

九章算法班2022版直播第3章

简约而不简单 —— 二分法的四重境界

主讲：夏天

457 Classical Binary Search 经典二分查找问题

Find **any** position of a target number in a **sorted array**. Return -1 if target does not exist.

在一个**有序数组**中找一个数，返回该数出现的**任意**位置，如果不存在，返回 -1。

输入：

nums = [1,2,2,4,5,5]

target = 6

输出：

-1

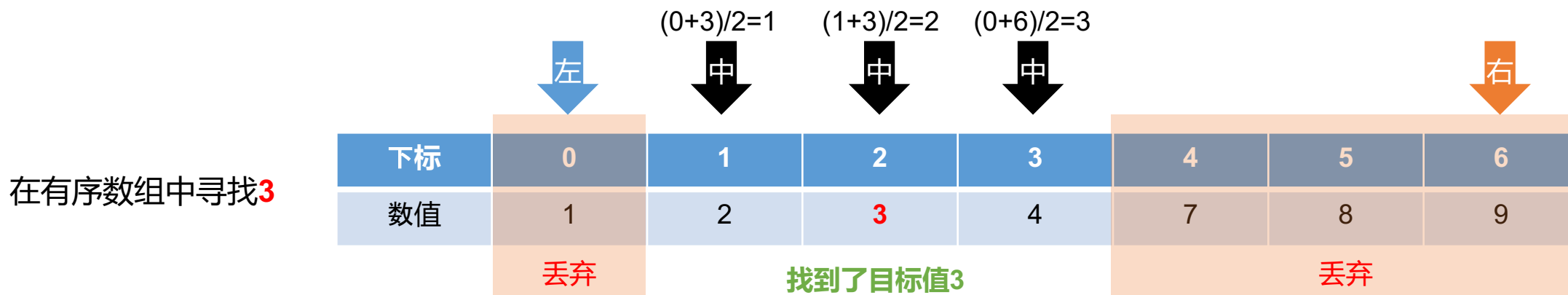
输入：

nums = [1,2,2,4,5,5]

target = 2

输出：

1 或者 2 (下标)



什么是二分法 Binary Search

二分法又叫**二分查找**，**折半查找**

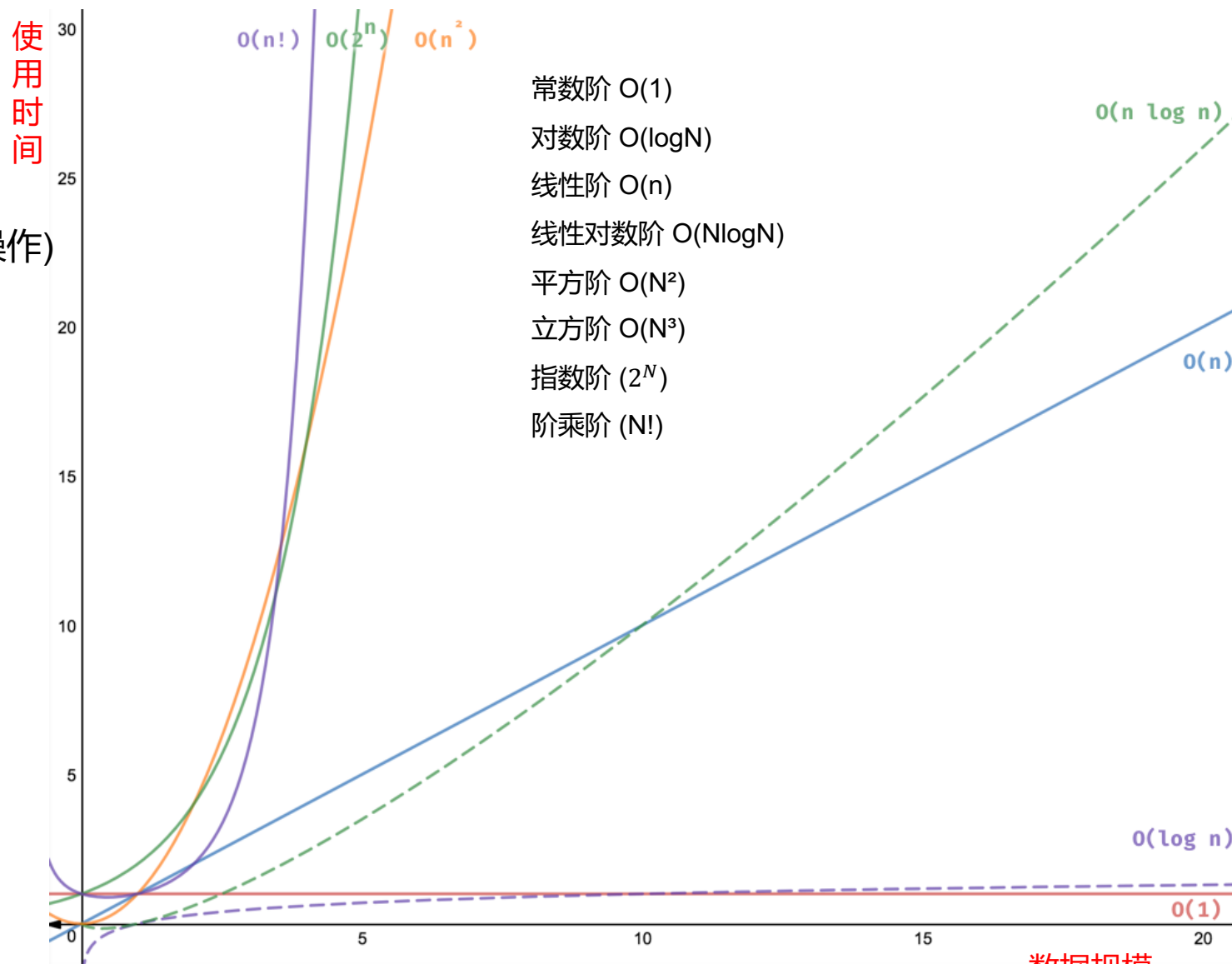
二分法是一种**高效**的查找算法, 时间复杂度为 $O(\log N)$ ，空间复杂度为 $O(1)$

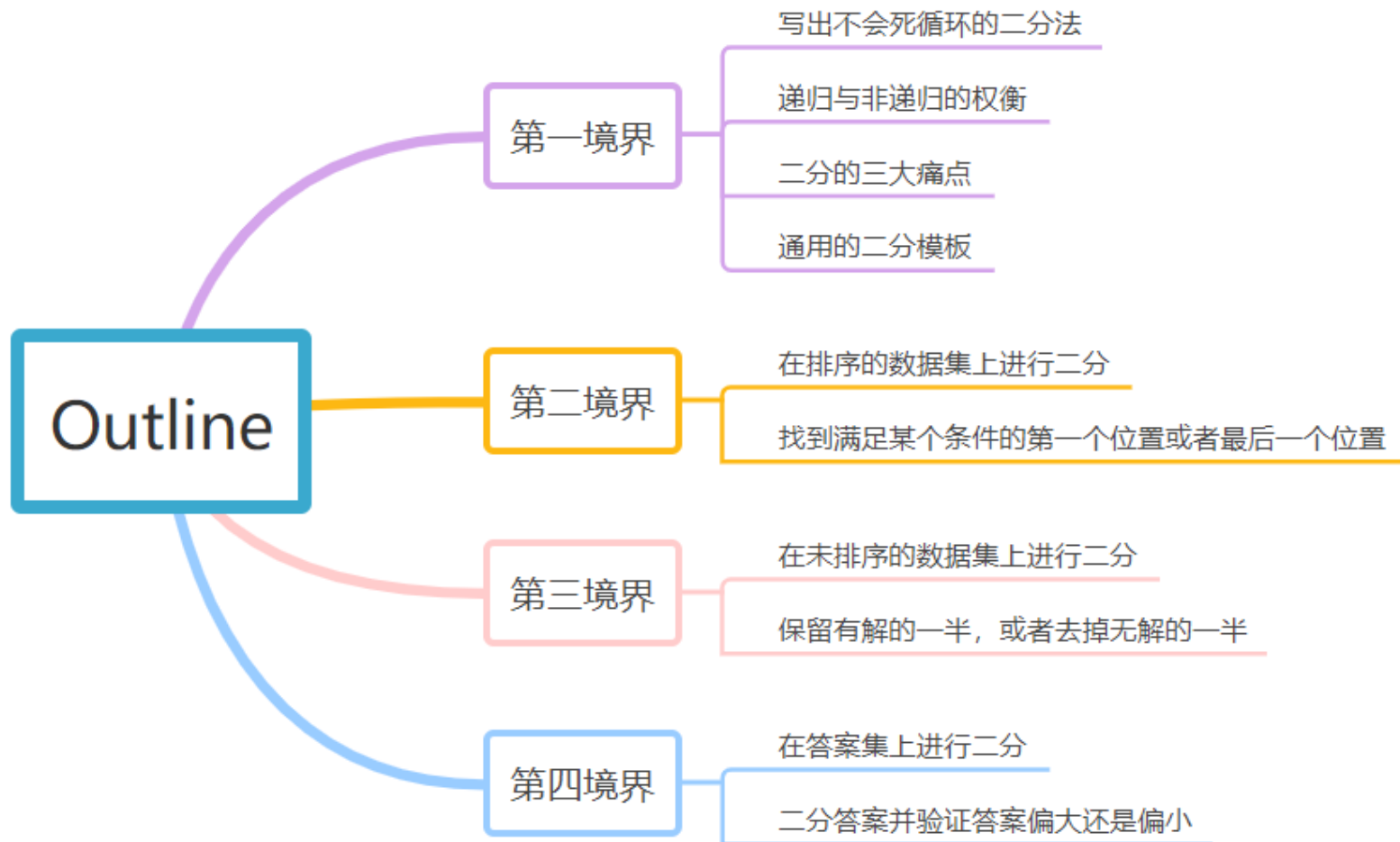
一般情况下，二分法用于在某个**有序**数据上寻找**target**值

二分法利用了**减治** (Decrease and Conquer) 的算法思想 (Algorithmic Paradigm)，不属于分治 (Divide and Conquer) 算法思想

常见的面试时间复杂度和对应的算法

- $O(\log N)$ **二分法**比较多
- $O(\sqrt{N})$ 分解质因数 (极少)
- $O(N)$ 双指针, 单调栈, 枚举法
- $O(N \log N)$ 排序, $O(N * \log N)$ 的数据结构上的操作)
- $O(N^2)$, $O(N^3)$, 动态规划等
- $O(2^N)$, 组合类 (combination) 的搜索问题
- $O(N!)$ 排列类 (permutation) 的搜索问题





第一境界 写出不会死循环的二分法

二分法模板 (必须背过)

```
def binarySearch(self, nums, target):
    if not nums:
        return -1

    start, end = 0, len(nums) - 1
    # 用 start + 1 < end 而不是 start < end 的目的是为了避免死循环
    # 在 first position of target 的情况下不会出现死循环
    # 但是在 last position of target 的情况下会出现死循环
    # 样例: nums=[1, 1] target = 1
    # 为了统一模板, 我们就都采用 start + 1 < end, 就保证不会出现死循环
    while start + 1 < end:
        # python 没有 overflow 的问题, 直接 // 2 就可以了
        # java和C++ 最好写成 mid = start + (end - start) / 2
        # 防止在 start = 2^31 - 1, end = 2^31 - 1 的情况下出现加法 overflow
        mid = (start + end) // 2

        # >, =, < 的逻辑先分开写, 然后在看看 = 的情况是否能合并到其他分支里
        if nums[mid] < target:
            # 写作 start = mid + 1 也是正确的
            # 只是可以偷懒不写, 因为不写也没问题, 不会影响时间复杂度
            # 不写的好处是, 万一你不小心写成了 mid - 1 你就错了
            start = mid
        elif nums[mid] == target:
            end = mid
        else:
            # 写作 end = mid - 1 也是正确的
            # 只是可以偷懒不写, 因为不写也没问题, 不会影响时间复杂度
            # 不写的好处是, 万一你不小心写成了 mid + 1 你就错了
            end = mid

    # 因为上面的循环退出条件是 start + 1 < end
    # 因此这里循环结束的时候, start 和 end 的关系是相邻关系 (1和2, 3和4这种)
    # 因此需要再单独判断 start 和 end 这两个数谁是我们的答案
    # 如果是找 first position of target 就先看 start, 否则就先看 end
    if nums[start] == target:
        return start
    if nums[end] == target:
        return end

    return -1
```

```
public int binarySearch(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int start = 0, end = nums.length - 1;
    // 用 start + 1 < end 而不是 start < end 的目的是为了避免死循环
    // 在 first position of target 的情况下不会出现死循环
    // 但是在 last position of target 的情况下会出现死循环
    // 样例: nums=[1, 1] target = 1
    // 为了统一模板, 我们就都采用 start + 1 < end, 就保证不会出现死循环
    while (start + 1 < end) {
        // python 没有 overflow 的问题, 直接 // 2 就可以了
        // java和C++ 最好写成 mid = start + (end - start) / 2
        // 防止在 start = 2^31 - 1, end = 2^31 - 1 的情况下出现加法 overflow
        int mid = start + (end - start) / 2;

        // >, =, < 的逻辑先分开写, 然后在看看 = 的情况是否能合并到其他分支里
        if (nums[mid] < target) {
            // 写作 start = mid + 1 也是正确的
            // 只是可以偷懒不写, 因为不写也没问题, 不会影响时间复杂度
            // 不写的好处是, 万一你不小心写成了 mid - 1 你就错了
            start = mid;
        } else if (nums[mid] == target) {
            end = mid;
        } else {
            // 写作 end = mid - 1 也是正确的
            // 只是可以偷懒不写, 因为不写也没问题, 不会影响时间复杂度
            // 不写的好处是, 万一你不小心写成了 mid + 1 你就错了
            end = mid;
        }
    }

    // 因为上面的循环退出条件是 start + 1 < end
    // 因此这里循环结束的时候, start 和 end 的关系是相邻关系 (1和2, 3和4这种)
    // 因此需要再单独判断 start 和 end 这两个数谁是我们的答案
    // 如果是找 first position of target 就先看 start, 否则就先看 end
    if (nums[start] == target) {
        return start;
    }
    if (nums[end] == target) {
        return end;
    }
    return -1;
}
```

[1, 1] 寻找最右1

```
7 def lastPosition(self, A, target):
8     if not A or target is None:
9         return -1
10
11     start = 0
12     end = len(A) - 1
13
14     while start < end:
15         mid = start + end // 2
16
17         if A[mid] <= target:
18             start = mid
19         elif A[mid] > target:
20             end = mid
21
22     if A[end] == target:
23         return end
24     else:
25         return -1
```

[1, 2] 寻找最右1

```
7 def lastPosition(self, A, target):
8     if not A or target is None:
9         return -1
10
11     start = 0
12     end = len(A) - 1
13
14     while start < end:
15         mid = start + end // 2
16
17         if A[mid] <= target:
18             start = mid + 1
19         elif A[mid] > target:
20             end = mid
21
22     if A[end] == target:
23         return end
24     else:
25         return -1
```


第二境界

在排序的数据集上进行二分

以target value = 5为例

| Index | Value | 二分问题 | |
|-------|-------|--------------------------------|--|
| 0 | 1 | | |
| 1 | 3 | | |
| 2 | 3 | | |
| 3 | 4 | 小于5的最大index(或value)， 插入5的index | |
| 4 | 5 | 大于等于5的最小index(或value) | 5出现的次数 = 等于5的最大index - 等于5的最小index + 1 |
| 5 | 5 | 任意一个出现5的index | |
| 6 | 5 | 小于等于5的最大index(或value) | |
| 7 | 7 | 大于5的最小index(或value)， 插入5的index | |
| 8 | 8 | | |
| 9 | 9 | | |

在**有序**的数组中寻找一个跟**target value**有关的**index**或**value**

447 Search in a Big Sorted Array

Given a big sorted array with non-negative integers sorted by non-decreasing order. **The array is so big** so that you can not get the length of the whole array directly, and you can only access the kth number by `ArrayReader.get(k)` (or `ArrayReader->get(k)` for C++).

Find the first index of a target number. Your algorithm should be in $O(\log k)$, where k is the first index of the target number.

Return -1, if the number doesn't exist in the array.

If you accessed an inaccessible index (outside of the array), `ArrayReader.get` will return 2,147,483,647.

给一个按照升序排序的非负整数数组。这个数组很大以至于你只能通过固定的接口 `ArrayReader.get(k)` 来访问第k个数 (或者C++里是`ArrayReader->get(k)`)，并且你也没有办法得知这个数组有多大。

找到给出的整数target第一次出现的位置。你的算法需要在 $O(\log k)$ 的时间复杂度内完成，k为target第一次出现的位置的下标。

如果找不到target，返回-1。

如果你访问了一个不可访问的下标（比如越界），`ArrayReader` 会返回2,147,483,647。

输入：

[1, 3, 6, 9, 21, ...]

target = 3

输出：

1

输入：

[1, 3, 6, 9, 21, ...]

target = 4

输出：

-1

倍增法 Exponential Backoff

寻找第一个9

reader.get(1 - 1)

| 下标 | 0 |
|----|---|
| 数值 | 1 |

reader.get(2 - 1)

| 下标 | 0 | 1 |
|----|---|---|
| 数值 | 1 | 3 |

reader.get(4 - 1)

| 下标 | 0 | 1 | 2 | 3 |
|----|---|---|---|---|
| 数值 | 1 | 3 | 3 | 5 |

reader.get(8 - 1)

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|---|---|---|---|---|---|---|---|
| 数值 | 1 | 3 | 3 | 5 | 6 | 6 | 6 | 7 |

reader.get(16 - 1)

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 数值 | 1 | 3 | 3 | 5 | 6 | 6 | 6 | 7 | 9 | 9 | 9 | 10 | 11 | 11 | 13 | 25 |

使用到倍增思想的场景：

- 动态数组 (List in Python, ArrayList in Java, vector in C++)
- 网络重试

寻找第一个9

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 数值 | 1 | 3 | 3 | 5 | 6 | 6 | 6 | 7 | 9 | 9 | 9 | 10 | 11 | 11 | 13 | 25 |

```
7 def searchBigSortedArray(self, reader, target):
8     # 初始化查找范围为1, 代表在前1个数中查找
9     range_total = 1
10    # 倍增法: 如果target在查找范围之外, 查找范围翻倍, 时间复杂度O(logK)
11    while reader.get(range_total - 1) < target:
12        range_total = range_total * 2
13
14    # start 也可以是 range_total // 2, 但是我习惯比较保守的写法
15    # 因为写为 0 也不会影响时间复杂度
16    # 如果target前rangeTotal中, 则index范围为 [0, rangeTotal - 1]
17    # 时间复杂度O(logK)
18    start, end = 0, range_total - 1
19    while start + 1 < end:
20        mid = start + (end - start) // 2
21        # 如果(中点值 < target), target在右边, 丢弃左边
22        if reader.get(mid) < target:
23            start = mid
24        # 如果(target <= 中点值), 丢弃右边, 去左边
25        # 要点: 为什么这里(target == 中点值), 不直接返回?
26        # 因为在中点左边还可能存在更靠左的target值
27        else:
28            end = mid
29
30    # 要点: 为什么这里先检查start, 再检查end?
31    # 因为我们要求的是first index, 所以先检查更靠左的start
32    # 如果start不为target, 再检查稍微靠右的end
33    # 比如, 在[9, 9]中寻找9, 应返回0 (第一个9的index)
34    if reader.get(start) == target:
35        return start
36    if reader.get(end) == target:
37        return end
38    return -1
```

```
7 public int searchBigSortedArray(ArrayReader reader, int target) {
8     // 初始化查找范围为1, 代表在前1个数中查找
9     int rangeTotal = 1;
10    // 倍增法: 如果target在查找范围之外, 查找范围翻倍, 时间复杂度O(logK)
11    while (reader.get(rangeTotal - 1) < target) {
12        rangeTotal = rangeTotal * 2;
13    }
14
15    // start 也可以是 rangeTotal / 2, 但是我习惯比较保守的写法
16    // 因为写为 0 也不会影响时间复杂度
17    // 如果target前rangeTotal中, 则index范围为 [0, rangeTotal - 1]
18    // 时间复杂度O(logK)
19    int start = 0, end = rangeTotal - 1;
20    while (start + 1 < end) {
21        int mid = start + (end - start) / 2;
22        // 如果(中点值 < target), target在右边, 丢弃左边
23        if (reader.get(mid) < target) {
24            start = mid;
25        }
26        // 如果(target <= 中点值), 丢弃右边, 去左边
27        // 要点: 为什么这里(target == 中点值), 不直接返回?
28        // 因为在中点左边还可能存在更靠左的target值
29        else {
30            end = mid;
31        }
32    }
33
34    // 要点: 为什么这里先检查start, 再检查end?
35    // 因为我们要求的是first index, 所以先检查更靠左的start
36    // 如果start不为target, 再检查稍微靠右的end
37    // 比如, 在[9, 9]中寻找9, 应返回0 (第一个9的index)
38    if (reader.get(start) == target) {
39        return start;
40    }
41    if (reader.get(end) == target) {
42        return end;
43    }
44    // 找不到, 返回-1, 具体返回值需要跟面试官核实
45    return -1;
}
```

寻找第一个9

| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 数值 | 1 | 3 | 3 | 5 | 6 | 6 | 6 | 7 | 9 | 9 | 9 | 10 | 11 | 11 | 13 | 25 |

```
7 def searchBigSortedArray(self, reader, target):
8     # 初始化查找范围为1, 代表在前1个数中查找
9     range_total = 1
10    # 倍增法: 如果target在查找范围之外, 查找范围翻倍, 时间复杂度O(logK)
11    while reader.get(range_total - 1) < target:
12        range_total = range_total * 2
13
14    # start 也可以是 range_total // 2, 但是我习惯比较保守的写法
15    # 因为写为 0 也不会影响时间复杂度
16    # 如果target前rangeTotal中, 则index范围为[0, rangeTotal - 1]
17    # 时间复杂度O(logK)
18    start, end = 0, range_total - 1
19    while start + 1 < end:
20        mid = start + (end - start) // 2
21        # 如果(中点值 < target), target在右边, 丢弃左边
22        if reader.get(mid) < target:
23            start = mid
24        # 如果(target <= 中点值), 丢弃右边, 去左边
25        # 要点: 为什么这里(target == 中点值), 不直接返回?
26        # 因为在左中点左边还可能存在更靠左的target值
27        else:
28            end = mid
29
30    # 要点: 为什么这里先检查start, 再检查end?
31    # 因为我们要求的是first index, 所以先检查更靠左的start
32    # 如果start不为target, 再检查稍微靠右的end
33    # 比如, 在[9, 9]中寻找9, 应返回0 (第一个9的index)
34    if reader.get(start) == target:
35        return start
36    if reader.get(end) == target:
37        return end
38    return -1
```

| | | |
|-------|-------------|--|
| 时间复杂度 | $O(\log K)$ | 倍增为 $O(x) \approx O(\log K)$ 二分为 $O(\log 2^{x-1}) = O(x) \approx O(\log K)$ x 为调用 reader.get() 的次数 k 为target第一次出现的位置的下标 |
| 空间复杂度 | $O(1)$ | |

$$2^{x-2} - 1 < K \leq 2^{x-1} - 1$$

⇓

$$x - 2 < \log(K + 1) \leq x - 1$$

⇓

$$\log(K + 1) + 1 \leq x < \log(K + 1) + 2$$

⇓

$$x \approx \log K$$

460 Find K Closest Elements在排序数组中找最接近的K个数

Given **target**, a non-negative integer **k** and an integer **array** **A** **sorted** in ascending order, find the **k** closest numbers to **target** in **A**, sorted in ascending order by the difference between the number and **target**. Otherwise, sorted in ascending order by number if the difference is same.

给一个目标数 **target**, 一个非负整数 **k**, 一个按照升序排列的数组 **A**。在**A**中找与**target**最接近的**k**个整数。返回这**k**个数并按照与**target**的接近程度从小到大排序，如果接近**程度相当**，那么**小的数排在前面**。

输入: **A** = [1, 2, 4, 6, 8], **target** = 3, **k** = 4

输出: [2, 4, 1, 6]

target值有可能不在数据中

array + **target** + **sorted**  **binary search**



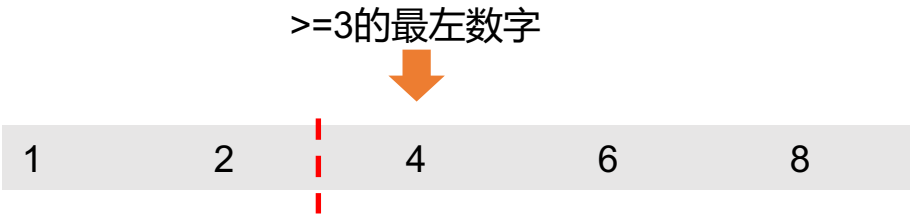
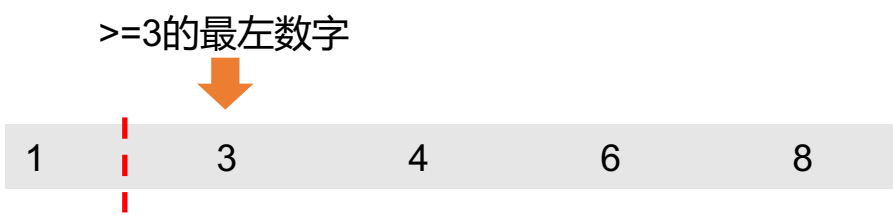
1 2 3 4 6 8

⋮
⋮

第一步：找到中界线（**target**的插入位置）

第二步：从中界线两边找最近的**k**个数字

题目解析 —— 第一步：找到中界线（target的插入位置）

| | 不存在3 | 存在3 |
|----------------|--|---|
| ≤ 3 的最右数字 | <p>≤ 3 的最右数字</p>  | <p>≤ 3 的最右数字</p>  |
| ≥ 3 的最左数字 | <p>≥ 3 的最左数字</p>  | <p>≥ 3 的最左数字</p>  |

其他思路？

< 3 的最右数字

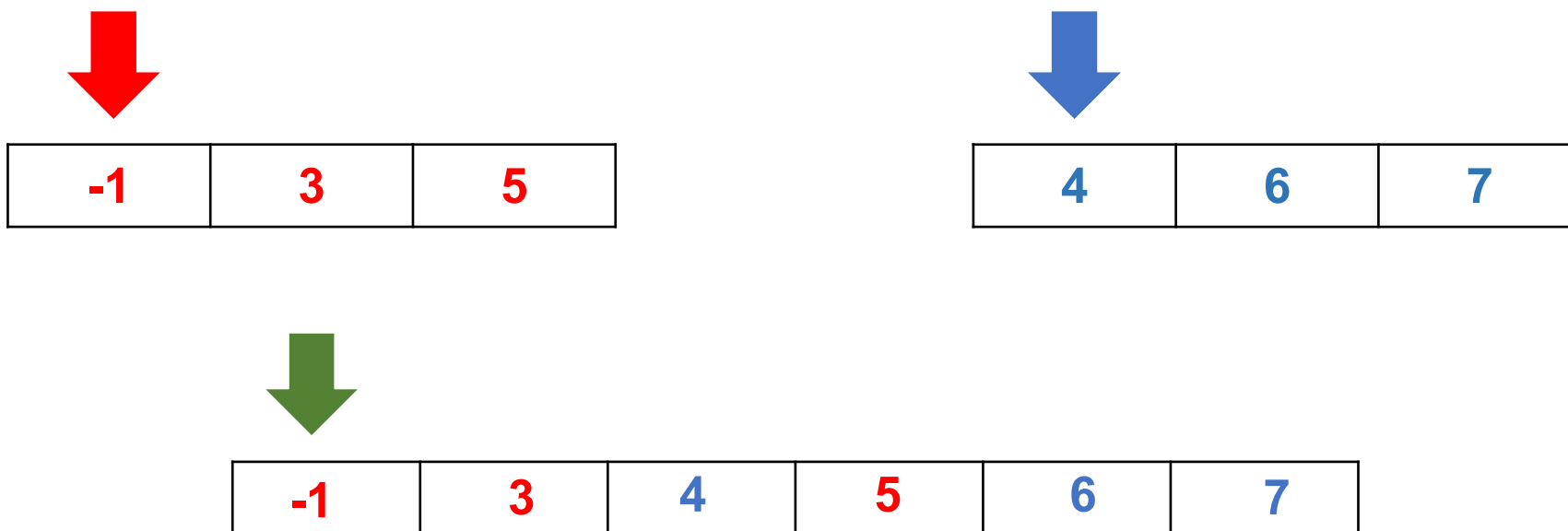
> 3 的最左数字

Merge Sorted Array解析

两个sorted arrays，每个array有一个指针指向参与比较的元素

初始化一个新的数组，长度为两数组之和，用来存放merge好的数组。用一个指针指向用于存放下一个元素的空位

使用两个指针分别对数组从小到大遍历，每次取二者中较小的放在新数组中，直到某个指针先到结尾，另一个数组中剩余的数字直接放在新数组后面



题目解析 —— 第二步：从中界线两边找最近的k个数字

思路

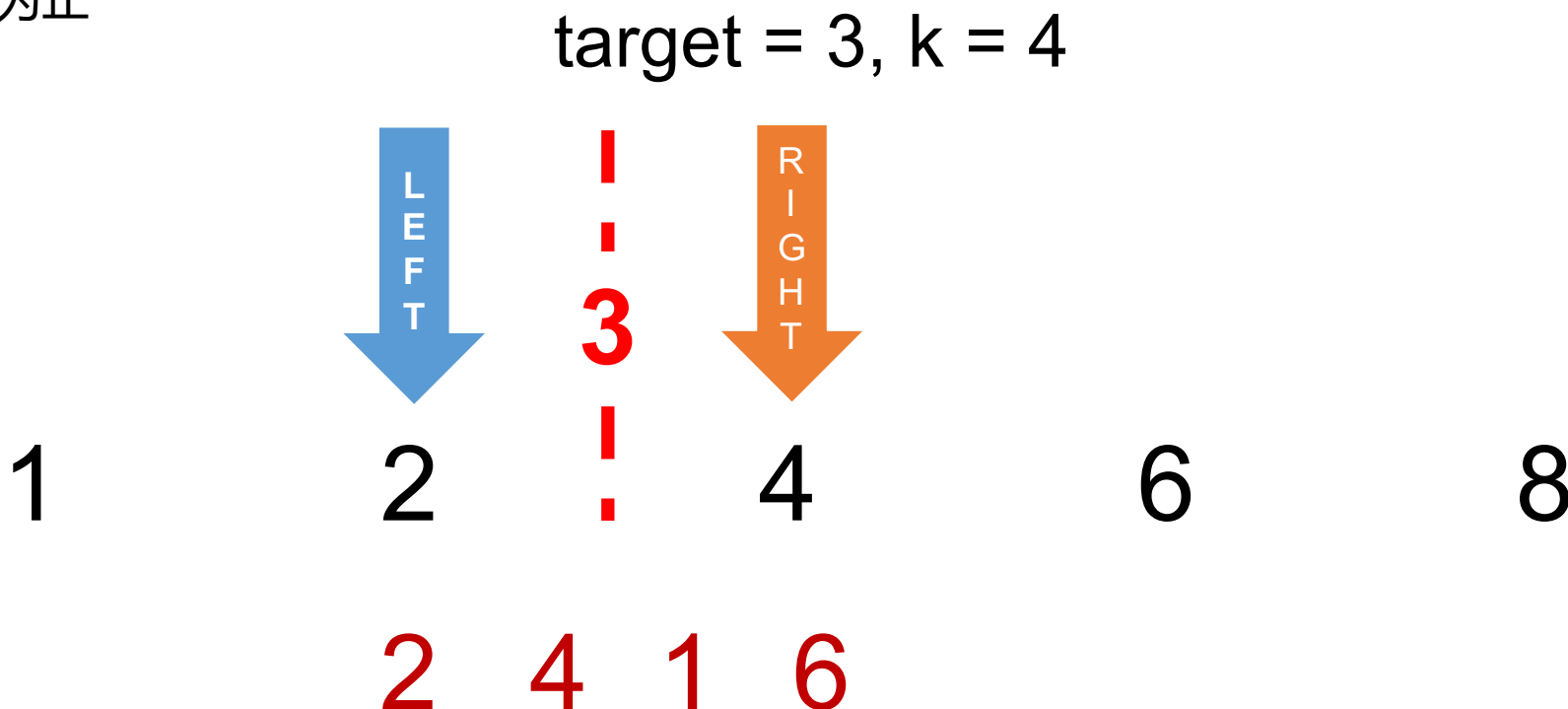
找到跟target最接近的数

以这个数为轴心，左边的数字离target越来越远，右边的数字也离target越来越远

不断从左右两边子数列中选择跟target更近的数字

直到选出k个数字为止

→ Merge
Sorted
Array



```

3  def kClosestNumbers(self, A, target, k):
4      right = self.findUpperClosest(A, target)
5      left = right - 1
6
7      # 两根指针从中间往两边扩展, 依次找到最接近的 k 个数
8      results = []
9      for _ in range(k):
10         # 如果左边更接近, 选左边
11         if self.isLeftCloser(A, target, left, right):
12             results.append(A[left])
13             left -= 1
14         else:
15             results.append(A[right])
16             right += 1
17
18     return results
19
20     def isLeftCloser(self, A, target, left, right):
21         # 如果左边已经耗尽, 返回false
22         if left < 0:
23             return False
24
25         # 如果右边已经耗尽, 返回true
26         if right == len(A):
27             return True
28
29         # 为什么有等号? 如果左右距离相等, 选左边
30         return target - A[left] <= A[right] - target
    
```

target = 3, k = 4 1 2 3 4 6 8

⋮
⋮

```

37     # 找到最左边的 >= target 的元素
38     def findUpperClosest(self, A, target):
39         start, end = 0, len(A) - 1
40         while start + 1 < end:
41             mid = (start + end) // 2
42             # 如果 mid >= target, mid 符合条件向左边去寻找
43             # 更靠左的符合条件的元素, 丢掉右边
44             if A[mid] >= target:
45                 end = mid
46             # 如果 mid < target, >= target 的元素在右边, 丢掉左边
47             else:
48                 start = mid
49
50         # 因为需要找最左数, 所以这里需要先判断 start
51         if A[start] >= target:
52             return start
53         # 如果 end 不行, 再判断 start
54         if A[end] >= target:
55             return end
56
57         # 找不到的情况
58         return len(A)
    
```

找不到时, 为什么返回 len(A) ?

如果插入位置在最左边怎么办?

时间复杂度

$O(\log N) + O(K)$

二分法寻找 target 需要 $O(\log N)$

Merge 过程需要 $O(K)$

空间复杂度

$O(K)$

需要长度为 K 的数组

记录数据

```

2 public int[] kClosestNumbers(int[] A, int target, int k) {
3     int left = findLowerClosest(A, target);
4     int right = left + 1;
5
6     int[] results = new int[k];
7     for (int i = 0; i < k; i++) {
8         // 如果左边更接近, 选左边
9         if (isLeftCloser(A, target, left, right)) {
10             results[i] = A[left];
11             left--;
12         } else {
13             results[i] = A[right];
14             right++;
15         }
16     }
17
18     return results;
19 }
20
21 private boolean isLeftCloser(int[] A, int target, int left, int right) {
22     // 如果左边已经耗尽, 返回false
23     if (left < 0) {
24         return false;
25     }
26
27     // 如果右边已经耗尽, 返回true
28     if (right == A.length) {
29         return true;
30     }
31
32     // 为什么有等号? 如果左右距离相等, 选左边
33     return target - A[left] <= A[right] - target;
34 }

```

target = 3, k = 4

1 2 3 4 6 8
: : :

```

42 // 找到比target小的最右一个数
43 private int findLowerClosest(int[] A, int target) {
44
45     int start = 0, end = A.length - 1;
46     while (start + 1 < end) {
47         int mid = start + (end - start) / 2;
48         // 如果mid < target, 答案在右边, 丢掉左边
49         if (A[mid] < target) {
50             start = mid;
51         }
52         // 如果mid >= target, 答案在左边, 丢掉右边
53         else {
54             end = mid;
55         }
56     }
57
58     // 因为需要找最右数, 所以这里需要先判断end
59     if (A[end] < target) {
60         return end;
61     }
62     // 如果end不行, 再判断left
63     if (A[start] < target) {
64         return start;
65     }
66     // 找不到, 说明所有的数据都>=target
67     return -1;
68 }

```

585 Maximum Number in Mountain Sequence 山脉序列中的最大值

Given a mountain sequence of n integers which **increase firstly and then decrease**, find the mountain top(Maximum).

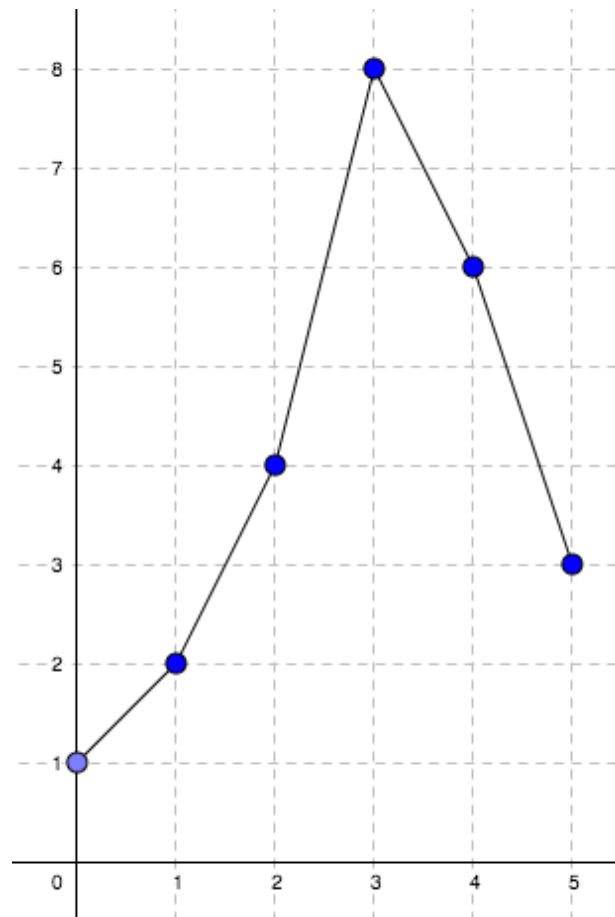
给 n 个整数的山脉数组，即先增后减的序列（没有相等），找到山顶（最大值）。

输入：

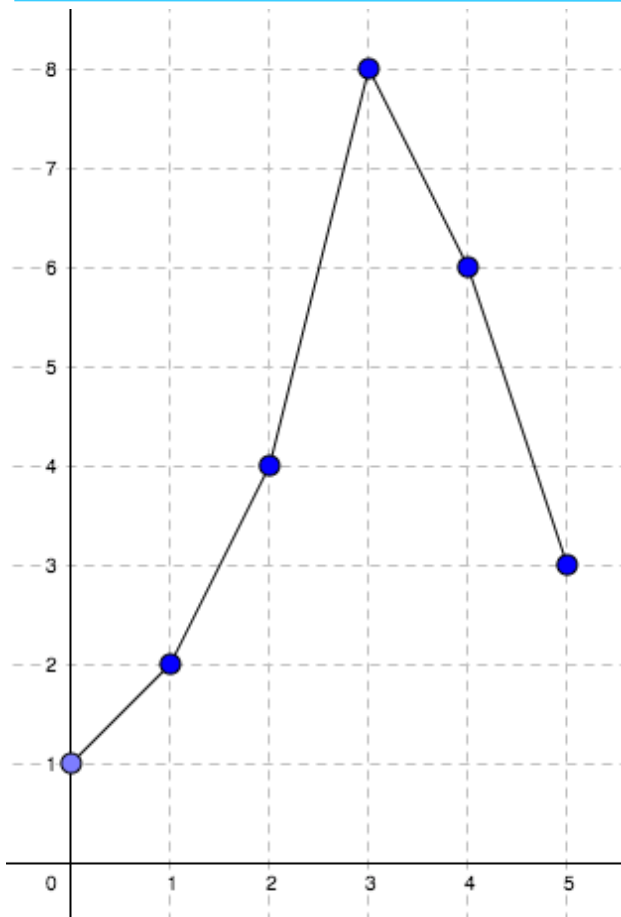
[1, 2, 4, 8, 6, 3]

输出：

8



代码解析



```

8  ✓ if not nums:
9      return -1
10
11  # 找到第一个符合条件nums[i] > nums[i + 1]的i
12  start, end = 0, len(nums) - 1
13  ✓ while start + 1 < end:
14      mid = (start + end) // 2
15      # mid + 1一定不会越界
16      # 因为while循环退出条件是start + 1 < end,
17      # 所以一定有start < mid + 1 < end
18      # 如果从mid点向右下方倾斜, 山峰一定在左边, 丢掉右边
19  ✓ if nums[mid] > nums[mid + 1]:
20      end = mid
21      # 如果从mid点向右上方倾斜, 山峰一定在右边, 丢掉左边
22  ✓ else:
23      start = mid
24      # 返回start和end中较大的值, 则为山顶
25      return max(nums[start], nums[end])

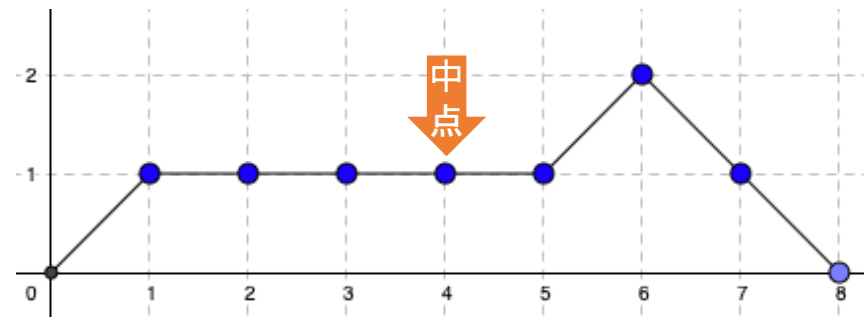
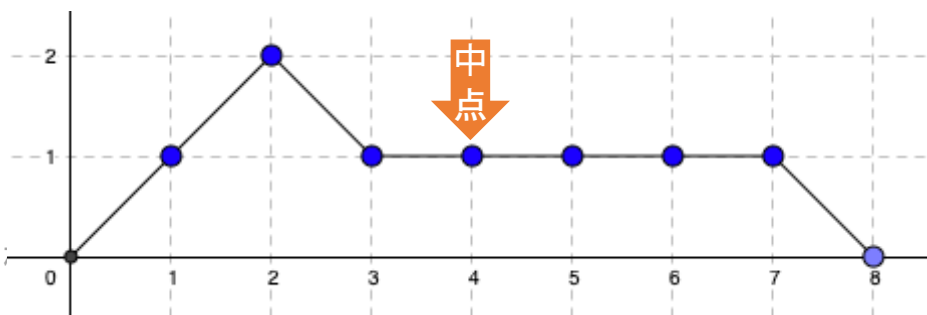
```

```

6  public int mountainSequence(int[] nums) {
7      // 特殊情况处理, 特殊情况返回值需要跟面试官确认
8      if (nums == null || nums.length == 0) {
9          return -1;
10     }
11
12     // 找到第一个符合条件nums[i] > nums[i + 1]的i
13     int start = 0, end = nums.length - 1;
14     while (start + 1 < end) {
15         int mid = (start + end) / 2;
16         // mid + 1一定不会越界
17         // 因为while循环退出条件是start + 1 < end,
18         // 所以一定有start < mid + 1 < end
19         // 如果从mid点向右下方倾斜, 山峰一定在左边, 丢掉右边
20         if (nums[mid] > nums[mid + 1]) {
21             end = mid;
22         }
23         // 如果从mid点向右上方倾斜, 山峰一定在右边, 丢掉左边
24         else {
25             start = mid;
26         }
27     }
28     // 返回start和end中较大的值, 则为山顶
29     return Math.max(nums[start], nums[end]);
30 }

```

如果输入数据只有两个数怎么办？比如[1, 2]
为什么没有重复点是个很重要的条件？



快扶我起来
我还能学



159. Find Minimum in Rotated Sorted Array 寻找旋转排序数组中的最小值

Suppose a sorted array in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the **minimum element**.

You can assume **no duplicate exists** in the array.

假设一个按升序排好序的数组在其某一未知点发生了旋转（比如0 1 2 4 5 6 7 可能变成4 5 6 7 0 1 2）。你需要找到其中**最小的元素**。

你可以假设数组中**不存在重复元素**。

输入：

[4, 5, 6, 7, **0**, 1, 2]

输出：

0（输出最小值）

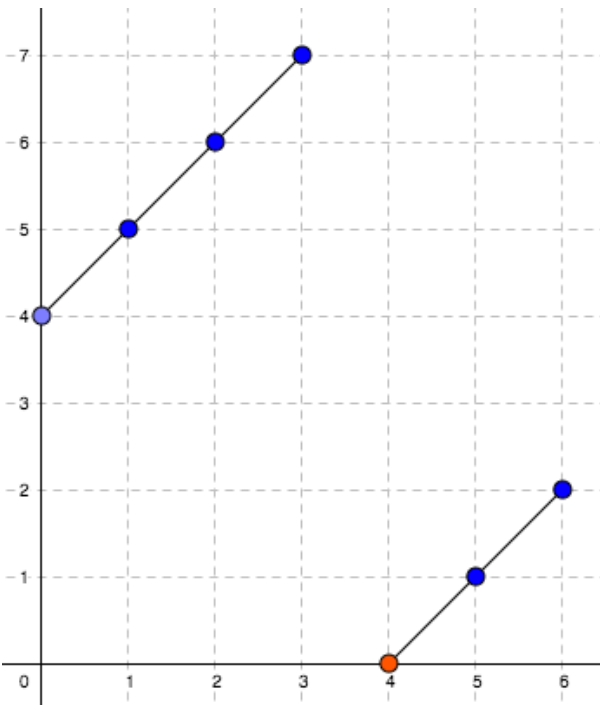
输入：

[2, **1**]

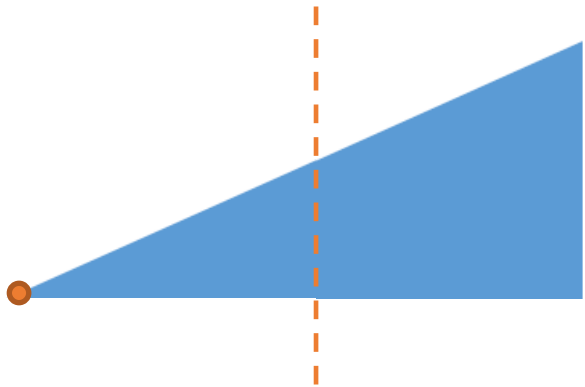
输出：

1（输出最小值）

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 4 | 5 | 6 | 7 | 0 | 1 | 2 |



```
3 def findMin(self, nums):
4     # 特殊情况处理, 需要跟面试官确认特殊情况的返回值
5     if not nums:
6         return -1
7
8     start, end = 0, len(nums) - 1
9     while start + 1 < end:
10        mid = (start + end) // 2
11        # 如果mid > end, 起点在右边, 抛弃左边
12        if nums[mid] > nums[end]:
13            start = mid
14        # 如果mid < end, 起点在左边, 抛弃右边
15        # 注意: 题目声明不存在相等元素, 所以这里else
16        # 等价于mid < end
17        else:
18            end = mid
19        # 返回start和end指向的最小值
20    return min(nums[start], nums[end])
```



```
6 public int findMin(int[] nums) {
7     // 特殊情况处理, 需要跟面试官确认特殊情况的返回值
8     if (nums == null || nums.length == 0) {
9         return -1;
10    }
11
12    int start = 0, end = nums.length - 1;
13
14    while (start + 1 < end) {
15        int mid = start + (end - start) / 2;
16        // 如果mid > end, 起点在右边, 抛弃左边
17        if (nums[mid] > nums[end]) {
18            start = mid;
19        }
20        // 如果mid < end, 起点在左边, 抛弃右边
21        // 注意: 题目声明不存在相等元素, 所以这里else
22        // 等价于mid < end
23        else {
24            end = mid;
25        }
26    }
27    // 返回start和end指向的最小值
28    return Math.min(nums[start], nums[end]);
29 }
```

62 Search in Rotated Sorted Array 搜索旋转排序数组

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

给定一个有序数组，但是数组以某个元素作为支点进行了旋转(比如，0 1 2 4 5 6 7 可能成为4 5 6 7 0 1 2)。给定一个目标值target进行搜索，如果在数组中找到目标值返回数组中的索引位置，否则返回-1。你可以假设数组中不存在重复的元素。你可以假设数组中不存在重复的元素。

输入：

array = [4, 5, 1, 2, 3]

target = 1

输出：

2

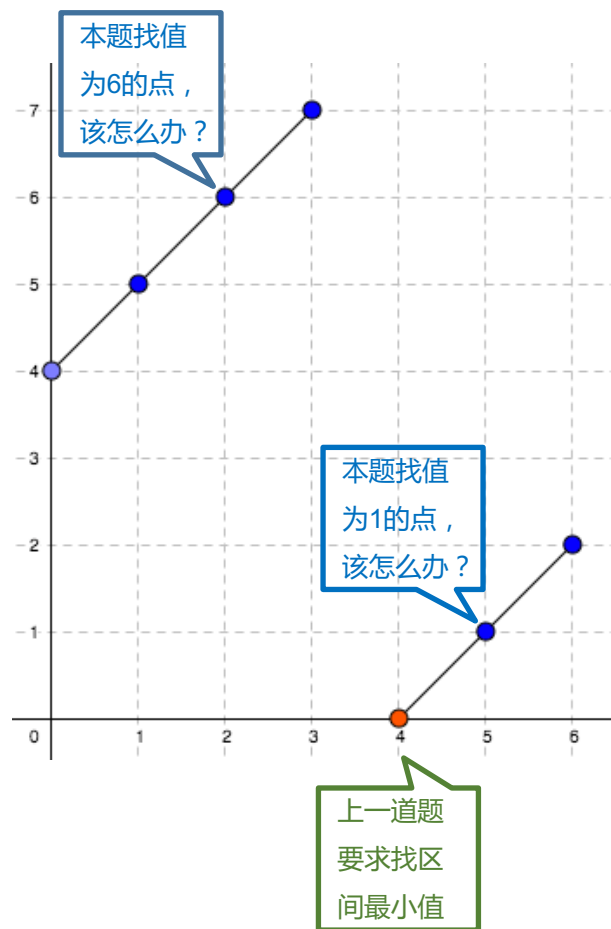
输入：

array = [4, 5, 1, 2, 3]

target = 0

输出：

-1



用两次二分法即可

- 第一次二分找到最小值
- 第二次二分在左上或右下区间找目标值

如果面试官继续纠缠

面试官：刚才这个题你写得太快了不算

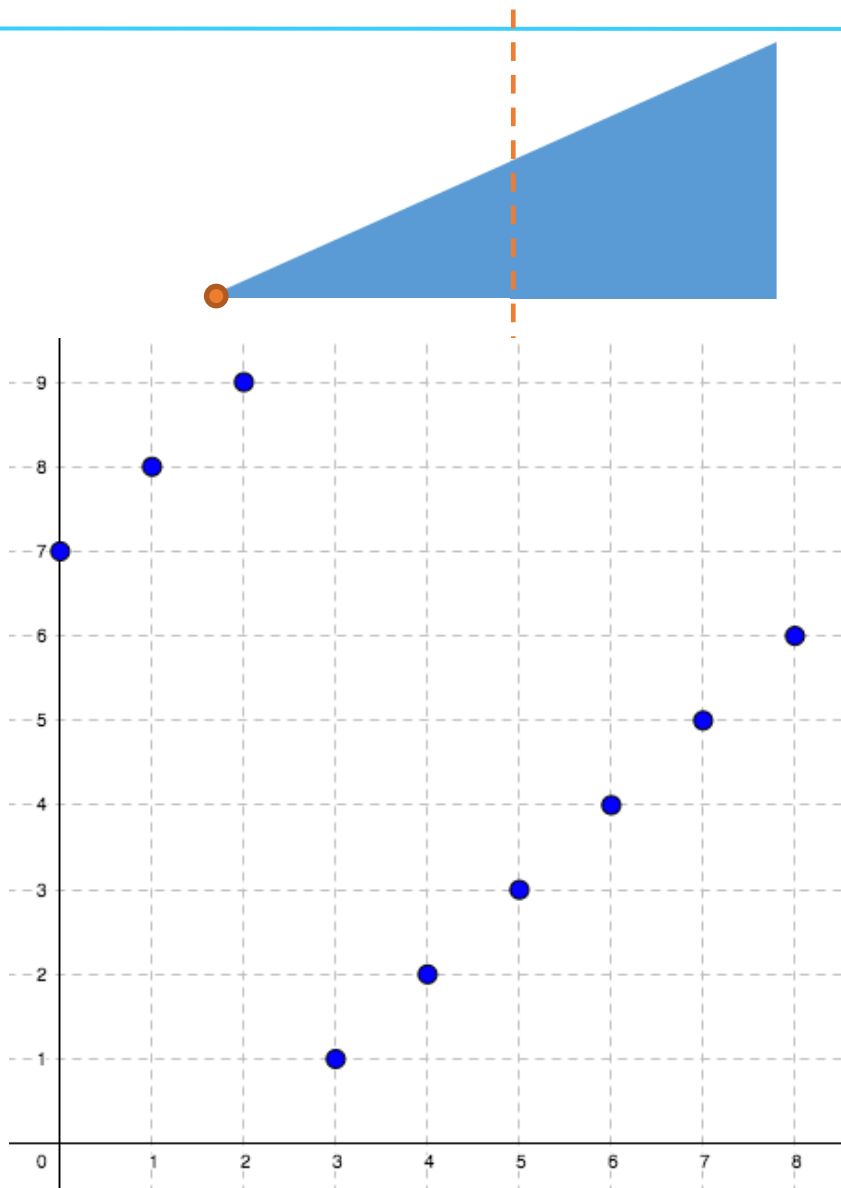
面试官：你能否**只用一次**二分就解决这个问题？

（答出来可以拿到 Strong Hire，答不出来也没关系）

我太南了



寻找1



```

7  def search(self, A, target):
8      # 特殊情况处理
9      if not A:
10         return -1
11
12     start, end = 0, len(A) - 1
13     while start + 1 < end:
14         mid = (start + end) // 2
15         # mid在左上面的线
16         if A[mid] > A[end]:
17             # target在“中段”
18             if A[start] <= target <= A[mid]:
19                 end = mid
20             # target在“头段”或“尾段”
21             else:
22                 start = mid
23         # mid在右下面的线
24         else:
25             # target在“中段”
26             if A[mid] <= target <= A[end]:
27                 start = mid
28             # target在“头段”或“尾段”
29             else:
30                 end = mid
31
32     # 在无重复数据中寻找target, 先判断start或end都可以
33     if A[start] == target:
34         return start
35     if A[end] == target:
36         return end
37     return -1
    
```

小结——在有序的输入集合中的二分查找

本质

在有序的数组中寻找一个跟Target有关的Index或值

时间复杂度

一次二分法是 $O(\log N)$

关键词

array, target, sorted , equal or close to target的词 (e.g. , 小于n的最大值) , $O(N\log N)$

套路

如果求两个数和的target , 固定一个值 , 另一个值二分查找

如果没有排序 , 可以先排序再求解

OOXX

OOOOOOO...OXX...XXXXXX

第二境界 OOXX 在排序的数据集上进行二分

一般会给你一个数组

让你找数组中第一个/最后一个满足某个条件的位置

OOOOOOO...O**O****X**X....XXXXXX

第三境界

在未排序的数据集上进行二分

并无法找到一个条件，形成 XXOO 的模型
但可以根据判断，保留下有解的那一半或者去掉无解的一半



75 Find Peak Element 寻找峰值

There is an integer array which has the following features:

The numbers in adjacent positions are different.

$A[0] < A[1]$ && $A[A.length - 2] > A[A.length - 1]$.

We define a position P is a peak if:

$A[P] > A[P-1]$ && $A[P] > A[P+1]$ Find a peak element in this array. Return the index of the peak.

It's guaranteed the array has at least one peak.

The array may contain multiple peaks, find any of them.

The array has at least 3 numbers in it.

给定一个整数数组(size为n)，其具有以下特点：

相邻位置的数字是不同的

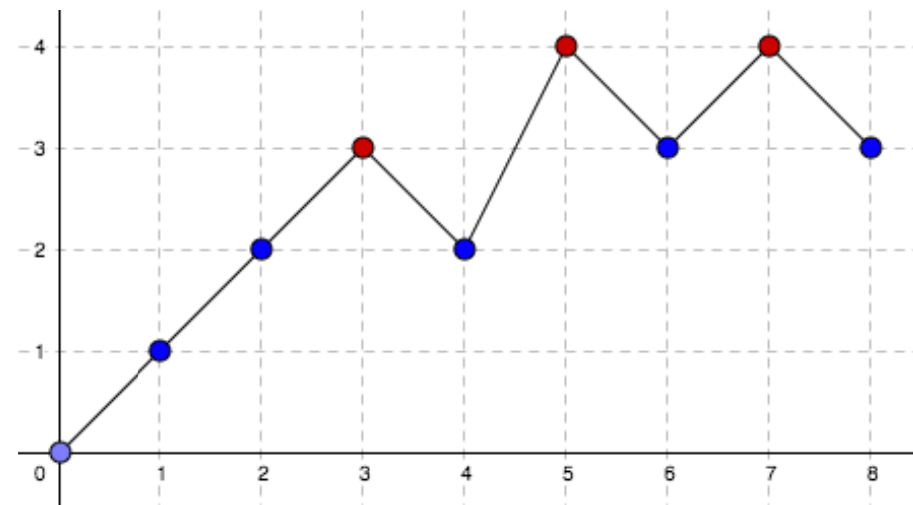
$A[0] < A[1]$ 并且 $A[n - 2] > A[n - 1]$

假定P是峰值的位置则满足 $A[P] > A[P-1]$ 且 $A[P] > A[P+1]$ ，返回数组中**任意一个**峰值的位置。

数组保证至少存在一个峰

如果数组存在多个峰，返回其中任意一个就行

数组至少包含 3 个数



输入：

[1, 2, 1, 3, 4, 5, 7, 6]

输出：

1 or 6 （输出下标）

输入：

[1,2,3,4,1]

输出：

3 （输出下标）

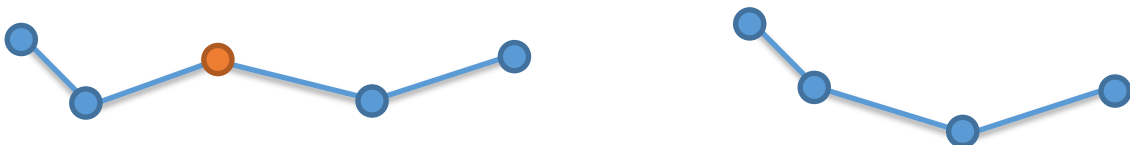


一起爬山吗
我还有机会吗

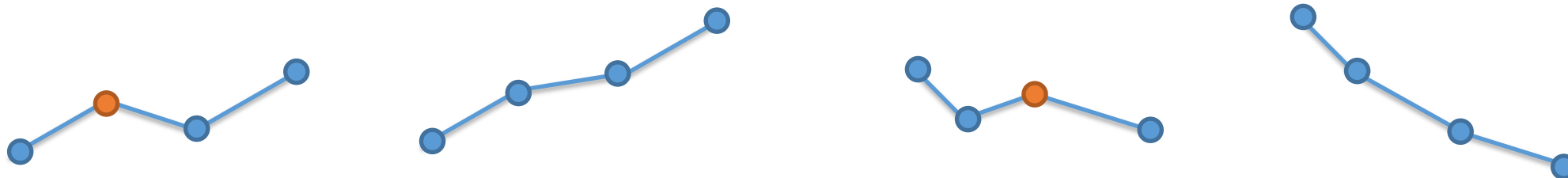
题目解析 —— 为什么 $A[0] < A[1]$ 并且 $A[n - 2] > A[n - 1]$ 很重要？

已知条件： $A[0] < A[1]$ 并且 $A[n - 2] > A[n - 1]$ 翻译成成人话（画图），就是**两端都是向中间上升的趋势**（或者理解为**两端先升后降**）

两端都向中间下降，**可能**没有山峰



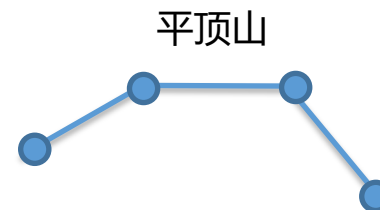
只有一端向中间上升，**可能**没有山峰



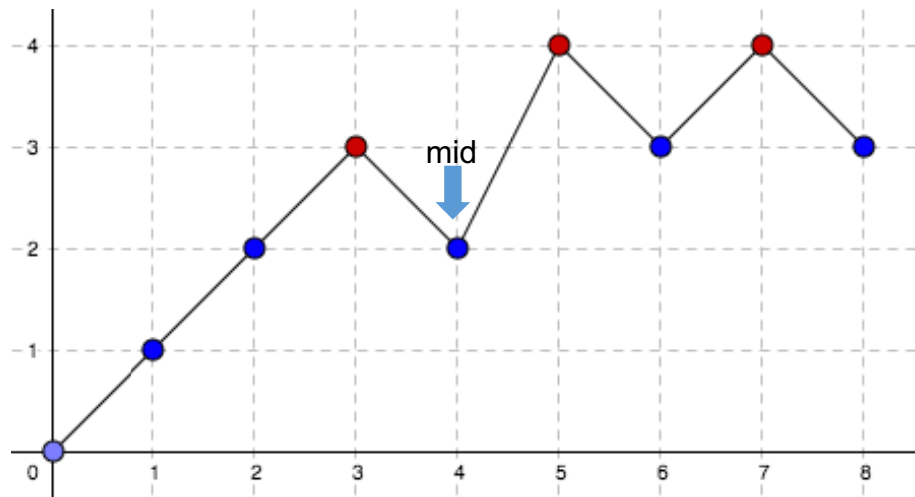
两端都是向中间上升的趋势，**一定**有山峰



真的**一定**有山峰吗？

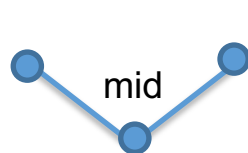


已知条件：**相邻位置的数字是不同的**

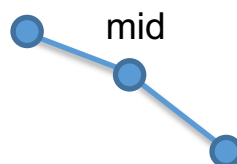


```

2  def findPeak(self, A):
3      # Peak不可能在两端, 所以在[1, A.length - 2]范围内寻找
4      start, end = 1, len(A) - 2
5      while start + 1 < end:
6          mid = (start + end) // 2
7          # 如果mid向左上方倾斜, 选左半边
8          if A[mid] < A[mid - 1]:
9              end = mid
10         # 如果mid向右上方倾斜, 选右半边
11         elif A[mid] < A[mid + 1]:
12             start = mid
13         # 如果mid为peak, 返回
14         else:
15             return mid
16     # 因为保证一定有peak, 所以返回start和end中大的一个
17     return end if A[start] < A[end] else start
    
```



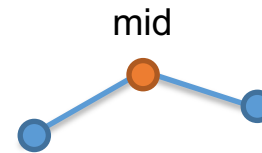
左边或右边一定有peak



左边一定有peak



右边一定有peak



找到了peak

```

2  public int findPeak(int[] A) {
3      // Peak不可能在两端, 所以在[1, A.length - 2]范围内寻找
4      int start = 1, end = A.length - 2;
5      while (start + 1 < end) {
6          int mid = start + (end - start) / 2;
7          // 如果mid向左上方倾斜, 选左半边
8          if (A[mid] < A[mid - 1]) {
9              end = mid;
10         }
11         // 如果mid向右上方倾斜, 选右半边
12         else if (A[mid] < A[mid + 1]) {
13             start = mid;
14         }
15         // 如果mid为peak, 返回
16         else {
17             return mid;
18         }
19     }
20     // 因为保证一定有peak, 所以返回start和end中大的一个
21     return A[start] < A[end] ? end : start;
22 }
    
```

第四境界 在答案集上进行二分

第一步：确定答案范围

第二步：验证答案大小

183. Wood Cut 木材加工

Given n pieces of wood with length $L[i]$ (integer array). Cut them into small pieces to guarantee you could have equal or more than k pieces with the same length. What is the longest length you can get from the n pieces of wood? Given L & k , return the maximum length of the small pieces. The unit of length is centimeter. The length of the woods are all positive integers, you couldn't cut wood into float length. If you couldn't get $\geq k$ pieces, return 0.

有一些原木，现在想把这些木头切割成一些长度相同的小段木头，需要得到的小段的数目至少为 k 。给定 L 和 k ，你需要计算能够得到的小段木头的**最大长度**。

木头长度的单位是厘米。原木的长度都是正整数，我们要求切割得到的小段木头的长度也要求是**整数**。无法切出要求至少 k 段的,则返回 0 即可。

题意：把很多根木头进行整数切割，最后可以得到 k 段等长的木头（木头不能拼接），切割后木头的最大长度是多少？

输入：

$L = [4, 6, 7, 8]$

$k = 3$

输出：

6

| 切割长度 \ 原木长度 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------|----|----|---|---|---|---|---|---|
| 4 | 4 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| 6 | 6 | 3 | 2 | 1 | 1 | 1 | 0 | 0 |
| 7 | 7 | 3 | 2 | 1 | 1 | 1 | 1 | 0 |
| 8 | 8 | 4 | 2 | 2 | 1 | 1 | 1 | 1 |
| 切割总段数 | 25 | 12 | 7 | 5 | 3 | 3 | 2 | 1 |

输入：

$L = [1, 2, 3]$

$k = 7$

输出：

0

解释：

如果木头长度为1，只能切6段，不能得出7段。

题目解析 —— 错误的想法

输入：

$L = [4, 6, 7, 8]$

$k = 3$

输出：

6

木头的最大值 = 所有原木长度之和 / 木头目标段数

$4 + 6 + 7 + 8 = 25$

$25 / 3 = 8$



这里求出的木头最大值是在某些理想数据情况下（边角料最少）的最大值。
木头是不可拼接的

| 切割长度 \ 原木长度 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------|----|----|---|---|---|---|---|---|
| 4 | 4 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| 6 | 6 | 3 | 2 | 1 | 1 | 1 | 0 | 0 |
| 7 | 7 | 3 | 2 | 1 | 1 | 1 | 1 | 0 |
| 8 | 8 | 4 | 2 | 2 | 1 | 1 | 1 | 1 |
| 切割总段数 | 25 | 12 | 7 | 5 | 3 | 3 | 2 | 1 |

理想数据

输入：

$L = [1, 8, 8, 8]$

$k = 3$

输出：

8

题目解析 —— 如何求切割后木头的最小值和最大值？

最小值

最小每段长度为1

最大值

想法1

木头的最大值 = 所有原木中的最大值。

无论k为多少，每段木头长度 \leq 原木中的最大值，否则一块木头也切不出来

想法2

木头的最大值 = 所有原木长度之和 / 木头目标段数

比如， $L = [3, 6, 7, 8]$, $k = 5$, $3 + 6 + 7 + 8 = 24$, $24 / 5 = 4$

如果长度为4，只能切出4段，不是5段。最终答案为3。

比如， $L = [1, 6, 8, 9]$, $k = 5$, $1 + 6 + 8 + 9 = 24$, $24 / 5 = 4$

如果长度为4，可以切出5段。最终答案为4。

这里最大值采用想法1或2或3，很重要吗？

不重要，这些最大值都正确，大一点小一点不会影响时间复杂度的量级

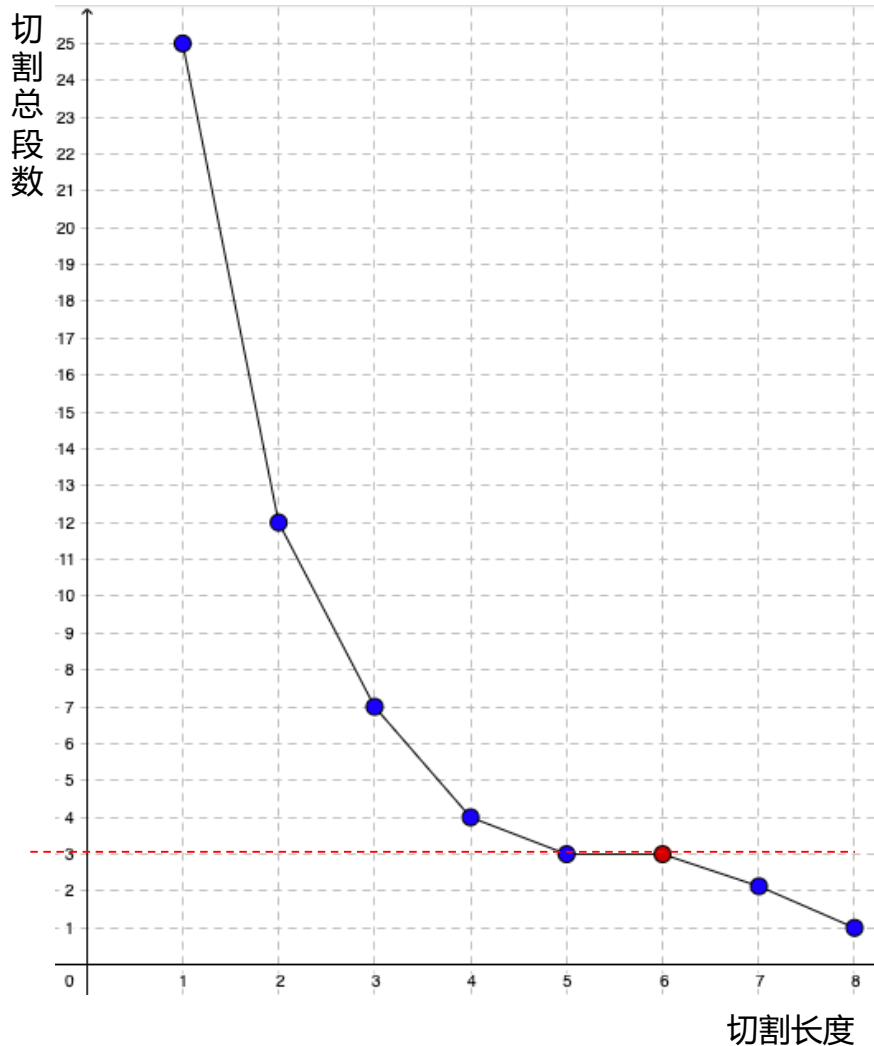
想法3

木头的最大值 = $\min(\text{所有原木中的最大值}, \text{所有原木总长度} / \text{木头目标段数})$

比如， $L = [3, 6, 7, 8]$, $k = 5$, $3 + 6 + 7 + 8 = 24$, $24 / 5 = 4$ ，木头的最大值 = $\min(8, 4) = 4$ 。最终答案为3。

比如， $L = [18, 20, 30]$, $k = 2$, $18 + 20 + 30 = 68$, $68 / 2 = 34$ ，木头的最大值 = $\min(30, 34) = 30$ 。最终答案为20。

切割长度和切割总段数有负相关映射关系



k = 3

| 切割长度 \ 原木长度 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------|----|----|---|---|---|---|---|---|
| 4 | 4 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| 6 | 6 | 3 | 2 | 1 | 1 | 1 | 0 | 0 |
| 7 | 7 | 3 | 2 | 1 | 1 | 1 | 1 | 0 |
| 8 | 8 | 4 | 2 | 2 | 1 | 1 | 1 | 1 |
| 切割总段数 | 25 | 12 | 7 | 5 | 3 | 3 | 2 | 1 |

```
2 def woodCut(self, L, k):
3     # 特殊情况处理
4     if not L:
5         return 0
6
7     # 初始化start和end, end = min(max in L, sumL / k)
8     start, end = 1, min(max(L), sum(L) // k)
9
10    # 如果end小于1, 不可能完成任务, 返回0
11    if end < 1:
12        return 0
```

```
14
15    while start + 1 < end:
16        mid = (start + end) // 2
17        # 长度为mid的木头总数 >= 目标总数, 继续增长木头长度, 选右边
18        if self.get_count(L, mid) >= k:
19            start = mid;
20        # 长度为mid的木头总数 < 目标总数, 继续缩短木头长度, 选左边
21        else:
22            end = mid;
23
24    # 因为之前排除了无解的状况, 所有这里一定有解, 非start即end
25    # 如果end符合要求, 首选end (因为end更长), 否则选start
26    return end if self.get_count(L, end) >= k else start
27
28 def get_count(self, L, length):
29     return sum(l // length for l in L)
```

| | | | | | | | | |
|--------|----|----|---|---|---|---|---|---|
| 每段木头长度 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 段数 | 25 | 12 | 7 | 4 | 3 | 3 | 2 | 1 |

切割长度和切割总段数

有**负相关映射关系**

| | | |
|-------|--------------|--------------------------|
| 时间复杂度 | $O(K\log N)$ | K = 原木段数, N = 答案范围大小 |
| 空间复杂度 | $O(1)$ | |

```
2 public int woodCut(int[] L, int k) {
3     // 特殊情况处理
4     if (L == null || L.length == 0) {
5         return 0;
6     }
7
8     // 初始化start和end, end = min(max in L, sumL / k)
9     int start = 1, end = 0, maxLen = 0;
10    long sum = 0;
11    for (int l : L) {
12        end = Math.max(end, l);
13        sum += l;
14    }
15    end = (int) Math.min(end, sum / k);
16
17    // 如果end小于1, 不可能完成任务, 返回0
18    if (end < 1) {
19        return 0;
20    }
21
22    while (start + 1 < end) {
23        int mid = start + (end - start) / 2;
24        // 长度为mid的木头总数 >= 目标总数, 继续增长木头长度, 选右边
25        if (getCount(L, mid) >= k) {
26            start = mid;
27        }
28        // 长度为mid的木头总数 < 目标总数, 继续缩短木头长度, 选左边
29        else {
30            end = mid;
31        }
32    }
33
34    // 因为之前排除了无解的状况, 所有这里一定有解, 非start即end
35    // 如果end符合要求, 首选end (因为end更长), 否则选start
36    return (getCount(L, end) >= k) ? end : start;
37 }
38
39 private int getCount(int[] L, int len) {
40     int cnt = 0;
41     for (int item : L) {
42         cnt += item / len;
43     }
44     return cnt;
45 }
```

小结 —— 在答案集上进行二分

本质：

求满足某条件的最大值或者最小值

最终结果是个有限的集合

每个结果有一个对应的映射

结果集合跟映射集合正相关或着负相关

可以通过在映射集合上进行二分，从而实现对结果集合的二分

关键词：

Array, 有限个答案, 映射, 正负相关, $O(N\log N)$

Thanks♪(・ω・)ﾉ



早晨一个学生背着 N 本出了图书馆，结果警报响了，管理员让学生看看是哪本书把警报弄响了，那学生把书倒出来，准备一本一本的测。管理员见状急了，把书分成两份，第一份过了一下，响了。又把这一份分成两份接着测，三回就找到了，管理员用鄙视的眼神看着学生，仿佛他不懂算法。

结果图书馆丢了 $N - 1$ 本书。



小丑竟然是我自己

同学们，下次再见👋！记得课后复习，课前预习！😊

做作业

做ladder

群里提问题

看回放



课前预习

课后复习

刷题

互动课