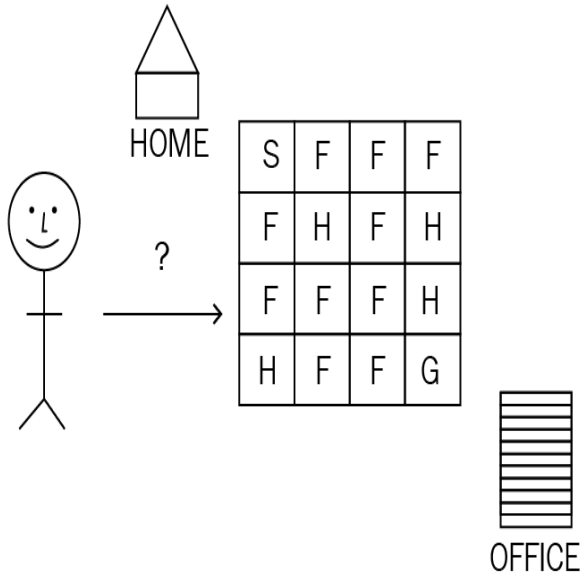
Solving The Frozen Lake Problem

The Problem Details

- Imagine there is a frozen lake stretching from your home to your office; you have to walk on the frozen lake to reach your office.
- But oops! There are holes in the frozen lake so you have to be careful while walking on the frozen lake to avoid getting trapped in the holes.



The symbols in the above picture:

- **S:** represents the strating position **(home)**
- **F:** represents the frozen lake where you can walk
- **H:** represents the holes, which you have to be so careful
- **G:** reprensents the goal **(office)**

The agent's goal : is to find the optimal path to go from **S** to **G** without getting trapped at **H**.

The **MDP** we have consists of the following:

- **States :** Set of states. Here we have 16 states (each little square box in the grid).
- **Actions :** Set of all possible actions (left, right, up, down; these are all the four possible actions our agent can take in our frozen lake environment).
- **Transition Function** $P^a_{s,s'}$ **:** The probability of moving from one state **(F)** to another state **(H)** by performing an action a.

- **Reward Function $R^a_{s,s'}$:** This is the expected reward we can recieve while moving from one state **(F)** to another state **(H)** by performing an action a.

The two possible ways to solve MDP in order to get the optimal policy:

- Value Iteration
- Policy Iteration

In [31]:
```
#import necessary libraries

import gym
import numpy as np
```

In [32]:
```
import math
```

In [33]:
```
#Make our frozen lake enviroment using OpenAI's Gym

env = gym.make('FrozenLake-v0')
```

```
[2020-08-19 17:33:57,743] Making new env: FrozenLake-v0
C:\Users\FADY-PC\anaconda3\lib\site-packages\gym\envs\registration.py:18: PkgRe
sourcesDeprecationWarning: Parameters to load are deprecated.  Call .resolve an
d .require separately.
  result = entry_point.load(False)
```

Exploring the Environment

In [34]:
```
#Explore the number of states in the enivronment
print(f'The number of states: {env.observation_space.n}')
```

```
The number of states: 16
```

In [35]:
```
#Explore the number of actions in the environment
print(f'The number of actions: {env.action_space.n}')
```

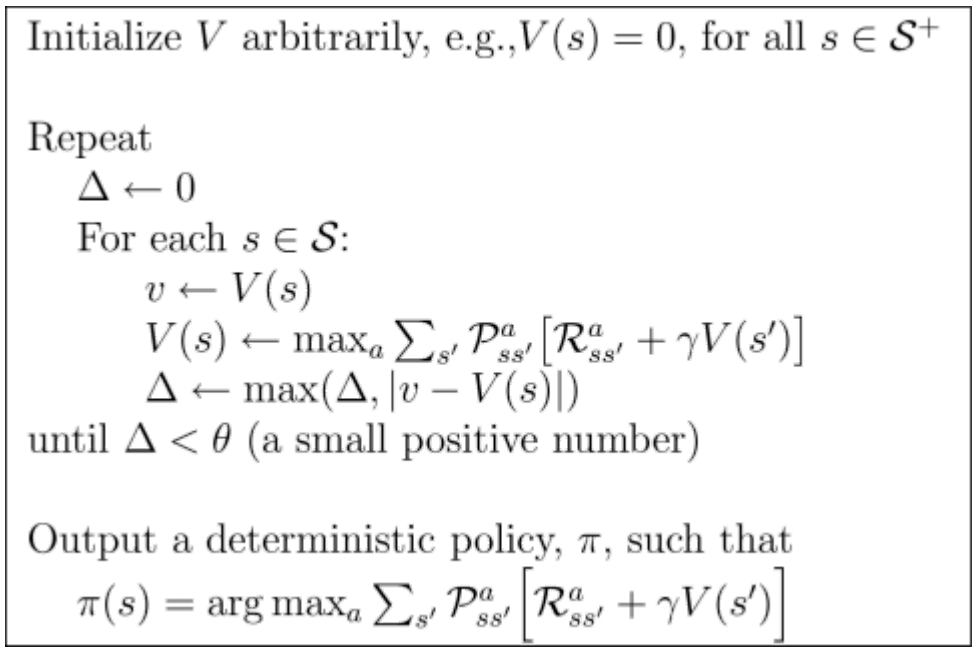
```
The number of actions: 4
```

Value Iteration

Recall the state-action value function:

$$Q(s,a) = \sum_{s'} [R^a_{s,s'} + V(s')]\, P^a_{s,s'} = \sum_{s'} next\ state\ reward\ for\ every$$

The Pseudo-code



The steps involved in the value iteration are as follows:

1. First, we initialize the random value function, that is, the random value for each state.
2. Then we compute the Q function for all state action pairs of Q(s, a).
3. Then we update our value function with the max value from Q(s,a).
4. We repeat these steps until the change in the value function is very small.

In [36]:
```
def value_iteration(env, gamma = 1.0):

    '''
    Usage:
      #value_iteration--> used for solving MDP by finding the optimal value funct

    Arguments:
      #env --> The enviroment that the agent interacts with
      #gamma --> represents the discount factor. The defalut value is 1.0

    Returns:
      #value_table -->is a the table of  the optimal value functions for all the
      #Q_value --> list of the updated values of the state-action value function
                  states

    '''


    #First, we initialize the random value table which is 0 for all the states
    value_table = np.zeros(env.observation_space.n)

    #Define the number of iterations
    no_of_iterations = 10000

    #Keep until convergence
    for i in range(no_of_iterations):
        #Upon starting each iteration, we copy the value_table to update_value_ta
        updated_value_table = np.copy(value_table)

        #For every space in the environment
        for state in range(env.observation_space.n):

            #instead of creating a Q-table for each state, we create a list listi
            #all the values of state-action pair
            Q_value = []

            #Scan every action
            for action in range(env.action_space.n):

                #Define empty list for storing next state reward for every transi
                next_states_rewards = []

                #Get some useful values of every state-action pair
                for next_sr in env.P[state][action]:

                    trans_prob, next_state, reward_func, _ = next_sr
                    #append the next-state reward value for that state-action pai
                    next_states_rewards.append((trans_prob * (reward_func + gamma

                #append the some of next_states_rewards for all the successor sta
                #every state-action pair
                Q_value.append(np.sum(next_states_rewards))

            #Pick up the maximum Q value and update it as value of a state
            value_table[state] = max(Q_value)
```

```
        #check if we have reached the convergence, that is, the difference of
        #the value-function between each iteration is small
        #at a result of that, we define a threshold ,a cutoff value, which we sto
        #the difference at least equals it
        threshold = 1e-10

        if(np.sum(np.fabs(updated_value_table - value_table)) <= threshold):
            print(f"Value-iteration converged at iteration: {i+1}")
            break

    return value_table, Q_value
```

In [37]:
```
#We can get the optimal value-function for all states(value_table) and the corres
value_table, Q_value  = value_iteration(env = env, gamma= 1.0)
```

```
Value-iteration converged at iteration: 877
```

In [38]:
```
#Print the values of the optimal value function
print(value_table)
```

```
[0.82352941 0.82352941 0.82352941 0.82352941 0.82352941 0.
 0.52941176 0.         0.82352941 0.82352941 0.76470588 0.
 0.         0.88235294 0.94117647 0.         ]
```

In [39]:
```
#Print the Q_values
print(Q_value)
```

```
[0.0, 0.0, 0.0, 0.0]
```

After finding the optimal value function, how can we extract the optimal policy from the optimal_value_function?

- We calculate the Q value using our optimal value action and pick up the actions which have the highest Q value for each state as the optimal policy.
- We do this via a function called **extract_policy()**

Note:

The functions, value_iteration() and extrac_policy(), **together** represent **the value iteration algorithm**, but in the lab we divide the algorithm into two functions just for explanation but we can encapsulate the two functions into one and call it value_iteraion().

In [46]:
```
def extract_policy(value_table,gamma = 1.0):
    '''
    Usage:
      #extract_policy--> used for getting the optimal policy

    Arguments:
      #value_table -->is a the table of  the optimal value functions for all the
      #gamma --> represents the discount factor. The defalut value is 1.0

    Returns:
      #policy--> list represents the best actions to perform for each state we th

    '''

    #Define a random policy for all the states
    policy = np.zeros(env.observation_space.n)

    #for each state , we build Q_table holds the possible action we can perform f
    for state in range(env.observation_space.n):

        #pre-allocating the Q_table
        Q_table = np.zeros(env.action_space.n)

        #for each action in the state, we compute the its state-action value func
        #And, append it to Q_table
        for action in range(env.action_space.n):
            #Get some useful values of every state-action pair
                for next_sr in env.P[state][action]:

                    trans_prob, next_state, reward_func, _ = next_sr

                    #Compute state-action value function for a certain (state,act
                    #And add it to Q_table
                    Q_table[action] += trans_prob * (reward_func + gamma * value_


        #Then Pick the action that has the highest Q_value for a specific state
        policy[state] = np.argmax(Q_table)


    return policy
```

In [47]:
```
#Get the optimal policy for each state
optimal_policy = extract_policy(value_table)
```

In [48]:
```
#Print the optimal_policy
optimal_policy
```

Out[48]:
```
array([0., 3., 3., 3., 0., 0., 0., 0., 3., 1., 0., 0., 0., 2., 1., 0.])
```

Policy Iteration

The Pseudo-code

Initialize a policy π' arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using π by
        solving the linear equations
            $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$

The steps involved in the policy iteration are as follows: :

1. First, we initialize some random policy
2. Then we find the value function for that random policy and evaluate to check if it is optimal which is called policy evaluation
3. If it is not optimal, we find a new improved policy, which is called policy improvement
4. We repeat these steps until we find an optimal policy

In [86]:
```
def compute_value_function(policy,gamma = 1.0):

    '''
    Usage:
      #compute_value_function--> Compute value function given a policy

    Arguments:
      #policy --> the policy that we compute the value fuction based on it
      #gamma --> represents the discount factor. The defalut value is 1.0

    Returns:

      #value_table--> is a the table of  the updated value functions for all the
                      to a better policy using them

    '''

    #first, we initialize the value_table as zeros for all the states
    value_table = np.zeros(env.observation_space.n)

    #check if we have reached the convergence, that is, the difference of
    #the value-function between each iteration is small
    #at a result of that, we define a threshold ,a cutoff value, which we stop up
    #the difference at least equals it
    threshold = 1e-20


    #Keep going until convergence
    while(True):

        #for each state, we get an action to perfom under the random policy we ir
        #we compute the value function according to that action and the state
        #we make a new variable called updated_value_table to update the value to
        updated_value_table = np.copy(value_table)

        #for every state in the environment
        for state in range(env.observation_space.n):

            #get the action the agent must perform under that policy
            action = policy[state]

            #compute the value_function under the given policy
            value_table[state] = sum([trans_prob * (reward_func + gamma * updated



        #Check for cenvergence
        if(np.sum(np.fabs(updated_value_table - value_table)) <= threshold):
            break


    return value_table
```

In [73]:
```
def policy_iteration(env,gamma = 1.0):
    '''
    Usage:
      #policy_iteration--> used for solving MDP by finding the optimal policy the


    Arguments:
      #env --> The enviroment that the agent interacts with
      #gamma --> represents the discount factor. The defalut value is 1.0

    Returns:
      #new_policy --> new_policy --> represents array of the optimal policy for e

    '''

    #first, we initialize a random policy
    random_policy = np.zeros(env.observation_space.n)

    #define the number of iterations
    no_of_iterations = 20000

    #then, for each iteration we calculate the new value function corresponding t
    #Extract the optimal policy for each state using the new value function we co
    for i in range(no_of_iterations):
        new_value_function = compute_value_function(random_policy,gamma = 1.0)
        new_policy = extract_policy(new_value_function)

        #then, we check whether we have reached convergence
        #we do that by comparing the random policy with the new policy
        #if the random policy equals the policy we break
        #else, we update the random_policy with the new policy we get from new va
        if (np.all(random_policy == new_policy)):
            print(f"Policy Iteration converged at iteration: {i+1}")
            break

        random_policy = new_policy

    return new_policy
```

In [87]:
```
#Get the optimal Policy for each state
optimal_policy = policy_iteration(env)
```

```
Policy Iteration converged at iteration: 7
```

In [88]:
```
#Explore the optimal policy
optimal_policy
```

Out[88]:
```
array([0., 3., 3., 3., 0., 0., 0., 0., 3., 1., 0., 0., 0., 2., 1., 0.])
```

Congratulations!