Articulation Point in C++

```
#include <bits/stdc++.h>
using namespace std;

//User function Template for C++

class Solution {
private:
   int timer = 1;
   void dfs(int node, int parent, vector<int> &vis, int
tin[], int low[],
          vector<int> &mark, vector<int>adj[]) {
       vis[node] = 1;
       tin[node] = low[node] = timer;
       timer++;
       int child = 0;
       for (auto it : adj[node]) {
          if (it == parent) continue;
          if (!vis[it]) {
             dfs(it, node, vis, tin, low, mark, adj);
             low[node] = min(low[node], low[it]);
             if (low[it] >= tin[node] && parent != -1) {
                mark[node] = 1;
             }
             child++;
          }
          else {
             low[node] = min(low[node], tin[it]);
          }
       }
       if (child > 1 && parent == -1) {
          mark[node] = 1;
       }
   }
public:
   vector<int> articulationPoints(int n, vector<int>adj[])
{
       vector<int> vis(n, 0);
       int tin[n];
       int low[n];
       vector<int> mark(n, 0);
       for (int i = 0; i < n; i++) {
          if (!vis[i]) {
             dfs(i, -1, vis, tin, low, mark, adj);
          }
       }
       vector<int> ans;
       for (int i = 0; i < n; i++) {
          if (mark[i] == 1) {
             ans.push_back(i);
          }
       }
       if (ans.size() == 0) return { -1};
       return ans;
   }
};
int main() {

   int n = 5;
   vector<vector<int>> edges = {
      {0, 1}, {1, 4},
      {2, 4}, {2, 3}, {3, 4}
```

Dry Run:

Let's dry run the algorithm with the following graph represented by edges:

```
int n = 5;
vector<vector<int>> edges = {
    {0, 1}, {1, 4},
    {2, 4}, {2, 3}, {3, 4}
};
```

The graph can be visualized as:

```
yaml
Copy code
   0
   |
   1
  / \
 4---2
     |
     3
```

Step 1: Initialize Variables

- `vis`: A boolean vector initialized to `[0, 0, 0, 0, 0]` (all nodes unvisited).
- `tin`: A vector initialized to `[-1, -1, -1, -1, -1]`.
- `low`: A vector initialized to `[-1, -1, -1, -1, -1]`.
- `mark`: A vector initialized to `[0, 0, 0, 0, 0]` (articulation points).
- `timer`: Set to `1`, used to assign discovery times.

Step 2: DFS Traversal

- Start DFS from node `0`:
  - For node `0`:
    - Set `tin[0] = low[0] = 1`.
    - Visit neighbors: `1` (child).
  - For node `1`:
    - Set `tin[1] = low[1] = 2`.
    - Visit neighbors: `0` (parent) and `4` (child).
  - For node `4`:
    - Set `tin[4] = low[4] = 3`.
    - Visit neighbors: `1` (parent), `2` (child).

```
    };

    vector<int> adj[n];
    for (auto it : edges) {
        int u = it[0], v = it[1];
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    Solution obj;
    vector<int> nodes = obj.articulationPoints(n, adj);
    for (auto node : nodes) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}
```

- For node `2`:
  - Set `tin[2] = low[2] = 4`.
  - Visit neighbors: `4` (parent), `3` (child).
- For node `3`:
  - Set `tin[3] = low[3] = 5`.
  - Visit neighbors: `2` (parent).
  - DFS ends for node `3`, return to `2`.
- For node `2`, update `low[2]` as `min(low[2], low[3]) = 4`.
- As `low[3] >= tin[2]`, mark node `2` as an articulation point.
- For node `4`, update `low[4]` as `min(low[4], low[2]) = 3`.
- As `low[2] >= tin[4]`, mark node `4` as an articulation point.
- For node `1`, update `low[1]` as `min(low[1], low[4]) = 2`.
- As `low[4] >= tin[1]`, mark node `1` as an articulation point.

Step 3: Collect and Sort Results

- After DFS completes, `mark` contains `[0, 1, 1, 0, 1]`, indicating that nodes `1`, `2`, and `4` are articulation points.
- The final output will be `1 4` (sorted articulation points).

Output:-
1 4