Problem 1:

Description: If you have ever played Genshin Impact, you must remember “Divine Ingenuity: Collector’s Chapter” event. In this event, players can create custom domains by arranging components, including props and traps, between the given starting point and exit point. Paimon does not want to design a difficult domain; she pursues the ultimate “automatic map”. In the given domain with a size of m × n, she only placed Airflow and Spikes. Specifically, Spikes will eliminate the player (represented by ‘x’), while the Airflow will blow the player to the next position according to the wind direction (up, left, down, right represented by ‘w’, ‘a’, ‘s’, ‘d’, respectively). The starting point and exit point are denoted by ‘i’ and ‘j’, respectively. Ideally, in Paimon’s domain, the player selects a direction and advances one position initially; afterward, the Airflow propels the player to the endpoint without falling into the Spikes. The player will achieve automatic clearance in such a domain.

However, Paimon, in her slight oversight, failed to create a domain that allows players to achieve automatic clearance. Please assist Paimon by making the minimum adjustments to her design to achieve automatic clearance. Given that the positions of the starting point and exit point are fixed, you can only adjust components at other locations. You have the option to remove existing component at any position; then, place a new direction of Airflow, or position a Spikes.

Solution:

Define the size of the domain as m rows and n columns.

Create a 2D array to represent the domain. Each cell in the array represents a position in the domain.

Initialize the starting point i and exit point j in the array.

Place the Airflow and Spikes components in the array according to the given positions.

Implement a recursive function that takes the current position and direction as input and returns a boolean value indicating whether the player can reach the exit point without falling into the Spikes.

In the recursive function, check if the current position is the exit point. If it is, return True.

Otherwise, check if the current position is out of bounds or contains a Spike. If it does, return False.

If the current position contains an Airflow, update the current position and direction according to the wind direction.

Recursively call the function with the updated position and direction.

If the function returns True, return True.

If the function returns False, try the next direction.

If all directions have been tried and the function has not returned True, return False.

Problem 2:

Description:

Give you a graph with n vertices and m edges. No two edges connect the same two vertices. For vertex ID from 1 to n, we do the following operation: If any two neighbors of a vertex have a k× relationship in terms of their IDs, we add a new edge between them. In other words, for any vertex i = 1 to n, if u = kv or v = ku, we add an edge (u, v), where u, v ∈ Neighbor(i). Besides, if there is already an edge between u and v, no operation is taken.

After the operation, we want you to output the BFS order starting from vertex s. Please traverse all neighbors in ascending order of their IDs when expanding a vertex.

Solution:

Create an adjacency list representation of the graph with n vertices and m edges.

Implement the operation to add new edges between vertices that have a k× relationship in terms of their IDs.

Implement the BFS algorithm to traverse the graph starting from vertex s.

Traverse all neighbors in ascending order of their IDs when expanding a vertex.

Here is the pseudocode for the BFS algorithm:

```
BFS(s):
    queue = [s]
    visited = set()
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in sorted(adj_list[vertex]):
                queue.append(neighbor)
    return visited
```

In this pseudocode, s is the starting vertex, adj_list is the adjacency list representation of the graph, queue is a list that stores the vertices to be visited, and visited is a set that stores the visited vertices.

To implement the operation to add new edges between vertices that have a k× relationship in terms of their IDs, we can iterate over all vertices and their neighbors, and check if any two neighbors have a k× relationship in terms of their IDs. If they do, we add a new edge between them.