学习目标

回溯是最重要的算法思想之一，主要解决一些暴力枚举也搞不定的问题，例如组合、分割、子集、排列，棋盘等。从性能角度来看回溯算法的效率并不高，但对于这些暴力都搞不定的算法能出结果就很好了，效率低点没关系。

回溯可以视为递归的拓展，很多思想和解法都与递归密切相关，在很多材料中都将回溯都与递归同时解释，例如本章2.1的路径问题就可以使用递归和回溯两种方法来解决。因此学习回溯时，我们对比递归来分析其特征会理解更深刻。

关于递归和回溯的区别，我们设想一个场景，某猛男想脱单，现在有两种策略：

- 1.递归策略：先与意中人制造偶遇，然后了解人家的情况，然后约人家吃饭，有好感之后尝试拉人家的手，没有拒绝就表白。
- 2.回溯策略：先统计周围所有的单身女孩，然后一个一个表白， 被拒绝就说“我喝醉了”，然后就当啥也没发生，继续找下一个。

其实回溯本质就这么个过程，请读者学习本章时认真揣摩这个过程。

回溯最大的好处是有非常明确的模板，所有的回溯都是一个大框架，因此透彻理解回溯的框架是解决一切回溯问题的基础。第一章我们只干一件事，那就是分析这个框架。

回溯不是万能的，而且能解决的问题也是非常明确的，例如组合、分割、子集、排列，棋盘等等，不过这些问题具体处理时又有很多不同，本章我们梳理了多个最为热门的问题来解释，请同学们认真对待。

回溯可以理解为递归的拓展，而代码结构又特别像深度遍历N叉树，因此只要知道递归，理解回溯并不难，难在很多人不理解为什么在递归语句之后要有个“撤销”的操作。我们会通过图示轻松给你解释该问题。这里先假设一个场景，你谈了个新女朋友，来你家之前，你是否会将你前任的东西赶紧藏起来？回溯也一样，有些信息是前任的，要先处理掉才能重新开始。

回溯最让人激动的是有非常清晰的解题模板，如下所示，大部分的回溯代码框架都是这个样子，具体为什么这样子我们后面再解释。

```
    void backtracking(参数) {
        if (终止条件) {
            存放结果;
            return;
        }
        for (选择本层集合中元素（画成树，就是树节点孩子的大小）){
            处理节点;
            backtracking();
            回溯，撤销处理结果;
        }
    }
```

1.纵横回溯

回溯是有明确的解题模板的，本章我们只干一件事——分析回溯的模板。

1.1 从N叉树说起

在解释回溯之前， 我们先看一下N叉树遍历的问题，这是回溯代码的前奏。我们知道在二叉树中，按照前序遍历的过程如下所示：

```
def travel(node root):
    if (root == null):
        return
    print root.val
    treeDFS(root.left)
    treeDFS(root.right)

#结点的定义
class Node(object):
    def __init__(self, val=-1, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

假如我现在是一个三叉、四叉甚至N叉树该怎么办呢？很显然这时候就不能用left和right来表示分支了，使用一个列表比较好，也就是这样子：

```
class Node(object):
   int val;
   nodes=[];
```

此时遍历代码如下：

```
def treeDFS(TreeNode root):
    //递归必须要有终止条件
    if (root == null):
      return

    print root.val
    //通过循环，分别遍历N个子树
    for ( i in range(len(nodes)):
        //2，todo一些操作，可有可无，视情况而定
        treeDFS("第i个子节点");
        //3，todo一些操作，可有可无，视情况而定
```

到这里，你有没有发现和上面说的回溯的模板非常像了？是的！非常像！既然很像，那说明两者一定存在某种关系。其他暂时不管，现在你只要先明白回溯的大框架就是遍历N叉树就行了。

1.2 为什么有的问题暴力搜索也不行

我们说回溯主要解决暴力枚举也解决不了的问题，什么问题这么神奇，暴力都搞不定？看个例子：

LeetCode77：给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。例如，输入n=4，k=2，则输出：

```
[[2,4], [3,4], [2,3], [1,2], [1,3], [1,4]]
```

首先明确这个题是什么意思，如果n=4，k=2，那就是从4个数中选择2个，问你最后能选出多少组数据。

这个是高中数学中的一个内容，过程大致这样：如果n=4，那就是所有的数字为{1,2,3,4}

- 1.先取一个1，则有[1,2]，[1,3]，[1,4]三种可能。
- 2.然后取一个2，因为1已经取过了，不再取，则有[2,3]，[2,4]两种可能。
- 3.再取一个3，因为1和2都取过了，不再取，则有[3,4]一种可能。
- 4.再取4，因为1，2，3都已经取过了，所以直接返回null。
- 5.所以最终结果就是[1,2]，[1,3]，[1,4]，[2,3]，[2,4]，[3,4]。

这就是我们思考该问题的基本过程，写成代码也很容易，双层循环轻松搞定：

```
n = 4
for ( i in range(n)):
   j=i+1
   for ( j in range(n)):
       print  i + " " + j
```

假如n和k都变大，比如n是200，k是3呢？也可以，三层循环基本搞定：

```
int n = 200;
 for ( i in range(n)):
      j=i+1
      for (  j in range(n)) :
           u=j+1
           for ( u in range(n)):
              print   i + " " + j + " " + u;
```

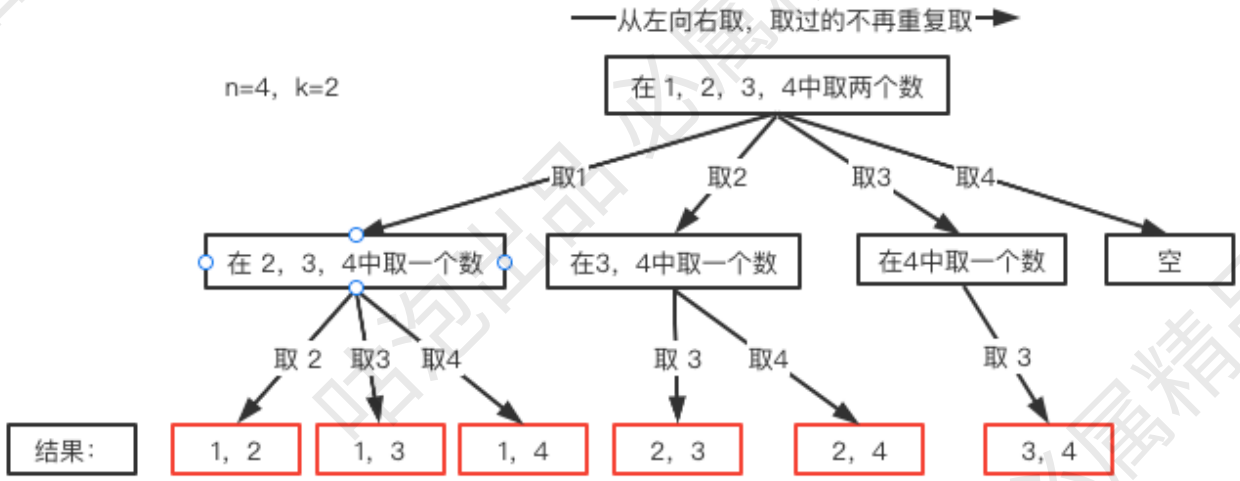
如何这里的K是5呢？甚至是50呢？你需要套多少层循环？甚至告诉你K就是一个未知的正整数k，你怎么写循环呢？是不是发现这时候已经无能为例了？所以暴力搜索就不行了。

这就是组合类型问题，除此之外子集、排列、切割、棋盘等方面都有类似的问题，暴力枚举也搞不定。

1.3 回溯=递归+局部枚举+放下前任

我们继续研究LeetCode77题，我们图示一下上面自己枚举时的过程。

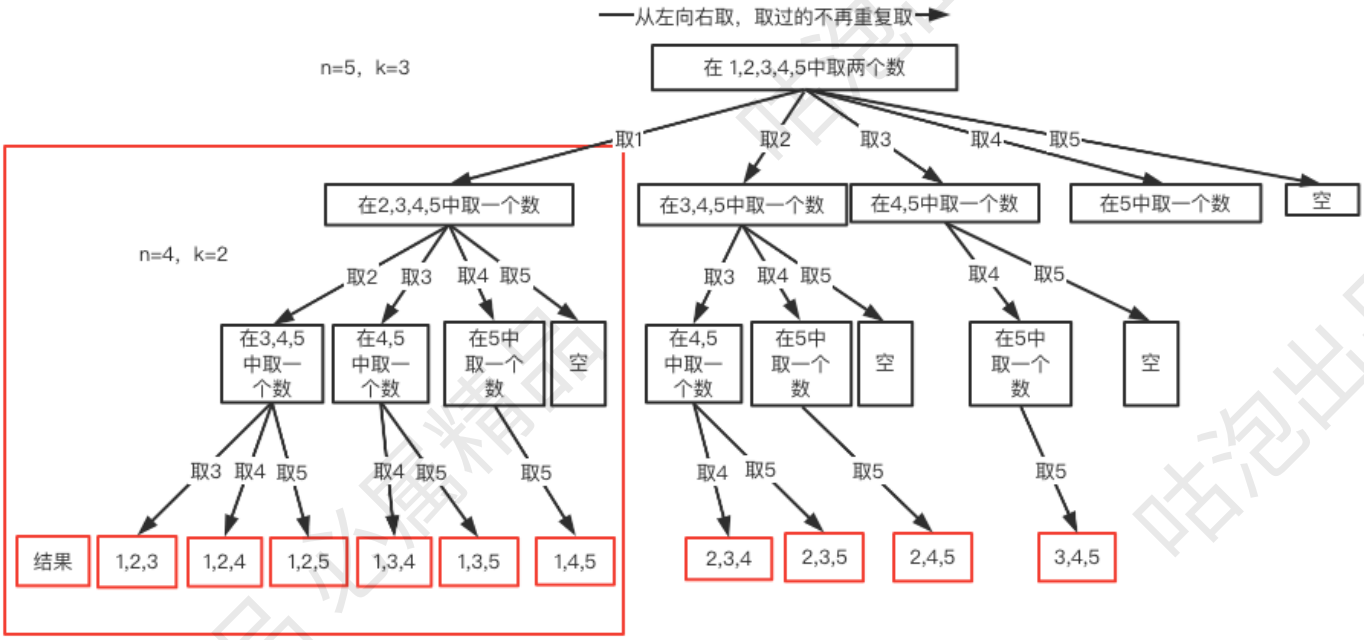
n=4时，我们可以选择的n有 {1,2,3,4}这四种情况，所以我们从第一层到第二层的分支有四个，分别表示可以取1，2，3，4。而且这里 从左向右取数，取过的数，不在重复取。 第一次取1，集合变为2，3，4 ，因为k为2，我们只需要再取一个数就可以了，分别取2，3，4，得到集合[1,2] [1,3] [1,4]，以此类推。



每次从集合中选取元素，可选择的范围会逐步收缩，到了取4时就直接为空了。

继续观察树结构，可以发现，图中每次访问到一次叶子节点(图中红框标记处)，我们就找到了一个结果。虽然最后一个是空，但是不影响结果。这相当于只需要把从根节点开始每次选择的内容（分支）达到叶子节点时，将其收集起来就是想要的结果。

如果感觉不明显，我们再画一个n=5，k=3的例子：



从图中我们发现元素个数n相当于树的宽度（横向），而每个结果的元素个数k相当于树的深度（纵向）。所以我们说回溯算法就是一纵一横而已。再分析，我们还发现几个规律：

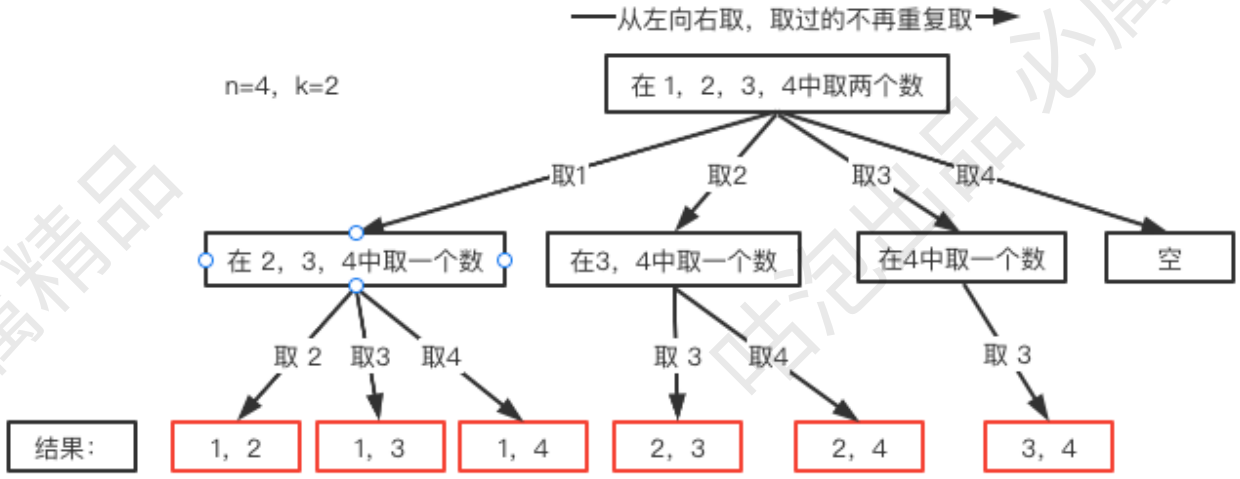
① 我们每次选择都是从类似{1,2,3,4}，{1,2,3,4,5}这样的序列中一个个选的，这就是局部枚举，而且越往后枚举范围越小。

② 枚举之后，我们要一个个暴力测试一遍的，从上图可以看到，这就是N叉树遍历的过程，因此两者代码也必然很像。

③ 我们再看上图中红色大框起来的部分，这个部分的执行过程与n=4，k=2的处理过程完全一致，很明显这是个可以递归的子结构。

这样我们就将回溯与N叉树的完美结合在一起了。

到此，还有一个大问题没有解决，回溯一般会有个手动撤销的操作，为什么要这样呢？继续观察纵横图：



我们可以看到，我们收集每个结果不是针对叶子结点的，而是针对树枝的，比如最上层我们首先选了1，下层如果选2，结果就是{1,2}，如果下层选了3，结果就是{1,3}，依次类推。
现在的问题是当我们得到第一个结果{1,2}之后，怎么得到第二个结果{1,3}呢？

继续观察纵横图，可以看到，我可以在得到{1,2}之后将2撤掉，再继续取3，这样就得到了{1,3}，同理可以得到{1,4}，之后当前层就没有了，我们可以将1撤销，继续从最上层取2继续进行。

现在明白为什么要手动撤销了吧，这个过程，我称之为"放下前任，继续前进"，后面所有的回溯问题都是这样的思路。

这几条就是回溯的基本规律，明白之后，一切都变得豁然开朗。如果还是不太明白，我们在1.4 小节用更完整的图示解释该过程。

还有问题要解决，我们要完整的保存结果，所以我们可以将找到的放在一个List里，因为要返回多个结果，最后的结果是二维数组结构，所以在方法的参数里，我们要放一个List<List>类型的res变量。

到此我们就可以写出完整的回溯代码了：

```
def combine(self, n, k):
        # 思路：回溯法，选好一个值，再选下一个值时，让start值加1， 让k值减1
        ans = []
        cur = []
        self.back_track(ans, cur, 1, n, k)
        return ans

    def back_track(self, ans, cur, start, n, k):
        # 递归终止条件
        if k == 0:
            ans.append(cur.copy())
            # cur = [] # 易错点：不用清空，回溯的过程会不断压入弹出
            return
        # 回溯，注意这里的range范围
        for i in range(start, n+1):
            cur.append(i)
            self.back_track(ans, cur, i+1, n, k-1)
            cur.pop()
```

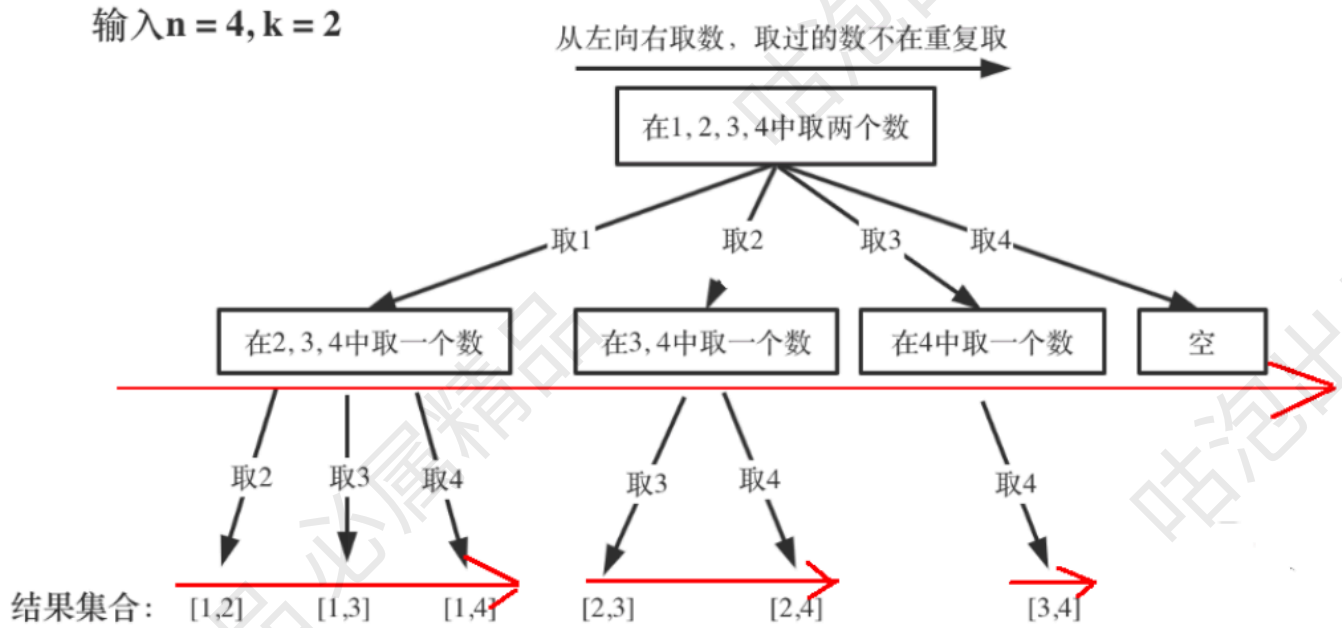
上面代码还有个问题要解释一下：startIndex和i是怎么变化的，为什么传给下一层时要加1。

深入分析startIndex的变化

我们可以看到在递归里有个循环

```
for (int i = startIndex; i <= n; i++) {
   dfs(n,k,i+1,path,res);
}
```

这里的循环有什么作用呢？看一下图就知道了，这里其实就是枚举，第一次n=4，可以选择1 ，2，3，4四种情况，所以就有四个分支，for循环就会执行四次：



而对于第二层第一个，选择了1之后，剩下的元素只有2 ，3， 4了，所以这时候for循环就执行3次，后面的则只有2次和1次。

1.4 图解为什么有个撤销的操作

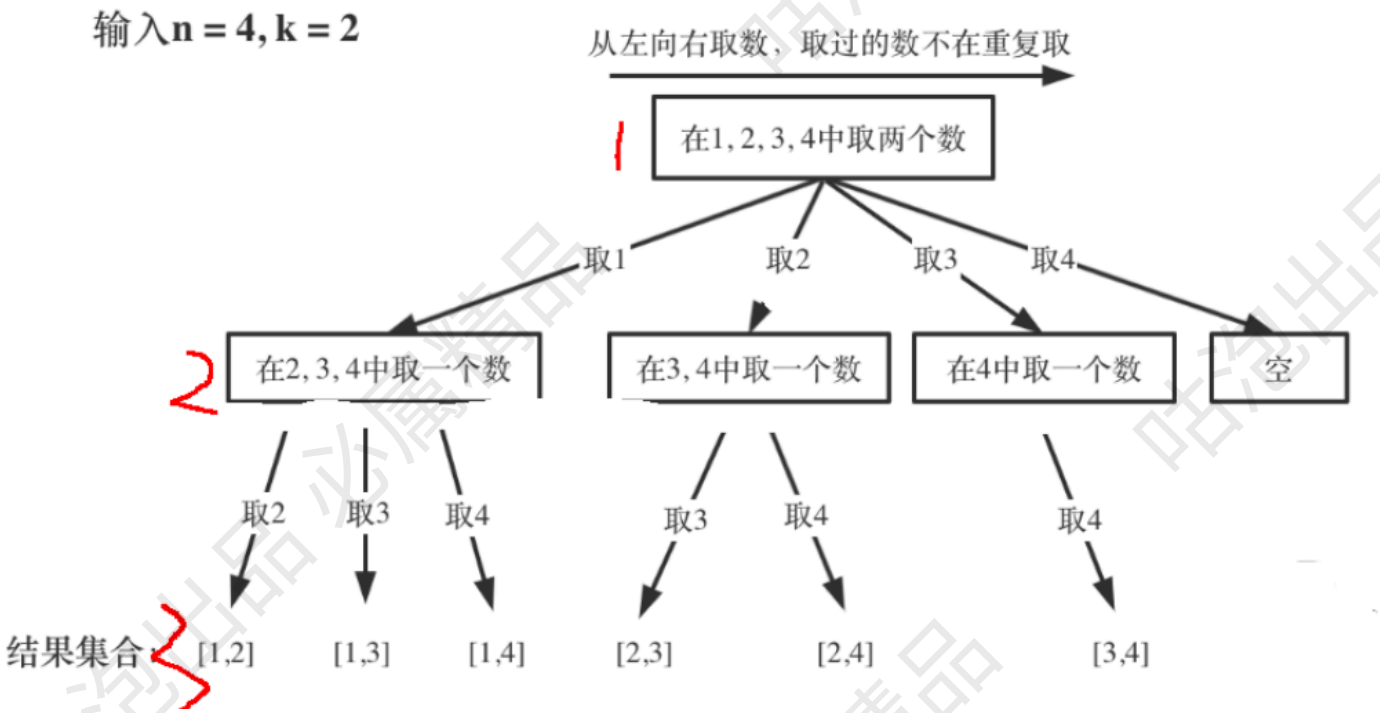
如果你已经明白上面为什么会有撤销过程，这一小节就不必看了。如果还是不懂，本节就用更详细的图示带你看一下。

回溯最难理解的部分是这个回溯过程，而且这个过程即使调试也经常会晕：

```
path.addLast(i);
dfs(n, k, i + 1, path, res);
path.removeLast();
```

最后为什么要remove这个撤销操作呢？我们可以对照下面的图看一下。当第一层取1时，最底层的边从左向右依次执行“取2”、“取3”和“取4”，而取3的时候，此时list里面存的是上一个结果<1,2>，所以必须提前将2撤销，这就path.removeLast();的作用。

用我们拆解递归的方法，将递归拆分成函数调用，输出第一条路径{1,2}的步骤如下如下：



我们在递归章节说过，递归是“不撞南墙不回头”，回溯也一样，接下来画代码的执行图详细看一下其过程，图中的手绘的序号是执行过程:

然后呢？{1,2}输出 之后会怎么执行呢？回归之后，假如我们将remove代码去掉，也就是这样子：

注意上面的4号位置结束之后，当前递归就结束了，然后返回到上一层继续执行for循环体，也就是上面的5。进入5之后，接着开始执行第6步：path.addLast(i)了，此时path的大小是3，元素是{1,2,3}，为什么会这样呢?

因为path是一个全局的引用，各个递归函数共用的，所以当{1,2}处理完之后，2污染了path变量。我们希望将1保留而将2干掉，然后让3进来，这样才能得到{1,3}，所以这时候需要手动remove一下。

同样3处理完之后，我们也不希望3污染接下来的{1,4}，1全部走完之后也不希望1污染接下来的{2,3}等等，这就是为什么回溯里会在递归之后有一个remove撤销操作。

2.高频面试题

先补充回溯算法的一个很大功能：能够获得完整的路径，并且能获得所有满足要求的路径，而我们后面要学习的动态规划一般只能获得最终的结果，无法获得完整的路径，我们看动态规划时再看。
既然能够获得完整路径，那在《递归和树》一章研究的二叉树专题中有个找路径的专题是不是可以做热身呢？是的！就是回溯的热身问题。
之后就研究一些回溯的经典问题，主要有组合、分割、子集、排列，棋盘等问题。虽然题目有难有易，但解题思路基本一致，也都是一个模板解题,所以我们整章都是在练习怎么梳理使用这个模板。
考虑到具体的题目有大量特定的细节需要处理，我们仍然分专题来讲解。

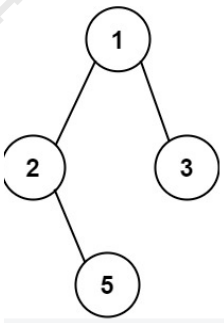
2.1 热身：二叉树中的路径问题

2.1.1 输出二叉树的所有路径

LeetCode257：给你一个二叉树的根节点root ， 按任意顺序 ， 返回所有从根节点到叶子节点的路径。

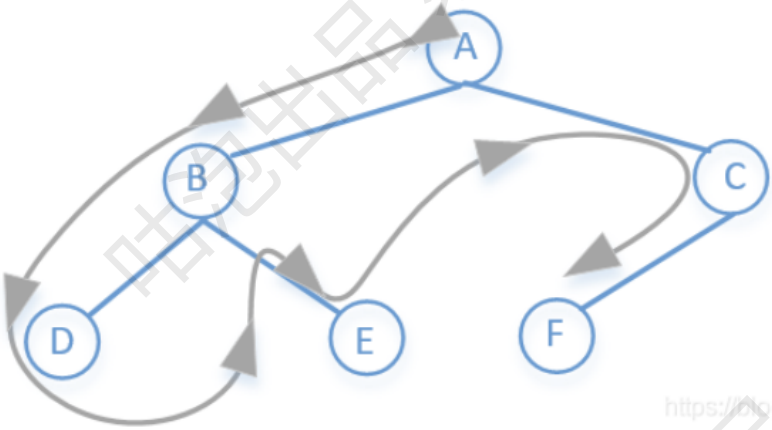
叶子节点是指没有子节点的节点。

```
示例：
输入：root = [1,2,3,null,5]
输出：["1->2->5","1->3"]
```



我们可以注意到有几个叶子节点，就有几条路径，那如何找叶子节点呢？我们知道深度优先搜索就是从根节点开始一直找到叶子结点，我们这里可以先判断当前节点是不是叶子结点，再决定是不是向下走，如果是叶子结点，我们就增加一条路径。

我们现在从回溯的角度来分析，得到第一条路径ABD之后怎么找到第二条路径ABE，这里很明显就是先将D撤销，然后再继续递归就可以了




```
class Solution(object):
    def binaryTreePaths(self, root):
        def get_paths(root, path, res):
            if root:
                path.append(str(root.val))
                left = get_paths(root.left, path, res)
                right = get_paths(root.right, path, res)
                if not left and not right:                  # 如果root是叶子结点
                    res.append("->".join(path))             # 把当前路径加入到结果列表中
                path.pop()                                  # 返回上一层递归时，要让当前路径
恢复原样
                return True
```

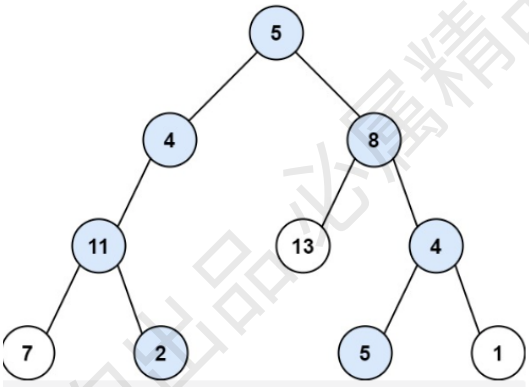
```
        res = []
        get_paths(root, [], res)
        return res
```

读者可以与《树和递归》一章中介绍的方法来对比一下，看看两种递归方式有什么区别。

2.1.1 路径总和问题

同样的问题是LeetCode113题，给你二叉树的根节点 root 和一个整数目标和 targetSum ，找出所有从根节点到叶子节点 路径总和等于给定目标和的路径。
叶子节点 是指没有子节点的节点。

```
示例1：
输入：root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
输出：[[5,4,11,2],[5,8,4,5]]
```



本题怎么做呢？我们直接观察题目给的示意图即可，要找的targetSum是22。我们发现根节点是5，因此只要从左侧或者右侧找到targetSum是17的即可。
继续看左子树，我们发现值为4，那只要从node(4)的左右子树中找targetSum是13即可，依次类推，当我们到达node(11)时，我们需要再找和为2的子链路，显然
此时node(7)已经超了，不是我们要的,此时就要将node(7)给移除掉，继续访问node(2).

同样在根结点的右侧，我们也要找总和为17的链路，方式与上面的一致。完整代码就是：

```
class Solution:
    def pathSum(self, root: TreeNode, targetSum: int) -> List[List[int]]:
        ret = list()
        path = list()

        def dfs(root: TreeNode, targetSum: int):
            if not root:
                return
            path.append(root.val)
            targetSum -= root.val
            if not root.left and not root.right and targetSum == 0:
                ret.append(path[:])
```

```
            dfs(root.left, targetSum)
            dfs(root.right, targetSum)
            path.pop()

        dfs(root, targetSum)
        return ret
```

2.2 组合总和问题

LeetCode39题目要求：给你一个无重复元素的整数数组candidates和一个目标整数 target ，找出 candidates 中可以使数字和为目标数 target 的 所有不同组合 ，并以列表形式返回。你可以按任意顺序返回这些组合。

candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。
数组中的元素满足1 <= candidates[i] <= 200。例子：

```
输入：candidates = [2,3,6,7], target = 7
输出：[[2,2,3],[7]]
解释：
2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意2可以使用多次。
7 也是一个候选， 7 = 7 ，仅有这两种组合。
```

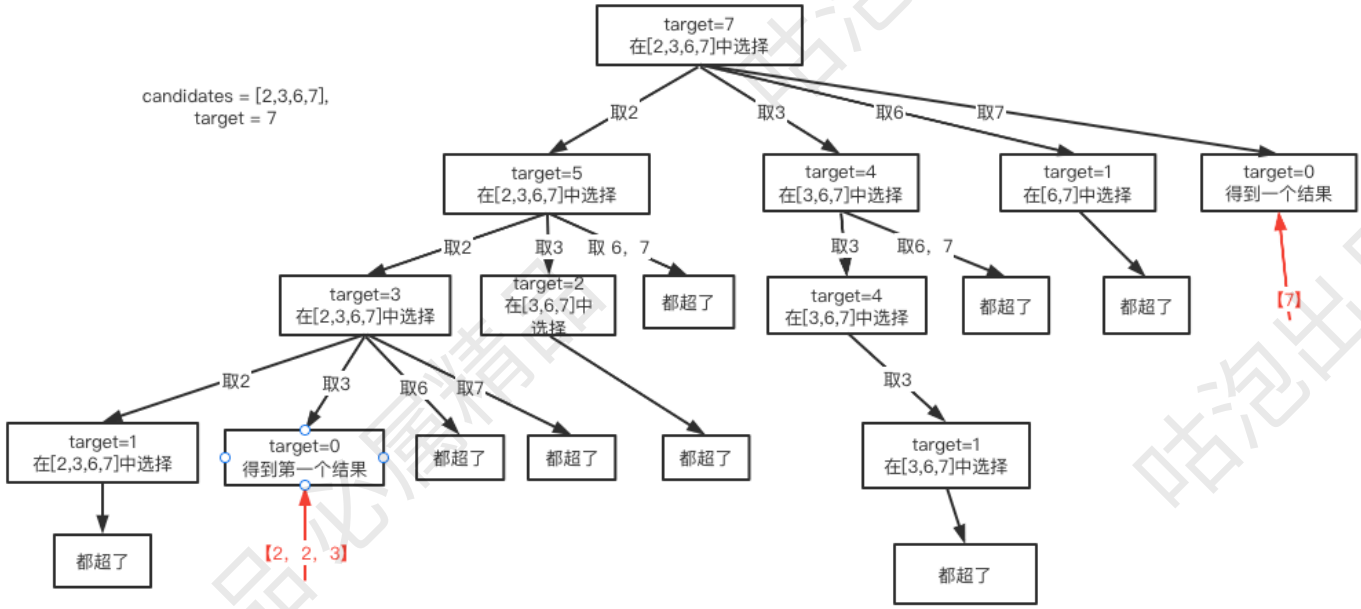
如果不考虑重复，本题与LeetCode113题就是一个题，如果可以重复，那是否会无限制取下去呢？也不会，因为题目给了说明，每个元素最小为1，因此最多也就target个1。

我们画图看该怎么做，对于序列{2,3,6,7}，target=7。很显然我们可以先选择一个2，然后剩下的target就是7-2=5。再选一个2，剩余5-2=3。之后再选一个2，剩余3-2=1。已经小于2了，我们不能继续向下了，要返回一下。看看有没有3。OK，序列中有3，那么就得到了第一个结果{2,2,3}。

之后我们继续回退到只选了一个2的时候，这时候不能再取2了，而是从{3,6,7}中选择，如下图所示，没有符合要求的！

依次类推，后面尝试从3、6和7开始选择。

所以我们最终得到的结果就是{2,2,3}和{2,5}。为了方便，我们可以先对元素做个排序，然后将上面的过程画成这个一个树形图：



这个图横向是针对每个元素的暴力枚举，纵向是递归，也是一个纵横问题，实现代码也不复杂：

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        res = []
        path = []
        def backtrack(candidates,target,sum,startIndex):
            if sum > target: return
            if sum == target: return res.append(path[:])
            for i in range(startIndex,len(candidates)):
                #如果 sum + candidates[i] > target 就终止遍历
                if sum + candidates[i] >target: return
                sum += candidates[i]
                path.append(candidates[i])
                #startIndex = i:表示可以重复读取当前的数
                backtrack(candidates,target,sum,i)
                sum -= candidates[i]  #回溯
                path.pop()  #回溯
        candidates = sorted(candidates)  #需要排序
        backtrack(candidates,target,0,0)
        return res
```

2.3 分割问题

分割问题也是回溯要解决的典型问题之一，常见的题目有分割回文串、分割IP地址，以及分割字符串等。

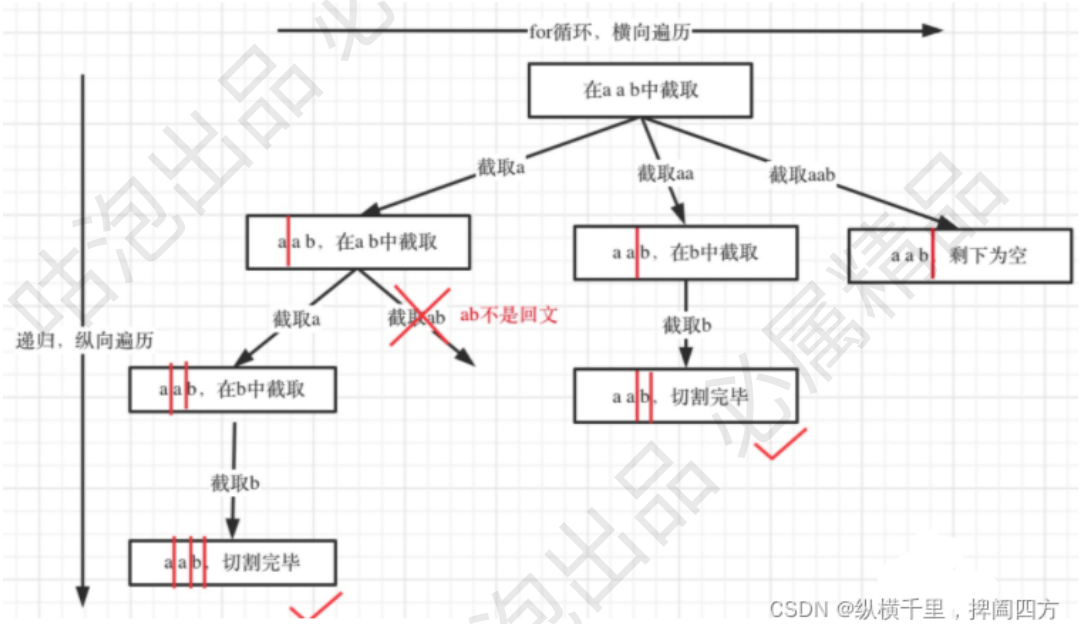
2.3.1 分割回文串

LeetCode131 分割回文串，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。

回文串是正着读和反着读都一样的字符串。

```
示例1：
输入：s = "aab"
输出：[["a","a","b"],["aa","b"]]
```

字符串如何判断回文本身就是一个道算法题，本题在其之上还要再解决一个问题：如何切割？如果暴力切割，是非常困难的，如果从回溯的角度来思考就清晰很多：



我们说回溯本身仍然会进行枚举，这里的也一样。切割线（就是图中的红线）切割到字符串的结尾位置，说明找到了一个切割方法。这里就是先试一试,第一次切'a',第二次切'aa',第三次切'aab'。这对应的就是回溯里的for循环，也就是横向方面。

我们还说回溯仍然会进行递归，这里也是一样的，第一次切了'a',剩下的就是'ab'。递归就是再将其再切一个回文下来，也就是第二个'a',剩下的'b'再交给递归进一步切割。这就是纵向方面要干的事情，其他以此类推。

至于回溯操作与前面是一样的道理，不再赘述。通过代码就可以发现，切割问题的回溯搜索的过程和组合问题的回溯搜索的过程是差不多的。

回溯+函数判断回文串

```
class Solution:
    def __init__(self):
        self.paths = []
        self.path = []

    def partition(self, s: str) -> List[List[str]]:
        '''
        当切割线迭代至字符串末尾，说明找到一种方法
```

```
        类似组合问题，为了不重复切割同一位置，需要start_index来做标记下一轮递归的起始位置(切割线)
        '''
        self.path.clear()
        self.paths.clear()
        self.backtracking(s, 0)
        return self.paths

    def backtracking(self, s: str, start_index: int) -> None:
        # Base Case
        if start_index >= len(s):
            self.paths.append(self.path[:])
            return

        # 单层递归逻辑
        for i in range(start_index, len(s)):
            # 此次比其他组合题目多了一步判断：
            # 判断被截取的这一段子串([start_index, i])是否为回文串
            if self.is_palindrome(s, start_index, i):
                self.path.append(s[start_index:i+1])
                self.backtracking(s, i+1)    # 递归纵向遍历：从下一处进行切割，判断其余是否仍为回
文串
                self.path.pop()              # 回溯
            else:
                continue

    def is_palindrome(self, s: str, start: int, end: int) -> bool:
        i: int = start
        j: int = end
        while i < j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True
```

2.3.2 复原IP地址

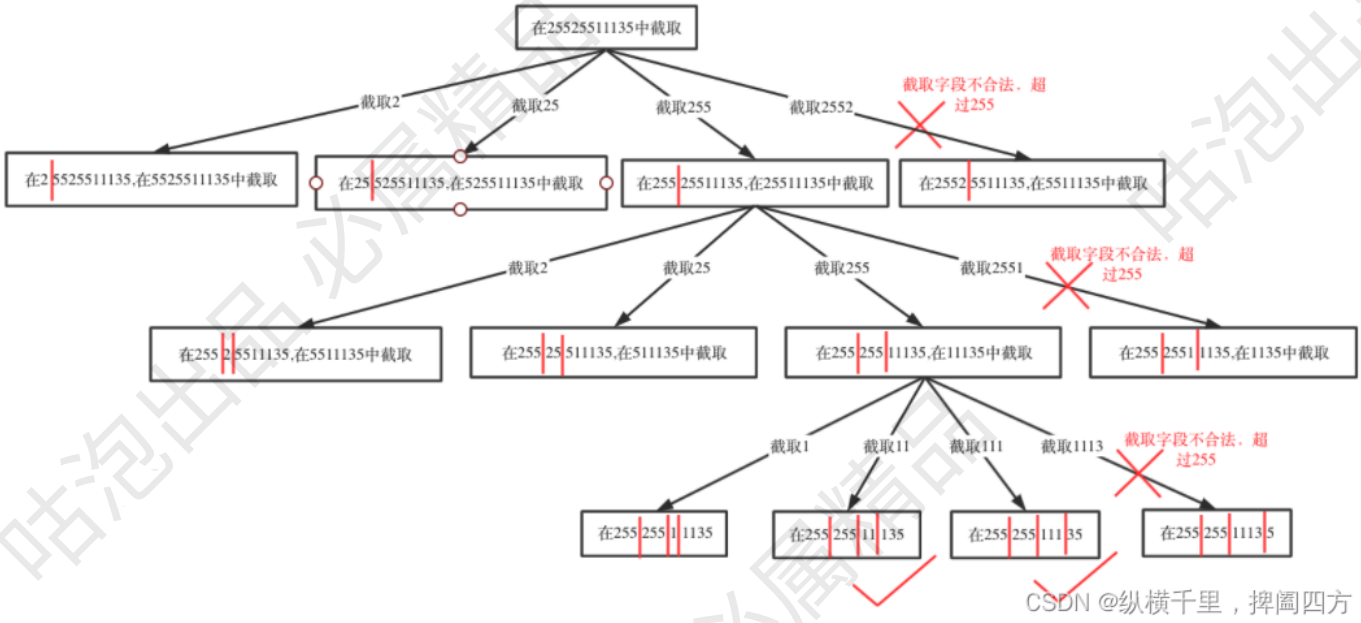
这也是一个经典的分割类型的回溯问题。LeetCode93.有效IP地址正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔。

例如："0.1.2.201" 和 "192.168.1.1" 是有效 IP 地址，但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是无效IP地址。
给定一个只包含数字的字符串s，用以表示一个IP地址，返回所有可能的有效IP地址，这些地址可以通过在s中插入 '.' 来形成。你不能重新排序或删除s中的任何数字。你可以按任何顺序返回答案。

```
示例1：
输入：s = "25525511135"
输出：["255.255.11.135","255.255.111.35"]
```

该问题的思路与与前面的分割回文串基本一致，也是切割问题。回溯的第一步就是使用枚举将所有可能性搜出来,找到一个符合要求的就先切下来，后面的部分继续进行枚举和切割，如果到了最后发现不符合要求，则开始回溯。

本题的难度明显比上一题要大，主要是判断是否合法的要求更高了，比如第一个元素我们可以截取2、25、255、2552，很显然到了2552之后就不合法了，此时就要回溯。后面也一样，假如我们第一层截取的是2，第二层就从”5525511135“中截取，此时可以有5，55，552，显然552已经不合法了，依次类推。
画出图来就如下所示：



当然这里还要判断是0的情况等等，在字符串转换成数字一章，我们讲解了很多种要处理的情况，为此我们可以可以写一个方法单独来执行相关的判断。代码如下：

```
// 判断字符串s在左闭又闭区间[start, end]所组成的数字是否合法
        def is_valid(self, s: str, start: int, end: int) -> bool:
        if start > end: return False
        # 若数字是0开头，不合法
        if s[start] == '0' and start != end:
            return False
        if not 0 <= int(s[start:end+1]) <= 255:
            return False
        return True
```

另外，IP地址只有四段，不是无限分割的，因此本题只会明确的分成4段，不能多也不能少。所以不能用切割线切到最后作为终止条件，而是分割的段数到了4就必须终止。考虑到我们构造IP地址时还要手动给添加三个小数点，所以我们用变量pointNum来表示小数点数量，pointNum为3说明字符串分成了4段了。

要手动添加一个小数点，这要增加一个位置来存储，所以下一层递归的startIndex要从i+2开始。其他的主要工作就是递归和回溯的过程了。这里的撤销部分要注意将刚刚加入的分隔符删掉，并且pointNum也要-1，完整代码如下：

```
class Solution:
    def __init__(self):
        self.result = []
```

```
    def restoreIpAddresses(self, s: str) -> List[str]:
        '''
        本质切割问题使用回溯搜索法，本题只能切割三次，所以纵向递归总共四层
        因为不能重复分割，所以需要start_index来记录下一层递归分割的起始位置
        添加变量point_num来记录逗号的数量[0,3]
        '''
        self.result.clear()
        if len(s) > 12: return []
        self.backtracking(s, 0, 0)
        return self.result

    def backtracking(self, s: str, start_index: int, point_num: int) -> None:
        # Base Case
        if point_num == 3:
            if self.is_valid(s, start_index, len(s)-1):
                self.result.append(s[:])
            return
        # 单层递归逻辑
        for i in range(start_index, len(s)):
            # [start_index, i]就是被截取的子串
            if self.is_valid(s, start_index, i):
                s = s[:i+1] + '.' + s[i+1:]
                self.backtracking(s, i+2, point_num+1)  # 在填入.后，下一子串起始后移2位
                s = s[:i+1] + s[i+2:]    # 回溯
            else:
                # 若当前被截取的子串大于255或者大于三位数，直接结束本层循环
                break

    def is_valid(self, s: str, start: int, end: int) -> bool:
        if start > end: return False
        # 若数字是0开头，不合法
        if s[start] == '0' and start != end:
            return False
        if not 0 <= int(s[start:end+1]) <= 255:
            return False
        return True
```

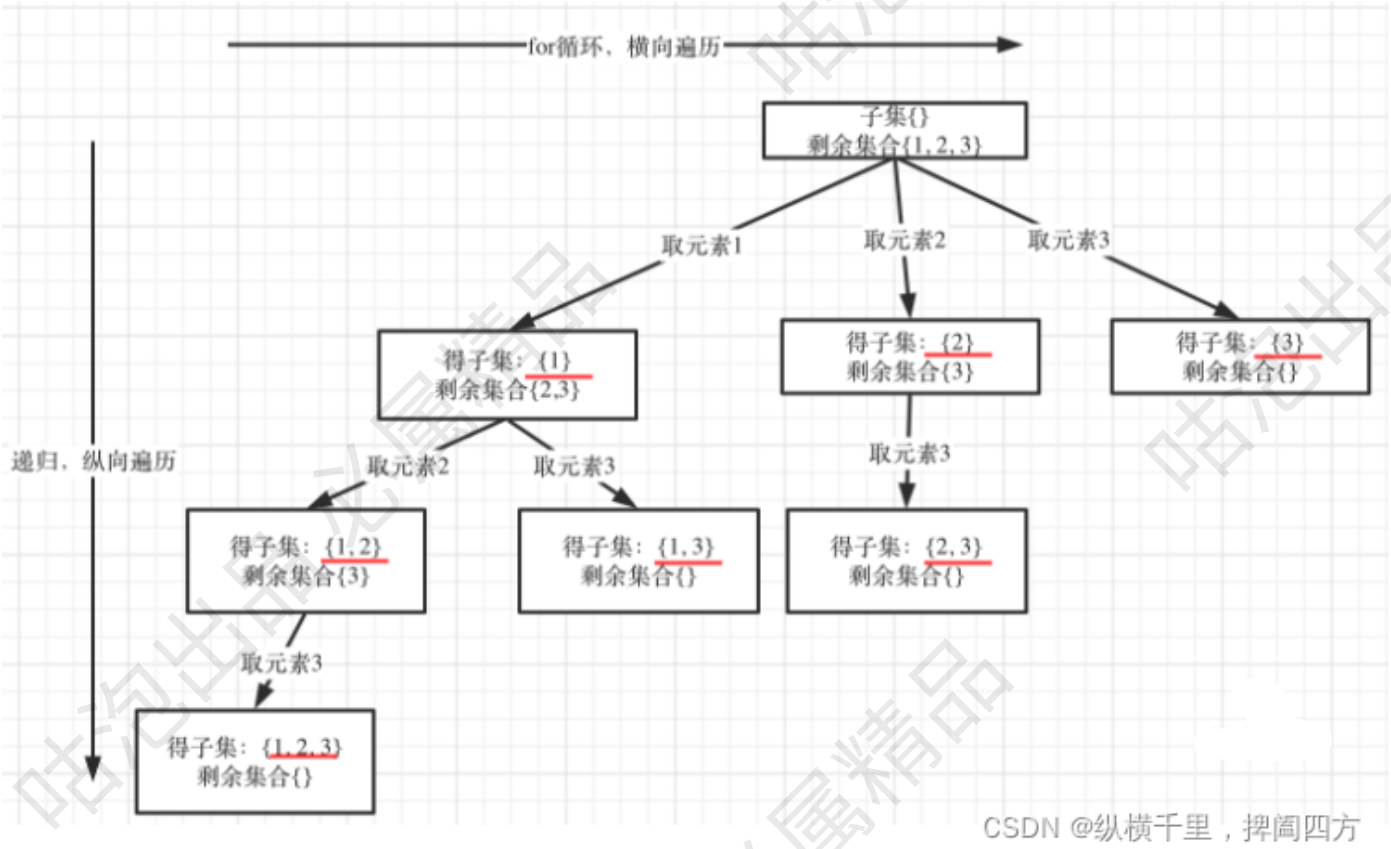
2.4 子集问题

子集问题也是回溯的经典使用场景。回溯可以画成一种树状结构，子集、组合、分割问题都可以抽象为一棵树，但是子集问题与其他的类型有个明显的区别，组合问题一般找到满足要求的结果即可，而集合则要找出所有的情况。

LeetCode78，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：
输入：nums = [1,2,3]
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

看上面的例子nums = [1,2,3]，将子集抽象为树型结构如下：



从图中红线部分，可以看出遍历这个树的时候，把所有节点都记录下来，就是要求的子集集合。

这里什么时候要停下来呢？其实可以不加终止条件，因为startIndex >= nums.size()，本层for循环本来也结束了。

而且求取子集问题，不需要任何剪枝！因为子集就是要遍历整棵树。这样实现起来也比较容易。

```
class Solution:
    def __init__(self):
        self.path: List[int] = []
        self.paths: List[List[int]] = []

    def subsets(self, nums: List[int]) -> List[List[int]]:
        self.paths.clear()
        self.path.clear()
        self.backtracking(nums, 0)
        return self.paths

    def backtracking(self, nums: List[int], start_index: int) -> None:
        # 收集子集，要先于终止判断
        self.paths.append(self.path[:])
        # Base Case
        if start_index == len(nums):
            return

        # 单层递归逻辑
        for i in range(start_index, len(nums)):
```

```
            self.path.append(nums[i])
            self.backtracking(nums, i+1)
            self.path.pop()     # 回溯
```

上面代码里定义了全局变量result和path，这样整体比较简洁。这里可以转换成方法参数的形式，只是看起来比较复杂一些。

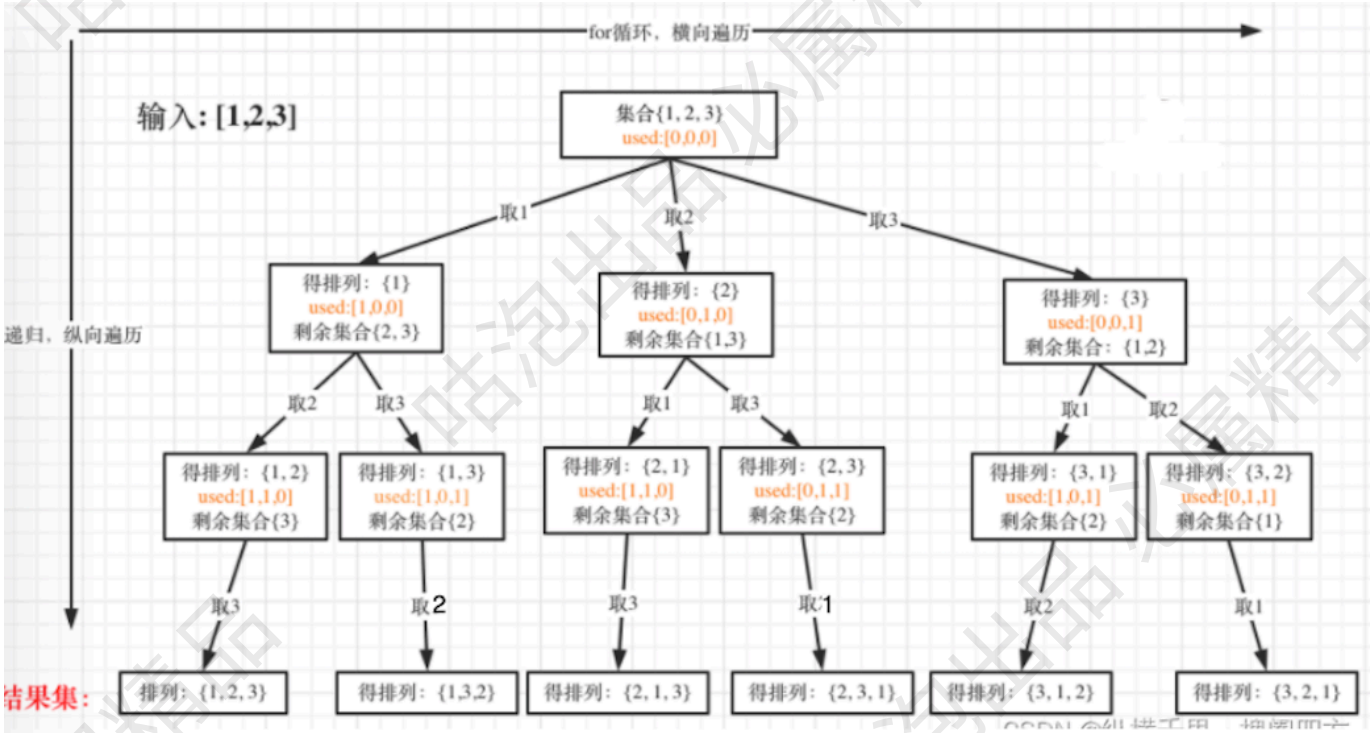
2.5 排列问题

LeetCode46.给定一个没有重复数字的序列，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] ]
```

排列问题是典型的小学生都会，但是难道众人的问题。这个问题与前面组合等问题的一个区别是使用过的后面还要再用，例如1，在开始使用了，但是到了2 和3的时候仍然要再使用一次。这本质上是因为 [1,2] 和 [2,1] 从集合的角度看是一个，但从排列的角度看是两个。

元素1在[1,2]中已经使用过了，但是在[2,1]中还要再使用一次，所以就不能使用startIndex了，为此可以使用一个used数组来标记已经选择的元素，完整过程如图所示:



这里的终止条件怎么判断呢？从上图可以看出叶子节点就是结果。那什么时候是到达叶子节点呢?

当收集元素的数组path的大小达到和nums数组一样大的时候，说明找到了一个全排列，也表示到达了叶子节点。

```
class Solution:
    def __init__(self):
        self.path = []
        self.paths = []
```

```
    def permute(self, nums: List[int]) -> List[List[int]]:
        '''
        因为本题排列是有序的，这意味着同一层的元素可以重复使用，但同一树枝上不能重复使用(usage_list)
        所以处理排列问题每层都需要从头搜索，故不再使用start_index
        '''
        usage_list = [False] * len(nums)
        self.backtracking(nums, usage_list)
        return self.paths

    def backtracking(self, nums: List[int], usage_list: List[bool]) -> None:
        # Base Case本题求叶子节点
        if len(self.path) == len(nums):
            self.paths.append(self.path[:])
            return

        # 单层递归逻辑
        for i in range(0, len(nums)):  # 从头开始搜索
            # 若遇到self.path里已收录的元素，跳过
            if usage_list[i] == True:
                continue
            usage_list[i] = True
            self.path.append(nums[i])
            self.backtracking(nums, usage_list)     # 纵向传递使用信息，去重
            self.path.pop()
            usage_list[i] = False
```

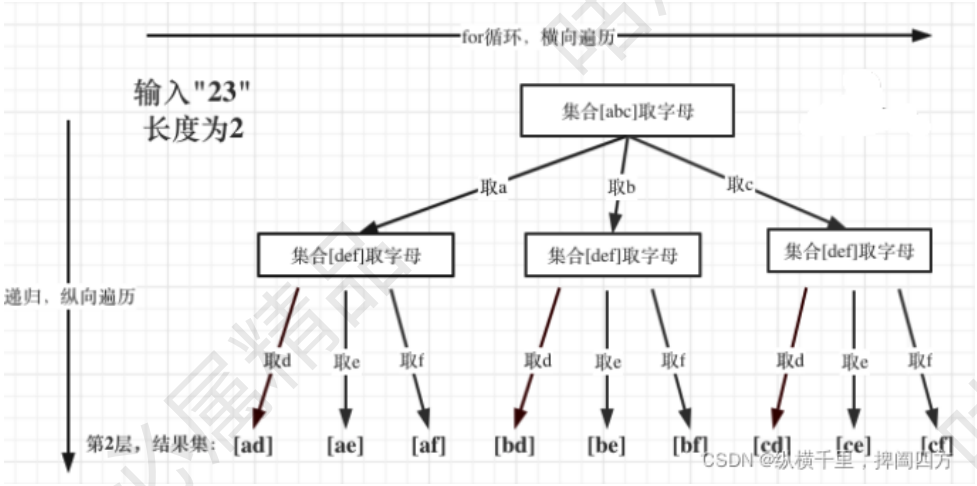
2.6 电话号码问题

LeetCode17.电话号码组合问题，也是热度非常高的一个题目，给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。 给出数字到字母的映射如下(与电话按键相同)。注意 1 不对应任何字母，9对应四个字母。


```
示例1:
输入: digits = "23"
输出: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

我们说回溯仍然会存在暴力枚举的情况，这个题就很典型，例如如果输入23，那么，2就有 a、b、c三种情况，3有d、e、f三种情况。组合一下就一共就有3*3=9种，如果是233，那么就是27种。

这里要注意的9对应4个字母，而1则没有，那该怎么建立字母和数字之间的映射呢？我们用一个list或者map来保存，而不写一堆的if else。接下来我们就用回溯来解决n层循环的问题，输入23对应的树就是这样的：



树的深度就是输入的数字个数，例如输入23，树的深度就是2。而所有的叶子节点就是我们需要的结果["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]。所以这里的终止条件就是，如果当前执行的index 等于输入的数字个数（digits.size）了。

使用for循环来枚举出来，然后循环体内就是回溯过程了。基本实现过程如下：

```
class Solution:
    def __init__(self):
        self.answers: List[str] = []
        self.answer: str = ''
        self.letter_map = {
            '2': 'abc',
            '3': 'def',
            '4': 'ghi',
            '5': 'jkl',
            '6': 'mno',
            '7': 'pqrs',
            '8': 'tuv',
            '9': 'wxyz'
        }

    def letterCombinations(self, digits: str) -> List[str]:
        self.answers.clear()
        if not digits: return []
        self.backtracking(digits, 0)
        return self.answers

    def backtracking(self, digits: str, index: int) -> None:
        # 回溯函数没有返回值
        if index == len(digits):    # 当遍历穷尽后的下一层时
            self.answers.append(self.answer)
            return
        # 单层递归逻辑
        letters: str = self.letter_map[digits[index]]
```

```
        for letter in letters:
            self.answer += letter   # 处理
            self.backtracking(digits, index + 1)    # 递归至下一层
            self.answer = self.answer[:-1]  # 回溯
```

2.8 括号生成问题

本题是一道非常典型的回溯问题，LeetCode22.数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

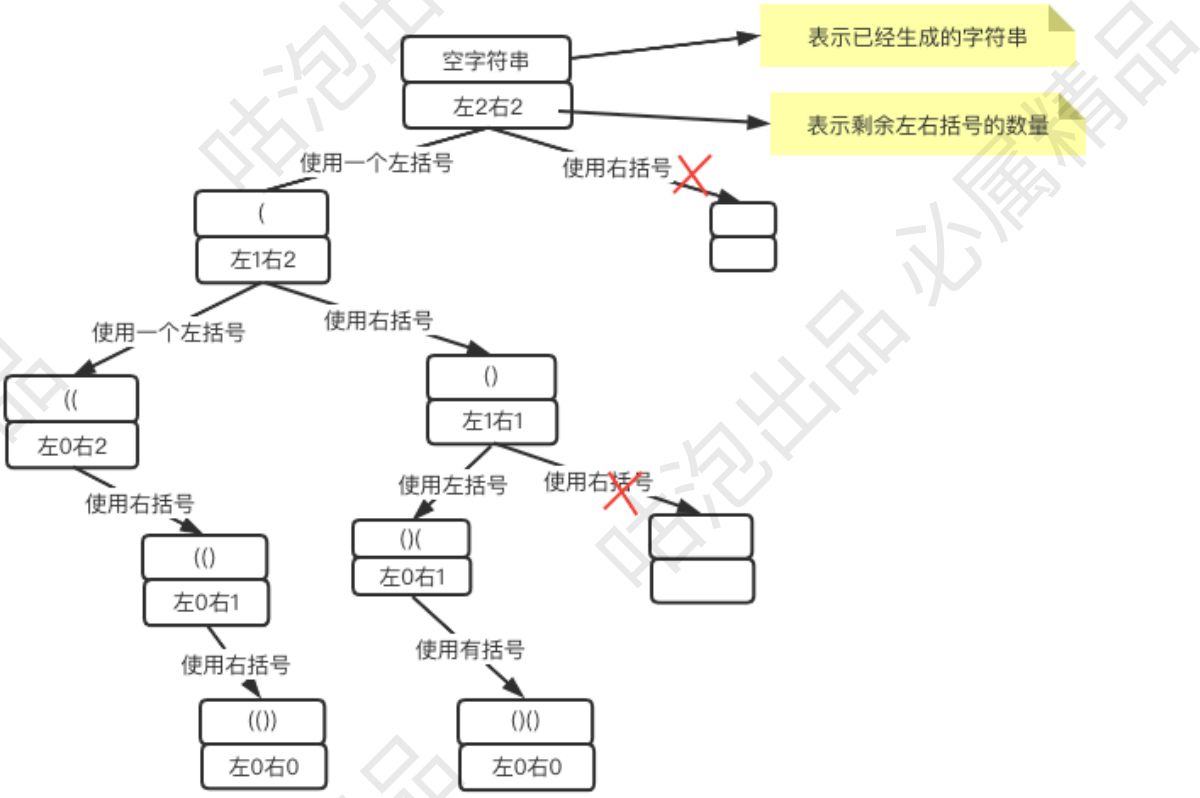
```
示例1：
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

要解决该问题，我们首先要明确一个问题，左右括号什么时候可以获得。我们知道左括号出现的数量一定等于n，观察题目给的示例，只要剩余左括号的数量大于0，就可以添加"("，例如"("、"(()"、"(()("、"()(("都是可以的，因为我们只要再给加几个右括号就行，例如可以将其变成"()"、"(())"、"(()()"、"()(())"。

那添加右括号的要求呢？结论是：序列中左括号的数量必须大于右括号的数量，例如上面的"("、"(()"、"(()("、"()((“，都可以，但是如果")"、"())"、")()("、")(()"就不可以了，此时的情况都是右括号数量大于等于左括号。所以我们可以得到两条结论：

- 1.只要剩余左括号的数量大于0，就可以添加"("。
- 2.序列中左括号的数量必须大于右括号的数量才可以添加"("，并且")"的剩余数量大于0。

接下来看如何用回溯解决，我们将添加"("视为left，将")"视为right，这样"("和")"各可以出现n次。我们以n=2为例画一下的图示：



图中红叉标记的位置是左括号大于有括号了，不能再向下走了，这个操作也称为剪枝。通过图示，可以得到如下结论：

- 当前左右括号都有大于 0 个可以使用的时候，才产生分支，否则就直接停止。
- 产生左分支的时候，只看当前是否还有左括号可以使用；
- 产生右分支的时候，除了要求还有右括号，还要求剩余右括号数量一定大于左括号的数量；
- 在左边和右边剩余的括号数都等于 0 的时候结束。

而且，从上图可以看到，不管是剪枝还是得到一个结果，返回的过程仍然可以通过回溯来实现：

```
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> ans = new ArrayList<String>();

        backtrack(ans, new StringBuilder(), 0, 0, n);
        return ans;
    }

    /**
     * @param ans 当前递归得到的结果
     * @param cur 当前的括号串
     * @param open   左括号已经使用的个数
     * @param close  右括号已经使用的个数
     * @param max    序列长度最大值
     */
    public void backtrack(List<String> ans, StringBuilder cur, int open, int close, int
max) {
        if (cur.length() == max * 2) {
            ans.add(cur.toString());
            return;
        }
        //本题需要两次回溯，比较少见的情况
        if (open < max) {
            cur.append('(');
            backtrack(ans, cur, open + 1, close, max);
            cur.deleteCharAt(cur.length() - 1);
        }
        if (close < open) {
            cur.append(')');
            backtrack(ans, cur, open, close + 1, max);
            cur.deleteCharAt(cur.length() - 1);
        }
    }
}
```

2.9 字母大小写全排列

LeetCode784. 字母大小写全排列：给定一个字符串 s ，通过将字符串 s 中的每个字母转变大小写，我们可以获得一个新的字符串。
返回所有可能得到的字符串集合。以任意顺序返回输出。

```
示例1：输入：s = "a1b2"
输出：["a1b2", "a1B2", "A1b2", "A1B2"]
```

如果本题去掉数字，只告诉你两个字母ab，让你每个字母变化大小写，那就是ab、Ab、aB、AB四种情况，题目比2.6电话号码问题还简单，这里的数字就是干扰项，我们需要做的是过滤掉数字， 只处理字母，代码如下：

```
class Solution:
    def letterCasePermutation(self, s: str) -> List[str]:
        ans = []
        def dfs(s: List[str], pos: int) -> None:
            while pos < len(s) and s[pos].isdigit():
                pos += 1
            if pos == len(s):
                ans.append(''.join(s))
                return
            dfs(s, pos + 1)
            s[pos] = s[pos].swapcase()
            dfs(s, pos + 1)
            s[pos] = s[pos].swapcase()
        dfs(list(s), 0)
        return ans
```

3.总结

我们本章分析了很多回溯问题，这些问题基本都是一个思路，一个模板，所以只要我们勤加锻炼，完全可以征服。回溯的难题还是不少的，例如在2.2，2.4，2.5中假如有重复元素该怎么处理呢？这正好对应了LeetCode的40、90和47、491题，感兴趣的同学可以再研究一下。

另外还有几个典型的游戏题目也能用回溯，例如LeetCode37 数独问题，51 N皇后问题，529扫雷问题等等，这些题目需要考虑的条件比较多，难度比较大，面试遇到的可能性不高，感兴趣的同学可以自己研究一下。