2699. Modify Graph Edge Weights

ou are given an **undirected weighted connected** graph containing `n` nodes labeled from `0` to `n - 1`, and an integer array `edges` where `edges[i] = [ai, bi, wi]` indicates that there is an edge between nodes `ai` and `bi` with weight `wi`.

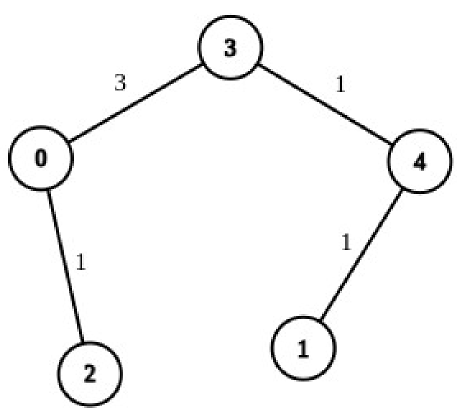
Some edges have a weight of `-1` (`wi = -1`), while others have a **positive** weight (`wi > 0`).

Your task is to modify **all edges** with a weight of `-1` by assigning them **positive integer values** in the range `[1, 2 * 109]` so that the **shortest distance** between the nodes `source` and `destination` becomes equal to an integer `target`. If there are **multiple modifications** that make the shortest distance between `source` and `destination` equal to `target`, any of them will be considered correct.

Return *an array containing all edges (even unmodified ones) in any order if it is possible to make the shortest distance from* `source` *to* `destination` *equal to* `target`*, or an* ***empty array*** *if it's impossible.*

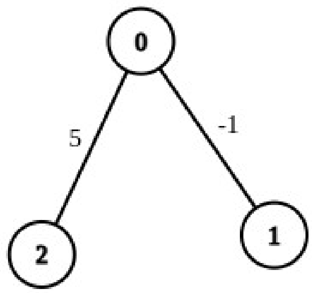
Note: You are not allowed to modify the weights of edges with initial positive weights.

Example 1:



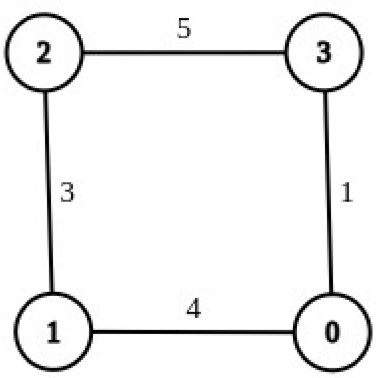

```
Input: n = 5, edges = [[4,1,-1],[2,0,-1],[0,3,-1],[4,3,-1]], source = 0, destination = 1, target = 5
Output: [[4,1,1],[2,0,1],[0,3,3],[4,3,1]]
Explanation: The graph above shows a possible modification to the edges, making the distance from 0 to 1 equal to 5.
```

Example 2:



Input: n = 3, edges = [[0,1,-1],[0,2,5]], source = 0, destination = 2, target = 6
Output: []
Explanation: The graph above contains the initial edges. It is not possible to make the distance from 0 to 2 equal to 6 by modifying the edge with weight -1. So, an empty array is returned.

Example 3:



Input: n = 4, edges = [[1,0,4],[1,2,3],[2,3,5],[0,3,-1]], source = 0, destination = 2, target = 6
Output: [[1,0,4],[1,2,3],[2,3,5],[0,3,1]]
Explanation: The graph above shows a modified graph having the shortest distance from 0 to 2 as 6.

Constraints:

- `1 <= n <= 100`
- `1 <= edges.length <= n * (n - 1) / 2`
- `edges[i].length == 3`
- `0 <= ai, bi < n`
- `wi = -1` or `1 <= wi <= 107`
- `ai != bi`
- `0 <= source, destination < n`
- `source != destination`
- `1 <= target <= 109`
- The graph is connected, and there are no self-loops or repeated edges

2699. Modify Graph Edge Weights

```
/*
   Dijkstra
   Time complexity: O((V+E)log V)
   Space complexity: O(V+E)
*/

typedef std::pair<int,int> ii;
typedef std::vector<ii> vii;
typedef std::vector<vii> vvii;

typedef std::vector<int> vi;
typedef std::vector<vi> vvi;

class Solution {
   public:
      vvii graph;
      vvi distances;
   public:
      void build_graph(int n,vvi& edges){
         graph.resize(n);
         int m=edges.size();
         for(int i=0;i<m;++i){
            vi edge=edges[i];
            int u=edge[0];
            int v=edge[1];
            graph[u].push_back({v,i});
            graph[v].push_back({u,i});
         }
      }
```

```
/*
   op=0 => assign to all modifiable edges 1
   op=1 => adjust the modifiable edges to achieve the exact path length
*/
void dijkstra(int n,vvi& edges,int source,int destination,int difference,int op){
   distances[source][op]=0;
   vi vis(n,0);
   std::priority_queue<ii,vii,std::greater<ii>> min_heap;
   min_heap.push({0,source});
   while(!min_heap.empty()){
      auto [cur_w,u]=min_heap.top();
      min_heap.pop();

      if(cur_w>distances[u][op]) continue;

      if(vis[u]) continue;
      vis[u]=1;

      if(u==destination) return;

      for(auto& neighbor: graph[u]){
         int v=neighbor.first;
         int i=neighbor.second;
         int w=edges[i][2];

         if(w==-1) w=1;

         if(op==1 && edges[i][2]==-1){
            int new_w=difference+distances[v][0]-distances[u][1];
            edges[i][2]=w=new_w;
         }

         if(distances[v][op]>distances[u][op]+w){
            distances[v][op]=distances[u][op]+w;
            min_heap.push({distances[v][op],v});
         }
      }
   }
}
```

```
    vvi modifiedGraphEdges(int n, vvi& edges, int source, int destination, int target) {
        build_graph(n,edges);

        distances.resize(n,vi(2,INT_MAX/2));

        dijkstra(n,edges,source,destination,0,0);

        int difference=target-distances[destination][0];

        // Same as: if(distances[destination][0]==INT_MAX || distances[destination][0]>target) return {};
        if(difference<0) return {};

        dijkstra(n,edges,source,destination,difference,1);

        if(distances[destination][1]<target) return {};

        for (auto& edge : edges) {
            if (edge[2] <=0) edge[2]=1;
        }

        return edges;
    }
};
```