A programfejlesztés gazdaságossága - a COCOMO

A programfejlesztés nem csak kimondottan programozást jelent, hanem gazdasági tevékenység is egyben. Ez azt jelenti, hogy egy feladat felvállalásakor reális tervekkel, ajánlatokkal kell fellépni. Ezzel kapcsolatban jelent meg 1981-ben a Barry W. Boehm által megírt Software Engineering Economics című könyv, amelyben leírta a COCOMO-t (Constructive Cost Model), a Konstruktív Költség Modellt. (Barry Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. ISBN 0-13-822122-7) Gyakran COCOMO 81 néven hivatkoznak rá, mivel 1997-re kidolgozták a mai viszonyokra jobban alkalmazható COCOMO II változatot. Lásd: Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ Prentice Hall, 2000. ISBN 0-13-026692-2 könyv. Az első verzióba tekintünk be. Ez egy gazdasági hátteret feszegető modell, amely választ igyekszik adni arra, hogy hogyan kalkuláljunk egy feladat (projekt) munkaigényességével, vállalt határidővel és létszámmal. A modellben fontos szerepet játszik a munkamennyiség mérőszáma, ami *emberhónap*, *PM*=Person Month, a *fejlesztési idő*, T_d=Development Time, valamint az elvégzendő munka dokumentációsorokban megjelenő mennyisége, elsősorban beleértve a kifejlesztendő programsorok számát. Ez utóbbit *KDSI*=Kilo-Delivered Source Instruction jelöli, a forrássorok száma ezres sorokban mérve.

Az empirikus (tapasztalati) adatokra támaszkodó elemzések azt az érdekes összefüggést tárták föl, hogy komolyabb méretű munkák esetén jó közelítéssel teljesül a

$$PM=2,4\cdot KDSI^{1,05}$$

összefüggés, azaz a befektetendő munka mennyisége emberhónapban kifejezve a véglegesített anyag méretével a megadott formula szerint változik. A formulában szereplő konstansok nem nagyon szigorúak, főleg a 2,4-es szorzószám. Ez a munkahelytől, annak felszereltségétől, az ott dolgozó csapat összeszokottságától, tapasztalatától függ, de jó átlagnak tekinthető, ha más nem áll rendelkezésre. Az látszik az egytől nagyobb kitevő révén, hogy ha egy munka mérete duplájára nő, akkor az elvégzéséhez szükséges idő várhatóan több mint duplája lesz (kb. 7%-kal lesz több). A modell más összetevőket is tartalmaz, maga a 2,4-es szorzó is több mérőszámból tevődik össze. PM ismeretében azonban még nem tudunk megbízható vállalási időbecslést adni a munka elvégzésére, mert elvileg sok ember ráállításával tetszőlegesen rövid határidőre tudnánk a vállalási időt lenyomni, ami nyilvánvalóan nem lehet igaz. Egy bizonyos létszám fölött már nem lehet a munkát hatékonyan végezni, veszélybe kerül a határidős teljesítés. A modell javasol egy fejlesztési időt,

$$T_d = 2,5\cdot\sqrt[3]{PM}\ .$$

Ez az az idő, amely alatt egy szokványos csapat a munkát megbízhatóan és csúszás nélkül várhatóan elvégzi. A *PM* és a T_d hányadosa alapján kalkulálhatjuk a leginkább megfelelő létszámot. Szokás az irányító személyek körében, hogyha fejlesztés közben valami ok miatt veszélybe kerül a határidő, vagy gyorsítani szeretnének, akkor további személyeket állítanak rá a munkára. Ez nem mindig válik be. *Brook* megfigyelései alapján ugyanis minél többen dolgoznak egy projekten, annál nehezebb a koordináció, a kommunikáció, és az új emberek betanítása. Gyakran a dolog fordítva sül el és a több ember ellenére még az eredeti határidőt sem tudják tartani. Tehát a T_d csökkentése nem ajánlatos, legalábbis meggondolandó. Az a tapasztalat, hogy ha T_d csökkentése a 75%-a alá megy, akkor óriásivá válik a rizikó a kudarcra.

Ezt a $T_d \cdot 0{,}75$ értéket hívják *Brook korlát*nak, *Brook limit*nek, ami alá nem tanácsos menni a létszám bővítése révén.

Példa. Legyen egy programozó csoport, amely egy munka elvállalása során becslések alapján a fentieket figyelembe véve kalkulál, és úgy találja, hogy a kifejlesztendő szoftver várhatóan 70 000 soros lesz. Határozzuk meg, hogy mennyi időre, és mennyiért vállalja a csoport a megbízást. Tegyük fel, hogy egy programozó havonta félmillió forintba kerül a cégnek (bér, járulékok, rezsi, stb.) A nyereséget ne kalkuláljuk most bele.

A munka elvégzéséhez szükséges emberhónapok száma: $PM=2{,}4 \cdot 70^{1{,}05}= 207{,}76$ emberhónap, azaz valamivel több, mint 17 év. Egy emberre ez nyilvánvalóan nem bízható. Ez a hónapszám beszorozva a havi félmillió forinttal 103,88 milliót, azaz körülbelül 104 millió forintot eredményez. A vállalási idő ezután $T_d = 2{,}5 \cdot \sqrt[3]{207{,}76} = 14{,}80$ hónap lenne a formula szerint. Célszerű tehát PM/T_d-számú embert ráállítani a munkára, ami kb. 14,03, azaz mondjuk, 14 főt állítunk a munkára. Ebben az esetben a vállalási idő 207,76/14=14,84 lenne. Legyen 15 hónap kerekítve. A Brook korlát $14{,}8 \cdot 0{,}75=11{,}1$ hónap. Tehát ha gyorsítani kellene a munkán további emberek bevonásával, akkor ez alá a 11,1 hónap alá rizikómentesen nem mehetünk.

Tegyük fel, hogy a piacon kapható olyan procedúra csomag, amely az elkészítendő programsorok 20%-át kiváltja, tehát azt nem kell elkészítenünk. A csomag ára 2 millió forint. A kalkuláció ebben a szituációban akkor 70-nek a 20 százaléka 14, tehát marad nekünk 70 – 14 = 56 ezersoros egység a fejlesztésre. Ebből $PM=2{,}4 \cdot 56^{1{,}05}= 164{,}36$ emberhónap, ami körülbelül 13,7 emberév. Forintosítva a programozók költsége $164{,}36 \cdot 0{,}5$MFt=82,18 millió forint. Hozzávéve a procedúra csomag vételárát 84,18 millió forint adódik, ami lényegesen alacsonyabb, mint a saját fejlesztés esetében. A vállalási idő pedig $T_d = 2{,}5 \cdot \sqrt[3]{164{,}36} = 13{,}7$ hónap, ami egy kicsivel kevesebb, mint az előző esetben volt. A munkára ráállítandó fők száma: $164{,}36/13{,}7 \cong 12$ fő. A rizikós határidő Brook limitje $13{,}7 \cdot 0{,}75 = 10{,}275$ hónap.

A példa azt sugallja, hogy érdemes kipróbált, dokumentált, garanciás piaci csomagokat alkalmazni. Persze ezt minden esetben gondosan elemezni kell. Mindenesetre a csomagok alkalmazásának további előnyei is lehetnek. Ezek például:

- Könnyebb lesz a szoftverünk tesztelése, mert a csomag már tesztelt
- A fejlesztés felgyorsul
- A csomag máskor is rendelkezésre áll
- A fejlesztés magasabb szinten történhet, mert a csomag erre adhat kereteket.

Programtesztelés

A programtesztelésre szükség van. Szükség van, mert a hibalehetőségek számtalan esete fordulhat elő. Egy nem nagyon elemi és nem nagyon kicsiny méretű programban majdnem törvényszerű a hiba megjelenése. Nem a hiba okozza a bajt, hanem a hiba kijavításától való elfordulás. Tévednünk szabad, kijavítani kötelesség. Nézzük a teljesség igénye nélkül, hogy milyen jellemző hibák fordulnak elő a programfejlesztés során a kezdeti algoritmizáláson túlmenően.

- *Szintaktikai hiba*. (Syntax error.) Ez a programozási nyelv nyelvtanával szemben elkövetett hiba, amit általában a fordítóprogram kijelez, miáltal általában könnyen javítható.
- *Szemantikai hiba*. Ez jelentéstani hiba. Az utasításokat nem a jelentésüknek megfelelően használtuk. Nem biztos, hogy jelzést kapunk róla, javítása nem mindig egyszerű.
- *Végrehajtási hiba*. (Run-Time error.) Az operációs rendszer a program végrehajtása közben hibát észlelt és a programot leállította. Oka lehet például zérussal osztás, vagy lehet tiltott memóriaterületre belépés kísérlete, stb.
- *Nem áll le a program*. Nem könnyű felderíteni a hibát. Gyakran végtelen ciklusról van szó.
- *Nem ír ki semmit, pedig kellene*. Nyilvánvalóan nem jut el a program a kiíratás helyéig, vagy más ágon fut le.
- *Nem azt írja ki, amit vártunk*. Valószínűleg nem jól számol, vagy nem azon az ágon fut, amit terveztünk. Biztos, hogy jó a program inputja?
- *Értelmezhetetlen hiba, hogy jön ez a hiba ide*? Lehet, hogy a hiba nem az adott helyen van, hanem valahol korábban, de most itt vettük észre, itt jelentkezett a hatása. Ez a hibaöröklődés, hibaterjedés jelensége. Nehéz az okot megtalálni.

A hibafeltárás, hibajavítás egyszerűbb eszközei, módjai általában:

- *Kiíratási utasítások elhelyezése a forráskódban* elsősorban a hiba gyanított helyének közelében. Mennyiségét és sűrűségét a szituáció szabja meg, lehet, hogy többszöri nekifutásra és módosításra van szükség. Ilyenkor a számítások részeredményeit és útvonalát igyekszünk követni, hogy az megfelel-e az eredeti tervünknek.
- *Nyomkövetés*. (Trace.) A nyomkövetés azt jelenti, hogy a programot utasításonként hajtjuk végre (utasítás nyomkövetés), minden utasítás után megállva, figyelve, hogy milyen a végrehajtási sorrend. Ahol az a várttól eltér, nagy eséllyel ott van a hiba. Adatnyomkövetés esetén egy, vagy néhány adat változását is figyeljük az utasítások során. A nyomkövetés lehet nagyon részletes, amikor minden utasítás után megállunk, de lehet olyan is, hogy mondjuk a procedúrákba, ciklusokba csak a belépést és a kilépést jelezzük.
- *Pillanatfelvétel*. (Snapshot.) A pillanatfelvétel a memória egy darabjának (kritikus esetben egészének) a megjelenítését és tárolását, elmentését (dump) jelenti elemzés céljából a kívánt időpontban.
- *Töréspontok elhelyezése*. (Breakpoint.) Ezek speciális jelek a fordítóprogram számára, amelyek a jelzett helyeken megállítják a programot és lehetővé teszik a helyzet elemzését, akár adatok megváltoztatását is.
- *Befordított ellenőrzések*. A programba a kiíratásokon kívül gyakran a fordítóprogramnak fordítási opciókat adhatunk meg, amelyek beépülnek a programba és programfutás közbeni ellenőrzéseket, figyeléseket végeznek. Hiba esetén hibajelzéssel leállítják a programot. Általában a program futásának az idejét jelentősen megnövelik, ezért csak a fejlesztés idején használjuk. Ha a program már jó, akkor

ellenőrzési opciók nélkül újra fordítjuk azt. Ilyen tipikus ellenőrzések szoktak lenni az alábbiak:

- *indexhatár figyelés*. Azt figyeltetjük, hogy tömbök esetén nem lépünk-e ki azokból.
- *túlcsordulás, alulcsordulás*. Az érvényes számtartományból történő kilépések.
- *típusellenőrzés*. Az adott helyen megfelelő-e a használt típus, pl. paraméterlistán.
- *típuskeveredés*. Típusok keveredésekor a műveletekben a konverzió jó-e.
- *értéket nem kapott változó*. Formulában értéket nem kapott változó okozhat hibát.
- *nem használt változó*. Valóban nem használjuk, vagy csak elírtuk a nevét?
- *mellékhatások figyelése*. (Side effekt.) Rafinált szituációk, amelyek mellékhatással járnak, mint például egy figyelmetlenül megírt függvény esetén előfordulhat az a furcsa jelenség, hogy $f(x)+f(x)\neq 2f(x)$.

A programtesztelés a program minőségbiztosításának a része. A tesztek kidolgozását egy részletes *tesztelési terv* kell, hogy megelőzze. A tesztelési terv egy dokumentum, amely felsorolja a tesztelési eseteket az inputokkal és a hozzájuk tartozó, helyesnek vélt, ismert outputokkal együtt. Megadja azokat a körülményeket, amelyek között megállapítható, hogy a teszt sikeres volt-e vagy sem (*elfogadási kritérium*). A tesztelési tervnek analizálhatónak kell lennie a teljességre és arra, hogy biztosítja-e a rendszer céljainak az elérését.

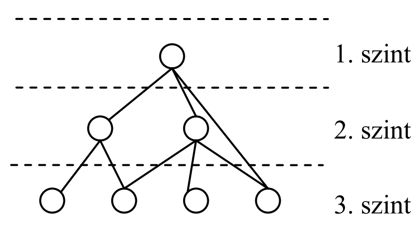
A tesztek programfuttatásokat jelentenek, amelyek esetén különböző, általában ismert eredményű input adatokkal vizsgáljuk a programot. Igyekezni kell lehetőleg minden programágat legalább egyszer kipróbálni. Ez azt jelent, hogy *minimálisan annyiféle tesztadatsort kell összeállítani, amennyi a program ciklikus bonyolultsága*. A tesztelésnek több fázisa van:

- tesztelés a program készítése közben
- tesztelés a program elkészülése után
- tesztelés a programnak a felhasználó, megrendelő számára történő átadás után
- tesztelés a program üzemelése során.

A tesztek futtatása gyakran automatizálható és az eredményeket célszerű újrafeldolgozásra eltenni, tárolni lehetőleg háttértárakon.

Az alulról-felfelé teszt (Bottom-Up test)

Megrajzoljuk a procedúrák hívási hálóját, amelyben kialakulnak az egyes hívási szintek, amint azt az alábbi ábrán láthatjuk, ahol a körök az egyes procedúrákat jelentik, az összekötő vonalak pedig a procedúrát a hívott procedúrával köti össze. A hívott van mélyebb szinten elhelyezve. A hívások feltérképezése után derül csak ki, hogy melyik procedúra melyik szintre kerül.



A tesztelési elv: amennyire lehetséges egy adott X procedúra tesztelése előtt az összes, az X által hívott procedúrát tesztelni kell, valamint azokat is, amelyek neki adatot készítenek elő. Tehát ilyen módon először a legalsó szinten lévő, egymástól nem függő procedúrákat teszteljük. Ez történhet akár egymással párhuzamosan is. Ezt követően lépünk egy szinttel feljebb abban a tudatban, hogy az onnan meghívott procedúrák már remény szerint helyesek.

Procedúra tesztelés. (Unit test.)
Ebben a tesztben a procedúra minden lehetséges viselkedését tesztelni kell. Ilyenek lehetnek

- lehetséges lefutási utak, elágazások
- határesetek és szokatlan esetek pl.
    - üres táblázat esete
    - megtelt táblázat esete
    - ciklusmag végrehatásainak száma zérus
    - rendezéskor az adatok már rendezettek stb.

- a procedúra interface (paraméterlista) átadása naiv felhasználóknak, azaz ne a fejlesztő teszteljen, mert ő a feladat hatása alatt van
- adatsruktúrák helyes inicializálási tesztje, pl. kezdő nullázások.
- adatstruktúrák megfelelő állapotának ellenőrzése pl. indexhatárok megfelelőek-e, mutatók be vannak-e állítva, stb.

Ilyenkor a használatra kerülő eszközök gyakran formázott hibakereső kiíratások (debug) beépítésében nyilvánulnak meg. Kiíratjuk az adatsruktúrát a procedúra hívása előtt, meghívjuk a procedúrát és kiíratjuk az adatstruktúrát a hívás után. Másik lehetőség a *tesztgenerátor* (test driver) használata. Ez olyan procedúra, amely generálja, felsorolja a teszteseteket és mindegyikre meghívja a tesztelendő procedúrát, ellenőrizve, hogy az eredmények megfelelőek-e.

Tesztelés egészben. (Integral test)
Az alulról-felfelé tesztet követően alkalmazzuk. Ellenőrzi, hogy a procedúrák, modulok összességükben együtt helyesen működnek-e. Az, hogy a procedúrák külön-külön jók, nem jelenti, hogy együtt is jók. Az egyik procedúra előállíthat egy másik számára olyan inputot, amelyre az nem volt felkészítve. Az egészben történő tesztelés szerepe a részek közötti együttműködés ellenőrzése, és hogy a részek valóban sikeresen kooperálnak a rendszer helyes céljainak az elérésére.

Átvételi teszt.
A rendszer aktív használatba vétele előtt fut le. Általában a megrendelő és a fejlesztő közötti formális szerződésben rögzítik, hogy a rendszer csak ez után a teszt után szállítható használatra. A szerződés előírhatja a teszt milyenségét is. Gyakran a tesztelést egy - a szerződésben kijelölt - független szervezet végzi.

Regressziós teszt. (Visszamenőleges teszt.)
Az éles üzemelés során hibák derülhetnek ki. Ezeket rögzítik, leírják, pontosan behatárolják. Felmerülhet új szolgáltatás beépítésének az igénye (Upgrade) is, például ha az üzemelés körülményei megváltoztak. A hiba javítása után tesztelendő, hogy

- tényleg nincs ott a hiba
- ha volt upgrade, akkor az működik is
- az is jól működik továbbra is, ami eddig jól működött.

A rendszer tesztcsomagjához az új teszteseteket hozzá kell fűzni.

Felülről-lefelé tesztek. (Top-Down test.)
A fejlesztés során gyakran még nem állnak rendelkezésre a finomított lépések procedúrái, de a magasabb szintűeket ellenőrizni kell. (A rendszer részeit párhuzamosan fejleszti a csoport, ez gyorsítja az implementációt.) Ilyenkor leegyszerűsített helyettesítő procedúrát (dummy procedure) készítünk a fejlesztés alatt levő helyébe, amely imitálja a tesztesetekre a procedúra outputját. Tulajdonképpen egy táblázatot tartalmaz, amelyben benne vannak a teszt inputjau

és a rájuk adandó válaszok. A procedúra megnézi, hogy melyik input jött be, és visszaadja a táblázatbeli választ. Használható akkor is, ha az egyes részek kölcsönösen hívják egymást és mindegyiket külön-külön tesztelni akarjuk.

A tesztelés kideríthet hibákat, de sohasem bizonyítja azok nem létét!

A program tesztelése mellett a hatékonyság sem elhanyagolandó szempont. A tapasztalatok azt mutatják, hogy nagy általánosságban a programkód 20%-a végzi a munka 80%-át. Ezért nem rossz ötlet kimérni az egyes összetevők munkamennyiségét, jellemzően a futási idejét. Egyszerűbb esetben ez történhet a programba beépített „stopper”-rel (óra lekérdezés előtte és utána). Modernebb változata az úgynevezett *execution-time profiler* szoftver használata, amely egy méréseket biztosító futtató környezet. Hangolásnak (tuning) nevezik, amikor a kevésbé hatékonynak bizonyuló részeket gondosabban átírják, hatékonyabbakra cserélik.

Programhelyesség bizonyítás

A programhelyesség bizonyítás a formális logika segítségével és módszereivel igyekszik belátni, hogy a program valóban azt végzi, amit tőle elvárnak. Előfordulási helyei:

- az algoritmus készítésének a kezdetén
- az algoritmus hibakeresése idején
- az algoritmus hatékonyságának a javítása során
- a procedúrák, mint egy nagyobb program részeinek helyességellenőrzésekor.

Hogyan fejezzük ki, hogy a *P* programnak mit kell tennie? Egy lehetséges gyümölcsöző út állítások megadása, amelyek leírják, hogy a *P* program futása előtt milyen és a *P* program futása után milyen állapotoknak kell fennállni. Az állítások lehetnek igazak, vagy hamisak és precíz logikai nyelven kerülnek leírásra.

Definíció: Az előfeltétel
Az előfeltétel leírja azokat a feltételeket, amelyek igazak a *P* program végrehajtása előtt. Legyen x a *P* program inputja. Az előfeltétel egy logikai függvény $\varphi(x)$, amely igaz kell legyen.

Definíció: Az utófeltétel
Az utófeltétel leírja azokat a feltételeket, amelyek igazak a *P* program végrehajtása után, feltéve, hogy az előfeltételek igazak voltak. Ha x az input és z az output ($z=P(x)$), akkor az utófeltétel egy $\psi(x, z)$ logikai függvény, amely igaz, ha $\varphi(x)$ igaz volt.

A formális felírás

$$\{\text{előfeltétel}\}$$
$$P$$
$$\{\text{utófeltétel}\}$$

Szemléltető példa.

Legyen a feladat az *A* valós számok tömbjében lévő számok csökkenő sorrendbe rendezése. Erre a feladatra készítettünk egy procedúra csomagot, amely az alábbi.

```
Procedúra MaxPozKeres(@A,m,n,@MaxPoz)
Input paraméterek: A∈R^n
                   m, n∈N, 1≤m≤n
Output paraméter:  MaxPoz∈ N
// Az A tömb a_m…n résztömbjében megkeresi a
   legnagyobb elemnek az indexét, ami MaxPoz.

i ← m
j ← m
REPEAT     INC( i )
           IF       a_i > a_j
                    THEN     j ← i
           UNTIL    i = n
MaxPoz ← j
RETURN( MaxPoz )
```

```
Procedúra Rendezés (@A, m, n)
Input paraméterek: A∈R^n
                   m, n∈N, 1 ≤ m ≤ n
Output paraméter:  A∈R^n
// Az A tömb a_m…n résztömbjében rendezi csökkenő
   sorrendbe az elemeket.

IF   m < n
     THEN   CALL MaxPozKeres(@A, m, n, @MaxPoz)
            a_m és a_MaxPoz felcserélése
            CALL Rendezés (@A, m+1, n)
RETURN(A)
```

A Rendezés procedúra esetén a helyességvizsgálat formálisan felírható az alábbi formában.

$\{ m \le n \}$	// legalább egy elemet kell rendezni állítás
Rendezés(@A, 1, n)	// a rendezés végrehajtása
$\{a_m \ge a_{m+1} \ge \ldots \ge a_n\}$	// állítja, hogy csökkenő a sorrend

Azt, hogy a Rendezés(@A, m, n) csökkenő sorrendbe rendez, azzal bizonyítjuk, hogy ha az előfeltétel igaz, és a Rendezés(@A, m, n) procedúrát végrehajtjuk, akkor igaz lesz az utófeltétel is, azaz $\{m \le n\}$ igaz esetén a **CALL** Rendezés(@A, 1, n) után igaz az $\{a_m \ge a_{m+1} \ge \ldots \ge a_n\}$ állítás. A bizonyítás céljából a következőket tesszük.

- Különböző feltételeket helyezünk el a P programban, melyek közbülső feltételeket írnak le.
- A P által használt procedúrákra is elvégezzük a helyességvizsgálatot, melynek eredményét P-nél felhasználjuk.

Először lássuk MaxPozKeres helyességbizonyítását! A formális felírás:

$$\{ m < n \}$$
MaxPozKeres(@A, m, n, @$MaxPoz$)
$$\{a_{MaxPoz} \ge a_{m\ldots n}\}$$

A MaxPozKeres procedúra kiegészítve további belső feltételekkel:

```
Procedúra MaxPozKeres(@A, m, n, @MaxPoz)
Input paraméterek: A∈R^n
                   m, n∈N, 1 ≤ m ≤ n
Output paraméter:  MaxPoz∈ N
// Az A tömb a_m…n résztömbjében megkeresi a
   legnagyobb elemnek az indexét, ami MaxPoz.

i ← m
j ← m
              {(i = m) ∧ (j = m) ∧ (m < n)}
REPEAT        INC( i )
              IF      a_i > a_j
                      THEN   j ← i
                      {( a_j ≥ a_m…i) ∧ (i ≤ n)}
              UNTIL   i = n
                         {( a_j ≥ a_m…n)}
MaxPoz ← j
RETURN( MaxPoz )
```

Logikai következtetések és programutasítások értelmezése által el kell jutnunk az előfeltételtől az utófeltételig.

Feltételezzük, hogy az előfeltétel igaz a procedúra indulásakor, azaz $m < n$. Az első két utasítás $i \leftarrow m$, $j \leftarrow m$ után mindkét lokális változó i és j értéke m, tehát igaz az $\{(i = m) \wedge (j = m) \wedge (m < n)\}$ feltétel. Következik a REPEAT…UNTIL ciklus, amelyben egyetlen feltételt helyeztünk el, ami most a *ciklusinvariáns*.

Definíció: A ciklusinvariáns
A *ciklusinvariáns* olyan állítás, amely mindig igaz, függetlenül attól, hogy a ciklusmag hányszor hajtódott végre.

A ciklusinvariancia bizonyítása emlékeztet a matematikai indukcióra. Belátjuk, hogy a ciklusinvariáns igaz az első menetben. Ezt követően belátjuk, hogy ha igaz az i. menetben, akkor igaznak kell lennie az i+1. menetben is.

Mivel a ciklusba belépéskor $\{(i = m) \wedge (j = m) \wedge (m < n)\}$ igaz, ezért $i = m$. INC(i) után $i = m+1$ lesz. Az IF utasítás előtt ezért $\{(i = m+1) \wedge (j = m)\}$ igaz. Ezért az IF után, ha $a_{m+1} > a_m$, akkor $j = m+1$ lesz, egyébként (ha $a_{m+1} \leq a_m$) pedig marad $j = m$. Mindenképpen ezután a j értéke az a_m és a_{m+1} közül a nagyobbik indexével lesz egyenlő, azaz $a_j \geq a_{m\ldots m+1}$ igaz lesz. Mivel $i = m+1$, ezért az állítás írható így is: $a_j \geq a_{m\ldots i}$, amely már a ciklusinvariáns első része. A ciklusinvariáns második része $(i \leq n)$. Ez azért igaz, mert bármely három szám: i, m, n esetén, ha $(m < n)$ és $(i = m+1)$, akkor $(i \leq n)$ igaz. Ugyanis $m < n$ azt jelenti, hogy az m legalább eggyel kevesebb, mint az n. Ezért az eggyel történő növelése nem adhat az n-nél nagyobb számot.

Most következik az indukciós lépés. Feltételezzük, hogy az i. menetben a ciklusinvariáns igaz. Mi történik az m+1. menetben? Az UNTIL $i = n$ ellenőrzést közvetlenül megelőzően igaz az $(a_j \geq a_{m\ldots i}) \wedge (i \leq n)$ állítás. Vizsgáljuk most azt az esetet, amikor az UNTIL $i = n$ utasításnál az $i \neq n$ áll fenn. Akkor ez azt jelenti, hogy

$(i \neq n)$ // az UNTIL-ból, amely hamis
$(a_j \geq a_{m\ldots i}) \wedge (i \leq n)$ // a ciklusinvariánsból, amely igaz

Ebből következik, hogy $(a_j \geq a_{m\ldots i}) \wedge (i < n)$ igaz a ciklusmag i+1. végrehajtásának kezdetén. Az i+1. menetben INC(i) után $(a_j \geq a_{m\ldots(i-1)}) \wedge (i \leq n)$ lesz igaz. Az IF utasítás ezután összehasonlítja az $a_{m\ldots(i-1)}$ elemek legnagyobbikát az a_i elemmel. Az IF-et követően j mindenképpen az $a_{m\ldots i}$ legnagyobbikának indexével lesz azonos. (Vagy marad a régi, ha a_i nem nagyobb, vagy i-re cserélődik, ha a_i nagyobb.) Tehát $(a_i \geq a_{m\ldots i})$ igaz $(i \leq n)$-et, kapjuk az igaznak bizonyuló ciklusinvariánst. Tehát a ciklusinvariáns igaz a ciklusmag minden lefutásakor.

Mi történik a ciklusból való kilépés után? A ciklusinvariáns igaz és a kilépéskor az UNTIL $i = n$ is igaz. Ezért igaz, hogy
$(i = n)$
$(a_j \geq a_{m\ldots i}) \wedge (i \leq n)$.
Ebből következik, hogy $(a_j \geq a_{m\ldots i}) \wedge (i = n)$, amiből írható, hogy $(a_j \geq a_{m\ldots n})$, mert $i = n$. Ez utóbbi viszont pontosan az utófeltétel, ami így igaz lesz. Ha ezután *MaxPoz* = j lesz, akkor *MaxPoz* az $a_{m\ldots n}$ elemek közül a legnagyobbnak az indexével lesz azonos.

Következzen most a Rendezés vizsgálata.

```
Procedúra Rendezés (@A, m, n)
Input paraméterek: A∈R^n
                   m, n∈N, 1 ≤ m ≤ n
Output paraméter:  A∈R^n
// Az A tömb a_m…n résztömbjében rendezi csökkenő
   sorrendbe az elemeket a saját helyükön.
---------------------------------------------------
IF   m < n
     THEN   CALL MaxPozKeres(@A, m, n, MaxPoz)
            {a_MaxPoz ≥ a_m…n}
            a_m és a_MaxPoz felcserélése
            {a_m ≥ a_m…n}
            CALL Rendezés (@A, m+1, n)
            {a_m ≥ a_m+1 ≥…≥ a_n}
{a_m ≥ a_m+1 ≥…≥ a_n}
RETURN( A )
```

A Rendezés előfeltétele: $m < n$., feltesszük, hogy igaz. Az IF utasításban ezért meghívódik a MaxPozKeres procedúra, melynek előfeltétele ($m < n$) teljesül. Beláttuk, hogy MaxPoz ezután az $a_{m\ldots n}$ legnagyobbikának az indexe lesz, azaz $\{a_{MaxPoz} \geq a_{m\ldots n}\}$ igaz lesz. A csere után pedig az $\{a_m \geq a_{m\ldots n}\}$ lesz igaz. Ezt a lépést követi a hátralévő $a_{m+1\ldots n}$ elem rekurzív rendezése csökkenő sorrendbe, amely eredményezi az $a_{m+1} \geq a_{m+2} \geq\ldots\geq a_n$ állítás igaz mivoltát, mivel a rekurzív híváskor a_m a helyén maradt és így $a_m \geq a_{m+1}$ igaz maradt a rekurzív hívásból való visszatérés után is. Azt kapjuk, hogy

$(a_m \geq a_{m+1})$
$(a_{m+1} \geq a_{m+2} \geq\ldots\geq a_n)$,

ahonnan $(a_m \geq a_{m+1} \geq a_{m+2} \geq\ldots\geq a_n)$ következik, de ez a végső állítás. Maradt még az IF feltételének hamis esete. Ha $m < n$ hamis, akkor az $m \leq n$ igaz miatt $m = n$ teljesül. Ebben az esetben azonban az $a_{m\ldots n}$ résztömb az $a_{m\ldots m}$ egyelemű résztömbbé degenerálódik, amely már rendezettnek tekinthető. A következtetés második részében *rekurzív indukció*t használtunk.

Definíció: A rekurzív indukció
A rekurzív indukció esetén a problémát kisebb méretű problémára vezetjük vissza úgy, hogy az előfeltétel teljesüljön, bizonyítjuk ezen visszavezetés helyességét és bizonyítjuk, hogy a rekurzió alapesetére (amit már nem vezetünk vissza kisebbre) szintén teljesül az utófeltétel. Bizonyítandó továbbá, hogy az alapeset véges sok lépés után elérődik, ami garantálja, hogy a rekurzió véget ér.

A rendezésben a rekurzió eljut az alapesetig, mivel a méret minden rekurzív hívásnál eggyel csökken.

Rámutatunk a programhelyesség bizonyítás egy gyengéjére a tárgyalt példán keresztül. Legyen egy csaló programozó, aki a következő rendezési procedúrát készíti el.

```
Procedúra Rendezés (@A, m, n)
FOR     i ← m TO n DO
        a_i ← a_m
RETURN( A )
```

Ennél a programnál, ha az $\{m \leq n\}$ előfeltétel igaz, akkor az $\{a_m \geq a_{m+1} \geq \ldots \geq a_n\}$ utófeltétel is igaz lesz. Ezért a programot helyesnek kell elfogadnunk, amikor pedig az nyilvánvalóan nem helyes. A probléma az, hogy amikor az utófeltételt megfogalmaztuk, az nem bizonyult *teljes*nek, mivel nincs benne, hogy csak az elemek sorrendjében legyen változás, maguk az egyes adatelemek ne változzanak meg. Törlés, duplikálás, új elem ne legyen.

Honnan lehetünk biztosak, hogy a formális logikai állítások az elő- és utófeltételekben pontosan azt fejezik ki, amit a programnak tudnia kell? Sehonnan, bízzunk a helyes intuíciónkban. Ezért is használunk intenzív tesztelést! Minden eszköz csak a bizonyosságunk szintjét emeli, teljes biztonságot nem ad.

Programdokumentáció

A feladat, a probléma megoldására elkészült programot dokumentálni kell. A dokumentációnak különböző szintjei vannak. Különböző területek, célcsoportok számára készül dokumentáció.

- *Felhasználói kézikönyv* (User's Manual). Gyakran van benne a szoftver lehetőségeit bemutató interaktív, animált program, amely lépésről-lépésre végigvezet a témán, tanítva az új felhasználót a szoftver alkalmazásásra.
- *Dokumentáció a projektvezető számára.* Jó, világos, magasszintű leírás mélyebb részek nélkül, hogy a vezető átlássa a végzett munkát.
- *Fejlesztői dokumentáció szakembereknek.* Részletes, sokszor a nyilvánvaló dolgok leírását is tartalmazó dokumentáció. Egy új, a témát még nem ismerő fejlesztő is úgy megérthesse, hogy akár át is veheti a munka folytatását.

A papír alapú dokumentációtól jobb az elektronikus szövegszerkesztési dokumentáció, mivel rugalmas, tág lehetőségekkel rendelkezik:

- navigációs gombok a szöveg más helyeire, fogalmak magyarázataira
- szöveg mutatása, vagy elrejtése, magyarázatok, diagramok az olvasó szintjétől függően
- hang lejátszása, beszédes magyarázatok
- animáció egy adott pont viselkedésére
- videofilm, klip

Úgy kell dokumentálni, ahogy azt másoktól elvárnánk. Jó stratégia a felülről-lefelé fejlesztés fokozatos finomítással. Ahogyan egy adott szinten dolgozunk, a program írásakor hozzáfűzzük a dokumentáló megjegyzéseinket. A finomításkor a megjegyzések is finomodnak, szinte automatikusan készül a dokumentáció a programmal párhuzamosan. Nagyon megtérülő tevékenység a kész, vagy majdnem kész dokumentáció átadása megértés végett olyan embereknek, csoportnak, akik nem vesznek részt a fejlesztésben, de az alkalmazásban már részt vesznek, vagy részt vehetnének. Az ő kérdéseik, megértési nehézségeik sokat segítenek azon, hogy a dokumentáció problémáit feltárjuk és orvosolni tudjuk.

Két fontos momentumot külön ki kell emelni a dokumentációval kapcsolatban.

- Az adatstruktúrákat is és a programmodulok interface-ét is dokumentálni kell. Pontosságban, részletezettségben nem maradhatnak el a procedúrák mögött.
- Minden procedúra elejére egy inicializáló megjegyzést kell tenni, amely a várt Input/Output paraméter jellemzőket is megadja. Formálisan ezek lesznek az előfeltételek és az utófeltételek.

A fejlesztő csapat számára jó, ha egységes munkastílust alakítanak ki minden téren.