

全排列、子集、分割回文串的递归回溯写法

必须掌握

Updated 2025-11-01 12:39 GMT+8

Compiled by Hongfei Yan (2025 Fall)

递归与回溯是算法中的核心思想之一，尤其在解决组合、排列、子集等问题时至关重要。如果你目前尚未完全理解其原理，建议先熟记经典模板，通过反复运行代码、调试过程、结合视频讲解来逐步深入理解。

“递归是思想，回溯是技巧；先走到底，再退回来。”

掌握这经典问题，你就迈出了通向回溯算法的第一步！

M46. 全排列

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

给定一个不含重复数字的数组 `nums`，返回其 所有可能的全排列。你可以 按任意顺序 返回答案。

示例 1：

```
1 | 输入: nums = [1,2,3]
2 | 输出: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

示例 2：

```
1 | 输入: nums = [0,1]
2 | 输出: [[0,1],[1,0]]
```

示例 3：

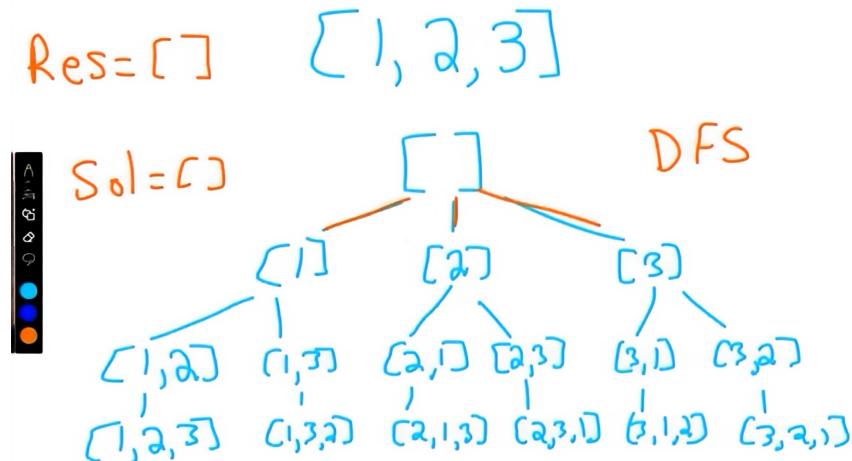
```
1 | 输入: nums = [1]
2 | 输出: [[1]]
```

提示：

- `1 <= nums.length <= 6`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有整数 互不相同

思路：递归 + 回溯

使用一个临时路径 `sol` 记录当前排列，通过遍历原数组并跳过已选元素的方式进行搜索。当路径长度等于数组长度时，将当前排列加入结果集。



```
1 class Solution:
2     def permute(self, nums: List[int]) -> List[List[int]]:
3         n = len(nums)
4         ans, sol = [], []
5
6         def backtrack():
7             # 终止条件：当前排列已满
8             if len(sol) == n:
9                 ans.append(sol[:]) # 深拷贝
10                return
11
12             # 尝试每个未被使用的数
13             for x in nums:
14                 if x not in sol: # 剪枝：避免重复使用
15                     sol.append(x) # 选择
16                     backtrack() # 递归
17                     sol.pop() # 回溯
18
19             backtrack()
20         return ans
```

全排列视频讲解：<https://pku.instructuremedia.com/embed/c76751c9-bc0e-49f1-8a99-624b955de668>

M78.子集

backtracking, <https://leetcode.cn/problems/subsets/>

给你一个整数数组 `nums`，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

示例 1:

```
1 | 输入: nums = [1,2,3]
2 | 输出: [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

示例 2:

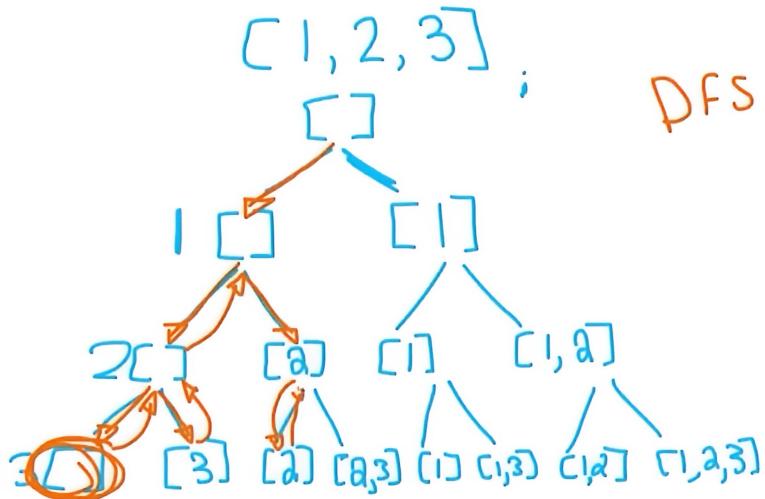
```
1 | 输入: nums = [0]
2 | 输出: [[], [0]]
```

提示:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有元素 互不相同

思路：递归回溯（选或不选）

对每个元素有两种选择：选入子集 或 不选入子集。递归遍历所有位置，到达末尾时将当前路径加入结果。



```
1 class Solution:
2     def subsets(self, nums: List[int]) -> List[List[int]]:
3         n = len(nums)
4         ans, sol = [], []
5
6         def backtrack(i):
# 终止条件：处理完所有元素
```

```

8     if i == n:
9         ans.append(sol[:])
10        return
11
12    # 分支1: 不选择 nums[i]
13    backtrack(i + 1)
14
15    # 分支2: 选择 nums[i]
16    sol.append(nums[i])
17    backtrack(i + 1)
18    sol.pop()  # 回溯
19
20    backtrack(0)
21    return ans

```

子集视频讲解: <https://pku.instructuremedia.com/embed/d8ccd717-3664-41bc-85d2-7170f348327b>

总结对比

问题	决策方式	终止条件	是否需要去重	时间复杂度
全排列	从剩余元素中选择	路径长度 = n	是 (避免重复使用)	$O(n \times n!)$
子集	每个元素选/不选	索引到达数组末尾	否 (天然无重)	$O(2^n \times n)$

⚠ 注意: 由于每次添加路径都需要复制 `sol[:]`, 因此总时间复杂度中乘以 `n`。

M131.分割回文串

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

给你一个字符串 `s`, 请你将 `s` 分割成一些子串, 使每个子串都是回文串。返回 `s` 所有可能的分割方案。回文串是指向前和向后读都相同的字符串。

示例 1:

```

1 | 输入: s = "aab"
2 | 输出: [[ "a" , "a" , "b" ], [ "aa" , "b" ]]

```

示例 2:

```

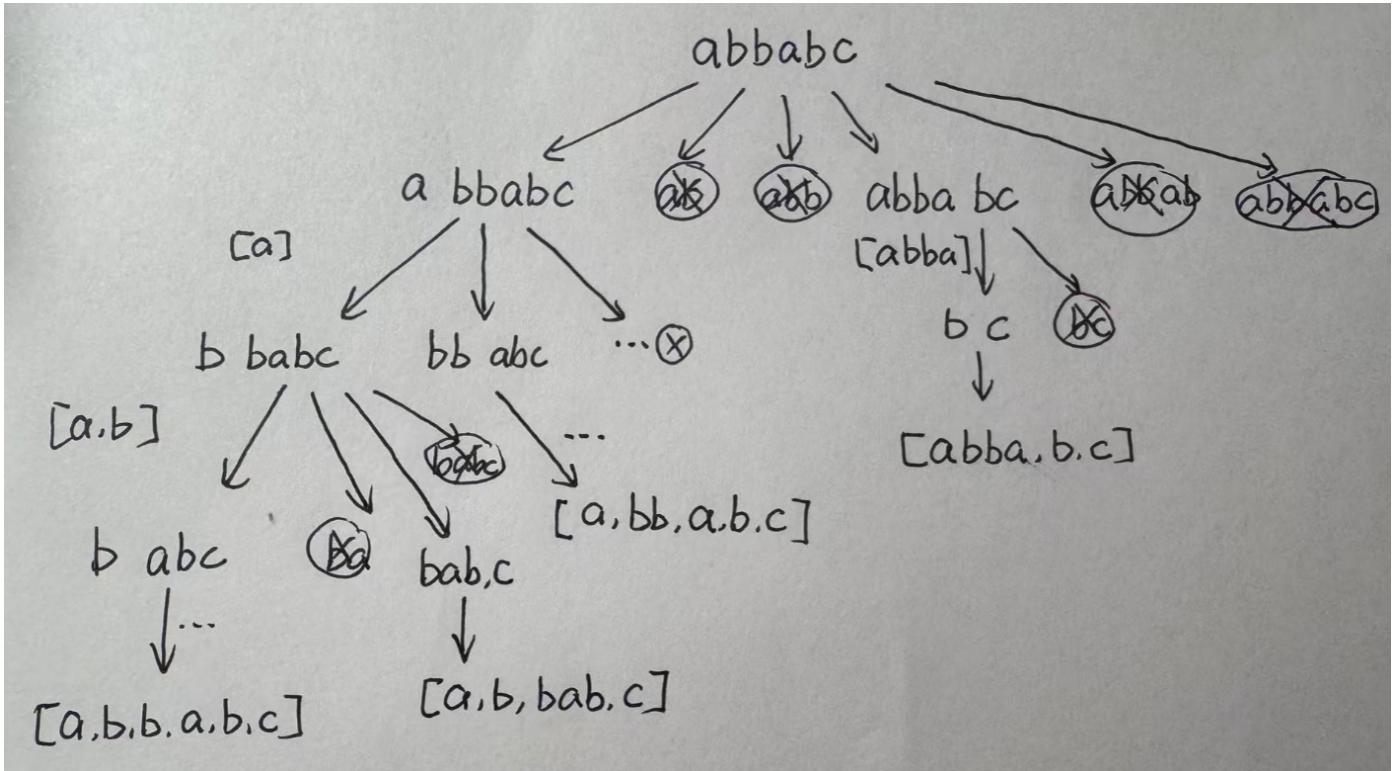
1 | 输入: s = "a"
2 | 输出: [[ "a" ]]

```

提示：

- $1 \leq s.length \leq 16$
- s 仅由小写英文字母组成

【陈林鑫 物理学院】思路：



如图所示，对于一个字符串 s ，依次判断从 $i=1$ 到 $i=\text{len}(s)+1$ ， $s[0:i]$ 是否为回文串，如果是，则在 i 处分割，前半部分为回文串，将它计入这条递归的列表 resi 中，剩下的部分 $s[i:]$ 则继续分割。如果剩下的字符串 $s[i:]$ 长度为 0，则说明分割完毕，返回 resi 。for 循环可以遍历所有情形。

如果字符串较长，可以使用 LRU 缓存递归判断（不建 DP 表）

```
1 from functools import lru_cache
2
3 class Solution:
4     def partition(self, s: str) -> List[List[str]]:
5         n = len(s)
6         ans = []
7         @lru_cache(None)
8         def is_pal(i, j):
9             return i >= j or (s[i] == s[j] and is_pal(i + 1, j - 1))
10
11     def dfs(start, path):
12         if start == n:
13             ans.append(path[:])
14             return
15         for end in range(start, n):
16             if is_pal(start, end):
```

```
17         path.append(s[start:end + 1])
18         dfs(end + 1, path)
19         path.pop()
20     dfs(0, [])
21
22
```