2097. Valid Arrangement of Pairs

Problem Description

Given a set of `n` nodes, there are `n` directed edges that form a sequence. The task is to find the correct arrangement of these edges to form the sequence. You are given a list `pairs` where pairs[i] = [a, b] indicates that there is a directed edge from node a to node b.

Return a list of pairs representing the correct arrangement of the edges. If there are multiple answers, return any of them.

Example:

Given pairs = [[1, 2], [2, 3], [3, 4], [4, 1]]

Output: [[1, 2], [2, 3], [3, 4], [4, 1]]

Approach

The problem is a graph traversal problem, and we will be using Hierholzer's algorithm to find the valid arrangement. More specifically, we will first create a directed graph using the given node pairs. Then, we'll build the correct arrangement of the edges by starting at the unique node with an out-degree of 1 greater than its in-degree, or arbitrarily choosing a starting node if no such unique node exists.

We use the following key data structures:

- We store the graph using an unordered_map with the node value as the key and stacks as the values to store the adjacent nodes.
- We use two unordered_maps to store the in-degree and out-degree of each node.

Step 1: Create the directed graph

We first create the directed graph using the given node pairs. We also populate the in-degree and out-degree maps while doing this.

Step 2: Find the starting node

We check for a unique node where the out-degree is one more than the in-degree. If such a node exists, we start from that node; otherwise, we arbitrarily choose a starting node from the given node pairs.

Step 3: Apply Hierholzer's algorithm to form the valid arrangement

We start at the starting node and traverse the edges in a depth-first manner using recursion and keeping track of the visited edges. We also keep track of the visited edges by popping them from the stack once traversed, which means that the stack will be empty when there is no more unvisited edge from that node.

Once all edges are visited, we reverse the order of the visited edges, representing the valid arrangement.

We'll implement this algorithm in a class called `Solution` for the following languages: Python, Java, JavaScript, C++, and C#.

Python Solution

```
python
class Solution:
    def validArrangement(self, pairs):
        ans = []
        graph = {}
        outDegree = {}
        inDegree = {}

        for pair in pairs:
            start, end = pair
            if not start in graph:
                graph[start] = []
            graph[start].append(end)
            outDegree[start] = outDegree.get(start, 0) + 1
            inDegree[end] = inDegree.get(end, 0) + 1

        startNode = self.getStartNode(graph, outDegree, inDegree, pairs)
        self.euler(graph, startNode, ans)
        ans.reverse()
        return ans

    def getStartNode(self, graph, outDegree, inDegree, pairs):
        for u in graph:
            if outDegree[u] - inDegree.get(u, 0) == 1:
                return u
        return pairs[0][0]

    def euler(self, graph, u, ans):
        while graph[u]:
            v = graph[u].pop()
            self.euler(graph, v, ans)
            ans.append((u, v))
```

Java Solution

```
java
import java.util.*;

class Solution {
    public List<List<Integer>> validArrangement(int[][] pairs) {
        List<List<Integer>> ans = new ArrayList<>();
        Map<Integer, Stack<Integer>> graph = new HashMap<>();
        Map<Integer, Integer> outDegree = new HashMap<>();
        Map<Integer, Integer> inDegree = new HashMap<>();

        for (int[] pair : pairs) {
            int start = pair[0];
            int end = pair[1];
            graph.computeIfAbsent(start, x -> new Stack<>()).push(end);
            outDegree.put(start, outDegree.getOrDefault(start, 0) + 1);
            inDegree.put(end, inDegree.getOrDefault(end, 0) + 1);
        }

        int startNode = getStartNode(graph, outDegree, inDegree, pairs);
        euler(graph, startNode, ans);
        Collections.reverse(ans);
        return ans;
    }

    private int getStartNode(Map<Integer, Stack<Integer>> graph, Map<Integer, Integer> outDegree, Map<Integer, Intege
        for (int u : graph.keySet()) {
            if (outDegree.get(u) - inDegree.getOrDefault(u, 0) == 1) {
                return u;
            }
        }
        return pairs[0][0];
    }

    private void euler(Map<Integer, Stack<Integer>> graph, int u, List<List<Integer>> ans) {
        Stack<Integer> stack = graph.get(u);
        while (!stack.isEmpty()) {
            int v = stack.pop();
            euler(graph, v, ans);
            ans.add(Arrays.asList(u, v));
        }
    }
}
```

JavaScript Solution

```
javascript
class Solution {
    validArrangement(pairs) {
        let ans = [];
        let graph = new Map();
        let outDegree = new Map();
        let inDegree = new Map();

        for (let pair of pairs) {
            let start = pair[0];
            let end = pair[1];
            if (!graph.has(start)) {
                graph.set(start, []);
            }
            graph.get(start).push(end);
            outDegree.set(start, (outDegree.get(start) || 0) + 1);
            inDegree.set(end, (inDegree.get(end) || 0) + 1);
        }

        let startNode = this.getStartNode(graph, outDegree, inDegree, pairs);
        this.euler(graph, startNode, ans);
        ans.reverse();
        return ans;
    }

    getStartNode(graph, outDegree, inDegree, pairs) {
        for (let u of graph.keys()) {
            if ((outDegree.get(u) || 0) - (inDegree.get(u) || 0) == 1) {
                return u;
            }
        }
        return pairs[0][0];
    }

    euler(graph, u, ans) {
        while (graph.get(u).length > 0) {
            let v = graph.get(u).pop();
            this.euler(graph, v, ans);
            ans.push([u, v]);
        }
    }
}
```

C++ Solution

```
cpp
#include <unordered_map>
#include <vector>
#include <stack>
#include <algorithm>

using namespace std;

class Solution {
 public:
  vector<vector<int>> validArrangement(vector<vector<int>>& pairs) {
    vector<vector<int>> ans;
    unordered_map<int, stack<int>> graph;
    unordered_map<int, int> outDegree;
    unordered_map<int, int> inDegree;

    for (const vector<int>& pair : pairs) {
      const int start = pair[0];
      const int end = pair[1];
      graph[start].push(end);
      ++outDegree[start];
      ++inDegree[end];
    }

    const int startNode = getStartNode(graph, outDegree, inDegree, pairs);
    euler(graph, startNode, ans);
    reverse(begin(ans), end(ans));
    return ans;
  }

 private:
  int getStartNode(const unordered_map<int, stack<int>>& graph,
                   unordered_map<int, int>& outDegree,
                   unordered_map<int, int>& inDegree,
                   const vector<vector<int>>& pairs) {
    for (const auto& [u, _] : graph)
      if (outDegree[u] - inDegree[u] == 1)
        return u;
    return pairs[0][0];  // Arbitrarily choose a node
  }

  void euler(unordered_map<int, stack<int>>& graph, int u,
             vector<vector<int>>& ans) {
    auto& stack = graph[u];
    while (!stack.empty()) {
      const int v = stack.top();
      stack.pop();
      euler(graph, v, ans);
      ans.push_back({u, v});
    }
  }
};
```

C# Solution

```
csharp
using System;
using System.Collections.Generic;

public class Solution {
    public IList<IList<int>> ValidArrangement(int[][] pairs) {
        List<IList<int>> ans = new List<IList<int>>();
        Dictionary<int, Stack<int>> graph = new Dictionary<int, Stack<int>>();
        Dictionary<int, int> outDegree = new Dictionary<int, int>();
        Dictionary<int, int> inDegree = new Dictionary<int, int>();

        foreach (int[] pair in pairs) {
            int start = pair[0];
            int end = pair[1];
            if (!graph.ContainsKey(start)) {
                graph[start] = new Stack<int>();
            }
            graph[start].Push(end);
            outDegree[start] = outDegree.GetValueOrDefault(start, 0) + 1;
            inDegree[end] = inDegree.GetValueOrDefault(end, 0) + 1;
        }

        int startNode = GetStartNode(graph, outDegree, inDegree, pairs);
        Euler(graph, startNode, ans);
        ans.Reverse();
        return ans;
    }

    private int GetStartNode(Dictionary<int, Stack<int>> graph, Dictionary<int, int> outDegree, Dictionary<int, int>
        foreach (int u in graph.Keys) {
            if (outDegree[u] - inDegree.GetValueOrDefault(u, 0) == 1) {
                return u;
            }
        }
        return pairs[0][0];
    }

    private void Euler(Dictionary<int, Stack<int>> graph, int u, List<IList<int>> ans) {
        Stack<int> stack = graph[u];
        while (stack.Count > 0) {
            int v = stack.Pop();
            Euler(graph, v, ans);
            ans.Add(new int[] {u, v});
        }
    }
}
```