1. Distance of Nearest 0 in a Binary Matrix

Problem: Given an m×nm \times nm×n binary matrix `mat`, return the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.

Solution: To solve this, we can use a multi-source Breadth-First Search (BFS). The BFS will start from all 0's in the matrix and propagate the distance to 1's. Here's the implementation in Python:

```
python
Copy code
from collections import deque

def updateMatrix(mat):
    m, n = len(mat), len(mat[0])
    dist = [[float('inf')] * n for _ in range(m)]
    queue = deque()

    # Initialize the queue with all 0s and their distances as 0
    for i in range(m):
        for j in range(n):
            if mat[i][j] == 0:
                dist[i][j] = 0
                queue.append((i, j))

    # BFS from all 0s
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    while queue:
        x, y = queue.popleft()
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and dist[nx][ny] > dist[x][y] +
1:
                dist[nx][ny] = dist[x][y] + 1
                queue.append((nx, ny))

    return dist

# Test Cases
print(updateMatrix([[0,0,0],[0,1,0],[0,0,0]]))  # [[0,0,0],[0,1,0],[0,0,0]]
print(updateMatrix([[0,0,0],[0,1,0],[1,1,1]]))  # [[0,0,0],[0,1,0],[1,2,1]]
```

2. Merge k Sorted Linked Lists

Problem: Merge `k` sorted linked-lists into one sorted linked-list and return it.

Solution: To solve this, we can use a Min-Heap to efficiently merge the linked lists. Here is the implementation:

```
python
Copy code
import heapq

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

```
def mergeKLists(lists):
    heap = []

    # Initialize the heap with the head of each list
    for l in lists:
        if l:
            heapq.heappush(heap, (l.val, l))

    dummy = ListNode()
    curr = dummy

    while heap:
        val, node = heapq.heappop(heap)
        curr.next = ListNode(val)
        curr = curr.next
        if node.next:
            heapq.heappush(heap, (node.next.val, node.next))

    return dummy.next

# Helper function to convert list to ListNode
def build_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for val in arr[1:]:
        current.next = ListNode(val)
        current = current.next
    return head

# Test Case
list1 = build_list([1,4,5])
list2 = build_list([1,3,4])
list3 = build_list([2,6])
result = mergeKLists([list1, list2, list3])

# Helper function to print ListNode
def print_list(node):
    while node:
        print(node.val, end="->")
        node = node.next
    print("None")

print_list(result)  # 1->1->2->3->4->4->5->6->None
```

3. Minimum Number of Operations to Make Array Strictly Increasing

Problem: Given two integer arrays `arr1` and `arr2`, return the minimum number of operations needed to make `arr1` strictly increasing by replacing elements in `arr1` with elements from `arr2`.

Solution: This problem can be solved using Dynamic Programming with Binary Search. Here is the implementation:

```
python
Copy code
from bisect import bisect_right
```

```
def makeArrayIncreasing(arr1, arr2):
    arr2.sort()
    dp = {-1: 0}

    for num in arr1:
        new_dp = {}
        for key in dp:
            if num > key:
                new_dp[num] = min(new_dp.get(num, float('inf')), dp[key])
            loc = bisect_right(arr2, key)
            if loc < len(arr2):
                new_dp[arr2[loc]] = min(new_dp.get(arr2[loc],
float('inf')), dp[key] + 1)
        dp = new_dp
        if not dp:
            return -1
    return min(dp.values())

# Test Case
print(makeArrayIncreasing([1,5,3,6,7], [1,3,2,4]))  # 1
```

4. Median of Two Sorted Arrays

Problem: Given two sorted arrays `nums1` and `nums2`, return the median of the two sorted arrays with an overall run time complexity of O(log⁡(m+n))O(\log (m+n))O(log(m+n)).

Solution: We can solve this problem using a binary search approach. Here is the implementation:

```
python
Copy code
def findMedianSortedArrays(nums1, nums2):
    A, B = nums1, nums2
    if len(A) > len(B):
        A, B = B, A
    m, n = len(A), len(B)

    imin, imax, half_len = 0, m, (m + n + 1) // 2
    while imin <= imax:
        i = (imin + imax) // 2
        j = half_len - i
        if i < m and B[j-1] > A[i]:
            imin = i + 1
        elif i > 0 and A[i-1] > B[j]:
            imax = i - 1
        else:
            max_of_left = 0
            if i == 0: max_of_left = B[j-1]
            elif j == 0: max_of_left = A[i-1]
            else: max_of_left = max(A[i-1], B[j-1])
            if (m + n) % 2 == 1:
                return max_of_left
            min_of_right = 0
            if i == m: min_of_right = B[j]
            elif j == n: min_of_right = A[i]
            else: min_of_right = min(A[i], B[j])
            return (max_of_left + min_of_right) / 2.0
```

```
# Test Case
print(findMedianSortedArrays([1,3], [2]))  # 2.00000
print(findMedianSortedArrays([1,2], [3,4]))  # 2.50000
```

5. Minimum Number of Repeats for Substring

Problem: 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.

Solution: To solve this, we can repeatedly append `a` to itself and check if `b` becomes a substring of it. Here is the implementation:

```
python
Copy code
def repeatedStringMatch(a, b):
    count = 1
    repeated_a = a
    while len(repeated_a) < len(b):
        repeated_a += a
        count += 1
    if b in repeated_a:
        return count
    if b in (repeated_a + a):
        return count + 1
    return -1

# Test Case
print(repeatedStringMatch("abcd", "cdabcdab"))  # 3
```

6. Four Sum

Problem: Given an array `nums` of n integers, return an array of all unique quadruplets `[nums[a], nums[b], nums[c], nums[d]]` such that `nums[a] + nums[b] + nums[c] + nums[d] == target`.

Solution: To solve this, we can use a combination of two-pointer technique with sorting. Here is the implementation:

```
python
Copy code
def fourSum(nums, target):
    nums.sort()
    result = []
    n = len(nums)

    for i in range(n - 3):
        if i > 0 and nums[i] == nums[i-1]:
            continue
        for j in range(i + 1, n - 2):
            if j > i + 1 and nums[j] == nums[j-1]:
                continue
            left, right = j + 1, n - 1
            while left < right:
                sum = nums[i] + nums[j] + nums[left] + nums[right]
                if sum == target:
```

```
                    result.append([nums[i], nums[j], nums[left],
nums[right]])
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1
                    left += 1
                    right -= 1
                elif sum < target:
                    left += 1
                else:
                    right -= 1

    return result

# Test Cases
print(fourSum([1,0,-1,0,-2,2], 0))  # [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
print(fourSum([2,2,2,2,2], 8))  # [[2,2,2,2]]
```

7. Missing Number

Problem: 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.

Solution: To solve this, we can use the formula for the sum of the first `n` natural numbers. Here is the implementation:

```
python
Copy code
def missingNumber(nums):
    n = len(nums)
    total_sum = n * (n + 1) // 2
    array_sum = sum(nums)
    return total_sum - array_sum

# Test Case
print(missingNumber([3,0,1]))  # 2
```

8. Majority Element

Problem: Given an array `nums` of size `n`, return the majority element.

Solution: This problem can be solved using the Boyer-Moore Voting Algorithm. Here is the implementation:

```
python
Copy code
def majorityElement(nums):
    count = 0
    candidate = None

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

    return candidate
```

```
# Test Case
print(majorityElement([3,2,3]))  # 3
print(majorityElement([2,2,1,1,1,2,2]))  # 2
```

9. Maximum Local Value in Grid

Problem: 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`.

Solution: To solve this, we can iterate through the grid and compute the maximum value for each `3 x 3` subgrid. Here is the implementation:

```
python
Copy code
def largestLocal(grid):
    n = len(grid)
    maxLocal = [[0] * (n - 2) for _ in range(n - 2)]

    for i in range(n - 2):
        for j in range(n - 2):
            maxLocal[i][j] = max([grid[i+k][j+l] for k in range(3) for l in
range(3)])

    return maxLocal

# Test Case
print(largestLocal([[9,9,8,1],[5,6,2,6],[8,2,6,4],[6,2,2,2]]))  #
[[9,9],[8,6]]
```

10. Sort Linked List

Problem: Given the head of a linked list, return the list after sorting it in ascending order.

Solution: To solve this, we can use the merge sort algorithm for linked lists. Here is the implementation:

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

def sortList(head):
    if not head or not head.next:
        return head

    mid = getMid(head)
    left = sortList(head)
    right = sortList(mid)

    return merge(left, right)

def getMid(head):
    slow, fast = head, head.next
```

```
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    mid = slow.next
    slow.next = None
    return mid

def merge(list1, list2):
    dummy = ListNode()
    tail = dummy

    while list1 and list2:
        if list1.val < list2.val:
            tail.next = list1
            list1 = list1.next
        else:
            tail.next = list2
            list2 = list2.next
        tail = tail.next

    tail.next = list1 if list1 else list2
    return dummy.next

# Helper function to convert list to ListNode
def build_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for val in arr[1:]:
        current.next = ListNode(val)
        current = current.next
    return head

# Test Case
head = build_list([4,2,1,3])
sorted_head = sortList(head)

# Helper function to print ListNode
def print_list(node):
    while node:
        print(node.val, end="->")
        node = node.next
    print("None")

print_list(sorted_head)  # 1->2->3->4->None
```