**Punjab University Lahore , Algo architects 1**

**Table of Content**

**Boilerplate Code**

**Two Pointer Approach**

**Gcd and itertools and Collection and numpy BFS anf DFS**

**Eulerian Circuit / Path Bridges Articulation/Cut Point**

**Strongly Connected Components (Kosaraju's Algorithm) Bellman-Ford Algorithm**

**Dijkstra's Algorithm Shortest Path Backtracking Solvable**

**Boilerplate Code**

import sys

import math import bisect as bs

import string as strn import heapq as hq import collections as clc import itertools as it import operator as op import copy as cp import queue as q to\_debug = True

def solve():

...

def main():

t = int(input()) for \_ in range(t):

solve()

def input():

return sys.stdin.readline().strip('\r\n') def inp\_int():

return int(input())

def inp\_map(f=None):

return map(f, input().split()) if f else map(int, input().split()) 3

def inp\_list(f=None):

return list(map(f, input().split())) if f else list(input()) def print(x='', end='\n'):

sys.stdout.write(str(x)) sys.stdout.write(end)

def debug(\*x, end='\n', sep=' '):

if not to\_debug:

return

for \_x in x: sys.stderr.write(str(\_x)) sys.stderr.write(str(sep)) sys.stderr.write(end) main()

# Two Pointer Approach

n = int(input()) t = int(input())

books = [int(input()) for \_ in range(n)] right = 0

left = 0

cur = 0

ans = 0

while left < n and right < n:

# Finding the maximum right for which cur is less than t. while right < n:

cur += books[right]

right += 1

# Subtracting the exceeded book from cur. if cur > t:

right -= 1

cur -= books[right] break

4

ans = max(ans, right - left) cur -= books[left]

left += 1 print(ans)

import math ..... print(math.**gcd**(20, 30)) # Greatest Common Divisor print(math.factorial(5))

#Factorial of 5..... print(math.comb(5, 2)) # Combinations of 5 taken 2 at a time ..... print(math.isqrt(25)) # Integer square root of 25 #end import itertools

arr = [1, 2, 3] ; print(list(**itertools**.permutations(arr))) # Generate permutations ; print(list(itertools.combinations(arr, 2))) # Generate combinations of 2 ; print(list(itertools.product(arr, repeat=2))) # Cartesian product #end

# import collections

counter = collections.Counter([1, 2, 2, 3, 3, 3]).... print(counter) # Output: Counter({3: 3, 2: 2, 1: 1}) # Deque (double-ended queue). dq = collections.deque([1, 2, 3]); dq.appendleft(0);

dq.append(4); print(dq) # deque([0, 1, 2, 3, 4]) # defaultdict ; dd = collections.defaultdict(int; dd["key"] += 1; print(dd["key"]) # Output: 1 #end

import heapq ; heap = [];heapq.heappush(heap, 10);heapq.heappush(heap, 20); heapq.heappush(heap, 5);print(heapq.heappop(heap)) # Output: 5 (smallest element ; print(heapq.nlargest(2, heap)) # Output: [20, 10] #end

import bisect ; arr = [1, 3, 4, 10, 12] ;print(bisect.bisect\_left(arr, 5)) # Output: 3 (Position where

5 should go);print(bisect.bisect\_right(arr, 10)) # Output: 4 (Position for 10); #end

import sys ; input = sys.stdin.read ; data = input().splitlines(); print(data) #end

# import numpy as np

from collections import deque ; dq = deque([1, 2, 3]) ; dq.appendleft(0); dq.append(4); dq.pop(); dq.popleft(); print(dq) # Output: deque([1, 2, 3]) #end

from collections import defaultdict frequency = defaultdict(int)

for char in "lubaba": frequency[char] += 1

print(frequency.values())

for value in frequency.values(): print(value)

l=list(frequency.values()) print(l[0:2]).............

def dfs(graph, node, visited):

visited.add(node) # Mark the node as visited print(node, end=" ") # Process the node

for neighbor in graph[node]: if neighbor not in visited:

dfs(graph, neighbor, visited)

graph = {1: [2, 3],2: [1, 4, 5],3: [1],4: [2],5: [2]}............. visited = set() dfs(graph, 1, visited)

# BFS anf DFS

from collections import deque def bfs(graph, start):

visited = set() # Set to keep track of visited nodes

queue = deque([start]) # Queue for BFS visited.add(start)

while queue:

node = queue.popleft() # Get the node from the front of the queue print(node, end=" ") # Process the node

for neighbor in graph[node]: if neighbor not in visited:

visited.add(neighbor) queue.append(neighbor)

bfs(graph, 1) #end

my\_dict = {'a': 1, 'b': 2, 'c': 3}

for key, value in my\_dict.items(): print(f"Key: {key}, Value: {value}") #end

# Eulerian Circuit / Path

Eulerian Path and Eulerian Circuit are algorithms to find a path that visits every edge of a graph exactly once. An Eulerian Circuit starts and ends at the same vertex, while an Eulerian Path may start and end at different vertices.

python Copy

# Check if the graph has an Eulerian Circuit or Path

def is\_eulerian(graph): odd\_degree\_vertices = 0 for node in graph:

if len(graph[node]) % 2 != 0:

odd\_degree\_vertices += 1

if odd\_degree\_vertices == 0: return "Eulerian Circuit"

elif odd\_degree\_vertices == 2: return "Eulerian Path"

else:

return "No Eulerian Path or Circuit" # Example graph

graph = { 0: [1, 2],

1: [0, 2, 3],

2: [0, 1, 3],

3: [1, 2]

}

print(is\_eulerian(graph)) # Eulerian Circuit

# Bridges

A bridge in a graph is an edge that, if removed, increases the number of connected components.

python Copy

# Find bridges in a graph using DFS

def dfs(graph, node, visited, parent, low, disc, bridges): visited[node] = True

disc[node] = low[node] = dfs.time dfs.time += 1

for neighbor in graph[node]: if not visited[neighbor]:

dfs(graph, neighbor, visited, node, low, disc, bridges) low[node] = min(low[node], low[neighbor])

if low[neighbor] > disc[node]: bridges.append((node, neighbor))

elif neighbor != parent:

low[node] = min(low[node], disc[neighbor])

def find\_bridges(graph): visited = [False] \* len(graph) disc = [-1] \* len(graph)

low = [-1] \* len(graph) bridges = []

dfs.time = 0

for node in range(len(graph)): if not visited[node]:

dfs(graph, node, visited, -1, low, disc, bridges) return bridges

graph = { 0: [1, 2],

1: [0, 2],

2: [0, 1, 3],

3: [2]

}

print(find\_bridges(graph)) # [(2, 3)]

# Articulation/Cut Point

An articulation point (or cut vertex) is a vertex which, when removed, increases the number of connected components.

python Copy

# Find articulation points using DFS

def dfs(graph, node, visited, parent, low, disc, ap, time): visited[node] = True

disc[node] = low[node] = time time += 1

children = 0

for neighbor in graph[node]: if not visited[neighbor]:

children += 1

dfs(graph, neighbor, visited, node, low, disc, ap, time) low[node] = min(low[node], low[neighbor])

if parent[node] == -1 and children > 1: ap[node] = True

if parent[node] != -1 and low[neighbor] >= disc[node]: ap[node] = True

elif neighbor != parent[node]:

low[node] = min(low[node], disc[neighbor])

def find\_articulation\_points(graph): visited = [False] \* len(graph)

disc = [-1] \* len(graph) low = [-1] \* len(graph) ap = [False] \* len(graph)

parent = [-1] \* len(graph) time = 0

for node in range(len(graph)): if not visited[node]:

dfs(graph, node, visited, parent, low, disc, ap, time) return [i for i, x in enumerate(ap) if x]

graph = { 0: [1, 2],

1: [0, 2],

2: [0, 1, 3],

3: [2]

}

print(find\_articulation\_points(graph)) # [2]

# Strongly Connected Components (Kosaraju's Algorithm)

A strongly connected component (SCC) is a maximal subgraph where each vertex is reachable from every other vertex in the component.

python Copy

from collections import defaultdict

# Kosaraju's Algorithm for Strongly Connected Components def dfs(graph, node, visited, stack):

visited[node] = True

for neighbor in graph[node]: if not visited[neighbor]:

dfs(graph, neighbor, visited, stack) stack.append(node)

def transpose(graph): transposed = defaultdict(list) for node in graph:

for neighbor in graph[node]: transposed[neighbor].append(node)

return transposed def kosaraju(graph):

stack = []

visited = [False] \* len(graph)

# Step 1: Fill stack with the finishing times of nodes for node in range(len(graph)):

if not visited[node]:

dfs(graph, node, visited, stack) # Step 2: Transpose the graph

transposed\_graph = transpose(graph) # Step 3: DFS on the transposed graph

visited = [False] \* len(graph) scc = []

while stack:

node = stack.pop() if not visited[node]:

component = []

dfs(transposed\_graph, node, visited, component) scc.append(component)

return scc graph = {

0: [1],

1: [2],

2: [0, 3],

3: [3]

}

print(kosaraju(graph)) # [[3], [2], [1], [0]]

* 1. Minimum Spanning Tree (Kruskal’s Algorithm)

Kruskal's algorithm finds the Minimum Spanning Tree (MST) of a graph by using a union-find data structure.

# Kruskal's Algorithm for Minimum Spanning Tree (MST) class UnionFind:

def init (self, n): self.parent = list(range(n)) self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x]) return self.parent[x]

def union(self, x, y): rootX = self.find(x) rootY = self.find(y) if rootX != rootY:

if self.rank[rootX] > self.rank[rootY]: self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]: self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX self.rank[rootX] += 1

def kruskal(n, edges): uf = UnionFind(n) mst = []

edges.sort(key=lambda x: x[2]) # Sort edges by weight

for u, v, w in edges:

if uf.find(u) != uf.find(v): uf.union(u, v) mst.append((u, v, w))

return mst

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

print(kruskal(4, edges)) # [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

# Bellman-Ford Algorithm

Bellman-Ford is used to find the shortest path from a single source to all other vertices in a weighted graph, and it can handle negative weights.

# Bellman-Ford Algorithm

def bellman\_ford(graph, V, start): distance = [float('inf')] \* V distance[start] = 0

# Relax all edges |V| - 1 times for \_ in range(V - 1):

for u, v, weight in graph:

if distance[u] != float('inf') and distance[u] + weight < distance[v]: distance[v] = distance[u] + weight

# Check for negative weight cycles for u, v, weight in graph:

if distance[u] != float('inf') and distance[u] + weight < distance[v]: print("Graph contains negative weight cycle")

return None return distance

# Example graph

graph = [(0, 1, -1), (0, 2, 4), (1, 2, 3), (1, 3, 2), (1, 4, 2), (3, 2, 5), (3, 1, 1), (4, 3, -3)]

print(bellman\_ford(graph, 5, 0)) # Shortest distances from source vertex

# Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest path from a single source to all other vertices in a weighted graph, but it only works with non-negative weights.

python Copy

import heapq

# Dijkstra's Algorithm def dijkstra(graph, start):

pq = [(0, start)] distances = {start: 0} while pq:

(dist, node) = heapq.heappop(pq)

if dist > distances.get(node, float('inf')): continue

for neighbor, weight in graph[node]: distance = dist + weight

if distance < distances.get(neighbor, float('inf')): distances[neighbor] = distance heapq.heappush(pq, (distance, neighbor))

return distances graph = {

0: [(1, 1), (2, 4)],

1: [(2, 2), (3, 5)],

2: [(3, 1)],

3: []

}

print(dijkstra(graph, 0)) # Shortest paths from source vertex #end Find Path in a Maze (DFS)

# Depth-First Search (DFS) to find a path in a maze def is\_path\_exists(maze, start, end):

rows, cols = len(maze), len(maze[0] # Directions: up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)

# Helper function to perform DFS def dfs(x, y):

if (x, y) == end: # Found the path return True

if not (0 <= x < rows and 0 <= y < cols): # Out of bounds return False

if maze[x][y] == 1: # Wall, can't go there return False

maze[x][y] = 1 # Mark as visited # Explore all four directions

for dx, dy in directions: nx, ny = x + dx, y + dy if dfs(nx, ny):

return True

return False

return dfs(start[0], start[1] # Example

maze = [

[0, 1, 0, 0],

[0, 1, 0, 1],

[0, 1, 0, 0],

[0, 0, 0, 0]

]

start = (0, 0)

end = (3, 3)

print(is\_path\_exists(maze, start, end)) # Output: True #end

# Find Shortest Path in a Maze (BFS)

from collections import deque

# BFS to find the shortest path in a maze def shortest\_path(maze, start, end):

rows, cols = len(maze), len(maze[0]) # Directions: up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

queue = deque([(start[0], start[1], 0)]) # (x, y, distance) visited = set()

visited.add(start) while queue:

x, y, dist = queue.popleft()

# If we reach the end, return the distance if (x, y) == end:

return dist

# Explore all four directions for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < rows and 0 <= ny < cols and maze[nx][ny] == 0 and (nx, ny) not in visited: visited.add((nx, ny))

queue.append((nx, ny, dist + 1)) return -1 # No path found

# Example maze = [

[0, 1, 0, 0],

[0, 1, 0, 1],

[0, 0, 0, 0],

[0, 1, 1, 0]

]

start = (0, 0)

end = (3, 3)

print(shortest\_path(maze, start, end)) # Output: 5 (path length) #end

# # Backtracking to find a path in a maze

def solve\_maze(maze, start, end): rows, cols = len(maze), len(maze[0]) # Directions: up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Helper function to perform backtracking def backtrack(x, y):

if (x, y) == end: # Found the path return [(x, y)]

if not (0 <= x < rows and 0 <= y < cols): # Out of bounds return []

if maze[x][y] == 1: # Wall, can't go there return []

maze[x][y] = 1 # Mark as visited # Explore all four directions

for dx, dy in directions: nx, ny = x + dx, y + dy path = backtrack(nx, ny)

if path: # If path is found, return the path return [(x, y)] + path

return [] # No path found, backtrack return backtrack(start[0], start[1]

# Example maze = [

[0, 1, 0, 0],

[0, 1, 0, 1],

[0, 0, 0, 0],

[0, 1, 1, 0]

]

start = (0, 0)

end = (3, 3)

path = solve\_maze(maze, start, end)

print(path) # Output: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3)] #end

# Problem: Find all possible paths from the top-left to the bottom-right of the maze.

Solution: Use DFS to explore all possible paths recursively. # Find all paths in a maze using DFS

def find\_all\_paths(maze, start, end): rows, cols = len(maze), len(maze[0]) all\_paths = []

# Directions: up, down, left, right directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def dfs(x, y, path): if (x, y) == end:

all\_paths.append(path) return

if not (0 <= x < rows and 0 <= y < cols): # Out of bounds return

if maze[x][y] == 1: # Wall, can't go there return

maze[x][y] = 1 # Mark as visited

# Explore all four directions for dx, dy in directions:

dfs(x + dx, y + dy, path + [(x + dx, y + dy)]) maze[x][y] = 0 # Unmark as visited

dfs(start[0], start[1], [start]) return all\_path

# Example maze = [

[0, 1, 0, 0],

[0, 0, 0, 1],

[0, 1, 0, 0],

[0, 0, 0, 0]

]

start = (0, 0)

end = (3, 3)

paths = find\_all\_paths(maze, start, end) for path in paths:

print(path) #END