1.

given an integer array, find the most k largest elements in it. duplicates numbers will be counted once.

```
    eg.  [1, 0, 19, 7, 7, 8]  k = 3  →     19, 8, 7
    eg. [8, 7, 6, 9, 8, 3, 4, 1]  k = 2  → 9, 8
    eg. [8 7 4 9 1 8]   k = 3   ---> 9 8 7
```

idea

Top k problen normally we can maintain a heap of size k

```
// 1
public static int[] getTopk(int[] nums, int k){
    int[] res = new int[k];
    if (nums == null || nums.length == 0) {
        return new int[0];
    }
    Queue<Integer> pq = new PriorityQueue<>(); // min heap
    Set<Integer> set = new HashSet<>(); // deduplicate
    for (int n : nums) {
        if (!set.contains(n)) {
            set.add(n);
            pq.offer(n);
            if (pq.size() > k) {
                pq.poll();
            }
        }
    }

    int i = k - 1;
    while (!pq.isEmpty()) {
        res[i--] = pq.poll();
    }

    return res;
}
```

2.

given a 2D (N*M) integer array, find the first column that contains 1. Notice that for each row, if [i, j] = 1 then every element afterwards in the same row will be 1. like [i, j+ 1] [i, j + 2], [i, j + 3]… Need to solve this in O(N + M)

```
eg.
[
[0 0 0 1 1 1]
[0 0 1 1 1 1]
[0 0 0 0 0 1]
]
The first column that contains 1 is 3rd column. return 3

[
[0 0 1 1]
[1 1 1 1]
[0 0 0 1]
]
The first column that contains 1 will be 1st column.  return 1

[
[0 0]
[0 0]
]
I return -1

[
[1 1]
[1 1]
]
return 1
```

idea

similar to LC 74. Search a 2D Matrix

```
// 2
public static int getFirstColumnContainsOne(int[][] nums) {
    if (nums == null || nums.length == 0 || nums[0].length == 0) {
        return -1;
    }
    int r = 0;
    int c = nums[0].length - 1;
    while (r < nums.length && c >= 0) {
        if (nums[r][c] == 1) {
            c--;
        } else {
            r++;
        }
    }

    return c + 2 > nums[0].length ? -1 : c + 2;
    // returning -1 if none is found.
}
```

3.

given an unsorted array, traverse all subsets. And calculate the amount of “perfect subset”. Assume there is no duplicate in the array.
“perfect subset” means the largest number in it equals to the sum of other numbers .

```
[2 3 1 6] --->  2 + 3 + 1 = 6  This is perfect
[3 1 4] ---> 3 + 1 = 4  This is perfect.
[1 2] ---> 1 != 2  This is not perfect.
```

eg. intput: [2 7 1 3 5]

perfet subsets will be [1, 2, 3], [2, 3, 5], [2, 5, 7] so the return value will be 3.

idea

Similarly to 3 sum, we fix one number and reduce the problem to 2 sum.
So we fix the largest number as target, then the rest is a combination sum problem.

```
// 3
public static int getPerfectCount(int[] nums) {
    Arrays.sort(nums);
```

```
        int count = 0;
        for (int i = nums.length - 1; i >= 0; i--) {
            count += combinationSum(nums, 0, i - 1, nums[i]);
        }

        return count;
    }

    private static int combinationSum(int[] nums, int start, int end, int target) {
        int[] f = new int[target + 1];
        f[0] = 1;
        for (int i = start; i <= end; ++i) {
            for (int  j = target; j >= nums[i]; j--) {
                f[j] += f[j - nums[i]];
            }
        }

        return f[target];
    }
```

4.

Imagine you have a special keyboard with the following keys:

- Key 1: (A): Print one 'A' on screen.
- Key 2: (Ctrl-A): Select the whole screen.
- Key 3: (Ctrl-C): Copy selection to buffer.
- Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.

```
Example_1 :
Input: N = 3
Output: 3
Explanation:
We can at most get 3 A's on screen by pressing following key sequence:
A, A, A
Example_2 :
Input: N = 7
Output: 9
Explanation:
We can at most get 9 A's on screen by pressing following key sequence:
```

```
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
```

idea

this takes me the most of the time. i have the instinct of this would be a dp problem.
I list the solution from 0 to 7 and found out if 0 <= N <= 6, I should return N.
Then for input 7 I have f[7] = 9

N	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**		
A	0	1	2	3	4	5	6	?		
						A	C	P	->f[4]+f[4]*1P	f[4]*2=8
					A	C	P	P	->f[3]+f[3]*2P	f[3]*3=9
				A	C	P	P	P	->f[2]+f[2]*3P	f[2]*4=8
			A	C	P	P	P	P	->f[1]+f[1]*4P	f[1]*5=5
		A	C	P	P	P	P	P	->f[0]+f[0]*5P	f[0]*6=0

Then for input 8 I have f[8] = 12

N	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
A	0	1	2	3	4	5	6	9	?	
							A	C	P	f[5]*2=10
						A	C	P	P	f[4]*3=12
					A	C	P	P	P	f[3]*4=12
				A	C	P	P	P	P	f[2]*5=10
			A	C	P	P	P	P	P	f[1]*6=6
		A	C	P	P	P	P	P	P	f[0]*7=0

*A for ctrl + All, C for ctrl + Copy, P for ctrl + Paste

So I have f[i], meaning if I can press i times, the maximum number of 'A's I can get is f[i]. So we return f[N]. The base cases are from 0 to 6 where f[i] = i. Starting from f[7], we need to see all the f[j] before f[i] and use f[j] * (i - j - 1) and get the maximum number. Finally we return f[N].

```
// 4
public static int maxA(int N) {
    if (N < 7) {
        return N;
    }

    int[] f = new int[N + 1];
    for (int i = 0; i < 7; i++) {
        f[i] = i;
    }


    for (int i = 7; i <= N; i++) {
        for (int j = i - 3; j >= 0; j--) {
            f[i] = Math.max(f[i], f[j] * (i - j - 1));
        }
    }

    return f[N];
}
```

5.

Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes. Left boundary is defined as the collection of nodes, each of which is the leftmost node of the same level(depth) in the binary tree. Right boundary is defined as the collection of nodes, each of which is the rightmost node of the same level(depth) in the binary tree

```
Example_1:
    ____1_____
   /          \
  2            3
   \          /
    5        6
   / \      / \
  7   8    9  10

Ouput:
[1,2,5,7,8,9,10,6,3]

Explanation:
The left boundary are node 1,2,5,7.
The leaves are node 7,8,9,10.
The right boundary are node 1,3,6,10..
So order them in anti-clockwise without duplicate nodes we have:
[1,2,5,7,8,9,10,6,3].

Example_2 :
Input:
  1
   \
    2
   / \
  3   4

Ouput:
[1, 2, 3, 4]
Example_1:
Explanation:
The left boundary are node 1,2,3.
The leaves are node 3 and 4.
The right boundary are node 1,2,4.
Note:
the anti-clockwise direction means you should output reversed right boundary.
So order them in anti-clockwise without duplicates and we have [1,2,3,4].
```

idea

similar to LC 199. Binary Tree Right Side View, we get left and right boundary. And then we get all the leaves. Finally we remove duplicates and return.

```
    // 5
    public static List<Integer> boundaryOfBinaryTree(TreeNode root) {
        List<Integer> leftBoundary = sideView(root, true);
        List<Integer> leaves = getLeaves(root);
        List<Integer> rightBoundary = sideView(root, false);

        // post process, deduplicate
        List<Integer> res = new ArrayList<>();
        Set<Integer> set = new HashSet<>();
        for (int i : leftBoundary) {
            if (!set.contains(i)) {
                res.add(i);
                set.add(i);
            }
        }

        for (int i : leaves) {
            if (!set.contains(i)) {
                res.add(i);
                set.add(i);
            }
        }

        for (int i : rightBoundary) {
            if (!set.contains(i)) {
                res.add(i);
                set.add(i);
            }
        }

        return res;
    }

    private static List<Integer> getLeaves(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) {
            return res;
        }
        if (root.left == null && root.right == null) {
            res.add(root.val);
            return res;
        }

        List<Integer> left = getLeaves(root.left);
        List<Integer> right = getLeaves(root.right);
        res.addAll(left);
        res.addAll(right);
        return res;
    }

    private static List<Integer> sideView(TreeNode root, boolean isLeft) {
        List<Integer> res = new ArrayList<>();
        dfsHelper(res, root, 0, isLeft);
        if (!isLeft) {
            Collections.reverse(res);
        }
        return res;
    }

    private static void dfsHelper(List<Integer> res, TreeNode root,
                                     int level, boolean isLeft) {
        if (root == null) {
            return;
        }

        if (res.size() == level) {
            res.add(root.val);
        }

        if (isLeft) {
            dfsHelper(res, root.left, level + 1, isLeft);
        }
        dfsHelper(res, root.right, level + 1, isLeft);
        if (!isLeft) {
            dfsHelper(res, root.left, level + 1, isLeft);
        }
    }

    static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        public TreeNode(int val) {
            this.val = val;
        }
    }

    private static TreeNode initNode1() {
        TreeNode n1 = new TreeNode(1);
        TreeNode n2 = new TreeNode(2);
        TreeNode n3 = new TreeNode(3);
        TreeNode n5 = new TreeNode(5);
        TreeNode n6 = new TreeNode(6);
        TreeNode n7 = new TreeNode(7);
        TreeNode n8 = new TreeNode(8);
        TreeNode n9 = new TreeNode(9);
        TreeNode n10 = new TreeNode(10);
        n1.left = n2;
        n1.right = n3;
        n2.right = n5;
        n3.left = n6;
        n5.left = n7;
        n5.right = n8;
        n6.left = n9;
        n6.right = n10;
        return n1;
    }

    private static TreeNode initNode2() {
        TreeNode n1 = new TreeNode(1);
        TreeNode n2 = new TreeNode(2);
        TreeNode n3 = new TreeNode(3);
        TreeNode n4 = new TreeNode(4);
        n1.right = n2;
        n2.left = n3;
        n2.right = n4;
        return n1;
    }
```

Test cases

```
public static void main(String[] args) {

    System.out.println("Q1:");
    System.out.println(
        Arrays.toString(getTopk(new int[]{1, 0, 19, 7, 7, 8}, 3)));
    System.out.println(
        Arrays.toString(getTopk(new int[]{8, 7, 6, 9, 8, 3, 4, 1}, 2)));
    System.out.println(
        Arrays.toString(getTopk(new int[]{8, 7, 4, 9, 1, 8}, 3)));
    System.out.println();

    System.out.println("Q2:");
    int[][] nums1 = {{0, 0, 0, 1, 1, 1},{0, 0, 1, 1, 1, 1},{0, 0, 0, 0, 0, 1}};
    System.out.println(getFirstColumnContainsOne(nums1));
    int[][] nums2 = {{0, 0, 1, 1}, {1, 1, 1, 1}, {0, 0, 0, 1}};
    System.out.println(getFirstColumnContainsOne(nums2));
    int[][] nums3 = {{0, 0}, {0, 0}};
    System.out.println(getFirstColumnContainsOne(nums3));
    int[][] nums4 = {{1, 1}, {1, 1}};
    System.out.println(getFirstColumnContainsOne(nums4));
    System.out.println();

    System.out.println("Q3:");
    System.out.println(getPerfectCount(new int[]{2,7,1,3,5}));
    System.out.println(getPerfectCount(new int[]{1,2,3,4,5}));
    System.out.println(getPerfectCount(new int[]{1,2,3,4,5,6}));
    System.out.println(getPerfectCount(new int[]{2,3,1,6}));
    System.out.println(getPerfectCount(new int[]{3,1,4}));
    System.out.println(getPerfectCount(new int[]{1,2}));
    System.out.println(getPerfectCount(new int[]{2}));
    System.out.println();

    System.out.println("Q4:");
    System.out.println(maxA(3));
    System.out.println(maxA(7));
    System.out.println(maxA(11));
    System.out.println(maxA(15));
```

```
        System.out.println(maxA(19));
        System.out.println();

        System.out.println("Q5:");
        System.out.println(boundaryOfBinaryTree(initNode1()).toString());
        System.out.println(boundaryOfBinaryTree(initNode2()).toString());
        System.out.println(boundaryOfBinaryTree(new TreeNode(1)).toString());
        System.out.println(boundaryOfBinaryTree(null).toString());

    }
```

Test results

```
Q1:
[19, 8, 7]
[9, 8]
[9, 8, 7]

Q2:
3
1
-1
1

Q3:
3
4
7
2
1
0
0

Q4:
3
9
27
81
256

Q5:
[1, 2, 5, 7, 8, 9, 10, 6, 3]
[1, 2, 3, 4]
[1]
[]
```