排列 / 组合 / 子集 问题

78. 子集

90. 子集 II

77. 组合

39. 组合总和

40. 组合总和 II

216. 组合总和 III

46. 全排列

47. 全排列 II

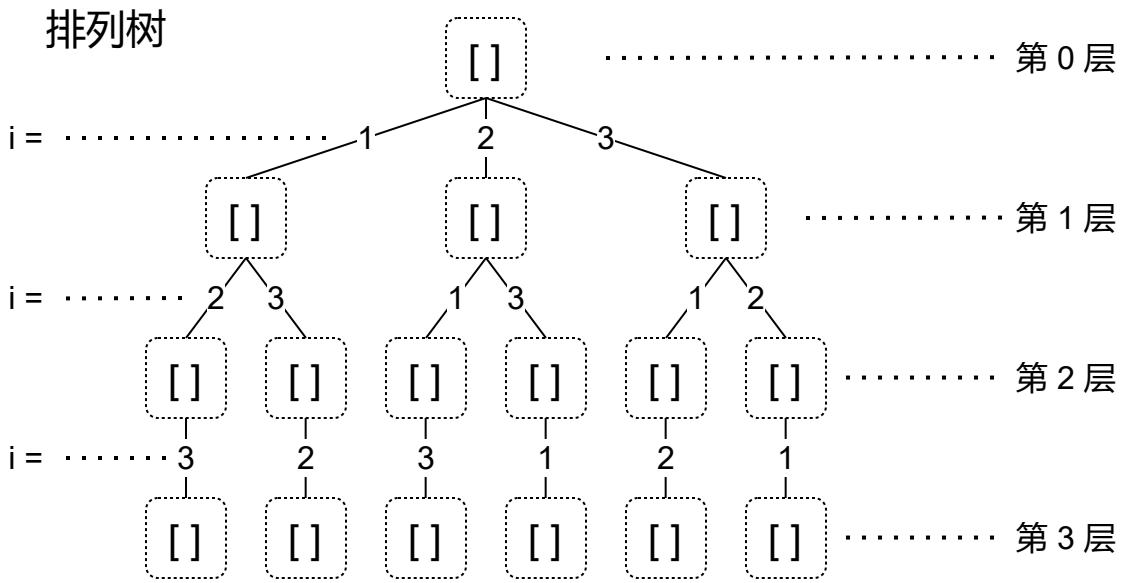
根据「元素是否重复」「元素是否可被重复选择」两个条件，可排列组合四种情况

- **元素无重不可复选**：即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次
- **元素可重不可复选**：即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次
- **元素无重可复选**：即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次
- **元素可重可复选**：即 `nums` 中的元素可以存在重复，每个元素可以被使用若干次

‼️ 注：由于第四种情况，元素又可重复，又可复选。如果进行一个去重后，和情况三一样了。所以第四种情况不单独考虑，基本上和第三种情况一样

在进行下面分析的时候，请记住下面两个「回溯树」。由于组合与子集问题差不多，所以大多数时候组合与子集放在一起讨论

‼️ 注：第 `n` 层的结果 = 第 `n` 层到第 `0` 层的路径上的数字集合



元素无重不可复选

这种情况是最简单也是最常见的一种

子集

我们先分析「子集」类型

抛开题目，如果让我们手动求集合 `[1,2,3]` 的子集，如何求?

- 首先肯定有空集 `[]`
- 其次求只有一个元素的子集 `[1]`、`[2]`、`[3]`
- 然后求只有两个元素的子集 `[12]`、`[13]`、`[23]`
- 最后求只有三个元素的子集 `[123]`

现在观察我们的子集树，是不是有种神奇的对应关系。第 `0` 层刚好对应「空集」，第 `1` 层对应只有一个元素的子集，以此类推

具体可见 78. 子集，详细代码如下:

```
// 存储路径上的信息
private List<Integer> track = new ArrayList<>();
// 存储最终结果
private List<List<Integer>> res = new ArrayList<>();
private void backtrack(int[] nums, int start) {
    // 二话不说，直接存储到结果中
    res.add(new ArrayList<>(track));

    for (int i = start; i < nums.length; i++) {
        track.add(nums[i]);
        // 下一层的 i 均为 该层 i+1
        backtrack(nums, i + 1);
        track.remove(track.size() - 1);
    }
}
// 解释：每个 for 循环代表一层，最开始还没有进入任何循环时是第 0 层
```

```
// 第 1 层:    0     1    2
// 第 2 层:  1   2   2
// 第 3 层:  2
// 对于每一次进入下一层的时候，就需要把路径信息存储到结果中
```

组合

下面分析「组合」类型

其实组合类型和子集几乎完全一样

对于一个数组 `[1,2,3]` ，大小为 2 的组合有 `[12]、[13]、[23]`

发现了吗？？？！！！大小为 2 的组合不就是第 2 层的集合嘛！！！所以只需要把代码稍微修改即可

具体可见 77. 组合 && 216. 组合总和 III，详细代码如下：

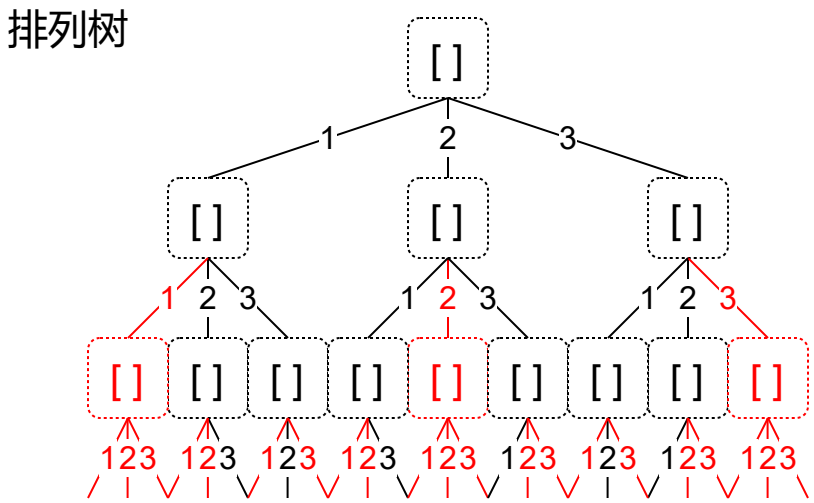
```
private void backtrack(int[] nums, int start, int k) {
    // 当 track 大小为 k 时，满足条件
    if (track.size() == k){
        res.add(new ArrayList<>(track));
        return ;
    }

    for (int i = start; i < nums.length; i++) {
        track.add(nums[i]);
        backtrack(nums, i + 1);
        track.remove(track.size() - 1);
    }
}
```

排列

下面分析「排列」类型

排列类型稍微稍微稍微麻烦一丢丢，不过也还好

仔细观察排列树，会发现其实这棵树是进行了一定剪枝的。未剪枝的排列树如下：



其中，红色标注出来的都是不符合条件的情况，因为一个元素重复使用了多次

我们可以用 `used[]` 记录每个节点的使用情况，如果节点已经被使用过了，直接跳过

具体可见 46. 全排列，详细代码如下：

```
private boolean[] used = new int[nums.length];
private void backtrack(int[] nums) {
    // 当 track 大小为 nums.length 时，满足条件
    if (track.size() == nums.length){
        res.add(new ArrayList<>(track));
        return ;
    }
    // 注意：这里的循环每次都是从 0 开始
    for (int i = 0; i < nums.length; i++) {
        // 如果使用过了，直接跳过
        if (used[i]) continue;
        // 标记使用
        used[i] = true;
        track.add(nums[i]);
        backtrack(nums);
        // 去除使用
        used[i] = false;
        track.remove(track.size() - 1);
    }
}
```

至此，「元素无重不可复选」情况下的三种题型都已经分析完毕！！！

元素可重不可复选

忘记提示一个前提：：：：： 这种情况均需要对数组先进行排序，让所有相同的元素相邻

子集

我们先分析「子集」类型

为了区分，我们在相同元素的右上角用角标区分

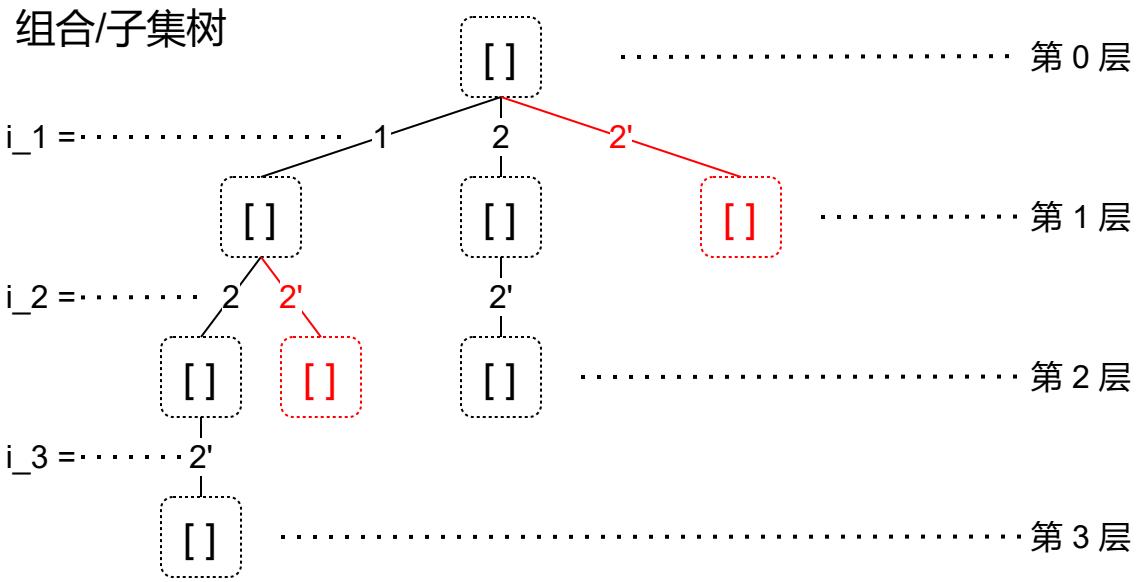
如果要求集合 `[1,2,2']` 的子集，如何求?

如果我们还是在上一种类型的代码下求解的话，出来的结果将会是：

- `[]`
- `[1]、[2]、[2']`
- `[12]、[12']、[22']`
- `[122']`

我们可以看到结果中 `[2]` 和 `[2']` 以及 `[12]` 和 `[12']` 重复

反应到子集树中，即为：



红色标注出来的分支即为重复的情况

我们如何把这种情况剪枝掉呢？？？

我们不难发现，每次发生重复的情况，均为前一元素和当前元素相同造成的，所以解决办法就很明显了

具体可见 90. 子集 II，详细代码如下：

```
Arrays.sort(nums);
private void backtrack(int[] nums, int start) {
    // 二话不说，直接存储到结果中
    res.add(new ArrayList<>(track));

    for (int i = start; i < nums.length; i++) {
        // 剪枝
        if (i > start && nums[i] == nums[i - 1]) continue;
        track.add(nums[i]);
        backtrack(nums, i + 1);
        track.remove(track.size() - 1);
    }
}
```

组合

「组合」类型的代码几乎一样，此处省略。**具体可见 40. 组合总和 II**

这里详细分析一下这个题目，先看代码

```
Arrays.sort(candidates);
private void backtrack(int[] candidates, int target, int start) {
    // base case 1 : sum == target
    if (sum == target) {
        res.add(new ArrayList<>(track));
        return ;
    }
    // base case 2 : sum > target
    if (sum > target) return;
    for (int i = start; i < candidates.length; i++) {
        if (i > start && candidates[i] == candidates[i - 1]) continue;
        sum += candidates[i];
        track.add(candidates[i]);
        backtrack(candidates, target, i + 1);
        sum -= candidates[i];
        track.remove(track.size() - 1);
    }
}
```

这个题目引入了一个「和」的概念，只有子集和为 target 才满足情况，所以 base case 1 就是 sum == target

需要注意的是 base case 2，如果没有加，可能会超时

排列

下面分析「排列」类型

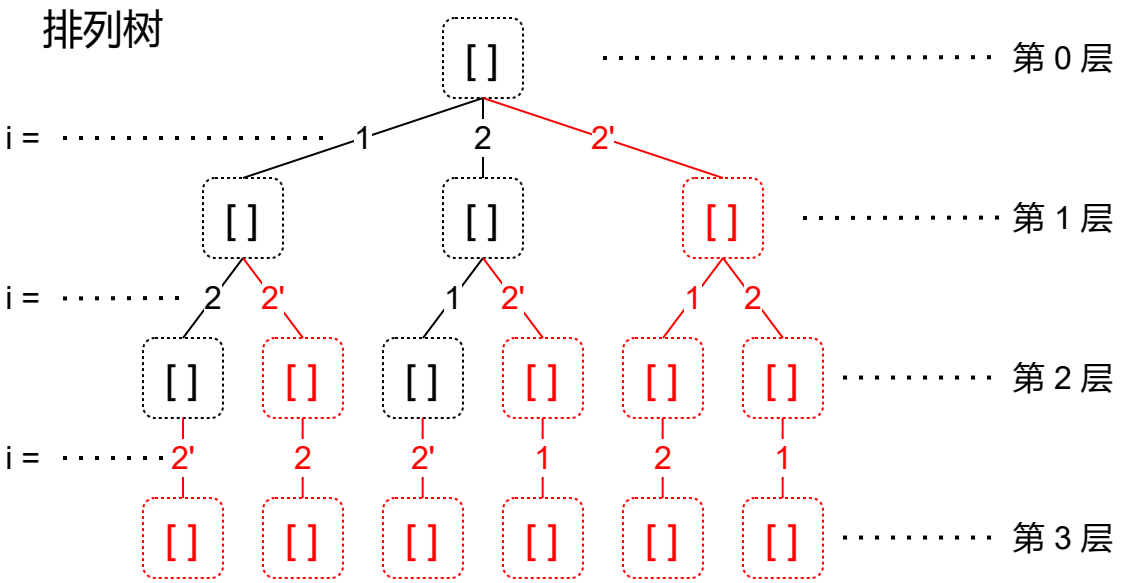
可以借鉴「子集」的思路，增加 `i > start && nums[i] == nums[i - 1]` 条件

具体可见 47. 全排列 II，部分代码如下：

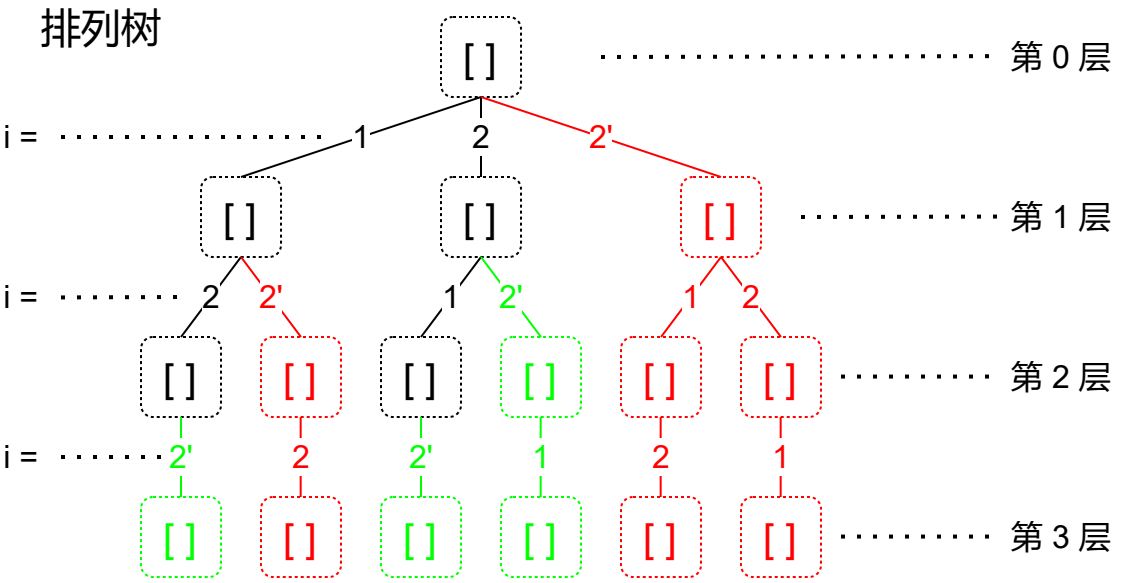
```
Arrays.sort(nums);
private void backtrack(int[] nums) {
    // 当 track 大小为 nums.length 时，满足条件
    if (track.size() == nums.length){
        res.add(new ArrayList<>(track));
        return ;
    }

    for (int i = 0; i < nums.length; i++) {
        // 剪枝
        if (used[i]) continue;
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        used[i] = true;
        track.add(nums[i]);
```

```
            backtrack(nums);
            used[i] = false;
            track.remove(track.size() - 1);
        }
    }
```

但是存在一个问题，先看排列树：



红色标注的部分均会被剪枝掉，显然 `track.size() == nums.length` 永远不会成立，所以最后结果只能为 `null`



绿色标注出的情况其实是不应该被剪枝掉的，如何排除掉这种情况呢？？

增加一个判断 `!used[i - 1]` ，只有当前一个元素没有使用时，才需要被剪枝掉

修改后的代码：

```
// 剪枝
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) continue;
```

元素无重可复选

这种情况好像没有「子集」「排列」类型，只有「组合」类型，不过问题不大，直接分析「组合」

组合

先直接看一个题目 39. 组合总和

组合树如下所示：

我们可以发现唯一的区别就是无限的递归，每次循环开始的下标都是从上次循环的下标开始，而不是 +1 后的下标

!! 需要注意的是，我们必须加一个 base case 2（如代码中所示），不然真的会停不下来

代码如下：

```
private void backtrack(int[] candidates, int target, int start) {
    // base case 1
    if (sum == target) {
        res.add(new ArrayList<>(track));
        return ;
    }
    // base case 2
    if (sum > target) return ;
    for (int i = start; i < candidates.length; i++) {
        sum += candidates[i];
        track.add(candidates[i]);
        // 注意：是 i，而不是 i+1
        backtrack(candidates, target, i);
        sum -= candidates[i];
        track.remove(track.size() - 1);
    }
}
```