Q1. Binary Search Related Problems:

还有好多变种，用到binary search 的各种variant. e.g.,

1.1) two sorted integer arrays, how to find the **median** of the two arrays. （和1.2是一个题）

1.2) two sorted integer arrays, how to find the k-th smallest element from them.

Example Input:

```
A[] = {2, 5, 7, 10, 13}
B[] = {1, 3, 4, 13, 20, 29}
k = 5
```

Output: 5

Solution1:

Two pointers i and j，谁小移谁。　　O(k)

Solution2:

A[] = xxxxxxxxxX　　xxxxxXxxxxx　　xxxxxxx　　k/2-th smallest

B[] = yyyyYyyyyy　　yyyyyyyyyyy　　yyyyyyy

(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]

```java
int findKthSmallest(int[] a, int aLeft, int[] b, int bLeft, int k) {
  if (aLeft >= a.length)
    return b[bLeft + k - 1];  // base case 1: if nothing left in a;
  if (bLeft >= b.length)
    return a[aLeft + k - 1];  // base case 2: if nothing left in b;
  if (k == 1)
    return Math.min(a[aLeft], b[bLeft]);  // base case 3

// Since index is from 0, so the k/2-th element should be = left + k/2 - 1
// why is correct? if a.length too small, then remove elements from b first.
  int aHalfK = aLeft + k/2 - 1 < a.length ? a[aLeft + k/2 - 1] : Integer.MAX_VALUE;
  int bHalfK = bLeft + k/2 - 1 < b.length ? b[aLeft + k/2 - 1] : Integer.MAX_VALUE;

  if (aHalfK < bHalfK) {
    return findKthSmallest(a, aLeft + k/2, b, bLeft, k - k/2);
  } else {
    return findKthSmallest(a, aLeft, b, bLeft + k/2, k - k/2);
  }
}
```

Class 2 Binary Search Variant 1.4 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, 8, 9};
```

L R →

solu = {3, 2, 1}

Solution:

Step1: we first move L and R by using binary search to make it close to the target number, until there are two or less elements in between L and R.

Step2: 谁小移谁

TIme = O(log(n) + k)

How to solve it in O(log(n) + log(k)) time?

A[] = 3 2 1 ⇒ 1 2 3

B[] = 8 9 ⇒ 4 5

Q2: (Array) Sliding window of size k, always return the **max element** in the window size.

```
index 0 1 2 3 4 5 6 7 8                    k == 3
      1 3 2 5 8 9 4 7 3,
        _____
```

Possible ways:

- DP
- BFS2
- Heap

Solution 1: MaxHeap

Initialization: insert all first k elements into the maxheap.

Then: When the sliding window moves to the right by 1 step…

- 1 new element (from right side) comes in ⇒ MaxHeap.push(X)
- 1 left-most element should be removed from the sliding window
  (but we can temporarily keep it in the heap until it becomes the top element in the heap)

Heap = {1 3 2 5}

Lazy deletion: when we want to call MaxHeap.top(), the only thing we should be careful about is to check whether this top element's index is < left border of the sliding window. If so, keep popping it out.

```
class Element {
        int value;        // as the key in the maxheap
        int index;
}
```

For each sliding: worst case time = O(log(n)) best case = O(log(k))

Total time for sliding the window (n-k) times = O((n-k) * log(n)) (n-k) log(k)

Solution 2: Deque

```
index 0 1 2 3 4 5 6 7 8                    k == 3
      1 3 2 5 8 9 4 7 3,
```

deque = { 7 ← 3 }

We must maintain all the values in the dequeue to keep them in a **descending** order. WHY????? Becasue when a new element X comes in, if it is bigger and

newer than the right most element r, then r cannot be the solution whatsoever, so we can just delete r.

So the dequeue[**left-most**] is the final result to return whenever the window slides one step to the right.

Time for each move (amortized time) = O(1)
Total time for all moves = O(n)

Q3 The Trick of using a combination of data structures?

Q3.1 How to design a LRU cache?
Least Recently Used (LRU) Cache

request_2 = {
country = can;
date_range = <yesterday>
gender = f;
language = en;
age =<18-38>
....
}
p1
p2
.l..
pn

Cache_size = 5000 ← 5001
<key = request_1, value = result1>
<key = request_2, value = result2>
...

<key = request_5001, value = result5001>

cache hit / cache miss

Use case:

1. Somehow store 5000 elements. (any data structure can do that)
2. Whenever we see a new request, we need to find out quickly whether this request is in the cache or not. (hash set/hash map)
3. (If cache hit) We need to adjust the priority of an entry in the cache efficiently. (DoublyLinkedList: O(n) to find X)

   Xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx

   newest oldest
4. (If cache miss) We need to add a new entry to the cache, and maybe delete the oldest entry from the cache. (LinkedList)

HashMap< key = request , value = reference to the ListNode >

```
class ListNode {
        String result;
        ListNode prev, next;
}
```

Q3.2 Given an unlimited stream of characters, find the **first** non-repeating character from stream. You need to tell the first non-repeating character in **O(1)** time at any moment.

```
index 0  1  2 3 4  ..............
data  a1 b1 c1 d  a2          c2        c3    b2
solu  a  a  a a   b1                          d
```

Use cases:

1. **Non-repeating.** We need to somehow record what kind of letters have appeared. (hash map)
2. **Which one is the first?** When a new element comes in: (DoublyLinkedList)
    a. we have a new solution candidate
    b. our current solution has changed ⇒ we need to update the solution to the next one
    c. although our current solution is unchanged, one of the solution candidates may be invalid ⇒ we need to delete it from the solution candidate.

HashMap< key = char , value = ListNode >

N1(a) <-> N2(b) <-> N3(c) <-> N4(d)

<a, N1>
<b, N2>
<c, N3>

Use cases:

1. **Non-repeating.** We need to somehow record what kind of letters have appeared. (hash map)
2. **Which one is the first?** When a new element comes in: (DoublyLinkedList)
    a. we have a new solution candidate
    b. our current solution has changed ⇒ we need to update the solution to the next one
    c. although our current solution is unchanged, one of the solution candidates may be invalid ⇒ we need to delete it from the solution candidate.

HashMap< key = char , value = ListNode >

N4(d)
head

<a, null> Case 1: If a is NOT in the HashMap ⇒ a has never appeared
Case 2: If a is in the HashMap with non-null value ⇒ a has appeared exactly once
Case 3: If a is in the HashMap with null value ⇒ a has appeared more than once
<b, null>
<c, null>
<d, N4>

Question4: voting algorithm

Q4.1 给一个integer array，允许duplicates，而且其中某个未知的integer的 duplicates的个数占了整个array的一大半(> 50%)。如何有效的找出这个integer?

Solution1:

Sort the array and return the middle element O(nlogn)

xxxxxxxxxxxxxxxXxxxxxxwzlfkjalsdkjf

Solution2:

hash_map <key = number, value = counter>

Time = O(n) Space = O(n)

Solution3:

Time = O(n) Space = O(1)

A	C
A	D
A	B
A	B
...	...
A	

Maintain a pair <E1 = candidate, Value = counter>

When a new element X comes in ,

If counter == 0, just set <E1 = X, and counter == 1>

Else

Case1: if X == candidate, counter++;

Case2: else, counter--,

Q4.3 what about **而且其中某个未知的integer的 duplicates的个数占了整个array的 > 1/k.**
need k-1 candidates.

A B C D ,... K

A B D

....

A

<candidate1, counter1 = 1>
<candidate2, counter2 = 2>
<candidate3, counter1 = x>
<candidate4, counter1 = y>

....

<candidate_k-1, counter1 = w>

Step1 When a new element X comes in, we check whether X is one of the keys in the hash_map,

Case1: if so, we do <x, counter_x ++>

Case2:

2.1) if the size of the hash_table == k-1, we decrement all values of all keys in the hash_map, if any counter's value == 0, we just remove the entry from the hash_table.

2.2) else if the size of hash_table < k-1, we just insert X into the hash_table,

Step2 Iterate over the array again, and count all frequency of candidate 1 to candidate k-1 to find which candidate has a counter that is larger than (1/k) * n

Time = O(n * k)

Space = O(k)