Problem Statement

There are two types of professional wrestlers: “babyfaces” (“good guys”) and “heels” (“bad guys”). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O(n + r)-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Main Idea

To determine if this wrestling situation is possible, we can use a graph. There are n nodes that represent each professional wrestler and their type of wrestler (”babyfaces” or ”heels”), and r edges that represent each rivalry between two wrestlers. The algorithm would consist of a Breadth-First Search, starting at any unvisited wrestler. This original wrestler would be labeled as a ”babyface,” and all direct neighboring nodes would be labeled as ”heels.” The next level would have your current node as a ”heel,” so all of its neighboring nodes would need to be labeled as ”babyfaces.” You continue this pattern until you either have looped through and labeled each individual node/wrestler , or you have connections between nodes where two of the same type of wrestler could have a rivalry.

Pseudocode

Algorithm 1 Determine possible wrestling rivalries

1: **function** WRESTINGRIVALRIES(void)
2: Initialize global variables:

- n = number of wrestlers
- RivalriesList = adjacency list of wrestler pairs, representing all possible rivalries
- WrestlerArray = array of size n+1, initialized to be all 0. Each item represents a wrestler's type
  - 0 = unassigned, 1= babyface, 2 = heels
- graphQueue = empty queue, to explore wrestlers

3: Iterate using for loop, for each wrestler (i) from 1 to n:

- if WrestlerArray[i] == 0 (wrester is unassigned):
  - WrestlerArray[i] == 1 (assign initial wrester as a babyface)
  - Enqueue i into graphQueue
  - Iterate using while loop, while graphQueue is not empty:
    - currentW = dequeue current wrestler from graphQueue
    - For loop, iterating through each rival in RivalriesList[currentW] :
      - if WrestlerArray[rival] == 0 (unassigned) and WrestlerArray[currentW] == 1 (babyface):
      - *WrestlerArray[rival] = 2 (heels)*
      - *Enqueue rival into graphQueue*
      - elif WrestlerArray[rival] == 0 (unassigned) and WrestlerArray[currentW] == 2 (heel):
      - *WrestlerArray[rival] = 1 (babyface)*
      - *Enqueue rival into graphQueue*
      - elif WrestlerArray[rival] == WrestlerArray[currentW]:
      - *return("IMPOSSIBLE") - two wrestlers in a rivalry were the same type*

4: return("POSSIBLE") - no conflicts were found
5: **end function**

Proof of Correctness

The base case for this algorithm is when there is only one wrestler (n=1). If there is only one wrestler, then they can be labeled as either "babyface" or "heel" without having to worry about potential same-type rivalries. This algorithm would assign the wrestler as a "babyface"

and return "POSSIBLE".

However, if there is more than one wrestler, then the more intensive Breadth-First Search will take place. Assuming there is a graph with N total wrestlers, and the algorithm correctly identifies one group of wrestlers as "babyfaces" and labels their rivals as "heels."

Now, consider a graph with N+1 total wrestlers. When adding a new wrestler, the previous pairings are still assumed to be correct. The algorithm would then be responsible for doing one of two possible choices: assign the wrestler to a valid non-conflicting group, or encounter an issue where two directly neighboring nodes are the same type. If there are two direct nodes with the same type, the algorithm should return "IMPOSSIBLE." Thus, by induction from the algorithm being correct with N wrestlers, the algorithm should be able to correctly determine which of these choices is correct for N+1 wrestlers as well.

Time Complexity

The time complexity of this algorithm is O(n+r). With the adjacency list, we iterate through each pair of rivalry wrestlers only once, which will be O(r). Then in the breadth-first search through the graph, each wrestler/node is only looked at once and only looks at this wrestler's rivalries only once, so you get a time complexity of O(n+r). Since these steps happen one after another, the larger value will be the overall time complexity of the algorithm. This means that the algorithm has a time complexity of O(n+r).