二分算法 Bisection Algorithm

1 二分查找

二分查找的应用场景是在一个递增顺序排列的整数数组中找到大于等于、大于、小于或小于等于某个整数的数组元素的索引。对数组进行暴力遍历会使得程序的时间复杂度为 $O(n)$，为了降低时间复杂度，采用二分查找快速找到所求值。

二分查找的过程：二分查找的关键在于根据二分处的值更新 L 和 R 指针。以用二分查找查找大于等于 `target` 值的元素的左索引的情况为例，L 的初始值为 0，R 的初始值为 $n-1$，M 的初始值为 $\lfloor (L+R)/2 \rfloor$。如果索引为 M 的数组元素小于 `target`，则 `target` 一定位于区间 $(M, R]$ 中，则指针 L 被更新为 $M+1$ 并进行下一次循环；如果索引为 M 的数组元素大于 `target`，则 `target` 一定位于区间 $[L, M)$ 中，指针 R 被更新为 $M-1$ 并进行下一次循环。由此得知，在循环过程中，$R+1$ **对应的数组元素始终大于等于 `target`；**$L-1$ **对应的数组元素始终小于 `target`**。据此，也可以得到循环的终止条件，即 R 在所有小于 `target` 值的元素的最右侧，L 在所有大于等于 `target` 值的元素的最左侧，也即 $R < L$。最后返回大于等于 `target` 的左索引即为 L 或 $R+1$。

闭区间与开区间：这里我们设定的 L, R 初始值为 $0, n-1$，这种设定被称为“闭区间”。存在其他设定方法。例如可以设置 R 为 n，此时相当于 R 一侧为开区间。在更新 R 时，就相应地设置 R 为 M 而不是 $M-1$。相应的，循环的终止条件也应改为 $R = L$。这种方法实质上等价于令一个新的 R_2 等于闭区间方法中的 $R+1$。

其他大小关系的二分查找：除了上述查找等于 `target` 的左索引的程序，查找大于 `target` 的左索引的程序等价于查找大于等于 `target+1` 的左索引的程序；查找小于 `target` 的左索引的程序等价于查找大于等于 `target` 的左索引再 `-1` 的程序；查找小于等于 `target` 的左索引的程序等价于查找大于等于 `target+1` 的左索引再 `-1` 的程序。

用“闭区间”方法编写查找大于等于 `target` 的初始索引的代码：

```python
def lower_bound(nums: List[int], target: int) -> int:
    left = 0
    right = len(nums) - 1

    while left <= right:
        middle = (left + right) // 2 # 注意在哪里更新middle

        if target > nums[middle]:
```

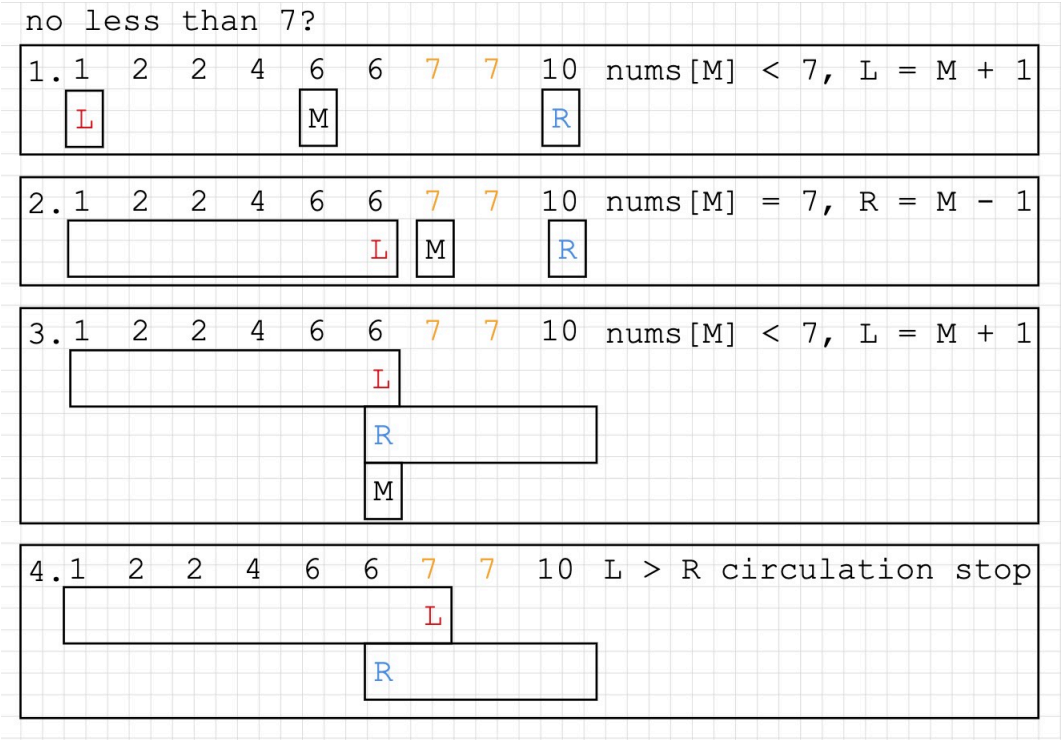

Figure. 1. bisect 用二分查找查找大于等于 target 值的元素的左索引

```
            left = middle + 1
        else:
            right = middle - 1

    return left # 注意return的值
```

使用二分查找:在实际编写代码时,可以通过 bisect_left(nums, target) 直接得到大于等于 target 的初始索引;通过 bisect_right(nums, target) 直接得到大于等于 target + 1 的初始索引。注意 bisect 还可以限制范围并修改用于比较的元素: bisect_left(nums, target, start, end, key = lambda x : folmula)。

二分查找后的判断:

target 不存在的情况:

1. 数组中的所有值都小于 target: 此时大于等于 target 的左索引为 n。
2. 数组中的所有值都大于 target: 此时大于等于 target 的左索引为 0。
3. target 存在在数组元素之间: 此时大于等于 target 的左索引等于等于 target+1 的初始索引, 都为 target 右侧首字符的索引。

1.1 easy_1385. 两个数组间的距离值----------

Problem 1.1

给你两个整数数组 *arr1*, *arr2* 和一个整数 *d*, 请你返回两个数组之间的距离值。

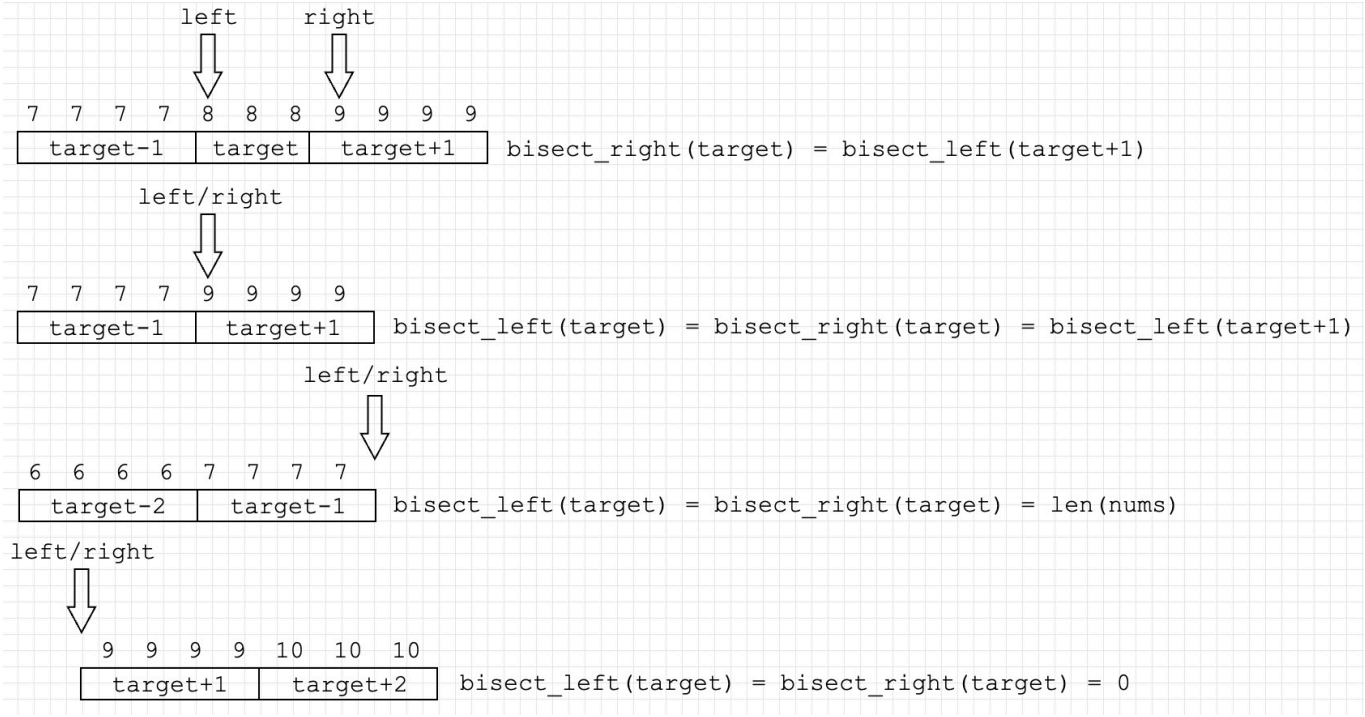

Figure. 2. bisect 方法与不同情况下的索引

> 「距离值」定义为符合此距离要求的元素数目：对于元素 *arr1[i]*，不存在任何元素 *arr2[j]* 满足 *|arr1[i]-arr2[j]| <= d*。

1.1.1 解法 1-暴力遍历

直接用两层循环嵌套分别遍历 arr1 与 arr2，遇到满足题目条件的数组元素时就将答案增加 1。

```
def findTheDistanceValue(self, arr1: List[int], arr2: List[int], d: int) -> int:
  count = 0
  for num1 in arr1:
        if all(abs(num1 - num2) > d for num2 in arr2): # all()用于判断一个可迭代对象中的所有元素是否都为True
            count += 1
    return count
```

这种方法的时间复杂度是 $n \times m$。

1.1.2 解法 2-二分查找

首先用 sort() 方法对 arr2 排序，随后先遍历 arr1，令 num1 = arr1[i]，若 num1 ± d 不存在于数组 arr2 中，则满足题目条件，答案增加 1。

```
def findTheDistanceValue(self, arr1: List[int], arr2: List[int], d: int) -> int:
    arr2.sort()
    ans = 0
```

```
    for num1 in arr1:
        start = bisect_left(arr2, num1 - d)

        if start == len(arr2) or arr2[start] > num1 + d: # 注意不需要再用一次二分查找。此时已知arr2[start] >= num1 - d，只需要arr2[start]同时大于num1 + d就可以证明num1 - d和num1 + d都不存在于arr2[start - 1]和arr2[start]之间。
            ans += 1

    return ans
```

这种方法的时间复杂度主要是排序算法和二分查找算法的贡献。排序算法的时间复杂度是 $O(m \log m)$，二分查找算法的时间复杂度由于 n 次循环的存在是 $O(n \log m)$，所以程序的总时间复杂度是 $O(m \log m + n \log m)$。

1.1.3 解法 3-双指针

我们在遍历 `arr1` 的同时用一个指针 `j` 维护满足 `arr2[j] < num1 - d` 的数的下标。如果在下一次循环中发现 `arr2[j + 1] > num1 + d`，则 `[num1 - d, num1 + d]` 必然在 `(arr[j], arr[j + 1])` 中。这表明 `arr[j] + d < num1 < arr[j + 1] - d`，即满足题目的条件，答案增加 1。

```
def findTheDistanceValue(self, arr1: List[int], arr2: List[int], d: int) -> int:
    arr1.sort()
    arr2.sort()

    ans = j = 0

    for num1 in arr1:
        while j < len(arr2) and arr2[j] < num1 - d:
            j += 1
        if j == len(arr2) or arr2[j] > num1 + d:
            ans += 1

    return ans
```

这种方法的时间复杂度主要是排序算法的贡献，即 $O(n \log n + m \log m)$。这里循环遍历的时间复杂度为 $O(n + m)$，可以忽略，因为对 `arr1` 排序后指针不需要每次都从 `j = 0` 开始遍历 `arr2`，而是单调递增。

1.2 easy_2389. 和有限的最长子序列----------

Problem 1.2

> 给你一个长度为 *n* 的整数数组 *nums*，和一个长度为 *m* 的整数数组 *queries*。
>
> 返回一个长度为 *m* 的数组 *answer*，其中 *answer[i]* 是 *nums* 中元素之和小于等于 *queries[i]* 的子序列的最大长度。

> 子序列是由一个数组删除某些元素（也可以不删除）但不改变剩余元素顺序得到的一个数组。

1.2.1 解法 1-暴力遍历

暴力遍历每个 queries 元素，对于每个元素，在 nums 中从小到大减去值直到小于 0，记录减去值的次数。

```python
def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:
    nums.sort()
    counts = []

    for num in queries:
        i = 0
        count = 0
        while i <= len(nums) - 1 and num - nums[i] >= 0: # 在nums中从小到大减去值直到小于0，注意索引条件
            num = num - nums[i]
            print(nums[i])
            i += 1
            count += 1
        counts.append(count)

    return counts
```

1.2.2 解法 2-前缀和与二分查找

我们通过二分查找查找 queries 在 nums 数组的前缀和中的索引。该索引与 0 的差再 +1 即可得到子序列的长度。

```python
def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:
    nums.sort() # 排序不能忘
    counts = [] # 初始化结果数组

    # nums的前缀和数组
    n = len(nums)
    prefixSum = [0] * n
    prefixSum[0] = nums[0]
    for i in range(1, n):
        prefixSum[i] = prefixSum[i - 1] + nums[i]

    # 在前缀和数组中二分查找num
    for num in queries:
        start = bisect_left(prefixSum, num)
        if start == n: # 如果num大于前缀和数组中的所有值，即start=n，则直接返回nums的长度n
            counts.append(n)
        elif prefixSum[start] == num: # 如果num存在于前缀和数组中，则需要返回索引+1
            counts.append(start + 1)
        else: # 如果num小于前缀和数组中的所有值，即start=0，此时直接返回start即可
            counts.append(start)

    return counts
```

这种方法还可以进行优化。注意到采用二分查找算法查找大于等于 numa 的初始索引并计算子序列长度完全等价于直接采用二分查找算法查找大于 num 的初始索引。同时前缀和可以直接记录在 nums 中，答案可以直接记录在 queries 中。

```
def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:
    nums.sort() # 排序不能忘

    for i in range(1, len(nums)):
        nums[i] = nums[i - 1] + nums[i] # 直接将前缀和保存在nums中，减少空间占用

    for i, target in enumerate(queries):
        queries[i] = bisect_right(nums, target) # 直接用queries保存结果，同时用bisect_right而不是bisect_left简化程序

    return queries
```

该方法的时间复杂度是 $n\log n + n + m\log n$，其中 $n\log n$ 表示排序算法的时间复杂度，n 表示计算前缀和的时间复杂度，$m\log n$ 表示循环中嵌套二分查找的时间复杂度。

1.3 medium_2080. 区间内查询数字的频率----------

Problem 1.3

请你设计一个数据结构，它能求出给定子数组内一个给定值的频率。

子数组中一个值的频率指的是这个子数组中这个值的出现次数。

请你实现 *RangeFreqQuery* 类：

- *RangeFreqQuery(int[] arr)* 用下标从 *0* 开始的整数数组 *arr* 构造一个类的实例。
- *int query(int left, int right, int value)* 返回子数组 *arr[left...right]* 中 *value* 的频率。

一个子数组指的是数组中一段连续的元素。*arr[left...right]* 指的是 *nums* 中包含下标 *left* 和 *right* 在内的中间一段连续元素。

1.3.1 解法 1-切割、排序与二分查找

```
def __init__(self, arr: List[int]):
    self.nums = arr

def query(self, left: int, right: int, value: int) -> int:
    arr = self.nums[left:right + 1]
    arr.sort()

    return bisect_left(arr, value) - bisect_right(arr, value)
```

这种方法的时间复杂度是 $n\log n + q\log n$，其中 q 为调用 query 的次数。因为需要先排序再进行二分查找。

1.3.2 解法 2-哈希表与二分查找

在这种方法中，我们不再需要进行排序，而是通过哈希表维护 arr 中每个数字的下标，随后直接判断 [left,right] 区间中是否存在该数字的下标即可。

假设需要在 arr = [1, 3, 2, 2, 3, 2, 1] 的区间 [1,4] 中统计数字 2 出现的次数：

1. 首先通过哈希表维护每个数字的下标，得到 pos = 1: [0, 6], 2: [2, 3, 5], 3: [1, 4]。
2. 提取统计值 value 对应的值：nums = self.pos[value] = [2, 3, 5](这里 value = 2)。
3. 随后判断 [1,4] 区间的位置。bisect_left(nums, left) = [0](这里 left = 1);bisect_right (nums, right)= [2](这里 right = 4)。所以频率 = 2 - 0 = 2。

```python
def __init__(self, arr: List[int]):
    pos = defaultdict(list) # defaultdict是一个可以自动为不存在的键创建一个默认值的字典
    for i, num in enumerate(arr):
        pos[num].append(i)
    self.pos = pos

def query(self, left: int, right: int, value: int) -> int:
    nums = self.pos[value]
    return (bisect_right(nums, right) - 1) - bisect_left(nums, left) + 1
```

2 二分答案求最小

对于二分答案的题目，应该直接对所求值进行二分。二分答案的关键点在于：1. 确定 left 与 right 的初始值。2. 确定更新 middle 时的条件判断语句/函数。

确定初始值的关键在于删除一定不满足题目要求的区间和一定满足题目要求的区间。

二分答案求最小的题目大多难度适中，在 left 和 right 的选择上相较于其他二分答案的题目难度更大但可接受，check 函数的编写思路较为简单直接。

一类非常典型的二分答案求最小的题目是求能够完成任务的时间或者能力最小值。这类题目的情景是非常容易理解的，因此程序编写的逻辑非常简易。**medium_2187 完成旅途的最少时间**、**medium_1011 在 D 天内送达包裹的能力**、**medium_3296 移山所需的最少秒数**都是此类题目。

2.1 medium_1283. 使结果不超过阈值的最小除数----------

Problem 2.1

> 给你一个整数数组 *nums* 和一个正整数 *threshold*，你需要选择一个正整数作为除数，然后将数组里每个数都除以它，并对除法结果求和。
>
> 请你找出能够使上述结果小于等于阈值 *threshold* 的除数中最小的那个。

每个数除以除数后都向上取整，比方说 *7/3 = 3*，*10/2 = 5*。
题目保证一定有解。

2.1.1 解法-二分答案求最小

假设除数为 m。根据题目要求，我们需要找到最小的满足 $\sum_{i=0}^{n-1}\left\lceil \frac{\texttt{nums[i]}}{\texttt{m}} \right\rceil$ <= threshold 的 m。

首先确定 left 与 right 的初始值：

left ：可以设置为 1，因为题目中要求了除数为正整数。

right ：可以设置为 max(nums)，因为此时 $\sum_{i=0}^{n-1}\left\lceil \frac{\texttt{nums[i]}}{\texttt{m}} \right\rceil$ = n <= threshold（据题目条件）。

接着确定更新 middle 时的条件判断语句。关于上取整的计算，当 a, b 均为正整数时，有：

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor = \left\lfloor \frac{a-1}{b} \right\rfloor + 1$$

所以：

$$\sum_{i=0}^{n-1}\left\lceil \frac{\texttt{nums[i]}}{\texttt{m}} \right\rceil = \sum_{i=0}^{n-1}\left(\left\lfloor \frac{\texttt{nums[i]}-1}{\texttt{m}} \right\rfloor + 1\right) = \sum_{i=0}^{n-1}\left\lfloor \frac{\texttt{nums[i]}-1}{\texttt{m}} \right\rfloor + \texttt{n}$$

因此条件 $\sum_{i=0}^{n-1}\left\lceil \frac{\texttt{nums[i]}}{\texttt{m}} \right\rceil$ <= threshold 变为 $\sum_{i=0}^{n-1}\left\lfloor \frac{\texttt{nums[i]}-1}{\texttt{m}} \right\rfloor$ <= threshold $- n$。

```python
def smallestDivisor(self, nums: List[int], threshold: int) -> int:
    left = 1
    right = max(nums)

    while left <= right:
        mid = (left + right) // 2

        if sum((x - 1) // mid for x in nums) <= threshold - len(nums):
            right = mid - 1 # 当sum <= threshold的时候需要减小除数以增大sum
        else:
            left = mid + 1

    return left
```

2.2 medium_2187. 完成旅途的最少时间----------

Problem 2.2

给你一个数组 *time*，其中 *time[i]* 表示第 *i* 辆公交车完成一趟旅途所需要花费的时间。
每辆公交车可以连续完成多趟旅途，也就是说，一辆公交车当前旅途完成后，可以立马

开始下一趟旅途。每辆公交车独立运行，也就是说可以同时有多辆公交车在运行且互不影响。

给你一个整数 *totalTrips*，表示所有公交车总共需要完成的旅途数目。请你返回完成至少 *totalTrips* 趟旅途需要花费的最少时间。

2.2.1 解法-二分答案求最小

假设需要花费的时间为 t。根据题目要求，我们需要找到最小的满足 $\sum_{i=0}^{n-1}\left\lfloor \frac{time[i]}{t} \right\rfloor$ >= titalTrips 的 t。

同样首先确定 left 和 right 的初始值：

left ：可以设置为 1，因为显然时间应该是一个正整数。t <= 0 是一定不满足题目要求的区间。

right ：可以设置为 totalTrips * min(time)。这个表达式意味着在这个时间 t 的要求下速度最快的公交车已经可以独自完成所有旅程。大于这个值的是一定满足题目条件的区间。

可以更进一步优化 left 和 right 的选择：

left ：假设所有的公交车的速度都是数组中最快的即 1/min(time)，最小时间应该使得这些速度最大化的公交车能够完成 totalTrips 趟旅途。即 (t * 1/min(time)) * n >= totalTrips。据此可以得到 left 的初始值：t_{min}>= $\left\lceil \frac{totalTrips}{n} \right\rceil$ * min(time)。显然小于这个值的是一定不满足题目条件的区间。

right ：假设所有的公交车的速度都是数组中最慢的即 1/max(time)，最大时间应该使得这些速度最小化的公交车能够完成 totalTrips 趟旅途，即 t_{max}>= $\left\lceil \frac{totalTrips}{n} \right\rceil$ * max(time)。同样的，大于这个值的是一定满足题目条件的区间。

总而言之：

- left 的初始值为 $\max\left(1, \left\lceil \frac{totalTrips}{n} \right\rceil * min(time)\right)$。
- right 的初始值为 $\min\left(totalTrips * min(time), \left\lceil \frac{totalTrips}{n} \right\rceil * max(time)\right)$。

我们需要将其中的上取整改为下取整。由于当 a, b 均为正整数时，有：

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor = \left\lfloor \frac{a-1}{b} \right\rfloor + 1$$

所以：

$$\left\lceil \frac{totalTrips}{n} \right\rceil = \left\lfloor \frac{totalTrips-1}{n} \right\rfloor + 1$$

可以新建一个变量来储存这个值。

```python
def minimumTime(self, time: List[int], totalTrips: int) -> int:
    avg = (totalTrips - 1) // len(time) + 1
    minTime = min(time)

    left = max(1, avg * minTime)
    right = min(totalTrips * minTime, avg * max(time))

    while left <= right:
        mid = (left + right) // 2

        if sum(mid // x for x in time) >= totalTrips:  # 大于totalTrips意味着t的设置过大了，所以应该减少right
            right = mid - 1
        else:
            left = mid + 1

    return left
```

2.3 medium_1011. 在 D 天内送达包裹的能力----------

Problem 2.3

> 传送带上的包裹必须在 *days* 天内从一个港口运送到另一个港口。
>
> 传送带上的第 *i* 个包裹的重量为 *weights[i]*。每一天，我们都会按给出重量（*weights*）的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。
>
> 返回能在 *days* 天内将传送带上的所有包裹送达的船的最低运载能力。

2.3.1 解法-二分答案求最小

首先需要定义一个判断在某一运载能力下需要多少天将所有包裹送达的 check 函数：

```python
def howManyDays(weights: List[int], capacity: int) -> int:
    current_weight = 0  # 当前已装载的重量
    days_needed = 0  # 需要的天数

    for weight in weights:
        if weight + current_weight > capacity: # 如果当前装载重量达到最大，则重置当前装载重量并增加天数
            current_weight = weight
            days_needed += 1
        else:
            current_weight += weight  # 继续装载
    return days_needed + 1  # 最后一天也需要计入
```

假设运载能力为 w。根据题目要求，我们需要找到最小的满足 howManyDays(weights, w) <= days 的 w。

同样首先确定 left 和 right 的初始值：

left ：可以设置为 1，因为显然承载能力应该是一个正整数。w <= 0 是一定不满足题目要求的区间。更进一步的，同样不满足题目要求的区间还有 w <= max(weights)，因为这意味

着重量最大的包裹永远无法被送上船。不满足题目要求的区间还有 `w < sum(weights) // days`，因为这意味着运力完全没有被浪费的理想情况。

`right` ：可以设置为 `sum(weights)`。这会使得所有包裹在 1 天内被全部送上船，大于这个值的是一定满足题目条件的区间。

总而言之：

- `left` 的初始值为 `max(max(weights),sum(weights) // days)`。
- `right` 的初始值为 `sum(weights)`。

```python
def shipWithinDays(self, weights: List[int], days: int) -> int:
    def howManyDays(weights: List[int], capacity: int) -> int:
        current_weight = 0
        days_needed = 0

        for weight in weights:
            if weight + current_weight > capacity:
                current_weight = weight
                days_needed += 1
            else:
                current_weight += weight
        return days_needed + 1

    max_weight = max(weights)
    sum_weight = sum(weights)
    left = max(max_weight, sum_weight // days)
    right = sum_weight

    while left <= right:
        middle = (left + right) // 2

        if howManyDays(weights, middle) <= days: # 实际的天数少于要求的天数，说明weight设置过大了，需要减小
            right = middle - 1
        else:
            left = middle + 1

    return left
```

2.4 medium_475. 供暖器----------

Problem 2.4

冬季已经来临。你的任务是设计一个有固定加热半径的供暖器向所有房屋供暖。

在加热器的加热半径范围内的每个房屋都可以获得供暖。

现在，给出位于一条水平线上的房屋 *houses* 和供暖器 *heaters* 的位置，请你找出并返回可以覆盖所有房屋的最小加热半径。

注意：所有供暖器 *heaters* 都遵循你的半径标准，加热的半径也一样。

2.4.1 解法 1-指针与二分答案求最小

首先需要定义一个判断所有房屋是否都至少在一个供暖器的范围内的 check 函数。这里可以仿照 easy_1385 的做法，利用一个指针进行判断。

```python
def canCoverAll(houses: List[int], heaters: List[int], r: int) -> int:
    i = 0 # 指针
    for house in houses:
        while i < len(heaters) and heaters[i] < house - r:
            i += 1
        if i >= len(heaters) or heaters[i] > house + r:
            return False
    return True
```

这里 i 指针维护满足 heaters[i] < house - r 的数的下标。如果在下一次循环中发现 heaters[i + 1] - r > house,则 [hourse - r, house + r] 必然在 (heaters[i], heaters[i + 1]) 中。反过来，这表明 heaters[i] + r < house < heaters[i + 1] - r，即该房屋没有被供暖器覆盖。还有一种情况是在下一次遍历时 i >= len(heaters)，这说明该房屋在 heaters[-1] + r 的右侧，也即没有被供暖器覆盖。

注意在调用这个函数前需要先用.sort() 方法。以及这个 check 函数单次调用的复杂度是 $O(nm)$ 因为需要分别遍历 houses 和 heaters。

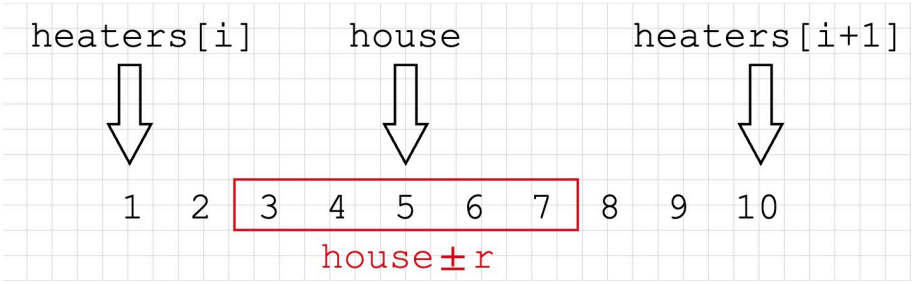

Figure. 3. 指针条件示意

假设供暖器的覆盖半径为 r。根据题目要求,我们需要找到最小的满足 canCoverAll(houses, heaters, r) == True 的 r。

确定 left 和 right 的初始值:

left ：可以设置为 0，因为覆盖半径一定是一个非负整数。(注意不一定是正整数，可能是 0)

right ：可以设置为 max(max(heaters) - min(houses), max(houses) - min(heaters))。大于这个值的是一定满足题目条件的区间，因为距离 heater 最远的 house 也可以被覆盖。

此时没有较好的优化方法。

```python
def findRadius(self, houses: List[int], heaters: List[int]) -> int:
    # check函数
    def canCoverAll(houses, heaters, r):
        i = 0
        for house in houses:
```

```
            while i < len(heaters) and heaters[i] + r < house:
                i += 1
            if i >= len(heaters) or heaters[i] - r > house:
                return False
        return True

    # 注意如果要用check函数中的指针方法需要先排序
    heaters.sort()
    houses.sort()

    left = 0
    right = max(houses[-1] - heaters[0], heaters[-1] - houses[0]) # 排序之后可以简化max()和min()

    while left <= right:
        mid = (left + right) // 2
        if canCoverAll(houses, heaters, mid):
            right = mid - 1
        else:
            left = mid + 1

    return left
```

此时的时间复杂度为 $n\log n + m\log m + nm\log(\texttt{right}-\texttt{left}+1)$。其中 $n\log n$ 和 $m\log m$ 是排序算法的时间复杂度，$nm\log(\texttt{right}-\texttt{left}+1)$ 是在二分查找中同时调用 `check` 函数的时间复杂度。

2.4.2 解法 2-二分查找

这种解法不需要定义 `check` 函数，而是将供暖器数组 `heaters` 排序，然后通过二分查找得到距离房屋 `houses[i]` 最近的供暖器，并更新当前要求的最远供暖器覆盖半径。

具体来说，该解法在将供暖器数组排序后通过 `bisect_right(heaters, house[i])` 得到在该房屋右侧最近的供暖器的索引 `j`，并由此得到在该房屋左侧最近的供暖器的索引 `j - 1`。最后通过比较左右供暖器和房屋之间的距离得到该房屋对供暖器覆盖半径的要求。

```
def findRadius(self, houses: List[int], heaters: List[int]) -> int:
    ans = 0
    heaters.sort() # 注意由于需要进行二分查找所以需要先排序

    for house in houses:
        j = bisect_right(heaters, house)
        i = j - 1
        rightDistance = heaters[j] - house if j < len(heaters) else float('inf') # j=len(heaters)对应house大于所有heaters的情况
        leftDistance = house - heaters[i] if i >= 0 else float('inf') # 当j取0的时候i是-1，对应house小于所有heaters的情况
        currentRadius = min(leftDistance, rightDistance) # 要求的供暖器覆盖半径是和距离较近的供暖器的距离
        ans = max(ans, currentRadius)
    return ans
```

2.5 medium_3296. 移山所需的最少秒数----------

Problem 2.5

给你一个整数 *mountainHeight* 表示山的高度。

同时给你一个整数数组 *workerTimes*，表示工人们的工作时间（单位：秒）。

工人们需要同时进行工作以降低山的高度。对于工人 *i*:

山的高度降低 *x*,需要花费 *workerTimes[i] + workerTimes[i] * 2 +...+ workerTimes[i] * x* 秒。例如:

山的高度降低 1，需要 *workerTimes[i]* 秒。

山的高度降低 2，需要 *workerTimes[i] + workerTimes[i] * 2* 秒，依此类推。

返回一个整数，表示工人们使山的高度降低到 0 所需的最少秒数。

2.5.1 解法-二分答案求最小

首先需要定义一个计算 `timeLimit` 秒内工人们可以降低山的总高度的 `check` 函数。第一种方法是利用 `while` 在 `timeUsed` 小于 `timeLimit` 前累加降低山的高度，并对每个工人分别进行这种操作。这种方法的时间复杂度大约是 $O(nt)$（n 是数组 `workerTimes` 的长度）。第二种方法是数学推导第 `i` 名工人可以降低的山的高度和 `timeUsed` 以及 `workerTimes[i]` 的关系。这种方法的时间复杂度是 $O(n)$。这里详细阐述第二种方法。

设 `time_needed = workerTimes[i]`，则一定有:

$$\texttt{time\_needed} + \texttt{time\_needed} * 2 + \cdots + \texttt{time\_needed} * \texttt{heightReduced} \texttt{ <= timeLimit}$$

该式可以化简为:

$$\texttt{time\_needed} * \frac{\texttt{heightReduced}(\texttt{heightReduced} + 1)}{2} \texttt{ <= timeLimit}$$

所以 `timeLimit` 下第 `i` 个工人可以降低的山的高度为:

$$\texttt{heightReduced} <= \left\lfloor \frac{-1 + \sqrt{1 + 8 \left\lfloor \frac{\texttt{timeLimit}}{\texttt{time\_needed}} \right\rfloor}}{2} \right\rfloor$$

`check` 函数可以累加所有工人的 `heightReduced` 并用于在二分查找时和 `moutainHeight` 对比:

```python
def check(moutainHeight: int, workerTimes: List[int], timeLimit: int) -> bool:
    height = moutainHeight

    for time_needed in workerTimes:
        height -= (isqrt(timeLimit // time_needed * 8 + 1) // 2)

    if height <= 0:
        return True

    return False
```

编写一个等差数列求和函数以方便后续计算：

```
def getSum(num: int) -> int:
    return num * (num + 1) //2
```

接下来确定 left 与 right 的初始值：

left ：可以设置为 1，因为 timeLimit 应该是一个正整数。更进一步的，假设所有工人的施工速度都是数组中最快的即 min(workerTimes)，那么每个工人需要施工的高度为 $\left\lceil \mathtt{h} = \frac{\mathtt{mountainHeight}}{\mathtt{n}} \right\rceil$，最短施工时间为 min(workerTimes) * (1+2+···+h) = min(workerTimes) * getSum(h)

right ：假设所有工人施工速度都是数组中最慢的即 max(workerTimes) 可以最长施工时间为 max(workerTimes) * getSum(h)。同时，最长时间还可以是施工速度最快的工人独自施工完所有高度的时间，即 min(workerTimes) * getSum(mountainHeight)。

总而言之：

- left 的初始值为 max(1, min(workerTimes) * getSum(h))。
- right 的初始值为 min(min(workerTimes) * getSum(mountainHeight), max(workerTimes) * getSum(h))。

```
def minNumberOfSeconds(self, mountainHeight: int, workerTimes: List[int]) -> int:
    # check函数
    def check(mountainHeight: int, workerTimes: List[int], timeLimit: int) -> bool:
        height = mountainHeight

        for time_needed in workerTimes:
            height -= (isqrt(timeLimit // time_needed * 8 + 1) - 1) // 2

        if height <= 0:
            return True

        return False

    # 辅助求和函数
    def getSum(num: int) -> int:
        return num * (num + 1) // 2

    minWorkerTime = min(workerTimes)
    maxWorkerTime = max(workerTimes)
    h = (mountainHeight - 1) // len(workerTimes) + 1
    getSum_h = getSum(h)

    left = max(1, minWorkerTime * getSum_h)
    right = min(minWorkerTime * getSum(mountainHeight), maxWorkerTime * getSum_h)

    while left <= right:
        print(left, right)
        mid = (left + right) // 2

        if check(mountainHeight, workerTimes, mid):
```

```
            right = mid - 1
        else:
            left = mid + 1

    return left
```

3 二分答案求最大

二分答案求最大和二分答案求最小的解题思路几乎一致，都是在确定 left 与 right 的初始值和更新 middle 时的条件判断语句/函数之后对所求值进行二分

二分答案求最大和二分答案求最小的区别在于：1.while 循环中更新 left 与 right 值的逻辑不同。2. 最后返回的值是 right 而不是 left。

一个典型的二分答案求最大的代码为：

```
while left <= right:
    mid = (left + right) // 2

    if check(mid):
        left = mid + 1
    else:
        right = mid - 1

return right
```

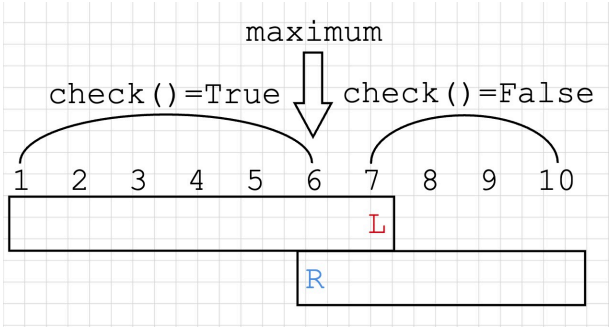

Figure. 4. 二分答案求最大

如图所示，在满足 check 函数的时候应该移动左指针，在不满足的时候应该移动右指针。

二分答案求最大的题目除了一些基础题目如 **medium_2226 每个小孩最多能分到多少糖果**在 check 函数的编写上比较简单之外，其余题目的 check 函数编写都有明显难度。

3.1 medium_2226. 每个小孩最多能分到多少糖果----------

Problem 3.1

> 给你一个下标从 0 开始的整数数组 *candies*。数组中的每个元素表示大小为 *candies[i]* 的一堆糖果。你可以将每堆糖果分成任意数量的子堆，但无法再将两堆合并到一起。

另给你一个整数 k。你需要将这些糖果分配给 k 个小孩，使每个小孩分到相同数量的糖果。每个小孩可以拿走至多一堆糖果，有些糖果可能会不被分配。

返回每个小孩可以拿走的最大糖果数目。

3.1.1 解法-二分答案求最大

假设每个小孩拿走的糖果数目为 n。根据题目要求,我们需要找到最大的满足 $\sum_{i=0}^{n-1} \left\lfloor \frac{\text{candies[i]}}{\text{n}} \right\rfloor$ >= k 的 n。

接下来确定 left 和 right 的初始值:

left ：可以设置为 1，因为除非 sum(candies) < k 否则显然每个孩子至少可以被分到 1 颗糖。

right ：可以设置为 max(candies)。因为糖果堆无法合并，所以 max(candies) 是所有孩子有可能获得的最多糖果。

```python
def maximumCandies(self, candies: List[int], k: int) -> int:
    if sum(candies) < k: # 如果糖果总数小于孩子的数量
        return 0

    left = 1
    right = max(candies)

    while left <= right:
        mid = (left + right) // 2

        if sum(x // mid for x in candies) >= k: # 如果糖果堆数大于人数则需要增大每个孩子拿走的糖果数目
            left = mid + 1
        else:
            right = mid - 1

    return right # 注意因为是二分答案求最大所以返回的是right而不是left
```

3.2 medium_2982. 找出出现至少三次的最长特殊子字符串 II----------

Problem 3.2

给你一个仅由小写英文字母组成的字符串 s。

如果一个字符串仅由单一字符组成，那么它被称为特殊字符串。例如，字符串 *"abc"* 不是特殊字符串，而字符串 *"ddd"*、*"zz"* 和 *"f"* 是特殊字符串。

返回在 s 中出现至少三次的最长特殊子字符串的长度，如果不存在出现至少三次的特殊子字符串，则返回 *-1*。

子字符串是字符串中的一个连续非空字符序列。

3.2.1 解法 1-二分答案求最大

首先编写一个函数用于获得所有字母的连续字符串的长度，随后对每个字母的连续字符串的长度进行排序并利用二分答案求最大得到该字母的最大的出现 3 次的子字符串长度。

首先编写获得所有字母的连续字符串的长度的函数：

```
def getSuccessiveStringLength(arr: str) -> List[List[int]]:  # 定义一个函数用于获得字符串中每个字母连续字符串的长度
    i, str_length = 0, len(arr)
    char_groups = [[] for _ in range(26)]  # 创建了一个包含 26 个空列表的列表，用于存储每个字母的连续长度
    while i < str_length:
        current_length = 0  # 一个字母的连续字符串的长度
        while i + current_length < str_length and arr[i] == arr[
            i + current_length]:  # 增加current_length的值直到不再是同一个字母
            current_length += 1
        char_groups[ord(arr[i]) - 97].append(current_length) # 将字母a-z映射为0-25以存储
        i += current_length # 跳到下一个字母处
    return char_groups
```

主程序编写如下：

```
def maximumLength(self, s: str, min_substrings: int = 3) -> int:
    max_length = 0
    groups = getSuccessiveStringLength(s) # 调用刚才的函数
    for group in groups:
        if not group: # 如果group为空则跳过
            continue
        group.sort(reverse=True) # 从大到小排序
        group = group[:min_substrings] # 只保留3个最大的连续长度，因为最长的3个连续字符串一定可以产生较长的3个子串

        # 二分查找最大的子串长度
        left = 0
        right = group[0] # 最长的子串的长度

        while left <= right:
            mid = (left + right) // 2

            valid_substrings = sum(max(0, length - mid + 1) for length in group) # 计算在mid作为最长的子串的长度时总的子串数量

            if valid_substrings >= min_substrings:
                left = mid + 1
            else:
                right = mid - 1

        max_length = max(max_length, right) # 注意要选出长度最长的

    return max_length if max_length else -1
```

3.2.2 解法 2-数学推导

在保留 3 个最大的连续长度之后，我们并不着急进行二分查找，而是直接数学推导最长的子串长度。

一共有以下这几种取出子串的方式：

1. 从最长的连续字符串 group[0] 中取出长度为 group[0] - 2 的子串。
2. 从最长和次长的连续字符串中取出三个子串。这里有两种取法。当 group[0] = group[1] 的时候，一定可以取出三个长度为 group[0] - 1 的子串。而当 group[0] >= group[1] + 1 的时候，可以取出长度为 group[1] 的子串（从最长中取 2 个，从次长中取 1 个）。
3. 从最长、次长和第三长的连续字符串中取出三个长度为 group[2] 的子串。

修改之前的代码：

```python
def maximumLength(self, s: str, min_substrings: int = 3) -> int:
    max_length = 0
    groups = getSuccessiveStringLength(s) # 调用刚才的函数
    for group in groups:
        if not group:
            continue
        group.sort(reverse=True) # 从大到小排序
        group = group[:min_substrings] # 只保留3个最大的连续长度

        # 注意该字母不一定有3个连续字符串
        if len(group) == 3:
            if len(group) >= 3 and group[0] == group[1]:
                current_max_length = max(group[0] - 1, group[2])
            else:
                current_max_length = max(group[0] - 2, group[1])
        elif len(group) == 2:
            if group[0] == group[1]:
                current_max_length = group[0] - 1
            else:
                current_max_length = max(group[0] - 2, group[1])
        else:
            current_max_length = group[0] - 2

        max_length = max(max_length, current_max_length) # 注意要选出长度最长的

    return max_length if max_length else -1
```

4 二分答案-最小化最大值

这种题目要求某一结果的最大值最小，其本质上是在求该最大值的最小值。因此我们对该最大值进行二分答案操作，找到满足 check 函数的最小值即可。

一种很典型的此类题目是要求将一个数组拆分成最大值有某一限制条件的 n 个子数组，对于此类题目，可以二分该最大值，在 check 函数中判断拆分的子数组的数量是否满足题目条件

(通常是 <= n)。**medium_1760 袋子里最少数目的球**、**hard_410 分割数组的最大值**、**medium_2064 分配给商店的最多商品的最小值**都是此类题目。

其他类型的题目的思路更加常规，都是二分答案，然后根据题意编写各式 check 函数。这类题目的难点和二分答案的难点类似，都是在 check 函数的编写方式上。**medium_2439 最小化数组中的最大值**是此类题目的一个代表。

4.1 medium_1760. 袋子里最少数目的球----------

Problem 4.1

> 给你一个整数数组 *nums*，其中 *nums[i]* 表示第 *i* 个袋子里球的数目。同时给你一个整数 *maxOperations*。
>
> 你可以进行如下操作至多 *maxOperations* 次：
>
> 选择任意一个袋子，并将袋子里的球分到 *2* 个新的袋子中，每个袋子里都有正整数个球。
>
> 比方说，一个袋子里有 *5* 个球，你可以把它们分到两个新袋子里，分别有 *1* 个和 *4* 个球，或者分别有 *2* 个和 *3* 个球。
>
> 你的开销是单个袋子里球数目的最大值，你想要最小化开销。
>
> 请你返回进行上述操作后的最小开销。

4.1.1 解法-二分答案求最小

要求袋子中球的数目的最大值的最小值，我们直接二分该最大值，随后通过 check 函数断是否可以在小于等于 maxOperations 次的操作中让所有袋子中的球的数量都小于这个最大值。

假设这个最大值为 k,则对于第 i 个袋子 nums[i],其需要 $\left\lceil \frac{\texttt{nums[i]}}{\texttt{k}} \right\rceil - 1 = \left\lfloor \frac{\texttt{nums[i]} - 1}{\texttt{k}} \right\rfloor$ 次操作才能分为 $\left\lceil \frac{\texttt{nums[i]}}{\texttt{k}} \right\rceil$ 份。所以所有袋子共需要 $\sum_{\texttt{i}=0}^{\texttt{len(nums)}} \left\lfloor \frac{\texttt{nums[i]} - 1}{\texttt{k}} \right\rfloor$ 次操作。这给出了我们的 check 函数：

```python
def check(k: int) -> bool:
    return sum((x - 1) // k for x in nums) <= maxOperations # 要求至多进行maxOperations次操作
```

接下来确定 left 和 right 的初始值。注意这里我们二分的是袋子中球的数目的最大值。

left ：可以设置为 1，每个袋子中至少应该有 1 个球。

right ：可以设置为 max(nums),因为当 maxOperations == 1 时的最大值最好设置为 max(nums)。

当然也可以用 $O(n \log n)$ 的时间复杂度对比 $\left\lceil \frac{\texttt{max(nums)}}{2} \right\rceil$ 和数组中第二大的数。

```python
def minimumSize(self, nums: List[int], maxOperations: int) -> int:
    left = 1
    right = max(nums)

    while left <= right:
        mid = (left + right) // 2

        if sum((x - 1) // mid for x in nums) <= maxOperations:
            right = mid - 1 # 如果操作次数小于maxOperations，说明袋子中球的数目的最大值太大了，需要减小
        else:
            left = mid + 1

    return left
```

4.2 medium_2064. 分配给商店的最多商品的最小值----------

Problem 4.2

给你一个整数 *n*，表示有 *n* 间零售商店。总共有 *m* 种产品，每种产品的数目用一个下标从 *0* 开始的整数数组 *quantities* 表示，其中 *quantities[i]* 表示第 *i* 种商品的数目。

你需要将所有商品分配到零售商店，并遵守这些规则：

一间商店至多只能有一种商品，但一间商店拥有的商品数目可以为任意件。

分配后，每间商店都会被分配一定数目的商品（可能为 *0* 件）。用 *x* 表示所有商店中分配商品数目的最大值，你希望 *x* 越小越好。也就是说，你想最小化分配给任意商店商品数目的最大值。

请你返回最小的可能的 *x*。

4.2.1 解法-二分答案求最小

要求分配商品数目的最大值的最小值，我们可以直接二分该最大值，随后通过 check 函数判断在该最大值下商品数量拆分的份数是否小于等于商店的数量。为什么不是判断拆分的份数是否**等于**商店的数量呢？这是因为题目中指出了商店可以被分配 0 件商品。所以拆分的份数可以小于等于商店的数量但是一定不能大于商店的数量。

quantities = [5,6]	n = 5(number_of_stores)	
maximum_amount	stores	len(stores)
2	[2,2,1,2,2,2]	6
3	[3,2,3,3]	4
4	[4,1,4,2]	4
5	[5,5,1]	3
6	[5,6]	2

Figure. 5. 当被拆分的份数小于商店的数量时只需要给剩余的商店分配 0 件商品即可

假设这个最大值为 k，则对于第 i 个商品，其可以拆分为 $\left\lceil \frac{\text{quantities[i]}}{\text{k}} \right\rceil$ 份，这对应了同等数量的商店。所以我们需要 $\sum_{\text{i=0}}^{\text{len(quan-)}} \left\lceil \frac{\text{quantities[i]}}{\text{k}} \right\rceil$ <= n，这给出了我们的 check 函数：

```
def check(maximum_amount: int) -> bool:
    return sum((x - 1) // maximum_amount + 1 for x in quantities) <= n
```

接下来确定 left 和 right 的初始值。

left ：可以设置为 1，只要 quantities 不是全为 0 就必然至少给一个商店分配数量为 1 的商品。

right ：可以设置为 max(quantities)，因为一个商店可能需要承载的最大商品数量为 max(quantities)。

```
def minimizedMaximum(self, n: int, quantities: List[int]) -> int:
    left = 1
    right = max(quantities)

    while left <= right:
        mid = (left + right) // 2

        if sum((x - 1) // mid + 1 for x in quantities) <= n: # 省略check函数
            right = mid - 1 # 小于等于商店数量说明设定的分配数量最大值太大，需要减小
        else:
            left = mid + 1

    return left
```

4.3 medium_2560. 打家劫舍----------

Problem 4.3

> 沿街有一排连续的房屋。每间房屋内都藏有一定的现金。现在有一位小偷计划从这些房屋中窃取现金。
>
> 由于相邻的房屋装有相互连通的防盗系统，所以小偷不会窃取相邻的房屋。
>
> 小偷的窃取能力定义为他在窃取过程中能从单间房屋中窃取的最大金额。
>
> 给你一个整数数组 *nums* 表示每间房屋存放的现金金额。形式上，从左起第 *i* 间房屋中放有 *nums[i]* 美元。
>
> 另给你一个整数 *k*，表示窃贼将会窃取的最少房屋数。小偷总能窃取至少 *k* 间房屋。
>
> 返回小偷的最小窃取能力。

4.3.1 解法-二分答案求最小

要求小偷的最小窃取能力，我们直接对该值进行二分，随后通过 check 函数判断能否窃取至少 k 间房屋。这里我们先定义 check 函数：

```python
def check(maximum_amount: int, nums: List[int], k: int) -> bool:
    count = 0 # 窃取的房屋数
    visited = False # 左侧相邻的房屋是否被窃取
    for num in nums:
        if num <= maximum_amount and (not visited): # 不能窃取相邻的房屋
            count += 1
            visited = True
        else:
            visited = False # 如果最大金额大于窃取能力则无法窃取
    return count >= k
```

接下来确定 left 和 right 的初始值。

left ：可以设置为 min(nums)，因为题目给出窃取的房屋数 k 大于等于 1，这意味着小偷至少应该能够窃取大小为 min(nums) 的金额。

right ：可以设置为 max(nums)。

```python
def minCapability(self, nums: List[int], k: int) -> int:
    # check函数
    def check(maximum_amount: int, nums: List[int], k: int) -> bool:
        count = 0 # 窃取的房屋数
        visited = False # 左侧相邻的房屋是否被窃取
        for num in nums:
            if num <= maximum_amount and (not visited): # 不能窃取相邻的房屋
                count += 1
                visited = True
            else:
                visited = False # 如果最大金额大于窃取能力则无法窃取
        return count >= k

    left = min(nums)
    right = max(nums)

    while left <= right:
        mid = (left + right) // 2

        if check(mid, nums, k):
            right = mid - 1 # 如果满足题目条件即可以窃取至少k间房屋，则尝试更小的值
        else:
            left = mid + 1

    return left
```

5 二分答案-最大化最小值

这种题目要求某一结果的最小值最大，其本质上是在求该最小值的最大值。因此我们对该最小值进行二分答案操作，找到满足 check 函数的最大值即可。