```
1) Divide and conquer
def square_large_number(n):
    if n < 10:
        return n * n  # Base case for single-digit numbers
    num_str = str(n)
    mid = len(num_str) // 2
    high = int(num_str[:mid])
    low = int(num_str[mid:])
    base = 10 ** (len(num_str) - mid)
    # Direct application of the squaring formula
    high_square = square_large_number(high)
    low_square = square_large_number(low)
    cross_term = 2 * high * low
    return high_square * base**2 + cross_term * base + low_square

# Example Usage
# num = 12345678901234567890
# print(square_large_number_simple(num))

try:
    num = int(input("Enter a large number to square: "))
    result = square_large_number(num)
    print(f"The square of {num} is: {result}")
except ValueError:
    print("Invalid input! Please enter a valid integer.")


2) Job scheduling
    def __init__(self, job_id, deadline, profit):
        self.job_id = job_id
        self.deadline = deadline
        self.profit = profit

def job_scheduling(jobs, n):
    jobs.sort(key=lambda x: x.profit, reverse=True)
    result, total_profit = [-1] * n, 0
    for job in jobs:
        for slot in range(min(n - 1, job.deadline - 1), -1, -1):
            if result[slot] == -1:
                result[slot] = job.job_id
                total_profit += job.profit
                break
    return [job_id for job_id in result if job_id != -1], total_profit

if __name__ == "__main__":
    n = int(input("Enter the number of jobs: "))
    jobs = []
    for _ in range(n):
        job_id = input(f"\nEnter job ID: ")
        deadline = int(input(f"Enter deadline: "))
        profit = int(input(f"Enter profit: "))
        jobs.append(Job(job_id, deadline, profit))

    scheduled_jobs, total_profit = job_scheduling(jobs, n)
    print(f"/nScheduled Jobs: {scheduled_jobs}")
    print(f"Total Profit: {total_profit}")

3) Flyod warshall
INF = float('inf')

def floyd_warshall(dist, n):
    for k in range(n):
        for i in range(n):
            for j in range(n):
```

```
                dist[i][j] = min(dist[i][j],(dist[i][k]+dist[k][j]))

    print("Result matrix: ")
    for i in range(n):
        for j in range(n):
            if dist[i][j]=="INF":
                print("INf", end= "\t")
            else:
                print(dist[i][j], end="\t")
        print()

def main():
    n = int(input("Enter no of ciites: "))
    dist= []
    print("Enter the matrix in A0 form also if no direct connect enter inf")
    for i in range(n):
        row = input().split()
        row = [INF if x =='inf' else int(x) for x in row]
        dist.append(row)

    floyd_warshall(dist,n)

if __name__ == "__main__":
    main()

4) Dijikstra
import heapq

class DAA4:
    def __init__(self):
        self.graph = []

    def create_graph(self, n):
        self.graph = [[0] * n for _ in range(n)]
        for i in range(n):
            for j in range(i, n):
                yn = int(input(f"Is there an edge between {i + 1} and {j + 1}?
(1/0): "))

                while yn != 0 and yn != 1:
                    yn = int(input("Enter either 1 or 0: "))

                if yn == 1:
                    wt = int(input("Enter weight of the edge: "))
                    self.graph[i][j] = self.graph[j][i] = wt
                else:                    self.graph[i][j] = self.graph[j][i] = 0

    def print_graph(self):
        for row in self.graph:
            print(" ".join(map(str, row)))

    def dijkstra(self, V, adj, S):
        distance = [float('inf')] * V
        distance[S] = 0
        visited = [False] * V

        pq = [(0, S)]  # Priority queue with (distance, vertex)

        while pq:
            dist_u, u = heapq.heappop(pq)

            if visited[u]:
                continue
            visited[u] = True
```

```
            for v in range(V):
                weight = adj[u][v]
                if weight != 0 and not visited[v] and dist_u + weight <
distance[v]:
                    distance[v] = dist_u + weight
                    heapq.heappush(pq, (distance[v], v))

        return distance

def main():
    obj = DAA4()
    n = int(input("Enter number of nodes: "))
    obj.create_graph(n)
    obj.print_graph()

    source = int(input(f"Enter the source vertex (1 to {n}): ")) - 1
    distances = obj.dijkstra(n, obj.graph, source)

    print(f"Shortest distances from source vertex {source + 1} to all other
vertices:")
    for i in range(n):
        print(f"Vertex {i + 1}: {distances[i]}")

if __name__ == "__main__":
    main()

5) Knight tour
N = 8

# Utility function to check if the move is valid
def is_safe(x, y, board):
    return (x >= 0 and x < N and y >= 0 and y < N and board[x][y] == -1)

# Backtracking function to solve the Knight's Tour problem
def solve_knights_tour_util(x, y, move_i, board, x_move, y_move):
    if move_i == N * N:
        return True  # All squares visited, solution found

    # Try all possible moves for the knight
    for i in range(8):
        next_x = x + x_move[i]
        next_y = y + y_move[i]
        if is_safe(next_x, next_y, board):
            board[next_x][next_y] = move_i
            if solve_knights_tour_util(next_x, next_y, move_i + 1, board,
x_move, y_move):
                return True
            # Backtrack: undo the move
            board[next_x][next_y] = -1

    return False  # If no move leads to a solution, backtrack

# Main function to solve the Knight's Tour problem using backtracking
def solve_knights_tour(start_x, start_y):
    # Initialize the solution board
    board = [[-1 for _ in range(N)] for _ in range(N)]

    # Moves of a knight (xMove, yMove represent possible moves in the x and y
directions)
    x_move = [2, 1, -1, -2, -2, -1, 1, 2]
    y_move = [1, 2, 2, 1, -1, -2, -2, -1]

    # Start the knight at the given starting point
```

```
    board[start_x][start_y] = 0

    # Use backtracking to find the solution
    if not solve_knights_tour_util(start_x, start_y, 1, board, x_move, y_move):
        print("Solution does not exist")
    else:
        print_solution(board)

# Utility function to print the solution board
def print_solution(board):
    for row in board:
        for val in row:
            print(f'{val:2}', end=' ')
        print()

# Driver Code
if __name__ == "__main__":
    start_x, start_y = map(int, input("Enter starting x and y positions (0-based
index): ").split())
    solve_knights_tour(start_x, start_y)
```

6) B&B class Job:

```
import math
from queue import PriorityQueue

class Node:
    def __init__(self, cost, level, assigned):
        self.cost = cost
        self.level = level
        self.assigned = assigned

    def __lt__(self, other):
        return self.cost < other.cost

def calculate_min_cost(cost_matrix):
    n = len(cost_matrix)
    pq = PriorityQueue()
    pq.put(Node(0, 0, [-1] * n))
    min_cost, min_assignment = math.inf, []

    while not pq.empty():
        node = pq.get()
        if node.level == n:
            if node.cost < min_cost:
                min_cost, min_assignment = node.cost, node.assigned
            continue

        for j in range(n):
            if j not in node.assigned:
                new_cost = node.cost + cost_matrix[node.level][j]
                if new_cost < min_cost:
                    new_assigned = node.assigned[:]
                    new_assigned[node.level] = j
                    pq.put(Node(new_cost, node.level + 1, new_assigned))

    return min_cost, min_assignment

def main():
    n = int(input("Enter the number of students (and clubs): "))
    print("Enter the cost matrix (row-wise):")
    cost_matrix = [list(map(int, input().split())) for _ in range(n)]
    min_cost, assignment = calculate_min_cost(cost_matrix)
    print(f"Minimum cost of assignment: {min_cost}")
```

```
    print(f"Optimal assignment: {assignment}")

if __name__ == "__main__":
    main()
```