1. GIVEN AN INTEGER ARRAY NUMS WHERE THE ELEMENTS ARE SORTED IN ASCENDING ORDER, CONVERT IT TO A HEIGHT-BALANCED BINARY SEARCH TREE. EXAMPLE 1: INPUT: NUMS = [-10,-3,0,5,9] OUTPUT: [0,-3,9,-10,NULL,5] EXPLANATION: [0,-10,5,NULL,-3,NULL,9] IS ALSO ACCEPTED

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def sorted_array_to_bst(nums):
    if not nums:
        return None

    # Find the middle element
    mid = len(nums) // 2

    # The middle element becomes the root
    root = TreeNode(nums[mid])

    # Recursively build the left and right subtrees
    root.left = sorted_array_to_bst(nums[:mid])
    root.right = sorted_array_to_bst(nums[mid+1:])

    return root

# Example usage:
nums = [-10, -3, 0, 5, 9]
tree_root = sorted_array_to_bst(nums)
```

2. GIVEN AN INTEGER ARRAY NUMS, REORDER IT SUCH THAT NUMS[0] < NUMS[1] > NUMS[2] < NUMS[3].... YOU MAY ASSUME THE INPUT ARRAY ALWAYS HAS A VALID ANSWER. EXAMPLE 1: INPUT: NUMS = [1,5,1,1,6,4] OUTPUT: [1,6,1,5,1,4] EXPLANATION: [1,4,1,5,1,6] IS ALSO ACCEPTED. EXAMPLE 2: INPUT: NUMS = [1,3,2,2,3,1] OUTPUT: [2,3,1,3,1,2]

```
def wiggle_sort(nums):
    nums.sort()
    half = len(nums[::2])
    nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]

# Example usage:
nums1 = [1, 5, 1, 1, 6, 4]
wiggle_sort(nums1)
print(nums1)  # Output: [1, 6, 1, 5, 1, 4]

nums2 = [1, 3, 2, 2, 3, 1]
wiggle_sort(nums2)
print(nums2)  # Output: [2, 3, 1, 3, 1, 2]
```

3.YOU ARE GIVEN AN ARRAY OF K LINKED-LISTS LISTS, EACH LINKED-LIST IS SORTED IN ASCENDING ORDER.MERGE ALL THE LINKED-LISTS INTO ONE SORTED LINKED-LIST AND RETURN IT. INPUT: LISTS = [[1,4,5],[1,3,4],[2,6]] OUTPUT: [1,1,2,3,4,4,5,6] EXPLANATION: THE LINKED-LISTS ARE: [1->4->5, 1->3->4, 2->6] MERGING THEM INTO ONE SORTED LIST: 1->1->2->3->4->4->5->6

```
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def merge_k_lists(lists):
    # Create a min-heap
    min_heap = []

    # Define a comparator for the heap to compare ListNode objects
    ListNode.__lt__ = lambda self, other: self.val < other.val
    # Initialize the heap with the head of each list
    for l in lists:
        if l:
            heapq.heappush(min_heap, l)

    # Dummy node to start the merged list
    dummy = ListNode(0)
    current = dummy

    # Pop the smallest element from the heap and add it to the merged list
    while min_heap:
        node = heapq.heappop(min_heap)
        current.next = node
        current = current.next
        if node.next:
            heapq.heappush(min_heap, node.next)

    return dummy.next
```

4. GIVEN TWO SORTED ARRAYS NUMS1 AND NUMS2 OF SIZE M AND N RESPECTIVELY, RETURN THE MEDIAN OF THE TWO SORTED ARRAYS. THE OVERALL RUN TIME COMPLEXITY SHOULD BE O(LOG (M+N)). EXAMPLE 1: INPUT: NUMS1 = [1,3], NUMS2 = [2] OUTPUT: 2.00000 EXPLANATION: MERGED ARRAY = [1,2,3] AND MEDIAN IS 2.

```
def find_median_sorted_arrays(nums1, nums2):
    # Ensure nums1 is the smaller array
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    low, high = 0, x

    while low <= high:
        partitionX = (low + high) // 2
        partitionY = (x + y + 1) // 2 - partitionX

        maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]

        minX = float('inf') if partitionX == x else nums1[partitionX]
        minY = float('inf') if partitionY == y else nums2[partitionY]

        if maxX <= minY and maxY <= minX:
            if (x + y) % 2 == 0:
                return (max(maxX, maxY) + min(minX, minY)) / 2
            else:
                return max(maxX, maxY)

        elif maxX > minY:
            high = partitionX - 1
        else:
            low = partitionX + 1

# Example usage:
nums1 = [1,3]
nums2 = [2]
median = find_median_sorted_arrays(nums1, nums2)
print(median)  # Output: 2.0
```

5. NUMS[B], NUMS[C], NUMS[D]] SUCH THAT: 0 <= A, B, C, D < N A, B, C, AND D ARE DISTINCT. NUMS[A] + NUMS[B] + NUMS[C] + NUMS[D] == TARGET YOU MAY RETURN THE ANSWER IN ANY ORDER. EXAMPLE 1: INPUT: NUMS = [1,0,-1,0,-2,2], TARGET = 0 OUTPUT: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]] EXAMPLE 2: INPUT: NUMS = [2,2,2,2,2], TARGET = 8 OUTPUT: [[2,2,2,2]]

```
def four_sum(nums, target):
    def k_sum(nums, target, k):
        res = []
        if not nums:
            return res
        average_value = target // k
        if average_value < nums[0] or nums[-1] < average_value:
            return res
        if k == 2:
            return two_sum(nums, target)
        for i in range(len(nums)):
            if i == 0 or nums[i - 1] != nums[i]:
                for subset in k_sum(nums[i + 1:], target - nums[i], k - 1):
                    res.append([nums[i]] + subset)
        return res

    def two_sum(nums, target):
        res = []
        s = set()
        for i in range(len(nums)):
            if len(res) == 0 or res[-1][1] != nums[i]:
                if target - nums[i] in s:
                    res.append([target - nums[i], nums[i]])
            s.add(nums[i])
        return res

    nums.sort()
    return k_sum(nums, target, 4)

# Example usage:
nums1 = [1, 0, -1, 0, -2, 2]
target1 = 0
print(four_sum(nums1, target1))  # Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

```
nums2 = [2, 2, 2, 2, 2]
target2 = 8
print(four_sum(nums2, target2))  # Output: [[2,2,2,2]]
```

6. GIVEN AN ARRAY NUMS OF SIZE N, RETURN THE MAJORITY ELEMENT. THE MAJORITY ELEMENT IS THE ELEMENT THAT APPEARS MORE THAN ⌊N / 2⌋ TIMES. YOU MAY ASSUME THAT THE MAJORITY ELEMENT ALWAYS EXISTS IN THE ARRAY. EXAMPLE 1: INPUT: NUMS = [3,2,3] OUTPUT: 3

```
def majority_element(nums):
    count = 0
    candidate = None

    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)

    return candidate

# Example usage:
nums = [3, 2, 3]
print(majority_element(nums))  # Output: 3
```

7. GIVEN THE HEAD OF A LINKED LIST, RETURN THE LIST AFTER SORTING IT IN ASCENDING ORDER. INPUT: HEAD = [4,2,1,3] OUTPUT: [1,2,3,4]

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def sort_list(head):
    if not head or not head.next:
        return head
    # Split the list into two halves
    slow, fast = head, head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    mid = slow.next
    slow.next = None

    # Sort each half
    left = sort_list(head)
    right = sort_list(mid)

    # Merge the sorted halves
    return merge(left, right)

def merge(l1, l2):
    dummy = ListNode(0)
    tail = dummy

    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next

    tail.next = l1 or l2
    return dummy.next
```

8. GIVEN AN ARRAY OF STRINGS STRS, GROUP THE ANAGRAMS TOGETHER. YOU CAN RETURN THE ANSWER IN ANY ORDER. AN ANAGRAM IS A WORD OR PHRASE FORMED BY REARRANGING THE LETTERS OF A DIFFERENT WORD OR PHRASE, TYPICALLY USING ALL THE ORIGINAL LETTERS EXACTLY ONCE. EXAMPLE 1: INPUT: STRS = ["EAT","TEA","TAN","ATE","NAT","BAT"] OUTPUT: [["BAT"],["NAT","TAN"],["ATE","EAT","TEA"]] EXAMPLE 2: INPUT: STRS = [""] OUTPUT: [[""]]

```
from collections import defaultdict


def group_anagrams(strs):

    anagrams = defaultdict(list)


    for s in strs:

        # Sort the string and use it as a key

        sorted_str = ''.join(sorted(s))

        anagrams[sorted_str].append(s)


    # Return the grouped anagrams

    return list(anagrams.values())


# Example usage:

strs1 = ["eat", "tea", "tan", "ate", "nat", "bat"]

print(group_anagrams(strs1))  # Output: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]


strs2 = [""]

print(group_anagrams(strs2))  # Output: [[""]]
```

9. YOU ARE GIVEN TWO 0-INDEXED ARRAYS NUMS1 AND NUMS2 OF LENGTH N, BOTH OF WHICH ARE PERMUTATIONS OF [0, 1, ..., N - 1]. A GOOD TRIPLET IS A SET OF 3 DISTINCT VALUES WHICH ARE PRESENT IN INCREASING ORDER BY POSITION BOTH IN NUMS1 AND NUMS2. IN OTHER WORDS, IF WE CONSIDER POS1V AS THE INDEX OF THE VALUE V IN NUMS1 AND POS2V AS THE INDEX OF THE VALUE V IN NUMS2, THEN A GOOD TRIPLET WILL BE A SET (X, Y, Z) WHERE 0 <= X, Y, Z <= N - 1, SUCH THAT POS1X < POS1Y < POS1Z AND POS2X < POS2Y < POS2Z. RETURN THE TOTAL NUMBER OF GOOD TRIPLETS. EXAMPLE 1: INPUT: NUMS1 = [2,0,1,3], NUMS2 = [0,1,2,3] OUTPUT: 1 EXPLANATION: THERE ARE 4 TRIPLETS (X,Y,Z) SUCH THAT POS1X < POS1Y < POS1Z. THEY ARE (2,0,1),

(2,0,3), (2,1,3), AND (0,1,3). OUT OF THOSE TRIPLETS, ONLY THE TRIPLET (0,1,3) SATISFIES POS2X < POS2Y < POS2Z. HENCE, THERE IS ONLY 1 GOOD TRIPLET.

```
def count_good_triplets(nums1, nums2):
    # Store the positions of each value in nums1
    pos1 = {num: i for i, num in enumerate(nums1)}
    count = 0

    # Iterate through all possible triplets in nums2
    for i in range(len(nums2)):
        for j in range(i + 1, len(nums2)):
            for k in range(j + 1, len(nums2)):
                # Check if they form a good triplet
                if pos1[nums2[i]] < pos1[nums2[j]] < pos1[nums2[k]]:
                    count += 1

    return count

# Example usage:
nums1 = [2, 0, 1, 3]
nums2 = [0, 1, 2, 3]
print(count_good_triplets(nums1, nums2))  # Output: 1
```

10. GIVEN AN INTEGER ARRAY NUMS AND AN INTEGER K, RETURN THE KTH LARGEST ELEMENT IN THE ARRAY. NOTE THAT IT IS THE KTH LARGEST ELEMENT IN THE SORTED ORDER, NOT THE KTH 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 = [3,2,3,1,2,4,5,5,6], K = 4 OUTPUT: 4 CONSTRAINTS: 1 <= K <= NUMS.LENGTH <= 105 -104 <= NUMS[I] <= 104

```
import heapq

def find_kth_largest(nums, k):
    # Create a min-heap with the first k elements
    min_heap = nums[:k]
    heapq.heapify(min_heap)
    # Iterate through the remaining elements
    for num in nums[k:]:
        if num > min_heap[0]:
            # Replace the smallest element in the heap if the current num is larger
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)

    # The root of the min-heap is the kth largest element
    return min_heap[0]

# Example usage:
nums1 = [3, 2, 1, 5, 6, 4]
k1 = 2
print(find_kth_largest(nums1, k1))  # Output: 5

nums2 = [3, 2, 3, 1, 2, 4, 5, 5, 6]
k2 = 4
print(find_kth_largest(nums2, k2))  # Output: 4
```