1. 题目

LC46.全排列

backtracking, https://leetcode.cn/problems/permutations/

思路：使用了回溯法，每次递归都加入一个元素并将对应位置标为 True，当到最后一个元素时加入 ans 最终列表，并将最后一个元素弹出，对应位置标为 False，继续递归还未使用的元素。

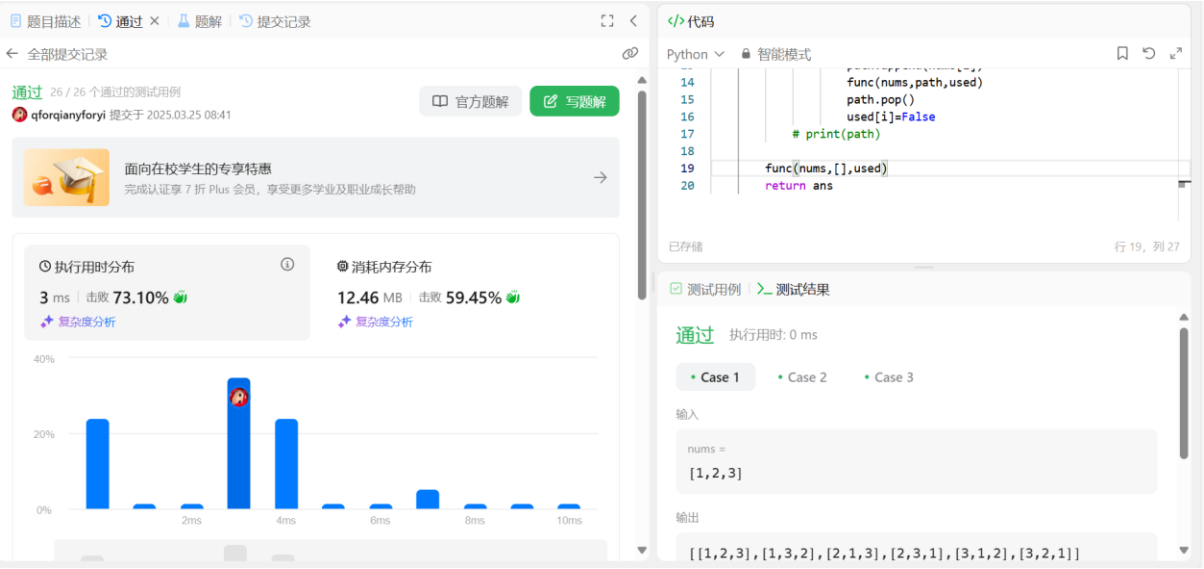
代码：

```
class Solution(object):
    def permute(self, nums):
        used = [False] * len(nums)
        ans = []
        def func(nums,path,used):
            if len(path)==len(nums):
                ans.append(path[:])
                return


            for i in range(len(nums)):
                if used[i] is False:
                    used[i]=True
                    path.append(nums[i])
                    func(nums,path,used)
                    path.pop()
                    used[i]=False
            # print(path)
```

func(nums,[],used)

return ans

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：50 分钟

LC79: 单词搜索

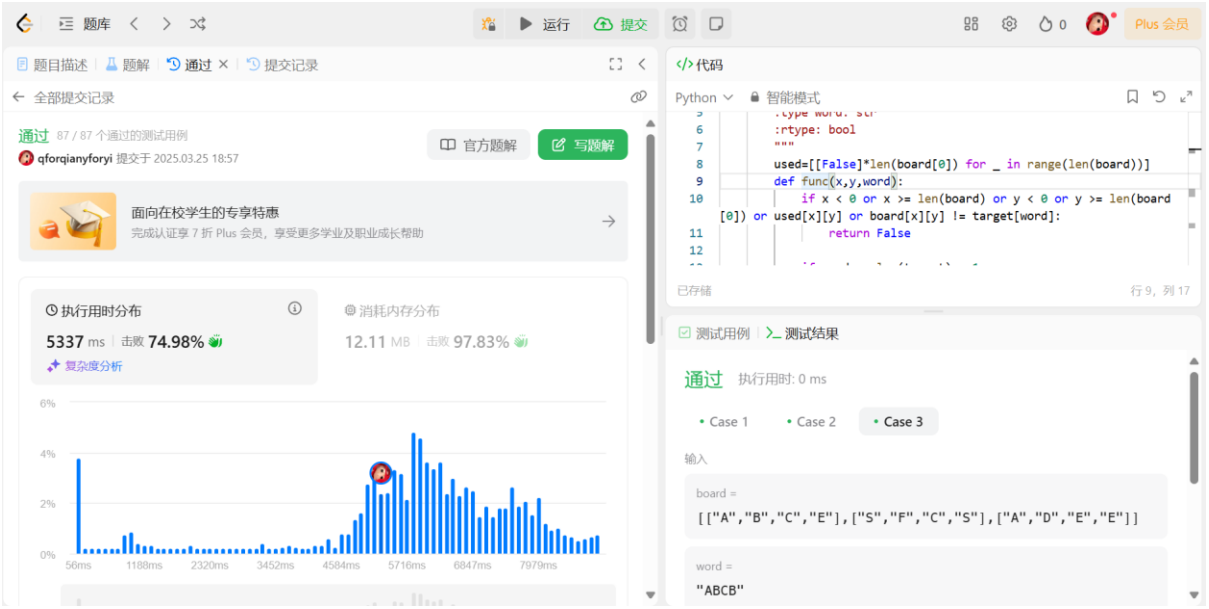
backtracking, https://leetcode.cn/problems/word-search/

思路：遍历整个 board 检查当前位置是否越界、是否已经被使用过，以及当前位置的字符是否与要找的字中对应索引的字符匹配，不满足条件则返回 False；当匹配到最后一个字符时则返回 True。每次将当前位置标记为 True 表示已经使用，并用递归 func 函数搜索上下左右四方向，只要其中一个能匹配成功则返回 True，递归结束后将当前位置标记为 False（未使用），以便后续能再次检查该位置。

代码：

```
class Solution(object):
    def exist(self, board, target):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """
        used=[[False]*len(board[0]) for _ in range(len(board))]
        def func(x,y,word):
            if x < 0 or x >= len(board) or y < 0 or y >= len(board[0]) or used[x][y] or board[x][y] != target[word]:
                return False

            if word == len(target) - 1:
                return True

            used[x][y] = True
            direction=(func(x+1,y,word+1) or func(x-1,y,word+1) or func(x, y + 1,word+1) or func(x, y - 1,word+1))
            used[x][y]=False
            return direction

        a,b=len(board),len(board[0])
        for i in range(a):
            for j in range(b):
                if func(i,j,0):
                    return True
        return False
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：1 小时

LC94.二叉树的中序遍历

dfs, https://leetcode.cn/problems/binary-tree-inorder-traversal/

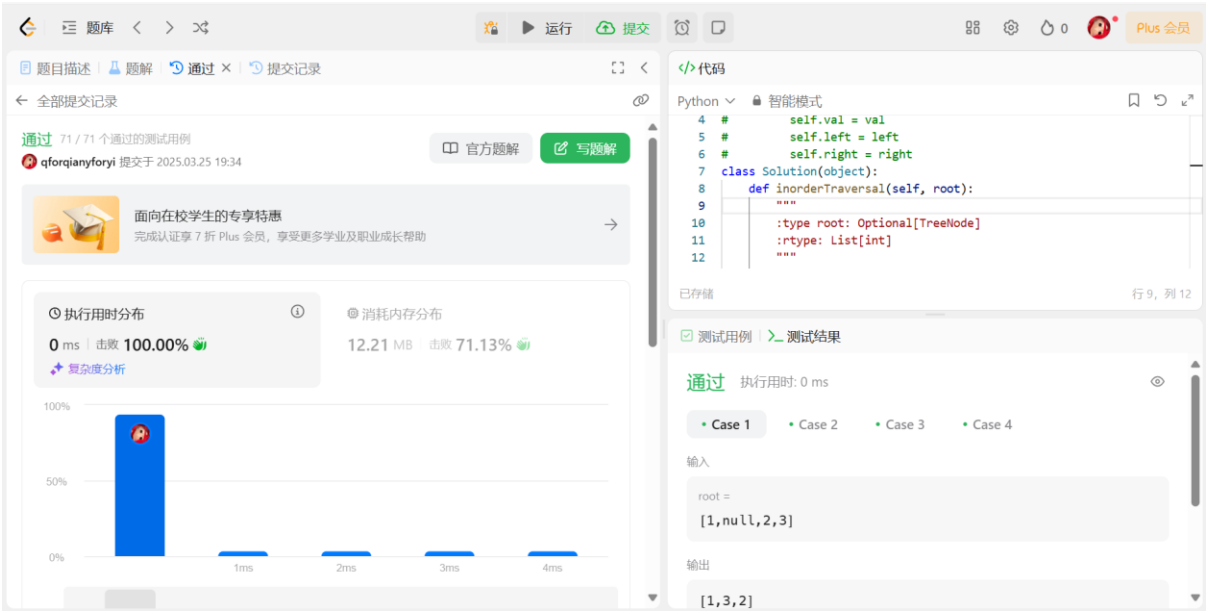
思路：回溯法先逐层遍历左子树，如果为 None 则将当前的节点加入 ans 并进入右子树再次逐层遍历。

代码：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
```

```
#         self.right = right
class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: List[int]
        """
        ans=[]
        def func(root):
            if not root:
                return
            func(root.left)
            ans.append(root.val)
            func(root.right)
        func(root)
        return ans
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：30 分钟

LC102.二叉树的层序遍历

bfs, https://leetcode.cn/problems/binary-tree-level-order-traversal/

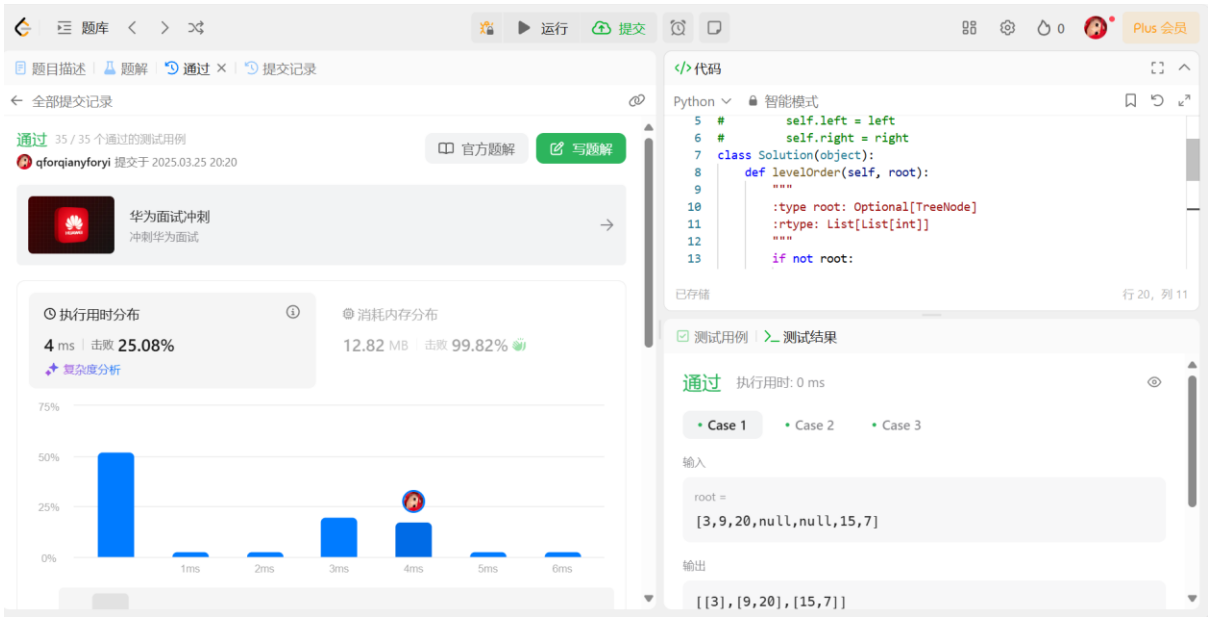
思路：使用了 bfs 和队列方法，首先用队列储存根节点，之后循环遍历当前层的所有节点并存到 res 列表中（仅记录当前层），之后逐层检查左子树和右子树并将节点加入队列中，最后将当前层的节点列表 res 存入最终列表 ans。

代码：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
```

```
def levelOrder(self, root):
    """
    :type root: Optional[TreeNode]
    :rtype: List[List[int]]
    """
    if not root:
        return []
    queue=deque([root])
    ans=[]
    while queue:
        res=[]
        for _ in range(len(queue)):
            node=queue.popleft()
            res.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        ans.append(res)
    return ans
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：45 分钟

LC131.分割回文串

dp, backtracking, https://leetcode.cn/problems/palindrome-partitioning/

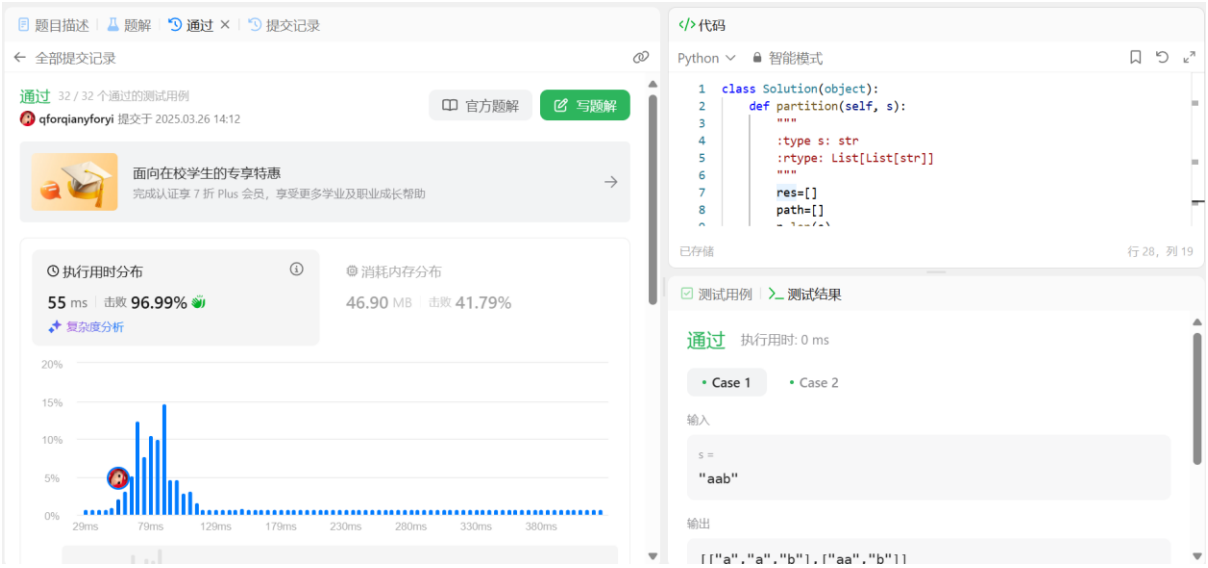
思路：首先用动态规划来判断回文子串（s[left] == s[right]判断首位是否相同，当单字符和两个字符是则 right-left<=1，本身就是回文；而 dp[left+1][right-1]用于判断更长的子串，如果 True 则是回文），再用回溯来遍历所有切割回文子串的方案（从 0 开始遍历，如果是回文就加入 path 列表，当遍历到末尾代表找到一种方案，将其记录到 res 中，然后用 pop()回撤，递归处理剩余部分。

代码：

```
class Solution(object):
    def partition(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
```

```
        """
        res=[]
        path=[]
        n=len(s)

        dp=[[False]*n for _ in range(n)]
        for right in range(n):
            for left in range(right+1):
                if s[left] == s[right] and (right-left <= 1 or dp[left+1][right-1]):
                    dp[left][right]=True

        def dfs(start):
            if start == n:
                res.append(path[:])
                return

            for end in range(start,n):
                if dp[start][end]:
                    path.append(s[start:end+1])
                    dfs(end+1)
                    path.pop()
        dfs(0)
        return res
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：1 小时 30 分钟

LC146.LRU 缓存

hash table, doubly-linked list, https://leetcode.cn/problems/lru-cache/

思路：用哈希表来查询数据，用双向链表更新数据的使用顺序，最近使用的项在头部，最久未使用的项在尾部。需要定义 get 和 put 操作，get 操作用于查找哈希表中的 key，如果存在返回对应的 value，否则返回-1（找不到），同时将该项移到双向链表的尾部；put 操作用于插入新的 key-value 对。如果 key 已经存在，更新其 value，并将其移到尾部，如果缓存满了，移除头部的最久未使用项；还要定义将节点移到尾部的函数，表示当访问一个节点时将其移到尾部（最近访问）。

代码：

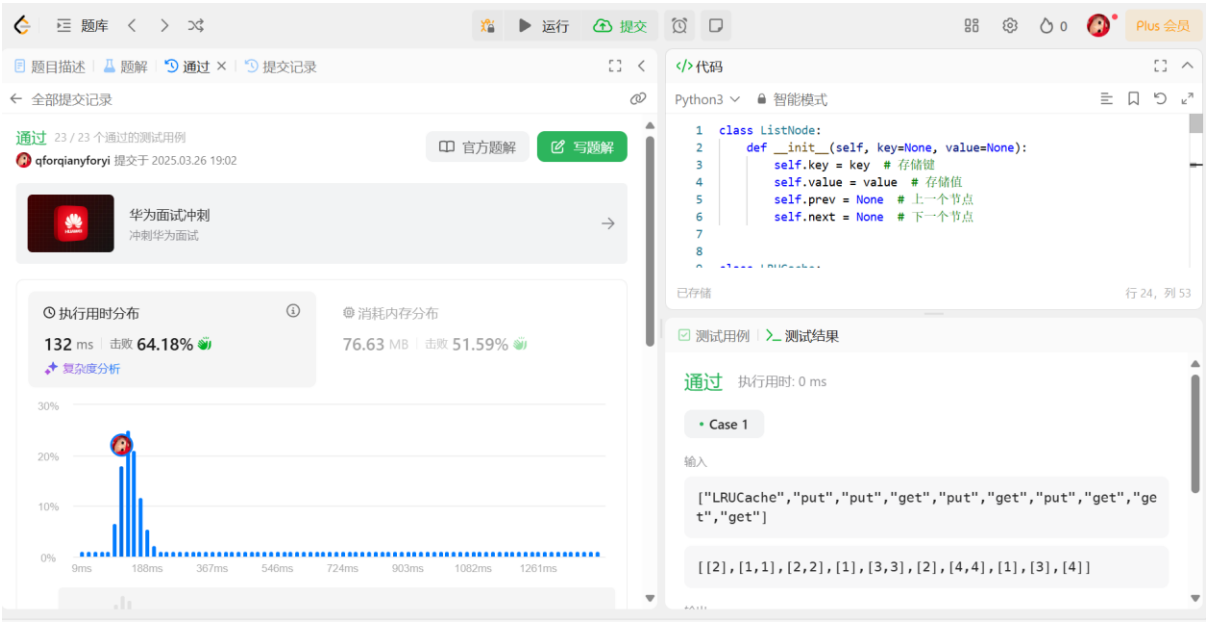
```
class ListNode:
    def __init__(self, key=None, value=None):
        self.key = key  # 存储键
        self.value = value  # 存储值
        self.prev = None  # 上一个节点
```

```
        self.next = None  # 下一个节点


class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity  # 缓存容量
        self.hashmap = {} # 哈希表用于存储 key 到 ListNode 的映射
        # 新建两个节点 head 头节点 和 tail 尾节点
        self.head = ListNode()
        self.tail = ListNode()
        # 初始化链表为 head <-> tail
        self.head.next = self.tail
        self.tail.prev = self.head
        # 头节点的 next 指向尾节点，尾节点的 prev 指向头节点，构成空（双向）链表


    # 因为 get 与 put 操作都可能需要将双向链表中的某个节点移到末尾，所以定义一个方法
    def move_node_to_tail(self, key):
            node = self.hashmap[key] # 获取哈希表中对应 key 的节点
            node.prev.next = node.next  # 先把节点从链表中移除
            node.next.prev = node.prev
            # 然后把节点插入到链表的尾部
            node.prev = self.tail.prev
            node.next = self.tail
            self.tail.prev.next = node
            self.tail.prev = node


    def get(self, key: int) -> int:
```

```
        if key in self.hashmap:
            # 如果已经在链表中了久把它移到末尾（变成最新访问的）
            self.move_node_to_tail(key)# 如果 key 在缓存中，移动到尾部（最近使用）
        res = self.hashmap.get(key, -1)  # 查找 key 对应的节点
        if res == -1:
            return res  # 如果找不到，返回 -1
        else:
            return res.value  # 如果找到，返回节点的值


    def put(self, key: int, value: int) -> None:
        if key in self.hashmap:
            # 如果 key 已经在缓存中，更新其 value，并移到尾部（如果 key 本身已经在哈希表中了就不需要在链表中加入新的节点)
            # 但是需要更新字典该值对应节点的 value
            self.hashmap[key].value = value
            # 之后将该节点移到末尾
            self.move_node_to_tail(key)
        else:
            if len(self.hashmap) == self.capacity:
                # 如果缓存满了，移除最久未使用的节点（即头节点的下一个节点）
                # 去掉哈希表对应项
                self.hashmap.pop(self.head.next.key)
                # 去掉最久没有被访问过的节点，即头节点之后的节点（删除哈希表中的最旧的节点）
                self.head.next = self.head.next.next # 从链表中删除最旧节点
                self.head.next.prev = self.head
            # 如果不在的话就插入到尾节点前
```

```
# 插入新的 key-value
new = ListNode(key, value)
self.hashmap[key] = new # 在哈希表中添加新节点
new.prev = self.tail.prev # 将新节点插入到尾部前面
new.next = self.tail
self.tail.prev.next = new
self.tail.prev = new
```

代码运行截图 <mark>（至少包含有"Accepted"）</mark>



大约用时：2 小时

2. 学习总结和收获

<mark>如果发现作业题目相对简单，有否寻找额外的练习题目，如“数算 2025spring 每日选做”、LeetCode、Codeforces、洛谷等网站上的题目。</mark>

一直想练习回溯，但是找的题目不是太难就是看不懂，这次因为做作业原因就算看不懂也做出来了 hhh，算是有所收获。第三题的链表稍微了解了一下发现并不难，还挺好理解的。第四题的 bfs 虽然知道有这个方法但之前好像没都写过，这次写了感觉在层次遍历上真的很方便，而且也顺便复习了队列~第五题的 dp+回溯看了题解，其中觉得 dp 真的好巧妙，但是自己写大概率写不出，之后打算找点简单题来练一练！第六题好难…完全看不懂，对我来说超纲了，只能看题解然后疯狂问 AI（还好用的是 leetcode 不然找题解可能都要我几个小时），看了几个小时终于把整个框架理解了 TvT，但是对各种节点的赋值（.next.prev.tail.....）还是很混乱，这回看懂了但是换个情况可能就看不懂了。