Program 1: Write an algorithm, draw a flowchart and develop a program to implement depth first search algorithm.

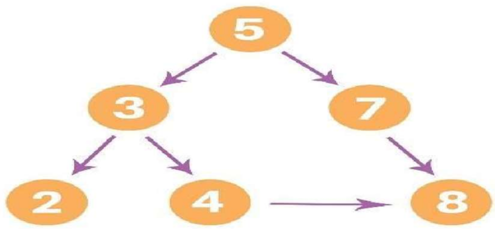
Program:
```
graph1 = {
'a': set(['b', 'c']),
'b': set(['a', 'd', 'e']),
'c': set(['a', 'f']),
'd': set(['b']),
'e': set(['b', 'f']),
'f': set(['c', 'e'])
} def dfs(graph, node, visited): if node not
in visited: visited.append(node) for n in
graph[node]:
            dfs(graph,n, visited)
    return visited
visited = dfs(graph1,'a', []) print(visited)
```

Output:

```
PS D:\PP> & C:/Users/ab/AppData/Local/Programs/Python/Python310/python.exe d:/PP/1.py
['a', 'b', 'e', 'f', 'c', 'd']
PS D:\PP>
```

Program 2: Write an algorithm, draw a flowchart and develop a program to implement breath first search algorithm.

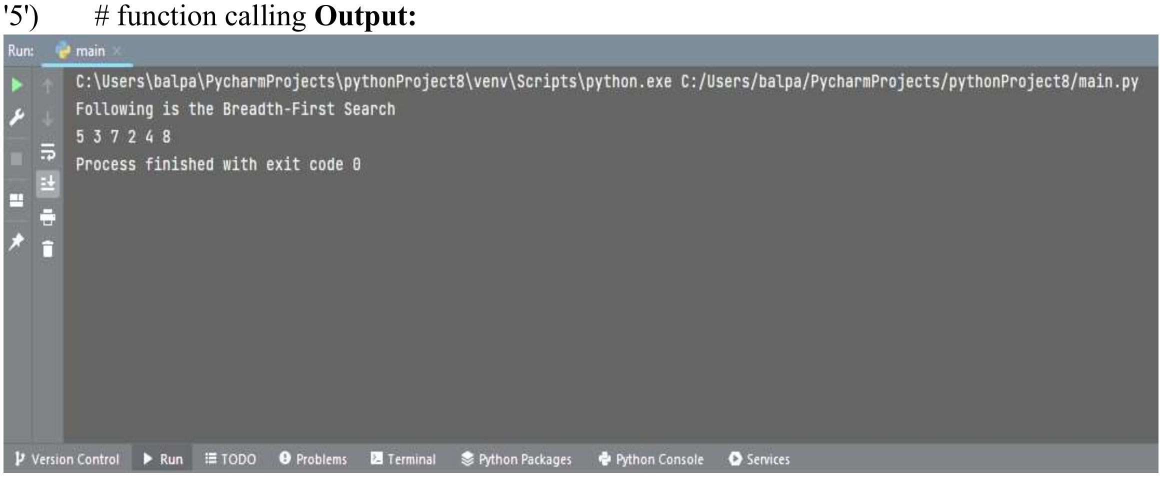
Graph :



Program:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : [] } visited = [] # List for visited nodes. queue = []
#Initialize a queue defbfs(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:



Program 3: Write an algorithm, draw a flowchart and develop a program to implement alpha beta search.

Program:

```
tree = [[[6, 2, 3], [9, -7, -8]], [[10, 5, 6], [-2, 5, 4]]] root = 0
pruned = 0
def children(branch, depth, alpha, beta):
    global tree global root global
    pruned i = 0 for child in branch:
        if type(child) is list:
            (nalpha, nbeta) = children(child, depth + 1, alpha, beta) if depth % 2 ==
            1:
                beta = nalpha if nalpha < beta else beta else:
                alpha = nbeta if nbeta > alpha else alpha
            branch[i] = alpha if depth % 2 == 0 else beta
            i += 1 else: if depth % 2 == 0 and alpha < child:
        alpha = child
            if depth % 2 == 1 and beta > child:
                beta = child
            if alpha >= beta: pruned += 1 break
    if depth == root:
        tree = alpha if root == 0 else beta
    return (alpha, beta)
def alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
    global tree global pruned global
    root
    (alpha, beta) = children(tree, start, upper, lower) if __name__==
    "__main__":
        print ("(alpha, beta): ", alpha, beta) print ("Result:
        ", tree) print ("Times pruned: ", pruned)
    return (alpha, beta, tree, pruned)
if __name__== "__main__":
    alphabeta(None)
```

Output:

```
PS D:\PP> & C:/Users/ab/AppData/Local/Programs/Python/Python310/python.exe d:/PP/5.py
(alpha, beta):  6 15
Result:  6
Times pruned:  1
PS D:\PP>
```

Program 4:Write an algorithm, draw a flowchart and develop a program to implement A* algorithm.
Program:

```
def aStarAlgo(start_node, stop_node):
        open_set = set(start_node)         closed_set = set()
        g = {} #store distance from starting node         parents = {}# parents
contains an adjacency map of all nodes
        #ditance of starting node from itself is zero
        g[start_node] = 0
        #start_node is root node i.e it has no parent nodes         #so
start_node is set to its own parent node
        parents[start_node] = start_node
        while len(open_set) > 0:
            n = None
            #node with lowest f() is found             for v in open_set:                 if n ==
None or g[v] + heuristic(v) < g[n] + heuristic(n):                     n = v
            if n == stop_node or Graph_nodes[n] == None:                 pass
else:                 for (m, weight) in get_neighbors(n):
                    #nodes 'm' not in first and last set are added to first
                    #n is set its parent                     if m not in open_set and m
not in closed_set:                         open_set.add(m)
parents[m] = n
                        g[m] = g[n] + weight
                    #for each node m,compare its distance from start i.e g(m) to the
                    #from start through n node                     else:
if g[m] > g[n] + weight:                             #update g(m)
g[m] = g[n] + weight                             #change parent of
m to n
                            parents[m] = n
                            #if m in closed set,remove and add to open
if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)

            if n == None:                 print('Path does not
exist!')                 return None
            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node             if n ==
stop_node:
                path = []
                while parents[n] != n:                     path.append(n)
                    n = parents[n]
                path.append(start_node)
                path.reverse()
                print('Path found: {}'.format(path))
                return path
            # remove n from the open_list, and add it to closed_list
# because all of his neighbors were inspected
            open_set.remove(n)
            closed_set.add(n)
        print('Path does not exist!')         return None
#define fuction to return neighbor and its distance
```

```
#from the passed node def
get_neighbors(v):      if v in Graph_nodes:
return Graph_nodes[v]      else:
        return None
#for simplicity we ll consider heuristic distances given #and this
function returns heuristic distance for all nodes def heuristic(n):
H_dist = {
            'A': 11,             'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
} aStarAlgo('A', 'G')
```
Output :

Program 5:Write an algorithm, draw a flowchart and develop a program to implement AO* algorithm.

Program:

```
def recAOStar(n): global finalPath
print("Expanding Node:",n)
    and_nodes = [] or_nodes =[]
if(n in allNodes):
if 'AND' in allNodes[n]: and_nodes =
            allNodes[n]['AND']
if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
if len(and_nodes)==0 and len(or_nodes)==0: return
solvable = False marked ={}
while not solvable:
if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least,min_cost_group_least = least_cost_group(and_nodes,or_nodes,{})
solvable = True change_heuristic(n,min_cost_least)
            optimal_child_group[n] = min_cost_group_least
continue min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)
        is_expanded = False
if len(min_cost_group)>1:
if(min_cost_group[0] in allNodes): is_expanded =
True recAOStar(min_cost_group[0])
if(min_cost_group[1] in allNodes): is_expanded =
True recAOStar(min_cost_group[1]) else:
if(min_cost_group in allNodes): is_expanded =
True recAOStar(min_cost_group) if
is_expanded:
            min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_nodes,
{}) if min_cost_group == min_cost_group_verify: solvable = True
change_heuristic(n, min_cost_verify) optimal_child_group[n] =
min_cost_group
else:
solvable = True change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group
marked[min_cost_group]=1 return heuristic(n)
def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
for node_pair in and_nodes: if not node_pair[0] +
node_pair[1] in marked:
cost = 0
cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2 node_wise_cost[node_pair[0] +
            node_pair[1]] = cost
for node in or_nodes: if not node in
marked:
cost = 0
cost = cost + heuristic(node) + 1 node_wise_cost[node] = cost
    min_cost = 999999 min_cost_group = None
for costKey in node_wise_cost:
if node_wise_cost[costKey] < min_cost: min_cost =
            node_wise_cost[costKey] min_cost_group = costKey
```

```
return [min_cost, min_cost_group]
def heuristic(n): return H_dist[n]
def change_heuristic(n, cost):
    H_dist[n] = cost return
def print_path(node):
print(optimal_child_group[node], end="") node =
optimal_child_group[node] if len(node) > 1: if node[0] in
optimal_child_group:
print("->", end="") print_path(node[0])
if node[1] in optimal_child_group:
print("->", end="") print_path(node[1])
else: if node in optimal_child_group:
print("->", end="") print_path(node)
H_dist = {
'A': -1,
'B': 4,
'C': 2,
'D': 3,
'E': 6,
'F': 8,
'G': 2,
'H': 0,
'I': 0,
'J': 0
}
allNodes = {
'A': {'AND': [('C', 'D')], 'OR': ['B']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': [('H', 'I')]},
 'D': {'OR': ['J']} } optimal_child_group = {} optimal_cost
= recAOStar('A') print('Nodes which gives optimal cost
are') print_path('A') print('\nOptimal Cost is :: ',
optimal_cost)
```

Output:

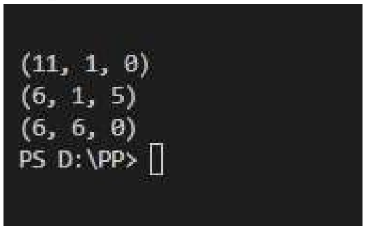
Program 6: Write an algorithm, draw a flowchart and develop a program to solve water jug problem.

Program:

```
# 3 water jugs capacity -> (x,y,z) where x>y>z # initial state (12,0,0)
# final state (6,6,0)
capacity = (12,8,5)
# Maximum capacities of 3 jugs -> x,y,z
x = capacity[0] y = capacity[1]
z = capacity[2]
# to mark visited states memory = {}
# store solution path ans = []
def get_all_states(state):
# Let the 3 jugs be called a,b,c a = state[0] b
   = state[1] c = state[2]
   if(a==6 and b==6): ans.append(state)
      return True
   # if current state is already visited earlier if((a,b,c) in
   memory): return False
   memory[(a,b,c)] = 1
   #empty jug a if(a>0):
      #empty a into b if(a+b<=y):
         if( get_all_states((0,a+b,c)) ):
            ans.append(state) return True
      else:
         if( get_all_states((a-(y-b), y, c)) ):
            ans.append(state) return True
            #empty a into c
      if(a+c<=z): if( get_all_states((0,b,a+c)) ):
      ans.append(state) return True else: if(
      get_all_states((a-(z-c), b, z)) ):
            ans.append(state) return True
   #empty jug b if(b>0):
      #empty b into a if(a+b<=x): if(
      get_all_states((a+b, 0, c)) ):
            ans.append(state) return True else: if(
      get_all_states((x, b-(x-a), c)) ):
            ans.append(state) return True #empty b
      into c if(b+c<=z): if( get_all_states((a, 0, b+c))
      ):
            ans.append(state) return True else: if(
      get_all_states((a, b-(z-c), z)) ):
            ans.append(state) return True
   #empty jug c if(c>0):
      #empty c into a if(a+c<=x): if(
      get_all_states((a+c, b, 0)) ):
            ans.append(state) return True else: if(
      get_all_states((x, b, c-(x-a))) ):
            ans.append(state) return True #empty c
      into b if(b+c<=y): if( get_all_states((a, b+c, 0))
      ):
            ans.append(state) return True else: if(
      get_all_states((a, y, c-(y-b))) ):
```

```
                ans.append(state) return True
    return False
initial_state = (12,0,0) print("Starting
work...\n") get_all_states(initial_state)
ans.reverse() for i in ans: print(i)
```

Output:



```
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
PS D:\PP>
```

Program 7: Write an algorithm, draw a flowchart and develop a program to implement Tic-Tac-Toe game using Min-Max algorithm.

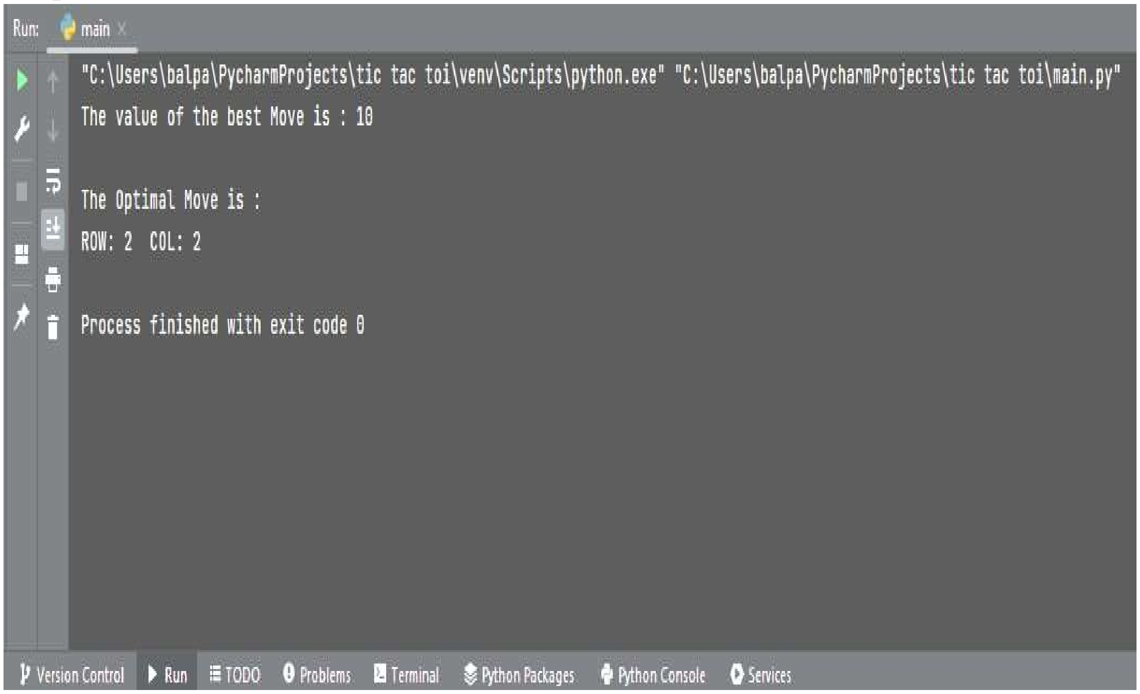
Program: player, opponent = 'x', 'o'

```
# This function returns true if there are moves # remaining
on the board. It returns false if # there are no moves left to
play. def isMovesLeft(board): for i in range(3):
        for j in range(3):
            if (board[i][j] == '_'):
                return True
    return False
# This is the evaluation function as discussed # in the previous
article ( http://goo.gl/sJgv68 ) def evaluate(b):
    # Checking for Rows for X or O victory. for row in
    range(3):
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]):
            if (b[row][0] == player):
                return 10
            elif (b[row][0] == opponent): return -10
    # Checking for Columns for X or O victory. for col in
    range(3):
        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]):
            if (b[0][col] == player):
                return 10
            elif (b[0][col] == opponent): return -10
    # Checking for Diagonals for X or O victory. if (b[0][0] ==
    b[1][1] and b[1][1] == b[2][2]):
        if (b[0][0] == player):
            return 10
        elif (b[0][0] == opponent):
            return -10
    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]):
        if (b[0][2] == player): return 10
        elif (b[0][2] == opponent): return -10
    # Else if none of them have won then return 0 return 0
# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board def minimax(board,
depth, isMax): score = evaluate(board)
    # If Maximizer has won the game return his/her
    # evaluated score if (score ==
    10):
        return score
    # If Minimizer has won the game return his/her
    # evaluated score if (score == -
    10):
        return score
    # If there are no more moves and no winner then
    # it is a tie if (isMovesLeft(board) == False):
    return 0
    # If this maximizer's move if (isMax):
    best = -1000
```

```
        # Traverse all cells for i in range(3):
            for j in range(3):
                # Check if cell is empty if (board[i][j]
                == '_'):
                    # Make the move
                    board[i][j] = player
                    # Call minimax recursively and choose # the
                    maximum value best = max(best, minimax(board,
                    depth + 1, not isMax))
                    # Undo the move
                    board[i][j] = '_'
        return best
    # If this minimizer's move else:
        best = 1000
        # Traverse all cells for i in range(3):
            for j in range(3):
                # Check if cell is empty if (board[i][j]
                == '_'):
                    # Make the move
                    board[i][j] = opponent
                    # Call minimax recursively and choose # the minimum
                    value
                    best = min(best, minimax(board, depth + 1, not isMax))
                    # Undo the move
                    board[i][j] = '_'
        return best
# This will return the best possible move for the player def
findBestMove(board): bestVal = -1000 bestMove = (-1, -1)
    # Traverse all cells, evaluate minimax function for # all empty
    cells. And return the cell with optimal # value.
for i in range(3):
        for j in range(3):
            # Check if cell is empty if (board[i][j]
            == '_'):
                # Make the move
                board[i][j] = player
                # compute evaluation function for this # move.
                moveVal = minimax(board, 0, False)
                # Undo the move
                board[i][j] = '_'
                # If the value of the current move is
                # more than the best value, then update
                # best/ if (moveVal > bestVal):
                bestMove = (i, j) bestVal = moveVal
    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code board = [
    ['x', 'o', 'x'],
    ['o', 'o', 'x'],
    ['_', '_', '_']
```

]
bestMove = findBestMove(board)
print("The Optimal Move is :") print("ROW:", bestMove[0], "
COL:", bestMove[1])

Output:

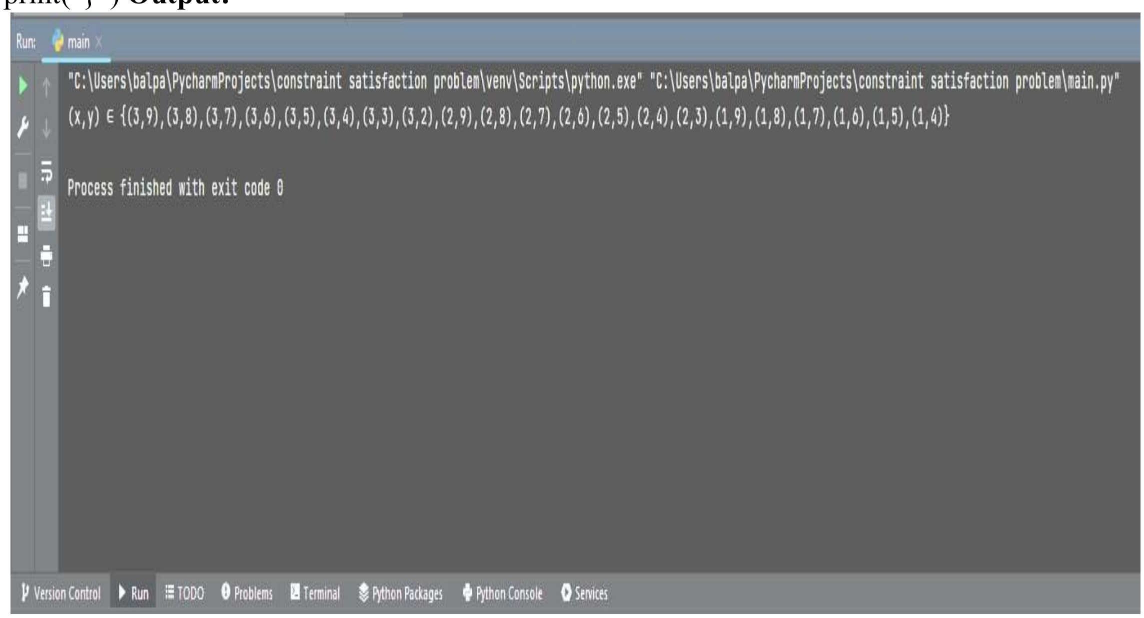


Pogram 8:Write an algorithm, draw a flowchart and develop a program to solve constraint satisfaction problem.

Program:

```
import constraint problem =

constraint.Problem()

problem.addVariable('x', [1,2,3]) problem.addVariable('y', range(10))
def our_constraint(x, y): if x + y >= 5: return True

problem.addConstraint(our_constraint, ['x','y']) solutions =

problem.getSolutions()

# Easier way to print and see all solutions # for solution
in solutions: # print(solution)
# Prettier way to print and see all solutions
length = len(solutions) print("(x,y)∈ {", end="") for index,
solution in enumerate(solutions):
    if index == length - 1:
        print("({},{})".format(solution['x'], solution['y']), end="")
    else:
        print("({},{}),".format(solution['x'], solution['y']), end="")
print("}")
```

Output:



Program 9:Write an algorithm, draw a flowchart and develop a program for Hill climbing problem.

Program: import math

```
increment = 0.1 startingPoint = [1,
1] point1 = [1,4] point2 = [4,2]
point3 = [3,2] point4 = [2,1]
def distance(x1, y1, x2, y2):
    dist = math.pow(x2-x1, 2) + math.pow(y2-y1, 2) return dist
def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):
    d1 = distance(x1, y1, px1, py1) d2 =
    distance(x1, y1, px2, py2) d3 = distance(x1, y1,
    px3, py3) d4 = distance(x1, y1, px4, py4) return
    d1 + d2 + d3 + d4
def newDistance(x1, y1, point1, point2, point3, point4):
    d1 = [x1, y1]
    d1temp = sumOfDistances(x1, y1, point1[0],point1[1], point2[0],point2[1],
point3[0],point3[1], point4[0],point4[1] )
    d1.append(d1temp) return d1
minDistance = sumOfDistances(startingPoint[0], startingPoint[1], point1[0],point1[1],
point2[0],point2[1],point3[0],point3[1], point4[0],point4[1] ) flag = True
def newPoints(minimum, d1, d2, d3, d4):
    if d1[2] == minimum:
        return [d1[0], d1[1]]
    elif d2[2] == minimum:
        return [d2[0], d2[1]]
    elif d3[2] == minimum:
        return [d3[0], d3[1]]
    elif d4[2] == minimum:
        return [d4[0], d4[1]]
i = 1 while flag: d1 =
newDistance(startingPoin
t[0]+increment,
startingPoint[1], point1,
point2, point3,
point4) d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point2, point3,
point4) d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point2, point3,
point4) d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point2, point3,
point4) print (i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2))
    minimum = min(d1[2], d2[2], d3[2], d4[2]) if minimum <
    minDistance: startingPoint = newPoints(minimum, d1, d2, d3, d4)
        minDistance = minimum
        #print i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2) i+=1 else:
        flag = False
```

Output:

```
PS D:\PP> & C:/Users/ab/AppData/Local/Programs/Python/Python310/python.exe d:/PP/6.py
1   1 1
2   1.1 1
3   1.2 1
4   1.3 1
5   1.3 1.1
6   1.4 1.1
7   1.4 1.2
8   1.5 1.2
9   1.5 1.3
10   1.6 1.3
11   1.6 1.4
12   1.7 1.4
13   1.7 1.5
```

Program 10:Write an algorithm, draw a flowchart and develop a program to shuffle Deck of cards.
Program:

```
# Python program to shuffle a deck of card
# importing modules import itertools,
random
# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))
# shuffle the cards random.shuffle(deck)
# draw five cards print("You got:")
for i in range(5):
  print(deck[i][0], "of", deck[i][1]) Output:
```