# Tema 9: Unit Testing with Python
# Testarea problemei **barbar** de pe infoarena.ro

AVRAM DANIEL

## 1 Descrierea problemei

Paftenie barbarul a fost capturat de către dușmanii săi și aruncat într-o temniță. Temnița este de fapt un grid de dimensiune R*C. În anumite celule există dragoni, unele sunt ocupate de pereți, iar altele sunt libere. Paftenie trebuie să iasă din temniță mergând numai prin celule libere (o celulă are maxim 4 vecini), și asta stând cat mai departe de dragoni (astfel încât minima din distanțele până la cel mai apropiat dragon din fiecare din celulele traseului său să fie maxim).

### 1.1 Cerința probleme

Ajutați-l pe barbarul Paftenie să iasă din temniță, determinând un traseu astfel încât distanța minimă până la cel mai apropiat dragon din fiecare dintre celulele traseului său să fie maximă.

### 1.2 Datele de intrare

Pe prima linie a fișierului de intrare *barbar.in* sunt date două numere întregi **R** și **C**, reprezentând numărul liniilor, respectiv al coloanelor temniței. Pe următoarele **R** linii se află câte **C** caractere, neseparate prin spații, cu următoarele semnificații:

**.** celula libera
***** perete
**D** dragon
**I** punctul de plecare al lui Paftenie
**O** iesirea din temnita

### 1.3 Datele de ieșire

Fișierul de ieșire *barbar.out* va conține pe prima linie un singur număr, reprezentând valoarea maximă pentru minima din distanțele până la cel mai apropiat dragon din fiecare din celulele traseului. În caz că nu există soluție se va afișa -1.

### 1.4 Restricții și precizări

- $2 \leq R, C \leq 500$
- Se garantează că în temniță există cel puțin un dragon

## 2 Rezolvare

Soluția problemei constă în încercarea a mai multor distanțe maxime prin care Paftenie poate să treacă. Mai întâi, este calculată matricea de distanțe $dist_{R,C}$ unde *dist[i][j]* va reprezenta distanța celulei *(i, j)* pana la cel mai apropiat dragon. Apoi, se incearca diferite valori pentru distanța $x$ pentru a vedea dacă există un traseu de la **I** la **O** care trece prin celule ale căror distanțe sunt mai mici sau egale cu $x$, $dist[i][j] \leq x, \forall (i, j)$ celule din traseu.

### 2.1 Distanțele de la dragoni

Pentru a calcula matricea *dist* se va folosi un *BFS*. La început, în coada $q$ din *BFS* sunt înserate pozițiile tuturor dragonilor, iar distanțele pe aceste poziții sunt inițializate cu 0. Apoi, *BFS* va începe o parcurgere de la pozițiile dragonilor spre vecinii săi. Valoarea *dist* a acestor celule nevizitate va fi egală cu $dist[i][j] + 1$, unde (i, j) e pozitia celulei de la care s-a ajuns la celula curentă.

**Complexitate:** $R * C$

### 2.2 Verificarea existenței unui traseu pentru o anumită distanță minimă

Pentru o distanță minimă $x$, pentru a verfica dacă există un traseu de la **I** la **O** care să treacă prin celule *(i, j)* a căror $dist[i][j] \leq x$, vom începe o parcurgere *BFS* de la **I** prin matricea *dist*. Dacă în timpul iterării prin coada *BFS*-ului am ajuns la coordonatele **O**, atunci $x$ este o valoarea validă.

**Complexitate:** $R * C$

### 2.3 Căutarea celei mai mici valori valide

Utilizând cautare binară, căutăm o valoare $x$ valida care sa fie cea mai mică posibilă. Dacă această valoare nu va fi găsită, răspunsul va fi *-1*. Pentru a verifica dacă $x$ este o valoare validă, folosim *BFS*-ul de la pasul anterior.

**Complexitate:** $log(max(R, C)) * (R * C)$

## 3 Testare

Pentru testare am folosit librăria standard `unittest` pentru scrierea testelor și `pytest` pentru rularea lor.

Testele sunt scrise in forma de clase care moștenesc de la `unittest.TestCase` și ai căror metode sunt funcții de testare, care trebuie să înceapă cu **test** pentru a fi detectate.

```
class InputFormattingTestCase(unittest.TestCase):

  def test_wrong_R(self):
    with self.assertRaisesRegex(Exception, R_NU_E_INT):
      in_file = 'wrong_R_test.in'
      Solver(in_file)
```

### 3.1 White Box - Formatarea datelor de intrare și ieșire

1. **R** nu este un integer;
2. **C** nu este un integer;
3. **R** este mai mic decât 2;
4. **C** este mai mic decât 2;
5. **R** este mai mare decât 500;
6. **C** este mai mare decât 500;
7. Una din linii nu are**C** caractere;
8. Fișierul nu are **R** linii;
9. Nu exista **I** in grid;
10. Nu exista **O** in grid;
11. Fișierul de intrare conține un caracter diferit de cele permise;
12. Fișierul de intrare nu conține **D** (dragoni).

Pentru aceste teste, se face aserțiunea că un anumit snippet de cod va arunca o Excepție de un anumit tip cu un anumit mesaj:

```
def test_wrong_ok_file(self):
    with self.assertRaisesRegex(ValueError, nu_este_numar('not_a_number')):
      out_file = 'wrong_ok.ok'
      get_output(out_file)
```

### 3.2 Black Box - Teste care verifică potențiale greșeli de implementare

- Testele preluate de pe *infoarena.ro*: grader_test1 - grader_test9 verifică edge cases ce au fost considerate de către autorii problemei, inclusiv cazul în care nu există un traseu de la **I** la **O**, deci răspunsul este *-1*.
- Testele grader_test11 - grader_test13 sunt teste introduse de mine cu scopul de a ucide mutanții apăruți în urma testării.
    1. grader_test11 are valoarea **R** și **C** minime, $R, C = 2$. Acest test ucidea mutanții care incrementau valoarea de **MIN_NR** în condițiile care verficau dacă **R** și **C** se încadrează;
    2. grader_test12 are valorile **R** și **C** diferite, $R \neq C$. Testele de pe infoarena nu ofereau acest caz. Mutanții uciși de acest test atribuiau lui **R** valoarea lui **C** sau vice-versa.
    3. grader_test13 poziționează I pe coordonate care nu sunt egale, (0, 1). În toate testele de pe infoarena, **I** se află pe coordonate egale ( (2, 2), de exemplu). Mutanții care au fost uciși de acest test memorau poziția **I** doar după valoarea coloanei sau a rândului și o foloseau și pentru cealaltă coloană.

Pentru aceste teste, se face aserțiunea că un anumit input va returna un anumit output:

```
class GroundTruthsTestCase(unittest.TestCase):

  def get_answer_and_ground_truth(self, index):
    file_name = 'grader_test'
    in_file = file_name + str(index) + '.in'
    out_file = file_name + str(index) + '.ok'
    solver = Solver(in_file)
    solver.solve()
    ground_truth = get_output(out_file)
    return (solver.answer, ground_truth)

  def test_1(self):
    ans, gt = self.get_answer_and_ground_truth(1)
    self.assertEqual(ans, gt)

  ...
```

## 4 Mutații

Pentru generarea de mutanți, a fost folosită librăria `mutmut` din urmatoarele motive:

1. **Ușurința utilizării:** Doar prin comanda `mutmut run`, librăria va încerca de sine stătător să găsească codul sursă care trebuie mutat și testele pe care trebuie să le ruleze. De asemenea, dacă `pytest` sau `unittest` este deja instalat, `mutmut` va ști să le utilizeze pentru rularea testelor.
2. **Configurație:** Dacă librăria nu e în stare să găsească codul sursă și testele, acestea pot fi configurate direct în comanda din terminal sau în `setup.cfg`.

3. **Continuitate:** Dacă procesul de generare și testare a mutanților a fost intrerupt, următoarea rulare va începe de la punctul la care s-a oprit precedenta rulare. De asemenea, lista mutanților poate fi accesată utilizând comanda `mutmut results`, iar detaliile privind un mutant anumit pot fi accesate cu `mutmut show MUTATION_ID`.

## 4.1 Rezultate

În urma rulării mutațiilor, au fost obținute 143 de mutanți, dintre care:

- **130** au fost uciși;
- **9** au expirat (rularea lor pe teste a durat mai mult decât codul original) din cauza unui loop infinit;
- 4 mutanți au supravețuit.

```
    @@ -82,7 +82,7 @@
-           i = curr[0] + ROW[p]
+           i = curr[0] - ROW[p]

    @@ -83,7 +83,7 @@
-           j = curr[1] + COL[p]
+           j = curr[1] - COL[p]

    @@ -108,7 +108,7 @@
-           i = curr[0] + ROW[p]
+           i = curr[0] - ROW[p]

    @@ -109,7 +109,7 @@
-           j = curr[1] + COL[p]
+           j = curr[1] - COL[p]
```

## 4.2 Motivarea mutanților ce au supravețuit

fisierul `constants.py` conține două liste care păstrează direcția în care *BFS* se va propaga:

```
ROW = [0, -1, 0, 1]
COL = [-1, 0, 1, 0]
```

*BFS*-ul alege următoarea celulă pe care să o introducă în coadă în modul următor:

```
curr = self.q[0]

for p in range(4):
    i = curr[0] + ROW[p]
    j = curr[1] + COL[p]
```

Conform codului, următoarele poziții vor fi alese în modul următor: *stânga, sus, dreapta, jos.* În urma mutațiilor, *i* sau *j* sunt calculate prin scăderea *ROW[p]* sau *COL[p]*, însă oricum toate direcțiile vor fi vizitate, se va schimba doar ordinea. În cazul:

```
i = curr[0] - ROW[p]
j = curr[1] + COL[p]
```

ordinea va fi: *stânga, jos, dreapta, sus*; In cazul:

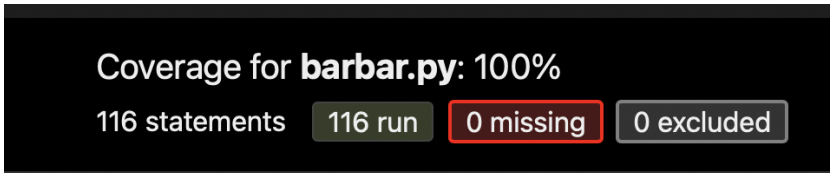
```
i = curr[0] + ROW[p]
j = curr[1] - COL[p]
```

ordinea va fi: *dreapta, sus, stânga, jos.*

În ambele cazuri algoritmul rămâne valid, din moment ce toți vecinii sunt vizitați, doar că în ordine diferită.

# 5 Coverage

Pentru verificarea acoperirii de testele scrise, se folosește librăria `coverage`. Aceasta monitorizează fiecare snippet de cod la execuția testelor. De asemenea, librăria generează un raport în format html la rularea comenzii `coverage html` care afișează snippeturile de cod ce au fost testate și care nu, precum și procentajul de acoperire. În cazul testelor implementate în cadrul acestui proiect, am obținut un coverage de 100%.




De asemenea, la rularea comenzii `coverage run --branch -m pytest` are loc rularea testelor cu monitorizarea arborelui de decizie a programului. În urma rulării, am obținut coverage de 100%. Rezultatele pot fi vizualizate direct în terminal cu `coverage report`:

| Name | Stmts | Miss | Branch | BrPart | Cover |
|---|---|---|---|---|---|
| barbar.py | 115 | 0 | 60 | 0 | 100% |
| constants.py | 5 | 0 | 0 | 0 | 100% |
| test_ground_truths.py | 47 | 0 | 2 | 0 | 100% |
| test_input_formatting.py | 57 | 0 | 2 | 0 | 100% |
| text_formatting.py | 17 | 0 | 0 | 0 | 100% |
| TOTAL | 241 | 0 | 64 | 0 | 100% |