Solutions to DSA Questions 80-110 (Trees, BST, Heaps, Graphs) For 1-2 Years

Experience Roles at EPAM Compiled on September 26, 2025

Introduction

This document provides detailed solutions for 31 Data Structures and Algorithms (DSA) problems (questions 80 to 110) from the Trees, Binary Search Trees (BST), Heaps, and Graphs categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity. Dynamic programming problems include both memoization and tabulation approaches where applicable.

Contents

1 Balanced Binary Tree

1.1 Problem Statement

Given a binary tree, determine if it is height-balanced (difference in heights of left and right subtrees <= 1).

1.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: True
- **Test Case 2**: root = [1,2,2,3,3,null,null,4,4] → Output: False
- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

1.3 Algorithm

1. Use DFS to compute height of each subtree.
2. Return -1 if unbalanced, else height.
3. Tree is balanced if root's height != -1.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$ (h = tree height)

1.4 Python Solution

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

def is_balanced(root):
    def check_height(node):
        if not node:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
```

```python
        right_height = check_height(node.right)
        if right_height == -1 or abs(left_height - right_height)
            > 1:
            return -1
        return max(left_height, right_height) + 1

    return check_height(root) != -1
```

2 Minimum Depth of Binary Tree

2.1 Problem Statement

Given a binary tree, find its minimum depth (shortest path from root to leaf).

2.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: 2
- **Test Case 2**: root = [2,null,3,null,4,null,5,null,6] → Output: 5
- **Test Case 3**: root = [] → Output: 0
- **Test Case 4**: root = [1] → Output: 1

2.3 Algorithm

1. Use BFS to find first leaf node.
2. Track depth while processing nodes level by level.
3. Return depth of first leaf.

Time Complexity: $O(n)$ **Space Complexity**: $O(w)$ (w = max width)

2.4 Python Solution

```python
from collections import deque

def min_depth(root):
    if not root:
        return 0

    queue = deque([(root, 1)])
    while queue:
        node, depth = queue.popleft()
        if not node.left and not node.right:
            return depth
        if node.left:
            queue.append((node.left, depth + 1))
        if node.right:
```

```
                queue.append((node.right, depth + 1))
```

3 Path Sum

3.1 Problem Statement

Given a binary tree and target sum, determine if there is a root-to-leaf path summing to target.

3.2 Dry Run on Test Cases

- **Test Case 1**: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], target = 22 → Output: True
- **Test Case 2**: root = [1,2,3], target = 5 → Output: False
- **Test Case 3**: root = [], target = 0 → Output: False
- **Test Case 4**: root = [1], target = 1 → Output: True

3.3 Algorithm

1. Use DFS, subtract node value from target.
2. At leaf, check if remaining sum is 0.
3. Return True if any path satisfies.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$

3.4 Python Solution

```python
def has_path_sum(root, target):
    if not root:
        return False
    if not root.left and not root.right:
        return target == root.val
    return has_path_sum(root.left, target - root.val) or
       has_path_sum(root.right, target - root.val)
```

4 Path Sum II

4.1 Problem Statement

Given a binary tree and target sum, return all root-to-leaf paths summing to target.

4.2 Dry Run on Test Cases

- **Test Case 1**: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], target = 22 → Output: [[5,4,11,2],[5,8,4,5]]
- **Test Case 2**: root = [1,2,3], target = 5 → Output: []
- **Test Case 3**: root = [1], target = 1 → Output: [[1]]
- **Test Case 4**: root = [], target = 0 → Output: []

4.3 Algorithm

1. Use DFS with backtracking.
2. Track current path and sum.
3. At leaf, add path to result if sum equals target.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$

4.4 Python Solution

```python
def path_sum(root, target):
    result = []

    def dfs(node, curr_sum, path):
        if not node:
            return
        curr_sum += node.val
        path.append(node.val)
        if not node.left and not node.right and curr_sum ==
           target:
            result.append(path[:])
        dfs(node.left, curr_sum, path)
        dfs(node.right, curr_sum, path)
        path.pop()

    dfs(root, 0, [])
    return result
```

5 Flatten Binary Tree to Linked List

5.1 Problem Statement

Given a binary tree, flatten it to a linked list in-place (preorder traversal).

5.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,5,3,4,null,6] → Output: [1,null,2,null,3,null,4,null,5,null,6]

- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,null,3] → Output: [1,null,2,null,3]

5.3 Algorithm

1. Use recursive preorder traversal.
2. For each node, flatten left and right subtrees.
3. Connect left subtree to right, set left to None.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$

5.4 Python Solution

```python
def flatten(root):
    def flatten_helper(node):
        if not node:
            return None
        if not node.left and not node.right:
            return node

        left_tail = flatten_helper(node.left)
        right_tail = flatten_helper(node.right)

        if left_tail:
            left_tail.right = node.right
            node.right = node.left
            node.left = None

        return right_tail if right_tail else left_tail if
           left_tail else node

    flatten_helper(root)
```

6 Construct Binary Tree from Preorder and Inorder Traversal

6.1 Problem Statement

Given preorder and inorder traversals, construct the binary tree.

6.2 Dry Run on Test Cases

- **Test Case 1**: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7] → Output: [3,9,20,null,null,15,7]
- **Test Case 2**: preorder = [1], inorder = [1] → Output: [1]

- **Test Case 3**: preorder = [], inorder = [] → Output: []
- **Test Case 4**: preorder = [1,2], inorder = [2,1] → Output: [1,2]

6.3 Algorithm

1. Use preorder's first element as root.
2. Find root in inorder to split left and right subtrees.
3. Recursively build left and right subtrees.

Time Complexity: $O(n)$ with hashmap **Space Complexity**: $O(n)$

6.4 Python Solution

```python
def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None

    inorder_index = {val: idx for idx, val in enumerate(inorder)}

    def build(pre_start, pre_end, in_start, in_end):
        if pre_start > pre_end:
            return None

        root_val = preorder[pre_start]
        root = TreeNode(root_val)
        root_idx = inorder_index[root_val]

        left_size = root_idx - in_start
        root.left = build(pre_start + 1, pre_start + left_size,
            in_start, root_idx - 1)
        root.right = build(pre_start + left_size + 1, pre_end,
            root_idx + 1, in_end)
        return root

    return build(0, len(preorder) - 1, 0, len(inorder) - 1)
```

7 Validate Binary Search Tree

7.1 Problem Statement

Given a binary tree, determine if it is a valid BST (left subtree < node < right subtree).

7.2 Dry Run on Test Cases

- **Test Case 1**: root = [2,1,3] → Output: True
- **Test Case 2**: root = [5,1,4,null,null,3,6] → Output: False

- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

7.3 Algorithm

1. Use DFS with range checking.
2. Each node must be within (min, max) range.
3. Update ranges for left (min, node.val) and right (node.val, max).

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$

7.4 Python Solution

```python
def is_valid_bst(root):
    def validate(node, min_val, max_val):
        if not node:
            return True
        if node.val <= min_val or node.val >= max_val:
            return False
        return validate(node.left, min_val, node.val) and
           validate(node.right, node.val, max_val)

    return validate(root, float('-inf'), float('inf'))
```

8 Kth Smallest Element in a BST

8.1 Problem Statement

Given a BST and integer k, find the kth smallest element.

8.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,1,4,null,2], k = 1 → Output: 1
- **Test Case 2**: root = [5,3,6,2,4,null,null,1], k = 3 → Output: 3
- **Test Case 3**: root = [1], k = 1 → Output: 1
- **Test Case 4**: root = [], k = 1 → Output: None

8.3 Algorithm

1. Perform inorder traversal (iterative).
2. Track count of visited nodes.
3. Return value when count == k.

Time Complexity: $O(h + k)$ **Space Complexity**: $O(h)$

8.4 Python Solution

```
def kth_smallest(root, k):
    stack = []
    curr = root
    count = 0

    while curr or stack:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        count += 1
        if count == k:
            return curr.val
        curr = curr.right
    return None
```

9 Lowest Common Ancestor in BST

9.1 Problem Statement

Given a BST and two nodes, find their lowest common ancestor.

9.2 Dry Run on Test Cases

- **Test Case 1**: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8 → Output: 6
- **Test Case 2**: root = [6,2,8], p = 2, q = 4 → Output: 2
- **Test Case 3**: root = [2,1], p = 2, q = 1 → Output: 2
- **Test Case 4**: root = [1], p = 1, q = 1 → Output: 1

9.3 Algorithm

1. Traverse from root.
2. If both p and q are less than root, go left.
3. If both greater, go right.
4. Else, root is LCA.

Time Complexity: $O(h)$ **Space Complexity**: $O(1)$

9.4 Python Solution

```python
def lowest_common_ancestor(root, p, q):
    curr = root
    while curr:
        if p.val < curr.val and q.val < curr.val:
            curr = curr.left
        elif p.val > curr.val and q.val > curr.val:
            curr = curr.right
        else:
            return curr
    return None
```

10 Binary Tree Zigzag Level Order Traversal

10.1 Problem Statement

Given a binary tree, return its zigzag level order traversal (left to right, then right to left).

10.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: [[3],[20,9],[15,7]]
- **Test Case 2**: root = [1] → Output: [[1]]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1,2,3,4,null,null,5] → Output: [[1],[3,2],[4,5]]

10.3 Algorithm

1. Use BFS with queue for level order.
2. Track direction (left-to-right or right-to-left).
3. Reverse level nodes if right-to-left.

Time Complexity: $O(n)$ **Space Complexity**: $O(w)$

10.4 Python Solution

```python
from collections import deque

def zigzag_level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])
    left_to_right = True
```

```python
    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        if not left_to_right:
            current_level.reverse()
        result.append(current_level)
        left_to_right = not left_to_right
    return result
```

11 Binary Tree Maximum Path Sum

11.1 Problem Statement

Given a binary tree, find the maximum path sum (path can include any node).

11.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3] → Output: 6 (2->1->3)
- **Test Case 2**: root = [-10,9,20,null,null,15,7] → Output: 42 (15->20->7)
- **Test Case 3**: root = [1] → Output: 1
- **Test Case 4**: root = [-3] → Output: -3

11.3 Algorithm

1. Use DFS to compute max path sum through each node.
2. Track global max sum.
3. For each node, return max path sum to parent (single branch).

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$

11.4 Python Solution

```python
def max_path_sum(root):
    max_sum = float('-inf')

    def max_gain(node):
        nonlocal max_sum
```

```python
        if not node:
            return 0
        left_gain = max(max_gain(node.left), 0)
        right_gain = max(max_gain(node.right), 0)
        current_sum = node.val + left_gain + right_gain
        max_sum = max(max_sum, current_sum)
        return node.val + max(left_gain, right_gain)

    max_gain(root)
    return max_sum
```

12 Insert into a BST

12.1 Problem Statement

Given a BST and a value, insert the value and return the root.

12.2 Dry Run on Test Cases

- **Test Case 1**: root = [4,2,7,1,3], val = 5 → Output: [4,2,7,1,3,5]
- **Test Case 2**: root = [], val = 1 → Output: [1]
- **Test Case 3**: root = [1], val = 2 → Output: [1,null,2]
- **Test Case 4**: root = [4,2,7], val = 4 → Output: [4,2,7,null,null,null,4]

12.3 Algorithm

1. If root is None, create new node.
2. If val < root.val, insert into left subtree.
3. If val > root.val, insert into right subtree.

Time Complexity: $O(h)$ **Space Complexity**: $O(h)$

12.4 Python Solution

```python
def insert_into_bst(root, val):
    if not root:
        return TreeNode(val)

    if val < root.val:
        root.left = insert_into_bst(root.left, val)
    elif val > root.val:
        root.right = insert_into_bst(root.right, val)
    return root
```

13 Delete Node in a BST

13.1 Problem Statement

Given a BST and a key, delete the node with that key and return the root.

13.2 Dry Run on Test Cases

* **Test Case 1**: root = [5,3,6,2,4,null,7], key = 3 → Output: [5,4,6,2,null,null,7]
* **Test Case 2**: root = [5,3,6,2,4,null,7], key = 0 → Output: [5,3,6,2,4,null,7]
* **Test Case 3**: root = [], key = 0 → Output: []
* **Test Case 4**: root = [1], key = 1 → Output: []

13.3 Algorithm

1. Find node to delete.
2. If leaf, remove it.
3. If one child, replace with child.
4. If two children, replace with successor (smallest in right subtree).

Time Complexity: $O(h)$ **Space Complexity**: $O(h)$

13.4 Python Solution

```python
def delete_node(root, key):
    if not root:
        return None

    if key < root.val:
        root.left = delete_node(root.left, key)
    elif key > root.val:
        root.right = delete_node(root.right, key)
    else:
        if not root.left:
            return root.right
        if not root.right:
            return root.left
        successor = root.right
        while successor.left:
            successor = successor.left
        root.val = successor.val
        root.right = delete_node(root.right, successor
            .val)
    return root
```

14 Binary Search Tree Iterator

14.1 Problem Statement

Implement an iterator over a BST with next() and hasNext() operations.

14.2 Dry Run on Test Cases

- **Test Case 1**: root = [7,3,15,null,null,9,20], next() → 3, next() → 7, hasNext() → True
- **Test Case 2**: root = [1], next() → 1, hasNext() → False
- **Test Case 3**: root = [], hasNext() → False
- **Test Case 4**: root = [3,1,4], next() → 1, next() → 3

14.3 Algorithm

1. Use stack for inorder traversal.
2. Push all left nodes from root.
3. next(): pop node, push all left nodes of its right subtree.
4. hasNext(): check if stack is non-empty.

Time Complexity: $O(1)$ average for next/hasNext **Space Complexity**: $O(h)$

14.4 Python Solution

```python
class BSTIterator:
    def __init__(self, root):
        self.stack = []
        self._push_left(root)

    def _push_left(self, node):
        while node:
            self.stack.append(node)
            node = node.left

    def next(self):
        node = self.stack.pop()
        self._push_left(node.right)
        return node.val

    def hasNext(self):
        return len(self.stack) > 0
```

15 Binary Tree Right Side View

15.1 Problem Statement

Given a binary tree, return the values of nodes visible from the right side.

15.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3,null,5,null,4] → Output: [1,3,4]
- **Test Case 2**: root = [1,null,3] → Output: [1,3]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1] → Output: [1]

15.3 Algorithm

1. Use BFS, process level by level.
2. Add last node of each level to result.

Time Complexity: $O(n)$ **Space Complexity**: $O(w)$

15.4 Python Solution

```python
from collections import deque

def right_side_view(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        for i in range(level_size):
            node = queue.popleft()
            if i == level_size - 1:
                result.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
    return result
```

16 Count Complete Tree Nodes

16.1 Problem Statement

Given a complete binary tree, count the number of nodes.

16.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3,4,5,6] $\rightarrow$ Output: 6
- **Test Case 2**: root = [] $\rightarrow$ Output: 0
- **Test Case 3**: root = [1] $\rightarrow$ Output: 1
- **Test Case 4**: root = [1,2,3,4] $\rightarrow$ Output: 4

16.3 Algorithm

1. Check if left and right heights are equal.
2. If equal, return $2^h - 1$ (perfect binary tree).
3. Else, recursively count left and right subtrees.

Time Complexity: $O(\log^2 n)$ **Space Complexity**: $O(\log n)$

16.4 Python Solution

```python
def count_nodes(root):
    def get_height(node, direction):
        height = 0
        while node:
            height += 1
            node = node.left if direction == 'left'
                else node.right
        return height

    if not root:
        return 0

    left_height = get_height(root, 'left')
    right_height = get_height(root, 'right')

    if left_height == right_height:
        return (1 << left_height) - 1
    return 1 + count_nodes(root.left) + count_nodes
        (root.right)
```

17 Kth Largest Element in an Array

17.1 Problem Statement

Given an array and integer k, find the kth largest element.

17.2 Dry Run on Test Cases

- **Test Case 1**: nums = [3,2,1,5,6,4], k = 2 → Output: 5
- **Test Case 2**: nums = [3,2,3,1,2,4,5,5,6], k = 4 → Output: 4
- **Test Case 3**: nums = [1], k = 1 → Output: 1
- **Test Case 4**: nums = [], k = 1 → Output: None

17.3 Algorithm

1. Use min-heap of size k.
2. Push elements; if heap size > k, pop smallest.
3. Return heap top.

Time Complexity: $O(n \log k)$ **Space Complexity**: $O(k)$

17.4 Python Solution

```python
import heapq

def find_kth_largest(nums, k):
    if not nums:
        return None
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap[0]
```

18 Top K Frequent Elements

18.1 Problem Statement

Given an array and integer k, return the k most frequent elements.

18.2 Dry Run on Test Cases

- **Test Case 1**: nums = [1,1,1,2,2,3], k = 2 → Output: [1,2]
- **Test Case 2**: nums = [1], k = 1 → Output: [1]
- **Test Case 3**: nums = [1,2], k = 2 → Output: [1,2]
- **Test Case 4**: nums = [], k = 1 → Output: []

18.3 Algorithm

1. Count frequency of each element.
2. Use min-heap of size k to store elements by frequency.
3. Return heap elements.

Time Complexity: $O(n \log k)$ **Space Complexity**: $O(n)$

18.4 Python Solution

```
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    if not nums:
        return []

    count = Counter(nums)
    heap = []
    for num, freq in count.items():
        heapq.heappush(heap, (freq, num))
        if len(heap) > k:
            heapq.heappop(heap)

    return [num for _, num in heap]
```

19 Median from Data Stream

19.1 Problem Statement

Design a data structure to find the median of a stream of numbers.

19.2 Dry Run on Test Cases

- **Test Case 1**: addNum(1), addNum(2), findMedian() $\rightarrow$ 1.5
- **Test Case 2**: addNum(3), findMedian() $\rightarrow$ 3
- **Test Case 3**: addNum(1), addNum(2), addNum(3), findMedian() $\rightarrow$ 2
- **Test Case 4**: addNum(4), addNum(5), findMedian() $\rightarrow$ 4.5

19.3 Algorithm

1. Use two heaps: max-heap for lower half, min-heap for upper half.
2. Balance heaps so max-heap has at most one more element.

3. Median is average of tops or top of max-heap.

Time Complexity: $O(\log n)$ for addNum, $O(1)$ for findMedian **Space Complexity**: $O(n)$

19.4 Python Solution

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # Max heap (negated values)
        self.large = []  # Min heap

    def addNum(self, num):
        if len(self.small) == 0 or num < -self.small[0]:
            heapq.heappush(self.small, -num)
        else:
            heapq.heappush(self.large, num)

        # Balance heaps
        if len(self.small) > len(self.large) + 1:
            heapq.heappush(self.large, -heapq.heappop(self.small))
        elif len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2
```

20 Merge k Sorted Lists

20.1 Problem Statement

Given k sorted linked lists, merge them into one sorted linked list.

20.2 Dry Run on Test Cases

- **Test Case 1**: lists = [[1,4,5],[1,3,4],[2,6]] → Output: [1,1,2,3,4,4,5,6]
- **Test Case 2**: lists = [] → Output: []
- **Test Case 3**: lists = [[]] → Output: []
- **Test Case 4**: lists = [[1]] → Output: [1]

20.3 Algorithm

1. Use min-heap to store heads of lists.
2. Pop smallest node, add to result, push next node if exists.
3. Continue until heap is empty.

Time Complexity: $O(n \log k)$ (n = total nodes) **Space Complexity**: $O(k)$

20.4 Python Solution

```python
import heapq

def merge_k_lists(lists):
    dummy = ListNode(0)
    curr = dummy
    heap = []

    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst.val, i, lst))

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

    return dummy.next
```

21 Find Median in Two Sorted Arrays

21.1 Problem Statement

Given two sorted arrays, find the median of the merged array.

21.2 Dry Run on Test Cases

- **Test Case 1**: nums1 = [1,3], nums2 = [2] → Output: 2.0
- **Test Case 2**: nums1 = [1,2], nums2 = [3,4] → Output: 2.5
- **Test Case 3**: nums1 = [], nums2 = [1] → Output: 1.0
- **Test Case 4**: nums1 = [2], nums2 = [] → Output: 2.0

21.3 Algorithm

1. Use binary search on smaller array to find partition.
2. Ensure left side of partition <= right side for both arrays.
3. Compute median based on total length (odd or even).

Time Complexity: $O(\log \min(m, n))$ **Space Complexity**: $O(1)$

21.4 Python Solution

```python
def find_median_sorted_arrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    left, right = 0, x

    while left <= right:
        partition_x = (left + right) // 2
        partition_y = (x + y + 1) // 2 - partition_x

        left_x = nums1[partition_x - 1] if partition_x > 0 else float('-inf')
        right_x = nums1[partition_x] if partition_x < x else float('inf')
        left_y = nums2[partition_y - 1] if partition_y > 0 else float('-inf')
        right_y = nums2[partition_y] if partition_y < y else float('inf')

        if left_x <= right_y and left_y <= right_x:
            if (x + y) % 2 == 0:
                return (max(left_x, left_y) + min(right_x, right_y)) / 2
            return max(left_x, left_y)
        elif left_x > right_y:
            right = partition_x - 1
        else:
            left = partition_x + 1
```

22 Design Min Heap

22.1 Problem Statement

Implement a min-heap with insert, extract$_{m}in, andget_{m}inoperations.$

22.2 Dry Run on Test Cases

- **Test Case 1**: insert(3), insert(2), get$_{m}in()$→ 2, extract$_{m}in()$→ 2
- **Test Case 2**: insert(1), get$_{m}in()$→ 1
- **Test Case 3**: extract$_{m}in()$→ None
- **Test Case 4**: insert(5), insert(1), extract$_{m}in()$→ 1

22.3 Algorithm

1. Use array to store heap.
2. Insert: append and bubble up.
3. Extract$_{m}in : removeroot, placelastelementatroot, bubbledown. Get_{m}in : returnroot$. **Time Complexity**: $O(\log n)$ for insert/extract$_{m}in$,O(1)$forget_{m}in$ **Space Complexity** :O(n)

22.4 Python Solution

```python
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self._bubble_up(len(self.heap) - 1)

    def extract_min(self):
        if not self.heap:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        min_val = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._bubble_down(0)
        return min_val

    def get_min(self):
        return self.heap[0] if self.heap else None

    def _bubble_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] < self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            self._bubble_up(parent)

    def _bubble_down(self, index):
```

```python
        smallest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < len(self.heap) and self.heap[left
           ] < self.heap[smallest]:
            smallest = left
        if right < len(self.heap) and self.heap[
           right] < self.heap[smallest]:
            smallest = right

        if smallest != index:
            self.heap[index], self.heap[smallest] =
                self.heap[smallest], self.heap[
               index]
            self._bubble_down(smallest)
```

23 Find K Closest Points to Origin

23.1 Problem Statement

Given an array of points and integer k, return the k closest points to the origin (0,0).

23.2 Dry Run on Test Cases

4. **Test Case 1**: points = [[1,3],[-2,2]], k = 1 → Output: [[-2,2]]

- **Test Case 2**: points = [[3,3],[5,-1],[-2,4]], k = 2 → Output: [[3,3],[-2,4]]
- **Test Case 3**: points = [[1,1]], k = 1 → Output: [[1,1]]
- **Test Case 4**: points = [], k = 1 → Output: []

23.3 Algorithm

1. Use max-heap of size k to store points by distance.
2. For each point, compute distance, push to heap, pop if size > k.
3. Return heap contents.

Time Complexity: $O(n \log k)$ **Space Complexity**: $O(k)$

23.4 Python Solution

```python
import heapq

def k_closest(points, k):
    if not points:
        return []
```

```python

    heap = []
    for x, y in points:
        dist = -(x*x + y*y)  # Negate for max heap
        heapq.heappush(heap, (dist, x, y))
        if len(heap) > k:
            heapq.heappop(heap)

    return [[x, y] for _, x, y in heap]
```

24 Course Schedule

24.1 Problem Statement

Given a number of courses and prerequisites, determine if all courses can be completed.

24.2 Dry Run on Test Cases

- **Test Case 1**: numCourses = 2, prerequisites = [[1,0]] → Output: True
- **Test Case 2**: numCourses = 2, prerequisites = [[1,0],[0,1]] → Output: False
- **Test Case 3**: numCourses = 1, prerequisites = [] → Output: True
- **Test Case 4**: numCourses = 3, prerequisites = [[1,0],[2,1]] → Output: True

24.3 Algorithm

1. Build adjacency list and in-degree count.
2. Use topological sort with queue for nodes with in-degree 0.
3. If all courses visited, return True; else, cycle exists.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V + E)$

24.4 Python Solution

```python
from collections import defaultdict, deque

def can_finish(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    queue = deque([i for i in range(numCourses) if
       in_degree[i] == 0])
```

```python
    count = 0

    while queue:
        node = queue.popleft()
        count += 1
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return count == numCourses
```

25 Course Schedule II

25.1 Problem Statement

Given a number of courses and prerequisites, return the order to take all courses.

25.2 Dry Run on Test Cases

- **Test Case 1**: numCourses = 2, prerequisites = [[1,0]] → Output: [0,1]
- **Test Case 2**: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]] → Output: [0,1,2,3] or [0,2,1,3]
- **Test Case 3**: numCourses = 1, prerequisites = [] → Output: [0]
- **Test Case 4**: numCourses = 2, prerequisites = [[1,0],[0,1]] → Output: []

25.3 Algorithm

1. Build adjacency list and in-degree count.
2. Use topological sort, add nodes with in-degree 0 to queue.
3. Collect order; return empty list if cycle detected.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V + E)$

25.4 Python Solution

```python
from collections import defaultdict, deque

def find_order(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1
```

```
        queue = deque([i for i in range(numCourses) if
            in_degree[i] == 0])
        order = []

        while queue:
            node = queue.popleft()
            order.append(node)
            for neighbor in graph[node]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

        return order if len(order) == numCourses else
            []
```

26 Clone Graph

26.1 Problem Statement

Given a graph node, return a deep copy (clone) of the graph.

26.2 Dry Run on Test Cases

- **Test Case 1**: node = [[2,4],[1,3],[2,4],[1,3]] → Output: Cloned graph
- **Test Case 2**: node = [[]] → Output: Cloned single node
- **Test Case 3**: node = [] → Output: None
- **Test Case 4**: node = [[2],[1]] → Output: Cloned graph

26.3 Algorithm

1. Use DFS with hashmap to store cloned nodes.
2. For each node, create clone, recursively clone neighbors.
3. Avoid cycles using hashmap.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V)$

26.4 Python Solution

```
class Node:
    def __init__(self, val=0, neighbors=None):
        self.val = val
        self.neighbors = neighbors if neighbors is
            not None else []

```

```python
def clone_graph(node):
    if not node:
        return None

    cloned = {}

    def dfs(node):
        if node in cloned:
            return cloned[node]

        clone = Node(node.val)
        cloned[node] = clone
        for neighbor in node.neighbors:
            clone.neighbors.append(dfs(neighbor))
        return clone

    return dfs(node)
```

27 Number of Islands

27.1 Problem Statement

Given a 2D grid of '1's (land) and '0's (water), count the number of islands.

27.2 Dry Run on Test Cases

- **Test Case 1**: grid = [["1","1","1"],["0","1","0"],["1","1","1"]] → Output: 1
- **Test Case 2**: grid = [["1","1","0","0","0"],["1","1","0","0","0"],["0","0","1","0","0"],["0","0","0","1","1"]] → Output: 3
- **Test Case 3**: grid = [] → Output: 0
- **Test Case 4**: grid = [["1"]] → Output: 1

27.3 Algorithm

1. Use DFS to mark all connected '1's as visited.
2. For each unvisited '1', increment island count and perform DFS.
3. Return total islands.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$ for recursion

27.4 Python Solution

```python
def num_islands(grid):
    if not grid:
        return 0
```

```python

    rows, cols = len(grid), len(grid[0])
    islands = 0

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != "1":
            return
        grid[i][j] = "0"  # Mark as visited
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == "1":
                islands += 1
                dfs(i, j)
    return islands
```

28 Flood Fill

28.1 Problem Statement

Given an image, starting pixel, and new color, flood fill the connected pixels of the same color.

28.2 Dry Run on Test Cases

- **Test Case 1**: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2 → Output: [[2,2,2],[2,2,0],[2,0,1]]
- **Test Case 2**: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, newColor = 0 → Output: [[0,0,0],[0,0,0]]
- **Test Case 3**: image = [[1]], sr = 0, sc = 0, newColor = 2 → Output: [[2]]
- **Test Case 4**: image = [], sr = 0, sc = 0, newColor = 1 → Output: []

28.3 Algorithm

1. Use DFS to change color of connected pixels.
2. If pixel matches old color, change to new color and recurse on neighbors.
3. Return modified image.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

28.4 Python Solution

```python
def flood_fill(image, sr, sc, newColor):
    if not image or image[sr][sc] == newColor:
        return image

    rows, cols = len(image), len(image[0])
    old_color = image[sr][sc]

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or image[i][j] != old_color:
            return
        image[i][j] = newColor
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    dfs(sr, sc)
    return image
```

29 Rotting Oranges

29.1 Problem Statement

Given a grid of oranges (fresh=1, rotten=2, empty=0), return the minimum time for all to rot.

29.2 Dry Run on Test Cases

- **Test Case 1**: grid = [[2,1,1],[1,1,0],[0,1,1]] → Output: 4
- **Test Case 2**: grid = [[2,1,1],[0,1,1],[1,0,1]] → Output: -1
- **Test Case 3**: grid = [[0,2]] → Output: 0
- **Test Case 4**: grid = [] → Output: 0

29.3 Algorithm

1. Use BFS starting from all rotten oranges.
2. Track time and fresh oranges.
3. If fresh remain, return -1; else return time.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

29.4 Python Solution

```python
from collections import deque

def oranges_rotting(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 2:
                queue.append((i, j))
            elif grid[i][j] == 1:
                fresh += 1

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

    while queue and fresh:
        for _ in range(len(queue)):
            i, j = queue.popleft()
            for di, dj in directions:
                ni, nj = i + di, j + dj
                if 0 <= ni < rows and 0 <= nj <
                    cols and grid[ni][nj] == 1:
                    grid[ni][nj] = 2
                    fresh -= 1
                    queue.append((ni, nj))
        time += 1

    return time if fresh == 0 else -1
```

30 Word Ladder

30.1 Problem Statement

Given two words and a word list, find the shortest transformation sequence length from beginWord to endWord.

30.2 Dry Run on Test Cases

- **Test Case 1**: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"] → Output: 5
- **Test Case 2**: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"] → Output: 0
- **Test Case 3**: beginWord = "a", endWord = "c", wordList = ["a","b","c"] → Output: 2

- **Test Case 4**: beginWord = "hit", endWord = "hit", wordList = [] → Output: 1

30.3 Algorithm

1. Use BFS to find shortest path.
2. For each word, generate all possible one-letter changes.
3. Track visited words and level.

Time Complexity: $O(N \cdot 26 \cdot L)$ (N = wordList size, L = word length) **Space Complexity**: $O(N)$

30.4 Python Solution

```python
from collections import deque, defaultdict

def ladder_length(beginWord, endWord, wordList):
    if endWord not in wordList:
        return 0

    word_set = set(wordList)
    queue = deque([(beginWord, 1)])
    visited = {beginWord}

    while queue:
        word, level = queue.popleft()
        if word == endWord:
            return level

        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                new_word = word[:i] + c + word[i
                    +1:]
                if new_word in word_set and
                    new_word not in visited:
                     visited.add(new_word)
                     queue.append((new_word, level +
                         1))

    return 0
```

31 Shortest Path in Binary Matrix

31.1 Problem Statement

Given an n x n binary matrix, find the shortest path length from (0,0) to (n-1,n-1) with 0s.

31.2 Dry Run on Test Cases

- **Test Case 1**: grid = [[0,1],[1,0]] → Output: 2
- **Test Case 2**: grid = [[0,0,0],[1,1,0],[1,1,0]] → Output: 4
- **Test Case 3**: grid = [[1,0],[0,1]] → Output: -1
- **Test Case 4**: grid = [[0]] → Output: 1

31.3 Algorithm

1. Use BFS starting from (0,0).
2. Explore 8 directions for each cell.
3. Return distance to (n-1,n-1) or -1 if unreachable.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(n^2)$

31.4 Python Solution

```
from collections import deque

def shortest_path_binary_matrix(grid):
    if not grid or grid[0][0] == 1:
        return -1

    n = len(grid)
    if n == 1:
        return 1 if grid[0][0] == 0 else -1

    queue = deque([(0, 0, 1)])
    grid[0][0] = 1  # Mark as visited
    directions = [(1,0), (-1,0), (0,1), (0,-1),
        (1,1), (-1,-1), (1,-1), (-1,1)]

    while queue:
        i, j, dist = queue.popleft()
        if i == n-1 and j == n-1:
            return dist

        for di, dj in directions:
            ni, nj = i + di, j + dj
            if 0 <= ni < n and 0 <= nj < n and grid
                [ni][nj] == 0:
                grid[ni][nj] = 1
                queue.append((ni, nj, dist + 1))

    return -1
```