Task-1 explanation:

Here the problem is solved using disjoint set union (DSU). It uses two main arrays 'parent' and 'size'. 'Parent' helps locating the leader or root of set and 'size' keeps track of size of each set. In the find_parent function path compression is employed which improves the efficiency of subsequent find operation. In the union function, when combining two sets, the smaller set is merged into larger one. This helps in keeping the set as flat as possible optimizing time complexity. For each friendship query, it executes a union operation on two villagers and. retrieves the size of set to which the newly connected pair belongs and prints the size.

Task-2

The code implements Kruskal's algorithm to solve an MST problem. The list of roads is sorted based on maintenance cost into a dd. Also it employs DSU to manage connected components of a graph. This helps in determining whether adding a particular road would form a cycle. 'Parent' keeps track of root node for each city and 'rank' helps in keeping tree flat'. The algorithm iterates through sorted roads and uses DSU to add roads that dont form cycles. After MST is constructed, total cost of roads in MST is considered minimal cost.

Task-3:

It is a dynamic programming problem. Here we implemented a recursive function that takes step aim as 'n' and a 'memo' array which stores previous results. If n is 1 it returns 1 and if n is 2 it returns 2 which are base case. The memo array is checked to implement memoisation. If the result for n is not computed, it calculates sum of for n-1 and n-2 which refers to single and double steps from n-1 and n-2. The main function climbingStairs initializes a memoization list of size n+1 which as all values as null or -1. Then it calls the recursive helper function with total step n to start calculation and returns the computed number of ways to climb 'n' steps.

Task - 4

The coinChange function utilizes dynamic programming to solve it. Here A list dp is made with size 'amount'+1' where dp[i] stores the minimum number of coins to make up the amount [i]. Initially all values are amount + 1 or 'infinity', except dp[0]. The function iterates through each sub-amount from 1 to 'amount', updating dp[i] for each coin denomination that is less or equal to i. It calculates minimum coin required by considering addition of one coin of current denomination to best solution found for remainder (i - coin). After dp array is filled, dp[amount] provides minimum coins required if it is lesser than amount, otherwise it returns -1.