1. Given an array of integers **arr[]** representing a permutation, implement the **next permutation** that rearranges the numbers into the lexicographically next greater permutation. If no such permutation exists, rearrange the numbers into the lowest possible order (i.e., sorted in ascending order).

Note - A permutation of an array of integers refers to a specific arrangement of its elements in a sequence or linear order.

Input: arr = [2, 4, 1, 7, 5, 0]

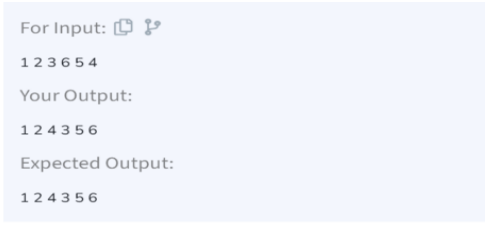
Output: [2, 4, 5, 0, 1, 7]

Explanation: The next permutation of the given array is {2, 4, 5, 0, 1, 7}.

CODE:

```
class Solution {
    void nextPermutation(int[] arr) {
        int pivot=-1;
        int n=arr.length;
        for(int i=n-2;i>=0;i--){
            if (arr[i]<arr[i+1]){
                pivot = i;
                break;
            }
        }
        if(pivot==-1){
            reverse(arr,0,n-1);
            return;
        }
        for(int i=n-1;i>pivot;i--){
            if (arr[i]>arr[pivot]){
                swap(arr,i,pivot);
                break;
            }
        }
        reverse(arr,pivot+1,n-1);
        for(int i=0;i<n;i++){
```

```
        }
    }
    public static void reverse(int[] arr, int start, int end) {
        while (start < end) {
            swap(arr, start++, end--);
        }
    }
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```



Time Complexity: O(n)

2. Given a matrix of size **N x M**. You have to find the **Kth** element which will obtain while traversing the matrix **spirally** starting from the top-left corner of the matrix.

Input:

N = 3, M = 3, K = 4

A[] = {{1, 2, 3},

{4, 5, 6},

{7, 8, 9}}

Output:

6

Explanation: Spiral traversal of matrix:

{1, 2, 3, 6, 9, 8, 7, 4, 5}. Fourth element

is 6.

CODE:

```
class Solution
{
        public int findK(int a[][], int n, int m, int k)
        {
            int top = 0, bottom = n - 1, left = 0, right = m - 1;
    int count = 0;
    while (top <= bottom && left <= right) {

        for (int i = left; i <= right; i++) {
            count++;
            if (count == k) {
                return a[top][i];
            }
        }
        top++;
        for (int i = top; i <= bottom; i++) {
            count++;
            if (count == k) {
                return a[i][right];
            }
        }
        right--;
        if (top <= bottom) {
            for (int i = right; i >= left; i--) {
                count++;
                if (count == k) {
                    return a[bottom][i];
                }
            }
            bottom--;
```

```
        }
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                count++;
                if (count == k) {
                    return a[i][left];
                }
            }
            left++;
        }
    }
    return -1;
  }
}
```

```
For Input:
3 3 4
1 2 3 4 5 6 7 8 9
Your Output:
6
Expected Output:
6
```

Time Complexity:O(n*m)

3. Given a string **s**, find the length of the longest substring with all distinct characters.

Input: s = "geeksforgeeks"

Output: 7

Explanation: "eksforg" is the longest substring with all distinct characters.

CODE:

```
class Solution {
    public int longestSubstrDistinctChars(String s) {
        int n = s.length();
        int res = 0;
        for (int i = 0; i < n; i++) {
            boolean[] visited = new boolean[256];
```

```
            for (int j = i; j < n; j++) {
                if (visited[s.charAt(j)]) {
                    break;
                }
                 else {
                    res = Math.max(res, j - i + 1);
                    visited[s.charAt(j)] = true;
                }
            }
        }
        return res;
    }
}
```

For Input:

geeksforgeeks

Your Output:

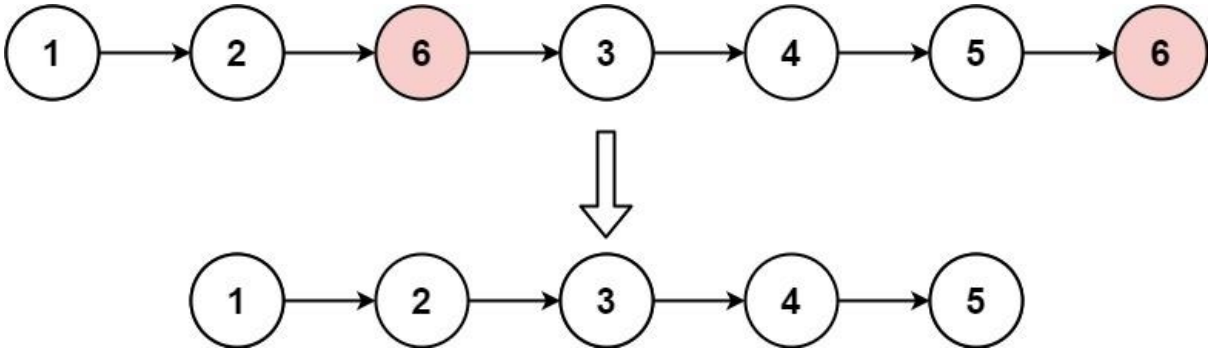
7

Expected Output:

7

Time Complexity: O(n^2)

4. Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.



Input: head = [1,2,6,3,4,5,6], val = 6

Output: [1,2,3,4,5]

CODE:

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if(head==null) return null;
        head.next=removeElements(head.next,val);
        return head.val==val?head.next:head;
    }
}
```

Output

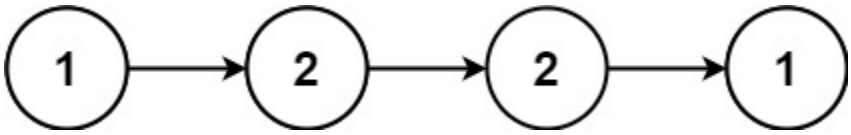
```
[1,2,3,4,5]
```

Expected

```
[1,2,3,4,5]
```

Time Complexity:O(n)

5. Given the head of a singly linked list, return true *if it is a palindrome* *or* false *otherwise*.



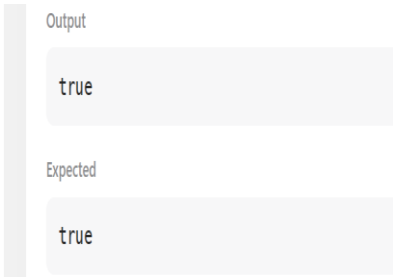
Input: head = [1,2,2,1]

Output: true

CODE:

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode slow = head, fast = head, prev, temp;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        prev = slow;
        slow = slow.next;
```

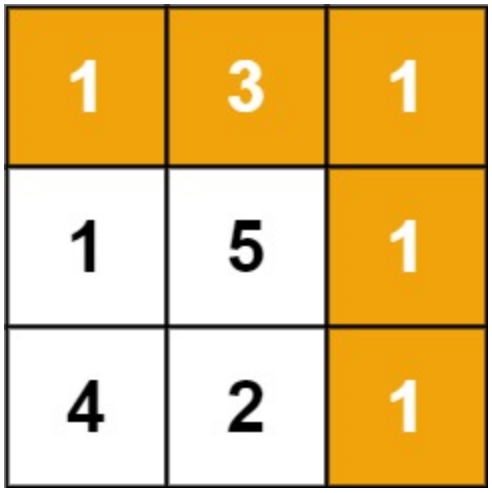
```
        prev.next = null;
        while (slow != null) {
            temp = slow.next;
            slow.next = prev;
            prev = slow;
            slow = temp;
        }
        fast = head;
        slow = prev;
        while (slow != null) {
            if (fast.val != slow.val) return false;
            fast = fast.next;
            slow = slow.next;
        }
        return true;
    }
}
```



Time Complexity: O(n)

6. Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.



Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

CODE:

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        grid[0][0] = grid[0][0];
        for (int i = 1; i < n; i++) {
            grid[0][i] += grid[0][i - 1];
        }
        for (int i = 1; i < m; i++) {
            grid[i][0] += grid[i - 1][0];
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
            }
        }
        return grid[m - 1][n - 1];
    }
}
```

Output

7

Expected

7

Time Complexity:O(m*n)

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

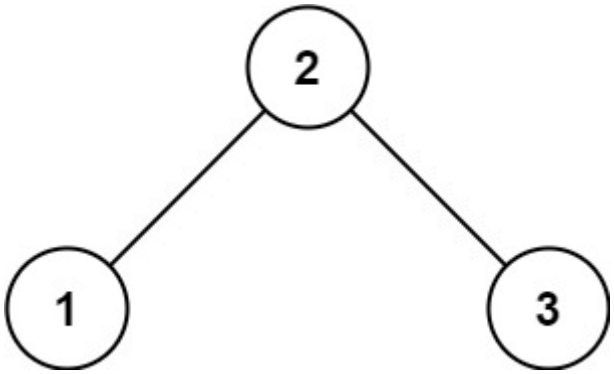
A **valid BST** is defined as follows:

- 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.



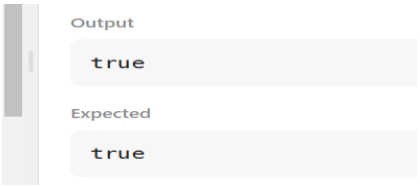
Input: root = [2,1,3]

Output: true

CODE:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    private boolean isValidBSTHelper(TreeNode node, long lower, long upper) {
        if (node == null) {
            return true;
```

```
        }
        if (node.val <= lower || node.val >= upper) {
            return false;
        }
        return isValidBSTHelper(node.left, lower, node.val) && isValidBSTHelper(node.right, node.val, upper);
    }
}
```



Time Complexity:O(n)

8. Given two words, beginWord and endWord, and a dictionary wordList, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* beginWord *to* endWord*, or* 0 *if no such sequence exists.*

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

CODE:

```
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) {
            return 0;
        }
        Queue<String> queue = new LinkedList<>();
        queue.offer(beginWord);
        Set<String> visited = new HashSet<>();
        visited.add(beginWord);
        int level = 1;
        while (!queue.isEmpty()) {
```

```
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String currentWord = queue.poll();
                for (int j = 0; j < currentWord.length(); j++) {
                    char[] wordArray = currentWord.toCharArray();
                    for (char c = 'a'; c <= 'z'; c++) {
                        wordArray[j] = c;
                        String nextWord = new String(wordArray);
                        if (nextWord.equals(endWord)) {
                            return level + 1;
                        }
                        if (wordSet.contains(nextWord) && !visited.contains(nextWord)) {
                            visited.add(nextWord);
                            queue.offer(nextWord);
                        }
                    }
                }
            }
            level++;
        }
        return 0;
    }
}
```

Output

```
5
```

Expected

```
5
```

Time Complexity:O(n*l)

9. Given two words, beginWord and endWord, and a dictionary wordList, return *all the **shortest transformation sequences** from* beginWord *to* endWord*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* [beginWord, s_1, s_2, ..., s_k].

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

Explanation: There are 2 shortest transformation sequences:

"hit" -> "hot" -> "dot" -> "dog" -> "cog"

"hit" -> "hot" -> "lot" -> "log" -> "cog"

CODE:

```
class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) {
            return new ArrayList<>();
        }
        Queue<String> queue = new LinkedList<>();
        queue.offer(beginWord);
        Map<String, List<String>> parents = new HashMap<>();
        parents.put(beginWord, new ArrayList<>());
        Set<String> visited = new HashSet<>();
        visited.add(beginWord);
        boolean found = false;
        while (!queue.isEmpty() && !found) {
            int size = queue.size();
            Set<String> levelVisited = new HashSet<>();
            for (int i = 0; i < size; i++) {
                String word = queue.poll();
                for (int j = 0; j < word.length(); j++) {
                    char[] wordArr = word.toCharArray();
                    for (char c = 'a'; c <= 'z'; c++) {
                        wordArr[j] = c;
                        String nextWord = new String(wordArr);
```

```
                if (nextWord.equals(endWord)) {
                    found = true;
                }
                if (wordSet.contains(nextWord) && !visited.contains(nextWord)) {
                    levelVisited.add(nextWord);
                    queue.offer(nextWord);
                    parents.putIfAbsent(nextWord, new ArrayList<>());
                    parents.get(nextWord).add(word);
                }
            }
        }
    }
    visited.addAll(levelVisited);
  }
  if (!found) {
    return new ArrayList<>();
  }
  List<List<String>> result = new ArrayList<>();
  List<String> path = new ArrayList<>();
  path.add(endWord);
  backtrack(result, path, parents, beginWord, endWord);
  return result;
 }
 private void backtrack(List<List<String>> result, List<String> path, Map<String, List<String>>
parents, String beginWord, String currentWord) {
  if (currentWord.equals(beginWord)) {
    List<String> validPath = new ArrayList<>(path);
    Collections.reverse(validPath);
    result.add(validPath);
    return;
  }
```

```
        for (String parent : parents.get(currentWord)) {
            path.add(parent);
            backtrack(result, path, parents, beginWord, parent);
            path.remove(path.size() - 1);
        }
    }
}
```

Output

```
[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog
"]]
```

Expected

```
[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog
"]]
```

Time Complexity: O(N * L + P * L)

10. There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Input: numCourses = 2, prerequisites = [[1,0]]

Output: true

Explanation: There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

CODE:

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] prerequisite : prerequisites) {
            graph.get(prerequisite[1]).add(prerequisite[0]);
        }
```

```
        int[] visited = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            if (visited[i] == 0) {
                if (hasCycle(graph, visited, i)) {
                    return false;
                }
            }
        }
        return true;
    }
    private boolean hasCycle(List<List<Integer>> graph, int[] visited, int course) {
        if (visited[course] == 1) {
            return true;
        }
        if (visited[course] == 2) {
            return false;
        }
        visited[course] = 1;
        for (int neighbor : graph.get(course)) {
            if (hasCycle(graph, visited, neighbor)) {
                return true;
            }
        }
        visited[course] = 2;
        return false;
    }
}
```

Output

```
true
```

Expected

```
true
```

Time Complexity: O(V + E)