

In [5]: 1 {i:x.count(i) for i in set(x)}

Out[5]: {'S': 5, 'A': 3, 'B': 2}

$O(n)$

In [ ]:

```

1 class Solution:
2     def preorderTraversal(self, root: TreeNode) -> List[int]:
3         # 特殊情况
4         if not root: return []
5         # 1. 前序递归!
6         return [root.val] + self.preorderTraversal(root.left) + self.preorderTraversal(root.right)
7
8         # 2. 前序递归2和1类似
9         def dfs(cur):
10             if not cur: return
11             res.append(cur.val)
12             dfs(cur.left)
13             dfs(cur.right)
14         res = []
15         dfs(root)
16         return res
17
18         # 3. 前序迭代1
19         stack = [root]; res = []
20         while stack:
21             cur = stack.pop()
22             res.append(cur.val) # 同时也会加入根节点的值
23             if cur.right: stack.append(cur.right) # 先打印左, 所以先放右
24             if cur.left: stack.append(cur.left)
25         return res
26
27         # 4. 前序迭代2
28         res = []
29         stack = []
30         cur = root
31         while stack or cur:
32             while cur:
33                 res.append(cur.val) # 一直找找到最底下最左边, 同时也会加入根节点的值
34                 stack.append(cur)
35                 cur = cur.left
36             cur = stack.pop()
37             cur = cur.right
38         return res

```

前

同理

[右, 左]

[左, 左]

↑ P ↑ 右

In [ ]:

```

1 class Solution:
2     def inorderTraversal(self, root: TreeNode) -> List[int]:
3         if not root: return []
4         # 1. 中序递归!
5         return self.inorderTraversal(root.left) + [root.val] + self.inorderTraversal(root.right)
6
7         # 2. 中序递归2和1类似
8         def dfs(cur):
9             if not cur: return
10            dfs(cur.left)
11            res.append(cur.val)
12            dfs(cur.right)
13        res = []
14        dfs(root)
15        return res
16
17        # 3. 中序迭代1
18        res = []
19        stack = []
20        cur = root
21        while stack or cur:
22            while cur: # 一直找找到最底下最左边
23                stack.append(cur)
24                cur = cur.left
25            cur = stack.pop()
26            res.append(cur.val) # 遍历到叶子再加入节点的值
27            cur = cur.right
28        return res

```

中

[左, 左]

↑ P ↑ 右

In [ ]:

```

1 class Solution:
2     def postorderTraversal(self, root: TreeNode) -> List[int]:
3         if not root: return []
4         # 1. 后序递归!
5         return self.postorderTraversal(root.left) + self.postorderTraversal(root.right) + [root.val]
6
7         # 2. 后序递归2和1类似
8         def dfs(cur):
9             if not cur: return
10            dfs(cur.left)
11            dfs(cur.right)
12            res.append(cur.val)
13        res = []
14        dfs(root)
15        return res
16
17        # 3. 后序迭代1, 和前序差不多, 调整顺序后翻转res
18        stack = [root]; res = []
19        while stack:
20            cur = stack.pop()
21            if cur.left: stack.append(cur.left)
22            if cur.right: stack.append(cur.right) # 先打印左, 所以先放右
23            res.append(cur.val) # 同时也会加入根节点的值
24        return res[::-1]
25
26        # 4. 后序迭代2
27        res = []
28        stack = []
29        cur = root
30        while stack or cur:
31            while cur:
32                res.append(cur.val) # 一直找找到最底下最左边, 同时也会加入根节点的值
33                stack.append(cur)
34                cur = cur.right
35            cur = stack.pop()
36            cur = cur.left
37        return res[::-1]

```

后

[右, 左]

↑ P ↑ 左

[::-1]

In [3]:

```
1 # 144 前序
2 def preorderTraversal(root):
3     # 递归1 中左右
4     if root == None: return []
5     return [root.val]+self.preorderTraversal(root.left)+self.preorderTraversal(root.right)
6
7     # 递归二
8     res = []
9     def dfs(root):
10         if root == None : return
11         res.append(root.val)
12         if root.left: dfs(root.left)
13         if root.right:dfs(root.right)
14     dfs(root)
15     return res
16
17     # 迭代法
18     if not root: return []
19     res = []
20     stack = [root]
21     while stack:
22         tempNode = stack.pop()
23         res.append(tempNode.val)
24         if tempNode.right: stack.append(tempNode.right)
25         if tempNode.left: stack.append(tempNode.left)
26     return res
27
28 # 145 后序
29 def postorderTraversal(root):
30     # 递归1
31     if not root: return []
32     return self.postorderTraversal(root.left)+self.postorderTraversal(root.right)+[root.val]
33
34     # 递归2
35     res = []
36     def dfs(root):
37         if not root: return
38         if root.left: dfs(root.left)
39         if root.right: dfs(root.right)
40         res.append(root.val)
41     dfs(root)
42     return res
43
44     # 迭代1
45     if not root: return []
46     res = []
47     stack = [root]
48     while stack:
49         tempNode = stack.pop() # 取了就拿
50         res.append(tempNode.val)
51         if tempNode.left : stack.append(tempNode.left) # 栈概念
52         if tempNode.right: stack.append(tempNode.right)
53     return res[::-1] # 后向, 中右左-> 左右中
54
55 # 94 中序
56 def inorderTraversal(root):
57     # 递归1
```

```
58     if not root: return []
59     return self.inorderTraversal(root.left)+[root.val]+self.inorderTraversal(root.right)
60
61     # 递归2
62     res = []
63     def dfs(root):
64         if not root: return
65         if root.left:dfs(root.left)
66         res.append(root.val)
67         if root.right:dfs(root.right)
68     dfs(root)
69     return res
70
71     # 迭代 - 结合动态理解
72     if not root: return []
73     res = []
74     stack = []
75     cur = root
76     while stack or cur:
77         while cur:
78             stack.append(cur)
79             cur = cur.left # 左到底
80         cur = stack.pop() # 再取
81         res.append(cur.val)
82         cur = cur.right # 再右
83     return res
84
85 # 116 层序遍历
86 def levelOrder(root):
87     if not root: return []
88     queue = [root]
89     res = []
90     while queue:
91         temp = []
92         length = len(queue)
93         for i in range(length): # 用for不用另外开空间
94             tempNode = queue.pop(0)
95             temp.append(tempNode.val)
96             if tempNode.left: queue.append(tempNode.left)
97             if tempNode.right: queue.append(tempNode.right)
98         res.append(temp)
99     return res
100
101 # 层序遍历右指针
102 def connect(root):
103     if not root: return None
104     queue = [root]
105     while queue:
106         length = len(queue)
107         for i in range(length):
108             tempNode = queue.pop(0)
109             if tempNode.left: queue.append(tempNode.left)
110             if tempNode.right: queue.append(tempNode.right)
111             if i == length - 1: break # 结束一条链表
112             tempNode.next = queue[0] # 连接节点
113     return root
114
115 # 226 镜像二叉树
116 def invertTree(root):
117     if not root: return root
118     # 层次遍历
```

```
119     queue = [root]
120     while queue:
121         length = len(queue)
122         for i in range(length):
123             tempNode = queue.pop(0)
124             tempNode.left, tempNode.right = tempNode.right, tempNode.left # 只改变了一下
125             if tempNode.left: queue.append(tempNode.left)
126             if tempNode.right: queue.append(tempNode.right)
127     return root
128
129     # 递归(前序)
130     root.left, root.right = root.right, root.left
131     self.invertTree(root.left)
132     self.invertTree(root.right)
133
134     # 迭代(深度优先), 前序
135     stack = [root]
136     while stack:
137         node = stack.pop()
138         node.left, node.right = node.right, node.left # 只改变这一点
139         if node.right: stack.append(node.right)
140         if node.left: stack.append(node.left)
141     return root
142
143 # 227 对称二叉树
144 def isSymmetric(root):
145     if not root: return False
146
147     # 递归
148     def compare(left, right):
149         # 四种情况: 00, 10, 01, 11
150         if not left and not right: return True
151         if not left or not right or left.val != right.val: return False
152         return compare(left.left, right.right) and compare(left.right, right.left)
153     return compare(root.left, root.right)
154
155     # 队列
156     queue = [root.left, root.right]
157     while queue:
158         leftNode = queue.pop(0)
159         rightNode = queue.pop(0)
160         if not leftNode and not rightNode: continue
161         if not leftNode or not rightNode or leftNode.val != rightNode.val: return False
162         queue.append(leftNode.left)
163         queue.append(rightNode.right)
164         queue.append(leftNode.right)
165         queue.append(rightNode.left)
166     return True
167
168 # 104 二叉树的最大深度
169 def maxDepth(root):
170     if not root: return 0
171     # 递归
172     return 1 + max(self.maxDepth(root.left), self.maxDepth(root.right))
173
174     # 迭代
175     # 层序遍历模板
176     queue = [root]
177     res = 0
178     while queue:
179         length = len(queue)
```

```
180         res += 1 # 接下来会遍历每一层
181         for i in range(length):
182             tempNode = queue.pop(0)
183             if tempNode.left: queue.append(tempNode.left)
184             if tempNode.right: queue.append(tempNode.right)
185         return res
186
187 # 559 N叉树的最大深度——层序遍历
188 def maxDepth(self, root: 'Node') -> int:
189     # 递归
190     if not root: return 0
191     res = 0
192     for i in range(len(root.children)):
193         res = max(res, self.maxDepth(root.children[i]))
194     return 1+ res
195
196     # 迭代法
197     queue = [root]
198     res = 0
199     while queue:
200         res += 1
201         length = len(queue)
202         for i in range(length):
203             tempNode = queue.pop(0)
204             if tempNode.children: queue.extend(tempNode.children)
205     return res
206
207 # 111 二叉树的最小深度
208 def minDepth(root):
209     # 递归法
210     if not root: return 0
211     if root.left and not root.right: return 1+ self.minDepth(root.left)
212     if root.right and not root.left: return 1+ self.minDepth(root.right)
213     return 1+ min(self.minDepth(root.left), self.minDepth(root.right)) # 两个都非空
214
215     # 迭代法, 层序遍历
216     if not root: return 0
217     queue = [root]
218     res = 0
219     while queue:
220         length = len(queue)
221         res += 1
222         for i in range(length):
223             tempNode = queue.pop(0)
224             if tempNode.left: queue.append(tempNode.left)
225             if tempNode.right: queue.append(tempNode.right)
226             if tempNode.left == None and tempNode.right == None: return res #退出条件
227     return res
228
229 # 222 完全二叉树的根节点
230 def countNodes(root):
231     # 递归
232     if not root: return 0
233     return 1+self.countNodes(root.left)+self.countNodes(root.right)
234
235     # 普通二叉树的迭代法——层次遍历
236     queue = [root]
237     res = 0
238     while queue:
239         length = len(queue)
240         res += length
```

```

241         for i in range(length):
242             tempNode = queue.pop(0)
243             if tempNode.left: queue.append(tempNode.left)
244             if tempNode.right: queue.append(tempNode.right)
245     return res
246
247     # 利用完全二叉树的性质
248     def countDepth(root):
249         # 计算最大深度
250         r = 0
251         while root:
252             root = root.left
253             r += 1
254         return r
255     if not root: return 0
256     leftDepth = countDepth(root.left)
257     rightDepth = countDepth(root.right)
258     if leftDepth == rightDepth:
259         # 左满右完全
260         return 2**leftDepth + self.countNodes(root.right) # 不用减1, 因为加了根节点
261     else:
262         # 右满左完全
263         return 2**rightDepth + self.countNodes(root.left)
264
265     # 110 判断平衡二叉树
266     def isBalanced(root):
267         # 递归法
268         def getdepth(root):
269             # 返回平衡树的高度, 后续遍历
270             if not root: return 0
271             leftDepth = getdepth(root.left)
272             rightDepth = getdepth(root.right)
273             if leftDepth == -1 or rightDepth == -1: return -1 # 有一边不是平衡树了
274             return -1 if abs(leftDepth-rightDepth) > 1 else 1+max(leftDepth, rightDepth)
275         return getdepth(root) != -1
276
277     # 257 二叉树的所有路径
278     def binaryTreePaths(root):
279         # 回溯
280         if not root: return
281         res = []
282         path = [str(root.val)]
283         def backtrak(root):
284             # 结束条件 叶子节点
285             if not root: return
286             if not root.left and not root.right: res.append('->'.join(path[:])) # 叶子节点了
287             if root.left:
288                 path.append(str(root.left.val))
289                 backtrak(root.left)
290                 path.pop()
291             if root.right:
292                 path.append(str(root.right.val))
293                 backtrak(root.right)
294                 path.pop()
295         backtrak(root)
296         return res
297
298     # 100 相同的树
299     def isSameTree(p, q):
300         # 层次遍历 56 % 60, [1, 2][1, null, 2] 不过
301         # 二叉树镜像, 递归判断

```

```
302     if not p and not q: return True
303     if not p or not q : return False
304     if p.val != q.val: return False
305     return self.isSameTree(p.left,q.left) and self.isSameTree(p.right,q.right)
306
307 # 404 左叶子之和
308 def sumOfLeftLeaves(root):
309     # 递归
310     self.res = 0
311     def findleft(root):
312         if not root: return 0
313         # 不能if not left: return 0,这排除了有右子树的情况
314         if root.left and not root.left.left and not root.left.right:
315             self.res += root.left.val # 修改全局变量
316             findleft(root.left)
317             findleft(root.right)
318         findleft(root)
319     return self.res # 返回全局变量
320
321 # 513 最左边的叶子的值
322 def findBottomLeftValue(root):
323     # 层序遍历
324     queue = [root]
325     while queue:
326         length = len(queue)
327         for i in range(length):
328             tempNode = queue.pop(0)
329             if i == 0:
330                 temp = tempNode.val
331                 if tempNode.left: queue.append(tempNode.left)
332                 if tempNode.right: queue.append(tempNode.right)
333     return temp
334
335 # 112 路径总和
336 def hasPathSum(root, targetSum):
337     # 递归
338     if not root: return False
339     targetSum -= root.val
340     if not root.left and not root.right and targetSum == 0: return True
341     return self.hasPathSum(root.left,targetSum) or self.hasPathSum(root.right,targetSum)
342
343 # 113 路径总和II
344 def pathSum(root, targetSum):
345     if not root: return []
346     res = []
347     path = [root.val]
348     def backtrack(cur, count):
349         if not cur.left and not cur.right and count == 0: return res.append(path[:]) #
350         if not cur.left and not cur.right: return #
351         if cur.left:
352             path.append(cur.left.val) #
353             count -= cur.left.val
354             backtrack(cur.left, count)
355             path.pop() #
356             count += cur.left.val
357         if cur.right:
358             path.append(cur.right.val) #
359             count -= cur.right.val
360             backtrack(cur.right, count)
361             path.pop() #
362             count += cur.right.val
```

```
363     backtrack(root, targetSum-root.val)
364     return res
365
366 # 106 中序和后续构造二叉树
367 def buildTree(inorder, postorder):
368     # 没有重复元素
369     if not len(inorder) or not len(postorder): return
370     rootval = postorder.pop()
371     root = TreeNode(rootval)
372     splitin = inorder.index(rootval)
373     inleft = inorder[:splitin]
374     inright = inorder[splitin+1:]
375     postleft = postorder[:splitin]
376     postright = postorder[splitin:]
377     root.left = self.buildTree(inleft, postleft)
378     root.right = self.buildTree(inright, postright)
379     return root
380
381 # 105 前序和中序构造二叉树
382 def buildTree(preorder, inorder):
383     if not len(inorder) or not len(preorder): return
384     rootval = preorder.pop(0)
385     root = TreeNode(rootval)
386     splitpoint = inorder.index(rootval)
387     root.left = self.buildTree(preorder[:splitpoint], inorder[:splitpoint])
388     root.right = self.buildTree(preorder[splitpoint:], inorder[splitpoint+1:])
389     return root
390
391 # 654 最大的二叉树
392 def constructMaximumBinaryTree(nums):
393     if not nums: return
394     rootval = max(nums)
395     root = TreeNode(rootval)
396     maxpoint = nums.index(rootval)
397     leftnums = nums[:maxpoint]
398     rightnums = nums[maxpoint+1:]
399     root.left = self.constructMaximumBinaryTree(leftnums)
400     root.right = self.constructMaximumBinaryTree(rightnums)
401     return root
402
403 # 617 合并二叉树
404 def mergeTrees(root1, root2):
405     if not root1: return root2
406     if not root2: return root1
407     if not root1 and not root2: return
408     rootval = root1.val+ root2.val
409     root = TreeNode(rootval)
410     root.left = self.mergeTrees(root1.left, root2.left)
411     root.right = self.mergeTrees(root1.right, root2.right)
412     return root
413
414 # 700 二叉搜索树中搜索
415 def searchBST(root, val):
416     if not root: return
417     if root.val == val: return root
418     # return self.searchBST(root.left, val) or self.searchBST(root.right, val)
419     # 左子树所有节点均小于根节点
420     # 右子树所有节点均大于根节点
421     if root.val > val: return self.searchBST(root.left, val)
422     if root.val < val: return self.searchBST(root.right, val)
423     return None
```



```
424
425     # 迭代
426     while root:
427         if root.val == val: return root
428         if root.val < val: root = root.right
429         else: root = root.left
430     return
431
432 # 98 验证二叉树
433 def isValidBST(root):
434     # 中序遍历输出有序数组
435     if not root: return True
436     pre = float("-inf") # 保存前一个访问节点的值
437     def traversal(root):
438         nonlocal pre
439         if not root: return True
440         if not traversal(root.left): return False
441         if root.val <= pre: return False
442         else: pre = root.val
443         if not traversal(root.right): return False
444         return True
445     return traversal(root)
446
447 # 纯递归 + 判断数值
448 if not root: return
449 def traversal(root):
450     if not root: return []
451     return traversal(root.left) + [root.val] + traversal(root.right)
452 res = traversal(root)
453 for i in range(1, len(res)):
454     if res[i] <= res[i-1]: return False
455 return True
456
457 # 迭代中序遍历
458 if not root: return True
459 stack = []
460 pre = float('-inf') # 保存前一个访问节点值
461 cur = root
462 while stack or cur:
463     while cur:
464         stack.append(cur)
465         cur = cur.left
466     cur = stack.pop()
467     if cur.val <= pre: return False
468     else: pre = cur.val
469     cur = cur.right
470 return True
471
472 # 530 二叉搜索树的最小绝对差
473 def getMinimumDifference(root):
474     # 递归
475     res = float('inf')
476     pre = float('-inf')
477     def traversal(root):
478         nonlocal res, pre
479         if not root: return
480         traversal(root.left)
481         res = min(res, root.val - pre)
482         pre = root.val
483         traversal(root.right)
484     traversal(root)
```

```

485     return res
486
487     # 迭代
488     res = float('inf')
489     pre = float('-inf')
490     stack = []
491     cur = root
492     while stack or cur:
493         while cur:
494             stack.append(cur)
495             cur = cur.left
496         cur = stack.pop()
497         res = min(res, cur.val - pre)
498         pre = cur.val
499         cur = cur.right
500     return res
501
502 # 501 二叉搜索树的众数
503 def findMode(self, root: TreeNode) -> List[int]:
504     # 普特二叉树对待
505     dic = {}
506     def search(root):
507         if not root: return
508         if root.val in dic: dic[root.val] += 1
509         else: dic[root.val] = 1
510         search(root.left)
511         search(root.right)
512     search(root)
513     sdict = sorted(dic.items(), key = lambda x:x[1], reverse = True) # 排序字典
514     res = []
515     for key, val in dic.items():
516         if val == sdict[0][1]:
517             res.append(key)
518     return res
519
520 # 二叉搜索树+迭代遍历
521 res = [] # 保存结果
522 maxCount = 0 # 统计最大频率
523 count = 0 # 统计当前频率
524 stack = []
525 cur = root
526 pre = float('-inf')
527 while stack or cur:
528     while cur:
529         stack.append(cur)
530         cur = cur.left
531     cur = stack.pop()
532     print(res, maxCount)
533     if cur.val == pre: count += 1
534     else: count = 1
535     if count == maxCount:
536         res.append(cur.val)
537     if count > maxCount:
538         maxCount = count
539         res = [cur.val] # 放弃之前所有元素
540     pre = cur.val # 记录前面的值
541     cur = cur.right
542 return res
543
544 # 236 二叉树的最近公共祖先
545 def lowestCommonAncestor(root, p, q):

```

```
546 # 从下往上递归
547 if not root or root == q or root == p: return root
548 left = self.lowestCommonAncestor(root.left, p, q)
549 right = self.lowestCommonAncestor(root.right, p, q)
550 if left and right: return root # 有返回值，只有一个两边都有，后面都是一个有一个没有，相对
551 if not left and right : return right
552 elif left and not right : return left
553 else: return
554
555 # 235 二叉搜索树的最近公共祖先
556 def lowestCommonAncestor(root, p, q):
557     if root.val > p.val and root.val > q.val:
558         return self.lowestCommonAncestor(root.left, p, q) # 在左边
559     elif root.val < p.val and root.val < q.val:
560         return self.lowestCommonAncestor(root.right, p, q) # 在右边
561     else: return root # 找到了，回传
562
563 # 701 二叉搜索树的插入
564 def insertIntoBST(root, val):
565     if not root:
566         # 能走到none这个位置
567         node = TreeNode(val)
568         return node
569     if root.val > val : root.left = self.insertIntoBST(root.left, val)
570     if root.val < val: root.right = self.insertIntoBST(root.right, val)
571     return root
572
573 # 450 删除二叉树的节点
574 def deleteNode(root, key):
575     if not root: return root
576     if root.val == key:
577         if not root.left: return root.right
578         elif not root.right: return root.left # 已经包含都为空情况了
579         else:
580             cur = root.right
581             while cur.left:
582                 cur = cur.left
583             cur.left = root.left
584             temp = root
585             root = root.right
586             del temp
587             return root
588     if root.val > key: root.left = self.deleteNode(root.left, key)
589     if root.val < key: root.right = self.deleteNode(root.right, key)
590     return root
591
592 # 669 修剪二叉搜索树
593 def trimBST(root, low, high):
594     if not root : return
595     if root.val < low:
596         right = self.trimBST(root.right, low, high)
597         return right
598     if root.val > high:
599         left = self.trimBST(root.left, low, high)
600         return left
601     root.left = self.trimBST(root.left, low, high)
602     root.right = self.trimBST(root.right, low, high)
603     return root
604
605 # 108 构建一课二叉搜索树
606 def sortedArrayToBST(nums):
```

```

607     def traversal(nums, left, right):
608         if left > right: return None
609         mid = left + (right-left) // 2
610         root = TreeNode(nums[mid])
611         root.left = traversal(nums, left, mid-1) # 不能是nums[:mid]
612         root.right = traversal(nums, mid+1, right)
613         return root
614     root = traversal(nums, 0, len(nums)-1)
615     return root
616
617 # 538 把二叉搜索树转换成累加树
618 def convertBST(root):
619     # 递归 反中序遍历
620     pre = 0
621     def traversal(cur):
622         nonlocal pre
623         if not cur: return
624         traversal(cur.right)
625         cur.val += pre
626         pre = cur.val
627         traversal(cur.left)
628     traversal(root)
629     return root
630     # 迭代
631     if not root: return
632     pre = 0
633     stack = []
634     cur = root
635     while stack or cur:
636         while cur:
637             stack.append(cur)
638             cur = cur.right
639         cur = stack.pop()
640         cur.val += pre # 这两行
641         pre = cur.val # 不同而已
642         cur = cur.left
643     return root

```

In [140]:

```

1 # 698 桶装法
2 def canPartitionKSubsets(nums, k):
3     # 每个子集的和为 nums / k
4     if sum(nums) % k != 0: return False
5     Sum = int(sum(nums) / k)
6     bucker = [0]*k # k个桶
7     def backtrack(nums, startIndex, k):
8         nonlocal Sum
9         if startIndex == len(nums) : return True
10        for i in range(k):
11            if bucker[i] + nums[startIndex] <= Sum:
12                bucker[i] += nums[startIndex]
13                if backtrack(nums, startIndex+1, k): return True
14                bucker[i] -= nums[startIndex]
15                if bucker[i] == 0: return False
16        return False
17    nums.sort(reverse=True)
18    return backtrack(nums, 0, k)

```