# **Selection Sort**

public class SelectionSort

{

public static void main(String[] args)

{

int [] arr= {5,8,10,4,1};

for(int i = 0;i&lt;=4;++i)

{

int min=arr[i];

for(int j=i+1;j&lt;arr.length;++j)

{

if(arr[i] &gt; arr[j] )

{

min = arr[j];

arr[j] = arr[i];

arr[i] = min;

}

}

System.out.print(arr[i]+&quot; &quot;);

}

}

}

# **BFS**

// Java program to print BFS traversal from a given source vertex.

// BFS(int s) traverses vertices reachable from s.

import java.io.\*;

import java.util.\*;

// This class represents a directed graph using adjacency list

// representation

class Graph

{

private int V; // No. of vertices

private LinkedList<Integer> adj[]; //Adjacency Lists

// Constructor

Graph(int v)

{

V = v;

adj = new LinkedList[v];

for (int i=0; i<v; ++i)

adj[i] = new LinkedList();

}

// Function to add an edge into the graph

void addEdge(int v,int w)

{

adj[v].add(w);

}

// prints BFS traversal from a given source s

void BFS(int s)

{

// Mark all the vertices as not visited(By default

// set as false)

boolean visited[] = new boolean[V];

// Create a queue for BFS

LinkedList<Integer> queue = new LinkedList<Integer>();

// Mark the current node as visited and enqueue it

visited[s]=true;

queue.add(s);

while (queue.size() != 0)

{

// Dequeue a vertex from queue and print it

s = queue.poll();

System.out.print(s+" ");

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it

// visited and enqueue it

Iterator<Integer> i = adj[s].listIterator();

while (i.hasNext())

{

int n = i.next();

if (!visited[n])

{

visited[n] = true;

queue.add(n);

}

}

}

}

// Driver method to

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Following is Breadth First Traversal "+

"(starting from vertex 2)");

g.BFS(2);

}

}

# **DFS**

// Java program to print DFS

// mtraversal from a given

// graph

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Graph {

private int V; // No. of vertices

// Array of lists for

// Adjacency List Representation

private LinkedList<Integer> adj[];

// Constructor

@SuppressWarnings("unchecked") Graph(int v)

{

V = v;

adj = new LinkedList[v];

for (int i = 0; i < v; ++i)

adj[i] = new LinkedList();

}

// Function to add an edge into the graph

void addEdge(int v, int w)

{

adj[v].add(w); // Add w to v's list.

}

// A function used by DFS

void DFSUtil(int v, boolean visited[])

{

// Mark the current node as visited and print it

visited[v] = true;

System.out.print(v + " ");

// Recur for all the vertices adjacent to this

// vertex

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

// The function to do DFS traversal.

// It uses recursive

// DFSUtil()

void DFS(int v)

{

// Mark all the vertices as

// not visited(set as

// false by default in java)

boolean visited[] = new boolean[V];

// Call the recursive helper

// function to print DFS

// traversal

DFSUtil(v, visited);

}

// Driver Code

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println(

"Following is Depth First Traversal "

+ "(starting from vertex 2)");

g.DFS(2);

}

}

# **A\***

graph= {'A':{'B':1, 'C':3, 'D':7},

'B':{'D':5},

'C':{'D':12},

}

state = {'A':1,

'B':1,

'C':1,

'D':1}

def A\_star(graph, start\_node, end\_node, state):

m ={}

value = 0

if start\_node == end\_node:

return

for node, v in graph[start\_node].items():

value = state[node] + v

m[node] = value

if (m != {}):

path[min(m, key=m.get)] = value

A\_star(graph, min(m, key=m.get), end\_node, state)

return path

path = {}

path['A'] = 0

bp = (A\_star(graph,'A','D',state))

print (list(bp))

print("Total cost: {}".format(sum(bp.values())))

print(bp)

# **NQUEEN**

/\* Java program to solve N Queen Problem using

backtracking \*/

public class NQueenProblem {

final int N = 4;

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

void printSolution(int board[][])

{

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

System.out.print(" " + board[i][j]

+ " ");

System.out.println();

}

}

/\* A utility function to check if a queen can

be placed on board[row][col]. Note that this

function is called when "col" queens are already

placed in columns from 0 to col -1. So we need

to check only left side for attacking queens \*/

boolean isSafe(int board[][], int row, int col)

{

int i, j;

/\* Check this row on left side \*/

for (i = 0; i < col; i++)

if (board[row][i] == 1)

return false;

/\* Check upper diagonal on left side \*/

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j] == 1)

return false;

/\* Check lower diagonal on left side \*/

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j] == 1)

return false;

return true;

}

/\* A recursive utility function to solve N

Queen problem \*/

boolean solveNQUtil(int board[][], int col)

{

/\* base case: If all queens are placed

then return true \*/

if (col >= N)

return true;

/\* Consider this column and try placing

this queen in all rows one by one \*/

for (int i = 0; i < N; i++) {

/\* Check if the queen can be placed on

board[i][col] \*/

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;

/\* If placing queen in board[i][col]

doesn't lead to a solution then

remove queen from board[i][col] \*/

board[i][col] = 0; // BACKTRACK

}

}

/\* If the queen can not be placed in any row in

this column col, then return false \*/

return false;

}

/\* This function solves the N Queen problem using

Backtracking. It mainly uses solveNQUtil () to

solve the problem. It returns false if queens

cannot be placed, otherwise, return true and

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.\*/

boolean solveNQ()

{

int board[][] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false) {

System.out.print("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

// driver program to test above function

public static void main(String args[])

{

NQueenProblem Queen = new NQueenProblem();

Queen.solveNQ();

}

}

# **PYTHON**

# **Selection Sort**

# Python program for implementation of Selection

# Sort

import sys

A = [64, 25, 12, 22, 11]

# 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]),

# **BFS**

# 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):

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] \* (len(self.graph))

# 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()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print ("Following is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)

# **DFS**

# Python3 program to print DFS traversal

# from a 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

# Create a graph given

# in the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")

g.DFS(2)

# **A\***

graph= {'A':{'B':1, 'C':3, 'D':7},

'B':{'D':5},

'C':{'D':12},

}

state = {'A':1,

'B':1,

'C':1,

'D':1}

def A\_star(graph, start\_node, end\_node, state):

m ={}

value = 0

if start\_node == end\_node:

return

for node, v in graph[start\_node].items():

value = state[node] + v

m[node] = value

if (m != {}):

path[min(m, key=m.get)] = value

A\_star(graph, min(m, key=m.get), end\_node, state)

return path

path = {}

path['A'] = 0

bp = (A\_star(graph,'A','D',state))

print (list(bp))

print("Total cost: {}".format(sum(bp.values())))

print(bp)

# **NQUEEN**

# Python program to solve N Queen

# Problem using backtracking

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

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

print()

# A utility function to check if a queen can

# be placed on board[row][col]. Note that this

# function is called when "col" queens are

# already placed in columns from 0 to col -1.

# So we need to check only left side for

# attacking queens

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

# Consider this column and try placing

# this queen in all rows one by one

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

# If placing queen in board[i][col

# doesn't lead to a solution, then

# queen from board[i][col]

board[i][col] = 0

# if the queen can not be placed in any row in

# this column col then return false

return False

# This function solves the N Queen problem using

# Backtracking. It mainly uses solveNQUtil() to

# solve the problem. It returns false if queens

# cannot be placed, otherwise return true and

# placement of queens in the form of 1s.

# note that there may be more than one

# solutions, this function prints one of the

# feasible solutions.

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()

# This code is contributed by Divyanshu Mehta