# LAB EXPERIMENTS

## LAB EXPERIMENT 1

**OBJECTIVE:**

Write a Program to Implement Breadth First Search using Python.

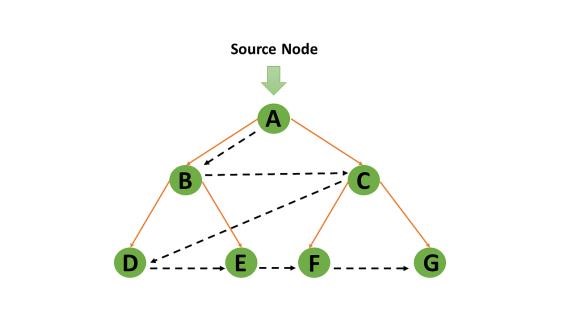
**BRIEF DESCRIPTION:**

Breadth-First Search Algorithm or BFS is the most widely utilized method.

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the nextlevel neighbor nodes.

According to the BFS, you must traverse the graph in a breadthwise direction:

* To begin, move horizontally and visit all the current layer's nodes.
* Continue to the next layer.



Breadth-First Search uses a [queue data structure](https://www.simplilearn.com/tutorials/python-tutorial/queue-in-python) to store the node and mark it as "visited" until it marks all the neighboring vertices directly related to it. The queue operates on the First in First out (FIFO) principle, so the node's neighbors will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

**PRE-EXPERIMENT QUESTIONS:**

1. What is searching?
2. What is QUEUE data structure?

**Explanation:** graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

} visited = [] # List for visited nodes. queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS visited.append(node) queue.append(node)

while queue: # Creating loop to visit each node m = queue.pop(0) print (m, end = " ")

for neighbour in graph[m]: if neighbour not in visited: visited.append(neighbour) queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search") bfs(visited, graph, '5') # function calling **Output:**



**POST EXPERIMENT QUESTIONS:**

1 .What do you understand by BFS?

2. What is the time and space complexity of BFS?

## LAB EXPERIMENT 2

**OBJECTIVE:**

Write a Program to Implement Depth First Search using Python.

**BRIEF DESCRIPTION:**

DFS is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depthfirst search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.

The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

* First, create a stack with the total number of vertices in the graph.
* Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
* After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
* Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
* If no vertex is left, go back and pop a vertex from the stack.
* Repeat steps 2, 3, and 4 until the stack is empty.

**PRE EXPERIMENT QUESTIONS:**

1. What is tree and graph data structure?
2. What do you understand by Stack data structure?

**Explanation:**

# Using a Python dictionary to act as an adjacency list graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs if node not in visited: print (node, end=" ") visited.add(node) for neighbour in graph[node]: dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search") dfs(visited, graph, '5')

**Output:**



**POST EXPERIMENT QUESTIONS:**

1. What do you understand by DFS?
2. What is the time and space complexity of DFS?
3. What is the difference between BFS & DFS?

## LAB EXPERIMENT 3

**OBJECTIVE:**

Write a Program to Implement Tic-Tac-Toe game using Python.

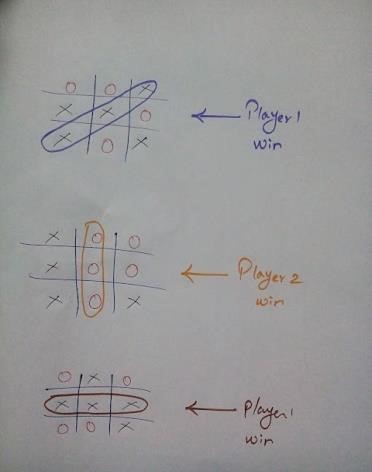
**BRIEF DESCRIPTION:**

The game Tic Tac Toe is also known as Noughts and Crosses or Xs and Os ,the player needs to take turns marking the spaces in a 3x3 grid with their own marks,if 3 consecutive marks (Horizontal, Vertical,Diagonal) are formed then the player who owns these moves get won.

Assume ,

Player 1 - X

Player 2 - O



So,a player who gets 3 consecutive marks first, they will win the game .

**PRE EXPERIMENT QUESTIONS:**

1. What do you understand by BFS?
2. What do you understand by DFS?

**Explanation:**

# Function to print Tic Tac Toe def print\_tic\_tac\_toe(values): print("\n")

print("\t | |") print("\t {} | {} | {}".format(values[0], values[1], values[2]))

print('\t\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_')

print("\t | |")

print("\t {} | {} | {}".format(values[3], values[4], values[5])) print('\t\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_')

print("\t | |")

print("\t {} | {} | {}".format(values[6], values[7], values[8]))

print("\t | |") print("\n")

# Function to print the score-board def print\_scoreboard(score\_board):

print("\t--------------------------------") print("\t SCOREBOARD ") print("\t--------------------------------")

players = list(score\_board.keys()) print("\t ", players[0], "\t ", score\_board[players[0]]) print("\t ", players[1], "\t ", score\_board[players[1]])

print("\t--------------------------------\n")

# Function to check if any player has won def check\_win(player\_pos, cur\_player):

# All possible winning combinations soln = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 4, 7], [2, 5, 8], [3, 6, 9], [1, 5, 9], [3, 5, 7]]

# Loop to check if any winning combination is satisfied for x in soln: if all(y in player\_pos[cur\_player] for y in x):

# Return True if any winning combination satisfies return True

# Return False if no combination is satisfied

return False

# Function to check if the game is drawn def check\_draw(player\_pos): if len(player\_pos['X']) + len(player\_pos['O']) == 9:

return True return False

# Function for a single game of Tic Tac Toe def single\_game(cur\_player):

# Represents the Tic Tac Toe values = [' ' for x in range(9)]

# Stores the positions occupied by X and O

player\_pos = {'X':[], 'O':[]}

# Game Loop for a single game of Tic Tac Toe while True:

print\_tic\_tac\_toe(values)

# Try exception block for MOVE input

try:

print("Player ", cur\_player, " turn. Which box? : ", end="") move = int(input()) except ValueError:

print("Wrong Input!!! Try Again")

continue

# Sanity check for MOVE inout if move < 1 or move > 9:

print("Wrong Input!!! Try Again")

continue

# Check if the box is not occupied already if values[move-1] != ' ':

print("Place already filled. Try again!!") continue

# Update game information

# Updating grid status values[move-1] = cur\_player # Updating player positions player\_pos[cur\_player].append(move)

# Function call for checking win if check\_win(player\_pos, cur\_player):

print\_tic\_tac\_toe(values) print("Player ", cur\_player, " has won the game!!")

print("\n") return cur\_player

# Function call for checking draw game if check\_draw(player\_pos): print\_tic\_tac\_toe(values) print("Game Drawn")

print("\n") return 'D'

# Switch player moves

if cur\_player == 'X': cur\_player = 'O' else:

cur\_player = 'X'

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

print("Player 1") player1 = input("Enter the name : ") print("\n")

print("Player 2") player2 = input("Enter the name : ") print("\n")

# Stores the player who chooses X and O cur\_player = player1

# Stores the choice of players player\_choice = {'X' : "", 'O' : ""}

# Stores the options options = ['X', 'O']

# Stores the scoreboard score\_board = {player1: 0, player2: 0} print\_scoreboard(score\_board)

# Game Loop for a series of Tic Tac Toe # The loop runs until the players quit while True:

# Player choice Menu print("Turn to choose for", cur\_player) print("Enter 1 for X") print("Enter 2 for O") print("Enter 3 to Quit")

# Try exception for CHOICE input

try:

choice = int(input()) except ValueError: print("Wrong Input!!! Try Again\n")

continue

# Conditions for player choice if choice == 1:

player\_choice['X'] = cur\_player if cur\_player == player1: player\_choice['O'] = player2

else:

player\_choice['O'] = player1

elif choice == 2:

player\_choice['O'] = cur\_player if cur\_player == player1: player\_choice['X'] = player2

else:

player\_choice['X'] = player1

elif choice == 3: print("Final Scores") print\_scoreboard(score\_board)

break

else:

print("Wrong Choice!!!! Try Again\n") # Stores the winner in a single game of Tic Tac Toe winner = single\_game(options[choice-1]) # Edits the scoreboard according to the winner

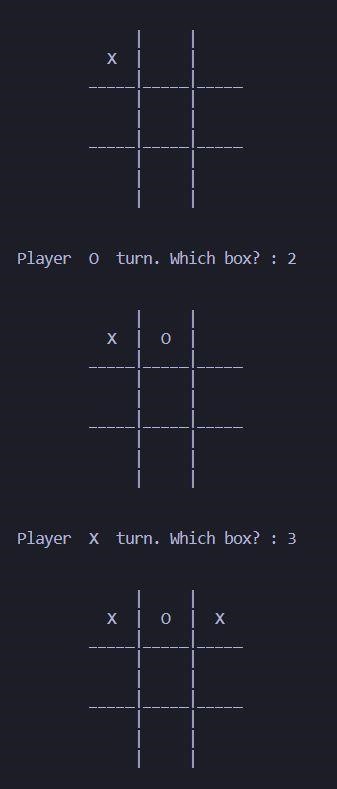
if winner != 'D' :

player\_won = player\_choice[winner] score\_board[player\_won] = score\_board[player\_won] + 1 print\_scoreboard(score\_board) # Switch player who chooses X or O if cur\_player == player1: cur\_player = player2

else:

cur\_player = player1

**Output:**



**POST EXPERIMENT QUESTIONS:**

1. How to implement tic-tac-toe using BFS?
2. How to implement tic-tac-toe using DFS?

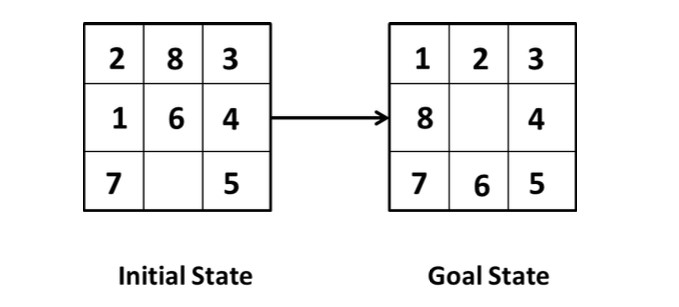
## LAB EXPERIMENT 4

**OBJECTIVE:**

Write a Program to implement 8-Puzzle problem using Python.

**BRIEF DESCRIPTION:**

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.



The program is to change the initial configuration into the goal configuration.

A solution to the problem is an appropriate sequence of moves, such as “move tile 5 to the right, move tile 7 to the left, move tile 6 to the down” etc…

To solve a problem, we must specify the global database, the rules, and the control strategy.

For the 8 puzzle problem that correspond to three components.

These elements are the problem states, moves and goal.

In this problem each tile configuration is a state.

The set of all possible configuration in the problem space, consists of 3,62,880 different configurations of the 8 tiles and blank space.

For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. Initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

A move transforms one problem state into another state.

**PRE EXPERIMENT QUESTIONS:**

1. What do you understand by BFS?
2. What do you understand by DFS?
3. What is branch and bound?

**Explanation:** import copy from heapq import heappush, heappop

n = 3 row = [ 1, 0, -1, 0 ] col = [ 0, -1, 0, 1 ] class priorityQueue:

def \_init\_(self):

self.heap = []

def push(self, k): heappush(self.heap, k) def pop(self):

return heappop(self.heap) def empty(self): if not self.heap:

return True else: return False

class node:

def \_init\_(self, parent, mat, empty\_tile\_pos,

cost, level): self.parent = parent

self.mat = mat

self.empty\_tile\_pos = empty\_tile\_pos

self.cost = cost self.level = level def \_lt\_(self, nxt):

return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n): if ((mat[i][j]) and (mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node: new\_mat = copy.deepcopy(mat)

x1 = empty\_tile\_pos[0] y1 = empty\_tile\_pos[1] x2 = new\_empty\_tile\_pos[0] y2 = new\_empty\_tile\_pos[1] new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1] cost = calculateCost(new\_mat, final) new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

def printMatrix(mat): for i in range(n):

for j in range(n): print("%d " % (mat[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

def solve(initial, empty\_tile\_pos, final):

pq = priorityQueue()

cost = calculateCost(initial, final) root = node(None, initial,

empty\_tile\_pos, cost, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop() if minimum.cost == 0:

printPath(minimum)

return

for i in range(4):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i], minimum.empty\_tile\_pos[1] + col[i], ] if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

child = newNode(minimum.mat,

minimum.empty\_tile\_pos, new\_tile\_pos,

minimum.level + 1, minimum, final,) pq.push(child)

initial = [ [ 1, 2, 3 ],

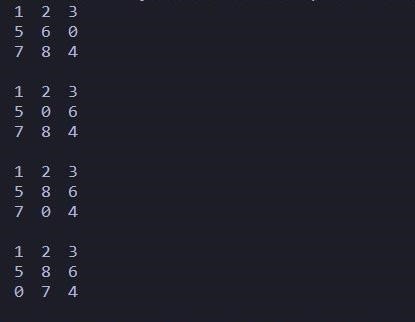
[ 5, 6, 0 ], [ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ], [ 0, 7, 4 ] ]

empty\_tile\_pos = [ 1, 2 ] solve(initial, empty\_tile\_pos, final)

**Output:**



**POST EXPERIMENT QUESTIONS:**

1. Which algorithm is used in 8-Puzzle problem?
2. How many states are there in the 8-puzzle problem?
3. What are the components of 8-puzzle problem?

## LAB EXPERIMENT 6

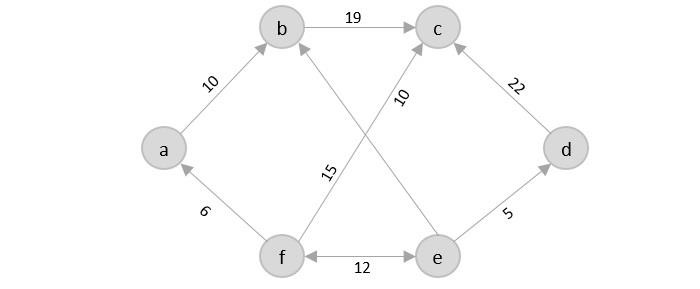
**OBJECTIVE:**

Write a Program to Implement Travelling Salesman Problem using Python.

**BRIEF DESCRIPTION:**

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the vertex ‘a’, they need to travel through all the remaining vertices b, c, d, e, f and get back to ‘a’ while making sure that the cost taken is minimum.



**PRE EXPERIMENT QUESTIONS:**

1. What is Brute –Force Searching?
2. What do you understand by Greedy approach?
3. What do you understand by Dynamic approach?

**Explanation:**

from sys import maxsize from itertools import permutations V = 4 def travellingSalesmanProblem(graph, s):

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

min\_path = maxsize next\_permutation=permutations(vertex) for i in next\_permutation:

current\_pathweight = 0

k = s

for j in i: current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][s]

min\_path = min(min\_path, current\_pathweight)

return min\_path

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

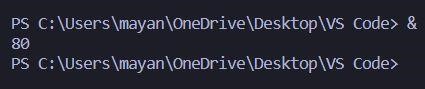
graph = [[0, 10, 15, 20], [10, 0, 35, 25],

[15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(travellingSalesmanProblem(graph, s))

**Output:**



**POST EXPERIMENT QUESTIONS:**

1. What are the practical applications of the travelling salesman problem?
2. What is the best algorithm for the travelling salesman problem?
3. How many possible routes are there in travelling salesman problem?
4. What is the time complexity of travelling salesman problem?

## LAB EXPERIMENT 7

**OBJECTIVE:**

Write a Program to Implement Tower of Hanoi using Python.

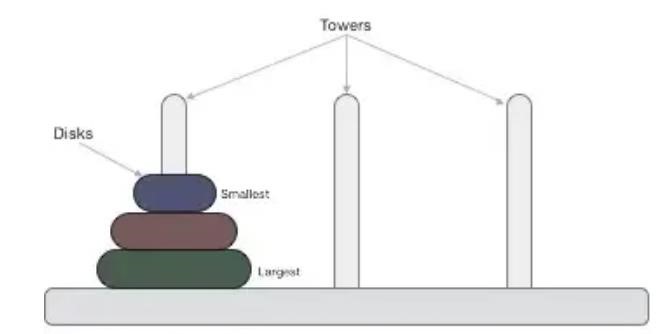
**BRIEF DESCRIPTION:**

Tower of Hanoi is mathematical game puzzle where we have three pile (pillars) and n numbers of disk.

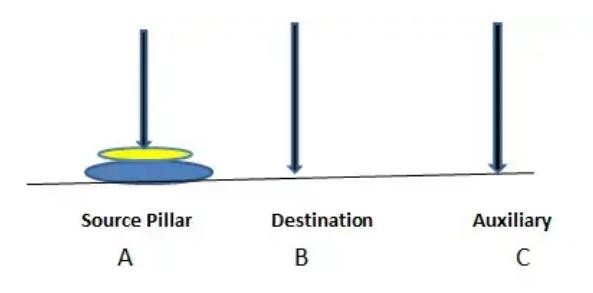
This game has some rules (Rules of game)

* Only one disk will move at a time.
* The larger disk should always be on the bottom and the smaller disk on top of it.(Even during intermediate move)
* Move only the uppermost disk.
* All disk move to destination pile from source pile.

So, here we are trying to solve that how many moves are required to solve a problem (It depends on number of disk).



### When we have two disk and 3 pillars (pile, A, B, C)



In the above diagram, following the rule of the game our target is move the disks from source pile (pillar) to the destination pillar. (Let’s take a look how many steps/ moves are required to make this happen).

Step1: Move small disk to the auxiliary pillar (A).

Step2: Move large disk to the Destination pillar (B).

Step3: Move small disk to the Destination pillar (B).4

So, basically when we have 2 disks we required 3 move to reach the destination.

What if we have 3, 4, 5...n disks?

Eventually, you figure out that there is some pattern to the puzzle and with each increment in disks; the pattern could be followed recursively.

Total move required to reach destination pillar is formula of moves means if we have 3 disks we required (4 moves to reach destination pillar), if 4 disks 8 moves required and so on...

**PRE EXPERIMENT QUESTIONS:**

1. What do you understand by Recursion?
2. What is Backtracking?

**Explanation:** class Tower:

def \_\_init\_\_(self): self.terminate = 1 def printMove(self, source, destination):

print("{} -> {}".format(source, destination)) def move(self, disc, source, destination, auxiliary):

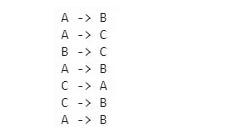
if disc == self.terminate:

self.printMove(source, destination) else:

self.move(disc - 1, source, auxiliary, destination) self.move(1, source, destination, auxiliary) self.move(disc - 1, auxiliary, destination, source) t = Tower();

t.move(3, 'A', 'B', 'C')

**Output:**



**POST EXPERIMENT QUESTIONS:**

1. What is the practical use of Tower of Hanoi?
2. Which data structure is used in the Tower of Hanoi?
3. Can we solve Tower of Hanoi problem without recursion?
4. Which algorithm approach is used to solve the Tower of Hanoi problem?

## LAB EXPERIMENT 8

**OBJECTIVE:**

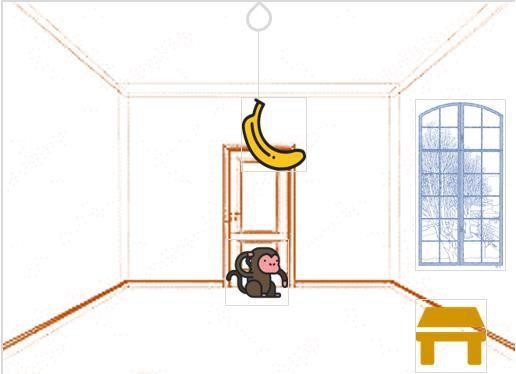
Write a Program to Implement Monkey Banana Problem using Python.

**BRIEF DESCRIPTION:**

The monkey and banana problem is a famous toy problem in artificial intelligence, particularly in logic programming and planning. A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey? The problem seeks to answer the question of whether monkeys are intelligent. Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter, or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other. Monkeys have the ability not only to remember how to hunt and gather but to learn new things, as is the case with the monkey and the bananas: despite the fact that the monkey may never have been in an identical situation, with the same artifacts at hand, a monkey is capable of concluding that it needs to make a ladder, position it below the bananas, and climb up to reach for them.

Suppose the problem is as given below −

* A hungry monkey is in a room, and he is near the door.
* The monkey is on the floor.
* Bananas have been hung from the center of the ceiling of the room.
* There is a block (or chair) present in the room near the window.
* The monkey wants the banana, but cannot reach it.



**So how can the monkey get the bananas?**

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case − • Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.

* If the block position is not at the center, then monkey can drag it to the center.
* If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
* When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

**PRE EXPERIMENT QUESTIONS:**

1. What is logic programming?
2. What do you understand by Planning?

**Explanation:** from poodle import Object, schedule from typing import Set class Position(Object): def \_\_str\_\_(self): if not hasattr(self, "locname"): return "unknown" return self.locname class HasHeight(Object): height: int class HasPosition(Object):

at: Position

class Monkey(HasHeight, HasPosition): pass class PalmTree(HasHeight, HasPosition):

def \_\_init\_\_(self, \*args, \*\*kwargs): super().\_\_init\_\_(\*args, \*\*kwargs) self.height = 2 class Box(HasHeight, HasPosition): pass class Banana(HasHeight, HasPosition):

owner: Monkey attached: PalmTree class World(Object): locations: Set[Position] p1 = Position() p1.locname = "Position A" p2 = Position() p2.locname = "Position B" p3 = Position() p3.locname = "Position C" w = World()

w.locations.add(p1)

w.locations.add(p2)

w.locations.add(p3) m = Monkey()

m.height = 0 # ground

m.at = p1 box = Box() box.height = 2 box.at = p2 p = PalmTree() p.at = p3 b = Banana()

b.attached = p

def go(monkey: Monkey, where: Position):

assert where in w.locations

assert monkey.height < 1, "Monkey can only move while on the ground" monkey.at = where

return f"Monkey moved to {where}" def push(monkey: Monkey, box: Box, where: Position):

assert monkey.at == box.at assert where in w.locations

assert monkey.height < 1, "Monkey can only move the box while on the ground" monkey.at = where box.at = where

return f"Monkey moved box to {where}" def climb\_up(monkey: Monkey, box: Box):

assert monkey.at == box.at monkey.height += box.height return "Monkey climbs the box" def grasp(monkey: Monkey, banana: Banana):

assert monkey.height == banana.height assert monkey.at == banana.at banana.owner = monkey return "Monkey takes the banana" def infer\_owner\_at(palmtree: PalmTree, banana: Banana):

assert banana.attached == palmtree banana.at = palmtree.at

return "Remembered that if banana is on palm tree, its location is where palm tree is" def infer\_banana\_height(palmtree: PalmTree, banana: Banana):

assert banana.attached == palmtree

banana.height = palmtree.height

return "Remembered that if banana is on the tree, its height equals tree's height" print('\n'.join(x() for x in schedule(

[go, push, climb\_up, grasp, infer\_banana\_height, infer\_owner\_at],

[w,p1,p2,p3,m,box,p,b], goal=lambda: b.owner == m)))

**Output:**

$ pip install poodle

$ python ./monkey.py

Monkey moved to Position B

Remembered that if banana is on the tree, its height equals tree's height

Remembered that if banana is on palm tree, its location is where palm tree is

Monkey moved box to Position C

Monkey climbs the box

Monkey takes the banana

**POST EXPERIMENT QUESTIONS:**

1. How do you solve monkey banana problem?
2. Which algorithm is used to solve monkey banana problem?

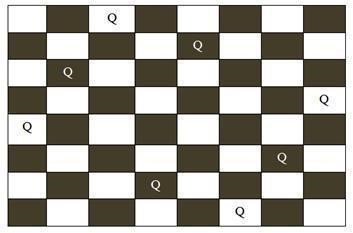
## LAB EXPERIMENT 10

**OBJECTIVE:**

Write a Program to implement 8-Queens Problem using Python.

**BRIEF DESCRIPTION:**

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. There are 92 solutions. We have 8 queens and an 8x8 Chess board having al ternate black and white squares. The queens are placed on the chessboard. Any queen can attack any other queen placed on same row, or column or diagonal. We have to find the proper placement of queens on the Chess board in such a way that no queen attacks other queen”.



In figure, the possible board configuration for 8-queen problem has been shown. The board has alternative black and white positions on it. The different positions on the board hold the queens.

The production rule for this game is you cannot put the same queens in a same row or same column or in same diagonal. After shifting a single queen from its position on the board, the user have to shift other queens according to the production rule.

Starting from the first row on the board the queen of their corresponding row and column are to be moved from their original positions to another position. Finally the player has to be ensured that no rows or columns or diagonals of on the table are same.

**PRE EXPERIMENT QUESTIONS:**

1. What is backtracking?
2. What do you understand by branch and bound?

**Explanation:**

# Python program to solve N Queen problem global N N = 4 def printSolution(board): for i in range(N): for j in range(N): print board[i][j], print def isSafe(board, row, col): # Check this row on left side for i in range(col): if board[row][i] == 1:

return False

# Check upper diagonal on left side for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1: return False

# Check lower diagonal on left side for i, j in zip(range(row, N, 1), range(col, -1, -1)): if board[i][j] == 1: return False return True def solveNQUtil(board, col):

# base case: If all queens are placed # then return true if col >= N: return True for i in range(N): if isSafe(board, i, col): # Place this queen in board[i][col] board[i][col] = 1 # recur to place rest of the queens if solveNQUtil(board, col + 1) == True:

return True board[i][col] = 0 return False def solveNQ(): board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False: print "Solution does not exist" return False printSolution(board) return True

# driver program to test above function solveNQ()

**Output:**

1. 0 1 0
2. 0 0 0

0 0 0 1

0 1 0 0

**POST EXPERIMENT QUESTIONS:**

1. What is the best method to solve the 8 queen problem?
2. What is the best method to solve the 8 queen problem?