

## KTH LARGEST ELEMENT

**215. Kth Largest Element in an Array** Solved

Medium Topics Companies

Given an integer array `nums` and an integer `k`, return the  $k^{\text{th}}$  largest element in the array.

Note that it is the  $k^{\text{th}}$  largest element in the sorted order, not the  $k^{\text{th}}$  distinct element.

Can you solve it without sorting?

**Example 1:**  
Input: `nums = [3,2,1,5,6,4]`, `k = 2`  
Output: `5`

**Example 2:**  
Input: `nums = [3,2,3,1,2,4,5,6]`, `k = 4`  
Output: `4`

**Constraints:**

- $1 \leq k \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

17.7K 275 286 Online

**Code**

```
Java Auto
8
9
10
11
12
13
}
return minHeap.peek();
}
```

Saved Ln 13, Col 1

**Testcase Test Result**

Accepted Runtime: 0 ms

Case 1 Case 2

Input

`nums =`  
`[3,2,1,5,6,4]`

`k =`  
`2`

## FIND PEAK ELEMENT

**162. Find Peak Element**

Medium Topics Companies

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.

**Code**

```
Java Auto
1
2
class Solution {
    public int findPeakElement(int[] nums) {
        // ...
    }
}
```

Saved Ln 15, Col 1

**Testcase Test Result**

Accepted Runtime: 0 ms

Case 1 Case 2

Input

`nums =`  
`[1,2,3,1]`

Output

`2`

Expected

`2`

Contribute a testcase

## FIRST BAD VERSION

The screenshot shows the LeetCode problem page for "278. First Bad Version". The problem description states that you are a product manager leading a team to develop a new product, but the latest version fails a quality check. Since each version is developed based on the previous one, all versions after a bad one are also bad. The goal is to find the first bad version out of  $n$  versions, minimizing the number of API calls to `isBadVersion(version)`.

**Example 1:**  
Input:  $n = 5$ ,  $\text{bad} = 4$   
Output: 4  
Explanation:   
call `isBadVersion(3)` -> false  
call `isBadVersion(5)` -> true  
call `isBadVersion(4)` -> true  
Then 4 is the first bad version.

**Example 2:**

The code editor on the right shows a Java solution using a binary search approach:

```
7         right = mid;  
8     } else {  
9         left = mid + 1;  
10    }  
11    }  
12    return left;  
13 }  
14 }  
15 }
```

The test result panel shows "Accepted" with a runtime of 1 ms. The input for Case 1 is  $n = 5$  and  $\text{bad} = 4$ .

## MERGE ELEMENT

The screenshot shows the LeetCode problem page for "56. Merge Intervals". The problem description asks to merge all overlapping intervals in an array and return the non-overlapping intervals.

**Example 1:**  
Input: `intervals = [[1,3],[2,6],[8,10],[15,18]]`  
Output: `[[1,6],[8,10],[15,18]]`  
Explanation: Since intervals `[1,3]` and `[2,6]` overlap, merge them into `[1,6]`.

**Example 2:**  
Input: `intervals = [[1,4],[4,5]]`  
Output: `[[1,5]]`  
Explanation: Intervals `[1,4]` and `[4,5]` are considered overlapping.

**Constraints:**

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].\text{length} == 2$

The code editor on the right shows a Java solution that sorts the intervals by their start value and then merges them:

```
9         } else {  
10            merged.get(merged.size() - 1)[1] = Math.max(merged.get(merged.size() - 1)  
11                [1], interval[1]);  
12        }  
13    }  
14    return merged.toArray(new int[merged.size()][]);  
15 }  
16 }
```

The test result panel shows "Accepted" with a runtime of 0 ms. The input for Case 1 is `intervals = [[1,3],[2,6],[8,10],[15,18]]` and the output is `[[1,6],[8,10],[15,18]]`.

## TOP K FREQUENT ELEMENT

Problem List

Description | Editorial | Solutions | Submissions

### 347. Top K Frequent Elements

Medium Topics Companies

Given an integer array `nums` and an integer `k`, return the `k` most frequent elements. You may return the answer in any order.

**Example 1:**

Input: `nums = [1,1,1,2,2,3]`, `k = 2`  
Output: `[1,2]`

**Example 2:**

Input: `nums = [1]`, `k = 1`  
Output: `[1]`

**Constraints:**

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `k` is in the range `[1, the number of unique elements in the array]`.

18K 228 337 Online

Code

Java

```
1 import java.util.*;
2
3 class Solution {
4     public int[] topKFrequent(int[] nums, int k) {
5         Map<Integer, Integer> countMap = new HashMap<>();
6         for (int num : nums) {
7             countMap.put(num, countMap.getOrDefault(num, 0) + 1);
8         }
9
10        PriorityQueue<Integer> minHeap = new PriorityQueue<>((Comparator.comparingInt
```

Testcase | Test Result

Case 1 Case 2

nums =

[1,1,1,2,2,3]

k =

2

Source