def count\_island(grid, i, j):  
 if i < 0 or j < 0 or i >= len(grid) or j >= len(grid[0]) or grid[i][j] != "1":  
 return 0  
  
 # count this cell  
 grid[i][j] = "0"  
  
 # count every cell connected to this one  
 count\_island(grid, i + 1, j)  
 count\_island(grid, i - 1, j)  
 count\_island(grid, i, j + 1)  
 count\_island(grid, i, j - 1)  
 return 1  
  
  
def get\_islands(grid):  
 islands = 0  
 for i in range(len(grid)):  
 for j in range(len(grid[i])):  
 islands = islands + count\_island(grid, i, j)  
 return islands

PYTHON

def shortest\_paths(graph, source):  
 num\_nodes = graph.shape[0]  
 distances = np.full(num\_nodes, np.inf)  
 distances[source] = 0  
 parents = np.full(num\_nodes, None)  
  
 heap = [(0, source)]  
 while heap:  
 current\_distance, current\_node = heapq.heappop(heap)  
 if current\_distance > distances[current\_node]:  
 continue  
 for neighbor in range(num\_nodes):  
 weight = graph[current\_node][neighbor]   
 if weight <= 0: # no edge  
 continue  
  
 distance = current\_distance + weight

# not good enough  
 if distance > distances[neighbor]:  
 continue  
  
 distances[neighbor] = distance  
 parents[neighbor] = current\_node  
 heapq.heappush(heap, (distance, neighbor))  
  
 return distances, parents  
  
  
def second\_shortest\_path(graph, source, terminal):  
 # shortest path from source to terminal  
 distances, parents = shortest\_paths(graph, source)  
 shortest\_path = []  
 current\_node = terminal  
 while current\_node is not None:  
 shortest\_path.append(current\_node)  
 current\_node = parents[current\_node]  
 shortest\_path.reverse()  
  
 path\_length = np.inf  
 path = None  
  
 for i in range(len(shortest\_path) - 1):  
 # remove this edge  
 node1 = shortest\_path[i]  
 node2 = shortest\_path[i+1]  
 temp\_weight = graph[node1][node2]  
 graph[node1][node2] = 0  
 graph[node2][node1] = 0  
  
 # shortest path without using this edge  
 distances, parents = shortest\_paths(graph, source)  
 if distances[terminal] < path\_length:  
 path\_length = distances[terminal]  
 path = []  
 current\_node = terminal  
 while current\_node is not None:  
 path.append(current\_node)  
 current\_node = parents[current\_node]  
 path.reverse()

# add this edge back  
 graph[node1][node2] = temp\_weight  
 graph[node2][node1] = temp\_weight  
  
 return path\_length, path

PYTHON

def get\_min\_vertex(costs, visited):  
 minimum\_cost = float('inf')  
 index = -1  
 for i in range(len(visited)):  
 if costs[i] < minimum\_cost and not visited[i]:  
 minimum\_cost = costs[i]  
 index = i  
 return index  
  
  
def get\_mst\_weight(parents, graph):  
 weight = 0  
 for i in range(1, len(parents)):  
 weight = weight + graph[parents[i]][i]  
 return weight  
  
  
def mst(graph):  
 vertex\_count = len(graph)  
 costs = [float('inf')] \* vertex\_count  
 costs[0] = 0  
 parents = [None] \* vertex\_count  
 visited = [False] \* vertex\_count  
  
 for \_ in range(vertex\_count):  
 min\_vertex = get\_min\_vertex(costs, visited)  
 if min\_vertex == -1:  
 break  
 visited[min\_vertex] = True  
  
 for v in range(vertex\_count):  
 if 0 < graph[min\_vertex][v] < costs[v] and not visited[v]:  
 costs[v] = graph[min\_vertex][v]  
 parents[v] = min\_vertex  
 return parents

def second\_best\_mst(graph):  
 vertex\_count = len(graph)  
 parents = mst(graph)  
 second\_best\_weight = np.inf  
 second\_best\_parents = [None] \* vertex\_count  
  
 for i in range(1, len(parents)):  
 parent = parents[i]  
 temp\_weight = graph[parent][i]  
 graph[parent][i] = 0  
 graph[i][parent] = 0  
  
 parents = mst(graph)  
 weight = get\_mst\_weight(parents, graph)  
 if weight < second\_best\_weight:  
 second\_best\_parents = parents  
 second\_best\_weight = weight  
  
 graph[parent][i] = temp\_weight  
 graph[i][parent] = temp\_weight  
  
 return second\_best\_parents

PYTHON

def cycle\_check(courses, prerequisites):  
 graph = [[False] \* courses for \_ in range(courses)]  
 for course, prerequisite in prerequisites:  
 graph[prerequisite][course] = True  
  
 visited = [False] \* courses  
 stack = [False] \* courses  
  
 def dfs(vertex):  
 if visited[vertex]:  
 return False  
  
 if stack[vertex]:  
 return True  
  
 stack[vertex] = True

for neighbor in range(courses):  
 if graph[vertex][neighbor] and dfs(neighbor):  
 return True  
  
 visited[vertex] = True  
 return False  
  
 for node in range(courses):  
 stack = [False] \* courses  
 if dfs(node):  
 return False  
  
 return True

PYTHON