Programozói szemlélet

Egy programozó programozási feladatainak megoldása során, anélkül, hogy észrevenné, egyre inkább egy sajátos viszonyba kerül a külvilággal. Kialakul az úgynevezett programozói szemlélete. A dologban az az érdekes, hogy ennek a szemléletnek a kialakulását sem siettetni, sem késleltetni nem lehet.

Ebben a fejezetben egy egyszerű matematikai feladat megoldásán keresztül szeretnénk bemutatni ennek a szemléletnek egy aspektusát. Ezt az aspektust, amit a probléma és környezete kapcsolatának töprengő felfedezéseként írhatunk körül, talán leginkább egy matematikából kölcsönzött szakszó jellemzi: *diszkusszív szemlélet*. Minden programozás-nyelvi környezet irányítja, illetve bizonyos mértékben korlátozza is a programozó szemléletét, azonban a programozói szemléletnek vannak nyelv független összetevői. Ilyen a diszkusszív megközelítés szemléleti összetevő is.

Feladat: Írjunk programot, amely kiszámolja a másodfokú egy ismeretlenes algebrai egyenlet gyökeit!

Minden másodfokú egy ismeretlenes algebrai egyenlet megadható $ax^2+bx+c=0$ alakban. Bizonyítható, hogy az egyenlet gyökei:

$$x1=\frac{-b+\sqrt{b2-4ac}}{2a} \text{ illetve } x2=\frac{-b-\sqrt{b2-4ac}}{2a}$$

A képletekben szereplő matematikai műveletek miatt két megszorítást kell tennünk. Egyrészt a főegyüttható nem lehet nulla, hiszen osztanunk kell *2a*-val, másrészt a gyökvonás műveletének értelmezése miatt a gyökjel alatti mennyiség nem lehet negatív.

Ezek az ismeretek elegendőek, ahhoz, hogy papír és ceruza segítségével elkezdhessünk megoldani egy ismeretlenes másodfokú algebrai egyenleteket.

A feladatot megoldó programnak, indítása után, először tájékoztatnia kell a felhasználót a program tevékenységéről: *„Ez a program kiszámolja egy másodfokú egy ismeretlenes algebrai egyenlet gyökeit”.* Az egyenlet általános alakjának illetve a megoldást szolgáltató képleteknek ismeretében a programnak ezután be kell kérnie az együtthatók értékeit: *„Az egyenlet általános alakja: $ax^2+bx+c=0$*”. *„Adja meg az a, b, c együtthatók értékeit!*” Innentől kezdve a feladat már nem matematikai, hanem gépi! Nem tételezhetjük fel ugyanis, hogy a bemenetet minden esetben humán intelligencia, azaz egy felhasználó fogja szolgáltatni. Elképzelhető, hogy számítógépünk egy műszerhez csatlakozik, amely rendszeres időközönként három számadatot küld feldolgozóprogramunk számára! Feltételezve a mindenkori felhasználói adatszolgáltatást, még ebben az esetben is fel kell készíteni programunkat a felhasználó esetleges tévedéseire vagy akár a „program határainak szándékos feszegetésére”.

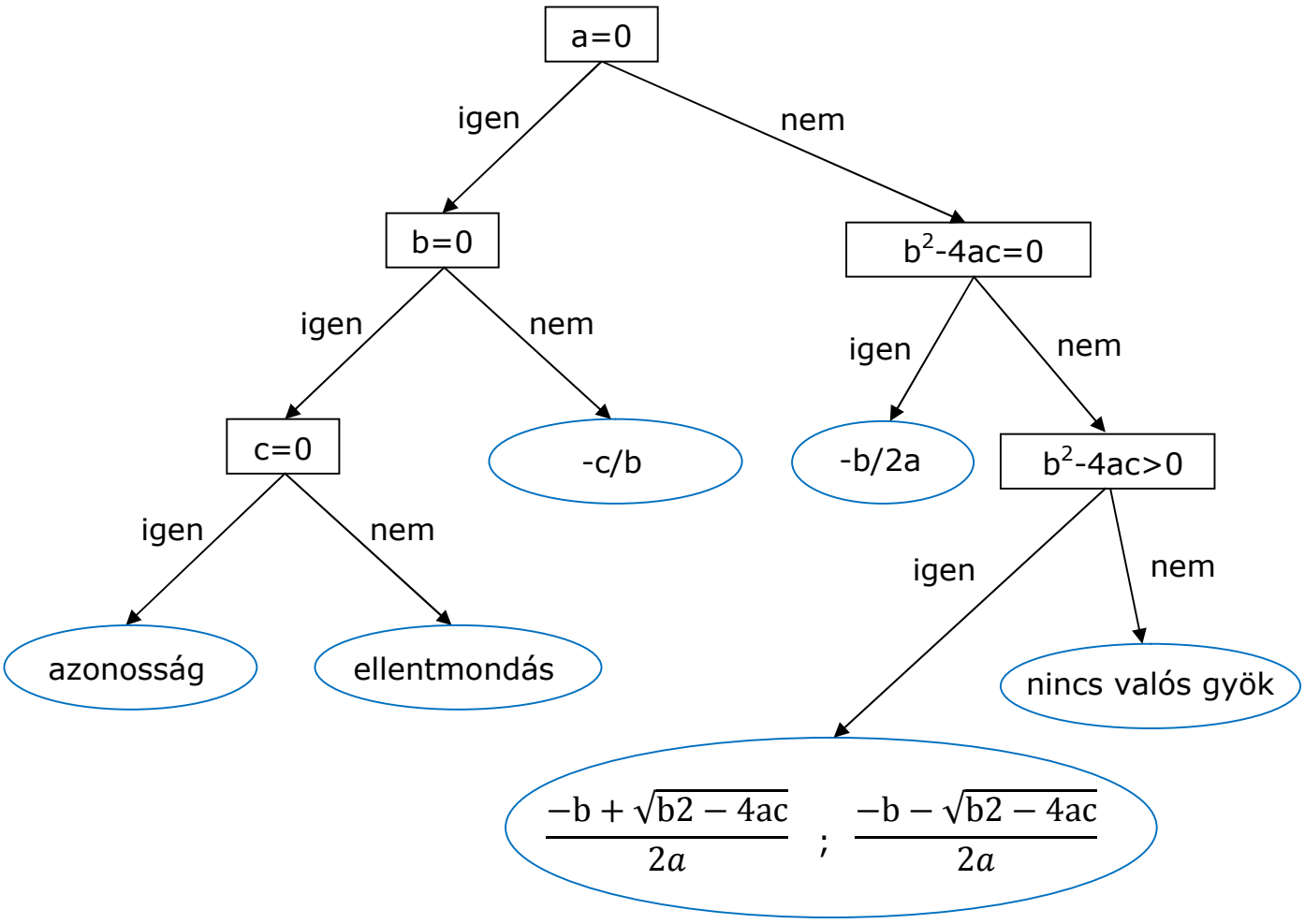
A programozónak tehát mindenképpen pontosan tisztába kell lennie azzal, hogy milyen *bemeneti adat*ok szükségesek illetve megengedettek, függetlenül attól, hogy ezt tájékoztatásul kiírja-e a képernyőre vagy nem, és

minden várható bemenetre fel kell készítenie programját. Az egyenlet megoldó programnak tehát minden lehetséges valós bemenetet kezelnie kell: *a, b, c* $\in$ *R.*

Vegyük sorra az eseteket! Ha a=0, képletünk nem működik, igaz egyenletünk már nem is másodfokú: $0x^2+bx+c = bx+c=0$. Ennek az egyenletnek a megoldása rendezés után: x=-c/b. Itt egy újabb probléma keletkezhet. Ha a=0 mellett b=0 is igaz, az elsőfokú egyenletet sem tudjuk megoldani! Most mit csináljunk? Nos, nem tehetünk mást, tiszta vizet öntünk a pohárba. Hogyan is néz ki az egyenlet most: $0x^2+0x+c=0$, azaz c=0. Ekkor további két eset lehetséges: c = 0 vagy c $\neq$ 0. Az első esetben egyenletünk végső alakja 0=0, ami egy azonosság, tehát minden valós szám megoldása az egyenletnek, a második esetben ellentmondó egyenlethez jutunk, tehát egyetlen valós megoldása sincs az egyenletnek. Ha a=0 mellett b $\neq$ 0, akkor megoldásunk mindenképpen x=-c/b, függetlenül c esetleges nulla voltától.

Ha a $\neq$ 0, valóban másodfokú egyenlethez jutunk. Ekkor viszont vizsgálnunk kell a gyökjel alatti mennyiséget, amely háromféle valós értéket vehet fel: pozitív, nulla, negatív. Ennek megfelelően megoldásunk újra elágazik három irányba. Az első esetben meg kell adnunk x1 és x2 képletekkel meghatározható értékét, a második esetben a megoldás –b/2a (kétszeres gyökként), a harmadik esetben pedig nincs valós megoldása az egyenletnek.

Az alábbiakban megrajzoltuk a programfolyamat hatlevelű logikai gráfját. A program lehetséges kimeneteit a kék körvonalú ellipszisek tartalmazzák.



A típus fogalmának kialakulása

Hogyan értelmezzük egy byte bitjeit?

Egy programváltozóhoz, melyben egész számokat szeretnénk tárolni, egybájtos memóriaterületet rendelünk. Ekkor 8 bit fog rendelkezésünkre állni a változóban tárolni kívánt információ kódolásához. Mivel minden biten 0 vagy 1 állhat, függetlenül a szomszédos bitek tartalmától, ezért $2^8 = 256$ féle különböző variációs lehetőségünk van, ennyi különböző nullákból és egyesekből álló sorozatot állíthatunk elő. Amennyiben az egybájtos változóban nem negatív egészeket tárolunk, a legkisebb lehetséges kettes számrendszerbeli, csak nullákat tartalmazó érték $00000000_2 = 0_{10}$, a legnagyobb lehetséges, csak egyeseket tartalmazó érték pedig $11111111_2 = 255_{10}$ lesz.

Ha negatív egészeket is szeretnénk tárolni a változóban, változtatnunk kell a tárolási struktúrán, hiszen valahogy jelezni kell a tárolt érték negatív voltát. Negatív szám tárolása esetén a legmagasabb helyiérték egyetlen bitjét használjuk fel, és 0-ról 1-esre billentjük azt. Ha például a -27_{10} értéket szeretnénk tárolni, mivel $27_{10} = 11011_2$, a vezető nullák kiírása után $27_{10} = 00011011_2$, ezért $-27_{10} = 10011011_2$ lesz. Ekkor a változót új típusú változóként kell deklarálnunk, a kiválasztott új típus nevén keresztül jelezve, hogy a legmagasabb helyiértéken álló 1-es most nem a $2^7=128$ számot jelenti, azaz a megadott jelkombináció által jelzett szám értéke nem 128+27=155. A korábbi típusnévtől eltérő új típusnevet éppen a szükséges értelmezés kikényszerítése miatt kell használnunk. Ekkor például az 11111111_2 nem a 255_{10} számot fogja jelenteni, hanem a -127_{10} számot. A típusok nevein keresztül tehát végeredményben a 0-1 kombinációk értelmezését rögzítjük. Érdekesség, hogy ebben a tárolási szisztémában két formálisan különböző *0* értékünk is keletkezik: $00000000_2 = 10000000_2 = 0_{10}$. Ezt az ábrázolási módot *előjeles abszolút értékes számábrázolás*nak nevezzük.

Az ***előjeles abszolút értékes számábrázolás***nak, azon a talán kevésbé zavaró tényen túl, hogy kétféleképpen is ábrázolhatjuk a nullát, van egy komoly hiányossága: egy számnak és a vele azonos abszolút értékű negatív párjának az összege nem nulla!

Az absztrakt algebrában egy halmaz elemein értelmezett műveletre nézve *neutrális elem*nek nevezzük a halmaz azon elemét, ha van ilyen, amely elem bármely halmaz elemmel együtt szerepeltetve a kérdéses műveletben, a művelet elvégzése során nem változtatja meg a halmazbeli elem értékét. Az egész számok halmazában az összeadásra nézve ilyen neutrális elem a *0*, ugyanis bármely egészre $n+0=n$. Ha a halmaz egy adott n eleméhez található olyan n^* halmazbeli elem, melyre igaz, hogy $n+n^*=0$, akkor n^* -ot *n additív inverz*ének nevezzük. A fentiek értelmében az előjeles abszolút értékes számábrázolásban a szám ellentettje sajnos nem úgy viselkedik az összeadási műveletben, ahogy azt elvárnánk. Új terminológiánk szerint ezt úgy is kifejezhetjük, hogy, eltekintve a *0* értéktől, az n szám és az n additív inverzének feltételezett n^* szám összege nem nulla!

Az úgynevezett ***kettes komplemens számábrázolás*** segítségével kiküszöbölhetjük az előjeles abszolút értékes számábrázolás hiányosságait. Egy tetszőleges szám additív inverzét ebben az esetben úgy képezzük, hogy a számot kettes számrendszerben leíró bitsorozat bitjeit ellentétesre változtatjuk, majd az így kapott értékhez egyet hozzáadunk. Az ily módon képzett additív inverz és az eredeti szám összege mindig nulla lesz. Az is teljesülni fog, hogy a *0* additív inverze formálisan is *0* lesz, megszűnik tehát a *0* jelölésének kétértelműsége.

Példák:

előjeles abszolút értékes additív inverz:

$27_{10}+(-27_{10})=0_{10}$, de $\boldsymbol{0}0011011_2+\boldsymbol{1}0011011_2=10110110_2 = -54_{10}$

kettes komplemens additív inverz:

$27_{10}+(-27_{10}) =0_{10}$, és $00011011_2+(\boldsymbol{11100100}_2+1_2)=\underline{1}\,\boldsymbol{00000000}_2$

$-0_{10}=0_{10}$, és $(-00000000_2)=\boldsymbol{11111111}_2+1_2=\underline{1}\,\boldsymbol{00000000}_2$

Az utóbbi két összeadás során a legnagyobb helyiértéken keletkező átvitel megőrzésére a nyolcbites tárolás következtében már nincs mód! Ezt úgy szokás kifejezni, hogy ekkor az átvitel a „semmibe fordul”. A ténylegesen eltárolt végeredmény így mindkét esetben valóban a *0* szám.

Megismerkedve a kettes komplemens számábrázolással felmerül egy érdekes kérdés: melyik a legkisebb ábrázolható negatív szám? Az eddigiek alapján egy ábrázolt 8 bites szám egyrészt jelentheti önmagát, de felfoghatjuk a számot úgy is, mintha egy fizikailag nem ábrázolt számnak a komplemens additív inverze lenne. Ebben az esetben a jelsorozat által szimbolizált érték kétféleképpen is értelmezhető lenne, ami semmiképpen sem lenne szerencsés! A problémát úgy oldhatjuk fel, hogy megtartjuk a komplemens additív inverz hasznosnak bizonyult szemléletet, de ugyanakkor felhasználjuk az előjeles abszolút értékes ábrázolásnál látottakat: a legnagyobb helyiértéken lévő egyes szám mindenképpen az őt követő hét bites szám negatív voltát jelezze! Figyeljük meg az alábbiakban, az alsóbb helyiértékek első hét bitjén egyesével növekvő pozitív számsorozatot illetve komplemens additív inverzeik sorozatát!

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	-0

0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	-1

0	0	0	0	0	0	1	0	2
1	1	1	1	1	1	1	0	-2

⋮

0	1	1	1	1	1	1	0	126
1	0	0	0	0	0	1	0	-126

0	1	1	1	1	1	1	1	127
1	0	0	0	0	0	0	1	-127

A képzett komplemens additív inverzek szisztematikusan fogynak az alsóbb helyiértékek első hét bitjén. A legkisebb kapott negatív érték a -127_{10}, ami a $+127_{10}$ komplemens additív inverze. Hét biten +127-nél nagyobb pozitív szám már nem képezhető, viszont a legkisebb értékű kapott negatív szám a csökkenés szabályosságát figyelembe véve még csökkenthető eggyel, igaz egyetlen szereplő pozitív számnak sem komplemens additív inverz párja. A legkisebb negatív szám tehát a -128_{10}, azaz az 10000000_2 bitsorozat. Elvárjuk, hogy egy additív inverz additív inverze is képezhető legyen, és az egyezzen meg az eredeti pozitív számmal. Ez így is van. Például -126_{10} additív inverze $01111101_2 + 1_2 = 01111110_2 = 126_{10}$. Képezhető-e -128_{10} additív inverze? Igen, az eredmény -128_{10}, de -128_{10} nem volt additív inverze egyetlen hétbites pozitív számnak sem!

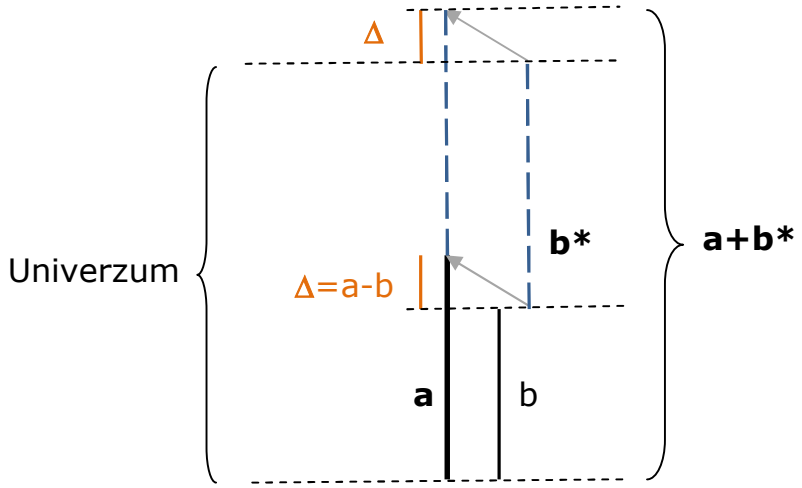
Fel kell még hívnunk a figyelmet a komplemens additív inverz eddigiekből már következő legfontosabb tulajdonságára: a kivonást ki tudjuk váltani a kivonandó abszolút értéke komplemens additív inverzének kisebbítendővel történő összeadásával! A processzornak tehát csak összeadni kell tudnia!

A memóriában tárolni kívánt kettes számrendszerbeli számok komplemensének képzése speciális eljárásnak tűnik, valójában azonban tetszőleges alapú számrendszerben megvalósítható. A következőkben tízes számrendszert használva is megmutatjuk az eljárást. Mivel rögzítettük a számrendszer alapszámát: 10, ezért az eljárást nevezhetjük 10-es komplemens képzésnek. Rögzítenünk kell még a rendelkezésünkre álló „digitek”, azaz ábrázolható helyiértékek számát is. Nem szabad ugyanis elfelejtenünk, hogy az eljárást egy számítógép processzora segítségével hajtjuk végre, és egy számítógép processzorának hardverében csak véges számú helyiérték alakítható ki. Az egyszerűség kedvéért tételezzük fel, hogy három helyiértéket használhatunk. Ekkor a legkisebb ábrázolható szám tízes számrendszerben *000*, a legnagyobb pedig *999*. Képezzük például a 16 komplemens additív inverzét: 999-16=983; 983+1=984. Tehát „-16” = 984! Ekkor ***16+(-16)=***16+984=1|***000*** !

Végezzünk el egy kivonást is az előbbi számítás felhasználásával! ***158-16 =*** 158+(-16)=158+(984)=1|***142***. Ebben a példában jól látható, hogy valóban sikerült kiváltanunk a kivonás műveletét egy összeadással. A tízes számrendszerben azonban a kivonás „kiváltása” nem valódi, hiszen a *16* additív inverzét kivonással kellett megalkotnunk: *999-16+1 = 984*. Műveleti táblát kellett tehát használnunk az egyes helyiértékeken: 9-6=3; 9-1=8; 9-0=9. A kettes számrendszerben azonban ténylegesen beszélhetünk a kivonás valódi logikai „kiváltásáról”, hiszen *1-0=1*; *1-1=0*, azaz nem kell műveleti táblát készítenünk, az egyes bitek egyszerűen alternálnak: ha töltött volt a bitnek megfelelő félvezető mikro kondenzátor, akkor kisütjük, ellenkező esetben egy áramlökéssel feltöltjük.

Az eddigiekben részletezett 2-es komlemens számábrázolást az egészek körében a JAVA nyelvben 1, 2, 4, és 8 byte-on is megvalósították. Az egyes típusok nevei sorrendben: ***byte*** [-128, 127]; ***short*** [-32768, 32767]; ***int*** $[-2^{31}, 2^{31}-1]$; ***long*** $[-2^{63}, 2^{63}-1]$.

A ***komplemens additív inverz*** előbbiekben tárgyalt fontos műveleti tulajdonsága algebrai eszközökkel tetszőleges számrendszerben bizonyítható. Az algebrai bizonyítás helyett most egy szemléletes, geometriai alapú, rajzos bizonyítást mutatunk:



A valós típusú számokat szokás ***lebegőpontos szám***oknak is nevezni, a tizedes pont ugyanis „lebeghet”, ide-oda vándorolhat a helyiértékeken. Ez azért lehetséges, mert a tízes számrendszer homogén csoportképzésű, azaz minden magasabb egység pontosan ugyanannyi alacsonyabb szintű egységből épül fel – tíz egyes egy tízes, tíz tízes egy százas, tíz százas egy ezres … és így tovább – tehát a csoportképzés struktúrája minden helyiértéken azonos. A maya számrendszerben ez nem volt így. Az első csoport csoportképzési alapszáma 20, a második csoport csoportképzési alapszáma viszont 18, majd minden újabb csoportszint ismét húsz egységből épült fel, a csoportképzés tehát inhomogén volt. Innentől kezdve viszont már nem mindegy, milyen helyiérték szinten osztunk. A tízes számrendszerben ismert osztási algoritmus nem érvényes a maya számrendszerben!

A fentiek értelmében a tízes számrendszerbeli valós számok tárolásakor elég csak az úgynevezett értékes jegyeket tárolni. Például a 0,00001703 tizedes tört tárolásakor elég az 1703 jegyeket eltárolni, illetve jelezni a tizedespont helyét. Mivel $0{,}00001703 = 0{,}1703 * 10^{-4}$, a tizedes pont ebben a jelölésben négy helyiértékkel jobbra mozgott. Ha megállapodunk abban, hogy az értékes jegyek alkotta számot minden esetben az előbbiekhez hasonlóan 0 és 1 közötti értékű számként ábrázoljuk, akkor az első 0 számjegyet és az őt követő tizedespontot sem szükséges tárolni, elégséges az *1703* és a *-4* számjegyek tárolása! Már csak abban kell megállapodni, hogy a négy bájt 32 helyiértékének mely szegmensét akarjuk az exponens tárolására használni, hogy az ily módon tárolt valós szám értéke mindig egyértelműen kiolvasható legyen. Figyelembe véve a rövid egészeknél mondottakat, a legmagasabb helyiértékű bájtot használjuk az exponens tárolására. Egyetlen bájton -128 és +127 között változhat az exponens értéke. Mivel a világegyetem kora másodpercekben 10^{17} nagyságrendű, a legkisebb ismert, még értelmezhető távolság méterben 10^{-35} nagyságrendű, a Tejútrendszer átmérője méterben kifejezve 10^{18} nagyságrendű szám, a

gyakorlati számításokban az exponens tárolására bőven elegendő egy bájt. A JAVA nyelvben ezt a típust *float*-nak (lebegőpontos valós) nevezik. A fenti megállapodásokban a helyiértékek szerepének leírása maga a típus leírás. A változók típusának megnevezése abban segít, hogy helyesen tudjuk visszakódolni akár egy négybájtos hosszú egész típusú változó bitjeit, akár egy esetleg ugyanazon 0-1 sorozatot tartalmazó, de valós típusú változó bitjeit.

Az alábbiakban érdekesség képpen közöljük Charles Babbage *Calculating Machin* című, az Edinburgh Review 1834 júliusi kötetében megjelent, informatikatörténeti szempontból híres cikkének egy rövid részletét, melynek magyar fordítását Győry Sándor közölte *Babbage Számoló Mozgonya* címmel, a Magyar Tudós Társaság *Tudománytár (4. kötet, 1834 december)* című, 1835 januárjában megjelent lapjában. [1]

BABBAGE' számoló mozgonya.

…

Mindazonáltal mind ezen munkálatokat is el lehet csupa öszveadás által végezni, a' kihuzandó helyett annak arithmeticai pótlékát irván, azután a' kettőt öszveadván 's a' resultatumból a' legfelsőbb egységet elvetvén. A' munkálat' menetelét 's okát könnyü lesz egy példából megérteni. Kivántassék: 768ból kivonni 357et. A' közönséges mód lenne a' mint következik:

	768 – ból
kivévén:	*357 – et*
marad	*411*

A' 357-nek arithmeticai pótolékja, vagy azon szám melly által 357 kipótoltatik 1000re = 643. Már ha ezen szám hozá adatik 768hoz 's a' balról eső egység az öszvettől elvettetik, a' munkálat következőkép megyen véghez:

	768 – hoz
adatván	*643*
leszen az öszvet	*1411*

's elvetvén az egységet a' keresett maradék 411.

…

[1] Közölte: Máté Eörs (2005)