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

FrozenLake 2

```
from frozen_lake_slippery import FrozenLakeEnv
import numpy as np

env = FrozenLakeEnv()
```

FrozenLake z poślizgiem

W notatniku **FrozenLake_1** pracowaliśmy ze środowiskiem w którym **nie był możliwy poślizg** (plik wykonaniu przez agenta pewnej akcji wiedzieliśmy do jakiego stanu agent przejdzie. Przypomnijm **FrozenLake_1**:

Rozważmy przykład: w stanie 0 agent wykonuje akcję 1 (porusz

```
[ ]  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 ins **frozen_lake_slippery.py**):

```
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 z prawdopodobieństwem 0.3333..., do stanu 1 z prawdopodobieństwem 0.3333... Wszystkie moż jest **poślizg na lodzie**.

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

(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 następują

```
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[1][1]
```

```
[(0.3333333333333333, 0, 0.0, False),
 (0.3333333333333333, 5, 0.0, True),
 (0.3333333333333333, 2, 0.0, False)]
```

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

```
env.P[10][0]
```

```
[(0.3333333333333333, 6, 0.0, False),
 (0.3333333333333333, 9, 0.0, False),
 (0.3333333333333333, 14, 0.0, False)]
```

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

```
env.P[14][2]
```

```
[(0.3333333333333333, 14, 0.0, False),
 (0.3333333333333333, 15, 1.0, True),
 (0.3333333333333333, 10, 0.0, False)]
```

Polityka stochastyczna

Polityka to mówiąc najprościej strategia postępowania agenta. **Polityka jest stochastyczna** jeżeli dopuszczalne akcje z jakimiś prawdopodobieństwami < 1. **Polityka jest deterministyczna** jeżeli w

prawdopodobieństem 1 .

W przypadku środowiska FrozenLake mamy **16 stanów** i **4 akcje**, a zatem **politykę stochastyczną**

```
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 akcj

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)

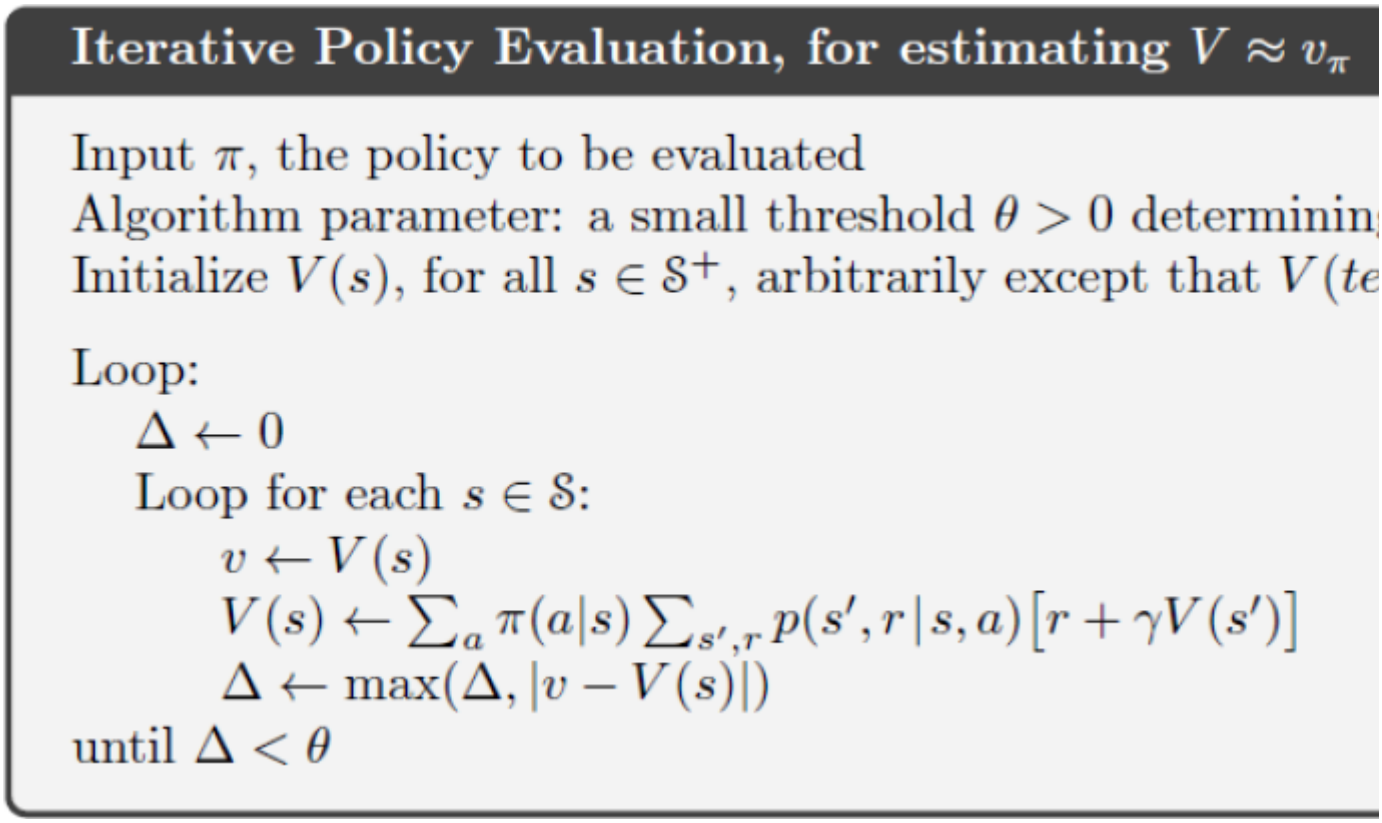
Zdefiniuj politykę stochastyczną w której **dla różnych akcji będa różne wartości prawdopodobieńs**

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

stochastic_policy/=stochastic_policy.sum(axis=1,keepdims=1)

print(stochastic_policy)
```

```
[[0.21536312 0.4354764  0.17790051 0.17125997]
 [0.0457116  0.17962332 0.40647394 0.36819114]
 [0.26299565 0.074882   0.61988765 0.0422347 ]
 [0.33293085 0.12765506 0.22624016 0.31317393]
 [0.16938568 0.24576977 0.40330705 0.1815375 ]
 [0.26211557 0.26057182 0.41060569 0.06670692]
 [0.33375841 0.3635139  0.11206396 0.19066373]
 [0.26303683 0.34778694 0.01200149 0.37717473]
 [0.23655667 0.57228221 0.13133708 0.05982404]
 [0.33755811 0.22285541 0.19591048 0.243676  ]
 [0.15798191 0.29199795 0.43581766 0.11420248]
 [0.42442916 0.01984698 0.15565622 0.40006764]
 [0.39171444 0.0757541  0.13008757 0.40244389]
 [0.32257389 0.22736699 0.33544124 0.11461788]
```

Algorytm iteracyjnego obliczenia polityki

Algorytm ten pozwala znaleźć **wartości oczekiwane zwrotów V(s)** dla każdego stanu **s** przy założ oznaczoną zwykle przez **pi**. Algorytm wygląda następująco:



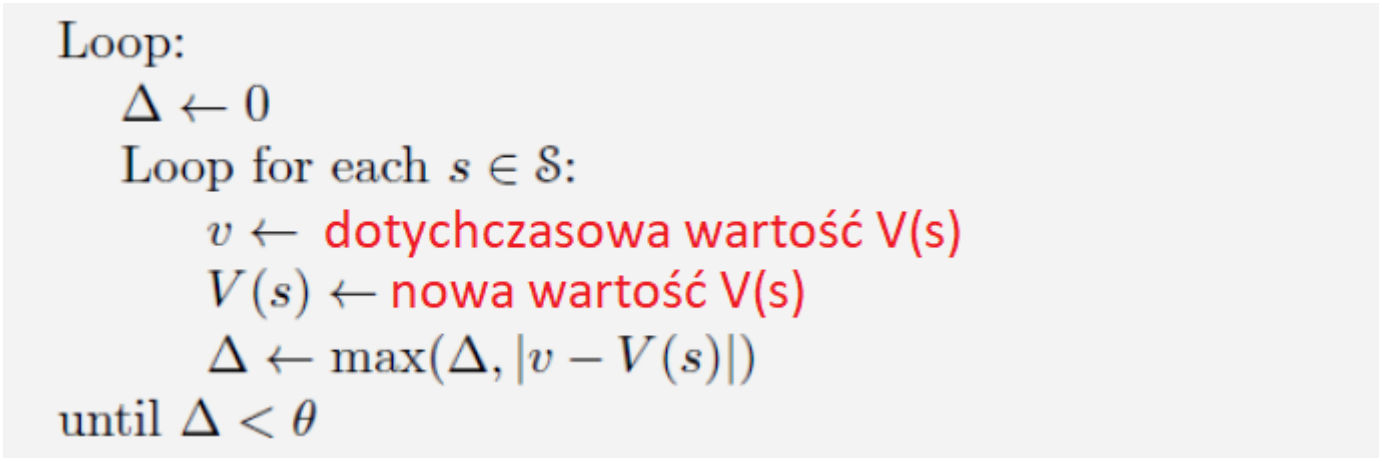
Początkowo przyjmujemy, że **V(s)=0** dla każdego stanu **s**. Możemy to zapisać tak:

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

Sprawdzamy:

```
print(V)
```

Jak działa algorytm? Zacznimy od uproszczonej postaci:



Zewnętrzna pętla (**Loop... until...**) służy do sprawdzenia jak duże były ostatnio **wprowadzone mod** (**Delta<Theta**) wówczas następuje przerwanie pętli (**Delta** jest zawsze modyfikowana po zmianie **największej z modyfikacji V(s)** biorąc pod uwagę wszysktie 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 prz

Zwróćmy uwagę, że sumujemy **po wszystkich akcjach**, które mogą być wykonane w stanie **s** i sun 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 prze

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_lab_4.pdf**. W tym celu uzupełnij definicję poniższej fun **polityki** (zdefiniowana powyżej) i **parametrów gamma i theta** znaleść **wartość oczekiwane zwrotó**

```
def policy_evaluation(env, policy, gamma=0.5, theta=1e-8):
    V = np.zeros(env.nS)

    while True:
      delta = .0
      for state in range(env.nS):
        v = V[state]

        V[state] = 0

        for action in range(env.nA):
          for next_state in range(len(env.P[state][action])):

            prob, next_state, reward, done = env.P[state][action][next_state]
            V[state] += policy[state][action] * prob * (reward + gamma * V[next_state])
        delta = max(delta,np.abs(V[state] - v))
      if delta < theta:
          break

    return V
```

Użycie funkcji:

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

Wypisanie wyliczonych wartości zwrotów:

```
print(V)
```