perplexity

LeetCode难题精解：中位数与二分查找的高阶应用技巧总览

在LeetCode的算法题库中，中位数相关问题和二分查找的变形应用是公认的高频难点。本文通过剖析六大类经典难题，系统梳理二分查找在非有序场景、动态规划优化、滑动窗口维护等场景中的高阶应用模式。所有解法均以Python实现，并着重揭示算法设计的核心思想与工程实践要点。

一、二分查找突破有序结构限制

1.1 二维空间中的元素定位

当处理非传统有序结构时，二分查找可通过维度投影实现高效搜索。典型例题包括：

1.1.1 搜索二维矩阵II（LeetCode 240）

通过在行列双维度实施剪枝策略，达到O(m+n)时间复杂度。关键点在于利用矩阵行列单调性特征：

```
def searchMatrix(matrix, target):
    if not matrix: return False
    row, col = 0, len(matrix[^0])-1
    while row < len(matrix) and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] > target:
            col -= 1
        else:
            row += 1
    return False
```

1.1.2 寻找峰值（LeetCode 162）

利用局部单调性判断趋势方向，只需O(log n)即可定位任意峰值：

```
def findPeakElement(nums):
    left, right = 0, len(nums)-1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid+1]:
            right = mid
        else:
            left = mid + 1
    return left
```

1.2 非数值型数据的二分处理

当数据不具备直接可比性时，可通过构造判断函数实现二分逻辑：

1.2.1 猜数字大小（LeetCode 374）

典型API交互型二分场景，重点在于安全计算中间值：

```
def guessNumber(n):
    left, right = 1, n
    while left < right:
        mid = left + (right - left) // 2
        res = guess(mid)
        if res == 0:
            return mid
        elif res == 1:
            left = mid + 1
        else:
            right = mid
    return left
```

二、动态规划与二分查找的融合

2.1 最长递增子序列优化（LeetCode 300）

传统$O(n^2)$解法升级为$O(n \log n)$的贪心+二分策略：

```
def lengthOfLIS(nums):
    tails = []
    for num in nums:
        idx = bisect.bisect_left(tails, num)
        if idx == len(tails):
            tails.append(num)
        else:
            tails[idx] = num
    return len(tails)
```

该算法的核心在于维护tails数组：tails[i]表示长度为i+1的所有LIS中最小末尾值。通过二分查找确定当前数字应插入的位置，保证数组始终有序[1] [2]。

2.2 俄罗斯套娃信封问题（LeetCode 354）

二维LIS问题可通过排序降维后应用相同技巧：

```
def maxEnvelopes(envs):
    envs.sort(key=lambda x: (x[^0], -x[^1]))
    heights = [h for _, h in envs]
    return lengthOfLIS(heights)
```

此处将宽度升序、高度降序排列，将问题转化为纯高度序列的LIS问题，有效避免同一宽度多次选择 [1:1]。

三、中位数问题的二分处理范式

3.1 双有序数组中位数（LeetCode 4）

通过分割线理论实现O(log min(m,n))时间复杂度：

```
def findMedianSortedArrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    left, right = 0, m
    total = m + n
    half = (total + 1) // 2

    while left <= right:
        i = (left + right) // 2
        j = half - i

        if i < m and nums2[j-1] > nums1[i]:
            left = i + 1
        elif i > 0 and nums1[i-1] > nums2[j]:
            right = i - 1
        else:
            if i == 0: max_left = nums2[j-1]
            elif j == 0: max_left = nums1[i-1]
            else: max_left = max(nums1[i-1], nums2[j-1])

            if total % 2 == 1:
                return max_left

            if i == m: min_right = nums2[j]
            elif j == n: min_right = nums1[i]
            else: min_right = min(nums1[i], nums2[j])

            return (max_left + min_right) / 2
```

关键点在于确保分割线左侧元素全小于右侧，通过二分调整分割位置 [3] [4]。

3.2 数据流中位数维护（LeetCode 295）

双堆结构实现动态平衡：

```
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # max heap
        self.large = []  # min heap
```

```python
    def addNum(self, num):
        if len(self.small) == len(self.large):
            heapq.heappush(self.large, -heapq.heappushpop(self.small, -num))
        else:
            heapq.heappush(self.small, -heapq.heappushpop(self.large, num))

    def findMedian(self):
        if len(self.small) == len(self.large):
            return (self.large[^0] - self.small[^0])/2
        else:
            return self.large[^0]
```

通过保持两个堆的大小差不超过1，确保中位数可快速获取[1:2] [2:1]。

四、滑动窗口中的中位数维护

4.1 滑动窗口中位数（LeetCode 480）

双堆+哈希表延迟删除技巧：

```python
import heapq
from collections import defaultdict

class SlidingWindowMedian:
    def __init__(self):
        self.small = []  # max heap
        self.large = []  # min heap
        self.balance = 0
        self.delay_remove = defaultdict(int)

    def _prune(self, heap, is_max_heap):
        while heap:
            num = -heap[^0] if is_max_heap else heap[^0]
            if self.delay_remove[num] > 0:
                self.delay_remove[num] -= 1
                heapq.heappop(heap)
            else:
                break

    def _make_balanced(self):
        if self.balance == 2:  # small比large多两个
            heapq.heappush(self.large, -heapq.heappop(self.small))
            self.balance -= 1
            self._prune(self.small, True)
        elif self.balance == -1:  # large比small多一个
            heapq.heappush(self.small, -heapq.heappop(self.large))
            self.balance += 1
            self._prune(self.large, False)

    def add_num(self, num):
        if not self.small or num <= -self.small[^0]:
            heapq.heappush(self.small, -num)
            self.balance += 1
```

```
        else:
            heapq.heappush(self.large, num)
            self.balance -= 1
        self._make_balanced()

    def remove_num(self, num):
        self.delay_remove[num] += 1
        if num <= -self.small[^0]:
            self.balance -= 1
            if num == -self.small[^0]:
                self._prune(self.small, True)
        else:
            self.balance += 1
            if num == self.large[^0]:
                self._prune(self.large, False)
        self._make_balanced()

    def find_median(self):
        if self.balance == 0:
            return (-self.small[^0] + self.large[^0]) / 2
        else:
            return -self.small[^0]
```

该实现通过延迟删除机制处理移出窗口元素，保持堆结构有效性[1:3] [2:2]。

五、分治策略下的中位数查找

5.1 多个有序序列的中位数

扩展到k个有序数组的场景，采用多路归并与二分结合：

```
def findMedianKSortedArrays(arrays):
    def count_less_equal(x):
        cnt = 0
        for arr in arrays:
            if not arr: continue
            if arr[-1] <= x:
                cnt += len(arr)
            else:
                cnt += bisect.bisect_right(arr, x)
        return cnt

    low = min(arr[^0] for arr in arrays if arr)
    high = max(arr[-1] for arr in arrays if arr)
    total = sum(len(arr) for arr in arrays)

    while low < high:
        mid = (low + high) // 2
        cnt = count_less_equal(mid)
        if cnt < (total + 1) // 2:
            low = mid + 1
        else:
```

```
            high = mid
    return low
```

通过二分猜测中位数值，并统计所有数组中不大于该值的元素总数[3:1] [2:3]。

六、几何空间中的中位数优化

6.1 最佳碰头地点（LeetCode 296）

在一维场景中，中位数位置即为最优解，二维场景可分解为两个独立的一维问题：

```
def minTotalDistance(grid):
    rows = []
    cols = []
    for i in range(len(grid)):
        for j in range(len(grid[^0])):
            if grid[i][j] == 1:
                rows.append(i)
                cols.append(j)

    def min_dist(points):
        points.sort()
        median = points[len(points)//2]
        return sum(abs(p - median) for p in points)

    return min_dist(rows) + min_dist(cols)
```

该算法证明中位数点在曼哈顿距离下的最优性[1:4] [2:4]。

结论

通过系统梳理可见，中位数问题与二分查找的高阶应用存在深層次的算法共性。核心在于：1）合理定义搜索空间 2）构造高效判定条件 3）动态维护数据结构平衡。掌握这些模式后，可显著提升对Hard难度题目的解析能力。建议在实际练习中，重点关注问题转化能力的培养，将复杂场景映射到经典算法框架，并注意边界条件的处理优化。

⁂

1. https://blog.csdn.net/beilizhang/article/details/108538433
2. https://liweiwei1419.github.io/leetcode-solution-blog/choice-goods/articles/【特别推荐】十分好用的二分查找法模板（Python 代码、Java 代码）.html
3. https://www.cnblogs.com/xiaoqiangink/p/12920570.html
4. https://liweiwei1419.github.io/leetcode-solution-blog/leetcode-problemset/binary-search/0004-median-of-two-sorted-arrays.html