Description

94. Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

```
Example:

Input: [1,null,2,3]
   1
    \
     2
    /
   3

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

Idea

Firstly we want to clearify the the inorder is left-root-right order. The recursive solution is trivial, we may want to point out that the call stack may overflow if the tree is large. We visited each node only once, so O(n) time, and call stack is O(H) space, where H is the maximum height of the tree.

Java

```java
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        helper(root, res);
        return res;
    }

    private void helper(TreeNode root, List<Integer> res) {
        if (root == null) {
            return;
        }
```

```
        helper(root.left, res);
        res.add(root.val);
        helper(root.right, res);
    }
}
```

The above approach is recursion top-down approach. Another recursive approach is divide and conquer bottom-up. We think left child has all the list of integers ready (leftList), and right child has all the list of integers ready (rightList). We can append leftList + root.val + rightList to get the final result. We visited each node only once, so O(n) time, and call stack is O(H) space, where H is the maximum height of the tree.

Java

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) {
            return res;
        }

        res.addAll(inorderTraversal(root.left));
        res.add(root.val);
        res.addAll(inorderTraversal(root.right));
        return res;
    }
}
```

Since we talked about the stack overflow, we might want to discuss a non-recursive version. Basically we just use a stack to simulate the call stack. The order is left-root-right, so when we handled left node, we need to go back to root, so we need a stack to store that informaiton. We visited each node only once, so O(n) time, and stack is O(H) space, where H is the maximum height of the tree.

Java

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> stack = new ArrayDeque<>();
```

```
        while (!stack.isEmpty() || root != null) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }

        return res;
    }
}
```

Or we can modulate the pushLeft into a push() function:

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> stack = new ArrayDeque<>();
        push(stack, root);

        while (!stack.isEmpty()) {
            TreeNode cur = stack.pop();
            res.add(cur.val);
            push(stack, cur.right);
        }

        return res;
    }

    private void push(Deque<TreeNode> stack, TreeNode node) {
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
    }
}
```

(The last version was introduced by Qinyuan and I think it is a very elegent solution.) When we visit to a node, we actually need to do three sub problem, 1) visit left child, 2) print current node, 3) visit right child. So for a node we have two operations visit and print. Each time the node is actually visited, we change it to the print state. For example, we have [1,2,3], if we visit root 2, we don't want to print it right now, instead we set its state to print, then next time we see node 2 and it's on print state, we print or add to the result list. Then we handle node 1 and 3 and so on. We

visited each node exactly twice, so O(n) time, and call stack is O(H) space, where H is the maximum height of the tree.

Java

```
class Solution {

    class Node {
        TreeNode node;
        boolean isPrint;
        public Node(TreeNode node) {
            this.node = node;
        }
    }

    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<Node> stack = new ArrayDeque<>();
        stack.push(new Node(root));

        while (!stack.isEmpty()) {
            Node cur = stack.pop();
            if (cur.node == null) {
                continue;
            }

            if (cur.isPrint) {
                res.add(cur.node.val);
            } else {
                stack.push(new Node(cur.node.right));
                cur.isPrint = true;
                stack.push(cur);
                stack.push(new Node(cur.node.left));
            }
        }

        return res;
    }
}
```

Summary

- Divid and conquer is a very common method to solve a binary tree question.
- Dividing a big problem into a sequence of sub problems is a very impoartant idea.

- Binary tree pre-/in-/post-/level-order traversal recursion top/down/bottom-up, non-recursion are the basic building blocks.

C++ implementation

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        dfsHelper(root, res);
        return res;
    }

    void dfsHelper(TreeNode* root, vector<int>& res) {
        if (root == NULL) {
            return;
        }

        dfsHelper(root->left, res);
        res.push_back(root->val);
        dfsHelper(root->right, res);
    }
};
```

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == NULL) {
            return res;
        }

        vector<int> left = inorderTraversal(root->left);
        vector<int> right = inorderTraversal(root->right);
        res.insert(res.end(), left.begin(), left.end());
        res.push_back(root->val);
        res.insert(res.end(), right.begin(), right.end());
        return res;
    }
};
```

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> stack;
        while (!stack.empty() || root != NULL) {
            while (root != NULL) {
                stack.push(root);
                root = root->left;
            }
            root = stack.top();
            stack.pop();
            res.push_back(root->val);
            root = root->right;
        }
        return res;
    }
};
```

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        deque<TreeNode*> stack;
        push(stack, root);

        while (!stack.empty()) {
            TreeNode* cur = stack.back();
            stack.pop_back();
            res.push_back(cur->val);
            push(stack, cur->right);
        }

        return res;
    }

private:
    void push(deque<TreeNode*>& stack, TreeNode* node) {
        while (node != NULL) {
            stack.push_back(node);
            node = node->left;
        }
    }
};
```

```cpp
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        deque<Node*> stack;
        stack.push_back(new Node(root));

        while (!stack.empty()) {
            Node* cur = stack.back();
            stack.pop_back();
            if (cur->node == NULL) {
                continue;
            }

            if (cur->isPrint) {
                res.push_back(cur->node->val);
            } else {
                stack.push_back(new Node(cur->node->right));
                cur->isPrint = true;
                stack.push_back(cur);
                stack.push_back(new Node(cur->node->left));
            }
        }

        return res;
    }

private:
    struct Node {
        TreeNode* node;
        bool isPrint;
        Node(TreeNode* node) : node(node), isPrint(false) {}
    };
};
```