LeetCode 例题精讲 | 09 排列组合问题再探：回溯法的去重策略

原创 nettee 面向大象编程 3月15日

来自专辑

LeetCode 例题精讲

本期例题（共两道）：

例题一，存在重复元素的子集问题：**LeetCode 90 - Subsets II**[1]（Medium）

> 给定一组**可能包含重复元素**的整数 *nums*，返回所有可能的子集（幂集）。例如：
>
> - 输入： `nums = [1,2,3]`
> - 输出：
>
> ```
> [
>  []
>  [1],
>  [2],
>  [1,2],
>  [2,2],
>  [1,2,2],
> ]
> ```

例题二，存在重复元素的全排列问题：**LeetCode 47 - Permutations II**[2]（Medium）

> 给定一个**可能重复**的整数集合，返回其所有可能的全排列。例如：
>
> - 输入： `[1, 2, 2]`
> - 输出：
>
> ```
> [
>   [1,1,2],
>   [1,2,1],
>   [2,1,1],
> ]
> ```

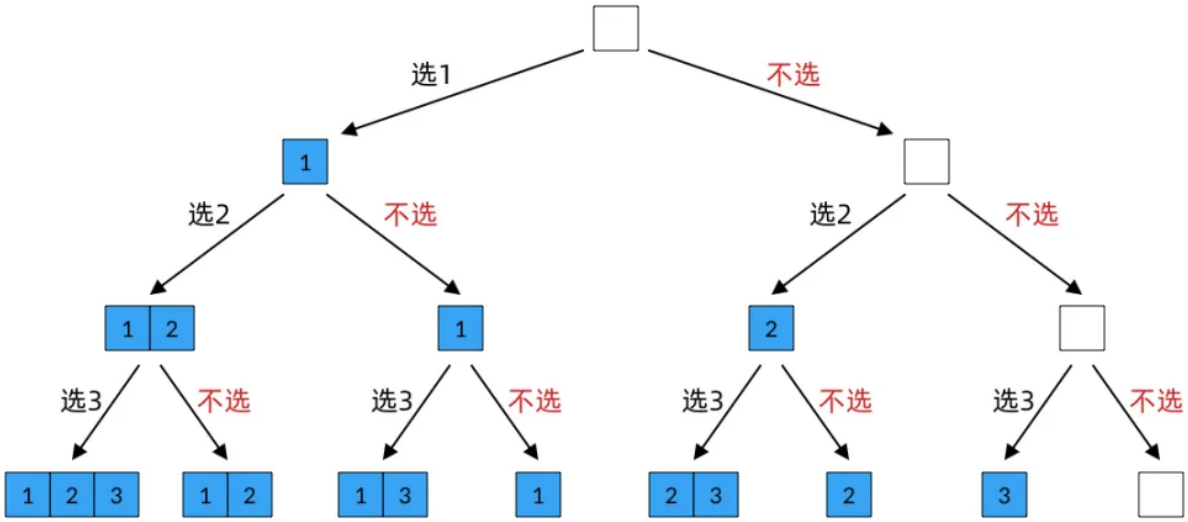
子集问题和全排列问题都是经典的回溯法问题，我们在上一篇文章中讲过详细的思路和题解。本期的例题是它们的变种，都是在输入中存在**重复元素**，要避免输出重复的结果。本文假设你已经清楚了子集（subset）问题、排列（permutation）问题、组合（combination）问题的思路与解法。如果你对这几个问题的解法还不是很清楚，可以先阅读上一篇文章：**LeetCode 例题精讲 | 08 排列组合问题：回溯法的候选集合**。

这两道题目考察的都是回溯法的**剪枝策略**。所谓剪枝，实际上就是在决策树上去除部分分支，不进行遍历。我们要根据题目的性质，思考如何去除重复的结果，以及如何将去重实现为决策树上的剪枝。

这篇文章将会包含子集问题、排列问题、组合问题的去重策略、剪枝方案与题解代码。

有重复元素的子集问题

首先我们来回顾一下子集问题的决策树。回溯法一共进行 n 次决策， 每次决策有两个分支，决定第 i 个元素是否放入子集。以 $[1,2,3]$ 为例，决策树的形状如下图所示。



子集问题的决策树

题解代码如下：

```
public List<List<Integer>> subsets(int[] nums) {
    Deque<Integer> current = new ArrayDeque<>(nums.length);
    List<List<Integer>> res = new ArrayList<>();
    backtrack(nums, 0, current, res);
    return res;
}

// 候选集合: nums[k..N)
```

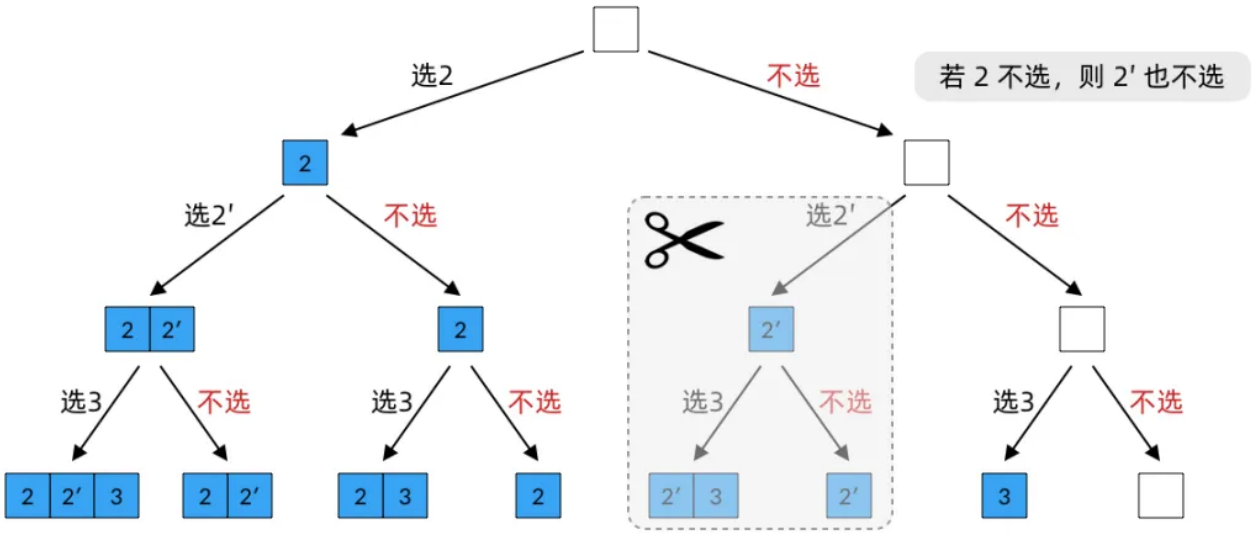
```
void backtrack(int[] nums, int k, Deque<Integer> current, List<List<Integer>> res) {
    if (k == nums.length) {
        res.add(new ArrayList<>(current));
        return;
    }

    // 不选择第 k 个元素
    backtrack(nums, k+1, current, res);

    // 选择第 k 个元素
    current.addLast(nums[k]);
    backtrack(nums, k+1, current, res);
    current.removeLast();
}
```

那么，当输入中有重复元素时，如何去除重复的结果呢？我们从具体的例子来寻找思路。以输入 $[1,2,2,2]$ 为例，$[1,2,2]$ 是它的一个子集。为了区分重复的 2，我们将三个 2 记为 2_1、2_2、2_3，那么子集 $[1,2,2]$ 有可能是 $[1,2_1,2_2]$、$[1,2_1,2_3]$ 或者 $[1,2_2,2_3]$。在这三个重复的结果中，我们只需要保留其中一个。

我们可以规定：**在一连串的 2 里面，如果第一个 2 没有放入子集，那么后面的 2 都不能放入子集。** 也就是说，如果不选择 2_1，就不能选择 2_2 或 2_3。这样要想在结果中出现两个 2，只可能是 2_1 和 2_2。去重成功！对应的决策树剪枝情况如下图所示。



有重复元素的子集问题的决策树剪枝

如何将这段剪枝逻辑翻译成代码呢？需要做到两点：

1. 在回溯之前，预先将数组中所有元素排序，这样数组中相等的元素就都是相邻的；
2. 每次对于元素 x 做决策时，如果不选择候选元素 x，那么 x 后面紧跟着的相等元素也都跳过，不再选择。

以下是题解代码。可以看到， 剪枝的逻辑是通过**一次删除多个候选集合中的元素**来实现的。候选集合这个概念，在理解“剪枝”的时候也能发挥重要作用。

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    // 对元素排序，保证相等的元素相邻
    Arrays.sort(nums);
    Deque<Integer> current = new ArrayDeque<>(nums.length);
    List<List<Integer>> res = new ArrayList<>();
    backtrack(nums, 0, current, res);
    return res;
}

// 候选集合: nums[k..N)
void backtrack(int[] nums, int k, Deque<Integer> current, List<List<Integer>> res) {
    if (k == nums.length) {
        res.add(new ArrayList<>(current));
        return;
    }

    // 选择 nums[k]
    current.addLast(nums[k]);
    backtrack(nums, k+1, current, res);
    current.removeLast();

    // 不选择 nums[k]
    // 将后续和 nums[k] 相等的元素 nums[k..j) 都从候选集合中删除
    int j = k;
    while (j < nums.length && nums[j] == nums[k]) {
        j++;
    }
    backtrack(nums, j, current, res);
}
```

有重复元素的全排列问题

有重复元素的全排列问题，其思路和子集问题非常像，只不过由于回溯方式的不同，代码上显得不太一样。首先回顾全排列问题的原始代码：

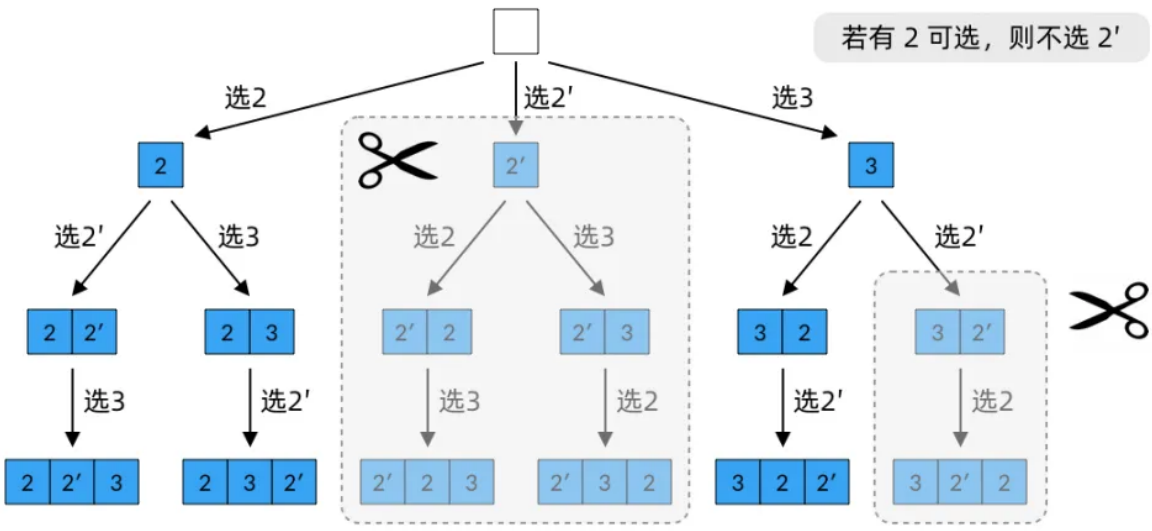
```
public List<List<Integer>> permute(List<Integer> nums) {
    List<Integer> current = new ArrayList<>(nums);
    List<List<Integer>> res = new ArrayList<>();
```

```
        backtrack(current, 0, res);
        return res;
    }

    // current[0..k) 是已选集合， current[k..N) 是候选集合
    void backtrack(List<Integer> current, int k, List<List<Integer>> res) {
        if (k == current.size()) {
            res.add(new ArrayList<>(current));
            return;
        }
        // 从候选集合中选择
        for (int i = k; i < current.size(); i++) {
            // 选择数字 current[i]
            Collections.swap(current, k, i);
            // 将 k 加一
            backtrack(current, k+1, res);
            // 撤销选择
            Collections.swap(current, k, i);
        }
    }
```

对于有重复元素的全排列问题，我们还是以输入 $[1,2,2,2]$ 为例思考剪枝策略：记其中的三个 2 分别为 2_1、2_2、2_3。对于排列 $[1,2,2,2]$ 来说，它实际上有可能是 $[1,2_1,2_2,2_3]$、$[1,2_1,2_3,2_2]$、$[1,2_2,2_1,2_3]$、$[1,2_2,2_3,2_1]$、$[1,2_3,2_1,2_2]$ 或者 $[1,2_3,2_2,2_1]$，共 6 种可能。我们该如何做剪枝，只保留其中一种结果呢？

我们可以规定：**如果有多个 2 候选，那么只能选第一个 2，后面的 2 此次不能选。** 也就是说，三个 2 只能先选 2_1，再选 2_2，最后选 2_3，只有排列 $2_1,2_2,2_3$ 才是合法的结果。去重成功！对应的决策树剪枝情况如下图所示。



有重复元素的全排列问题的决策树剪枝

不过对于全排列问题，我们不能通过对原数组排序来保证相等的元素一定相邻。在回溯的过程中会做各种元素的交换操作，打破原先的相邻关系。那么我们需要使用一个集合辅助判断候选元素是否是第一次出现。

以下是题解代码。可以看到，剪枝的逻辑是通过**跳过某些候选集合中的元素**实现的。

```
public List<List<Integer>> permuteUnique(List<Integer> nums) {
    List<Integer> current = new ArrayList<>(nums);
    List<List<Integer>> res = new ArrayList<>();
    backtrack(current, 0, res);
    return res;
}

// 已选集合 current[0..m)，候选集合 current[m..N)
void backtrack(List<Integer> current, int m, List<List<Integer>> res) {
    if (m == current.size()) {
        res.add(new ArrayList<>(current));
        return;
    }

    // 使用 set 辅助判断相等的候选元素是否已经出现过。
    Set<Integer> seen = new HashSet<>();
    for (int i = m; i < current.size(); i++) {
        int e = current.get(i);
        if (seen.contains(e)) {
            // 如果已经出现过相等的元素，则不选此元素
            continue;
        }
        seen.add(e);
        Collections.swap(current, m, i);
        backtrack(current, m+1, res);
        Collections.swap(current, m, i);
    }
}
```

有重复元素的组合问题

组合问题在 LeetCode 上并没有对应题目，不过出于讲解的完整性，我们不妨在子集问题、排列问题之后继续思考，有重复元素的组合问题该如何解决。同样地，首先回顾组合问题的题解代码：

```
public List<List<Integer>> combine(List<Integer> nums, int k) {
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(k, nums, 0, current, res);
    return res;
}

// current 是已选集合， nums[m..N) 是候选集合
void backtrack(int k, List<Integer> nums, int m, Deque<Integer> current, List<List<Integer>> res) {
    // 当已选集合达到 k 个元素时，收集结果并停止选择
    if (current.size() == k) {
        res.add(new ArrayList<>(current));
        return;
    }
    // 从候选集合中选择
    for (int i = m; i < nums.size(); i++) {
        // 选择数字 nums[i]
        current.addLast(nums.get(i));
        // 元素 nums[m..i) 均失效
        backtrack(k, nums, i+1, current, res);
        // 撤销选择
        current.removeLast();
    }
}
```

实际上，上文所讲解的全排列问题的去重思路，可以直接套用在组合问题的去重上。它们的剪枝思路一模一样！

以输入 $[1,2,2,2]$ 为例，和全排列问题相同的思路，我们规定：**如果有多个 2 候选，那么只能选第一个 2，后面的 2 此次不能选。** 也就是说，如果结果中只有一个 2，那么只可能是 2_1；如果结果中包含两个 2，那么只可能是 2_1 和 2_2。

如果把这个思路写成代码，组合问题反而比排列问题的代码要简单很多。这是因为组合问题中不存在交换操作打乱元素顺序的情况。我们只要在一开始对数组排序，就可以很容易地判断某元素是否是相等元素中的第一个。题解代码如下：

```
public List<List<Integer>> combine(int[] nums, int k) {
    // 对元素排序，保证相等的元素相邻
    Arrays.sort(nums);
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(k, nums, 0, current, res);
```

```
        backtrack(k, nums, 0, current, res);

        return res;
    }

    // current 是已选集合， nums[m..N) 是候选集合
    void backtrack(int k, int[] nums, int m, Deque<Integer> current, List<List<Integer>> res) {
        // 当已选集合达到 k 个元素时，收集结果并停止选择
        if (current.size() == k) {
            res.add(new ArrayList<>(current));
            return;
        }
        // 从候选集合中选择
        for (int i = m; i < nums.length; i++) {
            if (i > m && nums[i] == nums[i-1]) {
                // nums[i] 与前一个元素相等，说明不是相等元素中第一个出现的，跳过。
                continue;
            }
            // 选择数字 nums[i]
            current.addLast(nums[i]);
            // 元素 nums[m..i) 均失效
            backtrack(k, nums, i+1, current, res);
            // 撤销选择
            current.removeLast();
        }
    }
```

总结

我们在一篇文章中讲解了子集问题、排列问题、组合问题的有重复元素版本。可以看到，它们的解题思路都非常类似。总结起来，这一类需要去重的回溯法题目都可以遵循这几个步骤：

1. 首先把无重复元素版本的题目写对（这很重要！）；
2. 思考剪枝策略，一般是删除或跳过候选集合中的某些元素；
3. 根据代码的特点加入特定的剪枝判断，可以使用预排序，或是利用数据结构。

本期例题的题解代码看似容易，但还是需要多加练习并熟练为好。回溯法问题的难度除了在解题思路，还在于如何写出清晰、易懂、无 bug 的代码。多做练习，精简代码，这是解好回溯法问题的诀窍。

参考资料

[1] LeetCode 90 - Subsets II: *https://leetcode.com/problems/subsets-ii/*

[2] LeetCode 47 - Permutations II: *https://leetcode.com/problems/permutations-ii/*