递归与栈：解决表达式求值

1.二叉树的后序遍历

https://leetcode-cn.com/problems/binary-tree-postorder-traversal/

后续遍历我们可以不断的访问整个树的左节点，直到找出当前节点为最左边的节点然后输出，在去找当前节点的父节点的右节点直到找到最右边的节点然后输出，最后再输出父节点。直到将整个树遍历完成；

递归方法就是调用方法不断地访问当前节点的左节点直到左节点没有左子节点，再调用递归去访问当前节点的右节点直到没有右子节点，最后再输出当前节点。

递归：

```
/*
 * [145] 二叉树的后序遍历
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var postorderTraversal = function (root) {
    let res = new Array();
    return postorderTraversalNode(root,res);
};
var postorderTraversalNode = function (node,res) {
    if (node) {
        postorderTraversalNode(node.left,res);
        postorderTraversalNode(node.right,res);
```

```
        res.push(node.val);
    }
    return res;
}
```

迭代：

迭代的方法就是我们可以反向的进行后续遍历——中后前，然后将遍历的值加入到栈中，再通过栈来输出加入到数组中，由于栈是先进后出的，所以我们加入到数组的值是前后中，就为前序遍历了。

```
/*
 * [145] 二叉树的后序遍历
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var postorderTraversal = function (root) {
    var res =[];
    if(!root) return res
    var stack=[root];
    while(stack.length){
        root=stack.pop();
        res.unshift(root.val);
        if(root.left) stack.push(root.left);
        if(root.right) stack.push(root.right);
    }
    return res;
};
```

2.验证二叉树的前序序列化

https://leetcode-cn.com/problems/verify-preorder-serialization-of-a-binary-tree/

我们可以利用栈的特性来完成这件事，前序遍历的顺序是中前后。我们可以在栈中记录一个数字，代表当前节点可能有几个节点。首先我们存入一个1代表只有一个根节点。

然后我们开始遍历字符串，如果我们遇到了"#"就代表这个节点是空节点，我们就需要将栈顶的数字进行减一，代表我们已经找到了一个子节点。如果我们遇到的是数字，就代表当前节点不为空，我们就需要将栈顶的数字进行减一，代表我们已经找到了一个子节点，并且一个不为空的节点可能有两个子节点，所以我们要在栈中再压入一个2。

我们需要判断每次遍历，栈顶元素是否为0，如果为0代表这个中间节点的两个子节点都找到了，当前的节点遍历完成。要进行弹栈操作。

到最后我们就判断栈中是否有元素即可，如果还有元素，就证明序列化错误。

```
/*
 *
 * [331] 验证二叉树的前序序列化
 */
/**
 * @param {string} preorder
 * @return {boolean}
 */
var isValidSerialization = function (preorder) {
    let n = preorder.length, i = 0, stack = [1];
    while (i < n) {
        if (!stack.length) return false;
        if (preorder[i] === ',') i++;
        else if (preorder[i] === '#') {
            stack[stack.length - 1]--;
            if (stack[stack.length - 1] == 0) stack.pop();
            i++;
        } else {
            while (i < n && preorder[i] !== ',') {
                i++;
            }
            stack[stack.length - 1]--;
            if (stack[stack.length - 1] == 0) {
                stack.pop();
            }
            stack.push(2);
```

```
        }
    }
    return !stack.length;
};
```

3.基本计算器 II

https://leetcode-cn.com/problems/basic-calculator-ii/

我们在进行运算时，由于乘除法的优先级大于加减法，所以我们每次进行运算的时候，要根据下一个运算符进行判断。所以我们需要在最前面手动添加一个运算符来辅助我们进行运算。

我们可以创建一个栈，对于加减号后的数字，将其直接压入栈中；对于乘除号后的数字，可以直接与栈顶元素计算，并替换栈顶元素为计算后的结果。

具体来说，遍历字符串 s，并用变量preSign 记录每个数字之前的运算符，对于第一个数字，其之前的运算符视为加号。每次遍历到数字末尾时，根据 preSign 来决定计算方式：

最后我们将栈中的所有数字进行累加就得到了我们需要的计算值。

```
/*
 *
 * [227] 基本计算器 II
 */
/**
 * @param {string} s
 * @return {number}
 */
var calculate = function (s) {
    s = s.trim();
    let stack = new Array();
    let preSign = '+';
    let n = s.length;
    let num = 0;
    for (let i = 0; i < n; i++) {
        if (!isNaN(Number(s[i])) && s[i] !== ' ') {
            num = num * 10 + Number(s[i]);
        }
        if (isNaN(Number(s[i])) || i === n - 1) {
            switch (preSign) {
                case '+':
```

```
                    stack.push(num);
                    break;
                case '-':
                    stack.push(-num);
                    break;
                case '*':
                    stack.push(stack.pop() * num);
                    break;
                default:
                    stack.push(stack.pop() / num | 0);
                    break;
            }
            preSign = s[i];
            num = 0;
        }
    }
    let ans = 0;
    while (stack.length) {
        ans += stack.pop();
    }
    return ans;
};
```

4.函数的独占时间

https://leetcode-cn.com/problems/exclusive-time-of-functions/

我们可以先找到最后执行的任务(C)，去计算下这个任务花了多长时间，然后再去计算上一层任务(B)所花费的总时间，然后要减去C任务所花费的时间，就得到了B任务自己花费的时间。然后将每个任务花费的时间加入到数组的指定位置上。

我们可以利用栈的特性来完成这个操作；

```
/*
 * @lc app=leetcode.cn id=636 lang=javascript
 *
 * [636] 函数的独占时间
 */
/**
 * @param {number} n
 * @param {string[]} logs
 * @return {number[]}
```

```
 */
var exclusiveTime = function (n, logs) {
    let res = new Array(n).fill(0);
    let go = 0;
    function next() {
        let dura = 0;
        const start = logs[go].split(':');
        while (go < logs.length - 1 && logs[go + 1].indexOf('s')
!== -1) {
            go++;
            dura = dura + next();
        }
        const end = logs[++go].split(':');
        let sum = Number(end[2] - start[2]) + 1 - dura;
        res[Number(start[0])] = res[Number(start[0])] + sum;
        return sum + dura;
    }
    while (go < logs.length) {
        next(res);
        go++;
    }
    return res;
};
```

5.表现良好的最长时间段

https://leetcode-cn.com/problems/longest-well-performing-interval/

我们将工作时长大于8小时的那一天记作为1，小于等于八小时的那一天记为-1。这样我们就得到了一组只有1和-1的数组。我们从第一天到第n天的和如果大于0，就证明从第一天到第n天这段区间是表现良好的。所以我们可以计算下从第一天开始到第n天的和，也就是我们的前缀和。

我们要求的区间假设是第i天到第j天。要满足i<j并且preSum[j]-preSum[i]大于零。所以我们第i天的前缀和要尽可能的小。当这个前缀和为负数的时候，相减的结果就可能大于0。所以我们在前缀和数组的首位添加一个0来寻找preSum[i]为负数的位置。同时我们要将0压入栈中挨个比较，找到前缀和的值小于栈顶元素的值时，就将该前缀和的下标压入栈中，直到将整个数组遍历完。我们的i就从这个前缀和的数组中取值。

最后就遍历我们的整个hours数组，由于我们求的时j-i要尽可能的大，所以我们采用倒序遍历。当preSum[j]>preSum[i]的时候，我们就让j-i，并记录下相减的差值，然后我们的栈弹出最顶的元素。进行下一次循环操作，直到栈中的i全部遍历完，我们就将最大值求了出来。

```
/*
 * @lc app=leetcode.cn id=1124 lang=javascript
 *
 */
/**
 * @param {number[]} hours
 * @return {number}
 */
var longestWPI = function (hours) {
    let preSum = new Array(hours.length + 1).fill(0);
    for (let i = 0; i < hours.length; i++) {
        if (hours[i] > 8) preSum[i + 1] = preSum[i] + 1;
        else preSum[i + 1] = preSum[i] - 1;
    }
    let stack = [];
    stack.push(0);
    for (let i = 1; i < preSum.length; i++) {
        if (preSum[stack[stack.length - 1]] > preSum[i])
stack.push(i);
    }
    let max = 0;
    for (let i = preSum.length-1; i > max; i--) {
        while (preSum[stack[stack.length - 1]] < preSum[i]&&
stack.length) {
            max = Math.max(max, i - stack.pop());
        }
    }
    return max;
};
```