第三周学习总结

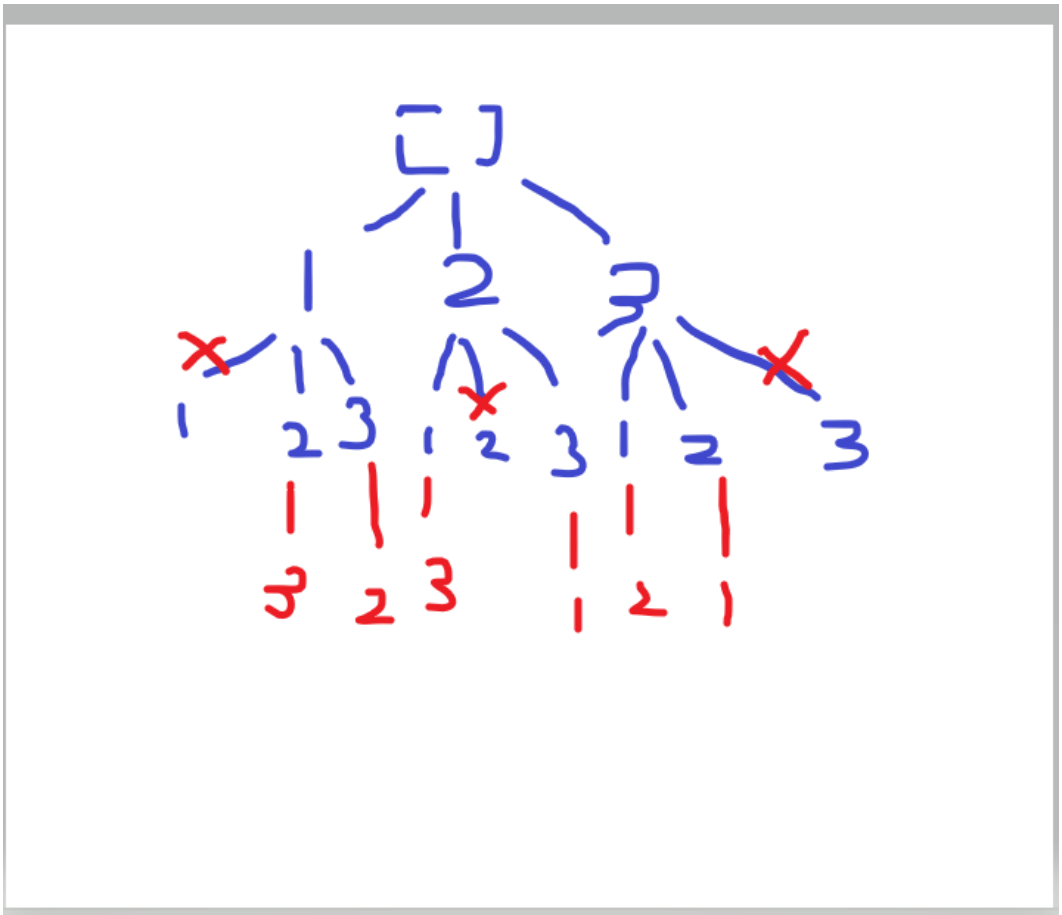
笔记本:	算法与数据结构		
创建时间:	2020/9/12 14:19	**更新时间:**	2020/9/13 12:06
作者:	pants君		

递归，分治，回溯

- 分治，回溯其实都算是递归的一种，分治把问题拆解为子问题，然后从子问题得到的解最终再合并为完整解；回溯则有一种不撞南墙回头的感觉，它会把所有分支都去遍历，如果遇到的解不是正确的解，它就会回到之前的节点上去重复一样的操作，
- 回溯相关的题目：
  - 组合，全排列i

*做回溯相关题目遇到的一些问题

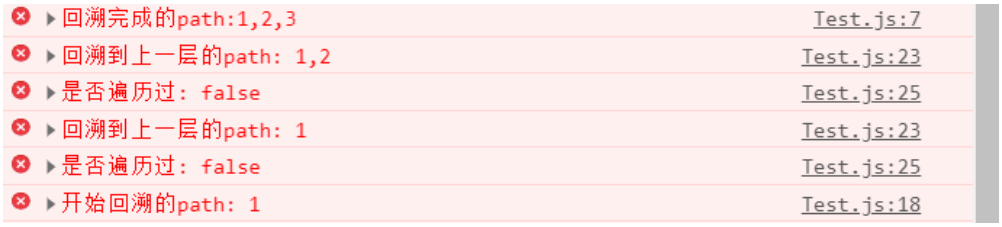
- 全排列i: 使用js编写，思路开始是先写出递归树：



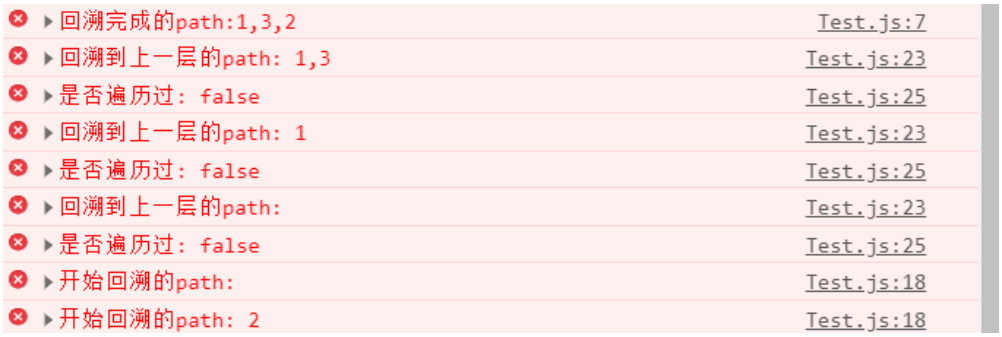
根据递归树所得出的规律，每下一层，就把所有可能都举出来（这里空间不够，所以第2层开始没吧所有情况都画出来）遇到与上一个节点相同的元素就跳过，遍历到path路径的长度等于元素个数，就把正确解推入结果数组中，原思路代码如下（初期思路，写得乱七八杂的）：

```
var permute = function (nums) {
    let path = [];
    let res = [];
    const permuteHelper = (start, path) => {
        if (path.length === nums.length) {
            res.push(path.slice());
        }
        for (let i = 0; i < nums.length; i++) { //列出当前节点与之搭档的所有节点可能，
            if (nums[start] === path[i]) continue;  // 如果候选节点和当前节点重复就排除
            path.push(nums[i]);  // 如果直接推入nums[i]还会报栈溢出 = =
            permuteHelper(i + 1, path);  // 从这里这就整不下去了，思路已乱 ==
        }
    }
    permuteHelper(0, path);  // 从0开始
    return res;
};
```

- 看完题解，和自己的辣鸡代码对比理解了一下，首先我缺少一个状态变量，就是比如说从1开始往下去寻找下一个可以搭配的元素，我可以设置一个状态变量，去记录当前节点是否已经在这个分支上访问过了，如果访问过就跳过，这比自己带入值，每个元素比较与这个值是否相等好一点（至少比较不烧脑），而且状态变量在回溯时只要把bool值改回false就好了。带值比较，等考虑回溯时就不合适了，
- 接下来就是如何正确考虑回溯的问题：
    - 我现在就是缺少当一个分支遍历完后怎么退回到之前的节点去组合新的解这么一个思路，当我把获取到的还没有访问过的节点push到path后，我要把它的访问状态设为true，然后开始递归，比如说 从1开始，path.push(1）,那就是path=[1]; 递归开始，还是在 nums[1,2,3]遍历，因为1已经访问过了，跳过，只剩下2和3, 2没被访问过，很好，2进来；设置已访问，然后继续递归；好家伙，1,2都被访问过了，都跳过了，就剩下3，进来。下一次递归发现已经到分支尽头了，正解输入到结果数组里，这一次循环里，因为所有元素都被访问过了，都被跳过，接着就是考虑让这玩意回溯到上一个节点去组合新的解了；
    - 要回溯，我们可以从path入手，把他后面的元素弹出，把状态变量都设为false，重新开始



这是一次求解结束后开始回溯的打印，从3开始被弹出，状态值被修改，之后2也被弹出以此类推，又从1开始去寻找新的解。



当1为领导的组合全部找到后，1也会被弹出(在写这个的时候我其实还没有完全想通，是怎么做到1在找到所有解之后才被弹出的)
全部代码如下（虽然是看题解才写出来的www）

```
var permute = function (nums) {
    let path = [];
    let res = [];
    let used = {};
    const permuteHelper = (path) => {
        if (path.length === nums.length) {
            console.error(`回溯完成的path:${path}`);
            res.push(path.slice());
        }
        //没有一次编译过的问题，变量没有加声明符号
        for (const num of nums) {
            if (used[num]) continue;
            console.error(`开始回溯的path: ${path}`);
            path.push(num);
            used[num] = true;
            permuteHelper(path);
            path.pop();
            console.error(`回溯到上一层的path: ${path}`);
            used[num] = false;
            console.error(`是否遍历过: ${used[num]}`);
        }
    }
    permuteHelper(path);
    return res;
```

```
};
```