Q1 (i). Towers of Hanoi.

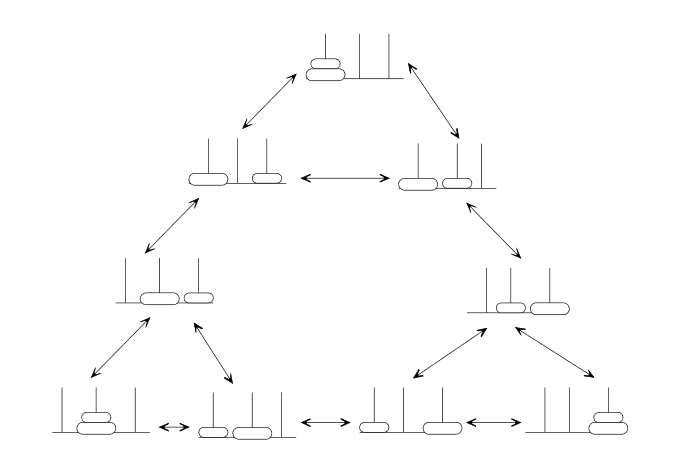
**Problem Statement**:

Tower of Hanoi consists of three pegs (or towers) with *n* disks placed one over the other.

The objective is to move the stack to another tower following these simple rules.

1. Only one disk can be moved at a time.
2. No disk can be placed on top of the smaller disk.

**State Space Representation**:



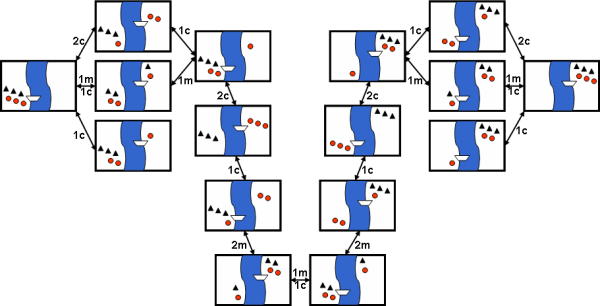
(ii). Missionaries and Cannibals

**Problem Statement**:

*On one bank of a river are three missionaries and three cannibals. There is one boat available that can hold up to two people and that they would like to use to cross the river. If the cannibals ever outnumber the missionaries on either of the river’s banks, the missionaries will get eaten.*

*How can the boat be used to safely carry all the missionaries and cannibals across the river?*

**State Space Representation**:



**Problem Solving**:

By looking at the graphical state space representation of a problem, we might be able to come up with a representation that we can use in our algorithm, which can give us an idea of how large our state space is and what kind of algorithm will be fit for it.

For example, in our graphical representation of Towers of Hanoi, we have three towers and two disks. So we can represent it as below:

2, 1

0, 0 ← current state (both disk 1 and 2 are on the first tower)

So other possible states will be like: (0, 1), (1, 0), (1, 1), (0, 2), (2, 0), (1, 2), (2, 1), (2, 2). So we can see that the size of state space will be 3n. Where n is the number of disks. That is an exponential state space, so depending on the number of disks, we can choose a search algorithm. For example, for a small number of disks, we may select DFS or BFS, and for a large number we may select a heuristic-based intelligent search.

from os import system

from random import random

from copy import deepcopy

class Player:

def \_\_init\_\_(self, name, type, symbol):

self.name = name

self.type = type

self.symbol = symbol

class Game:

def \_\_init\_\_(self, player1, player2):

self.players = []

self.players.append(player1)

self.players.append(player2)

self.matrix = [

1, 2, 3,

4, 5, 6,

7, 8, 9

]

self.turns\_taken = 0

def start(self):

while self.turns\_taken < 9:

curr\_player = self.turns\_taken % 2

choice = -1 # -1 is invalid choice

while self.is\_valid\_move(choice - 1) == False:

self.print\_matrix(self.matrix)

print(self.players[curr\_player].name + "'s turn: ")

choice = self.get\_player\_input(self.players[curr\_player])

system("cls")

# mark player's symbol on matrix

self.matrix[choice - 1] = self.players[curr\_player].symbol

# check victory

if (self.check\_victory(self.matrix, self.players[curr\_player].symbol)):

self.print\_matrix(self.matrix)

print(self.players[curr\_player].name + " has won!")

return

self.turns\_taken += 1

# if the loop completes, it means it's a draw

print("It's a draw!")

return

def print\_matrix(self, matrix):

for i in range(3):

for j in range(3):

print(matrix[i \* 3 + j], end='')

if j < 2:

print(" | ", end='')

else:

print("")

if i < 2:

print("---------")

# matches 1st to 3rd horizontal lines. Then 1st to 3rd vertical lines. And then left-to-right diagonal, followed by right-to-left diagonal.

def check\_victory(self, matrix, symbol):

m = matrix

if m[0] == m[1] == m[2] == symbol or \

m[3] == m[4] == m[5] == symbol or \

m[6] == m[7] == m[8] == symbol or \

m[0] == m[3] == m[6] == symbol or \

m[1] == m[4] == m[7] == symbol or \

m[2] == m[5] == m[8] == symbol or \

m[0] == m[4] == m[8] == symbol or \

m[2] == m[4] == m[6] == symbol:

return True

return False

def is\_draw(self, matrix):

for i in range(9):

if matrix[i] != 'X' and matrix[i] != 'O':

return False

return True

# checks if a matrix cell is already marked or not

def is\_valid\_move(self, index):

if index < 0 or index > 8:

return False

if self.matrix[index] == 'X' or self.matrix[index] == 'O':

return False

return True

# handles input based on whether the player is human or computer

def get\_player\_input(self, player):

if player.type == "human":

return int(input())

elif player.type == "computer":

return self.minimax(self.matrix, self.players.index(player), -10000, 10000)['index']

def get\_neighbours(self, matrix, symbol):

neighbours = []

for i in range(9):

if matrix[i] != 'X' and matrix[i] != 'O':

new\_matrix = deepcopy(matrix)

new\_matrix[i] = symbol

neighbours.append({'cell\_replaced': i + 1, 'matrix': new\_matrix})

return neighbours

# minimax WITH alpha-beta pruning

def minimax(self, matrix, player\_index, alpha, beta):

curr\_player = self.players[player\_index]

# terminal states

if self.check\_victory(matrix, 'O'):

return {'score': 10}

elif self.check\_victory(matrix, 'X'):

return {'score': -10}

elif self.is\_draw(matrix):

return {'score': 0}

# not a terminal state. So we go through all possible moves

scores = []

neighbours = self.get\_neighbours(matrix, curr\_player.symbol)

if curr\_player.type == "computer":

best = -10000

for neighbour in neighbours:

score = self.minimax(neighbour['matrix'], (player\_index + 1) % 2, alpha, beta)['score']

scores.append({'index': neighbour['cell\_replaced'], 'score': score})

best = max(best, score)

alpha = max(alpha, best)

if beta <= alpha:

break

elif curr\_player.type == "human":

best = 10000

for neighbour in neighbours:

score = self.minimax(neighbour['matrix'], (player\_index + 1) % 2, alpha, beta)['score']

scores.append({'index': neighbour['cell\_replaced'], 'score': score})

best = min(best, score)

beta = min(beta, best)

if beta <= alpha:

break

# evaluate the scores

best\_move = -1

if curr\_player.type == "computer":

max\_score = -10000

for i in range(len(scores)):

if max\_score < scores[i]['score']:

max\_score = scores[i]['score']

best\_move = i

elif curr\_player.type == "human":

min\_score = 10000

for i in range(len(scores)):

if min\_score > scores[i]['score']:

min\_score = scores[i]['score']

best\_move = i

return scores[best\_move]

# minimax WITHOUT alpha-beta pruning

def \_minimax(self, matrix, player\_index):

curr\_player = self.players[player\_index]

# terminal states

if self.check\_victory(matrix, 'O'):

return {'score': 10}

elif self.check\_victory(matrix, 'X'):

return {'score': -10}

elif self.is\_draw(matrix):

return {'score': 0}

# not a terminal state. So we go through all possible moves

scores = []

neighbours = self.get\_neighbours(matrix, curr\_player.symbol)

for neighbour in neighbours:

scores.append({'index': neighbour['cell\_replaced'], 'score': self.\_minimax(neighbour['matrix'], (player\_index + 1) % 2)['score']})

# evaluate the scores

best\_move = -1

if curr\_player.type == "computer":

max\_score = -10000

for i in range(len(scores)):

if max\_score < scores[i]['score']:

max\_score = scores[i]['score']

best\_move = i

elif curr\_player.type == "human":

min\_score = 10000

for i in range(len(scores)):

if min\_score > scores[i]['score']:

min\_score = scores[i]['score']

best\_move = i

return scores[best\_move]

def main():

p1 = Player("John", "human", "X")

p2 = Player("Comp", "computer", "O")

game = Game(p1, p2)

game.start()

if \_\_name\_\_ == "\_\_main\_\_":

main()