UWAGA: Wczytaj do Colab plik **frozen_lake_slippery.py** (intrukcja w pliku **COLAB_instrukcja.pdf**)

FrozenLake 2

```
import gym
import numpy as np

env = gym.make("FrozenLake-v0",is_slippery=True)
```

FrozenLake z poślizgiem

W notatniku **FrozenLake_1** pracowaliśmy ze środowiskiem w którym **nie był możliwy poślizg** (plik frozen_lake.py). Oznaczało to, że po wykonaniu przez agenta pewnej akcji wiedzieliśmy do jakiego stanu agent przejdzie. Przypomnijmy następujący fragment z notatnika **FrozenLake_1**:

Rozważmy przykład: w stanie 0 agent wykonuje akcję 1 (porusza się w dół):

```
[ ]  env.P[0][1]
```

```
[(1.0, 4, 0.0, False)]
```

Czyli agent przeszedł ze stanu 0 do stanu 4 (z prawdopodobieństwem 1). Wykonajmy tę samą instrukcję teraz (pracujemy z plikiem **is_slippery=True**):

```
env.P[0][1]
```

```
[(0.3333333333333333, 0, 0.0, False),
 (0.3333333333333333, 4, 0.0, False),
 (0.3333333333333333, 1, 0.0, False)]
```

A zatem otrzymujemy opis dynamiki: po wykonaniu akcji 1 w stanie 0 agent przejdzie do stanu 0 z prawdopodobieństwem 0.3333..., do stanu 4 z prawdopodobieństwem 0.3333..., do stanu 1 z prawdopodobieństwem 0.3333... Wszystkie możliwe nagrody wynoszą 0. Czyli uwzględniony jest **poślizg na lodzie**.

Zwróćmy uwagę na to, że powyższe wyrażenie jest listą, której elementami są krotki (tuples) zawierające:

(prawdopodobieństwo przejścia, nowy stan, nagrodę, czy nowy stan jest końcowy?)

Poszczególne z tych wartości dla stanu początkowego **s=0** i akcji **a=1** możemy uzyskać następująco:

```
for next_state in range(len(env.P[0][1])):

    prob, next_state, reward, done = env.P[0][1][next_state]

    print(prob," ",next_state," ",reward," ",done)
```

```
0.3333333333333333   0   0.0   False
0.3333333333333333   4   0.0   False
0.3333333333333333   1   0.0   False
```

Pętla powyższa przyda się nam w implementacji jednego z algorytmów na końcu notatatnika.

Polecenie 1 (do uzupełnienia)

Sprawdź dynamikę dla dla następujących przypadków:

W **stanie 1** agent **przechodzi w dół**:

```
env.P[0][1]
```

```
[(0.3333333333333333, 0, 0.0, False),
 (0.3333333333333333, 4, 0.0, False),
 (0.3333333333333333, 1, 0.0, False)]
```

W **stanie 10** agent **przechodzi w lewo**:

```
env.P[9][0]
```

```
[(0.3333333333333333, 5, 0.0, True),
 (0.3333333333333333, 8, 0.0, False),
 (0.3333333333333333, 13, 0.0, False)]
```

W **stanie 14** agent **przechodzi w prawo**:

```
env.P[13][2]
```

```
[(0.3333333333333333, 13, 0.0, False),
 (0.3333333333333333, 14, 0.0, False),
 (0.3333333333333333, 9, 0.0, False)]
```

Polityka stochastyczna

Polityka to mówiąc najprościej strategia postępowania agenta. **Polityka jest stochastyczna** jeżeli w każdym stanie agent może wybrać dopuszczalne akcje z jakimiś prawdopodobieństwami < 1. **Polityka jest deterministyczna** jeżeli w każdym stanie agent wybiera pewną akcję z prawdopodobieństem 1 .

W przypadku środowiska FrozenLake mamy **16 stanów** i **4 akcje**, a zatem **politykę stochastyczną** możemy zdefiniować np. tak:

```
stochastic_policy = np.ones([env.nS, env.nA]) / env.nA

print(stochastic_policy)
```

```
[[0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
```

Jest to bardzo prosta **polityka stochastyczna** w której prawdopodobieństwo wyboru każdej z akcji w dowolnym stanie wynosi **0.25**.

Prawdopodobieństwo wyboru w stanie **s** akcji **a** jest określone jako: `stochastic_policy[s][a]`

Przykład:

```
stochastic_policy[0][1]
```

```
0.25
```

Polecenie 2 (do uzupełnienia)

Zdefiniuj politykę stochastyczną w której **dla różnych akcji będa różne wartości prawdopodobieństwa ich wyboru**.

```
import random

stochastic_policy2=[]
for _ in range(16):
 r = [random.random() for i in range(4)]

 s =sum(r)

 r = [ round(i/s,3) for i in r]

 print(r)
 stochastic_policy2.append(r)


stochastic_policy2 = np.array(stochastic_policy2)
```

```
[0.185, 0.116, 0.25, 0.45]
[0.444, 0.249, 0.157, 0.15]
[0.237, 0.398, 0.265, 0.101]
[0.036, 0.444, 0.172, 0.348]
[0.174, 0.445, 0.177, 0.204]
```

```
[0.031, 0.461, 0.271, 0.238]
[0.124, 0.428, 0.329, 0.119]
[0.079, 0.572, 0.194, 0.155]
[0.366, 0.369, 0.164, 0.101]
[0.323, 0.246, 0.419, 0.013]
[0.26, 0.283, 0.223, 0.233]
[0.225, 0.339, 0.2, 0.236]
[0.097, 0.525, 0.163, 0.216]
[0.439, 0.126, 0.384, 0.051]
[0.36, 0.201, 0.102, 0.337]
[0.299, 0.289, 0.049, 0.363]
```

Algorytm iteracyjnego obliczenia polityki

Algorytm ten pozwala znaleźć **wartości oczekiwane zwrotów V(s)** dla każdego stanu **s** przy założeniu, że **agent wykorzystuje pewną politykę** oznaczoną zwykle przez **pi**. Algorytm wygląda następująco:

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Początkowo przyjmujemy, że **V(s)=0** dla każdego stanu **s**. Możemy to zapisać tak:

```
V = np.zeros(env.nS)
```

Sprawdzamy:

```
print(V)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Jak działa algorytm? Zacznimy od uproszczonej postaci:

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow$ dotychczasowa wartość V(s)
    $V(s) \leftarrow$ nowa wartość V(s)
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Zewnętrzna pętla (**Loop... until...**) służy do sprawdzenia jak duże były ostatnio **wprowadzone modyfikacje wartości V(s)**. Jeżeli były niewielkie (**Delta<Theta**) wówczas następuje przerwanie pętli (**Delta** jest zawsze modyfikowana po zmianie wartości V(s) i ostatecznie jest równa **największej z modyfikacji V(s)** biorąc pod uwagę wszystkie stany).

Powyższe pętle można zrealizować w następujący sposób:

```
while True:
  delta = .0
  for state in range(env.nS):# env.nS=16

    #tutaj musimy wyliczyć nową wartość V(s)

    delta = max(delta, np.abs(V[state] - Vs))
  if delta < theta:
    break
```

Jak wyliczyć wartość **V(s)**? Zgodnie z algorytmem musimy skorzystać z **równania Bellmana**:

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$$

Wartość **V(s)** obliczamy biorąc pod uwagę wartości **V(s')** wszystkich stanów do których może przejść agent ze stanu **s**.

Zwróćmy uwagę, że sumujemy **po wszystkich akcjach**, które mogą być wykonane w stanie **s** i sumujemy po wszystkich stanach **s'** do których agent może przejść ze stanu **s** oraz po wszystkich możliwych nagrodach **r**.

Wielkość **p(s',r|s,a)** jest prawdopodobieństem tego, że po wykonaniu w **stanie s akcji a** agent przejdzie do **stanu s'** i otrzyma przy tym **nagrodę r**.

Nową wartość Vs możemy wyliczyć następująco:

```
Vs = 0
#sumowanie po wszystkich akcjach możliwych do wykonania w stanie s
for action in range(env.nA):

  #sumowanie po wszystkich stanach do których może przejść agent ze stanu s
  for next_state in range(len(env.P[state][action])):

    prob, next_state, reward, done = env.P[state][action][next_state]
```

```
    Vs += policy[state][action] * prob * (reward + gamma * V[next_state])
```

Teraz już możesz wykonać **Zadanie 3** z **RL_zadania_4.pdf**. W tym celu uzupełnij definicję poniższej funkcji **policy_evaluation** pozwalającej dla danej **polityki** (zdefiniowana powyżej) i **parametrów gamma i theta** znaleść **wartość oczekiwane zwrotów V(s)**.

```
#def policy_evaluation(env, policy, gamma=0.8, theta=1e-8):
def policy_evaluation(env, policy, gamma=0.8, theta=1e-8):
    V = np.zeros(env.nS)

    while True:
        delta = .0

        for state in range(env.nS):
           #sumowanie po wszystkich akcjach możliwych do wykonania w stanie s
          Vs = 0
          for action in range(env.nA):

          #sumowanie po wszystkich stanach do których może przejść agent ze stanu s
             for next_state in range(len(env.P[state][action])):

               prob, next_state, reward, done = env.P[state][action][next_state]

               if(next_state==15):
                 reward=1
```

```
            else:
              reward=-1

          # print("r",reward)
            Vs += policy[state][action] * prob * (reward + gamma * V[next_state])
        #do uzupełnienia

          delta = max(delta, np.abs(V[state] - Vs))
          V[state]=Vs
        print("delta=",delta)
        if delta < theta:
            break

    return V
```

Użycie funkcji:

```
V = policy_evaluation(env,stochastic_policy)
```

```
delta= 1.5792
delta= 0.96
delta= 0.6975999999999998
delta= 0.5427840000000002
delta= 0.4181171199999998
delta= 0.32767999999999997
delta= 0.26214400000000015
delta= 0.2097152000000002
delta= 0.16777216000000017
delta= 0.13421772799999943
delta= 0.10737418240000007
delta= 0.08589934592000059
delta= 0.06871947673599976
delta= 0.05497558138879999
delta= 0.04398046511103981
delta= 0.03518437208883185
delta= 0.028147497671065835
delta= 0.022517998136851958
```

```
delta= 0.018014398509482277
delta= 0.014411518807586177
delta= 0.011529215046068408
delta= 0.009223372036854194
delta= 0.007378697629484066
delta= 0.005902958103586542
delta= 0.004722366482869944
delta= 0.0037778931862959553
delta= 0.0030223145490371195
delta= 0.0024178516392296956
delta= 0.0019342813113834012
delta= 0.0015474250491065433
delta= 0.0012379400392852347
delta= 0.0009903520314287206
delta= 0.0007922816251424436
delta= 0.0006338253001141325
delta= 0.0005070602400909507
delta= 0.00040564819207311587
delta= 0.00032451855365778215
delta= 0.000259614842926581
delta= 0.00020769187434144243
delta= 0.00016615349947368685
delta= 0.00013292279957788367
delta= 0.0001063382396626622
delta= 8.507059173012976e-05
delta= 6.805647338481435e-05
delta= 5.444517870678567e-05
delta= 4.355614296613908e-05
delta= 3.484491437255599e-05
delta= 2.7875931498222428e-05
delta= 2.2300745198045036e-05
delta= 1.7840596159501843e-05
delta= 1.427247692653566e-05
delta= 1.14179815415838e-05
delta= 9.134385233977582e-06
delta= 7.307508186116252e-06
delta= 5.846006549958815e-06
delta= 4.676805239967052e-06
delta= 3.7414441909078278e-06
```

Wypisanie wyliczonych wartości zwrotów:

```
print(V)
```

```
[-4.98437064 -4.98291689 -4.94729698 -4.98243231 -4.97019506 -4.99999996
 -4.82383875 -4.99999996 -4.89640964 -4.61544358 -4.17189687 -4.99999996
 -4.99999996 -4.0089114  -1.42020208  4.99999996]
```

```
from plot_utils import plot_values
```

```
plot_values(V)
```

0 s ukończono o 12:44