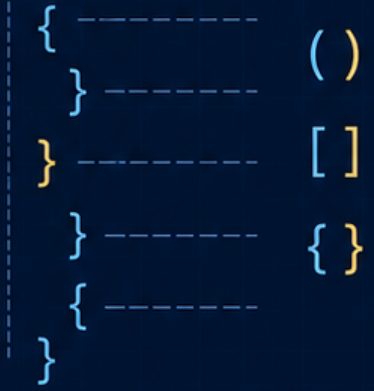
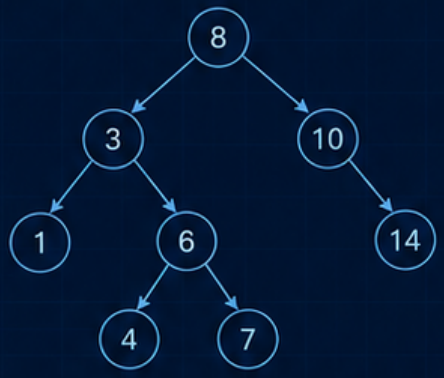
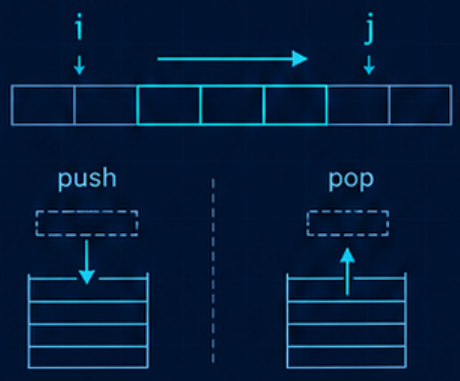
zero2Leetcode 蓝皮书

算法面试通关指南

高频手撕 · 八股文 · 大厂笔试 · AI Coding







Onefly

zero2Leetcode 蓝皮书

算法面试通关指南

版本：v0.1.0　日期：2026 年 8 月

作者：Onefly (github.com/ranxi2001)

目录

前言

《zero2Leetcode 蓝皮书》面向准备校招、实习与社招技术面试的读者，内容由四部分组成：面试高频手撕、计算机基础八股文、大厂笔试真题，以及语法、数据结构、测评和 AI Coding 技巧。

第一章以字节跳动高频手撕题单的六大分类为主线，其他公司统计只用于优先级参考和新增题补充。题解强调现场可讲、代码可写、复杂度可证，并在适合的题目中比较哈希表、双指针、滑动窗口、堆、二分和动态规划等多种方法。

第二至第四章在编译时直接读取网站原始资料，不在 `book` 目录维护副本。这样网站内容与出版内容共享同一事实来源，避免重复修改后产生版本漂移。

0.1 阅读建议

1. 面试临近时，优先完成第一章字节核心题，并练习每题的口述与追问。
2. 八股文按薄弱主题查缺补漏，不建议只背结论。
3. 笔试真题按公司和岗位限时完成，使用本机 Python 环境验证输入输出。
4. AI Coding 题重点训练需求拆解、测试设计、隐藏边界和限时交付。

第一章 面试高频手撕

1.1 链表

本文件维护第一章第一类题目的题解与面试追问。字节题单决定本类的核心范围和顺序，其他资料只用于覆盖数量参考与新增题补充。

1.1.1 字节核心题单

“其他统计覆盖”表示华为、美团、腾讯和 Hot 100 综合榜四份补充资料中，有多少份也记录了该题；不是可直接相加的考频。

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 206 反转链表	55	08 更新	4/4
2	LC 25 K 个一组翻转链表	55	08 更新	2/4
3	LC 19 删除链表的倒数第 N 个结点	43	08 更新	4/4
4	LC 23 合并 K 个升序链表	36	08 更新	4/4
5	LC 21 合并两个有序链表	29	08 更新	4/4
6	LC 141 环形链表	14	04 基线	4/4
7	LC 148 排序链表	13	04 基线	1/4
8	LC 92 反转链表 II	15	08 推算	2/4
9	LC 2 两数相加	11	04 基线	2/4

LC 92 的频次沿用字节原文推算值，不与其他来源频次相加。

1.1.2 其他来源新增题

题目	发现来源
LC 24 两两交换链表中的节点	腾讯
LC 82 删除排序链表中的重复元素 II	美团、腾讯
LC 138 随机链表的复制	Hot 100 综合榜
LC 142 环形链表 II	华为、美团、腾讯、Hot 100 综合榜
LC 143 重排链表	美团
LC 160 相交链表	华为、腾讯、Hot 100 综合榜
LC 234 回文链表	华为

1.1.3　题解与追问重点

- 反转类题目同时讲迭代、递归和区间反转，统一指针重连术语。
- 判环、找环入口和相交链表先给哈希表直观解法，再推导快慢指针或双指针的常数空间解法。
- 合并 K 个链表比较顺序合并、分治合并和最小堆。
- 删除、分组和重排题必须画清楚断链与重连位置，并覆盖头节点变化、空链表和不足一组等边界。

1.1.4　LC 206 反转链表

考频与考点　字节 55 次，其他统计覆盖 4/4。考查指针重连、原地修改，以及能否准确维护“上一结点、当前结点、下一结点”。

写代码前确认　确认是返回反转后的头结点；空链表和单结点链表应原样返回。若题目要求保留原链表，则不能使用下面的原地算法。

思路与解法

1. 迭代：先保存 `cur.next`，再令 `cur.next = prev`，最后整体向后移动。任一时刻，`prev` 都是已反转部分的头结点。
2. 递归：先反转后缀，再把后继结点指回当前结点。必须将原来的 `head.next` 置空，否则会形成环。

```
class Solution:
    def reverseList(self, head):
        prev, cur = None, head
        while cur:
            nxt = cur.next
            cur.next = prev
            prev, cur = cur, nxt
        return prev

    def reverseListRecursive(self, head):
        if head is None or head.next is None:
            return head
        new_head = self.reverseListRecursive(head.next)
        head.next.next = head
        head.next = None
        return new_head
```

复杂度　两种解法均为 `O(n)` 时间；迭代为 `O(1)` 额外空间，递归调用栈为 `O(n)`。

边界与易错点　不能在保存 `nxt` 前覆盖 `cur.next`；递归回溯时不能漏掉断开原指针；返回值是 `prev` 或递归得到的 `new_head`，不是原头结点。

面试追问　如何反转区间 `[left, right]`？如何只反转前 `k` 个结点并返回下一段起点？若链表极长，为什么迭代版本更稳妥？

1.1.5　LC 25 K 个一组翻转链表

考频与考点　字节 55 次，其他统计覆盖 2/4。重点是分组边界、局部反转和多段链表的正确拼接。

写代码前确认　不足 `k` 个结点的尾段保持原顺序；只能改变结点连接，不能交换结点值；通常可认为 `k >= 1`。

思路与解法

用哑结点统一处理头结点变化。`group_prev` 指向本组之前，先向后找到本组末尾 `kth`；找不到说明剩余不足一组。保存下一组起点 `group_next`，把区间 `[group_prev.next, group_next)` 原地反转，再将前后两段接回。递

归也可按同样边界处理，但调用栈为 O(n/k)，面试更推荐迭代。

```
class Solution:
    def reverseKGroup(self, head, k):
        dummy = ListNode(0, head)
        group_prev = dummy

        while True:
            kth = group_prev
            for _ in range(k):
                kth = kth.next
                if kth is None:
                    return dummy.next

            group_next = kth.next
            prev, cur = group_next, group_prev.next
            while cur is not group_next:
                nxt = cur.next
                cur.next = prev
                prev, cur = cur, nxt

            old_head = group_prev.next
            group_prev.next = kth
            group_prev = old_head
```

复杂度　每个结点至多被定位和反转各一次，时间 O(n)，额外空间 O(1)。

边界与易错点　必须先保存 group_next；反转时把 prev 初始化为 group_next，可直接接好尾部；下一轮的 group_prev 是本组反转前的头结点；k = 1 不应死循环。

面试追问　若不足 k 个也要反转，终止条件如何调整？如何抽象一个反转半开区间的函数？递归写法的栈深是多少？

1.1.6　LC 19 删除链表的倒数第 N 个结点

考频与考点　字节 43 次，其他统计覆盖 4/4。考查哑结点、固定间距双指针和一次遍历。

写代码前确认　通常保证 n 合法；删除的可能是头结点。若输入可能非法，需要约定返回原链表还是报错。

思路与解法

1. 两遍扫描：先求长度 L，再走到第 L−n 个前驱，逻辑直观。
2. 一遍双指针：从哑结点出发，让 fast 先走 n+1 步，使 slow 最终停在待删结点的前驱。哑结点消除了删除头结点的特判。

```
class Solution:
    def removeNthFromEnd(self, head, n):
        dummy = ListNode(0, head)
        fast = slow = dummy
        for _ in range(n + 1):
            fast = fast.next
        while fast:
            fast = fast.next
            slow = slow.next
        slow.next = slow.next.next
        return dummy.next
```

复杂度　两遍和双指针均为 O(n) 时间、O(1) 空间；双指针只扫描一轮。

边界与易错点　从 dummy 走 n+1 步与从 head 走 n 步是两套等价写法，不能混用；删除头结点时必须返回 dummy.next。

面试追问　若 n 可能大于链表长度怎样防御？只给待删结点而不给头结点能否完成删除？为什么尾结点不能用“复制后继值”的办法？

1.1.7　LC 23 合并 K 个升序链表

考频与考点　字节 36 次，其他统计覆盖 4/4。考查多路归并、最小堆、分治，以及 Python 堆中相同键的比较问题。

写代码前确认　输入可含空链表；结点总数记为 N、链表数记为 k；允许重用原结点。

思路与解法

1. 最小堆：每条非空链表只把当前头结点入堆。弹出最小结点后，再把它的后继入堆，堆大小不超过 k。元组中加入唯一序号，避免值相等时比较 ListNode。
2. 分治：两两合并，每轮链表数减半；每个结点参与 log k 层合并。它不依赖堆，且空间开销更低。

```
import heapq
from itertools import count


class Solution:
    def mergeKLists(self, lists):
        heap = []
        serial = count()
        for node in lists:
            if node:
                heapq.heappush(heap, (node.val, next(serial), node))

        dummy = tail = ListNode()
        while heap:
            _, _, node = heapq.heappop(heap)
            tail.next = node
            tail = node
            if node.next:
                heapq.heappush(
                    heap, (node.next.val, next(serial), node.next)
                )
        return dummy.next
```

复杂度　堆解法为 O(N log k) 时间、O(k) 空间；分治同为 O(N log k) 时间，迭代实现的辅助空间可为 O(1)。

边界与易错点　不要一次把全部 N 个结点放入堆；Python 中不能依赖 ListNode 直接比较；输入 lists=[] 或全为空时返回 None。

面试追问　顺序逐条合并为何最坏可达 O(Nk)？若链表来自流式数据如何处理？什么情况下分治比堆更合适？

1.1.8　LC 21 合并两个有序链表

考频与考点　字节 29 次，其他统计覆盖 4/4。考查链表基本操作、哑结点和归并排序的基础组件。

写代码前确认　链表按非递减顺序排列；可复用原结点；相等时先取哪一条链表不影响有序性，但应保持规则一致。

思路与解法

1. 迭代：用 `tail` 指向结果尾部，每次接入较小结点，最后一次性接入剩余链表。
2. 递归：较小头结点的 `next` 指向两条剩余链表的合并结果，代码短，但栈深可达 `m+n`。

```
class Solution:
    def mergeTwoLists(self, list1, list2):
        dummy = tail = ListNode()
        while list1 and list2:
            if list1.val <= list2.val:
                tail.next, list1 = list1, list1.next
            else:
                tail.next, list2 = list2, list2.next
            tail = tail.next
        tail.next = list1 or list2
        return dummy.next

    def mergeTwoListsRecursive(self, list1, list2):
        if not list1 or not list2:
            return list1 or list2
        if list1.val <= list2.val:
            list1.next = self.mergeTwoListsRecursive(list1.next, list2)
            return list1
        list2.next = self.mergeTwoListsRecursive(list1, list2.next)
        return list2
```

复杂度　时间 `O(m+n)`；迭代额外空间 `O(1)`，递归栈空间 `O(m+n)`。

边界与易错点　循环结束后不要逐个复制剩余结点；连接结点后要同步移动来源指针和 `tail`；空链表应直接接入另一条。

面试追问　如何改为降序合并？怎样合并数组形式的有序序列？这段逻辑如何复用于链表归并排序？

1.1.9　LC 141 环形链表

考频与考点　字节 14 次，其他统计覆盖 4/4。考查哈希判重和 Floyd 快慢指针。

写代码前确认　只需判断是否有环，不需要返回入口；不能通过结点值判断是否重复，因为值可以相同。

思路与解法

1. 哈希表：沿链表记录结点对象；再次遇到同一对象即有环。它最直观，时间 `O(n)`、空间 `O(n)`。
2. 快慢指针：慢指针每次一步、快指针每次两步。若无环，快指针到达空；若有环，两者进入环后相对距离每轮缩短一格，必然相遇。

```
class Solution:
    def hasCycle(self, head):
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow is fast:
                return True
        return False

    def hasCycleWithHash(self, head):
        seen = set()
```

```
        while head:
            if head in seen:
                return True
            seen.add(head)
            head = head.next
        return False
```

复杂度　两种解法时间均为 O(n)；哈希表空间 O(n)，快慢指针空间 O(1)。

边界与易错点　循环条件必须同时检查 fast 和 fast.next；比较的是结点身份；先移动再判断可自然处理自环。

面试追问　如何找环入口和环长？为什么快指针速度取两倍即可？若快指针每次走三步，是否一定能检测到环？

1.1.10　LC 148 排序链表

考频与考点　字节 13 次，其他统计覆盖 1/4。要求在 O(n log n) 时间完成链表排序，核心是快慢指针找中点和归并。

写代码前确认　允许修改链表连接；若严格要求 O(1) 额外空间，应使用自底向上的迭代归并，递归版有 O(log n) 调用栈。

思路与解法

递归归并排序：用 slow、fast 找到中点前驱并断链，将左右两半分别排序，再用 LC 21 的逻辑合并。分割时让 fast = head.next，可使双结点链表的 slow 停在第一个结点。

```
class Solution:
    def sortList(self, head):
        if head is None or head.next is None:
            return head

        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        right = slow.next
        slow.next = None

        left = self.sortList(head)
        right = self.sortList(right)
        dummy = tail = ListNode()
        while left and right:
            if left.val <= right.val:
                tail.next, left = left, left.next
            else:
                tail.next, right = right, right.next
            tail = tail.next
        tail.next = left or right
        return dummy.next
```

复杂度　时间 O(n log n)，递归栈 O(log n)；自底向上归并可将额外空间降到 O(1)。

边界与易错点　递归前必须断开左右链表，否则无法收敛；归并时要移动 tail；不要把链表转数组排序，那会使用 O(n) 额外空间。

面试追问　如何写自底向上归并？为什么链表适合归并排序而不适合常规快速排序？排序是否稳定？

1.1.11　LC 92 反转链表 II

考频与考点　字节推算 15 次，其他统计覆盖 2/4。考查局部反转、哑结点和区间边界。

写代码前确认　位置从 1 开始且 `left <= right`；区间可能包含头结点；通常保证位置合法。

思路与解法

头插法只扫描一次：先令 `prev` 停在第 `left-1` 个结点，`cur` 是区间首结点。之后每轮摘下 `cur.next`，插到 `prev` 后面。`cur` 始终是已反转区间的尾部，共执行 `right-left` 次。

```
class Solution:
    def reverseBetween(self, head, left, right):
        dummy = ListNode(0, head)
        prev = dummy
        for _ in range(left - 1):
            prev = prev.next

        cur = prev.next
        for _ in range(right - left):
            moved = cur.next
            cur.next = moved.next
            moved.next = prev.next
            prev.next = moved
        return dummy.next
```

复杂度　时间 `O(n)`，额外空间 `O(1)`。

边界与易错点　`left == right` 时不进入反转循环；包含头结点时依赖哑结点；头插过程中 `cur` 不移动，移动的是它的后继。

面试追问　如何用“反转半开区间”实现？怎样扩展为每 `k` 个一组反转？若区间位置可能越界，应如何处理已修改的前缀？

1.1.12　LC 2 两数相加

考频与考点　字节 11 次，其他统计覆盖 2/4。考查链表同步遍历、逐位进位和不同长度输入。

写代码前确认　数字按逆序保存，头结点是个位；每个结点是一位十进制数字；结果也按逆序返回。若题目改为正序存储，解法会不同。

思路与解法

同时遍历两条链表，把缺失位视为 0。每轮用 `divmod(x + y + carry, 10)` 得到新进位与当前位。循环条件包含 `carry`，因此最高位进位无需单独补结点。也可原地复用较长链表，但分支更多，面试中通常新建结果更清晰。

```
class Solution:
    def addTwoNumbers(self, l1, l2):
        dummy = tail = ListNode()
        carry = 0
        while l1 or l2 or carry:
            x = l1.val if l1 else 0
            y = l2.val if l2 else 0
            carry, digit = divmod(x + y + carry, 10)
            tail.next = ListNode(digit)
            tail = tail.next
            l1 = l1.next if l1 else None
```

```
            l2 = l2.next if l2 else None
        return dummy.next
```

复杂度　时间 O(max(m,n))，新结果链表占 O(max(m,n))；除返回结果外额外空间 O(1)。

边界与易错点　不要提前结束较长链表；最后可能多一位进位；divmod 返回顺序是商、余数；输入中的前导零规则应以题意为准。

面试追问　若数字正序存储如何用栈处理？若不允许修改输入且要求 O(1) 辅助空间能否完成？如何扩展到任意进制？

1.1.13　LC 24 两两交换链表中的节点

考频与考点　新增来源为腾讯。考查局部三指针重连，也是 K 组反转的最小特例。

写代码前确认　只能交换结点，不能只交换值；奇数长度链表的最后一个结点保持不动。

思路与解法

使用哑结点，令 prev 指向待交换二元组之前，first、second 是组内两个结点。按“first 接后段、second 接 first、prev 接 second”的顺序重连，再把 prev 移到 first。

```
class Solution:
    def swapPairs(self, head):
        dummy = ListNode(0, head)
        prev = dummy
        while prev.next and prev.next.next:
            first = prev.next
            second = first.next
            first.next = second.next
            second.next = first
            prev.next = second
            prev = first
        return dummy.next
```

复杂度　时间 O(n)，额外空间 O(1)。

边界与易错点　重连前保留两个结点；循环条件要保证一组有两个结点；交换后 first 成为该组尾部。

面试追问　递归版本的终止条件是什么？这题如何退化自 LC 25？若要求每三个结点循环右移一次如何修改？

1.1.14　LC 82 删除排序链表中的重复元素 II

考频与考点　新增来源为美团、腾讯。考查有序性、连续重复段识别和删除头部重复段。

写代码前确认　要删除所有出现重复的值，而不是每组保留一个；链表已经排序，因此相同值连续出现。

思路与解法

用哑结点和前驱 prev。若 cur 与后继值相同，记录该值并跳过整段；否则 prev、cur 同时前进。这样无需哈希表即可利用有序性完成原地删除。哈希计数也能做，但需两遍扫描和 O(n) 空间，没有利用题目条件。

```
class Solution:
    def deleteDuplicates(self, head):
        dummy = ListNode(0, head)
        prev, cur = dummy, head
        while cur:
            if cur.next and cur.val == cur.next.val:
```

```
                duplicate = cur.val
                while cur and cur.val == duplicate:
                    cur = cur.next
                prev.next = cur
            else:
                prev, cur = cur, cur.next
        return dummy.next
```

复杂度　时间 O(n)，额外空间 O(1)。

边界与易错点　发现重复后不能保留第一个；删除重复段时 prev 不移动；全重复或头部重复都依赖哑结点正确处理。

面试追问　若改为每个值保留一个怎样写？无序链表删除全部重复值需要什么额外结构？如何在不可修改原链表时返回新链表？

1.1.15　LC 138 随机链表的复制

考频与考点　新增来源为 Hot 100 综合榜。考查对象映射、深拷贝和利用链表结构进行空间优化。

写代码前确认　random 可指向任意结点或空；深拷贝要求新旧链表没有共享结点；结点值不保证唯一。

思路与解法

1. 哈希表：第一遍建立“原结点 → 新结点”映射，第二遍连接 next 和 random，最直观。
2. 穿插复制：把复制结点插在对应原结点之后，于是原结点 x 的复制结点是 x.next，其随机指针应指向 x.random.next；最后拆分两条链表，可省映射空间。

```
class Solution:
    def copyRandomList(self, head):
        if head is None:
            return None

        cur = head
        while cur:
            cur.next = Node(cur.val, cur.next)
            cur = cur.next.next

        cur = head
        while cur:
            if cur.random:
                cur.next.random = cur.random.next
            cur = cur.next.next

        copy_head = head.next
        cur = head
        while cur:
            copy = cur.next
            cur.next = copy.next
            copy.next = copy.next.next if copy.next else None
            cur = cur.next
        return copy_head
```

复杂度　两种解法时间均为 O(n)；哈希表空间 O(n)，穿插法额外空间 O(1)（不计返回链表）。

边界与易错点　映射键必须是结点对象而不是值；穿插法第三步必须恢复原链表；设置随机指针时要判断 random 是否为空。

面试追问　若链表含 next 环，穿插法还能否直接使用？如何复制一般有向图？为什么本题可以把映射关系编码在相邻位置？

1.1.16　LC 142 环形链表 II

考频与考点　新增来源覆盖 4/4。考查哈希定位、Floyd 判环及环入口的数学推导。

写代码前确认　无环返回 None；返回入口结点对象而非下标；不能修改链表。

思路与解法

1. 哈希表：第一个再次访问的结点就是环入口，直观但占 O(n) 空间。
2. Floyd：快慢指针相遇后，从头结点再出发一个指针；它与相遇点指针都每次走一步，首次相遇处就是入口。若头到入口距离为 a、入口到首次相遇距离为 b、环长为 c，相遇时有 2(a+b)=a+b+kc，故 a=(k−1)c+(c−b)。

```
class Solution:
    def detectCycle(self, head):
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow is fast:
                seeker = head
                while seeker is not slow:
                    seeker = seeker.next
                    slow = slow.next
                return seeker
        return None
```

复杂度　哈希解法 O(n) 时间和空间；Floyd 解法 O(n) 时间、O(1) 空间。

边界与易错点　第二阶段两个指针都走一步；不要在相遇后把快指针继续走两步；自环和入口为头结点都应正确返回。

面试追问　如何求环长？如何断开环？两条可能带环的链表如何判断是否相交？

1.1.17　LC 143 重排链表

考频与考点　新增来源为美团。考查找中点、反转后半段和交替合并三种链表基本操作的组合。

写代码前确认　要求原地重排为首、尾、次首、次尾的次序；不能改变结点值；函数通常不返回新头结点。

思路与解法

先用快慢指针找到前半段尾部并断链，再反转后半段，最后交替连接两条链表。数组保存全部结点后双指针重连更直观，但要 O(n) 空间；三阶段写法只用常数空间。

```
class Solution:
    def reorderList(self, head):
        if head is None or head.next is None:
            return

        slow = fast = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
```

```
        second = slow.next
        slow.next = None
        prev = None
        while second:
            nxt = second.next
            second.next = prev
            prev, second = second, nxt

        first, second = head, prev
        while second:
            next_first, next_second = first.next, second.next
            first.next = second
            second.next = next_first
            first, second = next_first, next_second
```

复杂度　时间 O(n)，额外空间 O(1)。

边界与易错点　反转前必须断开前半段；合并前保存双方后继；奇数长度时前半段多一个结点，可自然成为末尾。

面试追问　如何恢复原链表？若要求从中间向两端交替排列怎样处理？为什么这里选择前半段多一个结点更方便？

1.1.18　LC 160 相交链表

考频与考点　新增来源为华为、腾讯、Hot 100 综合榜。考查结点身份、哈希表和消除长度差的双指针技巧。

写代码前确认　相交指共享同一结点及其后缀，不是值相同；通常保证链表无环且结构不被修改。

思路与解法

1. 哈希表：记录 A 的所有结点，再遍历 B，首个命中的结点就是交点，便于直观说明。
2. 双指针：pa 走完 A 后转到 B，pb 走完 B 后转到 A。两者都走过 m+n 的组合路径，长度差被抵消；相交则在入口相遇，不相交则同时为 None。

```
class Solution:
    def getIntersectionNode(self, headA, headB):
        pa, pb = headA, headB
        while pa is not pb:
            pa = pa.next if pa else headB
            pb = pb.next if pb else headA
        return pa

    def getIntersectionNodeWithHash(self, headA, headB):
        seen = set()
        while headA:
            seen.add(headA)
            headA = headA.next
        while headB:
            if headB in seen:
                return headB
            headB = headB.next
        return None
```

复杂度　两者时间均为 O(m+n)；哈希表空间 O(m)，双指针空间 O(1)。

边界与易错点　使用 is 比较结点身份；切换链表发生在当前指针为空时；不相交时循环也会终止。

面试追问　如何先求长度再对齐？若链表可能有环，如何分类讨论相交？为什么交换路径不会无限循环？

1.1.19　LC 234 回文链表

考频与考点　新增来源为华为。考查中点、原地反转和链表结构恢复。

写代码前确认　空链表是否视为回文通常为是；是否允许修改输入链表；结点值可能重复或为负。

思路与解法

将值复制到数组后双指针比较最直观，时间、空间均为 O(n)。空间优化做法是快慢指针找中点，反转后半段，再从两端逐项比较。工程中宜在比较后再次反转，以恢复输入结构。

```
class Solution:
    def isPalindrome(self, head):
        if head is None or head.next is None:
            return True

        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        prev = None
        cur = slow
        while cur:
            nxt = cur.next
            cur.next = prev
            prev, cur = cur, nxt

        left, right = head, prev
        answer = True
        while right:
            if left.val != right.val:
                answer = False
                break
            left, right = left.next, right.next

        cur, prev = prev, None
        while cur:
            nxt = cur.next
            cur.next = prev
            prev, cur = cur, nxt
        return answer
```

复杂度　数组法为 O(n) 时间、O(n) 空间；反转法为 O(n) 时间、O(1) 空间。

边界与易错点　奇数长度时后半段可包含中点，不影响比较；提前发现不等也应恢复链表；不要用字符串拼接代替结点遍历。

面试追问　如何避免恢复时丢失后半段头结点？递归从两端比较为何仍需 O(n) 栈空间？只允许一次遍历是否可行？

1.2　二叉树与搜索图论

本文件维护第一章第二类题目的题解与面试追问。字节原文中的二叉树、网格搜索和图论题保持在同一分类。

1.2.1　字节核心题单

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 200 岛屿数量	50	08 更新	3/4
2	LC 236 二叉树的最近公共祖先	41	08 更新	2/4
3	LC 103 二叉树的锯齿形层序遍历	28	08 更新	2/4
4	LC 102 二叉树的层序遍历	27	08 更新	4/4
5	LC 199 二叉树的右视图	26	08 更新	2/4
6	LC 572 另一棵树的子树	17	08 更新	0/4
7	LC 104 二叉树的最大深度	22	04 基线	1/4
8	LC 94 二叉树的中序遍历	18	04 基线	1/4
9	LC 124 二叉树中的最大路径和	15	04 基线	3/4
10	LC 226 翻转二叉树	10	04 基线	0/4
11	LC 695 岛屿的最大面积	未单列	08 关联题	1/4

1.2.2　其他来源新增题

题目	发现来源
LC 108 将有序数组转换为二叉搜索树	Hot 100 综合榜
LC 111 二叉树的最小深度	美团
LC 114 二叉树展开为链表	Hot 100 综合榜
LC 144 二叉树的前序遍历	美团
LC 207 课程表	华为、Hot 100 综合榜
LC 208 实现 Trie（前缀树）	Hot 100 综合榜
LC 230 二叉搜索树中第 K 小的元素	Hot 100 综合榜
LC 437 路径总和 III	Hot 100 综合榜

1.2.3　题解与追问重点

- 每道递归题先定义函数语义、返回值和终止条件，再写代码。
- 遍历题比较递归与迭代；层序题统一使用队列模板，并讲清每层边界。
- 岛屿题比较 DFS、BFS 和并查集，说明原地标记与额外访问集合的取舍。
- 图论题补充有向图判环、拓扑排序和访问状态设计。
- 树题追问优先覆盖递归深度、退化树栈溢出、返回路径或具体节点等变体。

1.2.4　LC 200 岛屿数量

考频与考点：字节 50 次，本类最高频。考查网格建图、连通分量搜索和访问标记。

写代码前确认：网格是否允许原地修改；相邻是否只含上下左右。本题可修改时，将访问过的 `"1"` 改成 `"0"`，无需额外集合。

思路/解法：扫描每个格子。遇到尚未访问的陆地，答案加一，再从它出发用 DFS 淹没整个岛屿。递归 DFS 可写得更短，但极端网格可能超过 Python 递归深度，下面使用显式栈。BFS 只需将栈换成队列，复杂度相同。若

需要动态合并陆地或频繁查询连通性，可用并查集；对一次静态扫描没有优势。

```
from typing import List


class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid or not grid[0]:
            return 0

        rows, cols = len(grid), len(grid[0])
        answer = 0
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] != "1":
                    continue
                answer += 1
                grid[r][c] = "0"
                stack = [(r, c)]
                while stack:
                    x, y = stack.pop()
                    for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                        nx, ny = x + dx, y + dy
                        if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == "1":
                            grid[nx][ny] = "0"  # 入栈时标记，避免重复入栈
                            stack.append((nx, ny))
        return answer
```

复杂度：时间 O(mn)，每个格子至多处理一次；显式栈最坏 O(mn)。

边界/易错点：空网格、全水、单格岛屿；必须在入栈或入队时标记，不能等弹出后再标记。

面试追问：不能修改输入时如何用 visited？八方向相邻如何改？陆地逐个加入并实时询问岛屿数时如何用并查集？

1.2.5　LC 236 二叉树的最近公共祖先

考频与考点：字节 41 次。考查后序递归、子树信息向上传递，以及节点引用与节点值的区别。

写代码前确认：p、q 是否都存在且节点值是否唯一。原题保证都存在，应按节点对象比较。

思路/解法：定义 dfs(node)：若当前子树包含 p 或 q，返回能够代表该目标或其最近公共祖先的节点；都不含则返回空。左右子树均返回非空时，当前节点就是答案；只有一侧非空则向上传递该侧结果。

```
class Solution:
    def lowestCommonAncestor(self, root: "TreeNode", p: "TreeNode", q: "TreeNode") -> "TreeNode":
        def dfs(node: "TreeNode | None") -> "TreeNode | None":
            if node is None or node is p or node is q:
                return node
            left = dfs(node.left)
            right = dfs(node.right)
            if left is not None and right is not None:
                return node
            return left if left is not None else right

        return dfs(root)
```

复杂度：时间 `O(n)`；递归栈 `O(h)`，退化树最坏 `O(n)`。

边界/易错点：一个目标是另一个目标的祖先时，命中目标应立即返回；不要只比较 `val`。

面试追问：若节点可能不存在，如何同时返回命中数量？BST 如何利用大小关系？多次 LCA 查询如何做倍增预处理？

1.2.6　LC 103 二叉树的锯齿形层序遍历

考频与考点：字节 28 次。考查层序边界、双端队列以及方向切换。

写代码前确认：空树返回空列表；第一层方向为从左到右。

思路/解法：BFS 每轮先固定 `len(queue)`，确保只消费当前层。树节点始终按左右顺序入队；输出时根据方向追加到本层结果的右端或左端，避免每层结束后再反转。DFS 也可按深度建立结果，并依据奇偶深度插入两端，但会承受递归栈风险。

```
from collections import deque
from typing import List, Optional


class Solution:
    def zigzagLevelOrder(self, root: Optional["TreeNode"]) -> List[List[int]]:
        if root is None:
            return []
        queue = deque([root])
        left_to_right = True
        answer = []
        while queue:
            level = deque()
            for _ in range(len(queue)):
                node = queue.popleft()
                if left_to_right:
                    level.append(node.val)
                else:
                    level.appendleft(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            answer.append(list(level))
            left_to_right = not left_to_right
        return answer
```

复杂度：时间 `O(n)`，队列与结果之外的辅助空间最坏 `O(w)`，`w` 为最大层宽。

边界/易错点：不要改变子节点入队顺序来制造锯齿，否则下一层的结构顺序容易混乱。

面试追问：如何只用一个双端队列完成双向出队？若允许每层反转，代码和复杂度如何变化？

1.2.7　LC 102 二叉树的层序遍历

考频与考点：字节 27 次，多家公司共同高频。考查 BFS 队列和分层控制。

写代码前确认：返回值按层分组，而不是一维访问序列。

思路/解法：队列保存下一批待访问节点。每轮记录当前长度，只弹出这些节点，因此新加入的子节点自然属于下一层。递归方案可定义 `dfs(node, depth)`，首次到达新深度时创建列表，再追加节点值；BFS 更贴合题意。

```
from collections import deque
from typing import List, Optional


class Solution:
    def levelOrder(self, root: Optional["TreeNode"]) -> List[List[int]]:
        if root is None:
            return []
        answer, queue = [], deque([root])
        while queue:
            level = []
            for _ in range(len(queue)):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            answer.append(level)
        return answer
```

复杂度：时间 `O(n)`；队列辅助空间 `O(w)`，输出空间 `O(n)`。

边界/易错点：循环中必须保存本层长度；直接遍历不断增长的队列会丢失层边界。

面试追问：如何返回自底向上的层序？如何计算每层平均值？递归层序在退化树上有什么风险？

1.2.8　LC 199 二叉树的右视图

考频与考点：字节 26 次。考查层序遍历变体与先右后左的深度优先搜索。

写代码前确认：右视图是每一深度最右侧可见节点，并不等于沿 `right` 指针形成的链。

思路/解法：BFS 中每层最后弹出的节点就是该层最右节点。另一种常考写法是先右后左 DFS：定义 `dfs(node, depth)` 访问子树，当 `depth == len(answer)` 时说明这是该深度首次到达的节点，直接记录。

```
from collections import deque
from typing import List, Optional


class Solution:
    def rightSideView(self, root: Optional["TreeNode"]) -> List[int]:
        if root is None:
            return []
        answer, queue = [], deque([root])
        while queue:
            level_size = len(queue)
            for index in range(level_size):
                node = queue.popleft()
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
                if index == level_size - 1:
                    answer.append(node.val)
        return answer
```

复杂度：时间 `O(n)`，辅助空间 `O(w)`；DFS 方案为 `O(h)` 递归栈。

边界/易错点：某层最右节点可能来自左子树；实现时可直接在 `index == level_size - 1` 时记录。

面试追问：如何改为左视图？如何用 DFS 保证每层首个节点就是答案？如何返回每层最右节点对象？

1.2.9　LC 572 另一棵树的子树

考频与考点：字节 17 次。考查“枚举候选根 + 判断两棵树相同”的递归拆分。

写代码前确认：原题 `subRoot` 非空；相同必须同时满足结构和值一致，不能只比较某种遍历值序列。

思路/解法：定义 `same(a, b)`：判断两棵以 `a`、`b` 为根的树是否完全相同。再定义 `contains(node)`：判断 `node` 子树中是否存在与 `subRoot` 相同的根，检查当前节点后递归左右子树。

```
from typing import Optional


class Solution:
    def isSubtree(self, root: Optional["TreeNode"], subRoot: Optional["TreeNode"]) -> bool:
        def same(a: Optional["TreeNode"], b: Optional["TreeNode"]) -> bool:
            if a is None or b is None:
                return a is b
            return a.val == b.val and same(a.left, b.left) and same(a.right, b.right)

        def contains(node: Optional["TreeNode"]) -> bool:
            if node is None:
                return False
            return same(node, subRoot) or contains(node.left) or contains(node.right)

        return contains(root)
```

复杂度：最坏时间 `O(nm)`，递归栈 `O(h_root + h_sub)`；`n`、`m` 为两树节点数。

边界/易错点：序列化方案必须记录空节点和分隔符，否则不同结构或多位数可能误匹配。

面试追问：如何用带空标记的序列化加 KMP 降低匹配复杂度？如何用子树哈希比较？

1.2.10　LC 104 二叉树的最大深度

考频与考点：字节 22 次。考查最基础的树递归语义和高度定义。

写代码前确认：空树深度为 `0`，单节点树深度为 `1`。

思路/解法：定义 `depth(node)`：返回以 `node` 为根的子树最大深度。空节点为 `0`，非空节点为左右子树较大深度加一。迭代层序每完成一层将深度加一，可规避退化树的递归深度限制。

```
from typing import Optional


class Solution:
    def maxDepth(self, root: Optional["TreeNode"]) -> int:
        def depth(node: Optional["TreeNode"]) -> int:
            if node is None:
                return 0
            return max(depth(node.left), depth(node.right)) + 1

        return depth(root)
```

复杂度：时间 `O(n)`；递归栈 `O(h)`。

边界/易错点：不要混淆节点数定义的深度与边数定义的高度；本题按节点数返回。

面试追问：如何改为最小深度？如何在求深度的同时判断平衡树？极深树如何迭代实现？

1.2.11　LC 94 二叉树的中序遍历

考频与考点：字节 18 次。考查递归展开与显式栈模拟。

写代码前确认：顺序为左子树、根、右子树；是否要求避免递归。

思路/解法：递归中定义 `dfs(node)`：按中序把当前子树节点值追加到结果。迭代法用栈保存“左链上尚未访问的根”，先一路压左，弹栈访问，再转向右子树。面试通常应掌握两种。

```
from typing import List, Optional


class Solution:
    def inorderTraversal(self, root: Optional["TreeNode"]) -> List[int]:
        answer, stack = [], []
        current = root
        while current is not None or stack:
            while current is not None:
                stack.append(current)
                current = current.left
            current = stack.pop()
            answer.append(current.val)
            current = current.right
        return answer
```

递归核心等价于：`dfs(node.left)`、记录 `node.val`、`dfs(node.right)`；迭代法把函数调用栈显式化，复杂度不变。

复杂度：时间 `O(n)`；栈空间 `O(h)`。

边界/易错点：外层条件必须是 `current is not None or stack`；弹栈访问后转向右子树。

面试追问：如何写前序、后序迭代？Morris 中序如何做到 `O(1)` 额外空间，它是否临时修改树？

1.2.12　LC 124 二叉树中的最大路径和

考频与考点：字节 15 次，多家公司高频。考查树形 DP、全局答案与返回值语义分离。

写代码前确认：路径至少包含一个节点；可从任意节点开始和结束，不能重复节点；节点值可能全为负数。

思路/解法：定义 `gain(node)`：从 `node` 出发、只能向下选择一条分支时能提供给父节点的最大贡献。负贡献应舍弃为 `0`。经过当前节点且同时连接左右分支的路径不能继续向父节点延伸，只用于更新全局答案。

```
from typing import Optional


class Solution:
    def maxPathSum(self, root: Optional["TreeNode"]) -> int:
        answer = float("-inf")

        def gain(node: Optional["TreeNode"]) -> int:
            nonlocal answer
            if node is None:
```

```
                return 0
            left = max(gain(node.left), 0)
            right = max(gain(node.right), 0)
            answer = max(answer, node.val + left + right)
            return node.val + max(left, right)

        gain(root)
        return int(answer)
```

复杂度：时间 O(n)；递归栈 O(h)。

边界/易错点：全负树不能把答案初始化为 0；返回父节点时不能同时带上左右两条分支。

面试追问：如何返回具体路径？如何改为只允许叶子到叶子？与二叉树直径的递推关系有何异同？

1.2.13　LC 226 翻转二叉树

考频与考点：字节 10 次。考查递归定义、原地修改与树遍历。

写代码前确认：是否允许修改原树。原题要求返回翻转后的根，通常原地交换左右子树。

思路/解法：定义 invert(node)：翻转以 node 为根的整棵子树并返回其根。先递归翻转左右子树，再交换返回结果。也可 BFS 逐节点交换，避免递归栈过深。

```
from typing import Optional


class Solution:
    def invertTree(self, root: Optional["TreeNode"]) -> Optional["TreeNode"]:
        def invert(node: Optional["TreeNode"]) -> Optional["TreeNode"]:
            if node is None:
                return None
            left = invert(node.left)
            right = invert(node.right)
            node.left, node.right = right, left
            return node

        return invert(root)
```

复杂度：时间 O(n)；递归栈 O(h)。

边界/易错点：若先覆盖一个子指针再赋另一个，会丢失原子树；应使用并行赋值或临时变量。

面试追问：如何生成一棵新镜像树而不修改原树？如何迭代实现？翻转两次后树是否必然恢复？

1.2.14　LC 695 岛屿的最大面积

考频与考点：字节关联题。考查连通分量搜索过程中聚合节点数量。

写代码前确认：面积按四方向相邻的陆地格数计算；是否允许原地修改输入。

思路/解法：与 LC 200 相同地枚举每个未访问陆地，但每次搜索累计当前连通分量面积，并更新最大值。下面采用迭代 DFS。BFS 的访问顺序不同，结果和复杂度相同；并查集适合需要合并、查询多个分量大小的场景。

```
from typing import List


class Solution:
```

```
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        if not grid or not grid[0]:
            return 0
        rows, cols = len(grid), len(grid[0])
        answer = 0
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] != 1:
                    continue
                area = 0
                grid[r][c] = 0
                stack = [(r, c)]
                while stack:
                    x, y = stack.pop()
                    area += 1
                    for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                        nx, ny = x + dx, y + dy
                        if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                            grid[nx][ny] = 0
                            stack.append((nx, ny))
                answer = max(answer, area)
        return answer
```

复杂度：时间 `O(mn)`；栈最坏 `O(mn)`。

边界/易错点：全水时返回 `0`；应按连通分量重置 `area`，并在入栈时标记。

面试追问：如何返回最大岛屿的坐标集合？若翻转一个水格，如何求最大可能面积？

1.2.15　LC 108 将有序数组转换为二叉搜索树

考频与考点：其他来源新增。考查分治构造、BST 中序有序性质和平衡条件。

写代码前确认：输入严格递增；高度平衡只要求任意节点左右子树高度差不超过一，答案不唯一。

思路/解法：定义 `build(left, right)`：用 `nums[left:right + 1]` 构造一棵平衡 BST 并返回根。选中点为根后，左右区间长度至多差一，递归构造左右子树。使用下标而非切片，避免额外复制。

```
from typing import List, Optional


class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional["TreeNode"]:
        def build(left: int, right: int) -> Optional["TreeNode"]:
            if left > right:
                return None
            middle = left + (right - left) // 2
            root = TreeNode(nums[middle])
            root.left = build(left, middle - 1)
            root.right = build(middle + 1, right)
            return root

        return build(0, len(nums) - 1)
```

复杂度：时间 `O(n)`；递归栈 `O(log n)`，构造结果占 `O(n)`。

边界/易错点：空区间终止条件是 `left > right`；左右中点任选其一都合法。

面试追问：输入为有序链表时如何做到 `O(n)`？如何证明中点构造必然高度平衡？

1.2.16　LC 111 二叉树的最小深度

考频与考点：其他来源新增。考查叶节点定义和 BFS 最短性。

写代码前确认：最小深度是根到最近叶节点的节点数；只有一个孩子的节点不是叶子。

思路/解法：BFS 按层搜索，首次遇到左右孩子都为空的节点即可返回当前深度，因为队列保证后续节点不会更浅。递归也可定义 `depth(node)` 返回子树最小深度，但单侧为空时不能直接取 `min(0, x)`。

```
from collections import deque
from typing import Optional


class Solution:
    def minDepth(self, root: Optional["TreeNode"]) -> int:
        if root is None:
            return 0
        queue = deque([(root, 1)])
        while queue:
            node, depth = queue.popleft()
            if node.left is None and node.right is None:
                return depth
            if node.left:
                queue.append((node.left, depth + 1))
            if node.right:
                queue.append((node.right, depth + 1))
        return 0
```

复杂度：最坏时间 `O(n)`；队列空间 `O(w)`。若浅层很快出现叶子，BFS 可提前结束。

边界/易错点：根只有右子树时，答案必须沿右侧到叶子，不能把缺失的左子树深度当作 `0` 参与最小值。

面试追问：递归应如何处理单侧为空？若边带非负权，如何改用 Dijkstra 求最短根叶路径？

1.2.17　LC 114 二叉树展开为链表

考频与考点：其他来源新增。考查原地指针重排、前序顺序和后序递归信息。

写代码前确认：必须原地展开；所有 `left` 置空，`right` 链顺序等于原树前序遍历。

思路/解法：定义 `flatten_tree(node)`：原地展开当前子树，并返回展开链表的尾节点。先保存并展开左右子树；若左子树存在，将其接到根的右侧，再把原右链接到左链尾部。显式栈按前序取节点并连接前驱也可实现，空间同为 `O(h)` 到 `O(n)`。

```
from typing import Optional


class Solution:
    def flatten(self, root: Optional["TreeNode"]) -> None:
        def flatten_tree(node: Optional["TreeNode"]) -> Optional["TreeNode"]:
            if node is None:
                return None
            left_root, right_root = node.left, node.right
            left_tail = flatten_tree(left_root)
            right_tail = flatten_tree(right_root)
            if left_root is not None:
```

```
                node.right = left_root
                node.left = None
                left_tail.right = right_root
            return right_tail or left_tail or node

        flatten_tree(root)
```

复杂度：时间 `O(n)`；递归栈 `O(h)`，不计递归栈的额外空间为 `O(1)`。

边界/易错点：递归前先保存原右子树；不要每次扫描左链尾部，否则退化为 `O(n^2)`。

面试追问：如何用前驱节点实现 `O(1)` 额外空间？如何展开为中序顺序？

1.2.18　LC 144 二叉树的前序遍历

考频与考点：其他来源新增。考查根、左、右的访问顺序及迭代模拟。

写代码前确认：空树返回空列表；是否限制使用递归。

思路/解法：递归函数 `dfs(node)` 的语义是按前序把当前子树追加到结果。迭代法弹出根后先压右孩子、再压左孩子，利用栈后进先出保证左子树先访问。

```
from typing import List, Optional


class Solution:
    def preorderTraversal(self, root: Optional["TreeNode"]) -> List[int]:
        if root is None:
            return []
        answer, stack = [], [root]
        while stack:
            node = stack.pop()
            answer.append(node.val)
            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)
        return answer
```

复杂度：时间 `O(n)`；显式栈最坏 `O(h)`。

边界/易错点：压栈顺序与访问顺序相反；若先压左孩子，实际会先访问右子树。

面试追问：如何统一写出前、中、后序迭代模板？Morris 前序如何恢复临时线索？

1.2.19　LC 207 课程表

考频与考点：华为与 Hot 100 新增。考查有向图判环、拓扑排序和入度维护。

写代码前确认：`[course, prerequisite]` 表示从先修课指向课程；只需判断能否完成，不要求返回具体顺序。

思路/解法：Kahn BFS 先把所有入度为零的课程入队。每学完一门课，就删除它的出边并降低后继入度；最终处理数量等于课程总数则无环。三色 DFS 也可判环：`0` 未访问、`1` 当前递归路径、`2` 已完成，遇到状态 `1` 即发现环；BFS 无递归深度风险。

```
from collections import deque
from typing import List
```

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = [[] for _ in range(numCourses)]
        indegree = [0] * numCourses
        for course, prerequisite in prerequisites:
            graph[prerequisite].append(course)
            indegree[course] += 1

        queue = deque(i for i, degree in enumerate(indegree) if degree == 0)
        completed = 0
        while queue:
            course = queue.popleft()
            completed += 1
            for next_course in graph[course]:
                indegree[next_course] -= 1
                if indegree[next_course] == 0:
                    queue.append(next_course)
        return completed == numCourses
```

复杂度：时间 `O(V + E)`；邻接表、入度和队列空间 `O(V + E)`。

边界/易错点：建图方向必须与入度定义一致；无先修关系时所有课程都可完成；重复边是否存在应在面试中确认。

面试追问：如何返回一条拓扑序？如何用 DFS 返回具体环？若持续新增依赖，如何维护可行性？

1.2.20　LC 208 实现 Trie（前缀树）

考频与考点：Hot 100 新增。考查多叉树设计、前缀共享与单词结束标记。

写代码前确认：字符集范围；本题为小写英文字母。`search` 要求完整单词，`startsWith` 只要求前缀路径存在。

思路/解法：每个节点保存“字符到子节点”的映射和 `is_end` 标记。插入沿字符创建路径；查询沿路径行走，完整单词还需检查末节点标记。用字典便于适配稀疏字符集；固定 26 长度数组常数更稳定但占用更多空间。

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def _find(self, text: str):
```

```
        node = self.root
        for char in text:
            if char not in node.children:
                return None
            node = node.children[char]
        return node

    def search(self, word: str) -> bool:
        node = self._find(word)
        return node is not None and node.is_end

    def startsWith(self, prefix: str) -> bool:
        return self._find(prefix) is not None
```

复杂度：单次插入或查询时间 `O(L)`；总空间与所有已创建字符节点数成正比。

边界/易错点：不能用“没有孩子”代表单词结束，因为一个单词可能是另一个单词的前缀。

面试追问：如何支持删除并回收节点？如何返回指定前缀的前 `k` 个词？字符集很大时节点怎样存储？

1.2.21　LC 230 二叉搜索树中第 K 小的元素

考频与考点：Hot 100 新增。考查 BST 中序有序性质和迭代遍历提前终止。

写代码前确认：`k` 从 `1` 开始且合法；BST 是否允许重复值。原题通常按节点排名，不需要去重。

思路/解法：中序遍历 BST 会得到递增序列。使用显式栈压入左链，每弹出一个节点就将 `k` 减一，减到零立即返回，无需遍历剩余节点。递归也可通过外部计数提前记录答案，但 Python 中迭代更直接。

```
from typing import Optional


class Solution:
    def kthSmallest(self, root: Optional["TreeNode"], k: int) -> int:
        stack = []
        current = root
        while current is not None or stack:
            while current is not None:
                stack.append(current)
                current = current.left
            current = stack.pop()
            k -= 1
            if k == 0:
                return current.val
            current = current.right
        raise ValueError("k exceeds the number of nodes")
```

复杂度：时间平均 `O(h + k)`、最坏 `O(n)`；栈空间 `O(h)`。

边界/易错点：访问根后必须进入右子树；题目保证合法时不会走到异常分支。

面试追问：若频繁查询第 `k` 小，如何在节点维护子树大小并做到 `O(h)`？如何求第 `k` 大？

1.2.22　LC 437 路径总和 III

考频与考点：Hot 100 新增。考查树上前缀和、哈希计数和回溯恢复现场。

写代码前确认：路径必须向下但不必从根开始或到叶结束；节点值可为负，因此不能使用滑动窗口。

思路/解法：定义 `dfs(node, prefix)`：统计遍历当前子树时，以根到父节点的路径和为 `prefix` 的所有目标路径。到当前节点的新前缀和为 `current`，此前出现过 `current - targetSum` 的次数，就是以当前节点结尾的新路径数。哈希表保存当前递归路径上的前缀和频次；退出节点时减一，防止把兄弟分支拼成路径。

```
from typing import Optional


class Solution:
    def pathSum(self, root: Optional["TreeNode"], targetSum: int) -> int:
        count = {0: 1}

        def dfs(node: Optional["TreeNode"], prefix: int) -> int:
            if node is None:
                return 0
            current = prefix + node.val
            answer = count.get(current - targetSum, 0)
            count[current] = count.get(current, 0) + 1
            answer += dfs(node.left, current)
            answer += dfs(node.right, current)
            count[current] -= 1
            if count[current] == 0:
                del count[current]
            return answer

        return dfs(root, 0)
```

复杂度：时间 `O(n)`；哈希表与递归栈空间 `O(h)`。

边界/易错点：必须初始化 `count[0] = 1` 才能统计从根开始的路径；回溯时必须撤销当前前缀和。

面试追问：朴素的“从每个节点重新向下搜索”为何是 `O(n^2)`？如何返回所有具体路径？为什么负数使双指针失效？

1.3　动态规划

本文件维护第一章第三类题目的题解与面试追问。所有题解都要明确状态定义、转移来源、初始化、遍历顺序和答案位置。

1.3.1　字节核心题单

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 5 最长回文子串	46	08 更新	3/4
2	LC 300 最长递增子序列	43	08 更新	3/4
3	LC 72 编辑距离	34	08 更新	3/4
4	LC 1143 最长公共子序列	23	08 更新	1/4
5	LC 53 最大子数组和	30	04 基线	3/4
6	LC 121 买卖股票的最佳时机	23	04 基线	3/4
7	LC 322 零钱兑换	14	04 基线	2/4
8	LC 122 买卖股票的最佳时机 II	12	04 基线	0/4

顺序	题目	字节频次	数据版本	其他统计覆盖
9	LC 198 打家劫舍	8	04 基线	0/4
10	LC 70 爬楼梯	7	04 基线	3/4

1.3.2　其他来源新增题

题目	发现来源
LC 32 最长有效括号	美团、Hot 100 综合榜
LC 64 最小路径和	华为、Hot 100 综合榜
LC 118 杨辉三角	Hot 100 综合榜
LC 120 三角形最小路径和	美团
LC 279 完全平方数	Hot 100 综合榜
LC 416 分割等和子集	Hot 100 综合榜
LC 647 回文子串	美团
LC 718 最长重复子数组	美团、腾讯
LC 918 环形子数组的最大和	腾讯
LC 1262 可被三整除的最大和	腾讯

1.3.3　题解与追问重点

- 先给二维或完整状态，再讨论滚动数组、状态压缩和原地更新。
- LC 300 必须比较 $O(n^2)$ 动态规划与 $O(n \log n)$ 贪心加二分。
- 回文题比较动态规划与中心扩展，明确子串、子数组和子序列的区别。
- 股票题不仅给公式，还要解释状态机含义及交易次数变化时如何扩展。
- 背包题重点说明物品与容量的遍历顺序，避免把 0-1 背包写成完全背包。

1.3.4　LC 5 最长回文子串

考频与考点：字节高频题。考查区间动态规划、中心扩展，以及“子串必须连续”这一约束。

写代码前确认：输入非空；长度相同的最优答案返回任意一个即可。先确认面试官更关注 DP 模型，还是更短的中心扩展实现。

状态与思路：令 `dp[i][j]` 表示闭区间 `s[i:j+1]` 是否为回文。转移为 `s[i] == s[j] and (j - i <= 1 or dp[i+1][j-1])`。单字符初始化为真；`i` 必须从右向左遍历，`j` 从 `i` 向右遍历，保证内部状态已计算。答案不在固定单元格，而是在遍历中维护最长区间。中心扩展使用相同的回文结构，空间降为常数，是面试编码首选。

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        dp = [[False] * n for _ in range(n)]
        start = 0
        best_len = 1

        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                dp[i][j] = s[i] == s[j] and (j - i <= 1 or dp[i + 1][j - 1])
```

```
                if dp[i][j] and j - i + 1 > best_len:
                    start, best_len = i, j - i + 1
        return s[start:start + best_len]
```

```
from typing import Tuple

class SolutionCenter:
    def longestPalindrome(self, s: str) -> str:
        left = right = 0

        def expand(i: int, j: int) -> Tuple[int, int]:
            while i >= 0 and j < len(s) and s[i] == s[j]:
                i -= 1
                j += 1
            return i + 1, j - 1

        for center in range(len(s)):
            for i, j in (expand(center, center), expand(center, center + 1)):
                if j - i > right - left:
                    left, right = i, j
        return s[left:right + 1]
```

复杂度：两种解法时间均为 `O(n^2)`；DP 空间 `O(n^2)`，中心扩展空间 `O(1)`。

边界与易错点：偶数长度回文需要以两个字符为中心；DP 的遍历方向不能让 `dp[i+1][j-1]` 尚未计算。

面试追问：若求最长回文子序列，状态仍是区间 DP，但字符不等时可舍弃任一端；若要求线性时间，可进一步讨论 Manacher 算法。

1.3.5　LC 300 最长递增子序列

考频与考点：字节高频题。必须掌握 `O(n^2)` DP，以及 `O(n log n)` 的贪心加二分优化。

写代码前确认：题目要求严格递增，因此二分找第一个“大于等于”当前位置的元素；子序列不要求连续。

状态与思路：DP 定义 `dp[i]` 为以 `nums[i]` 结尾的最长递增子序列长度；若 `j < i` 且 `nums[j] < nums[i]`，转移为 `dp[i] = max(dp[i], dp[j] + 1)`。所有位置初始化为 1，按 `i` 从左向右、`j` 在其左侧遍历，答案为 `max(dp)`。优化解法维护 `tails[k]`：长度为 `k+1` 的递增子序列可取得的最小末尾值；它不是实际答案序列，但越小越利于接入后续元素。

```
from typing import List

class SolutionDP:
    def lengthOfLIS(self, nums: List[int]) -> int:
        dp = [1] * len(nums)
        for i in range(len(nums)):
            for j in range(i):
                if nums[j] < nums[i]:
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp)
```

```
from typing import List
```

```python
from bisect import bisect_left

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        tails: List[int] = []
        for value in nums:
            index = bisect_left(tails, value)
            if index == len(tails):
                tails.append(value)
            else:
                tails[index] = value
        return len(tails)
```

复杂度：DP 时间 `O(n^2)`、空间 `O(n)`；贪心加二分时间 `O(n log n)`、空间 `O(n)`。

边界与易错点：严格递增用 `bisect_left`；若允许相等则用 `bisect_right`。不要把 `tails` 直接当作原数组中的一条 LIS。

面试追问：如何恢复一条实际 LIS？为每个位置记录前驱与其在 `tails` 中的层级，再从最后一层回溯。

1.3.6　LC 72 编辑距离

考频与考点：字节高频二维 DP。考查两个字符串前缀之间的最优编辑代价。

写代码前确认：一次操作可以插入、删除或替换一个字符，每种代价均为 1。

状态与思路：`dp[i][j]` 表示 `word1[:i]` 转成 `word2[:j]` 的最少操作数。末字符相等时继承 `dp[i-1][j-1]`；否则取删除 `dp[i-1][j]`、插入 `dp[i][j-1]`、替换 `dp[i-1][j-1]` 的最小值再加 1。初始化 `dp[i][0] = i`、`dp[0][j] = j`；按前缀长度递增遍历；答案在 `dp[m][n]`。

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(m + 1):
            dp[i][0] = i
        for j in range(n + 1):
            dp[0][j] = j

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    dp[i][j] = 1 + min(
                        dp[i - 1][j],
                        dp[i][j - 1],
                        dp[i - 1][j - 1],
                    )
        return dp[m][n]
```

复杂度：时间 `O(mn)`，空间 `O(mn)`；只保留上一行可压缩到 `O(n)`。

边界与易错点：数组下标表示前缀长度，字符下标要减 1；“从 word1 插入字符”等价于目标前缀缩短一位，对应左侧状态。

面试追问：操作代价不同时如何修改？把三个候选分别加各自代价；如何输出操作序列？从 `dp[m][n]` 反向

追踪转移来源。

1.3.7　LC 1143 最长公共子序列

考频与考点：考查双序列 DP，是编辑距离、最长重复子数组等模型的基础。

写代码前确认：求子序列而非连续子串；相对顺序必须保留。

状态与思路：`dp[i][j]` 表示 `text1[:i]` 与 `text2[:j]` 的 LCS 长度。末字符相等时由 `dp[i-1][j-1] + 1` 转移；否则舍弃任一末字符，取 `max(dp[i-1][j], dp[i][j-1])`。空前缀所在行列初始化为 0；两个维度均递增；答案为 `dp[m][n]`。

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        return dp[m][n]
```

复杂度：时间 `O(mn)`，空间 `O(mn)`；滚动数组可降至 `O(min(m,n))`。

边界与易错点：字符相等时不能写成同一行或同一列加 1；这会重复使用字符。

面试追问：最长公共子串为何不同？其状态表示“以两个位置结尾的公共连续段”，字符不等时必须归零。

1.3.8　LC 53 最大子数组和

考频与考点：字节高频题。考查 Kadane 算法和连续子数组状态设计。

写代码前确认：子数组不能为空；全为负数时应返回最大的负数。

状态与思路：`dp[i]` 表示必须以 `nums[i]` 结尾的最大子数组和，转移为 `max(nums[i], dp[i-1] + nums[i])`。`dp[0] = nums[0]`，从左向右遍历，答案是所有 `dp[i]` 的最大值而非最后一个状态。由于只依赖前一项，可压缩为变量 `ending`。

```
from typing import List

class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        ending = answer = nums[0]
        for value in nums[1:]:
            ending = max(value, ending + value)
            answer = max(answer, ending)
        return answer
```

复杂度：时间 `O(n)`，空间 `O(1)`。

边界与易错点：不能将初值设为 0，否则全负数组会错误地选择空数组。

面试追问：如何返回区间？当 `ending` 从当前值重新开始时更新候选左端点，在全局答案变大时记录左右端点。

1.3.9　LC 121 买卖股票的最佳时机

考频与考点：单次交易模型，考查状态机 DP 与前缀最小值。

写代码前确认：只能买卖各一次，且必须先买后卖；允许不交易，此时利润为 0。

状态与思路：第 `i` 天结束后，`cash` 表示未持股最大利润，`hold` 表示持股最大利润。转移为 `cash = max(cash, hold + price)`、`hold = max(hold, -price)`，后者只能从未交易状态买入。初始化 `cash = 0`、`hold = -prices[0]`；按天从左到右；答案为最后的 `cash`。等价写法是维护历史最低价格。

```
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price = prices[0]
        answer = 0
        for price in prices[1:]:
            answer = max(answer, price - min_price)
            min_price = min(min_price, price)
        return answer
```

复杂度：时间 `O(n)`，空间 `O(1)`。

边界与易错点：不能用未来最低价减当前价格；最低价只能来自卖出日之前。

面试追问：与 LC 122 的差别在哪里？LC 121 的持股状态只能由初始现金买入，LC 122 可用此前卖出所得再次买入。

1.3.10　LC 322 零钱兑换

考频与考点：完全背包最少数量问题，重点是可重复选择与不可达状态。

写代码前确认：硬币可无限次使用；只求最少枚数，不要求方案数或具体组合。

状态与思路：`dp[x]` 表示凑成金额 `x` 的最少硬币数，转移为 `dp[x] = min(dp[x], dp[x-coin] + 1)`。`dp[0] = 0`，其余初始化为无穷。按硬币在外、金额从小到大遍历体现完全背包；若只求最少数，金额在外也能得到正确值。答案为 `dp[amount]`，仍为无穷则返回 `-1`。

```
from typing import List

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        unreachable = amount + 1
        dp = [unreachable] * (amount + 1)
        dp[0] = 0
        for coin in coins:
            for value in range(coin, amount + 1):
                dp[value] = min(dp[value], dp[value - coin] + 1)
        return -1 if dp[amount] == unreachable else dp[amount]
```

复杂度：时间 `O(n * amount)`，空间 `O(amount)`。

边界与易错点：金额必须正序遍历才允许同一硬币重复使用；不可达初值不能设为 0。

面试追问：若求组合数，硬币在外可避免不同排列重复计数；若求排列数，则金额在外、硬币在内。

1.3.11　LC 122 买卖股票的最佳时机 II

考频与考点：无限次交易的股票状态机，也可转化为累加所有正收益。

写代码前确认：同一时刻最多持有一股，卖出后可以再次买入；允许当天卖出再买入不影响最大利润。

状态与思路：`cash`、`hold` 分别表示当天结束未持股和持股的最大利润。转移为 `new_cash = max(cash, hold + price)`、`new_hold = max(hold, cash - price)`；初始化为 `0` 和 `-prices[0]`；从左到右遍历；答案是最终 `cash`。由于每段上涨都可独立兑现，也可贪心累加相邻正差值。

```
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        cash, hold = 0, -prices[0]
        for price in prices[1:]:
            cash, hold = max(cash, hold + price), max(hold, cash - price)
        return cash
```

```
from typing import List

class SolutionGreedy:
    def maxProfit(self, prices: List[int]) -> int:
        return sum(max(0, prices[i] - prices[i - 1]) for i in range(1, len(prices)))
```

复杂度：两种解法均为时间 `O(n)`、空间 `O(1)`。

边界与易错点：状态转移应基于上一日状态；Python 同时赋值右侧会先整体求值，因此这里可以安全更新。

面试追问：加入手续费、冷冻期或最多 `k` 次交易后，贪心通常失效，但持股/未持股状态机可以继续扩展。

1.3.12　LC 198 打家劫舍

考频与考点：线性 DP，核心是相邻位置不可同时选择。

写代码前确认：金额非负；房屋首尾不相邻。若首尾也相邻，则是环形版本。

状态与思路：`dp[i]` 表示考虑前 `i` 间房的最高金额，转移为 `dp[i] = max(dp[i-1], dp[i-2] + nums[i-1])`。初始化 `dp[0] = 0`、`dp[1] = nums[0]`；按房屋顺序向右；答案在 `dp[n]`。代码用两个变量保存 `dp[i-2]` 与 `dp[i-1]`。

```
from typing import List

class Solution:
    def rob(self, nums: List[int]) -> int:
        two_back = one_back = 0
        for money in nums:
            two_back, one_back = one_back, max(one_back, two_back + money)
        return one_back
```

复杂度：时间 `O(n)`，空间 `O(1)`。

边界与易错点：滚动更新时右侧必须同时基于旧值；只有一间房时也能由统一初始化处理。

面试追问：环形房屋如何处理？分别计算“不选最后一间”和“不选第一间”的线性答案，再取最大值。

1.3.13　LC 70 爬楼梯

考频与考点：最基础的线性 DP，用于考查状态定义与空间压缩。

写代码前确认：每次只能走 1 或 2 阶，`n >= 1`；顺序不同视为不同走法。

状态与思路：`dp[i]` 表示到达第 `i` 阶的方法数，最后一步来自 `i-1` 或 `i-2`，故 `dp[i] = dp[i-1] + dp[i-2]`。初始化 `dp[0] = 1`、`dp[1] = 1`；按阶数递增；答案为 `dp[n]`。只依赖前两项，可滚动压缩。

```
class Solution:
    def climbStairs(self, n: int) -> int:
        previous, current = 1, 1
        for _ in range(n):
            previous, current = current, previous + current
        return previous
```

复杂度：时间 `O(n)`，空间 `O(1)`。

边界与易错点：`dp[0] = 1` 表示“什么都不做”这一种空方案，是递推的中性起点。

面试追问：步长扩展为集合时可做完全背包；若 `n` 极大，可用矩阵快速幂将时间降至 `O(log n)`。

1.3.14　LC 32 最长有效括号

考频与考点：新增高频题。考查位置 DP，或用栈维护最近一个无法匹配的位置。

写代码前确认：求连续子串长度；输入只含左右括号。

状态与思路：`dp[i]` 表示必须以 `s[i]` 结尾的最长有效括号长度。只有 `s[i] == ')'` 才可能转移：若前一字符为左括号，加上 `dp[i-2]`；否则跳过前一段有效串，检查位置 `i-dp[i-1]-1` 是否为左括号，再接上更早的有效串。数组初始化为 0；`i` 从 1 向右；答案为 `max(dp)`。

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        dp = [0] * len(s)
        answer = 0
        for i in range(1, len(s)):
            if s[i] != ")":
                continue
            if s[i - 1] == "(":
                dp[i] = 2 + (dp[i - 2] if i >= 2 else 0)
            else:
                left = i - dp[i - 1] - 1
                if left >= 0 and s[left] == "(":
                    dp[i] = dp[i - 1] + 2
                    if left > 0:
                        dp[i] += dp[left - 1]
            answer = max(answer, dp[i])
        return answer
```

复杂度：时间 `O(n)`，空间 `O(n)`；双向计数扫描可做到 `O(1)` 空间。

边界与易错点：匹配 `...))` 时，要先跨过 `dp[i-1]` 找潜在左括号，并拼接左侧已有有效段。

面试追问：栈解法为何先压入 `-1`？它代表最近一个不可参与匹配的位置，使当前有效长度可以直接用下标差得到。

1.3.15　LC 64 最小路径和

考频与考点：新增二维网格 DP，考查边界初始化与原地状态压缩。

写代码前确认：只能向右或向下，网格非空；是否允许覆盖输入需要提前确认。

状态与思路：`dp[i][j]` 表示到达 `(i,j)` 的最小路径和，转移为当前值加上方、左方的较小者。左上角初始化为自身，首行只能来自左侧，首列只能来自上方；按行从左到右遍历；答案在右下角。代码直接把 `grid` 当作 DP 表，代价是覆盖原数据。

```
from typing import List

class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        for i in range(rows):
            for j in range(cols):
                if i == 0 and j == 0:
                    continue
                if i == 0:
                    grid[i][j] += grid[i][j - 1]
                elif j == 0:
                    grid[i][j] += grid[i - 1][j]
                else:
                    grid[i][j] += min(grid[i - 1][j], grid[i][j - 1])
        return grid[-1][-1]
```

复杂度：时间 `O(mn)`，除输入外空间 `O(1)`；不修改输入时可用 `O(n)` 滚动数组。

边界与易错点：首行和首列只有一个来源，不能用不存在的方向参与最小值比较。

面试追问：若允许四个方向，普通 DP 的无环依赖被破坏，应建图后使用 Dijkstra 等最短路算法。

1.3.16　LC 118 杨辉三角

考频与考点：新增基础 DP。考查上一行到下一行的构造关系。

写代码前确认：返回前 `numRows` 行，而不是某一行；每行两端固定为 1。

状态与思路：`triangle[i][j]` 表示第 `i` 行第 `j` 个数，内部位置由上一行相邻两数相加得到。第一行初始化为 `[1]`；按行递增构造，每行先填充 1，再更新内部位置；答案是完整二维列表。

```
from typing import List

class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        triangle: List[List[int]] = []
        for row_index in range(numRows):
            row = [1] * (row_index + 1)
            for j in range(1, row_index):
                row[j] = triangle[-1][j - 1] + triangle[-1][j]
            triangle.append(row)
        return triangle
```

复杂度：共生成 `O(numRows^2)` 个数，时间和输出空间均为 `O(numRows^2)`。

边界与易错点：只有一行时内部循环为空；输出本身需要二维空间，不能将其算作可省略的辅助空间。

面试追问：只求第 `k` 行时如何降到 `O(k)` 空间？使用一维数组并从右向左原地更新，避免覆盖上一层依赖。

1.3.17　LC 120 三角形最小路径和

考频与考点：新增不规则网格 DP，适合考查自底向上压缩。

写代码前确认：相邻下一层位置是同下标或下标加一；路径必须从顶到底。

状态与思路：自底向上定义 `dp[j]` 为从当前层位置 `j` 到底部的最小路径和。以最后一行为初始化；处理第 `i` 层时，转移为 `dp[j] = triangle[i][j] + min(dp[j], dp[j+1])`。层从倒数第二层向上、位置从左向右；最终答案在 `dp[0]`。

```
from typing import List

class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        dp = triangle[-1][:]
        for i in range(len(triangle) - 2, -1, -1):
            for j in range(i + 1):
                dp[j] = triangle[i][j] + min(dp[j], dp[j + 1])
        return dp[0]
```

复杂度：时间 `O(n^2)`，空间 `O(n)`。

边界与易错点：自底向上时 `dp[j]` 与 `dp[j+1]` 均来自下一层；若改为自顶向下原地更新，需要从右向左。

面试追问：若要恢复路径，可保留二维 DP，并从顶点依次选择代价更小的两个下一层状态。

1.3.18　LC 279 完全平方数

考频与考点：新增完全背包题，与零钱兑换同构。

写代码前确认：每个完全平方数可以重复使用，目标是最少数量。

状态与思路：`dp[value]` 表示组成 `value` 的最少平方数个数，枚举平方数 `square` 后转移 `dp[value] = min(dp[value], dp[value-square] + 1)`。`dp[0] = 0`，其余为无穷；平方数在外，容量从小到大；答案为 `dp[n]`。

```
from math import isqrt

class Solution:
    def numSquares(self, n: int) -> int:
        dp = [0] + [n + 1] * n
        for base in range(1, isqrt(n) + 1):
            square = base * base
            for value in range(square, n + 1):
                dp[value] = min(dp[value], dp[value - square] + 1)
        return dp[n]
```

复杂度：时间 `O(n * sqrt(n))`，空间 `O(n)`。

边界与易错点：容量正序表示完全背包；`isqrt` 可避免浮点开方误差。

面试追问：也可将每个整数看成图节点，用 BFS 找最少层数；数论解法可依据四平方和定理进一步优化。

1.3.19　LC 416 分割等和子集

考频与考点：新增 0-1 背包可达性问题，遍历方向是关键。

写代码前确认：每个元素只能使用一次；总和为奇数时必然无解。

状态与思路：目标容量是总和的一半，`dp[value]` 表示能否从已处理元素中选出和为 `value` 的子集。`dp[0] = True`，其余为假；对每个数，容量必须从目标值向下遍历，转移为 `dp[value] |= dp[value-num]`；答案在 `dp[target]`。倒序保证本轮不会重复使用当前元素。

```
from typing import List

class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total = sum(nums)
        if total % 2:
            return False
        target = total // 2
        dp = [False] * (target + 1)
        dp[0] = True
        for num in nums:
            for value in range(target, num - 1, -1):
                dp[value] = dp[value] or dp[value - num]
        return dp[target]
```

复杂度：时间 `O(n * target)`，空间 `O(target)`。

边界与易错点：容量若正序更新，会把一个元素使用多次，错误地变成完全背包。

面试追问：若要求恰好选 `k` 个数，应增加“已选数量”维度；若有负数，容量下标需平移或改用集合维护可达和。

1.3.20　LC 647 回文子串

考频与考点：新增回文 DP。与 LC 5 共用状态，但目标由最长长度变为计数。

写代码前确认：相同内容出现在不同位置要分别计数；单字符也是回文子串。

状态与思路：`dp[i][j]` 表示闭区间是否为回文，转移为 `s[i] == s[j] and (j - i <= 1 or dp[i+1][j-1])`。初始化依靠统一短区间条件；`i` 从右到左、`j` 从 `i` 向右；每个为真的状态给答案加一，答案是所有状态之和。

```
class Solution:
    def countSubstrings(self, s: str) -> int:
        n = len(s)
        dp = [[False] * n for _ in range(n)]
        count = 0
        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                dp[i][j] = s[i] == s[j] and (j - i <= 1 or dp[i + 1][j - 1])
                count += int(dp[i][j])
        return count
```

复杂度：时间和空间均为 `O(n^2)`；中心扩展可将空间降到 `O(1)`。

边界与易错点：这是按位置计数，不应使用集合去重；遍历方向仍需满足对内部区间的依赖。

面试追问：中心扩展总共有 `2n-1` 个中心，每成功扩展一次就得到一个不同位置的回文子串。

1.3.21　LC 718 最长重复子数组

考频与考点：新增双序列 DP。名称是“子数组”，因此要求连续。

写代码前确认：两个数组中的重复片段都必须连续；只返回长度。

状态与思路：`dp[i][j]` 表示分别以 `nums1[i-1]`、`nums2[j-1]` 结尾的最长公共连续片段长度。元素相等时由左上角加一，否则归零。空前缀行列初始化为 0；两个维度递增；答案是所有 `dp[i][j]` 的最大值，不一定在右下角。

```
from typing import List

class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        m, n = len(nums1), len(nums2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        answer = 0
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if nums1[i - 1] == nums2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    answer = max(answer, dp[i][j])
        return answer
```

复杂度：时间 `O(mn)`，空间 `O(mn)`；一维倒序更新可降到 `O(n)`。

边界与易错点：字符不等时状态必须为 0，不能像 LCS 那样取上方或左方最大值。

面试追问：数据规模更大时，可二分答案并用滚动哈希检查公共片段，或使用后缀自动机。

1.3.22　LC 918 环形子数组的最大和

考频与考点：新增 Kadane 变形。考查把跨边界区间转化为“总和减去中间最小子数组”。

写代码前确认：子数组不能为空；每个元素最多使用一次。

状态与思路：分别维护以当前位置结尾的最大和 `max_ending` 与最小和 `min_ending`，转移均为“从当前重启或接在前段之后”；初始化为首元素，按数组从左向右遍历。非环形答案是最大子数组和，环形答案是总和减最小子数组和。若最大值小于 0，说明全为负数，不能用总和减整个数组得到空数组，直接返回最大值。

```
from typing import List

class Solution:
    def maxSubarraySumCircular(self, nums: List[int]) -> int:
        total = nums[0]
        max_ending = max_sum = nums[0]
        min_ending = min_sum = nums[0]
        for value in nums[1:]:
            max_ending = max(value, max_ending + value)
            max_sum = max(max_sum, max_ending)
            min_ending = min(value, min_ending + value)
            min_sum = min(min_sum, min_ending)
            total += value
        return max_sum if max_sum < 0 else max(max_sum, total - min_sum)
```

复杂度：时间 `O(n)`，空间 `O(1)`。

边界与易错点：全负数组必须特判，否则环形候选会错误地选择空子数组。

面试追问：如何返回环形区间下标？同时记录最大、最小子数组边界；跨边界答案对应最小区间之外的两段。

1.3.23　LC 1262 可被三整除的最大和

考频与考点：新增余数状态 DP。考查将大数值容量压缩为有限模状态。

写代码前确认：每个元素最多选择一次，可以选择空集，因此答案至少为 0。

状态与思路：`dp[r]` 表示已处理元素中，和模 3 为 `r` 的最大值。初始化 `dp = [0, -inf, -inf]`；处理每个数时必须基于旧数组转移，选择它后更新余数 `(r + num) % 3`。按元素从左到右，余数维度遍历旧状态；答案在 `dp[0]`。

```
from typing import List

class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        negative_infinity = float("-inf")
        dp: List[float] = [0, negative_infinity, negative_infinity]
        for num in nums:
            next_dp = dp[:]
            for remainder in range(3):
                if dp[remainder] != negative_infinity:
                    new_remainder = (remainder + num) % 3
                    next_dp[new_remainder] = max(
                        next_dp[new_remainder], dp[remainder] + num
                    )
            dp = next_dp
        return int(dp[0])
```

复杂度：时间 `O(n)`，空间 `O(1)`，因为余数状态固定为 3 个。

边界与易错点：不可达状态不能初始化为 0；直接原地更新可能在同一轮重复选择当前元素。

面试追问：若要求和能被 `k` 整除，只需把状态扩展为 `k` 个余数，时间 `O(nk)`、空间 `O(k)`。

1.4　双指针、滑动窗口与字符串

本文件维护第一章第四类题目的题解与面试追问。本类重点不是背指针模板，而是说明指针为何可以单向移动，以及哈希表如何减少窗口内的重复查找。

1.4.1　字节核心题单

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 3 无重复字符的最长子串	174	08 更新	3/4
2	LC 15 三数之和	44	08 更新	4/4
3	LC 42 接雨水	38	08 更新	4/4
4	LC 88 合并两个有序数组	29	04 基线	2/4
5	LC 209 长度最小的子数组	12	04 基线	2/4
6	LC 239 滑动窗口最大值	9	04 基线	4/4

1.4.2　其他来源新增题

题目	发现来源
LC 8 字符串转换整数（atoi）	美团
LC 11 盛最多水的容器	华为
LC 14 最长公共前缀	美团

题目	发现来源
LC 16 最接近的三数之和	美团
LC 43 字符串相乘	美团、腾讯
LC 76 最小覆盖子串	腾讯、Hot 100 综合榜
LC 283 移动零	Hot 100 综合榜
LC 438 找到字符串中所有字母异位词	Hot 100 综合榜
LC 468 验证 IP 地址	美团

1.4.3　题解与追问重点

- LC 3 比较集合维护窗口与“最近出现位置”哈希表，解释左指针为什么只能前进，以及哈希表如何让它直接跳转。
- LC 15 必须给出排序加双指针和枚举加哈希表两种解法，重点比较去重难度、空间成本和面试推荐写法。
- LC 15 与第五类的 LC 1 组成专题：哈希表用于快速找补数，双指针用于在有序区间中缩小搜索范围。
- LC 42 比较前后缀最大值、双指针和单调栈，说明每种解法真正累加水量的时机。
- 滑动窗口题统一说明窗口含义、扩张条件、收缩条件和答案更新时机。
- 字符串解析题必须列出空串、前导零、符号、非法字符、溢出和分隔符边界。

1.4.4　LC 3 无重复字符的最长子串

考频与考点：字节 174 次，是本章最高频题。考查可变滑动窗口、哈希集合与最近位置哈希表。

写代码前确认：答案是子串长度，子串必须连续；字符集是否只含 ASCII 不影响字典写法；空串返回 `0`。

思路/解法一：集合维护合法窗口。窗口 `[left, right]` 始终无重复。若新字符已在集合中，持续移除左端字符并右移 `left`，直到可加入新字符。每个字符最多进出集合一次。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        window = set()
        left = answer = 0
        for right, char in enumerate(s):
            while char in window:
                window.remove(s[left])
                left += 1
            window.add(char)
            answer = max(answer, right - left + 1)
        return answer
```

思路/解法二：最近位置哈希表直接跳转。记录每个字符最近一次出现的下标。新字符上次出现在当前窗口内时，`left` 可直接跳到该位置后一格；必须取 `max`，因为左边界只能前进，不能被窗口外的旧位置拉回。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        last_index = {}
        left = answer = 0
        for right, char in enumerate(s):
            if char in last_index:
                left = max(left, last_index[char] + 1)
            last_index[char] = right
```

```
            answer = max(answer, right - left + 1)
        return answer
```

复杂度：两种方法时间均为 `O(n)`、空间均为 `O(min(n, 字符集大小))`。最近位置法减少了逐个收缩窗口的操作，表达更直接。

边界/易错点：`"abba"` 可检查左边界是否错误回退；更新最近位置和答案的先后顺序要与窗口定义一致。

面试追问：若每个字符最多出现 `k` 次，窗口状态如何维护？如何返回最长子串本身？输入是字符流且不能保存全部字符串时怎么办？

1.4.5　LC 15 三数之和

考频与考点：字节 44 次，所有统计源均覆盖。考查排序、双指针、哈希补数和结果去重，是 LC 1“两数之和”的自然扩展。

写代码前确认：返回数值三元组而非下标；三元组内使用三个不同位置；答案不能重复，输出顺序不限。

思路/解法一：排序 + 双指针。排序后枚举第一个数 `nums[i]`，剩余区间转化为目标值为 `-nums[i]` 的两数之和。和偏小则左指针右移，和偏大则右指针左移，这是有序性保证的单调排除。三层去重缺一不可：固定数与前一个相同时跳过；找到答案后分别跳过左侧重复值和右侧重复值；最后同时收缩两端。若 `nums[i] > 0`，之后不可能得到零，可提前结束。

```
from typing import List


class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        answer = []
        for i in range(len(nums) - 2):
            if nums[i] > 0:
                break
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            left, right = i + 1, len(nums) - 1
            while left < right:
                total = nums[i] + nums[left] + nums[right]
                if total < 0:
                    left += 1
                elif total > 0:
                    right -= 1
                else:
                    answer.append([nums[i], nums[left], nums[right]])
                    left_value, right_value = nums[left], nums[right]
                    while left < right and nums[left] == left_value:
                        left += 1
                    while left < right and nums[right] == right_value:
                        right -= 1
        return answer
```

思路/解法二：枚举 + 哈希表。固定一个位置后，从其右侧扫描第二个数；哈希集合保存已经扫描过的值，以平均 `O(1)` 查找补数。与 LC 1 的一次扫描哈希完全同构，但三数之和还要对三元组归一化并用集合去重，因此空间和去重成本更高，面试中通常优先排序双指针。

```
from typing import List


class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        triples = set()
        for i in range(len(nums) - 2):
            seen = set()
            for j in range(i + 1, len(nums)):
                complement = -nums[i] - nums[j]
                if complement in seen:
                    triples.add(tuple(sorted((nums[i], nums[j], complement))))
                seen.add(nums[j])
        return [list(triple) for triple in triples]
```

复杂度：排序双指针时间 `O(n^2)`，排序额外空间取决于实现；哈希法平均时间 `O(n^2)`，除答案外额外空间 `O(n)`。

边界/易错点：长度不足三、全为零、大量重复值；找到答案后只移动一侧会重复输出。排序会修改输入，若不允许应先复制。

面试追问：为何双指针必须依赖有序性？如何推广到目标值不为零、四数之和或 `kSum`？若要求返回原下标，应如何设计去重？

1.4.6　LC 42 接雨水

考频与考点：字节 38 次，所有统计源均覆盖。考查按列计算、边界最大值、双指针不变量和单调栈。

写代码前确认：每格宽度为 1，高度非负；长度小于三必然无法蓄水。

思路/解法一：前后缀最大值。第 `i` 列水量由其左侧最高柱与右侧最高柱中较矮者决定：`min(left_max[i], right_max[i]) - height[i]`。该方法最直观，适合先推导正确公式。

```
from typing import List


class Solution:
    def trap(self, height: List[int]) -> int:
        n = len(height)
        if n < 3:
            return 0
        left_max = [0] * n
        right_max = [0] * n
        left_max[0], right_max[-1] = height[0], height[-1]
        for i in range(1, n):
            left_max[i] = max(left_max[i - 1], height[i])
        for i in range(n - 2, -1, -1):
            right_max[i] = max(right_max[i + 1], height[i])
        return sum(min(left_max[i], right_max[i]) - height[i] for i in range(n))
```

思路/解法二：双指针。维护两端迄今最高值。若 `left_max <= right_max`，左侧当前列的短板已经确定为 `left_max`，未来右侧再高也不会改变它，因此可立即结算并右移左指针；反之结算右侧。空间降为常数。

```
from typing import List
```

```
class Solution:
    def trap(self, height: List[int]) -> int:
        left, right = 0, len(height) - 1
        left_max = right_max = water = 0
        while left <= right:
            if left_max <= right_max:
                left_max = max(left_max, height[left])
                water += left_max - height[left]
                left += 1
            else:
                right_max = max(right_max, height[right])
                water += right_max - height[right]
                right -= 1
        return water
```

思路/解法三：单调栈。栈保存高度单调不增的柱下标。当前柱更高时，弹出的柱是凹槽底部；新栈顶与当前柱形成左右边界，一次计算一层横向水量。它更适合追问“每个凹槽如何形成”。

```
from typing import List


class Solution:
    def trap(self, height: List[int]) -> int:
        stack, water = [], 0
        for right, current_height in enumerate(height):
            while stack and current_height > height[stack[-1]]:
                bottom = stack.pop()
                if not stack:
                    break
                left = stack[-1]
                width = right - left - 1
                bounded_height = min(height[left], current_height) - height[bottom]
                water += width * bounded_height
            stack.append(right)
        return water
```

复杂度：三种方法时间均为 `O(n)`；前后缀空间 `O(n)`，双指针空间 `O(1)`，单调栈空间 `O(n)`。

边界/易错点：单调序列、全等高度、多个相同边界；双指针比较的是已知边界最大值，而非简单比较当前两根柱后随意结算。

面试追问：三种算法分别按列还是按层累加？为什么单调栈在弹出后若为空不能计算？二维接雨水为何需要最小堆？

1.4.7　LC 88 合并两个有序数组

考频与考点：字节 29 次。考查逆向双指针和原地覆盖安全性。

写代码前确认：`nums1` 尾部有 `n` 个占位空间；有效元素分别为前 `m` 个和前 `n` 个；要求修改 `nums1`。

思路/解法：若从前向后写，`nums1` 中尚未比较的元素可能被覆盖。从两数组有效区末尾比较，把较大值写入 `nums1` 最末空位，写指针不断左移。最后只需补齐 `nums2` 剩余部分；若 `nums1` 有剩余，它们已在正确位置。

```
from typing import List
```

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        first, second, write = m - 1, n - 1, m + n - 1
        while second >= 0:
            if first >= 0 and nums1[first] > nums2[second]:
                nums1[write] = nums1[first]
                first -= 1
            else:
                nums1[write] = nums2[second]
                second -= 1
            write -= 1
```

复杂度：时间 `O(m + n)`；额外空间 `O(1)`。

边界/易错点：`m = 0`、`n = 0`、重复值；循环条件以 `second >= 0` 为准即可。

面试追问：若 `nums1` 没有尾部空间怎么办？如何合并两个有序流？为何正向合并需要额外数组？

1.4.8　LC 209 长度最小的子数组

考频与考点：字节 12 次。考查正数数组上的可变滑动窗口。

写代码前确认：原题元素均为正数，这是窗口和具有单调性的关键；不存在答案时返回 `0`。

思路/解法：窗口 `[left, right]` 表示当前连续子数组。右端加入新数使总和增大；当总和达到目标时，持续右移左边界寻找以当前右端结尾的最短合法窗口，并在删除左端前更新答案。若允许负数，该单调性消失，应考虑前缀和加单调队列。

```
from typing import List


class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        left = window_sum = 0
        answer = len(nums) + 1
        for right, value in enumerate(nums):
            window_sum += value
            while window_sum >= target:
                answer = min(answer, right - left + 1)
                window_sum -= nums[left]
                left += 1
        return 0 if answer == len(nums) + 1 else answer
```

复杂度：时间 `O(n)`，每个元素最多进入和离开窗口一次；额外空间 `O(1)`。

边界/易错点：单元素已达标、总和不足；答案要在收缩前更新，否则会漏掉合法窗口。

面试追问：如何用前缀和加二分做到 `O(n log n)`？包含负数时为什么普通滑窗错误，怎样改成单调队列？

1.4.9　LC 239 滑动窗口最大值

考频与考点：字节 9 次，所有统计源均覆盖。考查单调队列、过期下标和摊还复杂度。

写代码前确认：窗口大小 `k` 合法；返回每个完整窗口的最大值。

思路/解法：双端队列保存下标，并保证对应值从队首到队尾单调不增。加入新元素前，先移除队尾所有不

大于它的元素，因为它们更小且更早过期，不可能再成为最大值；再移除队首已离开窗口的下标。队首始终是当前窗口最大值。

```
from collections import deque
from typing import List


class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        queue = deque()
        answer = []
        for right, value in enumerate(nums):
            while queue and nums[queue[-1]] <= value:
                queue.pop()
            queue.append(right)
            left = right - k + 1
            if queue[0] < left:
                queue.popleft()
            if left >= 0:
                answer.append(nums[queue[0]])
        return answer
```

复杂度：时间 `O(n)`，每个下标至多入队、出队各一次；队列空间 `O(k)`。

边界/易错点：队列必须存下标才能判断过期；`k = 1` 时答案等于原数组；相等元素是否弹出不影响正确性，但弹出可保持队列更短。

面试追问：如何同时维护窗口最小值？为什么堆解法通常是 `O(n log n)`？单调队列的摊还 `O(1)` 如何证明？

1.4.10　LC 8 字符串转换整数（atoi）

考频与考点：美团新增。考查顺序解析、状态边界和 32 位整数溢出。

写代码前确认：只跳过前导空格；符号至多一个；遇到首个非数字后立即停止；无有效数字返回 `0`；结果限制在 `[-2^31, 2^31 - 1]`。

思路/解法：依次处理前导空格、可选符号和连续数字。累加新数字前用 `(limit - digit) // 10` 判断是否将溢出，从而不依赖语言的大整数能力。`limit` 根据符号分别取 `2^31 - 1` 或 `2^31`。

```
class Solution:
    def myAtoi(self, s: str) -> int:
        index, n = 0, len(s)
        while index < n and s[index] == " ":
            index += 1

        sign = 1
        if index < n and s[index] in "+-":
            sign = -1 if s[index] == "-" else 1
            index += 1

        limit = 2**31 if sign == -1 else 2**31 - 1
        value = 0
        while index < n and "0" <= s[index] <= "9":
            digit = ord(s[index]) - ord("0")
            if value > (limit - digit) // 10:
                return sign * limit
            value = value * 10 + digit
```

```
            index += 1
        return sign * value
```

复杂度：时间 O(n)；额外空间 O(1)。

边界/易错点：空串、只有符号、前导零、符号后空格、数字后字母、正负边界；不要把中间空格当作可跳过字符。

面试追问：如何用有限状态机表达？若解析任意进制或 64 位整数，需要调整哪些状态和边界？

1.4.11　LC 11 盛最多水的容器

考频与考点：华为新增。考查相向双指针及“移动短板”的正确性证明。

写代码前确认：两条竖线与横轴构成容器，面积为距离乘较短高度；不能倾斜容器。

思路/解法：从最宽区间开始计算面积。面积受较短边限制；若移动较长边，宽度减小而短板不变，面积不可能变大，因此只能移动较短边，寻找更高的短板。每一步都安全排除一批不可能更优的组合。

```
from typing import List


class Solution:
    def maxArea(self, height: List[int]) -> int:
        left, right = 0, len(height) - 1
        answer = 0
        while left < right:
            answer = max(answer, (right - left) * min(height[left], height[right]))
            if height[left] <= height[right]:
                left += 1
            else:
                right -= 1
        return answer
```

复杂度：时间 O(n)；额外空间 O(1)。

边界/易错点：面积宽度是下标差；两边等高时移动任意一边都不会漏掉更优解。

面试追问：如何严格证明移动长边无收益？与 LC 42 的双指针不变量有何区别？

1.4.12　LC 14 最长公共前缀

考频与考点：美团新增。考查字符串逐列比较和输入边界。

写代码前确认：空数组和包含空串时均返回空字符串；比较区分大小写。

思路/解法：把第一个字符串作为基准，逐列检查其余字符串。任一字符串长度不足或当前位置字符不同，当前下标之前就是答案。也可不断缩短候选前缀，但可能反复创建字符串。

```
from typing import List


class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if not strs:
            return ""
        for index, char in enumerate(strs[0]):
```

```
            for position in range(1, len(strs)):
                text = strs[position]
                if index == len(text) or text[index] != char:
                    return strs[0][:index]
        return strs[0]
```

复杂度：时间 `O(S)`，`S` 为实际检查的字符总数上界；除切片结果外空间 `O(1)`。

边界/易错点：不要访问短字符串越界；只有一个字符串时应返回它本身。

面试追问：大量字符串如何用 Trie？字符串分布在多台机器时如何归并公共前缀？

1.4.13　LC 16 最接近的三数之和

考频与考点：美团新增。考查排序双指针与最优差值维护。

写代码前确认：题目保证至少三个数且答案唯一；返回三数之和而非差值或下标。

思路/解法：排序后枚举第一个数，在其右侧使用双指针。每次先根据绝对差更新当前最佳和；和小于目标时只能右移左指针使其增大，和大于目标时只能左移右指针使其减小，恰好相等可立即返回。

```
from typing import List


class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        best = nums[0] + nums[1] + nums[2]
        for i in range(len(nums) - 2):
            left, right = i + 1, len(nums) - 1
            while left < right:
                total = nums[i] + nums[left] + nums[right]
                if abs(total - target) < abs(best - target):
                    best = total
                if total < target:
                    left += 1
                elif total > target:
                    right -= 1
                else:
                    return target
        return best
```

复杂度：时间 `O(n^2)`；排序额外空间取决于实现。

边界/易错点：最佳值应初始化为真实三数之和，不能随意设为 `0`；若题目不保证答案唯一，要约定差值相同时的选择。

面试追问：如何做最接近目标的两数之和？如何通过边界和进行剪枝？与 LC 15 的去重要求有何不同？

1.4.14　LC 43 字符串相乘

考频与考点：美团、腾讯新增。考查竖式乘法、进位与字符串大整数模拟。

写代码前确认：输入只含数字且无多余前导零；不能直接转换为大整数；任一输入为 `"0"` 时返回 `"0"`。

思路/解法：长度为 `m`、`n` 的数相乘结果最多 `m+n` 位。反向枚举两个数字，乘积累加到结果数组位置 `i+j+1`，个位留在该处，进位累加到 `i+j`。由于从低位向高位处理，累加进位可被后续计算继续归并。

```
class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        if num1 == "0" or num2 == "0":
            return "0"
        result = [0] * (len(num1) + len(num2))
        for i in range(len(num1) - 1, -1, -1):
            for j in range(len(num2) - 1, -1, -1):
                product = (ord(num1[i]) - 48) * (ord(num2[j]) - 48)
                total = product + result[i + j + 1]
                result[i + j + 1] = total % 10
                result[i + j] += total // 10
        first = 0
        while first < len(result) - 1 and result[first] == 0:
            first += 1
        return "".join(str(digit) for digit in result[first:])
```

复杂度：时间 `O(mn)`；结果数组空间 `O(m+n)`。

边界/易错点：下标对应关系是 `i+j` 和 `i+j+1`；结果前导零要移除，但不能把零乘积变成空串。

面试追问：如何实现字符串加法？超长数字如何用更大的进制分块？Karatsuba 乘法何时值得使用？

1.4.15　LC 76 最小覆盖子串

考频与考点：腾讯、Hot 100 新增。考查带重复需求的可变窗口和哈希计数。

写代码前确认：覆盖必须满足 `t` 中每个字符的频次而非仅包含字符种类；若不存在返回空串；大小写敏感。

思路/解法：`need` 保存尚缺的字符频次，`missing` 保存尚缺字符总数。右端字符若仍被需要，则 `missing` 减一；无论是否多余都将其需求减一。覆盖完成后持续收缩左端，并在收缩前更新最短答案；移出字符后若其需求变为正数，说明窗口重新缺字符。

```
from collections import Counter


class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if not s or not t:
            return ""
        need = Counter(t)
        missing = len(t)
        left = 0
        best_start, best_length = 0, len(s) + 1

        for right, char in enumerate(s):
            if need[char] > 0:
                missing -= 1
            need[char] -= 1

            while missing == 0:
                length = right - left + 1
                if length < best_length:
                    best_start, best_length = left, length
                left_char = s[left]
                need[left_char] += 1
                if need[left_char] > 0:
```

```
                missing += 1
            left += 1

        if best_length == len(s) + 1:
            return ""
        return s[best_start:best_start + best_length]
```

复杂度：时间 `O(|s| + |t|)`；哈希表空间 `O(| 字符集 |)`。

边界/易错点：`t` 含重复字符、`t` 比 `s` 长、刚好覆盖；多余字符的需求允许为负数。

面试追问：如何返回所有同长度最小窗口？若字符流只能向前读取，如何保存候选窗口？与 LC 438 的固定窗口有何联系？

1.4.16　LC 283 移动零

考频与考点：Hot 100 新增。考查同向双指针、原地稳定移动。

写代码前确认：保持非零元素相对顺序；必须原地操作，不需要返回数组。

思路/解法：`write` 指向下一个非零元素应放的位置。`read` 扫描数组，遇到非零就与 `write` 位置交换并推进 `write`。已扫描区间中，前 `write` 个元素始终是按原顺序收集的全部非零元素，其后均可视为待处理区域。

```
from typing import List


class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        write = 0
        for read in range(len(nums)):
            if nums[read] != 0:
                nums[write], nums[read] = nums[read], nums[write]
                write += 1
```

复杂度：时间 `O(n)`；额外空间 `O(1)`。

边界/易错点：全零、无零、只有一个元素；交换法在 `read == write` 时会自交换但不影响正确性。

面试追问：如何减少写操作次数？如何把满足任意谓词的元素稳定移到末尾？若不要求稳定可怎样做？

1.4.17　LC 438 找到字符串中所有字母异位词

考频与考点：Hot 100 新增。考查固定长度滑动窗口与频次哈希。

写代码前确认：异位词必须长度相同且字符频次一致；返回起始下标；原题字符为小写字母。

思路/解法：窗口长度固定为 `len(p)`。沿用 LC 76 的含义：`need` 表示当前窗口还缺多少字符，`missing` 表示缺失总数。加入右端后，若窗口过长就移出左端并恢复需求；长度刚好且 `missing == 0` 时记录起点。

```
from collections import Counter
from typing import List


class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        if len(p) > len(s):
            return []
```

```
        need = Counter(p)
        missing = len(p)
        left = 0
        answer = []
        for right, char in enumerate(s):
            if need[char] > 0:
                missing -= 1
            need[char] -= 1

            if right - left + 1 > len(p):
                left_char = s[left]
                need[left_char] += 1
                if need[left_char] > 0:
                    missing += 1
                left += 1
            if right - left + 1 == len(p) and missing == 0:
                answer.append(left)
        return answer
```

复杂度：时间 `O(|s| + |p|)`；哈希表空间 `O(| 字符集 |)`。

边界/易错点：`p` 比 `s` 长、重复字符、相邻答案；移出字符时先恢复 `need`，再判断是否重新产生缺失。

面试追问：小写字母场景如何用长度 26 的数组优化？如何判断两个字符串是否异位？与最小覆盖子串的收缩条件有何区别？

1.4.18　LC 468 验证 IP 地址

考频与考点：美团新增。考查严格格式解析、分隔符和字符范围校验。

写代码前确认：IPv4 恰有四段，IPv6 恰有八段；本题不接受 IPv6 压缩写法 `::`、前后分隔符或混合格式。

思路/解法：先按出现的分隔符决定候选类型。IPv4 每段必须是 1 至 3 个十进制数字，除单个 `0` 外不能有前导零，数值不超过 255。IPv6 每段必须是 1 至 4 个十六进制字符。先校验字符再转换，避免 `int` 接受题目不允许的符号或空白。

```
class Solution:
    def validIPAddress(self, queryIP: str) -> str:
        if "." in queryIP and ":" not in queryIP:
            parts = queryIP.split(".")
            if len(parts) != 4:
                return "Neither"
            for part in parts:
                if not 1 <= len(part) <= 3:
                    return "Neither"
                if not all("0" <= char <= "9" for char in part):
                    return "Neither"
                if len(part) > 1 and part[0] == "0":
                    return "Neither"
                if int(part) > 255:
                    return "Neither"
            return "IPv4"

        if ":" in queryIP and "." not in queryIP:
            parts = queryIP.split(":")
            hexadecimal = set("0123456789abcdefABCDEF")
            if len(parts) != 8:
```

```
            return "Neither"
        for part in parts:
            if not 1 <= len(part) <= 4 or any(char not in hexadecimal for char in part):
                return "Neither"
        return "IPv6"

    return "Neither"
```

复杂度：时间 `O(n)`；切分后的辅助空间 `O(n)`，IP 长度有固定上限时可视为常数。

边界/易错点：空段、尾部分隔符、IPv4 前导零、正负号、IPv6 非法字母、同时含点和冒号。

面试追问：如何支持 IPv6 的 `::` 压缩和 IPv4 映射地址？不用 `split` 如何写状态机？

1.5　栈、队列、哈希、贪心与设计

本文件维护第一章第五类题目的题解与面试追问。字节原文把哈希、堆、栈、队列、回溯、贪心和数据结构设计合并在本类，书稿保持这一安排。

1.5.1　字节核心题单

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 146 LRU 缓存	72	08 更新	4/4
2	LC 215 数组中的第 K 个最大元素	64	08 更新	4/4
3	LC 20 有效的括号	36	08 更新	3/4
4	LC 1 两数之和	17	04 基线	3/4
5	LC 46 全排列	16	04 基线	2/4
6	LC 165 比较版本号	16	04 基线	1/4
7	LC 232 用栈实现队列	14	04 基线	0/4
8	LC 415 字符串相加	13	04 基线	1/4
9	LC 22 括号生成	7	04 基线	2/4

1.5.2　其他来源新增题

题目	发现来源
LC 45 跳跃游戏 II	华为、Hot 100 综合榜
LC 49 字母异位词分组	Hot 100 综合榜
LC 51 N 皇后	Hot 100 综合榜
LC 55 跳跃游戏	华为
LC 84 柱状图中最大的矩形	Hot 100 综合榜
LC 93 复原 IP 地址	华为、美团
LC 128 最长连续序列	美团
LC 131 分割回文串	Hot 100 综合榜
LC 136 只出现一次的数字	华为、Hot 100 综合榜

题目	发现来源
LC 155 最小栈	华为、Hot 100 综合榜
LC 224 基本计算器	腾讯
LC 231 2 的幂	腾讯
LC 326 3 的幂	美团
LC 347 前 K 个高频元素	华为、腾讯、Hot 100 综合榜
LC 394 字符串解码	华为、Hot 100 综合榜
LC 406 根据身高重建队列	华为
LC 451 根据字符出现频率排序	华为
LC 470 用 Rand7() 实现 Rand10()	腾讯
LC 703 数据流中的第 K 大元素	华为
LC 739 每日温度	华为
LC 763 划分字母区间	华为、Hot 100 综合榜

1.5.3　题解与追问重点

- LC 1 必须比较暴力枚举、一次遍历哈希表和排序加双指针；若排序，必须处理原下标恢复。
- LC 1 与第四类的 LC 15 组成专题，统一解释补数查找、排序条件和去重策略。
- LC 146 必须手写哈希表加双向链表，禁止用有序字典代替核心实现，并覆盖更新已有键、容量为一和淘汰尾节点。
- LC 215 在本类重点维护最小堆解法；快速选择与排序解法在第六类交叉维护。
- 单调栈题要明确栈内保存值还是下标、单调方向及元素出栈时结算的含义。
- 回溯题统一维护选择、约束、递归和撤销选择，并说明剪枝条件。
- 设计题必须给出操作复杂度表，并解释多个数据结构如何协同保持复杂度承诺。

1.5.4　LC 146 LRU 缓存

考频与考点　字节 72 次，其他统计覆盖 4/4。设计题核心是用哈希表实现 `O(1)` 定位，用双向链表实现 `O(1)` 的访问顺序调整与淘汰。

写代码前确认　`get` 命中和 `put` 更新已有键都算一次访问；容量为正；淘汰的是最久未使用项而非最早插入项。

思路与解法

哈希表保存 `key -> node`。双向链表使用两个哨兵：越靠近头部越新，越靠近尾部越旧。每次命中先摘下结点再移到头部；插入新键后若超容量，删除尾哨兵之前的结点，并同步删除哈希映射。不能用普通单链表，因为已知结点时仍无法 `O(1)` 找到其前驱。

```
class Node:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None


class LRUCache:
```

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.head = Node()  # 最近使用端哨兵
        self.tail = Node()  # 最久未使用端哨兵
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add_first(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def _make_recent(self, node):
        self._remove(node)
        self._add_first(node)

    def get(self, key):
        node = self.cache.get(key)
        if node is None:
            return -1
        self._make_recent(node)
        return node.value

    def put(self, key, value):
        node = self.cache.get(key)
        if node is not None:
            node.value = value
            self._make_recent(node)
            return

        node = Node(key, value)
        self.cache[key] = node
        self._add_first(node)
        if len(self.cache) > self.capacity:
            oldest = self.tail.prev
            self._remove(oldest)
            del self.cache[oldest.key]
```

复杂度　`get`、`put` 均为平均 `O(1)`；存储至多 `capacity` 个有效结点，空间 `O(capacity)`。

边界与易错点　更新已有键不能增加容量；淘汰后必须同时删哈希表；容量为 1 时哨兵可避免头尾特判；结点中必须保存 `key`，才能淘汰时删除映射。

面试追问　若要求并发安全如何加锁或分片？怎样实现带过期时间的 LRU？LFU 需要增加哪些结构？操作复杂度为何能保持 `O(1)`？

1.5.5　LC 215 数组中的第 K 个最大元素

考频与考点　字节 64 次，其他统计覆盖 4/4。本类重点考查固定大小最小堆；快速选择与排序解法在第六类交叉维护。

写代码前确认　第 k 大按排序后的位置计算，重复元素分别占位；输入通常保证 `1 <= k <= n`；是否允许修改原数组会影响快速选择。

思路与解法

维护大小不超过 k 的最小堆，堆中始终保存当前最大的 k 个数，堆顶就是其中最小者，也即全局第 k 大。扫描后续元素时用 `heappushpop` 一次完成压入和弹出。相比整体排序的 `O(n log n)`，当 k 较小时堆更合适；快速选择平均 `O(n)`，但会修改数组且最坏 `O(n^2)`。

```
import heapq


class Solution:
    def findKthLargest(self, nums, k):
        heap = nums[:k]
        heapq.heapify(heap)
        for value in nums[k:]:
            if value > heap[0]:
                heapq.heapreplace(heap, value)
        return heap[0]
```

复杂度　建堆 `O(k)`，其余元素至多各调整一次，时间 `O(n log k)`，额外空间 `O(k)`。

边界与易错点　第 k 大应使用大小为 k 的最小堆，不是最大堆；重复值不能去重；`heapreplace` 仅在新值更大时调用。

面试追问　数据流场景为什么只能优先考虑堆？快速选择如何随机化主元？若求第 k 小如何改？为什么堆顶最终一定是答案？

1.5.6　LC 20 有效的括号

考频与考点　字节 36 次，其他统计覆盖 3/4。考查栈的后进先出特性与字符映射。

写代码前确认　输入只包含三类括号；空串通常有效；括号必须按类型和嵌套顺序匹配。

思路与解法

遇到左括号就把“期望的右括号”压栈，遇到右括号时必须与栈顶一致。保存期望字符比保存左括号少一次反向映射。结束时栈也必须为空。

```
class Solution:
    def isValid(self, s):
        expected = {"(": ")", "[": "]", "{": "}"}
        stack = []
        for ch in s:
            if ch in expected:
                stack.append(expected[ch])
            elif not stack or stack.pop() != ch:
                return False
        return not stack
```

复杂度　时间 `O(n)`，最坏空间 `O(n)`。

边界与易错点　右括号到来时先判断空栈；扫描结束后不能只返回 `True`；只统计数量无法识别 `([)]` 这类错误顺序。

面试追问　若字符串包含普通字符是否忽略？如何返回第一个错误位置？为什么此问题不能仅用三个计数器解决？

1.5.7　LC 1 两数之和

考频与考点　字节 17 次，其他统计覆盖 3/4。与 LC 15 共同构成哈希表和双指针对照专题：哈希表用空间换取补数的常数时间查找，双指针依赖有序性。

写代码前确认　要求返回原数组下标而非数值；同一元素不能使用两次；通常保证恰有一个答案。若输出所有不重复数值对，去重规则会不同。

思路与解法

1. 暴力枚举所有下标对，时间 `O(n^2)`，是复杂度比较基线。
2. 一次遍历哈希表：处理 `nums[i]` 时，只查询此前出现的 `target-nums[i]`，因此不会复用当前元素；命中即返回原下标。这是返回下标问题的首选。
3. 排序加双指针：先保存 **(值，原下标)** 再排序，从两端按和的大小收缩。时间 `O(n log n)`，若只排序数值会丢失原下标；当输入本来有序或题目要求返回数值对时更有优势。

```
class Solution:
    def twoSum(self, nums, target):
        index_of = {}
        for i, value in enumerate(nums):
            need = target - value
            if need in index_of:
                return [index_of[need], i]
            index_of[value] = i

    def twoSumBruteForce(self, nums, target):
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if nums[i] + nums[j] == target:
                    return [i, j]

    def twoSumWithTwoPointers(self, nums, target):
        pairs = sorted((value, i) for i, value in enumerate(nums))
        left, right = 0, len(pairs) - 1
        while left < right:
            total = pairs[left][0] + pairs[right][0]
            if total == target:
                return [pairs[left][1], pairs[right][1]]
            if total < target:
                left += 1
            else:
                right -= 1
```

复杂度　暴力为 `O(n^2)` 时间、`O(1)` 空间；哈希为平均 `O(n)` 时间、`O(n)` 空间；排序双指针为 `O(n log n)` 时间、`O(n)` 空间以保留原下标。

边界与易错点　必须先查补数再写入当前值，否则 `target = 2*x` 时可能复用同一位置；不能用 `if index_of.get(need)` 判断，因为下标 0 为假值；重复数值应由下标区分。

面试追问　数组有序时如何做到 `O(1)` 额外空间？如何返回所有不重复数值对？若数据持续到来怎样实时判断？LC 15 为什么标准答案更偏向排序双指针而不是哈希？

1.5.8　LC 46 全排列

考频与考点　字节 16 次，其他统计覆盖 2/4。考查回溯的选择、约束、递归和撤销选择。

写代码前确认　标准题中元素互不相同；输出所有排列，顺序通常不限；若有重复元素，需要排序并增加同层去重。

思路与解法

路径长度等于 n 时得到一个排列。每层尝试一个尚未使用的下标，加入路径并标记，递归返回后恢复标记和路径。原地交换也能避免 used 数组，但会修改输入顺序。

```
class Solution:
    def permute(self, nums):
        answer, path = [], []
        used = [False] * len(nums)

        def backtrack():
            if len(path) == len(nums):
                answer.append(path[:])
                return
            for i, value in enumerate(nums):
                if used[i]:
                    continue
                used[i] = True
                path.append(value)
                backtrack()
                path.pop()
                used[i] = False

        backtrack()
        return answer
```

复杂度　共有 n! 个结果，每次复制长度 n，时间 O(n*n!)，递归与路径空间 O(n)，不计输出。

边界与易错点　加入答案时必须复制 path；标记的是下标而非值；递归返回后撤销顺序要完整。

面试追问　含重复元素时如何剪枝？怎样用交换法实现？如何求字典序的下一个排列而不枚举全部结果？

1.5.9　LC 165 比较版本号

考频与考点　字节 16 次，其他统计覆盖 1/4。考查字符串分段、前导零和不同段数的对齐。

写代码前确认　修订号只含数字且可能有前导零；缺失段等价于 0；返回 -1/0/1。超长修订号在不支持大整数的语言中要避免直接转换。

思路与解法

按点分割后逐段比较，循环次数取两边段数最大值，缺失段补 0。Python 整数不限固定宽度，可安全使用 int。若追问常数额外空间，可用两个指针逐段解析。

```
class Solution:
    def compareVersion(self, version1, version2):
        parts1 = version1.split(".")
        parts2 = version2.split(".")
        for i in range(max(len(parts1), len(parts2))):
            x = int(parts1[i]) if i < len(parts1) else 0
            y = int(parts2[i]) if i < len(parts2) else 0
            if x < y:
                return -1
            if x > y:
                return 1
```

```
        return 0
```

复杂度　设输入总长度为 n，时间 O(n)，分割数组空间 O(n)；双指针解析可降至 O(1) 额外空间。

边界与易错点　不能按字符串字典序比较，如 "10" > "2" 的数值关系与字典序相反；尾部零段不影响结果；不要假设两边段数相同。

面试追问　不用 split 如何解析？修订号可能有百万位数字时如何比较？语义化版本中的预发布标识为何不能沿用此规则？

1.5.10　LC 232 用栈实现队列

考频与考点　字节 14 次。考查两个后进先出结构如何组合成先进先出结构，以及均摊复杂度。

写代码前确认　允许使用两个栈；调用 pop、peek 时队列通常非空；需要说明单次最坏复杂度与均摊复杂度的区别。

思路与解法

in_stack 只负责接收新元素，out_stack 只负责提供队首。当 out_stack 为空时，才把 in_stack 全部倒入，使顺序反转。每个元素最多入、出两个栈各一次，因此操作均摊 O(1)。

```
class MyQueue:
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def push(self, x):
        self.in_stack.append(x)

    def _move_if_needed(self):
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())

    def pop(self):
        self._move_if_needed()
        return self.out_stack.pop()

    def peek(self):
        self._move_if_needed()
        return self.out_stack[-1]

    def empty(self):
        return not self.in_stack and not self.out_stack
```

复杂度　push 为 O(1)；pop、peek 单次最坏 O(n)、均摊 O(1)；空间 O(n)。

边界与易错点　out_stack 非空时不能搬运，否则会破坏旧元素顺序；判空要同时检查两个栈；不要在每次 push 时反复整体搬运。

面试追问　如何用队列实现栈？怎样用势能法解释均摊 O(1)？如果要求每次操作严格最坏 O(1)，需要怎样渐进搬运？

1.5.11　LC 415 字符串相加

考频与考点　字节 13 次，其他统计覆盖 1/4。考查从低位到高位模拟加法和字符数字转换。

写代码前确认　输入是非负整数字符串，不能使用大整数库或直接转整数；结果不能有多余前导零。

思路与解法

两个指针从字符串末尾向前移动，缺失位视为 0；每轮用和的个位写入结果、十位作为进位。结果逆序产生，最后统一反转。该模板可直接推广到任意进制。

```
class Solution:
    def addStrings(self, num1, num2):
        i, j = len(num1) - 1, len(num2) - 1
        carry = 0
        digits = []
        while i >= 0 or j >= 0 or carry:
            x = ord(num1[i]) - ord("0") if i >= 0 else 0
            y = ord(num2[j]) - ord("0") if j >= 0 else 0
            carry, digit = divmod(x + y + carry, 10)
            digits.append(chr(ord("0") + digit))
            i -= 1
            j -= 1
        return "".join(reversed(digits))
```

复杂度　时间 `O(max(m,n))`，结果数组空间 `O(max(m,n))`。

边界与易错点　循环条件必须包含最终进位；指针越界时该位为 0；不可在循环头提前丢弃较长字符串剩余部分。

面试追问　如何实现字符串减法并处理符号？如何支持任意进制？链表形式的两数相加与本题的遍历方向有何关系？

1.5.12　LC 22 括号生成

考频与考点　字节 7 次，其他统计覆盖 2/4。考查生成型回溯及用约束保证只搜索合法前缀。

写代码前确认　生成 `n` 对圆括号的全部合法组合；输出顺序不限；`n = 0` 时数学上结果通常包含空串。

思路与解法

状态记录已使用的左右括号数。只要左括号未满就可添加；只有 `right < left` 时才能添加右括号，从而保证任意前缀中右括号不多于左括号。相比生成全部 `2^(2n)` 个串再校验，剪枝直接排除非法前缀。

```
class Solution:
    def generateParenthesis(self, n):
        answer, path = [], []

        def backtrack(left, right):
            if len(path) == 2 * n:
                answer.append("".join(path))
                return
            if left < n:
                path.append("(")
                backtrack(left + 1, right)
                path.pop()
            if right < left:
                path.append(")")
                backtrack(left, right + 1)
                path.pop()

        backtrack(0, 0)
```

```
        return answer
```

复杂度　合法结果数为第 n 个卡特兰数 C_n，生成和复制结果的时间为 O(n*C_n)；递归路径空间 O(n)，不计输出。

边界与易错点　右括号条件是 right < left；加入答案时要复制或拼接当前路径；不能把 LC 20 的事后校验当作主要剪枝。

面试追问　如何计算合法组合数量？怎样扩展到多种括号且要求正确嵌套？如何按字典序生成第 k 个合法串？

1.5.13　LC 45 跳跃游戏 II

考频与考点　新增来源为华为、Hot 100 综合榜。考查贪心分层和最少步数证明。

写代码前确认　通常保证能到达最后位置；nums[i] 是从位置 i 最远可跳距离；长度为 1 时答案为 0。

思路与解法

把一次跳跃能够到达的下标看作 BFS 的一层。扫描当前层 [0, current_end] 时，持续更新下一层最远边界 farthest；扫描到 current_end 才增加一次跳跃并扩展边界。无需真的保存图或队列。

```
class Solution:
    def jump(self, nums):
        jumps = 0
        current_end = 0
        farthest = 0
        for i in range(len(nums) - 1):
            farthest = max(farthest, i + nums[i])
            if i == current_end:
                jumps += 1
                current_end = farthest
        return jumps
```

复杂度　时间 O(n)，额外空间 O(1)；动态规划基线为 O(n^2) 时间。

边界与易错点　循环只到倒数第二个位置，避免到达终点后多计一步；不能在每次更新最远距离时都增加步数；若输入不保证可达，应在扩层时检查 farthest == current_end。

面试追问　为什么局部选择最远边界不会错过更少步数？如何输出一条最短跳跃路径？与 LC 55 的状态含义有何不同？

1.5.14　LC 49 字母异位词分组

考频与考点　新增来源为 Hot 100 综合榜。考查为等价类设计稳定哈希键。

写代码前确认　字符串通常只含小写英文字母；输出分组和组内顺序不限；空字符串的签名应合法。

思路与解法

异位词具有相同字符计数。对每个字符串构造长度 26 的计数元组作为哈希键，把原串加入对应列表。另一种写法用排序后字符串作键，代码更短，但每个长度为 k 的字符串需 O(k log k)。

```
from collections import defaultdict


class Solution:
```

```
    def groupAnagrams(self, strs):
        groups = defaultdict(list)
        for word in strs:
            counts = [0] * 26
            for ch in word:
                counts[ord(ch) - ord("a")] += 1
            groups[tuple(counts)].append(word)
        return list(groups.values())
```

复杂度　设所有字符总数为 N，固定字符集计数法时间 O(N)，哈希表及输出空间 O(N)。

边界与易错点　列表不可哈希，必须转换为元组；不能用字符集合做键，因为它会丢失次数；若字符集不限于小写字母，应改用排序键或规范化计数字典。

面试追问　Unicode 字符如何设计键？海量字符串无法全部放内存时怎样外排分组？计数键和排序键分别适合什么字符集？

1.5.15　LC 51 N 皇后

考频与考点　新增来源为 Hot 100 综合榜。考查回溯、列与两类对角线约束，以及剪枝状态的维护。

写代码前确认　每行恰放一个皇后；返回所有棋盘；主对角线可用 row−col 标识，副对角线可用 row+col 标识。

思路与解法

按行递归，每行尝试尚未被列集合、主对角线集合、副对角线集合占用的位置。选择后同时加入三个集合，递归返回时完整撤销。按行放置天然满足同行不冲突。

```
class Solution:
    def solveNQueens(self, n):
        answer = []
        board = [["."] * n for _ in range(n)]
        cols, diag1, diag2 = set(), set(), set()

        def backtrack(row):
            if row == n:
                answer.append(["".join(line) for line in board])
                return
            for col in range(n):
                d1, d2 = row - col, row + col
                if col in cols or d1 in diag1 or d2 in diag2:
                    continue
                cols.add(col)
                diag1.add(d1)
                diag2.add(d2)
                board[row][col] = "Q"
                backtrack(row + 1)
                board[row][col] = "."
                cols.remove(col)
                diag1.remove(d1)
                diag2.remove(d2)

        backtrack(0)
        return answer
```

复杂度　搜索树结点数的常用上界为 `O(n!)`，当前实现会在每个非叶结点扫描 `n` 列；设方案数为 `S`，每个棋盘快照还需 `O(n^2)` 时间，因此总时间上界写作 `O(n*n! + S*n^2)` 更完整。三个约束集合和递归路径为 `O(n)`，工作棋盘为 `O(n^2)`；输出空间为 `O(S*n^2)`。

边界与易错点　对角线标识不能写反后混用；三个约束都要撤销；保存答案时要构造字符串快照，不能保存同一个可变棋盘引用。

面试追问　只统计方案数怎样省去棋盘？如何用位运算表示三类约束？为什么逐行搜索已隐含“每行一个”的约束？

1.5.16　LC 55 跳跃游戏

考频与考点　新增来源为华为。考查可达性贪心和不变量。

写代码前确认　判断能否到达最后下标，不要求最少步数；元素非负；起点固定为下标 0。

思路与解法

维护从已确认可达的位置能够覆盖的最远下标 `farthest`。扫描到 `i` 时，若 `i > farthest`，说明中间出现不可跨越的断点；否则用 `i + nums[i]` 扩展覆盖。动态规划可令 `dp[i]` 表示是否可达，但空间和转移都没有必要。

```
class Solution:
    def canJump(self, nums):
        farthest = 0
        for i, step in enumerate(nums):
            if i > farthest:
                return False
            farthest = max(farthest, i + step)
            if farthest >= len(nums) - 1:
                return True
        return True
```

复杂度　时间 `O(n)`，额外空间 `O(1)`。

边界与易错点　只有可达位置才能扩展最远边界；中间的 0 不一定导致失败；长度为 1 时无需跳跃即可到达。

面试追问　从终点反向贪心如何写？怎样返回一条可行路径？为何 LC 45 需要“当前层边界”而本题只需最远边界？

1.5.17　LC 84 柱状图中最大的矩形

考频与考点　新增来源为 Hot 100 综合榜。考查单调栈中保存下标、单调方向，以及元素出栈时结算面积。

写代码前确认　柱宽均为 1，高度非负；矩形必须覆盖连续柱；相等高度应采用一致的入栈或弹栈规则。

思路与解法

维护高度单调不减的下标栈，并在两端使用高度 0 的哨兵。遇到更矮柱时，弹出的下标 `mid` 对应高度已确定右侧第一个更矮位置 `i`，弹栈后的新栈顶是左侧第一个更矮位置，因此宽度为 `i-stack[-1]-1`。每根柱只入栈、出栈一次。

```
class Solution:
    def largestRectangleArea(self, heights):
        values = [0] + heights + [0]
        stack = []
        answer = 0
        for i, height in enumerate(values):
```

```
            while stack and values[stack[-1]] > height:
                mid = stack.pop()
                width = i - stack[-1] - 1
                answer = max(answer, values[mid] * width)
            stack.append(i)
        return answer
```

复杂度　时间 O(n)，栈空间 O(n)；逐柱向两侧扩展的基线最坏为 O(n^2)。

边界与易错点　栈内保存下标而非高度；面积在出栈时结算；左边界是弹栈后的栈顶；末尾哨兵用于清空仍未结算的柱。

面试追问　为何可以在出栈时确定最大宽度？相等高度用 > 或 >= 有何差别？如何扩展到最大矩形矩阵？

1.5.18　LC 93 复原 IP 地址

考频与考点　新增来源为华为、美团。考查定长分段回溯与前导零剪枝。

写代码前确认　必须切成恰好 4 段，每段范围 0..255；除单独的 0 外不能有前导零；输入只含数字。

思路与解法

递归状态为当前位置和已选段。每段只尝试长度 1 到 3；选段前检查剩余字符数是否能填满剩余段，即是否位于 [remaining_parts, 3*remaining_parts]。遇到前导零时只允许长度 1。

```
class Solution:
    def restoreIpAddresses(self, s):
        answer, parts = [], []

        def backtrack(start):
            remaining_parts = 4 - len(parts)
            remaining_chars = len(s) - start
            if remaining_chars < remaining_parts or remaining_chars > 3 * remaining_parts:
                return
            if len(parts) == 4:
                if start == len(s):
                    answer.append(".".join(parts))
                return

            for end in range(start + 1, min(start + 3, len(s)) + 1):
                segment = s[start:end]
                if len(segment) > 1 and segment[0] == "0":
                    break
                if int(segment) > 255:
                    break
                parts.append(segment)
                backtrack(end)
                parts.pop()

        backtrack(0)
        return answer
```

复杂度　IP 固定 4 段、每段至多 3 种长度，搜索规模有常数上界；按一般输入长度记，校验与构造可视为 O(n)，递归空间 O(1)（固定深度）。

边界与易错点　"0" 合法而 "00" 不合法；段数够 4 后必须同时耗尽字符串；超过 255 后更长前缀也必然无效，可直接停止。

面试追问　若分成 k 段且每段长度、范围可配置，复杂度如何表示？怎样只判断是否存在合法切分？IPv6 的规则为何不能直接复用？

1.5.19　LC 128 最长连续序列

考频与考点　新增来源为美团。考查哈希集合如何把无序数组的连续性搜索优化到线性时间。

写代码前确认　连续指整数值相差 1，与原数组位置无关；重复元素不增加长度；要求期望 `O(n)` 时间。

思路与解法

把所有值放入集合。只从不存在前驱 `x-1` 的值开始向右扩展，因为它必然是一段连续序列的起点；非起点直接跳过。每个不同值只会在所属序列的扩展中被访问一次。排序法更直观，但需要 `O(n log n)`。

```
class Solution:
    def longestConsecutive(self, nums):
        values = set(nums)
        answer = 0
        for value in values:
            if value - 1 in values:
                continue
            end = value
            while end + 1 in values:
                end += 1
            answer = max(answer, end - value + 1)
        return answer
```

复杂度　期望时间 `O(n)`，集合空间 `O(n)`。

边界与易错点　必须只从序列起点扩展，否则连续长段会被反复扫描到 `O(n^2)`；遍历集合可自然去重；空数组答案为 0。

面试追问　如何返回最长序列本身？并查集能否解决，代价是什么？哈希表极端冲突时复杂度如何变化？

1.5.20　LC 131 分割回文串

考频与考点　新增来源为 Hot 100 综合榜。考查字符串切分回溯、回文判断和预处理优化。

写代码前确认　需要返回所有切分方案，每个片段都必须是非空回文串；输出顺序不限；空串的约定通常是一种空切分。

思路与解法

从 `start` 开始枚举当前片段终点，只有片段为回文时才选择并递归。直接双指针判断每个片段会重复扫描；先用动态规划预处理 `palindrome[i][j]`，其中两端字符相同且内部长度不超过 1 或内部也是回文，即可将每次判断降为 `O(1)`。

```
class Solution:
    def partition(self, s):
        n = len(s)
        palindrome = [[False] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                palindrome[i][j] = (
                    s[i] == s[j]
                    and (j - i <= 1 or palindrome[i + 1][j - 1])
                )
```

```
        answer, path = [], []

        def backtrack(start):
            if start == n:
                answer.append(path[:])
                return
            for end in range(start, n):
                if not palindrome[start][end]:
                    continue
                path.append(s[start:end + 1])
                backtrack(end + 1)
                path.pop()

        backtrack(0)
        return answer
```

复杂度　预处理 `O(n^2)` 时间和空间；切分方案最坏为 `2^(n-1)` 个，包含结果复制时总时间为 `O(n*2^n)` 量级。

边界与易错点　终点下标与切片右端点相差 1；保存方案时必须复制路径；DP 要按 `i` 递减计算，确保内部状态已知。

面试追问　如何求最少切割次数？不用 `O(n^2)` 预处理怎样写？能否只统计方案数而不枚举结果？

1.5.21　LC 136 只出现一次的数字

考频与考点　新增来源为华为、Hot 100 综合榜。考查异或的交换律、自反性和零元性质。

写代码前确认　除一个元素出现一次外，其余元素恰好出现两次；要求线性时间和常数额外空间。

思路与解法

将所有数异或。由于 `x ^ x = 0`、`x ^ 0 = x`，成对元素与遍历顺序无关地抵消，最终只剩单独元素。哈希计数也能做，但需要 `O(n)` 空间；求和公式可能溢出且仍依赖去重集合。

```
class Solution:
    def singleNumber(self, nums):
        answer = 0
        for value in nums:
            answer ^= value
        return answer
```

复杂度　时间 `O(n)`，额外空间 `O(1)`。

边界与易错点　异或适用于整数位模式，包括负数；不能用逻辑异或替代按位异或；前提必须是其他元素都恰好出现两次。

面试追问　其余元素出现三次时如何按位计数？有两个元素各出现一次时如何用最低有效差异位分组？为什么异或顺序不影响结果？

1.5.22　LC 155 最小栈

考频与考点　新增来源为华为、Hot 100 综合榜。考查用辅助状态为设计题维持所有操作 `O(1)`。

写代码前确认　`push`、`pop`、`top`、`getMin` 都要 `O(1)`；查询和弹出通常只在非空栈调用；重复最小值必须正确处理。

思路与解法

每次压入二元组（**当前值，压入后栈内最小值**）。这样弹出时，对应历史最小值会同步恢复。也可维护独立最小栈，但新值等于当前最小时也必须压入，否则弹出一个重复最小值后状态会错误。

```
class MinStack:
    def __init__(self):
        self.stack = []

    def push(self, val):
        current_min = val if not self.stack else min(val, self.stack[-1][1])
        self.stack.append((val, current_min))

    def pop(self):
        self.stack.pop()

    def top(self):
        return self.stack[-1][0]

    def getMin(self):
        return self.stack[-1][1]
```

复杂度　四个操作均为 `O(1)` 时间，存储 `n` 个元素需要 `O(n)` 空间。

边界与易错点　重复最小值不能只记录一次；`pop` 后不需要重新扫描；若用差值编码压缩空间，要注意固定宽度整数溢出。

面试追问　如何同时支持 `getMax`？只用一个数值栈如何通过差值编码最小值？怎样设计支持 `getMin` 的队列？

1.5.23　LC 224 基本计算器

考频与考点　新增来源为腾讯。考查表达式扫描、括号上下文和一元正负号。

写代码前确认　本题只含非负整数、`+`、`-`、括号和空格，不含乘除；表达式合法；负号既可能是二元减法，也可能是一元负号。

思路与解法

扫描时累积当前多位数 `number`，`sign` 表示它前面的符号。遇到运算符先把 `sign*number` 加入当前层结果。遇到左括号，把括号外的结果和符号压栈，并从新层开始；遇到右括号，先结算当前数字，再将括号内结果乘外层符号并加回外层结果。该状态机自然支持 `-(...)`。

```
class Solution:
    def calculate(self, s):
        result = 0
        number = 0
        sign = 1
        stack = []

        for ch in s:
            if ch.isdigit():
                number = number * 10 + int(ch)
            elif ch in "+-":
                result += sign * number
                number = 0
                sign = 1 if ch == "+" else -1
            elif ch == "(":
                stack.append(result)
                stack.append(sign)
```

```
                result, sign = 0, 1
            elif ch == ")":
                result += sign * number
                number = 0
                outer_sign = stack.pop()
                outer_result = stack.pop()
                result = outer_result + outer_sign * result

        return result + sign * number
```

复杂度　时间 `O(n)`，括号栈空间 `O(d)`，其中 `d` 为嵌套深度。

边界与易错点　遇到运算符和右括号都要先结算已有数字；空格应忽略；多位数字不能逐字符立即结算；压栈和出栈的顺序必须对应。

面试追问　加入乘除后如何处理优先级？怎样用逆波兰表达式求值？递归下降解析器的文法和状态应如何设计？

1.5.24　LC 231 2 的幂

考频与考点　新增来源为腾讯。考查二进制最低位性质。

写代码前确认　只有正整数可能是 2 的幂；0 和负数返回假；语言中的整数位宽和符号规则可能影响位运算解释。

思路与解法

正的 2 的幂二进制中恰有一个 1。`n-1` 会把该 1 变为 0，并把其低位全部变为 1，因此 `n & (n-1) == 0`。循环除以 2 也可做，但位运算直接表达核心性质。

```
class Solution:
    def isPowerOfTwo(self, n):
        return n > 0 and (n & (n - 1)) == 0
```

复杂度　固定宽度整数下时间、空间均为 `O(1)`。

边界与易错点　必须先限制 `n > 0`，否则 0 会误判；位运算括号应写清；不要用浮点对数判断，精度可能造成误差。

面试追问　如何统计二进制中 1 的个数？如何取得最低位的 1？判断 4 的幂还需增加什么条件？

1.5.25　LC 326 3 的幂

考频与考点　新增来源为美团。考查整除循环，以及不存在二进制单比特捷径时的替代思路。

写代码前确认　只有正整数可能是 3 的幂；输入位宽是否固定会影响“最大 3 的幂整除法”。

思路与解法

不断除以 3，若过程中出现不能整除则返回假，最终等于 1 即为 3 的幂。在 32 位有符号整数范围内，也可用最大 3 的幂 `3^19` 判断 `1162261467 % n == 0`，但该常量依赖输入范围，不如循环通用。

```
class Solution:
    def isPowerOfThree(self, n):
        if n <= 0:
            return False
        while n % 3 == 0:
```

```
            n //= 3
        return n == 1
```

复杂度　时间 `O(log_3 n)`，额外空间 `O(1)`；固定 32 位最大幂整除法为 `O(1)`。

边界与易错点　`n = 1` 是 `3^0`，应返回真；必须使用整数除法；浮点 `log` 方案有精度风险。

面试追问　为何最大 3 的幂能被所有较小 3 的幂整除？该技巧为何要求底数为质数？如何统一判断任意正整数是否为 `k` 的幂？

1.5.26　LC 347 前 K 个高频元素

考频与考点　新增来源为华为、腾讯、Hot 100 综合榜。考查计数哈希、固定大小堆和桶排序。

写代码前确认　返回元素而非频次；答案顺序通常不限；保证恰有足够元素且第 `k` 位不存在需要消解的歧义。

思路与解法

先用哈希表统计频次，再维护大小为 `k` 的最小堆（**频次，元素**）；堆顶是当前候选中频次最低者。若追求线性时间，可建立下标为频次的桶，从高频到低频收集 `k` 个元素，代价是 `O(n)` 额外桶空间。

```
from collections import Counter
import heapq


class Solution:
    def topKFrequent(self, nums, k):
        counts = Counter(nums)
        heap = []
        for value, frequency in counts.items():
            if len(heap) < k:
                heapq.heappush(heap, (frequency, value))
            elif frequency > heap[0][0]:
                heapq.heapreplace(heap, (frequency, value))
        return [value for _, value in heap]
```

复杂度　设不同元素数为 `m`，堆解法时间 `O(n + m log k)`、空间 `O(m+k)`（计数表占 `O(m)`）；桶排序时间 `O(n)`、空间 `O(n)`。

边界与易错点　堆按频次排序，不能直接对元素建堆；结果无需按频次有序；与 LC 215 不同，本题先聚合频次再选前 `k`。

面试追问　如何按频次降序输出？数据流中如何近似统计 Top K？桶排序为什么最多只需 `n+1` 个桶？

1.5.27　LC 394 字符串解码

考频与考点　新增来源为华为、Hot 100 综合榜。考查栈处理嵌套结构、多位重复次数和局部状态恢复。

写代码前确认　编码格式合法，重复次数位于方括号之前且可能为多位数；普通字母可连续出现；需要返回完全展开的字符串。

思路与解法

扫描时维护当前层的重复次数 `number` 和已解码字符列表 `current`。遇到 `[`，把外层列表和次数一起压栈，开始新的内层；遇到 `]`，弹出外层状态并拼接 `current` 的重复结果。也可递归解析，但显式栈更容易展示状态。

```
class Solution:
    def decodeString(self, s):
        stack = []
        current = []
        number = 0

        for ch in s:
            if ch.isdigit():
                number = number * 10 + int(ch)
            elif ch == "[":
                stack.append((current, number))
                current = []
                number = 0
            elif ch == "]":
                previous, repeat = stack.pop()
                current = previous + current * repeat
            else:
                current.append(ch)
        return "".join(current)
```

复杂度　设编码串长度为 n、最终输出长度为 L、嵌套深度为 d。列表拼接的时间等于各次] 产生的中间列表长度之和，最坏为 O(n*L)；例如多层重复次数为 1 的嵌套会反复复制同一段内容。最终结果占 O(L) 空间，栈另占 O(d) 个状态，处理中间列表的峰值空间为 O(n+L)。

边界与易错点　重复次数可能超过一位；进入新层后要清零 number；栈中必须同时保存外层内容和次数；复杂度不能只按编码串长度计算。

面试追问　如何写递归下降版本？若展开结果可能极大，怎样流式输出或只求长度？如何检测非法括号和缺失次数？

1.5.28　LC 406 根据身高重建队列

考频与考点　新增来源为华为。考查排序规则设计和贪心插入不变量。

写代码前确认　每个人表示为 [height, k]，k 是排在其前面且身高不低于他的人数；输入保证存在合法答案。

思路与解法

先按身高降序、同身高按 k 升序排序。依次处理时，结果中已有的人都不矮于当前人，因此把当前人插入下标 k，其前面恰有 k 个不矮者。后续插入的更矮者不会破坏这一条件。

```
class Solution:
    def reconstructQueue(self, people):
        people.sort(key=lambda person: (-person[0], person[1]))
        queue = []
        for person in people:
            queue.insert(person[1], person)
        return queue
```

复杂度　排序 O(n log n)，数组中间插入总计 O(n^2)，空间 O(n)（返回结果）。使用支持按秩插入的数据结构可进一步优化插入。

边界与易错点　同身高必须按 k 升序；若改为身高升序，需要从空位置角度设计完全不同的策略；不要把 k 理解为所有更高者数量。

面试追问　为什么后插入的矮个子不影响已处理的人？如何用树状数组在空位上重建？若同身高排序规则反过来会发生什么？

1.5.29　LC 451 根据字符出现频率排序

考频与考点　新增来源为华为。考查频次哈希与按频次桶排序。

写代码前确认　字符大小写敏感；相同频次的相对顺序通常不限；输出必须包含原字符串的全部字符。

思路与解法

哈希统计每个字符次数。频次最大不超过字符串长度 `n`，因此可建立 `n+1` 个桶，把字符放入对应频次下标，再从高到低输出 **字符 * 次数**。按哈希项排序也可做，时间为 `O(m log m)`，其中 `m` 是不同字符数。

```
from collections import Counter


class Solution:
    def frequencySort(self, s):
        counts = Counter(s)
        buckets = [[] for _ in range(len(s) + 1)]
        for ch, frequency in counts.items():
            buckets[frequency].append(ch)

        answer = []
        for frequency in range(len(s), 0, -1):
            for ch in buckets[frequency]:
                answer.append(ch * frequency)
        return "".join(answer)
```

复杂度　桶排序时间 `O(n)`，空间 `O(n)`；比较排序方案时间 `O(n + m log m)`。

边界与易错点　桶下标是频次而不是字符码；相同频次无需额外稳定排序，除非题目明确要求；空串应返回空串。

面试追问　若要求同频字符按字典序排列如何改？字符种类极多但字符串很短时桶和堆如何选择？怎样原地处理可变字符数组？

1.5.30　LC 470 用 Rand7() 实现 Rand10()

考频与考点　新增来源为腾讯。考查等概率样本空间构造和拒绝采样。

写代码前确认　`rand7()` 独立且均匀返回 `1..7`；目标是每个 `1..10` 概率相同，不能用取模直接处理 7 或 49 个非整倍数状态。

思路与解法

两次调用可等概率生成 `1..49`：`(rand7()-1)*7 + rand7()`。接受前 40 个状态并映射为 `1..10`，其余 9 个状态拒绝后重采样。因为 40 是 10 的整数倍，每个结果恰对应 4 个等概率状态。

```
class Solution:
    def rand10(self):
        while True:
            sample = (rand7() - 1) * 7 + rand7()
            if sample <= 40:
                return 1 + (sample - 1) % 10
```

复杂度　每轮接受概率为 `40/49`，期望轮数为 `49/40`，因此期望时间 `O(1)`、空间 `O(1)`；最坏调用次数没有有限上界。

边界与易错点　`rand7()+rand7()` 不是均匀分布；直接对 `1..49` 取模会产生偏差；应先减 1 映射到从 0 开始的等概率编号。

面试追问　如何回收被拒绝的 9 个状态以减少期望调用次数？怎样用 `RandM` 构造 `RandN`？如何证明拒绝后重新采样不会引入偏差？

1.5.31　LC 703 数据流中的第 K 大元素

考频与考点　新增来源为华为。考查固定大小最小堆在在线数据中的应用。

写代码前确认　初始化数组也属于数据流已有元素；每次 `add` 后至少有 `k` 个元素可供定义第 `k` 大；重复元素分别占位。

思路与解法

维护大小不超过 `k` 的最小堆，保存目前最大的 `k` 个元素。加入新值后先压堆，若大小超过 `k` 就弹出最小值，堆顶始终为第 `k` 大。初始化时可先建全部元素的堆再缩小，也可逐个调用同一逻辑。

```
import heapq


class KthLargest:
    def __init__(self, k, nums):
        self.k = k
        self.heap = []
        for value in nums:
            self._push(value)

    def _push(self, value):
        heapq.heappush(self.heap, value)
        if len(self.heap) > self.k:
            heapq.heappop(self.heap)

    def add(self, val):
        self._push(val)
        return self.heap[0]
```

复杂度　初始化 `n` 个数为 `O(n log k)`，每次 `add` 为 `O(log k)`，空间 `O(k)`；批量初始化也可优化为 `O(n + (n-k) log n)` 等实现。

边界与易错点　使用最小堆而非最大堆；堆只保留 `k` 个候选；负数和重复值不需特殊处理；不能每次重新排序全部历史数据。

面试追问　与 LC 215 的离线快速选择相比，为何数据流更适合堆？若 `k` 动态变化如何设计？如何支持删除任意历史元素？

1.5.32　LC 739 每日温度

考频与考点　新增来源为华为。考查单调栈保存下标，以及元素出栈时确定右侧第一个更大值。

写代码前确认　要返回等待天数而不是更高温度；必须严格更高，相等温度不能结算；没有更高温度的位置保持 0。

思路与解法

维护尚未找到更高温度的下标栈，栈内对应温度单调不增。当前温度高于栈顶温度时，当前下标就是栈顶位置右侧第一个更高温度，弹出并写入距离；随后把当前下标压栈。

```
class Solution:
    def dailyTemperatures(self, temperatures):
        answer = [0] * len(temperatures)
        stack = []
        for i, temperature in enumerate(temperatures):
            while stack and temperatures[stack[-1]] < temperature:
                previous = stack.pop()
                answer[previous] = i - previous
            stack.append(i)
        return answer
```

复杂度　每个下标至多入栈、出栈一次，时间 O(n)，栈空间 O(n)。

边界与易错点　栈中保存下标才能计算天数；相等温度不能弹出；未出栈位置答案保持初始化的 0。

面试追问　如何求右侧第一个更大元素的值或下标？循环数组版本如何处理？为什么总时间不是嵌套循环表面上的 O(n^2)？

1.5.33　LC 763 划分字母区间

考频与考点　新增来源为华为、Hot 100 综合榜。考查最后出现位置哈希与区间贪心。

写代码前确认　每个字母最多出现在一个片段中；目标是片段数尽可能多；返回各片段长度而非字符串本身。

思路与解法

先记录每个字符最后出现位置。扫描一个片段时，当前字符的最后位置可能把片段右边界扩展得更远；当下标到达动态右边界，说明片段中出现过的所有字符都不会在后面再出现，此时立即切分可得到最多片段。

```
class Solution:
    def partitionLabels(self, s):
        last = {ch: i for i, ch in enumerate(s)}
        answer = []
        start = end = 0
        for i, ch in enumerate(s):
            end = max(end, last[ch])
            if i == end:
                answer.append(end - start + 1)
                start = i + 1
        return answer
```

复杂度　时间 O(n)；哈希表空间为 O(u)，u 是不同字符数，小写字母条件下可视为 O(1)。

边界与易错点　边界要随片段内新字符持续扩展；长度为 end-start+1；在尚未到达右边界时切分会让某字符跨片段。

面试追问　如何从区间合并角度解释本题？若要求每个字符最多出现在两个片段中如何变化？为什么在首次可切位置立即切分具有最优性？

1.6　二分查找、排序与数组

本文件维护第一章第六类题目的题解与面试追问。数组题优先明确循环不变量，二分题必须固定区间定义，排序题需要说明稳定性、最坏复杂度和原地性。

1.6.1　字节核心题单

顺序	题目	字节频次	数据版本	其他统计覆盖
1	LC 215 数组中的第 K 个最大元素	64	08 更新	4/4
2	LC 56 合并区间	18	04 基线	3/4
3	LC 33 搜索旋转排序数组	15	04 基线	3/4
4	LC 34 在排序数组中查找元素的第一个和最后一个位置	12	04 基线	2/4
5	LC 912 排序数组	11	04 基线	1/4
6	LC 4 寻找两个正序数组的中位数	10	04 基线	1/4
7	LC 75 颜色分类	8	04 基线	2/4
8	LC 902 最大为 N 的数字组合	未单列	08 新增题	0/4

1.6.2　其他来源新增题

题目	发现来源
LC 35 搜索插入位置	华为、Hot 100 综合榜
LC 54 螺旋矩阵	华为、腾讯、Hot 100 综合榜
LC 73 矩阵置零	Hot 100 综合榜
LC 74 搜索二维矩阵	Hot 100 综合榜
LC 153 寻找旋转排序数组中的最小值	华为、腾讯、Hot 100 综合榜
LC 169 多数元素	华为
LC 238 除自身以外数组的乘积	华为、Hot 100 综合榜
LC 240 搜索二维矩阵 II	腾讯、Hot 100 综合榜

题目	发现来源
LC 287 寻找重复数	华为、腾讯、Hot 100 综合榜
LC 442 数组中重复的数据	腾讯
LC 704 二分查找	华为、腾讯

1.6.3　题解与追问重点

- 二分题统一声明使用左闭右闭还是左闭右开区间，循环条件、边界更新和返回位置必须保持一致。
- LC 33、LC 153 和 LC 704 组成二分模板专题，比较普通有序数组与旋转数组中仍然成立的单调性。
- LC 215 在本类重点维护排序、快速选择和分区写法；最小堆解法在第五类交叉维护。
- LC 912 至少覆盖快速排序与归并排序，并讨论随机基准、最坏情况、稳定性和额外空间。
- 矩阵题要先写清行列边界和遍历不变量，避免靠补丁式条件修复越界。
- 原地数组题说明哪些输入信息被覆盖，以及如何利用下标、符号位或固定数量变量保存状态。

1.6.4　LC 215 数组中的第 K 个最大元素

考频与考点：字节最高频题之一。重点考查排序基线、快速选择与分区不变量；最小堆解法在第五类题解中交叉维护。

写代码前确认：`k` 从 1 开始，重复元素分别参与排名；是否允许修改输入会决定能否原地分区。

思路：完整排序后答案是下标 `n-k`，时间 `O(n log n)`。更优的快速选择同样寻找下标 `target = n-k`：随机选择基准，将小于等于基准的元素放在左侧，分区结束时基准已处于最终位置；只继续搜索目标所在的一侧。分区循环的不变量是 `[left, store)` 中元素不大于基准，`[store, i)` 中元素大于基准。

```
from typing import List

class SolutionSort:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        nums.sort()
        return nums[len(nums) - k]
```

```
from typing import List

from random import randint

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        target = len(nums) - k
        left, right = 0, len(nums) - 1

        while left <= right:
            pivot_index = randint(left, right)
            nums[pivot_index], nums[right] = nums[right], nums[pivot_index]
            pivot = nums[right]
            store = left
            for i in range(left, right):
                if nums[i] <= pivot:
                    nums[store], nums[i] = nums[i], nums[store]
                    store += 1
            nums[store], nums[right] = nums[right], nums[store]
```

```
            if store == target:
                return nums[store]
            if store < target:
                left = store + 1
            else:
                right = store - 1
        raise RuntimeError("unreachable")
```

复杂度：排序时间 `O(n log n)`；Python 的 `list.sort()` 会修改输入，最坏辅助空间 `O(n)`。快速选择期望时间 `O(n)`、最坏 `O(n^2)`，额外空间 `O(1)`，也会修改输入。

边界与易错点：第 `k` 大对应升序下标 `n-k`；分区的比较符号与目标下标必须匹配。随机基准降低持续遇到极端划分的概率。

面试追问：数据流或不允许修改输入时可维护大小为 `k` 的最小堆，时间 `O(n log k)`、空间 `O(k)`；最坏时间必须稳定时可讨论 BFPRT。

1.6.5　LC 56 合并区间

考频与考点：高频排序扫描题。考查先建立顺序，再用局部信息维护全局合并结果。

写代码前确认：端点相接是否算重叠；本题闭区间 `[a,b]` 中 `[1,4]` 与 `[4,5]` 应合并。

思路：按左端点升序排序。扫描时保持循环不变量：`merged` 已覆盖所有处理过的区间，内部两两不重叠且按左端点有序。当前左端点不大于最后区间的右端点时更新右边界，否则开启新区间。

```
from typing import List

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda interval: interval[0])
        merged: List[List[int]] = []
        for left, right in intervals:
            if not merged or left > merged[-1][1]:
                merged.append([left, right])
            else:
                merged[-1][1] = max(merged[-1][1], right)
        return merged
```

复杂度：排序时间 `O(n log n)`，扫描 `O(n)`；Python 的 `list.sort()` 最坏辅助空间 `O(n)`，输出空间 `O(n)`。

边界与易错点：必须按左端点排序；本实现会重排输入列表，但为答案创建了新的区间对象，不会改写原区间的端点。若直接复用并修改输入区间对象，还会覆盖端点值。

面试追问：如何插入一个新区间？有序输入可依次处理左侧不相交、重叠、右侧不相交三段，做到 `O(n)`。

1.6.6　LC 33 搜索旋转排序数组

考频与考点：高频旋转数组二分。核心是每轮至少有一半仍然有序。

写代码前确认：元素互不相同。本文统一采用左闭右闭区间 `[left, right]`，循环条件为 `left <= right`。

思路：取中点后，若 `nums[left] <= nums[mid]`，左半段有序；判断目标是否落在 `[nums[left], nums[mid])`，据此舍弃另一半。否则右半段有序，判断目标是否落在 `(nums[mid], nums[right]]`。每次排除中点并收缩闭区间。

```
from typing import List
```

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid
            if nums[left] <= nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return -1
```

复杂度：时间 `O(log n)`，空间 `O(1)`。

边界与易错点：判断有序半段时要包含中点相等情形；各边界的不等号必须与闭区间一致。

面试追问：存在重复元素时，`nums[left] == nums[mid] == nums[right]` 无法判断哪边有序，只能收缩两端，最坏退化为 `O(n)`。

1.6.7　LC 34 在排序数组中查找元素的第一个和最后一个位置

考频与考点：高频边界二分，考查“找任意位置”与“找插入边界”的差别。

写代码前确认：数组非降序；不存在目标时返回 `[-1,-1]`。辅助函数统一在左闭右开区间 `[left, right)` 中找第一个大于等于目标的位置。

思路：`lower_bound(x)` 保持答案始终位于 `[left, right]`：当中点值小于 `x` 时，答案只能在右侧，令 `left = mid + 1`；否则中点可能是答案，令 `right = mid`。目标左边界是 `lower_bound(target)`，右边界是 `lower_bound(target + 1) - 1`。

```python
from typing import List

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        def lower_bound(value: int) -> int:
            left, right = 0, len(nums)
            while left < right:
                mid = left + (right - left) // 2
                if nums[mid] < value:
                    left = mid + 1
                else:
                    right = mid
            return left

        first = lower_bound(target)
        if first == len(nums) or nums[first] != target:
            return [-1, -1]
        return [first, lower_bound(target + 1) - 1]
```

复杂度：时间 `O(log n)`，空间 `O(1)`。

边界与易错点：二分结束后先检查 `first` 是否越界；在定宽整数语言中可另写 `upper_bound` 避免 `target + 1` 溢出。

面试追问：`lower_bound` 也直接解决搜索插入位置；将比较条件改为 `nums[mid] <= value` 可得到第一个严格大于目标的位置。

1.6.8　LC 912 排序数组

考频与考点：排序算法手写题。要求理解快速排序和归并排序在稳定性、最坏复杂度及空间上的取舍。

写代码前确认：是否允许修改输入、是否要求稳定、数据是否可能基本有序或包含大量重复值。快速排序原地但不稳定；归并排序稳定但需线性辅助空间。

思路：快速排序随机选基准并做三路分区，将区间划分成小于、等于和大于基准的三段，只继续处理两侧；这能一次跳过大量重复元素。代码始终递归处理较短一侧、循环处理较长一侧，将递归栈限制为 `O(log n)`。归并排序先分别排好左右半段，再以双指针稳定合并；合并前两半均有序是其循环不变量。

```
from typing import List

from random import randint

class SolutionQuickSort:
    def sortArray(self, nums: List[int]) -> List[int]:
        def quick_sort(left: int, right: int) -> None:
            while left < right:
                pivot = nums[randint(left, right)]
                less, current, greater = left, left, right
                while current <= greater:
                    if nums[current] < pivot:
                        nums[less], nums[current] = nums[current], nums[less]
                        less += 1
                        current += 1
                    elif nums[current] > pivot:
                        nums[current], nums[greater] = nums[greater], nums[current]
                        greater -= 1
                    else:
                        current += 1

                if less - left < right - greater:
                    quick_sort(left, less - 1)
                    left = greater + 1
                else:
                    quick_sort(greater + 1, right)
                    right = less - 1

        quick_sort(0, len(nums) - 1)
        return nums
```

```
from typing import List

class SolutionMergeSort:
    def sortArray(self, nums: List[int]) -> List[int]:
        buffer = [0] * len(nums)
```

```
        def merge_sort(left: int, right: int) -> None:
            if right - left <= 1:
                return
            mid = (left + right) // 2
            merge_sort(left, mid)
            merge_sort(mid, right)
            i, j, write = left, mid, left
            while i < mid or j < right:
                if j == right or (i < mid and nums[i] <= nums[j]):
                    buffer[write] = nums[i]
                    i += 1
                else:
                    buffer[write] = nums[j]
                    j += 1
                write += 1
            nums[left:right] = buffer[left:right]

        merge_sort(0, len(nums))
        return nums
```

复杂度：随机快速排序期望 `O(n log n)`、最坏 `O(n^2)`；较短侧递归使栈空间最坏为 `O(log n)`。归并排序稳定为 `O(n log n)`，辅助空间 `O(n)`。

边界与易错点：朴素固定基准会在有序输入上退化；二路分区也会在大量重复值下形成极不均衡的子问题，因此代码使用三路分区。归并时相等元素优先取左侧才保持稳定。

面试追问：标准库常使用混合策略；若要求严格原地且最坏 `O(n log n)`，可讨论堆排序，但它同样不稳定。

1.6.9　LC 4 寻找两个正序数组的中位数

考频与考点：高难度二分题。考查在较短数组上寻找能把两数组共同划分为左右两半的位置。

写代码前确认：两个数组不会同时为空；本解法令第一个数组较短。二分区间是划分位置的左闭右闭区间 `[0,m]`。

思路：设短数组左侧取 `i` 个元素，长数组左侧取 `j = (m+n+1)//2-i` 个。目标划分满足 `a_left <= b_right` 且 `b_left <= a_right`。若 `a_left > b_right`，`i` 太大；若 `b_left > a_right`，`i` 太小。用正负无穷统一处理切在数组边界的情况。

```
from typing import List

class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1
        m, n = len(nums1), len(nums2)
        left, right = 0, m

        while left <= right:
            i = (left + right) // 2
            j = (m + n + 1) // 2 - i
            a_left = float("-inf") if i == 0 else nums1[i - 1]
            a_right = float("inf") if i == m else nums1[i]
            b_left = float("-inf") if j == 0 else nums2[j - 1]
            b_right = float("inf") if j == n else nums2[j]

            if a_left > b_right:
```

```
                right = i - 1
            elif b_left > a_right:
                left = i + 1
            else:
                left_max = max(a_left, b_left)
                if (m + n) % 2:
                    return float(left_max)
                return (left_max + min(a_right, b_right)) / 2.0
        raise RuntimeError("unreachable")
```

复杂度：时间 `O(log(min(m,n) + 1))`，空间 `O(1)`；加一覆盖短数组为空的边界。

边界与易错点：必须在短数组上二分，才能保证 `j` 不越界；左半部分多放一个元素可统一奇数长度中位数。

面试追问：这也是“第 `k` 小”问题的特例；通用解法可每次排除某数组约 `k/2` 个元素。

1.6.10　LC 75 颜色分类

考频与考点：荷兰国旗问题，考查三区间循环不变量和原地交换。

写代码前确认：数组只含 0、1、2，要求原地完成，不能调用排序函数。

思路：维护 `[0, low)` 全为 0，`[low, current)` 全为 1，`[current, high]` 尚未分类，`(high,n)` 全为 2。遇 0 与 `low` 交换并同时右移；遇 1 只右移；遇 2 与 `high` 交换后只左移 `high`，因为换来的元素尚未检查。

```
from typing import List

class Solution:
    def sortColors(self, nums: List[int]) -> None:
        low, current, high = 0, 0, len(nums) - 1
        while current <= high:
            if nums[current] == 0:
                nums[low], nums[current] = nums[current], nums[low]
                low += 1
                current += 1
            elif nums[current] == 1:
                current += 1
            else:
                nums[current], nums[high] = nums[high], nums[current]
                high -= 1
```

复杂度：时间 `O(n)`，空间 `O(1)`，会覆盖输入数组的原顺序。

边界与易错点：交换 2 后不能立刻移动 `current`；从右侧换来的可能是 0 或 2，必须继续分类。

面试追问：若颜色种类扩展到 `k`，计数排序为 `O(n+k)`；要求稳定时不能直接使用这种交换方案。

1.6.11　LC 902 最大为 N 的数字组合

考频与考点：字节新增题。考查数位计数、前缀约束和组合数学。

写代码前确认：可用数字均为 1 到 9，可重复使用，不含前导零；统计正整数且不超过 `n`。

思路：设 `n` 有 `length` 位、数字集合大小为 `base`。先累加所有位数更短的数，共 `base^1 + ... + base^(length-1)`。再从高位到低位匹配 `n`：当前位置每个小于目标位的可用数字，都能与后续任意数字组成 `base^remaining` 个数；若目标位不在集合中，后续无法继续匹配，立即返回；若每位都匹配，最后把 `n` 本身计入。

```python
from typing import List

class Solution:
    def atMostNGivenDigitSet(self, digits: List[str], n: int) -> int:
        text = str(n)
        base = len(digits)
        answer = sum(base ** size for size in range(1, len(text)))

        for index, target_digit in enumerate(text):
            remaining = len(text) - index - 1
            smaller = sum(digit < target_digit for digit in digits)
            answer += smaller * (base ** remaining)
            if target_digit not in digits:
                return answer
        return answer + 1
```

复杂度：设 `n` 有 `L` 位、可用数字有 `D` 个，时间 `O(LD)`，空间 `O(1)`。

边界与易错点：较短位数必须从 1 位开始；只有完整匹配 `n` 的所有位后，才能把 `n` 自身加一。

面试追问：若数字集合包含 0，需要单独限制最高位；更通用的上下界、重复次数等约束可写成数位 DP。

1.6.12　LC 35 搜索插入位置

考频与考点：新增基础边界二分，是后续查找左右边界的模板。

写代码前确认：数组严格递增；目标不存在时返回保持有序的插入下标。使用左闭右开区间 `[left, right)`。

思路：寻找第一个大于等于 `target` 的位置。循环中答案始终位于闭合的候选边界 `[left,right]`；中点值小于目标时排除中点及左侧，否则保留中点并收缩右边界。`left == right` 时即为答案，也可能等于数组长度。

```python
from typing import List

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums)
        while left < right:
            mid = left + (right - left) // 2
            if nums[mid] < target:
                left = mid + 1
            else:
                right = mid
        return left
```

复杂度：时间 `O(log n)`，空间 `O(1)`。

边界与易错点：右边界初始化为 `len(nums)`，因此目标大于所有元素时自然返回末尾；半开区间更新右端时不能减一。

面试追问：将比较改为 `nums[mid] <= target`，可求第一个严格大于目标的插入位置。

1.6.13　LC 54 螺旋矩阵

考频与考点：新增矩阵模拟题。重点不是方向数组，而是维护尚未访问矩形的边界。

写代码前确认：矩阵非空且为规则矩形；只返回遍历结果，不修改矩阵。

思路：维护闭合边界 `top`、`bottom`、`left`、`right`。每轮开始时，边界内恰是尚未访问的矩形；依次取顶边、右边，再在仍有剩余行/列时取底边和左边。每访问一条边就向内收缩对应边界。

```
from typing import List

class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        top, bottom = 0, len(matrix) - 1
        left, right = 0, len(matrix[0]) - 1
        answer: List[int] = []

        while top <= bottom and left <= right:
            for col in range(left, right + 1):
                answer.append(matrix[top][col])
            top += 1

            for row in range(top, bottom + 1):
                answer.append(matrix[row][right])
            right -= 1

            if top <= bottom:
                for col in range(right, left - 1, -1):
                    answer.append(matrix[bottom][col])
                bottom -= 1

            if left <= right:
                for row in range(bottom, top - 1, -1):
                    answer.append(matrix[row][left])
                left += 1
        return answer
```

复杂度：每个元素访问一次，时间 `O(mn)`；除输出外空间 `O(1)`。

边界与易错点：单行或单列矩阵会在一轮内耗尽，访问底边、左边前必须再次检查边界，避免重复。

面试追问：若要求原地螺旋写入矩阵，可保留同样的边界不变量，只把“读取”替换为“写入”。

1.6.14　LC 73 矩阵置零

考频与考点：新增原地矩阵题。考查用首行、首列复用标记空间。

写代码前确认：某元素为 0 时整行整列置零；要求常数额外空间，允许覆盖矩阵。

思路：先单独记录首行、首列原本是否含零。扫描内部元素时，把其所在行的首列和所在列的首行置零作为标记。此时循环不变量是：已扫描内部区域中每个零对应的行列都已在边界留下标记。再据标记清零内部，最后处理首行首列。原地代价是首行和首列的原始值被用作元数据并最终可能被覆盖。

```
from typing import List

class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        rows, cols = len(matrix), len(matrix[0])
        first_row_zero = any(matrix[0][col] == 0 for col in range(cols))
        first_col_zero = any(matrix[row][0] == 0 for row in range(rows))

        for row in range(1, rows):
            for col in range(1, cols):
```

```
                if matrix[row][col] == 0:
                    matrix[row][0] = 0
                    matrix[0][col] = 0

        for row in range(1, rows):
            for col in range(1, cols):
                if matrix[row][0] == 0 or matrix[0][col] == 0:
                    matrix[row][col] = 0

        if first_row_zero:
            for col in range(cols):
                matrix[0][col] = 0
        if first_col_zero:
            for row in range(rows):
                matrix[row][0] = 0
```

复杂度：时间 `O(mn)`，额外空间 `O(1)`。

边界与易错点：首行首列共用 `matrix[0][0]`，所以必须用两个独立布尔值保存它们原本的状态；不能边发现零边直接清整行列，会污染后续判断。

面试追问：若不允许修改输入，只能另建结果或至少保存需要清零的行列集合，空间为 `O(m+n)`。

1.6.15　LC 74 搜索二维矩阵

考频与考点：新增二分题。矩阵每行有序且下一行首元素大于上一行末元素，可以视为一维有序数组。

写代码前确认：确认题目是 LC 74 的全局有序条件，而不是 LC 240 的行列分别有序。使用左闭右开虚拟下标区间 `[left,right)`。

思路：把虚拟下标 `index` 映射到 `row = index // cols`、`col = index % cols`，在长度 `rows*cols` 的有序序列上做普通二分。循环前目标若存在必位于当前半开区间内；每次排除中点或保留右侧候选。

```
from typing import List

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        rows, cols = len(matrix), len(matrix[0])
        left, right = 0, rows * cols
        while left < right:
            mid = left + (right - left) // 2
            value = matrix[mid // cols][mid % cols]
            if value < target:
                left = mid + 1
            else:
                right = mid
        return left < rows * cols and matrix[left // cols][left % cols] == target
```

复杂度：时间 `O(log(mn))`，空间 `O(1)`。

边界与易错点：虚拟下标的除数和模数都是列数；最终候选可能等于元素总数，访问前要检查。

面试追问：若只有每行、每列分别递增，展平后不再有序，应使用 LC 240 的右上角阶梯搜索。

1.6.16　LC 153 寻找旋转排序数组中的最小值

考频与考点：新增旋转数组二分。与 LC 33 不同，这里利用最小值相对右端点的单调分区。

写代码前确认：元素互不相同；使用左闭右闭区间 `[left,right]`，并保持最小值始终在区间内。

思路：当 `nums[mid] > nums[right]`，中点位于旋转前的较大段，最小值严格在右侧，令 `left = mid + 1`；否则中点可能就是最小值，令 `right = mid`。循环到单点时返回。

```
from typing import List

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = left + (right - left) // 2
            if nums[mid] > nums[right]:
                left = mid + 1
            else:
                right = mid
        return nums[left]
```

复杂度：时间 `O(log n)`，空间 `O(1)`。

边界与易错点：与右端点比较时，`right = mid` 不能写成 `mid - 1`，因为中点仍可能是最小值。

面试追问：存在重复值且 `nums[mid] == nums[right]` 时只能执行 `right -= 1`，最坏退化为 `O(n)`。

1.6.17　LC 169 多数元素

考频与考点：新增数组题。Boyer-Moore 投票考查“不同元素两两抵消”的不变量。

写代码前确认：多数元素出现次数严格大于 `n/2`，并保证存在；若不保证存在，最后必须再计数验证。

思路：维护候选人和净票数。净票为 0 时，当前元素成为新候选；相同则加一，不同则减一。扫描完任意前缀后，已经抵消的不同元素不会影响真正多数元素在剩余部分中的多数地位，因此最终候选必为答案。

```
from typing import List

class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        candidate = 0
        votes = 0
        for value in nums:
            if votes == 0:
                candidate = value
            votes += 1 if value == candidate else -1
        return candidate
```

复杂度：时间 `O(n)`，空间 `O(1)`，不修改输入。

边界与易错点：投票只产生候选人；题目不保证多数存在时，需要第二次扫描确认其次数是否超过 `n/2`。

面试追问：寻找出现次数超过 `n/3` 的元素时最多有两个候选，可维护两个候选及票数并在最后验证。

1.6.18　LC 238 除自身以外数组的乘积

考频与考点：新增前后缀题。考查不用除法、线性时间和常数额外空间的组合信息。

写代码前确认：不能使用除法；输出数组不计入额外空间。输入可能含零。

思路：第一次从左向右令 `answer[i]` 保存 `i` 左侧所有元素乘积；循环不变量是写入前 `prefix` 等于当前位置

左侧乘积。第二次从右向左维护右侧乘积 `suffix`，把它乘入答案后再吸收当前值。输入不被覆盖，所有前缀信息放在输出数组中。

```
from typing import List

class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        answer = [1] * len(nums)
        prefix = 1
        for i, value in enumerate(nums):
            answer[i] = prefix
            prefix *= value

        suffix = 1
        for i in range(len(nums) - 1, -1, -1):
            answer[i] *= suffix
            suffix *= nums[i]
        return answer
```

复杂度：时间 `O(n)`，除输出外空间 `O(1)`。

边界与易错点：先把当前 `prefix` 写入答案，再乘当前元素；零无需特判，前后缀乘积会自然处理一个或多个零。

面试追问：若允许除法，需要分别处理零的数量；相比之下前后缀方案更统一，也没有整除或浮点精度问题。

1.6.19　LC 240 搜索二维矩阵 II

考频与考点：新增矩阵搜索题。考查从特殊角点出发，每次排除一整行或一整列。

写代码前确认：每行从左到右递增，每列从上到下递增，但矩阵不能整体展平成有序数组。

思路：从右上角开始。当前位置大于目标，则该列当前位置以下也都不可能更小到命中，排除当前列；当前位置小于目标，则该行当前位置左侧也都更小，排除当前行。循环不变量是目标若存在，始终位于左下方尚未排除的矩形中。

```
from typing import List

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        row, col = 0, len(matrix[0]) - 1
        while row < len(matrix) and col >= 0:
            value = matrix[row][col]
            if value == target:
                return True
            if value > target:
                col -= 1
            else:
                row += 1
        return False
```

复杂度：时间 `O(m+n)`，空间 `O(1)`。

边界与易错点：从左上角无法确定应排除行还是列；右上和左下才同时提供一增一减的两个方向。

面试追问：也可逐行二分，复杂度 `O(m log n)`；应根据矩阵形状比较它与 `O(m+n)` 的实际代价。

1.6.20　LC 287 寻找重复数

考频与考点：新增数组题。经典解法把数组下标和值构成函数图，用 Floyd 判环定位入口。

写代码前确认：长度为 `n+1`，值域为 `[1,n]`，只有一个重复值但可能重复多次；不能修改输入且额外空间要求 `O(1)`。

思路：把 `index -> nums[index]` 看作单出边图。因为值域不含 0，从下标 0 出发必进入一个环；重复值是两条路径汇合的位置，也就是环入口。先用快慢指针找到环内相遇点，再让一个指针回到 0，两者同步前进，相遇处即入口。

```
from typing import List

class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        slow = nums[0]
        fast = nums[nums[0]]
        while slow != fast:
            slow = nums[slow]
            fast = nums[nums[fast]]

        slow = 0
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]
        return slow
```

复杂度：时间 `O(n)`，空间 `O(1)`，不修改输入。

边界与易错点：指针移动的是“值对应的下标”，不是按数组位置顺序移动；第二阶段必须把一个指针放回起点 0。

面试追问：也可在值域 `[1,n]` 上二分，统计小于等于中点的元素数是否超过中点，时间 `O(n log n)`、空间 `O(1)`。

1.6.21　LC 442 数组中重复的数据

考频与考点：新增原地下标标记题。利用值域与数组长度一致，把符号位当作访问标记。

写代码前确认：所有值位于 `[1,n]`，每个元素最多出现两次；允许修改输入数组。

思路：值 `v` 映射到下标 `v-1`。首次见到时将该位置取负；再次见到时发现该位置已为负，说明 `v` 重复。循环不变量是：已扫描元素对应的位置为负，当且仅当该值此前出现过。原地代价是数组元素的符号被覆盖。

```
from typing import List

class Solution:
    def findDuplicates(self, nums: List[int]) -> List[int]:
        answer: List[int] = []
        for raw_value in nums:
            value = abs(raw_value)
            index = value - 1
            if nums[index] < 0:
                answer.append(value)
            else:
                nums[index] = -nums[index]
        return answer
```

复杂度：时间 `O(n)`，除输出外空间 `O(1)`。

边界与易错点：读取当前值必须先取绝对值，因为它可能已被此前标记；若输入值可出现三次以上，第二、三次都会加入答案，需要额外约束或恢复标记。

面试追问：若需要恢复输入，可在结束后将所有元素取绝对值；若不允许修改输入，则使用哈希集合需要 `O(n)` 空间。

1.6.22　LC 704 二分查找

考频与考点：新增基础模板题。重点是区间定义、循环条件和边界更新三者一致。

写代码前确认：数组严格递增；不存在返回 `-1`。这里使用左闭右闭区间 `[left,right]`。

思路：循环不变量是：目标若存在，必在当前闭区间内。比较中点后若不相等，可以安全排除中点，所以分别更新为 `mid+1` 或 `mid-1`。当 `left > right` 时候选区间为空。

```
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid
            if nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return -1
```

复杂度：时间 `O(log n)`，空间 `O(1)`。

边界与易错点：闭区间必须使用 `left <= right`；若把右端初始化为 `len(nums)`，就应整体改用半开区间模板，不能混搭。

面试追问：二分的本质不是“有序数组”，而是可判断答案位于哪一侧的单调谓词；边界二分、答案二分都基于这一点。

zero2Leetcode 蓝皮书

代码能写，思路能讲，追问能接

https://github.com/ranxi2001/zero2Leetcode