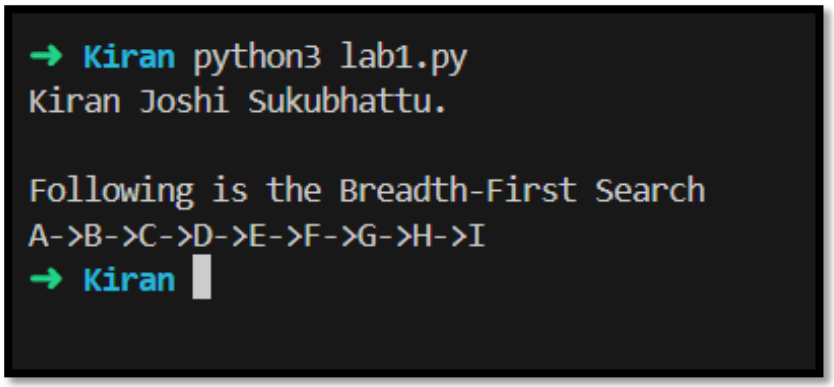
LAB 1
Breadth First Search

AIM: To Implement Breadth First Search using Python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': ['G', 'H'],
   'E': ['I'],
   'F': [],
   'G': [],
   'H': [],
   'I': []
}
visited = []
queue = []

def bfs(visited, graph, node):
   visited.append(node)
   queue.append(node)

   while queue:
      m = queue.pop(0)
      # print '->' after each node except the last one
      print(m, end='->' if m != 'I' else '\n')

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

print("\nFollowing is the Breadth-First Search")
bfs(visited, graph, 'A')
```

Output:



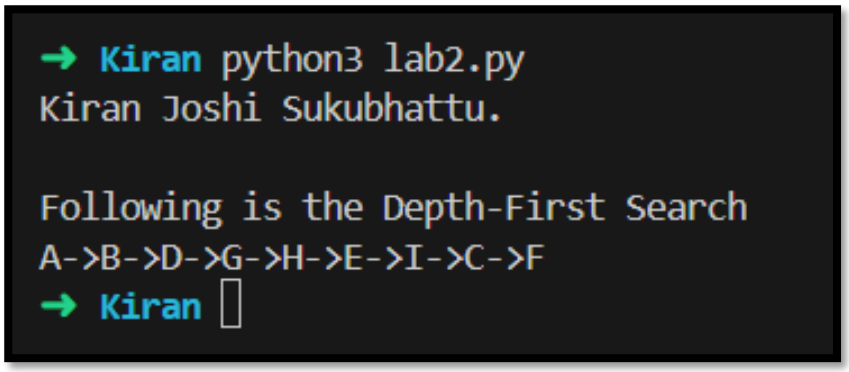
LAB 2
Depth First Search

AIM: To Implement Depth First Search using Python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': ['G', 'H'],
   'E': ['I'],
   'F': [],
   'G': [],
   'H': [],
   'I': []
}

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 '->' after each node except the last one
      print(node, end='->' if node != 'F' else '\n')
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)


# Driver Code
print("\nFollowing is the Depth-First Search")
dfs(visited, graph, 'A')
```

Output:



LAB 3
Tic-Tac-Toe

AIM: To Implement Tic-Tac-Toe game using python.

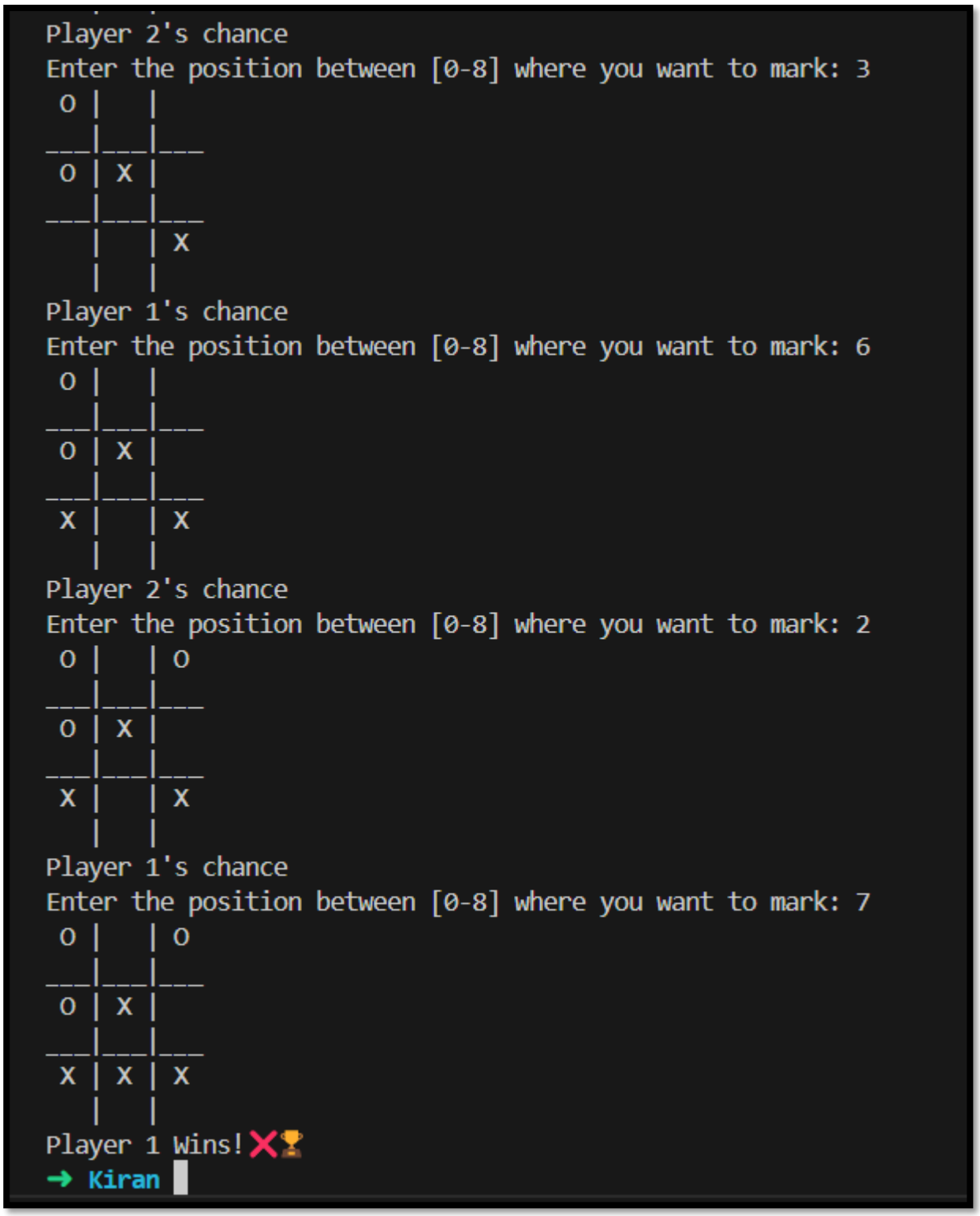
Source Code:

```
print("Kiran Joshi Sukubhattu.")
board = [' ' for x in range(9)]
player = 1

''' Win Flags '''
Win = 1
Draw = -1
Running = 0
Stop = 1
#############################
Game = Running
Mark = 'X'

# This Function Draws Game Board
def DrawBoard():
    print(" %c | %c | %c " % (board[0], board[1], board[2]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[3], board[4], board[5]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[6], board[7], board[8]))
    print("   |   |   ")


# This Function Checks position is empty or not
def CheckPosition(x):
    if (board[x] == ' '):
        return True
    else:
        return False


# This Function Checks player has won or not
def CheckWin():
    global Game
    # Horizontal winning condition
    if (board[0] == board[1] and board[1] == board[2] and board[0] != ' '):
        Game = Win
    elif (board[3] == board[4] and board[4] == board[5] and board[3] != ' '):
        Game = Win
    elif (board[6] == board[7] and board[7] == board[8] and board[6] != ' '):
        Game = Win

    # Vertical Winning Condition
    elif (board[0] == board[3] and board[3] == board[6] and board[0] != ' '):
        Game = Win
```

```
    elif (board[1] == board[4] and board[4] == board[7] and board[1] != ' '):
        Game = Win
    elif (board[2] == board[5] and board[5] == board[8] and board[2] != ' '):
        Game = Win

    # Diagonal Winning Condition
    elif (board[0] == board[4] and board[4] == board[8] and board[4] != ' '):
        Game = Win
    elif (board[2] == board[4] and board[4] == board[6] and board[4] != ' '):
        Game = Win

    # Match Tie or Draw Condition
    elif (board[0] != ' ' and
          board[1] != ' ' and
          board[2] != ' ' and
          board[3] != ' ' and
          board[4] != ' ' and
          board[5] != ' ' and
          board[6] != ' ' and
          board[7] != ' ' and
          board[8] != ' '):
        Game = Draw
    else:
        Game = Running

print("---- Tic-Tac-Toe ----\n\n")
print("Player 1 [X] --- Player 2 [O]\n\n\n")

while (Game == Running):
    DrawBoard()
    if (player % 2 != 0):
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'
    choice = int(
        input("Enter the position between [0-8] where you want to mark: "))
    if (CheckPosition(choice)):
        board[choice] = Mark
        player += 1
        CheckWin()
DrawBoard()
if (Game == Draw):
    print("Game is tied! 🏅 🏆 ")
elif (Game == Win):
    player -= 1
if (player % 2 != 0):
    print("Player 1 Wins! ❌ 🏆 ")
else:
    print("Player 2 Wins! ⭕ 🏆 ")
```

Output:

```
➜ Kiran python3 lab3.py
Kiran Joshi Sukubhattu.
---- Tic-Tac-Toe ----


Player 1 [X] --- Player 2 [O]


   |   |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 1's chance
Enter the position between [0-8] where you want to mark: 4
   |   |
___|___|___
   | X |
___|___|___
   |   |
   |   |
Player 2's chance
Enter the position between [0-8] where you want to mark: 0
 O |   |
___|___|___
   | X |
___|___|___
   |   |
   |   |
Player 1's chance
Enter the position between [0-8] where you want to mark: 8
 O |   |
___|___|___
   | X |
___|___|___
   |   | X
   |   |
```


Player 2's chance
Enter the position between [0-8] where you want to mark: 3
O | | |
X |
Player 1's chance
Enter the position between [0-8] where you want to mark: 6
O | | |
X |
X | | X
Player 2's chance
Enter the position between [0-8] where you want to mark: 2
O | | O
X |
X | | X
Player 1's chance
Enter the position between [0-8] where you want to mark: 7
O | | O
X |
X | X | X
Player 1 Wins!
Kiran

LAB 4

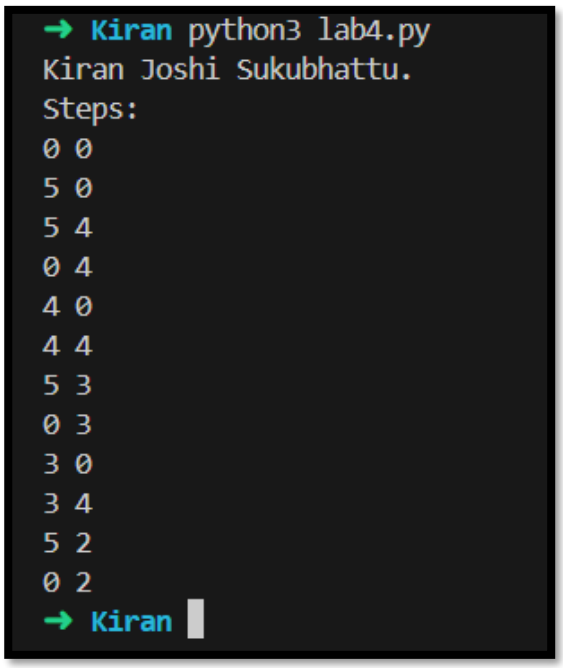
Water-Jug Problem

AIM: To Implement Water-Jug problem using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
from collections import defaultdict
# Max capacities of jugs and the target amount
jug1, jug2, aim = 5, 4, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    if not visited[(amt1, amt2)]:
        visited[(amt1, amt2)] = True
        print(amt1, amt2)

        return (waterJugSolver(0, amt2) or  # Empty jug 1
                waterJugSolver(amt1, 0) or  # Empty jug 2
                waterJugSolver(jug1, amt2) or  # Fill jug 1
                waterJugSolver(amt1, jug2) or  # Fill jug 2
                waterJugSolver(amt1 + min(amt2, jug1 - amt1), amt2 - min(amt2, jug1 - amt1)) or  #
Pour jug 2 into jug 1
                waterJugSolver(amt1 - min(amt1, jug2 - amt2), amt2 + min(amt1, jug2 - amt2)))  # Pour
jug 1 into jug 2
    return False
print("Steps:")
waterJugSolver(0, 0)
```

Output:



LAB 5

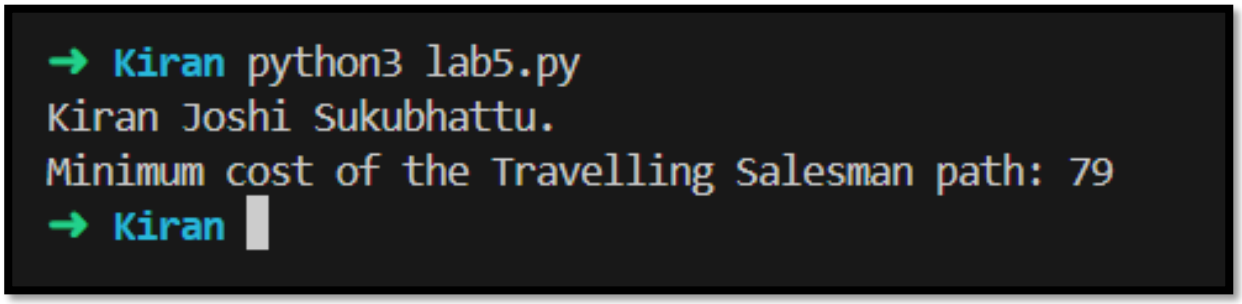
Travelling Salesman Problem

AIM: To Implement Travelling Salesman problem using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
from sys import maxsize
from itertools import permutations

# Number of vertices in the graph (V = 4)
V = 4

# Function to implement the Travelling Salesman Problem
def travellingSalesmanProblem(graph, start):
    # Store all vertices except the start vertex
    vertex = [i for i in range(V) if i != start]

    # Initialize minimum path weight as infinite
    min_path = maxsize

    # Generate all permutations of the vertex list
    next_permutations = permutations(vertex)

    # Calculate the path weight for each permutation
    for perm in next_permutations:
        current_pathweight = 0
        k = start

        # Compute the total path weight for this permutation
        for j in perm:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][start]  # Return to start

        # Update minimum path if the current one is smaller
        min_path = min(min_path, current_pathweight)

    return min_path

# Driver Code
if __name__ == "__main__":
    # Graph represented as an adjacency matrix
    graph = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
    start = 0
    print(f"Minimum cost of the Travelling Salesman path: {travellingSalesmanProblem(graph,
start)}")
```

Output:


➜ Kiran python3 lab5.py
Kiran Joshi Sukubhattu.
Minimum cost of the Travelling Salesman path: 79
➜ Kiran

LAB 6

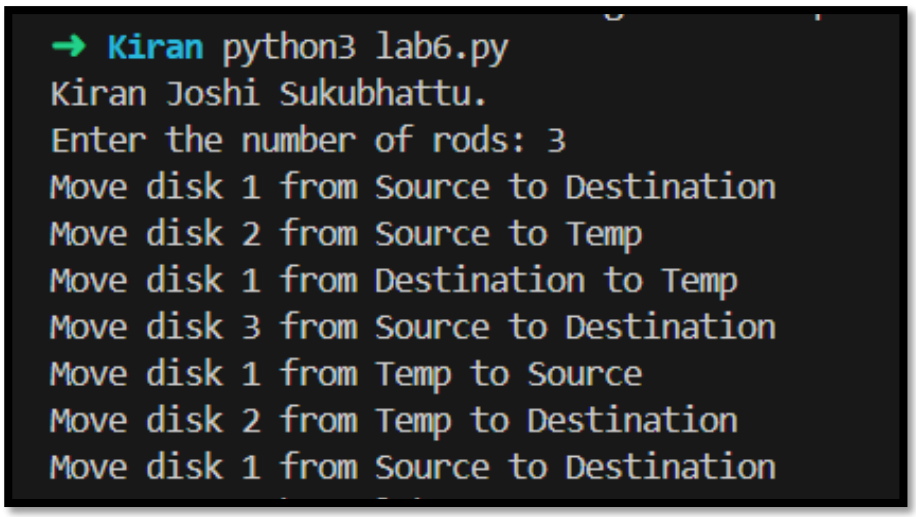
Tower of Hanoi

AIM: To Implement Tower of Hanoi using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
def toh(n, source, destination, temp):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    toh(n - 1, source, temp, destination)
    print(f"Move disk {n} from {source} to {destination}")
    toh(n-1, temp, destination, source)


# Enter the number of rods
n = int(input("Enter the number of rods: "))
toh(n, 'Source', 'Destination', 'Temp')
```

Output:



LAB 7

Monkey Banana Problem

AIM: To Implement Monkey Banana Problem using python.

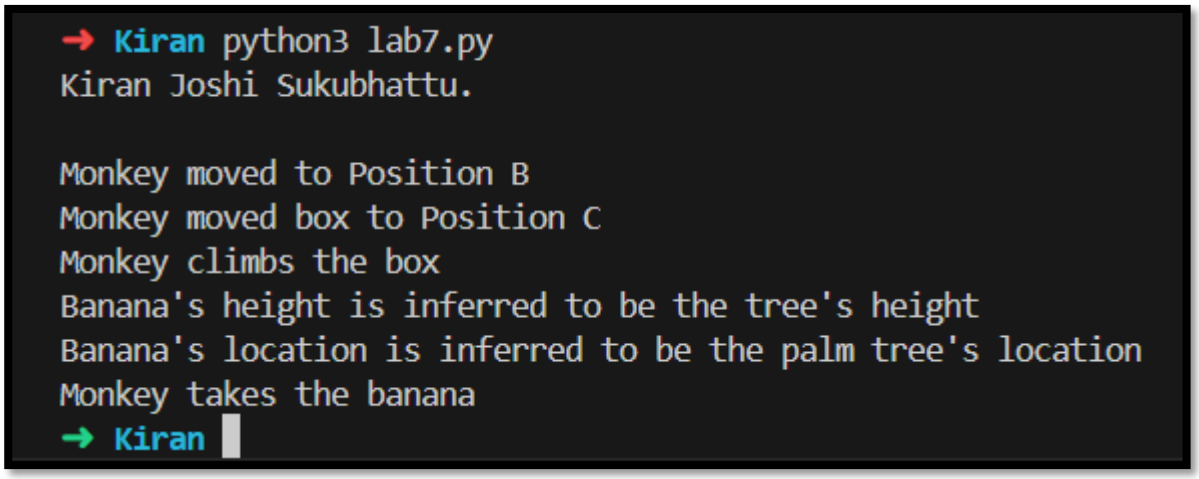
Source Code:

```
print("Kiran Joshi Sukubhattu.")
from typing import Set, Callable, List, Any

class Position:
    def __init__(self, locname=None):
        self.locname = locname

    def __str__(self):
        return self.locname if self.locname else "unknown"

class HasHeight:
    def __init__(self, height=0):
        self.height = height

class HasPosition:
    def __init__(self, at=None):
        self.at = at

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):
    def __init__(self, owner: Monkey = None, attached: PalmTree = None):
        super().__init__()
        self.owner = owner
        self.attached = attached

class World:
    def __init__(self):
        self.locations: Set[Position] = set()

# Create the world and positions
p1 = Position("Position A")
p2 = Position("Position B")
p3 = Position("Position C")

w = World()
w.locations = set([p1, p2, p3])
```

```
# Create the monkey, box, palm tree, and banana
m = Monkey()
m.height = 0  # ground level
m.at = p1

box = Box()
box.height = 2
box.at = p2

p = PalmTree()
p.at = p3

b = Banana()
b.attached = p

# Define the functions for actions and reasoning
def go(monkey: Monkey, where: Position):
    assert where in w.locations, "Invalid location"
    assert monkey.height == 0, "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, "Monkey and box must be at the same location"
    assert where in w.locations, "Invalid location"
    assert monkey.height == 0, "Monkey can only push 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 must be at the same location as the box"
    monkey.height += box.height
    return "Monkey climbs the box"

def grasp(monkey: Monkey, banana: Banana):
    assert monkey.height == banana.height, "Monkey must be at the same height as the banana"
    assert monkey.at == banana.at, "Monkey must be at the same location as the banana"
    banana.owner = monkey
    return "Monkey takes the banana"

def infer_owner_at(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree, "Banana must be attached to the palm tree"
    banana.at = palmtree.at
    return "Banana's location is inferred to be the palm tree's location"

def infer_banana_height(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree, "Banana must be attached to the palm tree"
    banana.height = palmtree.height
    return "Banana's height is inferred to be the tree's height"
```

```
# Define the schedule function
def schedule(actions: List[Callable], args: List[Any], goal: Callable):
    results = []
    for action in actions:
        try:
            result = action()  # Call the action without args, we'll provide them later in lambdas
            results.append(result)
            if goal():
                break
        except AssertionError as e:
            results.append(f"Action {action.__name__} failed: {str(e)}")
    return results

# Define the schedule of actions and reasoning
print('\n'.join(schedule(
    [lambda: go(m, p2), lambda: push(m, box, p3), lambda: climb_up(m, box), lambda:
infer_banana_height(p, b), lambda: infer_owner_at(p, b), lambda: grasp(m, b)],
    [w, p1, p2, p3, m, box, p, b],
    goal=lambda: b.owner == m
)))
```

Output:



LAB 8

N-Queens Problem

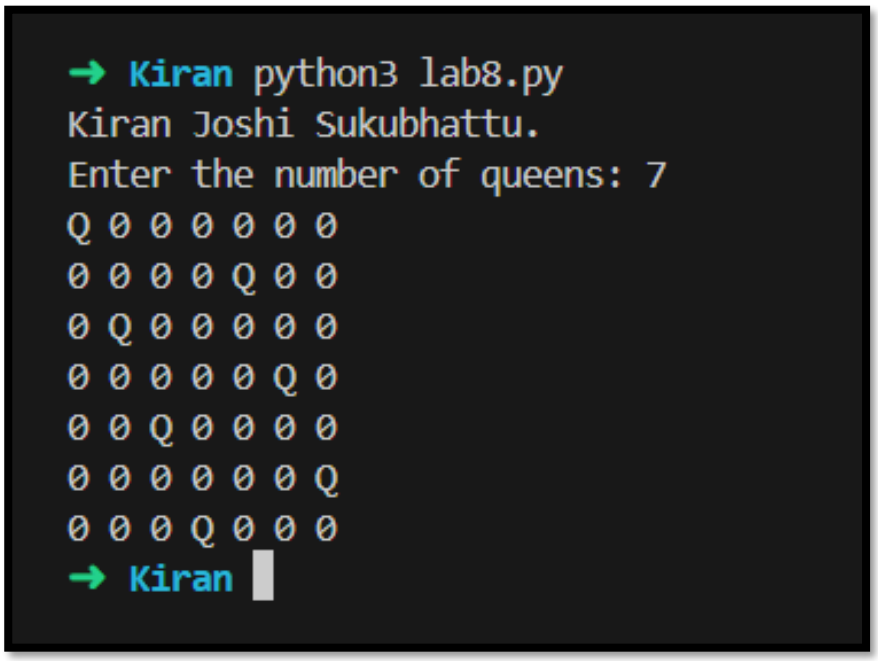
AIM: To Implement N-Queens Problem using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
'''
Program to implement for N-Queen problem
'''

'''
logic where the queen must not be placed:
1) row, col+-
2) row--, col--
3) row++, col--
'''

def is_safe(board, row, col, n):
    for c in range(col, -1, -1):  # check for the same row in left side of the board
        if board[row][c] == 'Q':
            return False

    i = row
    j = col

    while i >= 0 and j >= 0:  # check for the left diagonal in the upper side of the board
        if board[i][j] == 'Q':
            return False

        i -= 1
        j -= 1

    i = row
    j = col

    while i < n and j >= 0:  # check for the left diagonal in the bottom side of the board
        if board[i][j] == 'Q':
            return False

        i += 1
        j -= 1

    return True


def nqueen(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 'Q'
```

```
            if nqueen(board, col+1, n):
                return True
            board[i][col] = 0
    return False


n = int(input("Enter the number of queens: "))
board = [[0 for j in range(n)] for i in range(n)]

if nqueen(board, 0, n) == True:
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=' ')
        print()
else:
    print("Not possible")
```

Output:



LAB 9

Naive Bayes Algorithm

AIM: To Implement Naive Bayes Algorithm using python.

Source Code:

```
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Gaussian Naive Bayes classifier
model = GaussianNB()

# Train the model
model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Output the results
print("Kiran Joshi Sukubhattu.")
print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nConfusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(class_report)
```

Output:

```
➜ Kiran python3 lab9.py
Kiran Joshi Sukubhattu.

Accuracy: 100.00%

Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

➜ Kiran
```

LAB 10

Back Propagation Algorithm

AIM: To Implement Back Propagation Algorithm using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
# Input data (4 samples, 2 features each)
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

# Output labels (XOR truth table)
y = np.array([[0],
              [1],
              [1],
              [0]])

# Seed for reproducibility
np.random.seed(42)
# Initialize weights randomly with mean 0
input_layer_neurons = X.shape[1]  # Number of input neurons
hidden_layer_neurons = 2          # Number of hidden neurons
output_neurons = 1                # Number of output neurons

# Weights for the input to the hidden layer
W1 = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))

# Weights for the hidden to the output layer
W2 = np.random.uniform(size=(hidden_layer_neurons, output_neurons))

# Learning rate
learning_rate = 0.1

# Number of iterations for training
epochs = 10000
# Training process
for epoch in range(epochs):
    # Forward Propagation
    hidden_layer_input = np.dot(X, W1)  # Input to hidden layer
    hidden_layer_output = sigmoid(hidden_layer_input)  # Output from hidden layer

    output_layer_input = np.dot(hidden_layer_output, W2)  # Input to output layer
```

```
    predicted_output = sigmoid(output_layer_input)  # Output from output layer

    # Calculate the error (difference between actual and predicted)
    error = y - predicted_output

    # Backpropagation
    d_predicted_output = error * sigmoid_derivative(predicted_output)  # Derivative of output
    error_hidden_layer = d_predicted_output.dot(W2.T)  # Error propagated to hidden layer
    d_hidden_layer_output = error_hidden_layer * sigmoid_derivative(hidden_layer_output)  #
Derivative of hidden layer output

    # Update the weights
    W2 += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    W1 += X.T.dot(d_hidden_layer_output) * learning_rate

    # Print the error every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Error: {np.mean(np.abs(error))}")

# Final output after training
print("\nFinal output after training:")
print(predicted_output)
```

Output:

```
➜ Kiran python3 lab10.py
Kiran Joshi Sukubhattu.
Epoch 0, Error: 0.4994268057883715
Epoch 0, Error: 0.4994268057883715
Epoch 1000, Error: 0.4995279992650292
Epoch 2000, Error: 0.496475302936487
Epoch 3000, Error: 0.4763078141033909
Epoch 4000, Error: 0.43576723452875155
Epoch 5000, Error: 0.3951513944731264
Epoch 6000, Error: 0.3604774441101641
Epoch 7000, Error: 0.33148296848322656
Epoch 8000, Error: 0.30713273219637427
Epoch 9000, Error: 0.28649148390098955

Final output after training:
[[0.20369158]
 [0.73603066]
 [0.73604444]
 [0.34370702]]
```

LAB 11
Genetics Algorithm

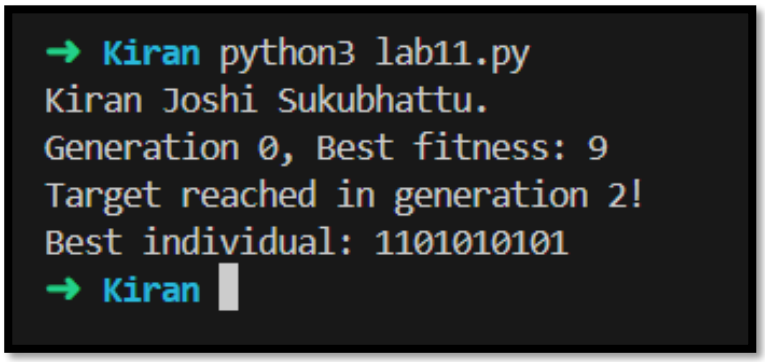
AIM: To Implement Genetics Algorithm using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")

import random
# Define the target string and population parameters
TARGET = "1101010101"  # Target binary string
POPULATION_SIZE = 100  # Size of the population
MUTATION_RATE = 0.01   # Mutation rate
GENERATIONS = 1000     # Maximum number of generations

# Individual class representing a binary string
class Individual:
    def __init__(self, chromosome=None):
        if chromosome:
            self.chromosome = chromosome
        else:
            # Randomly initialize a chromosome (binary string)
            self.chromosome = ''.join(random.choice('01') for _ in range(len(TARGET)))
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        # Fitness is the number of matching bits with the target string
        return sum(1 for i, j in zip(self.chromosome, TARGET) if i == j)

# Genetic Algorithm functions
def selection(population):
    # Select two individuals from the population based on fitness (roulette wheel selection)
    return random.choices(population, weights=[ind.fitness for ind in population], k=2)

def crossover(parent1, parent2):
    # Single-point crossover: combine the chromosomes of the two parents
    crossover_point = random.randint(0, len(TARGET) - 1)
    child_chromosome = parent1.chromosome[:crossover_point] +
parent2.chromosome[crossover_point:]
    return Individual(chromosome=child_chromosome)

def mutate(individual):
    # Randomly mutate the individual's chromosome based on the mutation rate
    mutated_chromosome = ''.join(
        gene if random.random() > MUTATION_RATE else random.choice('01')
        for gene in individual.chromosome
    )
    return Individual(chromosome=mutated_chromosome)

# Main Genetic Algorithm function
def genetic_algorithm():
    # Initialize the population with random individuals
```

```
    population = [Individual() for _ in range(POPULATION_SIZE)]

    for generation in range(GENERATIONS):
        # Sort population by fitness
        population = sorted(population, key=lambda x: x.fitness, reverse=True)

        # If the fittest individual matches the target, stop
        if population[0].fitness == len(TARGET):
            print(f"Target reached in generation {generation}!")
            print(f"Best individual: {population[0].chromosome}")
            break

        # Print best individual every 100 generations
        if generation % 100 == 0:
            print(f"Generation {generation}, Best fitness: {population[0].fitness}")

        # Create a new population using selection, crossover, and mutation
        new_population = []

        # Elitism: carry forward the best individual
        new_population.append(population[0])

        # Generate the rest of the population
        while len(new_population) < POPULATION_SIZE:
            # Select parents based on fitness
            parent1, parent2 = selection(population)

            # Perform crossover
            child = crossover(parent1, parent2)

            # Mutate the child
            child = mutate(child)

            # Add child to the new population
            new_population.append(child)

        # Replace the old population with the new one
        population = new_population

genetic_algorithm()
```

Output:



LAB 12

A* Search Algorithm

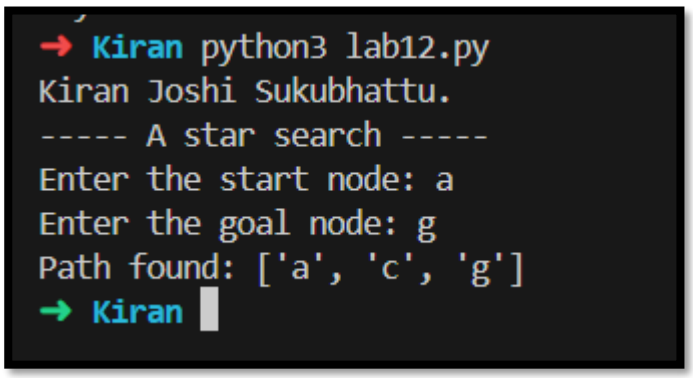
AIM: To Implement A* Search Algorithm using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
'''
Program to implement A Star Search Algorithm
'''

# Defining the graph nodes in dict with given costs to traverse
adj_list = {
   's': [('a', 1), ('g', 10)],
   'a': [('b', 2), ('c', 1)],
   'b': [('d', 5)],
   'c': [('d', 3), ('g', 4)],
   'd': [('g', 2)],
   'g': []
}

# Defining heuristic values for each nodes
heuristic = {
   's': 5,
   'a': 3,
   'b': 4,
   'c': 2,
   'd': 6,
   'g': 0
}

# A Star Search Algorithm


def astar_search(adj_list, heuristic, start_node, goal_node):
   open_list = set([start_node])
   closed_list = set([])
   g = {}
   g[start_node] = 0
   parents = {}
   parents[start_node] = start_node

   def get_neighbors(node):
      return adj_list[node]

   def h(node):
      return heuristic[node]

   while len(open_list) > 0:
      n = None
      for v in open_list:
         if n == None or g[v] + h(v) < g[n] + h(n):
```

```
            n = v

        if n == None:
            print('Path does not exist!')
            return None
        if n == goal_node:
            reconst_path = []
            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]
            reconst_path.append(start_node)
            reconst_path.reverse()

            print('Path found: {}'.format(reconst_path))
            return reconst_path

        for (m, weight) in get_neighbors(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)
        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None

print("----- A star search -----")
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")

astar_search(adj_list, heuristic, start_node, goal_node)
```

Output:



LAB 13

Greedy Search Algorithm

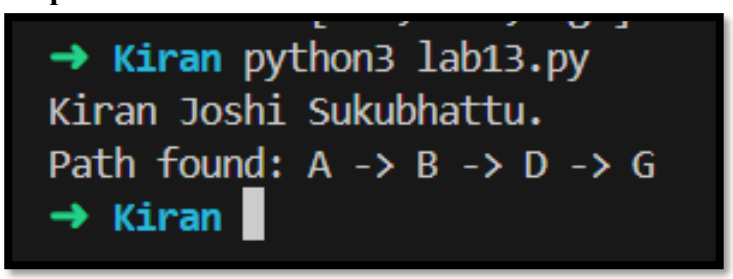
AIM: To Implement Greedy Search Algorithm using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
import heapq

# Class to represent the graph
class Graph:
    def __init__(self):
        self.edges = {}
        self.heuristics = {}

    # Add an edge between two nodes
    def add_edge(self, node1, node2, cost):
        if node1 not in self.edges:
            self.edges[node1] = []
        self.edges[node1].append((node2, cost))

    # Set heuristic value for a node
    def set_heuristic(self, node, h_value):
        self.heuristics[node] = h_value

    # Greedy Best-First Search algorithm
    def greedy_search(self, start, goal):
        # Priority queue to store (heuristic, node) pairs
        open_list = []
        heapq.heappush(open_list, (self.heuristics[start], start))

        # To keep track of visited nodes
        visited = set()

        # To store the path
        came_from = {}

        while open_list:
            # Get the node with the lowest heuristic value
            _, current_node = heapq.heappop(open_list)

            # If we reach the goal, reconstruct and return the path
            if current_node == goal:
                return self.reconstruct_path(came_from, start, goal)

            visited.add(current_node)

            # Explore neighbors
            for neighbor, cost in self.edges.get(current_node, []):
                if neighbor not in visited:
                    heapq.heappush(open_list, (self.heuristics[neighbor], neighbor))
                    came_from[neighbor] = current_node
```

```
        return None  # No path found

    # Function to reconstruct the path from the start to the goal
    def reconstruct_path(self, came_from, start, goal):
        path = [goal]
        while goal in came_from:
            goal = came_from[goal]
            path.append(goal)
        path.reverse()
        return path

# Example usage:
graph = Graph()

# Add edges: graph.add_edge('A', 'B', cost)
graph.add_edge('A', 'B', 1)
graph.add_edge('A', 'C', 3)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 1)
graph.add_edge('C', 'F', 5)
graph.add_edge('E', 'F', 2)
graph.add_edge('D', 'G', 6)
graph.add_edge('F', 'G', 2)

# Set heuristic values for each node (estimated cost to the goal)
graph.set_heuristic('A', 6)
graph.set_heuristic('B', 4)
graph.set_heuristic('C', 5)
graph.set_heuristic('D', 2)
graph.set_heuristic('E', 2)
graph.set_heuristic('F', 1)
graph.set_heuristic('G', 0)  # Goal node heuristic is 0

# Run Greedy Best-First Search from 'A' to 'G'
path = graph.greedy_search('A', 'G')
if path:
    print(f"Path found: {' -> '.join(path)}")
else:
    print("No path found")
```

Output:



LAB 14

Uniform Cost Search Algorithm

AIM: To Implement Uniform Cost Search Algorithm using python.

Source Code:

```
print("Kiran Joshi Sukubhattu.")
import heapq

# Class to represent the graph
class Graph:
    def __init__(self):
        self.edges = {}

    # Add an edge between two nodes with a given cost
    def add_edge(self, node1, node2, cost):
        if node1 not in self.edges:
            self.edges[node1] = []
        self.edges[node1].append((cost, node2))

    # Uniform Cost Search algorithm
    def uniform_cost_search(self, start, goal):
        # Priority queue to store (cumulative_cost, node, path)
        open_list = []
        heapq.heappush(open_list, (0, start, [start]))  # (cost, node, path)

        # To store the visited nodes and their costs
        visited = {}

        while open_list:
            # Get the node with the least cumulative cost
            current_cost, current_node, path = heapq.heappop(open_list)

            # If we have reached the goal, return the path and the cost
            if current_node == goal:
                return path, current_cost

            # If the node is not visited or found at a lower cost
            if current_node not in visited or current_cost < visited[current_node]:
                visited[current_node] = current_cost

                # Explore the neighbors
                for cost, neighbor in self.edges.get(current_node, []):
                    new_cost = current_cost + cost
                    new_path = path + [neighbor]
                    heapq.heappush(open_list, (new_cost, neighbor, new_path))

        return None, float('inf')  # No path found

# Example usage
graph = Graph()
```

```
# Add edges: graph.add_edge('A', 'B', cost)
graph.add_edge('A', 'B', 1)
graph.add_edge('A', 'C', 4)
graph.add_edge('B', 'D', 2)
graph.add_edge('C', 'D', 1)
graph.add_edge('D', 'E', 5)
graph.add_edge('B', 'E', 12)
graph.add_edge('C', 'E', 5)

# Run Uniform Cost Search from 'A' to 'E'
path, cost = graph.uniform_cost_search('A', 'E')
if path:
    print(f"Path found: {' -> '.join(path)}, with total cost: {cost}")
else:
    print("No path found")
```

Output: