Report

1. BFS & DFS

To solve this problem, I could've used DSU or the logic where if, after repeatedly removing all nodes with in-degree 0, some nodes still remain, those nodes must be part of a cycle. And also, as the problem itself suggests, BFS and DFS are solutions.

I personally used DFS with an Adjacency list to store spells because it's better than an Adjacency matrix. So my code just checks if there is any cycle, and if it detects one, then prints Fake, otherwise BALABALA. I treat each spell like a directed graph and explore learning all spells, and there becomes a loop of spells, then it's Fake. I marked all spells with white, grey, and black, like in normal DFS from the lecture, and if the next spell that I wanna learn is Grey, then it means there is a loop if I'm still learning spells.

My code is good because the DFS method detects cycles directly and efficiently without extra data structures, it's simple to implement, and runs in linear time. Also, DFS is better than BFS, because it's good for detecting cycles faster, while BFS needs additional structures to find them.

2. MST

In this problem, the solutions are data structures that build an MST, which are the Prim and Kruskal algorithms. Because MST is a weighted graph.

While solving this problem, I understood that using a Java-implemented PriorityQueue is not a good solution because even if the constraints are not too huge, PriorityQueue doesn't allow updating a vertex's key when a shorter edge is found, and cannot track where a given vertex is stored. So I wrote a few more functions for MinHeap, such as decreaseKey, swap, and minHeapify, and it will be faster and take less space than PriorityQueue in the Java library. Then I used the Prim algorithm logic to find the minimal mana cost. Firstly, I put all the crystals into a heap. Then, starting from the vertex 0, the algorithm repeatedly extracts the vertex with the smallest key, adds its cost to the total, and includes it in the MST. After choosing a vertex u, it checks all other vertices v not in the MST, then computes the Manhattan, and if that distance is smaller than key[v], it updates key[v] and calls decreaseKey to update the heap. This continues until all vertices are extracted.

My solution is fast and low-capacity because I used my own MinHeap and implemented Prim’s algorithm correctly. Also, it avoids duplicates and stale entries, which is common to PriorityQueue.