# Practical no 1 Code

## BFS:-checked working

*# Python3 Program to print BFS traversal # from a given source vertex. BFS(int s) # traverses vertices reachable from s.* **from** collections **import** defaultdict

*# This class represents a directed graph # using adjacency list representation* **class** Graph:

*# Constructor*

**def** init (self):

*# default dictionary to store graph*

self.graph = defaultdict(list)

*# function to add an edge to graph*

**def** addEdge(self,u,v):

*"""*

**:rtype***: object """*

self.graph[u].append(v)

*# Function to print a BFS of graph*

**def** BFS(self, s):

*# Mark all the vertices as not visited*

visited = [**False**] \* (max(self.graph) + 1)

*# Create a queue for BFS*

queue = []

*# Mark the source node as # visited and enqueue it* queue.append(s) visited[s] = **True**

**while** queue:

*# Dequeue a vertex from*

*# queue and print it* s = queue.pop(0) print (s, end = **" "**)

*# Get all adjacent vertices of the # dequeued vertex s. If a adjacent # has not been visited, then mark it # visited and enqueue it*

**for** i **in** self.graph[s]:

**if** visited[i] == **False**: queue.append(i) visited[i] = **True**

*# Driver code*

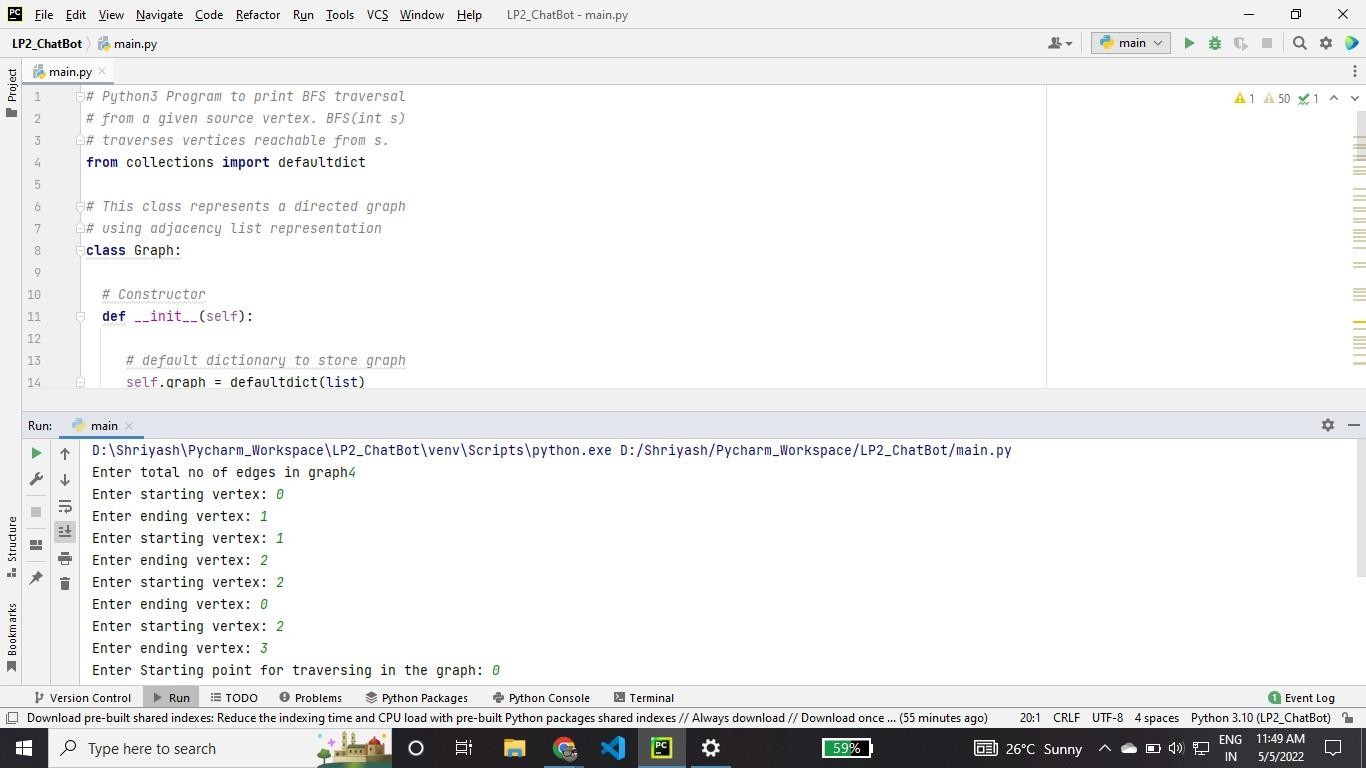
*# Create a graph given in # the above diagram*

g = Graph();

n=int(input(**"Enter total no of edges in graph"**)) **for** i **in** range(0,n):

u=int(input(**"Enter starting vertex: "**)) v=int(input(**"Enter ending vertex: "**)) g.addEdge(u,v)

s=int(input(**"Enter Starting point for traversing in the graph: "**)) g.BFS(s)



## DFS:-

# Python3 program to print DFS traversal # from a given given graph

from collections import defaultdict

# This class represents a directed graph using # adjacency list representation

class Graph:

# Constructor

def init (self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self, u, v):

self.graph[u].append(v)

# A function used by DFS

def DFSUtil(self, v, visited):

# Mark the current node as visited

# and print it

visited.add(v)

print(v, end=' ')

# Recur for all the vertices

# adjacent to this vertex

for neighbour in self.graph[v]:

if neighbour not in visited:

self.DFSUtil(neighbour, visited)

# The function to do DFS traversal. It uses

# recursive DFSUtil()

def DFS(self, v):

# Create a set to store visited vertices

visited = set()

# Call the recursive helper function

# to print DFS traversal

self.DFSUtil(v, visited)

# Driver code

g = Graph()

n = int(input("Enter total no of edges")) for i in range(0, n):

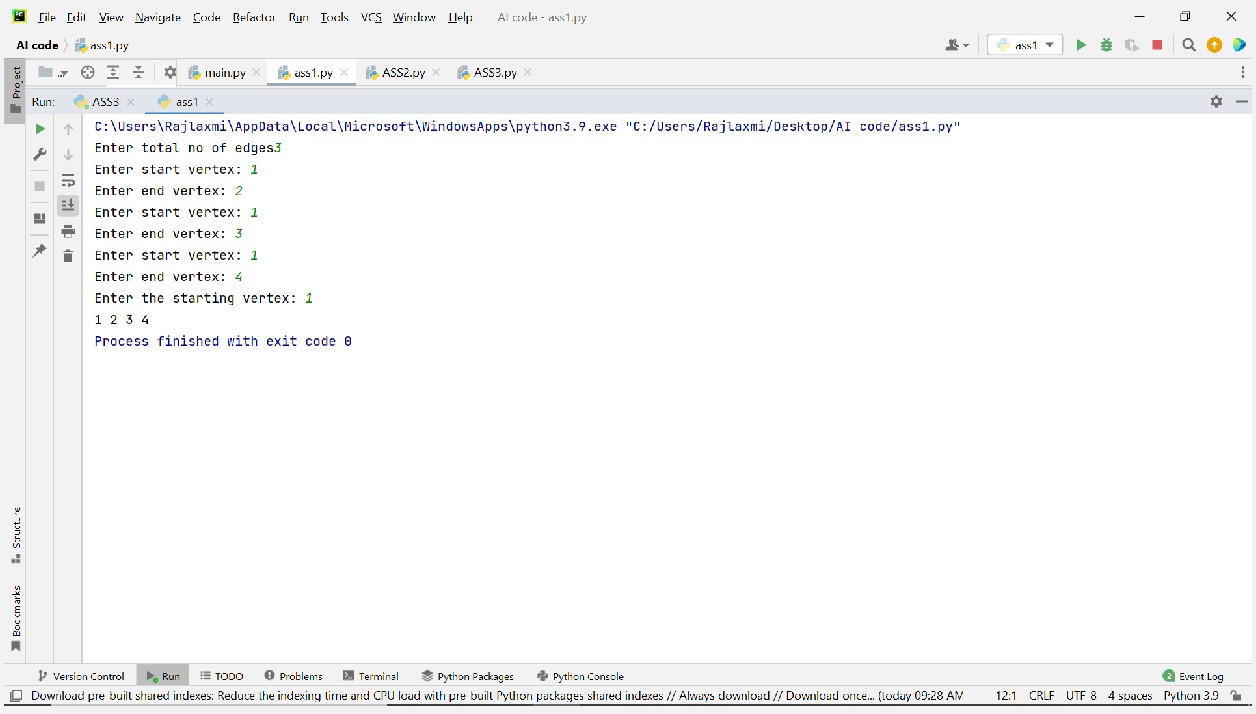
u = int(input("Enter start vertex: "))

v = int(input("Enter end vertex: "))

g.addEdge(u, v)

g.addEdge(v, u)

g.DFS(int(input("Enter the starting vertex: ")))



# Practical no 2 Code

[**A-Star Algorithm Python Tutorial - An Introduction To A\* Algorithm In Python**](https://www.simplifiedpython.net/a-star-algorithm-python-tutorial/)[**(simplifiedpython.net)**](https://www.simplifiedpython.net/a-star-algorithm-python-tutorial/)

A\* algorithm:-

from queue import PriorityQueue

*#Creating Base Class*

class State(object):

def init (self, value, parent, start = 0, goal = 0): self.children = []

self.parent = parent self.value = value self.dist = 0

if parent:

self.start = parent.start self.goal = parent.goal self.path = parent.path[:] self.path.append(value)

else:

self.path = [value] self.start = start self.goal = goal

def GetDistance(self): pass

def CreateChildren(self): pass

*# Creating subclass*

class State\_String(State):

def init (self, value, parent, start = 0, goal = 0 ): super(State\_String, self). init (value, parent, start, goal) self.dist = self.GetDistance()

def GetDistance(self):

if self.value == self.goal: return 0

dist = 0

for i in range(len(self.goal)): letter = self.goal[i]

dist += abs(i - self.value.index(letter)) return dist

def CreateChildren(self):

if not self.children:

for i in range(len(self.goal)-1):

val = self.value

val = val[:i] + val[i+1] + val[i] + val[i+2:] child = State\_String(val, self) self.children.append(child)

*# Creating a class that hold the final magic*

class A\_Star\_Solver:

def init (self, start, goal): self.path = [] self.vistedQueue =[]

self.priorityQueue = PriorityQueue() self.start = start

self.goal = goal

def Solve(self):

startState = State\_String(self.start,0,self.start,self.goal)

count = 0

self.priorityQueue.put((0,count, startState)) while(not self.path and self.priorityQueue.qsize()):

closesetChild = self.priorityQueue.get()[2] closesetChild.CreateChildren() self.vistedQueue.append(closesetChild.value) for child in closesetChild.children:

if child.value not in self.vistedQueue: count += 1

if not child.dist: self.path = child.path break

self.priorityQueue.put((child.dist,count,child)) if not self.path:

print("Goal Of is not possible !" + self.goal ) return self.path

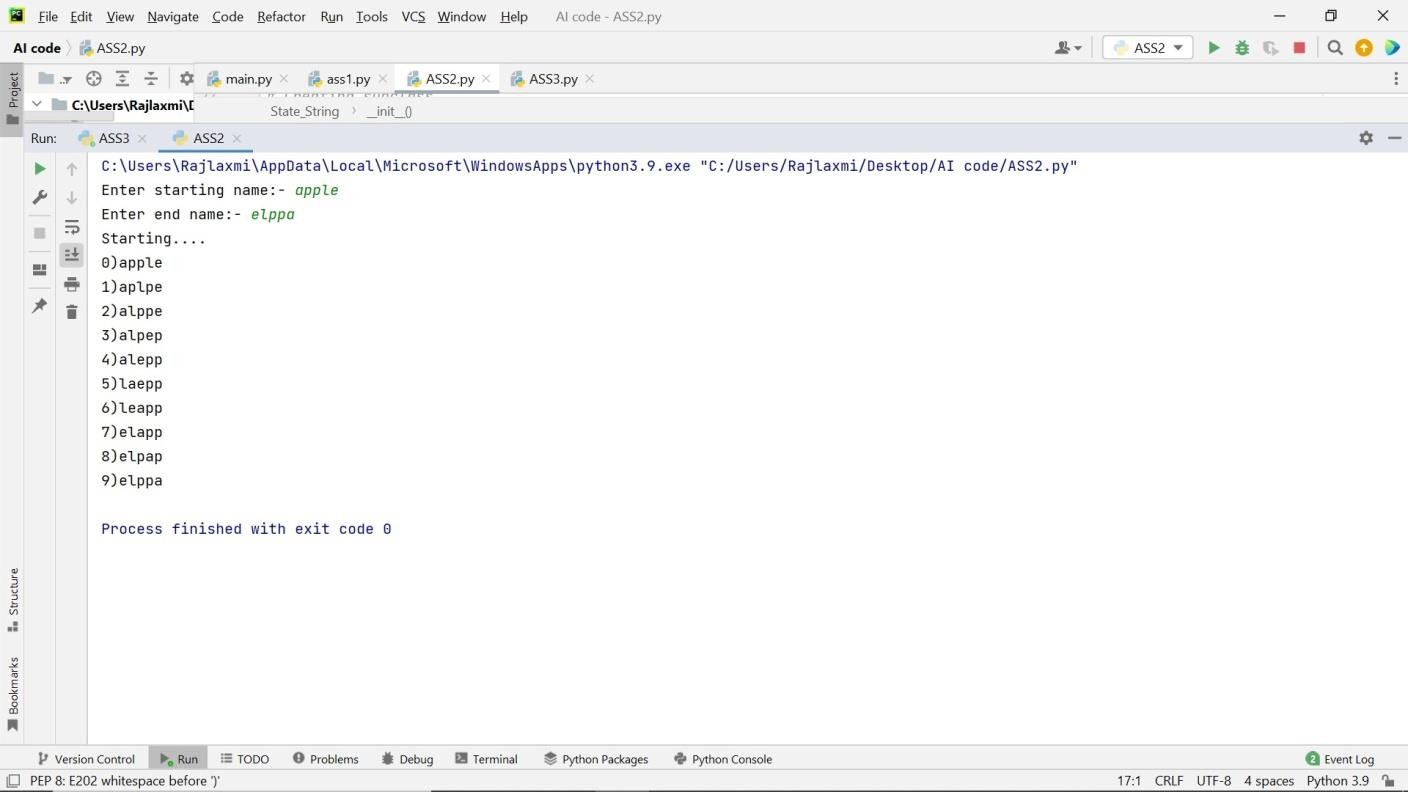
*# Calling all the existing stuffs*

if name == " main ":

start1 = str(input("Enter starting name:- ")) goal1 = str(input("Enter end name:- ")) print("Starting ")

a = A\_Star\_Solver(start1,goal1) a.Solve()

for i in range(len(a.path)): print("{0}){1}".format(i,a.path[i]))



# Practical No 3 Code

## Selection sort:-

***# Python program for implementation of Selection # Sort***

import sys

print("Enter no. of elements") a=int(input()) print("Enter",a,"integer elements") array=[]

for i in range(a): array.append(int(input()))

A=array

***# Traverse through all array elements***

for i in range(len(A)):

***# Find the minimum element in remaining # unsorted array***

min\_idx = i

for j in range(i + 1, len(A)): if A[min\_idx] > A[j]:

min\_idx = j

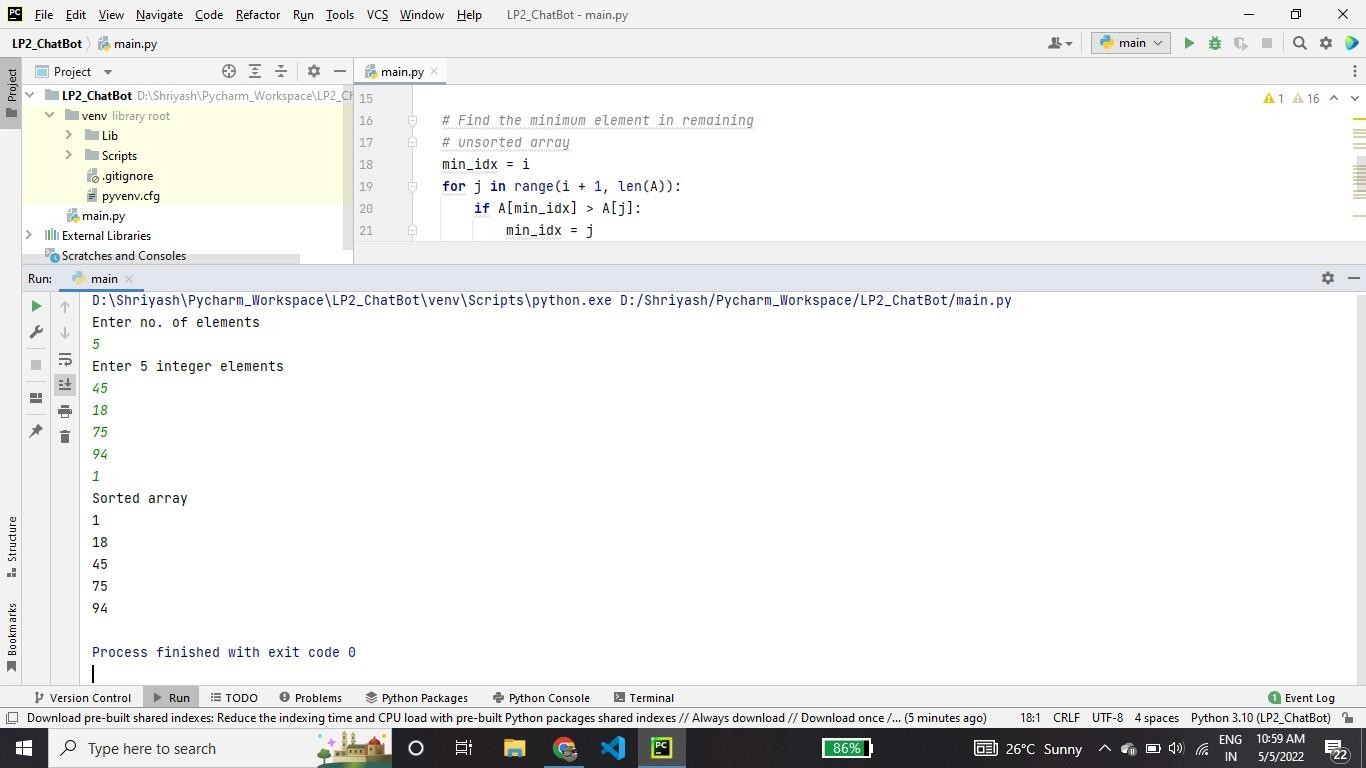
***# Swap the found minimum element with # the first element***

A[i], A[min\_idx] = A[min\_idx], A[i]

***# Driver code to test above***

print("Sorted array") for i in range(len(A)):

print("%d" % A[i]),



## Minimum Spanning Tree

***# A Python program for Prim's Minimum Spanning Tree (MST) algorithm. # The program is for adjacency matrix representation of the graph***

import sys # Library for INT\_MAX

class Graph():

**self.graph = [[0 for column in range(vertices)]**

**for row in range(vertices)]**

**def init (self, vertices): self.V = vertices**

# A utility function to print the constructed MST stored in parent[]

def printMST(self, parent):

print("Edge \tWeight")

for i in range(1, self.V):

print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minKey(self, key, mstSet):

# Initialize min value

min = sys.maxsize

|  |  |  |
| --- | --- | --- |
| **for v in range(self.V):** | |  |
| **if key[v] < min** | **and mstSet[v] == False:**  **v** | |
| **min = key[v]** |
| **min\_index =** |

return min\_index

# Function to construct and print MST for a graph

# represented using adjacency matrix representation

def primMST(self):

# Key values used to pick minimum weight edge in cut

key = [sys.maxsize] \* self.V

parent = [None] \* self.V # Array to store constructed MST

# Make key 0 so that this vertex is picked as first vertex

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1 # First node is always the root of

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minKey(key, mstSet)

# Put the minimum distance vertex in

# the shortest path tree

mstSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

# graph[u][v] is non zero only for adjacent vertices of m

# mstSet[v] is false for vertices not yet included in MST

# Update the key only if graph[u][v] is smaller than key[v]

if self.graph[u][v] > 0 and mstSet[v] == False and key[v]

> self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

self.printMST(parent)

V=int(input("Enter no of vertices")) g = Graph(V)

n=int(input("Enter total no of edges")) for i in range(0, n):

u = int(input("Enter start vertex: "))

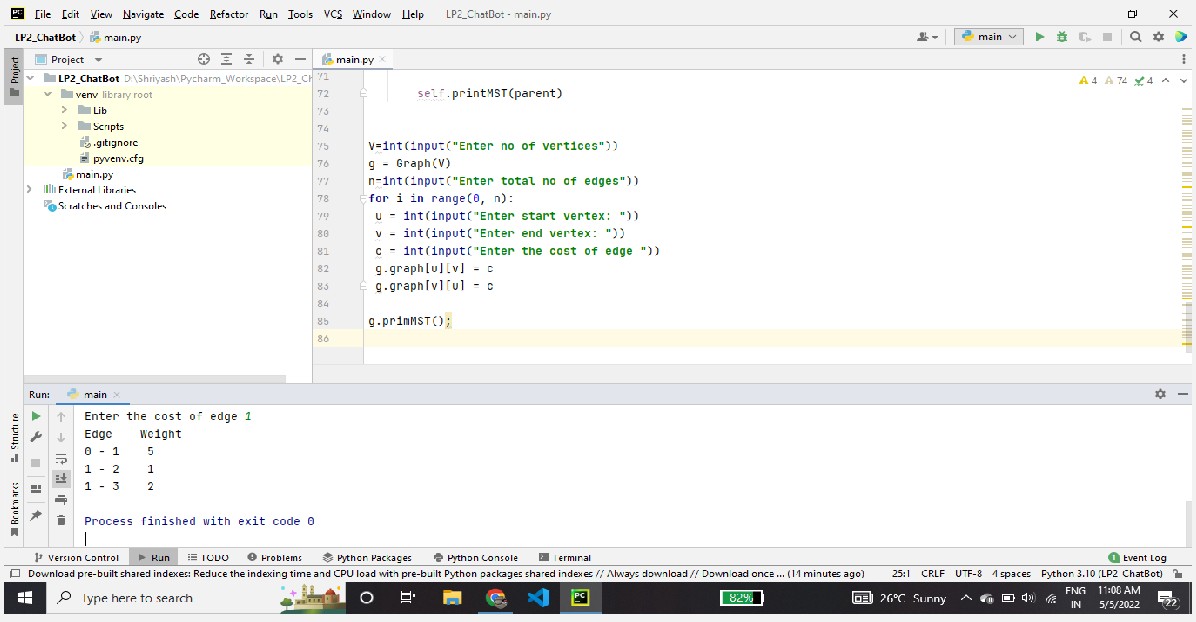
v = int(input("Enter end vertex: "))

c = int(input("Enter the cost of edge "))

g.graph[u][v] = c

g.graph[v][u] = c

g.primMST();



## Single-Source shortest path problem

# Python program for Dijkstra's single

# source shortest path algorithm. The program is # for adjacency matrix representation of the graph class Graph():

def init (self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printSolution(self, dist):

print("Vertex \t Distance from Source: ")

for node in range(self.V):

print(node, "\t\t", dist[node])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = 1e7

# Search not nearest vertex not in the

# shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def dijkstra(self, src):

dist = [1e7] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the

# shortest path tree

sptSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

if (self.graph[u][v] > 0 and

sptSet[v] == False and

dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program V=int(input("Enter no of vertices")) g = Graph(V)

n=int(input("Enter total no of edges")) for i in range(0, n):

u = int(input("Enter start vertex: "))

v = int(input("Enter end vertex: "))

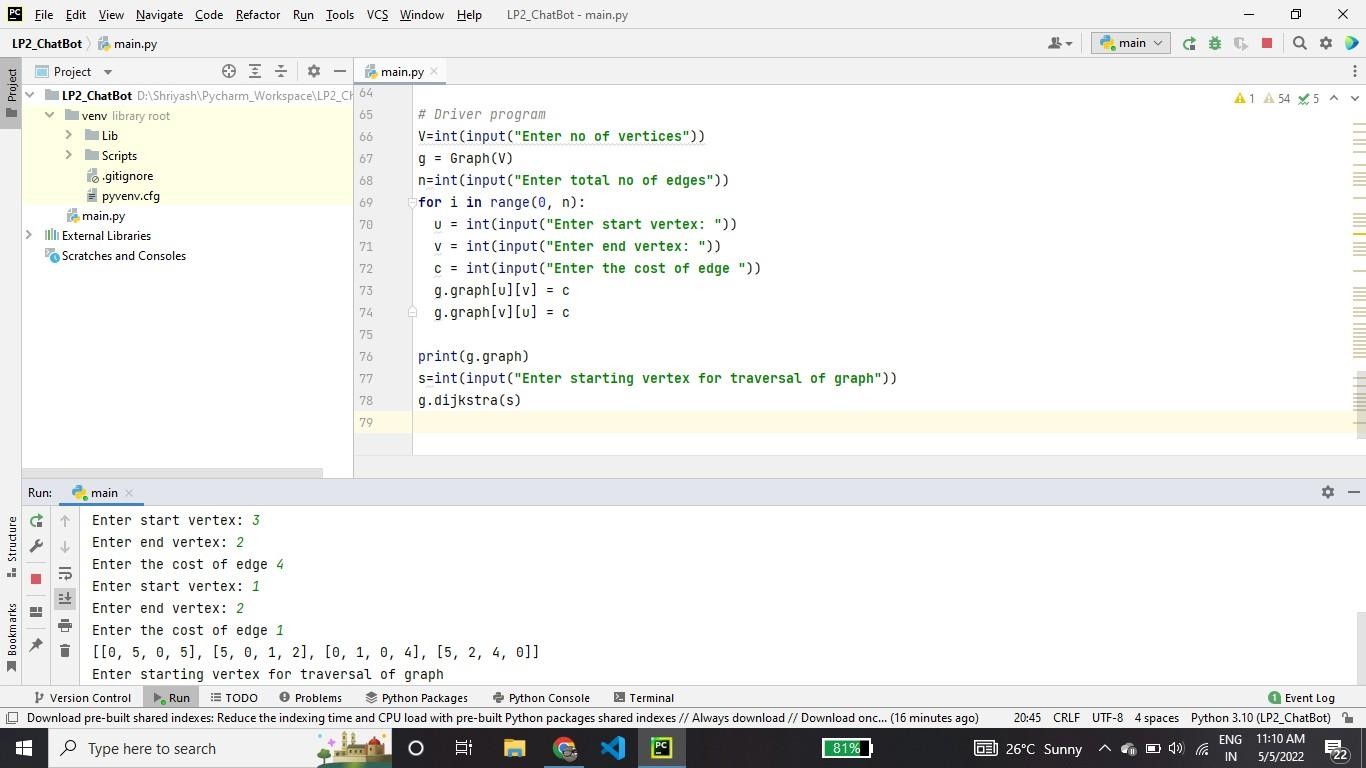
c = int(input("Enter the cost of edge "))

g.graph[u][v] = c

g.graph[v][u] = c

print(g.graph)

s=int(input("Enter starting vertex for traversal of graph")) g.dijkstra(s)



## Job Scheduling Problems

***# Program to find the maximum profit # job sequence from a given array # of jobs with deadlines and profits***

***# function to schedule the jobs take 2***

***# arguments array and no of jobs to schedule***

***def printJobScheduling(arr, t):***

***# length of array***

***n = len(arr)***

***# Sort all jobs according to***

***# decreasing order of profit***

***for i in range(n):***

***for j in range(n - 1 - i):***

***if arr[j][2] < arr[j + 1][2]:***

***arr[j], arr[j + 1] = arr[j + 1], arr[j]***

***# To keep track of free time slots***

***result = [False] \* t***

***# To store result (Sequence of jobs)***

***job = ['-1'] \* t***

***# Iterate through all given jobs***

***for i in range(len(arr)):***

***# Find a free slot for this job***

***# (Note that we start from the***

***# last possible slot)***

***for j in range(min(t - 1, arr[i][1] - 1), -1, -1):***

|  |
| --- |
| ***# Free slot found*** |
| ***if result[j] is False:*** |
| ***result[j] = True*** |
| ***job[j] = arr[i][0]*** |
| ***break*** |

***# print the sequence***

***print(job)***

***# Driver Code***

***t = int(input("Enter the total no of jobs"))***

***arr=[]***

***for i in range(t):***

***col = []***

***col.append(i+1)***

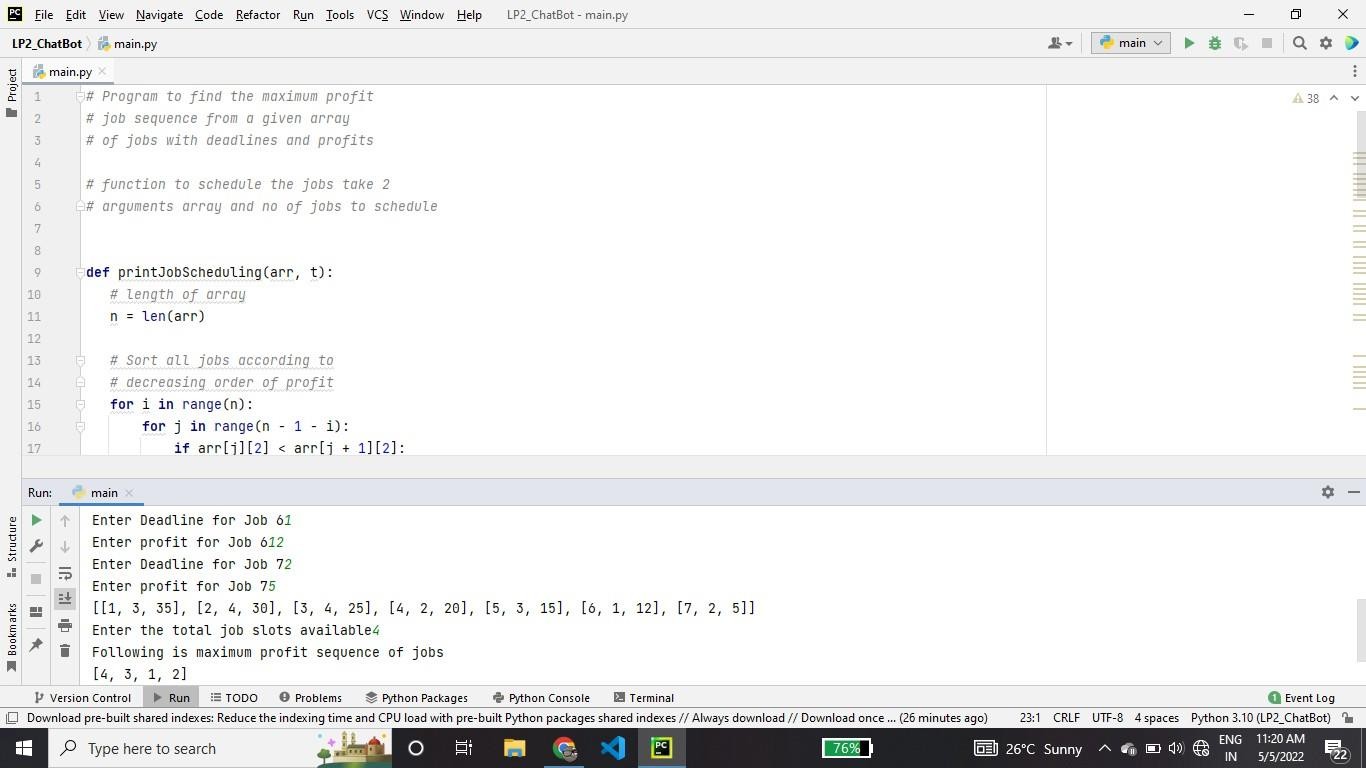
***col.append(int(input("Enter Deadline for Job "+str(i+1))))***

***col.append(int(input("Enter profit for Job "+str(i+1))))***

***arr.append(col) print(arr)***

***# Function Call***

***s=int(input("Enter the total job slots available")) print("Following is maximum profit sequence of jobs") printJobScheduling(arr, s)***



## Prim’s minimal Spanning Tree Algorithm

***# A Python program for Prim's Minimum Spanning Tree (MST) algorithm. # The program is for adjacency matrix representation of the graph***

import sys # Library for INT\_MAX

class Graph():

**self.graph = [[0 for column in range(vertices)]**

**for row in range(vertices)]**

**def init (self, vertices): self.V = vertices**

# A utility function to print the constructed MST stored in parent[]

def printMST(self, parent):

print("Edge \tWeight")

for i in range(1, self.V):

print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minKey(self, key, mstSet):

# Initialize min value

min = sys.maxsize

|  |  |  |
| --- | --- | --- |
| **for v in range(self.V):** | |  |
| **if key[v] < min** | **and mstSet[v] == False:**  **v** | |
| **min = key[v]** |
| **min\_index =** |

return min\_index

# Function to construct and print MST for a graph

# represented using adjacency matrix representation

def primMST(self):

# Key values used to pick minimum weight edge in cut

key = [sys.maxsize] \* self.V

parent = [None] \* self.V # Array to store constructed MST

# Make key 0 so that this vertex is picked as first vertex

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1 # First node is always the root of

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minKey(key, mstSet)

# Put the minimum distance vertex in

# the shortest path tree

mstSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

# graph[u][v] is non zero only for adjacent vertices of m

# mstSet[v] is false for vertices not yet included in MST

# Update the key only if graph[u][v] is smaller than key[v]

if self.graph[u][v] > 0 and mstSet[v] == False and key[v]

> self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

self.printMST(parent)

V=int(input("Enter no of vertices")) g = Graph(V)

n=int(input("Enter total no of edges")) for i in range(0, n):

u = int(input("Enter start vertex: "))

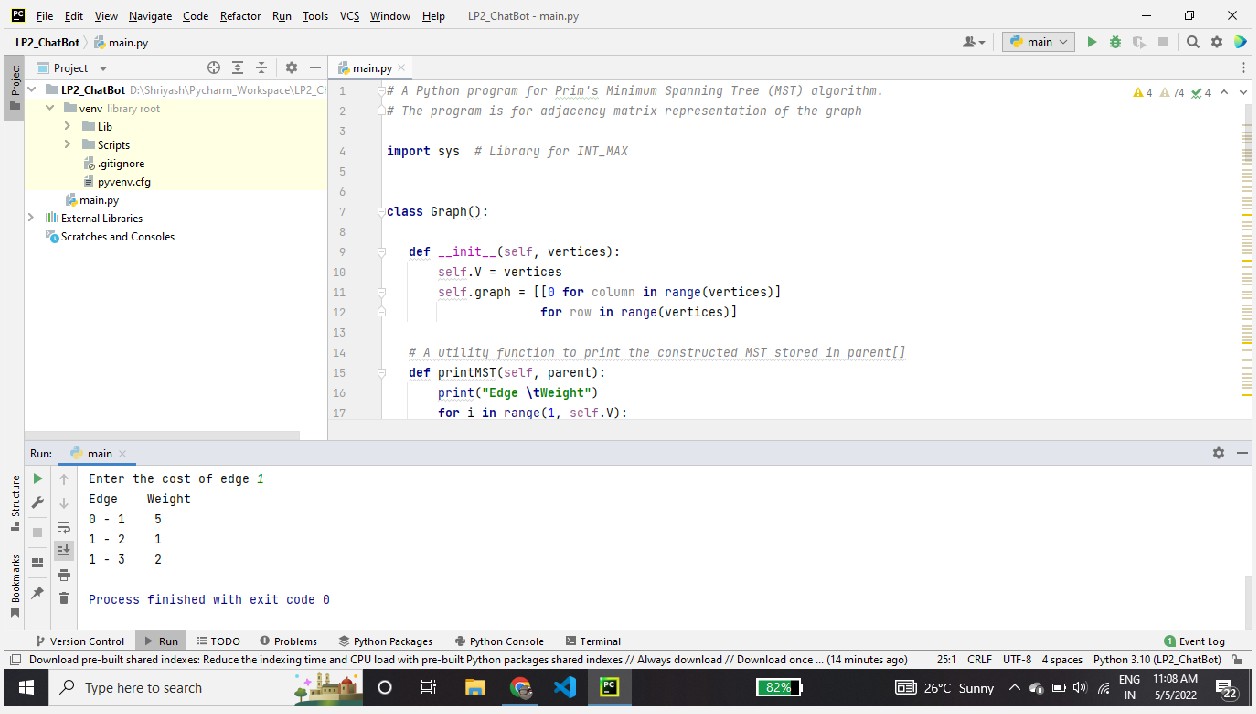
v = int(input("Enter end vertex: "))

c = int(input("Enter the cost of edge "))

g.graph[u][v] = c

g.graph[v][u] = c

g.primMST();



## Kruskal’s Minimal Spanning Tree algorithm

# Python program for Kruskal's algorithm to find

# Minimum Spanning Tree of a given connected, # undirected and weighted graph

from collections import defaultdict

# Class to represent a graph

class Graph:

def init (self, vertices):

self.V = vertices # No. of vertices

self.graph = [] # default dictionary

# to store graph

# function to add an edge to graph

def addEdge(self, u, v, w):

self.graph.append([u, v, w])

# A utility function to find set of an element i

# (uses path compression technique)

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

# A function that does union of two sets of x and y

# (uses union by rank)

def union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

# Attach smaller rank tree under root of

# high rank tree (Union by Rank)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

# If ranks are same, then make one as root

# and increment its rank by one

else:

parent[yroot] = xroot

rank[xroot] += 1

# The main function to construct MST using Kruskal's

# algorithm

def KruskalMST(self):

result = [] # This will store the resultant MST

# An index variable, used for sorted edges

i = 0

# An index variable, used for result[]

e = 0

# Step 1: Sort all the edges in

# non-decreasing order of their

# weight. If we are not allowed to change the

# given graph, we can create a copy of graph

self.graph = sorted(self.graph,

key=lambda item: item[2])

parent = []

rank = []

# Create V subsets with single elements

for node in range(self.V):

parent.append(node)

rank.append(0)

# Number of edges to be taken is equal to V-1

while e < self.V - 1:

# Step 2: Pick the smallest edge and increment

# the index for next iteration

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

# If including this edge does't

# cause cycle, include it in result

# and increment the indexof result

# for next edge

if x != y:

e = e + 1

result.append([u, v, w])

self.union(parent, rank, x, y)

# Else discard the edge

minimumCost = 0

print("Edges in the constructed MST")

for u, v, weight in result:

minimumCost += weight

print("%d -- %d == %d" % (u, v, weight))

print("Minimum Spanning Tree", minimumCost)

# Driver code

g = Graph(int(input("Enter no of vertex"))) t=int(input("Enter total no of edges: ")) for i in range (t):

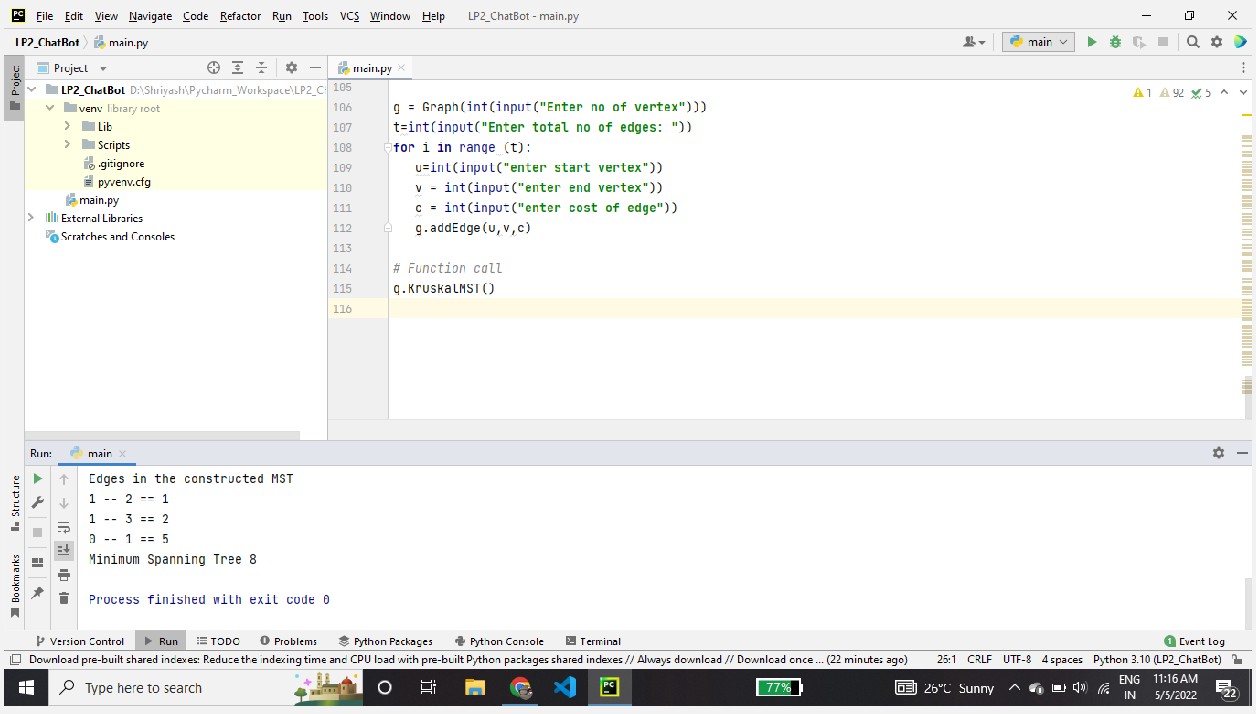
u=int(input("enter start vertex"))

v = int(input("enter end vertex"))

c = int(input("enter cost of edge"))

g.addEdge(u,v,c)

# Function call g.KruskalMST()



## Dijkstra's Shortest Path Algorithm

# Python program for Dijkstra's single

# source shortest path algorithm. The program is # for adjacency matrix representation of the graph class Graph():

def init (self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printSolution(self, dist):

print("Vertex \t Distance from Source: ")

for node in range(self.V):

print(node, "\t\t", dist[node])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = 1e7

# Search not nearest vertex not in the

# shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def dijkstra(self, src):

dist = [1e7] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the

# shortest path tree

sptSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

if (self.graph[u][v] > 0 and

sptSet[v] == False and

dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program V=int(input("Enter no of vertices")) g = Graph(V)

n=int(input("Enter total no of edges")) for i in range(0, n):

u = int(input("Enter start vertex: "))

v = int(input("Enter end vertex: "))

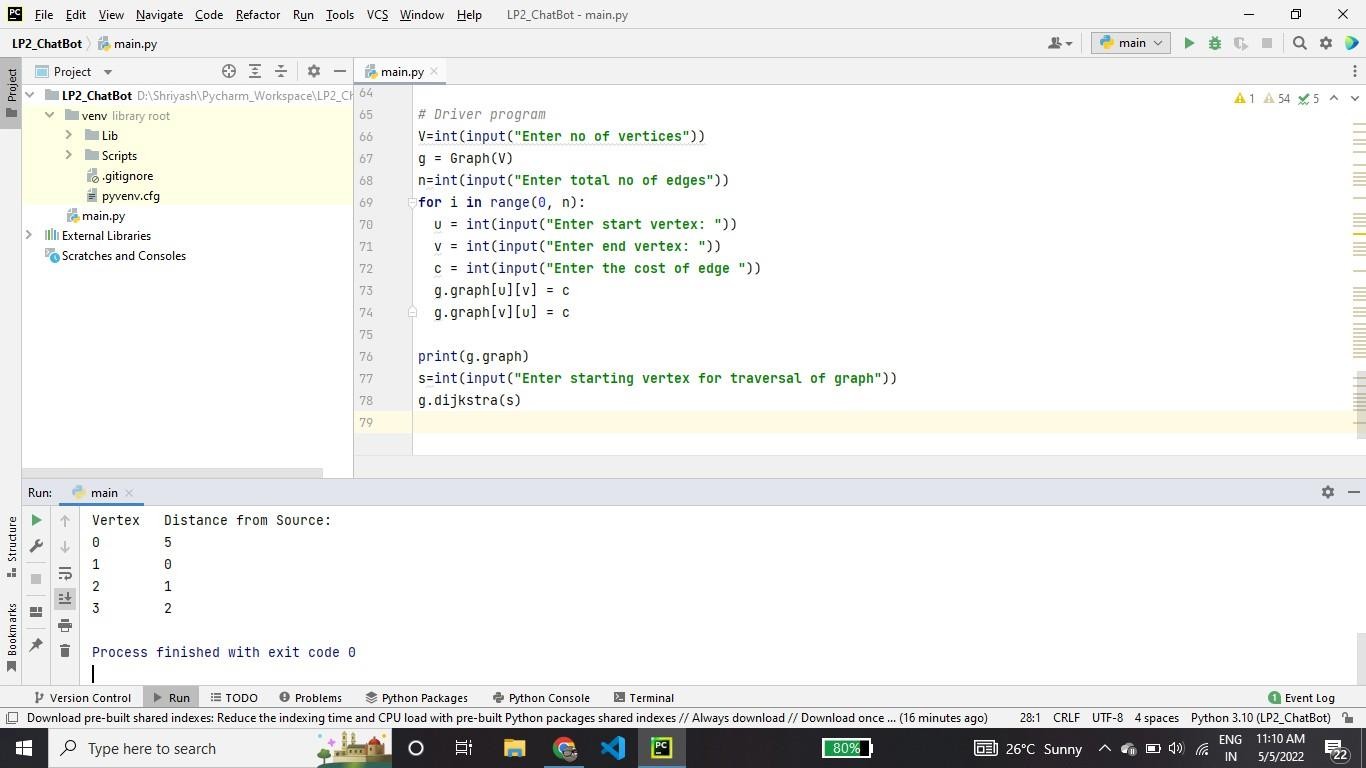
c = int(input("Enter the cost of edge "))

g.graph[u][v] = c

g.graph[v][u] = c

print(g.graph)

s=int(input("Enter starting vertex for traversal of graph")) g.dijkstra(s)



# Practical No 4 Code

## M Coloring Problem

***# Number of vertices in the graph # define 4 4***

***# check if the colored # graph is safe or not***

def isSafe(graph, color):

***# check for every edge***

for i in range(4):

for j in range(i + 1, 4):

if (graph[i][j] and color[j] == color[i]): return False

return True

***# /\* This function solves the m Coloring # problem using recursion. It returns***

***# false if the m colours cannot be assigned, # otherwise, return true and prints***

***# assignments of colours to all vertices. # Please note that there may be more than # one solutions, this function prints one # of the feasible solutions.\*/***

def graphColoring(graph, m, i, color):

***# if current index reached end***

if (i == 4):

***# if coloring is safe***

if (isSafe(graph, color)):

***# Print the solution* printSolution(color) return True**

return False

***# Assign each color from 1 to m***

for j in range(1, m + 1): color[i] = j

***# Recur of the rest vertices***

if (graphColoring(graph, m, i + 1, color)): return True

color[i] = 0 return False

***# /\* A utility function to print solution \*/***

def printSolution(color):

print("Solution Exists:" " Following are the assigned colors ") for i in range(4):

print(color[i],end=" ")

***# Driver code***

if name == ' main ':

***# /\* Create following graph and # test whether it is 3 colorable # (3)---(2)***

***# | / | # | / | # | / |***

***# (0)---(1)***

***# \*/***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **graph =** | **[** |  |  |  |
| **[ 0,** | **1,** | **1,** | **1** | **],** |
| **[ 1,** | **0,** | **1,** | **0** | **],** |
| **[ 1,** | **1,** | **0,** | **1** | **],** |
| **[ 1,** | **0,** | **1,** | **0** | **],** |
| **]** |  |  |  |  |

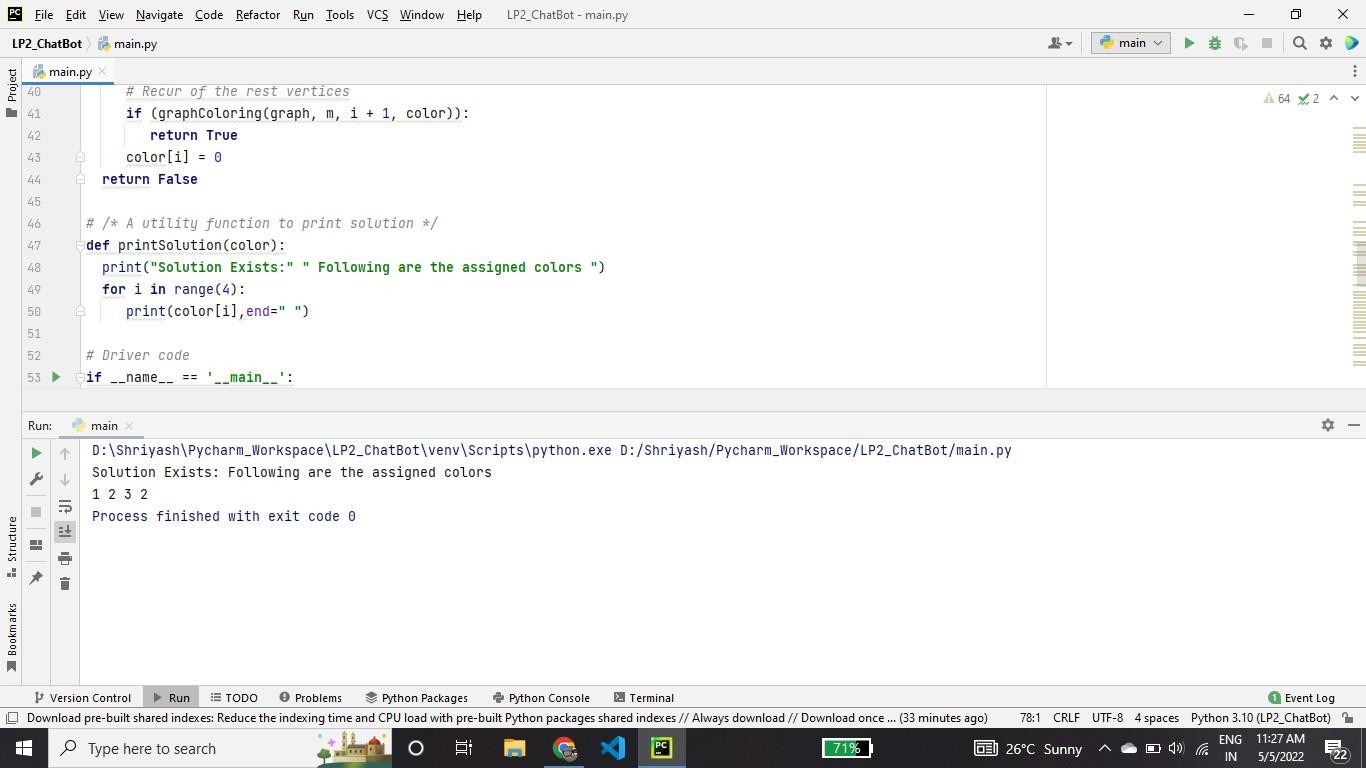
**m = 3 *# Number of colors***

***# Initialize all color values as 0. # This initialization is needed***

***# correct functioning of isSafe()***

color = [0 for i in range(4)]

if (not graphColoring(graph, m, 0, color)): print ("Solution does not exist")



## N Queen Problem

***""" Python3 program to solve N Queen Problem using Branch or Bound """***

N = 8

""" A utility function to print solution """

def printSolution(board): for i in range(N):

for j in range(N): print(board[i][j], end=" ")

print()

""" A Optimized function to check if

a queen can be placed on board[row][col] """

def isSafe(row, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):

if (slashCodeLookup[slashCode[row][col]] or backslashCodeLookup[backslashCode[row][col]] or rowLookup[row]):

return False return True

""" A recursive utility function to solve N Queen problem """

def solveNQueensUtil(board, col, slashCode, backslashCode,

rowLookup, slashCodeLookup, backslashCodeLookup):

***""" base case: If all queens are placed then return True """***

if (col >= N): return True

for i in range(N):

if (isSafe(i, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup)):

""" Place this queen in board[i][col] """ board[i][col] = 1

rowLookup[i] = True

slashCodeLookup[slashCode[i][col]] = True backslashCodeLookup[backslashCode[i][col]] = True

""" recur to place rest of the queens """ if (solveNQueensUtil(board, col + 1,

slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup)):

return True

""" If placing queen in board[i][col] doesn't lead to a solution,then backtrack """

""" Remove queen from board[i][col] """ board[i][col] = 0

rowLookup[i] = False slashCodeLookup[slashCode[i][col]] = False backslashCodeLookup[backslashCode[i][col]] = False

""" If queen can not be place in any row in this column col then return False """

return False

""" This function solves the N Queen problem using Branch or Bound. It mainly uses solveNQueensUtil()to solve the problem. It returns False if queens cannot be placed,otherwise return True or

prints placement of queens in the form of 1s. Please note that there may be more than one solutions,this function prints one of the feasible solutions."""

def solveNQueens():

board = [[0 for i in range(N)]

for j in range(N)]

***# helper matrices***

slashCode = [[0 for i in range(N)]

for j in range(N)] backslashCode = [[0 for i in range(N)]

for j in range(N)]

***# arrays to tell us which rows are occupied***

rowLookup = [False] \* N

***# keep two arrays to tell us # which diagonals are occupied***

x = 2 \* N - 1

slashCodeLookup = [False] \* x backslashCodeLookup = [False] \* x

***# initialize helper matrices***

for rr in range(N): for cc in range(N):

slashCode[rr][cc] = rr + cc backslashCode[rr][cc] = rr - cc + 7

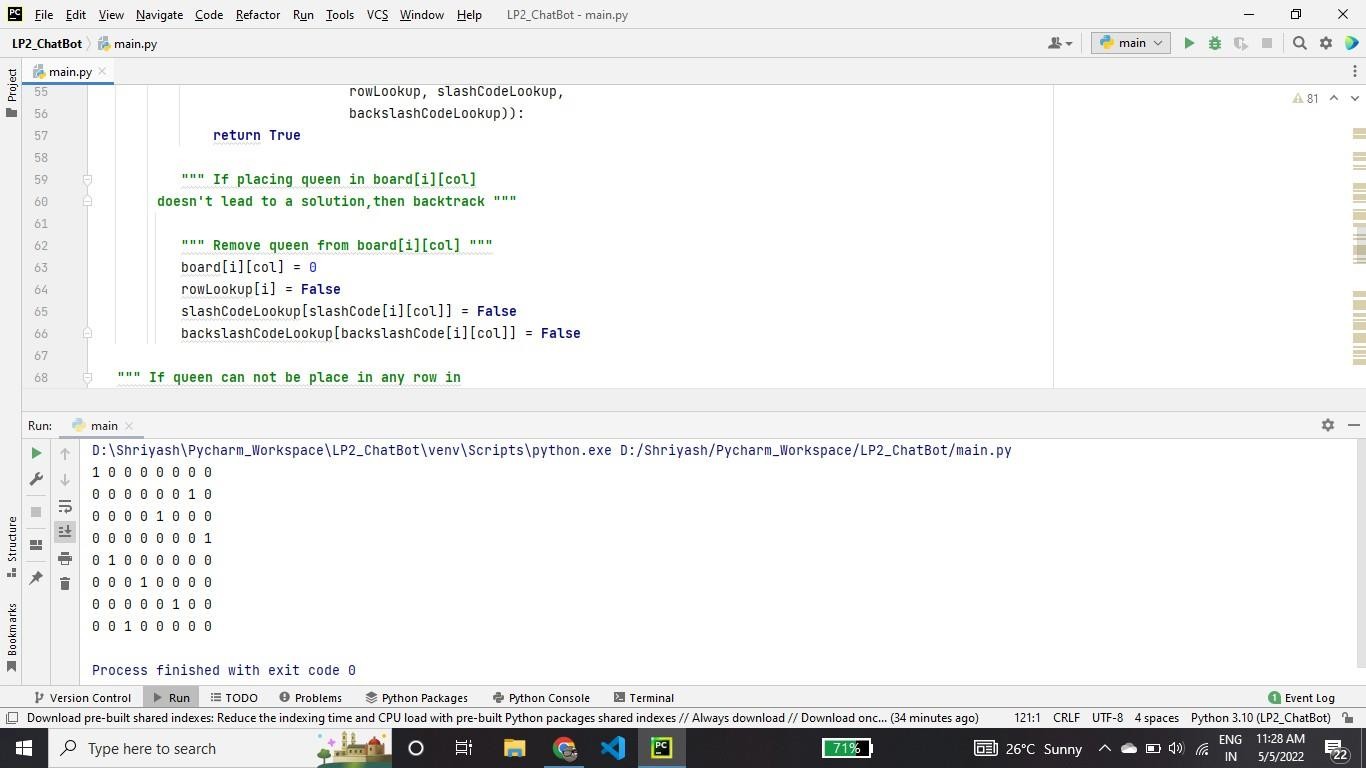
if (solveNQueensUtil(board, 0, slashCode, backslashCode,

rowLookup, slashCodeLookup, backslashCodeLookup) == False):

print("Solution does not exist") return False

***# solution found* printSolution(board) return True**

***# Driver Cde***

solveNQueens()

# Practical no 5 Code

**ChatBot**

#### Simple Python chatbot

import random

name = "Bot\_6282"

resp = {

"name":["My name is {0}".format(name)],

"deliver": [

"Currently due to covid-19 we only provide parcel delivery services through zomato/swiggy.",

"Yeah we are soon looking to open an outlet nearest to you, but for now we only have parcel service. ",

"Sorry only parcel service is available.",

"Our kitchen is situated near aissms college,pune ,only parcel service available."],

"safety": [

"Don't worry our food is 100% safe.",

"We are currently ranked A+ in food safety.",

"Don't worry,Your food is prepared with all safety measures.", ],

"menu":[

"poha --------- 30rs\n " "upma 35rs\n"

"vada sambhar 40rs\n"

"veg pulav 50rs\n"

"panner masala 100rs\n"

"mushroom 90rs\n"

"roti 10rs\n"

"coco-cola 20rs\n",

],

"":[

"I'm Sorry i didn't understand can u please enter valid keywords.", "What do you mean by these?, please enter valid keywords.", "OOps!! can u please enter valid keywords."],

"default": ["This is a default message"] }

def res(message):

if message in resp:

bot286\_message = random.choice(resp[message])

else:

bot286\_message = random.choice(resp["default"])

return bot286\_message

def real(xtext):

if "safety" in xtext:

ytext = "safety"

elif "secure" in xtext:

ytext ="safety"

elif "kitchen" in xtext:

ytext ="safety"

elif "deliver" in xtext:

ytext = "deliver"

elif "address" in xtext:

ytext = "deliver"

elif "name" in xtext:

ytext = "name"

elif "menu" in xtext:

ytext ="menu"

else:

ytext = ""

return ytext

def calculatebill():

n="start"

bill=0

while(n!="stop"):

dish= input("Enter name of dish: ").lower()

if "poha" in dish:

bill+=30

elif "upma" in dish:

bill+=35

elif "vada" in dish:

bill+=40

elif "veg" in dish:

bill+=50

elif "panner" in dish:

bill+=100

elif "mushroom" in dish:

bill+=90

elif "roti" in dish:

bill+=10

elif "coco" in dish:

bill+=20

elif "cancel" in dish:

bill=0

print("Total Bill is: "+str(bill))

n=input("Type 'stop' if done else enter, if you want to cancel order type 'enter' and then 'cancel'")

print("Thank You Order Placed/Cancelled!!!")

def send\_message(message):

#print((message))

response = res(message)

print((response))

if(message=="menu"):

calculatebill()

print("Hi there, My name is {0} \n "

"Welcome to Dhawalikar Pure Veg Restaurant \n"

"How may i assist you, type 'menu' for referring our Menu-Card ".format(name))

while 1:

my\_input = input()

my\_input = my\_input.lower()

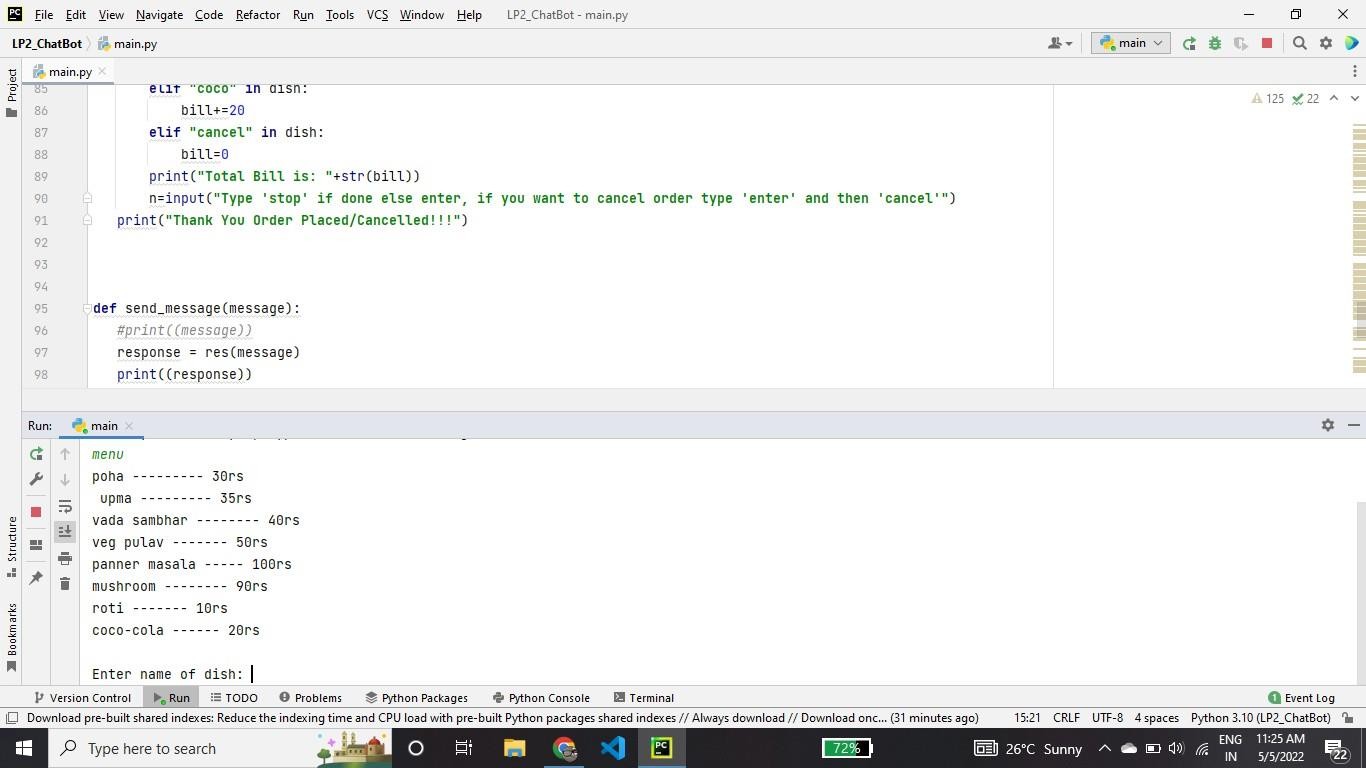
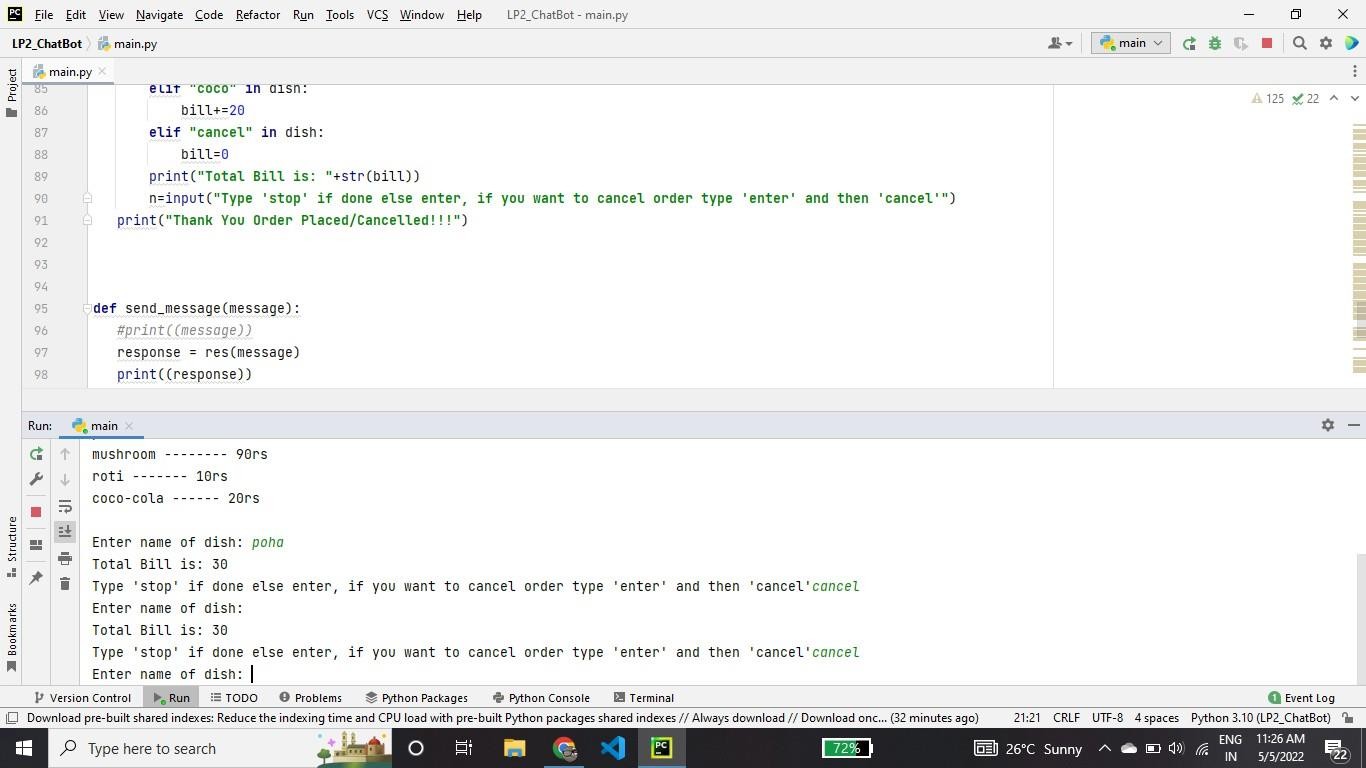
related\_text = real(my\_input)

send\_message(related\_text)

if my\_input == "exit" or my\_input == "stop":

print("BOT: GoodBye!!!")

break



# Practical No 6 Code

go:- hypothesis(Disease),

write('I believe that the patient have'), write(Disease),

nl,

write('TAKE CARE '),

undo.

/\*Hypothesis that should be tested\*/ hypothesis(cold) :- cold, !. hypothesis(flu) :- flu, !. hypothesis(typhoid) :- typhoid, !. hypothesis(measles) :- measles, !. hypothesis(malaria) :- malaria, !. hypothesis(unknown). /\* no diagnosis\*/

/\*Hypothesis Identification Rules\*/ cold :-

verify(headache), verify(runny\_nose), verify(sneezing), verify(sore\_throat), write('Advices and Sugestions:'), nl,

write('1: Tylenol/tab'), nl,

write('2: panadol/tab'), nl,

write('3: Nasal spray'), nl,

write('Please weare warm cloths Because'), nl.

flu :- verify(fever), verify(headache), verify(chills),

verify(body\_ache), write('Advices and Sugestions:'), nl,

write('1: Tamiflu/tab'), nl,

write('2: panadol/tab'), nl,

write('3: Zanamivir/tab'), nl,

write('Please take a warm bath and do salt gargling Because'), nl.

typhoid :- verify(headache), verify(abdominal\_pain), verify(poor\_appetite), verify(fever),

write('Advices and Sugestions:'),

nl,

write('1: Chloramphenicol/tab'), nl,

write('2: Amoxicillin/tab'), nl,

write('3: Ciprofloxacin/tab'), nl,

write('4: Azithromycin/tab'), nl,

write('Please do complete bed rest and take soft Diet Because'), nl.

measles :- verify(fever), verify(runny\_nose), verify(rash), verify(conjunctivitis),

write('Advices and Sugestions:'), nl,

write('1: Tylenol/tab'), nl,

write('2: Aleve/tab'), nl,

write('3: Advil/tab'), nl,

write('4: Vitamin A'), nl,

write('Please Get rest and use more liquid Because'), nl.

malaria :- verify(fever), verify(sweating), verify(headache), verify(nausea), verify(vomiting), verify(diarrhea),

write('Advices and Sugestions:'), nl,

write('1: Aralen/tab'), nl,

write('2: Qualaquin/tab'), nl,

write('3: Plaquenil/tab'), nl,

write('4: Mefloquine'), nl,

write('Please do not sleep in open air and cover your full skin Because'), nl.

/\* how to ask questions \*/ ask(Question) :-

write('Does the patient have following symptom:'), write(Question),

write('? '),

read(Response), nl,

( (Response == yes ; Response == y)

->

assert(yes(Question)) ; assert(no(Question)), fail).

:- dynamic yes/1,no/1.

/\*How to verify something \*/ verify(S) :-

(yes(S)

->

true ; (no(S)

->

fail ; ask(S))).

/\* undo all yes/no assertions\*/ undo :- retract(yes(\_)),fail. undo :- retract(no(\_)),fail. undo.

/\*Output\*/

