Proiect Testare Software

-Problema Examen-

Profesor:

Lect.dr. Sorina Predut

Student:

Naiboiu Teodor

Cuprins:

1. **Rezolvarea problemei “Examen”:**

Pentru rezolvarea acestei problemei si generarea testelor am folosit limbajul Python si modului unittest.

Am implementat o metoda care este destinata rezolvarii ecuatiilor problemei

```
def calculate(n=None, sums=None):
```

Am initializat by default doi parametrii cu None, pentru a putea apela metoda atat cu acesti parametrii trimisi ca argument cat si fara, urmand apoi sa verificam modul in care trimitem acesti parametrii si sa-i folosim in problema.

```
if n is None and sums is None:
    n, sums = define()
```

Observam ca daca acesti parametrii sunt None, adica functia a fost apelata fara parametrii, este nevoie sa ne definim cele doua variabile de input, anume “N”- numarul de studenti si “SUMS” - lista cu notele studentiilor dupa ce si-au calculat nota in functie de vecinul din stanga respectiv vecinul din dreapta.

```
def define():
    try:
        file_content = f.read()
        lines = file_content.strip().split("\n")
        nr = lines.pop(0)
        nr = int(nr)
        nested_sums = []
        g_sums = []
        for grade in lines:
            nested_sums.append(grade.split())
        for i in range(0, len(nested_sums[0])):
            g_sums.append(int(nested_sums[0][i]))
        return nr, g_sums
    except Exception as e:
        print("Error occurred parsing the file:", e)
        sys.exit(0)
```

In functia “define()” se citesc datele de intrare din fisier , N si sumele si apoi se initializeaza variabilele “NR” si ‘G_SUMS” cu numarul de studenti, respectiv sumele corespondente lor. Daca intampinam o eroare la parsarea si citirea din fisier, returnam un mesaj si oprim executia.
Verificam conditia din enunt ca “N” sa se afle intre 4 si 100.000

```
if not 4 <= n <= 100000:
   g.write(str(-1))
   return -1
```

Conform cerintei problemei daca orice conditie nu este indeplinita in fisier scriem valoarea “-1” si oprim executia problemei.

```
solutions = [-1] * n
```

Deoarece in Python nu putem manipula indexii unei liste fara ca acestia sa fie definiti, este nevoie sa definim o lista care sa contina pe toate pozitile “-1” si cu lungimea corespunzatoare numarului de studenti. In aceasta lista se vor returna notele initiale pentru studenti(studentului 1 ii corespunde indicele 0 din lista si tot asa).

```
try:# pentru a putea scrie None in teste
   if n != len(sums):
       g.write(str(-1))
       return -1
except TypeError:
   g.write(str(-1))
   return -1
```

Am continuat prin a verifica corectitudinea datelor de intrare, am testat daca numarul de note din fisierul de intrare corespunde cu numarul de studenti. Daca nu corespunde am scris in fisier mesajul specific pentru a reprezenta o solutie neadmisibila.

Am trecut apoi la rezolvarea algebrica a problemei

```
n_sum = 0
for s_grade in sums:
    n_sum += int(s_grade)
n_sum = n_sum // 2
s = n_sum
```

Am observat ca daca adunam toate sumele din fisierul de intrare si le impartim la 2, rezultatul este chiar suma notelor initiale. Am retinut aceasta valoare in variabila “N_SUM” si am copiat aceasta variabila in “S” pentru ca ne va folosi mai tarziu in rezolvare.

Stiind, din cerinta problemei, ca fiecare student s-a uitat in stanga si in dreapta si si-a trecut ca propriul rezultat suma rezultatelor vecinilor sai, am constatat ca se formeaza un sistem de N ecuatii. Am observat ca pentru a obtine ecuatii trebuie sa deducem metode de rezolvare astfel incat sa scadem din suma notelor initiale “N_SUM”, sumele deja stiute din ecutatii. Aici am observat ca, pentru un sistem cu solutie este nevoie ca numarul de ecuatii sa nu se divida cu 4, deoarece noi grupam cate 2 ecuatii, fiind nevoie sa ne ramana un numar de ecuatii libere intre 1 si 3. Asadar am dedus urmatoarele cazuri de rezolvare:

```
if n % 4 == 0:
    g.write(str(-1))
    return -1
```

In cazul in care numarul de ecuatii este divizibil cu 4 stim ca nu avem solutie unica, asadar scriem in fisier mesajul specific si iesim din executia programului.

Luam apoi pe rand fiecare caz in care restul impartirii lui N la 4 este, pe rand, 1, 2 si 3.

Pentru cazul in care N se imparte la 4 cu restul 1, stim ca avem posibile gruparile de cate 2 sume din care sa ne ramana o singura variabila fara grupare, pe care o aflam scazand din suma notelor initiale, sumele celorlalte variabile grupate.

```
if n % 4 == 1:
    for i in range(0, n-1):
        if i % 4 == 0 or i % 4 == 1:
            s -= sums[i]
    solutions[n - 1] = s
    i = n-1
    while i-3 >= 0:
        solutions[i - 3] = sums[i] - solutions[i]
        solutions[i - 1] = sums[i - 3] - solutions[i - 3]
        i -= 1
    for initial_grade in solutions:
        if not -1000000000 <= initial_grade <= 1000000000:
            g.write(str(-1))
            return -1
    solutions = rotate(solutions, 1)
    g.write(str(solutions))
    return solutions
```

Dupa ce aflam nota ultimului student prin aceasta metoda, scadem din suma acestuia nota initiala si observam din ecuatii ca obtinem suma initiala a studentului cu 3 pozitii mai departe.

Dupa aflarea tuturor notelor initiale verificam daca acestea se conformeaza conditiilor din enuntul problemei. Orice nota initiala trebuie sa se incadreze intre -1mld si 1mld, daca aceasta conditie nu este satisfacuta scriem in fisier rezultatul -1 si oprim executia, dar daca aceasta este satisfacuta rotim la stanga cu 1 pozitie lista de solutii, deoarece in formarea ei am rotit-o la dreapta, si scriem aceste solutii in fisier, return-ul ne ajuta sa testam.
Acest algoritm de rezolvare se aplica si le cele 2 resturi ramase necalculate cu mici variatii.

```
if n % 4 == 3:
    for i in range(0, n - 1):
        if (i % 4 == 0 or i % 4 == 1) and n-i >= 4:
            s -= sums[i]
        elif n-i < 4 and i % 4 == 0:
            s -= sums[i]
    solutions[n - 2] = s
    solutions[0] = sums[n - 2] - solutions[n - 2]
    for i in range(2, n, 2):
        solutions[i] = sums[i - 2] - solutions[i - 2]
    solutions[1] = sums[n - 1] - solutions[n - 1]
    for i in range(3, n, 2):
        solutions[i] = sums[i - 2] - solutions[i - 2]

    for initial_grade in solutions:
        if not -1000000000 <= initial_grade <= 1000000000:
            g.write(str(-1))
            return -1
    solutions = rotate(solutions, 1)
    g.write(str(solutions))
    return solutions
```

```
if n % 4 == 2:
    for i in range(0, n, 6):
        solutions[i + 5] = (sums[i + 5] + sums[i + 3] - sums[i + 1]) // 2
        solutions[i + 4] = (sums[i + 4] + sums[i + 2] - sums[i]) // 2
        solutions[i + 1] = sums[i + 5] - solutions[i + 5]
        solutions[i + 3] = sums[i + 1] - solutions[i + 1]
        solutions[i] = sums[i + 4] - solutions[i + 4]
        solutions[i + 2] = sums[i] - solutions[i]
    for initial_grade in solutions:
        if not -1000000000 <= initial_grade <= 1000000000:
            g.write(str(-1))
            return -1
    solutions = rotate(solutions, 1)
    g.write(str(solutions))
    return solutions
```

Acesta ultima conditie este finalul metodei “calculate()”.

Aceasta metoda este apelata in “main” fara niciun parametru:

```
if __name__ == "__main__":
    calculate()
```

2. Rezolvare cerinte proiect

2.1 Testare functionala

2.1.1 .Partitionare de echivalenta

Domeniul de intrari:

- N - numarul de studenti
- S - Suma notelor
- N trebuie sa fie intre 4 si 100.000 deci se disting 3 clase de echivalenta

N_1 : {4..100.000}

N_2 : {n | n<4}

N_3 : {n | n>100.000}

Observam ca lungimea listei de sumei trebuie sa corespunda cu numarul N, deci se disting inca 2 clase de echivalenta:

S_1 : {1 cand lungimea este egala cu N}

S_2 : {0 cand lungimea este diferita de N}

Domeniul de iesiri:

Consta din urmatoarele 2 raspunsuri:

- Notele initiale
- -1 - daca nu se admit solutii

I_1 : {note | cand ecutia este rezolvabila}

I_2 : {-1 | cand nu avem solutii}

Clasele de echivalenta:

I_111 : {(lista) | n este in N_1, lungimea listei de note este in S_1, note este in I_1}

I_112 : {(lista) | n este in N_1, lungimea listei de note este in S_1, note este in I_2}

I_122 : {(lista) | n este in N_1, lungimea listei de note este in S_2, note este in I_2}

I_2 : {(lista) | n este in N_2}

I_3 : {(lista) | n este in N_3}

```
def test_calculate(self):
    result = examen.calculate(5, [6, 13, 11, 10, 10])
    self.assertEqual(result, [4, 5, 9, 6, 1])
    result = examen.calculate(6, [6, 13, 11, 10, 11, 12])
    self.assertEqual(result, [8, 3, 5, 8, 4, 3])
    result = examen.calculate(7, [6, 13, 11, 10, 11, 12, 13])
    self.assertEqual(result, [11, 2, 2, 9, 8, 2, 4])
    result = examen.calculate(8, [6, 13, 11, 10, 11, 12, 13, 14])
    self.assertEqual(result, -1)
    result = examen.calculate(5, [1000000929, -10000006, 23123120,
232323111, -9282818220])
    self.assertEqual(result, -1)
    result = examen.calculate(5, [1000000929, 10000006, 23123120,
232323111, 928281822])
    self.assertEqual(result, -1)
    result = examen.calculate(5, [1, 2, 3, 4])
    self.assertEqual(result, -1)
    result = examen.calculate(1, None)
```

```
    self.assertEqual(result, -1)
    result = examen.calculate(100050, None)
    self.assertEqual(result, -1)
```

2.1.2 Analiza valorilor de frontiera

Am delimitat valorile 3,4,100.000, 100.001. Pentru aceasta problema s-a intamplat ca valorile de frontiera sa genereze solutii care nu sunt unice, asadar am generat teste si pentru valoarea de frontiera admisibila, 5 si 99.999 (nu se divid cu 4).

Deci testam pentru urmatoarele valori:
N_1 : 4,5,99.999.100.000
N_2 : 3
N_3 : 100.001

Observam ca avem valori de frontiera si pentru datele de iesire, acestea trebuie sa se incadreze intre -1mld si 1mld, asadar avem urmatoarele valori:
I_1 : -1.000.000.000, 1.000.000.000
I_2 : -1.000.000.001
I_2: 1.000.000.001
Am intampinat dificultati in a testa pentru frontiera superioara a lui N. Fiind nevoie de o lista admisibila cu 99.999 de note am dedus ca este un caz exceptional si nu poate fi testat in acest scop. Asadar am testat doar cazul 100.000 despre care stim ca este divizibil cu 4 deci ne asteptam sa primim rezultatul -1.

```
def test_calculate(self):
    result = examen.calculate(3, None)
    self.assertEqual(result, -1)
    result = examen.calculate(100001, None)
    self.assertEqual(result, -1)
    result = examen.calculate(4, None) # boundry admisibil dar ecuatie neadmisibila
    self.assertEqual(result, -1)
    result = examen.calculate(5, [5, 10, 12, 11, 11]) # boundry admisibil, ecuatie
admisibila
    self.assertEqual(result, [2, 2, 8, 9, 3])
    result = examen.calculate(4, [-5000, -100000, -5000, -100000]) # boundry admisibi,
```

```
ecuatie neadmisibila N11 + S11
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, -900010000, 10950000, 599990000, -999000000])
# boundry admisibi, ecuatie admisibila N11+S11 logic
    self.assertEqual(result, [-1000000000, 9950000, 99990000, 1000000, 500000000])
    result = examen.calculate(4, [-5000, 1999900000, -5000, 1999900000])  # boundry
admisibi, ecuatie neadmisibila N11 + S12
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, 1099990000, 10950000, 599990000, 1001000000])
# boundry admisibi, ecuatie admisibila N11 + S12
    self.assertEqual(result, [1000000000, 9950000, 99990000, 1000000, 500000000])
    result = examen.calculate(4, [-5000, -900010001, -5000, -900010001])  # boundry
admisibi, ecuatie neadmisibila N11 + S21
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, -900010001, 10950000, 599990000, -999000001])
# boundry admisibi, ecuatie admisibila N11 + S21
    self.assertEqual(result, -1)
    result = examen.calculate(4, [-5000, 1999900001, -5000, 1999900001])  # boundry
admisibi, ecuatie neadmisibila N11 + S22
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, 1099990001, 10950000, 599990000, 1001000001])
# boundry admisibi, ecuatie admisibila N11 + S22
    self.assertEqual(result, -1)
    result = examen.calculate(100000, [-5000, -100000, -5000, -100000])  # boundry
admisibi, ecuatie neadmisibila N12 + S11
    self.assertEqual(result, -1)
    result = examen.calculate(100000, [-5000, 1999900000, -5000, 1999900000])  # boundry
admisibi, ecuatie neadmisibila N12 + S12
    self.assertEqual(result, -1)
    result = examen.calculate(100000, [-5000, -900010001, -5000, -900010001])  # boundry
admisibi, ecuatie neadmisibila N12 + S21
    self.assertEqual(result, -1)
    result = examen.calculate(99999, [509950000, -900010001, 10950000, 599990000,
-999000001])  # boundry admisibi, ecuatie admisibila N12 + S21
    self.assertEqual(result, -1)
    result = examen.calculate(100000, [-5000, 1999900001, -5000, 1999900001])  # boundry
admisibi, ecuatie neadmisibila N12 + S22
    self.assertEqual(result, -1)
    result = examen.calculate(99999, [509950000, 1099990001, 10950000, 599990000,
1001000001])  # boundry admisibi, ecuatie admisibila N12 + S22
    self.assertEqual(result, -1)
```

2.1.3 Partitionare in categorii

Am descompus datele in urmatoarele categorii

- N
    1. {n | n<4}
    2. 4
    3. 5
    4. 5...99.999
    5. 99.999

6. 100.000
7. {n | n>100.000}

- Note
    1. { nota | lungimea sumelor este egala cu nr N}
    2. {nota | lungimea sumelor nu este egala cu N}

Am implementat astfel urmatoarele teste:

```
def test_calculate(self):
    result = examen.calculate(-6, None)# n<3
    self.assertEqual(result, -1)
    result = examen.calculate(3, None)# n=3
    self.assertEqual(result, -1)
    result = examen.calculate(4, None)  # n=4
    self.assertEqual(result, -1)
    result = examen.calculate(5, [6, 13, 11, 10, 11])  # n=5, len(s) = n
    self.assertEqual(result, [5, 4, 8, 6, 2])
    result = examen.calculate(5, [6, 13, 11, 10])  # n=5, len(s) != n
    self.assertEqual(result, -1)
    result = examen.calculate(99999, None)  # n=99999 Caz exceptional (nu
se poate testa) la fel si pentru n=100000
    self.assertEqual(result, -1)
    result = examen.calculate(100001, None)  # n=100001
    self.assertEqual(result, -1)
    result = examen.calculate(100005, None)  # n>100001
    self.assertEqual(result, -1)
```

2.2 Testare structurala

2.2.1 Acoperire la nivel de instructiune

Am transformat metoda intr-un graf orientat:

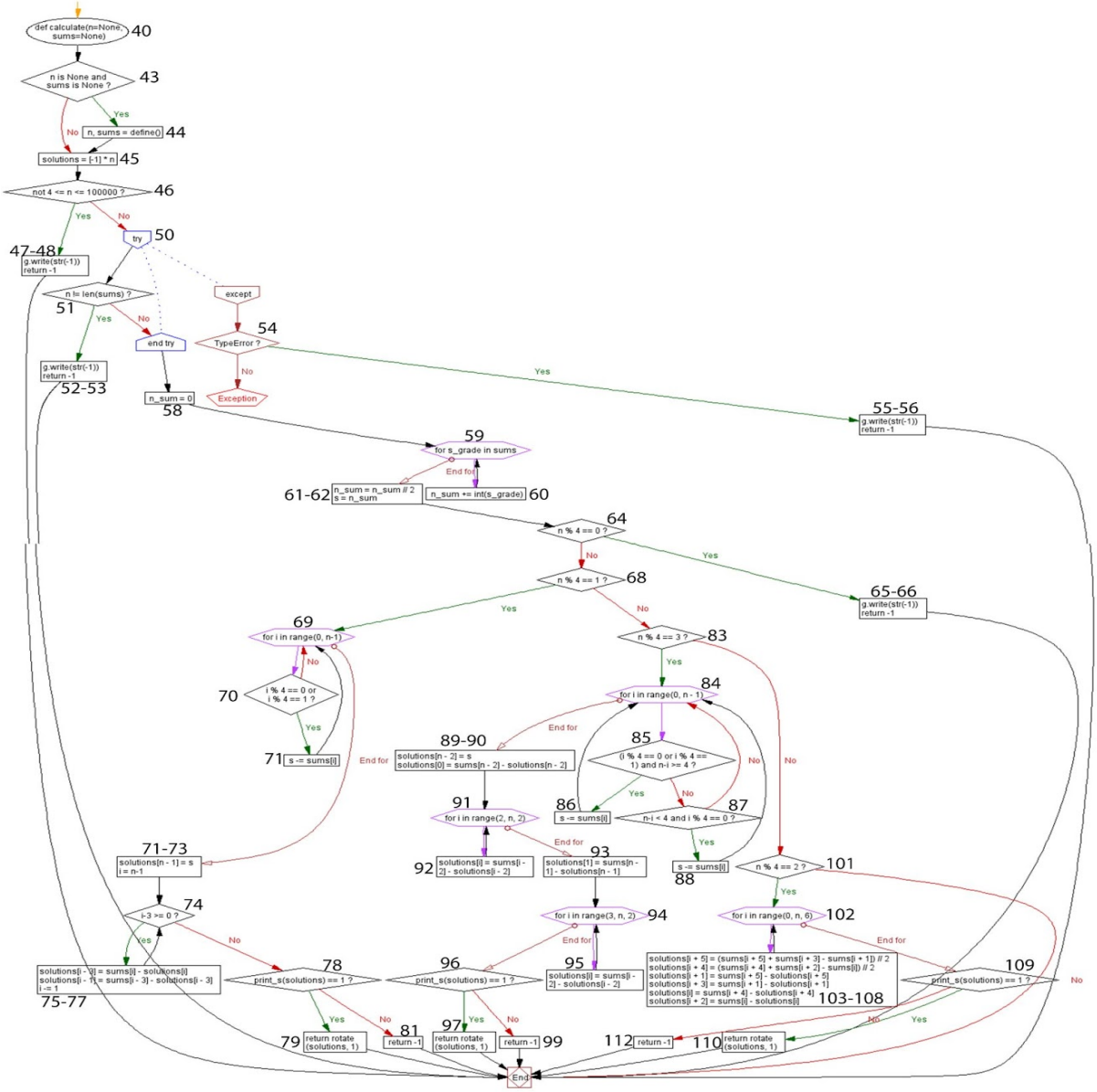

Fig 1. Graful metodei (numerotat) (zoom friendly)

Am generat apoi date de test pe acest graf orientat:

2.2.1 Acoperire la nivel de instructiune

```
# result = examen.calculate(None, None)result = examen.calculate(3, None)
self.assertEqual(result, -1)
result = examen.calculate(4, None)self.assertEqual(result, -1)
result = examen.calculate(5, [6, 13, 11, 10, 10])
self.assertEqual(result, [4, 5, 9, 6, 1])
result = examen.calculate(6, [6, 13, 11, 10, 11, 12])
self.assertEqual(result, [8, 3, 5, 8, 4, 3])
result = examen.calculate(7, [6, 13, 11, 10, 11, 12,
13])self.assertEqual(result, [11, 2, 2, 9, 8, 2, 4])
result = examen.calculate(8, [6, 13, 11, 10, 11, 12, 13, 14])
self.assertEqual(result, -1)
result = examen.calculate(5, [1, 2, 3, 4])
self.assertEqual(result, -1)
```

Testele de mai sus ating pe pe rand fiecare conditie

2.2.2 Acoperire la nivel de decizie

Deoarece metoda de rezolvare depinde foarte mult de input, cu testele precedente deja am testat fiecare decizie cu adevarat si fals, dar, totusi, ca sa aiba sens testarea am implemenetat urmatoarele teste care verifica acest lucru inca odata.

```
def test_calculate(self):
    result = examen.calculate(4, None)
    self.assertEqual(result, -1)
    result = examen.calculate(8, None)
    self.assertEqual(result, -1)
    result = examen.calculate(2, None)
    self.assertEqual(result, -1)
    result = examen.calculate(5, [6, 13, 11, 10, 10])
    self.assertEqual(result, [4, 5, 9, 6, 1])
    result = examen.calculate(4, [6, 13, 11, 10, 10])
    self.assertEqual(result, -1)
```

2.2.3 Acoperire la nivel de conditie

Am generat date de test astfel incat fiecare conditie sa ia pe rand valoarea adevarat si valoarea fals.

```
def test_calculate(self):
    result = examen.calculate(4, None)
    self.assertEqual(result, -1)
    result = examen.calculate(8, None)
    self.assertEqual(result, -1)
    result = examen.calculate(2, None)
    self.assertEqual(result, -1)
    result = examen.calculate(5, [6, 13, 11, 10, 10])
    self.assertEqual(result, [4, 5, 9, 6, 1])
    result = examen.calculate(4, [6, 13, 11, 10, 10])
    self.assertEqual(result, -1)
    result = examen.calculate(100001, None)
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, 1099990000, 10950000,
599990000, 1001000000])  # boundry admisibi, ecuatie admisibila N11 + S12
    self.assertEqual(result, [1000000000, 9950000, 99990000, 1000000,
500000000])
```

2.3 Complexitatea metodei. McCabe. Numarul de circuite independente

V(G) = e - n + 2p

e = numărul de muchii ale graficului.
n = numărul de noduri ale graficului.
p = numărul de componente conectate.

In cazul nostru avem o singura metoda asadar formula se simplifica astfel:
V(G) = e − n + 2.

In cazul nostru o sa consideram instructiunile de tip try-except ca o instructiune cu 2 noduri, ignorand nodurile si muchile “end-try” “raise exception”
Adaugam o muchie de la “END” catre primul nod pentru a avea un graf complet.
Avem:
n=49

e=70

V(G) = 21

Am scris doar cateva circuite independente.

Circuite independente:

a) 40, 43,45,46,47-48
b) 40, 43, 44, 45, 46, 47-48
c) 40, 43, 44, 45, 46, 50, 51, 52-53
d) 40, 43, 44, 45, 46, 50, 54, 55-56
e) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 65-66
f) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 69, 70, 71, 72-73, 74, 78, 79
g) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 69, 70, 71, 72-73, 74, 78, 81
h) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 69, 70, 71, 72-73, 74, 75-77, 78, 79
i) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 69, 70, 71, 72-73, 74, 75-77, 78, 81
j) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 83, 84, 85, 86, 89-90, 91, 92, 93, 94, 95, 96, 97
k) 40, 43, 44, 45, 46, 50, 51, 58, 59, 60, 61-62, 64 , 68, 83, 84, 85, 86, 89-90, 91, 92, 93, 94, 95, 96, 99

2.3.1 Teste acoperire circuit independent

Circuitele independente sunt testate prin introducerea unor date de test care acopera toate cazurile care pot exista in rezolvarea problemei, astfel stim ca este parcursa fiecare instructiune.

```
def test_calculate(self):
    # result = examen.calculate(None, None)    # self.assertEqual(result,
[11, 2, 2, 9, 8, 2, 4])
    result = examen.calculate(3, None)
    self.assertEqual(result, -1)
    result = examen.calculate(4, [6, 13, 11, 10, 10])
    self.assertEqual(result, -1)
    result = examen.calculate(5, [6, 13, 11, 10, 10])
    self.assertEqual(result, [4, 5, 9, 6, 1])
    result = examen.calculate(6, [6, 13, 11, 10, 11, 12])
    self.assertEqual(result, [8, 3, 5, 8, 4, 3])
    result = examen.calculate(7, [6, 13, 11, 10, 11, 12, 13])
    self.assertEqual(result, [11, 2, 2, 9, 8, 2, 4])
    result = examen.calculate(4, [1, 2, 3, 4])
```

```
    self.assertEqual(result, -1)
    result = examen.calculate(5, [509950000, -900010001, 10950000,
599990000, -999000001])
    self.assertEqual(result, -1)
```

2.4 Expresia regulata a grafului. Numarul de cai

Dat fiind numarul foarte ridicat de cai existente in graful nostru, am partitionat acest numar in clase de echivalenta.

Am calculat numarul de cai pentru fiecare caz in care problema are un numar de elevi divizibil cu 4, nedivizibil, cu restul 1, cu restul 2 si intr-un final cu restul 3.

Am notat ultimul nod (“END”) cu 113

Cazul in care prin impartirea la 4 obtinem rest 0:

40.43.44.45.46.50.1.58.(59.60)*.61+62.64.65+66.113

Cazul in care prin impartirea la 4 obtinem rest 1:

40.43.44.45.46.50.1.58.(59.60)*.61+62.64.68.(69+70.71)*.71.71+73.(74+77)*.78.79.113

Cazul in care prin impartirea la 4 obtinem rest 2:

40.43.44.45.46.50.1.58.(59.60)*.61+62.64.68.83.101.(102+108)*.109.110.113

Cazul in care prin impartirea la 4 obtinem rest 3:

40.43.44.45.46.50.1.58.(59.60)*.61+62.64.68.83.(84+85.86)*.(84+85.87)*.89+90.(91+92)*.93.(94+95)*.86.97.113

Expresia regulata a grafului este:

40.43.44.(43+null).45.(46+null).47+48.113..(50+null).54+56.113.51+53.113.(51+null).58.59.(60.59).61+62.64…

Numar cai:...

2.5 Generarea mutantilor

2.5.1 Rezultatul generarii mutantilor

Pentru generarea mutantilor am folosit modulul *mutmut*.

La o prima generare de mutanti, asupra testelor scrise anterior, observam urmatorul rezultat:

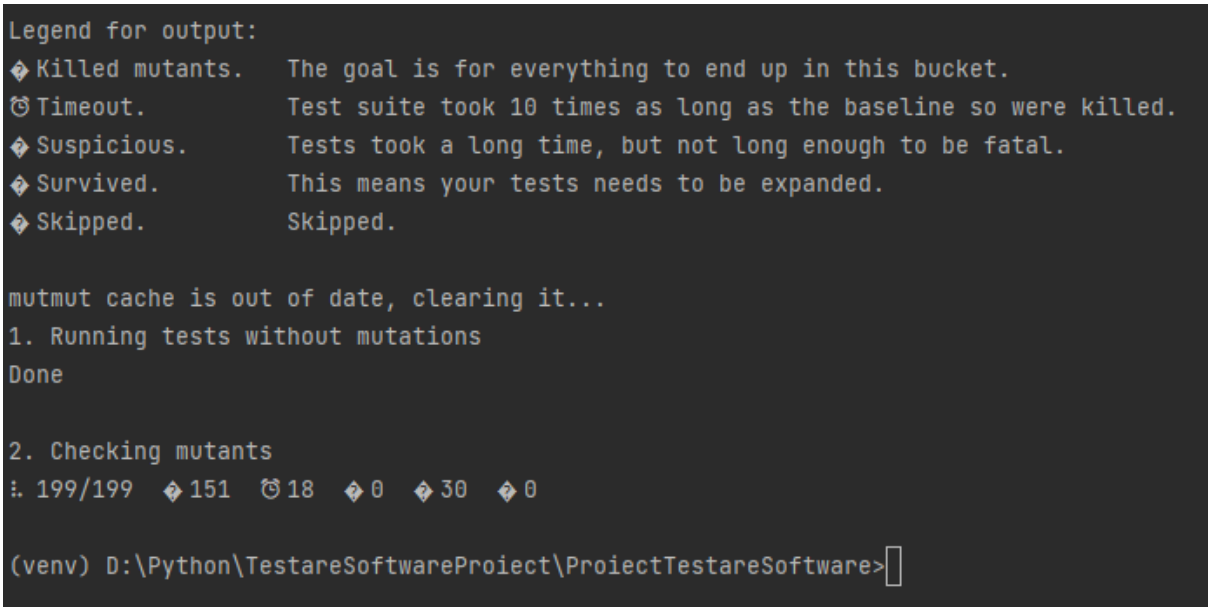

Fig 2. Rezultatul generarii mutantilor

Din legenda modulului de generare observam ca avem 151 de mutanti omorati, 30 de mutanti ramasi in viata si 18 mutanti care au depasit timpul limita de executie pe care ii omoram.

2.5.1 Generarea testelor suplimentare. Omorarea mutantilor.

Alegem la intamplare 2 mutanti dintre cei ramasi in viata si culegem informatii despre rezultatul acestora.

Observam ca mutantul generat presupune ca N si ‘sums’ raman *None*, caz care nu este acoperit de teste, asadar generam un nou test, care sa acopere si acest caz.

```
(venv) D:\Python\TestareSoftwareProiect\ProiectTestareSoftware>mutmut show 4
--- examen.py
+++ examen.py
@@ -42,7 +42,7 @@
     # g.truncate() daca vrem sa stergem continutul din fisier la fiecare test
     try:
         if n is None and sums is None:
-            n, sums = define()
+            n, sums = None
         solutions = [-1] * n
         if not 4 <= n <= 100000:
             g.write(str(-1))# pragma: no mutate
```

Fig.3 Mutantul cu ID 4

Stim ca in cazul in care ambele variabile sunt *None* inseamna ca apelul este facut in mod direct. Asta inseamna ca se vor citi datele din fisier si se vor genera solutii conform acelor date, deci pentru acest caz, trebuie sa generam un test care sa corespunda cu datele din fisier.

```
def test_calculate(self):
    result = examen.calculate(None, None)
    self.assertEqual(result, [2, 2, 8, 9, 3])
```

La o noua testare observam ca mutantul cu ID 4 nu mai exista printre mutantii in viata.
O alta observatie este ca am reusit sa omoram 2 mutanti cu 1 singur test.

```
Legend for output:
🎉 Killed mutants.   The goal is for everything to end up in this bucket.
⏰ Timeout.          Test suite took 10 times as long as the baseline so were killed.
🤔 Suspicious.       Tests took a long time, but not long enough to be fatal.
🙁 Survived.         This means your tests needs to be expanded.
🔇 Skipped.          Skipped.

1. Using cached time for baseline tests, to run baseline again delete the cache file

2. Checking mutants
⠸ 199/199  🎉 153  ⏰ 18  🤔 0  🙁 28  🔇 0
```

Fig.4 Mutantii dupa noul test

Daca ne uitam la vechea generare, fara noul test implmenentat observam ca pe langa mutantul cu ID 4 mai exista un mutant cu ID 2 asemanator:

```
(venv) D:\Python\TestareSoftwareProiect\ProiectTestareSoftware>mutmut show 2
--- examen.py
+++ examen.py
@@ -41,7 +41,7 @@
     # g.seek(0)
     # g.truncate() daca vrem sa stergem continutul din fisier la fiecare test
     try:
-        if n is None and sums is None:
+        if n is None and sums is not None:
             n, sums = define()
         solutions = [-1] * n
         if not 4 <= n <= 100000:
```

Fig 5. Mutantul cu ID 2

Acest mutant a fost omorat de noul test implementat odata cu mutantul cu ID 4.