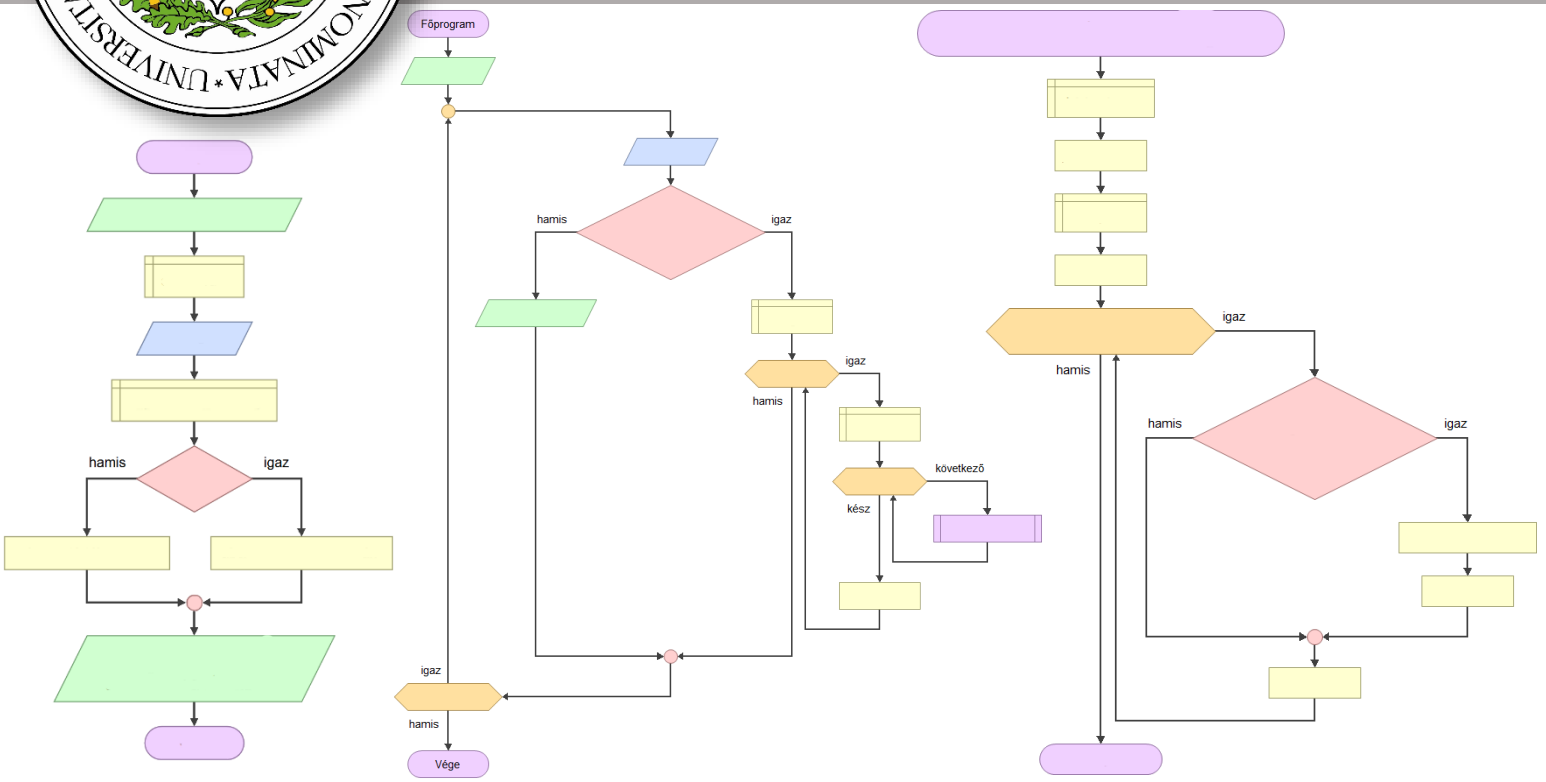
Belépő a tudás közösségébe

Szakköri segédanyagok tanároknak



Algoritmizáljunk Flowgorithm-mel!

Szabó Zsanett

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2019-ben.

Eötvös Loránd Tudományegyetem
Informatikai Kar

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Algoritmizáljunk Flowgorithm-mel!

Szerző

Szabó Zsanett

Felelős kiadó

ELTE Informatikai Kar
1117 Budapest, Pázmány Péter sétány 1/C.

ISBN 978-963-489-149-9

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2019-ben

Előszó

Kedves Kolléga!

Kezdő tanárként sokat gondolkoztam azon, hogy hogyan és mivel lenne érdemes algoritmizálást tanítani olyan (egyébként hatosztályos gimnáziumi) csoportokban, ahol a tanulók egy része találkozott már különféle (általában blokk alapú vagy rajzos) programozási környezetekkel, míg a tanulók másik részének egyáltalán nincs ilyen jellegű előzetes ismerete. Az eddigi saját tapasztalataim, illetve idősebb kollégáim algoritmizálás tanításával, illetve Flowgorithm-mel kapcsolatos tapasztalatai alapján úgy gondolom, hogy a Flowgorithm jó választás ilyen esetekben, hiszen egyformán segít a korábban megszerzett tudás rendszerezésében, illetve az új ismeretek elsajátításában kezdő és némi tapasztalattal rendelkező tanulók számára is. Nem egy konkrét programozási nyelvhez kötődik, hanem számos nyelv elsajátításának kezdeti lépéseit segíti, miközben a fő hangsúly az algoritmikus gondolkodásmód elsajátításán tud maradni.

Ez az anyag a Flowgorithm program lehetőségeinek rövid bemutatása mellett konkrét kidolgozott feladatokat tartalmaz némi módszertani útmutatással, illetve saját tapasztalatokkal kiegészítve. A konkrét feladatok mellett néhány kitűzött feladatnak is helyet adtunk, amelyekről úgy gondoljuk, hogy a kidolgozott példák alapján könnyen megoldhatók, akár házi feladatként is kiadhatók a tanulók számára.

Az itt leírt feladatok többségét 8. osztályos tanulókkal próbáltam ki, náluk jól működött. Úgy gondolom, hogy – esetenként némi módosítással – érdeklődő csoport esetén akár már 5. osztályban alkalmazható a feladatok jelentős része, illetve a Flowgorithm program. A korábban rajzos vagy blokkos környezetben tapasztalati úton megszerzett tudás rendszerezésére, összefoglalására pedig idősebbeknél is jól használhatónak tartom a Flowgorithm programot, illetve az itt található feladatokat.

A kidolgozott feladatok többségénél természetesen lehetőség van a feladat egyszerűsítésére, illetve nehezebbé tételére is. Diákjaim előszeretettel javasoltak az órákon a feladatokhoz módosítási, kiegészítési lehetőségeket, illetve őket érdeklő új feladatokat is. Ezen ötletek és saját feladatok beépítése pedig tovább növelte a csoport órai aktivitását és munkakedvét. Úgy gondolom, hogy más csoportok esetén is számítani és építeni lehet a tanulók kreativitására, ötleteire.

Minden tanulócsoport különböző, így a valószínűleg az egyes csoportokat más-más feladattal lehet jobban lekötni, motiválni, de bízom benne, hogy az itt szereplő feladatanyag jó kiindulópont lesz mindenkinek mindenféle korosztályhoz és a program használatához.

Jó munkát!

A szerző

Az anyagban használt jelölések

Az anyagban különböző jelöléseket használunk bizonyos információk kiemeléséhez, elkülönítéséhez, hogy megkönnyítsük többek között a program használatával, illetve az algoritmizálás témakör tanításával kapcsolatos tudnivalókat, megjegyzéseket. Ennek érdekében a következő jelöléseket vezettük be:

Ilyen sárga kiemeléssel jelöljük a Flowgorithm használatával, beállításaival és egyéb sajátosságaival kapcsolatos megjegyzéseket.

Az algoritmizálás témakör, illetve az anyagban szereplő feladatok tanításával kapcsolatos módszertani megjegyzéseket, saját tanítási tapasztalatból fakadó tanácsokat, javaslatokat ilyen szürkéskék háttérszínnel jelöltük.

A program bemutatásánál néhány esetben példákkal igyekeztünk érthetőbbé, szemléletesebbé tenni a mondandónkat. Ezeket, illetve a kidolgozott feladatok esetén a feladatok szövegét ilyen háttérszínnel emeltük ki.

A kidolgozott feladatoknál az egyes feladatok végén összegyűjtöttük a feladat megoldásához szükséges új ismereteket, amelyek a korábbi feladatokban még nem kerültek elő. Ezeket a részeke ilyen zöld háttérszínnel emeltük ki.

Források

A Flowgorithm-mel kapcsolatos információk elsődleges forrása a program weboldala (http://www.flowgorithm.org), melyeket saját tapasztalatokkal egészítettünk ki.

Az anyagban szereplő feladatok többsége sokak számára ismerős lehet, hiszen közismert, bárki által használható egyszerű feladatokról van szó. Ezt a gyűjteményt az eddigi tapasztalataim, a saját, illetve diákjaim ötletei alapján válogattam össze. A különböző tantárgyakhoz kapcsolódó feladatoknál pedig kollégáim voltak segítségemre.

A Flowgorithm bemutatása

A Flowgorithm elnevezésű ingyenesen letölthető program[1] neve két szó, a *flowchart* (folyamatábra) és az *algorithm* (algoritmus) összevonásából jött létre. A program egy olyan programozási környezetet biztosít számunkra, amiben folyamatábra formájában készíthetjük el programjaink algoritmusát, így az algoritmikus gondolkodásmód elsajátítását, fejlesztését segíti. Az elkészített algoritmust futtathatjuk egészében vagy lépésenként, futtatás közben figyelhetjük a változók értékeit és ráadásként még többféle (a 2.22.1-es verzióban 24 féle) programozási nyelven is megtekinthetjük az algoritmushoz tartozó programkódot.

A Flowgorithm programot folyamatosan fejlesztik. Első változata 2014-ben jelent meg, jelenleg pedig a 2.22.1-es a legfrissebb verzió, amit 2019. július 21-én tettek közzé[2]. A program nagy előnye, hogy a felhasználói felület nyelvének kiválasztásakor 30 nyelv közül választhatunk, melyek között a magyar is szerepel, ráadásul csak kis (6 Mb-nál kevesebb) helyet foglal el a teljes program. Némi hátrányt jelent azonban, hogy a Flowgorithm egyelőre csak Windows-os környezetben használható, de ígérik, hogy később Linux-ra és Macintosh-ra is telepíthető lesz.

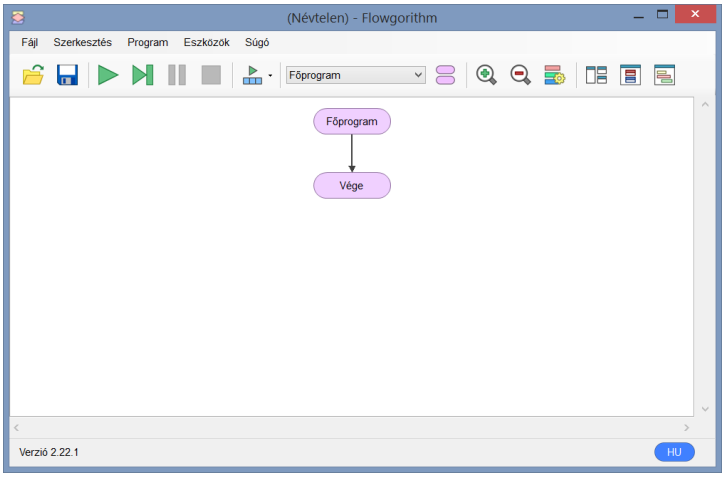
A program felülete rendkívül egyszerű, könnyen átlátható és kezelhető. Nincs zavaróan sok menüpont, így a Flowgorithm használata, lehetőségei egyszerűen megtanulhatók. A grafikus ikonok beszédesek, jelentésük könnyen elsajátítható, a magyarra állítható menü barátságos, logikusan felépített.

Program nyelvének beállítása:

Az ablak jobb alsó sarkában lévő kék ikonra kattintva vagy az F12 billentyű segítségével vagy angol nyelvű menü esetén a *Tools – Change Language…* menüpontra (harmadik főmenü első almenüjére) kattintva érhetjük el a *Nyelv módosítása…* ablakot, ahol magyar nyelvűre állíthatjuk a programot.

A diákok nagy örömére a *Stílus módosítása* ikonra kattintva számos színséma közül választhatunk a programban, illetve a „dobozok”, alakzatok formáit is többféle szabvány szerint állíthatjuk be.

A programhoz angol nyelvű súgó, illetve dokumentáció tartozik, a kezdeti lépések megkönnyítése érdekében pedig nyolc lépésből álló tutorial is elérhető a Flowgorithm weboldalán, ami egy *Hello world!* program elkészítését mutatja be. Ugyan ez is angol nyelvű, de a tutorial-ban szereplő képek garantálják a megértést az angol nyelvet nem értők számára is.



1. ábra: A Flowgorithm megnyitáskor

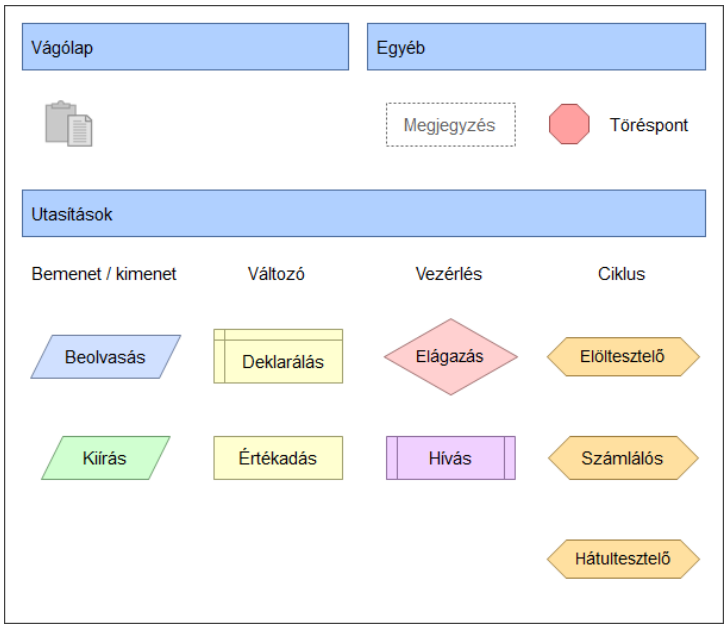
A Flowgorithm megnyitásakor egy üres program algoritmusát látjuk (1. ábra), ami egy **Főprogram** és egy **Vége** dobozból áll.

[1] A program hivatalos weboldala: http://www.flowgorithm.org/
[2] Az anyagban szereplő képernyőképek és feladatmegoldások ezzel a verzióval készültek.

A tutorial-ban szereplő magyarázat szerint – a folyamatábráknál megszokottal ellentétben – itt azért nem Start felirattal indul a folyamatábra, mert a legtöbb programozási nyelvben a Main-nél, vagyis a főprogramnál indul el az algoritmus futtatása. Így ez segíti a tanulók számára a programozási nyelvekre való későbbi átállást.

Az egyes dobozokat összekötő nyilakra kattintva tudunk újabb dobozokat beszúrni. A megjelenő lehetőségek (2. ábra) közül kell kiválasztanunk a számunkra megfelelőt.

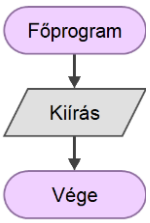
A beszúrható utasítások közül a bemenet/kimenethez tartozó **Beolvasás**, illetve **Kiírás** lehetőségek a legegyszerűbbek. A változókhoz kapcsolódóan két utasítás, a **Deklarálás** és az **Értékadás** közül választhatunk. A vezérléshez az **Elágazás** és a **Hívás** utasítások tartoznak. A ciklusoknál pedig mindhárom ciklus megtalálható, így az **Elöltesztelő**, a **Számlálós** vagy a **Hátultesztelő** utasítást szúrhatjuk be a meglévő algoritmusunkba.



2. ábra: Beszúrási lehetőségek

Az új utasításokhoz tartozó dobozok beszúrása mellett lehetőségünk van korábban vágólapra helyezett algoritmusrészlet beillesztésére is. Az egyéb kategóriába további két elem, a megjegyzés és töréspont került, melyek a tanítási folyamat során rendkívül hasznosak lehetnek.

A beszúrt utasítás először (az alapértelmezett színsémát használva) szürke háttérszínnel jelenik meg az algoritmusunkban. Ilyen módon jelzi számunkra a program, hogy az utasítás még nincs teljesen kész. **Amikor a szürke háttérszínű dobozon duplán kattintunk, akkor jelenik meg az utasításhoz tartozó szerkesztőablak, aminek segítségével működőképessé tehetjük a beszúrt utasítást.**



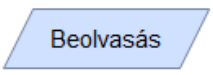
Más színsémák választása esetén nem feltétlenül szürke háttérszínnel, de mindenképp az adott doboz végleges színétől eltérő módon jelzi a program a befejezetlen utasatásokat.

Az utasításokra jobb gombbal kattintva tudjuk azokat kivágni, törölni vagy másolni. Egérhúzással (gumikerettel) több utasítás együttes kijelölésére is van lehetőség.

I. A használható utasításokról

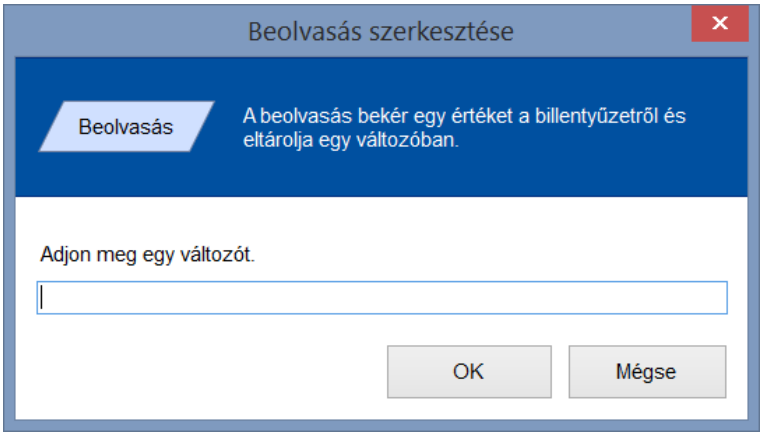
I.1. Bemenet/kimenet

A bemenet/kimenet-hez két utasítás kapcsolódódik, a Beolvasás és a Kiírás.



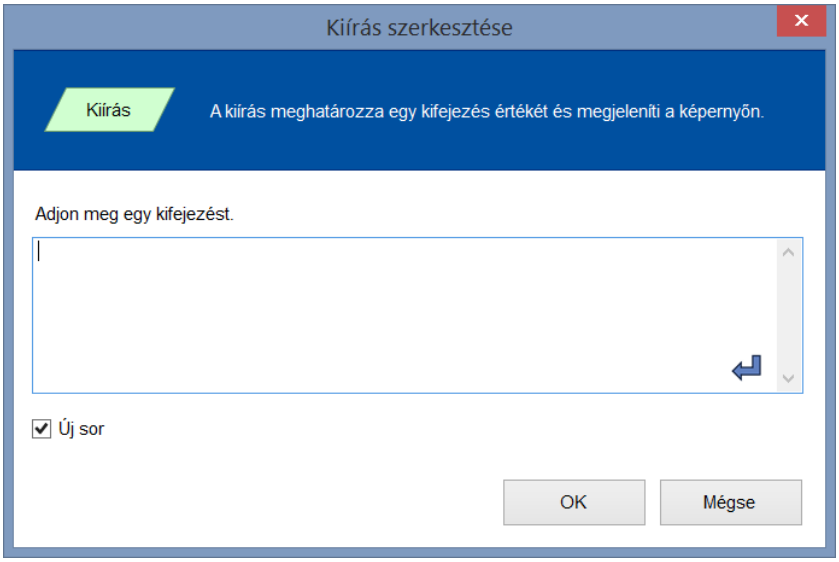
A Beolvasás utasításnál a program bekér egy értéket a felhasználótól és eltárolja azt egy változóban. A beolvasáshoz tartozó szerkesztőablakban (3. ábra) ennek megfelelően csak annak a változónak a nevét kell beírnunk, amiben tárolni szeretnénk a beolvasott adatot.

Fontos, hogy **a változót beolvasás előtt deklarálnunk kell**. Ennek elmulasztása esetén hibaüzenetet kapunk a program futtatásakor. (A hibaüzenetekkel később külön foglalkozunk.)



3. ábra: Beolvasás utasítás szerkesztőablaka

A Kiírás utasítás segítségével üzenetet tudunk kiírni a képernyőre a felhasználónak. Az utasításhoz tartozó szerkesztőablakban (4. ábra) a kiírni kívánt szöveget vagy kifejezést kell beírnunk. A kiírt szövegben természetesen a programban használt változók értékei is szerepelhetnek, emiatt itt több dologról kell említést tennünk.



4. ábra: Kiírás utasítás szerkesztőablaka

A kiírni kívánt üzenetet **idézőjelek közé** kell tennünk. Az idézőjelben lévő szöveg úgy kerül kiírásra a felhasználó számára, ahogyan azt leírtuk. Emiatt az algoritmusban használt változók értékének kiírásához nem elegendő beleírnunk a változó nevét az idézőjelek közé írt szövegbe. Ahhoz, hogy a futtatáskor megjelenő üzenetben a megfelelő változó értéke jelenjen meg, ahhoz a változó nevét idézőjelek nélkül kell szerepeltetnünk a kifejezésben. Az idézőjeleken kívül lévő változó nevének helyére a program futtatásakor kerül behelyettesítésre a változó értéke.

Ha a kiírni kívánt üzenet nem csak szó szerint megjelenítendő szöveg vagy nem csak egyetlen értéke, akkor szükségünk van a különböző szövegrészek összefűzésére. Az **összefűzéshez & jel**et kell használnunk.

> Az idézőjel és az & jel használata jól kapcsolatható a táblázatkezelés témakörhöz. A tanulók az algoritmizálás témakör előtt ott már találkozhattak az idézőjelek és az & ilyen módon történő használatával (pl. az Összefűz függvény & jellel történő kiváltásakor).

A kiírásban szereplő kifejezésben a változókkal műveleteket is végezhetünk. (A használható operátorokról később írunk.)

Példák kiírásban szereplő kifejezésekre:

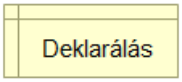
Algoritmusban megadott kifejezés:		Futtatáskor megjelenő szöveg:
`"Helló világ!"`	→	Helló világ!
`"Az eredmény: x."`	→	Az eredmény x.
`x`	→	6 *[Itt az x változó értéke 6.]*
`"Az eredmény: " & x & "."`	→	Az eredmény 6. *[Itt az x változó értéke 6.]*
`"Az eredmény: " & x+1 & "."`	→	Az eredmény 7. *[Itt az x változó értéke 6.]*

A kiírás utasításhoz tartozó szerkesztőablakban még egy dolgot állíthatunk be az **Új sor** jelölőnégyzete vagy a ↵ segítségével. Ha a jelölőnégyzetből kivesszük a pipát vagy a nyílra kattintunk, akkor a ↵ helyett ••• jelenik meg. Ennek akkor van jelentősége, ha egymás után több kiírás utasítást használunk. A ↵ esetén a második üzenet új sorba kerül, míg a ••• esetén a második üzenet az első folytatásaként, azzal egy sorban jelenik meg (szóköz nélkül).

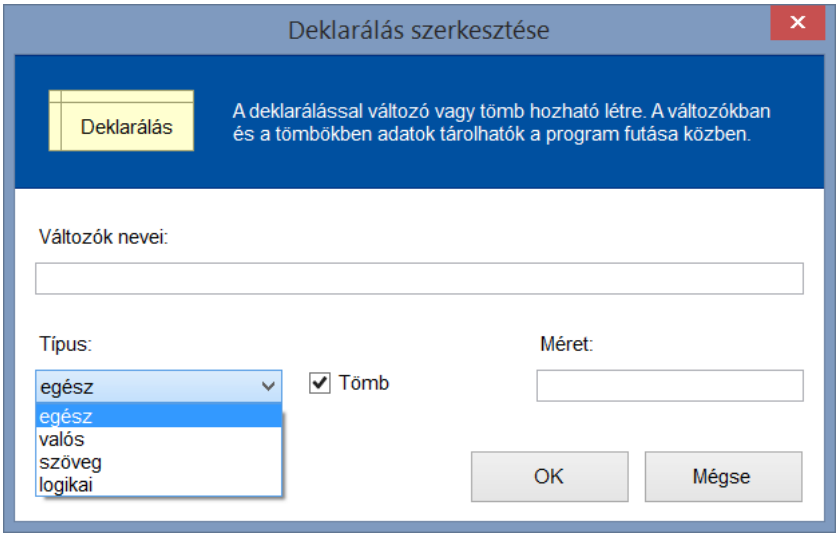
Tanítás során fontosnak tartom megemlíteni a szóközök szerepét. A diákjaimnál gyakori hiba volt, hogy az idézőjeleken belül nem tették ki a szükséges szóközöket, emiatt a futtatáskor megjelenő üzeneteikből számos szóköz hiányzott.

I.2. Változó

A változókhoz kapcsolódóan is két utasítás, a Deklarálás és az Értékadás közül választhatunk.



A deklarálás során az utasításhoz tartozó szerkesztőablakban (5. ábra) a változó nevét kell megadnunk, illetve a típusát kell kiválasztanunk egy legördülő listából. A programban **négyféle típus** közül választhatunk. Ezek az **egész**, a **valós**, a **szöveg**, illetve a **logikai**.

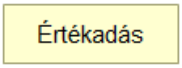


5. ábra: Deklarálás utasítás szerkesztőablaka

A változó neve tartalmazhat kis-és nagybetűket (változónevek esetén nem tesz közöttük különbséget a program), illetve ékezetes karaktereket, de a betűkön és a számokon kívül más karakterek nem engedélyezettek. Az első karakter csak betű lehet.

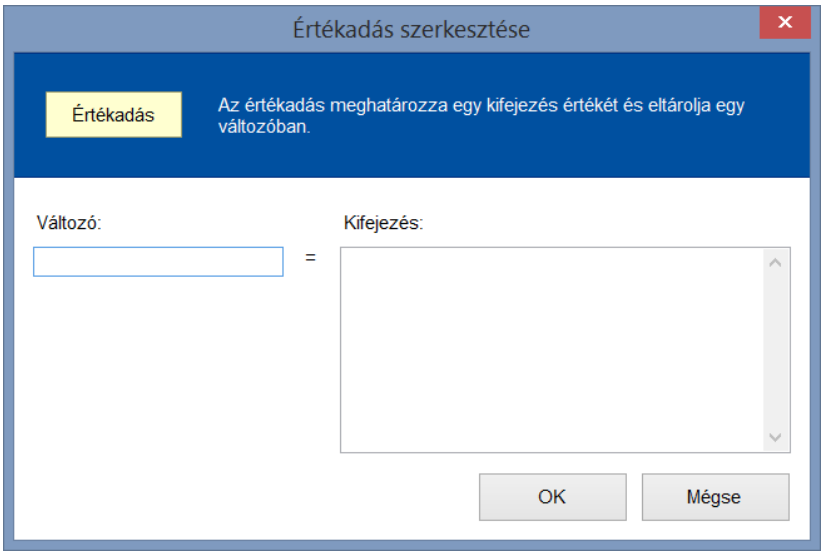
A *Tömb* címkével ellátott jelölőnégyzet segítségével mind a négy típussal tömböt is létrehozhatunk. A jelölőnégyzet bepipálása esetén jelenik meg a *Méret* beviteli mező, ahol a tömb méretét adhatjuk meg. A programozással kapcsolatos elvárásoknak megfelelően a tömb méreténél nem csak konkrét számot, hanem változót, illetve kiszámítandó kifejezést is megadhatunk (pl.: `2*3` vagy `x+3`).

A méretnél megadott változó típusa egész vagy valós változó lehet. Pozitív valós változó esetén értékének alsó egészrésze lesz a tömb mérete. Ha a méretben megadott érték típusa nem megfelelő vagy a szám vagy annak alsó egészrésze nem pozitív, akkor futtatáskor kapunk hibaüzenetet.



Az Értékadás utasítás esetén a változó nevét, illetve értékét kell megadnunk a megjelenő szerkesztőablakban (6. ábra). A változó értéke sokféle lehet, mert konkrét értékek mellett a tömb méretéhez hasonló módon itt is megadhatók kiszámítandó kifejezések is.

A **logikai változók értéke `true`, `false`** vagy valamilyen logikai kifejezés lehet.
A tizedestörtek esetén a tizedesvessző helyére pontot kell írnunk.
A **tömb elemeinek csak külön-külön adhatunk értéket**, nem állítható például 0-ra egy utasítással egyszerre a (legalább két elemből álló) tömb összes eleme. **A tömb `i`-edik elemére `tömb[i]` formában hivatkozhatunk** úgy, hogy **a tömb első eleme a 0 indexet kapja**.



6. ábra: Értékadás utasítás szerkesztőablaka

Az értékadási lehetőségek kapcsán a tanulók számára az `x=x+1` típusú értékadások bizonyultak a legnehezebbnek főként azon diákok számára, akik a Flowgorithm előtt nem találkoztak semmiféle programozási környezettel. Számukra matematikai ismereteik alapján nyilvánvaló, hogy a két oldal nem lehet egyenlő. El kell magyarázni nekik, hogy a bal oldalon a változó új értéke szerepel, míg a jobb oldalon a változó korábbi értéke kerül behelyettesítésre.

Fontos, hogy **a változót az értékadás előtt minden esetben deklarálnunk kell**. (Ezalól még a számlálós ciklus ciklusváltozója sem lesz kivétel.)

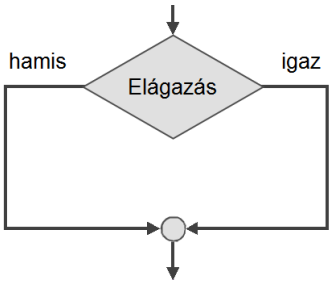
Ebben a környezetben a deklarálás és az értékadás utasítások nem vonhatók össze.

I.3. Vezérlés

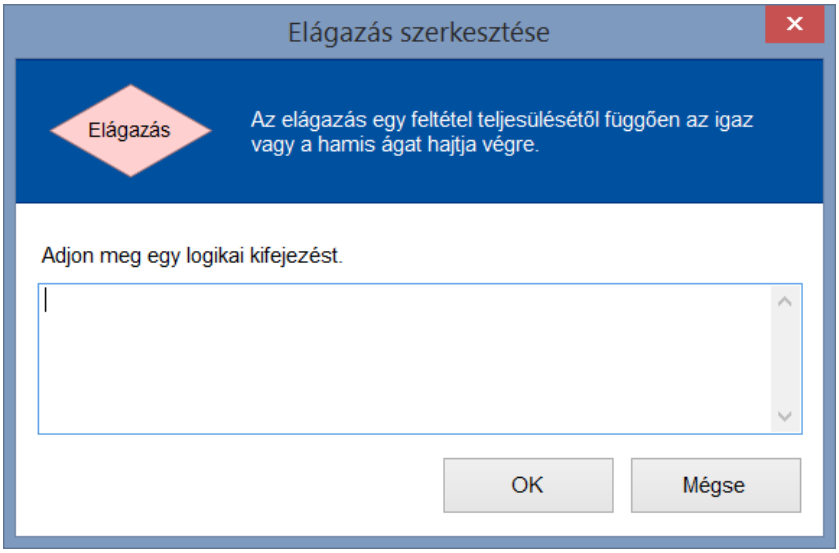
A vezérléshez is két utasítás került, méghozzá az Elágazás, illetve a Hívás.

Az elágazásnál **egy logikai kifejezést kell megadnunk** az utasításhoz tartozó szerkesztőablakban (7. ábra). A kifejezésben használható operátorokról később részletesen írunk. A logikai kifejezés kiértékelése után az eredménytől függően **az igaz vagy a hamis ágon lévő utasításokat hajtja végre a program**.

Az algoritmus alapján könnyen kikövetkeztethető, de nem árt hangsúlyozni és tudatosítani a diákokban, hogy egy futtatás során a két ág közül csak az egyiken lévő utasítások kerülnek végrehajtásra.



Az algoritmusban kezdők számára is könnyen értelmezhetően jelenik meg mindez. Minkét ágnál feltüntetésre kerül a megfelelő felirat, az igaz, illetve a hamis. Az elágazás végét pedig egy kis körrel jelzik. Ebbe futnak be az igaz és hamis ágakhoz tartozó nyilak, illetve ebből a körből indul el lefelé a nyíl az algoritmus további utasításaihoz.

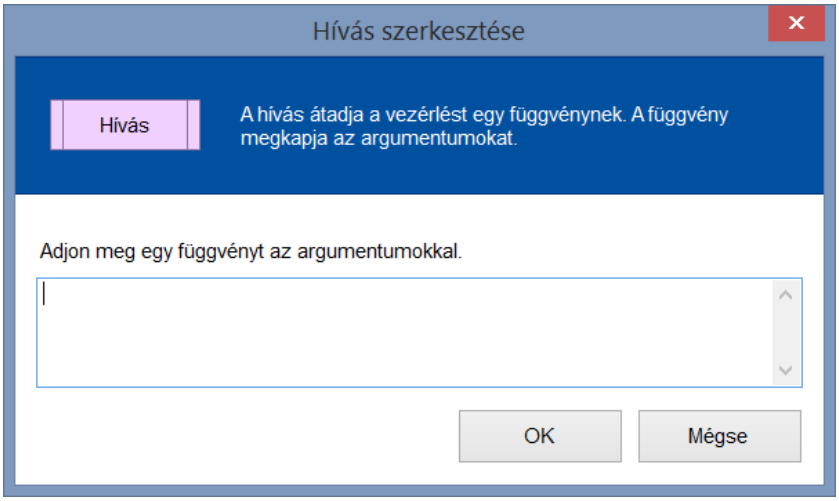


7. ábra: Elágazás utasítás szerkesztőablaka

A diákoknál a logikai kifejezéssel kapcsolatosan fontosnak tartom kiemelni azt, hogy egy logikai típusú változó önmagában is betöltheti a logikai kifejezés szerepét.

A Hívás utasítás segítségével saját korábban létrehozott függvényeinket hívhatjuk meg. Az utasításhoz tartozó szerkesztőablakban (8. ábra) **a függvény nevét és a függvény argumentumait kell megadnunk**. (A függvények létrehozásának módjáról később részletesen írunk.)

A függvény neve után a függvény argumentumait gömbölyű zárójelben, vesszőkkel elválasztva kell felsorolnunk. Például így: `lnko(a,b)`.



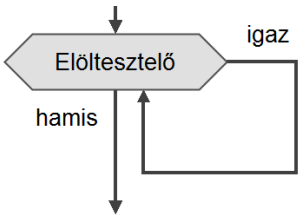
8. ábra: Hívás utasítás szerkesztőablaka

I.4. Ciklusok

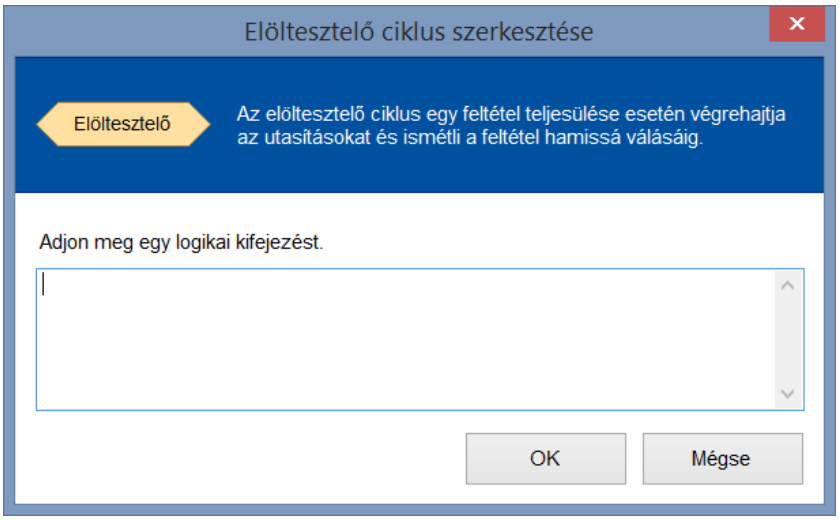
A ciklusokhoz kapcsolódóan három utasítás érhető el. A számlálós, az elöltesztelő és a hátultesztelő ciklushoz is egy-egy utasítás tartozik.

Elöltesztelő

Az elöltesztelő ciklusnál az utasításhoz tartozó szerkesztőablakban (9. ábra) **egy logikai kifejezést kell megadnunk. Ha a megadott logikai kifejezés értéke igaz, akkor hajtja végre a program a ciklusmagban, vagyis az igaz ágon szereplő utasításokat. Hamis érték esetén** pedig a lefelé mutató nyíl irányába fut tovább a program, vagyis **a ciklus alatt lévő utasítás hajtódik végre.**



Az elágazáshoz hasonló módon itt is látványosan, a kezdők számára is könnyen megérthető módon ábrázolták ezt. Az igaz ághoz tartozó nyíl a ciklus feltételére mutat vissza, így érzékeltetve azt, hogy **az igaz ág végén ismét kiértékelésre kerül a logikai kifejezés**.



9. ábra: Elöltesztelő ciklus utasításának szerkesztőablaka

Érdemes szót ejteni a tanítási folyamat során arról, hogy **előfordulhat olyan eset, amikor az elöltesztelő ciklus magjában lévő utasítások egyszer sem kerülnek végrehajtásra és olyan is, amikor az ott lévő utasítások többször is végrehajtásra kerülnek**.

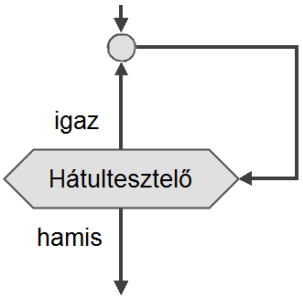
Előfordulhat, hogy a diákok számára némi magyarázatot igényel az is, hogy mitől változhat meg a logikai kifejezés értéke az igaz ág lefutása után, mert hajlamosak lehetnek azt gondolni, hogy ha a logikai kifejezés először igaz volt, akkor később is az lesz. Egy egyszerű példa segítségével viszont könnyen tisztázhatjuk ezt. (Pl.: Amíg egy változó értéke 100-nál kisebb, addig kétszeresére növeljük az értékét.)

Hátultesztelő

A hátultesztelő ciklus az elöltesztelőhöz hasonlóan működik. Itt is **egy logikai kifejezést kell megadnunk** az utasításhoz tartozó szerkesztőablakban (10. ábra).

Az utasítás jelölése a folyamatábrában itt is logikus, viszont mindenképpen szokniuk, értelmezniük kell azoknak, akik csak a Flowgorithm használata segítségével tanulják meg, hogy mi az a hátultesztelő ciklus.

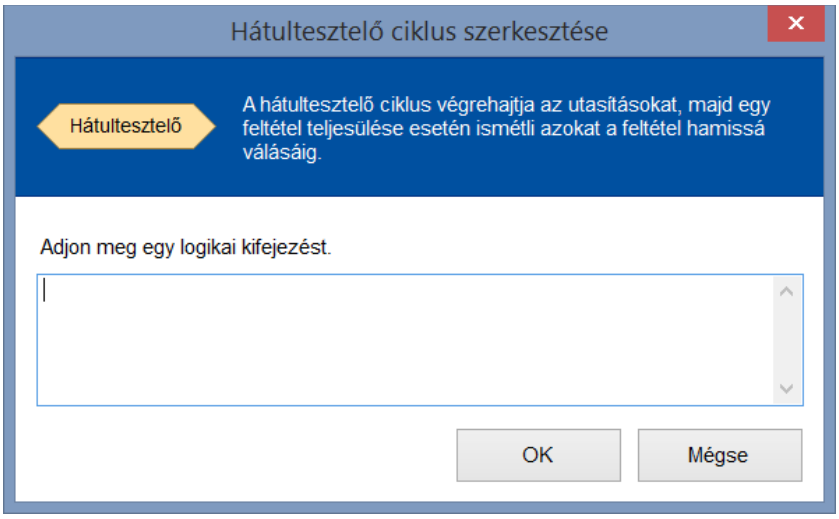
A ciklus kezdetét egy kis körrel jelzik, erre a körre mutat az hátultesztelő ciklus előtti utasításból érkező nyíl. A körből csak a hátultesztelő ciklus ciklusmagjában lévő utasítások felé mehetünk tovább. Ezek után a megadott feltételhez érünk, és a logikai kifejezés igazságértékétől függ az, hogy felfelé vagy lefelé megy tovább a program futtatása.



Ha a megadott feltétel igaz, akkor felfelé megyünk tovább és visszajutunk a hátultesztelő ciklus kezdetét jelölő körhöz, ahonnan **ismét csak a ciklusmag utasításai felé mehetünk tovább, így azok ismét végrehajtásra kerülnek. Ha a feltétel hamis, az a hátultesztelő ciklus végét jelenti**, és a lefelé, a következő utasítás felé fut tovább a program.

Fontosnak tartom megemlíteni azt, hogy a hátultesztelő ciklusnál **a ciklusmagban lévő utasítások legalább egyszer biztosan végrehajtásra kerülnek**.

A hátultesztelő ciklussal kapcsolatosan a legnagyobb nehézséget annak tudatosítása okozta a diákjaimnál, hogy itt is akkor hajtódnak végre ismét az utasítások, ha a megadott feltétel igaz, vagyis a ciklusban való bennmaradás feltételét kell megadniuk.



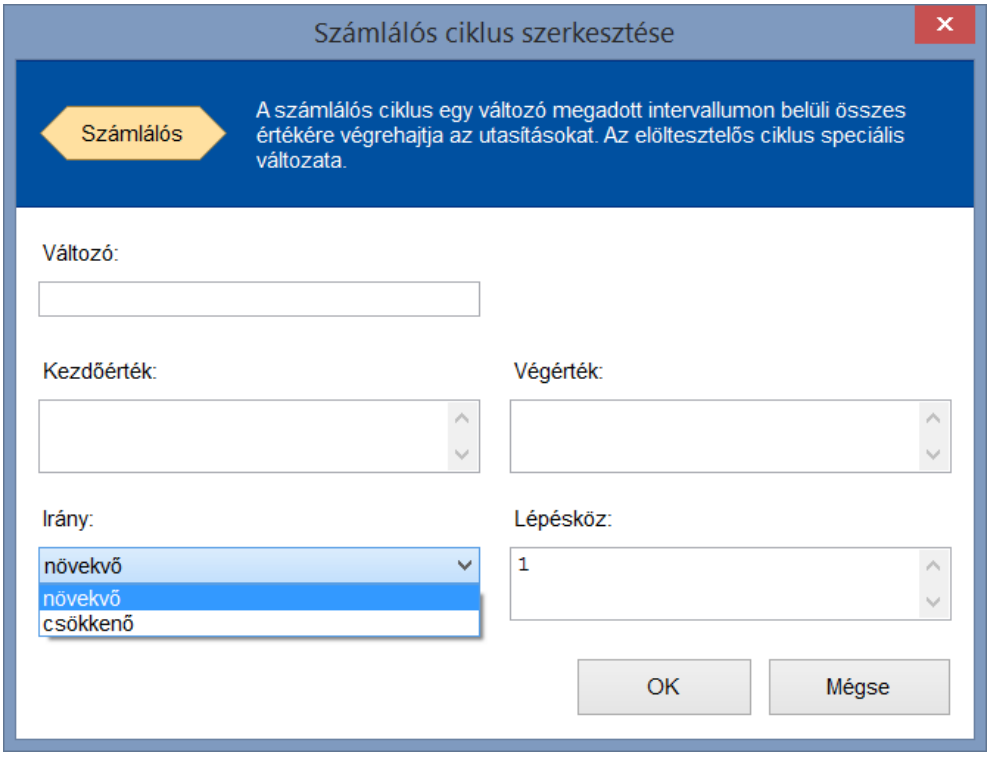
10. ábra: Hátultesztelő ciklus utasításának szerkesztőablaka

A hátultesztelő ciklusnál **a feltételből induló, felfelé mutató nyílra**, vagyis az igaz ágra az algoritmusban található többi nyíltól eltérő módon **nem szúrhatunk be további utasításokat**. Ennek magyarázata az, hogy később, a „kódolós programozás" során sem lesz ilyen lehetőség. Ez az oka annak, hogy ennek a nyílnak nem változik meg a színe, ha fölévisszük az egeret.

Számlálós

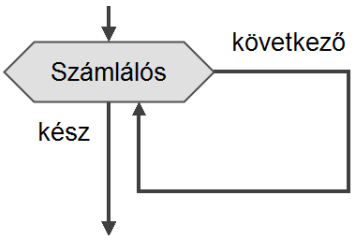
A számlálós ciklus esetén az eddigieknél jóval több dolgot kell megadnunk az utasításhoz tartozó szerkesztőablakban (11. ábra), de itt is igaz az, hogy mindezt könnyen értelmezhető formában kell megtennünk. **Meg kell adnunk egy változót, ami a ciklusváltozó** szerepét tölti be. Be kell írnunk **egy kezdő és egy végértéket**, ami lehet szám vagy valamilyen szám értékű kifejezés is. Ezek mellett **egy legördülő listából kell kiválasztanunk azt, hogy növekvő vagy csökkenő irányú-e a ciklus, illetve azt, hogy milyen lépésközzel haladjon a program** a kezdőértéktől a végértékig.

A kezdőértéknél, végértéknél, illetve lépésköznél is megadható tizedestört is. Ekkor természetesen figyelni kell a ciklusváltozó típusára is.



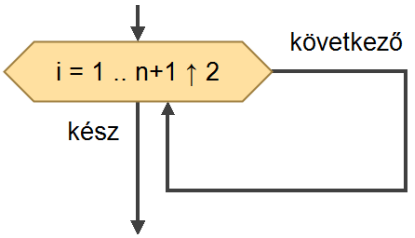
11. ábra: Számlálós ciklus utasításának szerkesztőablaka

Az utasítás beszúrásakor megjelenő, még kitöltetlen számlálós ciklus kinézete az elöltesztelő ciklusra emlékeztethet bennünket. Annyi a különbség, hogy itt az igaz és a hamis feliratok helyett következő és kész feliratok vannak.



A következő feliratú nyíl az, ami tartalmazza a ciklusmag utasításait és a számlálós ciklus elejére mutat vissza. Így a következő szó a következő ciklusléptés végrehajtását jelzi. **A kész feliratú nyíl pedig a számlálós ciklus végét jelenti** és a ciklus követő utasításra mutat.

A kitöltött számlálós ciklusnál minden információ megjelenik, amit a szerkesztőablakban megadtunk. A képen látható `i=1..n+1 ↑ 2` esetén az `i` a ciklusváltozó. A megadott kezdőérték `1`, míg a végérték `n+1` (az `n` egy változó). A **felfelé mutató nyíl a növekvő irányt jelzi, az utána lévő szám pedig a lépésközt jelöli**, így a példában 2 a lépésköz. **Csökkenő irány esetén** értelemszerűen **lefelé mutató nyilat látnánk**.



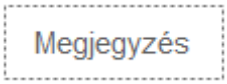
A tanítási folyamat során érdemes a számlálós ciklus végrehajtását lépésenkénti futtatással, a változók figyelésével végigkövetni. Fontos tisztázni, hogy ciklusmagban lévő utasítások végrehajtásra kerülnek akkor is, ha a ciklusváltozó értéke megegyezik a kezdőértékkel vagy a végértékkel. Emellett érdemes olyan esetet is bemutatni, amikor a ciklusváltozó értéke a futás során nem lesz egyenlő a végértékkel, csupán átlépi azt, illetve olyat is, amikor a ciklusmagban lévő utasítások egyszer sem kerülnek végrehajtásra.

Példák, amiket érdemes kipróbálni, végiggondolni a számlálós ciklusnál:

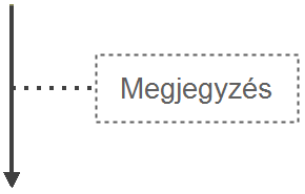
kezdőérték	végérték	irány	lépésköz	kiírások száma*	
1	6	növekvő	1	6	
1	6	növekvő	2	3	(amikor a ciklusváltozó értéke 1, 3, 5)
-3	6	növekvő	3	4	(amikor a ciklusváltozó értéke -3, 0, 3, 6)
1	1	növekvő	1	1	
1	6	csökkenő	1	0	
-3	6	csökkenő	3	0	
1	6	növekvő	0.5	∞	(ha a ciklusváltozó típusa egész, mert értéke végig 1 marad)
				11	(ha a ciklusváltozó típusa valós)
0.5	3	növekvő	1	4	(amikor a ciklusváltozó értéke 0, 1, 2, 3, ha a ciklusváltozó típusa egész)
				3	(amikor a ciklusváltozó értéke 0.5, 1.5, 2.5, ha a ciklusváltozó típusa valós)
0.5	3	növekvő	0.5	∞	(ha a ciklusváltozó típusa egész, mert értéke végig 0 marad)
				6	(ha a ciklusváltozó típusa valós)

*A ciklusmagban minden esetben egyetlen kiírás utasítás van.

I.5. Megjegyzés



A programban használható utasítások mellett említést kell még tennünk a megjegyzésről, ami az utasításokhoz hasonló módon jelenik meg az algoritmusunkban. **Megjegyzést az utasításokhoz hasonlóan bármelyik nyílra beszúrhatunk.** A megjegyzés szaggatott vonallal kapcsolódik az algoritmus többi részéhez.



A megjegyzések szövegében nem kell idézőjeleket használnunk, bármilyen szöveget beírhatunk, de itt a változók értékének megjelenítésére nincs lehetőség.

Hasznos lehet például magyarázatok, fontos tudnivalók rögzítéséhez, amelyek segítségével a tanulók később is fel tudják idézni vagy hiányzás esetén megérthetik a program készítésének folyamatát.

II. A logikai és egyéb kifejezésekben használható operátorok

A Flowgorithm-ben használható operátorok kiválasztásakor arra törekedett a program készítője, hogy azok a lehető legtöbb programozási nyelvhez illeszkedjenek. Emiatt **vannak olyan operátorok, amelyeknek többféle formáját** (a BASIC-típusú és a C-típusú nyelveknél használt megfelelőjét is) **használhatjuk**.

A később tanítani kívánt programozási nyelv alapján dönthetünk úgy, hogy a több lehetőség közül mindig csak azt tanítjuk, amelyeket a diákoknak később is használniuk kell majd. Illetve dönthetünk úgy is, hogy érdeklődő diákoknál vagy leendő programozók esetén mindegyik lehetőségről említést teszünk.

A programban használható operátorokat tartalmazó táblázatban (1. táblázat) megjelöltük, hogy melyik operátor melyik nyelvcsaládhoz tartozik, de **nyelvcsaládtól függetlenül vegyesen is használhatjuk őket**.

Operátor	C-típusú	BASIC-típusú	Visual Basic
tagadás	`!`	`not`	
modulo (osztási maradék)	`%`	`mod`	
egyenlőség	`==`	`=`	
nem egyenlőség	`!=`	`<>`	
és	`&&`	`and`	
vagy	`||`	`or`	
szövegek összefűzése			`&`
hatványozás			`^`

1. táblázat: Flowgorithm-ben használható operátorok

A Java és a C# a + operátort használja a szövegek összefűzésére, de a Flowgorithm-ben nehézséget okozott volna az, hogy a + kétféle műveletet jelenthet, ezért itt a + csak számoknál használható.

Az operátorokkal kapcsolatosan fontos tudnunk a kiértékelésük sorrendjét is. **A 2. táblázatban látható az operátorok végrehajtási sorrendje. A magasabb szinten lévő műveletek kerülnek előbb elvégzésre.** Az azonos szinten lévő műveletek esetén balról jobbra történik meg a kiértékelésük.

szint	megnevezés	műveletek
8.	egyváltozós (unáris) műveletek	`- ! not`
7.	hatványozás	`^`
6.	szorzás, osztás	`* / % mod`
5.	összeadás, kivonás	`+ -`
4.	összefűzés	`&`
3.	relációk	`> >= < <= == = != <>`
2.	és	`and &&`
1.	vagy	`or ||`

2. táblázat: Műveletek végrehajtási sorrendje

A műveletek **végrehajtásának sorrendjét** a matematikában megszokott módon itt is **zárójelek segítségével tudjuk befolyásolni**.

Példák az operátorok használatára, kiértékelésük sorrendjére

kifejezés	eredmény	megjegyzés
`1+3^2`	10	A hatványozás az első, az összeadás a másodikként elvégzett.
`7*4-1`	27	A szorzás a magasabb szintű, így azt végzi el előbb.
`7*(4-1)`	21	Először mindig a zárójelben lévő teljes kifejezés kerül kiszámításra, a műveleti sorrend csak ez után érvényesül.
`10 mod 3`	1	A 10 esetén a 3-mal való osztási maradék 1.
`3>5 && 7<=9`	false	Először a két reláció kerül kiértékelésre balról jobbra. Az első *hamis*, a második *igaz*. A *hamis* `&&` *igaz* értéke pedig *hamis*.
`!(3>5)`	true	A zárójelben lévő kifejezés *hamis*, a tagadás viszont az ellenkezőjére változtatja a kifejezés igazságértékét, így az eredmény végül *igaz* lesz.

Majdnem minden csoportomban előfordult olyan, hogy a tanulók ehhez hasonló feltételt szerettek volna megadni: `3<x<8`, mert matematika órán is találkoztak ilyennel, ott ez elfogadott jelölés. Emiatt fontosnak tartom megemlíteni, hogy az itt használható nem egyváltozós műveletek kétváltozósak, vagyis a `3<x<8` kifejezés `3<x && x<8` vagy `3<x and x<8` formában adható csak meg a programban.

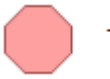
A `true` és a `false` mellett a `pi` használható még konstansként a programban.

III. Az algoritmus futtatása

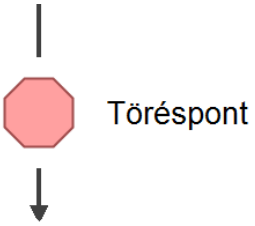
Az elkészített algoritmust egészében vagy lépésenként futtathatjuk. A menüszalagon lévő, **lejátszást szimbolizáló háromszöggel futtathatjuk le egészében a teljes programot**, míg **a mellette lévő ikonnal tudjuk lépésről lépésre végrehajtani az algoritmust**. **Lépésenkénti végrehajtás esetén mindig kijelölésre kerül a következő végrehajtandó utasítás az algoritmusban**, így könnyen nyomon lehet követni a történéseket. Az egyes lépések mindig újabb és újabb kattintás hatására hajtónak végre, így tetszőleges ideig elemezhetjük vagy magyarázhatjuk azokat a tanulók számára.

Lépésenkénti végrehajtás közben bármikor dönthetünk úgy, hogy az algoritmus további részét nem lépésenként, hanem egyben szeretnénk lefuttatni. Ekkor a lépésenkénti futtatás ikon helyett a futtatás ikonra kell kattintanunk.

Hosszabb algoritmusok esetén azonban hosszadalmas lenne lépésenkénti futtatással eljutni az algoritmus közepén vagy végén lévő, elemezni kívánt részhez. Ezt könnyíti meg a töréspont használata, amit az utasításokhoz hasonló módon szúrhatunk be a megfelelő helyre, az elemezni kívánt rész elé.

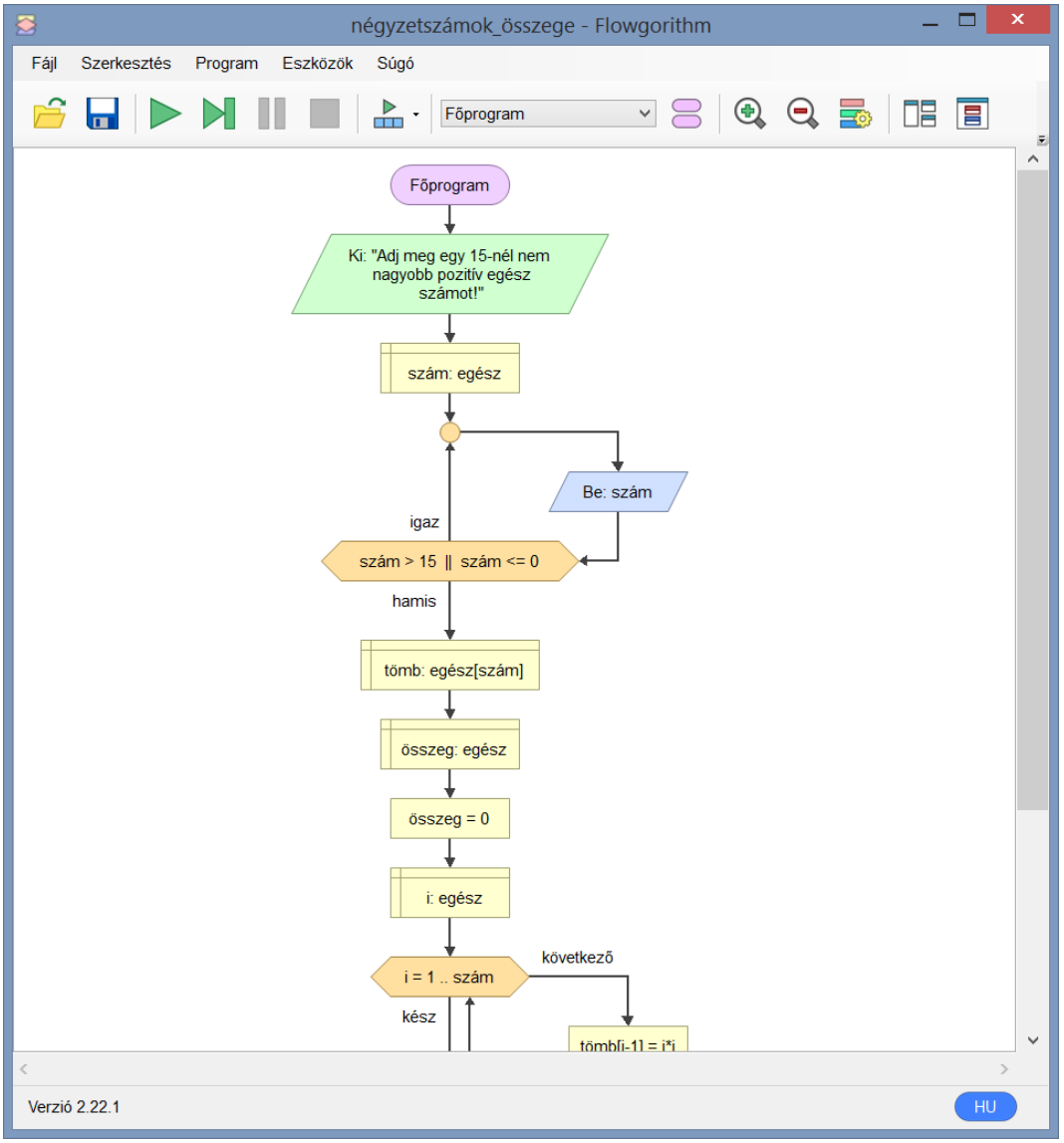


Ha az algoritmus valamely pontjára töréspontot helyezünk el, akkor ott a program megáll nem lépésenkénti futtatás közben is. Így meg tudjuk nézni, hogy mi történt addig, megnézhetjük a változók aktuális értékeit. Emellett eldönthetjük azt, hogy lépésenként vagy egyben szeretnénk tovább futtatni az algoritmust. Több töréspont esetén mindig csak a következő töréspontig hajtódik végre a program, ha a töréspont elérése után a futtatás lehetőséget választjuk. A töréspontokat, illetve funkciójukat **az algoritmusban a nyíl töréspontnál való megszakításával jelzik** számunkra.



IV. Ablakok, nézetek

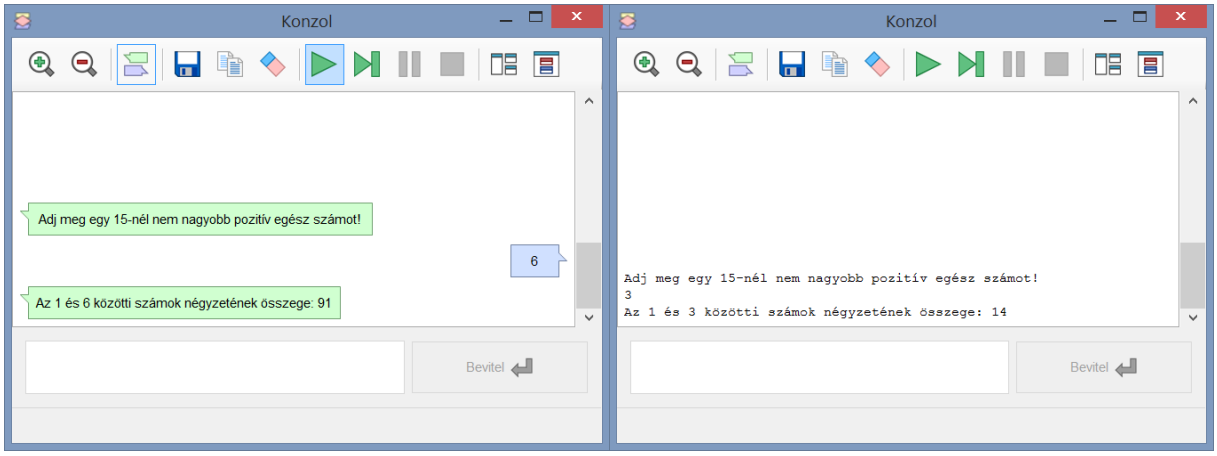
Az algoritmust tartalmazó fő ablak (12. ábra) mellé további ablakokat is megnyithatunk, amelyek nagy segítséget jelentenek az algoritmus elkészítse, illetve a tanítási folyamat során.



12. ábra: Az algoritmust tartalmazó fő ablak

Az egyik legfontosabb ablak a **Konzol**. **Ebben az ablakban adhatjuk meg a billentyűzetről bekért változókat, és itt jelennek meg a kiírás utasításokba írt üzeneteink.** Ez az ablak az algoritmus futtatásakor automatikusan kinyílik, de ahogyan a később említésre ketülő ablakok, úgy elérhető ez is az *Eszközök* menüből.

A konzolon alapértelmezett beállítás szerint a **Kiírás és a Beolvasás utasítások színeivel azonos színezéssel, párbeszédes formában, szövegbuborékokban** jelennek meg a képernyőre kiírt szövegek, illetve a felhasználó által futás közben megadott értékek épp úgy, ahogyan ezt számos „beszélgetős alkalmazás” esetén megszokhattuk (13. ábra). Ez a párbeszédes megjelenés kikapcsolható az ablak harmadik ikonja segítségével, de úgy gondolom, hogy jó, hogy ilyen módon könnyen megkülönböztethetővé válik a bemenet és a kimenet.



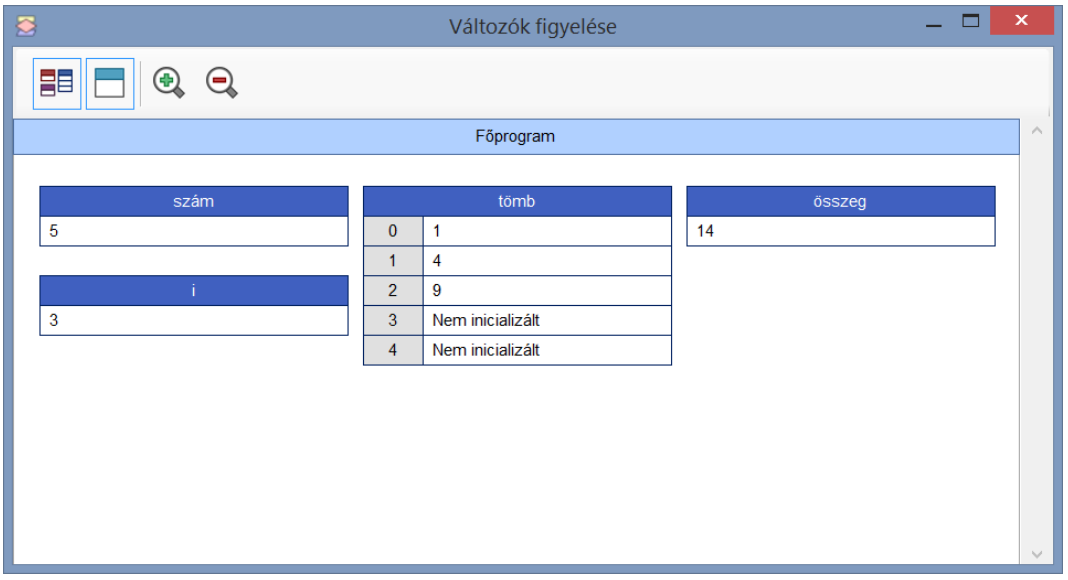
13. ábra: Konzol ablak szövegbuborékokkal és anélkül

Ha a futtatás során a beolvasáskor begépelt változó típusa nem megfelelő, akkor nem fut tovább a program, a Konzol ablak alján, az állapotsoron kapunk üzenetet arról, hogy új értéket kell megadnunk.

A **Változók figyelése ablak** (14. ábra) egy másik nagy segítség a tanítási folyamat során. Ennek köszönhetően az egész program során **végigkövethetjük minden változó aktuális értékét a deklarálásuktól a futtatás végéig**.

Az algoritmus lépésenkénti futtatásával kombinálva ez az ablak sokat segít a diákoknak az egyes utasítások értelmezésében. Előnyös az is, hogy a változókat típusonként eltérő színnel jelölik.

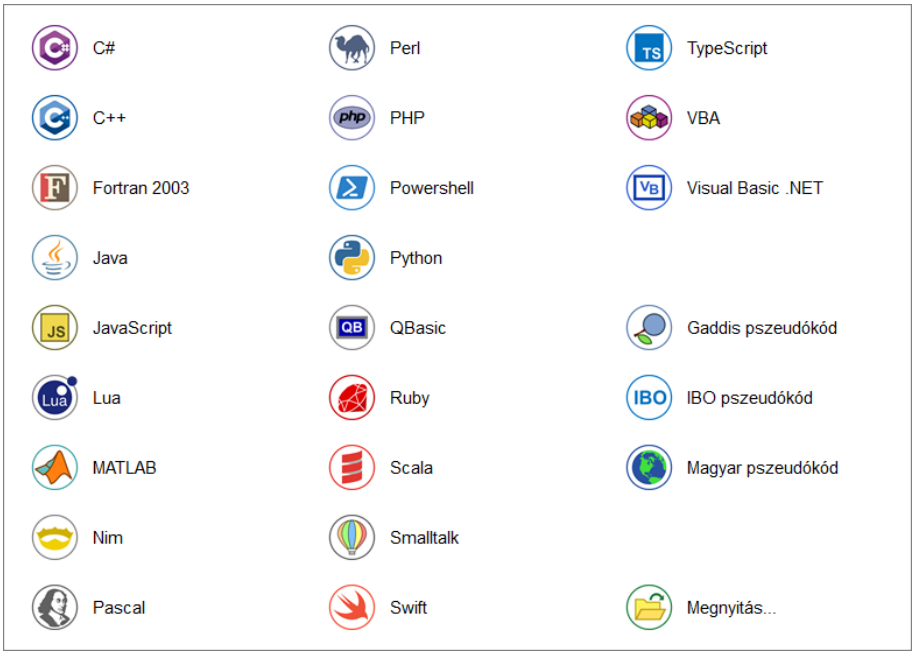
A tömbök táblázatos formában jelennek meg és jól látható, hogy a tömb első eleme a 0 indexet kapja. A deklarált de érték nélküli változóknál a *Nem inicializált* szöveg jelenik meg.



14. ábra: Változók figyelése ablak futtatás közben

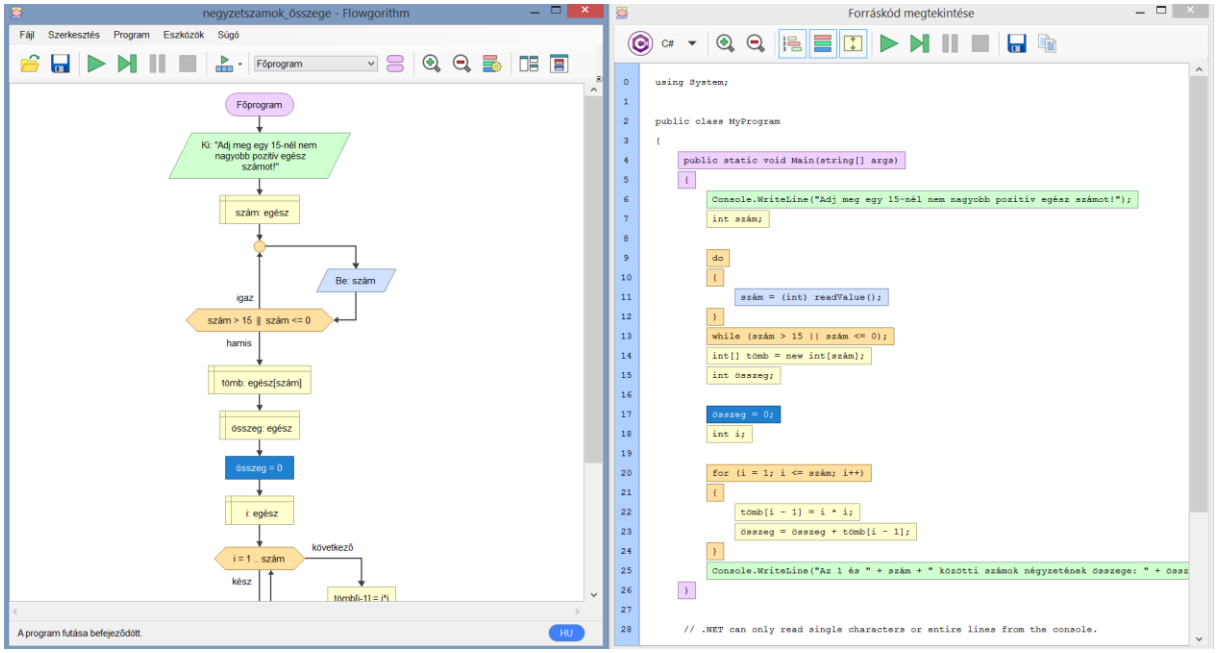
A változók figyelése ablaknál **a változók láthatóságára is figyeltek**, így például egy függvény hívása esetén csak a függvényhez tartozó változók jelennek meg itt, amikor a függvénynél tart az algoritmus futtatása. Ilyenkor az ábrán látható főprogram felirat a megfelelő függvény nevére cserélődik.

A negyedik, a **Forráskód megtekintése** ablak a programozás irányába történő továbbhaladás miatt fontos. Itt számos programozási nyelven (15. ábra) megtekinthetjük a létrehozott algoritmusunkat.



15. ábra: A 2.22.1-es verzióban elérhető programozási nyelvek listája

A megjelenő forráskód színezése az algoritmuséval megegyező. Ha az algoritmusban kijelölünk egy utasítást, akkor a hozzá tartozó kódrészlet a forráskódban is kijelölésre kerül (16. ábra). Ugyez történik lépésenkénti futtatáskor a következőként végrehajtandó utasítás forráskódja esetén is.



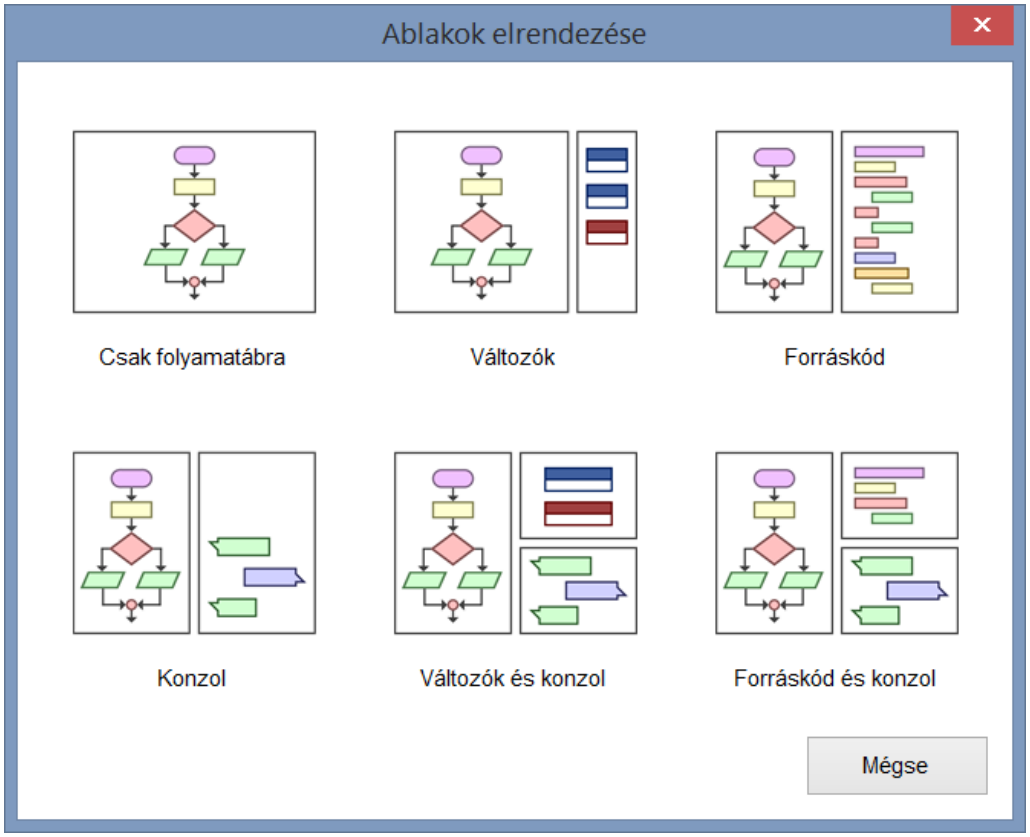
16. ábra: Az algoritmus és a hozzá tartozó forráskód

A forráskód különböző programozási nyelveken történő megjelenítése és az utasítások kiemelése nagy segítség a diákoknak egy-egy programozási nyelv utasításainak és szintaxisának elsajátításában.

Az ablakban megjelenő **forráskód egésze vágólapra másolható**, illetve **az adott programozási nyelvnek megfelelő formátumban** (pl.: *cs*, *cpp*, *pas*, *py*) **menthető**.

A különböző ablakok gyors, átlátható elrendezése érdekében hat beépített lehetőség közül választhatunk (17. ábra).

Az algoritmizálás témakör bevezetésekor a különböző utasítások tanításakor, a *Változók és konzol* elrendezést szoktam használni. A forráskód ablakot csak később mutatom meg a diákoknak. (Persze a programozás után érdeklődő tanulók korábban is felfedezik ezt a lehetőséget.) Idősebbeknél a programozási nyelvekre való áttéréskor már leginkább a *Forráskód* elrendezést használom az órákon.



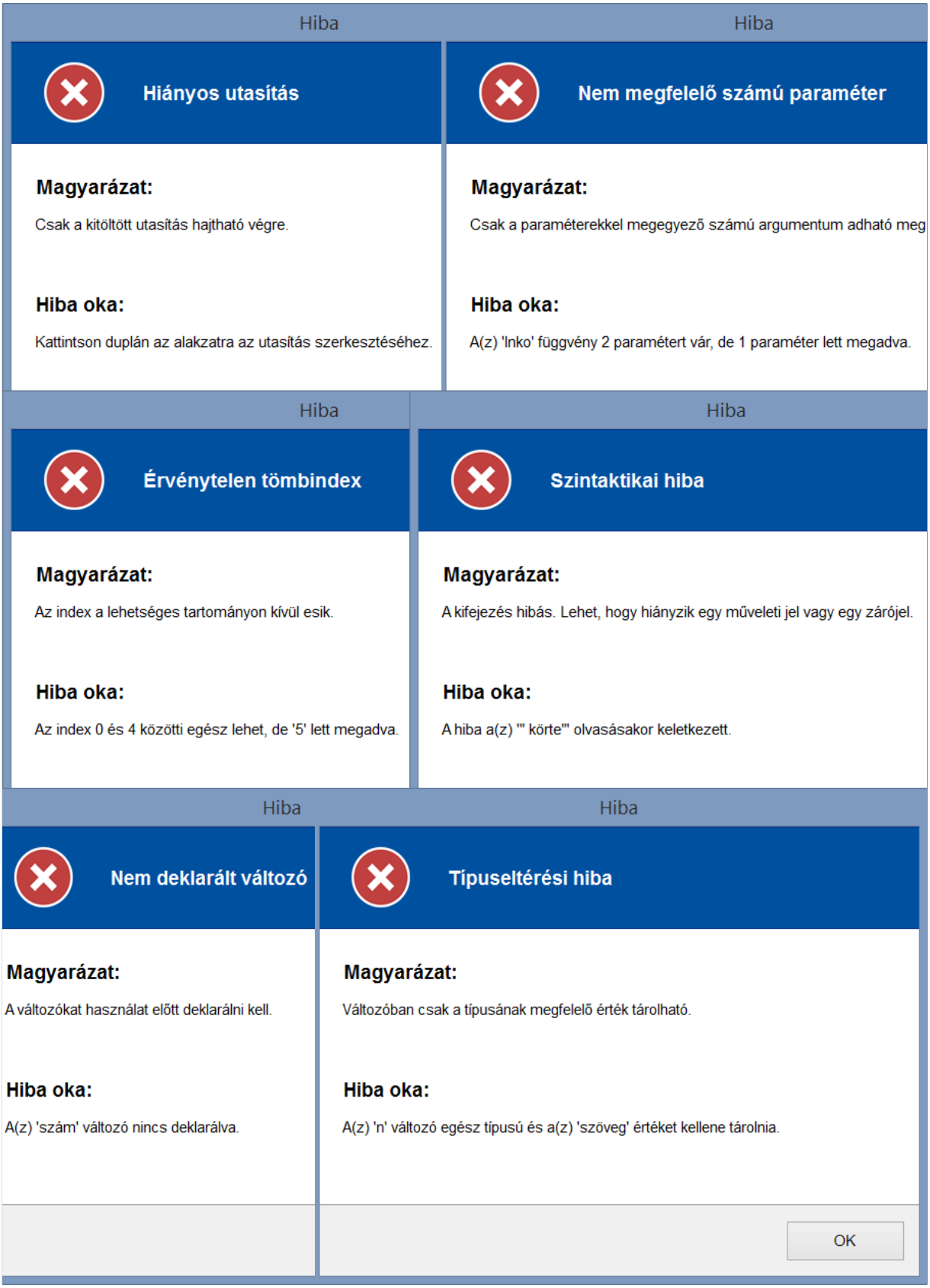
17. ábra: Az ablakok beépített elrendezési lehetőségei

V. Hibaüzenetek

Az algoritmus szerkesztése, illetve a program futtatása során számos hibaüzenettel találkozhatunk. Sok más környezettel szemben a Flowgorithm nagy előnye, hogy a hibaüzenetek is magyar nyelvűek és könnyen érthetőek. Nem tudunk minden lehetséges hibaüzenetet megmutatni, de remélhetőleg az itt lévő néhány példa (18. ábra) segít annak elképzelésében, hogy milyen lehet a többi.

A **hiba helyét mindig piros háttérszín jelzi** az algoritmusban.

Ezek a hibaüzenetek hozzájárulnak a tanítási folyamat sikeréhez. Fontos, hogy a diákok ne ijedjenek meg a hibaüzenetek láttán, hanem olvassák el figyelmesen, és próbálják meg önállóan megoldani a felmerülő problémát. Ez pedig sokkal könnyebb így, mintha angol nyelvű vagy különböző hibakódokat tartalmazó üzenetek jelennének meg.
A hibaüzenetektől való félelem legyőzése érdekében előfordul, hogy a csoport gondolatmenetét követve hibás megoldást gépelek be és vetítek ki az órákon, hogy utána közösen értelmezzük a megjelenő hibaüzenetet. Tapasztalataim szerint az első néhány hibaüzenet után a legtöbb, másolásból vagy figyelmetlenségből keletkező, többnyire gépelési hibát egyedül is meg tudják oldani a tanulók.



18. ábra: Példák a Flowgorithm hibaüzeneteire

VI. Függvények használata

A Flowgorithm-ben a korábban ismertetett utasítások mellett **beépített függvényeket is használhatunk, illetve saját függvényeket is létrehozhatunk**.

VI.1. Beépített függvények

Az előre definiált függvények között **főként matematikai, szövegekkel, illetve típuskonverzióval kapcsolatos függvények**et találunk.

A jól ismert **matematikai függvények közül a 3. táblázatban lévőket használhatjuk** az algoritmusainkban. A matematikai **függvények argumentumában szám típusú értékeket kell megadnunk**.

függvény	leírás
`abs(n)`	abszolútérték
`sgn(n)`	előjel (értéke negatív számoknál `-1`, 0-nál `0`, pozitív számoknál `1`)
`int(n)`	egészrész (pl.: `int(2.75)=2`, `int(-3,2)=-3`)
`sqrt(n)`	négyzetgyök
`sin(n)`	szögfüggvények
`cos(n)`	
`tan(n)`	
`arcsin(n)`	
`arccos(n)`	
`arctan(n)`	
`log(n)`	természetes alapú logaritmus (`e` alapú)
`log10(n)`	10-es alapú logaritmus

3. táblázat: Beépített matematikai függvények

Szövegekhez kapcsolódóan a 4. táblázatban látható két függvényt használhatjuk. A függvények argumentumainál az `s` helyére szöveget vagy szöveg típusú változót írhatunk, az `i` helyére pedig nemnegatív számot. (Ha a megadott szám valós, akkor annak egészrészével dolgozik a program.)

függvény	leírás
`len(s)`	Az `s` szöveg hosszát adja meg.
`char(s, i)`	Az `s` szöveg `i`. indexű helyen álló karakterét adja meg. (A szöveg első karaktere a 0 indexű.)

4. táblázat: Szövegekkel kapcsolatos beépített függvények

Van két függvény, amelyek funkciójuk miatt az előző két táblázat egyikébe sem sorolhatóak be, még fontos szerepük van bizonyos algoritmusok megvalósításakor. Ezek az 5. táblázatba kerültek.

függvény	leírás
`random(n)`	`1` és `n-1` közötti véletlen (egész) számot ad.
`size(t)`	A `t` tömb méretét (elemeinek számát) adja meg.

5. táblázat: További fontos beépített függvények

A beépített függvények segítségével **különböző típusok közötti konverzióra is lehetőségünk van**. Az 6. táblázatban gyűjtöttük össze az ide tartozó függvényekkel kapcsolatos tudnivalókat.

függvény	leírás
`ToInteger(n)`	Szöveg típusú számértéket alakít egész számmá.
`ToReal(n)`	Szöveg típusú számértéket alakít valós számmá.
`ToString(n)`	Szám típusú értéket alakít szöveg típusúvá.
`ToFixed(r, i)`	Valós számot alakít szöveggé úgy, hogy a tizedesvessző után `i` számjegyet jelenít meg.
`ToChar(n)`	Karakterkódot konvertál karakterré.
`ToCode(n)`	Karaktert konvertál karakterkóddá, az eredmény egész típusú).

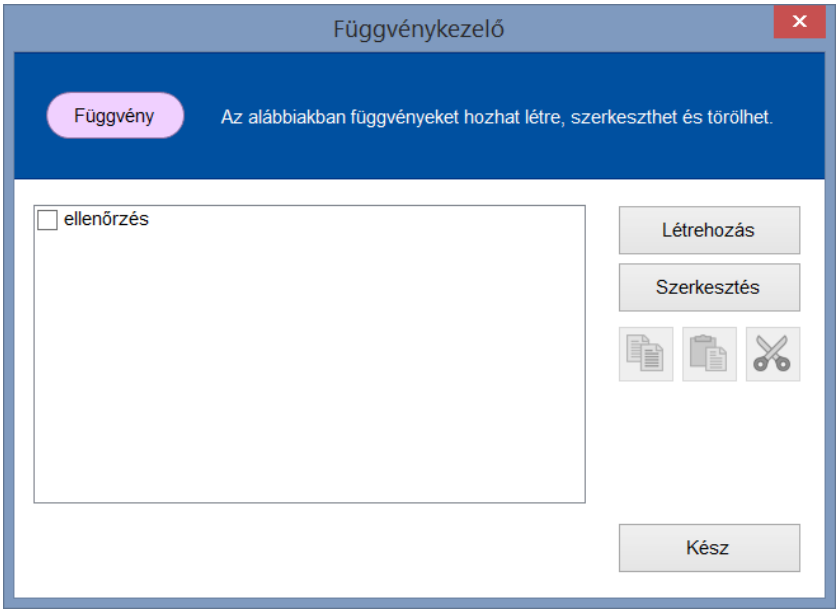
6. táblázat: Típusok közötti konverzióval kapcsolatos beépített függvények

A beépített függvények neveit nem használhatjuk változók és saját függvények neveiként. A későbbi bővítési tervek, újabb beépített függvények definiálása miatt nem használhatók függvények és változók neveiként a következők sem: `arccosh`, `cosh`, `arcsinh`, `sinh`, `arctanh`, `tanh`.

VI.2. Saját függvények létrehozása

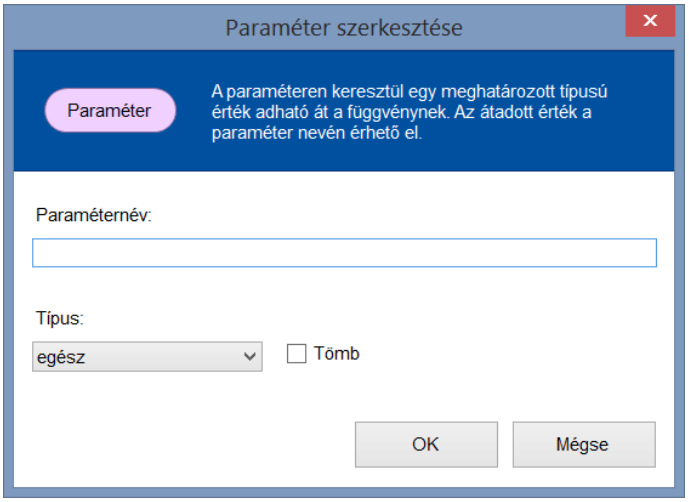
Az algoritmusok készítése során gyakran előfordul, hogy az algoritmus egy részére több alkalommal van szükségünk egy- egy programon belül. Ilyenkor az algoritmus bizonyos részeinek másolása nagyon jól tud jönni, de a másolt részben egy-egy apró utólagos módosítás esetén már nem örülünk annyira, ha több helyen kell ugyanazt átírnunk, javítanunk. Ezért hasznos, ha az ilyen ismétlődő részekhez vagy hosszú algoritmusok esetén annak egy-egy nagyobb részletéhez saját függvényeket hozunk létre.

Saját függvényeket a *Függvénykezelő*ben (19. ábra) **hozhatunk létre.** A már létrehozott függvények paramétereinek szerkesztésére is itt van lehetőségünk. A függvények neve előtti jelölőnégyzetek segítségével tudjuk kiválasztani azt, hogy melyik függvényt szeretnénk szerkeszteni.



19. ábra: Függvénykezelő egy korábban létrehozott függvénnyel

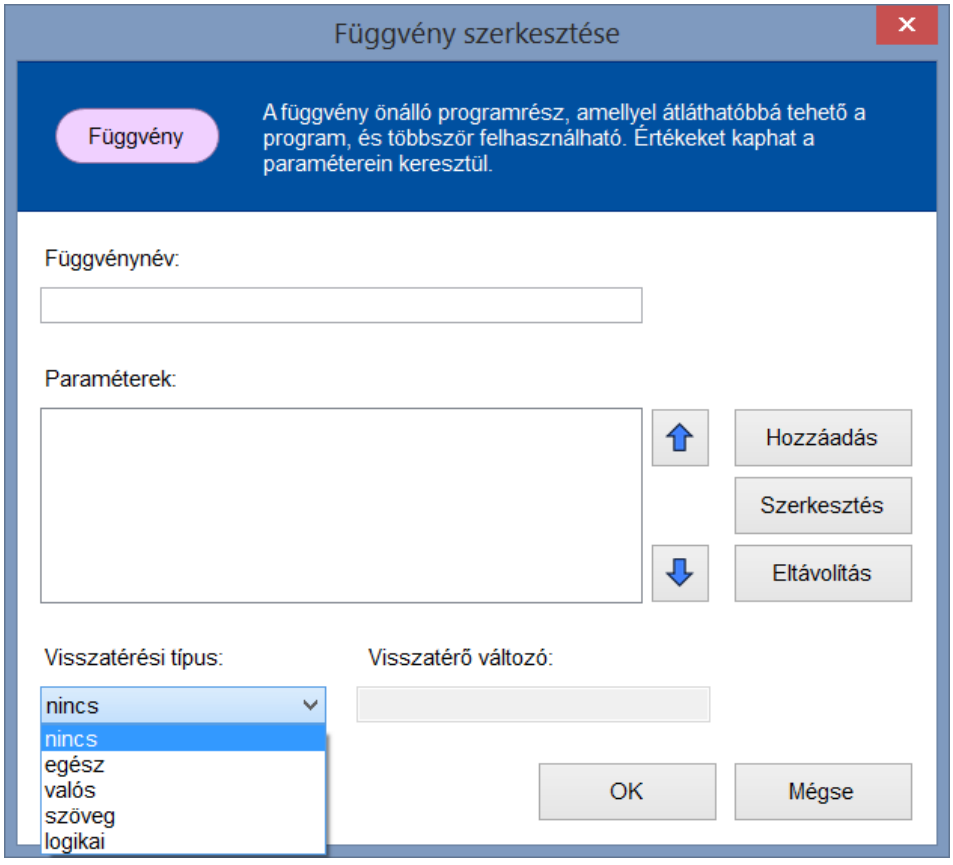
A *Létrehozás* gombra kattintva egy újabb ablak nyílik (21. ábra) meg, ahol legfelül **a függvény nevét kell megadnunk**. **A függvény paramétereinek létrehozásához a Hozzáadás gombra kell kattintanunk, ahol egy**, a változók létrehozásánál látotthoz hasonló **ablakban** (20. ábra) **kell beírnunk a paraméter nevét és kiválasztanunk annak típusát**. A létrehozott **paraméterek sorrendjét a felfelé, illetve lefelé mutató nyilak segítségével módosíthatjuk**.



20. ábra: A függvény egy paraméterének szerkesztése

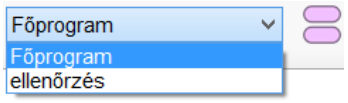
A függvény hívásakor fontos, hogy a behelyettesítendő változókat ugyanebben a sorrendben adjuk meg, ahogyan a függvény paramétereit, mert azok sorban kerülnek behelyettesítésre a program futtatásakor. Mivel ebben a környezetben a függvények gépelésekor nem jelenik meg az argumentumok listája, ezért a sorrend fontosságát és szerepét tudatosítanunk kell a diákokban is.

A függvény létrehozásakor a neve és a paraméterei mellett még **a visszatérési típust és a visszatérő változó nevét kell még megadnunk**.

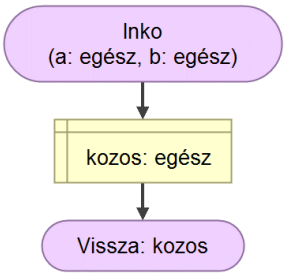


21. ábra: Függvény szerkesztése ablak

A létrehozott függvényeknek **legfeljebb egy visszatérő változója lehet**.



A létrehozott függvény algoritmusának szerkesztéséhez a *Főprogram*ról a szerkeszteni kívánt függvény nevére kell váltanunk a menüben látható legördülő listában.



Az függvény algoritmusában **a visszatérő változó automatikusan deklarálásra kerül**. A Főprogramnál látott Main és Vége feliratok helyett itt felül a függvény nevét, paramétereit és azok típusát láthatjuk, míg alul a Vissza szó és a visszatérő változó neve jelenik meg.

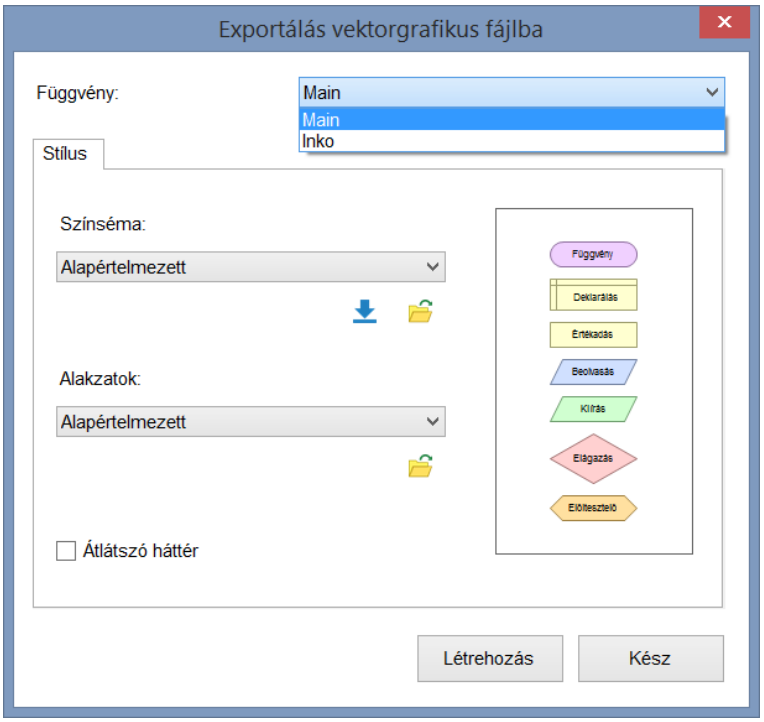
A függvények használatánál a tanulók számára eleinte nehézséget okoz az, hogy a függvény paramétereinek neve nem kell, hogy megegyezzen a függvény meghíváskor megadott változók nevével. Meg kell mutatni számukra, hogy a függvény paraméterei nem azonosak azokkal a változókkal, amikkel meghívjuk őket. Ebben segít a *Változók figyelése* ablak, ahol jól elkülönülnek ezek a változók. **A főprogramnál csak a globális változókat látjuk, míg a függvényeknél csak az aktuális függvényhez tartozókat.** A különbség megértését segíti az is, ha olyan példát mutatunk a diákoknak, ahol egy programon belül többször hívjuk meg ugyanazt a függvényt különböző változókkal.

VII. Mentési, exportálási lehetőségek

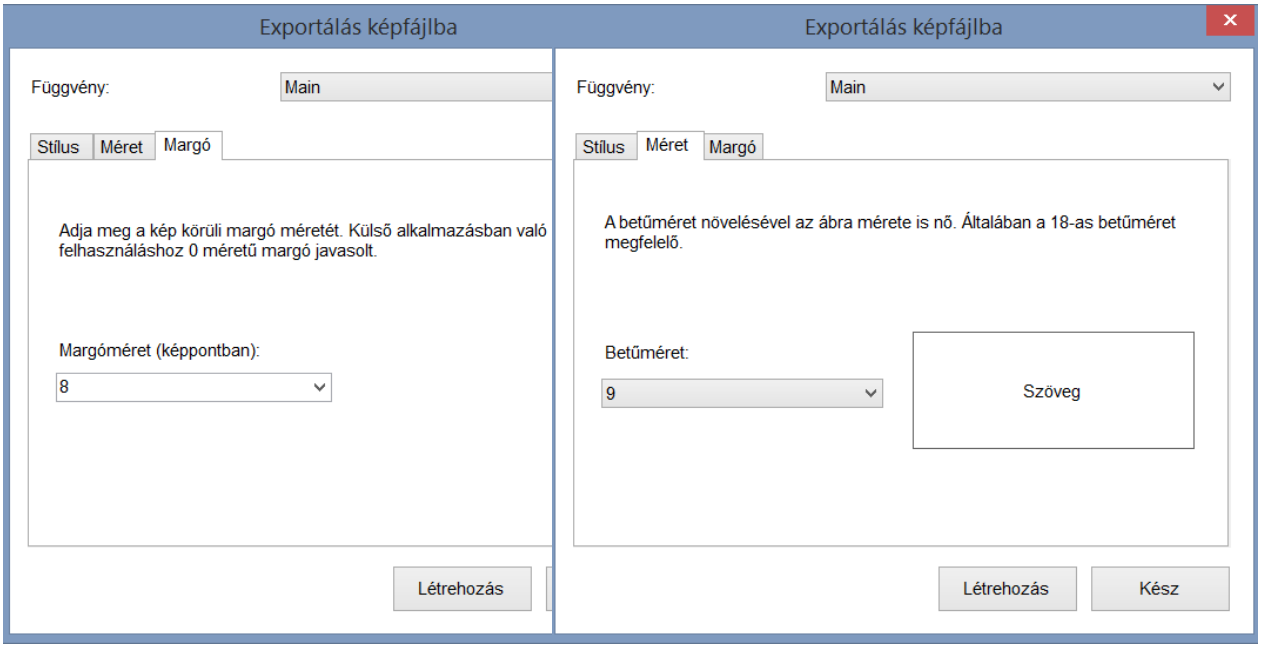
Az elkészített algoritmusokat a program alapértelmezett formátumába (*fprg*) menthetjük, így az később is futtatható lesz.

Az egyszerű mentésen kívül viszont **lehetőségünk van az algoritmus képfájlba** (*Eszközök – Exportálás képfájlba…* menüpont), **illetve vektorgrafikus képfájlba való exportálására** (*Eszközök – Exportálás vektorgrafikus képfájlba…* menüpont) is. Képfájl exportálása esetén *png*, vektorgrafikus képfájl esetén pedig *svg* vagy *emf* formátumban menthetjük el a munkánkat.

A képfájl és a vektorgrafikus képfájl exportálása esetén is beállíthatjuk, hogy milyen színsémával szeretnénk menteni a munkánkat és kiválaszthatjuk, hogy a főprogramot vagy valamelyik függvényt mentjük-e. Vektorgrafikus képfájl esetén dönthetünk úgy, hogy a kép hátterét átlátszóra állítjuk. Képfájl esetén megadhatjuk a betűméretet és a margók méretét is.



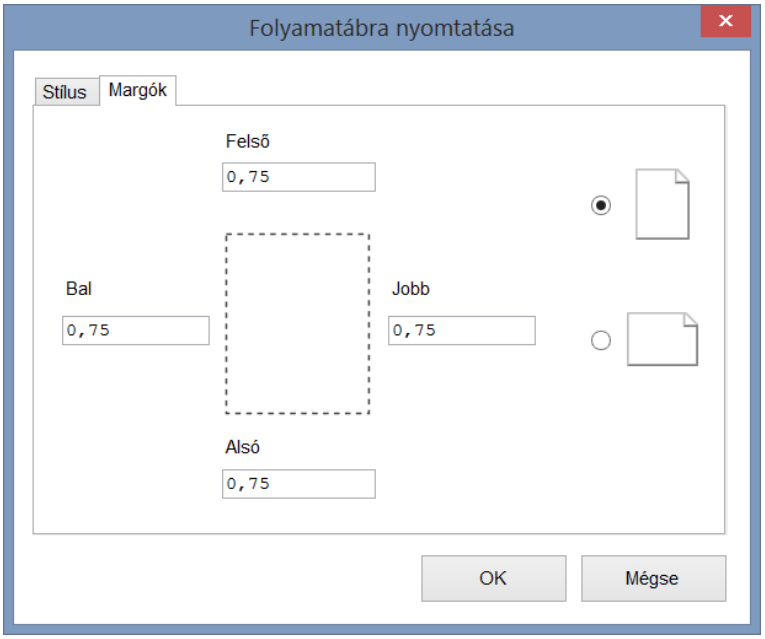
22. ábra: Exportálás vektorgrafikus fájlba



23. ábra: Exportálás képfájlba (margó és betűméret beállítása

A képfájlba való exportálás mellett **a teljes algoritmus pillanatnyi állapotáról készült képernyőképet is** könnyen **vágólapra másolhatjuk** az *Elrendezés – Pillanatkép másolása vágólapra* menüpont segítségével. Az képernyőn látható függvény vagy a főprogram algoritmusának képe kerül a vágólapra.

Az elkészített **algoritmus nyomtatására** (vagy *pdf*-be való nyomtatására) **is lehetőségünk van** a *Fájl – Nyomtatás* menüpont segítségével. Ekkor a színséma beállítása mellett a margókat és a laptájolást állíthatjuk be a felugró ablakban. **Nyomtatáskor a főprogram és a létrehozott függvények algoritmusai is nyomtatásra kerülnek**, külön oldalakon kapnak helyet.



24. ábra: Beállítási lehetőségek nyomtatáskor

Kidolgozott feladatok

Az anyag további részében különböző témakörökhöz kapcsolódó és különböző nehézségű feladatok szerepelnek. A feladatok többsége esetén a megoldás vagy a megoldást segítő utalások is szerepelnek. A hatékonyabb helykihasználás érdekében a megoldásokat helyenként megtörjük, és az egymás alatti részeket egymás mellett helyezzük el.

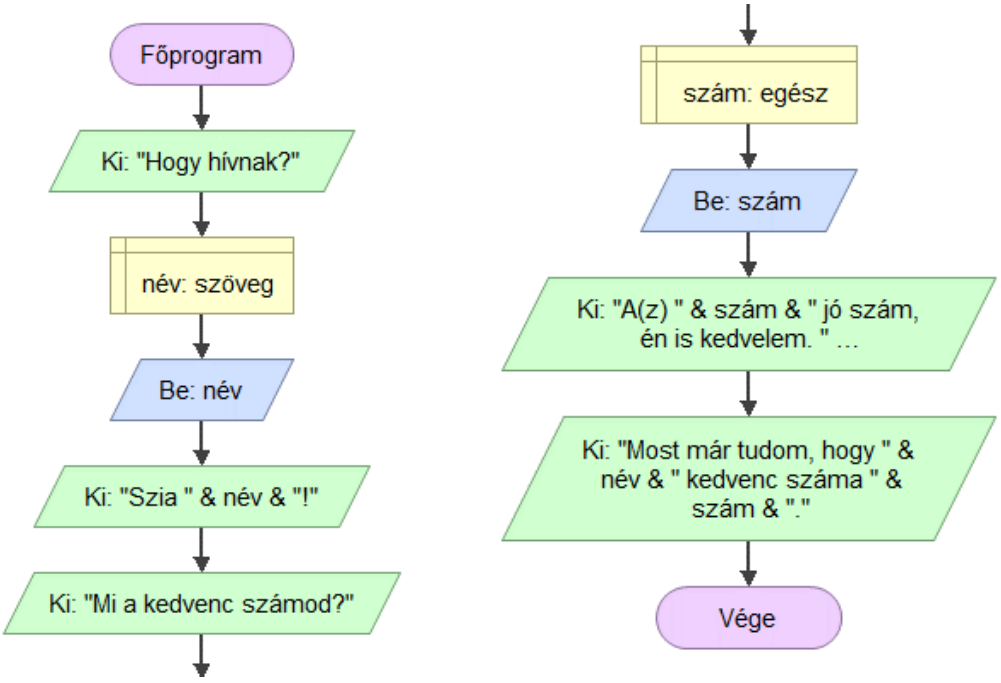
A feladatok segítségével fokozatosan bevezetve kerülnek elő a Flowgorithm-ben használható utasítások és a program adta lehetőségek. Így reményeink szerint az itt szereplő feladatok segítséget jelentenek a pedagógusok számára az algoritmizálás témakör Flowgorithm-mel való tanításához. Természetesen a feladatokat a tanított csoport összetételének, képességeinek és továbbtanulási terveinek megfelelően át lehet és valószínűleg helyenként át is kell formálnia mindenkinek. Az itteni feladatokat egyfajta vezérfonalnak, ötletgyűjteménynek szánjuk, miközben a program lehetőségeit is bemutatjuk a kidolgozott példákon keresztül.

I. Beolvasás, kiírás és változók

Az első feladatok a beolvasás és kiírás utasítások, valamint a változók használatának, típusainak megismerését szolgálják.

1. Beszélgetős, ismerkedős program

Valósítsunk meg egy ismerkedő beszélgetést a számítógép és a felhasználó között!



Új ismeretek:

- program használata (utasítások beszúrása, az algoritmus futtatása)
- Kiírás utasítás (" és & használata, *Új sor* funkció, változó értékének megjelenítése a kiírásban)
- Deklarálás utasítás (szöveg és egész típusok)
- Beolvasás utasítás

Házi feladatként vagy további gyakorlásra ehhez hasonló, párbeszédes feladatok megoldását javasoljuk.

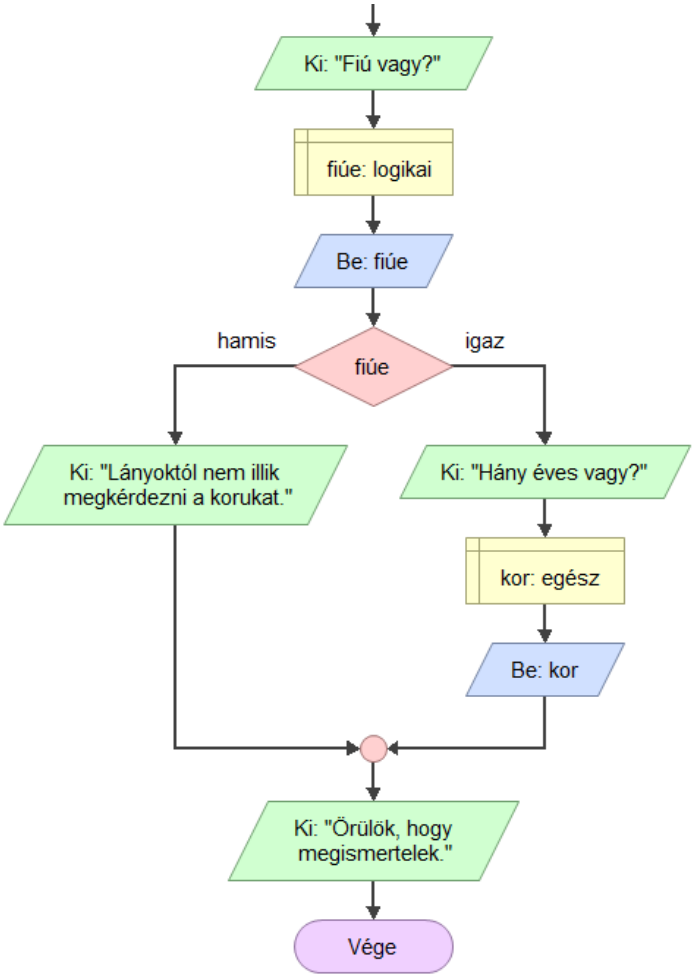
II. Elágazás

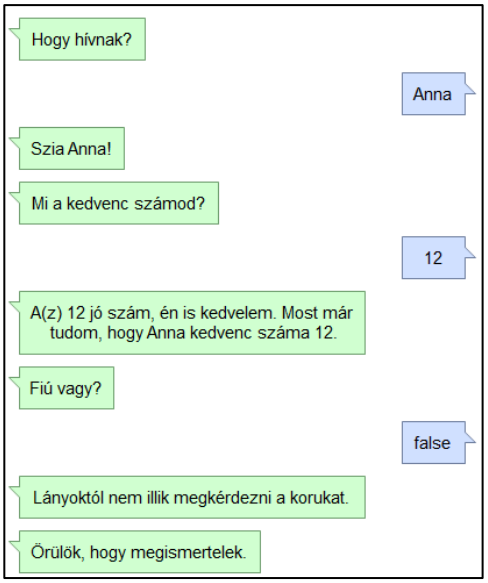
A kiírás és beolvasás utasítás után az elágazás a legegyszerűbb. Elágazással a való életben is találkozunk, így könnyű egyszerű, elágazást tartalmazó példákat mondani. Éppúgy példaként említhetjük a tinimagazinokban előforduló, folyamatábrához hasonlító kérdőíveket, ahogyan a táblázatkezelő programok `HA` függvényét.

2. Beszélgetős program elágazással

A számítógép és a felhasználó közötti korábbi beszélgetést bővítsük ki elágazással!

A beszélgetős program írásakor a tanulók ötletei kapcsán hamar elő szokott kerülni olyan kérdés, amihez már az elágazásra is szükség van.





Új ismeretek:

- Deklarálás utasítás (logikai típus)
- logikai változó lehetséges értékei (`true`, `false`)
- Elágazás utasítás
- logikai változó elágazás feltételeként

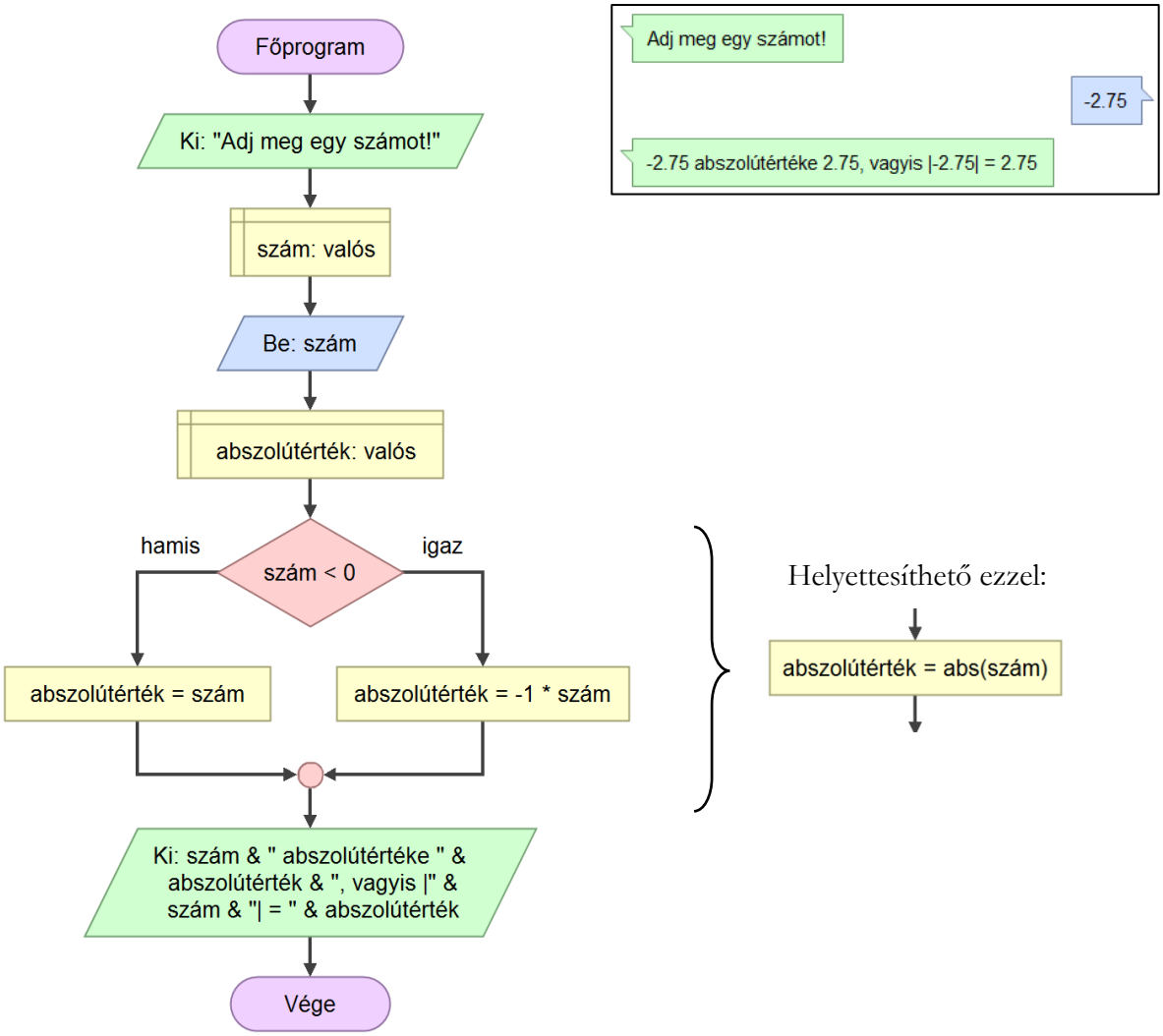
Diákok által javasolt hasonló feladat:

Kérdezzük meg a felhasználó életkorát, majd az alapján írjuk ki, hogy nagykorú-e vagy azt, hogy vásárolhat-e már energiaitalt vagy alkoholt. (Ehhez relációt tartalmazó feltételre is szükség van.)

3. Abszolútérték

Olvassunk be egy számot és adjuk meg az abszolútértékét!

A szám abszolútértékének meghatározására beépített függvény is van a programban. A feladat elágazással történő megoldása után ezt is érdemes kipróbálni.



Új ismeretek:

- valós típus (beolvasáskor tizedesvessző helyett ponttal való megadása)
- relációs feltétel elágazásban
- Értékadás utasítás (számított értékkel)
- beépített `abs` függvény

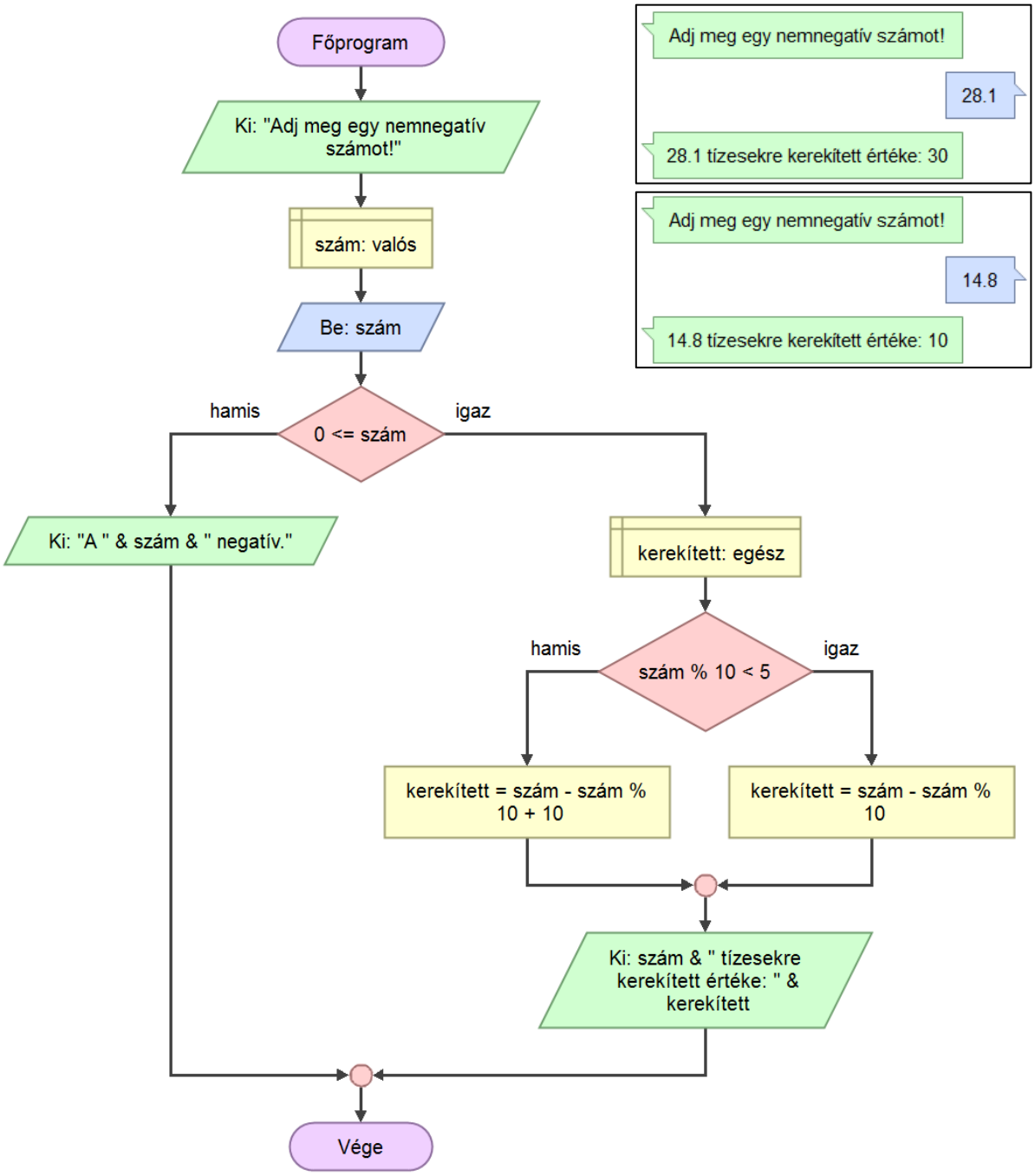
Hasonló feladat:

A beépített `sgn` függvény algoritmusának elkészítése. A `sgn` függvény értéke nulla esetén 0, pozitív számoknál 1, negatívaknál pedig -1. (A „szép” megoldáshoz elágazások egymásba ágyazására is szükség van, de üres ágakkal anélkül is megoldható.)

4. Kerekítés

Olvassunk be egy valós számot és adjuk meg a tízesre kerekített értékét!

A szám 10-zel való osztási maradéka segítségével oldjuk meg a feladatot.



Új ismeretek:

- elágazások egymásba ágyazása
- % vagy `mod` művelet (osztási maradék)

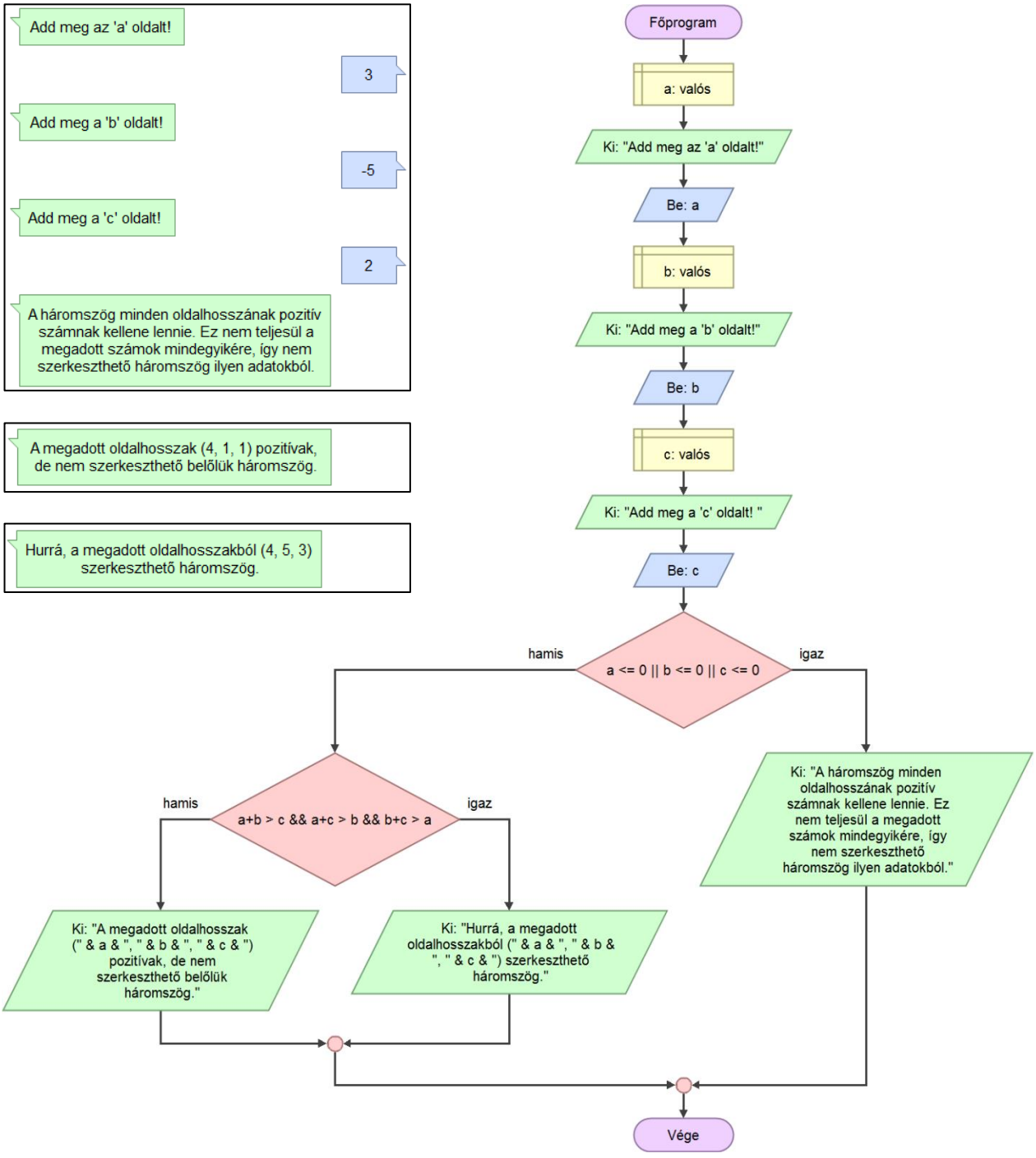
Hasonló feladatok:

- A feladat másik változatában az is bekérhetjük a felhasználótól, hogy hány számjegyre kerekítsünk. (Itt is utalhatunk a táblázatkezelő programok kerekítő függvényeire.)
- A beépített egészrész függvény (`int`) algoritmusának elkészítése.

5. Szerkeszthető-e a háromszög?

Olvassunk be három számot és írjuk ki, hogy szerkeszthető-e háromszög a megadott oldalhosszúságokkal!

A feladat megoldásához fel kell használnunk azt, hogy a háromszög oldalhosszúságai csak pozitív számok lehetnek, illetve a háromszög-egyenlőtlenséget (bármely két oldal hosszának összege nagyobb kell, hogy legyen a harmadik oldal hosszánál).



Új ismeret:

- és (`and`, `&&`), illetve vagy (`or`, `||`) művelet használata, kiértékelési sorrend logikai kifejezésben

Az algoritmusban szereplő feltételeket többféleképpen megadhatjuk úgy, hogy azok helyes eredményhez vezessenek. Érdemes a tanított csoport által javasolt gondolatmenet mentén elindulni.

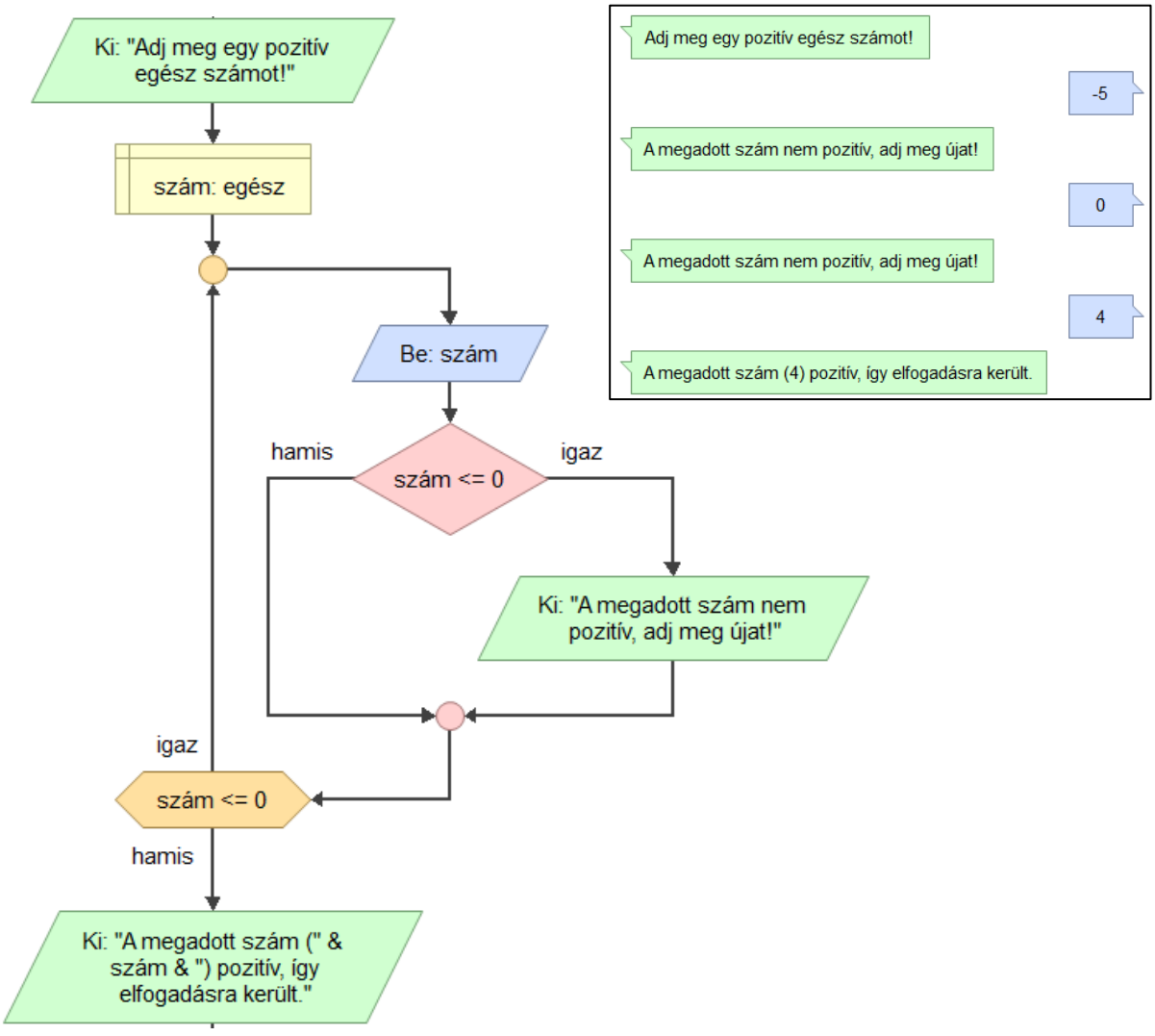
Valószínűleg a diákok először több elágazást tartalmazó megoldást javasolnak, ha nem találkoztak még az *és*, illetve a *vagy* operátorral. Így az algoritmusban több helyen kerülhet kiírásra ugyanaz hosszú a szöveg. Ehelyett javasolhatjuk például egy logikai változó bevezetését, amivel az algoritmus végén eldöntjük, hogy a szerkeszthető vagy a nem szerkeszthető üzenetet írjuk ki. Háromféle kiírás használatánál viszont ez a módszer már meglehetősen bonyolult, így remélhetőleg kérdéseik kapcsán felmerül a tanulói igény a két operátor megismerésére.

III. Hátultesztelő ciklus

Sok esetben az előző két feladat kapcsán felmerül az, hogy jobb lenne újra bekérni a felhasználótól a rosszul megadott értékeket (pl.: a negatív számoknál) annak kiírása helyett, hogy a megadott szám vagy a számok valamelyike nem megfelelő. A hátultesztelő ciklus bevezetéséhez pedig éppen erre az ötletre, felhasználói igényre van szükség.

6. Pozitív szám beolvasása

Olvassunk be egy pozitív számot! Ha a felhasználó negatív számot ad meg, akkor írjunk ki hibaüzenetet, és kérjünk új számot!



A feladat megoldása során érdemes hagyni, hogy a tanulók maguk találják ki, hogy hová érdemes írni az eredetileg szöveget kiíró utasítást, illetve a hibaüzenetét. Általában ilyenkor merül fel az a gondolat, hogy a hibaüzenet kiírása a hátultesztelő ciklusban a felfelé mutató nyílon történjen meg, de ez, ahogyan már korábban írtunk nem lehetséges.

Célszerű rámutatni a feladat kapcsán arra, hogy (egyféle hibaüzenet esetén) a hibaüzenet kiírásához tartozó elágazás és a hátultesztelő ciklus feltétele megegyezik (ha a hibaüzenet kiírását az igaz ágra tesszük). Tapasztalataim szerint ez segíti a diákok munkáját. Enélkül gyakori, hogy kétszer próbálják kitalálni a megfelelő feltételt, ez pedig különböző feltételekhez, nehezen észrevehető hibákhoz vezet.

Az előző két feladat beolvasását ehhez hasonlóan módosíthatjuk.

Gyakran felmerül a kérdés a tanulók részéről azzal kapcsolatban, hogy pozitív, negatív vagy nemnegatív számok beolvasása esetén elfogadható-e a 0 vagy sem. Erre érdemes figyelni és tudatosítani bennük, hogy a 0 nem pozitív és nem is negatív.

Új ismeretek:

- Hátultesztelő ciklus utasítás
- ellenőrzött beolvasás egyszerű feltétellel

Az ellenőrzött beolvasásnak sokféle, ennél jóval összetettebb feltétele is lehet, de ezekre itt most nem mutatunk több példát, mert a későbbi feladatok majdnem mindegyikében találkozunk majd ellenőrzött beolvasással.

7. Gondoltam egy számra! (egyszerűbb)

Hozzunk létre egy véletlen számot 1 és 100 között, majd találjuk ki, hogy melyik ez a szám!

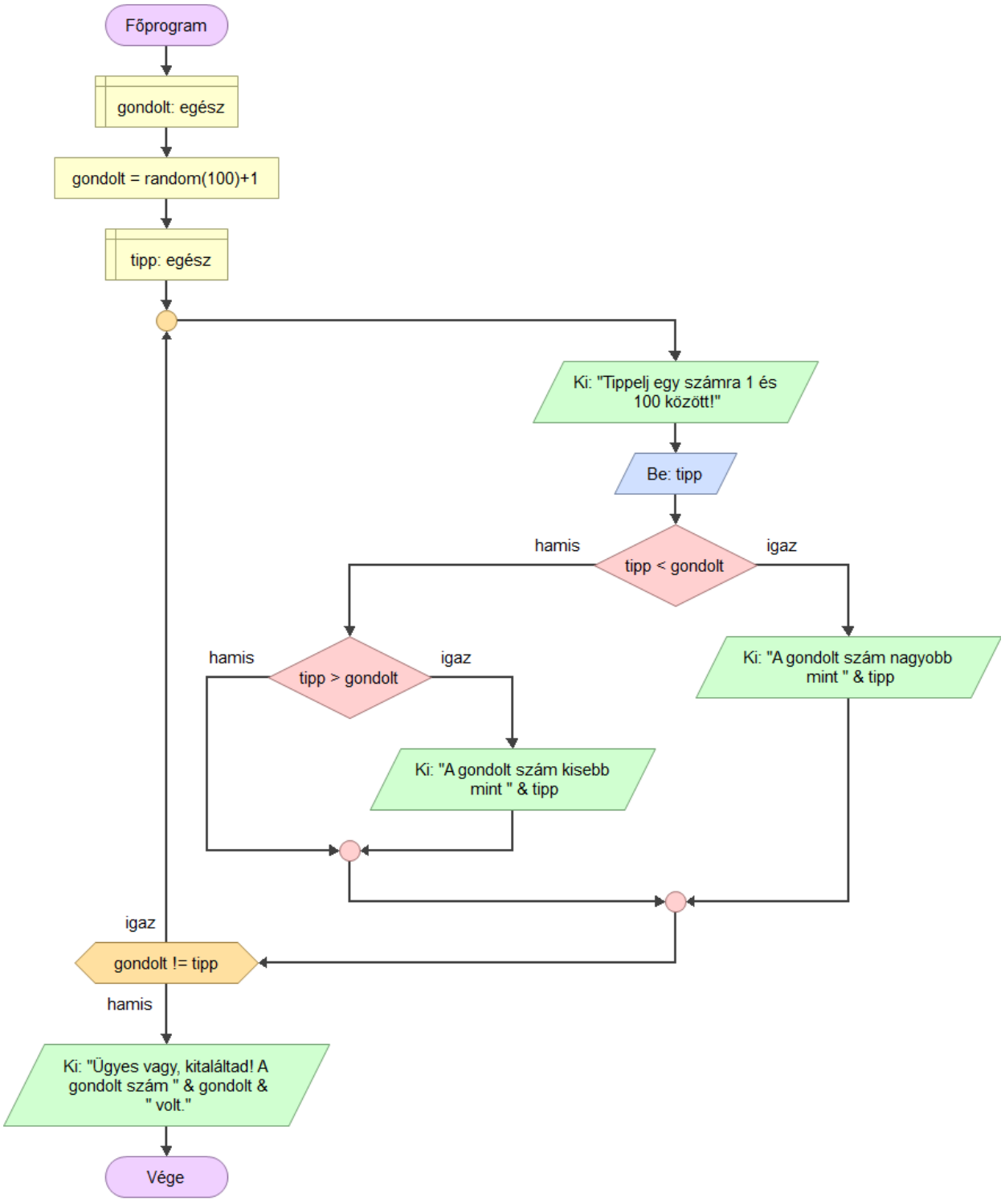
Azoknál a csoportoknál, ahol az ellenőrzött beolvasás igénye nem merül fel, jó alternatíva lehet ez a feladat. Ennek megoldását is előszeretettel kezdik úgy a tanulók, hogy elágazások sorozatát kezdik egymásba ágyazni, de általában hamar rájönnek arra, hogy nem tudhatják előre, hogy hány elágazásra lesz szükség futtatáskor. Így ezzel a feladattal is bevezethetjük a hátultesztelő ciklust.

Mivel a `random(100)` egy 0 és 99 közötti egész számot ad, így a kapott számhoz 1-et hozzá kell adni ahhoz, hogy a gondolt szám ténylegesen 1 és 100 közé essen (a határokat is beleértve).

Miután kiderül, hogy a `random(100)` nem megfelelő, fel szokott merülni a `random(101)` használatának gondolata, de az sem jó megoldás, mert az 0 és 100 közötti véletlen számot generál.

A háromféle felmerülő megoldás közötti különbséget legkönnyebben talán számegyenes segítségével szemléltetjük.

Az algoritmus tesztelése közben fontos, hogy tudjuk, hogy mi a gondolt szám. Könnyen adódik a korábbi feladatok alapján az, hogy kiírhatjuk az értékét üzenetként, de ez a feladat jó alkalom lehet arra, hogy megmutassuk a tanulóknak a *Változók figyelése* ablakot.



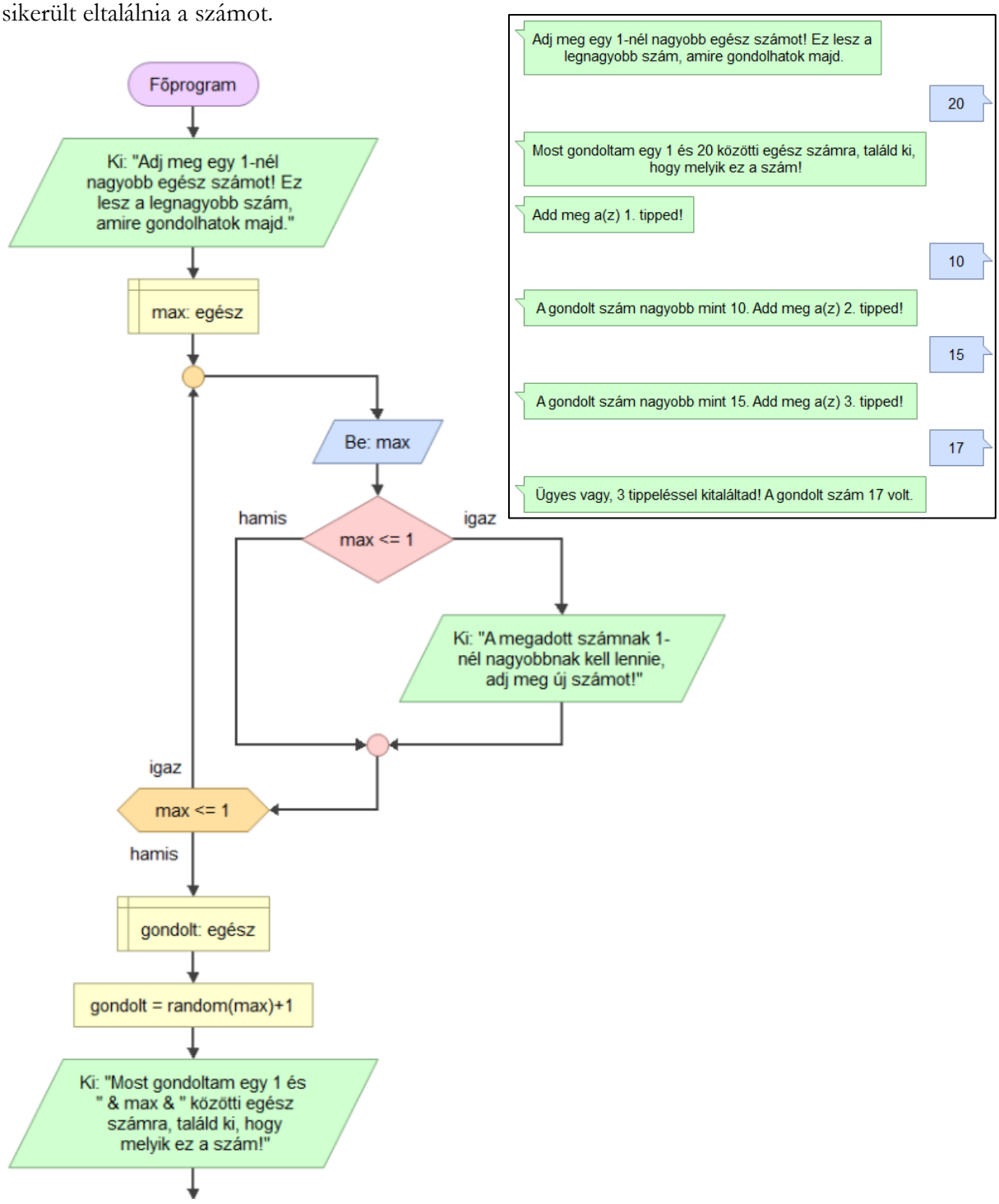
Új ismeretek:

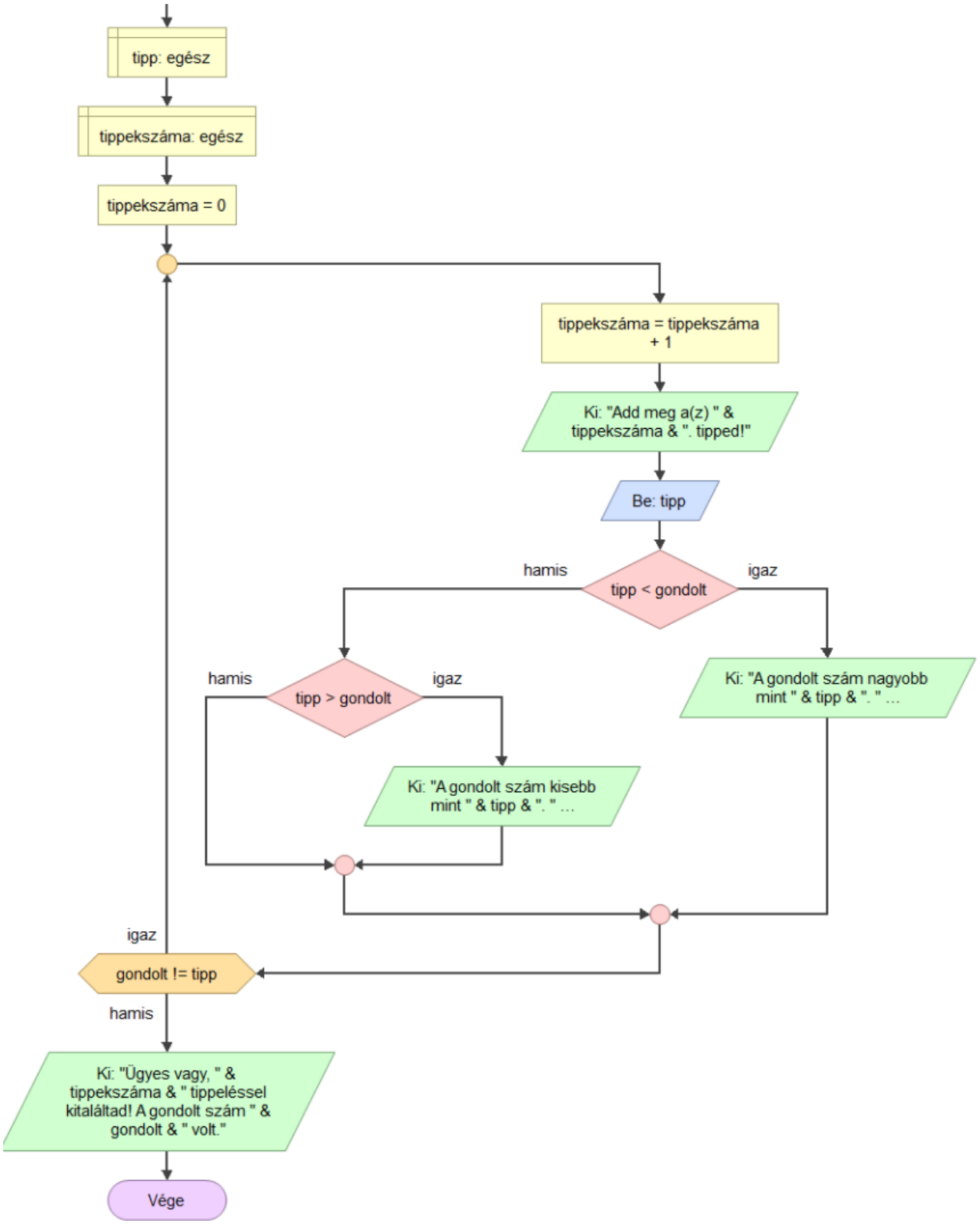
- ellenőrzött beolvasás összetettebb feltétellel
- véletlenszám generálása (`random` függvénnyel)
- nem egyenlő reláció (`!=` vagy `<>`) használata logikai kifejezésben
- *Változók figyelése* ablak használata (az érték nélküli változó nem inicializáltként jelenik meg)

8. Gondoltam egy számra! (bővített)

Hozzunk létre egy véletlen számot 0 és a felhasználó által megadott szám között, majd találjuk ki, hogy melyik szám lehet az. Közben számon tartjuk a próbálkozások számát is.

A feladat kicsit nehezített, összetettebb változata ez, amikor nem adjuk meg előre, hogy mi lehet a legnagyobb gondolt szám, hanem a felhasználó döntheti el, hogy mi legyen a felső határ. Az előző változathoz képest a másik módosítás az, hogy itt a szám kitalálása közben a felhasználó próbálkozásainak számát is tároljuk, illetve jelezzük neki, hogy hányadik próbálkozásnál tart, hányadikra sikerült eltalálnia a számot.





További módosítási lehetőség az, amikor nem csak a felső, hanem az alsó határt is a felhasználó adja meg. Ekkor a második határszám beolvasásánál arra kell figyelni, hogy az nagyobb vagy kisebb legyen az elsőként megadottnál a határok beolvasási sorrendjének megfelelően.

Új ismeret:

- változó értékének 1-esével való növelése

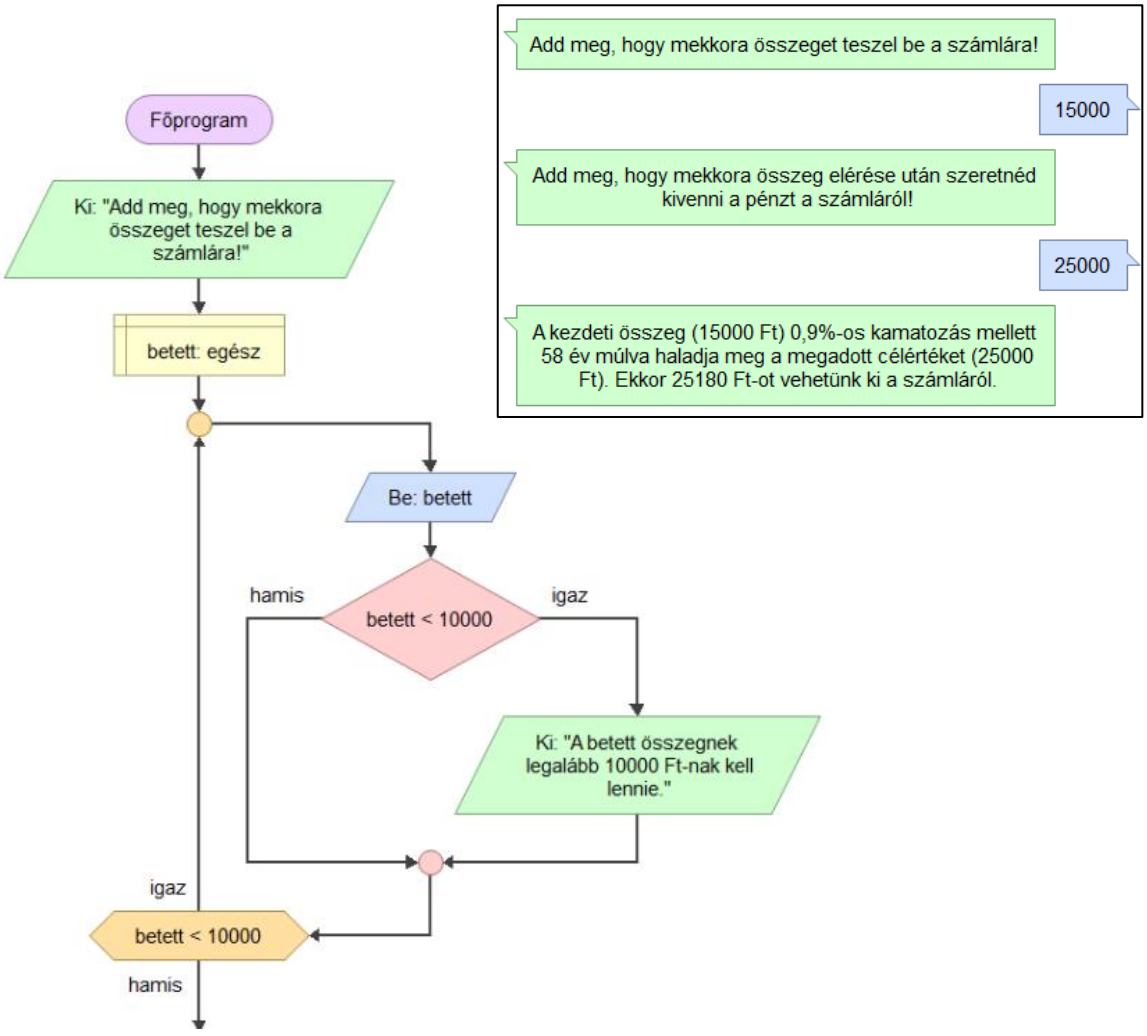
9. Banki megtakarítás

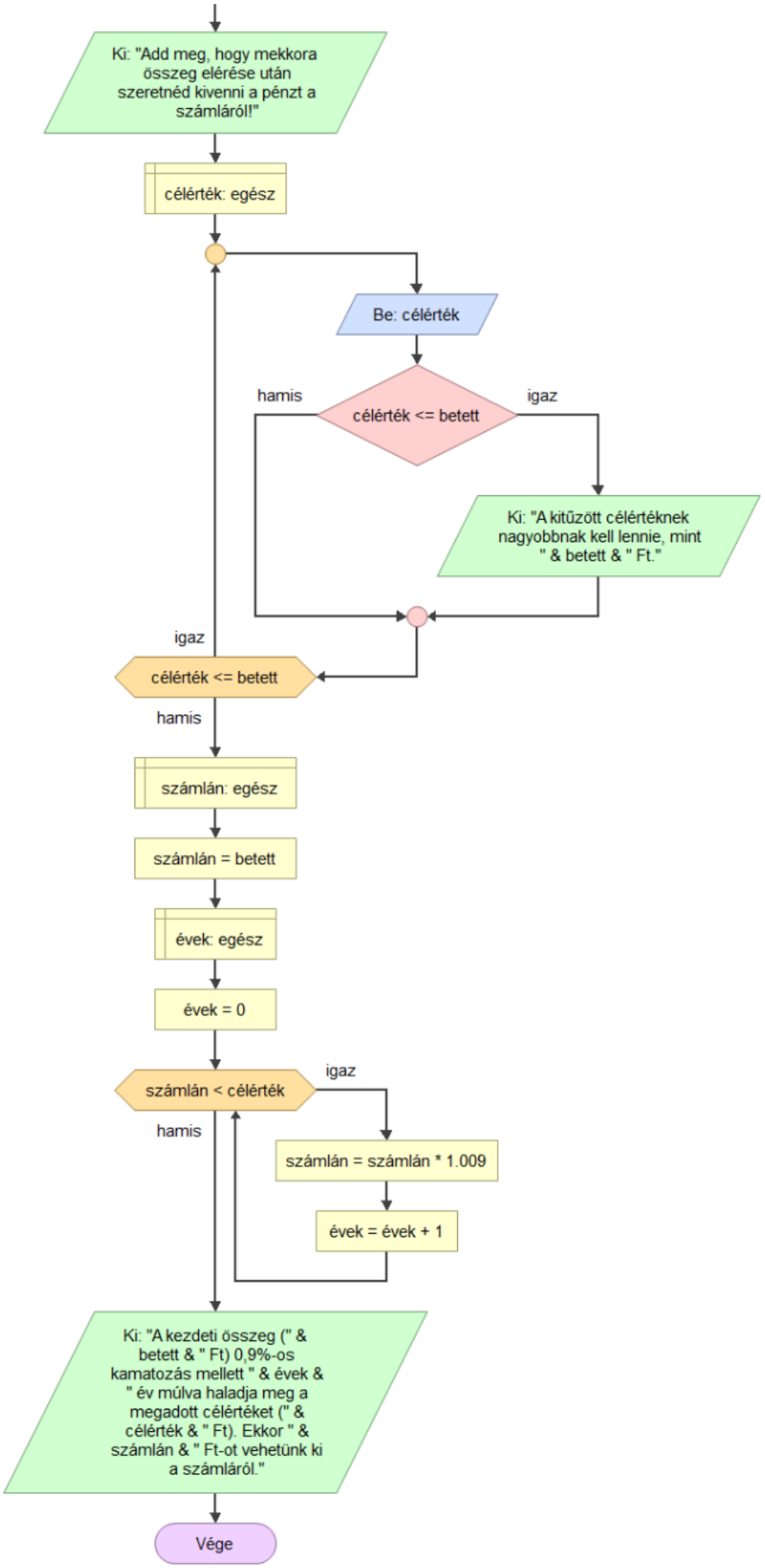
Újévi fogadalmunk részként megtakarítási számlát nyitunk, amelyre beteszünk egy bizonyos összeget (legalább 10000 Ft-ot). A számlán lévő pénz minden év végén 0,9%-ot kamatozik. Előre elhatározzuk, hogy mekkora összegnél vesszük majd ki a teljes összeget a számláról. Készítsünk algoritmus, amely meghatározza, hogy változatlan kamatozás mellett hányadik év végén kell kivennünk az összeget! (A számlára tett összeget és a célértéket a felhasználó adja meg. A célértéknek a betett összegnél nagyobbnak kell lennie.)

A megoldás kapcsán meggondolandó az, hogy engedélyezzük-e, hogy a számlán lévő összeg tizedestört legyen. Ha igen, akkor a `számlán` változó típusát *valós*ra kell módosítanunk, az algoritmus végén lévő kiírásban pedig az `Int` vagy a `ToFixed` beépített függvények valamelyikével szépíthetjük a számlán lévő összeg kiírásra kerülő formátumát.

Hasonló feladatok:

- A feladat módosított változatában a kamat mértékét is megadhatja a felhasználó. Egy másik módosítási lehetőség az, hogy a második évtől kezdve minden év elején egy fix összeggel növelhetjük a számlán lévő összeget.
- A célérték, a kamat és a rendelkezésre álló évek ismeretében határozzuk meg, hogy mekkora összeget kell kezdetben betenni a számlára. (Ez a feladat számlálós ciklussal is megoldható.)
- Kamatos kamat számításához kötődően elöltesztelő ciklussal kapcsolatos feladatok is kitűzhetők a feltételek módosításával.




Ki: "Add meg, hogy mekkora összeg elérése után szeretnéd kivenni a pénzt a számláról!"
célérték: egész
Be: célérték
hamis
célérték <= betett
igaz
Ki: "A kitűzött célértéknek nagyobbnak kell lennie, mint " & betett & " Ft."
igaz
célérték <= betett
hamis
számlán: egész
számlán = betett
évek: egész
évek = 0
számlán < célérték
igaz
hamis
számlán = számlán * 1.009
évek = évek + 1
Ki: "A kezdeti összeg (" & betett & " Ft) 0,9%-os kamatozás mellett " & évek & " év múlva haladja meg a megadott célértéket (" & célérték & " Ft). Ekkor " & számlán & " Ft-ot vehetünk ki a számláról."
Vége

IV. Elöltesztelő ciklus

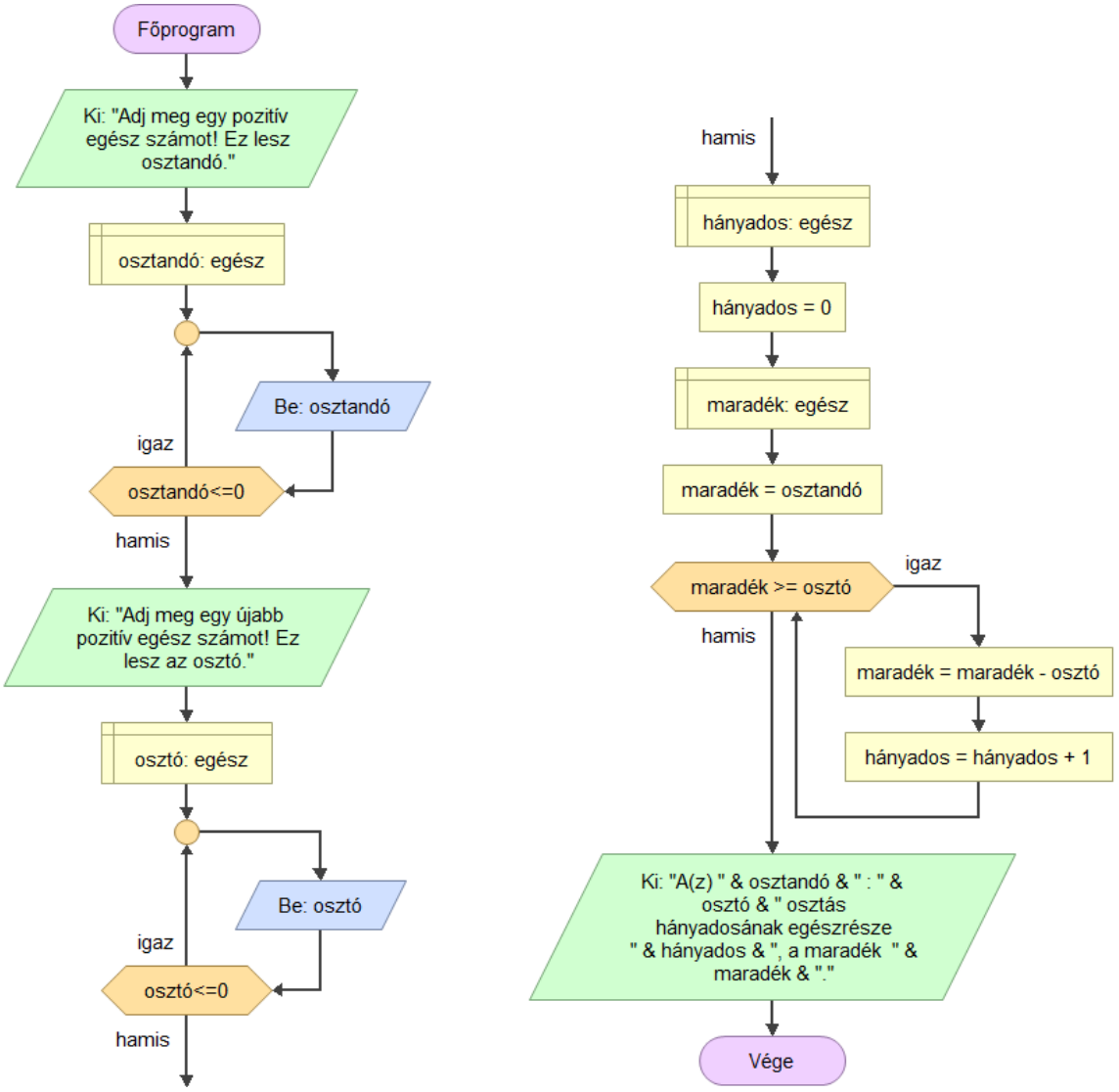
A hátultesztelő ciklus után érdemes rögtön bevezetni az elöltesztelő ciklust is a két utasítás szerkezetének hasonlósága miatt. Fontos tisztázni azt is, hogy mi a különbség a két szerkezet között, mitől függ, hogy melyiket használjuk.

Mindkét ciklusnál a megadott feltételtől függ az, hogy hányszor kerülnek végrehajtásra a ciklusmagban lévő utasítások. Az elöltesztelő ciklusnál a feltételvizsgálat van előbb, a hátultesztelőnél pedig a feltétel vizsgálata előtt végrehajtásra kerülnek a ciklusmagban lévő utasítások. Ebből adódik az, hogy **a hátultesztelő ciklus magjában lévő utasítások legalább egyszer biztosan lefutnak, míg az elöltesztelő ciklusnál lehet, hogy a ciklusmag utasításai egyszer sem kerülnek végrehajtásra**.

10. Maradékos osztás

Olvassunk be két pozitív egész számot, egy osztandót és egy osztót, majd (az osztás és a mod (%) műveletek használata nélkül) határozzuk meg a két szám hányadosának egészrészét és az osztási maradékot!

A hányados egészrészének értelmezéséhez fiatalabb csoportok esetén némi magyarázatra lehet szükség. Egy konkrét, számokkal felírt példa segítségével viszont könnyen érthetővé tehetjük számukra is a megoldandó feladatot.



Új ismeret:

- elöltesztelő ciklus

A feladat osztás és mod (%) műveletek nélkül ismételt kivonással oldható meg. Az ismételt kivonás gondolatához jó rávezető példa az, ha egy kupac cukorka (*osztandó* db) adott számú gyerek (*osztó* db) közötti szétosztásához hasonlítjuk feladatot. Ha nem végezhetünk előzetes számításokat, nem számolhatjuk meg előre, hogy összesen hány cukorka van, akkor könnyen adódik az a megoldási lehetőség, hogy ha tudunk, akkor mindenkinek adunk egy-egy cukorkát, és ezt ismételgetjük addig, ameddig lehet.

Néha nehézséget szokott okozni az, hogy a `maradék` változó kezdőértéke miért az `osztandó`. Ennek magyarázatához is jól jön az előbb említett cukorkás példa, illetve az, hogyha felhívjuk a tanulók figyelmét arra, hogy az `osztandó` értékét nem változtathatjuk meg, mert a kiírásnál szükség van a felhasználó által megadott értékre.

A feladatot csoporttól függően érdemes két lépésben megoldani úgy, hogy először csak a hányados egészrészének vagy a maradék kiszámítását tartjuk szem előtt, majd ezt egészítjük ki a másik változó számításához szükséges lépésekkel.

A tanulók számára az elöltesztelő ciklus megértéséhez nagy segítség lehet, ha lépésenként futtatjuk az algoritmust, miközben a változók értékeinek változását is figyelemmel kísérjük a *Változók figyelése* ablak segítségével. Emellett további segítséget jelenthet az, ha minden ciklusléptésben kiírjuk a változók aktuális értékeit a konzolra, hogy a futtatás végén egyben is látható legyen az, hogy mi történt.

A feladat kiegészíthető a számok beolvasásánál azzal, hogy hibás értékek megadásakor hibaüzenetet is kiírunk a felhasználónak. (Úgy, ahogyan azt korábban a *Pozitív szám beolvasása* feladatnál tettük.)

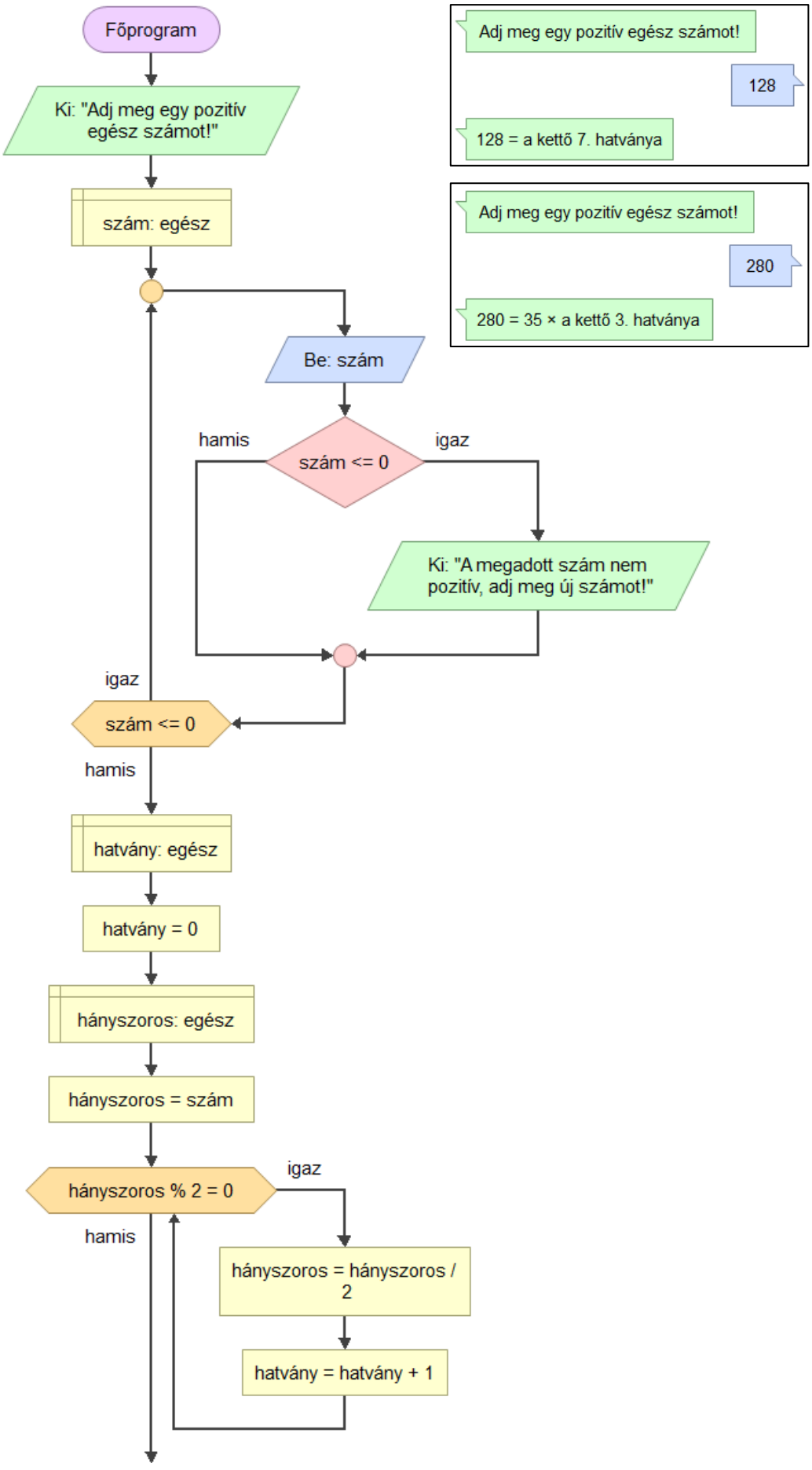
A feladat megoldása után érdemes megmutatni a tanulóknak azt, hogy mi történik akkor, ha az elöltesztelő ciklus helyett `maradék = osztandó / osztó` értékadást használunk. Ez jó lehetőség arra, hogy megfigyeljék, hogy mi történik, ha két egész szám hányadosát egész típusú változóban tároljuk. Emellett a feladat kapcsán megemlíthető a mod (%) művelet is, ami a programba beépített művelet, az osztási maradék meghatározására.

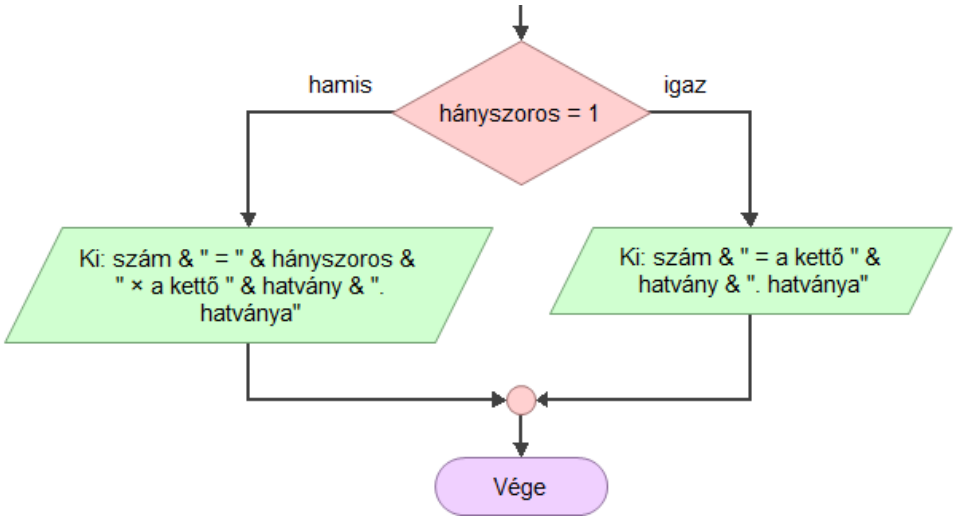
11. A szám 2-hatvány?

Olvassunk be egy számot, majd állapítsuk meg, hogy 2-hatvány-e! Ha 2-hatvány, akkor adjuk meg, hogy hányadik hatványa. Ha a szám nem 2-hatvány, akkor adjuk meg a legnagyobb 2-hatványt, amelynek többszöröse, és azt hogy annak hányszorosa.

A megoldás során a számot addig osztjuk 2-vel, ameddig maradék nélkül osztható 2-vel. Az osztogatás közben számoljuk, hogy hány alkalommal osztunk 2-vel, mert az az információ kell a válasz kiírásához. Ha a folyamat végén a szám 1, akkor az eredeti szám 2-hatvány. Ha az osztások után a kapott szám bármilyen más szám, akkor az eredeti szám nem 2-hatvány.

Ebben a feladatban a `hányszoros` változó szerepe hasonló az előző feladatban lévő `maradék` változó szerepéhez. Valószínűleg itt is némi magyarázat szükséges annak megértéséhez, hogy miért a `szám` változó értéke lett a `hányszoros` változó kezdőértéke. A magyarázat egyrészt az, hogy a szám változó felhasználó által megadott értékére szükség van a kiírásban, ezért annak értékét az algoritmus során nem változtatjuk meg. Másrészt pedig az, hogy az algoritmus végén a `hányszoros` változóban pontosan az az érték lesz, amivel a megfelelő 2-hatványt megszorozva az eredeti számot kapjuk vissza.


Főprogram
Ki: "Adj meg egy pozitív egész számot!"
szám: egész
Be: szám
hamis
szám <= 0
igaz
Ki: "A megadott szám nem pozitív, adj meg új számot!"
igaz
szám <= 0
hamis
hatvány: egész
hatvány = 0
hányszoros: egész
hányszoros = szám
hányszoros % 2 = 0
igaz
hamis
hányszoros = hányszoros / 2
hatvány = hatvány + 1
Adj meg egy pozitív egész számot!
128
128 = a kettő 7. hatványa
Adj meg egy pozitív egész számot!
280
280 = 35 × a kettő 3. hatványa



Új ismeret:

- egyenlőségvizsgálat logikai kifejezésben (a = jel helyett az == is használható)

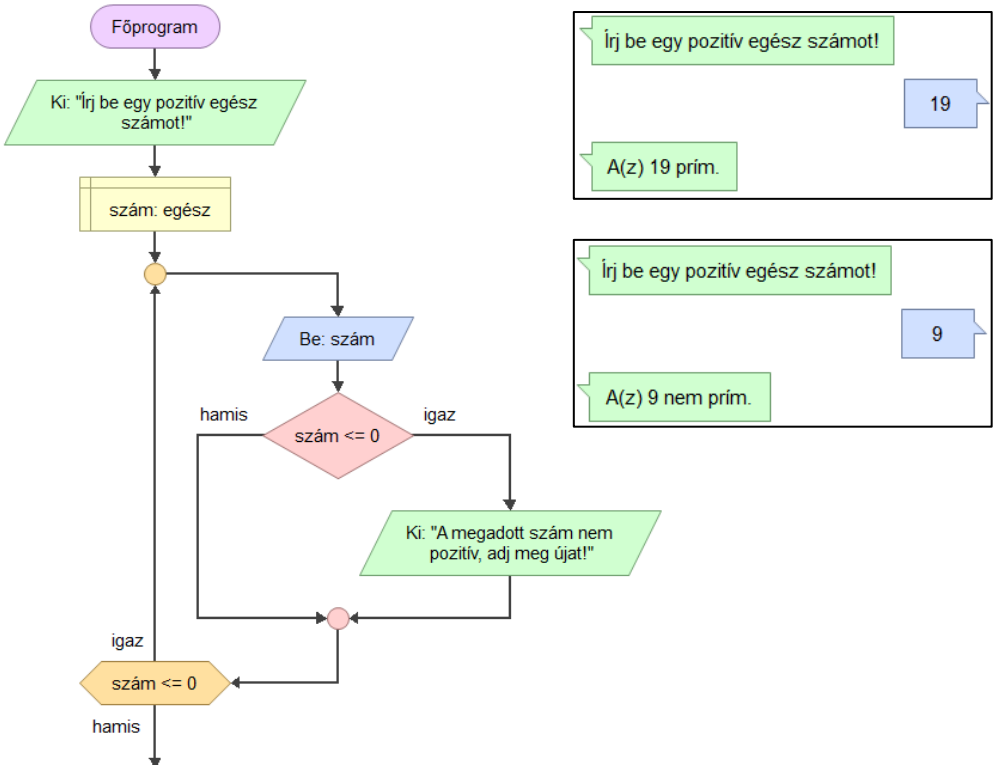
Hasonló feladatok:

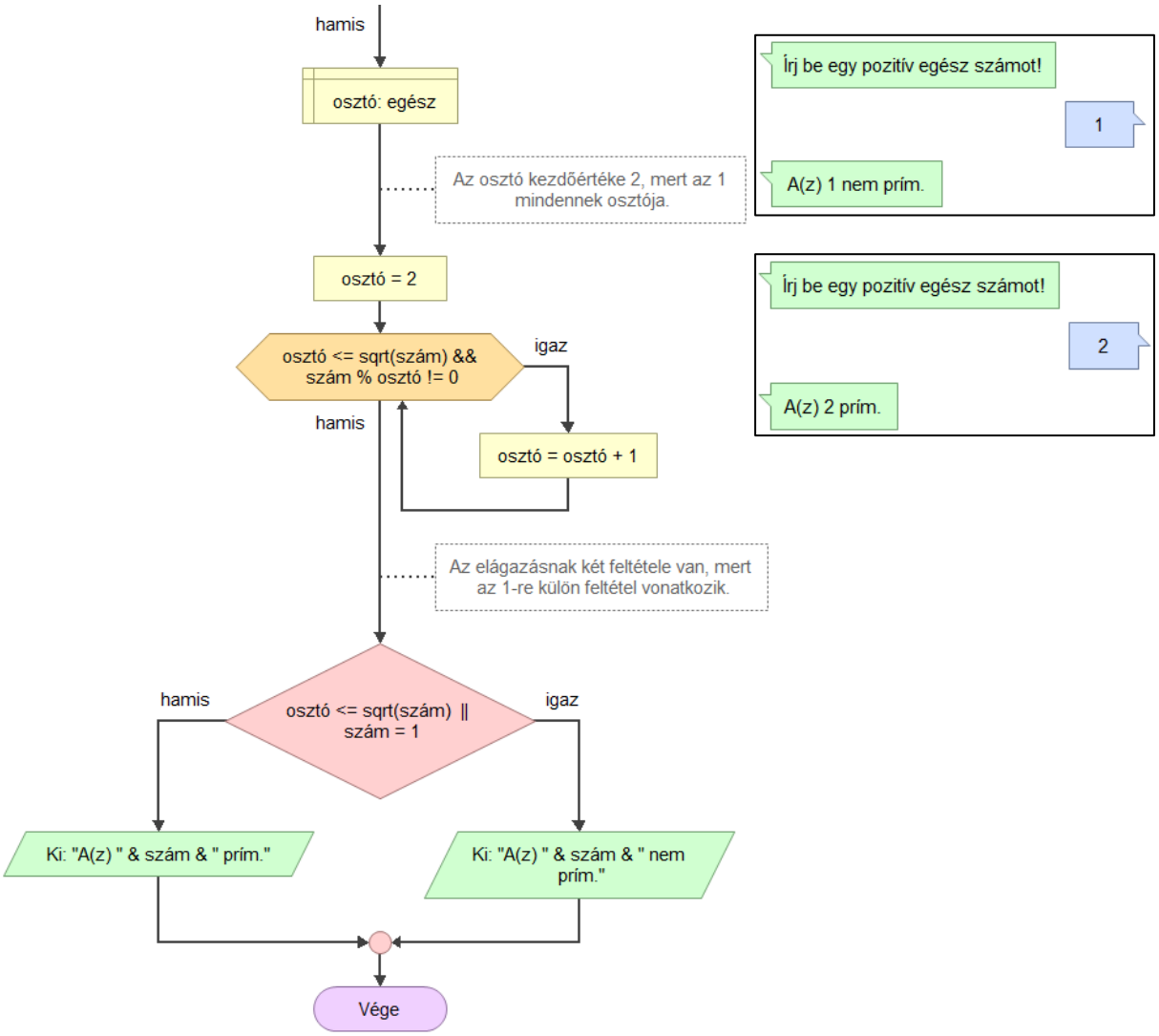
A feladat természetesen nem csak a 2-hatványaival, hanem más számok hatványaival is feladható.

12. Prímszám-e?

Olvassunk be egy pozitív egész számot és állapítsuk meg, hogy prímszám-e!

A feladat megoldásához szükség van némi számelméleti ismeretekre. A megoldás során kihasználjuk azt, hogy minden pozitív egész szám osztója az 1 és önmaga. Azt vizsgáljuk, hogy a vizsgált számnak van-e más osztója az 1-en és önmagán kívül, vagyis van-e olyan szám, amivel osztva 0-t ad maradékul. Ha van, akkor a szám nem prímszám, ha nincs, akkor pedig prímszám. Az 1-re speciális helyzete miatt különösen figyelnünk kell a megoldás során, mert csak egy osztója van és nem prímszám.





A diákok matematikai ismereteitől függően érdemes eldönteni azt, hogy meddig keressük a lehetséges osztókat. A kevésbé hatékony megoldás az, ha *szám-1-ig* vizsgáljuk a lehetséges osztókat. Ennél valamivel jobb megoldás az, ha *a szám feléig* keresgélünk. (Ehhez az osztópárok ismeretére van szükség.) **A leghatékonyabb megoldás pedig az, ha a *szám négyzetgyökéig* vizsgáljuk, hogy van-e osztó az 1-en kívül.** (Ehhez a négyzetgyök fogalmának ismerete szükséges.)

A megfelelő szintű matematikai tudás esetén is gyakori, hogy a tanulóknak nem a leghatékonyabb megoldási lehetőség jut először eszükbe, és segítő kérdésekkel afelé kell terelgetni őket.

Új ismeretek:

- összetett feltétel elöltesztelő ciklusban
- beépített `sqrt` függvény

Hasonló feladatok:

- Egy pozitív egész szám 1-től és önmagától különböző legnagyobb/legkisebb osztójának meghatározása.
- Egy pozitív egész szám osztóinak számának/összegének meghatározása.

13. Euklideszi algoritmus

Olvassunk be két pozitív egész számot, majd határozzuk meg a két szám legnagyobb közös osztóját Euklideszi algoritmussal!

Az Euklideszi algoritmusban a nagyobb szám kisebb számmal való osztási maradékát számoljuk ki, majd a nagyobb számot a kisebb számra, a kisebb számot pedig az osztási maradékra cseréljük. Ezeket a lépéseket ismételjük egészen addig, ameddig az osztási maradék 0 nem lesz. A folyamat végén a kisebb szám lesz az eredeti két szám legnagyobb közös osztója. Ha a legnagyobb közös osztó 1, akkor azt mondjuk, hogy a számok relatív prímek.

A csoport matematikai ismereteitől függően a relatív prímség fogalmát (vagyis az utolsó elágazást) kihagyhatjuk a feladatmegoldásból.

Példák az algoritmusra:

megadott számok: 585 és 225

585 : 225 maradéka 135
225 : 135 maradéka 90
135 : 90 maradéka 45
90 : 45 maradéka 0

eredmény: 45

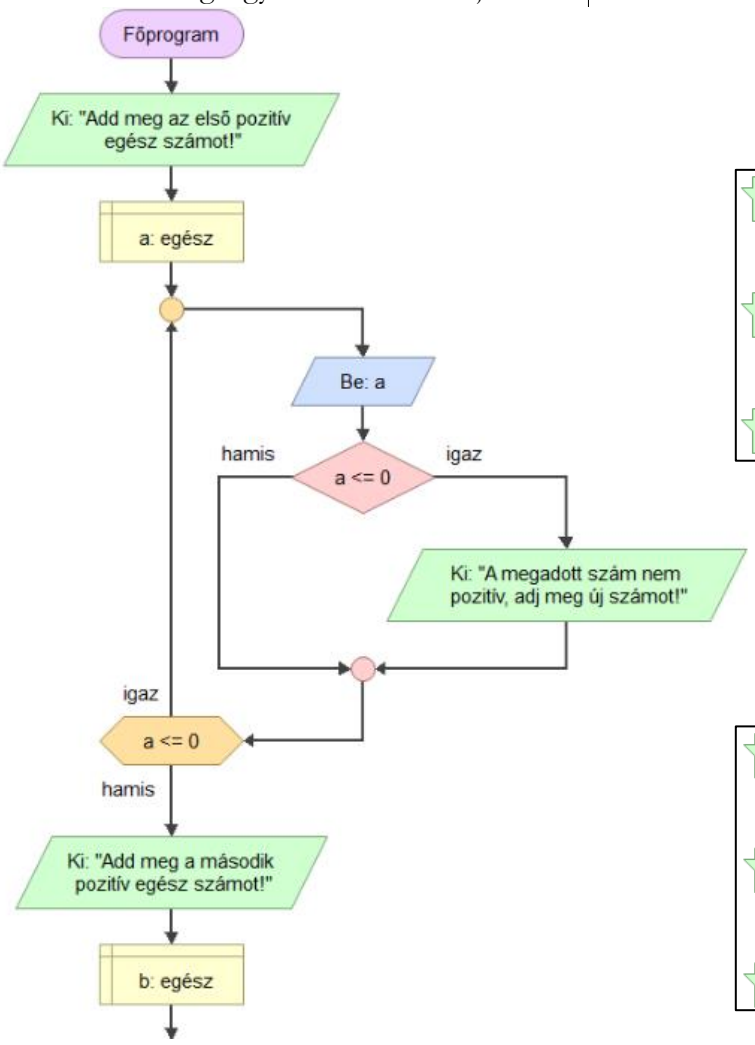
kimenet:
585 és 225 legnagyobb közös osztója: 45

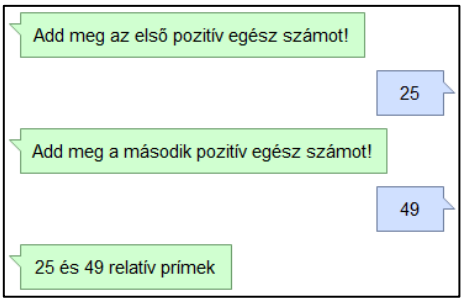
megadott számok: 52 és 35

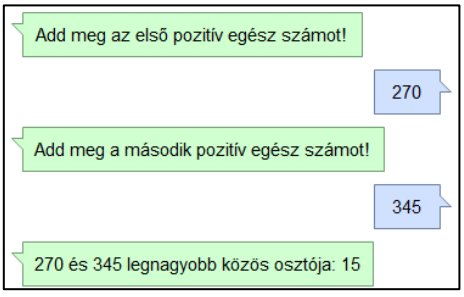
52 : 35 maradéka 17
35 : 17 maradéka 1
17 : 1 maradéka 0

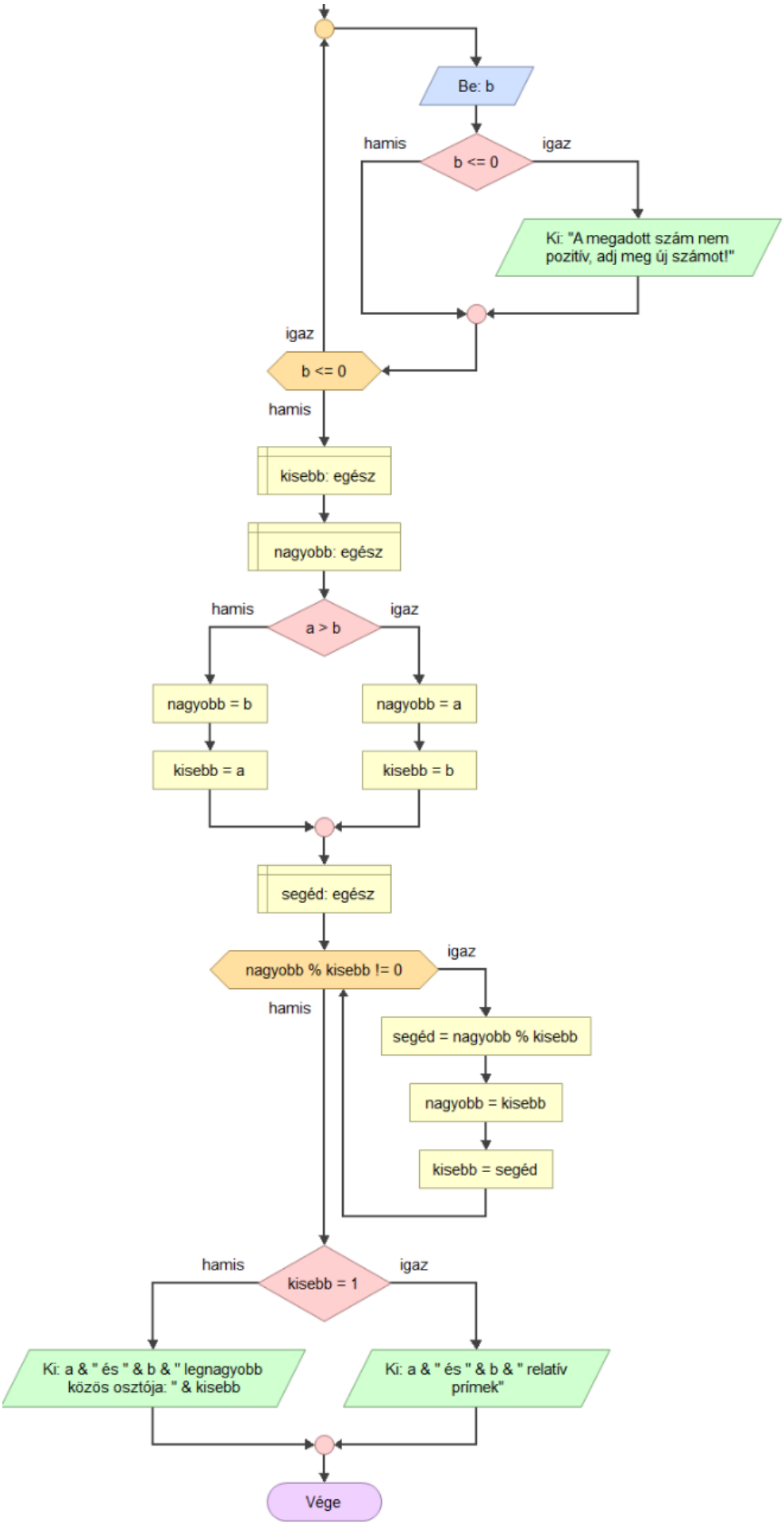
eredmény: 1

kimenet:
52 és 35 relatív prímek








Be: b
hamis
b <= 0
igaz
Ki: "A megadott szám nem pozitív, adj meg új számot!"
igaz
b <= 0
hamis
kisebb: egész
nagyobb: egész
hamis
a > b
igaz
nagyobb = b
nagyobb = a
kisebb = a
kisebb = b
segéd: egész
nagyobb % kisebb != 0
igaz
hamis
segéd = nagyobb % kisebb
nagyobb = kisebb
kisebb = segéd
hamis
kisebb = 1
igaz
Ki: a & " és " & b & " legnagyobb közös osztója: " & kisebb
Ki: a & " és " & b & " relatív prímek"
Vége

A feladat feladható úgy is, hogy a tanulóknak a megadott példák alapján kell kitalálniuk, hogy mi lehet a megfelelő algoritmus. Ebből is látszik az, hogy a tanulóknak nem kell ismerniük az Euklideszi algoritmust a feladat sikeres megoldásához.

Új ismeret:

- változók értékének cseréje (segédváltozó segítségével)

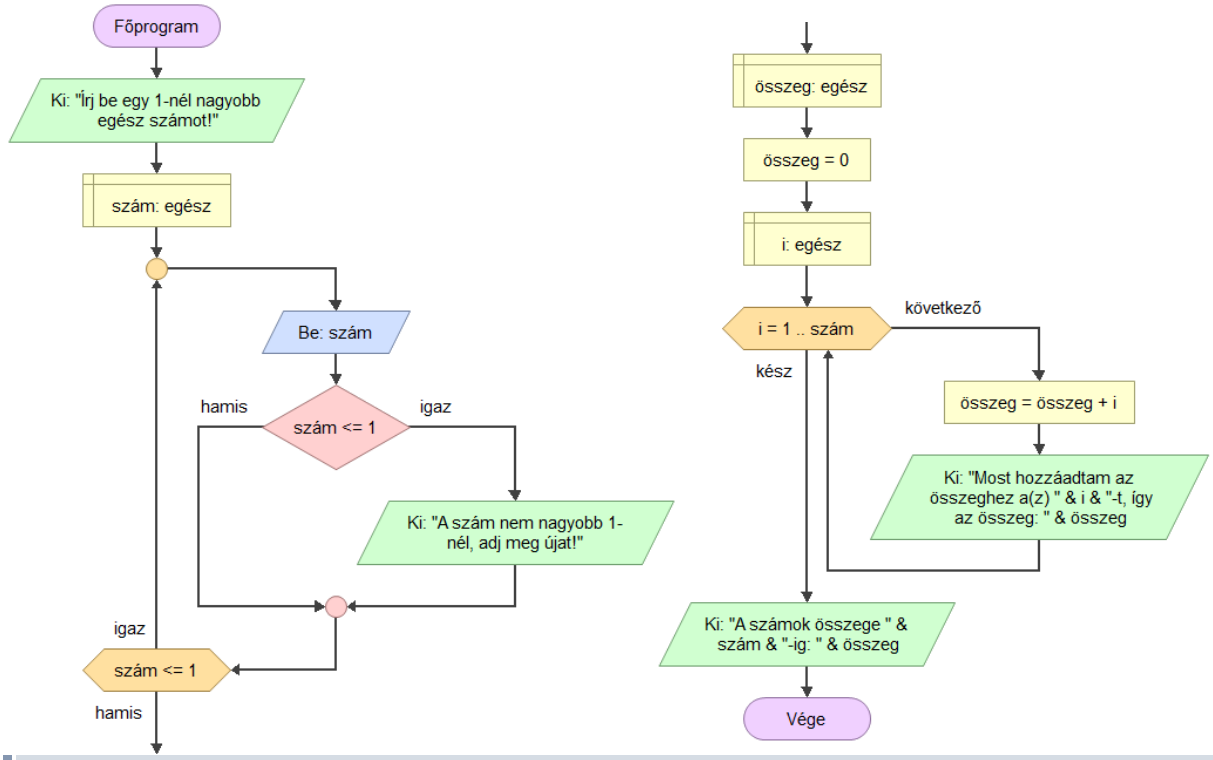
V. Számlálós ciklus

Ebben a részben az elöltesztelő és hátultesztelő ciklussal kapcsolatos feladatok után a harmadik fajta ciklushoz, a számlálós ciklushoz kapcsolódó feladatokat gyűjtöttünk össze. A tanítási folyamat során fontos kiemelni azt, hogy ez a ciklus abban különbözik az előző kettőtől, hogy már a ciklus elején tudjuk azt, hogy mi a ciklusváltozó kezdőértéke, illetve végértéke. Újdonság az is, hogy itt ciklusváltozóra is szükség van. A ciklusváltozó szerepe és értékének automatikus változásának megfigyeléséhez nagy segítséget ad az algoritmus lépésenkénti futtatása és a *Változók figyelése* ablak, melyek segítségével jól szemléltethető, hogy hol és hogyan változik a ciklusváltozó értéke.

A Flowgorithm abban különbözik a „kódolós” programozástól, hogy itt a ciklusváltozót a számlálós ciklus előtt, a cikluson kívül kell deklarálni. Nem tudjuk úgy létrehozni a ciklusváltozót, hogy az csak a cikluson belül legyen látható, használható. Ez szokatlan lehet majd a diákok számára a kódolásra való átálláskor.

14. Számok összege

Olvassunk be egy 1-nél nagyobb számot, majd határozzuk meg a pozitív számok összegét a felhasználó által megadott számig!



A számlálós ciklus magjában lévő kiírás itt csupán a megértést segíti, faktoriális számítása esetén viszont nagyon jól jön az egész típus korlátosságának felismeréséhez.

Új ismeretek:

- számlálós ciklus
- ciklusváltozó szerepe és használata

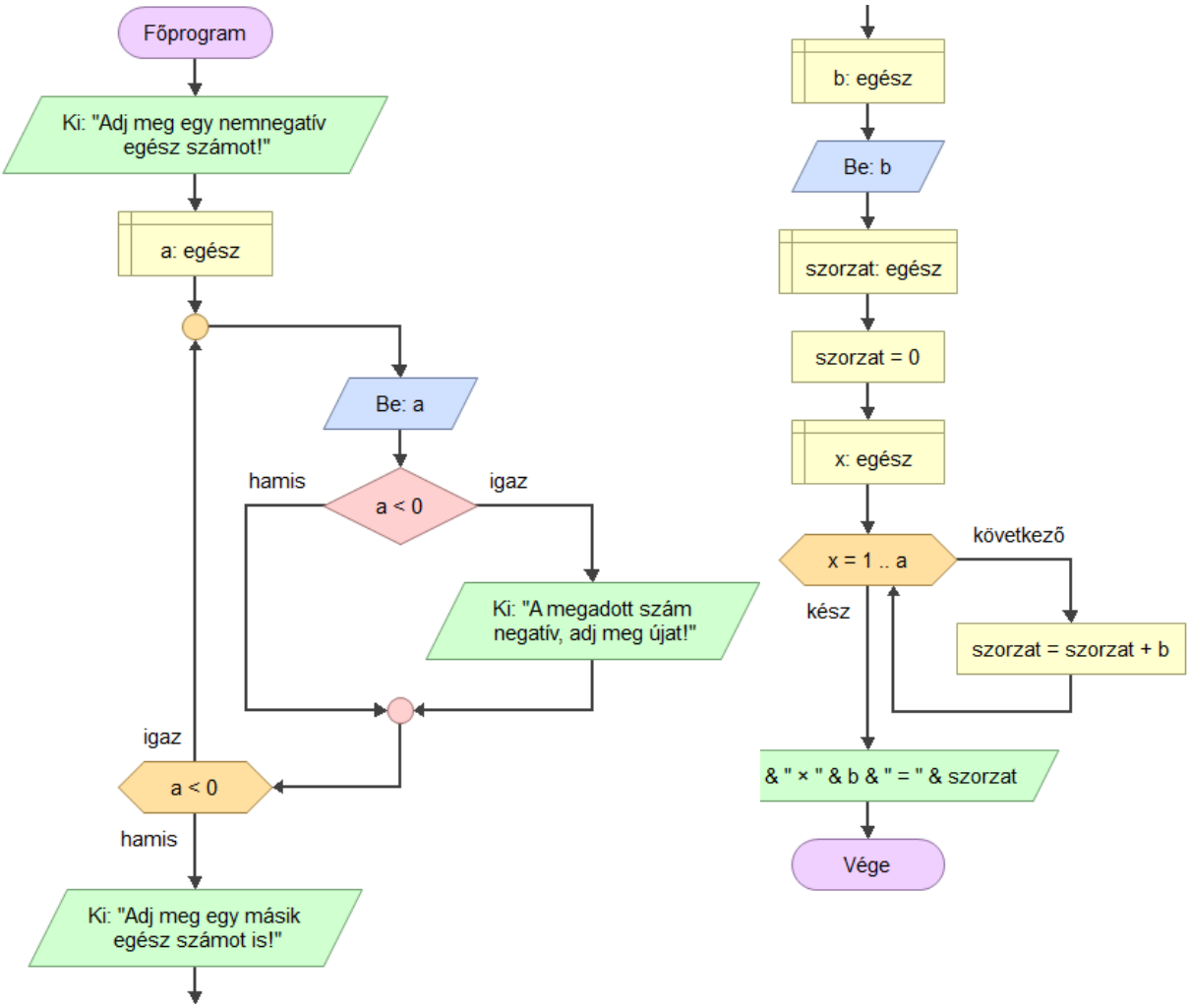
Hasonló feladatok:

- A számok összege helyett a szorzatukat, vagyis faktoriálist is számolhatunk.
- A feladat módosítható úgy is, hogy csak bizonyos feltételeknek megfelelő (pl. 3-mal osztható) számok összegét/szorzatát határozzuk meg.

15. Szorzás, mint ismételt összeadás

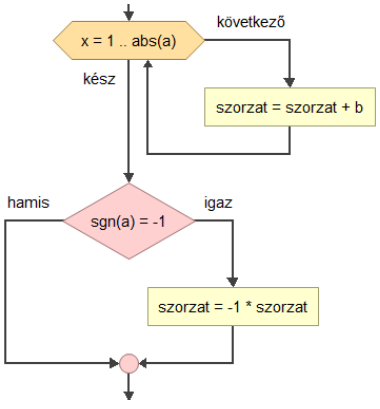
Olvassunk be két egész számot, és határozzuk meg a szorzatukat a szorzás művelet használata nélkül! Az elsőként beolvasott szám ne legyen negatív!

A feladat megoldása során azt használjuk ki, hogy a szorzás ismételt összeadást jelöl. Vagyis azt, hogy például `5·3=3+3+3+3+3`.



Módosítási lehetőség:

A megoldás módosításával elérhető az is, hogy az elsőként beolvasott számnál is megadható legyen negatív szám is. Ekkor az első szám abszolútértékét kell megadni a számlálós ciklus végértékeként, a ciklus után pedig szükség van még egy elágazásra, ahol a negatív `a` szám esetén ellentettjére változtatjuk a `szorzat` változó értékét.

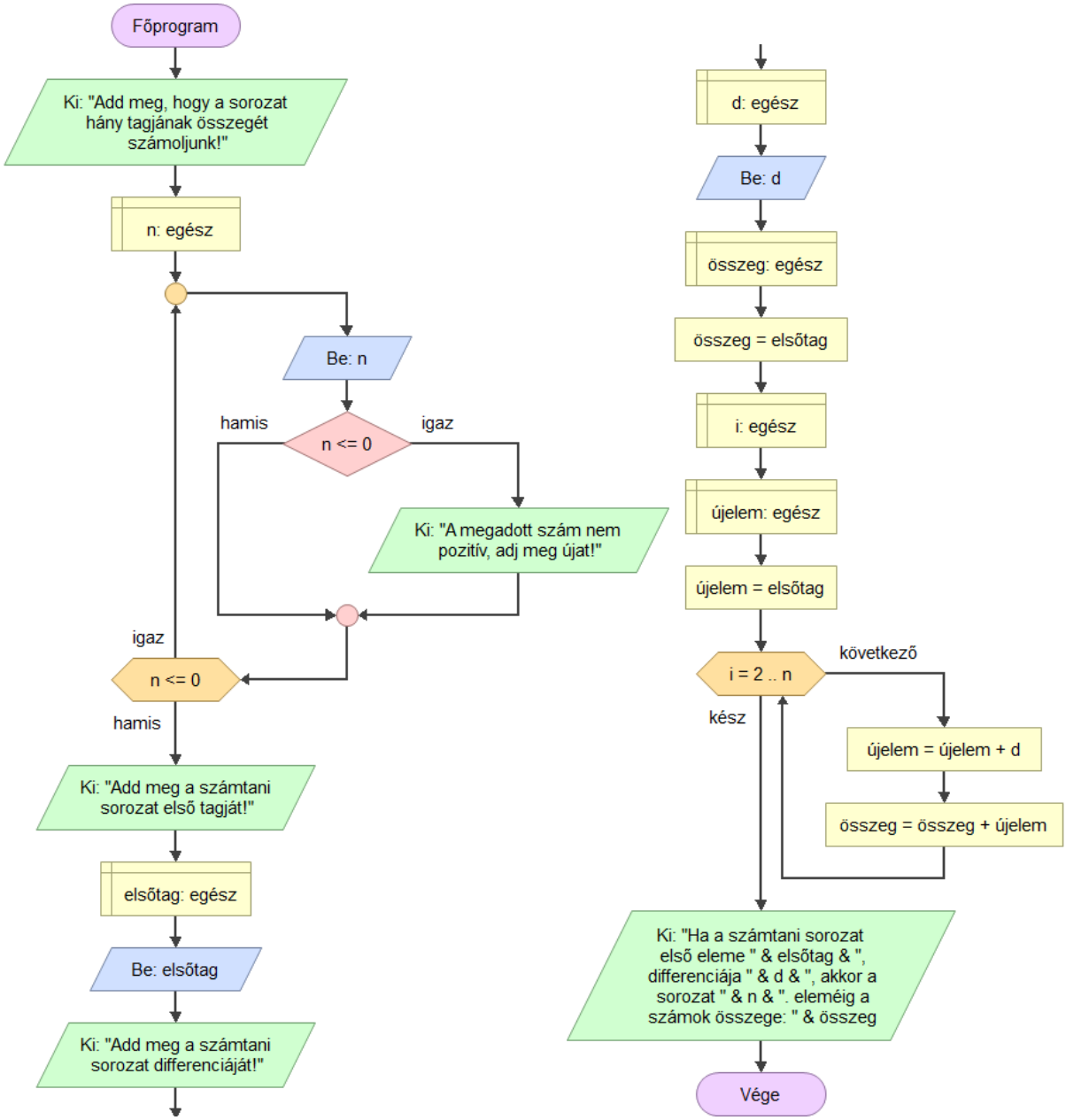


Új ismeret:

- beépített `sgn` függvény (enélkül is megoldható)

16. Számtani sorozat első n tagjának összege

Kérjünk be a felhasználótól egy számtani sorozatot annak első tagjával és differenciájával, illetve egy `n` egész számot, majd számítsuk ki a sorozat első `n` tagjának összegét!

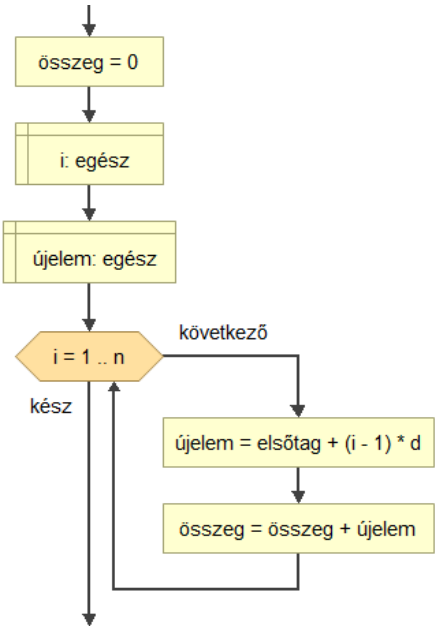


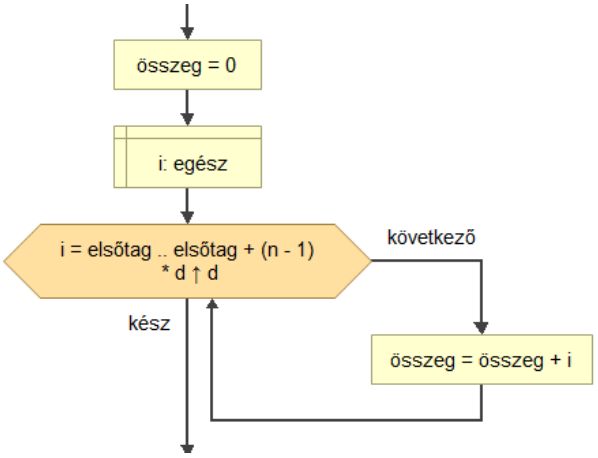
A feladat többféleképpen megoldható. A következő megoldásban azt használtuk ki, hogy a számtani sorozat tagjait (a második tagtól kezdve) úgy számíthatjuk ki, hogy a sorozat előző tagjához hozzáadjuk a sorozat differenciáját.

```
Ha a számtani sorozat első eleme 10, differenciája 3, akkor a sorozat 6. eleméig a számok összege: 105
```

Az alábbi két módosított megoldásban pedig azt használjuk ki, hogy a sorozat `n`-edik tagja kiszámítható az első tag és a differencia ismeretében `elsőtag+(n-1)·d` képlettel.

Ezeknél a megoldásoknál az algoritmus eleje (a beolvasás és az `összeg` változó deklarálása) és az eredmény kiírásának módja megegyezik a fenti megoldással, ezért csak a köztes részhez tartozó algoritmusokat tüntetjük itt fel.





Új ismeretek:

- számlálós ciklus 1-től különböző lépésközzel

A harmadik megoldási lehetőség kapcsán érdemes megemlíteni, illetve megmutatni a tanulóknak, hogy a ciklus lépésköze negatív szám is lehet.

VI. Tömbök

Az alapvető utasítások megismerése után a tömbökkel kapcsolatos feladatok következnek. Mivel a Flowgorithm-ben (a 2.22.1-es verzióban még) nem lehet adatokat fájlból beolvasni, és nem tudunk konstans tömböt létrehozni úgy, hogy egyszerre több tömbelemnek is értéket adunk, ezért elsősorban olyan feladatokat válogattunk össze, ahol a tömb elemeit a felhasználó adja meg, valamilyen szabály alapján számítjuk vagy generáljuk.

A tömbök elemeit 0-től kell indexelni. Erre a *Változók figyelése* ablak és a jól érthető hibaüzenetek segítségével könnyen rá lehet vezetni a tanulókat. A feladatok kapcsán viszont nagyon kell figyelni arra, hogy például a beolvasáskor ne 0., hanem 1. elemként hivatkozzunk az elsőként bekért értékre.

Tanárként érdemes eldönteni azt, hogy a ciklusváltozót 0-tól vagy 1-től indítjuk, majd ezt tudatosan, egységesen alkalmazni a feladatok megoldásakor. A döntés kapcsán a diákok véleményét is kikérhetjük, megkérdezhetjük azt, hogy ők melyik lehetőséget érzik természetesebbnek, melyiket választják. Fontosnak tartom, hogy ők is tudjanak arról, hogy ez egy programozói döntés és bármelyik lehetőség választható, nincs jó vagy rossz.

Ha a 0 mellett döntünk, akkor minden olyan esetben, amikor a felhasználónak írjuk ki egy tömbelem sorszámát, akkor 1-et hozzá kell adnunk a ciklusváltozó értékéhez.

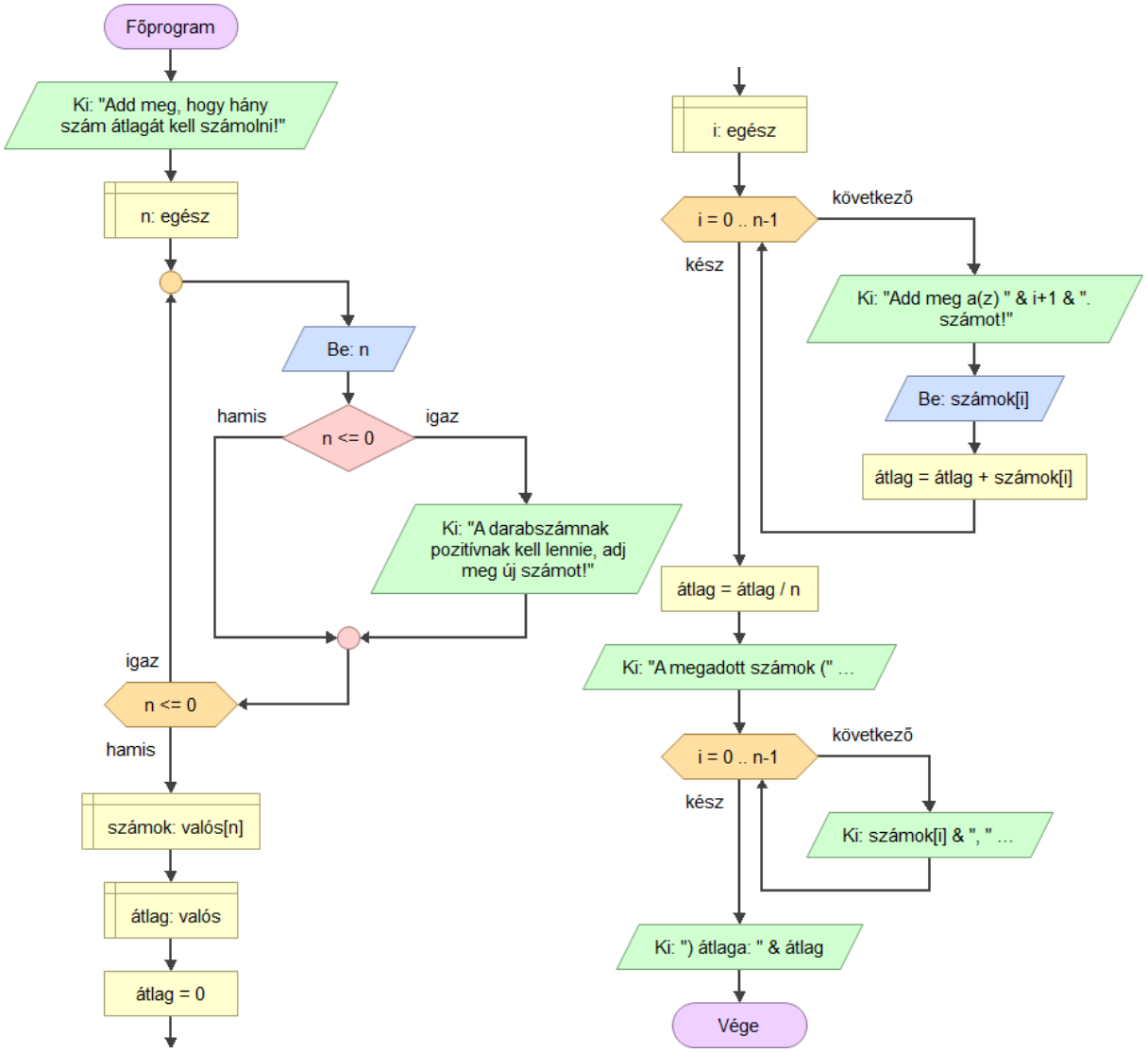
Ha az 1 mellett döntünk, akkor pedig minden tömbelemre való hivatkozáskor 1-et le kell vonnunk a ciklusváltozó értékéből.

Ebben az anyagban mindkét lehetőségre mutatunk példát a feladatok megoldása során. (A következő feladatnál 0-ról indítjuk a ciklusváltozót, az azt követően pedig 1-ről.)

17. Számok átlaga

Olvassunk be a felhasználótól `n` db számot, majd határozzuk meg a számok átlagát! Az átlag mellett a felhasználó által megadott számokat is soroljuk fel! (Az `n` egy pozitív egész szám, amit szintén a felhasználó ad meg a futtatás során.)

A feladat megoldása során figyelni kell a változók típusaira, illetve az egész típusú számok osztására.



Az utolsó kiírásban `ToFixed(átlag,2)`-re cserélve az `átlag` változó értékét két tizedesjegy pontossággal jeleníthetjük meg az eredményt.

```
Add meg a(z) 4. számot!
16
Add meg a(z) 5. számot!
2.8
A megadott számok (2.5, 6, 4.11, 16, 2.8, )
átlaga: 6.282
```

A tömb elemeinek beolvasása és az átlag számítása akár külön ciklusba is tehető lenne.

Új ismeretek:

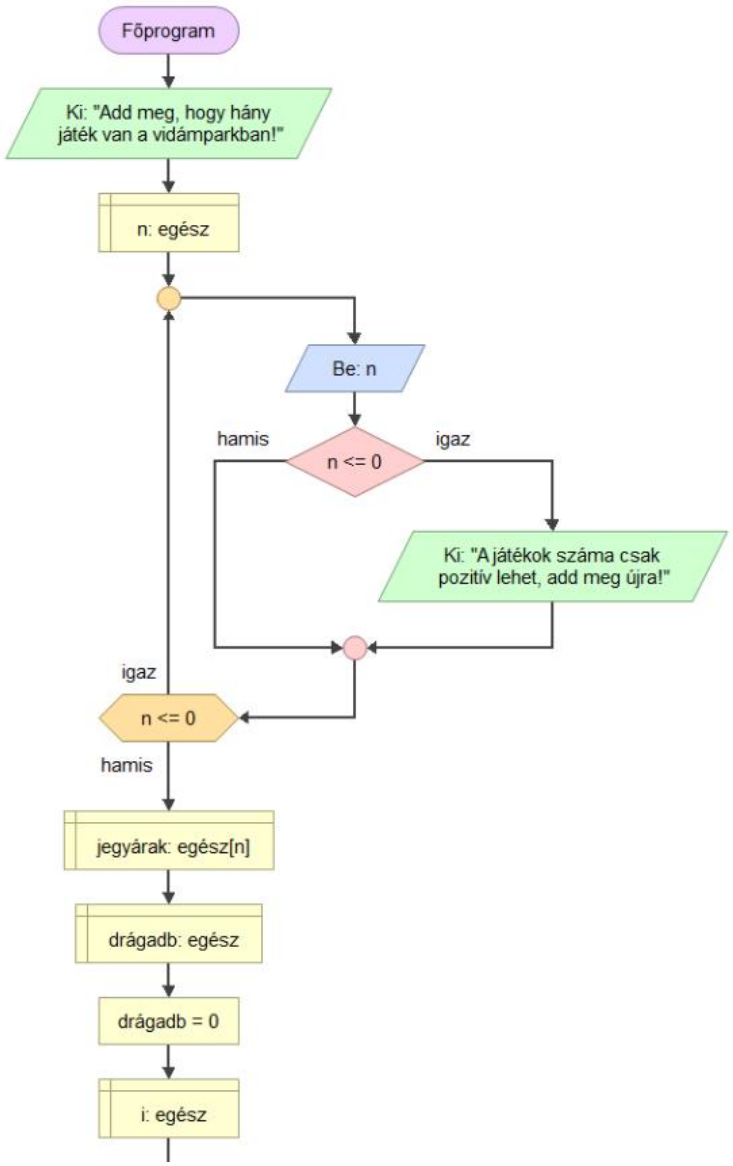
- tömbök (tömb létrehozása, elemek indexelése)
- művelet végzése a kiírásban
- sorozatszámítás feladattípus

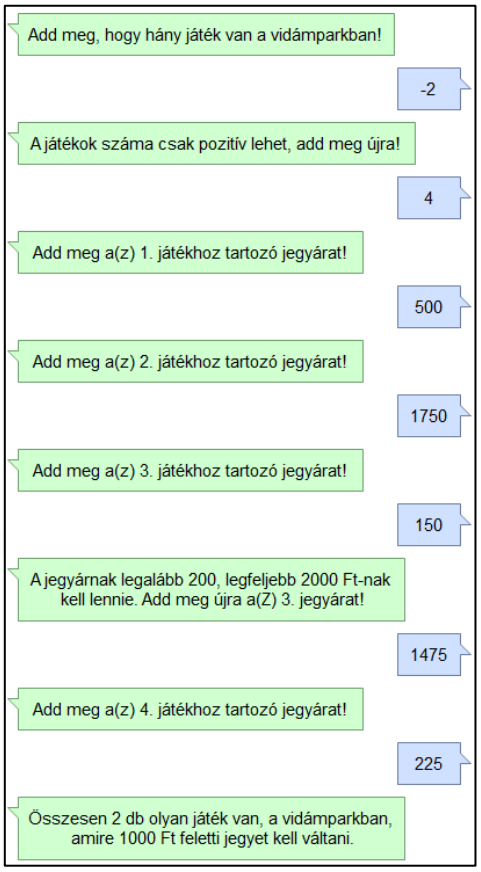
Hasonló feladatok:

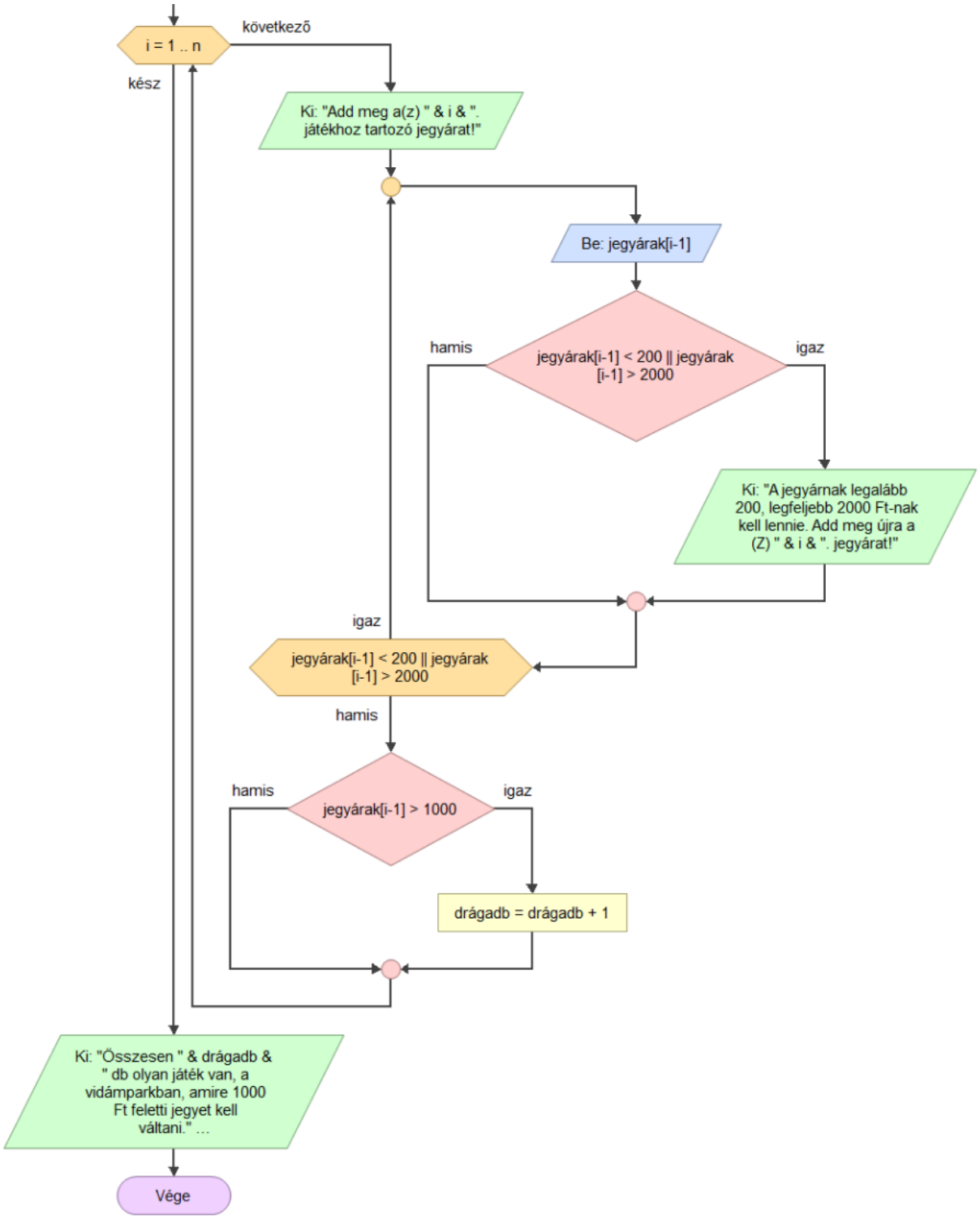
- Átlag helyett a számok összegét vagy akár szorzatát is számolhatjuk.
- A feladatot módosíthatjuk úgy is, hogy nem az összes tömbelemmel végzünk el valamilyen műveletet, csak azokkal, amelyek megfelelnek egy bizonyos feltételnek (pl. 100-nál nagyobbak, 3-mal oszthatók, …)
- A feladatot szöveges környezetbe is ágyazhatjuk. Lehetnek a számok például hőmérséklet értékek, osztályzatok vagy bármilyen adat, aminél van értelme az összegzésnek, átlagszámításnak vagy valamilyen feltételes számításnak.

18. Vidámpark (megszámolás)

Egy vidámparkban n darab játék van. Ezek mindegyikére külön-külön kell jegyet váltani. A játékokra váltható jegyek ára 200 és 2000 Ft között mozog. Olvasd be az adatokat (játékok száma, egyes játékok jegyeinek ára), majd határozd meg, hogy hány olyan játék van, amelynek jegyára 1000 Ft felett van!







Új ismeret:

- megszámolás feladattípus

Hasonló feladatok:

A feladathoz tartozó kerettörténet és a megszámolás feltétele is számtalan módon változtatható a csoport érdeklődési körének megfelelően.

19. Legmelegebb nap

Egy településen n napon keresztül mérték a hőmérsékletet. Készíts programot, amely beolvassa az adatokat (napok száma, napi hőmérsékletek), majd megadja a legmagasabb mért értéket és azt is, hogy mely napon vagy napokon mérték azt az értéket! (A mért értékeknek -50 és 50°C között kell lenniük.)

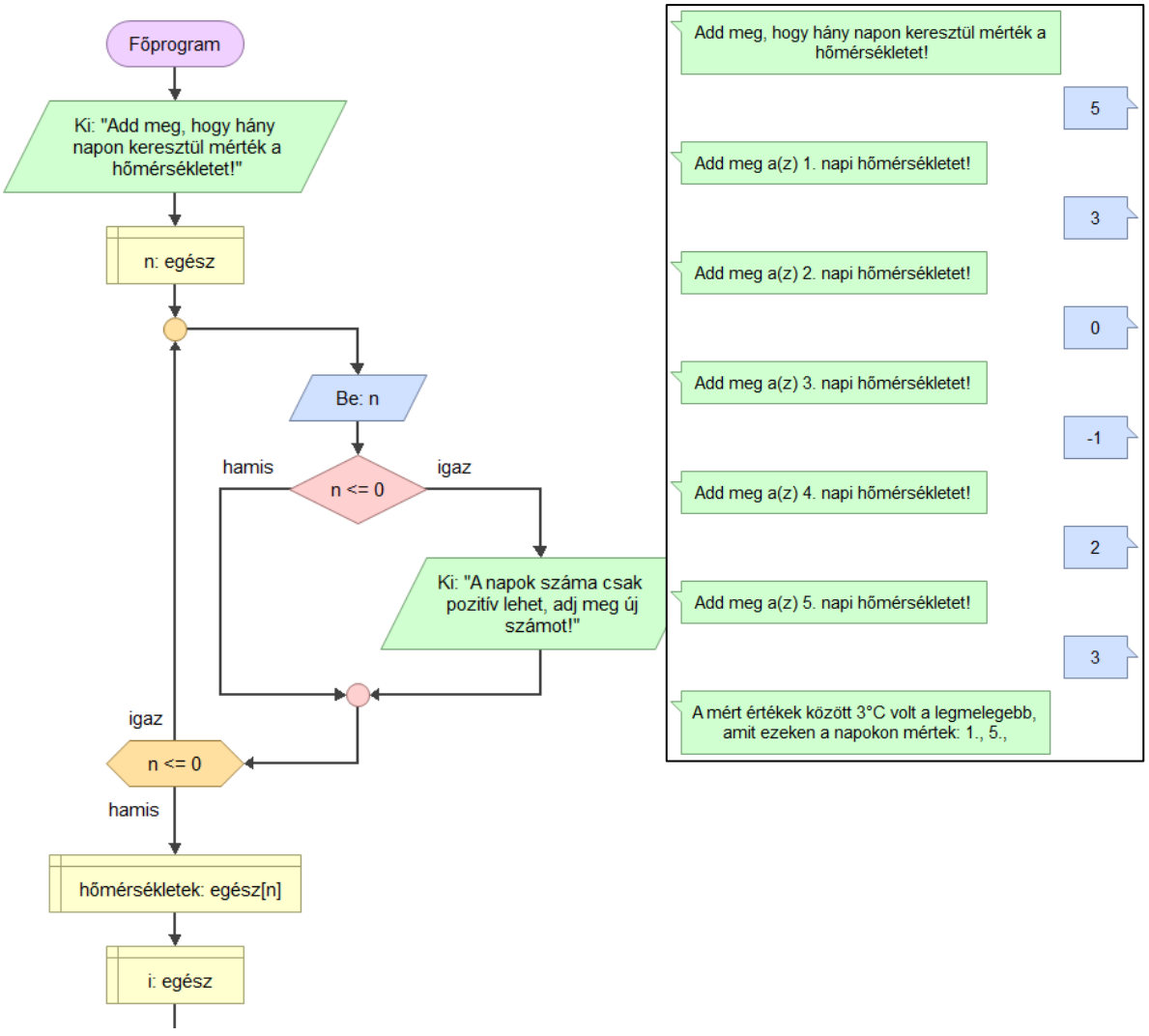
A megoldás hatékonyabbá tehető, ha a `maxérték` keresése során eltároljuk a legkisebb olyan tömbindexet, ahol maximális érték van. Ekkor a végén a napok sorszámának kiírásakor már csak az ezt követő értékeket kell megvizsgálni a számlálós ciklusban.

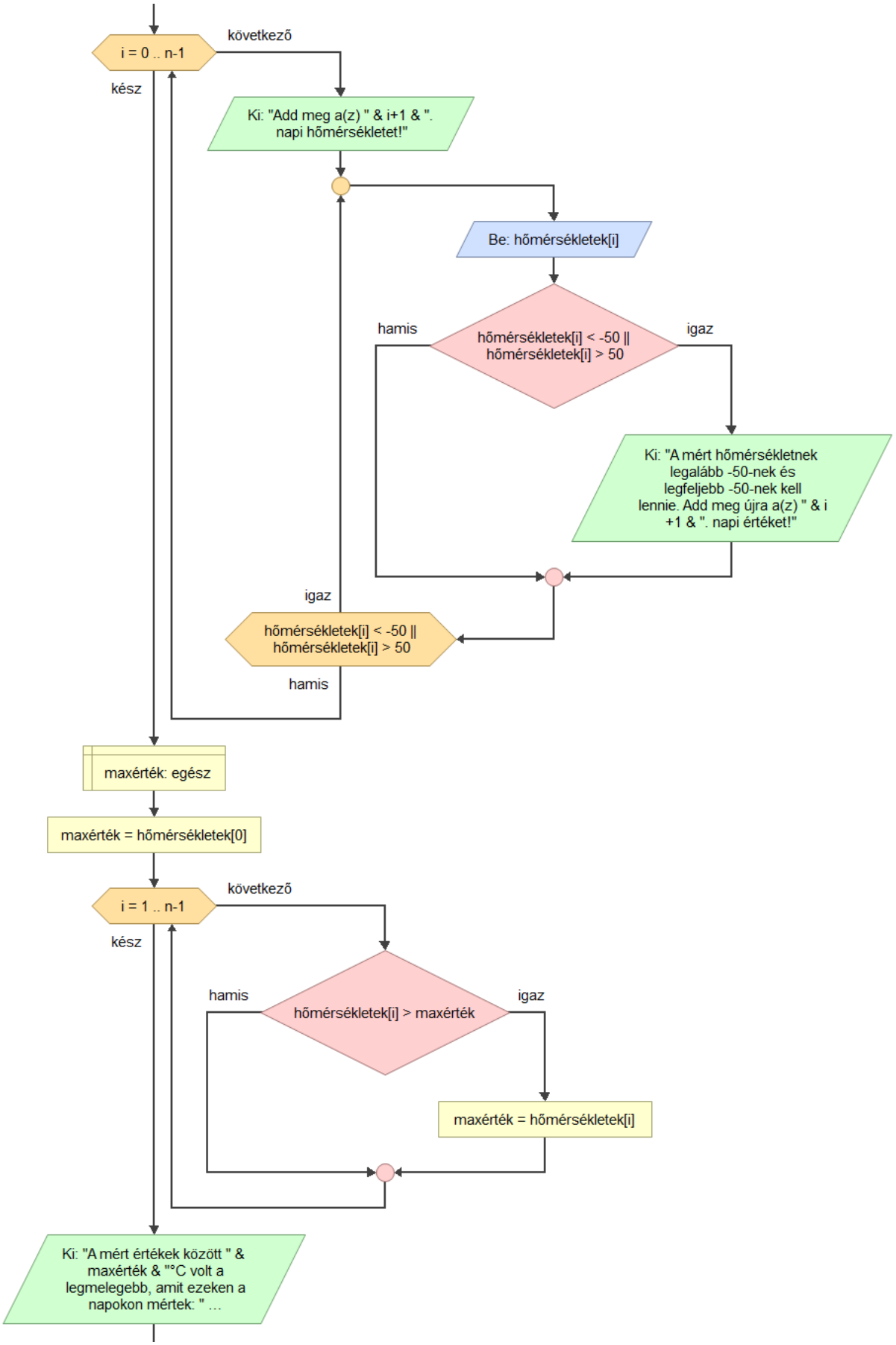
Egy másik lehetőség a maximális értékek számának, illetve a megfelelő indexek tárolása lehetne. A maximális tömbelemek indexének tárolásával viszont sok memóriát használnánk feleslegesen, mert nem tudjuk előre, hogy hány értéket kell majd tárolni, így feleslegesen nagy tömböt hoznánk létre neki.

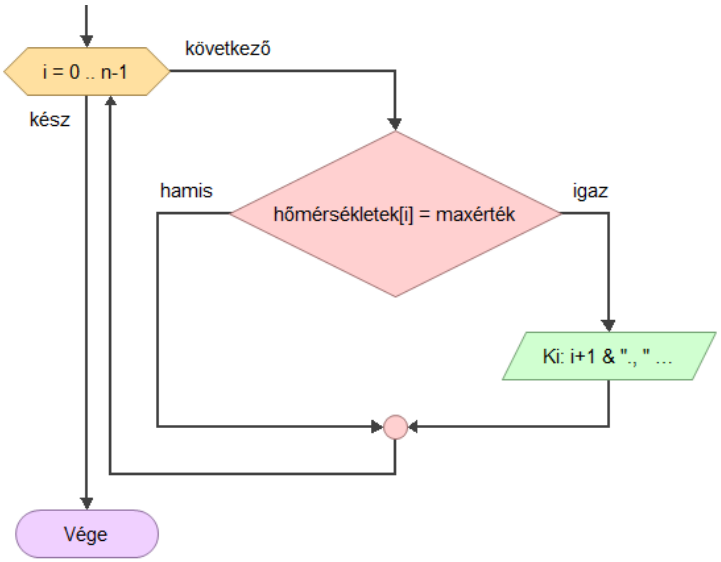
A Flowgorithm-ben a tömb méretét mindig meg kell adnunk annak létrehozásakor, így nem tudunk dinamikus méretű tömbbel dolgozni.

Új ismeretek:

- maximumkiválasztás feladattípus
- kiválogatás feladattípus




i = 0 .. n-1
következő
kész
Ki: "Add meg a(z) " & i+1 & ". napi hőmérsékletet!"
Be: hőmérsékletek[i]
hőmérsékletek[i] < -50 || hőmérsékletek[i] > 50
hamis
igaz
Ki: "A mért hőmérsékletnek legalább -50-nek és legfeljebb -50-nek kell lennie. Add meg újra a(z) " & i +1 & ". napi értéket!"
hőmérsékletek[i] < -50 || hőmérsékletek[i] > 50
igaz
hamis
maxérték: egész
maxérték = hőmérsékletek[0]
i = 1 .. n-1
következő
kész
hőmérsékletek[i] > maxérték
hamis
igaz
maxérték = hőmérsékletek[i]
Ki: "A mért értékek között " & maxérték & "°C volt a legmelegebb, amit ezeken a napokon mértek: " ...

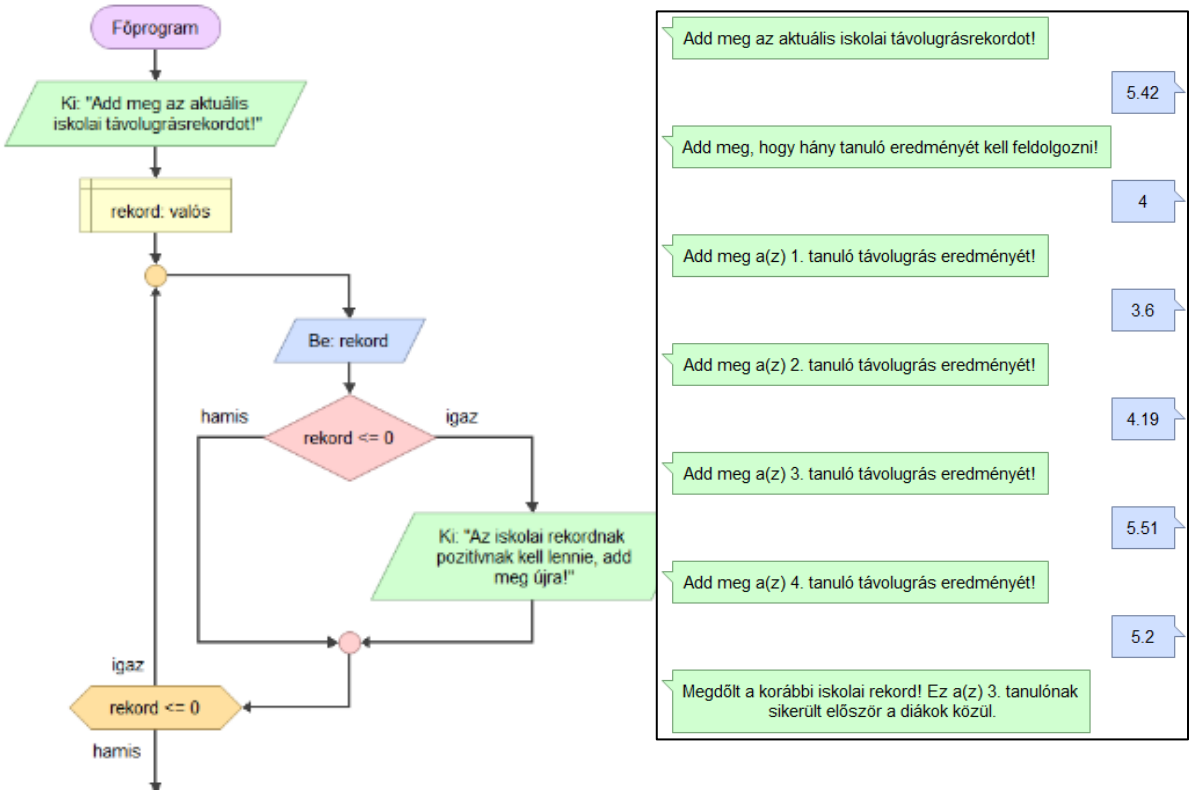


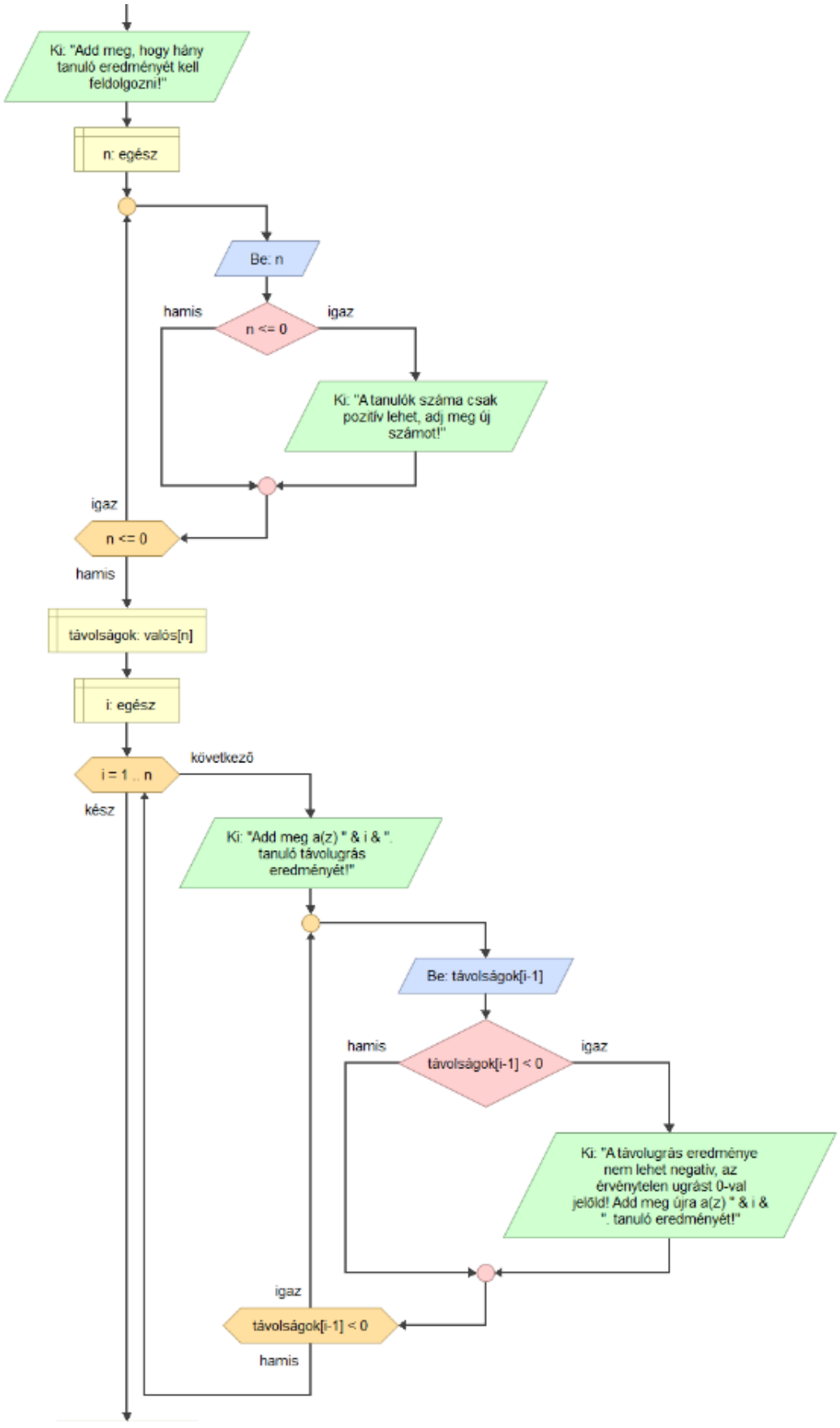
Hasonló feladatok:

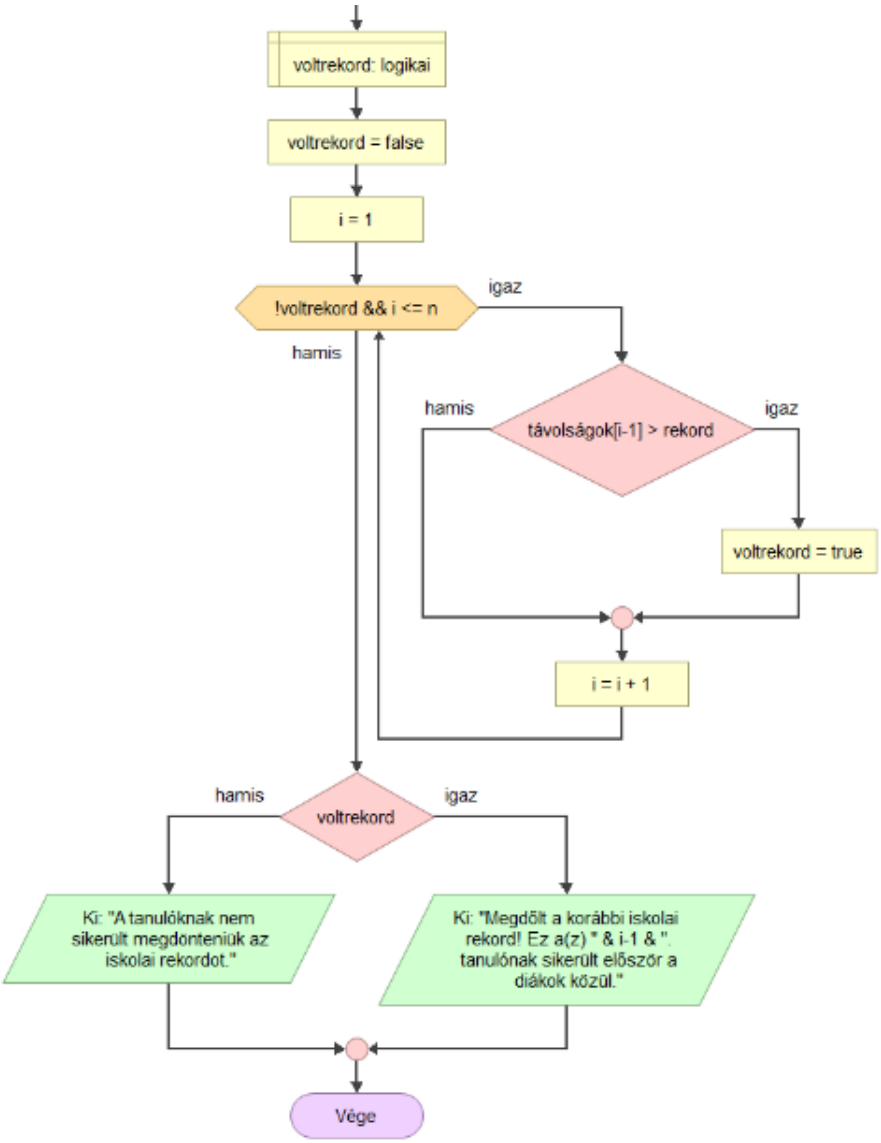
- Változtathatunk a feladathoz tartozó kerettörténeten, a maximumot minimumra cserélhetjük.
- Kereshetünk maximumot vagy minimumot bizonyos feltételeknek megfelelő tömbelemek között.

20. Távolugrás rekord

Egy iskolai tornaórán n gyerek távolugrás eredményét mérték le. Az iskolai rekord megdöntéséért külön elismerés jár. Készíts egy programot, amelyben beolvasod az iskolai rekordot és a tanulók eredményeit, majd határozd meg, hogy sikerült-e megdönteni az iskolai rekordot azon az órán! Ha sikerült, akkor írd ki, hogy hányadik tanulónak sikerült ez először.




Ki: "Add meg, hogy hány tanuló eredményét kell feldolgozni!"
n: egész
Be: n
hamis
n <= 0
igaz
Ki: "A tanulók száma csak pozitív lehet, adj meg új számot!"
igaz
n <= 0
hamis
távolságok: valós[n]
i: egész
i = 1 .. n
következő
kész
Ki: "Add meg a(z) " & i & ". tanuló távolugrás eredményét!"
Be: távolságok[i-1]
hamis
távolságok[i-1] < 0
igaz
Ki: "A távolugrás eredménye nem lehet negatív, az érvénytelen ugrást 0-val jelöld! Add meg újra a(z) " & i & ". tanuló eredményét!"
igaz
távolságok[i-1] < 0
hamis



Az elöltesztelő ciklus feltételében nem szerepelhet a `távolságok[i-1]`, mert ha a voltrekord változó kihagyásával `i<=n && távolságok[i-1]<rekord` alakban írjuk a feltételt, akkor hibaüzenetet kapunk a futtatás során. Ennek oka az, hogy akkor is kiértékelésre kerül a teljes logikai kifejezés, ha az *és* első fele hamis, és így a kifejezés második felének kiértékelése nélkül is tudjuk, hogy a teljes kifejezés is hamis biztosan hamis lesz. Vagyis abban az esetben, ha nem találunk a feltételnek megfelelő tömbelemet, és az `i<=n` már hamis, akkor a `távolságok[i-1]`-ben az `i` helyére `n+1` kerül behelyettesítésre, vagyis a program a távolságok tömb `n`-edik elemét próbálja a `rekord`hoz hasonlítani, de a tömbnek nincs olyan eleme, amelynek indexe `n`.

Új ismeretek:

- keresés feladattípus
- kiválasztás feladattípus

Hasonló feladatok:

A korábbi tömbös feladatokhoz hasonlóan ezt is nagyon sokféle változatban feladhatjuk a diákoknak. Emellett pedig a különböző feladattípusok kombinálására is lehetőségünk van.

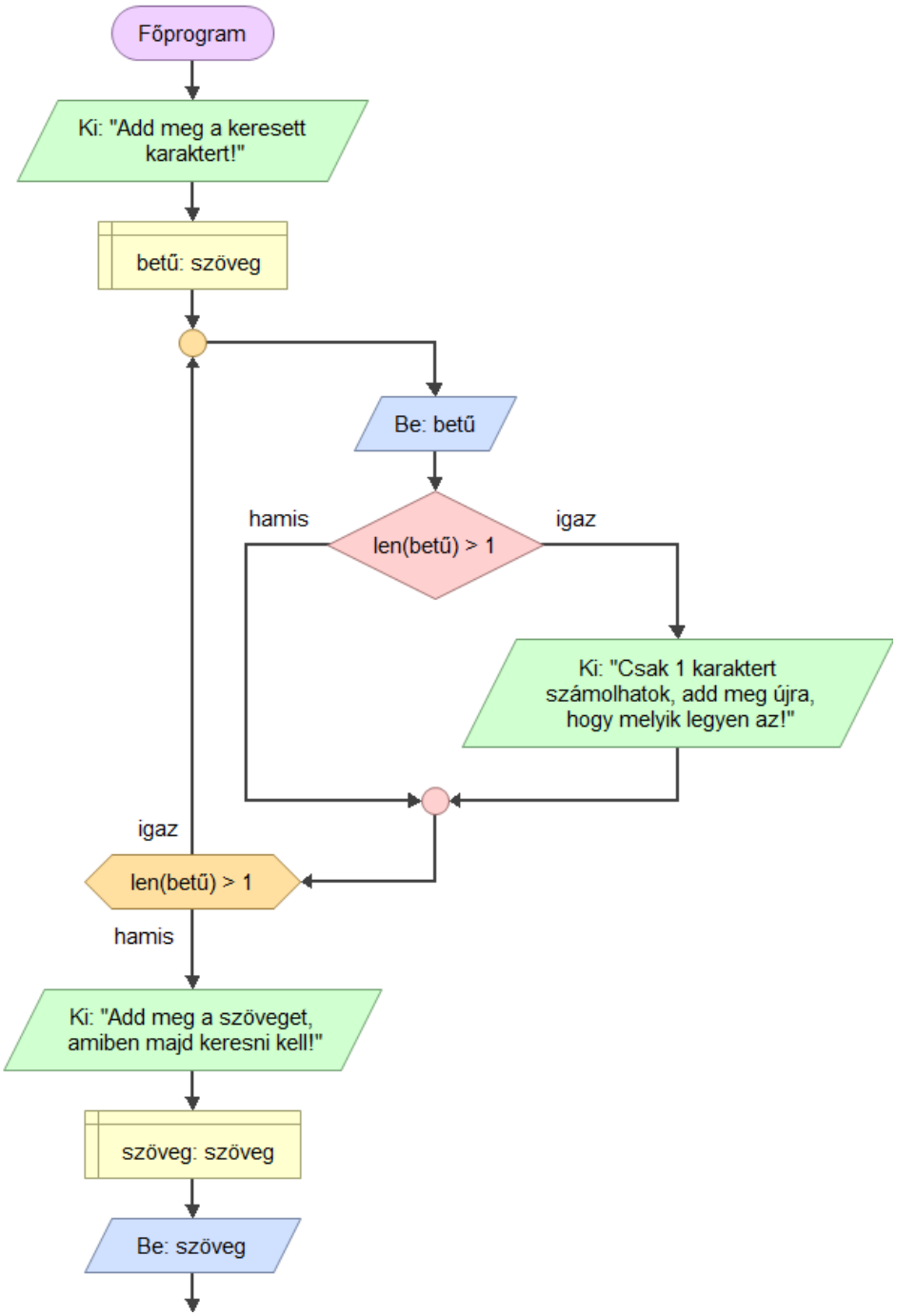
VII. Szöveges változókkal kapcsolatos feladatok

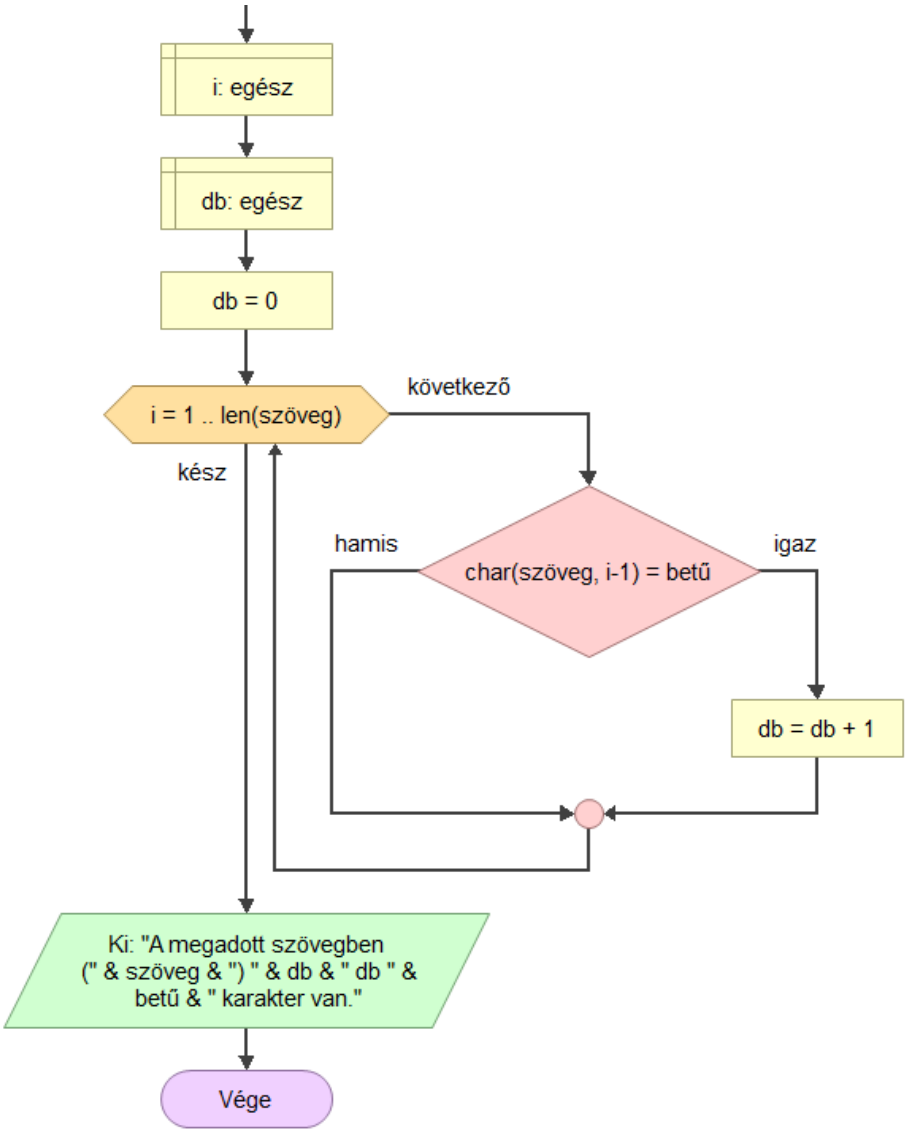
A matematika iránt kevésbé érdeklődő diákok nagy örömére a szöveges változókhoz kapcsolódóan is változatos feladatokat állíthatunk össze.

A szöveg típusú változók valójában karakterekből álló tömbök. Ebből adódóan a szöveges változók egy-egy karakterére való hivatkozás a tömbök egy-egy elemére való hivatkozáshoz hasonlóan történik. A szöveges változók első karakterére is a 0 index segítségével tudunk hivatkozni.

21. Betűkereső

Készítsünk egy programot, amelyben bekérünk egy betűt (karaktert) és egy szöveget a felhasználótól, majd vizsgáljuk meg, hogy hányszor szerepel megadott szövegben a keresett karakter!





A felhasználó által megadott `betű` változó speciális karakter is lehet. A tanulók maguktól is rá szoktak jönni, de érdemes megemlíteni is, hogy míg a számoknál könnyen tudjuk ellenőrizni, hogy megfelel-e a feltételeknek, addig a beolvasott szöveg értelmességét nem tudjuk garantálni.

A kis- és nagybetűk összehasonlításkor különbözőnek számítanak, vagyis pl. `"a"="A"` hamis.

Új ismeretek:

- szöveges változó, mint karakterek tömbje (0-tól kell indexelni)
- egy karakterből álló szöveg beolvasása
- beépített `len` és `char` függvények

Hasonló feladatok:

- Egyszerűsített változatban a keresendő karakter előre adott (szóköz esetén szavak száma).
- Nehezítésként kereséssé alakíthatjuk a feladatot az előző tömbös példához hasonlóan.
- A beolvasott szöveget egy szóra is korlátozhatjuk. (Ekkor a beolvasáskor arra kell figyelni, hogy a szöveg ne tartalmazzon szóközt.)
- További nehezítés, ha nem egy karaktert, hanem egy szórészletet, szövegrészletet keresünk a beolvasott szövegben. Vagy a szavak (szóközök) számolásakor figyelünk a dupla szóközökre.

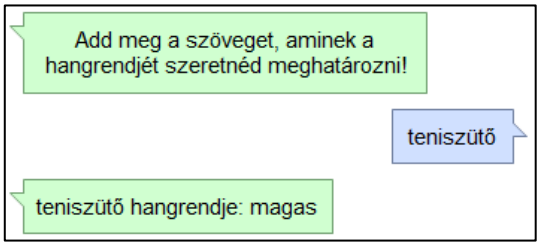
22. Hangrend

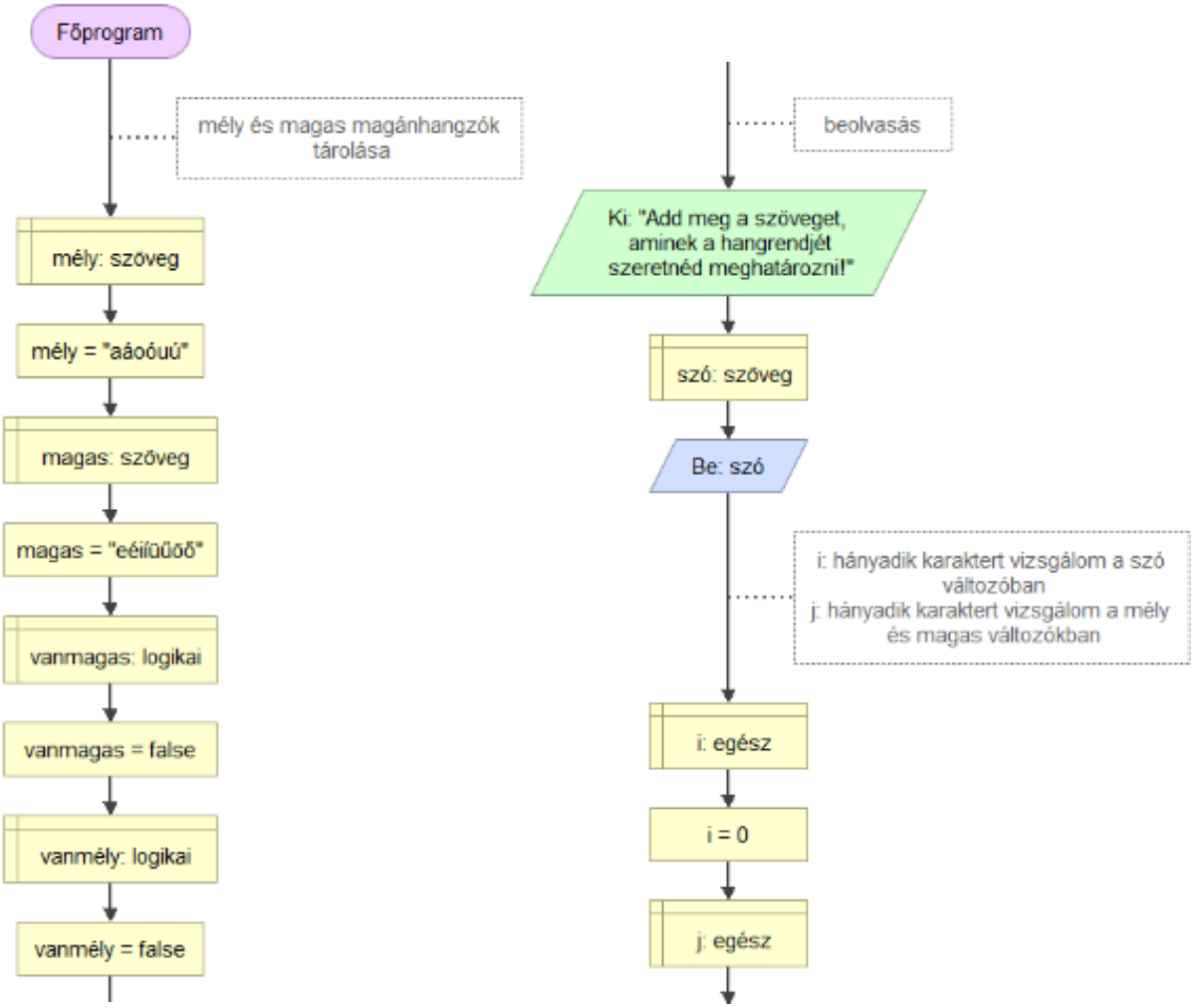
Olvassunk be egy szöveget, majd határozzuk meg, hogy milyen hangrendű (mély, magas vagy vegyes)!

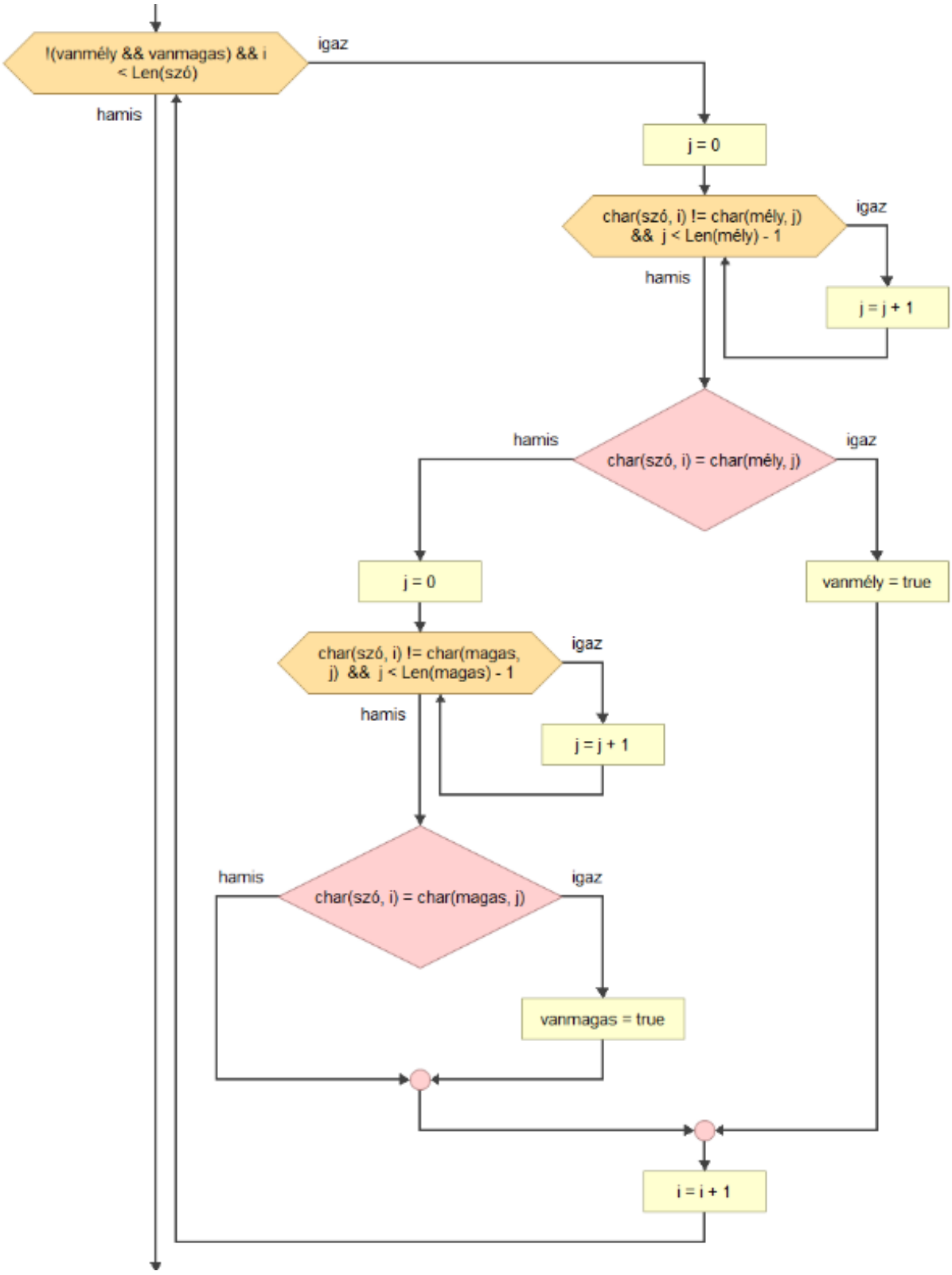
A feladat megoldásához fel kell elevenítenünk a hangrenddel kapcsolatos ismereteinket. A szó hangrendje a benne lévő magánhangzóktól függ. Magas magánhangzók az e, é, i, í, ö, ő, ü, ű. Mély magánhangzók az a, á, o, ó, u, ú. Ha egy szóban csak mély vagy csak magas magánhangzó van, akkor az mély, illetve magas hangrendű, míg ha mély és magas magánhangzó is előfordul a szóban, akkor azt vegyes hangrendűnek mondjuk.

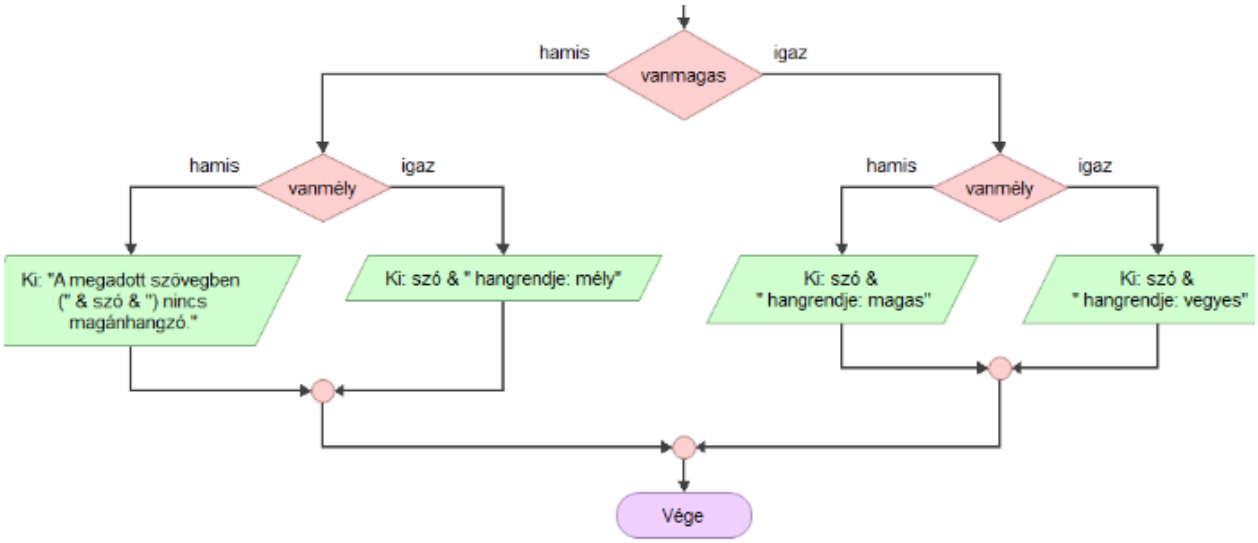
Mivel a Flowgorithm-ben hosszadalmasan tudnánk csak a megfelelő magánhangzókat tartalmazó tömböt létrehozni, ezért a tömb helyett egy-egy szöveges változóban tároljuk a mély és magas magánhangzókat, és karakterek tömbjeként dolgozunk vele. (Kihasználjuk azt, hogy minden magánhangzó egy karakter. A mássalhangzókkal nehezebb dolgunk lenne a két vagy három karakterből álló mássalhangzók miatt.

A megoldás során elöltesztelő ciklust használunk, mert ha már mély és magas hangrendű magánhangzót is találtunk, akkor nem kell megvizsgálnunk a többi karaktert, mert a szöveg már csak vegyes hangrendű lehet.






!(vanmély && vanmagas) && i < Len(szó)
igaz
hamis
j = 0
char(szó, i) != char(mély, j) && j < Len(mély) - 1
igaz
hamis
j = j + 1
hamis
char(szó, i) = char(mély, j)
igaz
j = 0
vanmély = true
char(szó, i) != char(magas, j) && j < Len(magas) - 1
igaz
hamis
j = j + 1
hamis
char(szó, i) = char(magas, j)
igaz
vanmagas = true
i = i + 1



Új ismeretek:

- tömbelemek összehasonlítása
- algoritmus több egymás utáni utasításának másolása (Ha eddig nem került rá sor, akkor itt kimondottan hasznos lehet.)

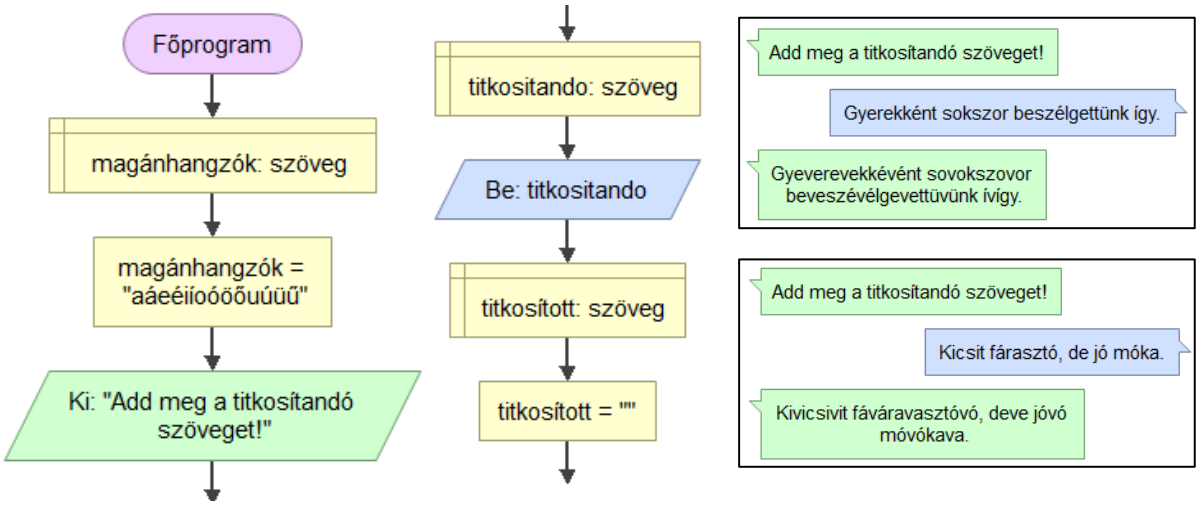
Hasonló feladatok:

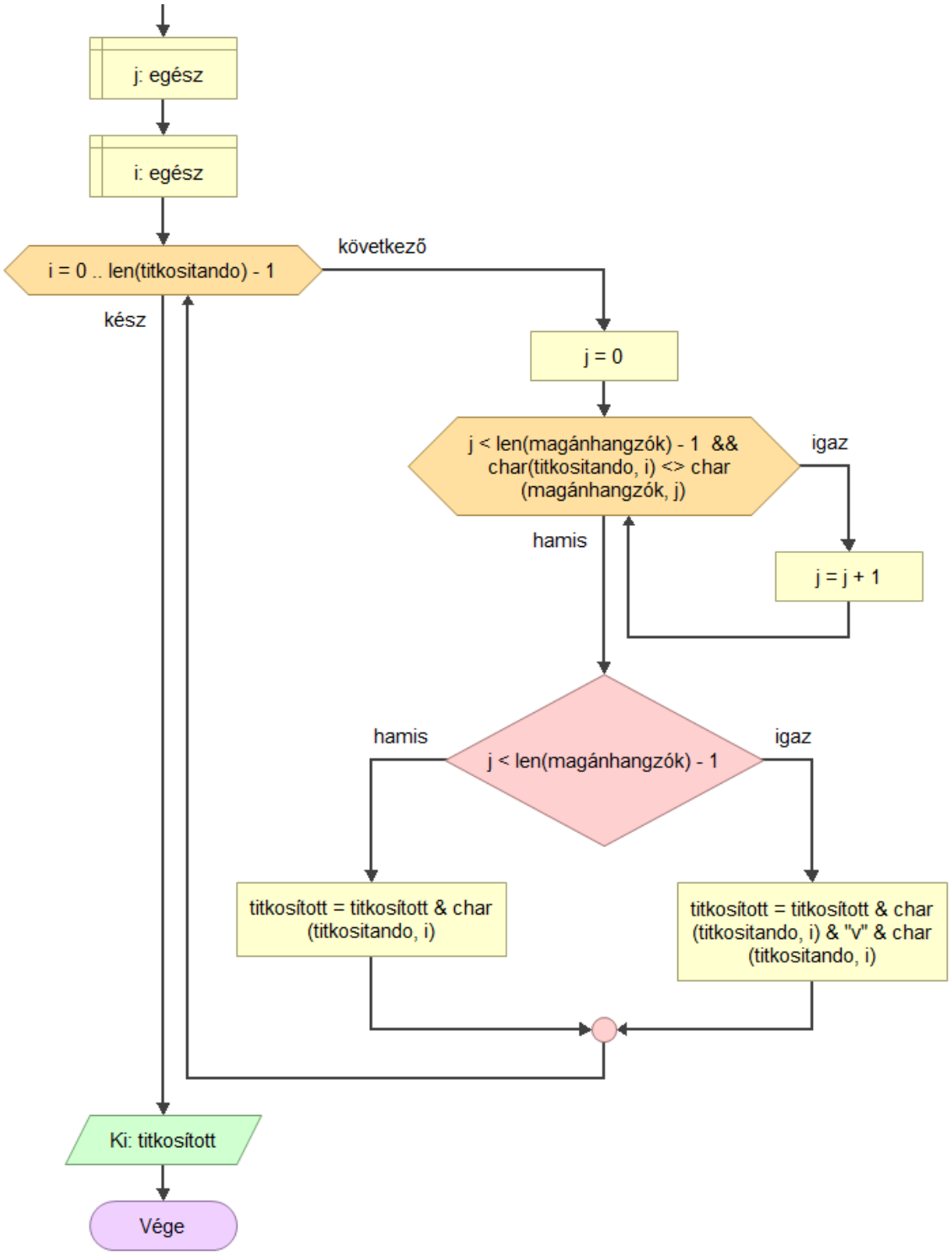
- A feladat nehezíthető úgy, hogy a bemenetet egyetlen szóra korlátozzuk, vagyis figyelünk arra, hogy az ne tartalmazhasson szóközt.
- Határozzuk meg, hogy a beolvasott szó vagy szöveg hány szótagból áll! (A szövegben lévő magánhangzókat kell megszámolni.)

23. Tuvudsz ívígy beveszévélnivi?

Olvassunk be egy szöveget, majd „fordítsuk le” a jól ismert gyereknyelvre!

Ez a feladat az egyik személyes kedvencem, a diákjaim is szerették ezt a feladatot, bár volt közöttük olyan, aki korábban nem hallott erről a nyelvről. A nyelv szabályai viszont könnyen és gyorsan megérthetőek. A szövegben minden magánhangzó után beszúrunk egy v-t, majd megismételjük a magánhangzót.



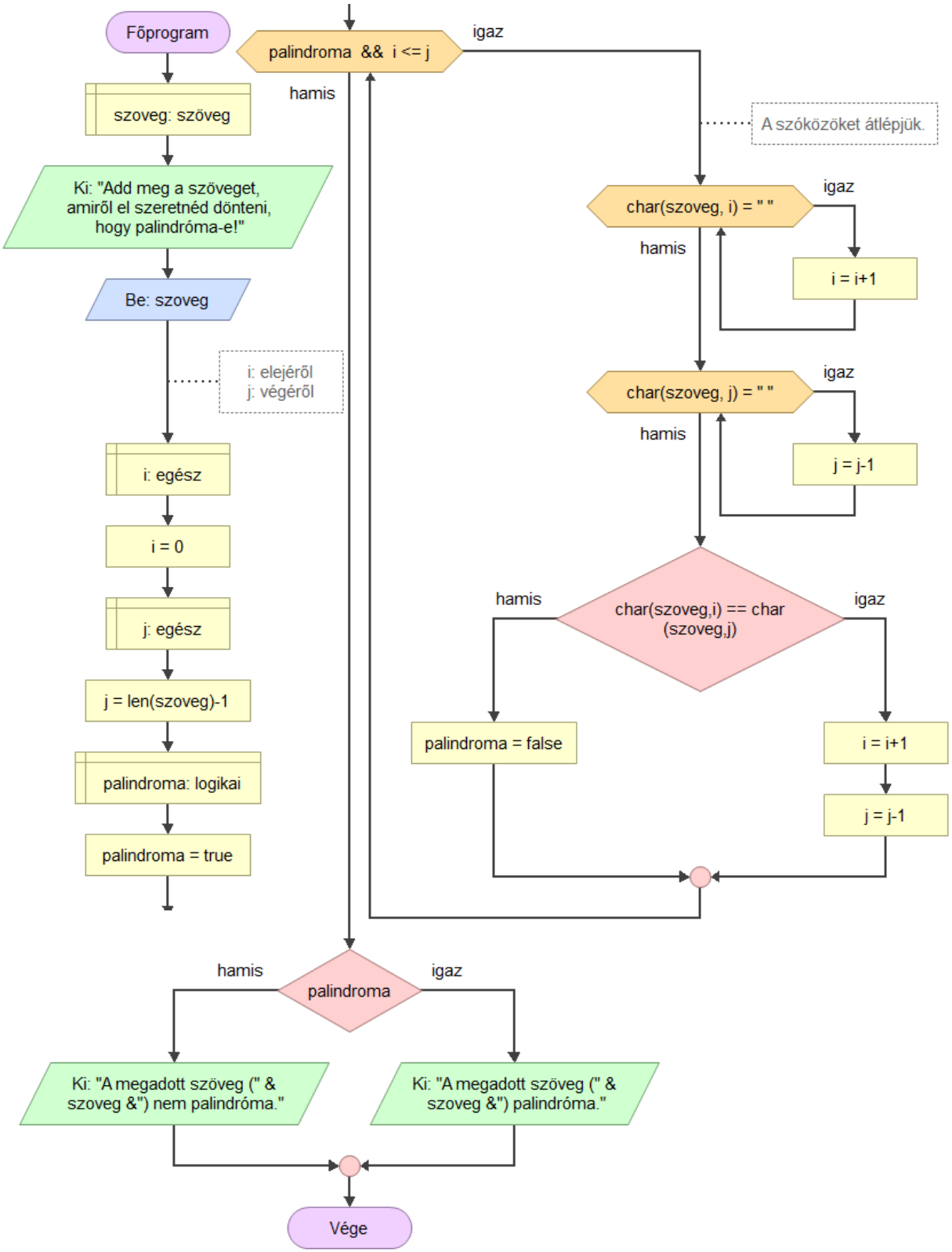


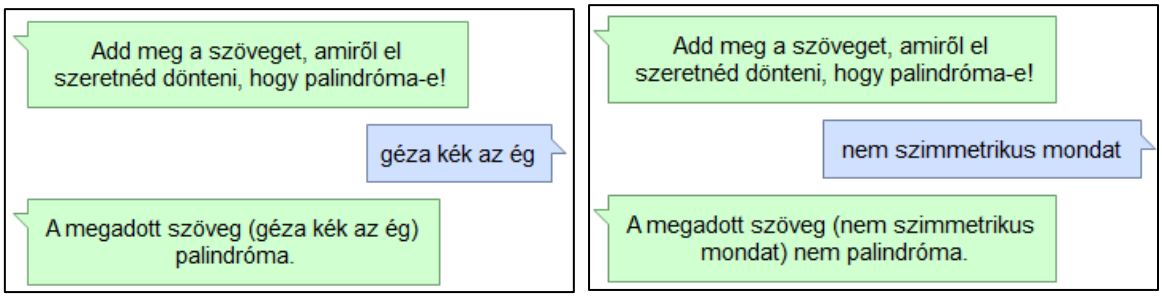
Kiegészítési lehetőség:

A feladat kibővíthető úgy, hogy a fordítóprogramban meg lehessen adni, hogy milyen nyelvjárásra szeretnénk lefordítani a felhasználó által megadott szöveget. Ha a fenti példát tekintjük v-s nyelvjárásnak, akkor például a g-s nyelvjárás esetén a szöveg minden magánhangzója után egy g betűt szúrunk be, majd megismételjük a magánhangzót. Ekkor tehát a magánhangzók után beszúrásra kerülő karaktert is a felhasználó adhatja meg. Hátultesztelő ciklussal a korábbi példához hasonlóan tudjuk megakadályozni, hogy több karaktert adjon meg a felhasználó nyelvjárásként, de dönthetünk úgy, hogy a több karakterből álló nyelvjárásokat is elfogadjuk bemenetként. (Ez a megoldást igazából nem befolyásolja.)

24. Palindróma?

Olvassunk be egy szöveget, majd állapítsuk meg, hogy palinróma-e! A szóközöket hagyjuk figyelmen kívül!





Nem okoz gondot az, ha az `i` (a szöveg elejéről indulva tartja számon, hogy hányadik karakternél tartunk) és a `j` változók (a szöveg végétől indulva tartja számon, hogy hányadik karakternél tartunk) a szóközök figyelmen kívül hagyásánál, átlépésénél felcserélődnek.

A kis- és nagybetűk különbözőnek számítanak, ha összehasonlítjuk őket, ezért a helyes eredmény érdekében érdemes kisbetűkkel begépelni a szöveget.

A szóközök figyelmen kívül hagyása nem nehéz, az összes központozáshoz használt egyéb karakterek (pl. vesszők, pontok) kihagyása viszont már nehezebb feladat lenne. Ennek megoldásához szükség lenne egy olyan változóra, amiben minden olyan karaktert felsorolunk, amit figyelmen kívül szeretnénk hagyni. A karakterek hasonlításakor pedig mindig ellenőrizni kellene azt, hogy az éppen vizsgált karakter szerepel-e azok között, amiket ki kell hagyni.

Hasonló feladatok:

- Készítsünk olyan programot, ami a beolvasott szöveg minden szavát visszafelé, tükrözve írja ki! (pl. *Ez a példa szövege.* → *zE a adlép .egevözs*)
- Olvassunk be egy szóláncot, majd állapítsuk meg, hogy helyes-e! (Minden szó kezdőbetűjének meg kell egyeznie a megelőző szó utolsó betűjével.)

VIII. Függvény használata

Az algoritmusok átláthatósága és a kevesebb ismétlődő rész érdekében gyakran érdemes saját függvényeket bevezetni, melyek alkalmasak bizonyos részfeladatok elvégzésére. Természetesen igaz az is, hogy a függvények túlzott használata is nehezítheti az algoritmus megértését, de van középút.

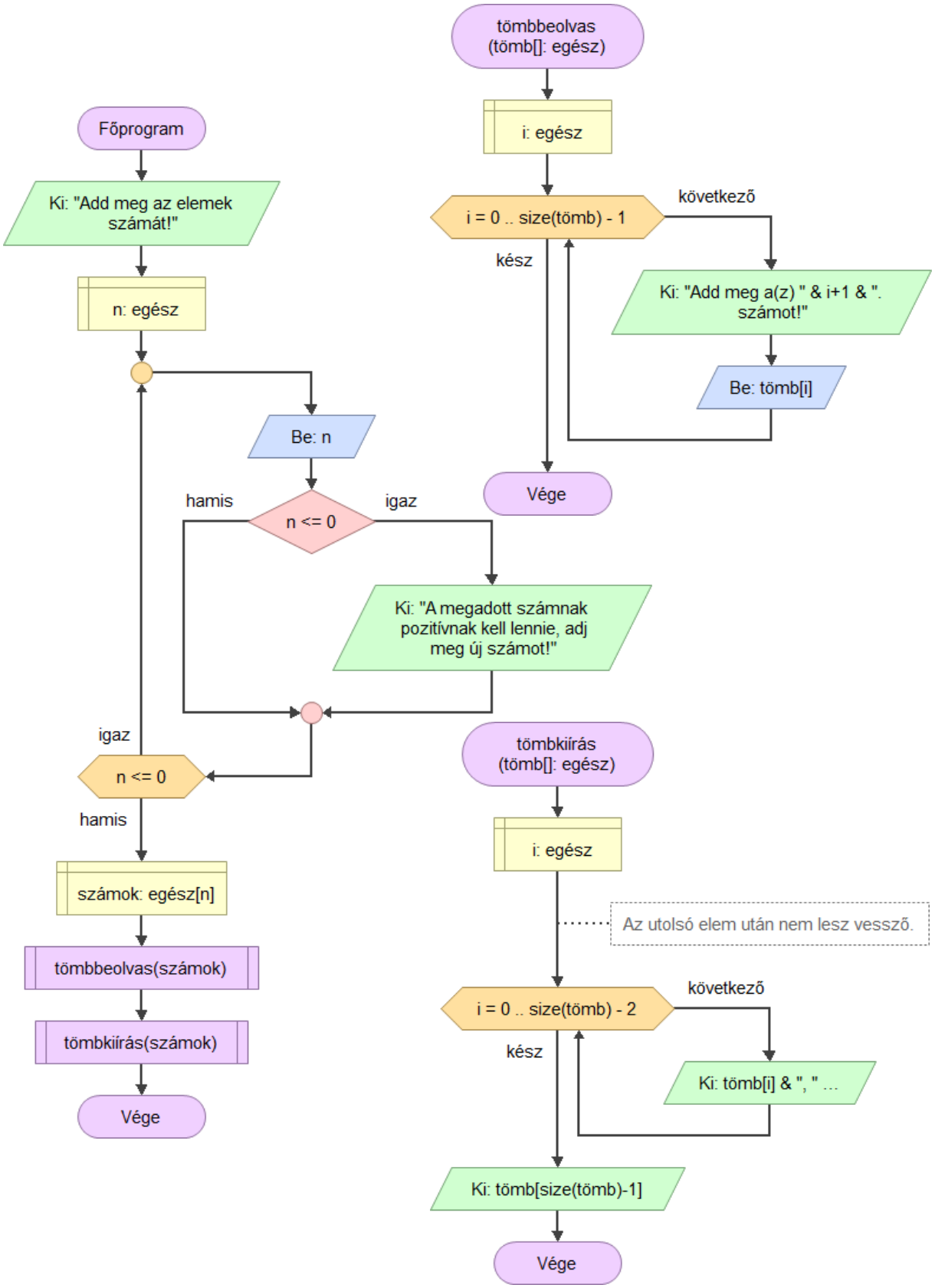
A függvények használatának megismerésére érdemes rászánni egy kis időt, és kitapasztalni azt, hogy mi az, ami megoldható a Flowgorithm-ben, és mi az, ami nem. A következő feladatokban ehhez nyújtunk segítséget, bemutatunk néhány alkalmazási lehetőséget.

25. Tömb elemeinek beolvasása és kiírása függvénnyel

Készítsünk egy-egy függvényt egy tömb elemeinek beolvasására, illetve kiírására!

A feladathoz tartozó algoritmus három részből áll össze. Az egyik a főprogram, amelybe a tömb méretének beolvasása és a tömb deklarálása, valamint a függvény beolvasásához, illetve kiírásához tartozó függvényhívás (Hívás utasítás) kerül. A másik kettő pedig a `tömbbeolvas` és a `tömbkiír` függvény, melyek a tömb elemeinek beolvasását és kiírását végzik. A függvények egyetlen paramétere minkét esetben egy egész típusú tömb, visszatérési értékük pedig nincs.

A tömb méretét tartalmazó változót nem kell paraméterként átadnunk a függvénynek, mert a Flowgorithm beépített `size` függvénye segítségével meg tudjuk állapítani, hogy hány elemű a függvény számára paraméterként átadott tömb.



A paraméterátadás és a különböző helyen (főprogramban vagy függvényben) deklarált változók láthatóságának megértéséhez nagy segítséget nyújt a *Változók figyelése* ablak, amelyben a program lépésenkénti futtatásakor minden lépésnél csak azok a változók jelennek meg, amelyek az adott részen láthatók, használhatók.

Új ismeretek:

- függvények létrehozása
- Hívás utasítás
- paraméterek átadása
- a főprogramban és a függvényekben deklarált változók láthatósága
- beépített `size` függvény használata
- értékadás függvény segítségével tömbök esetén

Érdemes megmutatni a diákoknak, hogy amellett, hogy a függvényekkel átláthatóbbá, rövidebbé tettük a főprogramban szereplő algoritmust, ezek a függvények egy programon belül többször, akár különböző méretű tömbökkel is meghívhatók. Így például egy tömb elemeink többszöri (pl. rendezés előtti és utáni) kiírása vagy több tömb elemeinek ugyanolyan szerkezetű kiírása esetén nem kell újra végiggondolnunk vagy lemásolnunk az algoritmus egy részletét, hanem elegendő a Hívás utasítást megismételnünk.

26. Egy változó beolvasása függvénnyel

Készítsünk egy függvényt, amellyel egy pozitív egész számot tudunk ellenőrzötten beolvasni!

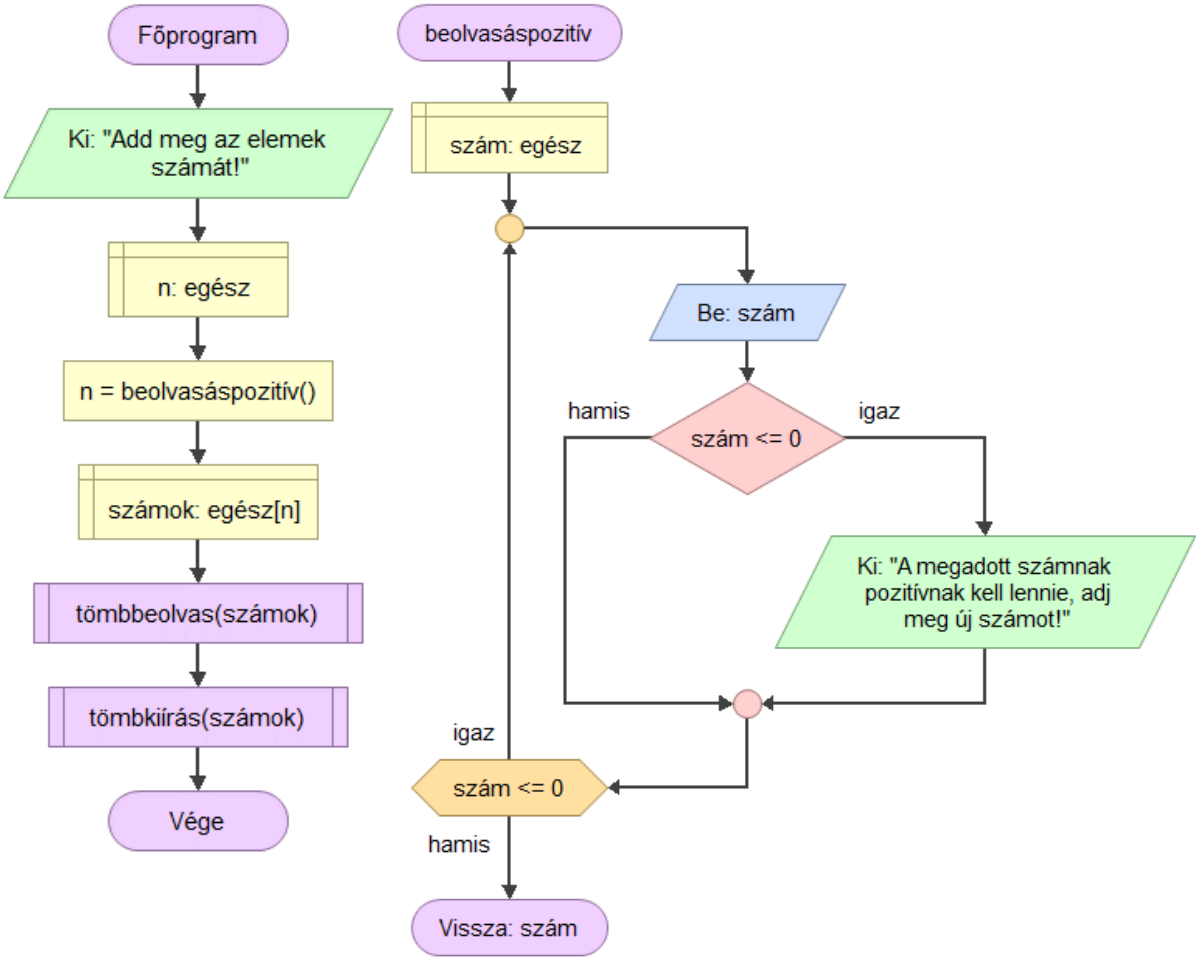
A tömb elemeinek beolvasása mellett külön említést kell tennünk arról, hogy hogyan tudjuk a Flowgorithm-ben egyetlen változó értékét függvény segítségével beolvasni. **A tömböknél alkalmazott módszer csak tömböknél működik**. Ha egy nem tömb típusú változó (pl. egész) beolvasását próbáljuk megvalósítani a tömböknél látott módon, akkor futtatáskor hibaüzenetet kapunk. A hibaüzenet oka az, hogy **nem tömb típusú változót csak akkor adhatunk át paraméterként egy függvénynek, ha az korábban már értéket kapott.** Viszont önmagában az sem jelent megoldást, ha a változó beolvasását végző függvény hívása előtt a változónak adunk valamilyen értéket, amit a beolvasás során megváltoztatunk. Ilyen módon a függvényen belül tudunk az új változóértékkel dolgozni, a főprogramba visszatérve viszont a változó értéke ismét az lesz, ami a függvény hívása előtt volt. Ezt úgy tudjuk kiküszöbölni, ha a függvény visszatérési értékeként a beolvasott változót állítjuk be. Ekkor viszont a főprogramban Hívás utasítás helyett értékadást kell használnunk.

Egy nem tömb típusú változó értékét tehát úgy tudjuk beolvasni függvény segítségével, hogy létrehozunk egy – akár paraméter nélküli – függvényt, ahol a visszatérési típus a beolvasni kívánt változó típusának megfelelő. (Ez a típus a mi esetünkben most *egész*.) A függvényben elvégezzük a visszatérő változó beolvasását, a főprogramban pedig a beolvasni kívánt változó értékeként a függvényt adjuk meg, így a változó azt az értéket kapja meg, amivel a függvény visszatér a főprogram futtatásához.

A pozitív egész szám ellenőrzött beolvasását az előző feladatban létrehozott algoritmusban használtuk fel. Az ottani `n` változó beolvasását helyeztük át a `beolvaspozitív` függvénybe.

Új ismeretek:

- a főprogramban és a függvényekben deklarált változók láthatósága
- visszatérési értékkel rendelkező függvény létrehozása
- saját függvény használata értékadásban
- értékadás függvény segítségével nem tömb típusú változók esetén



A Flowgorithm-ben általánosan is igaz az, hogy a tömbök egyes elemeinek főprogrambeli értékét tudjuk módosítani anélkül, hogy az érték a függvény visszatérési érték lenne, de a nem tömb típusú változók értékeit csak akkor tudjuk függvény segítségével megváltoztatni, ha az érték a függvény visszatérési értéke.
A függvény visszatérési érték nem lehet tömb.

Hasonló feladat:

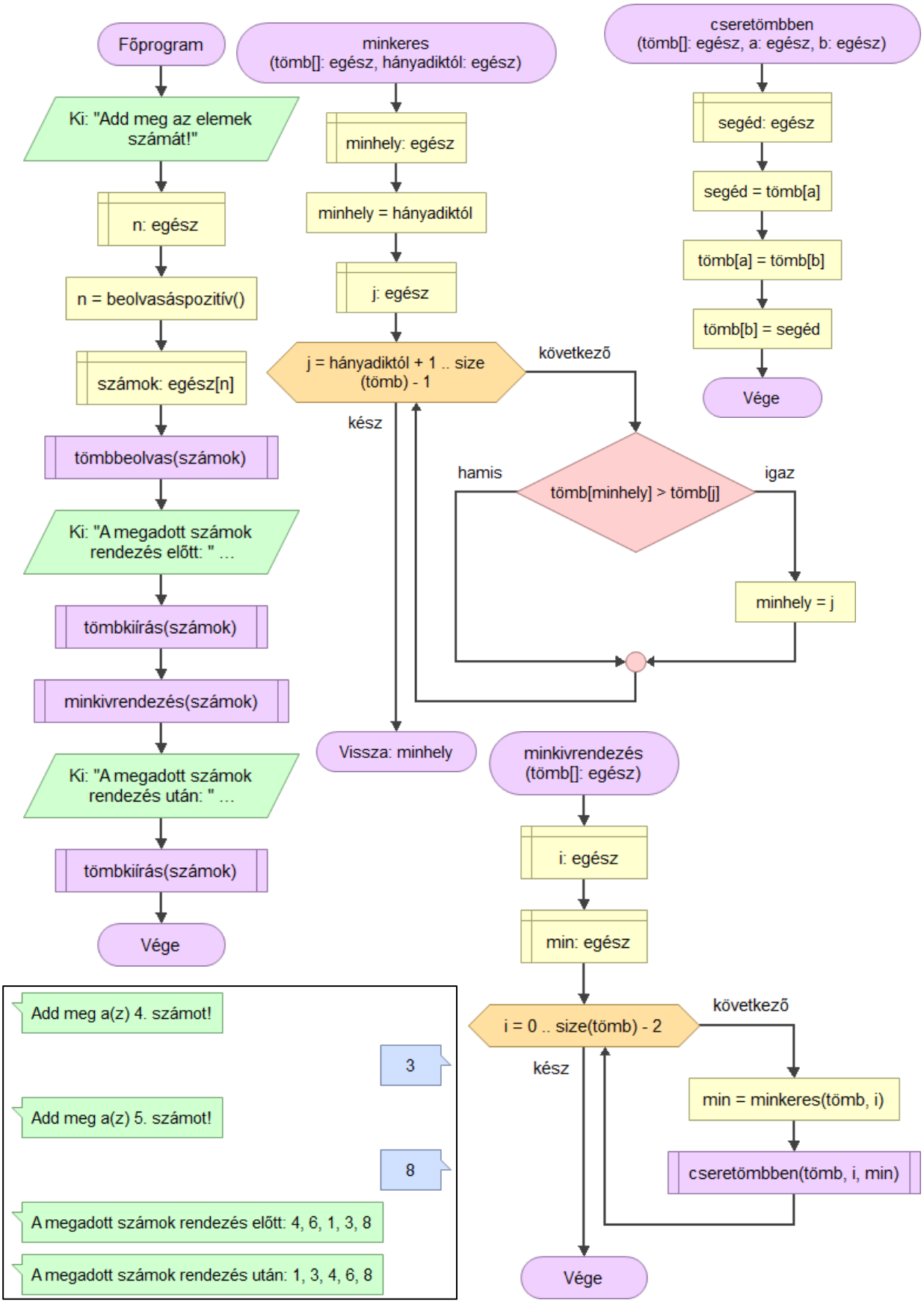
Készíthetünk olyan számot beolvasó függvényt is, ahol a függvény paramétereként megadhatjuk, hogy milyen határok fogadható el a szám.

27. Minimumkiválasztásos rendezés

Olvassunk be egy egész számokból álló tömböt, majd rendezzük a számokat növekvő sorrendbe minimumkiválasztásos rendezés segítségével! A megoldásban alkalmazzunk függvényeket!

A minimumkiválasztásos rendezés során megkeressük a tömb legkisebb elemét, majd a legkisebb elemet kicseréljük a tömb első elemével. Ezután ugyanezt megtesszük a második elemtől kezdődő résztömbben is. Az ottani legkisebb elemet is a résztömb első elemével cseréljük ki. Ezt ismételjük a tömb vége felé haladva az egyre kisebb résztömbökkel. Az utolsó résztömb, amit vizsgálnunk kell, az az eredeti tömb utolsó két eleméből áll. Így a folyamat végére a tömb elemei növekvő sorrendbe kerülnek.

A feladat megoldásához felhasználjuk az előző két feladatban létrehozott függvényeket (`beolvaspozitív`, `tömbbeolvas`, `tömbkiírás`). Ezeknek az algoritmusait nem másoljuk be újra.


Főprogram
Ki: "Add meg az elemek számát!"
n: egész
n = beolvasáspozitív()
számok: egész[n]
tömbbeolvas(számok)
Ki: "A megadott számok rendezés előtt: " …
tömbkiírás(számok)
minkivrendezés(számok)
Ki: "A megadott számok rendezés után: " …
tömbkiírás(számok)
Vége
minkeres (tömb[]: egész, hányadiktól: egész)
minhely: egész
minhely = hányadiktól
j: egész
j = hányadiktól + 1 .. size (tömb) - 1
következő
kész
hamis
tömb[minhely] > tömb[j]
igaz
minhely = j
Vissza: minhely
cseretömbben (tömb[]: egész, a: egész, b: egész)
segéd: egész
segéd = tömb[a]
tömb[a] = tömb[b]
tömb[b] = segéd
Vége
minkivrendezés (tömb[]: egész)
i: egész
min: egész
i = 0 .. size(tömb) - 2
következő
kész
min = minkeres(tömb, i)
cseretömbben(tömb, i, min)
Vége
Add meg a(z) 4. számot!
3
Add meg a(z) 5. számot!
8
A megadott számok rendezés előtt: 4, 6, 1, 3, 8
A megadott számok rendezés után: 1, 3, 4, 6, 8

A korábbi függvények mellé készítünk egy új, `minkivrendzés` függvényt. Ebben a függvényben valósítjuk meg a minimumkiválasztásos rendezést. Ennek egyetlen bemeneti paramétere egy egész típusú tömb, visszatérési érték pedig nincs.

A `minkivrendezés` függvényen belül további két függvényt használunk. Az egyik a `minkeres`, ami egy tömb adott elemétől kezdve megkeresi a legkisebb elemet, és a legkisebb elem indexét adja vissza eredményként. Két paramétere van: az első egy egész típusokból álló tömb, a második pedig egy egész szám, ami azt jelöli, hogy hányadik elemnél kezdődik az a résztömb, amiben a legkisebb elemet keresni kell. A másik függvény a `cseretömbben`, amelynek három paramétere van, egy tömb és két egész szám, amik a tömb egy-egy indexét jelölik. Ez a függvény kicseréli a paraméterül kapott tömbben a két indexhez tartozó tömbelemet.

Új ismeretek:

- minimumkiválasztásos rendezés
- saját függvény több paraméterrel visszatérési értékkel és anélkül
- függvény hívása függvényben

Hasonló feladatok:

A minimumkiválasztásos rendezéshez hasonló módon más rendezési algoritmusok is megvalósíthatók.

A különböző rendezések hatékonyságának összehasonlításához érdemes a rendezések során az algoritmus lépésszámát és az összehasonlítások számát nyomon követni.

IX. Összetettebb, vegyes feladatok

A következő feladatok nem illettek bele egyik korábbi kategóriába sem, vagy épp több kategóriába is beleillettek volna, ezért inkább itt kaptak helyet. Az itteni feladatok valamivel összetettebbek az egy-egy utasítást bevezető egyszerű feladatoknál, de a korábbiakhoz hasonlóan ezekre is igaz, hogy a legtöbbnél van egyszerűsítési vagy épp nehezítési lehetőség. A feladatok egy részénél függvényeket is alkalmazunk, de a tanulók szintjének megfelelően a függvények használata akár ki is hagyható a megoldásokból.

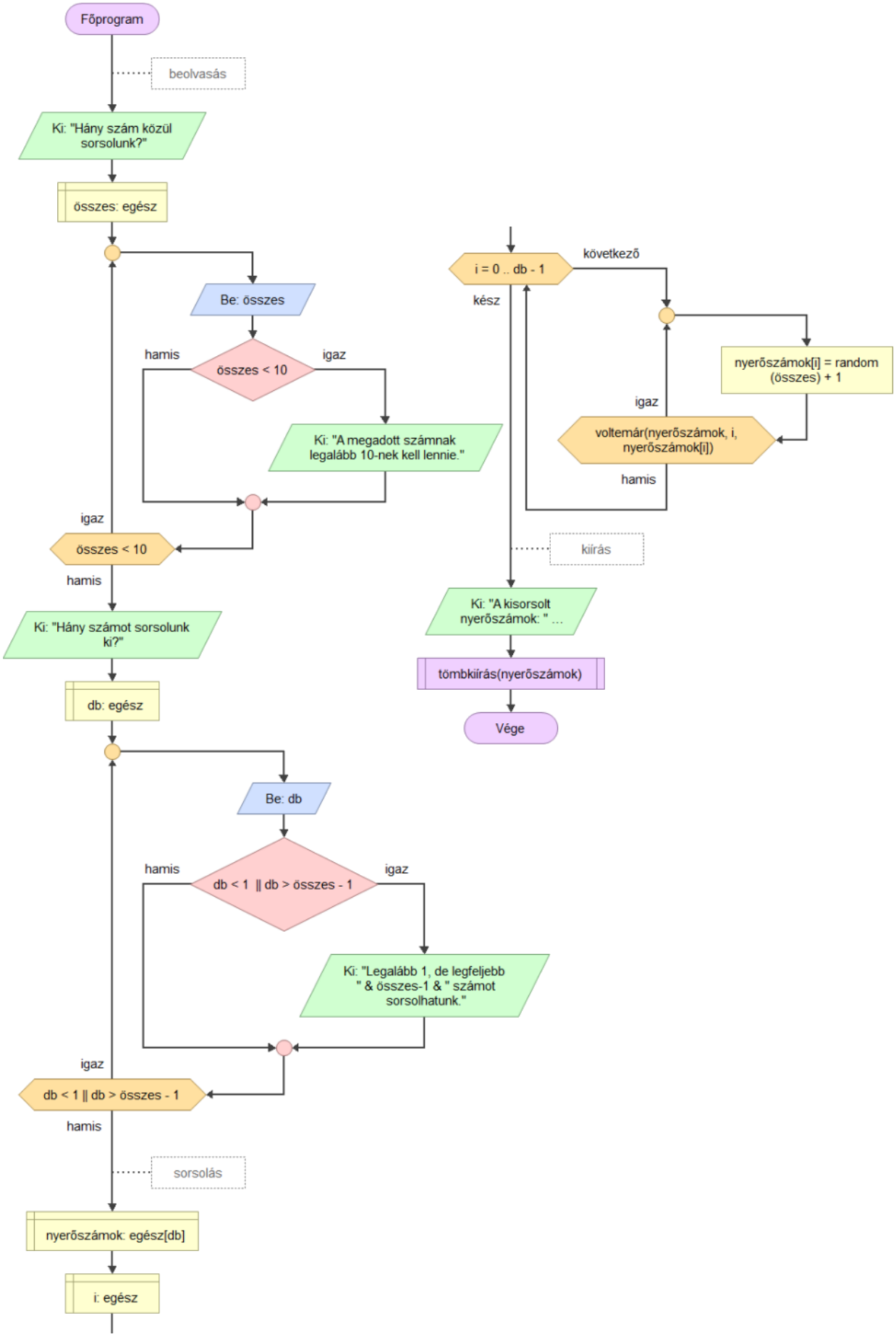
28. Lottósorsoló

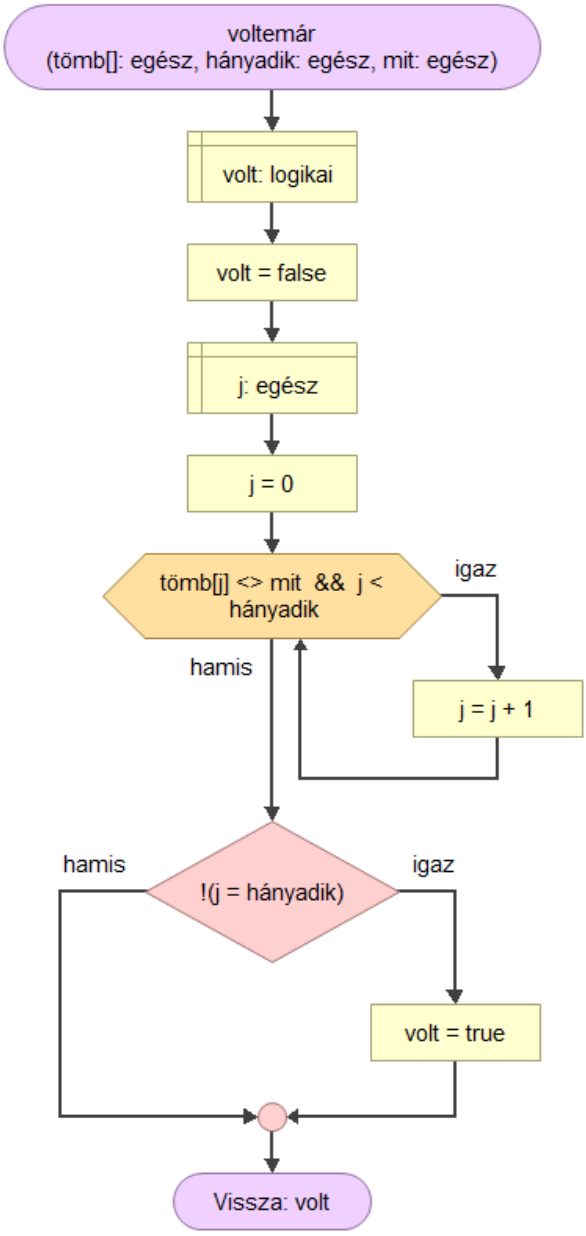
Készítsünk egy programot, amellyel kisorsolhatjuk a lottó nyerőszámait! A programban az összes lehetséges szám és a kisorsolt számok darabszámát a felhasználó adja meg. Figyeljünk arra, hogy ne lehessen ugyanazt a számot többször kisorsolni!

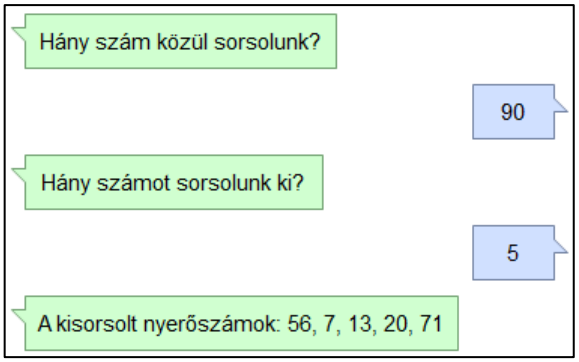
A megoldásban függvényt készítettünk `voltemár` néven annak ellenőrzésére, hogy az új sorsolt szám szerepel-e már a kisorsoltak között. Ez a függvény paraméterként kapja a kisorsolt számok tömbjét, azt, hogy hányadik szám sorsolásánál tartunk (eddig kell megvizsgálni a tömb elemeit) és az új sorsolt számot, amiről tudni szeretnénk, hogy benne van-e már a tömbben. A nyerőszámok kiírásához itt is a 25. feladatban létrehozott `tömbkiírás` függvényt használtuk.

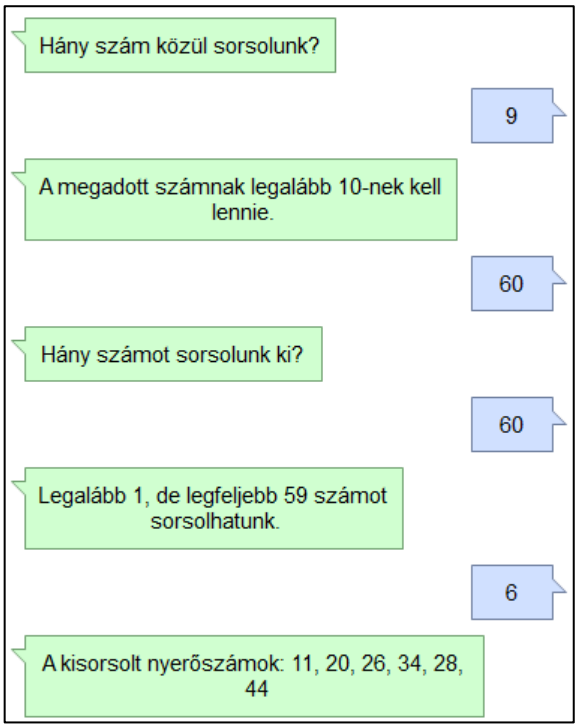
Új ismeretek:

- tömb feltöltése véletlen számokkal
- tömbelemek ismétlődésének kizárása (saját függvénnyel)
- saját függvény logikai típusú visszatérési értékkel
- saját függvény használata hátultesztelő ciklus feltételeként


Főprogram
beolvasás
Ki: "Hány szám közül sorsolunk?"
összes: egész
Be: összes
hamis
összes < 10
igaz
Ki: "A megadott számnak legalább 10-nek kell lennie."
igaz
összes < 10
hamis
Ki: "Hány számot sorsolunk ki?"
db: egész
Be: db
hamis
db < 1 || db > összes - 1
igaz
Ki: "Legalább 1, de legfeljebb " & összes-1 & " számot sorsolhatunk."
igaz
db < 1 || db > összes - 1
hamis
sorsolás
nyerőszámok: egész[db]
i: egész
i = 0 .. db - 1
következő
kész
nyerőszámok[i] = random (összes) + 1
igaz
voltemár(nyerőszámok, i, nyerőszámok[i])
hamis
kiírás
Ki: "A kisorsolt nyerőszámok: " …
tömbkiírás(nyerőszámok)
Vége







Ez a feladat egyike azoknak, amelyek a diákjaim ötletei alapján kerültek bővítésre, továbbfejlesztésre. Egy egyszerű, tömbökkel kapcsolatos feladatként indult, amellyel be lehet vezetni a véletlenszámok generálását. Ekkor még a 0 is a kisorsolható számok kövzött szerepelt, majd jöttek a feltételek, hogy ne lehessen minden számot kisorsolni, ne legyen sorsolható a 0, és ne legyen többször sorsolható ugyanaz a szám. Ezek megvalósítása után a tanulók ötletei nyomán a különböző „szabálytalan" feltételek irányába mentünk tovább, melyekkel befolyásoltuk a sorsolást, de a számok rendezésének ötlete is felmerült.

Módosítási lehetőségek:

- A feladat egyszerűsíthető úgy, hogy megengedjük, hogy egy-egy szám akár többször is kisorsolásra kerüljön.
- A megoldás kiegészíthető a számok növekvő sorrendbe való rendezésével.
- Nehezíthető a feladat úgy, hogy a kisorsolt számokkal kapcsolatosan feltételeket szabunk (pl. csak páros számok sorsolhatók ki; ugyanannyi páros és páratlan számot kell kisorsolni…).

29. Legnagyobb közös osztó (több számra)

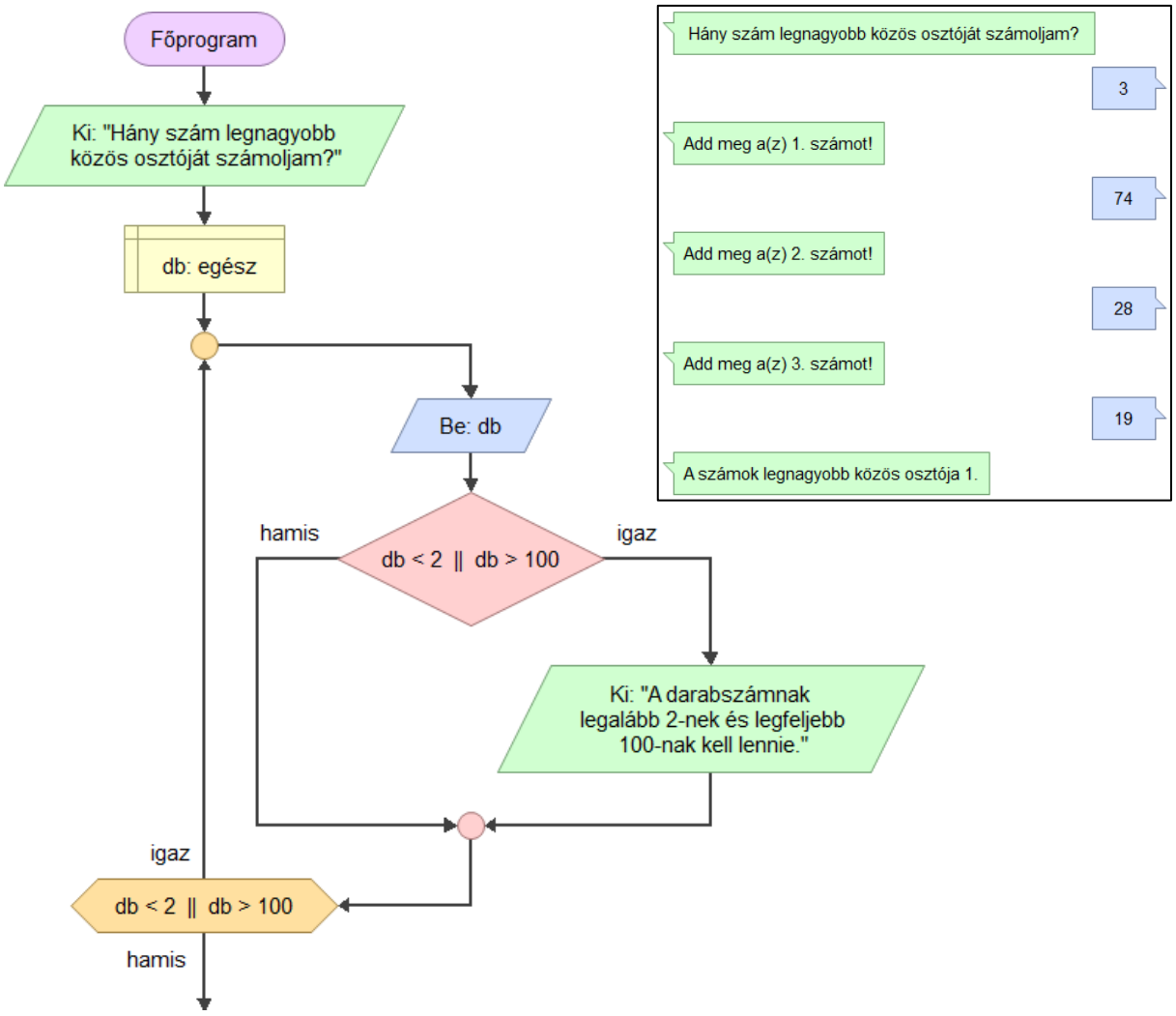
Olvassunk be `n` darab számot, majd határozzuk meg a legnagyobb közös osztójukat!

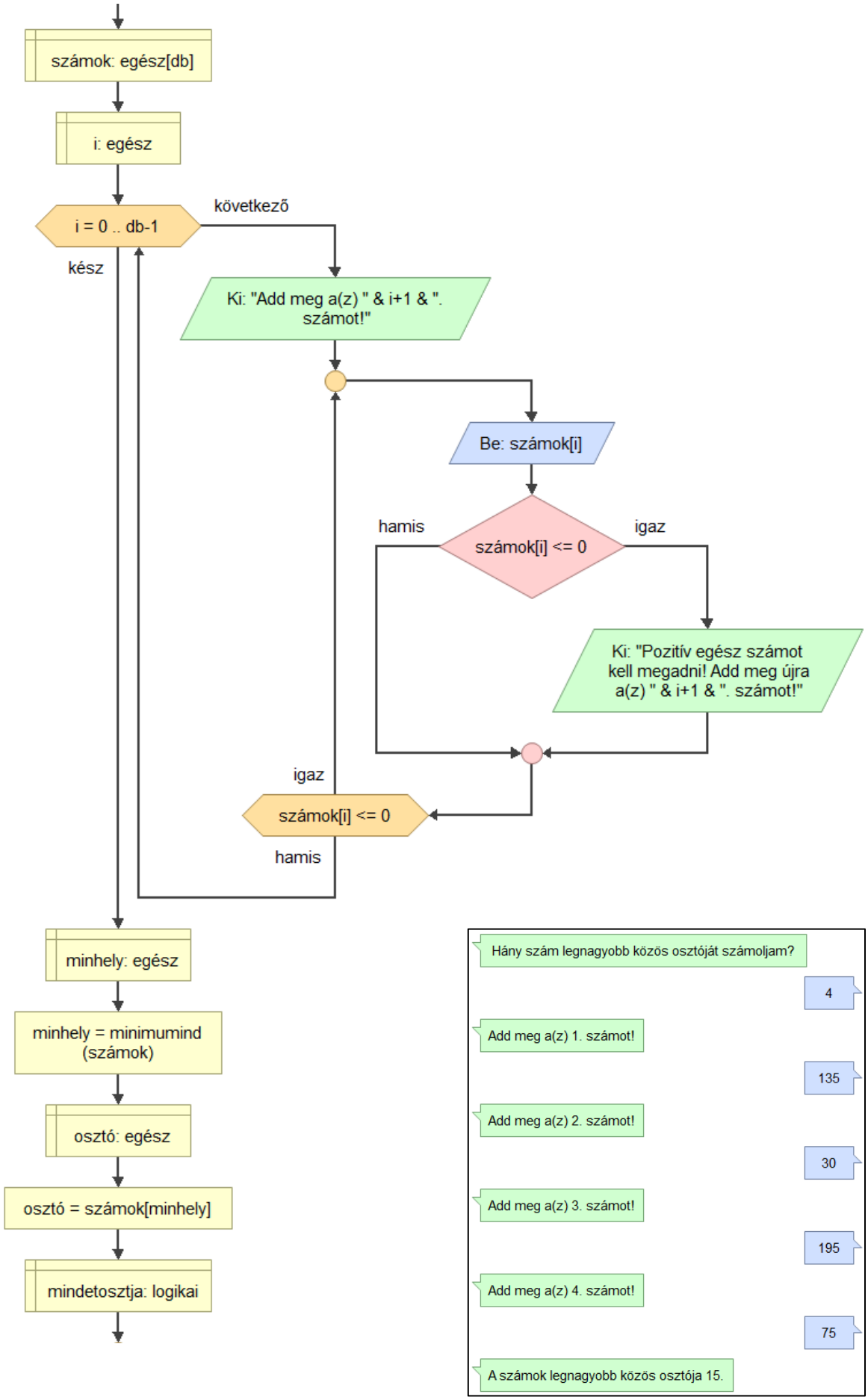
Matematikaórákon a legnagyobb közös osztó kiszámítását úgy szoktuk kezdeni, hogy elkészítjük a számok prímtényezős felbontását. Ez azonban nehezen programozható és rendkívül műveletigényes folyamat. Ebből adódóan a feladat Flowgorithm-mel való megoldásához egy másik módszert választottunk.

A számok legnagyobb közös osztója osztója a számok közül a legkisebbnek is. Így a megoldásban először megkeressük a beolvasott számok közül a legkisebbet, majd azt vizsgáljuk meg, hogy ennek a számnak melyik az az osztója, amelyik mindegyik számot osztja. Mivel a számok közös osztói közül a legnagyobbat keressük, ezért először azt vizsgáljuk, hogy a számok közül a legkisebb osztója-e a többi számnak, majd a legkisebb szám egyre kisebb osztóival próbálkozunk. A legkisebb szám osztói közül legkésőbb az 1 biztosan jó lesz, így a folyamat mindig véget ér, minden esetben találunk megfelelő osztót, ami mindegyik beolvasott számnak osztója.

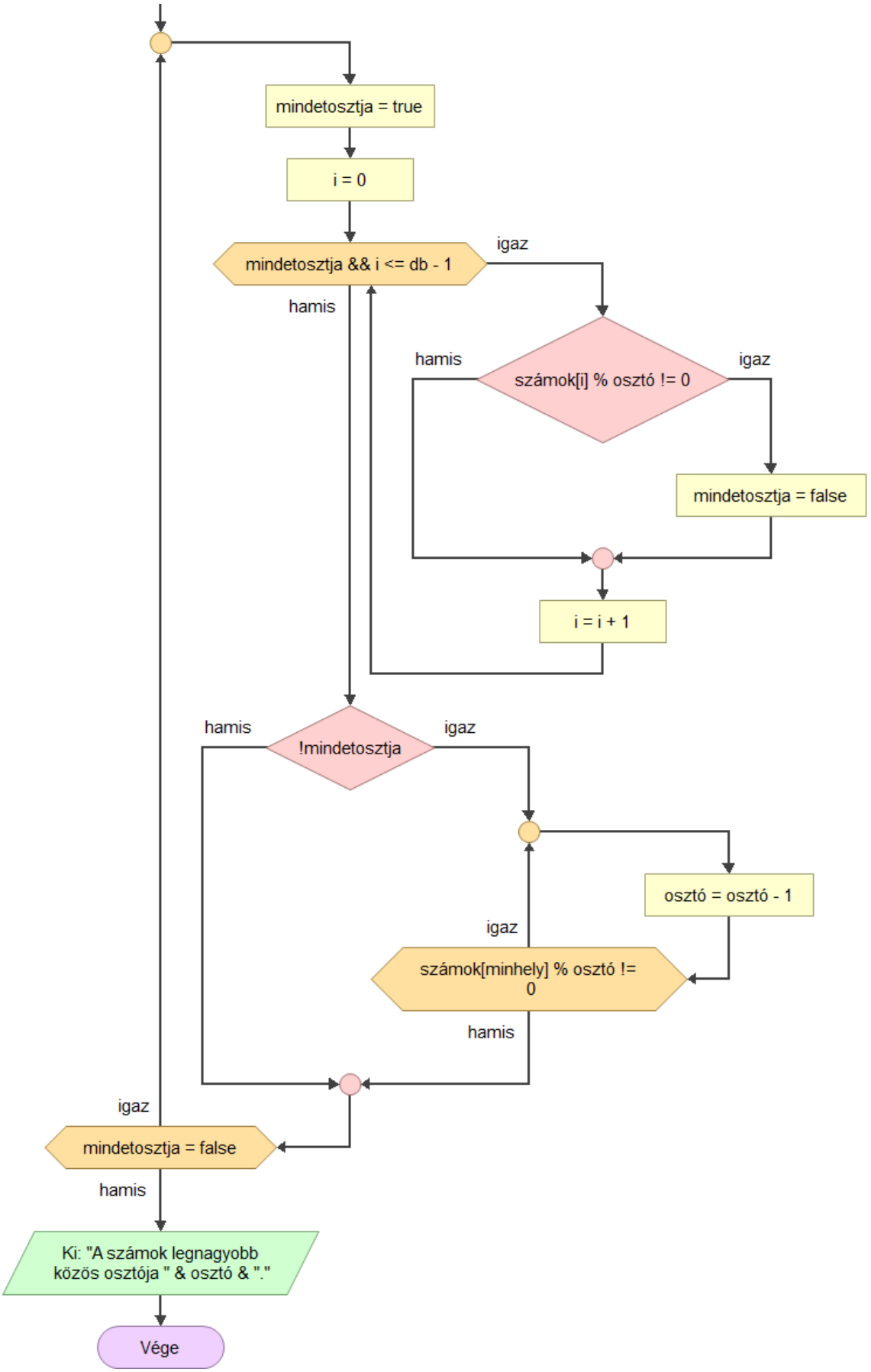
A megoldásban itt egyetlen függvényt használtunk (`minimumind`), ami a tömb legkisebb elemének indexét adja vissza (az első elemé a 0 index).

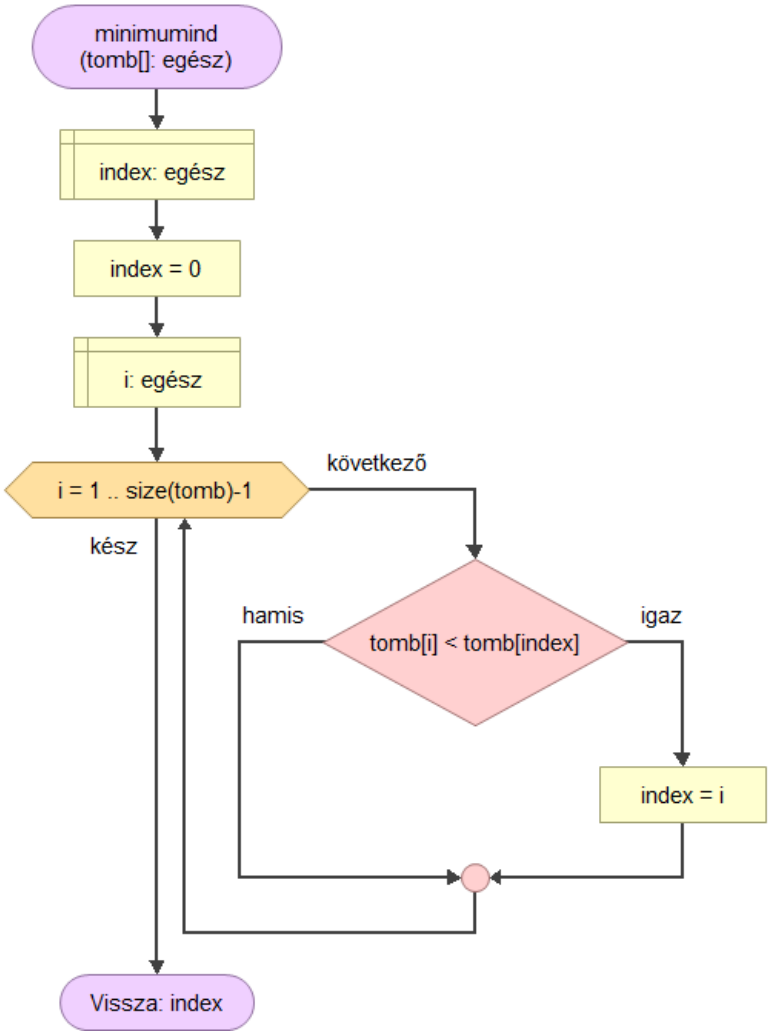
Természetesen itt is használható lenne akár több függvény is a megoldáshoz, illetve függvény használata nélkül is megoldható lenne a feladat.




számok: egész[db]
i: egész
i = 0 .. db-1
következő
kész
Ki: "Add meg a(z) " & i+1 & ". számot!"
Be: számok[i]
hamis
számok[i] <= 0
igaz
Ki: "Pozitív egész számot kell megadni! Add meg újra a(z) " & i+1 & ". számot!"
igaz
számok[i] <= 0
hamis
minhely: egész
minhely = minimumind (számok)
osztó: egész
osztó = számok[minhely]
mindetosztja: logikai

Hány szám legnagyobb közös osztóját számoljam?
4
Add meg a(z) 1. számot!
135
Add meg a(z) 2. számot!
30
Add meg a(z) 3. számot!
195
Add meg a(z) 4. számot!
75
A számok legnagyobb közös osztója 15.


mindetosztja = true
i = 0
mindetosztja && i <= db - 1
igaz
hamis
hamis
számok[i] % osztó != 0
igaz
mindetosztja = false
i = i + 1
hamis
!mindetosztja
igaz
osztó = osztó - 1
igaz
számok[minhely] % osztó != 0
hamis
igaz
mindetosztja = false
hamis
Ki: "A számok legnagyobb közös osztója " & osztó & "."
Vége



Hasonló feladat:

Ehhez hasonló módon számolhatjuk n szám legkisebb közös többszörösét is. Ennél a feladatnál érdemes a beolvasott számok közül a legnagyobbat megkeresni, ezután pedig majd megvizsgálni azt, hogy ezt a szám többszöröse-e a többi számnak. Ha nem, akkor ennek a legnagyobb számnak az egyre nagyobb többszöröseit vizsgáljuk, mert ezek között szerepelni fog a számok legkisebb közös többszöröse is. A folyamat itt is biztosan véget ér, mert legkésőbb a beolvasott számok szorzata közös többszörös lesz, bár az előző feladattal ellentétben itt gondot okozhat az, hogy a számok szorzata már nem biztos, hogy elfér az egész típusú változóban.

30. Átváltás más számrendszerbe

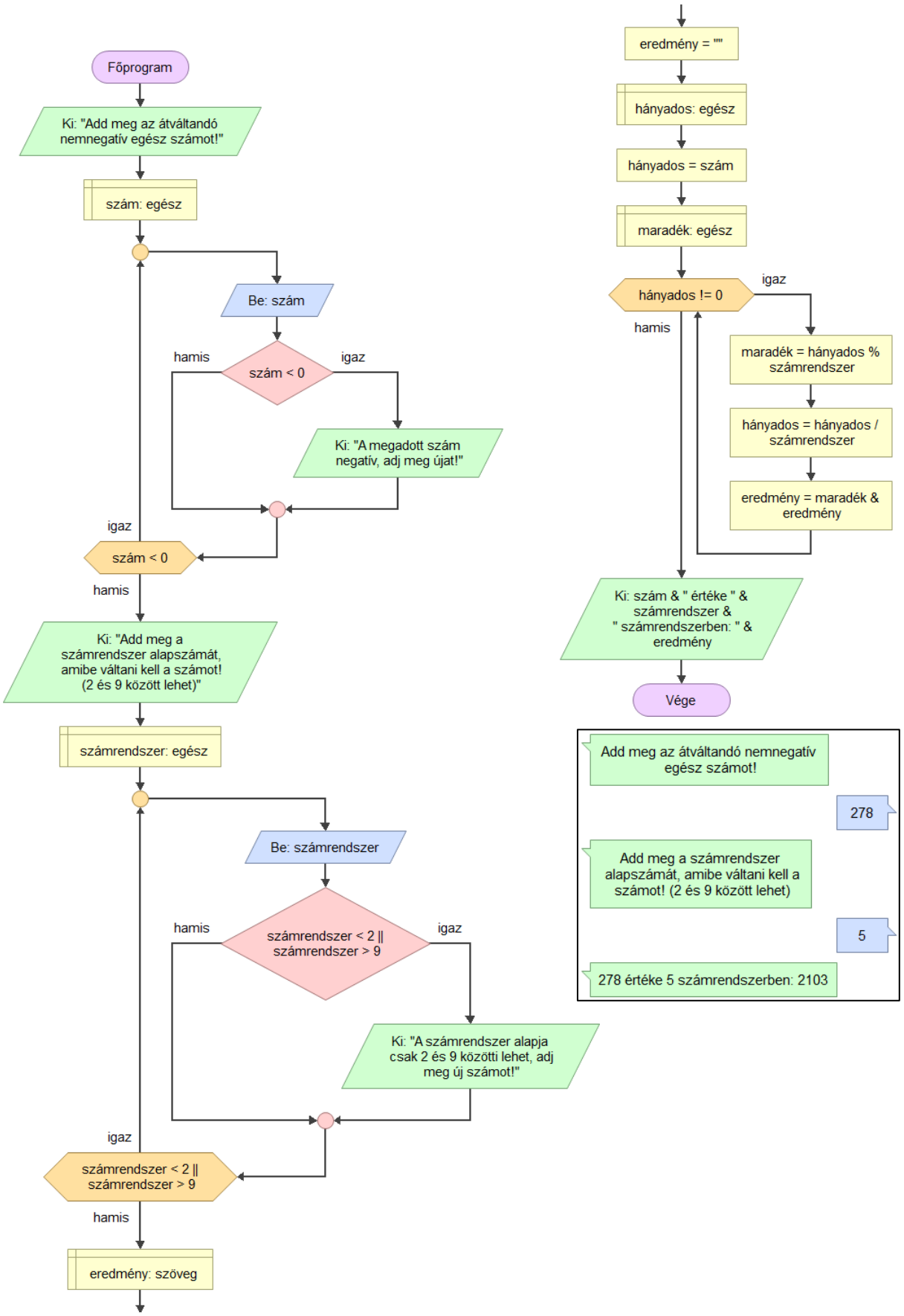
Készíts egy programot, ami átvált egy 10-es számrendszerbeli számot egy másik (2 és 9 közötti) számrendszerbe!

A 10-es számrendszerből más számrendszerbe való átváltás folyamatát az alábbi példán szemléltetjük:

821	: 6 =	136	maradék: 5
136	: 6 =	22	maradék: 4
22	: 6 =	3	maradék: 4
3	: 6 =	0	maradék: 3

$821_{10} = 3445_6$

A módszer előnye, hogy így a 6 megfelelő hatványainak kiszámolása nélkül is meg tudjuk oldani a feladatot.



Mivel a megfelelő 10-hatványokkal való szorzás alkalmazásával bonyolultabb lenne a keletkező számjegyeket az addigiak elé írni, emiatt az eredményt nem *egész*, hanem *szöveg* típusú változóban tároljuk. (A két típus közötti konverzióra szükség esetén használhatjuk a rendelkezésünkre álló beépített `ToInteger` vagy `ToString` függvényeket.)

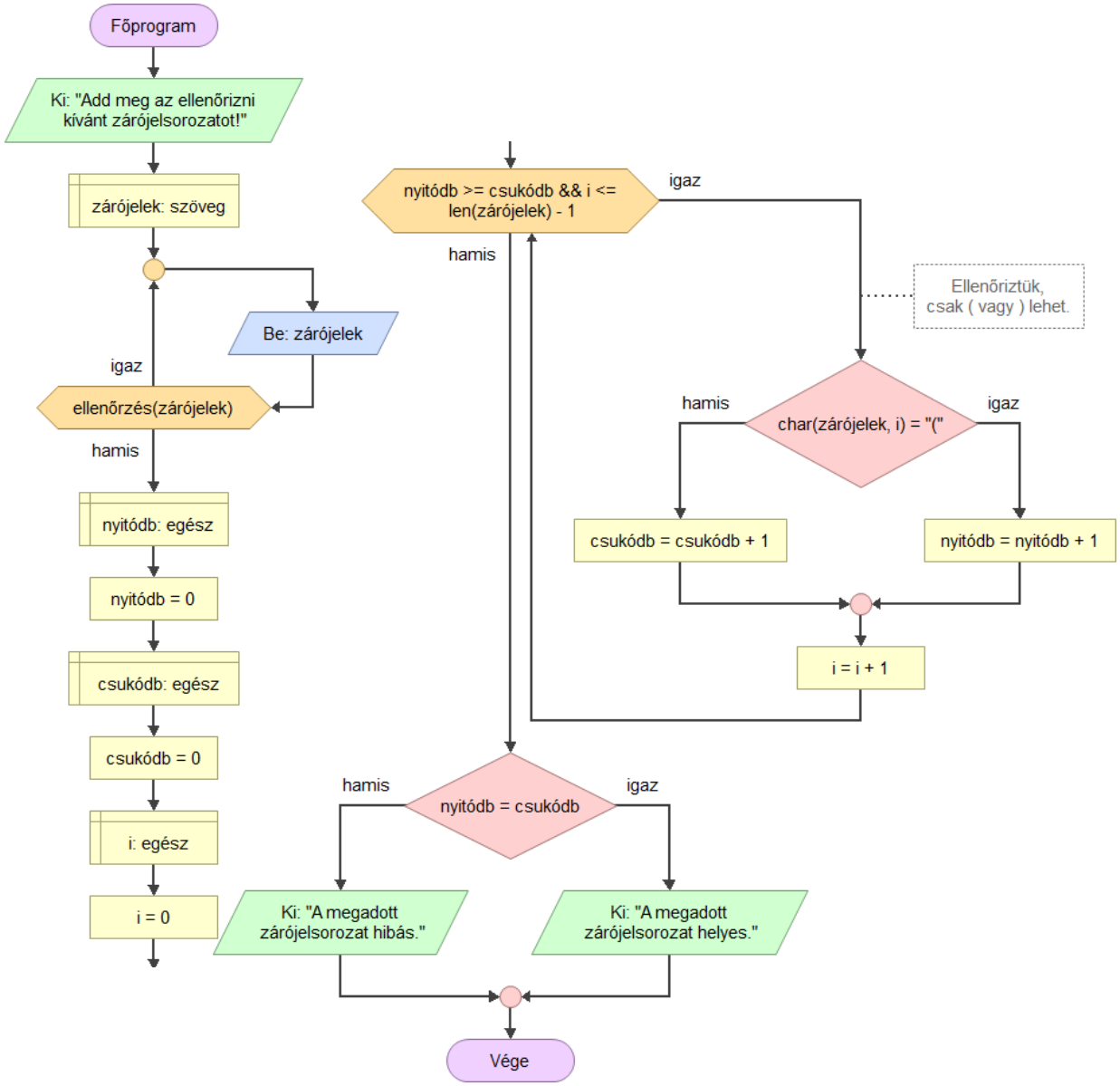
További számokkal, átváltásokkal kapcsolatos feladatok:

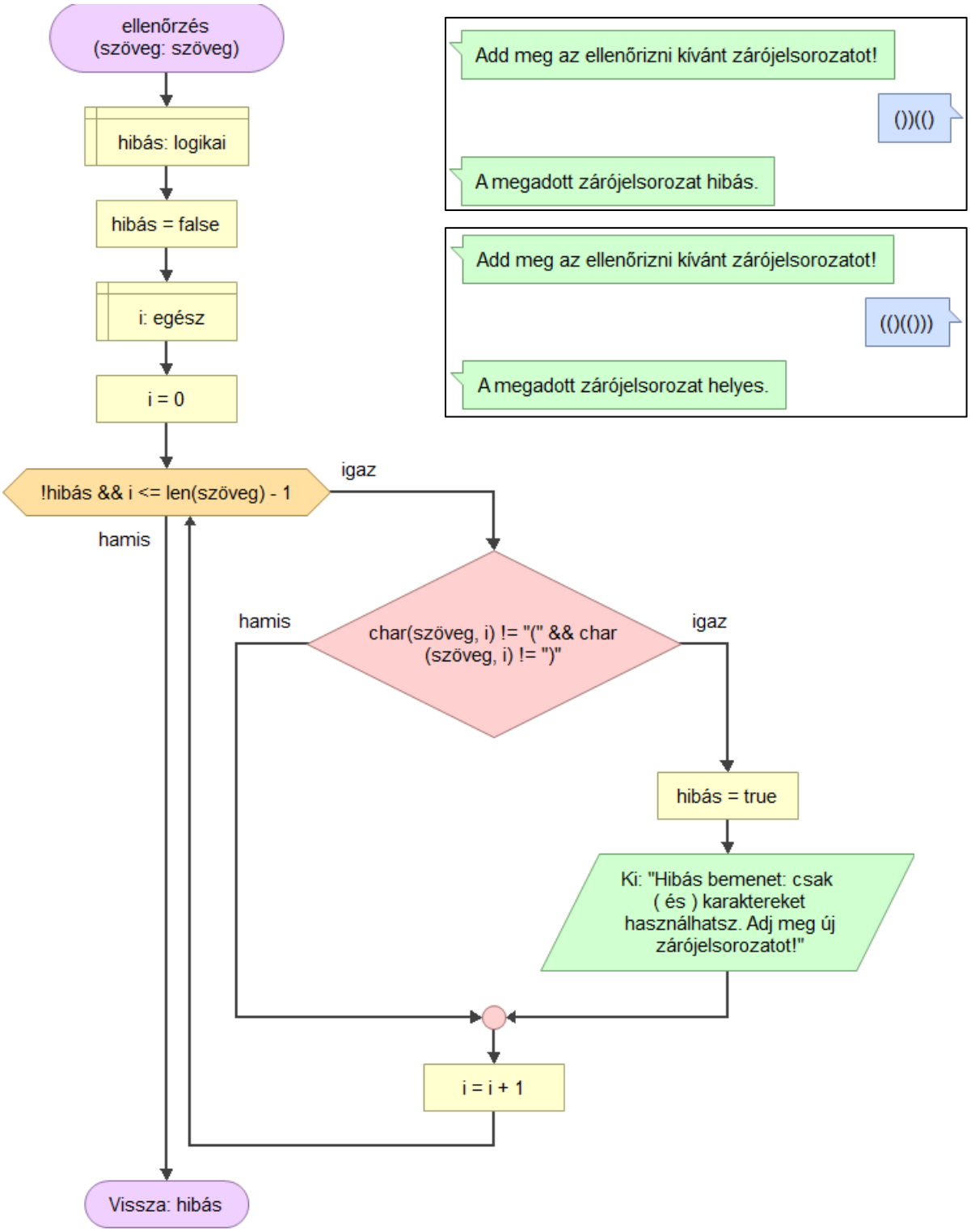
- Váltsunk át egy adott számrendszerbeli számot 10-es számrendszerbe!
- Váltsunk át egy arab számot római számmá!
- Váltsunk át egy római számot arab számmá!
- Határozzuk meg n darab számadat mediánját!
- Kérjünk be egy évszámot a felhasználótól és állapítsuk meg, hogy az adott év szökőév-e!

31. Helyes a zárójelezés?

Olvassunk be egy nyitó és csukó zárójelekből álló jelsorozatot, és ellenőrizzük, hogy helyes-e! (Minden nyitó zárójelnek van-e csukó párja, minden csukó zárójel előtt megfelelő számú nyitó zárójel van-e.)

Pl.: `( ( ( ) ( ) ) )` helyes zárójelezés `( ) ) ( ( ) ( ) )` helytelen zárójelezés





A megoldásban a bemenet helyességének ellenőrzését az `ellenőrzés` függvény segítségével ellenőriztük, ami egy logikai értékkel tér vissza. A visszatérési érték akkor *igaz*, ha a bemenet hibás.

Módosítási lehetőségek:

- A feladat egyszerűsíthető a bemenet ellenőrzésének kihagyásával.
- A feladat nehezíthető többféle zárójel használatával.

32. Anagramma?

Olvassunk be két szöveget, majd állapítsuk meg, hogy azok egymás anagramma változatai-e!

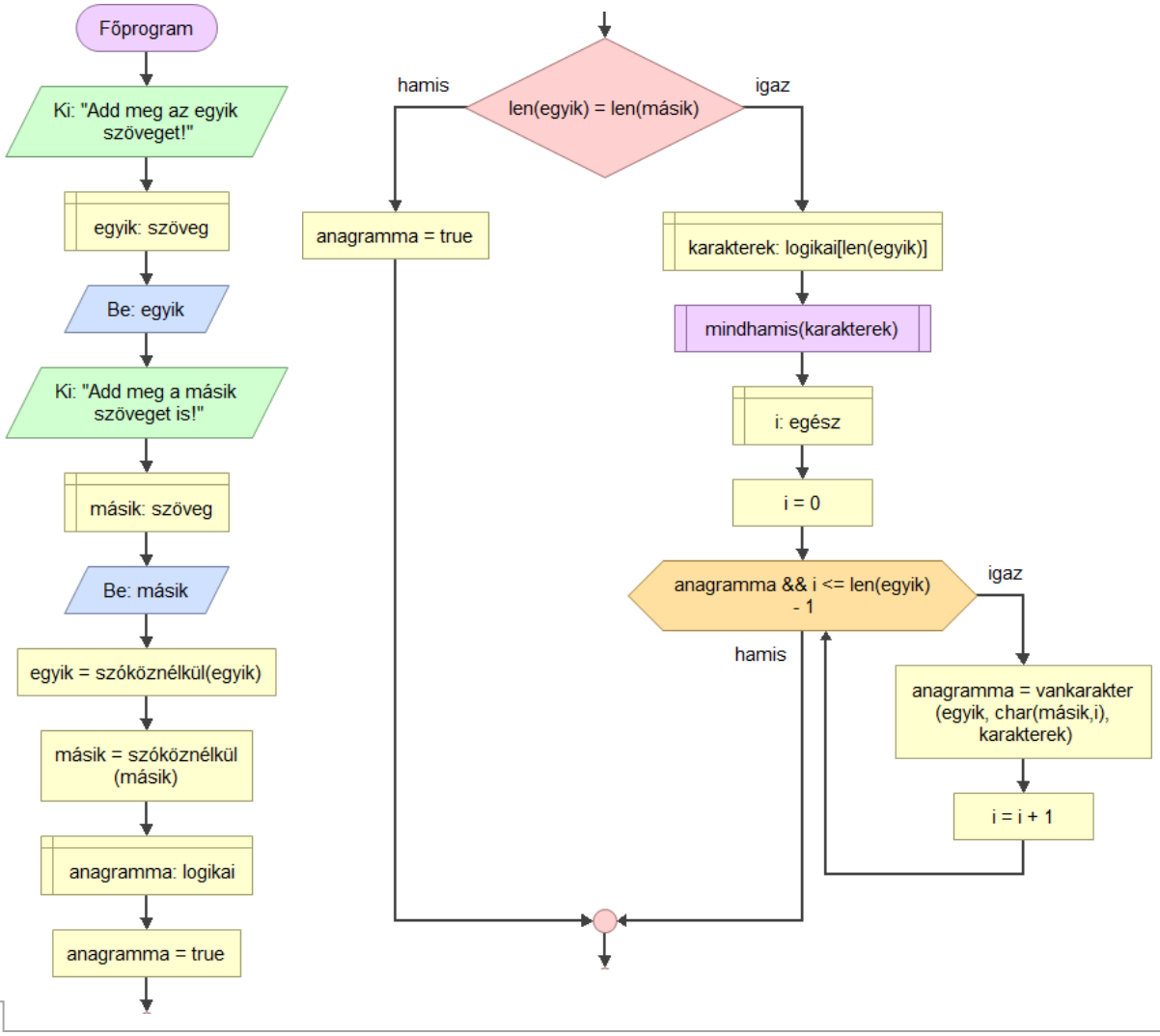
Két szöveg egymás anagramma változata, ha a szövegek csak a betűk sorrendjében különböznek. A szövegek összehasonlításakor a szóközöket figyelmen kívül hagyjuk.

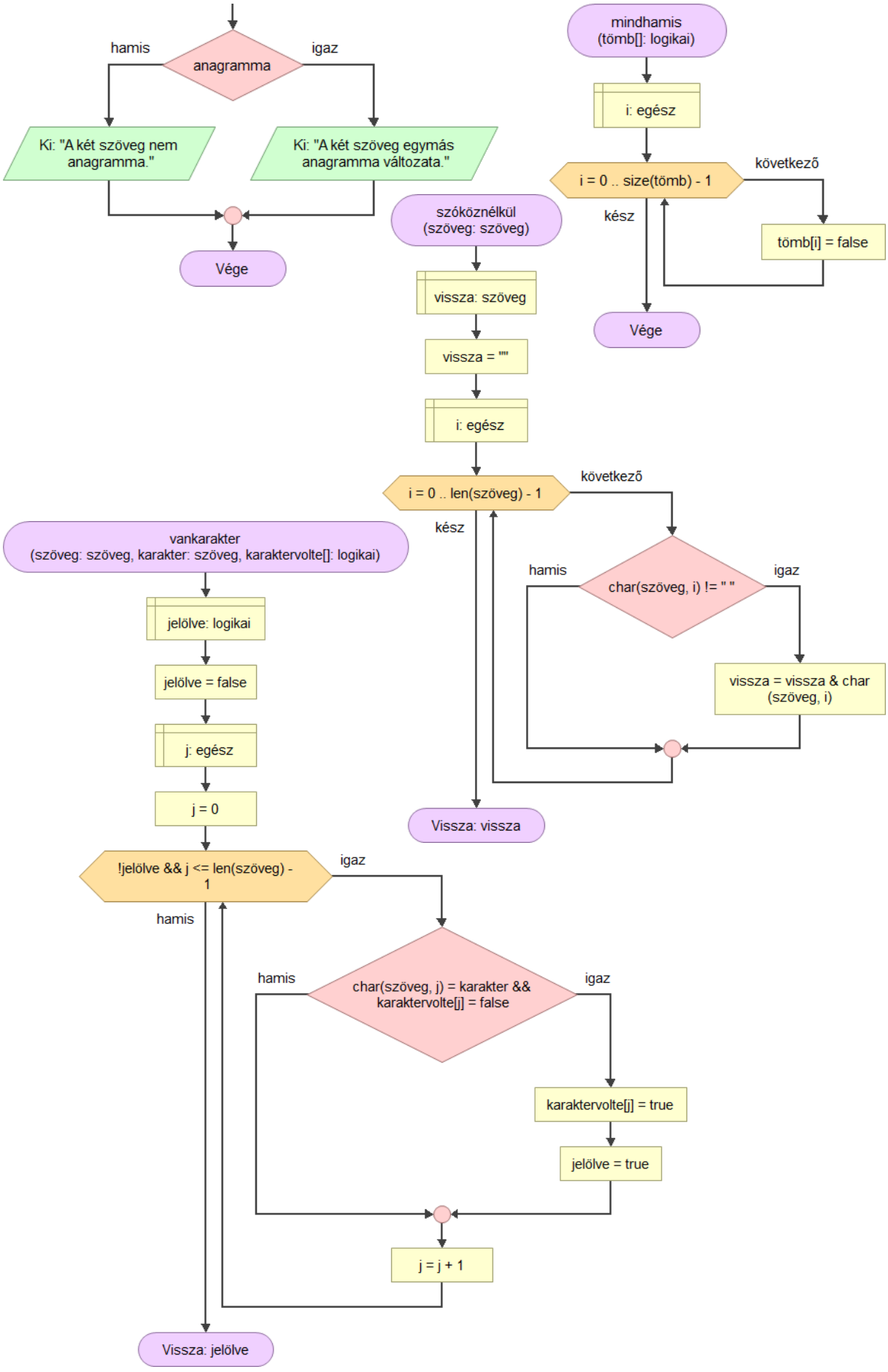
A megoldásban a szövegek szóköz nélküli változatait hasonlítjuk össze. Létrehozunk egy logikai típusú tömböt, amelyben azt tartjuk számon a második szöveg karakterein sorban végighaladva, hogy az első szöveg mely karaktereit használtuk már fel a második szöveghez. Csak addig vizsgáljuk a második szöveg karaktereit, ameddig nem találunk benne olyat, amiből nincs már több az első szövegben.

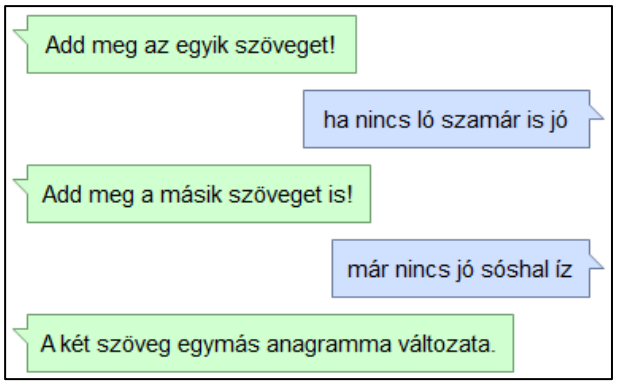
A megoldásban három függvényt használtunk fel. Az egyik, a `szóköznélkül` előállítja a megadott szövegek szóközöktől mentes változatát. Erre azért van szükség, mert ha a szövegek hossza nem egyezik meg, akkor nem lehetnek egymás anagrammái, felesleges további vizsgálatokat végrehajtani.

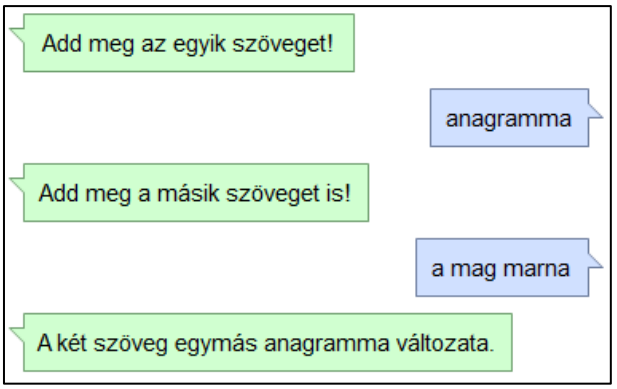
A `mindhamis` függvény a paraméterül kapott logikai típusú tömb minden elemét *hamis*-ra állítja.

A `vankarakter` függvény pedig megvizsgálja, hogy a paraméterül kapott szövegben szerepel-e a megadott karakter. Ha igen, akkor a szintén paraméterül kapott tömbben *igaz*-ra állítja a karakter sorszámának megfelelő tömbelemet, ezzel jelezve, hogy azt a karaktert felhasználtuk, így többször már nem használható. A `vankarakter` függvény *igaz*-zal tér vissza, ha sikerült megfelelő, még szabad karaktert találnia a szövegben, és *hamis* értéket ad, ha nincs megfelelő karakter az első szövegben, ami még szabad lenne.




anagramma
hamis
igaz
Ki: "A két szöveg nem anagramma."
Ki: "A két szöveg egymás anagramma változata."
Vége
mindhamis
(tömb[]: logikai)
i: egész
i = 0 .. size(tömb) - 1
következő
kész
tömb[i] = false
Vége
szóköznélkül
(szöveg: szöveg)
vissza: szöveg
vissza = ""
i: egész
i = 0 .. len(szöveg) - 1
következő
kész
hamis
char(szöveg, i) != " "
igaz
vissza = vissza & char (szöveg, i)
Vissza: vissza
vankarakter
(szöveg: szöveg, karakter: szöveg, karaktervolte[]: logikai)
jelölve: logikai
jelölve = false
j: egész
j = 0
!jelölve && j <= len(szöveg) - 1
igaz
hamis
hamis
char(szöveg, j) = karakter && karaktervolte[j] = false
igaz
karaktervolte[j] = true
jelölve = true
j = j + 1
Vissza: jelölve





A megoldás során a szóközök számát figyelmen kívül hagyjuk, de más karaktereket (pl. mondatvégi írásjelek, vesszők) nem. A Flowgorithm-ben a kis- és nagybetűk különbözőnek számítanak összehasonlításkor, ezért érdemes a szöveget kisbetűkkel begépelni.

További szövegekkel kapcsolatos feladatok:

- Állapítsuk meg egy beolvasott szövegről, hogy eszperente nyelven íródott-e! (Vagyis vizsgáljuk meg, hogy az e-n kívül tartalmaz-e más magánhangzót. (A szöveg értelmességét itt sem tudjuk ellenőrizni.)
- Határozzuk meg egy beolvasott szöveg átlagos szóhosszát!

Tartalomjegyzék