Programgráf bonyolultsága

Definíció: A programgráf ciklikus bonyolultsága

A P programgráf ciklikus bonyolultságának nevezzük az $m(P) = e - c$ mérőszámot, ahol e a programgráf éleinek a száma, c pedig a programgráf csúcsainak a száma.

A ciklikus bonyolultság azt próbálja mérni, hogy egy programgráf kódolása (az algoritmus programkódjának elkészítése) során milyen mennyiségű munkát kell elvégeznünk. Nem ad viszont választ arra, hogy milyen kódolási nehézségeink lesznek. Ezáltal egy algoritmus két különböző implementációjának a bonyolultságát nem tudjuk érdemben általa összehasonlítani.

Példa:

Egy terjedelmesebb programgráfban nem könnyű összeszámolni az éleket és a csúcsokat. Ennek megkönnyítése végett mondjuk ki a következő tételt.

Tétel: <u>A programgráf ciklikus bonyolultságának tétele</u>

A P programgráf ciklikus bonyolultsága kiszámítható a következő módon is:
$m(P) = p + 1$, ahol p a programgráf predikátumainak a száma.

Bizonyítás:

Jelölje a P programgráf predikátumainak a számát p, a függvények (szekvenciák) számát f és a gyűjtők számát g! Az élek száma továbbra is e, a csúcsok száma c legyen! Ezen jelölések mellett előbb belátjuk a következő **állítást**. Állítás:

1. $e = 3p + f + 1$
2. $g = p$.

Az állítás belátása egyszerű. A csúcsok száma nyilvánvalóan $c = p + f + g$. Az élek számát kétféleképpen fogjuk meghatározni. Egyszer vesszük az egyes csomópontokba bevezető éleket, amelyekhez még hozzávesszük a STOP-ba vezető élt, a programgráf kimenő élét, majd másodszor vesszük a csomópontokból kivezető éleket és hozzávesszük a START-ból induló kezdőélt, a programgráf bemenő élét. Eszerint egyrészt $e = p + f + 2g + 1$, másrészt $e = 2p + f + g + 1$. A két formulát egymással egyenlővé téve kapjuk, hogy $p+f+2g+1 = 2p+f+g+1$, ahonnan összevonások után a $p = g$ összefüggésre jutunk, azaz a programgráfban a predikátumok és a gyűjtők száma megegyezik. Ha most a g helyére az élekre vonatkozó két formula bármelyikébe behelyettesítjük a p értékét, akkor kapjuk az első állításunkat: $e = p + f + 2p + 1 = 3p + f + 1$.

Ezen állítás segítségével pedig a ciklikus bonyolultság

$$m(P) = e - c = (3p + f + 1) - (p + f + g) = p + 1$$

■

Tétel: <u>A nem struktúrált program ciklikus bonyolultságának tétele</u>

A nem struktúrált P programgráf ciklikus bonyolultsága legalább három, azaz $m(P) \geq 3$, ha P nem struktúrált.

A nemstruktúráltságot karakterizáló négy alapeset bármelyikének a ciklikus bonyolultsága a programgráfjáról leolvashatóan három, és miután a nem struktúrált programban ezek legalább egyike előfordul, ezért a programé legalább három.

Többet mond a bonyolultságra az összehasonlíthatóság szempontjából a következő definíció.

Definíció: A programgráf lényeges bonyolultsága

Legyen k a P programgráf azon részgráfjainak a száma, amelyek az ***E***, ***C***, ***I*** formulák valamelyikével felírhatók (lebontási lépésszám kizárva belőle a szekvenciák esetét). A P programgráf lényeges bonyolultságának nevezzük az $M(P)=m(P)-k$ számot. Ez tulajdonképpen a lebontással elérhető ciklikus bonyolultság.

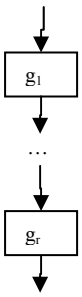
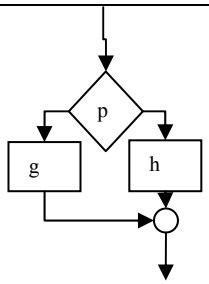
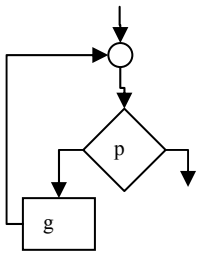
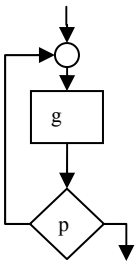
Tétel: <u>A struktúrált program lényeges bonyolultságának tétele</u>

A struktúrált P programgráf lényeges bonyolultsága egy, azaz $M(P)=1$, ha P struktúrált.

Nem struktúrált esetben nem tudjuk három alá csökkenteni a lényeges bonyolultságot.

Példa programgráfok.

A struktúrált programgráf elemeinek a ciklikus bonyolultságát az alábbi táblázatban foglaltuk össze.

Név / Leírás	Üres program	Szekvencia	Elágazás	Ciklus	Iteráció
gráf					
formula	$\varnothing$	$\boldsymbol{S}(g_1,\ldots,g_r)$	$\boldsymbol{E}(p;g,h)$	$\boldsymbol{C}(p;g)$	$\boldsymbol{I}(p;g)$
n	1	$r+1$	6	5	5
c	0	r	4	3	3
$m(P)=n-c$	1	1	2	2	2

Legyen egy program, amely az $y=f(x)$ függvényt számítja ki.
Input: $x \in X$, Output $y \in Y$.

1.	Legyen *f* a *g* és a *h* függvények *kompozíciója* $f=h \circ g$, azaz $f(x)=h(g(x))$	Programozása szekvenciával $z \leftarrow g(x)$ $y \leftarrow h(z)$
2.	Legyen *f alternatíva*, azaz $y=f(x)=\begin{cases} g(x) & ha\ \ T(x) \\ h(x) & ha\ \ \overline{T(x)} \end{cases}$	Programozása elágazással **IF** $T(x)$ **THEN** $y \leftarrow g(x)$ **ELSE** $y \leftarrow h(x)$
3.	Legyen *f több függvény összege*, azaz $y=f(x)=\sum_{i=1}^{n} g_i(x)$	Programozása ciklussal $y \leftarrow 0$ **FOR** $i \leftarrow 1$ TO n DO $y \leftarrow y+g_i(x)$

4.	Legyen *f rekurzió*, azaz
	$y=f(x)=\begin{cases} g(x) & ha\ \ T(x) \\ (h \circ f \circ i)(x) & ha\ \ \overline{T(x)} \end{cases}$
	
	Formális pszeudokódja rekurzióval
	IF $T(x)$
	THEN $y \leftarrow g(x)$
	ELSE $z \leftarrow i(x)$
	$z \leftarrow f(z)$
	$y \leftarrow h(z)$
		Formális pszeudokódja iteratív ciklussal
		$k \leftarrow 0$
		WHILE $\overline{T(x)}$ **DO**
		$x \leftarrow i(x)$
		INC(k)
		$v \leftarrow g(x)$
		FOR $j \leftarrow 1$ **TO** k **DO**
		$v \leftarrow h(v)$
		$y \leftarrow v$

Példa rekurzióra:

$$i(x)=x-1$$
$$h(x)=x+1$$
$$g(x)=c$$
$$T(x)=(x<1)$$
$$y=f(x)=\begin{cases} c & ha\ \ x<1 \\ f(x-1)+1 & ha\ \ x \geq 1 \end{cases}$$

Néhány számított érték:

$$f(0)=c$$
$$f(1)=f(0)+1=c+1$$
$$f(2)=f(1)+1=(f(0)+1)+1=(c+1)+1=c+2$$
$$f(3)=f(2)+1=(f(1)+1)+1=((f(0)+1)+1)+1=c+3$$
$$\vdots$$
$$f(n)=f(n-1)+1=\ldots=c+n$$

Programtervezés felülről-lefelé (Top-Down Program Design)

A *felülről-lefelé programtervezési elv* az algoritmusnak először egy nagyvonalú leírását adja, amelyben csak absztrakt szinten írjuk le a lépéseket és az adatstruktúrákat. Szokás ezt az elvet kiegészíteni a *lépésenkénti finomítási elv*vel (stepwise refinement), amely az egyes lépéseket fokozatosan részletezi addig, amíg már az egyes apró lépéseknek a programkódja világos módon elkészíthetővé nem válik. A felbontás egyre kisebb, finomabb, szűkebb részekre történik, eközben egyre nyilvánvalóbbá válik, hogy melyek lesznek a konkrét programnyelvi utasítások. Egy rövid, durva vázlat fimomítódik, amíg a legalsó szintű részletek is nem tisztázódnak. A legfelső szinten absztrakt adattípusokat és objektum műveleteket használunk, amelyek implementációja még esetleg meg sem történt, és amelyeknek még csak a viselkedési leírása, specifikációja adott. Ez elősegíti a pontos, világos gondolkodást, mely által tiszta programot írhatunk, amelyet az alsóbb részletek nem ködösítenek el. A későbbi fázisokban megválasztjuk az adatreprezentációkat és az algoritmusokat, hogy a magasszintű absztrakciót implementáljuk. Ez teszi a részekből álló rendszert együttműködővé, miközben megőrizzük a felső szint tisztaságát.

A felülről-lefelé elvet a struktúrált programozással összekötve a programgráf megtervezésekor felülről-lefelé haladva mindig struktúrált ábraelemekre korlátozódunk. Ennek az előnye, hogy struktúrált lesz az elkészült program, várhatóan kevesebb lesz az elkövetett hiba, könnyebb lesz a program helyességének belátása (bizonyítása), és a helyesség lépésről-lépésre „beépül” a programba. Célszerűnek látszi az eltolt bekezdések használata és az egy-kétoldalas programmodulok készítése.

A *funkcionális felbontás elve* azt tartja szem előtt, hogy az algoritmus leírásakor törekedni kell arra, hogy a funkcionálisan együvé tartozó, egymással szorosan összefüggő részeket, lépéseket együtt, egy helyen tárgyaljuk. A lazábban kapcsolódó részeket szétválaszthatjuk, ezáltal az algoritmus természetes módon esik szét kisebb, jobban kezelhető részekre.

A fenti elveket általában nem lehet élesen elválasztani egymástól, a fejlesztés során ezek egymással együtt működnek.

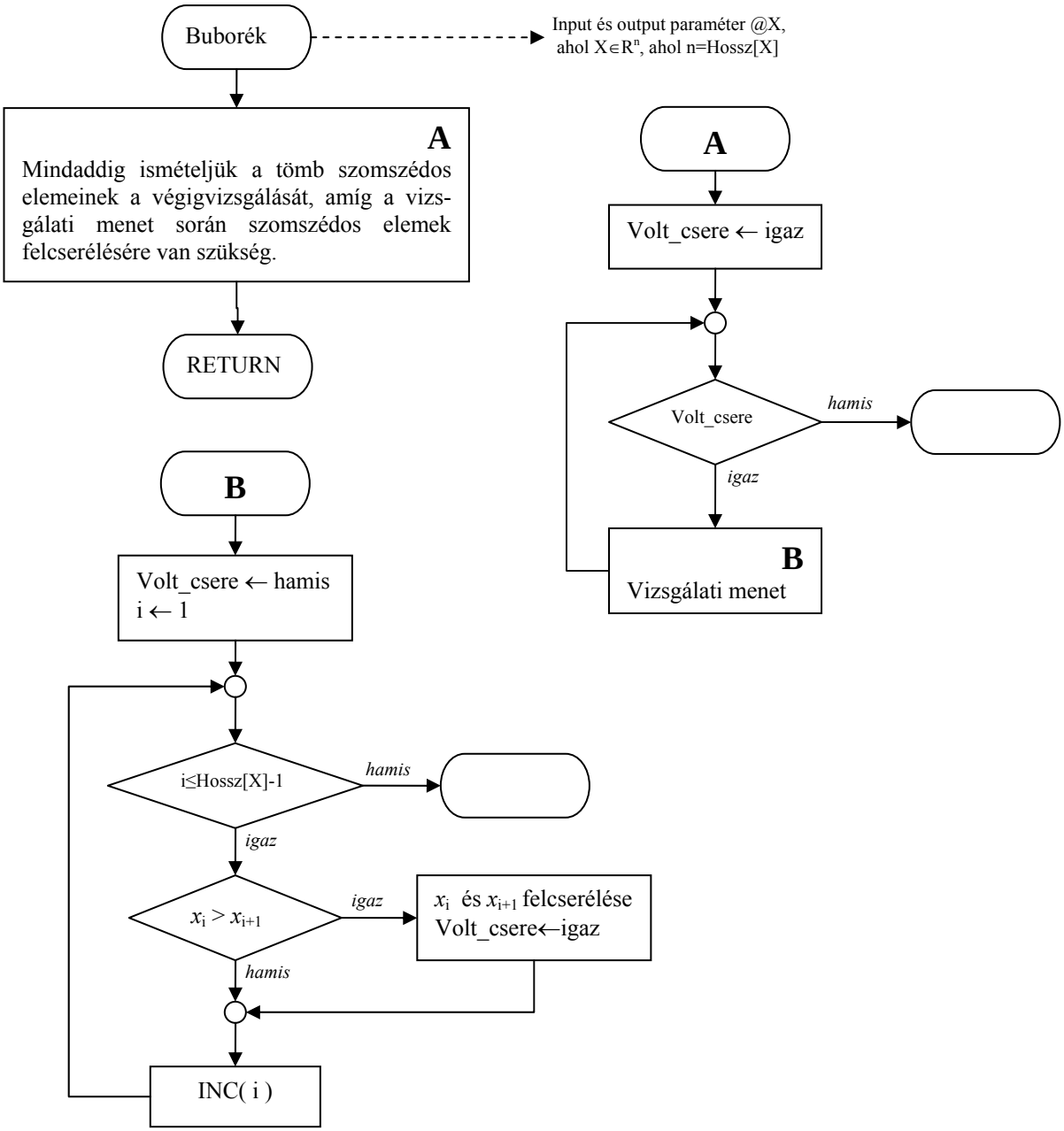
Az *alulról-felfelé programtervezés*t (Bottom-Up Program Design) akkor célszerű követni, ha sok előzetesen összeállt, korábbról összegyűjtött apró részalgoritmusnak már ki van dolgozva a realizálása. Ekkor ezekből az apró építőelemekből kombináljuk, rakjuk össze a bonyolultabb algoritmust.

Tekintsük a rendezési feladatnak a megoldásaként a Buborék rendezési algoritmus folyamatábrája megrajzolását. Az algoritmus a következőképpen foglalható össze szavakba.

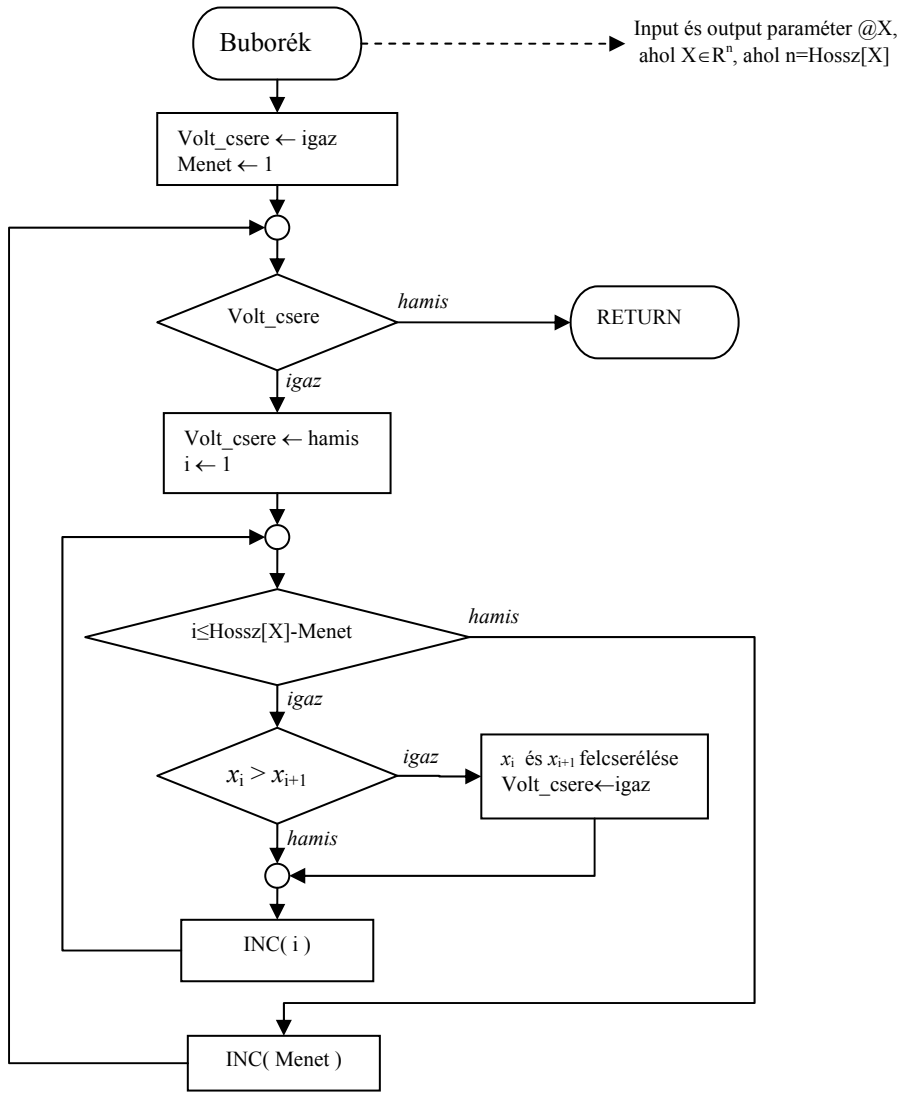
Legyen a feladat egy számokat tartalmazó tömb elemeinek a növekvő sorrendbe rakása. A tömb legyen az X tömb, melynek attribútuma n=Hossz[X]. A tömbelemek indexelése 1-gyel kezdődik. A módszer: megvizsgáljuk a tömb szomszédos elemeit, hogy növekvő sorrendben vannak-e, vagy sem. Amennyiben helytelen a sorrend, akkor helycserét hajtunk végre. Ha a tömbön végighaladva nem kell cserét végezni, akkor kilépünk az algoritmusból.

Konkrét számpéldán szemléltetjük az algoritmust. Legyen a kiinduló tömb 6,2,7,8,3,1. Az első vizsgálati menetben a 6,2 felcserélése után kapjuk a 2,6,7,8,3,1 sorrendet, amelyben folytatva a vizsgálatot a 8,3, majd a 8,1 cseréje következik, azaz az első vizsgálati menet végén a tömb 2,6,7,3,1,8 lesz. Mivel a menet során volt csere, ezért újabb menet jön. Most a 7,3 és a 7,1 cseréjére van szükség. A menet végén a tömb 2,6,3,1,7,8 lesz. Volt csere, tehát

jön a harmadik menet. A cserék: 6,3, azután a 6,1. Ezzel kialakul a 2,3,1,6,7,8 sorrend. Az újabb menetben a 3,1 cserét végezzük el, amire a sorrend 2,1,3,6,7,8 lesz. A következő menetben a 2,1 cserére van szükség. Ezáltal a sorrend 1,2,3,6,7,8 lesz. Mivel volt a vizsgálati menetben csere, ezért még egy menet lesz, amely már csere nélkül zajlik le. Ezzel kilépünk az algoritmusból. Vegyük észre, hogy az első menet során a legnagyobb elem a végleges helyére kerül, ezért a második menetben eggyel kevesebb összehasonlítást kell tenni. Mindegyik menetben egy elem a végleges helyére kerül, tehát minden menet után a következőben az összehasonlítások száma eggyel kevesebb lesz. Tehát, ha *n* elem van, akkor legfeljebb *n*-1 menetre van szükség a legrosszabb esetben. Ha a számok eleve rendezettek, akkor egyetlen menet után az algoritmus véget ér.
A folyamatábra tervezése során igyekszünk figyelembe venni a felülről-lefelé programtervezési elvet.



Látjuk, hogy a tervezés során nem használtuk ki, hogy minden menetben az összehasonlítások száma eggyel csökken. Most, amikor az ábrát összeszerkesztjük egybe, akkor ezt a korrekciót elvégezhetjük, amint azt az alábbi ábrán meg is tesszük.



Pszeudokód

Procedúra **Buborék_rendezés** (*@X*)

Input paraméter: $X \in \boldsymbol{R}^n$, ahol n=Hossz[X]
Output paraméter: $X \in \boldsymbol{R}^n$

// Valós számokat az eredeti tömbben növekvő sorrendbe
// rendez legfeljebb konstans mennnyiségű tovább
// memória felhasználásával.

Volt_csere ← *igaz*
Menet ← 1

WHILE *Volt_csere* **DO**
    Volt_csere ← *hamis*
    FOR *i* ← 1 **TO** *Hossz*[*X*]-*Menet* **DO**
        IF $x_i > x_{i+1}$
            THEN x_i és x_{i+1} felcserélése
                Volt_csere ← *igaz*
    INC(*Menet*)

RETURN (*X*)

Struktogram

Procedúra **Buborék_rendezés** (*@X*)

Input paraméter: $X \in \boldsymbol{R}^n$, ahol n=Hossz[X]
Output paraméter: $X \in \boldsymbol{R}^n$

// Valós számokat az eredeti tömbben növekvő sorrendbe
// rendez legfeljebb konstans mennnyiségű tovább
// memória felhasználásával.

Volt_csere ← *igaz*

Menet ← 1

Volt_csere

Volt_csere ← *hamis*

i ← 1

i ≤ *Hossz*[*X*]-*Menet*

$x_i > x_{i+1}$

igen | *nem*

$y \leftarrow x_i$
$x_i \leftarrow x_{i+1}$
$x_{i+1} \leftarrow y$
Volt_csere ← *igaz*

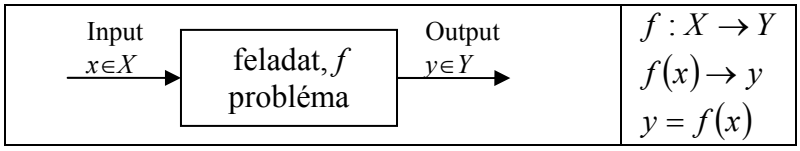
INC(*i*)

INC(*Menet*)

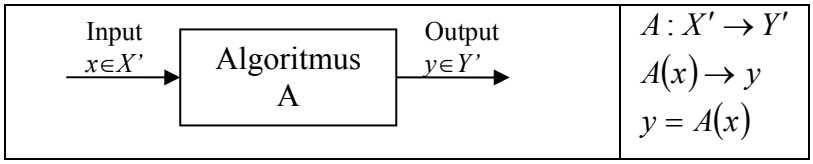
Egy teljes feladat megoldásának a fázisai lehetnek az alábbiak.

1. A feladat megfogalmazása
2. A feladat modelljének a felépítése
3. A modell keretein belül működő algoritmus kidolgozása, elkészítése (szöveg, pszeudokód, folyamatábra, struktogram, stb.)
4. Az algoritmus helyességének ellenőrzése
5. Az algoritmus bonyolultságának az elemzése
6. Az algoritmus realizálása, implementálása, programkészítés
7. A program tesztelése és javítása
8. Dokumentáció elkészítése (az 1-7 pontok leírása ismertetése már menet közben)

A feladat vagy probléma megfogalmazását az alábbi sematikus ábrával jellemezhetjük.



Az *f* összefüggés helyett, amely az *x*-ből *y*-t készít, valójában mi egy A algoritmust fogunk alkalmazni A helyzet az, hogy *f* a valóságban hat, mi pedig egy modell keretei között dolgozunk, amely csak utánozni igyekszik a valóságot.



Kérdés, hogy az A algoritmus valóban alkalmas-e arra, hogy az *f* függvény helyett álljon.

Definíció: Függvény helyes realizációja algoritmussal
Azt mondjuk, hogy az A algoritmus helyesen realizálja az *f* függvényt, ha $X \subset X'$, $Y \subset Y'$ és $A(x)=f(x)$, $\forall x \in X$.

Vizsgáljuk meg a tárgyalt négyzetgyökvonási algoritmusunkat ezen szemszögből. Annak a bemenetére elvileg bármilyen nemzérus kezdőértéket ráengedhetünk. Mit ad negatív kezdőértékekre az algoritmus? (Pozitívakra a kért pontosság mellett a pozitív négyzetgyököt szolgáltatja, láttuk.)

A programhelyesség bizonyítása azt jelenti, hogy belátjuk, hogy a program teljesíti a kitűzött célokat, megvalósítja, eleget tesz az előírásoknak, azaz a program specifikációnak, amely logikai formában leírható. A bizonyítás matematikai bizonyításnak tekinthető.

A programtesztelésre azért van szükség, mert a helyességbizonyításban is lehetnek hibák, akár az érvelésben, akár a specifikációban. A tesztelés szükséges. A tesztelés kimutatja a hiba létezését, de sohasem bizonyítja azok teljes hiányát. A célunk az, hogy minél nagyobb megbízhatósággal állíthassuk, hogy a programunk pontosan azt teszi, amit tennie kell.

A program az algoritmus implementációja az adott hardverre. Másrészt a program tulajdonképpen egy leképezés a bemenő és a kimenő adatok között, ezért beszélhetünk *programfüggvényről*. A program a leképezés kiszámítási szabályát foglalja magában és azt a programgráffal tudjuk szemléltetni, szerkezetét feltárni. Minden valódi programhoz meg lehet

szerkeszteni egy vele ekvivalens (struktúrált) programszekezetet. A programokhoz hozzárendelhető egy programszerkezeti bonyolultsági mérőszám, amely a struktúrált programok esetén a legkisebb.

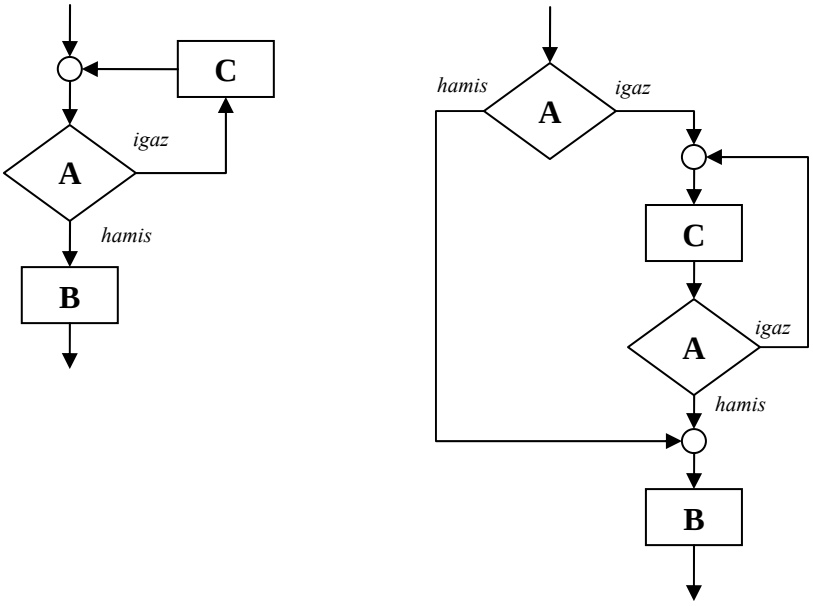
A program által megvalósítandó leképezést *specifikáció*nak nevezzük, a program rögzíti a specifikáció megvalósításához tartozó *kiszámítási szabály*t. A programozás alapvető problémája olyan kiszámítási szabály (program) előállítása, amely egy előre rögzített függvényt (specifikációt) definiál. A fogalmak párba állíthatók függvény↔kiszámítási szabály, valamint specifikáció↔program. Egy függvény megadásához szükség van az értelmezési tartományra, az értékkészletre, valamint a kettő közötti leképezésre. Ezeket formálisan az alábbi módon írhatjuk le.

Értelmezési tartomány	$D(f)=\{x\|P(x)\}$
Értékkészlet	$R(f)=\{y\|Q(y)\}$
A leképezés	$f=\{(x,y)\|F(x,y)\wedge x\in D(f)\wedge y\in R(f)\}$

Itt $P(x)$ és $Q(y)$ predikátumok, $F(x,y)$ pedig az (x,y) pár elemeit egymáshoz rendelő predikátum. A szokásos jelölés $y=f(x)$, ahol y neve függvény érték, x neve argumentum. Véges $D(f)$ és $R(f)$ esetén a párok felsorolhatók, ekkor $f=\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$. Szokásos a párok formulával történő megadása. Például: $f=\{(x,y)\|y=x(x-1)\wedge x\in R\}$, vagy $y=x(x\text{-}1)$ valós x számra.

A függvény vizsgálatához általában nincs szükség a kiszámítási szabályra, amely a függvényértéket az argumentumból előállítja. Amikor viszont használjuk a függvényértéket, akkor a kiszámítási szabály, és az őt realizáló, implementáló algoritmus fontossá válik. A kiszámítási szabály általában nem egyértelműen meghatározott. Például az $y = x^2 - x$ és az $y = x(x - 1)$ két szabály ugyanazt a függvényt határozza meg. Általánosan kérdés, hogy két szabály ugyanazt a leképezést definiálja-e. Ennek eldöntése nem mindig egyszerű probléma. Az implementáció során pedig további problémák lehetnek. Például, ha az első esetben az x^2 kiszámítása túlcsordulást eredményez, akkor nincs helyes eredmény, míg mondjuk a második esetben előfordulhat, hogy éppen nincs túlcsordulás.

Példa két ekvivalens program gáfjára, amelyeknek különböző a ciklikus bonyolultságuk.



A programot valamely programozási nyelv utasításainak a segítségével készítjük el. Minden utasítás végülis egy függvénynek fogható fel. Belőlük épül fel a program. Mindegyiknek ugyanaz az értelmezési tartománya, éspedig a program adatmezeje. $D(f)=\{d_0,d_1,d_2,\ldots\}$ adatmező. Minden utasítás egy d_i adatot a program egy s_{i+1} állapotára képez le. $s_{i+1}=f(d_i)$. Az s állapot összetevői $s = (d, f)$. adat és utasítás pár. A program végrehajtása az $s_i = (d_i, f)$, i=0,1,2,… állapotok sorozata. Minden egyes utasítás végrehajtása egy új d_{i+1} és egy új f_{i+1} utasítást eredményez. Az f_i utasítás értékkészlete $R(f_i)=S=D\times F$, ahol $F=\{ f_0, f_1, f_2,\ldots\}$ a programozási nyelv utasításainak a halmaza. A programvégrehajtás kezdő állapota s_0. A programvégrehajtás eredménye v, ha véges sok lépésben befejeződik, és $u= s_0, s_1, s_2,\ldots, s_n=v$. A program az adott F révén egy kiszámítási szabályt ad meg $\{(u_1, v_1), (u_2, v_2),\ldots, (u_n, v_n)\}$, ha véges sok lépésben véget ér. Azon, hogy a program végrehajtása véges sok lépésben véget ér, azt értjük, hogy a programot végrehajtó gép egy speciális utasítás hatására abbahagyja az adott program utasításainak a végrehajtását és átadja a vezérlést egy másik programnak, általában a rendszerprogramnak, az operációs rendszernek. A program végrehajtása befejeződik az i-dik lépés után, ha $f_i(d_i)\notin S$. Olyan P programhoz, amely befejeződik, egy p programfüggvény rendelhető és maga a P program erre a programfüggvényre egy kiszámítási szabályt rögzít. Ugyanazt a programfüggvényt két vagy több különböző program is meghatározhatja.

A program egy feladatot old meg. A feladatot le kell írni, specifikálni kell, úgynevezett előfeltételt és utófeltételt kell adnunk. Az előfeltétel megadja, hogy mi lehet az input, az input adatok minek kell, hogy megfeleljenek, az utófeltétel pedig azt ellenőrzi, hogy a helyes inputok esetén helyes-e az eredmény. A programban használt változók, memóriarekeszek minden egyes utasítás végrehajtását követően egy állapotot határoznak meg. Általában a programot egy állapottér jellemzi és a program végrehajtása az ezen állapottér egyes elemein történő végighaladás a végállapotig. Például, ha az $ax^2+bx+c=0$ tetszőleges valós együtthatójú egyenlet valós megoldásait keressük, akkor az inputra felírhatjuk, hogy a, b, $c\in\boldsymbol{R}$, az outputra, hogy legyen $m\in\boldsymbol{S}$, $x_1,x_2\in\boldsymbol{R}$. Az outputban a m szerepe egy magyarázó szöveg, hogy az eredményt hogyan kell értelmezni. Az eredmény lehet ugyanis olyan, hogy két valós megoldás van, egybeeső valós megoldások vannak, nincs valós megoldás, de lehet, hogy az egyenlet elsőfokú, akkor csak egy valós megoldás lehet, vagy ha nulladfokú az egyenlet, akkor pedig lehet minden valós x megoldás, vagy lehet, hogy egy megoldás sincs, mert ellentmondó az egyenlet. Ezek az esetek a abból következnek, hogy mely együtthatók zérusok. Ha másodfokú esetről van szó, tehát $a\neq0$, akkor ki kell számolnunk a $d=b^2-4ac$ diszkriminánst, $c\in\boldsymbol{R}$, és ennek a milyensége dönt a megoldások számáról. A program állapotterét ekkor formálisan leírhatjuk, mint az $\boldsymbol{R}^3\times\boldsymbol{S}\times\boldsymbol{R}^2\times\boldsymbol{R}$ halmazt, ahol a Descartes szorzatban szereplő összetevők (koordináták) rendre az a, b, c, m, x_1, x_2, d változók.