```
Int[] arr = {3,2,3,5,6}
for (int i = 0; i <  arr.length; i++){
        SOP();
}
public int binary_search(int a[], int target){
        if(a == null || a.length = 0){
                return -1;
        }
        int left = 0;
        int right = a.length - 1;
        while(left <= right){
                middle = left + (right - left) / 2;
                if(a[mid] == target){
                        return mid;
                }else if(a[mid] < target){
                        left = mid + 1;
                }else{
                        right = mid -1;
                        }
        }
        return -1;
}

if(a == null || a.length = 0 ){
        return -1;
}
int left = 0;
int right = a.length -1;
while(left <= right){
        if(left <= right){
                mid = left + (right -left)/2;

                if(a[mid] == target){

                        return mid;
                }else if(a[mid] < target){
                        left = mid + 1;
                }else{
                        right = mid -1;
                        }
        }
}
return -1;
```

```
        L          M          R
index   0  1  2    3  4  5  6
array   1  2  3    6  7  8  9
```

target:4

Step1： 6 > 4; M > target; 需要向左边找；把右边的切掉; R = M -1; mid 并不是我想要的元素，所以我写了mid -1；

```
              L  M  R
        index 0  1  2  3  4  5  6
        array 1  2  3  6  7  8  9
```

target:4

Step2：2 < 4; M < target; 需要往右边走；把左边的切掉；L = M + 1;

```
                 LM R
        index 0  1  2  3  4  5  6
        array 1  2  3  6  7  8  9
```

target:4

Step3：3 < 4 ； M < target；需要往右边走；把左边的切掉；L = M + 1;

```
                    LM R
        index 0  1  2  3  4  5  6
        array 1  2  3  4  6  7  8  9
```

target:4

记住：target 要是 小的话， 一定往 左边移动,左边移动就得变 R；target要大的话， 一定往右边移动， 向右移动就得变 L；只看target；

theta,找不到最坏最好的

1，Class Binary Search

Question 1: Classical Binary Search

--- to find an element/number in an array, → sorted array.

Example: a[7] = 1 2 4 5 7 8 9 whether target == **4** is in this array or not.

L=M+1R =M-1

index	0	1	2	3	4	5	6
A[7]	1	2	**4**	5	7	8	9

Iteration 1: L = 0, R = 6, M = 3 A[M] == A[3] == 5 > target == 4, so R = M-1 = 2;
Iteration 2: L = 0, R = 2, M = 1 A[M] == A[1] == 2 < target == 4, so L = M+1 = 2;
Iteration 3: L = 2, R = 2, M = 2 A[M] == A[2] == 4 == target , so Done!!!

2, 为什么binary search 是log_2(n)？$\log_2(8) = 3$

因为binary search 实在sorted array上面折半查找，所以

n -> 200 -> 100->50->25-> 12->6->....2 1

2^n…… 2^3, 2^2,2^1,2^0

所以log2_（n）

2a：为什么是logn？

n， 1/2n, 1/4n， 1/((2^k)n)= 1；所以把分子分母合并一下为：

n/2^K = 1； 所以变为：2^K = n；再用指数换对数的公式：

⇒ 2^k = n; log2n = k; so k = logn;

2b: 为什么要mid+1or mid - 1； 直接是mid 不行么？看上图粉红色的提示；

2c: 什么时候L 和 R 就不需要再继续看了呢？应该是所有的口红的色号都看完了，就不需要在看了；！= 什么时候循环结束；

3，while（left < right），错的，这样的话有一个元素的时候不能进while循环，比如下面这个例子，left 和right都是0，我难道这一个元素就不看了么？我还得进while循环，继续， a[MID]：5 <7; left = mid；那我这个left 一直没有动啊，left永远等于0，等于MIDDLE;根本走不出 代码里面的 else if 的代码；

index 0

l==R==0==M target = 7

array 5

4， a[mid] < target

left = mid 如果没有mid+1，left就一直是0，一直是0

input = [5] target=6， 所以mid = 0+(1-0)/2 = 0+0=0

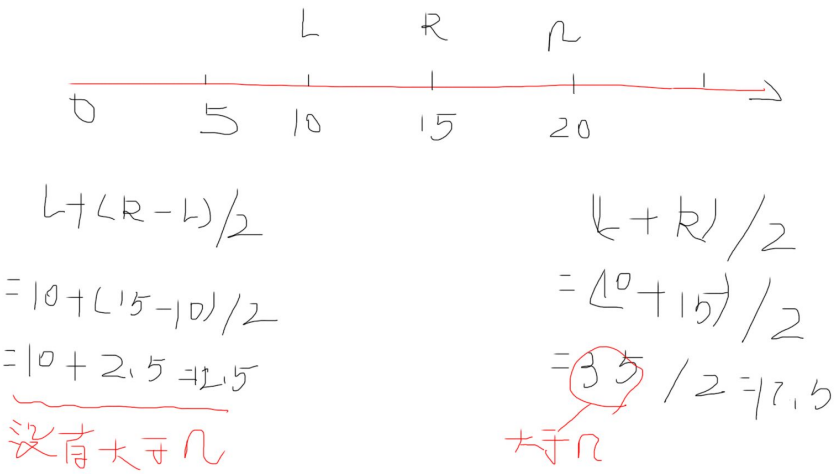
a[mid] < 6, left = mid; left 根本就没有移动呀；那不是死循环了么？

index 0 1

LM R

array 5 7

5，避免溢出：




```
public int binary_search(int [] a, int target){
        if (a == null || a.length == 0)
                return -1;
        int left = 0;
        int right = a.length -1;                    // bug 1 右边必须是length -1 否则 bounddary
        while(left <= right){                       // bug 2
             int mid = left + (right - left) / 2;
             if (a[mid] == target){
                  return mid;
             } else if (a[mid] < target){
                  left = mid + 1;                   // buy 3,4
             }
              else {
                  right = mid -1;                   // bug 5
             }
    }
```

```
        return -1;
}
```

时间复杂度：Log（n）
空间复杂度是：O（1）

面试的时候需要和面试官交流的东西：

1. Clarification
    a. 给女朋友找口红的问题， color？ int
    b. 口红是sorted么？ sorted？ ascending？
    c. duplicate？ why？
2. Examples
3. Solutions
    a. Assumptions
        i. not in the array: -1
    b. Input/output
        i. input: int []
        ii. output: int index
    c. Corner Case
        i. empty, null
    d. Algorithm
        i. binary search
    e. Time/space complexity
        i. time: logn
        ii. space: O1
4. Coding
5. Test
    a. Test corner cases: null, empty, 1, 2 elements
    b. Test general cases

2，Binary Search in sorted 2d array I

[0][0]

1 2 3 4

5 6 7 8

9 10 11 12 index = [2][3]

target:6;

N：row = 3

M：col = 4　　m*n - 1 = 3*4 -1=11 个index；

1, 怎么样拉成丝？

left = 0；right = M * N - 1；

1 2 3 4 5 6 7 8 9 10 11 12 array

0 1 2 3 4 5 6 7 8 9 10 11 index

l　　m　　r　　M = 0+ 11/ 2 = 5.5=5

2，这个mid = 5， 对应的是 二位空间里的 哪行哪列？

row ： 5/ col = 5 / 4 = 1

col： 5%col = 5 % 4 = 1， 所以找到的mid应该是1行1列，不就是6么？

这个题目用method2：convert 2d to 1d array and do binary search

```
 int cols = matrix[0].length;//?
```

这个是什么意思

1）在定义变量的时候要定义rows 和 cols

2）row = mid/cols；

  col = mid % cols；

3）时间复杂度是：O（log m * n）；空间复杂度：O（1）

```
public class Solution {
 public int[] search(int[][] matrix, int target) {
   // Write your solution here
   if( matrix.length == 0 || matrix[0].length == 0){
```

```
        return new int[] {-1,-1};
    }
    int left = 0;
    int right = rows * cols -1;
    int rows = matrix.length;
    int cols = matrix[0].length;//?
       while(left <= right){
        int mid = left + (right - left) / 2;
        //convert 2d to 1d to the col and row
        int row = mid / cols;
        int col = mid % cols;
        if(matrix[row][col]== target){
            return new int[] {row, col};
        }else if(matrix[row][col] < target){
            left = mid + 1;
        }else{
            right = mid - 1;
        }
    }
    return new int[] {-1,-1};
  }
}
```

Question 3：Closest Elements to target

1 5 7 9 11 12 T : 2
L M R

1 5 7 9 11 12 T : 2
L M R

1 5 7 9 11 12 T : 2
LM R

a：为什么不能把mid 错过去，即mid + 1？ 因为mid 在这里很有可能是最接近的那个元素，所以不能违背错过正确答案的rule；

b: 如果是left = right -1； 像上面的这个 1，5 的例子，会出现什么情况？通过跑代码，array[mid] < target； left = mid；发现每次return的都是mid； 1<2; left = mid；不能让left 和mid 在同一个位置，or right 和mid 在同一个位置，这样会死循环；所以要写 left < right -1； 还剩2个数的时候跳出来；因为会mid不会缩小，会死循环；

1) 一定是left 在Mid 左边， left < right - 1；当里面的元素，少于2个或等于2个的时候，要提前停下来。**why要提前一步停下来? 如果不提前停下来，会有dead loop；**
2) 还有post-processing：如果left < right 的绝对值， return left
3) 时间复杂度是：？

   Variant 1.1 how to find an element in the array that is closest to the target number?
4)
5) Target == 4; L = 0 M=2 R=4
6) index 0 1 2 3 4
7) // e.g. int a[5] = {1, 2, 3,4 8, 9};
8) 在这里a[m] < target；left = mid;应该往右找；变成下方
   l m r
9) index 0 1 2 3 4
10) // e.g. int a[5] = {1, 2, 3,4 8, 9}; 不能把3给ruleout 掉
    a[m] > target, right = mid;
    lm r
    // e.g. int a[5] = {1, 2, 3,4 8, 9}; 到现在为止，还是a[m] > target; 如果还是做right = mid；那么lmr都挤到3 都上去了，根本没有把2遍的差值计算一下，所以需要post-processing，在还有2个元素的时候，停下来，不要进while 循环，分别做abs.3-4 和abs.8-4，谁小返回谁；
11) while (left< right -1)
12)
13)
14)
15)
16) L M R
17) xxxxxxxxxxxxxxxxxxXxxxxxxxxTxxxxx 一看middle比Target小太多，立马把mid左边的丢掉
18)
19) L M R
20) xxxxxxxxxxxxxxxxxxXxxxxxxxXxTxxxxx 这个mid又比target小太多，果断抛弃mid左边的

21)
22) L M R
23) xxxxxxxxxxxxxxxxxxXxxxxxxxXxTx xxxx mid又比target大太多，把mid右边的果断抛弃
24)
25) L M R
26) xxxxxxxxxxxxxxxxxxXxxxxxxxXxTx xxxx mid又比target小， 果断再抛弃mid左边的一个
27)
28) LR
29) xxxxxxxxxxxxxxxxxxXxxxxxxxXxTxxxx 提前一步停下来，Left = right
30) 搜索范围不断缩小，提前一步停下来，可以看到left = right，就可以找到相邻的元素，当找到相邻的元素，一定可以找到4夹在相邻的3， 和 4 中间
31)
32) int BinarySearch (int a[], int left, int right, int target) {
33) int mid;
34) while (left < right - 1){ // **left == right - 1，**等于的时候意味着相邻，所以<小一位可以继续
35) mid = left + (right - left) / 2;
36) if (a[mid] == target){
37) return mid;
38) }else if (a[mid] < target){
39) left = mid; // mid + 1 both are not ok
40) }else{
41) right = mid; // mid - 1 wrong, but why? 因为不想相互错过
42) }
43) }
44) // post processing
45) if (Math.abs(array[left] - target) <= Math.abs(array[right] - target)){
46) return left;
47) }else{
48) return right;
49) }
50)
51) 这个例子找到最为接近的数，不能马上排除，不能太agreesive，而上一个例子是找到具体的元素
52) xxxxxxxxxx **3** _ xxxxxxxxx

53) 4
54) m L+1 就错过了3
55)
56)

Question 4: First Target

1）right = mid； 对的；因为我也不知道现在的这个mid是不是第一个5；

2）left = mid 和left + 1 都是对的，因为我现在还没有找到5，可以继续右面找

3）post-processing，先check左边的，如果左边是5那就return了

	L			M				R
index	0	1	2	3	4	5	6	7
array	2	**2**	2	3	6	7	8	9

target: 第一个2

step1: M > target; 3 > 2; target 比较 小，应该向左边移动，r = mid - 1；mid - 1 和mid 都work，但是r = mid 用的更多，没有那么agressive

	L	m	R					
index	0	1	2	3	4	5	6	7
array	2	**2**	2	3	6	7	8	9

target: 第一个2

step2: M = target; 2 = 2; return mid；但这个并不是第一个2，所以我们把while 缩小到了 <=2; 在post-processing check 一下；

为什么是a[mid] == target, right = mid;

1,我们求的是first occurance，是在最左边，当while执行的条件是有 L M R，左右两边加一个元素；当我们想要第一个2的时候，我们应该 R=M； 再往左边去一点；也可以这么理解：while 马上要终止的条件是 L M R；while再往下就不能执行了，必须是left 和right中间夹一个元素，到这时我们就要post-processing check 了；

Question 5: Last Target

Variant 1.2

// return the index of the first occurrence of an element, say 5

```
int BinarySearch (int a[], int left, int right, int target) {
      int mid;
      while(left < right -1){
            mid = left + (right - left ) / 2;
            if (a[mid] == target){
                right = mid;
            }else if(a[mid] < target){
                left = mid;   // left = mid + 1 both are ok
            }else{
                right = mid;// right = mid - 1 both are ok
            }
}
       if(a(left) == target)
            return left;
       if(a(right) == target)
            return right;
     return -1;
}
```

Variant 1.3

// return the last occurrence of an element
// eg int a[6] = {4,5,5,5,5,5};
// if target == 5; then index 5 is returned;
// if target == 10; then -1 is returned;

```
int BinarySearch (int a[], int left, int right, int target) {
       int mid;
       while(left < right -1){
            mid = left - (right - left) / 2;
            if (a[mid] == target){
                 left =mid;
            }else if(a[mid] < target){
                left = mid;// left = mid + 1 both are ok

            }else{
                right = mid;// right = mid - 1 both are ok

            }
}
      if(a[right] == target)
            return right;
      if(a[left] == target)
            return left;
       return -1;
}
```

Question 6: Closet K Elements

Variant 1.4 how to find K elements in the array that is closest to the target number?

Target == 4;

// e.g. int a[5] = {1, 2, 3, 4 8, 9}

<-L <-L <-L R

返回： 3 2 1 8

step 1: binary search to find the L and R that are adjacent to each other. closet element

step 2: 中心开花， 谁小谁移动 expanded from both left and right, move the pointer to the closest to the target

Time = O(log(n) + k)

1) 先把这个加单的数找到，是4，也就是之前讲的找到最近的算法
2) 谁最近移谁 （中心开花，谁小移谁）

Target == 4;

// e.g. int a[5] = {1, 2, 3, 4 8, 9}
<-L R-> 3 最近，打印出3，所以left --

// e.g. int a[5] = {1, 2, 3, 4 8, 9} 2和8比，2 离4 近，打印出2， 所以left--
<-L R->

// e.g. int a[5] = {1, 2, 3, 4 8, 9} 1和8比，1 离4 近，打印出1， 所以left--
<-L R-> 这时要check coner case，不能到-1

3) Time : step 1: 1st closest log(n)

step2: O(k)

Total time: log(n) + k

空间：不算输出是O1

4）如果K 接近于n，会出现： log（n）+ n = O（n）因为取worst case 是O（n）嘛，怎么办？5分钟之后揭晓？哪一个

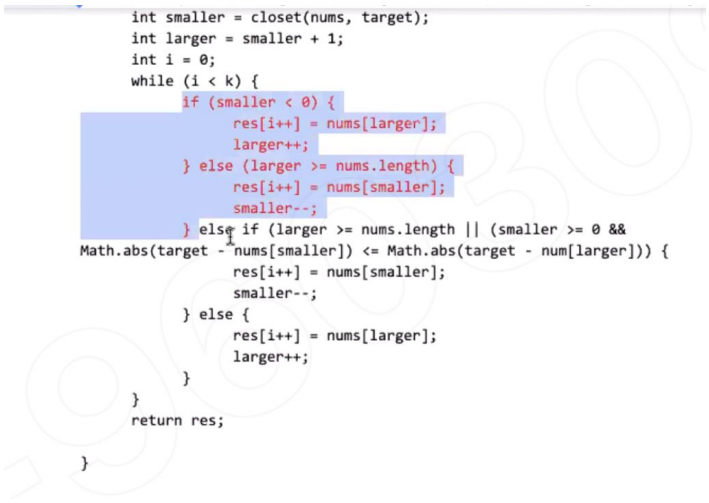
```
public class Solution {
 public int[] kClosest(int[] array, int target, int k) {
   // Write your solution here
   if(array == null || array.length == 0){
```

```
        return array;
    }
    if(k == 0){//corner case check,  k最好不要为0；
        return new int[0];
    }
    int left = largestSmallEqual(array, target);
    int right = left  + 1;
    int [] result = new int[k];// 新建一个array来存最近的几个元素
    for(int i = 0; i < k; i++){
//注意越界问题，left 和right 一直在--，++ 很有可能会越界对吧；left 很有可能变为-1；
        if(right >= array.length ||
 (left >= 0 && Math.abs(array[left]-target) <= Math.abs(array[right]-target))){
            result[i] = array[left--];
        }else{
            result[i] = array[right++];
        }
    }
    return result;
 }

private int largestSmallEqual(int[] array, int target){
    int left = 0;
    int right = array.length - 1;
    while(left < right - 1){
        int mid = left + (right - left) / 2;
        if(array[mid] == target){
            return mid;
        }else if(array[mid] < target){
            left = mid;
        }else{
            right = mid;
        }
    }
```

```
if(Math.abs(array[left] - target) <= Math.abs(array[right] - target)){
    return left;
}else{
    return right;
}
```

看一下这个图上的，中心开花谁小移动谁的判断越界的条件怎么来的；



```
int smaller = closet(nums, target);
int larger = smaller + 1;
int i = 0;
while (i < k) {
    if (smaller < 0) {
        res[i++] = nums[larger];
        larger++;
    } else (larger >= nums.length) {
        res[i++] = nums[smaller];
        smaller--;
    } else if (larger >= nums.length || (smaller >= 0 &&
Math.abs(target - nums[smaller]) <= Math.abs(target - num[larger])) {
        res[i++] = nums[smaller];
        smaller--;
    } else {
        res[i++] = nums[larger];
        larger++;
    }
}
return res;

}
```

}如果smaller跑到负数那边，就越界了，我们就得移动larger；同理larger 越界，我们就移动smaller；合并条件的时候是把larger>num.length 和 math.abs(left-target)<right - target;因为他们的条件都是移动}smaller；但是在写larger>num.length || math.abs(left-target)<right - target时，还要考虑smaller是否越界，所以把smaller>=0 写在了or 后面的语句了，而or后面的语句是small>=0&& Math.abs；即 **A || (smaller>=0 && B);** A larger越界，移动smaller；B 是Math越界，移动smaller，无论谁为ture，都会移动smaller；

Debug 老师的版本：

```
public int[] kclosest(int[] nums, int target, int k) {
      if (k > nums.length) {

      }
      int[] res = new int[k];
      int smaller = closet(nums, target);
      int larger = smaller + 1;
      int i = 0;
      while (i < k) {
            if (larger >= nums.length || (smaller >= 0 &&
Math.abs(target - nums[smaller]) <= Math.abs(target - num[larger])) {
```

```
                    res[i++] = nums[smaller];
                    smaller--;
              } else {
                    res[i++] = nums[larger];
                    larger++;
              }
        }
        return res;
}


public int closet(int[] nums, int target) {
        //corner case
        if (nums==null || nums.length==0) return -1;

        //binary search
        int left=0, right=nums.length-1;
        while (left < right -1) {    //only remain 1/2 element
              int mid = left + (right-left)/2;
              if ( nums[mid] == target) {
                    return mid;
              }
              if ( nums[mid] < target) {
                    left =mid;
              } else {
                    right = mid;
              }
        }
        if (nums[right] <= target) {
              return right;
        }
        if (nums[left] <= target) {
              return left;
```

```
        }
        return -1;
}
```

老师的版本，在找最近的元素的method 的时，用的不是先找最近的那个元素，再中心开花，谁小移动谁；老师的版本不要求掌握，把1 找最近的元素，2 中心开花谁小移动谁掌握就好

Question 7: Smallest Element that is Larger than Target

How to find the smallest element that is larger than a target number?

input[N] = {1,3,4,5,8,9} target = 7; return 8 的index

7

这个题和哪个题目相似？我觉得是closest target像，但其实是个Last Target相似

Analysis：

sssssssssssss eeeee Bbbbbbbbbbbbbbb

L R

是不是就找 找b里面最左侧的B呀？步骤是先找到b，要想找到b就得用Last Target，然后把post-processing 改为left or right > target就行了

```
int binarySearch(int [] a, int left, int right, int target){
        int mid;
        while(left < right - 1){
                mid = left + (right - left ) / 2;
                if(a[mid] <= target){
                        left = mid;
                }else if (a[mid] > target){
                        right = mid;
```

```
            }
        }
        //post-processing
        if(a[left] > target){
            return left;
        }
        if(a[right] > target){
            return right;
        }
        return -1;
}
```

Question 8 : K-th Smallest in Two Sorted Arrays

2个sorted array怎么可以找到k个最小的

A[] = {2,5,7,10,13}

B[] = {1,3,4,13,20,29}

k = 5

output = 5

Solution 1:

A[] = {2,5,7,10,13}

        i

B[] = {1,3,4,13,20,29}

        j

result: [1,2,....]

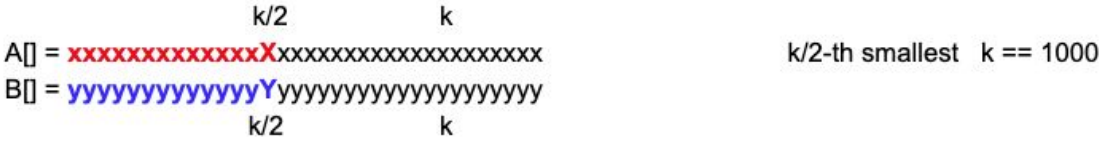
2个指针谁小移动谁，时间复杂度太大了，是O（k），如果k是n的话，都linear了太慢；不行

Solutions 2：

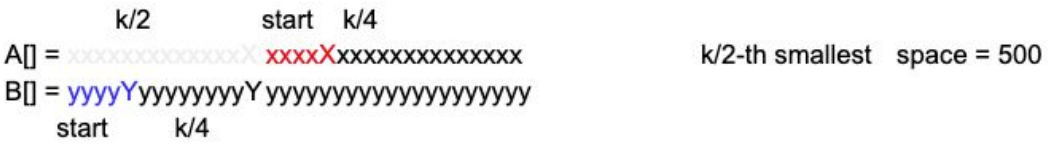
2个array，每次都从头上找，然后依次递减

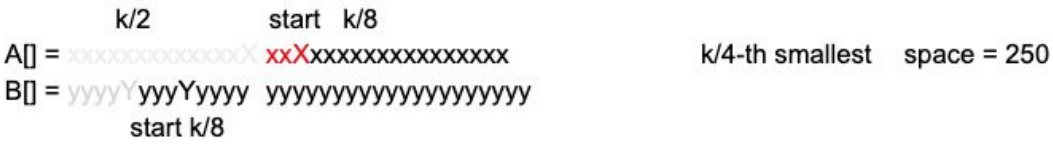
只看A和B 的前500个数，谁小就把谁抛弃掉，不放入solution

2个sorted Array run binary search，并不是从中间折半查找，而是从头上开始往后跳



k ⇒ k/2 ⇒ k/4 ⇒ k/8





k/2 + k/4 + ... = k-1

(High Level) 核心思想是什么： 把A[N]和前B[M]的各自前k/2比较，以每次删除k/2。
(Details) How to delete k/2 ? A[k/2 - 1] 和 B[k/2 - 1]**谁小就删谁**的前k/2
(Proof) Why is it correct? Because result (=k-th smallest) cannot be among A[0] -- A[k/2 -1] and **A[k/2 - 1]** must be smaller than the k-th smallest element

Time： O（logk）

1：如果A < B，那么A 里面的前500个都可以把他们放到solution里面去；

2：在从A 后面数出250 个元素，从B 中数出250个元素，比较A 和B，假设这次A>B，那么把第二部里面蓝色的y都放到solution里面

3：再算125，逐次缩小空间

Kth Smallest of Two Sorted Arrays

1. Problem:There are two sorted arrays **nums1** and **nums2** of size m and n respectively. Find the kth smallest of the two sorted arrays. You may assume **nums1** and **nums2** cannot be both empty.

Example 1:

nums1 = [1, 2]

nums2 = [3, 4], k = 2

return 2

2. Algorithm Analysis

1) High-level Idea:

```
two sorted array                    ===>
from left to right, find the kth smallest by two pointers
advance the pointer with smaller values
Time O(k) Space O(1)                ===>
Divide & Conquer
k-size problem ---> k/2-size problem ---> k==1 problem O(1)
how to reduce the problem size by k/2?
remove the part with k/2 elements ===>
A<k/2th> <= B<k/2th>, A<0~k/2th> cannot include the kth smallest
A<k/2th> > B<k/2th>,  B<0~k/2th> cannot include the kth smallest
A: ------ -(k/2)5 kth smallest
B: ------ -(k/2)
6 <= A<k/2> k/2-1 k/2 + k/2 -1 == k - 1
guess:
<===
1) pos of kth < k/2 pos ===> t (pos in A) + k/2 < k
2) pos of kth == k/2 pos ===> t + k/2 == k
A<k/2th> < B<k/2th> ===> # of ele(<= kth) in B < k/2
A: ------ -(k/2) Y 5 kth smallest
B: ------ -(k/2)
# of ele <= Y in A(k/2) + # of ele <= Y in B(??k/2) == k
A<k/2th> <= B<k/2th>
```

```java
private int kth(int[] a, int aleft, int[] b, int bleft, int k) {
  // base cases
  if (aleft >= a.length) {
    return b[bleft + k - 1];
  }
  if (bleft >= b.length) {
    return a[aleft + k - 1];
  }
  if (k == 1) {
    return Math.min(a[aleft], b[bleft]);
  }

  // Since index starts from left,
  // the k/2-the element should be left + k/2 - 1
  int amid = aleft + k / 2 - 1;
  int bmid = bleft + k / 2 - 1;
  // why is correct?
  // if a.size too small, then remove elements from b first.
  int aval = amid >= a.length ? Integer.MAX_VALUE : a[amid];
  int bval = bmid >= b.length ? Integer.MAX_VALUE : b[bmid];

  if (aval <= bval) {
    return kth(a, amid + 1, b, bleft, k - k / 2);
  } else {
    return kth(a, aleft, b, bmid + 1, k - k / 2);
  }
}
```

2) Implementation Trick

i. k/2th element index

mid = binary search starting index + k / 2 - 1(k is 1-based)

ii. index out of bound for one array A

A: ------ -(k/2) Y 5 kth smallest

B: ------ -(k/2)

kth smallest

A: --- - 3 k = 1000 500

B: ---------- 10000

A<k/2> >= B<k/2>

the kth element must be in the left part of A and the right part of B

===>

val == Max value if out of bound : real val

ii. base case

- k == 1, only one, pick the smaller one
- A run out of elements, directly find the k-th element in B

```
/*
assumption:
A, B sorted ascending order
A empty, find kth in B
```

```
B empty, find kth in A
corner case
k <= 0, or input == null --> exception
solution 1: one by one checking
time O(k) space O(1)
solution 2: divide and conquer
time O(logk) space O(logk)
* */
class Solution {
  public int kth(int[] A, int[] B, int k) {
    if (A == null || B == null || k <= 0) {
      throw new IllegalStateException("Invalid Input");
    }
    return recurKth(A, 0, B, 0, k);
  }
  //start from a, b find kth smallest
  private int recurKth(int[] A, int a, int[] B, int b, int k) {
    //corner cases
    if (a >= A.length) return B[b + k - 1];
    if (b >= B.length) return A[a + k - 1];
    //base case
    if (k == 1) return Math.min(A[a], B[b]);
    //divide part
    int aMid = a + k / 2 - 1, bMid = b + k / 2 - 1;
    //corner case: run out of elements
    int aVal = aMid >= A.length ? Integer.MAX_VALUE : A[aMid];
    int bVal = bMid >= B.length ? Integer.MAX_VALUE : B[bMid];
    if (aVal <= bVal) {
      return recurKth(A, aMid + 1, B, b, k - k / 2);
    } else {
      return recurKth(A, a, B, bMid + 1, k - k / 2);
    }
  }
}
A: ------ -(k/2) -------
   a       amid amid+1
B: ------ -(k/2)
   b
```

```java
private int kth(int[] a, int aleft, int[] b, int bleft, int k) {
  // base cases
  if (aleft >= a.length) {
    return b[bleft + k - 1];
  }
  if (bleft >= b.length) {
    return a[aleft + k - 1];
  }
  if (k == 1) {
    return Math.min(a[aleft], b[bleft]);
  }

  // Since index starts from left,
  // the k/2-the element should be left + k/2 - 1
  int amid = aleft + k / 2 - 1;
  int bmid = bleft + k / 2 - 1;
  // why is correct?
  // if a.size too small, then remove elements from b first.
  int aval = amid >= a.length ? Integer.MAX_VALUE : a[amid];
  int bval = bmid >= b.length ? Integer.MAX_VALUE : b[bmid];

  if (aval <= bval) {
    return kth(a, amid + 1, b, bleft, k - k / 2);
  } else {
    return kth(a, aleft, b, bmid + 1, k - k / 2);
  }
}
```

Question 9： K-th cloest element(Q6 2nd Solutions) 用log（n）时间复杂度，也就是2个array

How to find closest k elements in the array that is **closest** to a target number?

k = 3

Target == 4; **L=2 R=3**

index **0 1 2 3 4**

```
// e.g. int a[5] = {1, 2, 3,48, 9};
                       ← L  R →
```

Hint: How is this question related to **Question 8 K-th Smallest in Two Sorted Array**?

k/2

A[] = xxxxxxxxxxxxxXxxxxxxxxxxxxxx k/2-th smallest

B[] = yyyyyyyyyyyyyyYyyyyyyyyyyyyyyyyyyyy

Target == 4; **L=2 R=3**

index 0 1 2 3 4

```
// e.g. int a[5] = {1, 2, 3, 8, 9};
                        ← L   R →
```

left half = { 3 2 1}

right half = {4, 5}

Step1: run binary search to place L R

Step2: Q8 solution

Log(n) + Log(k) → Log(n)

Step1：run binary search to place LR，也就是第6题找到closed element

Step2：在脑袋里面人工的把这个array给断开，如何在这2个array里面run binary search呀？然后就是left half{3 2 1} right half{4，5} run k/2，也就是question 8 的步骤

时间复杂度： Log（n）正好回答了第六题的疑问

Question 10: given a sorted dictionary with unknown size, how to determine whether a word is in this dictionary or not:

Example: dictionary[x] = {1 3 5 9 ...10000 ..1000000000…...}
target == 9999
Assumption if a[index] == Null then we know the size of dictionary is index;

Step1: 跳一步
step 2：跳2步
step 3： 跳2^2 = 4
step 4: 跳 2^3
-
--

----------2^n -1
------------------------------ 2^n

xxxxxxxxxxxxxxxxxxYxxxxxxxxxxX size = n, 为了找到右边的边界，最终要耗费多少时间
TIME：O(log_2(n)), 因为每次都是折半折半，2倍的跳
Solution1:
Step1: do for loop to keep jumping 2^i steps,until we jump out of the boundary
Step2: run binary search [0, 2^r] to find the solution log_2(n)

```
/*
*  interface Dictionary {
*    public Integer get(int index);
*  }
*/

// You do not need to implement the Dictionary interface.
// You can use it directly, the implementation is provided when
testing your solution.
public class Solution {
 public int search(Dictionary dict, int target) {
   // Write your solution here
   if(dict == null){
       return -1;
   }
   int left = 0;
```

```
    int right = 1;
    //find the right boundary for binary serarch
    //extends untill we are sure the target is within in
[left,right]range.
    while(dict.get(right) != null && dict.get(right)<target){//
只要是右边界小于target，我们就一直可以进whileloop，当while 条件不成立的
时候就代表，jump out of boudary了，
        //move left to right
        //double right index
        left = right;
        right = 2 * right;
    }
    return binarySearch(dict,target,left,right);
 }
 private int binarySearch(Dictionary dict, int target, int
left, int right){

     //classical binary search

     while(left <= right){
         int mid = left + (right - left)/2;
         if(dict.get(mid) == null || dict.get(mid) > target){
             right = mid -1;
         }else if(dict.get(mid)<target){
             left = mid + 1;
         }else{
             return mid;
         }
     }
     return -1;
 }
```

```
}
```

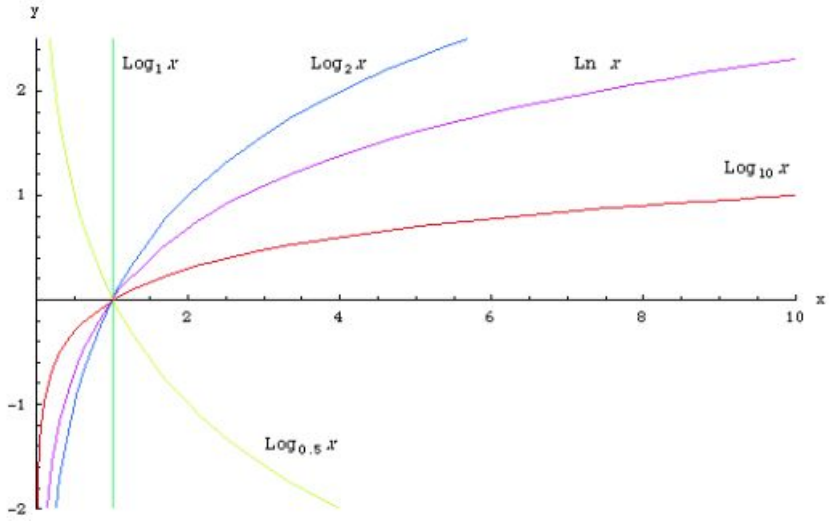
solution Follow up
jump by 2 times each time
how about jump by 10 times each time, 哪个好，为什么？
定性分析
1 to jump out, 10 times jump is better，肯定找的快呀，但是一点跳出去了，在往回找的时候，2倍的时间是不是好一点？能够更仔细的找到那个字
2 to jump in,　2 times jump is better, because the over-shot distance is shorter
定量分析

	2 times	10times
To jump out	log_2(n)	log_10(n)
To jump in	log_2(2n)	log_2(10n) 往里跳永远是折半跳

2 times - 10times =, <, >
log_2(n) + log_2(2n) -(log_10(n) + log_2(10n)) < 0　　2-10 <0 谁好？ 2 好
log_2(n) + log_2(2n) -(log_10(n) + log_2(10n)) == 0　　2 好
log_2(n) + log_2(2n) -(log_10(n) + log_2(10n)) > 0　　10好

10 times - 2 times =
（ log_10(n) + log_2(10n)）- （ log_2(n) + log_2(2n) ）< 0
= log_10(n) - log_2(n) + log_2(10n) - log_2(2n)
= 通过看图应该是个负数 + log_2(5)
= -10000 + 2.5 < 0
=》10times - 2 times < 0
=> 10 times < 2 times
10倍的时间< 2 倍的时间， 10倍的快

Conclusion: So jump 10 steps is faster than 2 steps!



Question 1，Class Binary Search

Question 2，Binary Search in sorted 2d array I

Question 3：Closest Elements to target ： 1：注意不能相互错过

Question 4: First Target

Question 5: Last Target

Question 6: Closet K Elements：1：先找到加单的数 2：中心开花谁小移动谁

Question 7: Smallest Element that is Larger than Target：ssssssssssss eeeee

Bbbbbbbbbbbbbbb； 1：Last Target； 2： post-processing left， right > target

Question 8 : K-th Smallest in Two Sorted Arrays 1：不会

Question 9： K-th cloest element(Q6 2nd Solutions) 用log（n）时间复杂度，也就是2个array； Step1：run binary search to place LR，也就是第6题找到closed element

Step2：在脑袋里面人工的把这个array给断开，如何在这2个array里面run binary search呀？然后就是left half{3 2 1} right half{4，5} run k/2，也就是question 8 的步骤

时间复杂度： Log（n）正好回答了第六题的疑问

Question 10: given a sorted dictionary with unknown size, how to determine whether a word is in this dictionary or not:

==========

对于ListNode， LinkedList， newArray空间复杂度的理解：

1，recursion Linkedlist

时间： On

空间：heap：

stack：只要没有hit 到base case，压到栈里面的东西，还是存在的，一个栈是O1，那么我要压多少次？不是要压n次么？不就是On么？存100个元素的stack和10万个元素的stack当然不一样了；

2， ListNode node1 = new ListNode(11);

ListNode node2 = new ListNode(12);

ListNode node3 = new ListNode(13);

空间复杂度：3个 node嘛，

3， new ListNode 和newLinkedList 空间复杂度分别是多少？

Deque<Integer> q = new LinkedList<>();

在这里的 linkedlist 的空间复杂度是 O1，因为我有一个linkedlist，但里面什么都没有啊，是个空的；linkedlist是一个背包，背包里面什么都没有；但这个背包本身是存在的；LinkedList的空间复杂度取决于里面包含的元素个数，我刚new 出linkedlist的时候是空的，没有东西呀，所以空间是O1；

4，同理可得 listNode，当我new 出来一个listNode时，我只是new 出来了一个结点，一个结点就只占 O1的空间嘛；

5， array 在heap的空间复杂度是On，因为array的特点是，size不能change，一开始长度定下来了，这个array就包含这么多东西，比如，一个array，size 是n，一开始这个array就默认有n个元素，这n个元素的初始值默认为0；reference 初始值是null；无论哪一个，这个array要占用这么多空间嘛，我要new 的时候，我要指定它的大小嘛，要制定array所占用

的空间嘛；但是我在 creat 一个arrayList or LinkedList 的时候，一开始是空的呀，空的就可以理解为，只有很小的空间耗用，O1，它里面没有任何元素；

6，new array的空间复杂度之所以是n，是因为这个size，我们在new 一个array的时候，这个array 本身的属性就有size，而这个size 里面具体有多少个元素要看题目，一般都是给 int n，也就是n个元素；这样来说的话，我只要new array，这个array 本身属性有带有了n个元素，所以是On；

7，简单来说，new ListNode， new linkedlist，new arraylist， 空间都是O1，因为我new出来的东西是空的，只占很小的一部分空间；而new array时，根据array本身size不能改变的属性，当我new 一个array时，这个array就已经划分好了元素的size，不管有n个还是2个，只要new array，他就在heap里面划分好区域来存储array size了，0 或者null，这个array并不是空的，所以时间复杂度是On；

8， reference 是指向 下一个结点，而不是++；