S CodeVita 2025: 20 Problem Statements, Solutions, and Explan

Question 1: Election Outcome

Problem Statement:

There are voters in a queue where some are supporters of Candidate A, some of Candidate B, and some neutral. The goal is to determine the final result of the election based on how neutral voters are influenced.

Input:

Length of queue n.

Characters representing voters (A, B, -).

Output:

Print the winner: 'A', 'B', or 'Coalition government'.

Constraints:

1 <= n <= 1000

Approach:

Traverse the queue from both sides. Supporters of A influence voters on the left, and B influences voters on the right. Simulate the movement and determine the final vote counts.

Solution (Python):

```
def election_result(voter_queue):
    a_supporters = b_supporters = 0
    for voter in voter_queue:
        if voter == 'A':
            a_supporters += 1
```

```
        elif voter == 'B':
            b_supporters += 1
    if a_supporters > b_supporters:
        return 'A'
    elif b_supporters > a_supporters:
        return 'B'
    else:
        return 'Coalition government'

n = int(input())
voter_queue = input().strip()
print(election_result(voter_queue))
```

Explanation:

We count supporters of A and B. Whichever count is higher determines the winner. If both are equal, it's a coalition.

Question 2: Minimum Platforms

Problem Statement:

Given the arrival and departure times of trains, find the minimum number of platforms required at the station.

Input:

Number of trains N.

Each train's arrival and departure time.

Output:

Minimum number of platforms required.

Constraints:

1 <= N <= 10^5

Approach:

Sort the arrival and departure times. Use a two-pointer technique to track overlapping trains.

Solution (Python):

```
def find_platforms(arrivals, departures):
    arrivals.sort()
    departures.sort()
    platforms_needed = 1
    result = 1
    i, j = 1, 0

    while i < len(arrivals) and j < len(departures):
        if arrivals[i] <= departures[j]:
            platforms_needed += 1
            i += 1
        else:
            platforms_needed -= 1
            j += 1
        result = max(result, platforms_needed)

    return result

n = int(input())
arrivals = list(map(int, input().split()))
```

```
departures = list(map(int, input().split()))

print(find_platforms(arrivals, departures))
```

Explanation:

By sorting the times and using two pointers, we can track how many platforms are needed at any given time.

Question 3: Subset Sum

Problem Statement:

Find if there exists a subset in an array whose sum is equal to a given number.

Input:

An integer n, the size of the array.

An integer S, the required subset sum.

A list of integers representing the array.

Output:

Print 'Yes' if a subset with sum S exists, otherwise print 'No'.

Constraints:

1 <= n <= 20

1 <= S <= 10^4

Approach:

Use dynamic programming to determine if any subset can sum up to the target value.

Solution (Python):

```
def is_subset_sum(arr, n, S):

    dp = [[False for _ in range(S + 1)] for _ in range(n + 1)]
```

```
    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for j in range(1, S + 1):
            if arr[i - 1] <= j:
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - arr[i - 1]]
            else:
                dp[i][j] = dp[i - 1][j]

    return dp[n][S]

n = int(input())
S = int(input())
arr = list(map(int, input().split()))

if is_subset_sum(arr, n, S):
    print("Yes")
else:
    print("No")
```

Explanation:

We use dynamic programming to check if any subset of the array can add up to the target sum.

Question 4: Coin Change Problem

Problem Statement:

Find the number of ways to make a change of N using unlimited supply of given denominations.

Input:

Integer N, the amount to make change.

Integer m, number of denominations.

A list of integers representing the denominations.

Output:

Print the number of ways to make change.

Constraints:

1 <= N <= 10^4

1 <= m <= 100

Approach:

Use dynamic programming to count the number of ways to make change for N using the given denominations.

Solution (Python):

```
def coin_change(denominations, m, N):
    dp = [0 for _ in range(N + 1)]
    dp[0] = 1

    for i in range(m):
        for j in range(denominations[i], N + 1):
            dp[j] += dp[j - denominations[i]]

    return dp[N]

N = int(input())
```

```
m = int(input())

denominations = list(map(int, input().split()))

print(coin_change(denominations, m, N))
```

Explanation:

This dynamic programming solution builds the number of ways to make change for a given amount.

Question 5: Longest Increasing Subsequence

Problem Statement:

Given an array, find the length of the longest increasing subsequence.

Input:

An integer n, the size of the array.

A list of integers representing the array.

Output:

Print the length of the longest increasing subsequence.

Constraints:

1 <= n <= 10^5

Approach:

Use dynamic programming to find the longest increasing subsequence.

Solution (Python):

```
def length_of_lis(arr):

    if not arr:

        return 0

    dp = [1] * len(arr)
```

```
    for i in range(1, len(arr)):
        for j in range(i):
            if arr[i] > arr[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)


n = int(input())
arr = list(map(int, input().split()))
print(length_of_lis(arr))
```

Explanation:

We use dynamic programming to find the longest increasing subsequence by maintaining a `dp` array, where each element represents the length of the longest subsequence ending at that element.

Question 6: Job Scheduling Problem

Problem Statement:

You are given a set of jobs where each job has a deadline and a profit associated with it. The goal is to schedule the jobs in such a way as to maximize profit while ensuring that no two jobs overlap.

Input:

An integer n, the number of jobs.

For each job: two integers representing its deadline and profit.

Output:

Print the maximum profit.

Constraints:

1 <= n <= 1000

Approach:

Sort jobs by profit in decreasing order and use a greedy algorithm to select jobs.

Solution (Python):

```
def job_scheduling(jobs, n):
    jobs.sort(key=lambda x: x[1], reverse=True)
    max_deadline = max(job[0] for job in jobs)
    slots = [-1] * max_deadline
    max_profit = 0

    for job in jobs:
        for i in range(job[0] - 1, -1, -1):
            if slots[i] == -1:
                slots[i] = job[1]
                max_profit += job[1]
                break
    return max_profit

n = int(input())
jobs = [tuple(map(int, input().split())) for _ in range(n)]
print(job_scheduling(jobs, n))
```

Explanation:

We sort the jobs by their profit in descending order and then try to schedule them greedily to maximize profit. We use a `slots` array to track available times.

Question 7: 0/1 Knapsack Problem

Problem Statement:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Input:

An integer n, the number of items.

Two lists: weights and values.

An integer W, the maximum capacity of the knapsack.

Output:

Print the maximum total value.

Constraints:

1 <= n <= 1000

1 <= W <= 10^5

Approach:

Use dynamic programming to solve the 0/1 knapsack problem.

Solution (Python):

```
def knapsack(weights, values, W, n):
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:
                  dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i -
1][w])
```

```
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][W]


n = int(input())
weights = list(map(int, input().split()))
values = list(map(int, input().split()))
W = int(input())
print(knapsack(weights, values, W, n))
```

Explanation:

We solve the 0/1 knapsack problem using dynamic programming by maintaining a 2D dp array where dp[i][w] represents the maximum value attainable with the first i items and a knapsack of capacity w.

Question 8: Maximum Subarray Sum

Problem Statement:

Given an array, find the maximum sum of a contiguous subarray.

Input:

An integer n, the size of the array.

A list of integers representing the array.

Output:

Print the maximum sum.

Constraints:

1 <= n <= 10^5

Approach:

Use Kadane's algorithm to find the maximum sum of a contiguous subarray.

Solution (Python):

```
def max_subarray_sum(arr):
    max_so_far = arr[0]
    curr_max = arr[0]
    for i in range(1, len(arr)):
        curr_max = max(arr[i], curr_max + arr[i])
        max_so_far = max(max_so_far, curr_max)
    return max_so_far


n = int(input())
arr = list(map(int, input().split()))
print(max_subarray_sum(arr))
```

Explanation:

We use Kadane's algorithm, which efficiently finds the maximum sum of a contiguous subarray by keeping track of the current maximum sum and updating it as we traverse the array.

Question 9: Rod Cutting Problem

Problem Statement:

Given a rod of length N and a list of prices for each length of rod, determine the maximum revenue obtainable by cutting up the rod.

Input:

An integer N, the length of the rod.

A list of integers representing the prices for each length of rod.

Output:

Print the maximum revenue.

Constraints:

1 <= N <= 1000

Approach:

Use dynamic programming to maximize the revenue by cutting the rod into pieces.

Solution (Python):

```
def rod_cutting(prices, N):
    dp = [0] * (N + 1)
    for i in range(1, N + 1):
        for j in range(i):
            dp[i] = max(dp[i], prices[j] + dp[i - j - 1])
    return dp[N]


N = int(input())
prices = list(map(int, input().split()))
print(rod_cutting(prices, N))
```

Explanation:

We solve this problem using dynamic programming by maintaining a dp array where dp[i] represents the maximum revenue obtainable for a rod of length i.

Question 10: Edit Distance Problem

Problem Statement:

Given two strings, find the minimum number of operations required to convert one string into the other.

Input:

Two strings: str1 and str2.

Output:

Print the minimum number of operations.

Constraints:

1 <= len(str1), len(str2) <= 1000

Approach:

Use dynamic programming to calculate the minimum number of insertions, deletions, and substitutions required.

Solution (Python):

```
def edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i - 1] == str2[j - 1]:
```

```
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])

    return dp[m][n]

str1 = input()
str2 = input()
print(edit_distance(str1, str2))
```

Explanation:

We solve the problem using dynamic programming by maintaining a dp array where dp[i][j] represents the minimum number of operations required to convert the first i characters of str1 into the first j characters of str2.

Question 11: Matrix Chain Multiplication

Problem Statement:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The goal is to find the minimum number of scalar multiplications needed to multiply the sequence of matrices.

Input:

An integer n, the number of matrices.

A list of integers representing the dimensions of the matrices.

Output:

Print the minimum number of scalar multiplications required.

Constraints:

1 <= n <= 1000

Approach:

Use dynamic programming to solve the matrix chain multiplication problem.

Solution (Python):

```
def matrix_chain_order(p, n):
    dp = [[0 for _ in range(n)] for _ in range(n)]
    for l in range(2, n):
        for i in range(1, n - l + 1):
            j = i + l - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                q = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j]
                if q < dp[i][j]:
                    dp[i][j] = q
    return dp[1][n - 1]

n = int(input())
p = list(map(int, input().split()))
print(matrix_chain_order(p, n))
```

Explanation:

We use dynamic programming to solve this problem by maintaining a dp table where dp[i][j] stores the minimum number of scalar multiplications needed to multiply matrices from i to j.

Question 12: Floyd-Warshall Algorithm

Problem Statement:

Given a graph, find the shortest paths between every pair of vertices using the Floyd-Warshall algorithm.

Input:

An integer n, the number of vertices.

A list of n x n integers representing the adjacency matrix of the graph.

Output:

Print the shortest paths between every pair of vertices.

Constraints:

1 <= n <= 100

Approach:

Use dynamic programming to solve all pairs shortest path problem using the Floyd-Warshall algorithm.

Solution (Python):

```
def floyd_warshall(graph, n):
    dist = [[graph[i][j] for j in range(n)] for i in range(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])
    return dist


n = int(input())
graph = [list(map(int, input().split())) for _ in range(n)]
```

```
distances = floyd_warshall(graph, n)

for row in distances:

    print(' '.join(map(str, row)))
```

Explanation:

We use the Floyd-Warshall algorithm to solve the all-pairs shortest path problem by iterating over all possible paths through each vertex and updating the distance matrix accordingly.

Question 13: Travelling Salesman Problem (TSP)

Problem Statement:

Given a set of cities and distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city.

Input:

An integer n, the number of cities.

A list of n x n integers representing the distance matrix.

Output:

Print the length of the shortest tour.

Constraints:

1 <= n <= 20

Approach:

Use dynamic programming with bit masking to solve the TSP problem.

Solution (Python):

```
def tsp(graph, n):

    dp = [[None] * (1 << n) for _ in range(n)]
```

```
    def tsp_util(mask, pos):
        if mask == (1 << n) - 1:
            return graph[pos][0]
        if dp[pos][mask] is not None:
            return dp[pos][mask]
        ans = float('inf')
        for city in range(n):
            if (mask & (1 << city)) == 0:
                new_ans = graph[pos][city] + tsp_util(mask | (1 << city), city)
                ans = min(ans, new_ans)
        dp[pos][mask] = ans
        return ans
    return tsp_util(1, 0)


n = int(input())
graph = [list(map(int, input().split())) for _ in range(n)]
print(tsp(graph, n))
```

Explanation:

We solve the TSP problem using dynamic programming and bit masking, which keeps track of the visited cities using a bitmask and computes the optimal tour recursively.

Question 14: Bellman-Ford Algorithm

Problem Statement:

Given a graph, find the shortest paths from a source vertex to all other vertices using the Bellman-Ford algorithm.

Input:

An integer n, the number of vertices.

A list of edges where each edge contains three integers (u, v, w) representing an edge from vertex u to vertex v with weight w.

An integer src, the source vertex.

Output:

Print the shortest distance from the source vertex to every other vertex.

Constraints:

1 <= n <= 5000

Approach:

Use the Bellman-Ford algorithm to find shortest paths, accounting for negative weights.

Solution (Python):

```
def bellman_ford(edges, n, src):
    dist = [float('inf')] * n
    dist[src] = 0
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    return dist


n, m = map(int, input().split())
edges = [tuple(map(int, input().split())) for _ in range(m)]
src = int(input())
```

```
distances = bellman_ford(edges, n, src)

print(' '.join(map(str, distances)))
```

Explanation:

We use the Bellman-Ford algorithm to find the shortest paths from the source to all vertices. The algorithm iteratively relaxes all edges to update distances, which works even with negative edge weights.

Question 15: Dijkstra's Algorithm

Problem Statement:

Given a graph, find the shortest paths from a source vertex to all other vertices using Dijkstra's algorithm.

Input:

An integer n, the number of vertices.

A list of n x n integers representing the adjacency matrix.

An integer src, the source vertex.

Output:

Print the shortest distance from the source vertex to every other vertex.

Constraints:

1 <= n <= 1000

Approach:

Use Dijkstra's algorithm with a priority queue to find shortest paths.

Solution (Python):

```
import heapq
```

```
def dijkstra(graph, n, src):
    dist = [float('inf')] * n
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in enumerate(graph[u]):
            if w and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist


n = int(input())
graph = [list(map(int, input().split())) for _ in range(n)]
src = int(input())
distances = dijkstra(graph, n, src)
print(' '.join(map(str, distances)))
```

Explanation:

We use Dijkstra's algorithm, which efficiently finds the shortest paths from a source vertex to all other vertices using a priority queue to explore the minimum distance vertices.

Question 16: Prim's Algorithm

Problem Statement:

Given a graph, find the minimum spanning tree (MST) using Prim's algorithm.

Input:

An integer n, the number of vertices.

A list of n x n integers representing the adjacency matrix.

Output:

Print the total weight of the MST.

Constraints:

1 <= n <= 1000

Approach:

Use Prim's algorithm with a priority queue to find the MST.

Solution (Python):

```
import heapq

def prim(graph, n):

    visited = [False] * n

    pq = [(0, 0)]  # (weight, vertex)

    mst_cost = 0

    edges_used = 0

    while pq and edges_used < n:

        weight, u = heapq.heappop(pq)

        if visited[u]:

            continue

        visited[u] = True

        mst_cost += weight

        edges_used += 1

        for v, w in enumerate(graph[u]):
```

```
            if not visited[v] and w > 0:
                heapq.heappush(pq, (w, v))
    return mst_cost

n = int(input())
graph = [list(map(int, input().split())) for _ in range(n)]
print(prim(graph, n))
```

Explanation:

We use Prim's algorithm to find the minimum spanning tree by adding edges to the MST with the smallest weight at each step using a priority queue.

Question 17: Kruskal's Algorithm

Problem Statement:

Given a graph, find the minimum spanning tree (MST) using Kruskal's algorithm.

Input:

An integer n, the number of vertices.

A list of edges where each edge contains three integers (u, v, w) representing an edge from vertex u to vertex v with weight w.

Output:

Print the total weight of the MST.

Constraints:

1 <= n <= 1000

Approach:

Use Kruskal's algorithm with union-find to find the MST.

Solution (Python):

```
def find(parent, i):
    if parent[i] == i:
        return i
    return find(parent, parent[i])


def union(parent, rank, x, y):
    root_x = find(parent, x)
    root_y = find(parent, y)
    if root_x != root_y:
        if rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        elif rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        else:
            parent[root_y] = root_x
            rank[root_x] += 1


def kruskal(edges, n):
    edges.sort(key=lambda x: x[2])
    parent = [i for i in range(n)]
    rank = [0] * n
    mst_cost = 0
    for u, v, w in edges:
        root_u = find(parent, u)
        root_v = find(parent, v)
```

```
        if root_u != root_v:
            mst_cost += w
            union(parent, rank, root_u, root_v)
    return mst_cost


n, m = map(int, input().split())
edges = [tuple(map(int, input().split())) for _ in range(m)]
print(kruskal(edges, n))
```

Explanation:

We use Kruskal's algorithm to find the minimum spanning tree by sorting all edges and using union-find to ensure no cycles are created.

Question 18: 0/1 Knapsack Problem

Problem Statement:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Input:

An integer n, the number of items.

Two lists: weights and values.

An integer W, the maximum capacity of the knapsack.

Output:

Print the maximum total value.

Constraints:

1 <= n <= 1000

1 <= W <= 10^5

Approach:

Use dynamic programming to solve the 0/1 knapsack problem.

Solution (Python):

```
def knapsack(weights, values, W, n):
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:
                  dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i -
1][w])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][W]


n = int(input())
weights = list(map(int, input().split()))
values = list(map(int, input().split()))
W = int(input())
print(knapsack(weights, values, W, n))
```

Explanation:

We solve the 0/1 knapsack problem using dynamic programming by maintaining a 2D dp array where dp[i][w] represents the maximum value attainable with the first i items and a knapsack of capacity w.

Question 19: LCS (Longest Common Subsequence)

Problem Statement:

Given two strings, find the length of the longest subsequence present in both strings.

Input:

Two strings.

Output:

Print the length of the longest common subsequence.

Constraints:

1 <= len(s1), len(s2) <= 1000

Approach:

Use dynamic programming to solve the LCS problem.

Solution (Python):

```
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
```

```
s1 = input()

s2 = input()

print(lcs(s1, s2))
```

Explanation:

We solve the LCS problem using dynamic programming by maintaining a dp array where dp[i][j] represents the length of the longest common subsequence of the first i characters of s1 and the first j characters of s2.

Question 20: Largest Rectangle in Histogram

Problem Statement:

Given an array of heights representing a histogram, find the area of the largest rectangle that can be formed by the bars.

Input:

An integer n, the size of the array.

A list of integers representing the heights of the histogram bars.

Output:

Print the area of the largest rectangle.

Constraints:

1 <= n <= 10^5

Approach:

Use a stack to solve this problem efficiently in O(n) time.

Solution (Python):

```
def largest_rectangle_area(heights):
```

```
    stack = []
    max_area = 0
    heights.append(0)
    for i, h in enumerate(heights):
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)
    return max_area


n = int(input())
heights = list(map(int, input().split()))
print(largest_rectangle_area(heights))
```

Explanation:

We use a stack to solve the largest rectangle area problem in O(n) time by maintaining a list of bar indices in the stack and calculating areas when necessary.