Proiect TSS laborator

Cuciureanu Dragos-Adrian
Grupa 506

Table of Contents

Problema 3
Programul in java 3
Subiectul 1 4
Partitionarea in clase de echivalenta 4
Analiza valorilor de frontiera 5
Graf cauza efect 7
Tabele efecte 7
Tabel final 8
Subiectul 2 9
Subiectul 3 11
Programul numerotat: 11
Graful orientat rezultat: 12
MC/DC: 12
Subiectul 4 14
Mutant echivalent: 14
Concluzii 14
Subiectul 5 15
Mutant ne-echivalent omorat de test: 15
Concluzii 15
Mutant ne-echivalent omorat de test: 16
Concluzii 16

Problema

Se da un numar intreg prim n mai mare decat 100. Sa se verifice daca cifrele lui n sunt in ordine crescatoare (in ordine doar crescatoare, nu strict). Programul va intoarce un string:

- Daca n nu este in mai mare decat 100 sau nu este prim, se va intoarce „Numar invalid”
- Da/Nu daca respecta cerinta sau nu

Observatie: Programul implementat o sa rezolve o cerinta alternativa, dar mai usor de implementat, si anume faptul ca cifrele numarului n luate invers sunt in ordine descrescatoare.

Programul in java

Metoda „verificareCifreCrescatoare” rezolvarea problema de mai sus si se foloseste de functia ajutatoare „verificarePrim”.

```
public static String verificareCifreCrescatoare(int n){
    boolean ok = true;
    if (n < 100 || !verificarePrim(n)) {
        return "Numar invalid";
    }
    int cifraAnterioara = n % 10;
    n /= 10;
    while (n != 0 && ok) {
        int cifraCurenta = n % 10;
        if (cifraCurenta > cifraAnterioara)
            ok = false;
        cifraAnterioara = cifraCurenta;
        n /= 10;
    }
    if (ok)
        return "Da";
    else
        return "Nu";
}

static boolean verificarePrim(int n){
    if (n < 2)
        return false;
    for (int i = 2; i <= n - 1; i++) {
        if (n % i == 0){
            return false;
        }
    }
    return true;
}
```

Subiectul 1

Partitionarea in clase de echivalenta

Intrarile programului: n

Clasele intrarilor:

N1 = {n | n prim, n >= 100}
N2 = {n | n prim, n < 100}
N3 = {n | n nu e prim, n >= 100}
N4 = {n | n nu e prim, n < 100}

Clasele iesirilor:

I1 = “Da”
I2 = “Nu”
I3 = “Numar invalid”

Clase de echivalenta:

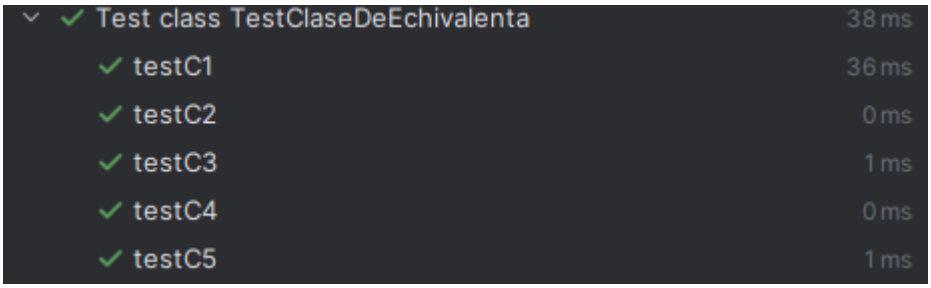
C1 = {n din N1, iesirea I1} -> 113
C2 = {n din N1, iesirea I2} -> 109
C3 = {n din N2, iesirea I3} -> 79
C4 = {n din N3, iesirea I3} -> 120
C5 = {n din N4, iesirea I3} -> 12

n	Output
113	“Da”
109	“Nu”
79	“Numar invalid”
120	“Numar invalid”
12	“Numar invalid”

Testele implementate:

```
public class TestClaseDeEchivalenta {
    @Test
    public void testC1() {
        // C1: n = 113, output = "Da"
        int n = 113;
        Assertions.assertEquals("Da", Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testC2() {
        // C2: n = 109, output = "Nu"
        int n = 109;
        Assertions.assertEquals("Nu", Main.verificareCifreCrescatoare(n));
    }
```

```
    @Test
    public void testC3() {
        // C3: n = 79, output = "Numar invalid"
        int n = 79;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testC4() {
        // C4: n = 120, output = "Numar invalid"
        int n = 120;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testC5() {
        // C5: n = 12, output = "Numar invalid"
        int n = 12;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }
}
```

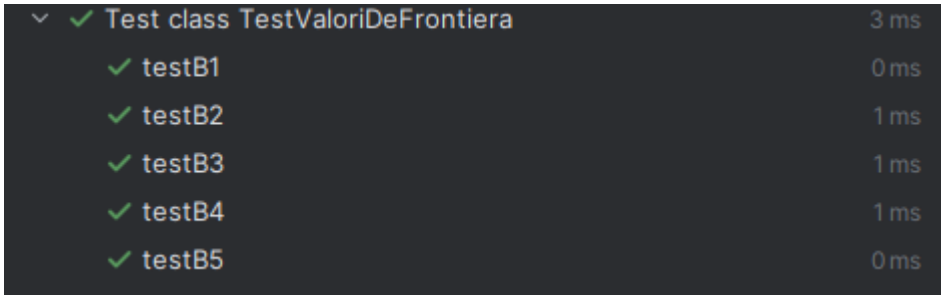


Analiza valorilor de frontiera
Intrari: n
Valori de frontiera: 99 si 100
Cazuri speciale: numere prime mai mici de 100 precum 2, 3, 5, 7, 11, 13... 97 etc., toate aceste cazuri vor rezulta in „Numar invalid”. Al doilea tip de caz special este primul numar care returneaza „Da” si anume 113 in cazul nostru si al treilea caz este 101, care returneaza „Nu”. Cazurile speciale nu sunt explicit valori de frontiera, dar sunt in vecinatatea aproapiata a acestora.

Teste	n	Output
B1	99	“Numar invalid”
B2	100	“Numar invalid”
B3	97	“Numar invalid”
B4	113	„Da”
B5	101	„Nu”

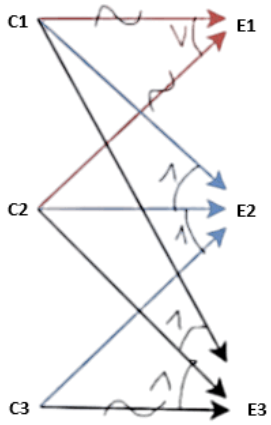
Testele implementate:

```
public class TestValoriDeFrontiera {
    @Test
    public void testB1() {
        // B1: n = 99, output = "Numar invalid"
        int n = 99;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testB2() {
        // C2: n = 100, output = "Numar invalid"
        int n = 100;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testB3() {
        // C3: n = 97, output = "Numar invalid"
        int n = 97;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testB4() {
        // C4: n = 113, output = "Da"
        int n = 113;
        Assertions.assertEquals("Da", Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testB5() {
        // C5: n = 101, output = "Nu"
        int n = 101;
        Assertions.assertEquals("Nu", Main.verificareCifreCrescatoare(n));
    }
}
```



Graf cauza efect

Cauze:
C1: n >= 100
C2: n prim
C3: n verifica enuntul, adica cifrele lui n sunt in ordine crescatoare



Efecte:
E1: „Numar invalid”
E2: „Da”
E3: „Nu”

Tabele efecte

E1: ~C1 V ~C2 echivalent cu ~(C1 ^ C2) => dam toate combinatiile lui C1 si C2 pentru a iesi 1, cum C3 nu ne intereseaza ii dam valoarea 0

C1	0	0	1
C2	0	1	0
C3	0	0	0
E1	1	1	1
E2	0	0	0
E3	0	0	0

E2: C1 ^ C2 ^ C3 => C1, C2 si C3 vor avea valoarea 1 ca sa dea E2 cu valoarea 1

C1	0	0	1	1
C2	0	1	0	1
C3	0	0	0	1
E1	1	1	1	0
E2	0	0	0	1
E3	0	0	0	0

E3: C1 ^ C2 ^ ~C3 => vom da valoarare 1 lui C1 si C2 si valoarea 0 lui C3 pentru a iesi E3 cu valoarea 1

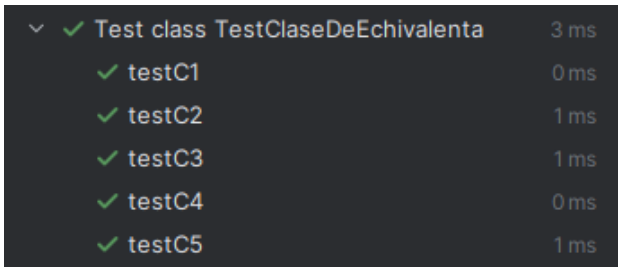
C1	0	0	1	1	1
C2	0	1	0	1	1
C3	0	0	0	1	0
E1	1	1	1	0	0
E2	0	0	0	1	0
E3	0	0	0	0	1

Tabel final

C1	0	0	1	1	1
C2	0	1	0	1	1
C3	0	0	0	1	0
E1	1	1	1	0	0
E2	0	0	0	1	0
E3	0	0	0	0	1
Input	65	73	150	239	151
Output	“Numar invalid”	“Numar invalid”	“Numar invalid”	„Da”	„Nu”

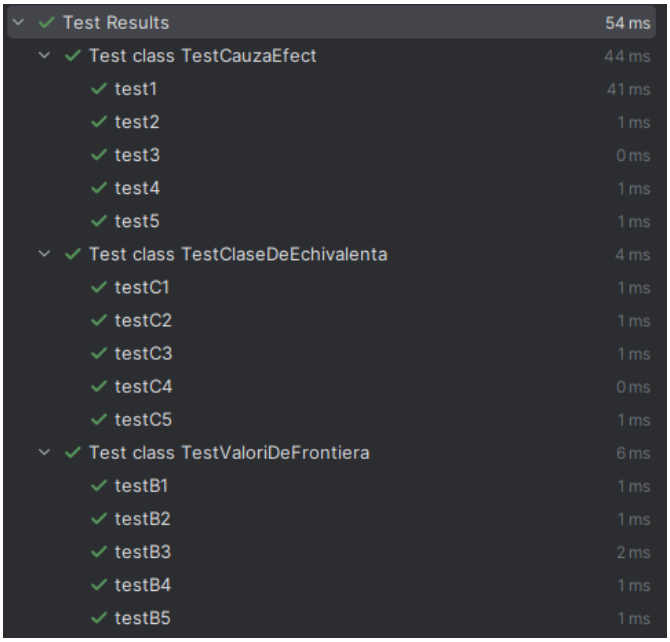
Testele implementate:

```
public class TestCauzaEfect {
    @Test
    public void test1() {
        // 1: n = 239, output = "Da"
        int n = 239;
        Assertions.assertEquals("Da", Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void test2() {
        // 2: n = 151, output = "Nu"
        int n = 151;
        Assertions.assertEquals("Nu", Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void test3() {
        // 3: n = 65, output = "Numar invalid"
        int n = 65;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void test4() {
        // 4: n = 73, output = "Numar invalid"
        int n = 73;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void test5() {
        // 5: n = 150, output = "Numar invalid"
        int n = 150;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }
}
```

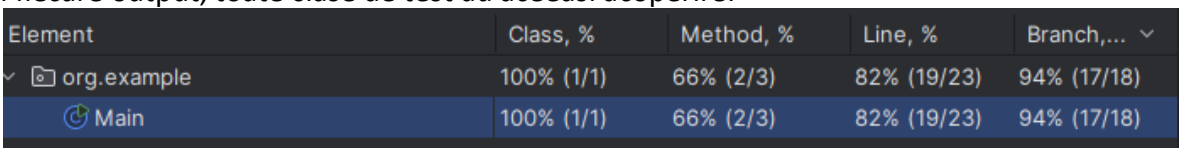


Subiectul 2

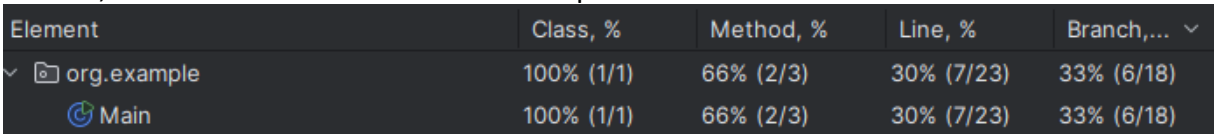
Acesta este un screenshot cu toate testele trecute:



Am realizat acoperirea cu JaCoCo. Cum in toate clasele de test am acoperit cate cel putin un caz din fiecare output, toate clase de test au aceeasi acoperire:



Singura valoare care ar fi diferita ar fi daca am scoate cazurile speciale din analiza valorilor de frontiera, si atunci am obtine urmatoarea acoperire:



Iar aici putem vedea ce a fost acoperit si ce nu a fost acoperit din program:

```
5       public static void main(String[] args) {
6           System.out.println(verificareCifreCrescatoare( n: 109));
7           System.out.println(verificareCifreCrescatoare( n: 113));
8           System.out.println(verificareCifreCrescatoare( n: 89));
9       }
10
11      public static String verificareCifreCrescatoare(int n)  18 usages
12      {
13          boolean ok = true;
14          if (n < 100 || !verificarePrim(n)) {
15              return "Numar invalid";
16          }
17          int cifraAnterioara = n % 10;
18          n /= 10;
19          while (n != 0 && ok) {
20              int cifraCurenta = n % 10;
21              if (cifraCurenta > cifraAnterioara)
22                  ok = false;
23              cifraAnterioara = cifraCurenta;
24              n /= 10;
25          }
26          if (ok)
27              return "Da";
28          else
29              return "Nu";
30      }
31
32      static boolean verificarePrim(int n)  1 usage
33      {
34          if (n < 2)
35              return false;
36          for (int i = 2; i <= n - 1; i++) {
37              if (n % i == 0)
38                  return false;
39          }
40          return true;
41      }
42  }
```

Vom lua in ordine componentele ce sunt reprezentate in code coverage. In primul rand la nivel de method singura metoda netestata este mainul, dupa cum se poate vedea si in imaginea de mai sus, in schimb ambele metode ce contribuie efectiv la rezultat au fost acoperite. La nivel de line ne lipsesc 4 linii, 3 dintre acestea sunt pe testele mele din main, iar cea de a 4 a este din metoda „verificarePrim”, linia respectiva nu va fi niciodata atinsa, deoarce daca un numar este mai mic decat 2, atunci metoda „verificareCifreCrescatoare” va returna direct „Numar invalid” pentru ca numarul este mai mic decat 100. Iar in ultimul caz, la nivel de branch este legat tot de linia 35, fiind cea tocmai explicata mai sus.
Astfel, putem considera ca am testat tot codul care contribuie la rezultatul final.

Subiectul 3

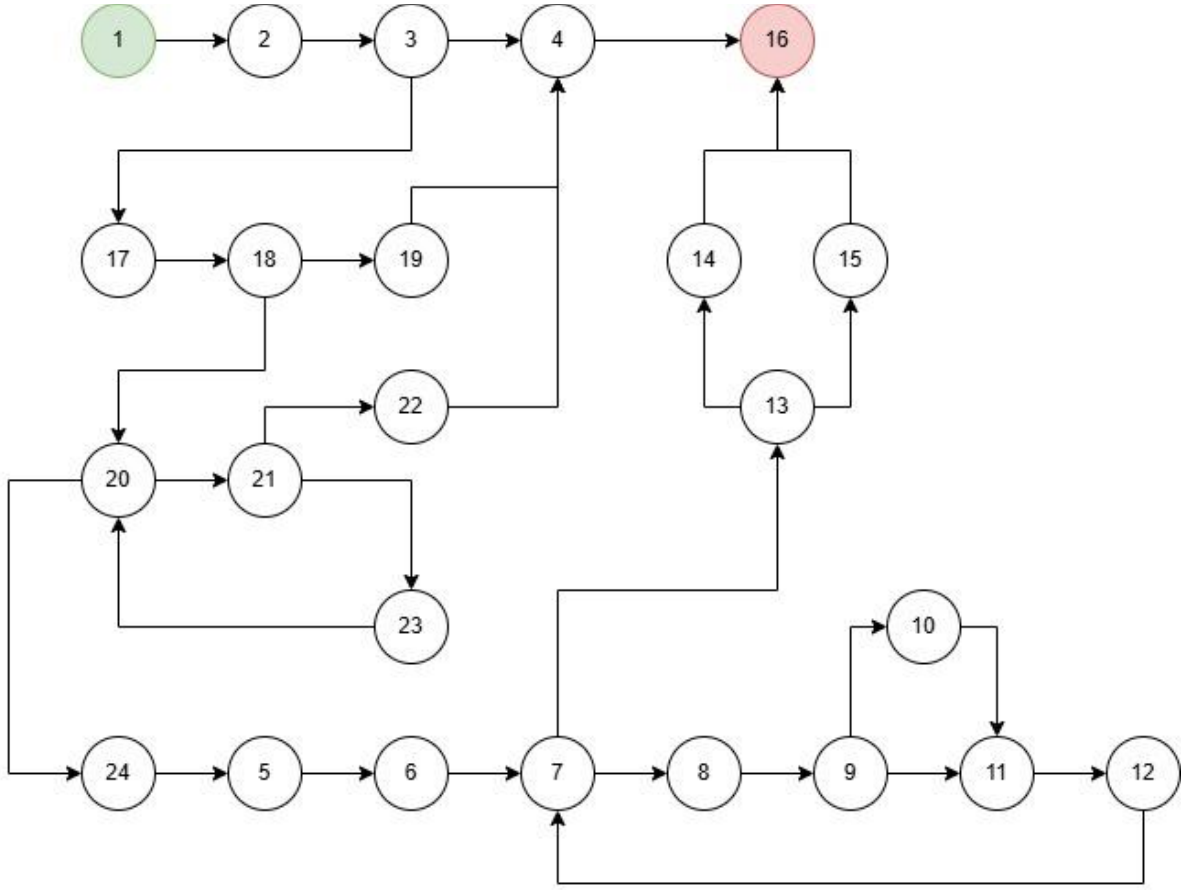
Programul numerotat:

```
1  public static String verificareCifreCrescatoare(int n){
2      boolean ok = true;
3      if (n < 100 || !verificarePrim(n)) {
4          return "Numar invalid";
       }
5      int cifraAnterioara = n % 10;
6      n /= 10;
7      while (n != 0 && ok) {
8          int cifraCurenta = n % 10;
9          if (cifraCurenta > cifraAnterioara)
10             ok = false;
11         cifraAnterioara = cifraCurenta;
12         n /= 10;
       }
13     if (ok)
14         return "Da";
       else
15         return "Nu";
16   }

17 static boolean verificarePrim(int n){
18     if (n < 2)
19         return false;
20     for (int i = 2; i <= n - 1; i++) {
21         if (n % i == 0){
22             return false;
23         }
       }
24     return true;
   }
```

Pe baza acestei numerotari ale liniilor vom realiza graful orientat al programului. Pentru urmatorul graf am marcat cu verde nodul de intrare (nodul 1) si cu rosu nodul de iesire (nodul 16).

Cum am mentionat si la subiectul anterior, instructiunea de la randul 19 (adica nodul 19) nu va fi niciodata atinsa.

Graful orientat rezultat:



MC/DC:
Am ales conditia: if (n < 100 || !verificarePrim(n))

Conditii individuale:
C1: n < 100
C2: verificarePrim(n)

Decizia: D: (n < 100 || ! verificarePrim(n)) <=> C1 V ~C2

Acoperire la nivel de instructiune: D = true, D = false

Acoperire la nivel de decizie: D = true, D = false
D = true: C1 = true, C2 = false
D = false: C1 = false, C2 = true

Acoperire la nivel de conditie: C1 = true/false si C2 = true/false
C1 = true si C2 = false => D = true
C1 = false si C2 = true => D = false

Acoperire la nivel de decizie/conditie: C1 = true/false si C2 = true/false, D = true/false
C1 = true si C2 = false => D = true
C1 = false si C2 = true => D = false

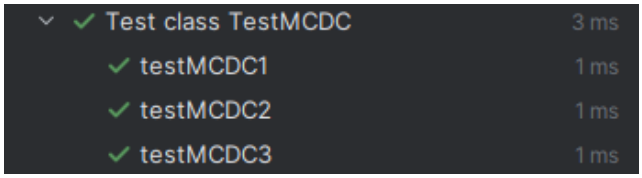
Acoperire la nivel de conditii multiple:
D = true: C1 = false, C2 = false
D = false: C1 = false, C2 = true
D = true: C1 = true, C2 = false
D = true: C1 = true, C2 = true

Acoperire la nivel de conditie/decizie modificata: fiecare conditie individuala sa fie atat true cat si false: C1 = true/false, C2 = true/false; fiecare decizie este true/false: D = true/false si fiecare conditie individuala influenteaza independent decizia.

D = true: C1 = false, C2 = false => 200 (va returna „Numar invalid”)
D = false: C1 = false, C2 = true => 179 (va returna „Da”)
D = true: C1 = true, C2 = true => 31 (va returna „Numar invalid”)

Testele implementate:

```
public class TestMCDC {
    @Test
    public void testMCDC1() {
        // 1: n = 200, output = "Numar invalid"
        int n = 200;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testMCDC2() {
        // 2: n = 179, output = "Da"
        int n = 179;
        Assertions.assertEquals("Da", Main.verificareCifreCrescatoare(n));
    }

    @Test
    public void testMCDC3() {
        // 3: n = 65, output = "Numar invalid"
        int n = 31;
        Assertions.assertEquals("Numar invalid",
Main.verificareCifreCrescatoare(n));
    }
}
```



Subiectul 4

Mutant echivalent:

```
public static String verificareCifreCrescatoare(int n) {
    boolean ok = true;
    if (n < 100 || !verificarePrim(n)) {
        return "Numar invalid";
    }
    int cifraAnterioara = n % 10;
    n /= 10;
    while (n != 0 && ok) {
        int cifraCurenta = n % 10;
        if (cifraCurenta > cifraAnterioara)
            ok = false;
        cifraAnterioara = cifraCurenta;
        n /= 10;
    }
    if (ok)
        return "Da";
    else
        return "Nu";
}

static boolean verificarePrim(int n) {
    if (n < 2)
        return false;
    // am modificat din n - 1 in radical de n
    for (int i = 2; i <= Math.sqrt(n); i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Concluzii

Am facut o singura modificare asupra programului, si aceea este de a schimba „n – 1” cu „Math.sqrt(n)” in interiorul for-ului care verifica daca un numar este prim sau nu. Aceasta schimbare face ca forul sa aiba mai putine iteratii in cazul unui numar prim, insa nu modifica niciun output al programului, fiind un mutant de ordin 1 echivalent.

Subiectul 5

Vom utiliza pentru ambii mutanti al treilea caz de la „Partitionarea in clase de echivalenta”, mai precis n = 79, care in mod normal ar trebui sa dea „Numar invalid”.

Mutant ne-echivalent omorat de test:

```
public static String verificareCifreCrescatoare(int n) {
    boolean ok = true;
    // am modificat valoarea pe care trebuie sa o depaseasca n din 100 in 60
    if (n < 60 || !verificarePrim(n)) {
        return "Numar invalid";
    }
    int cifraAnterioara = n % 10;
    n /= 10;
    while (n != 0 && ok) {
        int cifraCurenta = n % 10;
        if (cifraCurenta > cifraAnterioara)
            ok = false;
        cifraAnterioara = cifraCurenta;
        n /= 10;
    }
    if (ok)
        return "Da";
    else
        return "Nu";
}

static boolean verificarePrim(int n) {
    if (n < 2)
        return false;
    for (int i = 2; i <= n - 1; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Concluzii

Pentru modificare threshold-ului din 100 in 60, outputul programului se modifica pentru o serie de numere prime precum 61, 67, 89, cat si pentru valoarea testului nostru, adica 79, care trebuia sa dea „Numar invalid” si acum returneaza „Da”, astfel testul il omoara.

Mutant ne-echivalent neomorat de test:

```
public static String verificareCifreCrescatoare(int n) {
    boolean ok = true;
    // am modificat valoarea pe care trebuie sa o depaseasca n din 100 in 80
    if (n < 80 || !verificarePrim(n)) {
        return "Numar invalid";
    }
    int cifraAnterioara = n % 10;
    n /= 10;
    while (n != 0 && ok) {
        int cifraCurenta = n % 10;
        if (cifraCurenta > cifraAnterioara)
            ok = false;
        cifraAnterioara = cifraCurenta;
        n /= 10;
    }
    if (ok)
        return "Da";
    else
        return "Nu";
}

static boolean verificarePrim(int n) {
    if (n < 2)
        return false;
    for (int i = 2; i <= n - 1; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Concluzii

Pentru modificare threshold-ului din 100 in 80, outputul programului se modifica pentru o serie de numere prime precum 83, 89, 97, insa nu si pentru valoarea testului nostru, adica 79, care inca returneaza „Numar invalid”, astfel testul nu il omoara.