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

FrozenLake 2

In [26]:

```
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 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 **frozen_lake_slippery.py**):

In [27]:

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

Out[27]:

```
[(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:

In [28]:

```
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
```

```
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**:

In [29]:

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

Out[29]:

```
[(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**:

In [30]:

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

Out[30]:

```
[(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**:

In [31]:

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

Out[31]:

```
[(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 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:

In [32]:

```
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:

In [33]:

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

Out[33]:

```
0.25
```

Polecenie 2 (do uzupełnienia)

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

In [34]:

```
import numpy as np, numpy.random

my_stochastic_policy = np.ones([env.nS, env.nA]) #* np.random.dirichlet(np.ones(4),size=1)


for i in range(0, env.nS):
    rand = np.random.dirichlet(np.ones(4),size=1)
    sum = 0
    for j in range(0, env.nA):
        my_stochastic_policy[i,j] = round(rand[0,j], 2)
    for j in range(0, env.nA):
        sum += my_stochastic_policy[i,j]
    #print(round(sum, 1))


print(my_stochastic_policy)
print(stochastic_policy[0][0]+stochastic_policy[0][1]+stochastic_policy[0][2]+stochastic_policy[0][
3])
```

```
[[0.59 0.37 0.04 0.01]
 [0.11 0.11 0.28 0.5 ]
 [0.29 0.1  0.15 0.46]
 [0.04 0.54 0.38 0.04]
 [0.09 0.48 0.29 0.14]
 [0.44 0.02 0.21 0.33]
 [0.41 0.02 0.53 0.05]
 [0.1  0.24 0.28 0.38]
 [0.54 0.07 0.3  0.09]
 [0.13 0.6  0.27 0.  ]
 [0.46 0.37 0.01 0.17]
 [0.4  0.49 0.08 0.04]
 [0.1  0.18 0.26 0.47]
 [0.12 0.07 0.23 0.58]
 [0.05 0.58 0.1  0.26]
 [0.23 0.48 0.24 0.04]]
1.0
```

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:

In [35]:

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

Sprawdzamy:

In [36]:

```
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:

In [41]:

```
while True:
    delta = 0.0
    theta = 1e-8
    for state in range(env.nS):# env.nS=16
        delta = max(delta, np.abs(V[state] - Vs))
```

```
        delta = max(delta, np.abs(V[state] - Vs))
    #tutaj musimy wyliczyć nową wartość V(s)
    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:

In [43]:

```
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])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-43-20b09132c3e5> in <module>
      8     prob, next_state, reward, done = env.P[state][action][next_state]
      9
---> 10     Vs += policy[state][action] * prob * (reward + gamma * V[next_state])

NameError: name 'policy' is not defined
```

Teraz już możesz wykonać **Zadanie 3** z **RL_lab_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)**.

In [44]:

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

    while True:
        delta = .0
        for state in range(env.nS):
            #do uzupełnienia
            Vs = 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]
                    Vs += policy[state][action] * prob * (reward + gamma * V[next_state])

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

    return V
```

Użycie funkcji:

In [45]:

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

Wypisanie wyliczonych wartości zwrotów:

In [46]:

```
print(V)
```

```
[0.01393977 0.01163091 0.02095297 0.01047648 0.01624865 0.
 0.04075153 0.         0.03480619 0.08816993 0.14205316 0.
 0.         0.17582037 0.43929118 0.        ]
```