Néha egy-egy speciális felépítésű csomagra "vadásznak", így az összes többi csomag számukra szükségtelen. Amennyiben egy cikluson belül elemzik az áthaladó csomagokat úgy egy *continue* utasítás segítségével könnyen figyelmen kívül hagyhatják azokat melyek nem bírnak jelentőséggel és elegendő csupán azokat feldolgozniuk amikre ténylegesen szükségük van.

- A (le)számláló ciklusok leírásánál említésre került, hogy a ciklusváltozó egy ideiglenes változó. Ez egyszerűen annyit jelent, hogy néhány programozási nyelvben amint a ciklus véget ér, a változó magától törlődik, így később nem tudunk rá hivatkozni. Ez azonban tényleg nyelvfüggő viselkedés, hisz más nyelvekben a ciklusváltozóra ugyanúgy hivatkozhatunk a későbbiekben, mintha mi hoztuk volna létre őket. Így pl. a

```
1->     FOR i=0;i<10;i=i+1
2->         KIÍR(i)
3->     KIÍR(i)
```

  kódrészlet nyelvtől függően dobhat hibát, figyelmeztetést, vagy mutathat hibátlan viselkedést is (hisz amint a ciklusnak vége a 3. sorban kiíratjuk az i változó értékét. Amennyiben az törlődött úgy természetesen hibát kapunk).

- A (le)számláló ciklusnak mind az inicializációs, mind a feltétel, mind pedig az inkrementációs lépése elhagyható (akár mindhárom egyszerre). Ekkor az alábbiak történnek:

  1. Az *inicializációs* lépés elhagyásakor a program a korábbi sorokban szereplő értékekkel dolgozik. Így pl. ha egy programkód $5.$ sorában létrehozunk egy i nevű változót 13-as kezdeti értékkel úgy ha elhagyjuk az inicializációs lépést a ciklus i kezdeti értékének 13-at veszi. Ez akkor lehet hasznos ha a kezdeti érték változhat, így nem tudjuk előre, hogy honnan kell indulnunk (például ha a webshopban az első néhány termék forgalmazása megszűnik akkor lehet, hogy a $13.$ terméktől kell indítanunk a számolást hisz nem létezik $0.$, $1.$, $2.$, $\ldots$, $12.$ termék).

     ```
     szöveges fájl megnyitása
     i = első olyan sor száma ahol az "alma" szó szerepel
     FOR ;i<fájl sorainak a száma;i=i+1
         KIÍR(i-edik sor)
     ```

     A fenti kódban megnyitunk egy egyszerű szöveges állományt, valahogy megkeressük az első olyan sort amelyben szerepel az "alma" szó, majd ettől a sortól kezdve kiírjuk a fájl összes sorát. Ily módon a *txt* fájl első "pár" sorát figyelmen kívül hagyjuk. Vegyük észre, hogy annak ellenére, hogy elhagytuk az inicializációs lépést, annak az (üres) helye szerepel (az első pontosvessző előtt).

2. A *feltétel* elhagyásakor egy ún. végtelen ciklust (olyan ciklus melyből a program soha nem lép ki) kapunk. Korábban megtanultuk, hogy egy feltételnek mindig vagy igaznak, vagy hamisnak kell lennie (az üres feltételnek is), így felmerülhet a kérdés, hogy az üres feltételt miért tekintjük igaznak? A válasz meglepő: egyszerűen azért, mert annak idején az igaz értéket rendelték hozzá. Tehát semmilyen matematikai, programozói megfontolás nem áll a dolog hátterében. Természetesen e mögött a választás mögött (részben) az áll, hogy sok értelme nincs eleve le sem futó (hamis feltétellel ellátott) ciklusokat írni, így felesleges hamis-nak választani az üres feltétel, hisz "akkor úgysem történne semmi". Persze aki tanult logikát az sejtheti, hogy valami más is meghúzódik a dolgok hátterében, ám jelen könyvnek nem célja ennek magyarázatába jobban belemerülni.

   ```
   FOR i=0;;i=i+1
       KIÍR(i)
   ```

   Ebben a kódban 0-tól kezdve "végtelenig" kiírunk minden egész számot, hiszen a feltétel rész üres, azaz automatikusan mindig igaznak tekintendő. Vegyük észre, hogy annak ellenére, hogy elhagytuk a feltételt, a helyét megtartottuk (a két pontosvessző közötti üres rész).

3. Az *inkrementációs* lépés elhagyásakor egy (elöl)tesztelő ciklushoz hasonló működést kapunk. Egyszerűen a ciklus belsejében szereplő kód minden egyes lefutását követően egyből visszaugrik a feltétel vizsgálatára ahelyett, hogy végrehajtaná az inkrementációs lépést. Erre akkor lehet szükség ha a kód írásakor nem tudjuk pontosan, hogy az inkrementáció milyen módon fog történni. Például ha egy osztályteremben csak a női hallgatókat szeretnénk megszólítani akkor a névsorolvasás során kisebb-nagyobb hézagok fordulhatnak elő (a Dóra-Péter-Juli-Ági-Károly-Tamás-Eszter sorozatban az inkrementáció Dóráról Julira 2 lépésben történik, Juliról Ágira 1 lépésben, Ágiról Eszterre viszont 3 lépésben).

   ```
   FOR i=0;i<20;
       HA i osztható 5-el AKKOR:
           i = i + 1
       EGYÉBKÉNT:
           i = i + 2
   ```

   A kódban az inkrementációs lépésnek hagytunk ki a sor végén üres helyet (az utolsó pontosvessző után). Mivel az inkrementáció i pillanatnyi értékétől függ ezért egyszerűbb a cikluson belül egy elágazással befolyásolni ahelyett, hogy bonyolultabb matematikai összefüggéseket írnánk rá. Az i ciklusváltozó értéke az alábbiak szerint változik: $i = 0, 1, 3, 5, 6, 8, 10, 11, 13, 15, 16, 18, 20$.

- Az inicializációs, feltétel és inkrementációs lépések egymástól függetlenek. Ez azt jelen-

ti, hogy nem kötelező ugyanazt változót használni mindegyikben. Tekintsük például az alábbi kódrészletet:

```
n = 12
s = 0
k = 0
FOR i=0;s+k<100;n=n+1
    s = 2*i + n
    k = i - k
    i = 3*n + 1
```

Látható, hogy sem a feltétel, sem pedig az inkrementációs lépés nem tartalmazza az inicializáció során megadott i változót. Érdekességképp tekintsük meg, hogy hogyan változik i, n, s, k értéke az egyes "körök" során.

kör	$s+k$	s	k	i	n
0	0	12	0	37	13
1	12	87	37	40	14
2	124 ($\not< 100$)	$-$	$-$	$-$	$-$

- A ciklusváltozóként használt i jelölés szinte bármi lehet. Az i a matematikából ered ahol főként az indexek jelölésére használják. Ám néha kényelmesebb és átláthatóbb kódot eredményez ha "beszédesebb" nevű ciklusváltozókat használunk. Például ha a már sokszor említett webshopban lévő termékeket szeretnénk megvizsgálni úgy i helyett írhatjuk azt, hogy *cikkszam, termekszam,...*. Míg ha egy teremben lévő hallgatói létszámot kívánjuk megállapítani akkor i helyett érthetőbb a *diak* szó használata.

```
FOR cikkszam=0;cikkszam<termékek száma;cikkszam=cikkszam+1:
    KIÍR(cikkszam-adik termék ára)

FOR diak=0;diak<diákok száma;diak=diak+1:
    KIÍR(diak-adik diák neve)
```

Természetesen figyeljünk arra, hogy egyrészt ne használjunk olyan változónevet, melyet korábban már felhasználtunk valahol hisz ilyenkor az felülíródhat (nem lenne szerencsés ha például egy 3 órán keresztül számolt x nevű változó értékét lenulláznánk azzal, hogy `FOR x=0;...`), másrészt pedig lehetőség szerint tartsuk be az előző fejezetben megismert nevezéktani ajánlásokat.

Most, hogy alapszinten megismerkedtünk az elágazásokkal, illetve a ciklusokkal már majdnem készen állunk, hogy egy valamivel nehezebb témakör – a függvények – felé vegyük az

irányt. Azonban mielőtt rátérnénk ezek tárgyalására, egy rövid alfejezetben kitérünk az egymásba ágyazott ciklusokra, hisz annak ellenére, hogy ezek használata egyáltalán nem különbözik a már tanultaktól, kezdő programozók számára mégis fejtörést szokott okozni a használatuk.

3.4. Egymásba ágyazott ciklusok

Az egymásba ágyazott ciklus (angolul *nested loops*) elnevezés egyszerűen egy olyan vezérlőszerkezetre utal ahol egy ciklus belsejében (törzsében) egy másik ciklus található. Természetesen emellett szerepelhetnek elágazások, értékadó utasítások, stb. is, a kulcsmondat csupán az, hogy "cikluson belül ciklus". És bár itt meg is állhatnánk és mondhatnánk azt, hogy "innentől kezdve a kezelésüket illetően a korábbi alfejezetben leírtak a mérvadóak" mégis érdemes egy példával szemléltetni ezek működését.

- Az egyik leggyakrabban előforduló felhasználási mód az amikor egy 2 dimenziós tömb elemeit szeretnénk bejárni (vagy bármilyen olyan feladat melyben sorokkal és oszlopokkal kell dolgoznunk). Ehhez felhasználjuk a második fejezetben szereplő tömböt:

$$tomb2d = [\ [1,2,3],[4,5,6],[7,8,9],[10,11,12]\ ],$$

melyet a jobb szemléltetés végett felírhatunk táblázatos alakban is.

sor \ oszlop	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

Ha valaki arra kérne minket, hogy soroljuk fel, hogy ennek a tömbnek milyen elemei vannak akkor az emberek jelentős része vagy soronként, vagy oszloponként haladva tenné ezt. Nincs ez másként a programozásban sem. Ha azt a feladatot kapjuk, hogy írjuk ki *tomb2d* összes elemét a képernyőre akkor egy lehetőség az, hogy soronként haladunk és amint "belépünk" egy sorba kiírjuk az adott sor összes oszlopában található számot. Egy egyszerű kódrészlet lehet például a következő:

```
FOR i=0;i<4;i=i+1
    FOR j=0;j<3;j=j+1
        KIÍR(tomb2d[i][j])
```

A fenti kódrészlet teljes részletességében az alábbiakat csinálja:

– i értékét beállítja 0-ra, majd megnézi, hogy a $0 < 4$ feltétel teljesül-e. Mivel igen, így...

  * j értékét beállítja 0-ra, megnézi, hogy teljesül-e a $0 < 3$ feltétel. Mivel igen ezért kiírja *tomb2d[0][0]*-t ami nem más, mint 1. Végül j értékét 1-el növeli 0-ról 1-re.
  * Mivel $1 < 3$ még mindig teljesül ezért most kiírja *tomb2d[0][1]*-et ami 2, majd j értékét 1-ről 2-re növeli.
  * Mivel a $2 < 3$ feltétel még mindig igaz ezért most *tomb2d[0][2]*-t írja ki ami 3 és j-t 2-ről 3-ra növeli.
  * Mivel a $3 < 3$ feltétel már **hamis** ezért a j-hez tartozó ciklusnak vége, visszalépünk az i-hez tartozóba. Mivel azonban itt több sor nincs ezért i értékét 0-ról 1-re növeljük.

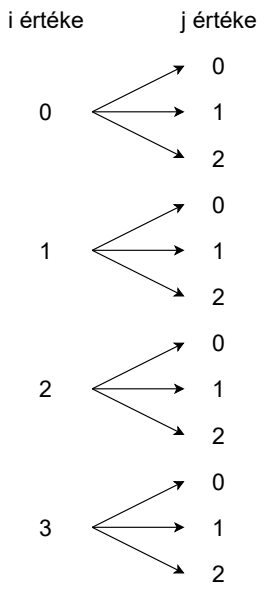
– Mivel az $1 < 4$ feltétel igaz, ezért...

  * j értékét beállítja 0-ra, megnézi, hogy teljesül-e a $0 < 3$ feltétel. Mivel igen ezért kiírja *tomb2d[1][0]*-t ami nem más, mint 4. Végül j értékét 1-el növeli 0-ról 1-re.
  * Mivel $1 < 3$ még mindig teljesül ezért most kiírja *tomb2d[1][1]*-et ami 5, majd j értékét 1-ről 2-re növeli.
  * Mivel a $2 < 3$ feltétel még mindig igaz ezért most *tomb2d[1][2]*-t írja ki ami 6 és j-t 2-ről 3-ra növeli.
  * Mivel a $3 < 3$ feltétel már **hamis** ezért a j-hez tartozó ciklusnak vége, visszalépünk az i-hez tartozóba. Mivel azonban itt több sor nincs ezért i értékét 1-ről 3-re növeljük.

– Mivel a $2 < 4$ feltétel igaz, ezért...

A ciklus hátralévő sorait már nem írjuk ki, az olvasó számára valószínűleg már egyértelmű, hogy mi történik.

- A legegyszerűbb szemléltetési módja a beágyazott ciklusoknak az nem más mintha elképzelünk egy minden lakásban fellelhető vízórát. Ha megnyitjuk a csapot úgy a legutolsó számlap kezd pörögni 0-tól egészen 9-ig, majd ha elérte a 9-et úgy visszafordul a kiindulási 0 értékre **miközben** az előtte lévő számlap eggyel nő. Lényegében a fenti példában is erről van szó. Így az összes lehetséges esetet könnyen bemutathatjuk az alábbi ábra segítségével:



Megjegyzés: Látható, hogy i-vel "beleragadunk" egy-egy értékbe egészen addig amíg a j változó összes lehetséges értéke végig nem fut. Így az (i, j) párok az alábbiak szerint változnak:

$$(0,0),(0,1),(0,2) \;||\; (1,0),(1,1),(1,2) \;||$$
$$(2,0),(2,1),(2,2) \;||\; (3,0),(3,1),(3,2).$$

- A fenti példában megfigyelhető volt, hogy a két ciklusban szereplő ciklusváltozók különböző nevet kaptak: i és j. Ennek oka igen egyszerű: ha mindkét változó ugyanolyan névvel rendelkezne úgy követhetetlen lenne, hogy pl. a *tomb2d[i][i]* alatt mit értünk. Sok programozási nyelv eleve nem is engedi, hogy egymásba ágyazott ciklusok ciklusváltozói ugyanúgy legyenek elnevezve, ám néhány nyelv esetén (pl. C, C++) ez nem jelent akadályt, bár nehezen követhető és sok hibára ad lehetőséget. Példaként tekintsük az alábbi két kódrészletet melyek C-ben készültek:

```
int i;
for(i=0;i<10;i=i+1){
    for(i=0;i<3;i=i+1){
        printf("%d\n",i);
    }
}
```

```
for(int i=0;i<10;i=i+1){
    for(int i=0;i<3;i=i+1){
        printf("%d\n",i);
    }
}
```

  A különbség a két kód között látszólag csupán annyi, hogy az i nevű változót a bal oldali kódban a ciklus előtt hoztuk létre, míg a jobb oldaliban a ciklusban történt az inicializáció. Ennek ellenére a bal kódrészlet egy végtelen ciklus mely a $0, 1, 2$ számokat írja ki egymás alá, a jobb kód viszont 10-szer írja ki a $0, 1, 2$ számokat, majd megáll. A jelenség magyarázata nem része ennek a műnek, ám ebből talán érthető, hogy miért nem szerencsés ugyanazt a ciklusváltozót használni egymásba ágyazott ciklusok esetén.

Most, hogy alaposan kivesézték az elágazásokat, mind pedig a hagyományos és az egymásba ágyazott ciklusokat, néhány gyakorló feladatot követően rátérünk arra a témakörre, mely valamelyest túlmutat a kezdő szinten, ám alapos megismerése jelentősen hozzájárul a hatékonyabb programozáshoz.

3.5. Feladatok

Tervezzünk algoritmust az alábbi problémák megoldására! A tömböt tartalmazó feladatokban használhatjuk a `HOSSZ(tomb_neve)` utasítást mely megmondja, hogy hány elem van a tömbben. Például az alábbi kód esetén

```
t = [5,7,-1,0,16,9,3]
h = HOSSZ(t)
KIÍR(h)
```

a képernyőn a 7 szám fog megjelenni.

1. Adottak az a, b, c *double* típusú számok. Készítsünk algoritmust mely megadja az $ax^2 + bx + c = 0$ egyenlet összes megoldását. Figyeljünk arra, hogy mi történik ha $a = 0$.

2. Adott x, y egész számok esetén tervezzünk algoritmust mely megcseréli x és y értékét!

3. Adott x *double* szám esetén számoljuk ki x abszolút értékét.

4. Egy országban többkulcsos adórendszer működik. Ha valakinek a fizetése maximum 1000 peták akkor nem kell adót fizetnie. 1001 és 2000 peták között 10% az adó az 1000 peták feletti részre, míg 2000 peták felett a 2000 peták feletti részre 15% az adó (természetesen az $1001-2000$ közötti részen ugyanúgy megmarad a 10%). Készítsük el azt az algoritmust mely a $fizetes$ változó megkapását követően kiszámolja az ado változó értékét.

5. Döntsük el egy háromszögről a három oldalhosszának (a, b, c) ismeretében, hogy az szerkeszthető avagy nem (egy háromszög szerkeszthető ha bármely két oldalának összege nagyobb, mint a harmadik)!

6. Egy $f(x) = a\sin(bx)+c$ alakú függvény esetén adottak az a, b, c számok (az egyszerűség kedvéért $b > 0$ és $a = \pm 1$). Írjuk ki, hogy milyen lépéseket kell végrehajtanunk ha le akarjuk azt rajzolni. A felhasználható parancsok:

   - `RAJZOL`, mely kirajzolja az alapfüggvényt. Ezt csak a legelején kell egyszer meghívni, hogy felkerüljön a "táblára" a $\sin$ függvény.
   - `NYUJT(b)`, `ZSUGORIT(b)`, melyek nyújtják vagy zsugorítják a már lerajzolt függvényt az x tengely mentén. A nyújtás csak akkor működik ha $b > 1$, míg a zsugorítás csak akkor ha $0 < b < 1$, egyébként hibát kapunk.
   - `FELTOL(c)`, `LETOL(c)`, mely az y tengelyen felfelé vagy lefelé tolja a függvényt d-vel attól függően, hogy d pozitív vagy negatív. A `FELTOL(d)` nem működik ha b negatív, hasonlóan a `LETOL(d)` nem működik ha d pozitív.
   - `TUKROZ`, mely az x tengelyre tükrözi a függvényt ha $a = -1$, de hibát ad ha $a = 1$.

   Segítségképp a függvénytranszformációk sorrendje az alábbi:

(a) Alapfüggvény kirajzolása.

(b) Az x tengelyen nyújtás vagy zsugorítás.

(c) Az x tengelyre tükrözés (ha kell).

(d) Fel-, vagy letolás az y tengelyen.

7. Adott n pozitív egész esetén számoljuk ki $n!$ értékét.

8. Adott t egészeket tartalmazó tömb esetén tervezzünk algoritmust mely megkeresi t legkisebb elemét.

9. Adott n és a egészek esetén számoljuk ki n^a értékét! Figyeljünk a speciális esetekre ($n = 0, a = 0$, netán mindkettő egyszerre 0) is!

10. Készítsük el az ún. *FizzBuzz* játékot: adott n pozitív egész esetén írjuk ki a számokat 1-től n-ig úgy, hogy a 2-vel osztható számok helyére a *Fizz* szót írjuk, a 3-al oszthatóak helyére a *Buzz*-ot míg ami mindkettővel osztható oda *FizzBuzz* kerül.

11. Adott t egészeket tartalmazó tömb és x egész szám esetén írjuk ki a képernyőre, hogy hányszor szerepel t-ben az x szám. Például a `t=[3,3,8,12,-5,0,3,2,3,0,-5]` tömb és $x = 3$ esetén a válasz 4, hisz 4 darab hármas van t-ben.

12. Adott egy torta melyen egymás mellett gyertyák vannak. A gyertyák hosszait egy t tömb tartalmazza. Írjuk ki a képernyőre, hogy a gyertyákra oldalról ráfújva hány darab gyertya alszik el! A magasabb gyertyák leárnyékolják az alacsonyabbakat, így ha például a tömb egy részlete `...,10,3,7,8,2,11,5,...` úgy a $3, 7, 8, 2$ és 5 hosszúságú gyertyák biztosan nem alszanak el hisz a $3, 7, 8, 2$-t leárnyékolja a 10, míg az 5-öt a 11 (illetve a 10 is, de az már nem számít).

13. Készítsük el a híres $3k + 1$ probléma algoritmusát. A probléma lényege, hogy adott k pozitív egész esetén amennyiben k páros úgy elosztjuk 2-vel, míg ha páratlan akkor megszorozzuk 3-al és 1-et hozzáadunk. Egy híres matematikai sejtés szerint akármilyen számból indulunk ki idővel 1-et kapunk. Így a feladatunk az, hogy adott k esetén kiírjuk a képernyőre a lépések végrehajtását követően kapott számokat egészen 1-ig (k-t és 1-et is írjuk ki). Feltehető, hogy mindig eljutunk 1-ig. Például ha $k = 13$, úgy a képernyőre írandó számok: $13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

14. Írjuk ki a képernyőre egy adott n pozitív egész esetén n összes pozitív osztóját.

15. Számoljuk ki az n és m pozitív egészek legnagyobb közös osztóját és legkisebb közös többszörösét tetszőleges módon (sok különböző megoldás létezik a feladatra).

16. Adott egy t tömb mely a $0, 1, 2$ számokat tartalmazhatja (de bármelyikből tetszőlegesen sokat). Írjuk ki, hogy melyik számból van benne a legtöbb.

17. Adott `tomb` két dimenziós tömb esetén írjuk ki `tomb` elemeit táblázatos formában. Gyakorlásképp a feladatban csak a "speciális" `MODKIÍR` módosított kiíró-utasítást használjuk mely a kiírt szöveg után nem tesz sem szóközt és új sort sem kezd. Tehát például az

```
a = 5
b = "alma"
c = "körte"
MODKIÍR(a)
MODKIÍR(b)
MODKIÍR(c)
```

kód esetén a kimenet `5almakörte` lesz. Az új sor kezdéséhez használjuk a `MODKIÍR(új sor)` utasítást mely csupán egy darab "entert" üt le. Így ha a fenti kódban az 5-öt és az `alma`-t egy sorban szeretnénk szóközzel elválasztva megjeleníteni, míg a `körte`-t új sorba írnánk akkor az alábbiak szerint kell módosítani a kódot:

```
a = 5
b = "alma"
c = "körte"
MODKIÍR(a)
MODKIÍR(" ") (ez a szóköz)
MODKIÍR(b)
MODKIÍR(új sor)
MODKIÍR(c)
```

18. Adott n pozitív egész esetén rajzoljunk ki a képernyőre $\star$-okból egy:

(a) n magasságú, balra zárt derékszögű háromszöget:
```
*
*  *
*  *  *
```

(b) n magasságú jobbra zárt derékszögű háromszöget:
```
      *
    *  *
  *  *  *
```

(c) n magasságú egyenlő szárú háromszöget:
```
    *
  *  *  *
*  *  *  *  *
```

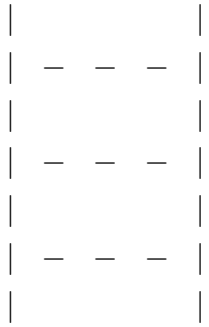
19. Adott n, m egészek esetén rajzoljunk ki a $\star$ jelekből egy $n \times m$ nagyságú teli téglalapot.

Például $n = 3, m = 4$ esetén
```
*  *  *  *
*  *  *  *
*  *  *  *
```
.

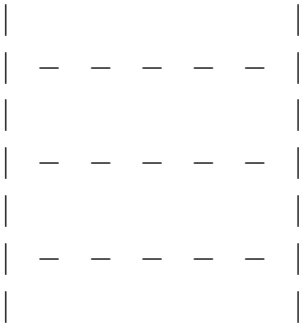
20. Adott t egészeket tartalmazó tömb és x egész szám esetén tervezzünk algoritmust mely kiírja az `IGEN` vagy `NEM` szavak valamelyikét annak megfelelően, hogy van-e t-ben két olyan szám melyek összege x. Például a `t=[3,5,7,9]` tömb és $x = 14$ esetén a válasz `IGEN`, hisz $5 + 9 = 14$.

21. Tervezzünk algoritmust mely kirajzol a képernyőre egy létrát. Az alfeladatok egymástól függetlenek és egyre nehezednek.

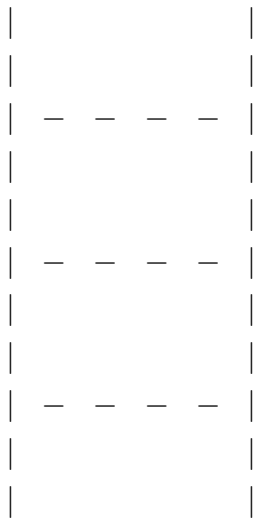
    (a) A létrának egy adott n pozitív egész foka van és fixen 3 széles. Minden fok között egy "üres" rész (szóközökből) található az alábbiak szerint. Például egy $n = 3$ fokú létra így néz ki:

    (b) A létrának egy adott n pozitív egész foka van és minden fok m széles. Ezen kívül a feladat ugyanaz, mint az előző pontban. Például egy $n = 3$ fokú, $m = 5$ széles létra így néz ki:

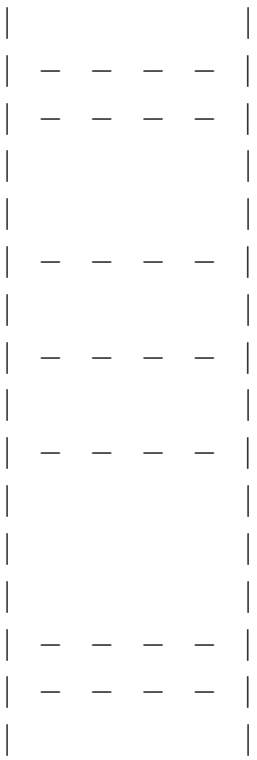
    

    (c) A létrának egy adott n pozitív egész foka van és minden fok m széles. Minden fok között k "üres" rész található. Például egy $n = 3$ fokú, $m = 4$ széles, $k = 2$ hézagos létra így néz ki:

    (d) A létra szélességét egy m egész adja meg. A létra egyetlen üres résszel kezdődik és

végződik ám az egyes fokok közötti hézagok méretét egy t nemnegatív egészeket tartalmazó tömb adja meg. Például egy $m = 4$ széles, $t = [0, 2, 1, 1, 3, 0]$ közökkel megadott létra kinézete:



4. Függvények

Ha máshol nem is, a középiskolai matematikaórán mindenki találkozott már függvényekkel. Kicsit programozói fejjel gondolkozva ezek olyan "struktúrák" melyek valamilyen előre megadott szabály szerint átalakítják a bemenetként kapott információt. Sőt, igazából már általános iskola elején is megismerkednek velük a diákok, bár akkor még rendszerint olyan feladatok keretében melyek szövege a *"A gépbe bedobunk* 3 *kört és* 6 *körte esik ki belőle. Mit csinál a gép?"* mondatokhoz hasonlít. A programozás során használandó függvények felépítés terén valójában nagyon hasonlóak mind a középiskolában, mind az általános iskolában tanultakhoz, így ha valaki ilyen módon próbálja őket megközelíteni akkor ez jelentősen megkönnyíti a megértésüket.

A programozásban a függvények olyan, a fő programtól (javarészt) független kódrészletek, melyek a matematikában látottakhoz hasonlóan adatot, vagy adatokat kapnak a "külvilágból", majd a bennük szereplő kód futtatását követően valamilyen kimenetet adnak a fő programnak. A függetlenség alatt azt értjük, hogy ami egy függvényen belül történik az általában "ott is marad", így ha például a fő programunkban **és** a függvényünkben is létrehozunk egy-egy x nevű változót úgy azoknak egymáshoz semmi közük nem lesz, ha a függvényen belül megváltoztatom az x értékét úgy a fő programban szereplő x változatlan marad. A könnyebb érthetőség kedvéért ezt egy teljesen leegyszerűsített példán keresztül be is mutatjuk:

```
FÜGGVÉNY f(x):
    x = x + 20
```

```
----FŐ PROGRAM----
x = 100
f(x)
KIÍR(x)
----FŐ PROGRAM VÉGE----
```

A példában először a "fő programunkon" kívül definiáltunk egy f függvényt, mely egy x-nek nevezett értéket kap a külvilágból (például a felhasználótól), majd ennek az értékét 20-al növeli és a végeredményt szintén x néven menti el. A "fő programon" belül először létrehozunk egy *int* x változót 100-as kezdőértékkel, majd ezt az x-et átadjuk az f függvénynek, végül kiírjuk az x értékét. Első meggondolásra a képernyőn ekkor 120-at kéne látnunk, azonban a kiírt szám csupán 100 lesz. Ahhoz, hogy ezt megértsük tekintsük át lépésről lépésre, hogy mi is történik a programban:

1. Az x változót "átadjuk" a függvénynek.
2. Ekkor az x **helyett** csupán a 100 íródik bele a függvénybe (tehát **nem** az x változót adtuk oda f-nek hanem **csak annak az értékét**, a 100-at).
3. Az előző lépésben átadott 100 behelyettesítődik a függvényen **belüli** x változó helyére. Azért x helyére, mert a függvény definiálásakor az f után az x-et használtuk. Ha például `FÜGGVÉNY f(y)`-t írtunk volna akkor a 100 az y változó helyét venné át a függvényen belül (erről még később szó lesz).
4. A függvényen belüli x változó, melynek az értéke a kívülről kapott 100 megnő 20-ra, így a **belső** x értéke 120 lesz. A függvény itt véget is ér.
5. Mivel a 2. pontban láttuk, hogy a **külső** x-nek csak az értéke adódott át, így a fő programban szereplő x változó valójában nem módosult, ezért a kiírás során az eredeti 100-at fogjuk látni (a függvény belsejében lévő x pedig "megsemmisült" amint kiléptünk a függvényből).

A fentiekből kitűnik, hogy a függvényben szereplő x-eknek és a fő programban szereplő x-eknek semmi közük egymáshoz. Így a korábbiakkal teljesen megegyező eredményt kapunk akkor is ha a kódunkat az alábbiak szerint módosítjuk:

```
FÜGGVÉNY f(valami123):
    valami123 = valami123 + 20

----FŐ PROGRAM----
x = 100
```

```
f(x)
KIÍR(x)
----FŐ PROGRAM VÉGE----
```

Ekkor a fenti felsorolás 3. pontja fog oly módon megváltozni, hogy *"Az előző lépésben átadott* 100 *behelyettesítődik a függvényen* ***belüli*** $valami123$ *változó helyére."* Látható tehát, hogy igazából az $f(x)$ és az $f(valami123)$ esetében is lényegében $f(100)$-al dolgozunk, az x és a $valami123$ csupán ahhoz kell, hogy erre a 100-ra tudjunk valahogy hivatkozni.

Bár a korábbi néhány bekezdésben már láttunk egy leegyszerűsített példát arra vonatkozóan, hogy nagyjából, hogyan is néz ki egy függvény és milyen módon kell azt használni, ahhoz, hogy alaposabban elsajátíthassuk a működésüket és megértsük, hogy miért is kimondottan hasznosak, elengedhetetlen megismerkednünk az általános felépítésükkel. A használandó nyelv típusától függően két fő módja van a leírásuknak. Az ún. *statikusan típusos* nyelveknél (pl. C, C++, C#, Java,...) szükséges megadnunk a változók, paraméterek, stb. típusát, így az ilyen programozási nyelvek esetén egy függvény nagyjából így adható meg általánosan:

```
<visszatérési típus> NÉV(<típus> paraméter1, <típus> paraméter2,...):
    FÜGGVÉNY TÖRZSE
```

Ezzel szemben a *dinamikus típusrendszerrel* rendelkező nyelveknél (pl. Python, Lua, Ruby,...) elegendő csupán a neveket megadni típusok nélkül:

```
NÉV(paraméter1, paraméter2,...):
    FÜGGVÉNY TÖRZSE
```

- A *visszatérési típus* annak a változónak a típusát jelenti melyet a függvény a lefutását követően a felhasználó rendelkezésére bocsájt. Például ha a középiskolából megismert $f(x) = x^2$ függvényre gondolunk úgy ennek a visszatérési típusa egy (pozitív) *double* szám. Ha egy olyan függvényt készítünk mely eldönti egy számról, hogy prímszám-e, vagy nem úgy annak a visszatérési típusa *boolean* (igaz, vagy hamis). Hasonlóképp ha a függvényünk egy diák nevét bocsájtja rendelkezésünkre úgy a visszatérési típus egy *string* (str).

- Az előző pont mintájára a *paraméter típus* annak a változónak a típusát jelenti amit a függvény a "külvilágból" kap. Például az $f(x) = \sqrt{x}$ függvény a felhasználótól csupán nemnegatív számokat kaphat, hisz a valós számok körében a negatív számokból nem lehetséges a gyökvonás. Ugyanígy ha a már látott "prímszám eldöntő" függvényt képzeljük el, úgy az a felhasználótól egy pozitív egész számot kell, hogy kapjon, hisz a prímséget törtek, vagy negatív számok esetén nem szokás értelmezni.

Az előző pontokban szó esett a *visszatérési értékről*, illetve a *paraméterek* is felmerültek. Az alábbiakban áttekintjük, hogy ezek pontosan mit is jelentenek:

- **Visszatérési érték** (angolul *return value*): ahogy korábban már említésre került ami a függvényeken belül történik az (általában) a függvényeken belül is marad. Viszont nagyon gyakran (sőt, szinte mindig) szükségünk van arra, hogy a függvényben végrehajtott műveletek eredményét a fő programunkban felhasználjuk. (Egy egyszerű példa: ha a legkisebb ötjegyű prímszámot keressük akkor nem elég, hogy egy függvényen belül el tudjuk dönteni a 10067-ről, hogy prím (és a $10000, \ldots, 10066$-ról, hogy nem az), ezt az információt valahogy a fő programunk tudtára is kell hoznunk.) Erre a feladatra alkották meg a *visszatérési értéket*, vagy ismertebb nevén a *return value*-t (vagy "félmagyarosan" *return értéket*). Ennek a szerepe nagyon konyhanyelven és leegyszerűsítve annyi, hogy a *return* kulcsszó után szereplő értéket "behelyettesíti" a függvény helyére (ez egy túlzottan is leegyszerűsített és igen pontatlan megfogalmazás, ám a könnyebb megérthetőséghez megfelelő), így ha egy változó értékének a függvényt adjuk meg, úgy amint az lefutott a változó értéke ez a return érték lesz. Egy egyszerű példa:

```
FÜGGVÉNY tesztfv(y):
    alma = y*20
    HA alma > 100 AKKOR:
        return 50
    EGYÉBKÉNT:
        return alma/2

----FŐ PROGRAM----
x = 3
eredmeny = tesztfv(x)
KIÍR(eredmeny)
----FŐ PROGRAM VÉGE----
```

A példában először definiáltuk a *teszfv* nevű függvényt, mely egy y értéket kap bemenetként, ezt megszorozza 20-al, majd az eredményt elmenti az *alma* nevű változóban. Végül megvizsgálja az *alma* változó értékét és ha ez 100-nál nagyobb úgy 50-et "ad vissza", egyébként pedig a $alma/2$-t. A fő programunkban a létrehozott $x = 3$ változót átadjuk a függvénynek (ezt úgy szokás mondani, hogy **meghívjuk a függvényt** x**-el**). Így a 3 bekerül a függvénybe y **helyére** (tehát lényegében a *tesztfv(*3*)* fut le). Az *alma* változó értéke 60 lesz, ezért az elágazás egyébként ága fog kiértékelődni, így a ***tesztfv(***3***)* értéke** 30 **lesz** (hisz *alma*=60, így $alma/2 = 30$). Ezt a 30-at rendeljük hozzá az *eredmeny* változóhoz, ezért a kiírás során a képernyőn a 30 fog megjelenni. Maga az *alma* változó és az y is "megsemmisül" a függvény futását követően.

- *Megjegyzés 1:* több programozási nyelv engedélyezi az "üres" return utasítást. Ennek a lényege, hogy a függvény leáll, ám a fő programnak semmilyen értéket nem bocsájt rendelkezésére. Az ilyen kimondottan hasznos lehet például akkor ha a függvényünknek csupán annyi a szerepe, hogy a képernyőre írjon egy üzenetet, majd kilépjen (*"HA van a megadott számok között prímszám AKKOR írd ki, hogy melyik az, majd return"*). Azoknál a nyelveknél ahol a függvény definíciójában meg kell adnunk a visszatérési érték típusát (statikusan típusos nyelvek) az "üres" return-re csak akkor van lehetőségünk ha a visszatérési érték típusa ún. *void* (üres, semmi) típus. Minden egyéb esetben (*int, string,...*) muszáj visszaadnunk valamilyen értéket akkor is ha azt nem használjuk fel. Int esetén gyakori a *return 0* utasítás mely hagyományosan a hiba nélküli futást jelzi (ún. 0-s return kód), míg ha stringeknél szeretnénk a "semmit" visszaadni úgy azt nyelvtől függően return NULL, return None, return NIL, stb.-vel tehetjük meg.
- *Megjegyzés 2:* sok programozási nyelv csupán egyetlen return értéket engedélyez, így ha pl. egy függvény visszaadná egy diák nevét és születési évét is úgy erre különböző trükköket kell alkalmazni melyekkel azonban jelen könyv keretein belül nem

foglalkozunk. Más nyelvek (pl. a Python, Lua, stb.) engedik a több érték visszaadását is (pl. `return nev,eletkor`), így itt ilyen probléma nem merül fel. **Fontos** kiemelni azonban, hogy az, hogy egy nyelvben a függvények csak egy return értékkel rendelkezhetnek nem jelenti azt, hogy a függvényben csupán egyetlen return utasítás szerepelhet. Ha visszatekintünk a korábban már látott *tesztfv*-re úgy megfigyelhetjük, hogy abban **két** return utasítás található, ám mindkét utasítás csupán **egyetlen** értéket ad vissza. Tehát ez egy olyan függvény ami egyetlen return értékkel rendelkezik annak ellenére, hogy többször szerepel benne a return szó.

– *Megjegyzés 3:* a return utasítás valamelyest hasonlít a ciklusoknál megismert break kulcsszóhoz abban az értelemben, hogy amint a függvény egy ilyen paranccsal találkozik úgy abban a pillanatban kilép és visszaadja a return után szereplő értéket/értékeket. Így nagyon meg kell gondolni az utasítások sorrendjét, hisz a return utáni sorok már nem kerülnek kiértékelésre. Egy könnyen érthető példa ennek a problémának a bemutatására a következő: tegyük fel, hogy van 100 dobozunk és az utolsótól kezdve kidobjuk egyesével mindet egészen addig amíg valamelyikben nincs egy piros labda. Amint a labdát megtaláltuk visszaadjuk ennek a doboznak a számát, majd beállítjuk, hogy ez a doboz legyen a sorban az utolsó (hisz az utána következőket már kidobtuk). Ekkor ha például a $73.$ dobozban van a labda akkor a $74., 75, \ldots, 100.$ dobozokat már mind kidobtuk, így az utolsó doboz a sorban a $73.$ lesz. Jó ötletnek tűnik, hogy a függvényünk belsejét egy ehhez hasonló kóddal töltsük fel:

```
utolso_doboz = 100
FOR i=100;i>0;i=i-1:
    HA van piros labda az i-edik dobozban AKKOR:
        return i (ez a doboz száma)
        utolso_doboz = i
```

Először beállítjuk, hogy az utolsó dobozunk a $100.$, hiszen az elején még egyet sem dobtunk ki. Ezt követően 100-tól visszafelé haladva egyesével megnézzük, hogy a $100., 99., \ldots$ dobozokban van-e piros labda. Ha nincs akkor egyszerűen megyünk tovább és nézzük a következőt, ha viszont van akkor visszaadjuk a doboznak a számát, majd beállítjuk az utolsó doboznak azt amiben a labda volt. Emberi aggyal ez egy teljesen jól működő kód, ám a fentebb említettek alapján mégis hibás, hiszen amint a return-el visszaadtuk a doboz számát az utolsó sor már nem értékelődik ki, így az *utolso_doboz* változó végig 100 fog maradni. Ennek megoldása jelen feladatban nagyon könnyű, csak meg kell cserélni a return utasítást az azt követővel. Természetesen a tényleges programozási feladatok között gyakran felmerülnek olyanok ahol nem ilyen egyszerű a megoldás, ám ha végig szem előtt tartjuk, hogy a return utáni részek olyanok mintha "ott sem lennének" úgy viszonylag könnyen rá lehet jönni

arra, hogy hogyan kell felépíteni a kódunkat. **Fontos** azonban tisztában lenni azzal, hogy ez a bekezdés arra az esetre vonatkozik amikor a programnak **ténylegesen ki kell értékelnie** egy return utasítást. Tekintsük példának az alábbi kódrészletet:

```
HA pontszam < 60 AKKOR:
    return "Elégtelen"
HA pontszam < 70 AKKOR:
    return "Elégséges"
HA pontszam < 80 AKKOR:
    return "Közepes"
HA pontszam < 90 AKKOR:
    return "Jó"
EGYÉBKÉNT:
    return "Jeles"
```

Ha a diák 85 pontot ért el úgy annak ellenére, hogy a program átsiklik három return utasításon (< 60, < 70, < 80) mégsem áll meg egyiknél sem, hanem csupán a < 90 ágba lép be. Így hiába "találkozik" a számítógép több return paranccsal, ha azok nem kerülnek kiértékelésre (pl. egy elágazás miatt) úgy a program ott természetesen nem fog megállni. Ez kellő gyakorlással idővel teljesen magától értetődő lesz, ám a kezdők gyakran futnak bele abba a hibába, hogy félnek leírni a return szót mert attól tartanak, hogy a program minden egyes sort lefuttat (még azokat is ahol a feltétel nem teljesül) és így ott ki fog lépni.

- **Paraméter/argumentum**: a paraméterek, vagy argumentumok azok az értékek melyeket a függvény a "külvilágból" kap. Ezeket a korábbi példákban már használtuk mindenféle részletesebb magyarázat nélkül, ám a függvények pontos megértéséhez elengedhetetlen ezek részletesebb megismerése. Mielőtt azonban erre rátérnénk, a pontosság látszatának fenntartása érdekében beiktatunk egy rövid megjegyzést:

  – Bár sok programozó (a könyv szerzőjét is beleértve) általában egymás szinonimájaként használja a paraméter és argumentum szavakat, a valóságban ezek két különböző dolgot jelölnek: a paramétereket a függvény definíciójában használjuk (pl. a néhány oldallal fentebbi *tesztfv(y)* esetén az *y* egy paraméter), míg az argumentumok a függvény meghívása során átadott értékek (tehát ha azt írjuk, hogy *tesztfv(*100*)*, vagy, hogy *tesztfv(x)*, ahol *x* a fő programban létrehozott változó akkor argumentumokat adtunk át a függvénynek). Bár ez a különbség igen fontosnak látszik, a programozók jelentős hányada nem is tud róla (vagy nem foglalkozik vele), így nagy valószínűséggel senki nem fog furcsán ránk nézni ha ezen két fogalmat szinonimaként használjuk :)

Bár a korábbi példákban szándékosan egyparaméteres függvényeket használtunk (hisz a

középiskolában is főként olyan függvényekkel futhattunk össze melyben csak az x volt a változó), a gyakorlatban rendszerint több paraméterrel dolgozunk. Ezeket egyszerűen vesszővel elválasztva soroljuk fel a függvény definíciója során, nyelvtől függően vagy a típusukkal együtt (pl. C, C++, Java, stb. esetén *int sor, str nev, str azonosotio*-ként) vagy egyszerűen a nevükkel (pl. Pythonnál, Lua-nál, Rubynál *sor, nev, azonosito*-ként). Az ún. *paramétersoron* megadott paraméterek sorrendje kötött, mely azt jelenti, hogy a függvény meghívása során az átadott argumentumoknak a definícióban megadott sorrendben kell követniük egymást. Így például ha egy függvény definíciója

```
FÜGGVÉNY szamOszto(x,y):
    return x/y
```

úgy ha végeredményként $\frac{3}{2}$-t szeretnénk kapni úgy a függvényünket `szamOszto(3,2)`-vel kell meghívnunk. Bár ez első ránézésre triviálisnak tűnik, a függvényekkel most ismerkedők számára ez néha mégis problémát okoz. A probléma forrásának megértéséhez tekintsük az alábbi kódrészletet:

```
FÜGGVÉNY furaFuggveny(w,x,y,z):
    a = x + y
    b = z * w
    c = a + b - x
    return a + b + c
```

Maga a függvény első ránézésre nem tűnik bonyolultnak, csak néhány alap matematikai művelet van benne. Ha például `furaFuggveny(5,10,15,20)`-al hívjuk meg akkor könnyen kiszámolható, hogy a visszatérési értéke

$$\begin{aligned} a = &\quad x + y &= &\quad 10 + 15 &= 25 \\ b = &\quad z \cdot w &= &\quad 20 \cdot 5 &= 100 \\ c = a + b - x &= 25 + 100 - 10 = 115, \end{aligned}$$

240 lesz $(25 + 100 + 115)$. Azonban mit látunk a képernyőn ha az alábbi kódot futtatjuk?

```
----FŐ PROGRAM----
a = 10
b = 20
c = 30
w = 15
x = 20
y = 10
```

```
z = 5
eredmeny = furaFuggveny(z,y,w,x)
KIÍR(a,b,c,eredmeny)
----FŐ PROGRAM VÉGE----
```

Ha visszaemlékszünk a néhány oldallal korábban tanultakra akkor a kiírásban szereplő a, b, c-re egyből mondhatjuk, hogy $10, 20, 30$, hisz "ami a függvényen belül történik az (javarészt) független a fő programtól", így a függvényen belüli a, b, c-nek semmi köze a "kinti" a, b, c-hez. Az érdekesebb kérdés az, hogy mi lesz az *eredmeny* változó értéke? A problémát az jelentheti, hogy a függvény definíciójában is w, x, y, z-k szerepelnek és a fő program is ilyen nevű változókat használ. *Így logikus gondolatnak tűnik, hogy a fő programban szereplő w, x, y, z-t egyszerűen behelyettesítsük a függvényen belüli w, x, y, z helyére és így számoljuk ki a végeredményt.* Azonban ez **helytelen** megoldást adna. Ennek oka a már említett sorrendiségben keresendő. A `furaFuggveny` négy argumentummal rendelkezik melyeknek a nevét a programozó választja ki. Azonban ez a név tetszőleges lehet és semmi köze nincs a fő programban szereplő w, x, y, z-hez. Kezdők számára segíteni szokott ha a függvény definícióját az alábbiak szerint átírjuk:

```
FÜGGVÉNY furaFuggveny(param1,param2,param3,param3):
a = param2 + param3
b = param4 * param1
c = a + b - param2
return a + b + c
```

A fenti kódban semmi mást nem csináltunk csak a w-t $param1$-re, az x-et $param2$-re, az y-t $param3$-ra, míg a z-t $param4$-re cseréltük. Így már nincs ütközés a fő programban szereplő w, x, y, z változókkal, tehát ha a `furaFuggveny(z,y,w,x)`-re vagyunk kíváncsiak akkor a $param1$ értéke 5 (hisz $z = 5$), a $param2$ értéke 10 ($y = 10$), a $param3$ értéke 15 ($w = 15$) és végül a $param4$ pedig 20 ($x = 20$). Összefoglalva így a `furaFuggveny(5,10,15,20)`-at kell kiszámolnunk amit korábban már megtettünk és 240-et kaptunk. *Tehát amit a fentiekből érdemes fejben tartanunk az csupán annyi, hogy a függvényekben a paraméterek sorrendje számít és nem a nevük.*

A fejezetet végül néhány megjegyzéssel/érdekességgel zárjuk:

- Többször említésre került, hogy a függvények nem képesek közvetlenül hatni a fő programban szereplő változókra. Ám minden ilyen mondatnál volt egy zárójeles megjegyzés mely azt sugallta, hogy ez a kijelentés nem mindig igaz. Bizonyos esetekben lehetőségünk van arra, hogy a függvény belsejéből módosítsunk "kívül lévő" változókat. A könyv bevezető jellegéből adódóan itt csupán $2 + 1$ példát említünk:

– A leggyakoribb és igen hasznos módja annak, hogy egy függvényen belül megváltoztassunk egy külső változót az ha ez ennek a külső változónak a típusa "tömbszerű" (tömb, lista, stb.). Amennyiben egy tömböt paraméterként adunk át a függvényünknek úgy ha annak elemeit megváltoztatjuk ezek a változások return nélkül is megjelennek a fő programban szereplő tömbben. Tekintsük az alábbi kódrészletet:

```
FÜGGVÉNY valtoztato(t):
    t[0] = 20
    t[1] = 30

----FŐ PROGRAM----
tomb = [5,2,-3,7]
valtoztato(tomb)
KIÍR(tomb)
----FŐ PROGRAM VÉGE----
```

A `valtoztato` függvény nem rendelkezik return értékkel, így látszólag nincs lehetőségünk a fő programban lévő értékeket módosítani. Azonban a "hagyományos" változókkal ellentétben a tömbökön történő változások megmaradnak, így a kiírás során a képernyőn a `[20,30,-3,7]` értékek fognak feltűnni. Ez a technika kimondottan hasznos az olyan programnyelvekben melyekben nincs lehetőségünk egynél több értéket visszaadni a return utasítással. Ha például három számot szeretnénk átadni a fő programnak egyetlen függvény segítségével úgy ennek egy lehetséges megoldása az, hogy paraméterként egy háromelemű, egészeket tartalmazó tömböt adunk meg és ennek a tömbnek a három elemét állítjuk be a visszaadandó értékekre. Így a függvényből való kilépés után rendelkezésünkre fog állni a kívánt három változó pl. `tomb[0], tomb[1]` és `tomb[2]` neveken.

– Egy másik, viszonylag ritkán használt lehetőség a külső változók megváltoztatására az, hogy a változókat ún. globális változókként definiáljuk. Így a függvények akkor is elérik ezeket ha nem adjuk át nekik őket paraméterként. Azonban az, hogy a globális változók függvényen belüli változtatása megmarad-e a függvény kilépését követően erősen függ a használt programozási nyelvtől, így egyrészt ez nem egy biztos módszer arra, hogy elérjük a kívánt célt, másrészt a globális változók összeakadhatnak az ugyanolyan nevű lokális változókkal, így az elnevezés során figyelni kell arra, hogy ez ne történhessen meg. Így összességében a tapasztaltabb programozók ezt a technikát erre a célra nem szokták javasolni.

– Az utolsó bemutatott technika csupán azoknál a programozási nyelveknél működik melyekben lehetőség van a memóriacímekhez való (közvetlen) hozzáféréshez (pl. a C nyelv). A módszer lényege az, hogy a függvénynek paraméterként nem a változót adjuk át, hanem azt a memóriacímet ahol a változó található. Ezt úgy lehet

egyszerűen (és elnagyoltan) elképzelni, hogy a memóriában (RAM) minden létrehozott változó elfoglal néhány "doboznyi szabad helyet". Minden doboznak van egy "száma" (például $0x8048552$) amellyel hivatkozni lehet rá. Ha a függvénynek argumentumként a változó memóriában elfoglalt helyét adjuk meg, úgy ha az ezen a helyen lévő értéket (a "dobozban lévő számot") módosítjuk maga a hely ugyanaz marad. Így a szó szoros értelmében nem változtatjuk a fő program részeit (hisz a memóriahely változatlan) ám az ott szereplő változó értéke mégis más lesz a függvény futását követően. Ez azonban egy nagyon elnagyolt és pontatlan leírása az ún. *pointerek* kezelésének, így azon olvasók akik alaposabban szeretnék beleásni magukat ebbe a témába sokkal jobban járnak egy olyan könyv tanulmányozásával mely kimondottan egy erre a feladatra alkalmas programozási nyelv (pl. a C/C++) ismertetésével foglalkozik.

A fejezetet az előzőekhez hasonlóan néhány gyakorló feladattal zárjuk.

4.1. Feladatok

Készítsünk függvényt az alábbi problémákra. Készítsünk mindig egy "fő programot" (a fentieknek megfelelően) ahol meghívjuk az elkészült függvényt.

1. Készítsük el a "létrás" feladat kivételével a 3.5. fejezet végén lévő példákat függvények segítségével. A paraméterek azok a változók melyek a feladatokban "adott"-ként vannak feltüntetve. A visszatérési érték a feladat szövege alapján kerüljön kitalálásra (pl. a $3k + 1$ problémánál és a *FizzBuzz* feladatnál nincs visszatérési érték, hisz csak a képernyőre írunk, a legnagyobb közös osztónál és legkisebb közös többszörösnél viszont egy tömb melynek 0. eleme az *lnko*-t tartalmazza az 1. eleme az *lkkt*-t.)

2. Készítsünk egy függvényt mely paraméterként egy pozitív egész n-et kap és kimenetként `true` vagy `false` értékkel tér vissza attól függően, hogy n prím-e. Ezt követően a fő programban írjuk ki a függvény segítségével, hogy 1-től 1000-ig hány prímszám található.

3. Készítsük el a korábban már látott "létrarajzoló" feladatot (feladat száma: 21) függvényekkel. Készítsünk `uresFok, teliFok` függvényeket melyek a paraméterként megadott m szélességű "üres", vagy "teli" fokot készítik el, majd hívjuk meg őket a fő programon belül egy megfelelő ciklusban, hogy kirajzolják az egész létrát.

4. Készítsünk függvényt mely paraméterként egy két dimenziós t tömböt kap, visszatérési értékként pedig a sor- és oszlopösszegek maximumát adja. Például a $\begin{matrix} 1 & 0 & -5 \\ 3 & 2 & 0 \\ 7 & 1 & 4 \end{matrix}$ tömb esetén (az egyszerűbb bemutatás miatt táblázatos formában ábrázoltuk) a visszatérési érték 12, hiszen a sorok összegei $-4, 5, 12$, az oszlopoké pedig $11, 3, -1$. Ezen hat szám maximuma pedig 12.

5. Készítsünk függvényt mely paraméterként egy két dimenziós listát kap melynek minden eleme egy `[név,születési év]` alakú lista és visszaadja a legöregebb embernek a nevét. Ha több ilyen van akkor elég az egyiket visszaadni. A bemenetre egy példa: `[ ["Károly",1990],["Juli",1986],["Bea",1991] ]`. Ekkor a kimenet *"Juli"*.

6. Készítsünk egy függvényt mely növekvő sorrendbe rendezi a paraméterként kapott egészeket tartalmazó tömb elemeit oly módon, hogy minden elemet összehasonlít mindennel és az egyes összehasonlítások után ha szükséges (ha a későbbi elem kisebb) megcseréli a két elemet. Például a `SORBARENDEZ([5,7,3,4,0])` esetén az alábbi összehasonlítások és cserék történnek meg (csak az elejét tüntetjük fel):

```
t[0]=5 összehasonlítva t[0]=5-el: az 5 marad a helyén
t[0]=5 összehasonlítva t[1]=7-el: az 5 marad a helyén.
t[0]=5 összehasonlítva t[2]=3-al: az 5 és a 3 helyet cserél.
t[0]=3 összehasonlítva t[3]=0-el: a 3 marad a helyén.
t[0]=3 összehasonlítva t[4]=0-val: a 3 és a 0 helyet cserél.
Így a tömb eddig: [0,7,5,4,3].
t[1]=7 összehasonlítva t[0]=0-val: a 7 marad a helyén.
t[1]=7 összehasonlítva t[1]=7-el: a 7 marad a helyén.
t[1]=7 összehasonlítva t[2]=5-el: a 7 és 5 helyet cserél.
t[1]=5 összehasonlítva t[3]=4-el: az 5 és 4 helyet cserél.
t[1]=4 összehasonlítva t[4]=3-al: a 4 és a 3 helyet cserél.
Így a tömb eddig: [0,3,7,5,4].
Stb...
```

7. Készítsünk el egy olyan függvényt mely egy paraméterként megadott számokat tartalmazó tömb esetén visszaadja annak a legkisebb elemét és a legkisebb elem helyét egy kételemű tömbként. A kilépés előtt a paraméterként kapott tömb legkisebb elemét cserélje ki 100-ra. Ha több legkisebb elem van úgy csakis az egyiket cserélje ki és csakis annak a helyét adja vissza. Példaprogram:

```
FÜGGVÉNY legkisebb(tomb):
    hely = ...
    ertek = ...
    ...
    tomb-ben az ertek-edik helyen lévő elem 100 lesz
    return [hely,ertek]

----FŐ PROGRAM----
```

```
tomb = [4,8,16,2,8,4,3,2,7,3,2]
eredmeny = legkisebb(tomb)
KIÍR(eredmeny) -> [10,2]
KIÍR(legkisebb) -> [4,8,16,2,8,4,3,2,7,3,100]
----FŐ PROGRAM VÉGE----
```

Természetesen kódtól függően bármelyik 2-es számnak a helyét visszaadhatta volna.

8. Készítsünk el egy olyan függvényt mely növekvő sorrendbe rendezi a bemenetként megadott egészeket tartalmazó tömböt az alábbi megkötésekkel:

   (a) A tömb minden eleme 100-nál kisebb.

   (b) A rendezést úgy kell megcsinálni, hogy megkeressük a tömb legkisebb elemét majd azt hozzáadjuk egy másik tömbhöz. Ezt követően megkeressük a következő legkisebb elemét és azt is hozzáadjuk. Ehhez használjuk a `HOZZÁAD(tomb,elem)` parancsot mely a `tomb` végéhez hozzáfűzi a megadott `elem`-et.

   Tipp: használjuk az előző feladatban elkészített függvényt.

9. Készítsünk egy olyan függvényt mely bemenetként egy egészeket tartalmazó tömböt kap és `true`-val tér vissza akkor ha igaz az, hogy a tömb első "néhány" elemének összege megegegyezik a maradék elemek összegével. Egyébként a visszatérési érték `false`. Például a `tomb=[3,4,6,1,2]` esetén a visszatérési érték `false`, hisz:

$$3 \neq 4+6+1+2$$
$$3+4 \neq 6+1+2$$
$$3+4+6 \neq 1+2$$
$$3+4+6+1 \neq 2.$$

   Ellenben a `tomb=[4,1,0,3,6,2]` esetén a függvény `true`-t ad, hisz:

$$4 \neq 1+0+3+6+2$$
$$4+1 \neq 0+3+6+2$$
$$4+1+0 \neq 3+6+2$$
$$\mathbf{4+1+0+3=3+2}$$

10. Készítsünk egy függvényt mely `true`-t ad vissza ha a bemenetként megadott szó palindrom (azaz visszafelé olvasva ugyanaz), egyébként `false`-ot. A szavakat kezeljük úgy, mintha tömbök lennének, tehát az 1. betűjük a `szo[0]`, a 2. betűjük a `szo[1]`, stb. helyeken vannak.