刷题笔记 - python - (27-43题)

画图辅助思路：

二叉树的镜像：使用递归法。

1. **终止条件**： 当节点 `root` 为空时（即越过叶节点），则返回 `null` ；
2. **递推工作**：
    1. 初始化节点 `tmp` ，用于暂存 `root` 的左子节点；
    2. 开启递归 右子节点 `mirrorTree(root.right)` ，并将返回值作为 `root` 的 左子节点 。
    3. 开启递归 左子节点 `mirrorTree(tmp)` ，并将返回值作为 `root` 的 右子节点 。
3. **返回值**： 返回当前节点 `root` ；

这里面要注意一点：如果使用别的编程语言，没有Python的平行赋值特性。在递归右子节点 `root.left = mirrorTree(root.right)` 执行完毕后， `root.left` 的值已经发生改变，此时递归左子节点 `mirrorTree(root.left)` 则会出问题，所以要暂存 `root` 的左子节点.

```
class Solution:
    def mirrorTree(self, root: TreeNode) -> TreeNode:
        if not root: return
        root.left, root.right = self.mirrorTree(root.right), self.mirrorTree(root.left)
        return root

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-shu-de-jing-xiang-lcof/solution/mian-shi-ti-27-er-cha-shu-de-jing-xiang-di-gui-fu-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

对称的二叉树:

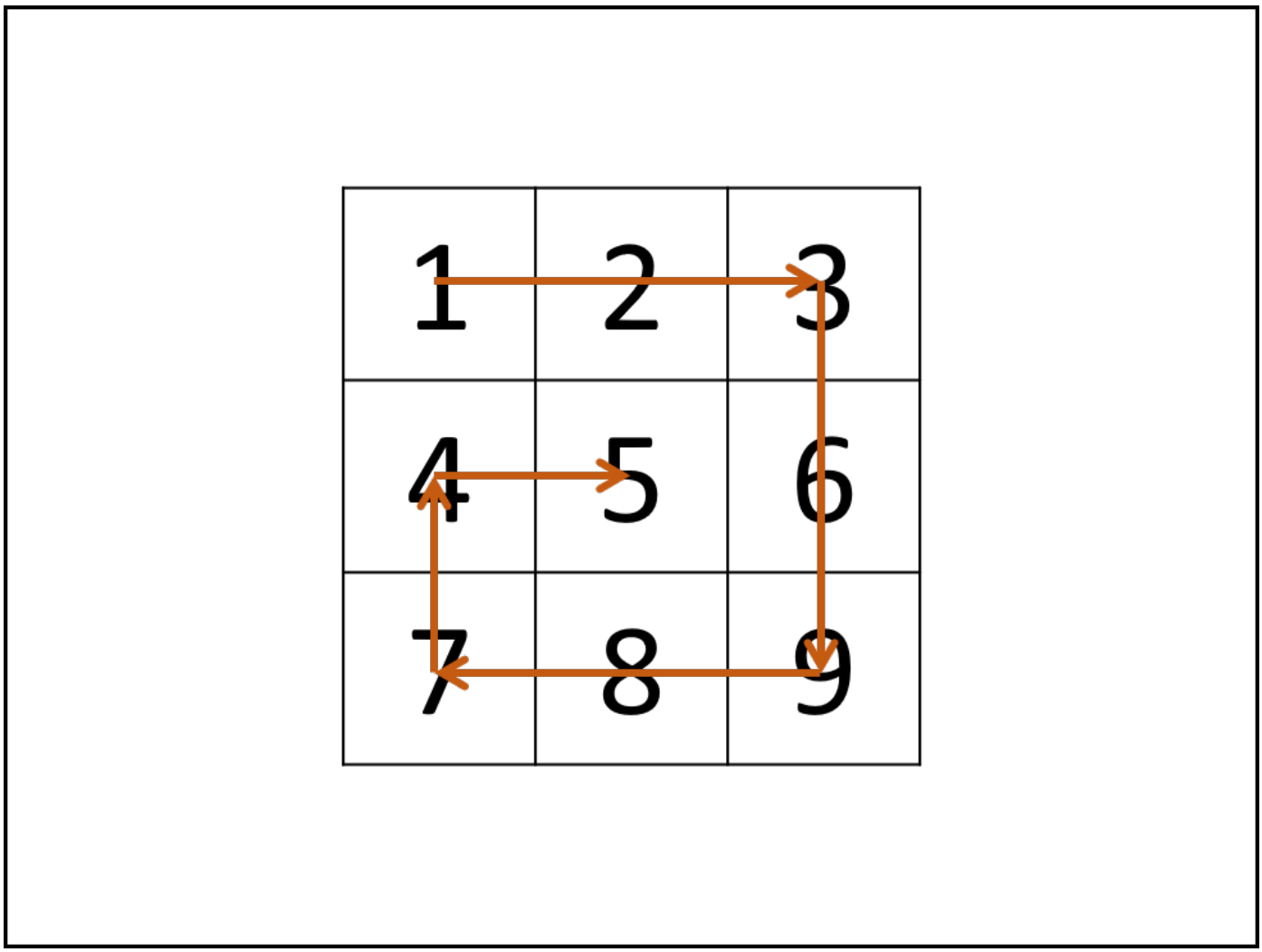
之前有提到，对于二叉树的遍历有前序遍历，中序遍历等等，我们可以定义一个新的定义方式叫做**对称前序遍历**，即**根，右，左**。如果**考虑二叉树的空节点**，则若一个树的前序遍历和另一个树的对称前序遍历相同，则它俩就是对称二叉树。

这里面要注意一下，书上定义的函数是`isSymmetric(L, R)`，但是力扣预设的函数是`isSymmetric(root)`。唯一的区别是，只要在函数里面再设置一个函数`recur(L, R)`，并用这个来递归。

```python
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        def recur(L, R):
            if not L and not R: return True
            if not L or not R or L.val != R.val: return False
            return recur(L.left, R.right) and recur(L.right, R.left)

        return recur(root.left, root.right) if root else True

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/dui-cheng-de-er-cha-shu-lcof/solution/mian-shi-ti-28-dui-cheng-de-er-cha-shu-di-gui-qing/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

顺时针打印矩阵:

考虑设定矩阵的“左、上、右、下”四个边界，模拟以上矩阵遍历顺序。



1. **空值处理**： 当 `matrix` 为空时，直接返回空列表 `[]` 即可。
2. **初始化**： 矩阵 左、右、上、下 四个边界 `l` , `r` , `t` , `b` ，用于打印的结果列表 `res` 。

3. **循环打印**： “从左向右、从上向下、从右向左、从下向上” 四个方向循环，每个方向打印中做以下三件事 （各方向的具体信息见下表）；
    1. 根据边界打印，即将元素按顺序添加至列表 `res` 尾部；
    2. 边界向内收缩 `1` （代表已被打印）；
    3. 判断是否打印完毕（边界是否相遇），若打印完毕则跳出。
4. **返回值**： 返回 `res` 即可。

打印方向	1. 根据边界打印	2. 边界向内收缩	3. 是否打印完毕
从左向右	左边界 `l` ，右边界 `r`	上边界 `t` 加 1	是否 `t > b`
从上向下	上边界 `t` ，下边界 `b`	右边界 `r` 减 1	是否 `l > r`
从右向左	右边界 `r` ，左边界 `l`	下边界 `b` 减 1	是否 `t > b`
从下向上	下边界 `b` ，上边界 `t`	左边界 `l` 加 1	是否 `l > r`

```python
class Solution:
    def spiralOrder(self, matrix:[[int]]) -> [int]:
        if not matrix: return []
        l, r, t, b, res = 0, len(matrix[0]) - 1, 0, len(matrix) - 1, []
        while True:
            for i in range(l, r + 1): res.append(matrix[t][i]) # left to right
            t += 1
            if t > b: break
            for i in range(t, b + 1): res.append(matrix[i][r]) # top to bottom
            r -= 1
            if l > r: break
            for i in range(r, l - 1, -1): res.append(matrix[b][i]) # right to left
            b -= 1
            if t > b: break
            for i in range(b, t - 1, -1): res.append(matrix[i][l]) # bottom to top
            l += 1
            if l > r: break
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shun-shi-zhen-da-yin-ju-zhen-lcof/solution/mian-shi-ti-29-shun-shi-zhen-da-yin-ju-zhen-she-di/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

举例让抽象问题具体化

包含min函数的栈：需要复杂度为O(1)

一般来说，获得栈的最小值，需要遍历整个栈，复杂度为O(N)，如果要降为O(1)，需要建立一个辅助栈。

- 数据栈 A： 栈 A 用于存储所有元素，保证入栈 `push()` 函数、出栈 `pop()` 函数、获取栈顶 `top()` 函数的正常逻辑。
- 辅助栈 B： 栈 B 中存储栈 A 中所有 **非严格降序** 的元素，则栈 A 中的最小元素始终对应栈 B 的栈顶元素，即 `min()` 函数只需返回栈 B 的栈顶元素即可。

因此，只需设法维护好 栈 B 的元素，使其保持非严格降序。

push(x) 函数： 重点为保持栈 B 的元素是 **非严格降序** 的。

1. 将 x 压入栈 A （即 `A.add(x)` ）；
2. 若 ① 栈 B 为空 或 ② x **小于等于** 栈 B 的栈顶元素，则将 x 压入栈 B （即 `B.add(x)` ）。

pop() 函数： 重点为保持栈 A,B 的 **元素一致性** 。

1. 执行栈 A 出栈（即 `A.pop()` ），将出栈元素记为 y；
2. 若 y 等于栈 B 的栈顶元素，则执行栈 B 出栈（即 `B.pop()` ）。

top() 函数： 直接返回栈 A 的栈顶元素即可，即返回 `A.peek()` 。

min() 函数： 直接返回栈 B 的栈顶元素即可，即返回 `B.peek()` 。

```
class MinStack:
    def __init__(self):
        self.A, self.B = [], []

    def push(self, x: int) -> None:
        self.A.append(x)
        if not self.B or self.B[-1] >= x:
            self.B.append(x)

    def pop(self) -> None:
        if self.A.pop() == self.B[-1]:
            self.B.pop()

    def top(self) -> int:
        return self.A[-1]

    def min(self) -> int:
        return self.B[-1]

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/bao-han-minhan-shu-de-zhan-lcof/solution/mian-shi-ti-30-bao-han-minhan-shu-de-zhan-fu-zhu-z/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

从上到下打印二叉树:

我们发现这是使用**BFS**和**队列**可以解决的按层遍历算法。每次打印一个节点，就将它的子节点放到队列里，然后按顺序打印即可。

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        if not root: return []
        res, queue = [], collections.deque()
        queue.append(root)
        while queue:
            node = queue.popleft()
            res.append(node.val)
            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/solution/mian-shi-ti-32-i-cong-shang-dao-xia-da-yin-er-ch-4/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

从上到下打印二叉树- II：区别在于，这返回一个二维列表，每一层一个子列表。

1. **特例处理**： 当根节点为空，则返回空列表`[]`；
2. **初始化**： 打印结果列表`res = []`，包含根节点的队列`queue = [root]`；
3. **BFS循环**： 当队列`queue`为空时跳出；
    1. 新建一个临时列表`tmp`，用于存储当前层打印结果；
    2. **当前层打印循环**： 循环次数为当前层节点数（即队列`queue`长度）；
        1. **出队**： 队首元素出队，记为 node；
        2. **打印**： 将`node.val`添加至`tm`尾部；
        3. **添加子节点**： 若`node`的左（右）子节点不为空，则将左（右）子节点加入队列`queue`；
    3. 将当前层结果`tmp`添加入`res`。
4. **返回值**： 返回打印结果列表`res`即可。

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        res, queue = [], collections.deque()
        queue.append(root)
        while queue:
            tmp = []
            for _ in range(len(queue)):
                node = queue.popleft()
                tmp.append(node.val)
```

```
                if node.left: queue.append(node.left)
                if node.right: queue.append(node.right)
            res.append(tmp)
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/solution/mian-shi-ti-32-ii-cong-shang-dao-xia-da-yin-er-c-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

从上到下打印二叉树- III：区别在于：第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        res, deque = [], collections.deque([root])
        while deque:
            tmp = collections.deque()
            for _ in range(len(deque)):
                node = deque.popleft()
                if len(res) % 2: tmp.appendleft(node.val) # 偶数层 -> 队列头部
                else: tmp.append(node.val) # 奇数层 -> 队列尾部
                if node.left: deque.append(node.left)
                if node.right: deque.append(node.right)
            res.append(list(tmp))
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/solution/mian-shi-ti-32-iii-cong-shang-dao-xia-da-yin-er--3/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

其中，只是在内层循环里加了一个判断语句，队列添加元素的时候换用`appendleft`即可。

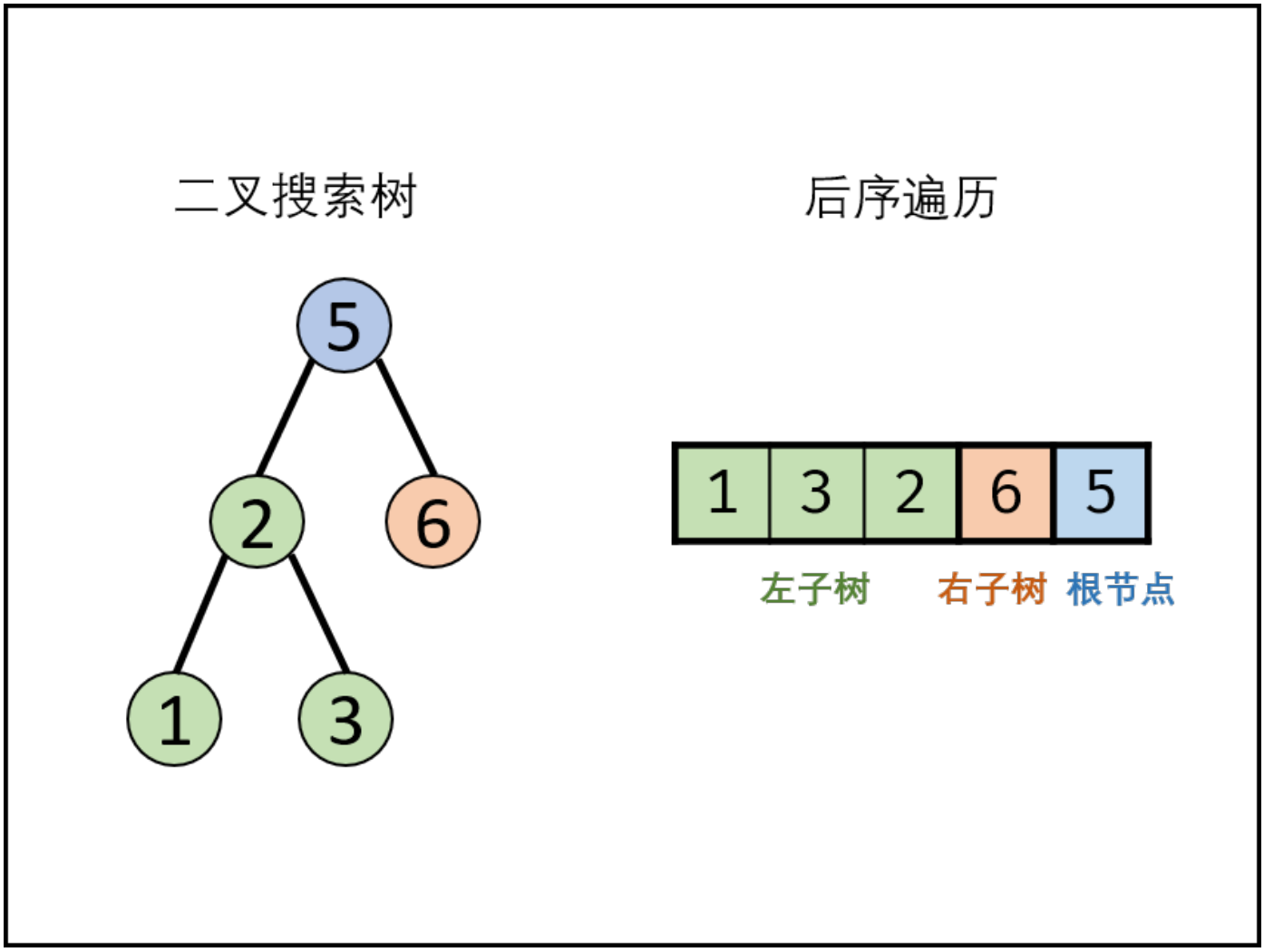
二叉搜索树的后序遍历序列:

这里有两个概念：**二叉搜索树**和**后续遍历**

后序遍历定义：遍历顺序为**左，右，根**。

二叉搜索树定义： 左子树中所有节点的值 << 根节点的值；右子树中所有节点的值 >> 根节点的值；其左、右子树也分别为二叉搜索树。

可以通过递归，判断所有子树的**正确性** （即其后序遍历是否满足二叉搜索树的定义） ，若所有子树都正确，则此序列为二叉搜索树的后序遍历。



终止条件： 当 $i \geq j$ ，说明此子树节点数量 ≤ 1 ，无需判别正确性，因此直接返回 `True` ；

递推工作：

1. **划分左右子树**： 遍历后序遍历的 `[i, j]` 区间元素，寻找**第一个大于根节点**的节点，索引记为 `m` 。此时，可划分出左子树区间 `[i, m - 1]` 、右子树区间 `[m, j - 1]` 、根节点索引 `j` 。
2. **判断是否为二叉搜索树**：
   - **左子树区间** `[i, m - 1]` 内的所有节点都应 $< postorder[j]$ 。而第 `1.划分左右子树` 步骤已经保证左子树区间的正确性，因此只需要判断右子树区间即可。
   - **右子树区间** `[m, j-1]` 内的所有节点都应 $> postorder[j]$ 。实现方式为遍历，当遇到 $\leq postorder[j]$ 的节点则跳出；则可通过 $p = j$ 判断是否为二叉搜索树。

返回值： 所有子树都需正确才可判定正确，因此使用 与**逻辑符 &&** 连接。

1. $p = j$： 判断**此树**是否正确。
2. $recur(i, m - 1)$： 判断**此树的左子树**是否正确。
3. $recur(m, j - 1)$： 判断**此树的右子树**是否正确。

```
class Solution:
    def verifyPostorder(self, postorder: [int]) -> bool:
        def recur(i, j):
            if i >= j: return True
            p = i
```

```
            while postorder[p] < postorder[j]: p += 1
            m = p
            while postorder[p] > postorder[j]: p += 1
            return p == j and recur(i, m - 1) and recur(m, j - 1)

        return recur(0, len(postorder) - 1)

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/solution/mian-shi-ti-33-er-cha-sou-suo-shu-de-hou-xu-bian-6/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

二叉树中和为某一值的路径:

使用回溯法解决问题，包含**先序遍历 + 路径记录**两部分。

- **先序遍历**： 按照 **“根、左、右”** 的顺序，遍历树的所有节点。
- **路径记录**： 在先序遍历中，记录从根节点到当前节点的路径。当路径为 ① 根节点到叶节点形成的路径 且 ② 各节点值的和等于目标值`sum`时，将此路径加入结果列表。

`pathSum(root, sum)`函数：

- **初始化**： 结果列表`res`，路径列表`path`。
- **返回值**： 返回`res`即可。

`recur(root, tar)`函数：

- **递推参数**： 当前节点`root`，当前目标值`tar`。
- **终止条件**： 若节点`root`为空，则直接返回。
- 递推工作：
    1. **路径更新**： 将当前节点值`root.val`加入路径`path`；
    2. **目标值更新**： `tar = tar - root.val`（即目标值`tar`从`sum`减至0）；
    3. **路径记录**： 当 ① `root` 为叶节点 且 ② 路径和等于目标值 ，则将此路径 `path` 加入 `res` 。
    4. **先序遍历**： 递归左 / 右子节点。
    5. **路径恢复**： 向上回溯前，需要将当前节点从路径 `path` 中删除，即执行 `path.pop()` 。

值得注意的是，记录路径时若直接执行 `res.append(path)` ，则是将 `path` 对象加入了 `res` ；后续 `path` 改变时， `res` 中的 `path` 对象也会随之改变。

正确做法： `res.append(list(path))` ，相当于复制了一个 `path` 并加入到 `res` 。

```
class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> List[List[int]]:
        res, path = [], []
        def recur(root, tar):
            if not root: return
            path.append(root.val)
            tar -= root.val
            if tar == 0 and not root.left and not root.right:
```

```
                res.append(list(path))
            recur(root.left, tar)
            recur(root.right, tar)
            path.pop()

        recur(root, sum)
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-shu-zhong-he-wei-mou-yi-zhi-de-lu-jing-lcof/solution/mian-shi-ti-34-er-cha-shu-zhong-he-wei-mou-yi-zh-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

分解让复杂的问题简单化

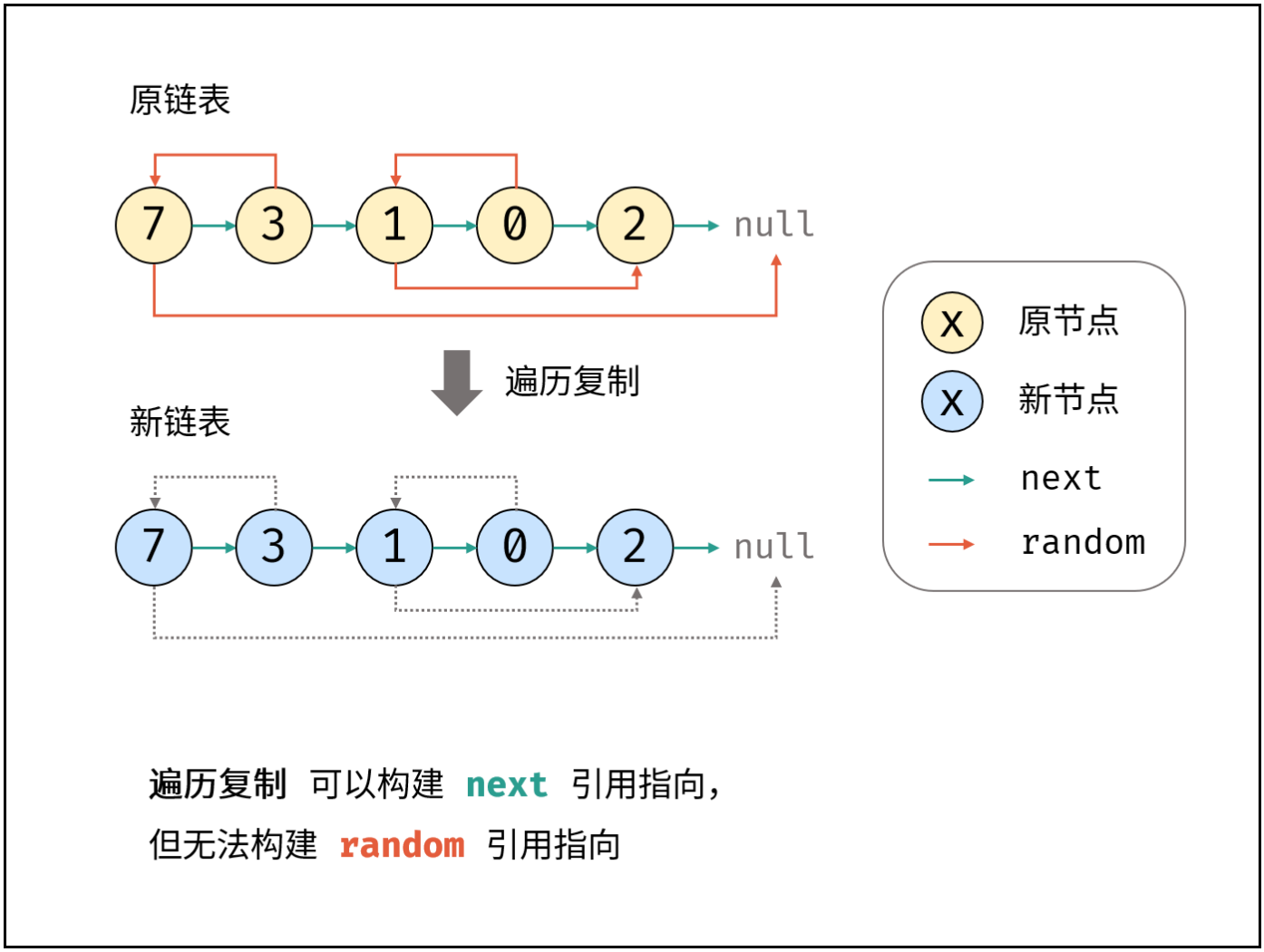
复杂链表的复制:

在这里，复杂链表的定义如下：

```
# Definition for a Node.
class Node:
    def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
        self.val = int(x)
        self.next = next
        self.random = random
```

给定链表的头节点 `head` ，复制普通链表很简单，只需遍历链表，每轮建立新节点 + 构建前驱节点 `pre` 和当前节点 `node` 的引用指向即可。

本题链表的节点新增了 `random` 指针，指向链表中的 **任意节点** 或者 `null` 。这个 `random` 指针意味着在复制过程中，除了构建前驱节点和当前节点的引用指向 `pre.next` ，还要构建前驱节点和其随机节点的引用指向 `pre.random` 。



方法一：哈希表

利用哈希表的查询特点，考虑构建 **原链表节点** 和 **新链表对应节点** 的键值对映射关系，再遍历构建新链表各节点的 `next` 和 `random` 引用指向即可。

算法流程：

1. 若头节点 `head` 为空节点，直接返回 `null` ；
2. **初始化**： 哈希表 `dic` ， 节点 `cur` 指向头节点；
3. **复制链表**：
    1. 建立新节点，并向 `dic` 添加键值对 (原 `cur` 节点, 新 `cur` 节点) ；
    2. `cur` 遍历至原链表下一节点；
4. **构建新链表的引用指向**：
    1. 构建新节点的 `next` 和 `random` 引用指向；
    2. `cur` 遍历至原链表下一节点；
5. **返回值**： 新链表的头节点 `dic[cur]` ；

```python
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head: return
        dic = {}
        # 3. 复制各节点，并建立 “原节点 -> 新节点” 的 Map 映射
```

```
        cur = head
        while cur:
            dic[cur] = Node(cur.val)
            cur = cur.next
        cur = head
        # 4. 构建新节点的 next 和 random 指向
        while cur:
            dic[cur].next = dic.get(cur.next)
            dic[cur].random = dic.get(cur.random)
            cur = cur.next
        # 5. 返回新链表的头节点
        return dic[head]

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/solution/jian-zhi-offer-35-fu-za-lian-biao-de-fu-zhi-ha-xi-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

方法二：拼接 + 拆分

看了上种方法再直接看这个代码，应该很好理解：

```
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head: return
        cur = head
        # 1. 复制各节点，并构建拼接链表
        while cur:
            tmp = Node(cur.val)
            tmp.next = cur.next
            cur.next = tmp
            cur = tmp.next
        # 2. 构建各新节点的 random 指向
        cur = head
        while cur:
            if cur.random:
                cur.next.random = cur.random.next
            cur = cur.next.next
        # 3. 拆分两链表
        cur = res = head.next
        pre = head
        while cur.next:
            pre.next = pre.next.next
            cur.next = cur.next.next
            pre = pre.next
            cur = cur.next
        pre.next = None # 单独处理原链表尾节点
        return res      # 返回新链表头节点
```

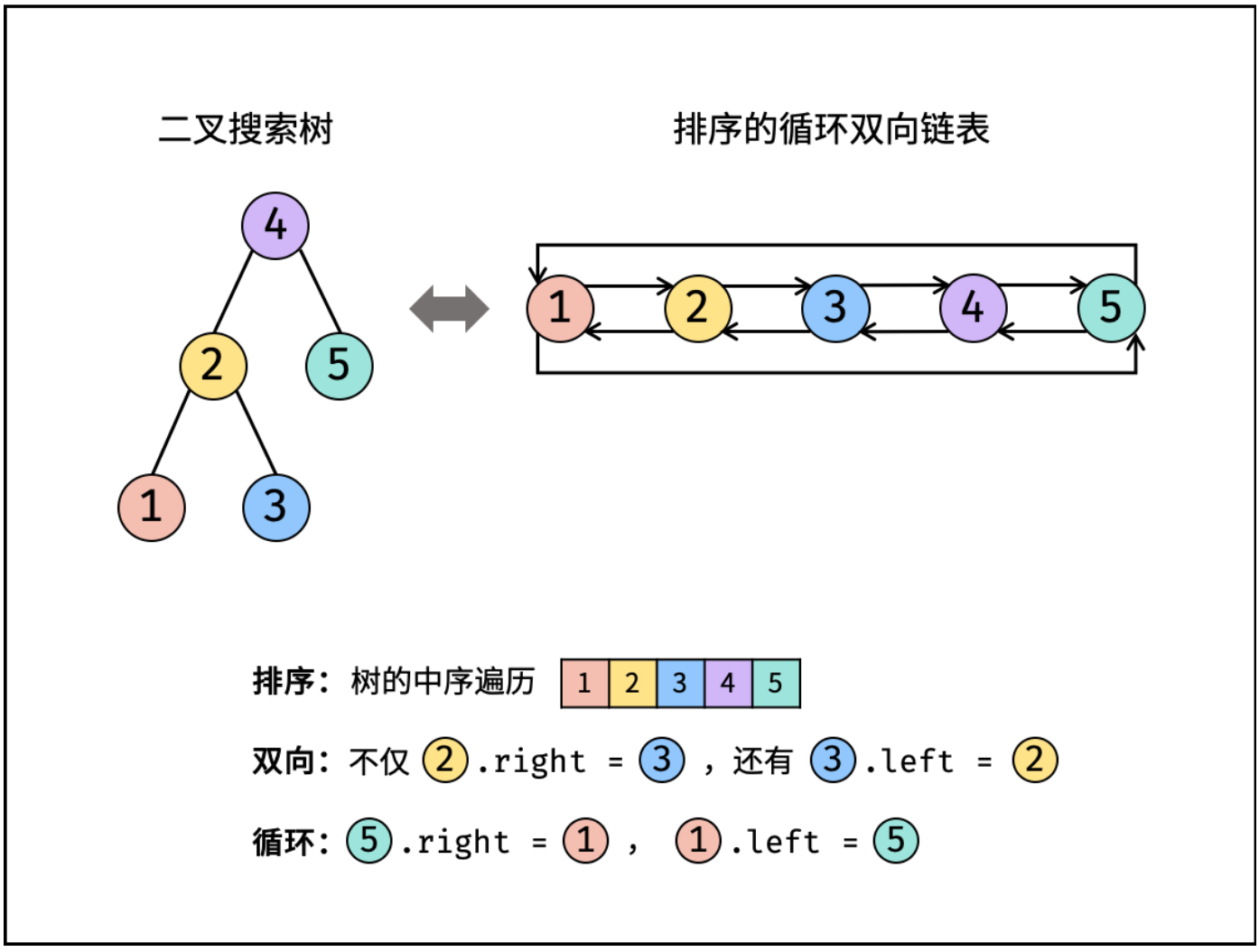
```
# 作者：jyd
# 链接：https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/solution/jian-zhi-offer-35-fu-za-lian-biao-de-fu-zhi-ha-xi-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

二叉搜索树与双向链表:中序遍历

解题思路:
本文解法基于性质：二叉搜索树的中序遍历为 **递增序列** 。

将 二叉搜索树 转换成一个 “排序的循环双向链表” ，其中包含三个要素：

1. **排序链表**： 节点应从小到大排序，因此应使用 **中序遍历** “从小到大”访问树的节点。
2. **双向链表**： 在构建相邻节点的引用关系时，设前驱节点 `pre` 和当前节点 `cur` ，不仅应构建 `pre.right = cur` ，也应构建 `cur.left = pre` 。
3. **循环链表**： 设链表头节点 `head` 和尾节点 `tail` ，则应构建 `head.left = tail` 和 `tail.right = head` 。



中序遍历 为对二叉树作 “左、根、右” 顺序遍历，递归实现如下：

```
# 打印中序遍历
def dfs(root):
    if not root: return
    dfs(root.left)  # 左
    print(root.val) # 根
    dfs(root.right) # 右

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof/solution/mian-shi-ti-36-er-cha-sou-suo-shu-yu-shuang-xian-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

算法流程：

`dfs(cur)`：递归法中序遍历；

1. **终止条件**： 当节点 `cur` 为空，代表越过叶节点，直接返回；
2. 递归左子树，即 `dfs(cur.left)` ；
3. **构建链表**：
    1. **当 `pre` 为空时**： 代表正在访问链表头节点，记为 `head` ；
    2. **当 `pre` 不为空时**： 修改双向节点引用，即 `pre.right = cur` ， `cur.left = pre` ；
    3. **保存 `cur`** ： 更新 `pre = cur` ，即节点 `cur` 是后继节点的 `pre` ；
4. 递归右子树，即 `dfs(cur.right)` ；

`treeToDoublyList(root)`：

1. **特例处理**： 若节点 `root` 为空，则直接返回；
2. **初始化**： 空节点 `pre` ；
3. **转化为双向链表**： 调用 `dfs(root)` ；
4. **构建循环链表**： 中序遍历完成后， `head` 指向头节点， `pre` 指向尾节点，因此修改 `head` 和 `pre` 的双向节点引用即可；
5. **返回值**： 返回链表的头节点 `head` 即可；

```
class Solution:
    def treeToDoublyList(self, root: 'Node') -> 'Node':
        def dfs(cur):
            if not cur: return
            dfs(cur.left) # 递归左子树
            if self.pre: # 修改节点引用
                self.pre.right, cur.left = cur, self.pre
            else: # 记录头节点
                self.head = cur
            self.pre = cur # 保存 cur
            dfs(cur.right) # 递归右子树

        if not root: return
        self.pre = None
```

```
        dfs(root)
        self.head.left, self.pre.right = self.pre, self.head
        return self.head

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof/solution/mian-shi-ti-36-er-cha-sou-suo-shu-yu-shuang-xian-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

序列化二叉树:

题目要求的 序列化 和 反序列化 是 **可逆操作** 。因此，序列化的字符串应携带 **完整的二叉树信息** 。序列化的字符串实际上是二叉树的 “层序遍历”（BFS）结果，本文也采用层序遍历。

为完整表示二叉树，考虑将叶节点下的 `null` 也记录。在此基础上，对于列表中任意某节点 `node` ，其左子节点 `node.left` 和右子节点 `node.right` 在序列中的位置都是 **唯一确定** 的。

设 m 为列表区间 $[0, n]$中的 `null` 节点个数，则可总结出根节点、左子节点、右子节点的列表索引的递推公式：

node.val	node 的列表索引	node.left 的列表索引	node.right 的列表索引
$\neq$ `null`	n	$2(n-m)+1$	$2(n-m)+2$
$=$ `null`	n	无	无

反序列化 通过以上递推公式反推各节点在序列中的索引，进而实现。

序列化 Serialize：

借助队列，对二叉树做层序遍历，并将越过叶节点的 `null` 也打印出来。

1. **特例处理**： 若 `root` 为空，则直接返回空列表 `"[]"` ；
2. **初始化**： 队列 `queue` （包含根节点 `root` ）；序列化列表 `res` ；
3. **层序遍历**： 当 `queue` 为空时跳出；
    1. **节点出队**，记为 `node` ；
    2. 若 `node` 不为空：① 打印字符串 `node.val` ，② 将左、右子节点加入 `queue` ；
    3. 否则（若 `node` 为空）：打印字符串 `"null"` ；
4. **返回值**： 拼接列表，用 `','` 隔开，首尾添加中括号；

反序列化 Deserialize：

基于本文开始推出的 `node` , `node.left` , `node.right` 在序列化列表中的位置关系，可实现反序列化。

利用队列按层构建二叉树，借助一个指针 `i` 指向节点 `node` 的左、右子节点，每构建一个 `node` 的左、右子节点，指针 `i` 就向右移动 1 位。

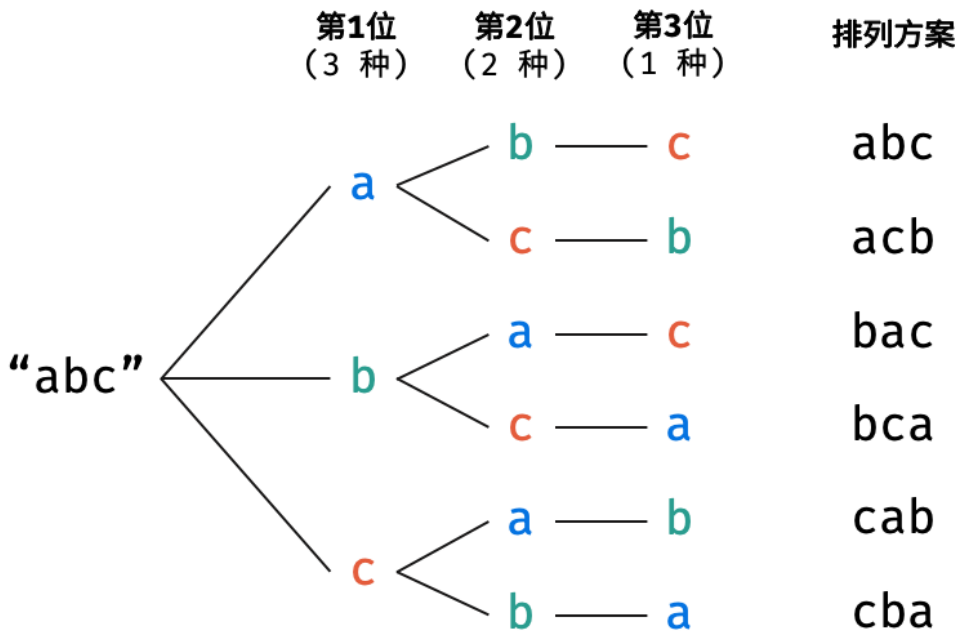
1. **特例处理**： 若 `data` 为空，直接返回 `null` ；
2. **初始化**： 序列化列表 `vals` （先去掉首尾中括号，再用逗号隔开），指针 `i = 1` ，根节点 `root` （值为 `vals[0]` ），队列 `queue` （包含 `root` ）；

3. **按层构建**： 当 `queue` 为空时跳出；
    1. 节点出队，记为 `node` ；
    2. 构建 `node` 的左子节点： `node.left` 的值为 `vals[i]` ，并将 `node.left` 入队；
    3. 执行 `i += 1` ；
    4. 构建 `node` 的右子节点： `node.left` 的值为 `vals[i]` ，并将 `node.left` 入队；
    5. 执行 `i += 1` ；
4. **返回值**： 返回根节点 `root` 即可；

```
class Codec:
    def serialize(self, root):
        if not root: return "[]"
        queue = collections.deque()
        queue.append(root)
        res = []
        while queue:
            node = queue.popleft()
            if node:
                res.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else: res.append("null")
        return '[' + ','.join(res) + ']'

    def deserialize(self, data):
        if data == "[]": return
        vals, i = data[1:-1].split(','), 1
        root = TreeNode(int(vals[0]))
        queue = collections.deque()
        queue.append(root)
        while queue:
            node = queue.popleft()
            if vals[i] != "null":
                node.left = TreeNode(int(vals[i]))
                queue.append(node.left)
            i += 1
            if vals[i] != "null":
                node.right = TreeNode(int(vals[i]))
                queue.append(node.right)
            i += 1
        return root

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/xu-lie-hua-er-cha-shu-lcof/solution/mian-shi-ti-37-xu-lie-hua-er-cha-shu-ceng-xu-bian-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

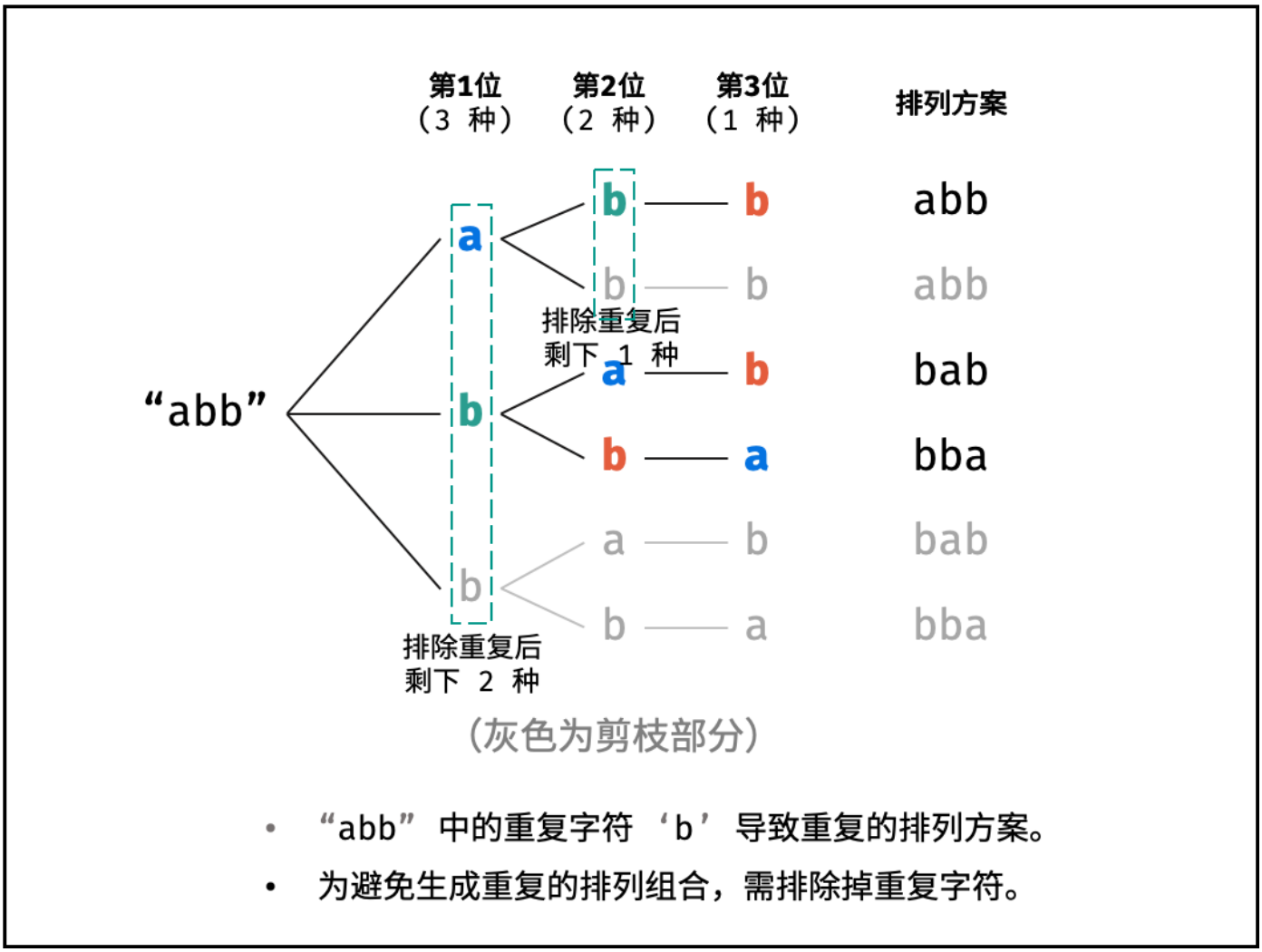
字符串的排列：生成排列方案，剪枝重复方案

根据字符串排列的特点，考虑深度优先搜索所有排列方案。即通过字符交换，先固定第 1 位字符（ n 种情况）、再固定第22 位字符（ n−1 种情况）、… 、最后固定第 n 位字符（ 1 种情况）。



- 对于长度为 n 的字符串（字符互不重复），其排列方案共有 $n\times(n-1)\times(n-2)\times\ldots\times2\times1$ 种。
- 对于以上图例，共有 $3\times2\times1=6$ 种排列方案。

当字符串存在重复字符时，排列方案中也存在重复的排列方案。为排除重复方案，需在固定某位字符时，保证 “每种字符只在此位固定一次” ，即遇到重复字符时不交换，直接跳过。从 DFS 角度看，此操作称为 “剪枝” 。



1. **终止条件**： 当 `x = len(c) - 1` 时，代表所有位已固定（最后一位只有 1 种情况），则将当前组合 `c` 转化为字符串并加入 `res` ，并返回；
2. **递推参数**： 当前固定位 `x` ；
3. **递推工作**： 初始化一个 Set ，用于排除重复的字符；将第 `x` 位字符与 i ∈ `[x, len(c)]` 字符分别交换，并进入下层递归；
    1. **剪枝**： 若 `c[i]` 在 Set 中，代表其是重复字符，因此“剪枝”；
    2. 将 `c[i]` 加入 Set ，以便之后遇到重复字符时剪枝；
    3. **固定字符**： 将字符 `c[i]` 和 `c[x]` 交换，即固定 `c[i]` 为当前位字符；
    4. **开启下层递归**： 调用 `dfs(x + 1)` ，即开始固定第 `x + 1` 个字符；
    5. **还原交换**： 将字符 `c[i]` 和 `c[x]` 交换（还原之前的交换）；

```
class Solution:
    def permutation(self, s: str) -> List[str]:
        c, res = list(s), []
        def dfs(x):
            if x == len(c) - 1:
                res.append(''.join(c))    # 添加排列方案
                return
            dic = set()
            for i in range(x, len(c)):
                if c[i] in dic: continue # 重复，因此剪枝
```

```
                dic.add(c[i])
                c[i], c[x] = c[x], c[i]  # 交换，将 c[i] 固定在第 x 位
                dfs(x + 1)               # 开启固定第 x + 1 位字符
                c[i], c[x] = c[x], c[i]  # 恢复交换
        dfs(0)
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/zi-fu-chuan-de-pai-lie-lcof/solution/mian-shi-ti-38-zi-fu-chuan-de-pai-lie-hui-su-fa-by/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

优化时间和空间效率 - 时间效率

数组中出现次数超过一半的数字:

本题常见的三种解法：

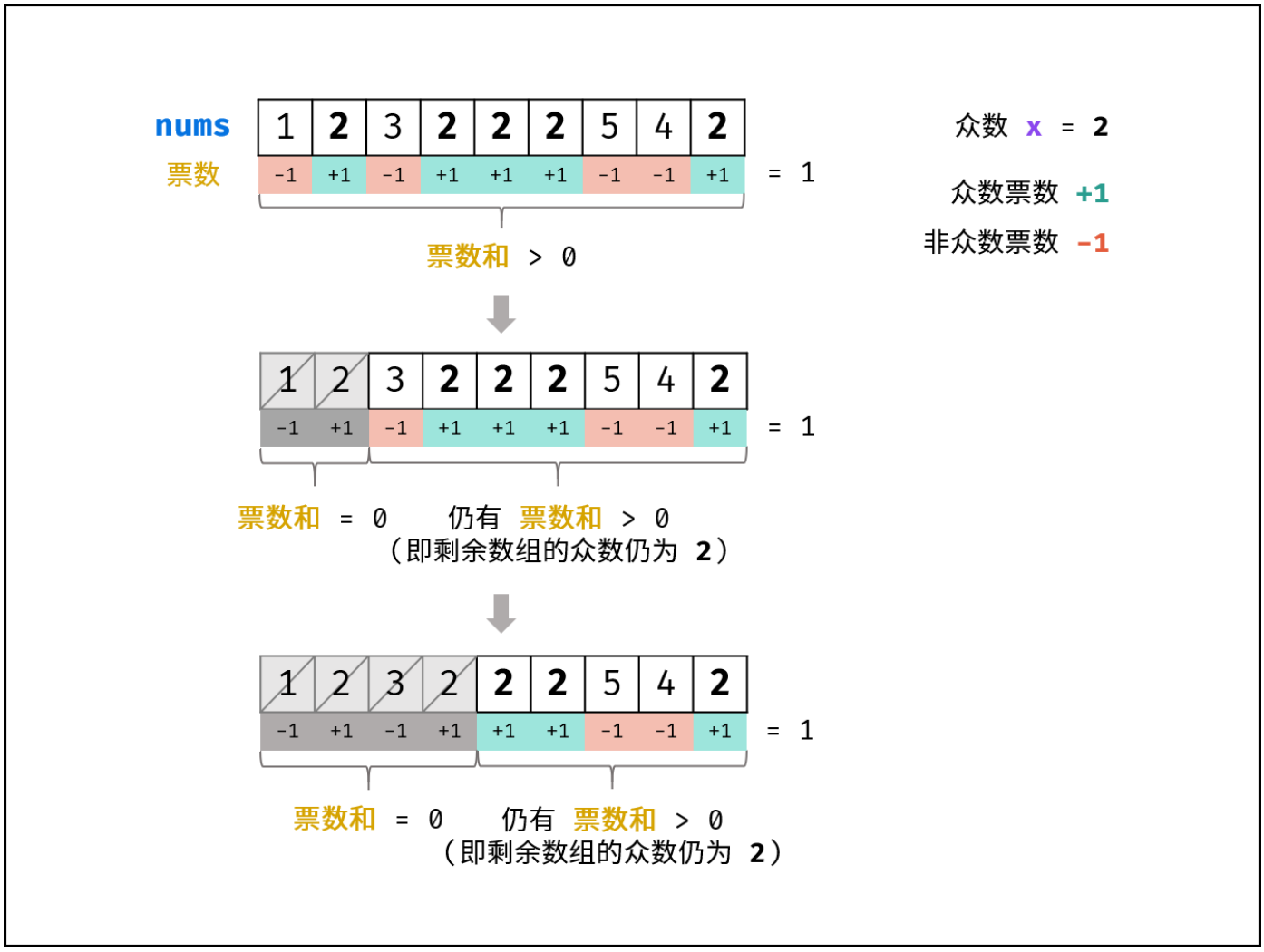
1. **哈希表统计法**： 遍历数组 `nums` ，用 HashMap 统计各数字的数量，即可找出 “众数” 。此方法时间和空间复杂度均为 $O(N)$ 。
2. **数组排序法**： 将数组 `nums` 排序，**数组中点的元素** 一定为“众数”。
3. **摩尔投票法**： 核心理念为 **票数正负抵消** 。此方法时间和空间复杂度分别为 $O(N)$ 和 $O(1)$ ，为本题的最佳解法。

数组排序法：使用快速排序法，然后找中位数，就是“众数”。快速排序的**分治**代码在之前就写过，用这个方法，这道题是*第k小的数*的特殊情况。

摩尔投票法：设输入数组 `nums` 的众数为 x ，数组长度为 n 。

推论一： 若记 **“众数”** 的票数为 +1 ，**非“众数”** 的票数为 −1 ，则一定有所有数字的 **票数和** >0 。

推论二： 若数组的前 a 个数字的 **票数和** =0 ，则 数组剩余 $(n-a)$ 个数字的 **票数和一定仍** >0 ，即后 $(n-a)$ 个数字的 **“众数”仍为** x 。



根据以上推论，记数组首个元素为 n_1，“众数”为 x，遍历并统计票数。当发生 **票数和** =0 时，**剩余数组的“众数”一定不变**，这是由于：

- 当 $n_1 = x$： 抵消的所有数字，有一半是“众数” x 。
- 当 $n_1 \neq x$： 抵消的所有数字，“众数” x 的数量为一半或 0 个。

利用此特性，每轮假设发生 **票数和** =0 都可以 **缩小剩余数组区间** 。当遍历完成时，最后一轮假设的数字即为“众数”。

算法流程:

1. **初始化**： 票数统计 `votes = 0` ， 众数 `x`；
2. **循环**： 遍历数组 `nums` 中的每个数字 `num` ；
    1. 当 票数 `votes` 等于 0 ，则假设当前数字 `num` 是“众数”；
    2. 当 `num = x` 时，票数 `votes` 自增 1 ；当 `num != x` 时，票数 `votes` 自减 1 ；
3. **返回值**： 返回 `x` 即可；

如果考虑**不存在“众数”**的情况，需要加一个“验证环节”：

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        votes, count = 0, 0
        for num in nums:
```

```
            if votes == 0: x = num
            votes += 1 if num == x else -1
        # 验证 x 是否为众数
        for num in nums:
            if num == x: count += 1
        return x if count > len(nums) // 2 else 0 # 当无众数时返回 0

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-zu-zhong-chu-xian-ci-shu-chao-guo-yi-ban-de-shu-zi-lcof/solution/mian-shi-ti-39-shu-zu-zhong-chu-xian-ci-shu-chao-3/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

最小的k个数:

我们可以借鉴**快速排序**的思想。我们知道快排的划分函数每次执行完后都能将数组分成两个部分，小于等于分界值 `pivot` 的元素的都会被放到数组的左边，大于的都会被放到数组的右边，然后返回分界值的下标。与快速排序不同的是，快速排序会根据分界值的下标递归处理划分的两侧，而这里我们只处理划分的一边。

我们定义函数 `randomized_selected(arr, l, r, k)` 表示划分数组 `arr` 的 `[l,r]` 部分，使前 `k` 小的数在数组的左侧，在函数里我们调用快排的划分函数，假设划分函数返回的下标是 `pos`（表示分界值 `pivot` 最终在数组中的位置），即 `pivot` 是数组中第 `pos - l + 1` 小的数，那么一共会有三种情况：

- 如果 `pos - l + 1 == k`，表示 `pivot` 就是第 k 小的数，直接返回即可；
- 如果 `pos - l + 1 < k`，表示第 k 小的数在 `pivot` 的右侧，因此递归调用 `randomized_selected(arr, pos + 1, r, k - (pos - l + 1))`；
- 如果 `pos - l + 1 > k`，表示第 k 小的数在 `pivot` 的左侧，递归调用 `randomized_selected(arr, l, pos - 1, k)`。

函数递归入口为 `randomized_selected(arr, 0, arr.length - 1, k)`。在函数返回后，将前 `k` 个数放入答案数组返回即可。

```
class Solution:
    def partition(self, nums, l, r):
        pivot = nums[r]
        i = l - 1
        for j in range(l, r):
            if nums[j] <= pivot:
                i += 1
                nums[i], nums[j] = nums[j], nums[i]
        nums[i + 1], nums[r] = nums[r], nums[i + 1]
        return i + 1

    def randomized_partition(self, nums, l, r):
        i = random.randint(l, r)
        nums[r], nums[i] = nums[i], nums[r]
        return self.partition(nums, l, r)

    def randomized_selected(self, arr, l, r, k):
```

```
        pos = self.randomized_partition(arr, l, r)
        num = pos - l + 1
        if k < num:
            self.randomized_selected(arr, l, pos - 1, k)
        elif k > num:
            self.randomized_selected(arr, pos + 1, r, k - num)

    def getLeastNumbers(self, arr: List[int], k: int) -> List[int]:
        if k == 0:
            return list()
        self.randomized_selected(arr, 0, len(arr) - 1, k)
        return arr[:k]

# 作者：LeetCode-Solution
# 链接：https://leetcode-cn.com/problems/zui-xiao-de-kge-shu-lcof/solution/zui-xiao-de-kge-shu-by-leetcode-solution/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

数据流中的中位数：

给定一长度为 N 的无序数组，其中位数的计算方法：首先对数组执行排序（使用 $O(N \log N)$ 时间），然后返回中间元素即可（使用 $O(1)$ 时间）。

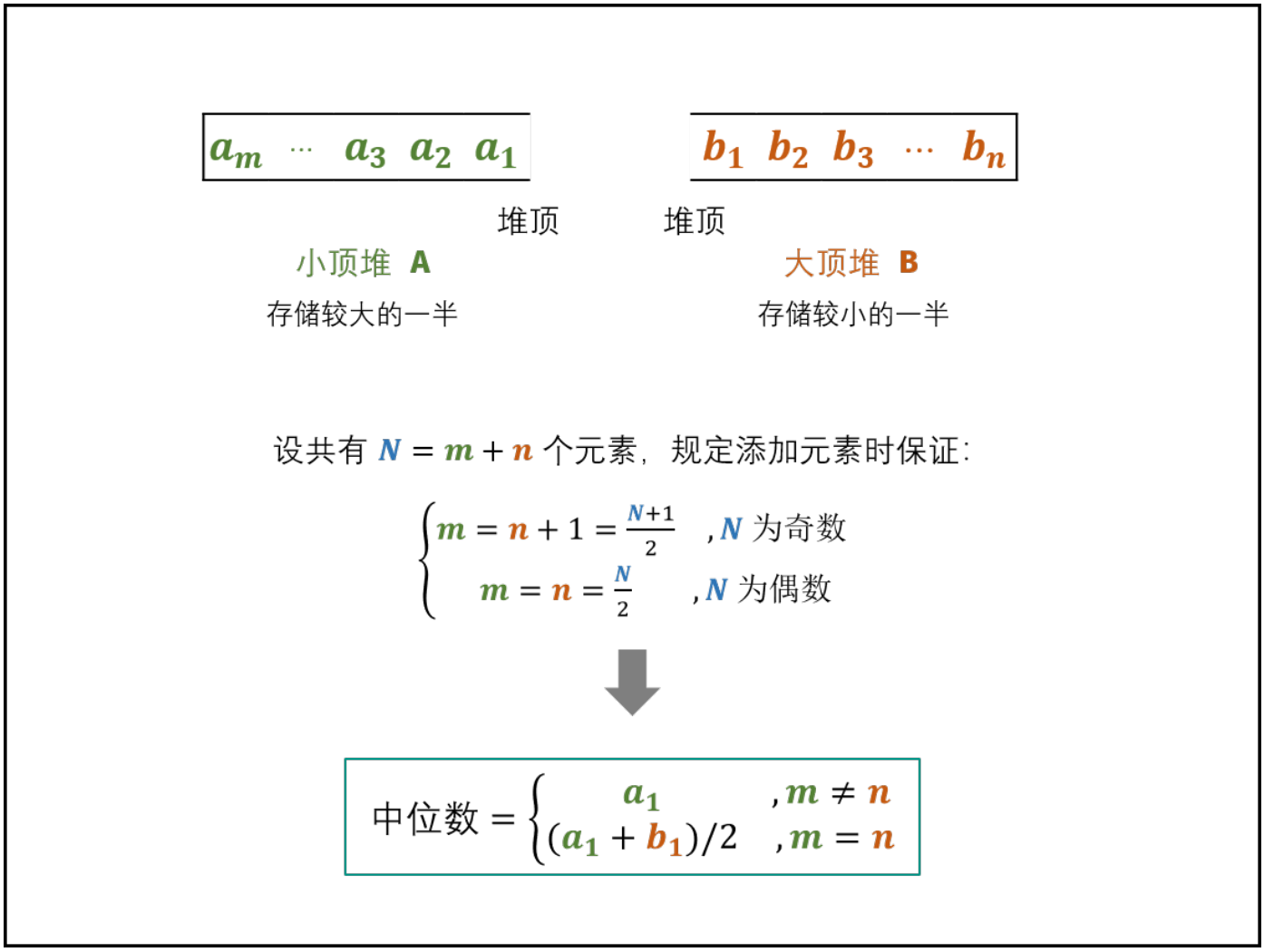
针对本题，根据以上思路，可以将数据流保存在一个列表中，并在添加元素时 **保持数组有序** 。此方法的时间复杂度为 $O(N)$，其中包括： 查找元素插入位置 $O(\log N)$（二分查找）、向数组某位置插入元素 $O(N)$（插入位置之后的元素都需要向后移动一位）。

借助 **堆** 可进一步优化时间复杂度。

建立一个 **小顶堆** A 和 **大顶堆** B，各保存列表的一半元素，且规定：

- A 保存 **较大** 的一半，长度为 $\frac{N}{2}$（N 为偶数）或 $\frac{N+1}{2}$（N 为奇数）；
- B 保存 **较小** 的一半，长度为 $\frac{N}{2}$（N 为偶数）或 $\frac{N-1}{2}$（N 为奇数）；

随后，中位数可仅根据 A, B 的堆顶元素计算得到。



算法流程：

设元素总数为 $N = m + n$，其中 m 和 n 分别为 A 和 B 中的元素个数。

`addNum(num)` 函数：

1. 当 $m = n$（即 N 为 **偶数**）：需向 A 添加一个元素。实现方法：将新元素 num 插入至 B，再将 B 堆顶元素插入至 A；
2. 当 $m \neq n$（即 N 为 **奇数**）：需向 B 添加一个元素。实现方法：将新元素 num 插入至 A，再将 A 堆顶元素插入至 B；

假设插入数字 num 遇到情况 `1`. 。由于 num 可能属于 “较小的一半” （即属于 B），因此不能将 $nums$ 直接插入至 A。而应先将 num 插入至 B，再将 B 堆顶元素插入至 A。这样就可以始终保持 A 保存较大一半、B 保存较小一半。

`findMedian()` 函数：

1. 当 $m = n$（即 N 为 **偶数**）：则中位数为 (A 的堆顶元素 + B 的堆顶元素)/2。
2. 当 $m \neq n$（即 N 为 **奇数**）：则中位数为 A 的堆顶元素。

Python 中 `heapq` 模块是小顶堆。实现 **大顶堆** 方法： 小顶堆的插入和弹出操作均将元素 **取反** 即可。

```
from heapq import *

class MedianFinder:
```

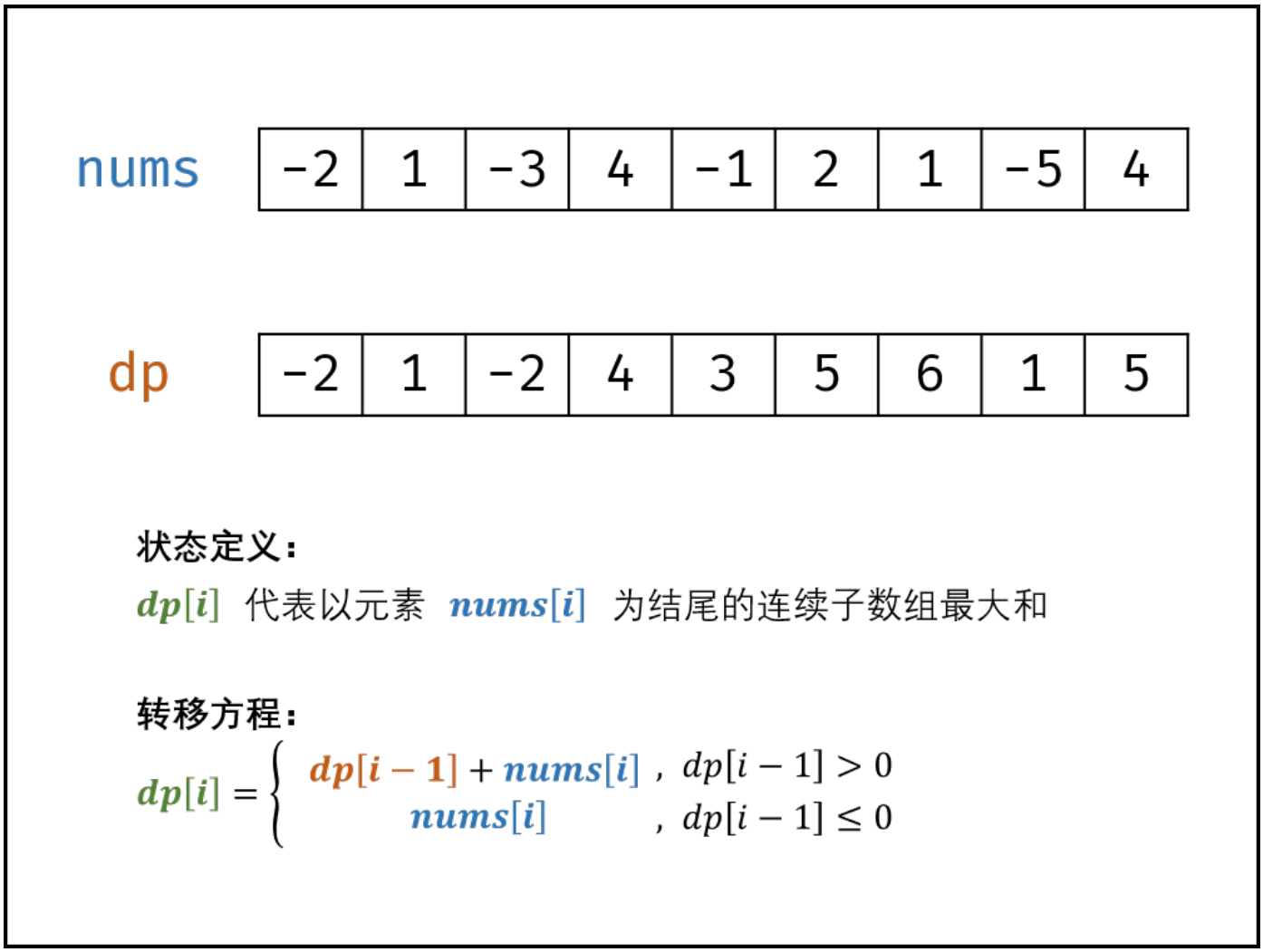
```
    def __init__(self):
        self.A = [] # 小顶堆，保存较大的一半
        self.B = [] # 大顶堆，保存较小的一半

    def addNum(self, num: int) -> None:
        if len(self.A) != len(self.B):
            heappush(self.A, num)
            heappush(self.B, -heappop(self.A))
        else:
            heappush(self.B, -num)
            heappush(self.A, -heappop(self.B))

    def findMedian(self) -> float:
        return self.A[0] if len(self.A) != len(self.B) else (self.A[0] - self.B[0]) /
2.0

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-ju-liu-zhong-de-zhong-wei-shu-
lcof/solution/mian-shi-ti-41-shu-ju-liu-zhong-de-zhong-wei-shu-y/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

```
from heapq import *

class MedianFinder:
    def __init__(self):
        self.A = [] # 小顶堆，保存较大的一半
        self.B = [] # 大顶堆，保存较小的一半

    def addNum(self, num: int) -> None:
        if len(self.A) != len(self.B):
            heappush(self.B, -heappushpop(self.A, num))
        else:
            heappush(self.A, -heappushpop(self.B, -num))

    def findMedian(self) -> float:
        return self.A[0] if len(self.A) != len(self.B) else (self.A[0] - self.B[0]) /
2.0

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-ju-liu-zhong-de-zhong-wei-shu-
lcof/solution/mian-shi-ti-41-shu-ju-liu-zhong-de-zhong-wei-shu-y/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

连续子数组的最大和：使用**动态规划**是最优的

- **状态定义**： 设动态规划列表 dp ，$dp[i]$ 代表以元素 $nums[i]$ 为结尾的连续子数组最大和。
  - 为何定义最大和 $dp[i]$ 中必须包含元素 $nums[i]$ ： 保证 $dp[i]$ 递推到 $dp[i+1]$ 的正确性；如果不包含 $nums[i]$ ，递推时则不满足题目的 **连续子数组** 要求。
- **转移方程**： 若 $dp[i-1] \leq 0$ ，说明 $dp[i-1]$ 对 $dp[i]$ 产生负贡献，即 $dp[i-1]+nums[i]$ 还不如 $nums[i]$ 本身大。
  - 当 $dp[i-1] > 0$ 时：执行 $dp[i] = dp[i-1] + nums[i]$ ；
  - 当 $dp[i-1] \leq 0$ 时：执行 $dp[i] = nums[i]$ ；
- **初始状态**： $dp[0] = nums[0]$，即以 $nums[0]$ 结尾的连续子数组最大和为 $nums[0]$ 。
- **返回值**： 返回 dp 列表中的最大值，代表全局最大值。



```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        for i in range(1, len(nums)):
            nums[i] += max(nums[i - 1], 0)
        return max(nums)

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/lian-xu-zi-shu-zu-de-zui-da-he-lcof/solution/mian-shi-ti-42-lian-xu-zi-shu-zu-de-zui-da-he-do-2/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

1～n 整数中 1 出现的次数:

如果用常规方法（$O(N^2)$），会直接超时。

设数字 n 是个 x 位数，记 n 的第 i 位为 n_i ，则可将 n 写为 $n_x n_{x-1} \cdots n_2 n_1$ ：

- 称 " n_i " 为 **当前位** ，记为 cur ，
- 将 " $n_{i-1} n_{i-2} \cdots n_2 n_1$ " 称为 **低位** ，记为 low ；
- 将 " $n_x n_{x-1} \cdots n_{i+2} n_{i+1}$ " 称为 **高位** ，记为 $high$ 。
- 将 10^i 称为 **位因子** ，记为 $digit$ 。

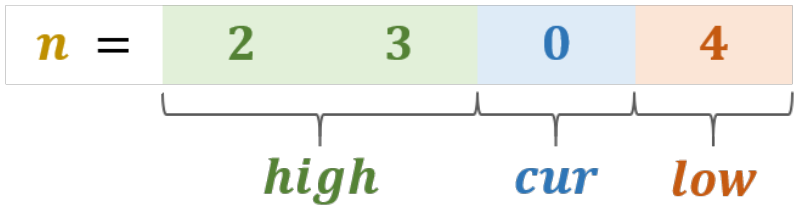
某位中 1 出现次数的计算方法：

根据当前位 cur 值的不同，分为以下三种情况：

- 当 $cur = 0$ 时： 此位 1 的出现次数只由高位 $high$ 决定，计算公式为：$high \times digit$

$digit = 10$　　即求“十位”的 1 的个数

将数字 n 划分为三部分：



出现 1 的数字范围：　$0010 \sim 2219$

只看高低位：　$000 \sim 229$

易得 1 出现次数为：　$229 - 0 + 1 = \mathbf{230}$

结论：

当此位 $cur = 0$ 时，此位 1 的个数的计算公式为：

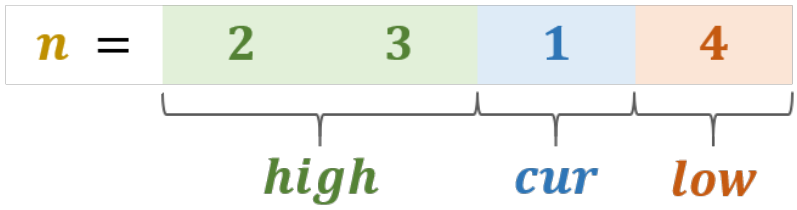
$$high \times digit$$

即 $23 \times 10 = 230$

- 当 $cur = 1$ 时：此位 1 的出现次数由高位 $high$ 和低位 low 决定，计算公式为：$high \times digit + low + 1$

$digit = 10$　　即求“十位”的 1 的个数

将数字 n 划分为三部分：



出现 1 的数字范围：　$0010 \sim 2314$

只看高低位：　$000 \sim 234$

易得 1 出现次数为：　$234 - 0 + 1 = \mathbf{235}$

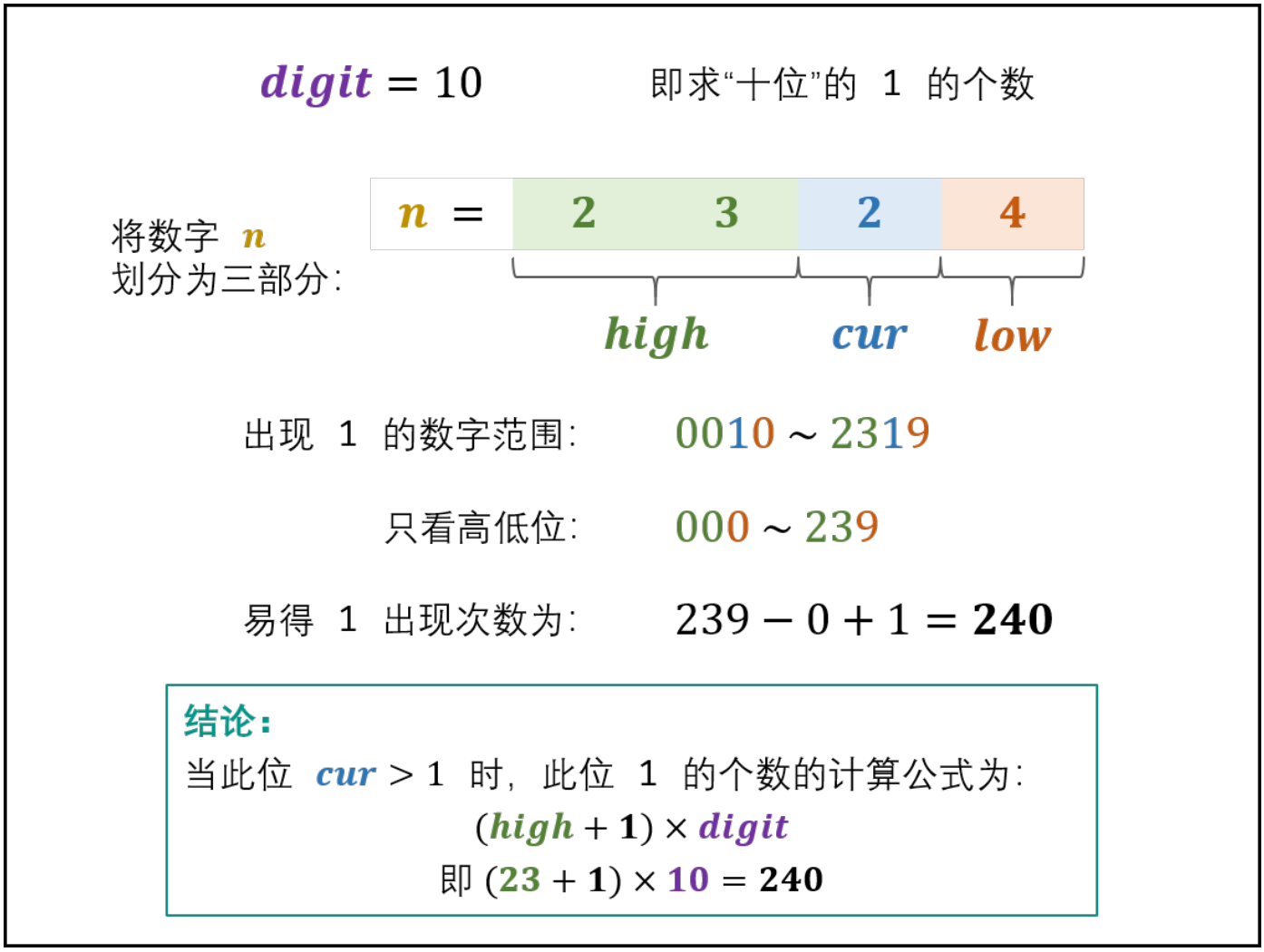
结论：

当此位 $cur = 1$ 时，此位 1 的个数的计算公式为：

$$high \times digit + low + 1$$

即 $23 \times 10 + 4 + 1 = 235$

- 当 $cur = 2, 3, \cdots, 9$ 时：此位 1 的出现次数只由高位 $high$ 决定，计算公式为：$(high + 1) \times digit$



设计按照 “个位、十位、...” 的顺序计算，则 $high/cur/low/digit$ 应初始化为：

```
high = n // 10
cur = n % 10
low = 0
digit = 1 # 个位
```

因此，从个位到最高位的变量递推公式为：

```
while high != 0 or cur != 0: # 当 high 和 cur 同时为 0 时，说明已经越过最高位，因此跳出
    low += cur * digit # 将 cur 加入 low ，组成下轮 low
    cur = high % 10 # 下轮 cur 是本轮 high 的最低位
    high //= 10 # 将本轮 high 最低位删除，得到下轮 high
    digit *= 10 # 位因子每轮 × 10
```

```
class Solution:
    def countDigitOne(self, n: int) -> int:
        digit, res = 1, 0
        high, cur, low = n // 10, n % 10, 0
        while high != 0 or cur != 0:
            if cur == 0: res += high * digit
```

```
            elif cur == 1: res += high * digit + low + 1
            else: res += (high + 1) * digit
            low += cur * digit
            cur = high % 10
            high //= 10
            digit *= 10
        return res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/1nzheng-shu-zhong-1chu-xian-de-ci-shu-lcof/solution/mian-shi-ti-43-1n-zheng-shu-zhong-1-chu-xian-de-2/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

-TBC-