



ELTE | IK

PROGRAMOZÁS

Tesztelés, hibakeresés, dokumentálás

Horváth Győző, Szlávi Péter

Tartalom



Tartalom

- Tesztelés

Helyes-e a program?

Igen

Nem

- Hibakeresés

Hol és mi a hiba?

- Hibajavítás

Hogyan javítható a hiba?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Tartalom

- Tesztelés

Helyes-e a program?

- Hibakeresés

Hol és mi a hiba?

- Hibajavítás

Hogyan javítható a hiba?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Tartalom

- Tesztelés

Helyes-e a program?

- Hibakeresés

Hol és mi a hiba?

- Hibajavítás

Hogyan javítható a hiba?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Algoritmizálás,
kódolás



Tartalom

- Tesztelés

Hatékony-e a program?

Igen

Nem

- Hibakeresés

Hol és mi a probléma?

- Hibajavítás

Hogyan javítható a probléma?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Tartalom

- Tesztelés

Hatékony-e a program?

- Hibakeresés

Hol és mi a probléma?

- Hibajavítás

Hogyan javítható a probléma?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Tartalom

- Tesztelés

Hatékony-e a program?

- Hibakeresés

Hol és mi a probléma?

- Hibajavítás

Hogyan javítható a probléma?

Algoritmizálás,
kódolás

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?



Tartalom

- Tesztelés

Helyes-e / jó-e a program?

- Hibakeresés

Hol és mi a probléma?

- Hibajavítás

Hogyan javítható a probléma?

- Dokumentálás

Hogyan használható, hogyan karbantartható a program?

Tesztelés



Célja és alapfogalmai

Cél: a hibás/nem hatékony működés kimutatása

Eredménye: hibajelenség vagy annak „hiánya”

Alapfogalmak:

- Teszteset = **bemenet** + **kimenet**
- Próba = teszteset-halmaz
- Jó teszteset: nagy valószínűséggel felfedetlen hibát mutat ki
- Ideális próba: minden hibát kimutat
- Megbízható próba: nagy valószínűséggel minden hibát kimutat

A „hiba” helyett jobb lenne „problémát” mondani, mivel a tesztelés nemcsak a hibakeresés, hanem a hatékonyságvizsgálat eszköze is.

Tesztelési elvek

- Érvényes (megengedett) és érvénytelen (hibás) bemenetre is kell tesztelni.
- Minden tesztet által nyújtott információt maximálisan ki kell használni (a következő tesztesetek kiválasztásánál).
- Helyesség ellenőrzésére minimális méretű (fejben könnyen ellenőrizhető) tesztbemeneteket definiáljunk.
- Csak más (mint a szerző) tudja jól tesztelni a programot.
- A hibák nagy része a kód kis részében van.
- Rossz a meg nem ismételhető tesztet.

Ez nem a tesztelés, hanem a tesztelendő program vonása. Ilyen programok: a véletlenszámokat (l. később) használó vagy folyamatvezérlő programok.

Az elv megvalósítása: a tesztelés idejére garantálni kell (ha lehet), hogy csak a bemenettől függjön a futás. Ilyenkor speciális beállítások (l. véletlenszámoknál), vagy a folyamatkörnyezet szoftveres szimulációja kell a tesztelés-hez.

Tesztelés

Módszercsaládok:

- Statikus tesztelés:
a programszöveget vizsgáljuk, a program futtatása nélkül.
- Dinamikus tesztelés:
a programot futtatjuk különböző bemenetekkel és a kapott eredményeket vizsgáljuk.

A tesztelés alapkérdése:

- Meddig kell tesztelni?

Statikus tesztelések

Kódelőellenőrzés:

- Algoritmus ↔ kód megfeleltetés
 - kódolási hibák kimutatására
- Algoritmus + kód elmagyarázása másnak
 - algoritmikus + kódolás hibák kimutatására

Szintaktikus ellenőrzés:

- fordítóprogram esetén automatikus
 - bár nem mindig kellően szigorú
- értelmező esetén sok futtatással jár
 - valójában dinamikus tesztelés (l. később)

Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- felhasználatlan változó/érték

```
i=1;  
for (i=2; ...) {  
    ...  
}
```

- „identikus” transzformáció

```
i=1*i+0; //n1? K0 vagy 0?
```

- Inicializálatlan változó

```
static int n;           //meggondolatlan kódolása  
int[] k=new int[n]; //a specifikációbeli adatléírásnak
```

Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- definiálatlan (?) értékű kifejezés

```
static int n; ← globális n
```

```
...
```

```
static int fv()
```

```
{
```

```
    for (int n=0; n<9; ++n) { ← lokális n
```

```
        ... n ... ; ← a lokális n felhasználása
```

```
    }
```

```
    return ...n...; ← a globális n-et tartalmazó formula
```

```
}
```


Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- azonosan **igaz/hamis** feltétel

$(N > 1 \ || \ N < \text{MaxN}) \ / \ (N < 1 \ \&\& \ N > \text{MaxN})$

Gyakori hiba a **beolvasás-ellenőrzés** kódolásakor.

- végtelen számlálás ciklus

```
for (int i=0; i<N; ++i) {
```

...

~~--i;~~ ← *nem vált ki fordítási hibaüzenetet*

```
}
```

Talán egy **while**-os ciklus átírása során maradhatott benn.

$N \geq 1 \ \&\& \ N \leq \text{MaxN}$
Téves „fejben” negálásai

Vannak olyan programozási
nyelvek, amelyek fordítási
szinten utasítják vissza a
ciklusváltozó ciklusmag-
beli, direkt módosítását.

Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- pontatlan ciklus-szervezés

```
for (int i=0; i<N; ++i) {  
    ...  
    ++i;   
}
```

Vannak olyan programozási nyelvek, amelyek fordítási szinten utasítják vissza a ciklusváltozó ciklusmag-beli, direkt módosítását.

Talán egy **while**-os ciklus átírása során maradhatott benn.

- konstans értékű, (bár) változókat tartalmazó kifejezés

```
y=sin(x)*cos(x)-sin(2*x)/2
```

Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- végtelen feltételes ciklus ($i < N$ feltételű ciklusban sem i , sem N nem változik, vagy „szinkronban” változik)

```
i=1;  
while (i<=N) {  
    ...  
    i=+1; ← talán i+=1 akart lenni?  
}
```

Statikus tesztelés * kódellenőrzés

Szemantikus ellenőrzés, ellentmondás-keresés:

- végtelen feltételes ciklus ($i < N$ feltételű ciklusban sem i , sem N nem változik, vagy „szinkronban” változik)

```
i=1;  
while (i<=N) {  
    ...  
    i=i++; ← i_masolat=i; i=i+1; i=i_masolat vagyis i=i  
}
```

Statikus tesztelés * kódellenőrzés

Nem szemantikusan hibás, de **kódolási vétség**:

- A ciklusfajta **rossz** választása (**eldöntés tétel**):

```
Van=false; általánosított összegzés tétel  
for (int i=1; i<=N; ++i) {  
    Van=Van || T(...i...);  
}
```

Mi a baj vele? **Eltérés** a programozási tételtől + **rossz hatékonyság**.

- A **jó** választás, azt teszi és olyan hatékonyan, amit és ahogyan kell:

```
int i=1;  
while (i<=N && !T(...i...)) {  
    ++i;  
}  
Van=i<=N;  $\leftarrow$  van-e T-tulajdonságú?
```

Statikus tesztelés * kódellenőrzés

Nem szemantikusan hibás, de **kódolási vétség**:

- A ciklusfajta **rossz** választása, majd annak **megerőszakolása** (kiválasztás tétel):

```
for (int i=1; i<=N; ++i) {  
    if (T(...i...)) {  
        sorsz=i; i=N; ← a kilépés kieroszakolása  
    }  
}
```

- ... a **jó** választás (erőszakmentesen) ugyanazt teszi:

```
i=1;  
while (!T(...i...)) {  
    ++i;  
}  
sorsz=i;
```

Statikus tesztelés * kódellenőrzés

Nem szemantikusan hibás, de **kódolási vétség**:

- A ciklusfajta **rossz** választása, majd annak **megerőszakolása** (kiválasztás tétel):

```
for (int i=1; i<=N; ++i) {  
    if (T(...i...)) {  
        sorsz=i; break; ← a kilépés kierőszakolása  
    }  
}
```

- ... a **jó** választás (erőszakmentesen) ugyanazt teszi:

```
i=1;  
while (!T(...i...)) {  
    ++i;  
}  
sorsz=i;
```

Statikus tesztelés * kódellenőrzés

Nem szemantikusan hibás, de **kódolási vétség**:

- A ciklusfajta **rossz** választása, majd annak **megerőszakolása** (kiválasztás tétel):

```
for (int i=1; i<=N; ++i) {  
    if (T(...i...)) {  
        sorsz=i; break; ← a kilépés kierőszakolása  
    }  
}
```

- ... a **jó** választás (erőszakmentesen) ugyanazt még „**olcsóbban**” teszi:

```
sorsz=1;  
while (!T(...sorsz...))  
{  
    ++sorsz;  
}
```


Dinamikus tesztelések

Tesztelési módszerek:

- **Fekete doboz** módszerek (←*nincs kimerítő bemenet* – nem lehet minden lehetséges bemenetre kipróbálni): a teszteseteket a program **specifikációja** alapján, **optimálisan** választjuk.
- **Fehér doboz** módszerek (←*nincs kimerítő út* – nem lehet minden végrehajtási sorrendre kipróbálni): a teszteseteket a **program struktúrája** alapján, **optimálisan** választjuk.
- **Szürke doboz** módszerek – a konkrét algoritmust nem ismerjük, de a típusát igen, a tesztelést erre alapozzuk.

Dinamikus tesztelés * fekete doboz módszerek

- **Ekvivalencia-osztályok módszere:** a bemeneteket (vagy a kimeneteket) soroljuk olyan diszjunkt osztályokba, amelyekre a program várhatóan egyformán működik; ezután osztályonként egy tesztesetet válasszunk!
- **Határeset elemzés módszere:** az ekvivalencia-osztályok határáról válasszunk tesztesetet (be- és kimeneti osztályokra is)!

Ekvivalencia-osztályok módszere * osztályozás

- Ha a bemeneti feltétel értéktartományt definiál, az **érvényes** ekvivalencia osztály legyen a megengedett bemenő értékek halmaza, az **érvénytelen** ekvivalencia osztályok pedig az alsó és a felső határoló tartomány. Pl. ha az adatok osztályzatok (értékük 1 és 5 között van), akkor ezek az ekvivalencia osztályok rendre: $\{1 \leq i \leq 5\}$, $\{i < 1\}$ és $\{i > 5\}$.
- Ha a bemeneti feltétel értékek számát határozza meg, akkor az előzőhöz hasonlóan járjunk el. Pl. ha be kell olvassunk legfeljebb 6 karaktert, akkor az érvényes ekvivalencia osztály: 0-6 karakter beolvasása, az érvénytelen ekvivalencia osztály: 6-nál több karakter beolvasása. (0-nál kevesebb nem fordulhat elő.)

Ekvivalencia-osztályok módszere * osztályozás

- Ha a bemenet feltétele azt mondja ki, hogy a bemenő adatnak valamilyen meghatározott jellemzővel kell rendelkezni, akkor két ekvivalencia osztályt kell felvenni: egy **érvényes**et és egy **érvénytelen**t.
- Ha okunk van feltételezni, hogy a program valamelyik ekvivalencia osztályba eső elemeket különféleképpen kezeli, akkor a feltételezésnek megfelelően bontsuk az ekvivalencia osztályt további osztályokra.
- Alkalmazzuk ugyanezeket az elveket a kimeneti ekvivalencia osztályokra is!

Ekvivalencia-osztályok módszere * tesztesetek

A teszteseteket a következő két elv alapján határozhatjuk meg:

- Amíg az **érvényes** ekvivalencia osztályokat le nem fedtük, addig készítsünk olyan teszteseteket, amelyek minél több érvényes ekvivalencia osztályt lefednek!
- Minden **érvénytelen** ekvivalencia osztályra írjunk egy-egy, az osztályt lefedő tesztesetet. Több hiba esetén ugyanis előfordulhat, hogy a hibás adatok lefedik egymást, a második hiba kijelzésére az első hibajelzés miatt már nem kerül sor. Megjegyzés: mindegyikhez 1-1 hibajelzésnek kell tartoznia a programban.

Ekvivalencia-osztályok módszere * tesztesetek * példafeladat

Bemenet dátum (év+hó+nap)

Kimenet: A) hányadik napja az évnek; B) melyik évszakba esik; ...

Érvényes ekvivalenciaosztályok:

- **év_A:** 1) $400 \mid \text{év}$,
2) nem $100 \mid \text{év}$ és $4 \mid \text{év}$,
3) nem $4 \mid \text{év}$
(Szökőév: 1), 2))
- **hó_B:** 1) $\{12, 1, 2\}$,
2) $\{3..5\}$,
3) $\{6..8\}$,
4) $\{9..11\}$
- **nap_A:** 1) Szökőév(év) és $\text{hó} \in \{3..12\}$,
2) Szökőév(év) és $\text{hó}=2$ és $\text{nap}=29$,
3) nem Szökőév(év)

Ekvivalencia-osztályok módszere * tesztesetek * példafeladat

Bemenet dátum (év+hó+nap)

Kimenet: A) hányadik napja az évnek; B) melyik évszakba esik; ...

Érvénytelen ekvivalenciaosztályok (a beolvasáshoz)

- **év:** a) nem természetes szám,
b) $\{..1581\}$,
c) $\{2024..\}$
- **hó:** a) nem természetes szám,
b) $\{..0\}$,
c) $\{13..\}$
- **nap:** a) nem pozitív természetes szám,
b) $hó \in \{4,6,9,11\}$ és $nap \in \{31..\}$,
c) $hó \in \{1,3,5,7,8,10,12\}$ és $nap \in \{32..\}$,
d) Szökőév(év) és $hó=2$ és $nap \in \{30..\}$,
e) nem Szökőév(év) és $hó=2$ és $nap \in \{29..\}$

Ekvivalencia-osztályok módszere * tesztesetek * példafeladat

Bemenet dátum (év+hó+nap)

Kimenet: A) hányadik napja az évnek; B) melyik évszakba esik; ...

Teszt-bemenetek

Érvényesekre:

- **2016 2 29** reprezentálja az év/2, hó/1, nap/2 ekvivalenciaosztályokat
- **2017 3 31** reprezentálja az év/3, hó/2, nap/3 (év/c) ekvivalenciaosztályokat
- **2015 6 30** reprezentálja a(z év/3,) hó/3 (nap/3, év/c) ekvivalenciaosztályokat
- ...

Érvénytelenekre:

(határeset elemzéssel [l. a következő diától])

- **2016 2 29**-on túl: **2016 2 30**, **2016 2 1**, **2016 2 0**
- ...

Határeset elemzés módszere

- Ha a bemeneti feltétel egy értéktartományt jelöl meg, írjunk teszteseteket az érvényes tartomány alsó és felső határára és az érvénytelen tartománynak a határ közelébe eső elemére! Pl.: ha a bemeneti tartomány a $(0,1)$ nyílt intervallum, akkor a 0, 1, 0.01, 0.99 értékekre érdemes kipróbálni a programot.
- Ha egy bemeneti feltétel értékek számosságát adja meg, akkor hasonlóan járjunk el, mint az előző esetben. Pl.: ha rendeznünk kell 1-128 nevet, akkor célszerű a programot kipróbálni 0, 1, 128, 129 névvel.

Dinamikus tesztelés * fekete doboz módszer * példa₁

Specifikáció:

Be: $n \in \mathbb{N}$

Ki: $o \in \mathbb{N}$, $van \in \mathbb{L}$

Ef: $n > 1$

Uf: $van = \exists i \in [2..n-1] : (i | n) \text{ és } van \rightarrow (2 \leq o < n \text{ és } o | n \text{ és } \forall i \in [2..o-1] : (i \nmid n))$

Feladat: Adjuk meg egy n természetes szám valódi (1-től és önmagától különböző) osztóját!

Érvényes adatokra

Ekvivalencia osztályok (bemenet alapján):

1. n prímszám: 3

2. n -nek egy(-féle) valódi osztója van: $25 = 5 * 5$

3. n -nek több, különböző valódi osztója is van: $77 = 7 * 11$

4. n páros

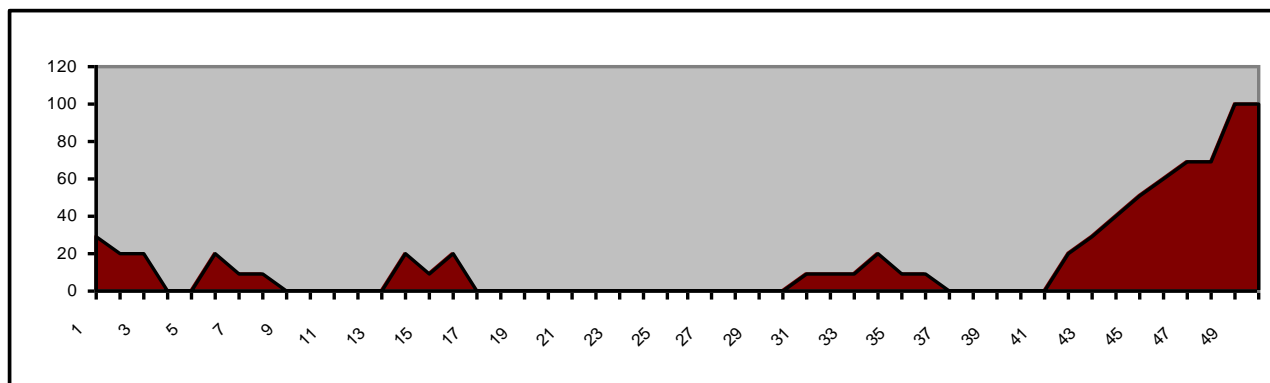
5. $n \leq 1$, vagy bármilyen, ami nem természetes szám

Érvénytelen adatokra

Dinamikus tesztelés * fekete doboz módszer * példa₂

Feladat:

Egy repülőgéppel Európából Amerikába repültünk. Az út során bizonyos kilométerenként mértük a felszín tengerszint feletti magasságát (≥ 0). 0 magasságot ott mértünk, ahol tenger van, >0 -t pedig ott, ahol szárazföld. Adjuk meg a legszélesebb szigetet!



Dinamikus tesztelés * fekete doboz módszer * példa₂

Specifikáció:

Be: $n \in \mathbb{N}$, $\text{mag} \in \mathbb{Z}[1..n]$

Ki: $\text{van} \in \mathbb{L}$, $k, v \in \mathbb{N}$

Ef: $n \geq 3$ és
 $\text{mag}[1] > 0$ és $\text{mag}[n] > 0$ és
 $\forall i \in [2..n-1]: \text{Mag}[i] \geq 0$ és
 $\exists i \in [2..n-1]: \text{Mag}[i] = 0$

Uf: ...


Feladat:

Egy repülőgéppel Európából Amerikába repültünk. Az út során bizonyos km-ként mértük a felszín tengerszint feletti magasságát (≥ 0). 0 magasságot ott mértünk, ahol tenger van, >0 -t pedig ott, ahol szárazföld. Adjuk meg a legszélesebb szigetet!

Érvénytelen ekvivalencia osztályok:

- $n < 3$
- $\text{mag}[1] \leq 0$
- $\exists i \in [2..n-1]: \text{mag}[i] < 0$
- $\forall i \in [2..n-1]: \text{mag}[i] > 0$
- $\text{mag}[n] \leq 0$

Dinamikus tesztelés * fekete doboz módszer * példa₂



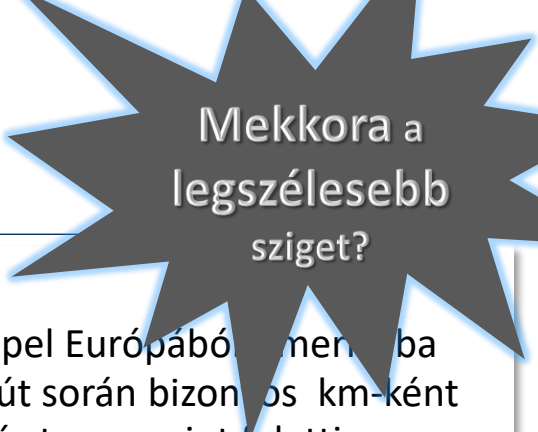
Melyik a
legszélesebb
sziget?

Érvényes ekvivalencia osztályok (a kimenet alapján):

- nincs sziget (1.)
- van sziget
 - egy sziget van (2.)
 - több sziget van
 - egyforma szélességűek (3.)
 - nem egyforma szélességűek
 - az első a legszélesebb (4.)
 - az utolsó a legszélesebb (5.)
 - egy közbülső a legszélesebb (6.)

6 ekvivalencia-osztály

Dinamikus tesztelés * fekete doboz módszer * példa₂



Mekkora a
legszélesebb
sziget?

Határesetek:

- Európa 1 szélességű,
nem 1 szélességű;
- Amerika 1 szélességű,
nem 1 szélességű;
- a legszélesebb sziget 1 szélességű,
nem 1 szélességű;
- a legszélesebb szigetet (bal, ill. jobb) szomszédjától 1 szélességű
tenger választja el,
nem 1 szélességű tenger választja el.

Feladat:

Egy repülőgéppel Európából Amerikába repültünk. Az út során bizonyos km-ként mértük a felszín tengerszint feletti magasságát (≥ 0). 0 magasságot ott mértünk, ahol tenger van, >0 -t pedig ott, ahol szárazföld. Adjuk meg a legszélesebb szigetet!

Ez hány teszteset? Az első kettő négyféleképpen kombinálható. A harmadik ezek számát megduplázza. A negyedik pedig négyszerezi.

32 teszteset!

Dinamikus tesztelés * fekete doboz módszer * példa₂

Megtaláljuk ezzel az **összes** hibát?

Tételezzük fel, hogy a megoldásban először megkeressük Európa utolsó és Amerika első pontját, majd e két pont között keressük a szigeteket. Ha az Amerika első pontját tartalmazó változót elrontjuk, pl. $N/2$ -re vagy $N-10$ -re állítjuk, akkor mi a garancia arra, hogy a korábbi tesztek ezt felfedezik?

A csökkentett tartományban van az, amire a teszt fókuszál, akkor ez a hiba nem vevődik észre.

Mi van, ha a programunk Európát és Amerikát is szigetként veszi számításba, és adja legszélesebb szigetnek?

Ha olyan teszteset nincs, amelyben a legszélesebbnél szélesebb valamelyik kontinens, akkor ez a hiba nem vevődik észre.

Dinamikus tesztelés * szürke doboz módszer

Lényeg:

az algoritmus

- valamely globális jellemzőjét,
- a végrehajtás jellegzetes „logikáját”,
- a részletek mellőzésével... „madártávlatból”

vizsgáljuk.

Dinamikus tesztelés * szürke doboz módszer

„Globális” jellemző
vizsgálata

Határok vizsgálata sorozatoknál

- Első elem feldolgozásra kerül-e
- Utolsó elem feldolgozásra kerül-e
- Közbenső elem feldolgozásra kerül-e

Sorozat mérete szerint:

- Üres sorozat kezelése
- Egy elemű sorozat kezelése
- (Két elemű sorozat kezelése)
- Több elemű sorozat kezelése

Dinamikus tesztelés * szürke doboz módszer

Tétel-specifikus
vizsgálat

Összegzés

- Két különböző elemet tartalmazó sorozat.
- Terheléses teszt, túlcsordulás vizsgálat.

Megszámolás

- A sorozatban az adott tulajdonságnak eleget tevők száma
 - nulla,
 - egy,
 - legalább kettő,
 - az összes.

Kiválasztás

- A kiválasztandó elem a sorozat első eleme.
- A kiválasztandó elem a sorozatnak nem az első eleme.

Dinamikus tesztelés * szürke doboz módszer

Tétel-specifikus
vizsgálat

Keresés

- A keresett elem a sorozat első eleme.
- A keresett elem a sorozat utolsó eleme.
- Létezik a keresett tulajdonságnak megfelelő közbenső elem.
- Nem létezik a keresett tulajdonságnak megfelelő elem.

Maximum-kiválasztás

- Két elemű sorozat, első eleme a nagyobb.
- Két elemű sorozat, második eleme a nagyobb.
- Több elemű sorozat közbenső eleme a legnagyobb.
- Több elemű sorozatban több maximális elem van.

Dinamikus tesztelés * fehér doboz módszer

Lépések

1. egy kipróbálási **stratégiát** választunk a program szerkezete alapján,
2. a stratégia **tesztutak**at jelöl ki,
3. a tesztutakhoz **tesztpredikátumok**at rendelünk,
4. a tesztpredikátumok **ekvivalencia osztályok**at határoznak meg,
5. amelyekből egy-egy **teszteset**et választunk.

Dinamikus tesztelés * fehér doboz módszer

Kipróbálási stratégiák

- utasítás lefedés: minden utasítást legalább egyszer hajtsunk végre!
- feltétel lefedés: minden feltétel legyen legalább egyszer igaz, illetve hamis!
- részfeltétel lefedés: minden részfeltétel legyen legalább egyszer igaz, illetve hamis!

Dinamikus tesztelés * fehér doboz módszer

Teszteset-generálás

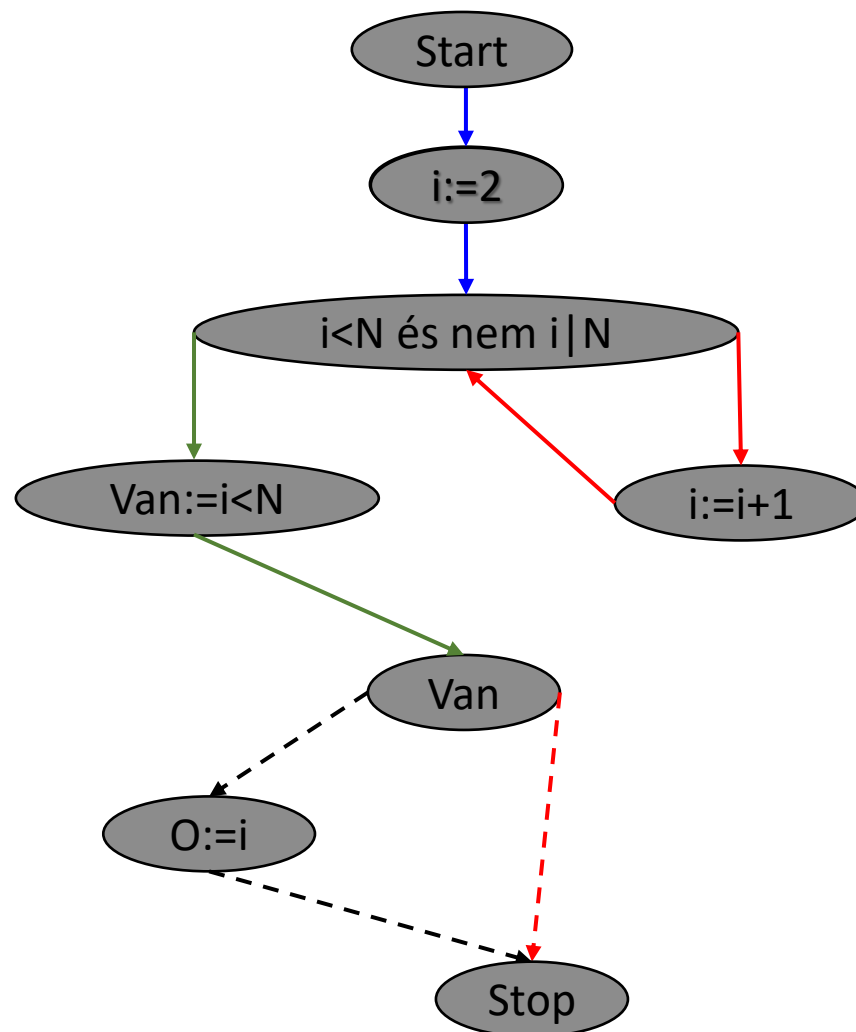
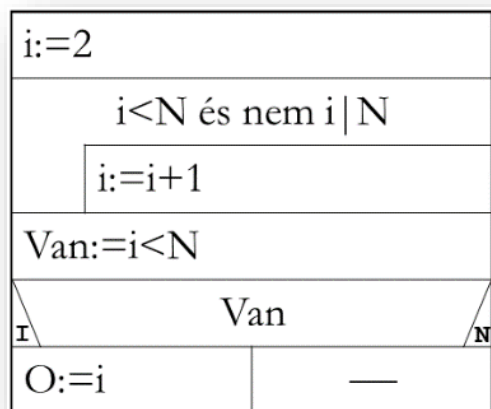
Bázisútaknak nevezzük a programgráf olyan útjait, amely

- a kezdőponttól a legelső elágazás- vagy ciklusfeltétel kiértékeléséig tart, vagy
- elágazás- vagy ciklusfeltételtől a következő elágazás- vagy ciklusfeltétel helyéig vezet, vagy
- elágazás- vagy ciklusfeltételtől a program végéig tart, s közben más feltétel kiértékelés nincs.



Dinamikus tesztelés * fehér doboz módszer

Programgráf



A különféle színű és/vagy mintázatú élekkel jelöltük az egyes bázisutakat.

Dinamikus tesztelés * fehér doboz módszer

Tesztutaknak nevezzük a programgráfon átvezető, a kezdőponttól a végpontig haladó olyan bázisút-sorozatok, amelyek minden bennük szereplő élt pontosan egyszer tartalmazznak.

Tesztpredikátumnak nevezzük azokat a bemenetre vonatkozó feltételeket, amelyek teljesülése esetén pontosan egy tesztúton kell végighaladni.

A **teszteset-generálás** első lépése a minimális számú olyan tesztút meghatározása, amelyek lefedik a kipróbálási stratégiának megfelelően a programgráfot.

Dinamikus tesztelés * fehér doboz módszer

A tesztpredikátum előállítása:

Ehhez a program **szimbolikus végrehajtás**sára van szükség. Induljunk ki az **előfeltételből**! Haladjunk a programban az első elágazás- vagy ciklusfeltételig, s a formulát a közbülső műveleteknek megfelelően **transzformáljuk**! A tesztútnak megfelelő **ág feltételét és kapcsolattal kapcsoljuk hozzá a tesztpredikátumhoz**, majd folytassuk a szimbolikus végrehajtást egészen a program végpontjáig!

Dinamikus tesztelés * fehér doboz módszer * példa

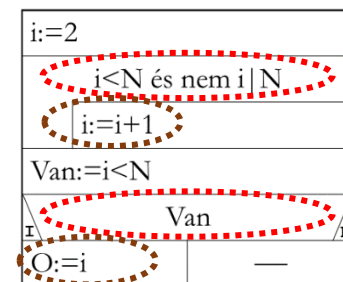
Feladat: Egy N természetes szám valódi (1-től és önmagától különböző) osztója...

Utasítás lefedés:

- $i:=i+1$ végrehajtandó: $N=3$
- $O:=i$ végrehajtandó: $(\leftarrow \text{Van}=\text{Igaz})$ $N=4$

Feltétel lefedés:

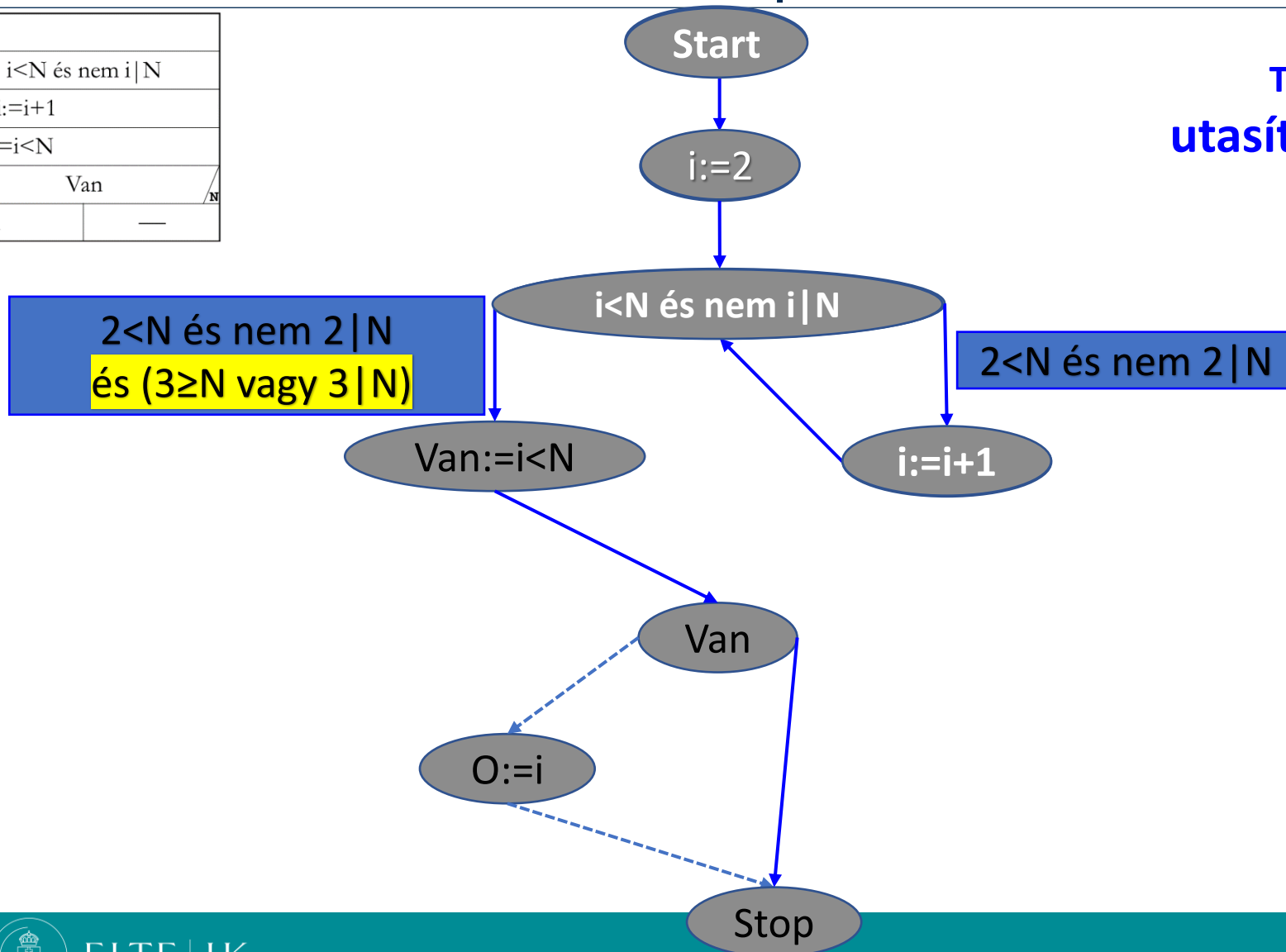
- Ciklusfeltétel igaz: $N=3$
- Ciklusfeltétel hamis: $N=2$ (be sem lép)
- Elágazásfeltétel igaz: $(\leftrightarrow \text{Van}=\text{Igaz})$ $N=4$
- Elágazásfeltétel hamis: $(\leftrightarrow \text{Van}=\text{Hamis})$ $N=2$



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

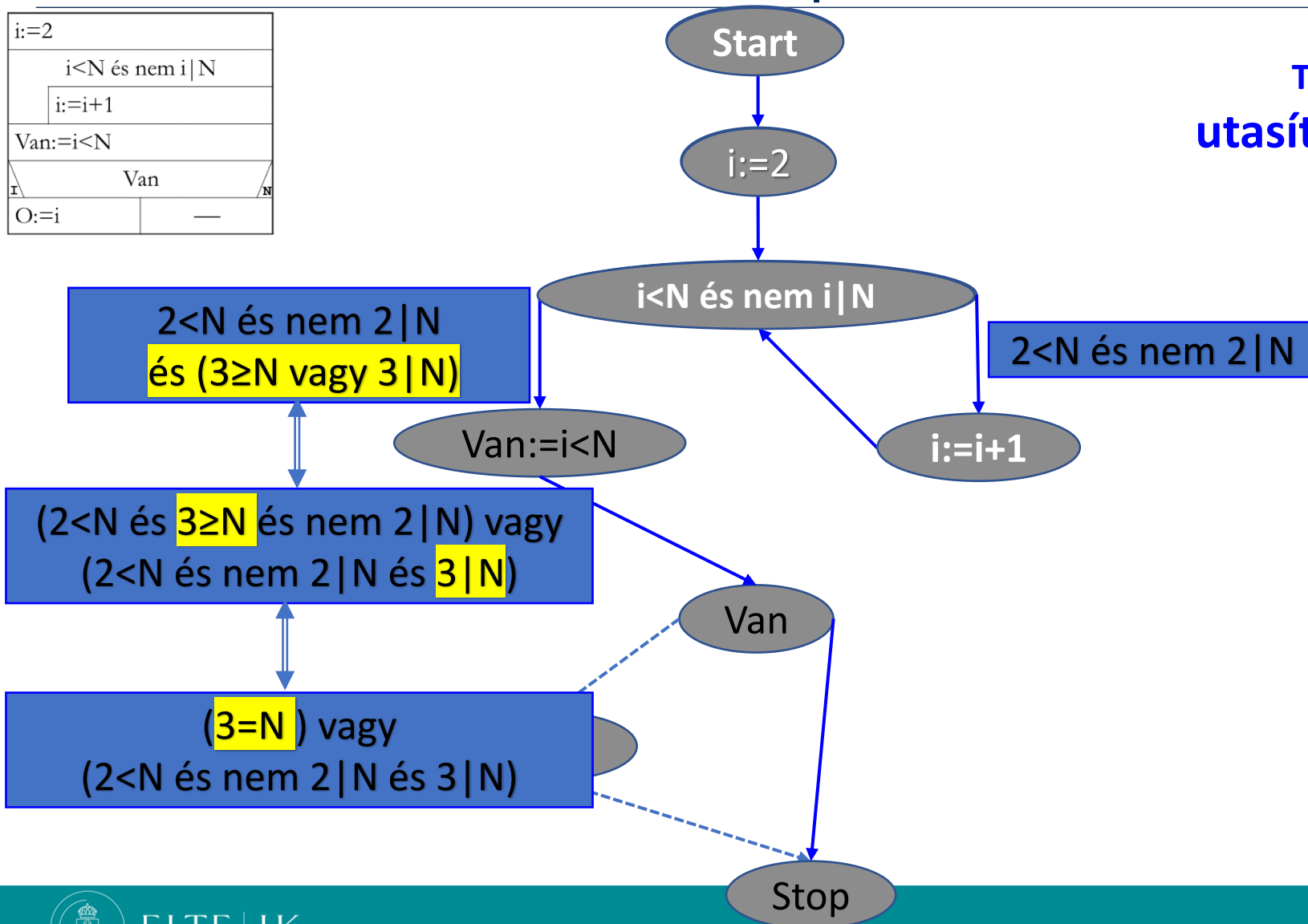
**Tesztút₁
utasításlefedés**



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

Tesztút₁
utasításlefedés

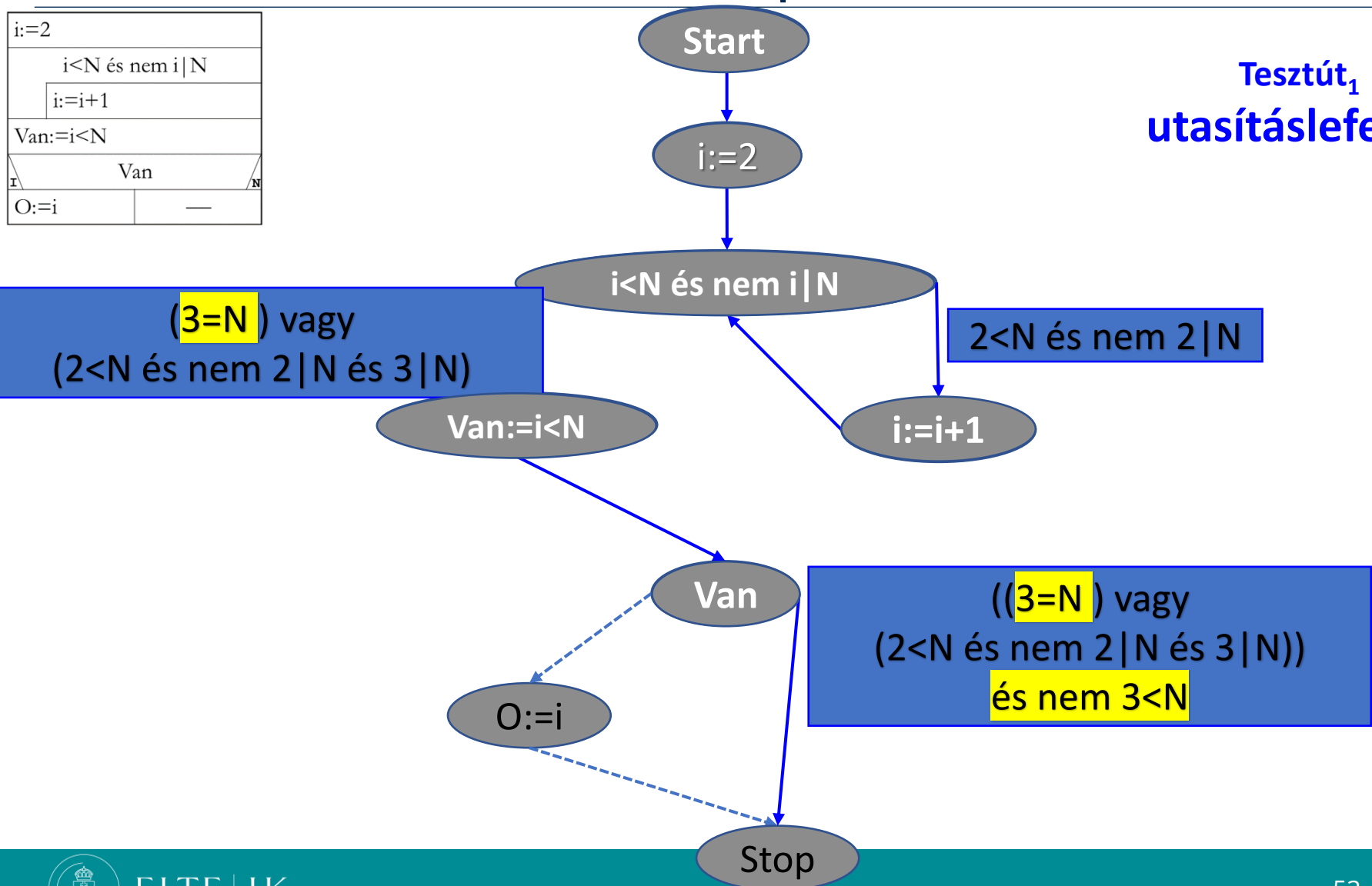
i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

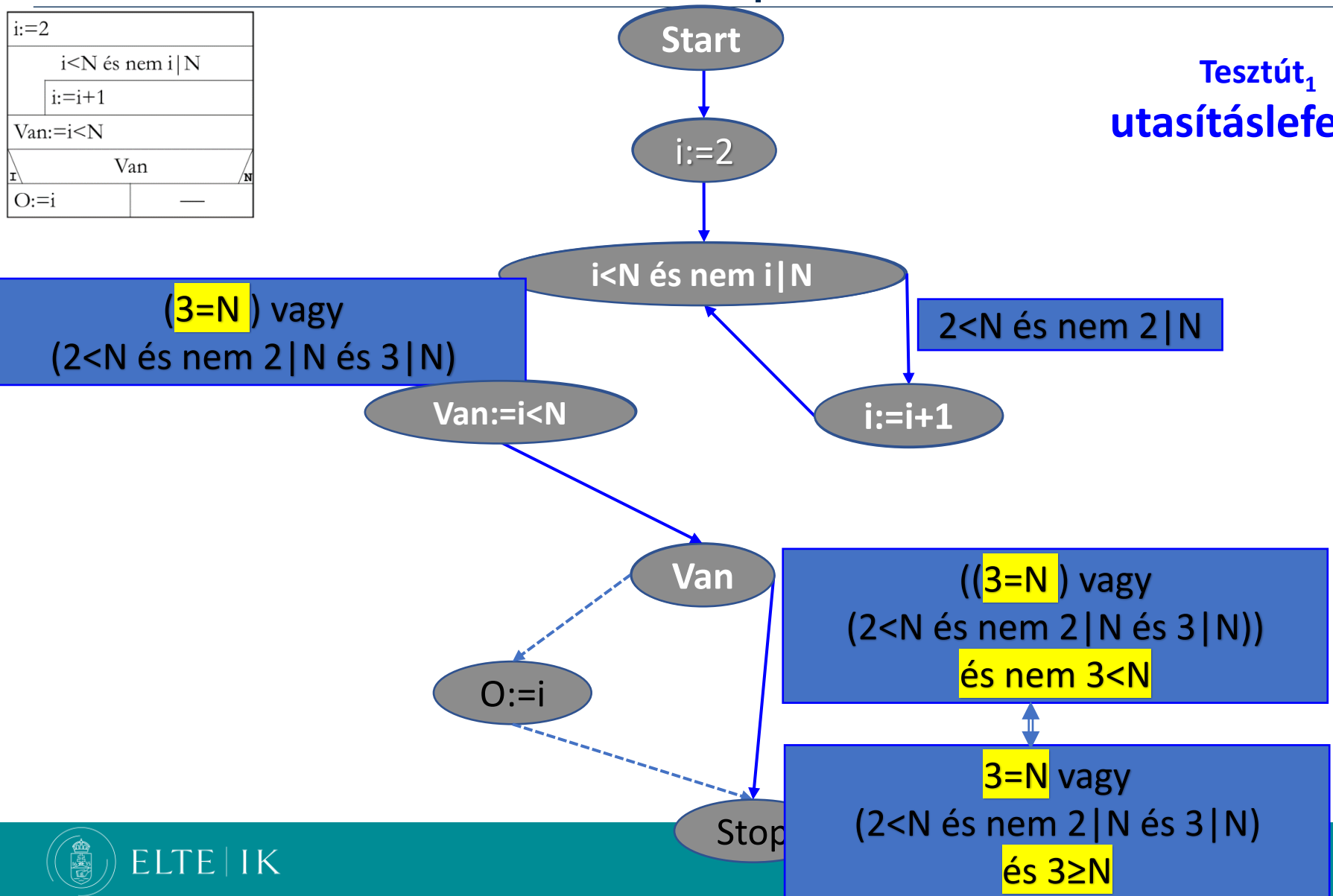
**Tesztút₁
utasításlefedés**



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

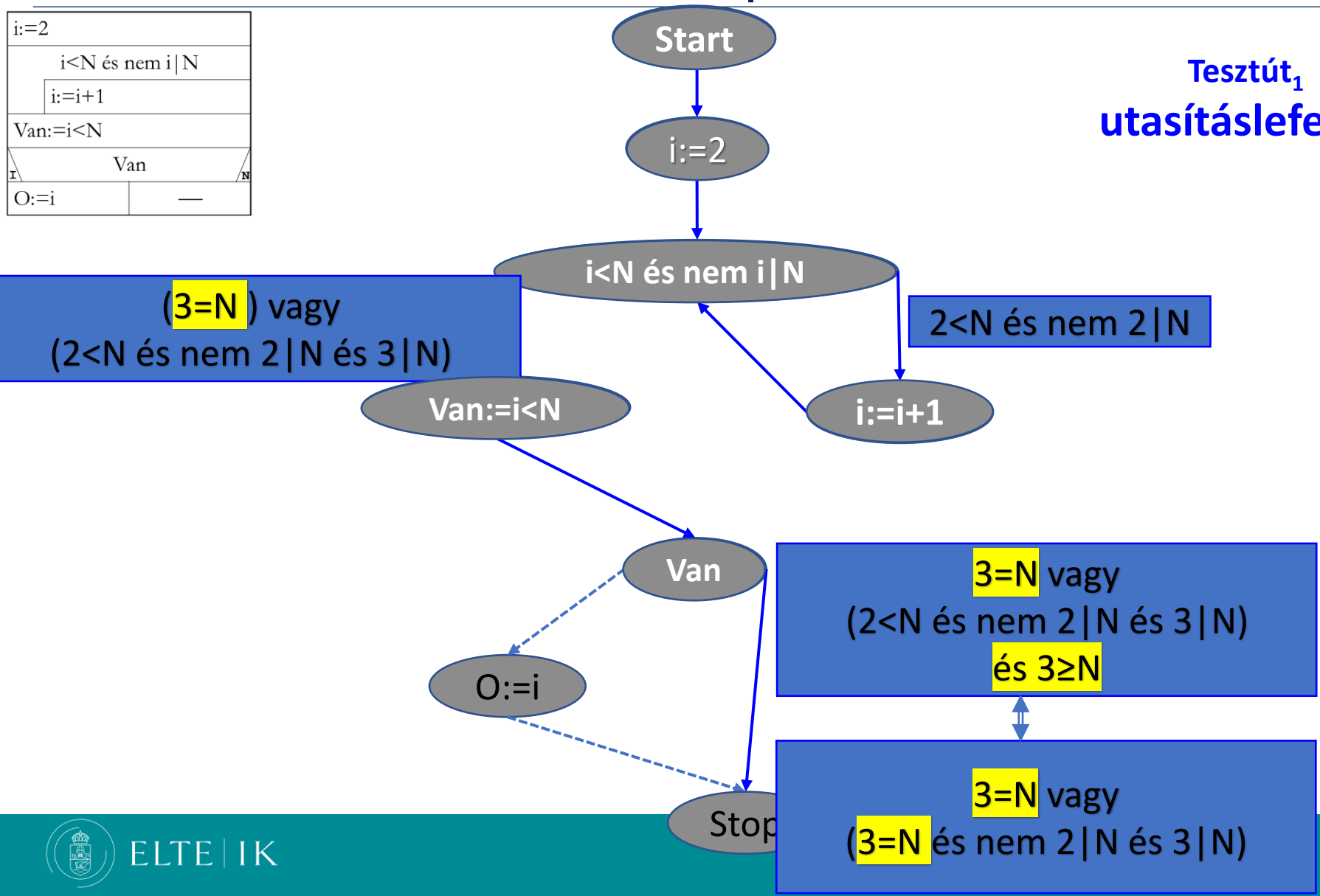
**Tesztút₁
utasításlefedés**



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

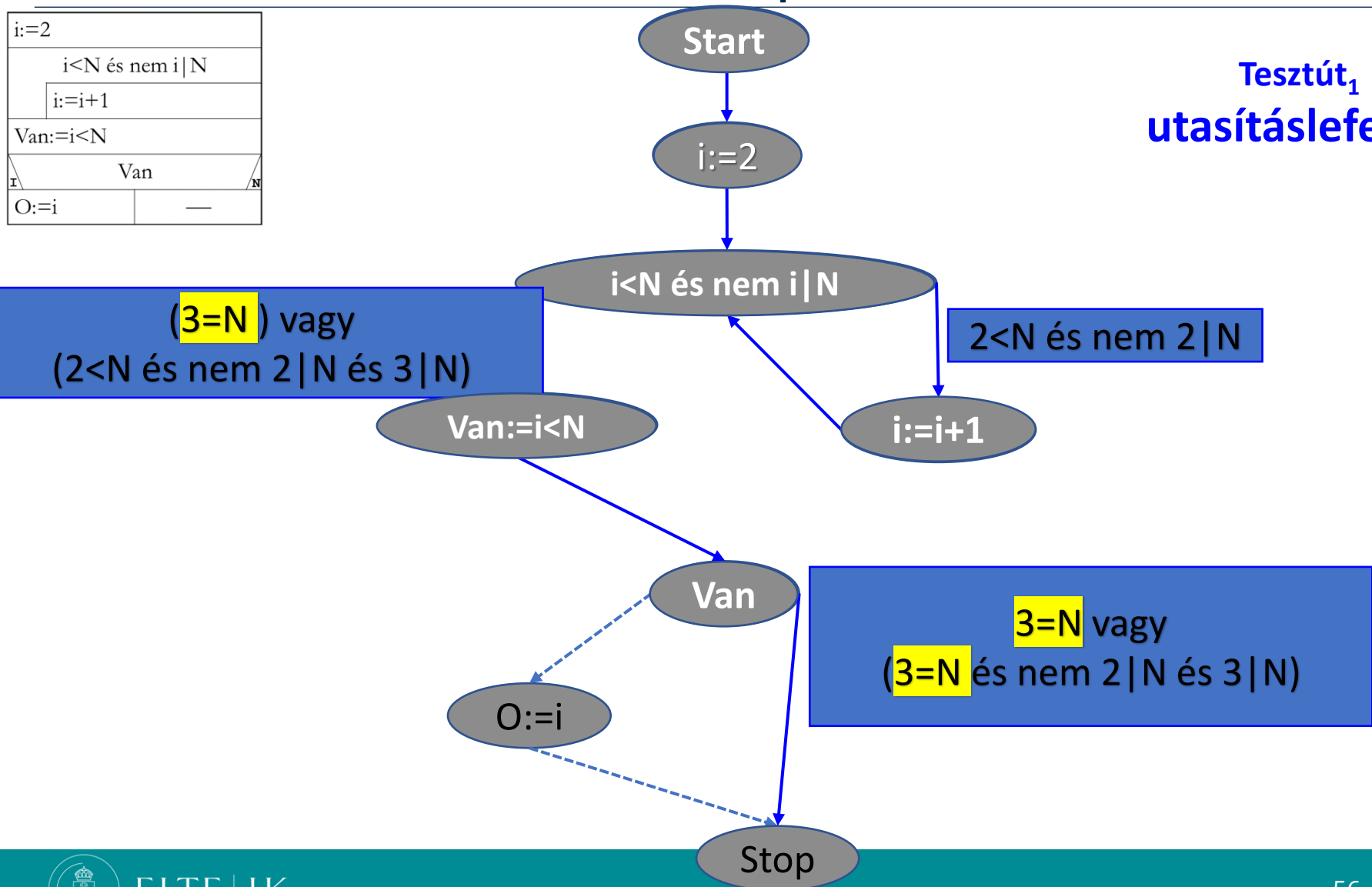
**Tesztút₁
utasításlefedés**



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

**Tesztút₁
utasításlefedés**



Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

i:=2	
i<N és nem i N	
i:=i+1	
Van:=i<N	
Van	
O:=i	—

N=3

Start

i:=2

i<N és nem i | N

(3=N) vagy
(2<N és nem 2 | N és 3 | N)

2<N és nem 2 | N

Van:=i<N

i:=i+1

Van

3=N vagy
(3=N)

O:=i

Stop

Tesztút₁
utasításlefedés

Dinamikus tesztelés * fehér doboz módszer * példa (formális levezetés)

$i:=2$
$i < N$ és nem $i \mid N$
$i:=i+1$
$\text{Van}:=i < N$
Van
$O:=i$

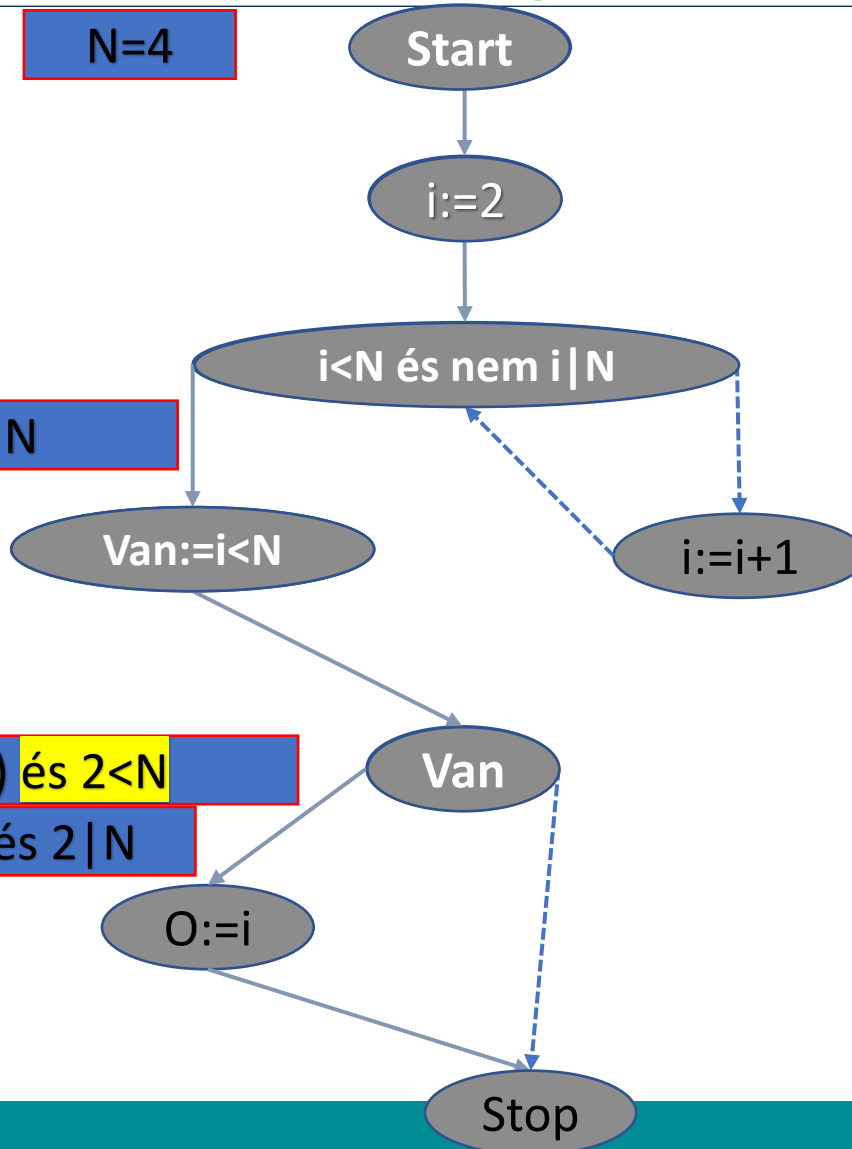
N=4

**Tesztút₂
utasításlefedés**

$2 \geq N$ vagy $2 \mid N$

$(2 \geq N \text{ vagy } 2 \mid N) \text{ és } 2 < N$

$2 < N$ és $2 \mid N$



Speciális tesztelések

- Biztonsági teszt: ellenőrzések vannak?
- Hatékonysági teszt

Speciális programokhoz

- Funkcióteszt: tud minden funkciót?
- Stressz-teszt: gyorsan jönnek a feldolgozandók, ...
- Volumen-teszt: sok adat sem zavarja

Tesztelés automatizálása

Teszt-generálás:

- kézi
- automatikus (generáló program)
 - szabályos
 - véletlenszerű

Teszt-futtatás:

- kézi
- automatikus (parancsfájl, be- és kimeneti állományok, automatikus értékelés)

L. 4. előadás

Tesztelés automatizálása * futtatás adatfájlal

Elv:

A standard input/output átirányítható fájlba. Ekkor a program **fájl**t használ az inputhoz és az outputhoz. Következmény: **szervezetileg a konzol inputtal/outputtal megegyező kell legyen / lesz a megfelelő fájl.**

„Technika”:

A lefordított kód mögé kell paraméterként írni a megfelelő fájlok nevét:

```
prog.exe <inputfájl >outputfájl
```

Ilyenek futtatását (az összes tesztre) szervező szkriptfájlal tovább gyorsítható a tesztelés.

Nyereség:

Gyors, kényelmes és adminisztrálható (pl. a dokumentáció számára) tesztelés.

L. 5. előadás

Tesztelés automatizálása * futtatás adatfájlal

L. 5. előadás

Kódolás:

Ügyeljen az outputok elkülönítésére!

- **Csak a felhasználóknak** küldött üzenet esetén:

```
Console.Error.Write("...kérdés a felhasználónak...");  
vagy  
Console.Error.WriteLine("...üzenet a felhasználónak...");
```

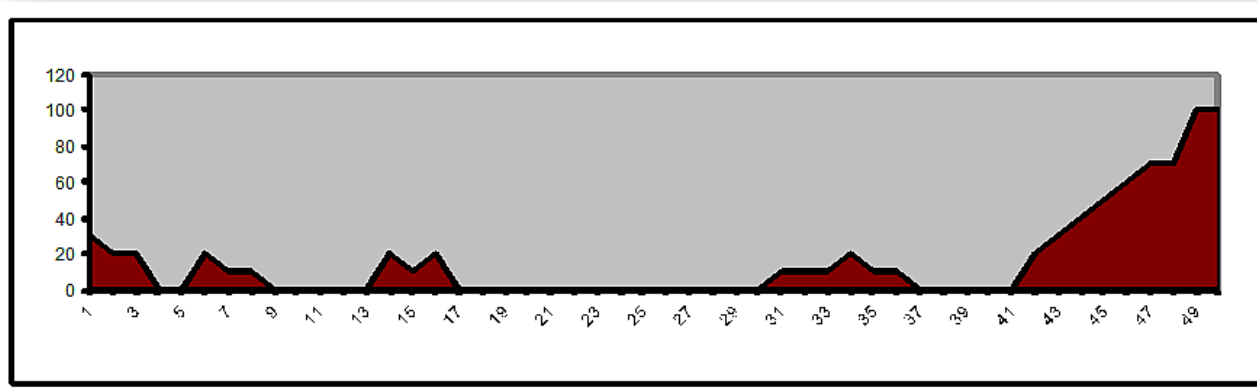
- A lényegi (pl. a tesztelés szempontjából érdekes, vagy a Bíró számára értékelendő) output:

```
Console.Write("...kérdés a felhasználónak...");  
Vagy  
Console.WriteLine("...üzenet a felhasználónak...");
```

Tesztek előállítása * példa

Feladat (teszteléshez):

Egy repülőgéppel Európából Amerikába repültünk. Az út során X kilométerenként mértük a felszín tengerszint feletti magasságát (≥ 0). 0 magasságot ott mértünk, ahol tenger van, >0 -t pedig ott, ahol szárazföld. Adjuk meg a szigeteket!



Tesztek előállítása * példa

Specifikáció:

Be: $n \in \mathbb{N}$, $\text{mag} \in \mathbb{Z}[1..n]$

Ki: $\text{db} \in \mathbb{N}$, $k, v \in \mathbb{N}[1..\text{db}]$

...

Tesztelés:

- **Kis** tesztek a tesztelési elveknek megfelelően, például:
 - $n=3$, $\text{mag}=(1,0,1)$ → nincs sziget
 - $n=5$, $\text{mag}=(1,0,1,0,1)$ → egy „rövid” sziget
 - $n=7$, $\text{mag}=(1,0,1,0,1,0,1)$ → több „rövid” sziget
 - $n=7$, $\text{mag}=(1,0,1,1,1,0,1)$ → hosszabb sziget
 - $n=8$, $\text{mag}=(1,0,1,1,0,1,1,1)$ → hosszabb sziget, még hosszabb kontinens
- ...
- Hogyan készítünk **nagy** (pl. hatékonysági) tesztek?

Tesztek előállítása * szabályos tesztek

Generálhatunk „**szabályos**” tesztek (egyszerű ciklusokkal) tesztbemenet(fájlok) gyártására.

Például így:

n:=1000		Változó i:Egész	
i=1..10			
	mag[i]:=11-i		⇐ Európa
i=11..900			
	mag[i]:=0		⇐ tenger
i=901..n			
	mag[i]:=i-900		⇐ Amerika

Probléma: sokféle teszteset sok célprogramot igényel.

Megoldás: véletlen tesztbemenetek gyártása.

Tesztek előállítása * véletlen tesztek *

elvi alapok

Mitől véletlen egy számsorozat? Pl. $x[1..] \in \{0,1\}$, akkor véletlenek-e az alábbiak:

a) 0,0,0,0,...,0

←csupa 0

elvárás: minden lehetséges érték előforduljon

b) 0,1,0,1,...,0,1

←csupa 0,1 (1,0)

elvárás: minden pár (0,0/0,1/1,0/1,1) előforduljon

c) 0,0,0,1,1,1,0,0,0,...,1,1,1

Tesztek előállítása * véletlen tesztek *

elvi alapok

Mitől véletlen egy számsorozat? Pl. $x[1..] \in \{0,1\}$, akkor véletlenek-e az alábbiak:

a) 0,0,0,0,...,0

←csupa 0

elvárás: minden lehetséges érték előforduljon

b) 0,1,0,1,...,0,1

←csupa 0,1 (1,0)

elvárás: minden pár (0,0/0,1/1,0/1,1) előforduljon

c) 0,0, 0,1, 1,1, 0,0,0,...,1,1,1

←3-asok közül csak a 0,0,0 és 1,1,1; ...

elvárás: minden 3-s, 4-es ... előforduljon

elvárások:

1. Minden részsorozat kb. a matematikai valószínűségével forduljon elő!
2. Kellően „meglepően”, „szabálytalanul” következzenek egymás után!

Tesztek előállítása * véletlen tesztek *

elvi alapok

Mitől véletlen egy számsorozat? Pl. $x[1..] \in \{0,1\}$, akkor véletlenek-e az alábbiak:

a) 0,0,0,0,...,0

←csupa 0

elvárás: minden lehetséges érték előforduljon

b) 0,1,0,1,...,0,1

←csupa 0,1

elvárás: minden 2-s (0,0/0,1/1,0/1,1) legyen benne

c) 0,0,0,1,1,1,0,0,0,...,1,1,1

←3-asok közül csak a 0,0,0 és 1,1,1

elvárás: minden 3-s ... is legyen benne

d) 0, 0,1, 1,0, 1,1, 1,0,0, 1,0,1,
1,1,0, 1,1,1, 1,0,0,0,...

←jónak látszik

pedig nem véletlen, a „szabály”: $0_2, 1_2, 2_2, 3_2, 4_2, 5_2, 6_2, 7_2, 8_2, \dots$

Tesztek előállítása * véletlen tesztek * elvi alapok

A **véletlenszám**okat a számítógép egy algoritmussal állítja elő egy kezdőszámból kiindulva.

$$x_0 \rightarrow f(x_0)=x_1 \rightarrow f(x_1)=x_2 \rightarrow \dots$$

A „véletlenszerűséghez” megfelelő függvény és jó kezdőszám szükséges.

- **Kezdőszám:** (pl.) a belső órából vett érték.
- **Függvény** (az ún. lineáris kongruencia módszernél):

$$f(x) = (A \cdot x + B) \text{ Mod } M,$$

ahol A, B és M a függvény belső konstansai.

Tesztek előállítása * véletlen tesztek * algorithmus és C#

C#: `rnd.Next(...)` véletlen egész számot ad a paramétere(ek) által meghatározott intervallumban, ahol az `rnd` egy `Random` típusú (osztálybeli) **változó** (objektum): `Random rnd=new Random();`

- Véletlen($a..b$) $\in\{a,...,b\}$

```
v=rnd.Next(a,b)
```

- Véletlen(N) $\in\{1,...,N\}$

```
v=rnd.Next(N)+1
```

- Véletlenszám $\in[0..1)\subset\mathbb{R}$

```
v=(double)rnd.Next()/Int32.MaxValue
```

A véletlenszám generátor x_0 induló értéke a deklaráláskor beállítható:

```
Random rnd = new Random((int)DateTime.Now.Ticks);
```

- Az **x** kiinduló véletlen szám szerepe:

- x-hez egyedi véletlenszám-sorozat

- A pillanatnyi rendszetidő véletlen, miért **nem jó** mégsem, ha a `:Next()` függvény helyett minduntalan a rendszeridőből újrainicializáljuk?

Tesztek előállítása * véletlen tesztek * algorithmus

A **Véletlenszám** $\in [0..1)$ függvényhez:

Példafeladat: kockadobás szimulálása

„Cinkelt” kocka nem azonos eséllyel fordul az egyes oldalaira.

Legyen $P[i]$ az i ($i=1..6$) oldalára fordulás esélye. Elvárás

Ef: $\forall i \in [1..6]: P[i] \in [0..1)$ és $SZUMMA(i=1..6, P[i])=1$

Uf: $oldal \in [0..6]$

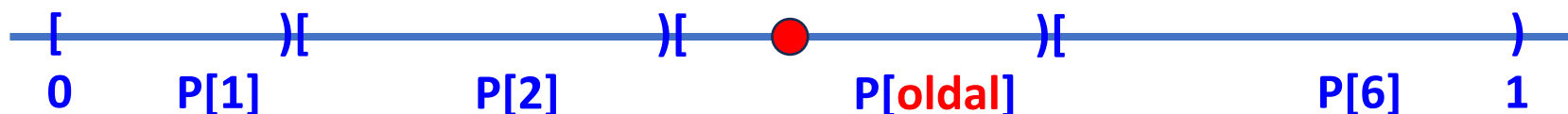
```
rnd:=Véletlenszám; oldal:=1; ps:=P[1]
```

```
rnd>=ps
```

```
    oldal:=oldal+1; ps:=ps+P[oldal]
```

Változó

rnd,ps:Valós



Tesztek előállítása * véletlen tesztek *

C# alapok

Próba:

1. A véletlenszám kezdőértéke
 1. a rendszeridőből származik,
 2. konstans paraméter
 3. újraállítható
2. A véletlenszám kezdőértékétől függ a generált sorozat

Tesztek előállítása * véletlen tesztek *

C# alapok

```
VéletlenPróba((int)DateTime.Now.Ticks, 1, /* > */ 0, 5); g, int db)
```

```
VéletlenPróba(1, 1, 10, 10);  
VéletlenPróba(2, 1, 10, 10);
```

```
Console.WriteLine("\n----- újraindul? -----");  
VéletlenPróba(1, 1, 10, 10);  
VéletlenPróba(2, 1, 10, 10);
```

```
rnd.Next({1},{2}) sorozat:",
```

```
{  
    Console.Write(" | {0}.:{1}", i, rnd.Next(tól, ig));  
}  
}  
else  
{  
    Console.WriteLine("r  
    for (int i = 1; i <= 10; i++)  
    {  
        Console.Write(" | {0}.:{1}", i, rnd.Next(tól, ig));  
    }  
}  
Console.WriteLine();  
}
```

```
C:\Users\Szlávi Péter\Docume x + v  
rnd = new Random(-717569779); rnd.Next() sorozat:  
| 1.:2089383348 | 2.:850210450 | 3.:1328595787 | 4.:187843709 | 5.:152724339  
rnd = new Random(1); rnd.Next(1,10) sorozat:  
| 1.:3 | 2.:1 | 3.:5 | 4.:7 | 5.:6 | 6.:4 | 7.:4 | 8.:9 | 9.:1 | 10.:6  
rnd = new Random(2); rnd.Next(1,10) sorozat:  
| 1.:7 | 2.:4 | 3.:2 | 4.:9 | 5.:1 | 6.:3 | 7.:8 | 8.:5 | 9.:3 | 10.:1  
  
----- újraindul? -----  
rnd = new Random(1); rnd.Next(1,10) sorozat:  
| 1.:3 | 2.:1 | 3.:5 | 4.:7 | 5.:6 | 6.:4 | 7.:4 | 8.:9 | 9.:1 | 10.:6  
rnd = new Random(2); rnd.Next(1,10) sorozat:  
| 1.:7 | 2.:4 | 3.:2 | 4.:9 | 5.:1 | 6.:3 | 7.:8 | 8.:5 | 9.:3 | 10.:1  
|
```

Tesztek előállítása * véletlen tesztek

Véletlen tesztekhez használjunk véletlenszámokat! Például így:

n:=1000	
m:=Véletlen(9)	
i=1..m	
mag[i]:=Véletlen(4..10)	
i=m+1..900	
Véletlenszám<0.5	
mag[i]:=0	mag[i]:=1
i=901..n	
mag[i]:=Véletlen(2..8)	

Változó
i:Egész

⇐ Európa

⇐ tenger és szigetek

⇐ Amerika

Véletlen tesztek előállítása * véletlen sorozatok

Feladat:

Rendezést is tartalmazó feladatoknál gyakori egy rendezetlen $x[1..n]$ bemenet előállítása (1 és m közötti **különböző** n darab érték). A megoldhatóság nyilvánvaló feltétele: $n \leq m$.

Specifikáció:

Be: $n \in \mathbb{N}$, $m \in \mathbb{N}$

Ki: $x \in \mathbb{N}[1..n]$

Ef: $n \leq m$

Uf: $\forall i \in [1..n]:$

$(x[i] \in [1..m] \text{ és } \forall j \neq i \in [1..n]: x[j] \neq x[i])$

Megoldás:

Algoritmusváltozatokat írunk némileg eltérő Ef mellett.

Az algoritmusok helyességéhez be kell látni: az Uf teljesülését.

Véletlen tesztek előállítása * véletlen sorozatok

Be: $n \in \mathbb{N}$, $m \in \mathbb{N}$

Ki: $x \in \mathbb{N}[1..n]$

Ef: $n \leq m$ és $m \gg n$ [m sokkal nagyobb, mint n]

Uf: $\forall i \in [1..n]:$

$(x[i] \in [1..m] \text{ és } \forall j \neq i \in [1..n]: x[j] \neq x[i])$

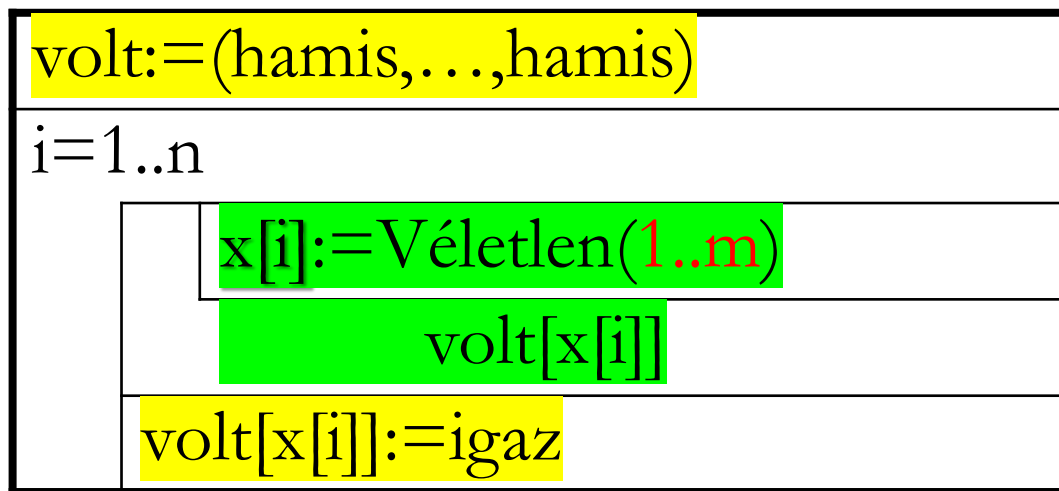
Ötlet:

1. Generáljunk $1..m$ intervallumból véletlen értéket! Ez az $1..m$ méretől, azaz az m nagyságától független idejű lépés.
2. Ha azt már generáltuk, akkor dobjuk el, és generáljunk újból!
3. Időspórolás érdekében: hogy ne kelljen eldöntés tétellel ellenőrizni az éppen generált szám meglétét, egy logikai tömbben jegyezzük, hogy generáltuk-e már vagy sem! (Ez m -től függő méretű tömb.)
4. Mindezt addig, amíg mind az n véletlen érték ki nem jön.

Véletlen tesztek előállítása * véletlen sorozatok

Uf: $\forall i \in [1..n]:$
 $(x[i] \in [1..m] \text{ és } \forall j \neq i \in [1..n]: x[j] \neq x[i])$

Algoritmus:



Változó

i: Egész

volt: Tömb[1..m:
Logikai]

Helyesség:

1. $\forall i \in [1..n]: x[i] \in [1..m] \leftarrow x[i] := \text{Véletlen}(1..m)$
2. $\forall i, j \neq i \in [1..n]: x[j] \neq x[i] \leftarrow$ inicializálás + belső ciklus + volt beállítás

Véletlen tesztek előállítása * véletlen sorozatok

Be: $n \in \mathbb{N}$, $m \in \mathbb{N}$

Ki: $x \in \mathbb{N}[1..n]$

Ef: $n \leq m$ és $m = n$

Uf: $\forall i \in [1..n]:$

$(x[i] \in [1..m] \text{ és } \forall j \neq i \in [1..n]: x[j] \neq x[i])$

Ötlet:

1. Induljunk ki abból, hogy kezdetben x fel van töltve a kellő $1..n$ értékekkel ($n=m$)!
2. Cseréljük meg az $1..n-1$ elemeket a mögöttük levők véletlenszerűen választott valamelyikével, és ő már a végleges helyén lesz!

Véletlen tesztek előállítása * véletlen sorozatok

Uf: $\forall i \in [1..n]:$
 $(x[i] \in [1..m] \text{ és } \forall j \neq i \in [1..n]: x[j] \neq x[i])$

Algoritmus:

$x := (1, 2, \dots, n)$ [x egy helyes permutáció]
$i = 1..n-1$
$j := \text{Véletlen}(i..n)$
$\text{Csere}(x[i], x[j])$ [x[i] már a helyén]

Változó
i, j: Egész

Helyesség:

1. $\forall i \in [1..n]: x[i] \in [1..m] \leftarrow x := (1, 2, \dots, n) [E_f \rightarrow n=m]$
2. $\forall i, j \neq i \in [1..n]: x[j] \neq x[i] \leftarrow \text{inicializálás} + \text{csere}$

Továbbá az is teljesül, hogy bármelyik érték bárhol azonos eséllyel lehet, azaz bármely permutáció azonos eséllyel jöhet ki. (Ezt most nem bizonyítjuk.)

Hibakeresés



Hibakeresés * hibajelenségek

Hibajelenségek a tesztelés során...

- hibás az eredmény,
- futási hiba keletkezett,
- nincs eredmény,
- részleges eredményt kaptunk,
- olyat is kiír, amit nem vártunk,
- túl sokat (sokszor) ír,
- nem áll le a program,
- ...

Hibakeresés * célok, elvek

Célja:

a felfedett hibajelenség okának, helyének megtalálása.

Elvek:

- Eszközök használata előtt alapos végiggondolás.
- Egy megtalált hiba a program más részeiben is okozhat hibát.
- A hibák száma, súlyossága a program méretével nemlineárisan (annál gyorsabban!) nő.
- Egyformán fontos, hogy *miért* nem csinálja a program, amit várunk, illetve, hogy *miért* csinál olyat, amit nem várunk.
- Csak akkor javítani, ha megtaláltuk a hibát!

Hibakeresési eszközök

L. Feltételes fordítás

Hibakeresési eszközök:

- Változó-, memória-kiírás
- Töréspont elhelyezése
- Lépésenkénti végrehajtás
- Adat-nyomkövetés
- Állapot-nyomkövetés (pl. paraméterekre vonatkozó előfeltételek, ciklus-invariánsok)
- Postmortem nyomkövetés: hibától visszafelé
- Speciális ellenőrzések

Hibakeresés *

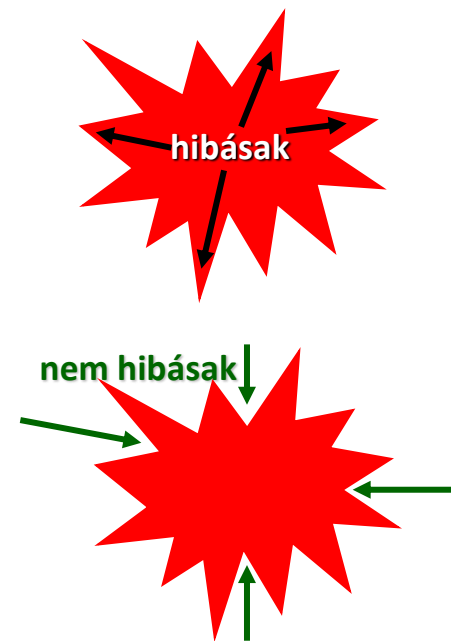
Célok, módszerek

Cél:

- A **bemenetnek mely** része, amelyre hibásan működik a program?
- **Hol** található a **programban** a hibát okozó utasítás?

Módszerfajták:

1. Indukciós módszer (hibásak körének **bővítése**)
2. Dedukciós módszer (hibásak körének **szűkítése**)
3. Hibakeresés hibától visszafelé
4. Teszteléssel segített hibakeresés (olyan teszteset kell, amely az ismert hiba helyét fedí fel)



Hibakeresés *

Indukciós módszer * példa

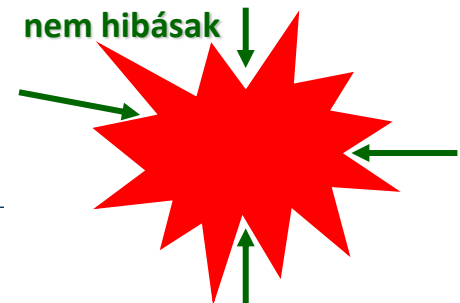


Feladat: 1 és 99 közötti N szám kiírása betűkkel.

- Tesztesetek: $N=8 \Rightarrow$ jó, $N=17 \Rightarrow$ jó, $N=30 \Rightarrow$ hibás.
- Próbáljunk a hibásakból általánosítani: tegyük fel, hogy minden 30-cal kezdődőre rossz!
- Ha beláttuk (teszteléssel), akkor próbáljuk tovább általánosítani, pl. tegyük fel, hogy minden 30 felettire rossz!
- Ha nem lehet tovább általánosítani, akkor tudjuk mit kell keresni a hibás programban.
- Ha nem ment az általánosítás, próbáljuk másképp: hibás-e minden 0-ra végződő számra!
- ...

Hibakeresés *

Dedukciós módszer * példa



Feladat: *1 és 99 közötti N szám kiírása betűkkel.*

- Tesztesetek: $N=8 \Rightarrow$ jó, $N=17 \Rightarrow$ jó, $N=30 \Rightarrow$ hibás.
- Tegyük fel, hogy minden nem jóra hibás!
- Próbáljunk a hibás esetek alapján szűkíteni:
tegyük fel, hogy a 20-nál kisebbekre jó!
- Ha beláttuk (teszteléssel), akkor szűkítsünk tovább, jó-e minden 39-nél nagyobbra?
- Ha nem szűkíthető tovább, akkor megtaláltuk, mit kell keresni a hibás programunkban.
- Ha nem, szűkítsünk másképp: tegyük fel, hogy jó minden nem 0-ra végződő számra!
- ...

Hibajavítás



Hibajavítás *

Cél, elvek

Cél:

a megtalált hiba kijavítása.

Elvek:

- A hibát kell javítani és nem a tüneteit.
- A hiba kijavítása a program más részében hibát okozhat (rosszul javítunk, illetve korábban elfedett más hibát).
- Javítás után a tesztelés megismételendő!
- A jó javítás valószínűsége a program méretével fordítva arányos.
- A hibajavítás a tervezési fázisba is visszanyúlhat (a módszertan célja: lehetőleg ne nyúljon vissza).

Dokumentálás



Dokumentálás *

Célok, fajták

Célok:

A vásárlónak, a felhasználónak és a (tovább)fejlesztőnek szükséges információk biztosítása.

Dőlten szedve, ami az aktuális komplex program estén a dokumentációból elhagyható.

Fajtái:

- *Programismertető*
- Felhasználói dokumentáció
- Fejlesztői dokumentáció



Dokumentálás * felhasználói dokumentáció

E nélkül be sem adható!

Tartalma:

- **feladatszöveg** (összefoglaló és részletes is)
- futási környezet (szg.+or.+hw/sw-elvárások)
- használat leírása (*telepítés*, kérdések + lehetséges válaszok,...):
 - bemenő adatok, eredmények, *szolgáltatások*
 - mintaalkalmazások – példafutások
 - hibaüzenetek és a hibák lehetséges okai

Dőlten szedve, ami az aktuális komplex program estén a dokumentációból elhagyható.



Dokumentálás * fejlesztői dokumentáció

Dőlten szedve, ami az aktuális komplex program estén a dokumentációból elhagyható.

Tartalma:

- **feladatszöveg**, specifikáció, követelményanalízis
- fejlesztői környezet (op.rsz.+fordító program, ...)
- adateleírás (feladatparaméterek reprezentálása)
- algoritmusok leírása, döntések (pl. tételekre utalás), *más alternatívák, érvek*, magyarázatok
- kód, *implementációs szabványok*, ~ döntések
- próba (=tesztesetek)
- *hatékonysági mérések*
- fejlesztési lehetőségek
- **szerző(k)**

Összefoglalás



Összefoglalás

- Tesztelés
- Hibakeresés
- Hibajavítás
- Dokumentáció