347. Top K Frequent Elements - Medium

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 <= 10^5
-10^4 <= nums[i] <= 10^4
k is in the range [1, the number of unique elements in the array].
It is guaranteed that the answer is unique.

Follow up: Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

```
impl Solution {
    pub fn top_k_frequent(nums: Vec<i32>, k: i32) -> Vec<i32> {

    }
}
```

238. Product of Array Except Self - Medium

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:
Input: nums = [1,2,3,4]
Output: [24,12,8,6]

Example 2:
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]

Constraints:

2 <= nums.length <= 10^5
-30 <= nums[i] <= 30
The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

Follow up: Can you solve the problem in O(1) extra space complexity? (The output array does not count as extra space for space complexity analysis.)

```
impl Solution {
    pub fn product_except_self(nums: Vec<i32>) -> Vec<i32> {

    }
}
```

230. Kth Smallest Element in a BST - Medium

Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:
Input: root = [3,1,4,null,2], k = 1
Output: 1

Example 2:
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3

Constraints:

The number of nodes in the tree is n.
1 <= k <= n <= 10^4
0 <= Node.val <= 10^4

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//   pub val: i32,
//   pub left: Option<Rc<RefCell<TreeNode>>>,
//   pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
//   #[inline]
//   pub fn new(val: i32) -> Self {
//     TreeNode {
//       val,
//       left: None,
//       right: None
//     }
//   }
// }
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
    pub fn kth_smallest(root: Option<Rc<RefCell<TreeNode>>>, k: i32) -> i32 {

    }
}
```

98. Validate Binary Search Tree -Medium

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:
Input: root = [2,1,3]
Output: true

Example 2:
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.

Constraints:
The number of nodes in the tree is in the range [1, 104].
-2^31 <= Node.val <= 2^31 – 1

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//   pub val: i32,
//   pub left: Option<Rc<RefCell<TreeNode>>>,
//   pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
//   #[inline]
//   pub fn new(val: i32) -> Self {
//     TreeNode {
//       val,
//       left: None,
//       right: None
//     }
//   }
// }
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
    pub fn is_valid_bst(root: Option<Rc<RefCell<TreeNode>>>) -> bool {

    }
}
```

367. Valid Perfect Square - Easy

Given a positive integer num, return true if num is a perfect square or false otherwise.

A perfect square is an integer that is the square of an integer. In other words, it is the product of some integer with itself.

You must not use any built-in library function, such as sqrt.

Example 1:
Input: num = 16
Output: true
Explanation: We return true because 4 * 4 = 16 and 4 is an integer.

Example 2:
Input: num = 14
Output: false
Explanation: We return false because 3.742 * 3.742 = 14 and 3.742 is not an integer.

Constraints:
1 <= num <= 2^31 – 1

```
impl Solution {
    pub fn is_perfect_square(num: i32) -> bool {

    }
}
```

112. Path Sum - Easy

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

Example 1:
Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
Output: true
Explanation: The root-to-leaf path with the target sum is shown.

Example 2:
Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.

Example 3:
Input: root = [], targetSum = 0
Output: false
Explanation: Since the tree is empty, there are no root-to-leaf paths.

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//   pub val: i32,
//   pub left: Option<Rc<RefCell<TreeNode>>>,
//   pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {

    pub fn has_path_sum(root: Option<Rc<RefCell<TreeNode>>>, target_sum: i32) -> bool {

    }
}
```

113. Path Sum II - Medium

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]
Explanation: There are two paths whose sum equals targetSum:
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22

Example 2:
Input: root = [1,2,3], targetSum = 5
Output: []

Example 3:
Input: root = [1,2], targetSum = 0
Output: []

Constraints:
- The number of nodes in the tree is in the range [0, 5000].
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//   pub val: i32,
//   pub left: Option<Rc<RefCell<TreeNode>>>,
//   pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
    pub fn path_sum(root: Option<Rc<RefCell<TreeNode>>>, target_sum: i32) -> Vec<Vec<i32>>
{

    }
}
```

437. Path Sum III - Medium

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:
Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3
Explanation: The paths that sum to 8 are shown.

Example 2:
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: 3

Constraints:

- The number of nodes in the tree is in the range [0, 1000].
- -10^9 <= Node.val <= 10^9
- -1000 <= targetSum <= 1000

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//   pub val: i32,
//   pub left: Option<Rc<RefCell<TreeNode>>>,
//   pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
//   #[inline]
//   pub fn new(val: i32) -> Self {
//     TreeNode {
//       val,
//       left: None,
//       right: None
//     }
//   }
// }
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
    pub fn path_sum(root: Option<Rc<RefCell<TreeNode>>>, target_sum: i32) -> i32 {

    }
}
```