Long Coding Contest – April 2025 (Amrita School of Computing, Bengaluru)

# Q1: Pascal’s Triangle

def generate\_pascals\_triangle(N):  
 triangle = []  
 for i in range(N):  
 row = [1] \* (i + 1)  
 for j in range(1, i):  
 row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j]  
 triangle.append(row)  
 return triangle  
  
N = int(input("Enter the number of rows for Pascal's Triangle: "))  
result = generate\_pascals\_triangle(N)  
print("Pascal's Triangle:")  
for row in result:  
 print(row)

# Q2: Longest Increasing Subsequence (LIS)

import bisect  
  
def longest\_increasing\_subsequence(nums):  
 sub = []  
 for x in nums:  
 i = bisect.bisect\_left(sub, x)  
 if i == len(sub):  
 sub.append(x)  
 else:  
 sub[i] = x  
 return len(sub)  
  
arr = list(map(int, input("Enter the array elements: ").split()))  
print("Length of LIS:", longest\_increasing\_subsequence(arr))

# Q3: Minimum Number of Perfect Squares to Sum to N

import math  
  
def min\_perfect\_squares(n):  
 dp = [0] + [float('inf')] \* n  
 for i in range(1, n + 1):  
 j = 1  
 while j\*j <= i:  
 dp[i] = min(dp[i], dp[i - j\*j] + 1)  
 j += 1  
 return dp[n]  
  
N = int(input("Enter a number: "))  
print("Minimum number of perfect squares:", min\_perfect\_squares(N))

# Q4: Unique Paths in Grid with Obstacles

def unique\_paths\_with\_obstacles(grid):  
 m, n = len(grid), len(grid[0])  
 if grid[0][0] == 1:  
 return 0  
  
 dp = [[0]\*n for \_ in range(m)]  
 dp[0][0] = 1  
  
 for i in range(m):  
 for j in range(n):  
 if grid[i][j] == 1:  
 dp[i][j] = 0  
 else:  
 if i > 0: dp[i][j] += dp[i-1][j]  
 if j > 0: dp[i][j] += dp[i][j-1]  
 return dp[m-1][n-1]  
  
m = int(input("Enter number of rows: "))  
n = int(input("Enter number of columns: "))  
grid = []  
print("Enter grid row-wise (0 for open, 1 for obstacle):")  
for \_ in range(m):  
 grid.append(list(map(int, input().split())))  
print("Number of unique paths:", unique\_paths\_with\_obstacles(grid))

# Q5: Knight's Tour

N = int(input("Enter the size of the chessboard (N): "))  
  
moves = [(-2, -1), (-1, -2), (1, -2), (2, -1),  
 (2, 1), (1, 2), (-1, 2), (-2, 1)]  
  
def is\_valid(x, y, board):  
 return 0 <= x < N and 0 <= y < N and board[x][y] == -1  
  
def solve(x, y, movei, board):  
 if movei == N\*N:  
 return True  
 for dx, dy in moves:  
 nx, ny = x + dx, y + dy  
 if is\_valid(nx, ny, board):  
 board[nx][ny] = movei  
 if solve(nx, ny, movei + 1, board):  
 return True  
 board[nx][ny] = -1  
 return False  
  
board = [[-1 for \_ in range(N)] for \_ in range(N)]  
board[0][0] = 0  
  
if solve(0, 0, 1, board):  
 for row in board:  
 print(row)  
else:  
 print("No solution exists.")

# Q6: Hamiltonian Cycle

def is\_valid(v, pos, path, graph):  
 if graph[path[pos - 1]][v] == 0:  
 return False  
 if v in path:  
 return False  
 return True  
  
def hamiltonian\_util(graph, path, pos):  
 if pos == len(graph):  
 return graph[path[pos - 1]][path[0]] == 1  
  
 for v in range(1, len(graph)):  
 if is\_valid(v, pos, path, graph):  
 path[pos] = v  
 if hamiltonian\_util(graph, path, pos + 1):  
 return True  
 path[pos] = -1  
 return False  
  
def find\_hamiltonian\_cycle(graph):  
 path = [-1] \* len(graph)  
 path[0] = 0  
 if not hamiltonian\_util(graph, path, 1):  
 print("No Hamiltonian Cycle exists")  
 else:  
 path.append(path[0])  
 print("Hamiltonian Cycle:", path)  
  
n = int(input("Enter number of vertices: "))  
print("Enter adjacency matrix:")  
graph = [list(map(int, input().split())) for \_ in range(n)]  
find\_hamiltonian\_cycle(graph)

# Q7: Minimum Jumps to Reach End

def min\_jumps(arr):  
 n = len(arr)  
 if n <= 1:  
 return 0  
 if arr[0] == 0:  
 return -1  
  
 maxReach = arr[0]  
 step = arr[0]  
 jump = 1  
  
 for i in range(1, n):  
 if i == n - 1:  
 return jump  
 maxReach = max(maxReach, i + arr[i])  
 step -= 1  
 if step == 0:  
 jump += 1  
 if i >= maxReach:  
 return -1  
 step = maxReach - i  
 return -1  
  
arr = list(map(int, input("Enter array elements: ").split()))  
print("Minimum jumps to reach end:", min\_jumps(arr))

# Q8: Graph Coloring

def is\_safe(node, graph, color, c):  
 for i in range(len(graph)):  
 if graph[node][i] == 1 and color[i] == c:  
 return False  
 return True  
  
def graph\_coloring(graph, m, color, node):  
 if node == len(graph):  
 return True  
 for c in range(1, m + 1):  
 if is\_safe(node, graph, color, c):  
 color[node] = c  
 if graph\_coloring(graph, m, color, node + 1):  
 return True  
 color[node] = 0  
 return False  
  
V = int(input("Enter number of vertices: "))  
graph = []  
print("Enter adjacency matrix:")  
for \_ in range(V):  
 graph.append(list(map(int, input().split())))  
m = int(input("Enter number of colors: "))  
color = [0] \* V  
  
if graph\_coloring(graph, m, color, 0):  
 print("Colors assigned to vertices:", color)  
else:  
 print(0)

# Q9: Job Sequencing

class Job:  
 def \_\_init\_\_(self, id, deadline, profit):  
 self.id = id  
 self.deadline = deadline  
 self.profit = profit  
  
def job\_sequencing(jobs):  
 jobs.sort(key=lambda x: x.profit, reverse=True)  
 n = len(jobs)  
 result = [-1] \* n  
 slot = [False] \* n  
 total\_profit = 0  
  
 for job in jobs:  
 for j in range(min(n - 1, job.deadline - 1), -1, -1):  
 if not slot[j]:  
 result[j] = job.id  
 slot[j] = True  
 total\_profit += job.profit  
 break  
  
 print("Scheduled jobs:", [j for j in result if j != -1])  
 print("Maximum profit:", total\_profit)  
  
n = int(input("Enter number of jobs: "))  
jobs = []  
for i in range(n):  
 p, d = map(int, input(f"Enter profit and deadline for job {i+1}: ").split())  
 jobs.append(Job(i + 1, d, p))  
  
job\_sequencing(jobs)

# Q10: Maximize Sum of A with K Swaps

def maximize\_sum\_after\_swaps(A, B, K):  
 A.sort()  
 B.sort(reverse=True)  
 for i in range(K):  
 if A[i] < B[i]:  
 A[i], B[i] = B[i], A[i]  
 else:  
 break  
 return sum(A)  
  
A = list(map(int, input("Enter array A: ").split()))  
B = list(map(int, input("Enter array B: ").split()))  
K = int(input("Enter maximum number of swaps (K): "))  
  
result = maximize\_sum\_after\_swaps(A, B, K)  
print("Maximum sum of A after swaps:", result)