# LeetCode 例题精讲 | 08 排列组合问题：回溯法的候选集合

原创 nettee 面向大象编程 3月9日

来自专辑

LeetCode 例题精讲

本期例题：**LeetCode 46 - Permutations**[1]（Medium）

> 给定一个**不重复**的数字集合，返回其所有可能的全排列。例如：
>
> - **输入**：`[1, 2, 3]`
> - **输出**：
>
> ```
> [
>   [1,2,3],
>   [1,3,2],
>   [2,1,3],
>   [2,3,1],
>   [3,1,2],
>   [3,2,1]
> ]
> ```

在第三讲中，我们就讲过了回溯法问题的基本思想。回溯法问题用递归求解，可以联系上树的遍历，我们可以将决策路径画成一棵树，回溯的过程就是这棵树的遍历过程。

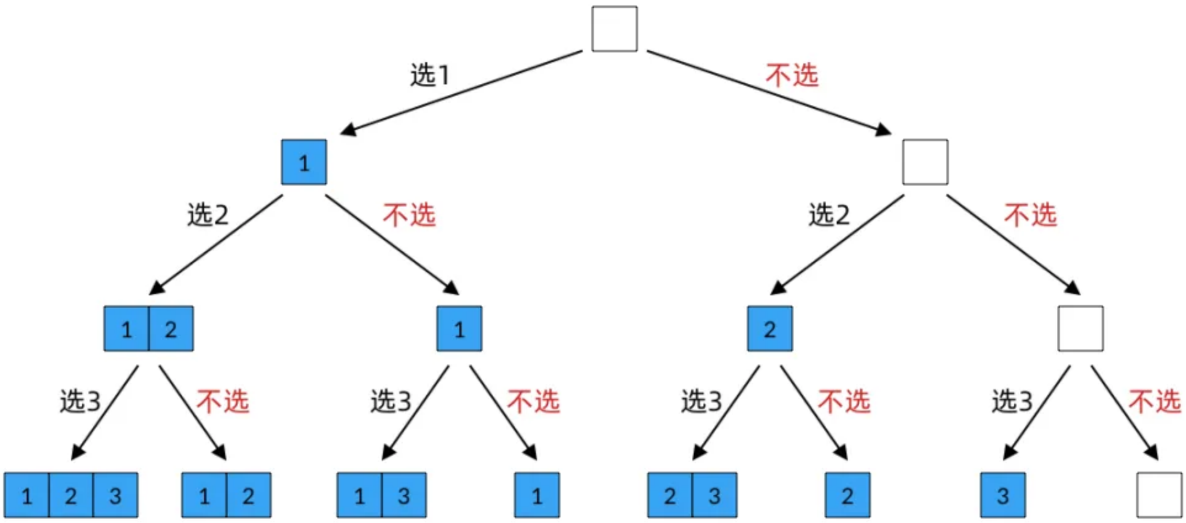
不过在那篇文章中，我们只求解了一道非常简单的回溯法问题：子集（subset）问题。在面试中，我们需要有能力更加复杂的回溯法问题，并应对题目的各种变种。本篇以经典的排列（permutation）和组合（combination）问题为例，讲讲求解回溯法问题的要点：**候选集合**。

这篇文章将会包含：

- 回溯法的"选什么"问题与候选集合
- 全排列、排列、组合问题的回溯法解法
- 回溯法问题中，如何维护候选集合
- 回溯法问题中，如何处理失效元素

## 回溯法的重点："选什么"

我们说过，回溯法实际上就是在一棵决策树上做遍历的过程。那么，求解回溯法题目时，我们首先要思考所有决策路径的形状。例如，子集问题的决策树如下图所示：




子集问题的决策树

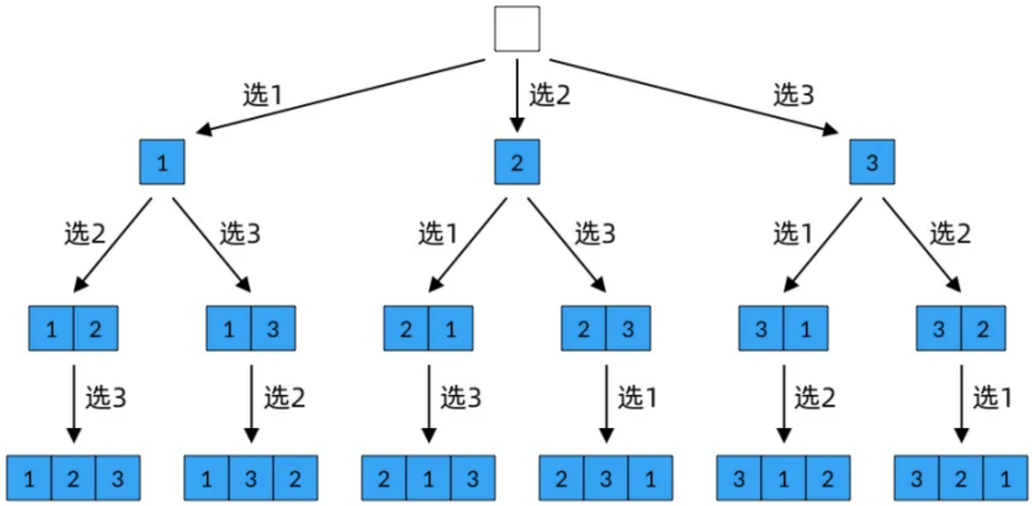
决策树形状主要取决于每个结点处可能的分支，换句话说，就是在每次做决策时，我们 **“可以选什么”**、 **“有什么可选的”**。

对于子集问题而言，这个“选什么”的问题非常简单，每次只有一个元素可选，要么选、要么不选。不过，对于更多的回溯法题目，“选什么”的问题并不好回答。这时候，我们就需要分析问题的**候选集合**，以及候选集合的变化，以此得到解题的思路。

## 全排列问题：如何维护候选集合

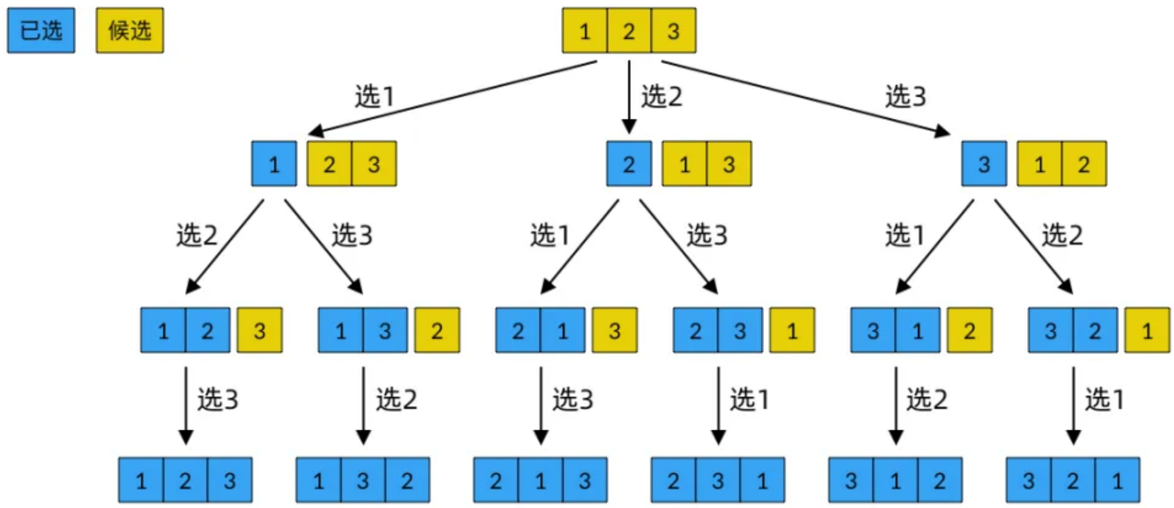
让我们拿经典的全排列问题来讲解回溯法问题的**候选集合**概念。

在全排列问题中，决策树的分支数量并不固定。我们一共做 $n$ 次决策，第 $i$ 次决策会选择排列的第 $i$ 个数。选择第一个数时，全部的 $n$ 个数都可供挑选。而由于已选的数不可以重复选择，越往后可供选择的数越少。以 $n=3$ 为例，决策树的形状如下图所示：




全排列问题的决策树

如果从候选集合的角度来思考，在进行第一次选择时，全部的 3 个数都可以选择，候选集合的大小为 3。在第二次选择时，候选集合的大小就只有 2 了；第三次选择时，候选集合只剩一个元素。可以看到，全排列问题候选集合的变化规律是：每做一次选择，候选集合就少一个元素，直到候选集合选完为止。我们可以在上面的决策树的每个结点旁画上候选集合的元素，这样看得更清晰。



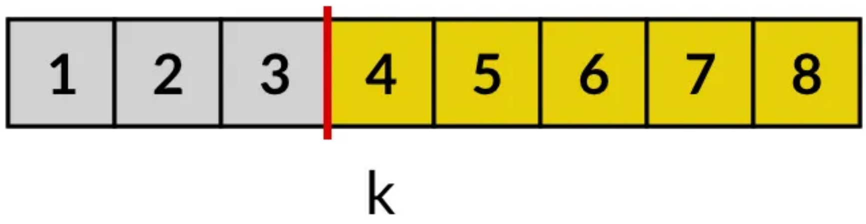

全排列问题有候选集合的决策树

可以看到，**已选集合**与**候选集合**是补集的关系，它们加起来就是全部的元素。而在回溯法的选择与撤销选择的过程中，已选集合和候选集合是此消彼长的关系。

如何根据这样的思路写出代码呢？当然可以用 `HashSet` 这样的数据结构来表示候选集合。但如果你这么去写的话，会发现代码写起来比较啰嗦，而且 set 结构的“遍历-删除”操作不太好写。在这里，我不展示使用 set 结构的代码。大家只要明白一条要点：**在一般情况下，候选集合使用数组表示即可。** 候选集合上需要做的操作并不是很多，使用数组简单又高效。

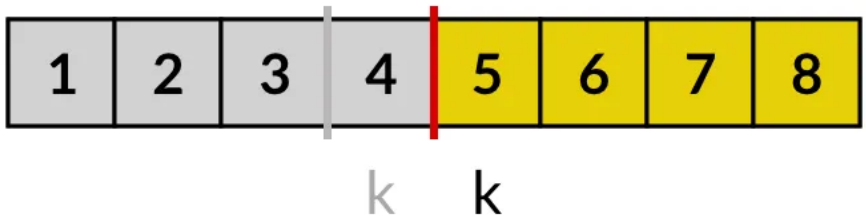
在子集问题中，我们定义了变量 `k` ，表示当前要对第 `k` 个元素做决策。实际上，变量 `k` 就是候选集合的边界，指针 `k` 之后的元素都是候选元素，而 `k` 之前都是无效元素，不可以再选

了。



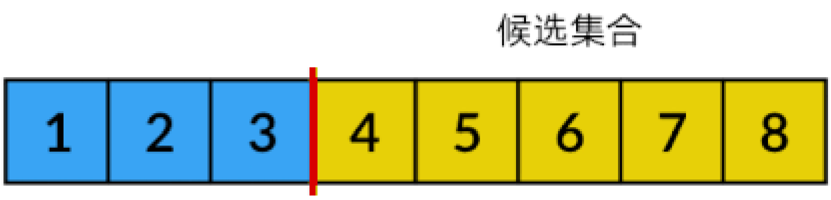

用数组表示候选集合

而每次决策完之后将 k 加一，就是将第 k 个元素移出了候选集合。




将第 k 个元素移出候选集合

在全排列问题中，我们要处理的情况更难一些。每次做决策时，候选集合中的所有元素都可以选择，也就是有可能删除候选集合中间的元素，这样数组中会出现"空洞"。这种情况该怎么处理呢？我们可以使用一个巧妙的方法，先将要删除的元素与第 k 个元素交换，再将 k 加一，过程如下方动图所示：




从候选集合中部删除元素（动图）

不知道你有没有注意到，上图中候选集合之外的元素画成了蓝色，这些实际上就是已选集合。前面分析过，已选集合与候选集合是互补的。将蓝色部分看成已选集合的话，我们从候选集合中删除的元素，正好加入了已选集合中。也就是说，我们可以只用一个数组同时表示已选集合和候选集合！

理解了图中的关系之后，题解代码就呼之欲出了。我们只需使用一个 `current` 数组，左半边表示已选元素，右半边表示候选元素。指针 `k` 不仅是候选元素的开始位置，还是已选元素的结束位置。我们可以得到一份非常简洁的题解代码：

```
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);
    }
}
```

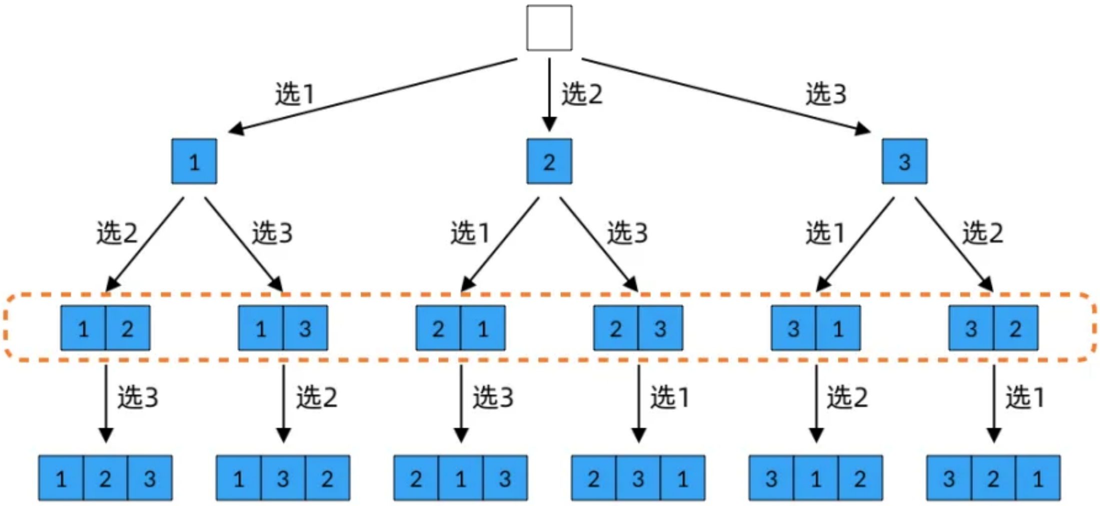
注意写在递归函数上方的注释。在写回溯法问题的代码时，你需要时刻清楚什么是已选集合，什么是候选集合。注释中的条件叫做“不变式”。一方面，我们在函数中可以参考变量 k 的含义，另一方面，我们在做递归调用的时候，要保证这个条件始终成立。特别注意代码中递归调用传入的参数是 `k+1` ，即删除一个候选元素，而不是传入 `i+1` 。

## n 中取 k 的排列

全排列问题是 $n$ 中取 $n$ 的排列，可以记为 $P(n,n)$。在面试中，我们很可能会遇到各种各样的变种题，那么 $n$ 中取 $k$ 的排列 $P(n,k)$、组合 $C(n,k)$ 我们也要掌握。

$P(n,k)$ 问题非常简单，我们只需要在全排列的基础上，做完第 $k$ 个决策后就将结果返回。也就是说，只遍历决策树的前 $k$ 层。例如 $n=3, k=2$ 时，决策树的第 2 层，已选集合中有两个元

素，将这里的结果返回即可。




n 中取 k 的排列的决策树

题解代码如下所示，只需要修改递归结束的条件即可。

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

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

注意这里 $k$ 是题目的输入，所以原先我们代码里的变量 `k` 重命名成了 `m` 。此外，就是递归函数开头的 if 语句条件发生了变化，当已选集合达到 $k$ 个元素时，就收集结果停止递归。

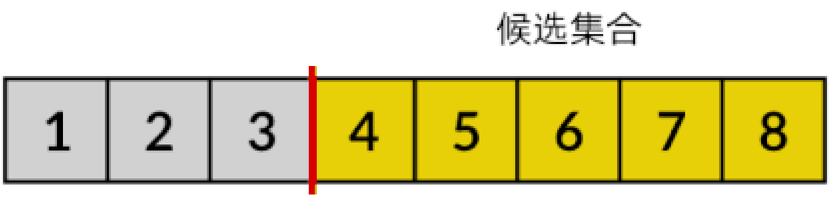
## 组合问题：失效元素

由于排列组合的密切联系，组合问题 $C(n, k)$ ，即 $n$ 中取 $k$ 的组合，可以在 $P(n, k)$ 问题的解法上稍加修改而来。

我们先思考一下组合和排列的关系。元素相同，但顺序不同的两个结果视为不同的排列，例如 $[1, 2, 3]$ 和 $[2, 1, 3]$。但顺序不同的结果会视为同一组合。那么，我们只需要考虑 $P(n, k)$ 中所有**升序**的结果，就自然完成了组合的去重，得到 $C(n, k)$。

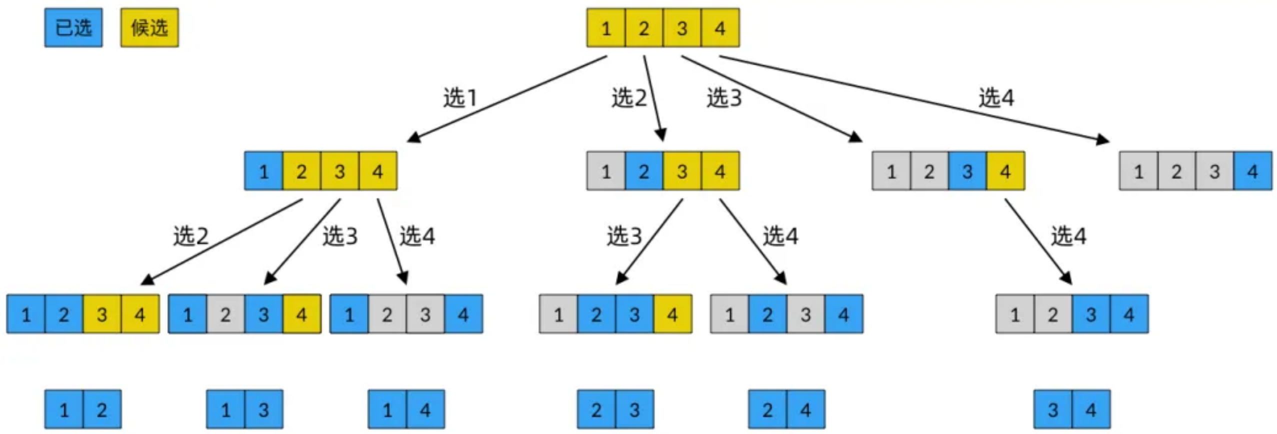
那么，如何让回溯只生成升序的排列呢？这需要稍微动点脑筋，但也不是很难，只需要做到：**每当选择了一个数 $x$ 时，将候选集合中的所有小于 $x$ 的元素删除，不再作为候选元素。**

再仔细想想的话，在排列问题为了维护候选集合而进行的交换操作，这里也不需要了。例如下面的例子，选择元素 6 之后，为了保持结果升序，前面的元素 4、5 也不能要了。不过，我们并不需要关注失效元素，我们只需要关注候选集合的变化情况。我们发现，剩下的候选集合仍然是数组中连续的一段，不会出现排列问题中的“空洞”情况。我们只用一个指针就能表示新的候选集合。




从候选集合中删除多个元素

下图是 $C(n, k)$ 的决策树，可以看到，候选集合都是连续的。已选集合不连续没有关系，我们可以另开一个数组保存已选元素。




组合问题的决策树

按照这个思路，我们可以写出 $C(n,k)$ 的代码。

```
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();
    }
}
```

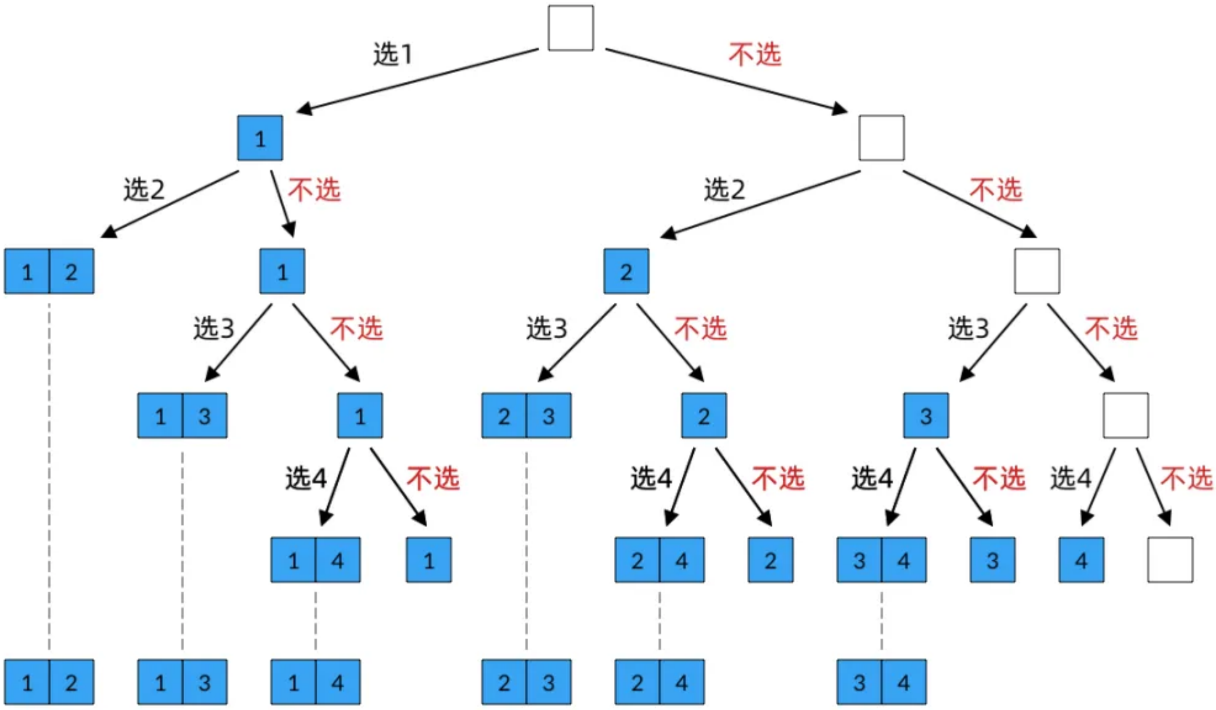
由于已选集合与候选集合并非互补，这里用单独的数组存储已选元素，这一点上与子集问题类似。

## 组合问题与子集问题的关系

也许是排列 & 组合的 CP 感太重，所以我们在思考组合问题的解法的时候会自然地从排列问题上迁移。其实，组合问题和子集问题有很密切的联系。

### 由子集问题求解组合问题

组合问题可以看成是子集问题的特殊情况。从 $n$ 中取 $k$ 个数的组合，实际上就是求 $n$ 个元素的所有大小为 $k$ 的子集。也就是说，组合问题的结果是子集问题的一部分。我们可以在子集问题的决策树的基础上，当已选集合大小为 $k$ 的时候就不再递归，就可以得到组合问题的决策树。




在子集问题决策树基础上得到的组合问题决策树

具体的代码这里就不展示了，请读者自行在子集问题的题解代码的基础上修改得到 $C(n,k)$ 的代码。

### 由组合问题求解子集问题

对于子集问题，大小为 $n$ 的集合共有 $2^n$ 个可能的子集。对于组合问题 $C(n,k)$，得到的结果个数是组合数 $C_n^k$，或者记为 $\binom{n}{r}$。根据二项式定理：

$$2^n = C_n^0 + C_n^1 + \cdots + C_n^n$$

要得到全部的 $2^n$ 个子集，我们可以计算所有 $n$ 中取 $0, 1, \ldots, n$ 的组合，再把这些组合加起来。根据这个思路，我们可以在组合问题的题解代码上稍加修改得到子集问题的解：

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

// current 是已选集合， nums[m..N) 是候选集合
void backtrack(List<Integer> nums, int m, Deque<Integer> current, List<List<Integer>> res) {
    // 收集决策树上每一个结点的结果
    res.add(new ArrayList<>(current));
    if (m == nums.size()) {
            // 当候选集合为空时，停止递归
        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(nums, i+1, current, res);
        // 撤销选择
        current.removeLast();
    }
}
```

可以看到，每次做决策都会增加一个已选元素。当递归到第 $k$ 层时，计算的就是大小为 $k$ 的子集。不过，这样写出的子集问题解法没有原解法易懂，我还是更推荐原解法。

## 总结

排列组合问题是回溯法中非常实际也非常典型的例题，可以通过做这些题目来体会回溯法的基本技巧。不过它们在 LeetCode 中没有完全对应的例题。文章开头的例题是全排列问题。对于组合问题，LeetCode 只有一个简化版 **77. Combinations**[2]，其中数字固定为 1 到 n 的整数。

排列组合问题展示了在求解回溯法问题时，**候选集合**的概念对理清思路的重要性。实际上，回溯法中的“选择”与“撤销选择”，实际上就是从候选集合中删除元素与添加回元素的操作。而我们

在写代码的时候要注意在递归函数上方写注释，明确数组的哪一部分是候选集合。

排列组合问题还存在着一些变种，例如当输入存在重复元素的时候，如何避免结果重复，就需要使用决策树的剪枝方法。下一篇文章将会讲解回溯法问题中如何正确地剪枝。

## 参考资料

[1] LeetCode 46 - Permutations: *https://leetcode.com/problems/permutations/*

[2] 77. Combinations: *https://leetcode.com/problems/combinations/*