0093. 复原 IP 地址

ITCharge 大约 4 分钟

- 标签：字符串、回溯
- 难度：中等

题目链接

- 0093. 复原 IP 地址 - 力扣

题目大意

描述：给定一个只包含数字的字符串 `s`，用来表示一个 IP 地址

要求：返回所有由 `s` 构成的有效 IP 地址，这些地址可以通过在 `s` 中插入 `'.'` 来形成。不能重新排序或删除 `s` 中的任何数字。可以按任何顺序返回答案。

说明：

- **有效 IP 地址**：正好由四个整数（每个整数由 $0 \sim 255$ 的数构成，且不能含有前导 0），整数之间用 `.` 分割。
- $1 \leq s.length \leq 20$。
- `s` 仅由数字组成。

示例：

- 示例 1：

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

- 示例 2：

```py
输入：s = "0000"
输出：["0.0.0.0"]
```

解题思路

思路 1：回溯算法

一个有效 IP 地址由四个整数构成，中间用 3 个点隔开。现在给定的是无分隔的整数字符串，我们可以通过在整数字符串中间的不同位置插入 3 个点来生成不同的 IP 地址。这个过程可以通过回溯算法来生成。

根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择**：全排列中每个位置上的元素都可以从剩余可选元素中选出，对此画出决策树，如下图所示。
2. **明确终止条件**：
    - 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。
3. **将决策树和终止条件翻译成代码：**
    1. 定义回溯函数：
        - `backtracking(index)`：函数的传入参数是 `index`（剩余字符开始位置），全局变量是 `res`（存放所有符合条件结果的集合数组）和 `path`（存放当前符合条件的结果）。
        - `backtracking(index)`：函数代表的含义是：递归从 `index` 位置开始，从剩下字符中，选择当前子段的值。
    2. 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。
        - 从当前正在考虑的字符，到字符串结束为止，枚举出所有可作为当前子段值的字符。对于每一个子段值：
            - 约束条件：只能从 `index` 位置开始选择，并且要符合规则要求。
            - 选择元素：将其添加到当前子集数组 `path` 中。
            - 递归搜索：在选择该子段值的情况下，继续递归从剩下字符中，选择下一个子段值。
            - 撤销选择：将该子段值从当前结果数组 `path` 中移除。

```py
for i in range(index, len(s)):    # 枚举可选元素列表
    sub = s[index: i + 1]
    # 如果当前值不在 0 ~ 255 之间，直接跳过
    if int(sub) > 255:
        continue
    # 如果当前值为 0，但不是单个 0（"00..."），直接跳过
    if int(sub) == 0 and i != index:
        continue
    # 如果当前值大于 0，但是以 0 开头（"0XX..."），直接跳过
    if int(sub) > 0 and s[index] == '0':
        continue

    path.append(sub)              # 选择元素
    backtracking(i + 1)           # 递归搜索
    path.pop()                    # 撤销选择
```

3. 明确递归终止条件（给出递归终止条件，以及递归终止时的处理方法）。
   - 当遍历到决策树的叶子节点时，就终止了。也就是存放当前结果的数组 `path` 的长度等于 4，并且剩余字符开始位置为字符串结束位置（即 `len(path) == 4 and index == len(s)` ）时，递归停止。
   - 如果回溯过程中，切割次数大于 4（即 `len(path) > 4` ），递归停止，直接返回。

思路 1：代码

```py
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        res = []
        path = []
        def backtracking(index):
            # 如果切割次数大于 4，直接返回
            if len(path) > 4:
                return

            # 切割完成，将当前结果加入答案结果数组中
            if len(path) == 4 and index == len(s):
                res.append('.'.join(path))
                return

            for i in range(index, len(s)):
                sub = s[index: i + 1]
```

```
            # 如果当前值不在 0 ~ 255 之间，直接跳过
            if int(sub) > 255:
                continue
            # 如果当前值为 0，但不是单个 0（"00..."），直接跳过
            if int(sub) == 0 and i != index:
                continue
            # 如果当前值大于 0，但是以 0 开头（"0XX..."），直接跳过
            if int(sub) > 0 and s[index] == '0':
                continue

            path.append(sub)
            backtracking(i + 1)
            path.pop()


    backtracking(0)
    return res
```

思路 1：复杂度分析

- **时间复杂度**：$O(3^4 \times |s|)$，其中 $|s|$ 是字符串 `s` 的长度。由于 IP 地址的每一子段位数不会超过 3，因此在递归时，我们最多只会深入到下一层中的 3 种情况。而 IP 地址由 4 个子段构成，所以递归的最大层数为 4，则递归的时间复杂度为 $O(3^4)$。而每次将有效的 IP 地址添加到答案数组的时间复杂度为 $|s|$，所以总的时间复杂度为 $3^4 \times |s|$。
- **空间复杂度**：$O(|s|)$，只记录除了用来存储答案数组之外的空间复杂度。