Problem 1-

Intuition-

Table: RequestAccepted

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| requester_id   | int     |
| accepter_id    | int     |
| accept_date    | date    |
+----------------+---------+
```

(requester_id, accepter_id) is the primary key (combination of columns with unique values) for this table.
This table contains the ID of the user who sent the request, the ID of the user who received the request, and the date when the request was accepted.

Write a solution to find the people who have the most friends and the most friends number.

The test cases are generated so that only one person has the most friends.

The result format is in the following example.

Example 1:

Input:
RequestAccepted table:

```
+--------------+-------------+-------------+
| requester_id | accepter_id | accept_date |
+--------------+-------------+-------------+
| 1            | 2           | 2016/06/03  |
| 1            | 3           | 2016/06/08  |
| 2            | 3           | 2016/06/08  |
| 3            | 4           | 2016/06/09  |
+--------------+-------------+-------------+
```

Output:

```
+----+-----+
| id | num |
+----+-----+
| 3  | 3   |
+----+-----+
```

Explanation:
The person with id 3 is a friend of people 1, 2, and 4, so he has three friends in total, which is the most number than any others.

Solution –

```sql
WITH requester_count AS (
    SELECT COUNT(requester_id) AS req_count, requester_id
    FROM requestaccepted
    GROUP BY requester_id
),
accepter_count AS (
    SELECT COUNT(accepter_id) AS acc_count, accepter_id
    FROM requestaccepted
    GROUP BY accepter_id
```

```sql
),
combined_table as (
    select req_count, requester_id, accepter_id,coalesce(acc_count, 0) as acc_count
    from requester_count left join accepter_count on requester_id = accepter_id
    union
    select coalesce(req_count, 0) as num2, requester_id, accepter_id, acc_count
    from requester_count right join accepter_count on requester_id = accepter_id
)
select case
        when requester_id is null then accepter_id
        when accepter_id is null then requester_id
        else requester_id
        end as id,
        (req_count + acc_count) as num
from combined_table
where (req_count + acc_count) = (select max(req_count + acc_count) from combined_table);
```

Problem 2-

Intuition

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.

Example 2:

Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

Example 3:

Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Constraints:

- n == nums.length
- 1 <= n <= 5000
- -5000 <= nums[i] <= 5000
- All the integers of nums are **unique**.
- nums is sorted and rotated between 1 and n times.

Solution –

```
class Solution {
    public int findMin(int[] nums) {
        int right = nums.length - 1;
        int left = 0, mid = 0;
        int min = nums[0];
        while(left <= right){
            mid = (left + right)/2;
            if(nums[left] <= nums[mid])
```

```
            {
                min = Math.min(nums[left], min);
                left = mid + 1;
            }else{
                min = Math.min(min, nums[mid]);
                right = mid - 1;
            }
        }
        return min;
    }
}
```

Problem 3-

Intuition-

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,4,4,5,6,7] might become:

- [4,5,6,7,0,1,4] if it was rotated 4 times.
- [0,1,4,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums that may contain **duplicates**, return *the minimum element of this array*.

You must decrease the overall operation steps as much as possible.

Example 1:

Input: nums = [1,3,5]
Output: 1

Example 2:

Input: nums = [2,2,2,0,1]
Output: 0

Constraints:

- n == nums.length
- 1 <= n <= 5000
- -5000 <= nums[i] <= 5000
- nums is sorted and rotated between 1 and n times

Solution –

```
class Solution {
    public int findMin(int[] nums) {
        int left = 0, right = nums.length - 1, mid = 0;
        while(left <= right){
            mid = left + (right - left)/2;
            // System.out.println("left = "+left+" mid = "+mid+" right = "+right);

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

```
    }
}
```

Problem 4-

Intuition-

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

Input: nums = [1,2,1,3,5,6,4]
Output: 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:

- 1 <= nums.length <= 1000
- -2^{31} <= nums[i] <= $2^{31} - 1$
- nums[i] != nums[i + 1] for all valid i.

Solution-

```
class Solution {
    public int findPeakElement(int[] nums) {
        int len = nums.length;
        if(len == 1) return 0;
        int right = len - 1;
        int mid = 0;
        int left = 0;
        while(left <= right){
            mid = left + (right - left)/2;
            if(mid == 0){
                if(mid + 1 < len && nums[mid] > nums[mid + 1]) { return mid;}
                else { left = mid + 1; }
            }else if(mid == len - 1){
                if(mid - 1 >= 0 && nums[mid] > nums[mid - 1]){ return mid;}
                else{ right = mid - 1; }
            }else if(nums[mid] > nums[mid + 1] && nums[mid] > nums[mid - 1]){
                return mid;
            }else if(nums[mid] < nums[mid + 1]){
```

```
                left = mid + 1;
            }else{
                right = mid - 1;
            }
        }
        return mid;
    }
}
```