Binary Tree Algorithms - Advanced Patterns

1. Burn a Binary Tree from a Leaf Node

Problem

Given a binary tree and a start node, simulate infection/burning of all nodes minute by minute. Return the time it takes to infect the entire tree.

Approach

1. Mark Parents using BFS
2. Do BFS from the start node and infect unvisited neighbors (left, right, parent)
3. Track minutes as levels

Code

```
int amountOfTime(TreeNode* root, int start) {
    unordered_map<TreeNode*, TreeNode*> parent;
    queue<TreeNode*> q;
    TreeNode* source = nullptr;

    // Step 1: Mark parents
    q.push(root);
    while (!q.empty()) {
        TreeNode* node = q.front(); q.pop();
        if (node->val == start) source = node;
        if (node->left) parent[node->left] = node, q.push(node->left);
        if (node->right) parent[node->right] = node, q.push(node->right);
    }

    // Step 2: BFS from source
    unordered_map<TreeNode*, bool> visited;
    q.push(source);
    visited[source] = true;
    int time = -1;

    while (!q.empty()) {
```

```
        int n = q.size();
        while (n--) {
            TreeNode* curr = q.front(); q.pop();
            for (TreeNode* neighbor : {curr->left, curr->right, parent[cu
                if (neighbor && !visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        time++;
    }
    return time;
}
```

2. Distance K from Target Node

Problem

Find all nodes at distance K from a target node.

Approach

1. Build Parent Map using BFS
2. Do a Level Order BFS starting from target
3. Stop when level == K

Code

```
vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
    unordered_map<TreeNode*, TreeNode*> parent;
    queue<TreeNode*> q;
    q.push(root);

    // Step 1: Mark Parents
    while (!q.empty()) {
        TreeNode* node = q.front(); q.pop();
        if (node->left) parent[node->left] = node, q.push(node->left);
        if (node->right) parent[node->right] = node, q.push(node->right);
    }
```

```
    // Step 2: BFS from target
    unordered_map<TreeNode*, bool> visited;
    q.push(target);
    visited[target] = true;
    int dist = 0;

    while (!q.empty()) {
        if (dist == k) break;
        int n = q.size();
        while (n--) {
            TreeNode* curr = q.front(); q.pop();
            for (TreeNode* neighbor : {curr->left, curr->right, parent[cu
                if (neighbor && !visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        dist++;
    }

    vector<int> res;
    while (!q.empty()) res.push_back(q.front()->val), q.pop();
    return res;
}
```

3. Construct Binary Tree from Traversals

A. Preorder + Inorder → Binary Tree

Approach

1. The first element in preorder is root
2. Find this root in inorder → elements to left = left subtree, right = right subtree
3. Recursively build left and right

Code

```
TreeNode* build(vector<int>& preorder, int preStart, int inStart, int inE
    if (inStart > inEnd) return nullptr;
```

```
    TreeNode* root = new TreeNode(preorder[preStart]);
    int inIndex = inMap[root->val];
    int leftSize = inIndex - inStart;

    root->left = build(preorder, preStart + 1, inStart, inIndex - 1, inMa
    root->right = build(preorder, preStart + 1 + leftSize, inIndex + 1, i
    return root;
}

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    unordered_map<int, int> inMap;
    for (int i = 0; i < inorder.size(); i++) inMap[inorder[i]] = i;
    return build(preorder, 0, 0, inorder.size() - 1, inMap);
}
```

B. Postorder + Inorder → Binary Tree

Approach

1. The last element in postorder is root
2. Similar partitioning using inorder
3. Recursively build right subtree first (postorder: Left → Right → Root)

Code

```
TreeNode* build(vector<int>& postorder, int postEnd, int inStart, int inE
    if (inStart > inEnd) return nullptr;

    TreeNode* root = new TreeNode(postorder[postEnd]);
    int inIndex = inMap[root->val];
    int rightSize = inEnd - inIndex;

    root->right = build(postorder, postEnd - 1, inIndex + 1, inEnd, inMap
    root->left = build(postorder, postEnd - rightSize - 1, inStart, inInc
    return root;
}

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    unordered_map<int, int> inMap;
    for (int i = 0; i < inorder.size(); i++) inMap[inorder[i]] = i;
    return build(postorder, postorder.size() - 1, 0, inorder.size() - 1,
}
```

4. Serialize and Deserialize a Binary Tree (BFS)

Serialize: Level Order with Nulls

```
string serialize(TreeNode* root) {
    if (!root) return "#";
    queue<TreeNode*> q;
    q.push(root);
    string result;

    while (!q.empty()) {
        TreeNode* node = q.front(); q.pop();
        if (!node) result += "# ", continue;
        result += to_string(node->val) + " ";
        q.push(node->left);
        q.push(node->right);
    }
    return result;
}
```

5. Morris Traversal (Inorder and Preorder)

Inorder (Left → Root → Right)

Code

```
vector<int> morrisInorder(TreeNode* root) {
    vector<int> res;
    TreeNode* curr = root;

    while (curr) {
        if (!curr->left) {
            res.push_back(curr->val);
            curr = curr->right;
        } else {
            TreeNode* pred = curr->left;
            while (pred->right && pred->right != curr) pred = pred->right

            if (!pred->right) {
                pred->right = curr;
```

```
                curr = curr->left;
            } else {
                pred->right = nullptr;
                res.push_back(curr->val);
                curr = curr->right;
            }
        }
    }
    return res;
}
```

Preorder (Root → Left → Right)

Code

```
vector<int> morrisPreorder(TreeNode* root) {
    vector<int> res;
    TreeNode* curr = root;

    while (curr) {
        if (!curr->left) {
            res.push_back(curr->val);
            curr = curr->right;
        } else {
            TreeNode* pred = curr->left;
            while (pred->right && pred->right != curr) pred = pred->right

            if (!pred->right) {
                res.push_back(curr->val);  // Visit before threading
                pred->right = curr;
                curr = curr->left;
            } else {
                pred->right = nullptr;
                curr = curr->right;
            }
        }
    }
    return res;
}
```

Key Notes

- **Diameter ≠ Burn Time always**
  - Burn time is from a specific node outward (like BFS)
  - Diameter is longest path in tree (from any node to any node)
- **Morris Traversal** is threading based and uses O(1) space