入选理由

1.昨天题目的进阶，变了条件你还会么？

标签

- 回溯
- 剪枝

难度

- 中等

题目地址（47 全排列 II）

https://leetcode-cn.com/problems/permutations-ii/

题目描述

```
给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例:

输入: [1,1,2]
输出:
[
[1,1,2],
[1,2,1],
[2,1,1]
```

前置知识

- 回溯
- 数组
- 剪枝

思路

其实这个题应该和 46. 全排列做对比，因为 46 和 47 和前两天的 39，40 很相似的，46 题大家有兴趣也可以去自己做。

该题的题干很简单，就是让我输出所有不重复的全排列，为什么全排列需要强调不重复，因为可能含有重复元素。分析：

- 得到的每个全排列都是由数组中所有元素构成的且每个元素只出现一次，因此和前两天得不一样，反而不能去限制顺序（避免剪掉可行解），那么我们就需要一个辅助数组 visit 来避免重复使用元素。
- 如何去重的：其实和昨天的也很像，如果单纯用 set 则复杂度过高。
- 下面简单说一下两种剪掉重复解的方式：假设数据是[1,1,2]（这里注意要给数组先排序再 dfs）
    - nums[i] == nums[i - 1] && visit[i - 1]： 该种情况是优先取右，举个简单例子，第一个我们从左到右是 1，1，2，这种情况是不可取的，因为当到第二个 1 时候，第一个 1 已经用过了，正相反，当我们从第二个 1 开始的时候,取第二个数也就是 nums[0]=1 还没用过，符合条件，故两个 1，1，2 只会存下来 1 个。
    - nums[i] == nums[i - 1] && !visit[i - 1]： 这个跟第一种过滤方式刚好相反，不过多解释。

代码

代码支持：Java

Java Code:

```java
public List<List<Integer>> permuteUnique(int[] nums) {

    List<List<Integer>> res = new ArrayList<>();

    if (nums == null || nums.length == 0)
        return res;

    boolean[] visited = new boolean[nums.length];
    Arrays.sort(nums);
    dfs(nums, res, new ArrayList<Integer>(), visited);

    return res;
}

public void dfs(int[] nums, List<List<Integer>> res, List<Integer> tmp, boolean[] visited) {

    if (tmp.size() == nums.length) {

        res.add(new ArrayList(tmp));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
```

```
        if (i > 0 && nums[i] == nums[i - 1] && visited[i - 1])
            continue;

        //backtracking
        if (!visited[i]) {

            visited[i] = true;
            tmp.add(nums[i]);
            dfs(nums, res, tmp, visited);
            visited[i] = false;
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

JS Code:

```
function backtrack(list, nums, tempList, visited) {
  if (tempList.length === nums.length) return list.push([...tempList]);
  for (let i = 0; i < nums.length; i++) {
    if (visited[i]) continue;
    // visited[i - 1] 容易忽略
    if (i > 0 && nums[i] === nums[i - 1] && visited[i - 1]) continue;

    visited[i] = true;
    tempList.push(nums[i]);
    backtrack(list, nums, tempList, visited);
    visited[i] = false;
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permuteUnique = function (nums) {
  const list = [];
  backtrack(
    list,
    nums.sort((a, b) => a - b),
    [],
    []
  );
  return list;
};
```

Python Code:

```
```

```python
class Solution:
    def backtrack(numbers, pre):
        nonlocal res
        if len(numbers) <= 1:
            res.append(pre + numbers)
            return
        for key, value in enumerate(numbers):
            if value not in numbers[:key]:
                backtrack(numbers[:key] + numbers[key + 1:], pre+[value])

    res = []
    if not len(nums): return []
    backtrack(nums, [])
    return res
```

复杂度分析

- 时间复杂度：由于由 visit 数组的控制使得每递归一次深度-1，因此递归的时间复杂度是$N * (N - 1) * ... * 1$也就是$O(N! * op(res))$，其中 N 是数组长度，op 操作即是昨天说的 res.add 算子的时间复杂度。

- 空间复杂度：$O(N * N!)$，考虑数组 N 个不重复元素，每一个排列占$O(N)$，共有 N!个排列。

上一页 下一页