31. Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.

Input: mat = [[0,0,0],[0,1,0],[0,0,0]] ; Output: [[0,0,0],[0,1,0],[0,0,0]]

Input: mat = [[0,0,0],[0,1,0],[1,1,1]] ;Output: [[0,0,0],[0,1,0],[1,2,1]]

Program:

```
from collections import deque

def updateMatrix(mat):

    rows, cols = len(mat), len(mat[0])

    dist = [[float('inf')] * cols for _ in range(rows)]

    queue = deque()

    for r in range(rows):

        for c in range(cols):

            if mat[r][c] == 0:

                dist[r][c] = 0

                queue.append((r, c))

    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    while queue:

        r, c = queue.popleft()

        for dr, dc in directions:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols:

                if dist[nr][nc] > dist[r][c] + 1:

                    dist[nr][nc] = dist[r][c] + 1

                    queue.append((nr, nc))

    return dist

mat1 = [[0,0,0],[0,1,0],[0,0,0]]
```

```
mat2 = [[0,0,0],[0,1,0],[1,1,1]]

print(updateMatrix(mat1))

# Output: [[0,0,0],[0,1,0],[0,0,0]]

print(updateMatrix(mat2))

# Output: [[0,0,0],[0,1,0],[1,2,1]]
```

31. 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

Program:

```
import heapq

class ListNode:

    def _init_(self, val=0, next=None):

        self.val = val

        self.next = next

def merge_k_sorted_lists(lists):

    if not lists:

        return None

    dummy = ListNode()

    curr = dummy

    pq = []  # Priority queue to store (value, list index)

    # Initialize the priority queue with the first element from each list

    for i, lst in enumerate(lists):

        if lst:

            heapq.heappush(pq, (lst.val, i))
    while pq:
        val, idx = heapq.heappop(pq)
        curr.next = ListNode(val)
        curr = curr.next

        if lists[idx].next:
            heapq.heappush(pq, (lists[idx].next.val, idx))
            lists[idx] = lists[idx].next

    return dummy.next

# Example usage:
# Create linked lists from the input lists
lists = [[1, 4, 5], [1, 3, 4], [2, 6]]
linked_lists = [ListNode(val) for val in lists]

# Merge the linked lists with error handling
try:
    merged_list = merge_k_sorted_lists(linked_lists)

    # Convert the merged list to a Python list for display
    result = []
    while merged_list:
        result.append(merged_list.val)
        merged_list = merged_list.next

    print(result)  # Output: [1, 1, 2, 3, 4, 4, 5, 6]

except Exception as e:
    print(f"An error occurred: {e}")
```

32. Given two integer arrays arr1 and arr2, return the minimum number of operations (possibly zero) needed to make arr1 strictly increasing. In one operation, you can choose two indices 0 <= i < arr1.length and 0 <= j < arr2.length and do the assignment arr1[i] = arr2[j]. If there is no way to make arr1 strictly increasing, return -1.

Example 1: Input: arr1 = [1,5,3,6,7], arr2 = [1,3,2,4] Output: 1

Explanation: Replace 5 with 2, then arr1 = [1, 2, 3, 6, 7]

Program:

```
def min_operations(arr1, arr2):
    n = len(arr1)
    arr2.sort()
    dp = {-1: 0}
    for i in range(n):
        new_dp = {}
        for key in dp:
            if arr1[i] > key:
                new_dp[arr1[i]] = min(new_dp.get(arr1[i], float('inf')), dp[key])
            while arr2 and arr2[0] <= key:
                arr2.pop(0)
            if arr2:
                new_dp[arr2[0]] = min(new_dp.get(arr2[0], float('inf')), dp[key] + 1)
        dp = new_dp
    if dp:
        return min(dp.values())
    return -1
arr1 = [1, 5, 3, 6, 7]
arr2 = [1, 3, 2, 4]
print(min_operations(arr1, arr2))
# Output: 1
```

32. 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.

Program:

```
def find_median_sorted_arrays(nums1, nums2):
    merged = sorted(nums1 + nums2)
    total_len = len(merged)
    if total_len % 2 == 1:
        return float(merged[total_len // 2])
    else:
        mid1 = merged[total_len // 2 - 1]
        mid2 = merged[total_len // 2]
        return (mid1 + mid2) / 2.0
# Example usage:
nums1 = [1, 3]
nums2 = [2]
print(find_median_sorted_arrays(nums1, nums2))
```

33. Given two strings a and b, return the minimum number of times you should repeat string a so that string b is a substring of it. If it is impossible for b to be a substring of a after repeating it, return -1. Notice: string "abc" repeated 0 times is "", repeated 1 time is "abc" and repeated 2 times is "abcabc". Example 1: Input: a = "abcd", b = "cdabcdab" ; Output: 3

Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of it

Program:

```
def min_repeats_v2(a, b):
   if b in a:
      return 1
   for i in range(1, len(b) + 1):
      if b == a[:i] * (len(b) // i) + a[:len(b) % i]:
         return len(b) // i + (len(b) % i != 0)
   return -1
a = "abcd"
b = "cdabcdab"
result = min_repeats_v2(a, b)
print(result)
# Output: 3
```

33. Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], 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]]

Program:

```
def four_sum(nums, target):
   nums.sort()  # Sort the array
```

```
result = []
n = len(nums)

for a in range(n - 3):
    if a > 0 and nums[a] == nums[a - 1]:
        continue  # Skip duplicates

    for b in range(a + 1, n - 2):
        if b > a + 1 and nums[b] == nums[b - 1]:
            continue  # Skip duplicates

        left, right = b + 1, n - 1

        while left < right:
            total = nums[a] + nums[b] + nums[left] + nums[right]

            if total == target:
                result.append([nums[a], nums[b], nums[left], nums[right]])
                while left < right and nums[left] == nums[left + 1]:
                    left += 1  # Skip duplicates
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1  # Skip duplicates
                left += 1
                right -= 1
            elif total < target:
                left += 1
            else:
                right -= 1
    return result
# 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]]
```

34. Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

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

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Program:

```
def missing_number(nums):
    n = len(nums)
    total_sum = n * (n + 1) // 2
    array_sum = sum(nums)
    return total_sum - array_sum
nums = [3, 0, 1]
print(missing_number(nums))
# Output: 2
```

34. 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

Program:

```
def majority_element(nums):
    candidate = None
    count = 0
    for num in nums:
        if count == 0:
            candidate = num
            count = 1
        elif num == candidate:
            count += 1
        else:
            count -= 1
    return candidate
# Example usage:
nums = [3, 2, 3]
print(majority_element(nums))  # Output: 3
```

35. You are given an n x n integer matrix grid.Generate an integer matrix maxLocal of size (n - 2) x (n - 2) such that: maxLocal[i][j] is equal to the largest value of the 3 x 3 matrix in grid centered around row i + 1 and column j + 1. In other words, we want to find the largest value in every contiguous 3 x 3 matrix in grid. Return the generated matrix.

Input: grid = [[9,9,8,1],[5,6,2,6],[8,2,6,4],[6,2,2,2]] Output: [[9,9],[8,6]]

Explanation: The diagram above shows the original matrix and the generated matrix. Notice that each value in the generated matrix corresponds to the largest value of a contiguous 3 x 3 matrix in grid.

Program:

```
def generate_max_local(grid):
    n = len(grid)
    max_local = [[max(grid[i-1][j-1], grid[i-1][j], grid[i-1][j+1],
                      grid[i][j-1], grid[i][j], grid[i][j+1],
                      grid[i+1][j-1], grid[i+1][j], grid[i+1][j+1])
                  for j in range(1, n-1)]
                 for i in range(1, n-1)]
    return max_local
grid = [[9, 9, 8, 1], [5, 6, 2, 6], [8, 2, 6, 4], [6, 2, 2, 2]]
result = generate_max_local(grid)
print(result)
```

35. 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]

Program:

```
class ListNode:
    def _init_(self, val=0, next=None):
        self.val = val
        self.next = next
def merge_sorted_lists(left, right):
    dummy = ListNode()
    curr = dummy
    while left and right:
        if left.val < right.val:
            curr.next = left
            left = left.next
        else:
            curr.next = right
            right = right.next
        curr = curr.next

    curr.next = left or right
    return dummy.next

def sort_linked_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

    left, right = head, slow.next
    slow.next = None

    # Recursively sort both halves
    left_sorted = sort_linked_list(left)
    right_sorted = sort_linked_list(right)
    # Merge the sorted halves
    return merge_sorted_lists(left_sorted, right_sorted)
# Example usage:
# Create a linked list from the input
head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))
sorted_head = sort_linked_list(head)
# Convert the sorted linked list to a Python list for display
result = []
while sorted_head:
    result.append(sorted_head.val)
    sorted_head = sorted_head.next
print(result)  # Output: [1, 2, 3, 4]
```