

Proiect Laborator - Testare si Verificare

Student: Grecu Elena-Alexandra

Master: Inginerie Software (506)

Curs: Testare si Verificare

1. Problema aleasa



Problema aceasta a fost selectata de pe www.infoarena.ro si poate fi gasita [aici](#).

Timbre

Dupa cum stiti cu totii, Adriana este o mare colectionara de timbre. In fiecare zi se duce la magazinul de pe strada ei pentru a-si mari colectia. Intr-o zi, vanzatorul (nimeni altul decat Balaurul Arhirel) s-a gandit sa-i faca o surpriza. A scos dintr-un dulap vechi niste timbre foarte valoroase pe care erau scrise cu fir de aur si de argint numere naturale. Stiind ca fetita nu are bani prea multi, Balaurul i-a spus urmatoarele: "Eu pot sa impart timbrele in M intervale de forma $[1, \dots, m_i]$. Tu poti sa iei din orice interval o singura subsecventa de maxim K elemente. Desigur, daca ai ales o subsecventa din intervalul i vei plati o anumita suma..."

Adriana s-a gandit ca ar fi frumos sa-si numereze toate cele N pagini ale clasorului ei cu astfel de timbre. Fiind si o fetita pofticioasa si-a zis: "Tare as vrea sa mananc o inghetata din banii pe care ii am la mine, dar nu stiu daca o sa-mi ajunga sa platesc timbrele. Cum sa fac?"

Cerinta

Stiind cele M intervale, precum si costurile acestora, ajutati-o pe Adriana sa cumpere timbrele necesare numerotarii clasorului, platind o suma cat mai mica.

Date de intrare

Pe prima linie a fisierului timbre.in se afla N, M, si K. N reprezinta numarul de pagini ale clasorului, M reprezinta numarul de intervale, iar K lungimea maxima a unei subsecvente. Pe urmatoarele M linii se afla doua numere separate printr-un spatiu, m_i si c_i , unde m_i reprezinta marginea superioara a intervalului i, iar c_i costul acestuia.

Date de iesire

Pe prima linie a fisierului timbre.out se va afla Smin, reprezentand suma minima pe care trebuie sa o plateasca Adriana pentru a cumpara timbrele necesare numerotarii clasorului.

Restrictii si precizari

- $0 < N < 1\ 001$
- $0 < M < 10\ 001$
- $0 < K < 1\ 001$
- $0 < m_i < 100\ 000$
- $0 < c_i < 10\ 000$
- pentru a numera toate cele pagini ale clasorului, Adriana are nevoie de timbre cu numerele de la 1 la N

Exemplu

timbre.in	timbre.out
-----------	------------

4 3 2 5 3 2 1 6 2	3
-------------------	---

2. Solutia problemei



Rezolvarea acestui proiect este realizata in Python.

main.py

```
def min_cost_to_buy_stamps(input_data):
    '''Returns the minimum cost for buying stamps'''
    N, M, K, intervals = input_data

    # check inputs
    if not (1 <= N <= 1000):
        return "N (Numarul de pagini) trebuie sa fie intre 1 si 1000"

    if not (1 <= M <= 10000):
        return "M (Numarul de intervale) trebuie sa fie intre 1 si 10000"

    if not (1 <= K <= 1000):
        return "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000"

    if len(intervals) != M:
        return "Numarul de perechi (mi, ci) trebuie sa fie M"

    for i, (mi, ci) in enumerate(intervals):
        if not (1 <= mi <= 100000):
            return "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)"

        if not (1 <= ci <= 10000):
            return "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)"

    # sort the given intervals by cost
    intervals.sort(key=lambda x: x[1])

    # create a set to store the costs
    cost = set()
    i = M - 1
    c = 0

    # loop until all pages have stamps
    while N > 0:
        # add the cost of the interval if it has enough stamps for the remaining pages
        while i >= 0 and intervals[i][0] >= N:
            cost.add(intervals[i][1])
            i -= 1

        # check if the cost set is not empty before finding the minimum value
        if cost:
            # add the minimum cost to the total cost and remove it from the set
            c += min(cost)
            cost.remove(min(cost))

        # decrease the number of pages by the maximum length of a subsequence
        N -= K

    return c
```

- Am creat o functie `min_cost_to_buy_stamps(input_data)` care are ca scop gasirea costului minim al timbrelor pe care Adriana le poate cumpara pentru a-si numerota clasorul. Datele de intrare au fost validate, am sortat crescator intervalele cu timbre dupa cost si am iterat prin aceste intervale ca sa gasim costul minim.

3. Cerinta 1

Pe baza cerintelor programului, sa se genereze date de test folosind:

- a) *equivalence partitioning*
- b) *boundary value analysis*

c) cause-effect graphing

3. a) Equivalence partitioning

Vom utiliza constrangerile mentionate in enuntul problemei pentru a imparti domeniul de intrare pentru fiecare variabila:

```
Clasele intrarilor:
- N (Numarul de pagini ale clasorului)  $0 < N < 1001$ 
- M (Numarul de intervale de timbre)  $0 < M < 10001$ 
- K (Lungimea maxima a unei subsecvente de timbre care poate fi luata din interval)
   $0 < K < 1001$ 
- (mi, ci)  $i \leq M$ :
  - mi (Marginea superioara a intervalului i)  $0 < mi < 100\ 000$ ,  $i \leq M$ 
  - ci (Costul intervalului i)  $0 < ci < 10\ 000$ ,  $i \leq M$ 

1) N:
N_1 = {N |  $0 < N < 1001$ } # N valid
N_2 = {N |  $N < 1$ } # N invalid
N_3 = {N |  $N > 1001$ } # N invalid
2) M:
M_1 = {M |  $0 < M < 10001$ } # M valid
M_2 = {M |  $M < 1$ } # M invalid
M_3 = {M |  $M > 10001$ } # M invalid
3) K:
K_1 = {K |  $0 < K < 1001$ } # K valid
K_2 = {K |  $K < 1$ } # K invalid
K_3 = {K |  $K > 10001$ } # K invalid
4) mi:
mi_1 = {m[i] |  $0 < m[i] < 100\ 000$ ,  $0 < i < M$ } # mi valid
mi_2 = {m[i] |  $m[i] < 0$ ,  $0 < i < M$ } # mi invalid
mi_3 = {m[i] |  $m[i] > 100\ 000$ ,  $0 < i < M$ } # mi invalid
mi_4 = {m[i] |  $0 < m[i] < 100\ 000$ ,  $\text{count}((m[i], c[i])) \neq M$ } # numarul total
de perechi (mi, ci) este invalid
5) ci:
ci_1 = {c[i] |  $0 < c[i] < 10\ 000$ ,  $0 < i \leq M$ } # ci valid
ci_2 = {c[i] |  $c[i] < 0$ ,  $0 < i \leq M$ } # ci invalid
ci_3 = {c[i] |  $c[i] > 10\ 000$ ,  $0 < i \leq M$ } # ci invalid
ci_4 = {c[i] |  $0 < c[i] < 10\ 000$ ,  $\text{count}((m[i], c[i])) \neq M$ } # numarul total de perechi
(mi, ci) este invalid
-----
Clase iesiri:
i_1 = Solutia dorita (toate datele de intrare sunt valide)
i_2 = "N (Numarul de pagini) trebuie sa fie intre 1 si 1000"
i_3 = "M (Numarul de intervale) trebuie sa fie intre 1 si 10000"
i_4 = "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000"
i_5 = "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)"
i_6 = "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)"
i_7 = "Numarul de perechi (mi, ci) trebuie sa fie M"
-----
Clase de echivalenta finala:
C_1 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_1, ci in ci_1 si i_1}
C_2 = {N, M, K, (mi, ci) | N in N_2, M in M_1, K in K_1, mi in mi_1, ci in ci_1 si i_2}
C_3 = {N, M, K, (mi, ci) | N in N_3, M in M_1, K in K_1, mi in mi_1, ci in ci_1 si i_2}
C_4 = {N, M, K, (mi, ci) | N in N_1, M in M_2, K in K_1, mi in mi_1, ci in ci_1 si i_3}
C_5 = {N, M, K, (mi, ci) | N in N_1, M in M_3, K in K_1, mi in mi_1, ci in ci_1 si i_3}
C_6 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_2, mi in mi_1, ci in ci_1 si i_4}
C_7 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_3, mi in mi_1, ci in ci_1 si i_4}
C_8 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_2, ci in ci_1 si i_5}
C_9 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_3, ci in ci_1 si i_5}
C_10 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_1, ci in ci_2 si i_6}
C_11 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_1, ci in ci_3 si i_6}
C_12 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_4, ci in ci_4 si i_7}
```

Pentru fiecare clasa am creat cate un test, acestea fiind:

`equivalence_partitioning.py`

```
class TestStampMinCost(unittest.TestCase):
    def test_C_1(self):
        # input bun
        N, M, K, intervals, expected = (5, 3, 2, [(3, 1), (6, 2), (8, 3)], 6)
        result = min_cost_to_buy_stamps(N, M, K, intervals)
        self.assertEqual(result, expected)
```

```

def test_C_2(self):
    # N < 1
    N, M, K, intervals, expected = (
        0, 3, 2, [(3, 1), (6, 2), (8, 3)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_3(self):
    # N > 1000
    N, M, K, intervals, expected = (
        1001, 3, 2, [(3, 1), (6, 2), (8, 3)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_4(self):
    # M < 1
    N, M, K, intervals, expected = (
        5, 0, 2, [(3, 1), (6, 2), (8, 3)], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_5(self):
    # M > 10000
    N, M, K, intervals, expected = (
        5, 10001, 2, [(i, i) for i in range(1, 10002)], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_6(self):
    # K < 1
    N, M, K, intervals, expected = (
        5, 3, 0, [(3, 1), (6, 2), (8, 3)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_7(self):
    # K > 1000
    N, M, K, intervals, expected = (
        5, 3, 1001, [(3, 1), (6, 2), (8, 3)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_8(self):
    # mi < 1
    N, M, K, intervals, expected = (
        5, 3, 2, [(0, 1), (6, 2), (8, 3)], "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_9(self):
    # mi > 100 000
    N, M, K, intervals, expected = (5, 3, 2, [(100001, 1), (6, 2), (8, 3)],
        "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_10(self):
    # ci < 1
    N, M, K, intervals, expected = (
        5, 3, 2, [(3, 0), (6, 2), (8, 3)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_11(self):
    # ci > 10 000
    N, M, K, intervals, expected = (
        5, 3, 2, [(3, 10001), (6, 2), (8, 3)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

def test_C_12(self):
    # numarul de perechi (mi, ci) nu este M
    N, M, K, intervals, expected = (5, 3, 2, [(3, 1), (6, 2)], "Numarul de perechi (mi, ci) trebuie sa fie M")
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    self.assertEqual(result, expected)

if __name__ == "__main__":
    unittest.main()

```

3. b) Boundary Value Analysis

Pentru a crea teste pentru Boundary Value Analysis, ne-am folosit de constrangerile pentru fiecare variabila a problemei. Pentru fiecare variabila am ales valorile minime si maxime valide, valori valide aproape de minimul si maximul intervalului precum si valorile maxime si minime invalide (cele mai mici si mari valori care nu se gasesc in intervalul de valori valide).

```
Clasele intrarilor:

1) N:
N_1 = 1 # valoarea minima valida
N_2 = 1000 # valoarea maxima valida
N_3 = 2 # valoarea valida aproape de minimul intervalului
N_4 = 999 # valoarea valida aproape de maximul intervalului
N_5 = 0 # valoarea maxima invalida de langa limita inferioara a intervalului de valori valide
N_6 = 1001 # valoarea minima invalida de langa limita superioara a intervalului de valori valide

2) M:
M_1 = 1 # valoarea minima valida
M_2 = 10 000 # valoarea maxima valida
M_3 = 2 # valoarea valida aproape de minimul intervalului
M_4 = 9999 # valoarea valida aproape de maximul intervalului
M_5 = 0 # valoarea maxima invalida de langa limita inferioara a intervalului de valori valide
M_6 = 10 001 # valoarea minima invalida de langa limita superioara a intervalului de valori valide

3) K:
K_1 = 1 # valoarea minima valida
K_2 = 1000 # valoarea maxima valida
K_3 = 2 # valoarea valida aproape de minimul intervalului
K_4 = 999 # valoarea valida aproape de maximul intervalului
K_5 = 0 # valoarea maxima invalida de langa limita inferioara a intervalului de valori valide
K_6 = 10 001 # valoarea minima invalida de langa limita superioara a intervalului de valori valide

4) mi:
mi_1 = 1 # valoarea minima valida
mi_2 = 99 999 # valoarea maxima valida
mi_3 = 2 # valoarea valida aproape de minimul intervalului
mi_4 = 99 998 # valoarea valida aproape de maximul intervalului
mi_5 = 0 # valoarea maxima invalida de langa limita inferioara a intervalului de valori valide
mi_6 = 100 001 # valoarea minima invalida de langa limita superioara a intervalului de valori valide

5) ci:
ci_1 = 1 # valoarea minima valida
ci_2 = 9 999 # valoarea maxima valida
ci_3 = 2 # valoarea valida aproape de minimul intervalului
ci_4 = 9 998 # valoarea valida aproape de maximul intervalului
ci_5 = 0 # valoarea maxima invalida de langa limita inferioara a intervalului de valori valide
ci_6 = 10 001 # valoarea minima invalida de langa limita superioara a intervalului de valori valide

-----

Clase iesiri:
i_1 = Numar - Solutia dorita (toate datele de intrare sunt valide)
i_2 = "N (Numarul de pagini) trebuie sa fie intre 1 si 1000"
i_3 = "M (Numarul de intervale) trebuie sa fie intre 1 si 10000"
i_4 = "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000"
i_5 = "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 10000)"
i_6 = "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)"

-----

Clase de echivalenta finala:
C_1 = {N, M, K, (mi, ci) | N in N_1, M in M_1, K in K_1, mi in mi_1, ci in ci_1 si iesirea i_1}
C_2 = {N, M, K, (mi, ci) | N in N_2, M in M_2, K in K_2, mi in mi_2, ci in ci_2 si iesirea i_1}
C_3 = {N, M, K, (mi, ci) | N in N_3, M in M_3, K in K_3, mi in mi_3, ci in ci_3 si iesirea i_1}
C_4 = {N, M, K, (mi, ci) | N in N_4, M in M_4, K in K_4, mi in mi_4, ci in ci_4 si iesirea i_1}
C_5 = {N, M, K, (mi, ci) | N in N_5, M, K, mi, ci - toate valide si iesirea i_2}
C_6 = {N, M, K, (mi, ci) | N in N_6, M, K, mi, ci - toate valide si iesirea i_2}
C_7 = {N, M, K, (mi, ci) | M in M_5, N, K, mi, ci - toate valide si iesirea i_3}
C_8 = {N, M, K, (mi, ci) | M in M_6, N, K, mi, ci - toate valide si iesirea i_3}
C_9 = {N, M, K, (mi, ci) | K in M_5, N, M, mi, ci - toate valide si iesirea i_4}
C_10 = {N, M, K, (mi, ci) | K in M_6, N, M, mi, ci - toate valide si iesirea i_4}
C_11 = {N, M, K, (mi, ci) | mi in mi_5, N, M, K, ci - toate valide si iesirea i_5}
C_12 = {N, M, K, (mi, ci) | mi in mi_6, N, M, K, ci - toate valide si iesirea i_5}
C_13 = {N, M, K, (mi, ci) | ci in ci_5, N, M, K, mi - toate valide si iesirea i_6}
C_14 = {N, M, K, (mi, ci) | ci in ci_6, N, M, K, mi - toate valide si iesirea i_6}
```

Pentru fiecare clasa am creat un test, acestea fiind:

boudary_value_analysis.py

```

class TestStampMinCost(unittest.TestCase):
    def test_C_1(self):
        N, M, K, intervals, expected = (1, 1, 1, [(1, 1)], 1)
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_2(self):
        N, M, K, intervals, expected = (1000, 10, 1000, [(1000 * i, 1) for i in range(1, 11)], 1)
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_3(self):
        N, M, K, intervals, expected = (2, 2, 2, [(2, 1), (3, 2)], 1)
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_4(self):
        N, M, K, intervals, expected = (1000, 5, 1000, [(1000 * i, 1) for i in range(1, 6)], 1)
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_5(self):
        N, M, K, intervals, expected = (0, 3, 2, [(3, 1), (6, 2), (8, 3)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_6(self):
        N, M, K, intervals, expected = (1001, 3, 2, [(3, 1), (6, 2), (8, 3)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_7(self):
        N, M, K, intervals, expected = (5, 0, 2, [(3, 1), (6, 2), (8, 3)], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_8(self):
        N, M, K, intervals, expected = (5, 10001, 2, [(3, 1), (6, 2), (8, 3)], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_9(self):
        N, M, K, intervals, expected = (5, 3, 0, [(3, 1), (6, 2), (8, 3)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 10")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_10(self):
        N, M, K, intervals, expected = (5, 3, 1001, [(3, 1), (6, 2), (8, 3)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 10")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_11(self):
        N, M, K, intervals, expected = (5, 3, 2, [(0, 1), (6, 2), (8, 3)], "mi (Limita superioara a intervalului i) trebuie sa fie intre (0 si 10000)")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_12(self):
        N, M, K, intervals, expected = (5, 3, 2, [(100001, 1), (6, 2), (8, 3)], "mi (Limita superioara a intervalului i) trebuie sa fie in")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_13(self):
        N, M, K, intervals, expected = (5, 3, 2, [(3, 0), (6, 2), (8, 3)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

    def test_C_14(self):
        N, M, K, intervals, expected = (5, 3, 2, [(3, 10001), (6, 2), (8, 3)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)")
        self.assertEqual(min_cost_to_buy_stamps((N, M, K, intervals)), expected)

if __name__ == '__main__':
    unittest.main()

```

3. c) Cause-Effect graphing

Pe baza constrangerilor variabilelor, am definit 6 cauze si 7 efecte:

Clasele intrarilor:

- N (Numarul de pagini ale clasorului) $0 < N < 1001$
- M (Numarul de intervale de timbre) $0 < M < 10001$

- K (Lungimea maxima a unei subsecvente de timbre care poate fi luata din interval) $0 < K < 1001$
- (mi, ci) $i \leq M$:
 - mi (Marginea superioara a intervalului i) $0 < mi < 100\ 000$, $i \leq M$
 - ci (Costul intervalului i) $0 < ci < 10\ 000$, $i \leq M$

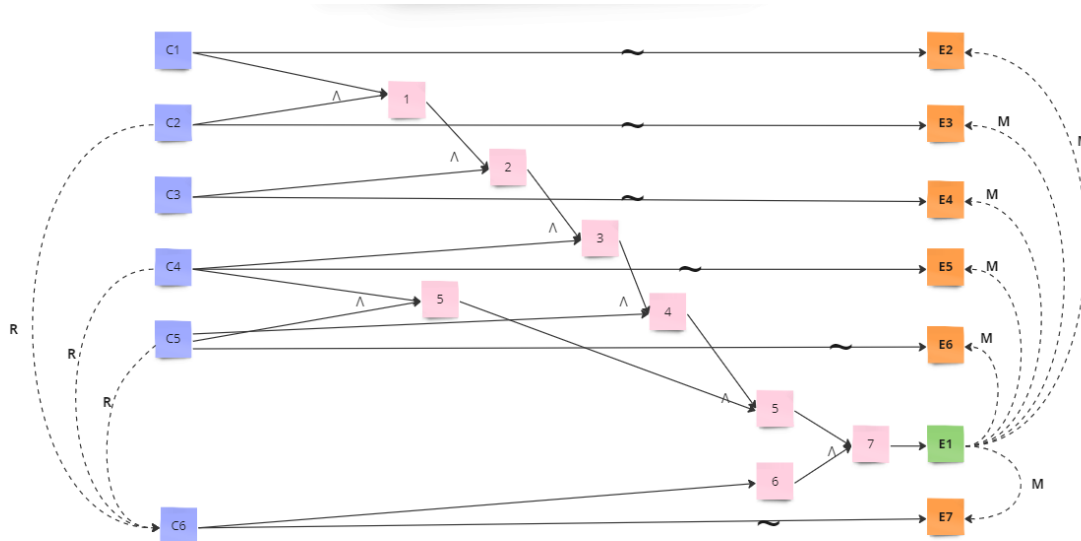
Cauze:

- 1) Alegerea variabilei N
- 2) Alegerea variabilei M
- 3) Alegerea variabilei K
- 4) Alegerea variabilei mi
- 5) Alegerea variabilei ci
- 6) Alegerea numarului total de intervale (ci, mi)

Efecte:

- 1) Afisarea raspunsului (suma minima pe care copila trebuie sa o plateasca pentru a cumpara timbrele necesare)
- 2) Afisarea mesajului de eroare: "N (Numarul de pagini) trebuie sa fie intre 1 si 1000"
- 3) Afisarea mesajului de eroare: "M (Numarul de intervale) trebuie sa fie intre 1 si 10000"
- 4) Afisarea mesajului de eroare: "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000"
- 5) Afisarea mesajului de eroare: "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)"
- 6) Afisarea mesajului de eroare: "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)"
- 7) Afisarea mesajului de eroare: "Numarul de perechi (mi, ci) trebuie sa fie M"

Graful cauza-efect:



Pentru fiecare efect am creat cate un test, acestea fiind:

`cause_effect.py`

```
class TestStampMinCost(unittest.TestCase):
    def test_C_1(self):
        # valid inputs
        N, M, K, intervals, expected = (10, 3, 5, [(10, 4), (20, 7), (30, 8)], 11)
        result = min_cost_to_buy_stamps(N, M, K, intervals)
        self.assertEqual(result, expected)

    def test_C_2(self):
        # N invalid
        N, M, K, intervals, expected = (-1, 3, 5, [(10, 4), (20, 7), (30, 8)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000")
        result = min_cost_to_buy_stamps(N, M, K, intervals)
        self.assertEqual(result, expected)

    def test_C_3(self):
        # M invalid
        N, M, K, intervals, expected = (10, 10001, 5, [(10, 4), (20, 7), (30, 8)], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000")
        result = min_cost_to_buy_stamps(N, M, K, intervals)
        self.assertEqual(result, expected)
```

```

def test_C_4(self):
    # K invalid
    N, M, K, intervals, expected = (10, 3, 1001, [(10, 4), (20, 7), (30, 8)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre
    result = min_cost_to_buy_stamps(N, M, K, intervals)
    self.assertEqual(result, expected)

def test_C_5(self):
    # mi invalid
    N, M, K, intervals, expected = (10, 3, 5, [(10, 4), (100001, 7), (30, 8)], "mi (Limita superioara a intervalului i) trebuie sa fie
    result = min_cost_to_buy_stamps(N, M, K, intervals)
    self.assertEqual(result, expected)

def test_C_6(self):
    # ci invalid
    N, M, K, intervals, expected = (10, 3, 5, [(10, 4), (20, 10001), (30, 8)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 1
    result = min_cost_to_buy_stamps(N, M, K, intervals)
    self.assertEqual(result, expected)

def test_C_7(self):
    # numarul de perechi (mi, ci) este invalid (Nu este M)
    N, M, K, intervals, expected = (10, 3, 5, [(10, 4), (20, 7)], "Numarul de perechi (mi, ci) trebuie sa fie M")
    result = min_cost_to_buy_stamps(N, M, K, intervals)
    self.assertEqual(result, expected)

if __name__ == "__main__":
    unittest.main()

```

- Crearea tabelului de decizie:

Nr crt	1	2	3	4	5	6	7
C1	1	0	1	1	1	1	1
C2	1	0	0	1	1	1	1
C3	1	0	0	0	1	1	1
C4	1	0	0	0	0	1	1
C5	1	0	0	0	0	0	1
C6	1	0	0	0	0	0	0
E1	1	0	0	0	0	0	0
E2	0	1	0	0	0	0	0
E3	0	0	1	0	0	0	0
E4	0	0	0	1	0	0	0
E5	0	0	0	0	1	0	0
E6	0	0	0	0	0	1	0
E7	0	0	0	0	0	0	1

Tabelul 3.1: Tabelele de decizie pentru toate cele 7 efecte

1. $C1 \wedge C2 \wedge C3 \wedge C4 \wedge C5 \wedge C6 = 1$ ($C1 = C2 = \dots = C6 = 1$) \Rightarrow Ef1 = 1
2. $C1 = 0 \Rightarrow$ Ef2 = 1
3. $C2 = 0 \Rightarrow$ Ef3 = 0
4. $C3 = 0 \Rightarrow$ Ef4 = 0
5. $C4 = 0 \Rightarrow$ Ef5 = 0
6. $C5 = 0 \Rightarrow$ Ef6 = 1
7. $\text{count}((mi, ci)) \neq M \Rightarrow$ Ef7 = 1

4. Cerinta 2

Sa se stabileasca nivelul de acoperire realizat de **fiecare** dintre seturile de teste de la 1) a), b) si c) folosind unul dintre utilitarele de code coverage. Sa se compare si sa se comenteze rezultatele obtinute de cele trei seturi de teste.



Nivelul de acoperire a fost testat folosind `Coverage.py` din Python. Mai multe detalii despre acesta puteti gasi [aici](#).

4.1 Nivelul de acoperire pentru *equivalence partitioning*:

Coverage report: 100%				
coverage.py v7.2.3, created at 2023-04-09 17:00 +0300				
Module	statements	missing	excluded	coverage
equivalence_partitioning.py	53	0	0	100%
main.py	28	0	0	100%
Total	81	0	0	100%
coverage.py v7.2.3, created at 2023-04-09 17:00 +0300				

Figura 4.1.1: Raportul de acoperire pentru `main.py` + `equivalence_partitioning.py`

- Testele din `equivalence_partitioning.py` au acoperit toate cazurile din `main.py`.

4.2 Nivelul de acoperire pentru *boundary value analysis*

Coverage report: 99%				
coverage.py v7.2.3, created at 2023-04-09 17:04 +0300				
Module	statements	missing	excluded	coverage
boundary_value_analysis.py	47	0	0	100%
main.py	28	1	0	96%
Total	75	1	0	99%
coverage.py v7.2.3, created at 2023-04-09 17:04 +0300				

Figura 4.2.1: Raportul de acoperire pentru `main.py` + `boundary_value_analysis.py`

- Am obtinut un nivel de acoperire de 99%. In `main.py` nu a fost acoperit cazul in care este verificat daca numarul de intervale (mi, ci) este M (Deoarece testele de `boundary_value_analysis` nu se ocupa cu testarea acestui aspect).

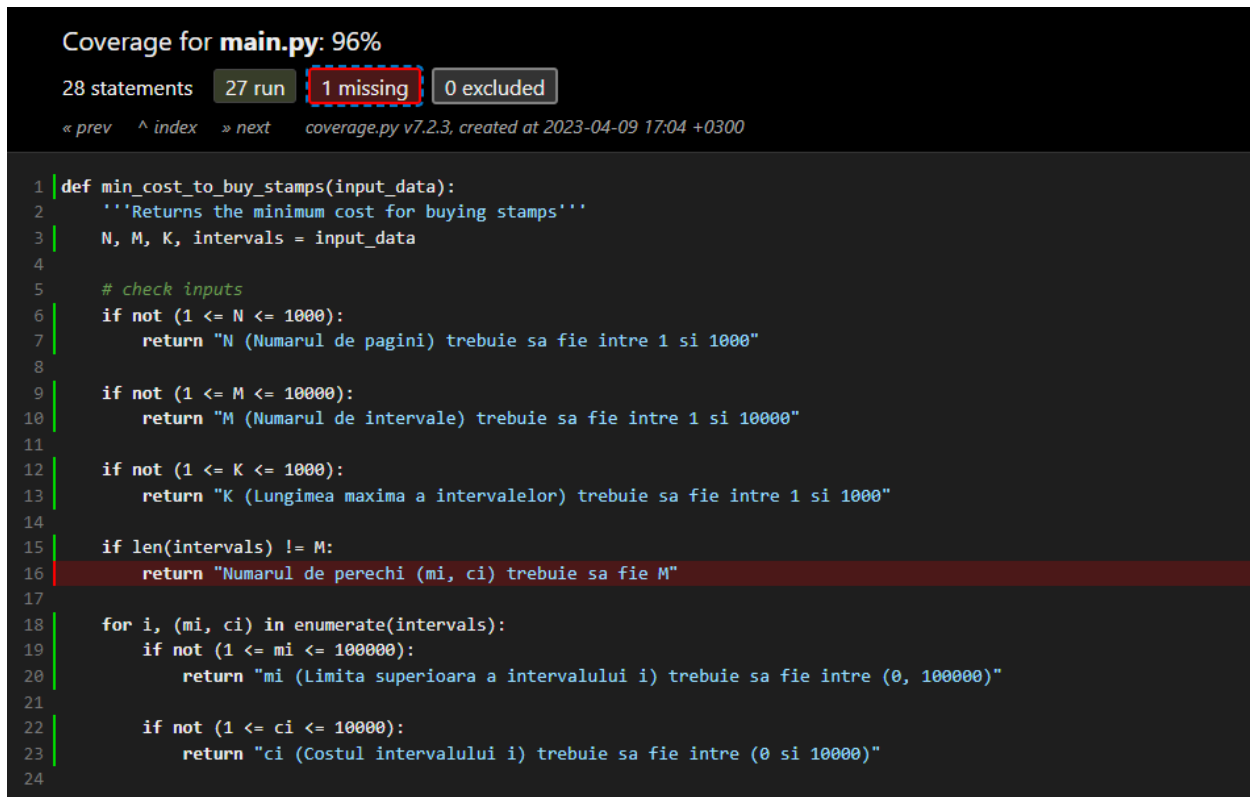


Figura 4.2.2: Raportul detaliat pentru fisierul `main.py`. (Se poate vedea ca linia 16 nu a fost acoperita)

4.3 Nivelul de acoperire pentru *cause-effect*

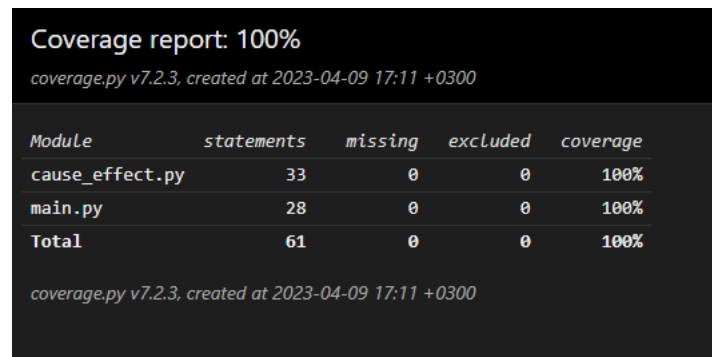


Figura 4.3.1: Raportul de acoperire pentru `main.py` + `cause_effect.py`

- Testele din `cause_effect.py` au acoperit toate cazurile din `main.py`.

5. Cerinta 4

Sa se scrie un mutant de ordinul 1 echivalent al programului.

- Pentru a obtine un mutant de ordin 1 echivalent al programului, am introdus o noua variabila `min_cost` pentru a stoca valoarea minima de cost din setul de costuri, astfel incat functia `min()` sa fie apelata o singura data. Aceasta schimbare nu afecteaza functionalitatea sau output-ul codului, creandu-se astfel un mutant echivalent.

cod original:

```
# loop until all pages have stamps
while N > 0:
    while i >= 0 and intervals[i][0] >= N:
        cost.add(intervals[i][1])
        i -= 1

    if cost:
        c += min(cost)
        cost.remove(min(cost))

    N -= K
```

cod modificat:

```
# loop until all pages have stamps
while N > 0:
    while i >= 0 and intervals[i][0] >= N:
        cost.add(intervals[i][1])
        i -= 1

    if cost:
        min_cost = min(cost)
        c += min_cost
        cost.remove(min_cost)

    N -= K
```

6. Cerinta 5

Pentru unul dintre cazurile de testare sa se scrie un mutant ne-echivalent care sa fie omorat de catre test si un mutant ne-echivalent care sa nu fie omorat de catre test.

6.1 Mutant ne-echivalent care sa fie omorat

Modificarea pe care am facut-o in functia principala a fost sa cresc cu 1 costul fiecarui interval de timbre, astfel rezultatul de output in acest caz va fi diferit fata de codul original.

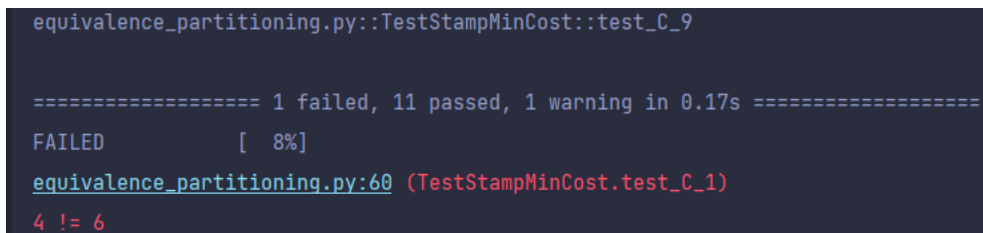
cod original:

```
# loop until all pages have stamps
while N > 0:
    while i >= 0 and intervals[i][0] >= N:
        cost.add(intervals[i][1])
        i -= 1
```

cod modificat:

```
# loop until all pages have stamps
while N > 0:
    while i >= 0 and intervals[i][0] >= N:
        cost.add(intervals[i][1] + 1)
        i -= 1
```

Pentru a omori mutantul, am folosit testele create pentru equivalence partitioning testing.



```
equivalence_partitioning.py::TestStampMinCost::test_C_9

===== 1 failed, 11 passed, 1 warning in 0.17s =====
FAILED [ 8%]
equivalence_partitioning.py:60 (TestStampMinCost.test_C_1)
4 != 6
```

Figura 6.1.1: Mesajul de eroare primit in urma modificarilor facute, mutantul a fost omorat (Asteptam costul minim 4, dar dupa modificarile facute costul minim gasit este 6.)

6.2 Mutant ne-echivalent care sa nu fie omorat

Modificarea pe care am facut-o in functia principala a fost sa sortez intervalele dupa limita superioara a acestora, nu dupa cost. Acest mutant va genera rezultate diferite fata de situatia initiala, insa nu va fi omorat de catre teste.

cod original:

cod modificat:

```
# sortarea intervalelor dupa costul acestora
intervals.sort(key=lambda x: x[1])
```

```
# sortarea intervalelor dupa limita superioara a acestora
intervals.sort(key=lambda x: x[0])
```

Pentru a testa daca mutantul a fost omorat sau nu, am folosit testele create pentru *boundary value analysis*.

```
(base) PS C:\Users\User\PycharmProjects\test4> coverage run .\boundary_value_analysis.py
.....
-----
Ran 14 tests in 0.002s

OK
(base) PS C:\Users\User\PycharmProjects\test4>
```

Figura 6.2.1: Mesajul de success primit in urma rularii celor 7 teste din *boundary_value_analysis.py* dupa ce am facut modificarile mentionate, testele nu au prins eroarea introdusa.

7. Cerinta 3

Sa se transforme programul intr-un graf orientat si, pe baza acestuia, sa se gaseasca un set de teste care satisface criteriul *modified condition/decision coverage* (MC/DC).

Liniile de cod la care se iau decizii sunt urmatoarele:

```
1. if not (1 <= N <= 1000)
2. if not (1 <= M <= 10000)
3. if not (1 <= K <= 1000)
4. if len(intervals) != M
5. for i, (mi, ci) in enumerate(intervals):
    5.a. if not (1 <= mi <= 100000)
    5.b. if not (1 <= ci <= 10000)
6. while N > 0
7. while i >= 0 and intervals[i][0] >= N
8. if cost
```

Am creat 7 test case-uri pentru a testa fiecare linie de cod din cele de mai sus:

`MC_DC_criterion.py`

```
from main import min_cost_to_buy_stamps

test_cases = [
    (0, 1, 1, [(1, 1)], "N (Numarul de pagini) trebuie sa fie intre 1 si 1000"),
    (1, 0, 1, [], "M (Numarul de intervale) trebuie sa fie intre 1 si 10000"),
    (1, 1, 0, [(1, 1)], "K (Lungimea maxima a intervalelor) trebuie sa fie intre 1 si 1000"),
    (1, 1, 1, [(1, 1), (2, 1)], "Numarul de perechi (mi, ci) trebuie sa fie M"),
    (1, 1, 1, [(0, 1)], "mi (Limita superioara a intervalului i) trebuie sa fie intre (0, 100000)"),
    (1, 1, 1, [(1, 0)], "ci (Costul intervalului i) trebuie sa fie intre (0 si 10000)"),
    (4, 3, 2, [(5, 3), (2, 1), (6, 2)], 3), # input valid + solutie valida
]

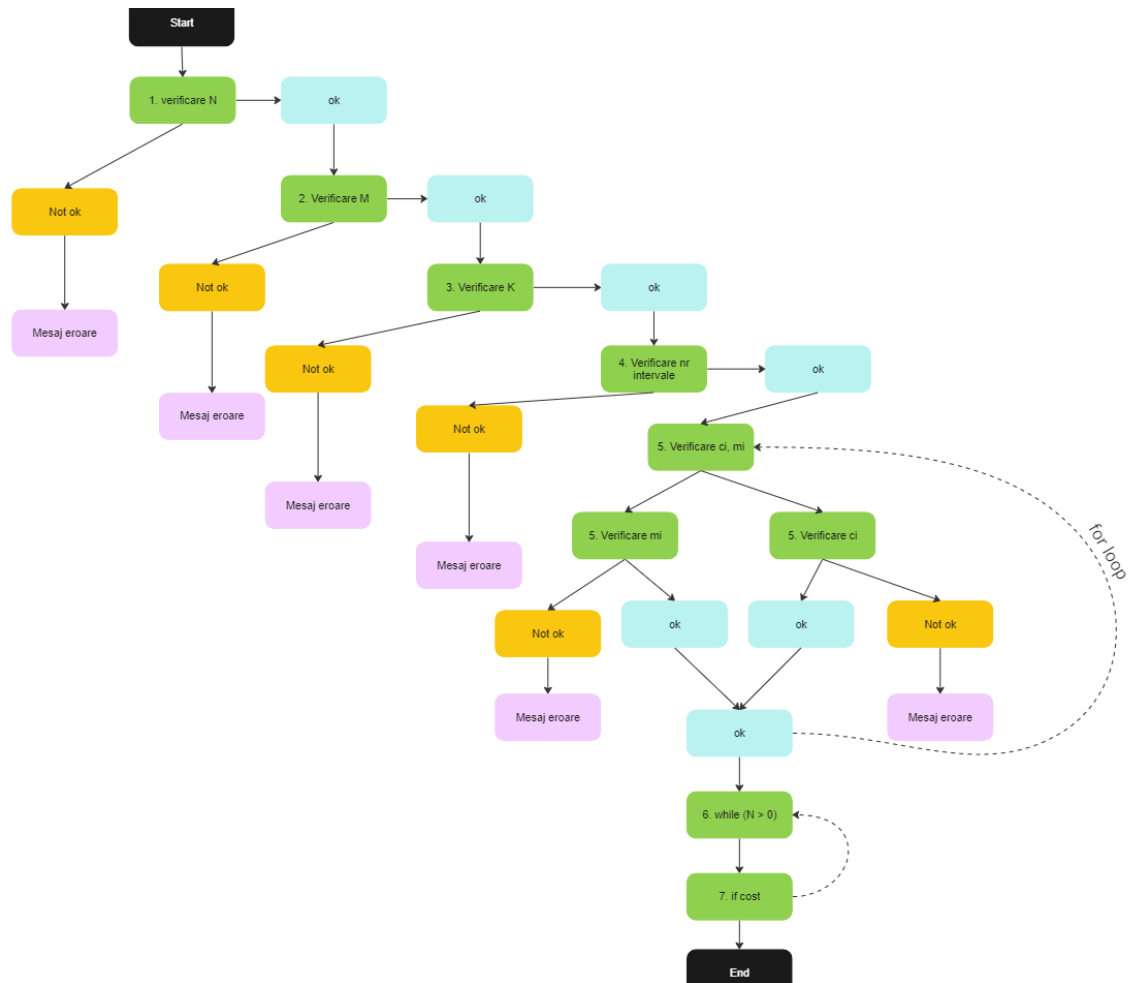
for i, (N, M, K, intervals, expected_output) in enumerate(test_cases, start=1):
    result = min_cost_to_buy_stamps((N, M, K, intervals))
    assert result == expected_output, f"Test {i} failed: expected {expected_output}, got {result}"
    print(f"Test {i} passed")
```

Toate cele 7 teste au trecut.

Graful orientat al programului este urmatorul:



In construirea grafului nu am luat toate liniile de cod fiindca ar fi rezultat un graf foarte mare, asa ca am luat liniile de decizie specificate anterior si am construit graful.



8. Bibliografie

1.


Coverage.py — Coverage.py 7.2.3 documentation

 <https://coverage.readthedocs.io/en/7.2.3/>

2.

Mutation Testing with Python


Test the tests—automatically, by applying common mistakes

 <https://medium.com/analytics-vidhya/unit-testing-in-python-mutation-testing-7a70143180d8>



3.

Equivalence partition « Python recipes « ActiveState Code

 <https://code.activestate.com/recipes/499354-equivalence-partition/>


4.

<https://www.guru99.com/equivalence-partitioning-boundary-value-analysis>

5.

Cause-Effect Graph Technique in Black Box Testing - javatpoint

Cause-Effect Graph Technique in Black Box Testing with introduction, software development life cycle, design, development, testing, quality assurance, quality control, methods, black box testing, white box testing, etc.

 <https://www.javatpoint.com/cause-and-effect-graph-technique-in-black-box-testing>