**DYNAMIC PROGRAMMING AND BACKTRACKING**

DYNAMIC PROGRAMMING

**PROGRAM 1:** LONGEST COMMON SUBSEQUENCE

**DESCRIPTION:**

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

**CODE:**

def lcs\_algo(S1, S2, m, n):

L = [[0 for x in range(n+1)] for x in range(m+1)]

for i in range(m+1):

for j in range(n+1):

if i == 0 or j == 0:

L[i][j] = 0

elif S1[i-1] == S2[j-1]:

L[i][j] = L[i-1][j-1] + 1

else:

L[i][j] = max(L[i-1][j], L[i][j-1])

index = L[m][n]

lcs\_algo = [""] \* (index+1)

lcs\_algo[index] = ""

i = m

j = n

while i > 0 and j > 0:

if S1[i-1] == S2[j-1]:

lcs\_algo[index-1] = S1[i-1]

i -= 1

j -= 1

index -= 1

elif L[i-1][j] > L[i][j-1]:

i -= 1

else:

j -= 1

print("S1 : " + S1 + "\nS2 : " + S2)

print("LCS: " + "".join(lcs\_algo))

S1 = "ACADB"

S2 = "CBDA"

m = len(S1)

n = len(S2)

lcs\_algo(S1, S2, m, n)

**OUTPUT: **

**PROGRAM 2:** OPTIMAL BINARY SEARCH TREE

**DESCRIPTION:**

Optimal BST problem has both properties of a dynamic programming problem. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result.

**CODE:**

INT\_MAX = 2147483647

def optimalSearchTree(keys, freq, n):

cost = [[0 for x in range(n)]

for y in range(n)]

for i in range(n):

cost[i][i] = freq[i]

for L in range(2, n + 1):

for i in range(n - L + 2):

j = i + L - 1

off\_set\_sum = sum(freq, i, j)

if i >= n or j >= n:

break

cost[i][j] = INT\_MAX

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

c = 0

if (r > i):

c += cost[i][r - 1]

if (r < j):

c += cost[r + 1][j]

c += off\_set\_sum

if (c < cost[i][j]):

cost[i][j] = c

return cost[0][n - 1]

def sum(freq, i, j):

s = 0

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

s += freq[k]

return s

if \_\_name\_\_ == '\_\_main\_\_':

keys = [10, 12, 20]

freq = [34, 8, 50]

n = len(keys)

print("Cost of Optimal BST is",

optimalSearchTree(keys, freq, n))

**OUTPUT: **

**BACKTRACKING**

**PROGRAM 3:** 8 QUEENS PROBLEM

**DESCRIPTION:**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens.

**CODE:**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print(board[i][j], end = " ")

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1),

range(col, -1, -1)):

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

return False

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):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

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:

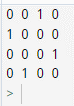
print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

**OUTPUT: **

**PROGRAM 4:**

**DESCRIPTION:**

The idea is to assign colors one by one to different vertices, starting from vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not.

**CODE:**

class Graph():

def \_\_init\_\_(self, vertices):

self.V = vertices

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

for row in range(vertices)]

def isSafe(self, v, colour, c):

for i in range(self.V):

if self.graph[v][i] == 1 and colour[i] == c:

return False

return True

def graphColourUtil(self, m, colour, v):

if v == self.V:

return True

for c in range(1, m + 1):

if self.isSafe(v, colour, c) == True:

colour[v] = c

if self.graphColourUtil(m, colour, v + 1) == True:

return True

colour[v] = 0

def graphColouring(self, m):

colour = [0] \* self.V

if self.graphColourUtil(m, colour, 0) == None:

return False

print("Solution exist and Following are the assigned colours:")

for c in colour:

print(c, end=' ')

return True

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph(4)

g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]

m = 3

g.graphColouring(m)

**OUTPUT: **

**PROGRAM 5:** HAMILTONIAN CYCLE

**DESCRIPTION:**

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.

**CODE:**

class Graph():

def \_\_init\_\_(self, vertices):

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

for row in range(vertices)]

self.V = vertices

def isSafe(self, v, pos, path):

if self.graph[ path[pos-1] ][v] == 0:

return False

for vertex in path:

if vertex == v:

return False

return True

def hamCycleUtil(self, path, pos):

if pos == self.V:

if self.graph[ path[pos-1] ][ path[0] ] == 1:

return True

else:

return False

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

if self.isSafe(v, pos, path) == True:

path[pos] = v

if self.hamCycleUtil(path, pos+1) == True:

return True

path[pos] = -1

return False

def hamCycle(self):

path = [-1] \* self.V

path[0] = 0

if self.hamCycleUtil(path,1) == False:

print ("Solution does not exist\n")

return False

self.printSolution(path)

return True

def printSolution(self, path):

print ("Solution Exists: Following",

"is one Hamiltonian Cycle")

for vertex in path:

print (vertex, end = " ")

print (path[0], "\n")

g1 = Graph(5)

g1.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],

[0, 1, 0, 0, 1,],[1, 1, 0, 0, 1],

[0, 1, 1, 1, 0], ]

g1.hamCycle();

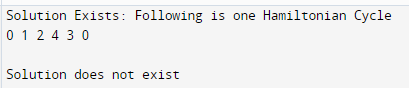
g2 = Graph(5)

g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],

[0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],

[0, 1, 1, 0, 0], ]

g2.hamCycle();

**OUTPUT: **