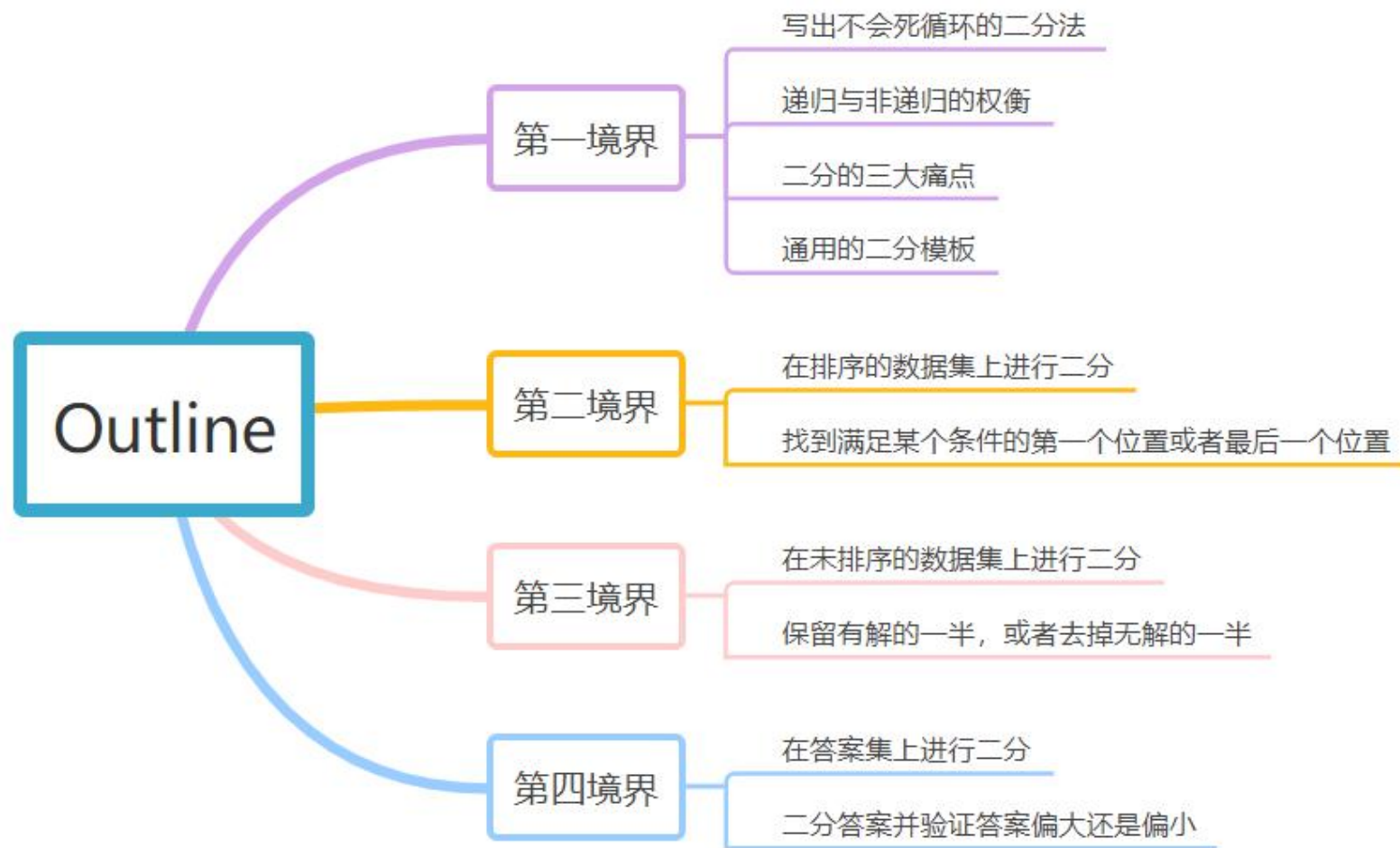


简约而不简单—— 二分法的四重境界

主讲人 令狐冲
课程版本 v7.0

版权声明

九章的所有课程均受法律保护，不允许录像与传播录像
一经发现，将被追究法律责任和赔偿经济损失



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
- $O(2^n)$, 组合类的搜索问题
- $O(n!)$ 排列类的搜索问题

独孤九剑 —— 破刀式

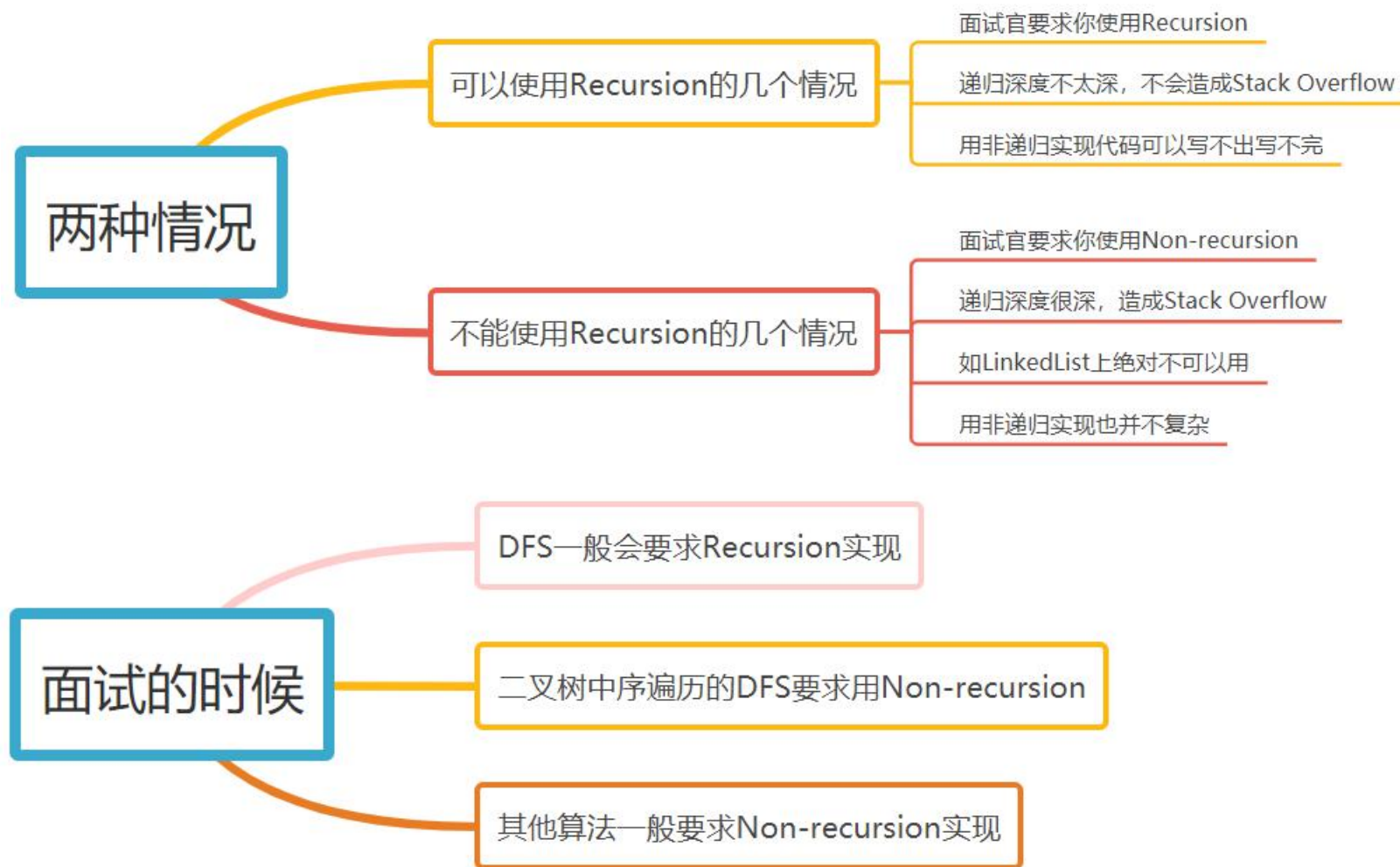
根据时间复杂度倒推算法是面试常用策略
如：比 $O(n)$ 更优的时间复杂度只能是 $O(\log n)$ 的二分法

Recursion or While Loop?

R: Recursion

W: While loop

B: Both work



第一境界

写出不会死循环的二分法

<http://www.jiuzhang.com/solutions/binary-search/>

$start + 1 < end$

$start + (end - start) / 2$

$A[mid] ==, <, >$

$A[start] A[end] ? target$

又死循环了！
what are you 弄撒
捏！

循环结束条件到底是
哪个？！

$\text{start} \leq \text{end}$

$\text{start} < \text{end}$

$\text{start} + 1 < \text{end}$

挠头...



指针变化到底是
哪个？

$\text{start} = \text{mid}$

$\text{start} = \text{mid} + 1$

$\text{start} = \text{mid} - 1$

```
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;
}
```

死循环的发生

Last Position of Target

nums = [1,1], target = 1

使用 `start < end` 无论如何都会出现死循环

任意位置 vs 第一个位置 vs 最后一个位置

<http://www.lintcode.com/problem/classical-binary-search/>

<http://www.lintcode.com/problem/first-position-of-target/>

<http://www.lintcode.com/problem/last-position-of-target/>

Search In a Big Sorted Array

<http://www.lintcode.com/problem/search-in-a-big-sorted-array/>

<http://www.jiuzhang.com/solutions/search-in-a-big-sorted-array/>

大小未知的排序数组中找 **target** 出现的位置

无法直接确定二分右边界怎么办？

倍增法

Exponential Backoff

使用到倍增思想的场景：

动态数组（ArrayList in Java, vector in C++）

网络重试

```
def searchBigSortedArray(self, reader, target):
    kth = 1
    while reader.get(kth - 1) < target:
        kth = kth * 2

    # start 也可以是 kth // 2, 但是我习惯比较保守的写法
    # 因为写为 0 也不会影响时间复杂度
    start, end = 0, kth - 1
    while start + 1 < end:
        mid = start + (end - start) // 2
        if reader.get(mid) < target:
            start = mid
        else:
            end = mid
    if reader.get(start) == target:
        return start
    if reader.get(end) == target:
        return end
    return -1
```

```
public int searchBigSortedArray(ArrayReader reader, int target) {
    int kth = 1;
    while (reader.get(kth - 1) < target) {
        kth = kth * 2;
    }

    // start 也可以是 kth / 2, 但是我习惯比较保守的写法
    // 因为写为 0 也不会影响时间复杂度
    int start = 0, end = kth - 1;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (reader.get(mid) < target) {
            start = mid;
        } else {
            end = mid;
        }
    }
    if (reader.get(start) == target) {
        return start;
    }
    if (reader.get(end) == target) {
        return end;
    }
    return -1;
}
```


为什么没有三分法？

Trinary Search

时间复杂度上会不会更好？
实际运行速度上会不会更快？

分的越多，系数越大

三分法实现起来更加复杂
时间复杂度是一样的
实际运行速度反而更慢了

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

Find K Closest Elements

<http://www.lintcode.com/problem/find-k-closest-elements/>

<http://www.jiuzhang.com/solutions/find-k-closest-elements/>

排序数组中找离 **target** 最接近的 **k** 个整数

如何提高代码的可读性？

```
def kClosestNumbers(self, A, target, k):
    # 找到 A[left] < target, A[right] >= target
    # 也就是最接近 target 的两个数，他们肯定是相邻的
    right = self.findUpperClosest(A, target)
    left = right - 1

    # 两根指针从中间往两边扩展，依次找到最接近的 k 个数
    results = []
    for _ in range(k):
        if self.isLeftCloser(A, target, left, right):
            results.append(A[left])
            left -= 1
        else:
            results.append(A[right])
            right += 1

    return results
```

```
def findUpperClosest(self, A, target):
    # find the first number >= target in A
    start, end = 0, len(A) - 1
    while start + 1 < end:
        mid = (start + end) // 2
        if A[mid] >= target:
            end = mid
        else:
            start = mid

    if A[start] >= target:
        return start

    if A[end] >= target:
        return end

    # 找不到的情况
    return len(A)
```

```
def isLeftCloser(self, A, target, left, right):
    if left < 0:
        return False
    if right >= len(A):
        return True
    return target - A[left] <= A[right] - target
```

```
public int[] kClosestNumbers(int[] A, int target, int k) {  
    // 找到 A[left] < target, A[right] >= target  
    // 也就是最接近 target 的两个数，他们肯定是相邻的  
    int right = findUpperClosest(A, target);  
    int left = right - 1;  
  
    // 两根指针从中间往两边扩展，依次找到最接近的 k 个数  
    int[] results = new int[k];  
    for (int i = 0; i < k; i++) {  
        if (isLeftCloser(A, target, left, right)) {  
            results[i] = A[left];  
            left -= 1;  
        } else {  
            results[i] = A[right];  
            right += 1;  
        }  
    }  
    return results;  
}
```

```
private int findUpperClosest(int[] A, int target) {  
    // find the first number >= target in A  
    int start = 0, end = A.length - 1;  
    while (start + 1 < end) {  
        int mid = start + (end - start) / 2;  
        if (A[mid] >= target) {  
            end = mid;  
        } else {  
            start = mid;  
        }  
    }  
  
    if (A[start] >= target) {  
        return start;  
    }  
  
    if (A[end] >= target) {  
        return end;  
    }  
  
    // 找不到的情况  
    return A.length;  
}
```

```
private boolean isLeftCloser(int[] A, int target, int left, int right) {  
    if (left < 0) {  
        return false;  
    }  
    if (right >= A.length) {  
        return true;  
    }  
    return (target - A[left]) <= (A[right] - target);  
}
```

Maximum Number in Mountain Sequence

<http://www.lintcode.com/problem/maximum-number-in-mountain-sequence/>

<http://www.jiuzhang.com/solutions/maximum-number-in-mountain-sequence/>

在先增后减的序列中找最大值

直接用for循环, $O(n)$ 太过复杂

比 $O(n)$ 更快 复杂度 - $O(n \log n)$ 二分
假设山峰及右边部分为X, 山峰左边部分为O
那么X的条件是什么?

休息 5 分钟

Take a break

Find Minimum in Rotated Sorted Array

<http://www.lintcode.com/problem/find-minimum-in-rotated-sorted-array/>

<http://www.jiuzhang.com/solutions/find-minimum-in-rotated-sorted-array/>

右半部分的条件

A: < 第一个数

B: <= 最后一个数

建立OOXX模型

将左半边有序数组看做O, 将右半边有序数组看做X
可以把最后一个数当成target, 找最后一个小于等于target的位置

```
def findMin(self, nums):  
    if not nums:  
        return -1  
  
    start, end = 0, len(nums) - 1  
    target = nums[-1]  
  
    # find the first element <= target  
    while start + 1 < end: # 用来控制区间大小  
        mid = (start + end) // 2  
        if nums[mid] <= target: # 如果mid位置上的数字小于等于最右端的数字时，区间向左移动  
            end = mid  
        else:  
            start = mid  
  
    return min(nums[start], nums[end]) # 最终返回start和end位置上较小的数字即可
```

```
public int findMin(int[] nums) {  
    if (nums == null || nums.length == 0) {  
        return -1;  
    }  
  
    int start = 0, end = nums.length - 1;  
    int target = nums[nums.length - 1];  
  
    // find the first element <= target  
    while (start + 1 < end) { //用来控制区间大小  
        int mid = start + (end - start) / 2;  
        if (nums[mid] <= target) { //如果mid位置上的数字小于等于最右端的数字时，区间向左移动  
            end = mid;  
        } else {  
            start = mid;  
        }  
    }  
    return Math.min(nums[start], nums[end]); //最终返回start和end位置上较小的数字即可  
}
```

Follow up: 如果有重复的数?

可以证明，无法保证在 $\text{Log}(N)$ 的时间复杂度内解决

例子：[1,1,1,1,1.....,1] 里藏着一个0

最坏情况下需要把每个位置上的1都看一遍，才能找到最后一个有0的位置

考点：能否想到这个最坏情况的例子

不是写代码！不是写代码！不是写代码！

Search in Rotated Sorted Array

<http://www.lintcode.com/problem/search-in-rotated-sorted-array/>

<http://www.jiuzhang.com/solutions/search-in-rotated-sorted-array/>

在旋转数组中找到一个数的位置

方法1： 实现简单

如何利用 Find Minimum 来解决这道题？

方法1： 实现简单

做两次二分

第一次找到最小值

然后在最小值左侧或者右侧去找 **target**

```
def search(self, A, target):
    if not A:
        return -1

    index = self.find_min_index(A)
    if A[index] <= target <= A[-1]:
        return self.binary_search(A, index, len(A) - 1, target)
    return self.binary_search(A, 0, index - 1, target)

def find_min_index(self, A):
    start, end = 0, len(A) - 1
    while start + 1 < end:
        mid = (start + end) // 2
        if A[mid] < A[end]:
            end = mid
        else:
            start = mid

    if A[start] < A[end]:
        return start
    return end
```

```
def binary_search(self, A, start, end, target):
    if end < 0:
        end += len(A)
    while start + 1 < end:
        mid = (start + end) // 2
        if A[mid] < target:
            start = mid
        else:
            end = mid
    if A[start] == target:
        return start
    if A[end] == target:
        return end
    return -1
```



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

    int index = findMinIndex(A);
    if (A[index] <= target && target <= A[A.length - 1]) {
        return binarySearch(A, index, A.length - 1, target);
    }

    return binarySearch(A, 0, index - 1, target);
}

private int findMinIndex(int[] A) {
    int start = 0, end = A.length - 1;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (A[mid] < A[end]) {
            end = mid;
        } else {
            start = mid;
        }
    }

    if (A[start] < A[end]) {
        return start;
    }
    return end;
}
```

```
private int binarySearch(int[] A, int start, int end, int target) {
    if (end < 0) {
        end += A.length;
    }
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (A[mid] < target) {
            start = mid;
        } else {
            end = mid;
        }
    }

    if (A[start] == target) {
        return start;
    }
    if (A[end] == target) {
        return end;
    }
    return -1;
}
```

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）

方法2：加大难度

在一个Rotated Sorted Array上切一刀
可以确定这一刀在左半部分还是右半部分
这一刀的两边仍然是Rotated Sorted Array

```
def search(self, A, target):
    if not A:
        return -1

    start, end = 0, len(A) - 1
    while start + 1 < end:
        mid = (start + end) // 2
        if A[mid] >= A[start]:
            if A[start] <= target <= A[mid]:
                end = mid
            else:
                start = mid
        else:
            if A[mid] <= target <= A[end]:
                start = mid
            else:
                end = mid

    if A[start] == target:
        return start
    if A[end] == target:
        return end
    return -1
```

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

    int start = 0, end = A.length - 1;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (A[mid] >= A[start]) {
            if (A[start] <= target && target <= A[mid]) {
                end = mid;
            } else {
                start = mid;
            }
        } else {
            if (A[mid] <= target && target <= A[end]) {
                start = mid;
            } else {
                end = mid;
            }
        }
    }

    if (A[start] == target) {
        return start;
    }
    if (A[end] == target) {
        return end;
    }
    return -1;
}
```

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

Find Peak Element

<http://www.lintcode.com/problem/find-peak-element/>

<http://www.jiuzhang.com/solutions/find-peak-element/>

follow up: Find Peak Element II (by 高频题冲刺班)

如果求的是最高峰或所有峰，复杂度都不可能低于 $O(n)$

上升区间右侧必有峰

下降区间左侧必有峰

谷的两边都有峰

```
def findPeak(self, A):
    # write your code here
    start, end = 1, len(A) - 2
    while start + 1 < end:
        mid = (start + end) // 2
        if A[mid] < A[mid - 1]:
            end = mid
        elif A[mid] < A[mid + 1]:
            start = mid
        else:
            return mid

    if A[start] < A[end]:
        return end
    else:
        return start
```

```
public int findPeak(int[] A) {
    int start = 1, end = A.length - 2;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (A[mid] < A[mid - 1]) {
            end = mid;
        } else if (A[mid] < A[mid + 1]) {
            start = mid;
        } else {
            return mid;
        }
    }

    if (A[start] < A[end]) {
        return end;
    } else {
        return start;
    }
}
```


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

Wood Cut

<https://www.lintcode.com/problem/wood-cut/>

<https://www.jiuzhang.com/solution/wood-cut/>

将一些原木切成 k 段等长小木头，求小木头最长是多少

为什么可以二分

如果小木头的长度可以是 l ，一定能切出 k 段更短的小木头

如果小木头的长度达不到 l ，一定切不出 k 段更长的小木头

小木头的长度不可能比最长原木更长

小木头的长度不可能比所有原木总和除 k 更长

$L = [232, 124, 456]$									
length	1	2	3	...	113	114	115	116	...
k	812	406	270	...	7	7	6	6	...

```
def woodCut(self, L, k):
    if not L:
        return 0

    start, end = 1, min(max(L), sum(L) // k)
    if start > end:
        return 0

    while start + 1 < end:
        mid = (start + end) // 2
        if self.get_pieces(L, mid) >= k:
            start = mid
        else:
            end = mid

    if self.get_pieces(L, end) >= k:
        return end
    if self.get_pieces(L, start) >= k:
        return start

    return 0

# O(n)
def get_pieces(self, L, length):
    return sum(l // length for l in L)
```

```
public int woodCut(int[] L, int k) {
    if (L == null || L.length == 0) {
        return 0;
    }

    long maxValue = 0, sumValue = 0;
    for (int i = 0; i < L.length; i++) {
        sumValue += L[i];
        maxValue = Math.max(maxValue, L[i]);
    }

    int start = 1, end = (int) Math.min(maxValue, sumValue / k);
    if (start > end) {
        return 0;
    }

    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (getPieces(L, mid) >= k) {
            start = mid;
        } else {
            end = mid;
        }
    }

    if (getPieces(L, end) >= k) {
        return end;
    }
    if (getPieces(L, start) >= k) {
        return start;
    }

    return 0;
}

// O(n)
private long getPieces(int[] L, int length) {
    long reslut = 0;
    for (int i = 0; i < L.length; i++) {
        reslut += L[i] / length;
    }
    return reslut;
}
```

Thank You

Q & A