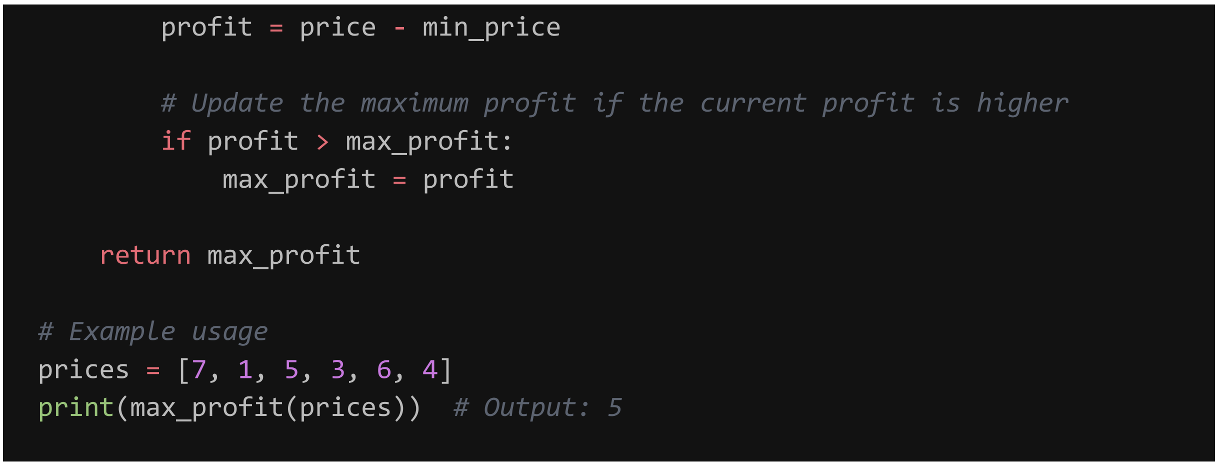
BLIND 75 Sheet

2Sum Problem

```python
def two_sum(nums, target):
    # Create a dictionary to store the indices of the numbers we have seen so far
    num_to_index = {}

    for i, num in enumerate(nums):
        # Calculate the complement that we need to find
        complement = target - num

        # If the complement is in the dictionary, return the indices
        if complement in num_to_index:
            return [num_to_index[complement], i]

        # Otherwise, store the index of the current number in the dictionary
        num_to_index[num] = i

    # If no solution is found, return an empty list
    return []

# Example usage
nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # Output: [0, 1]
```

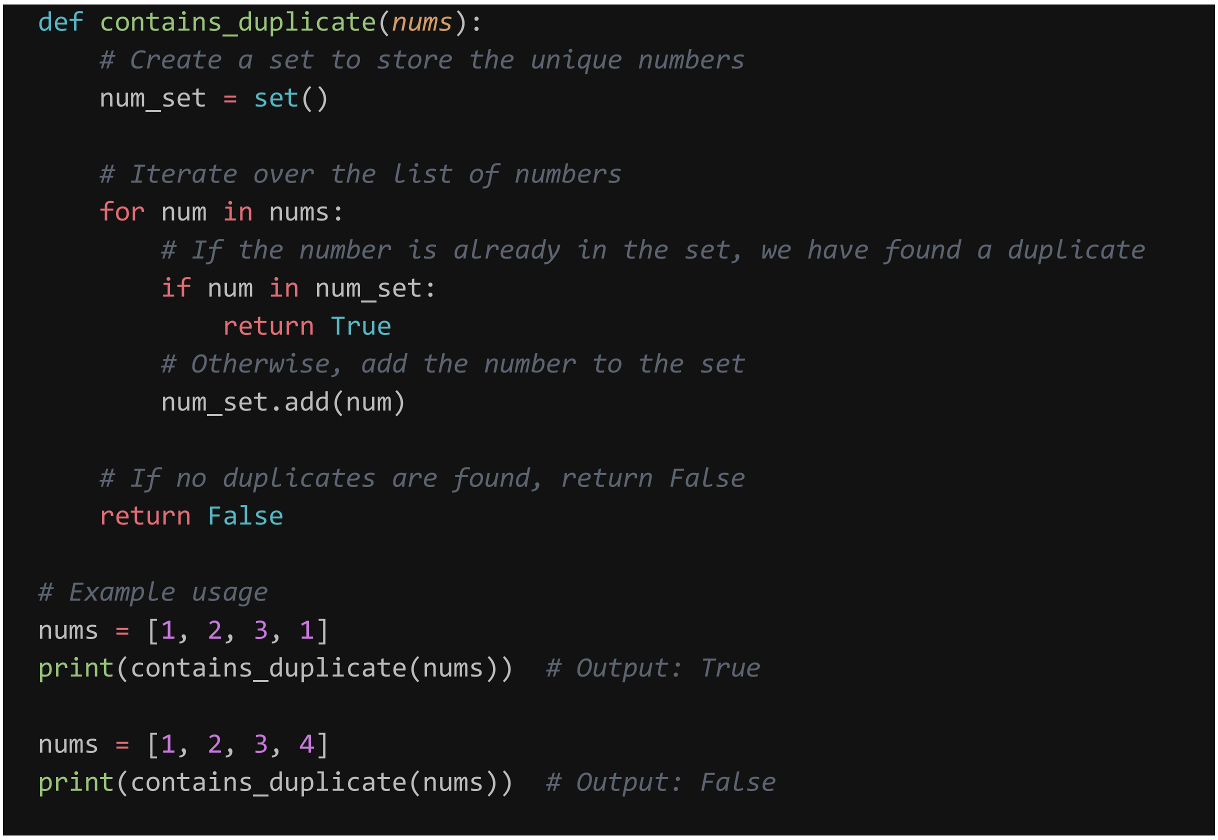
Best Time to Buy and Sell Stock |(DP-35)

```python
def max_profit(prices):
    if not prices:
        return 0

    # Initialize the minimum price to the first price in the list
    min_price = prices[0]
    max_profit = 0

    # Iterate over the list of prices
    for price in prices:
        # Update the minimum price if the current price is lower
        if price < min_price:
            min_price = price

        # Calculate the profit if selling at the current price
```



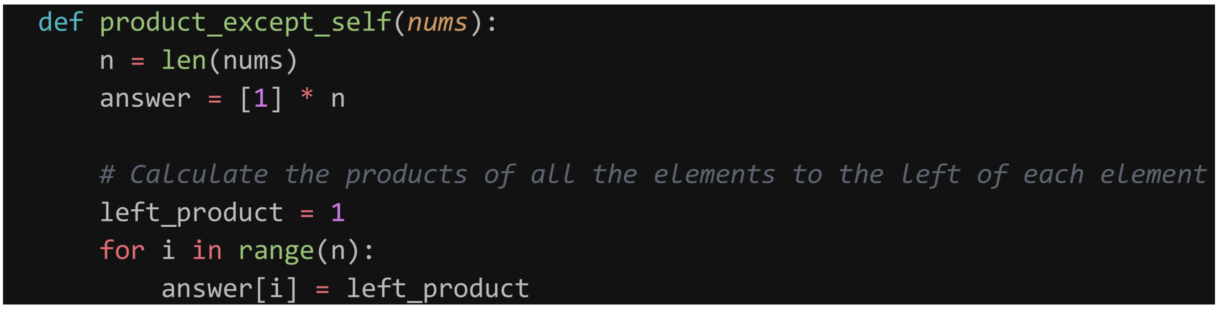
```
        profit = price - min_price

        # Update the maximum profit if the current profit is higher
        if profit > max_profit:
            max_profit = profit

    return max_profit

# Example usage
prices = [7, 1, 5, 3, 6, 4]
print(max_profit(prices))  # Output: 5
```

Contains Duplicate



```
def contains_duplicate(nums):
    # Create a set to store the unique numbers
    num_set = set()

    # Iterate over the list of numbers
    for num in nums:
        # If the number is already in the set, we have found a duplicate
        if num in num_set:
            return True
        # Otherwise, add the number to the set
        num_set.add(num)

    # If no duplicates are found, return False
    return False

# Example usage
nums = [1, 2, 3, 1]
print(contains_duplicate(nums))  # Output: True

nums = [1, 2, 3, 4]
print(contains_duplicate(nums))  # Output: False
```

Product of Array Except Self



```
def product_except_self(nums):
    n = len(nums)
    answer = [1] * n

    # Calculate the products of all the elements to the left of each element
    left_product = 1
    for i in range(n):
        answer[i] = left_product
```

```
          left_product *= nums[i]

      # Calculate the products of all the elements to the right of each
element
      right_product = 1
      for i in range(n - 1, -1, -1):
          answer[i] *= right_product
          right_product *= nums[i]

      return answer

  # Example usage
  nums = [1, 2, 3, 4]
  print(product_except_self(nums))  # Output: [24, 12, 8, 6]
```

Kadane's Algorithm, maximum subarray sum

```
  def max_subarray_sum(nums):
      # Initialize variables to keep track of the maximum sum so far and the
current sum
      max_so_far = nums[0]
      max_ending_here = nums[0]

      # Iterate through the array starting from the second element
      for num in nums[1:]:
          # Update the current sum by including the current number or starting
anew from the current number
          max_ending_here = max(num, max_ending_here + num)

          # Update the maximum sum so far if the current sum is greater
          max_so_far = max(max_so_far, max_ending_here)

      return max_so_far

  # Example usage
  nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
  print(max_subarray_sum(nums))  # Output: 6 (subarray: [4, -1, 2, 1])
```

Maximum Product Subarray

```
  def max_product(nums):
      if not nums:
          return 0

      # Initialize variables to store the maximum and minimum product up to
the current position
```

```
    max_product_so_far = nums[0]
    min_product_so_far = nums[0]
    result = nums[0]

    # Iterate through the array starting from the second element
    for num in nums[1:]:
        if num < 0:
            # Swap max and min when encountering a negative number
            max_product_so_far, min_product_so_far = min_product_so_far,
max_product_so_far

        # Update max and min product so far
        max_product_so_far = max(num, max_product_so_far * num)
        min_product_so_far = min(num, min_product_so_far * num)

        # Update the result to be the maximum product so far
        result = max(result, max_product_so_far)

    return result

# Example usage
nums = [2, 3, -2, 4]
print(max_product(nums))  # Output: 6 (subarray: [2, 3])
```

Find minimum in Rotated Sorted Array

```
def find_min(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2

        # If mid element is greater than the rightmost element,
        # the minimum element must be in the right part
        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            # If mid element is less than or equal to the rightmost element,
            # the minimum element is in the left part (including mid)
            right = mid

    return nums[left]

# Example usage
nums = [4, 5, 6, 7, 0, 1, 2]
print(find_min(nums))  # Output: 0
```

```
nums = [3, 4, 5, 1, 2]
print(find_min(nums))  # Output: 1
```

Search in Rotated Sorted Array I

```
def search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        if nums[mid] == target:
            return mid

        # Determine which side is properly sorted
        if nums[left] <= nums[mid]:
            # Left side is sorted
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            # Right side is sorted
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1

# Example usage
nums = [4, 5, 6, 7, 0, 1, 2]
target = 0
print(search(nums, target))  # Output: 4

target = 3
print(search(nums, target))  # Output: -1
```

3 sum

```
def three_sum(nums):
    nums.sort()  # Sort the array to use two-pointer technique
    result = []
    n = len(nums)

    for i in range(n - 2):
        if i > 0 and nums[i] == nums[i - 1]:
            continue  # Skip duplicate values

        left, right = i + 1, n - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            if total < 0:
                left += 1
            elif total > 0:
                right -= 1
            else:
                result.append([nums[i], nums[left], nums[right]])
                # Skip duplicates
                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

    return result

# Example usage
nums = [-1, 0, 1, 2, -1, -4]
print(three_sum(nums))  # Output: [[-1, -1, 2], [-1, 0, 1]]
```

Container with most water

```
def max_area(height):
    left, right = 0, len(height) - 1
    max_water = 0

    while left < right:
        # Calculate the area with the current left and right pointers
        width = right - left
        current_height = min(height[left], height[right])
        current_area = width * current_height
        max_water = max(max_water, current_area)

        # Move the pointer that is at the smaller height
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1

    return max_water
```

```python
# Example usage
height = [1,8,6,2,5,4,8,3,7]
print(max_area(height))  # Output: 49
```

Sum of Two Integers

```python
def get_sum(a, b):
    # 32-bit mask to get the last 32 bits
    mask = 0xFFFFFFFF

    # Perform addition while considering the range of integers
    while b != 0:
        # Perform addition without carrying using XOR
        a, b = (a ^ b) & mask, ((a & b) << 1) & mask

    # Handle overflow for negative numbers
    return a if a <= 0x7FFFFFFF else ~(a ^ mask)

# Example usage
a = 3
b = 2
print(get_sum(a, b))  # Output: 5

a = -2
b = 3
print(get_sum(a, b))  # Output: 1
```

Number of 1 Bits

```python
def hamming_weight(n):
    count = 0
    while n:
        count += n & 1
        n >>= 1
    return count

# Example usage
n = 11  # Binary representation: 1011
print(hamming_weight(n))  # Output: 3 (three 1 bits)

n = 128  # Binary representation: 10000000
print(hamming_weight(n))  # Output: 1 (one 1 bit)
```

Counting Bits

```
def count_bits(n):
    # Initialize a list to store the number of 1 bits for each number from 0 to n
    bits = [0] * (n + 1)

    # Iterate through each number from 1 to n
    for i in range(1, n + 1):
        # The number of 1 bits in i is equal to the number of 1 bits in i // 2
        # plus 1 if i is odd (i & 1 checks if i is odd)
        bits[i] = bits[i >> 1] + (i & 1)

    return bits

# Example usage
n = 5
print(count_bits(n))  # Output: [0, 1, 1, 2, 1, 2]

n = 10
print(count_bits(n))  # Output: [0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2]
```

Find missing number in an array

```
def find_missing_number(nums):
    n = len(nums)
    missing_number = n
    for i in range(n):
        missing_number ^= i ^ nums[i]
    return missing_number

# Example usage
nums = [3, 0, 1]
print(find_missing_number(nums))  # Output: 2
```

Reverse Bits

```
def reverse_bits(n):
    result = 0
    for _ in range(32):  # Since it's a 32-bit integer
        result = (result << 1) | (n & 1)  # Shift result left and add the least significant bit of n
        n >>= 1  # Shift n to the right
    return result

# Example usage
n = 43261596  # Binary representation: 00000010100101000001111010011100
```

```
  print(reverse_bits(n))  # Output: 964176192  (Binary representation after
reversing: 00111001011110000010100101000000)
```

Climbing Stars

```
  def climb_stairs(n):
      if n == 0:
          return 1  # Base case: 1 way to stay at the ground level (doing
nothing)

      # Initialize an array to store the number of ways to reach each step
      ways = [0] * (n + 1)
      ways[0] = 1  # Base case: 1 way to stay at step 0 (doing nothing)
      ways[1] = 1  # 1 way to reach step 1 (take 1 step)

      # Calculate number of ways for each step up to n
      for i in range(2, n + 1):
          ways[i] = ways[i - 1] + ways[i - 2]

      return ways[n]

  # Example usage
  n = 4
  print(climb_stairs(n))  # Output: 5 (ways to climb: [1, 1, 1, 1], [1, 1, 2],
[1, 2, 1], [2, 1, 1], [2, 2])
```

Coin change

```
  def coin_change(coins, amount):
      # Initialize dp array with a value larger than any possible number of
coins (amount + 1)
      dp = [float('inf')] * (amount + 1)
      dp[0] = 0  # 0 coins needed to make amount 0

      for i in range(1, amount + 1):
          for coin in coins:
              if coin <= i:
                  dp[i] = min(dp[i], dp[i - coin] + 1)

      return dp[amount] if dp[amount] != float('inf') else -1

  # Example usage:
  coins = [1, 2, 5]
  amount = 11
  print(coin_change(coins, amount))  # Output: 3 (11 = 5 + 5 + 1)
```

Longest Increasing Subsequence

```
def length_of_lis(nums):
    if not nums:
        return 0

    n = len(nums)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Example usage:
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis(nums))  # Output: 4 (The longest increasing subsequence
is [2, 3, 7, 101])
```

Longest Common Subsequence

```
def longest_common_subsequence(text1, text2):
    m = len(text1)
    n = len(text2)

    # Initialize dp table
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage:
text1 = "abcde"
text2 = "ace"
print(longest_common_subsequence(text1, text2))  # Output: 3 (LCS is "ace")
```

Word Break

```python
def word_break(s, wordDict):
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True  # Base case: empty string can always be segmented

    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break

    return dp[n]

# Example usage:
s = "leetcode"
wordDict = ["leet", "code"]
print(word_break(s, wordDict))  # Output: True (can be segmented into "leet" and "code")
```

Combination Sum

```python
def combination_sum(candidates, target):
    # Sort candidates to handle duplicates efficiently
    candidates.sort()
    results = []

    def backtrack(start, target, path):
        if target == 0:
            results.append(path)
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                break
            backtrack(i, target - candidates[i], path + [candidates[i]])

    backtrack(0, target, [])
    return results

# Example usage:
candidates = [2, 3, 6, 7]
target = 7
print(combination_sum(candidates, target))  # Output: [[2, 2, 3], [7]]
```

Maximum sum of non-adjacent elements (DP 5)

```python
def max_sum_non_adjacent(nums):
    if not nums:
        return 0

    n = len(nums)
    if n == 1:
        return nums[0]

    incl = nums[0]
    excl = 0

    for i in range(1, n):
        new_incl = excl + nums[i]
        excl = max(incl, excl)
        incl = new_incl

    return max(incl, excl)

# Example usage:
nums = [2, 4, 6, 2, 5]
print(max_sum_non_adjacent(nums))  # Output: 13 (maximum sum of non-adjacent elements: 2 + 6 + 5)
```

House Robber (DP 6)

```
def rob(nums):
    n = len(nums)
    if n == 0:
        return 0
    elif n == 1:
        return nums[0]

    dp = [0] * n
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in range(2, n):
        dp[i] = max(dp[i-1], dp[i-2] + nums[i])

    return dp[-1]

# Example usage:
nums = [2, 7, 9, 3, 1]
print(rob(nums))  # Output: 12 (Rob houses with amounts [2, 9, 1])
```

Decode Ways

```
def num_decodings(s):
    n = len(s)
    if n == 0 or s[0] == '0':
        return 0

    dp = [0] * (n + 1)
    dp[0] = 1  # Empty string has one way to decode
    dp[1] = 1 if s[0] != '0' else 0

    for i in range(2, n + 1):
        # Single digit
        if s[i-1] != '0':
            dp[i] += dp[i-1]

        # Two digits
        two_digit = int(s[i-2:i])
        if 10 <= two_digit <= 26:
            dp[i] += dp[i-2]

    return dp[n]

# Example usage:
s = "226"
print(num_decodings(s))  # Output: 3 (Possible decodings are "BZ", "VF", and "BBF")
```

Grid Unique Paths

```
def unique_paths(m, n):
    # Initialize a 2D dp array with all zeros
    dp = [[0] * n for _ in range(m)]

    # Base case: There is one way to reach the starting point
    dp[0][0] = 1

    # Fill the dp array
    for i in range(m):
        for j in range(n):
            if i > 0:
                dp[i][j] += dp[i-1][j]  # Add ways from above
            if j > 0:
                dp[i][j] += dp[i][j-1]  # Add ways from left

    return dp[m-1][n-1]
```

```python
# Example usage:
m = 3
n = 7
print(unique_paths(m, n))  # Output: 28 (Number of unique paths from top-left to bottom-right)
```

Jump Game

```python
def can_jump(nums):
    max_reach = 0
    n = len(nums)

    for i in range(n):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + nums[i])
        if max_reach >= n - 1:
            return True

    return True  # In case the array is empty or has only one element

# Example usage:
nums = [2, 3, 1, 1, 4]
print(can_jump(nums))  # Output: True (can jump to the last index)
```

Clone Graph

```python
from collections import deque

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

def cloneGraph(node):
    if not node:
        return None

    # Dictionary to store original node -> cloned node mapping
    clone_map = {}
    queue = deque([node])

    # Create clone of the starting node
```

```
      clone_map[node] = Node(node.val)

      while queue:
          current_node = queue.popleft()

          for neighbor in current_node.neighbors:
              if neighbor not in clone_map:
                  # If neighbor hasn't been cloned, create a new clone
                  clone_map[neighbor] = Node(neighbor.val)
                  queue.append(neighbor)

         # Add the cloned neighbor to the cloned node's neighbors list
              clone_map[current_node].neighbors.append(clone_map[neighb
or])

      # Return the clone of the starting node
      return clone_map[node]
```

Course Schedule – I

```
  def canFinish(numCourses, prerequisites):
      # Build the graph
      graph = [[] for _ in range(numCourses)]
      for course, prereq in prerequisites:
          graph[course].append(prereq)

      # Track visited states: 0 - not visited, 1 - visiting, 2 -
visited
      visited = [0] * numCourses

      def dfs(course):
          if visited[course] == 1:
              return False  # Cycle detected
          if visited[course] == 2:
              return True   # Already visited and checked

          # Mark the current node as visiting
          visited[course] = 1

          # Visit all neighbors (prerequisites)
          for neighbor in graph[course]:
              if not dfs(neighbor):
                  return False

          # Mark the current node as visited
```

```
        visited[course] = 2
        return True

    # Check for cycle in each course
    for course in range(numCourses):
        if not dfs(course):
            return False

    return True

# Example usage:
numCourses = 2
prerequisites = [[1,0]]
print(canFinish(numCourses, prerequisites))  # Output: True (Can
finish all courses)
```

Pacific Atlantic Water Flow

```
def pacificAtlantic(matrix):
    if not matrix or not matrix[0]:
        return []

    m, n = len(matrix), len(matrix[0])
    pacific_reachable = [[False] * n for _ in range(m)]
    atlantic_reachable = [[False] * n for _ in range(m)]

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down,
Left, Right

    def dfs(r, c, reachable):
        reachable[r][c] = True
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < m and 0 <= nc < n and not reachable[nr][nc]
and matrix[nr][nc] >= matrix[r][c]:
                dfs(nr, nc, reachable)

    # Pacific side: Top and Left edges
    for i in range(m):
        dfs(i, 0, pacific_reachable)
    for j in range(n):
        dfs(0, j, pacific_reachable)

    # Atlantic side: Bottom and Right edges
    for i in range(m):
```

```python
        dfs(i, n-1, atlantic_reachable)
    for j in range(n):
        dfs(m-1, j, atlantic_reachable)

    # Collect the result
    result = []
    for i in range(m):
        for j in range(n):
            if pacific_reachable[i][j] and atlantic_reachable[i][j]:
                result.append([i, j])

    return result

# Example usage:
matrix = [
    [1, 2, 2, 3, 5],
    [3, 2, 3, 4, 4],
    [2, 4, 5, 3, 1],
    [6, 7, 1, 4, 5],
    [5, 1, 1, 2, 4]
]
print(pacificAtlantic(matrix))  # Output: [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]
```

Number of islands(Do in Grid and Graph Both)

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

    m, n = len(grid), len(grid[0])
    num_islands = 0

    def dfs(r, c):
        # Base case: out of bounds or water
        if r < 0 or r >= m or c < 0 or c >= n or grid[r][c] == '0':
            return
        # Mark the current cell as visited (change '1' to '0')
        grid[r][c] = '0'
        # Visit all 4 neighbors (up, down, left, right)
        dfs(r-1, c)
        dfs(r+1, c)
        dfs(r, c-1)
        dfs(r, c+1)
```

```python
      # Iterate through each cell in the grid
      for i in range(m):
          for j in range(n):
              if grid[i][j] == '1':
                  # Found a new island, perform DFS to mark all connected land cells as visited
                  num_islands += 1
                  dfs(i, j)

      return num_islands

  # Example usage:
  grid = [
      ['1', '1', '1', '1', '0'],
      ['1', '1', '0', '1', '0'],
      ['1', '1', '0', '0', '0'],
      ['0', '0', '0', '0', '0']
  ]
  print(numIslands(grid))  # Output: 1 (There is one island in the given grid)
```

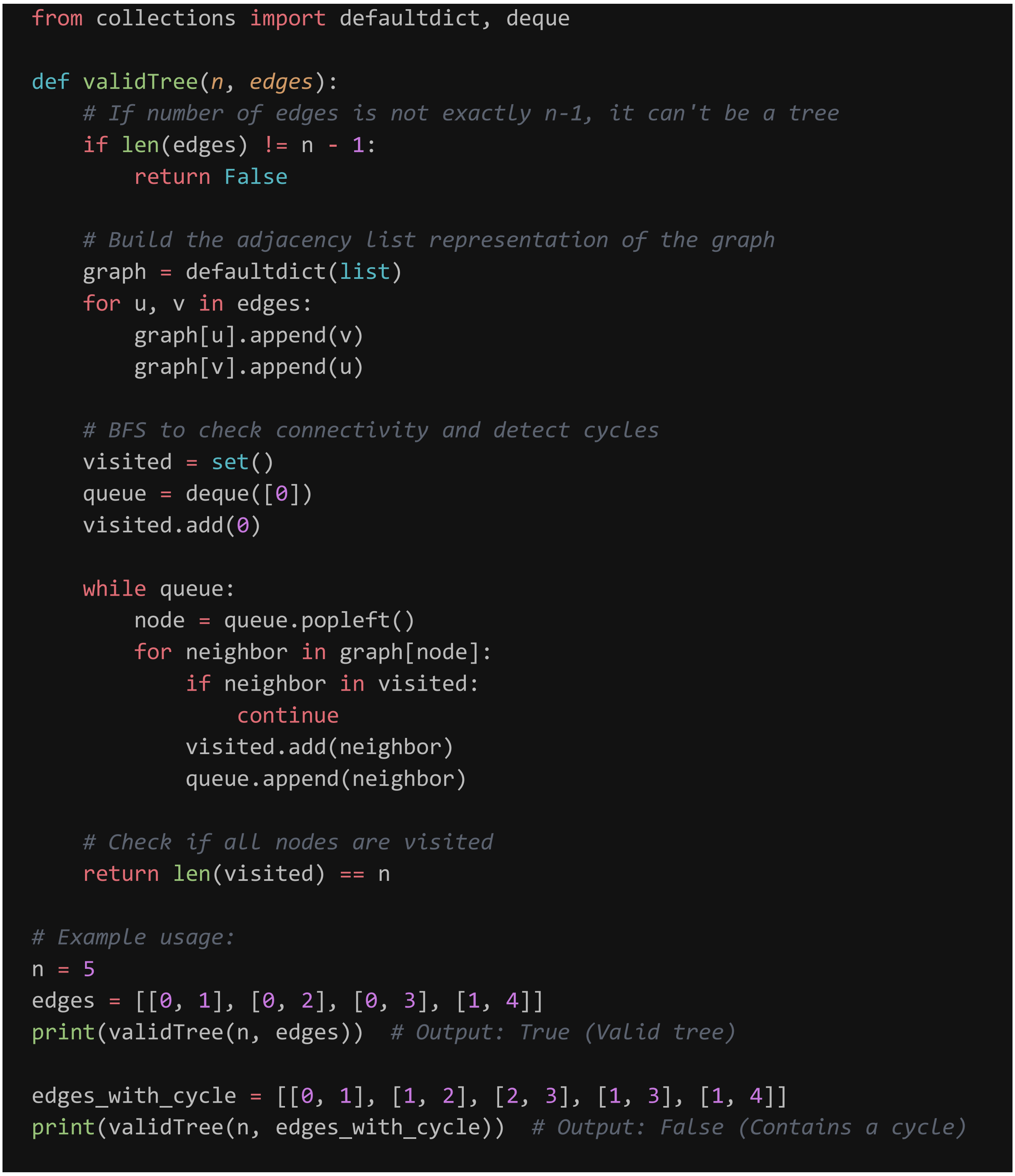
Longest Consecutive Sequence

```python
  def longestConsecutive(nums):
      num_set = set(nums)
      max_len = 0

      for num in num_set:
          if num - 1 not in num_set:  # Only start counting from the beginning of a sequence
              curr_num = num
              curr_len = 1

              while curr_num + 1 in num_set:
                  curr_num += 1
                  curr_len += 1

              max_len = max(max_len, curr_len)
      return max_len


  nums = [100, 4, 200, 1, 3, 2]
  print(longestConsecutive(nums))  # Output: 4 (Longest consecutive sequence is [1, 2, 3, 4])
```

Alien dictionary

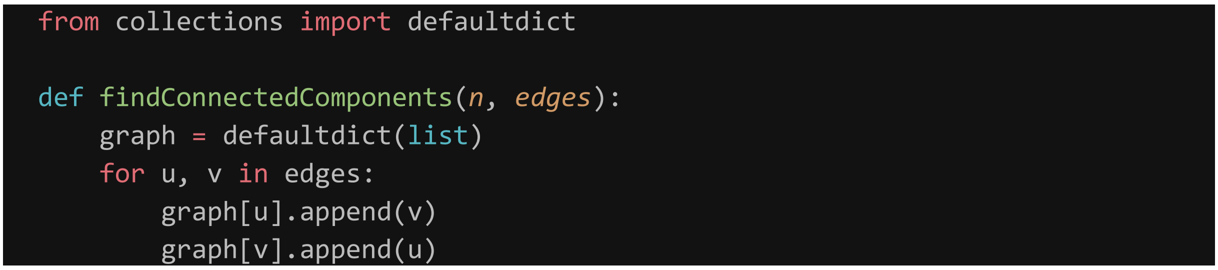
```
from collections import defaultdict, deque

def alienOrder(words):
    # Step 1: Build the graph
    graph = defaultdict(set)
    in_degree = {c: 0 for word in words for c in word}

    for i in range(1, len(words)):
        word1, word2 = words[i-1], words[i]
        for c1, c2 in zip(word1, word2):
            if c1 != c2:
                if c2 not in graph[c1]:
                    graph[c1].add(c2)
                    in_degree[c2] += 1
                break
        else:
            if len(word1) > len(word2):
                return ""  # Invalid order, words are not sorted correctly

    # Step 2: Topological sorting using Kahn's algorithm
    result = []
    zero_in_degree_queue = deque([c for c in in_degree if in_degree[c] ==
0])

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

    # Step 3: Check for cycle
    if len(result) != len(in_degree):
        return ""  # Cycle detected

    return "".join(result)

words = [
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
print(alienOrder(words))  # Output: "wertf"
```

Graph Valid Tree

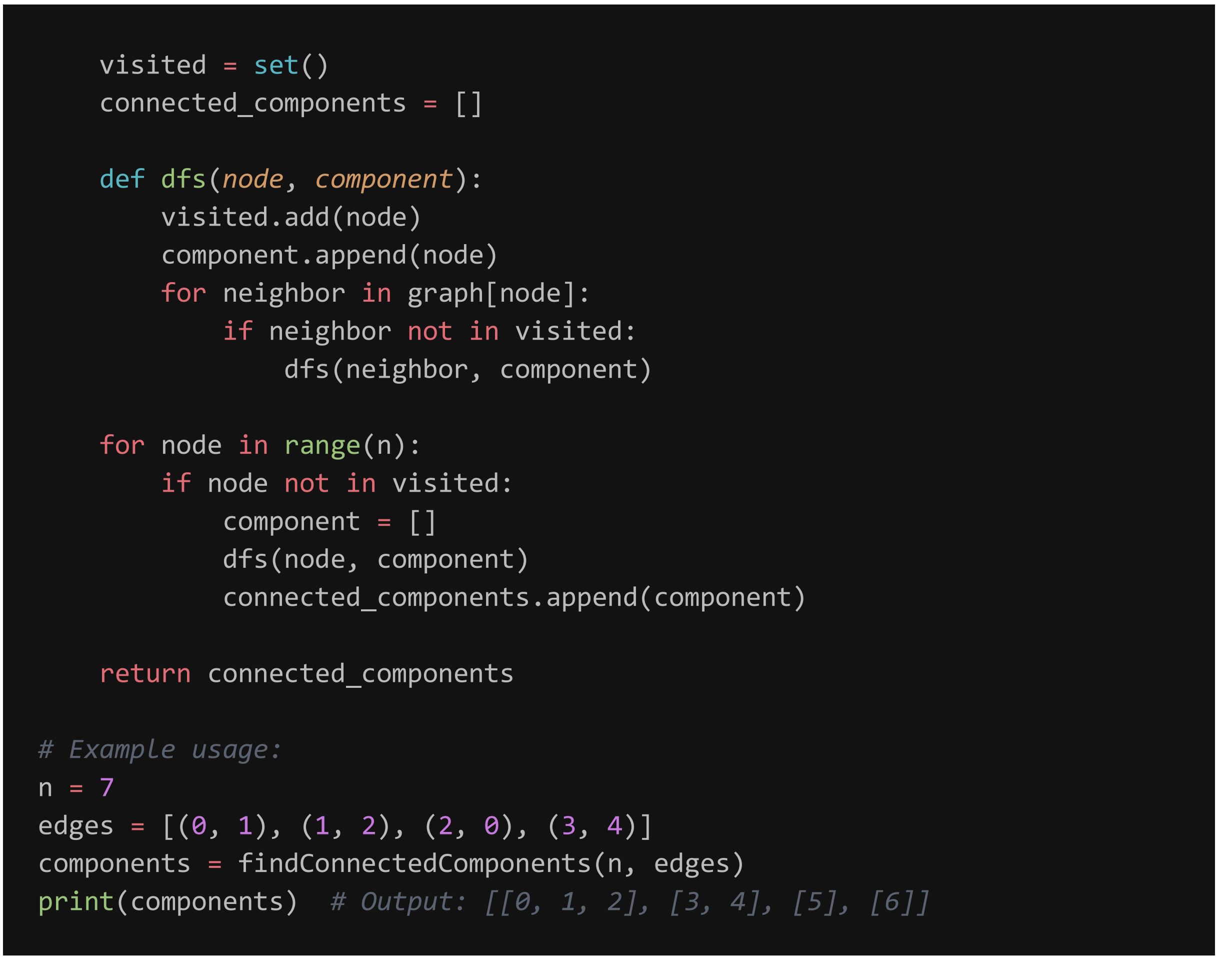


```python
from collections import defaultdict, deque

def validTree(n, edges):
    # If number of edges is not exactly n-1, it can't be a tree
    if len(edges) != n - 1:
        return False

    # Build the adjacency list representation of the graph
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    # BFS to check connectivity and detect cycles
    visited = set()
    queue = deque([0])
    visited.add(0)

    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor in visited:
                continue
            visited.add(neighbor)
            queue.append(neighbor)

    # Check if all nodes are visited
    return len(visited) == n

# Example usage:
n = 5
edges = [[0, 1], [0, 2], [0, 3], [1, 4]]
print(validTree(n, edges))  # Output: True (Valid tree)

edges_with_cycle = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]
print(validTree(n, edges_with_cycle))  # Output: False (Contains a cycle)
```

Connected Components | Logic Explanation

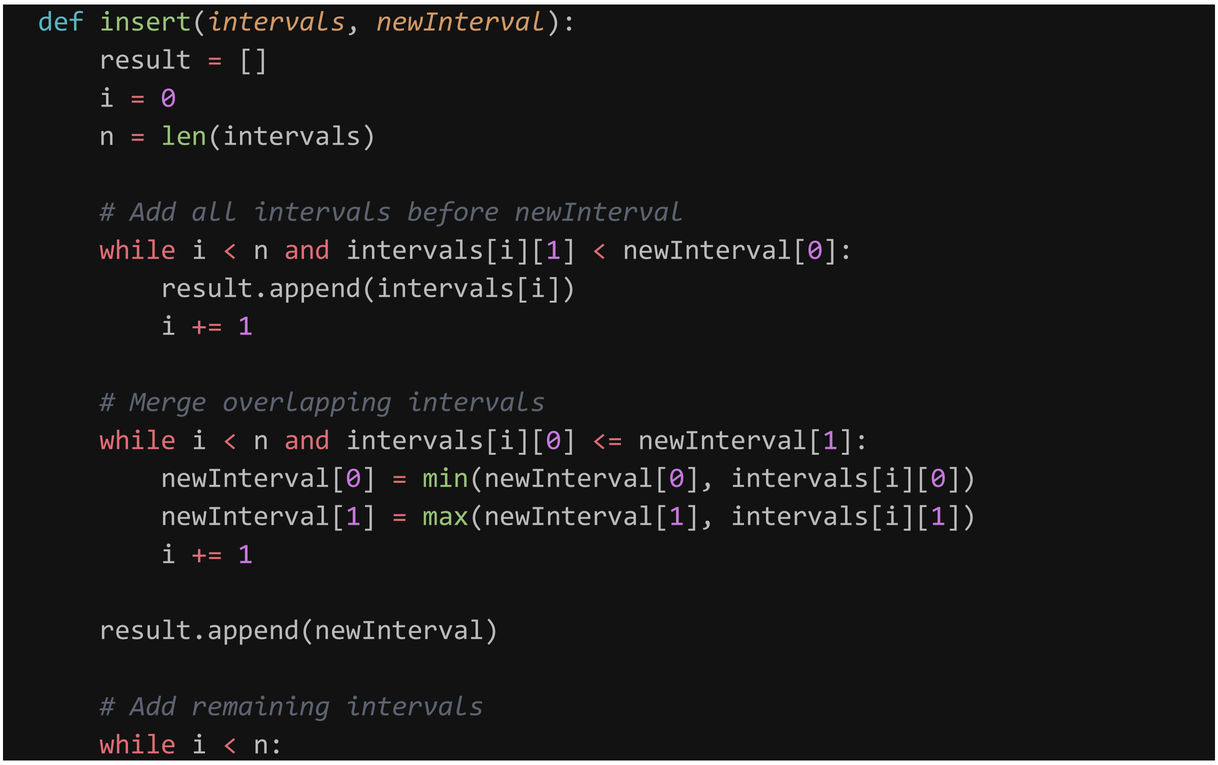


```python
from collections import defaultdict

def findConnectedComponents(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
```



```python
    visited = set()
    connected_components = []

    def dfs(node, component):
        visited.add(node)
        component.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor, component)

    for node in range(n):
        if node not in visited:
            component = []
            dfs(node, component)
            connected_components.append(component)

    return connected_components

# Example usage:
n = 7
edges = [(0, 1), (1, 2), (2, 0), (3, 4)]
components = findConnectedComponents(n, edges)
print(components)  # Output: [[0, 1, 2], [3, 4], [5], [6]]
```

Insert Interval



```python
def insert(intervals, newInterval):
    result = []
    i = 0
    n = len(intervals)

    # Add all intervals before newInterval
    while i < n and intervals[i][1] < newInterval[0]:
        result.append(intervals[i])
        i += 1

    # Merge overlapping intervals
    while i < n and intervals[i][0] <= newInterval[1]:
        newInterval[0] = min(newInterval[0], intervals[i][0])
        newInterval[1] = max(newInterval[1], intervals[i][1])
        i += 1

    result.append(newInterval)

    # Add remaining intervals
    while i < n:
        result.append(intervals[i])
        i += 1

    return result

# Example usage:
intervals = [[1,3],[6,9]]
newInterval = [2,5]
print(insert(intervals, newInterval))  # Output: [[1,5],[6,9]]
```

Merge Intervals

```
def merge_intervals(intervals):
    # Sort the intervals based on the start value
    intervals.sort(key=lambda x: x[0])

    merged = []
    for interval in intervals:
        # If the list of merged intervals is empty or if the current
        # interval does not overlap with the previous, simply append it.
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            # There is overlap, so we merge the current and previous
intervals.
            merged[-1][1] = max(merged[-1][1], interval[1])

    return merged

# Example usage:
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
print("Merged intervals:", merge_intervals(intervals))
```

Non-overlapping Intervals

```
def erase_overlap_intervals(intervals):
    if not intervals:
        return 0

    # Sort intervals by their end time
    intervals.sort(key=lambda x: x[1])

    end = intervals[0][1]
    count = 0

    for i in range(1, len(intervals)):
          if intervals[i][0] < end:
              # If current interval overlaps with the previous, increment
count
              count += 1
          else:
              # No overlap, update the end to the current interval's end
              end = intervals[i][1]

      return count

  # Example usage:
  intervals = [[1, 2], [2, 3], [3, 4], [1, 3]]
  print("Minimum number of intervals to remove to make non-overlapping:",
erase_overlap_intervals(intervals))
```

Repeat and Missing Number

```
  def find_missing_and_duplicate(arr):
      n = len(arr)
      sum_n = n * (n + 1) // 2
      sum_n_sq = n * (n + 1) * (2 * n + 1) // 6

      sum_arr = sum(arr)
      sum_arr_sq = sum(x * x for x in arr)

      # Let x be the missing number and y be the duplicate number
      # We have:
      # x - y = sum_n - sum_arr  (1)
      # x^2 - y^2 = sum_n_sq - sum_arr_sq  (2)
      # (2) can be written as (x - y)(x + y) = sum_n_sq - sum_arr_sq

      diff = sum_n - sum_arr  # x - y
      diff_sq = sum_n_sq - sum_arr_sq  # x^2 - y^2

      sum_xy = diff_sq // diff  # x + y

      x = (diff + sum_xy) // 2
      y = sum_xy - x

      return x, y

  # Example usage:
  arr = [4, 3, 6, 2, 1, 1]
  missing, duplicate = find_missing_and_duplicate(arr)
  print("Missing number:", missing)
  print("Duplicate number:", duplicate)
```

Meeting Rooms (Leetcode Premium).

```
def can_attend_meetings(intervals):
    # Sort the intervals based on the start time
    intervals.sort(key=lambda x: x[0])

    for i in range(1, len(intervals)):
        # If the current meeting starts before the previous meeting ends, return False
        if intervals[i][0] < intervals[i-1][1]:
            return False
    return True

# Example usage:
intervals = [(0, 30), (5, 10), (15, 20)]
print("Can attend all meetings:", can_attend_meetings(intervals))  # Output: False

intervals = [(7, 10), (2, 4)]
print("Can attend all meetings:", can_attend_meetings(intervals))  # Output: True
```

Meeting Rooms II (Leetcode Premium)

```
import heapq

def min_meeting_rooms(intervals):
    if not intervals:
        return 0

    # Sort the intervals by start time
    intervals.sort(key=lambda x: x[0])

    # Initialize a heap to keep track of the end times of meetings
    end_times = []

    # Add the end time of the first meeting
    heapq.heappush(end_times, intervals[0][1])

    for i in range(1, len(intervals)):
        # If the room due to free up the earliest is free, reuse that room
        if intervals[i][0] >= end_times[0]:
            heapq.heappop(end_times)

        # Add the end time of the current meeting
        heapq.heappush(end_times, intervals[i][1])

    # The size of the heap is the number of rooms required
```

```
      return len(end_times)

  # Example usage:
  intervals = [(0, 30), (5, 10), (15, 20)]
  print("Minimum number of meeting rooms required:",
min_meeting_rooms(intervals))  # Output: 2

  intervals = [(7, 10), (2, 4)]
  print("Minimum number of meeting rooms required:",
min_meeting_rooms(intervals))  # Output: 1
```

Reverse a LinkedList

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

  def reverse_linked_list(head):
      prev = None
      current = head

      while current is not None:
          next_node = current.next
          current.next = prev
          prev = current
          current = next_node

      return prev

  def print_list(head):
      current = head
      while current:
          print(current.value, end=" -> ")
          current = current.next
      print("None")

  # Creating a linked list: 1 -> 2 -> 3 -> 4 -> None
  node1 = ListNode(1)
  node2 = ListNode(2)
  node3 = ListNode(3)
  node4 = ListNode(4)
  node1.next = node2
  node2.next = node3
  node3.next = node4

  print("Original linked list:")
```

```
print_list(node1)

# Reversing the linked list
reversed_head = reverse_linked
```

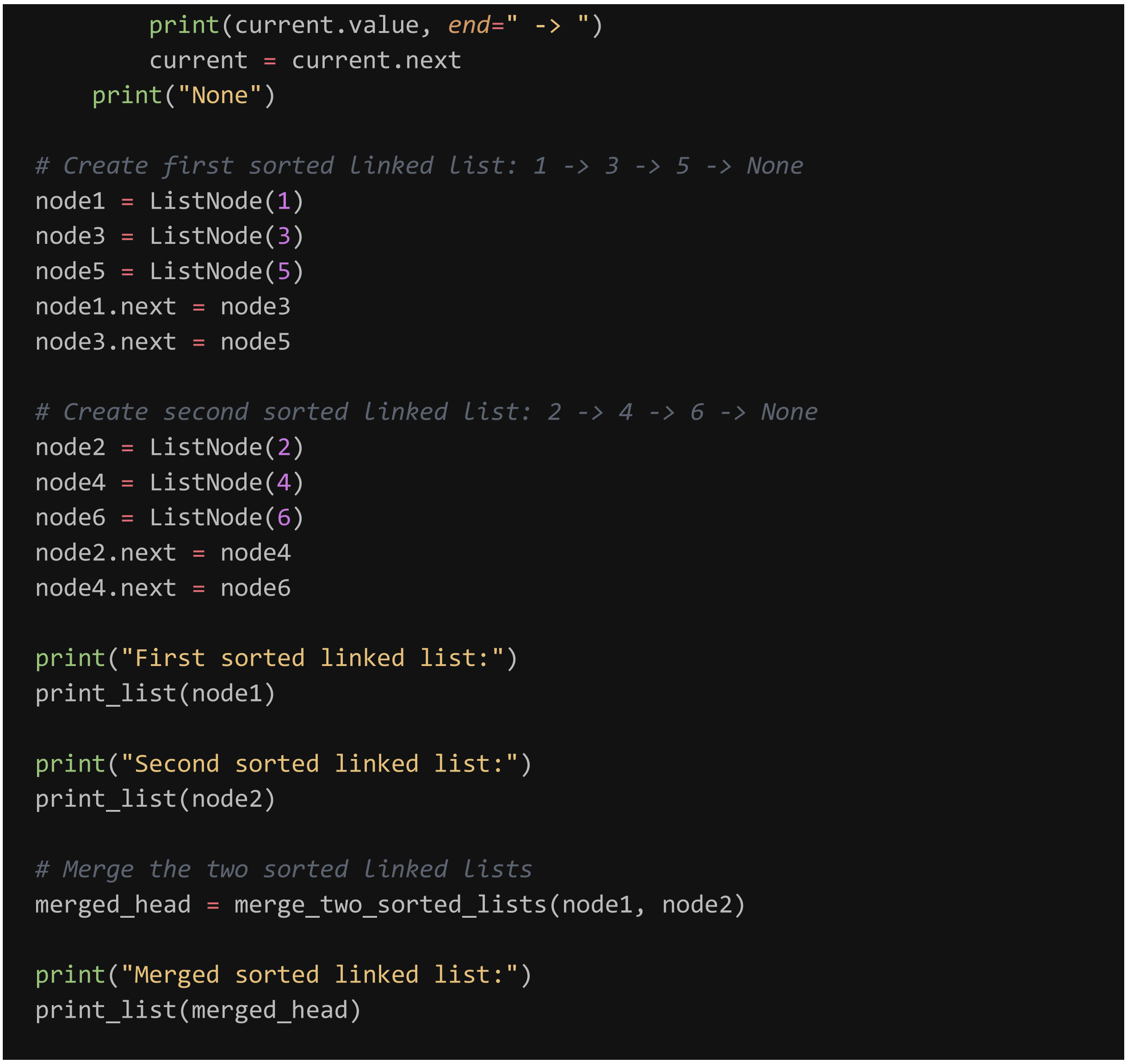
Detect a cycle in Linked List

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

def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next

    return True

# Helper function to print the list (for debugging purposes)
def print_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Create nodes
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)

# Link nodes to form a cycle: 1 -> 2 -> 3 -> 4 -> 2 (cycle)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node2
```

```
# Test cycle detection
print("Does the linked list have a cycle?")
print(has_cycle(node1))  # Output: True

# Create another list without a cycle: 1 -> 2 -> 3 -> 4 -> None
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node1.next = node2
node2.next = node3
node3.next = node4

# Test cycle detection
print("Does the linked list have a cycle?")
print(has_cycle(node1))  # Output: False
```

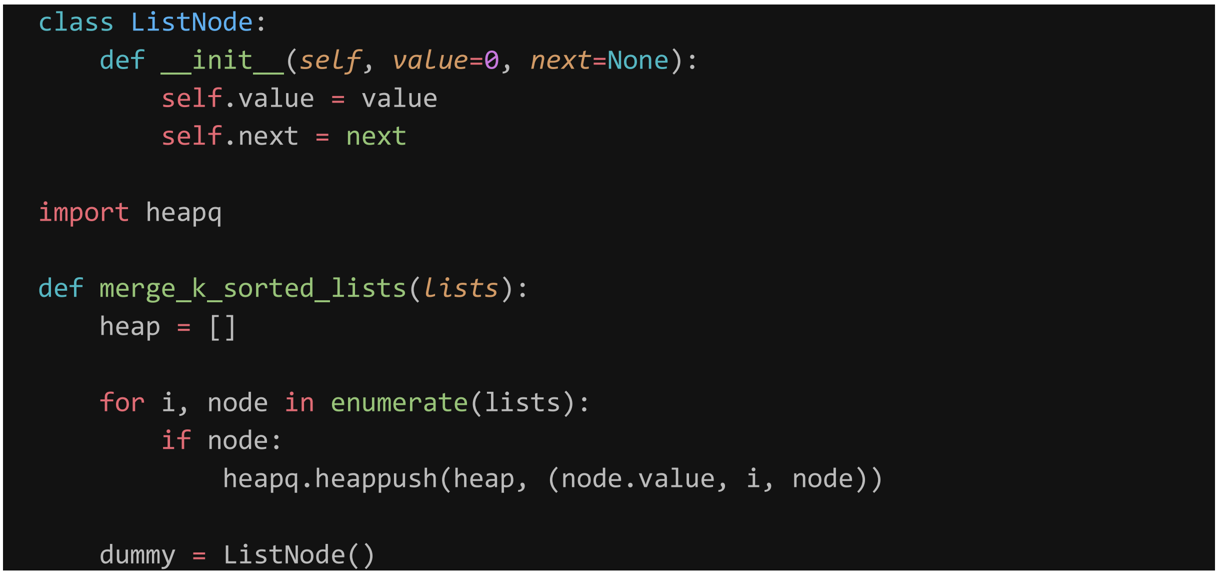
Merge two sorted Linked List (use method used in mergeSort)

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

def merge_two_sorted_lists(l1, l2):
    dummy = ListNode()
    current = dummy

    while l1 and l2:
        if l1.value < l2.value:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    if l1:
        current.next = l1
    else:
        current.next = l2

    return dummy.next

def print_list(head):
    current = head
    while current:
```



```python
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Create first sorted linked list: 1 -> 3 -> 5 -> None
node1 = ListNode(1)
node3 = ListNode(3)
node5 = ListNode(5)
node1.next = node3
node3.next = node5

# Create second sorted linked list: 2 -> 4 -> 6 -> None
node2 = ListNode(2)
node4 = ListNode(4)
node6 = ListNode(6)
node2.next = node4
node4.next = node6

print("First sorted linked list:")
print_list(node1)

print("Second sorted linked list:")
print_list(node2)

# Merge the two sorted linked lists
merged_head = merge_two_sorted_lists(node1, node2)

print("Merged sorted linked list:")
print_list(merged_head)
```

Merge K sorted arrays



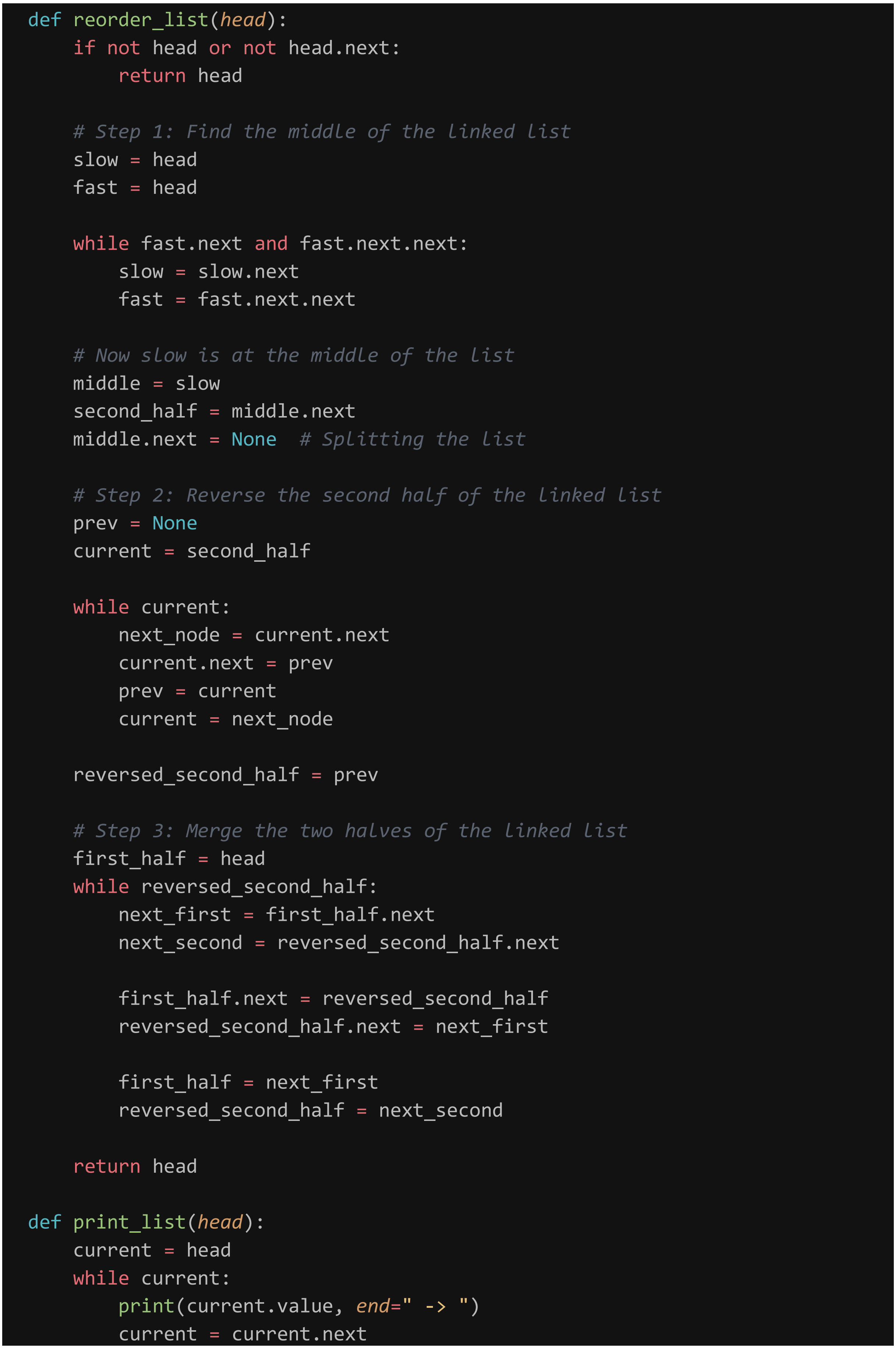
```python
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

import heapq

def merge_k_sorted_lists(lists):
    heap = []

    for i, node in enumerate(lists):
        if node:
            heapq.heappush(heap, (node.value, i, node))

    dummy = ListNode()
    current = dummy

    while heap:
        value, i, node = heapq.heappop(heap)
        current.next = ListNode(value)
        current = current.next
        if node.next:
            heapq.heappush(heap, (node.next.value, i, node.next))

    return dummy.next

def print_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Create multiple sorted linked lists
node1 = ListNode(1, ListNode(4, ListNode(5)))
node2 = ListNode(1, ListNode(3, ListNode(4)))
node3 = ListNode(2, ListNode(6))

lists = [node1, node2, node3]

print("Original sorted linked lists:")
for i, l in enumerate(lists):
    print(f"List {i+1}: ", end="")
    print_list(l)

# Merge the sorted linked lists
merged_head = merge_k_sorted_lists(lists)

print("Merged sorted linked list:")
print_list(merged_head)
```

Remove N-th node from back of LinkedList

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

def remove_nth_from_end(head, n):
    dummy = ListNode(0)
    dummy.next = head
    first = dummy
    second = dummy

    for _ in range(n + 1):
        first = first.next

    while first is not None:
        first = first.next
        second = second.next

    second.next = second.next.next

    return dummy.next

def print_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> None
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node5 = ListNode(5)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

print("Original linked list:")
print_list(node1)

# Remove the 2nd node from the back (the node with value 4)
updated_head = remove_nth_from_end(node1, 2)

print("Updated linked list:")
print_list(updated_head)
```

Reorder List

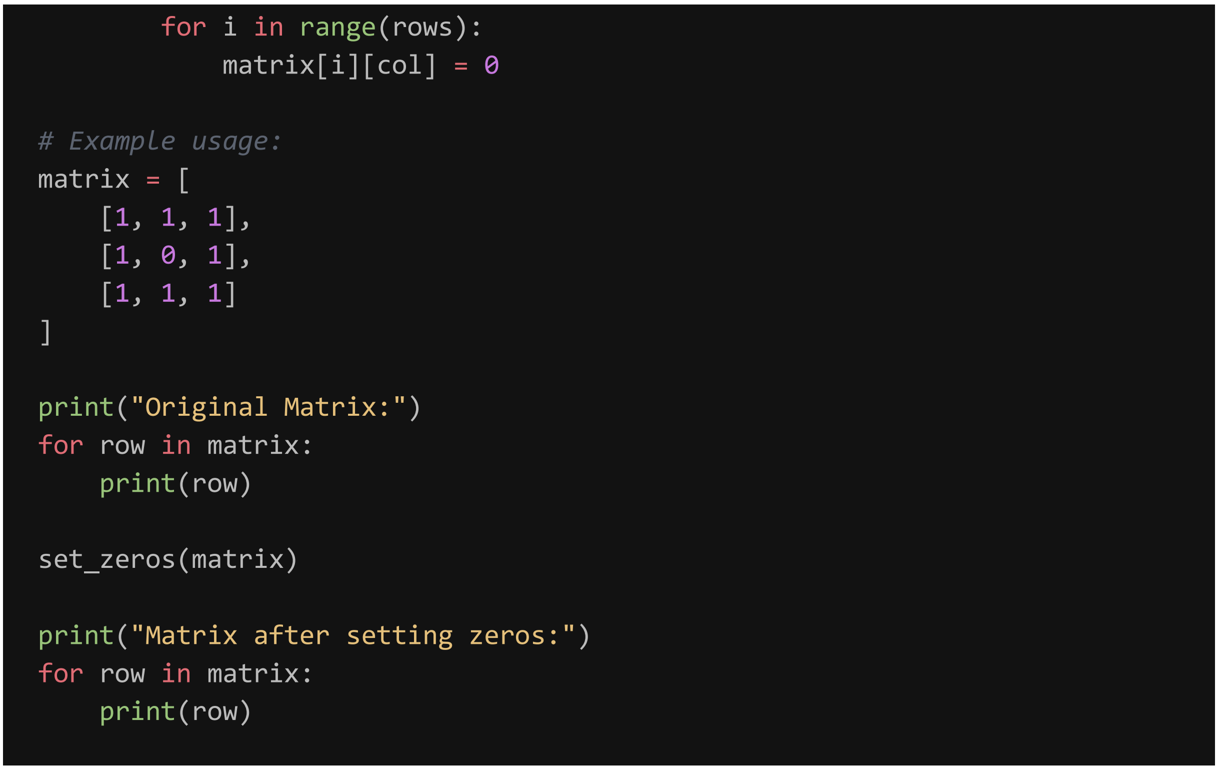
```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next
```



```python
def reorder_list(head):
    if not head or not head.next:
        return head

    # Step 1: Find the middle of the linked list
    slow = head
    fast = head

    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next

    # Now slow is at the middle of the list
    middle = slow
    second_half = middle.next
    middle.next = None  # Splitting the list

    # Step 2: Reverse the second half of the linked list
    prev = None
    current = second_half

    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    reversed_second_half = prev

    # Step 3: Merge the two halves of the linked list
    first_half = head
    while reversed_second_half:
        next_first = first_half.next
        next_second = reversed_second_half.next

        first_half.next = reversed_second_half
        reversed_second_half.next = next_first

        first_half = next_first
        reversed_second_half = next_second

    return head

def print_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

# Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> None
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node5 = ListNode(5)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

print("Original linked list:")
print_list(node1)

# Reorder the linked list
reorder_list(node1)

print("Reordered linked list:")
print_list(node1)
```

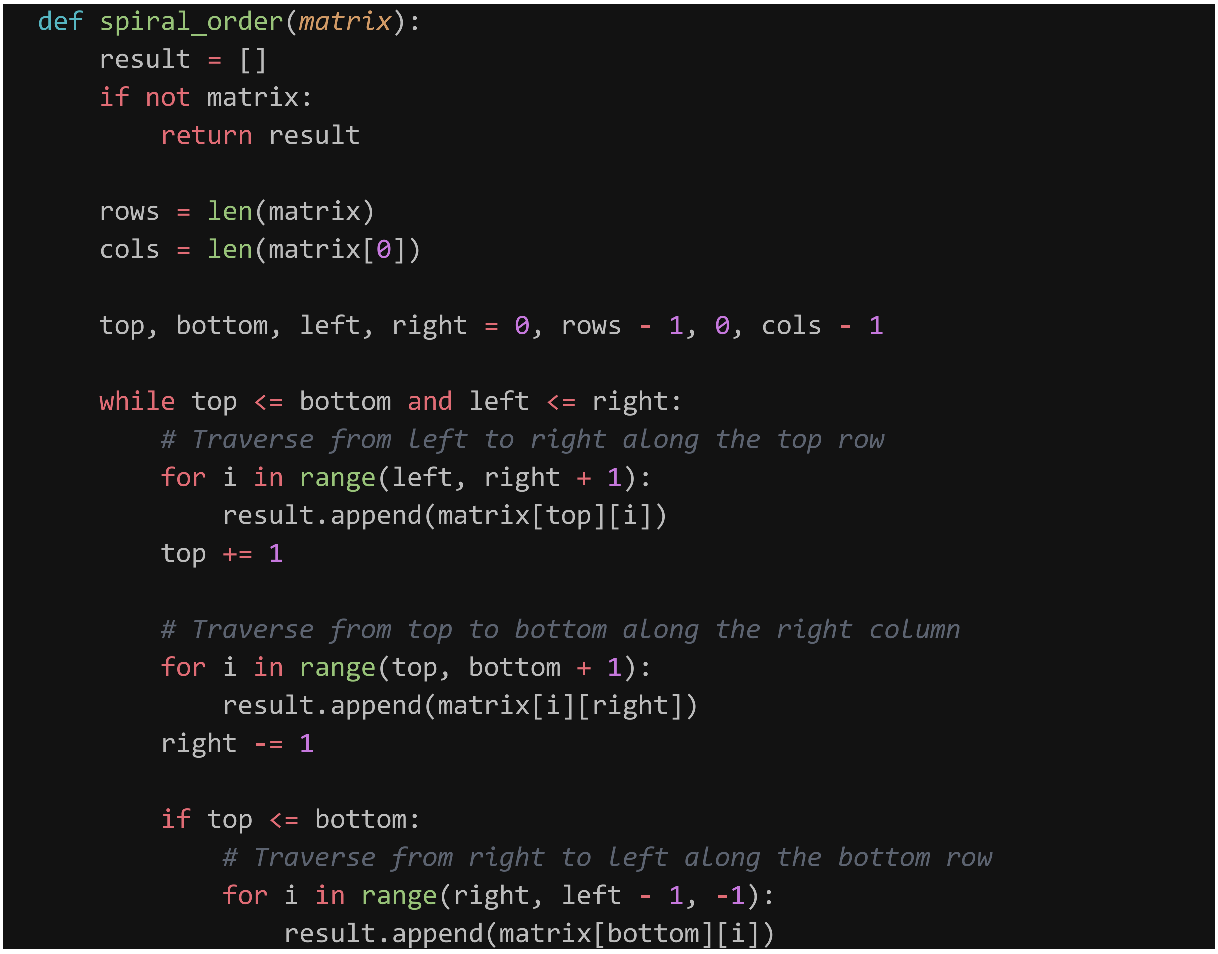
Set Matrix Zeros

```
def set_zeros(matrix):
    if not matrix:
        return

    rows = len(matrix)
    cols = len(matrix[0])

    zero_rows = set()
    zero_cols = set()

    # Step 1: Identify rows and columns containing zeros
    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 0:
                zero_rows.add(i)
                zero_cols.add(j)

    # Step 2: Set zeros based on zero_rows and zero_cols
    for row in zero_rows:
        for j in range(cols):
            matrix[row][j] = 0

    for col in zero_cols:
```



```python
        for i in range(rows):
            matrix[i][col] = 0

# Example usage:
matrix = [
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
]

print("Original Matrix:")
for row in matrix:
    print(row)

set_zeros(matrix)

print("Matrix after setting zeros:")
for row in matrix:
    print(row)
```

Print the matrix in spiral manner



```python
def spiral_order(matrix):
    result = []
    if not matrix:
        return result

    rows = len(matrix)
    cols = len(matrix[0])

    top, bottom, left, right = 0, rows - 1, 0, cols - 1

    while top <= bottom and left <= right:
        # Traverse from left to right along the top row
        for i in range(left, right + 1):
            result.append(matrix[top][i])
        top += 1

        # Traverse from top to bottom along the right column
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1

        if top <= bottom:
            # Traverse from right to left along the bottom row
            for i in range(right, left - 1, -1):
                result.append(matrix[bottom][i])
            bottom -= 1

        if left <= right:
            # Traverse from bottom to top along the left column
            for i in range(bottom, top - 1, -1):
                result.append(matrix[i][left])
            left += 1

    return result

# Example usage:
matrix = [
    [ 1, 2, 3, 4 ],
    [ 5, 6, 7, 8 ],
    [ 9, 10, 11, 12 ],
    [13, 14, 15, 16 ]
]

print("Matrix in spiral order:")
print(spiral_order(matrix))
```

Rotate Matrix by 90 degrees

```
def rotate(matrix):
    n = len(matrix)

    # Step 1: Transpose the matrix
    for i in range(n):
        for j in range(i + 1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # Step 2: Reverse each row
    for i in range(n):
        matrix[i] = matrix[i][::-1]

    return matrix

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print("Original Matrix:")
for row in matrix:
    print(row)
```

```
rotated_matrix = rotate(matrix)

print("\nMatrix after 90 degrees clockwise rotation:")
for row in rotated_matrix:
    print(row)
```

Word Search

```
def exist(board, word):
    if not board:
        return False

    rows = len(board)
    cols = len(board[0])

    def dfs(i, j, index):
        if index == len(word):
            return True

        if i < 0 or i >= rows or j < 0 or j >= cols or board[i][j] !=
word[index]:
            return False

        # Temporarily mark the cell as visited
        temp = board[i][j]
        board[i][j] = '#'

        # Explore neighbors in all four directions
        found = (dfs(i+1, j, index+1) or
                 dfs(i-1, j, index+1) or
                 dfs(i, j+1, index+1) or
                 dfs(i, j-1, index+1))

        # Restore the cell
        board[i][j] = temp

        return found

    # Start DFS from each cell to find the word
    for i in range(rows):
        for j in range(cols):
            if dfs(i, j, 0):
                return True

    return False
```

```
  # Example usage:
  board = [
      ['A','B','C','E'],
      ['S','F','C','S'],
      ['A','D','E','E']
  ]
  word = "ABCCED"
  print(f"Does the word '{word}' exist in the board? {exist(board, word)}")

  word = "SEE"
  print(f"Does the word '{word}' exist in the board? {exist(board, word)}")

  word = "ABCB"
  print(f"Does the word '{word}' exist in the board? {exist(board, word)}")
```

Longest Substring Without Repeating Characters

```
  def length_of_longest_substring(s):
      char_map = {}
      max_length = 0
      left = 0

      for right in range(len(s)):
          if s[right] in char_map and char_map[s[right]] >= left:
              left = char_map[s[right]] + 1

          char_map[s[right]] = right
          max_length = max(max_length, right - left + 1)

      return max_length

  # Example usage:
  input_string = "abcabcbb"
  print(f"Longest substring without repeating characters in '{input_string}':
{length_of_longest_substring(input_string)}")

  input_string = "bbbbb"
  print(f"Longest substring without repeating characters in '{input_string}':
{length_of_longest_substring(input_string)}")

  input_string = "pwwkew"
  print(f"Longest substring without repeating characters in '{input_string}':
{length_of_longest_substring(input_string)}")
```

Longest repeating character replacement

```python
def longest_repeating_character_replacement(s, k):
    if not s:
        return 0

    n = len(s)
    char_count = [0] * 26  # Assuming only uppercase English letters

    max_length = 0
    max_count = 0
    left = 0

    for right in range(n):
        char_index = ord(s[right]) - ord('A')
        char_count[char_index] += 1
        max_count = max(max_count, char_count[char_index])

        # If the window size - max_count > k, shrink the window from the left
        while (right - left + 1 - max_count) > k:
            char_count[ord(s[left]) - ord('A')] -= 1
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
input_string = "ABAB"
k = 2
print(f"Longest substring with repeating characters for '{input_string}' and k = {k}: {longest_repeating_character_replacement(input_string, k)}")

input_string = "AABABBA"
k = 1
print(f"Longest substring with repeating characters for '{input_string}' and k = {k}: {longest_repeating_character_replacement(input_string, k)}")
```

Minimum Window Substring

```python
import collections

def min_window(s, pattern):
    if not s or not pattern:
        return ""

    pattern_map = collections.Counter(pattern)
    window_map = {}
    min_length = float('inf')
    min_window = ""
    left, count = 0, 0

    for right in range(len(s)):
        char_right = s[right]

        # Add character to the window_map
        if char_right in window_map:
            window_map[char_right] += 1
        else:
            window_map[char_right] = 1

        # Check if the current window contains all characters of the pattern
        if char_right in pattern_map and window_map[char_right] <= pattern_map[char_right]:
            count += 1

        # Try to minimize the window size by moving the left pointer
        while count == len(pattern):
            char_left = s[left]

            # Update the minimum window if a smaller valid window is found
            if right - left + 1 < min_length:
                min_length = right - left + 1
                min_window = s[left:right + 1]

            # Move the left pointer to shrink the window
            window_map[char_left] -= 1
            if char_left in pattern_map and window_map[char_left] < pattern_map[char_left]:
                count -= 1
            left += 1

    return min_window

# Example usage:
s = "ADOBECODEBANC"
pattern = "ABC"
print("Minimum window substring:", min_window(s, pattern))
```

Check for Anagrams

```python
import collections

def check_anagrams(s1, s2):
    # If lengths are different, they cannot be anagrams
    if len(s1) != len(s2):
        return False

    # Use a hashmap to count character frequencies
    count1 = collections.Counter(s1)
    count2 = collections.Counter(s2)

    # Compare the two frequency maps
    return count1 == count2

# Example usage:
s1 = "listen"
s2 = "silent"
print(f"Are '{s1}' and '{s2}' anagrams? {check_anagrams(s1, s2)}")

s1 = "triangle"
s2 = "integral"
print(f"Are '{s1}' and '{s2}' anagrams? {check_anagrams(s1, s2)}")

s1 = "hello"
s2 = "world"
print(f"Are '{s1}' and '{s2}' anagrams? {check_anagrams(s1, s2)}")
```

Group Anagrams

```
def group_anagrams(words):
    anagram_map = {}

    for word in words:
        sorted_word = tuple(sorted(word))  # Convert sorted characters to
tuple
        if sorted_word in anagram_map:
            anagram_map[sorted_word].append(word)
        else:
            anagram_map[sorted_word] = [word]

    return list(anagram_map.values())

# Example usage:
words = ["eat", "tea", "tan", "ate", "nat", "bat"]
print("Grouped Anagrams:")
print(group_anagrams(words))

words = ["listen", "silent", "triangle", "integral", "hello", "world"]
print("Grouped Anagrams:")
print(group_anagrams(words))
```

Check for balanced paranthesis

```
  def is_balanced_parentheses(s):
      stack = []
      mapping = {')': '('}  # Mapping for closing to opening parentheses

      for char in s:
          if char in mapping:
              top_element = stack.pop() if stack else '#'
              if mapping[char] != top_element:
                  return False
          else:
              stack.append(char)

      return not stack

  # Example usage:
  test_string = "()"
  print(f"Is '{test_string}' balanced?
{is_balanced_parentheses(test_string)}")

  test_string = "()[]{}"
  print(f"Is '{test_string}' balanced?
{is_balanced_parentheses(test_string)}")

  test_string = "(]"
  print(f"Is '{test_string}' balanced?
{is_balanced_parentheses(test_string)}")

  test_string = "([)]"
  print(f"Is '{test_string}' balanced?
{is_balanced_parentheses(test_string)}")

  test_string = "{[]}"
  print(f"Is '{test_string}' balanced?
{is_balanced_parentheses(test_string)}")
```

Check Palindrome

```
  def is_palindrome(s):
      # Normalize string: convert to lowercase and remove non-alphanumeric
characters
      s = ''.join(char.lower() for char in s if char.isalnum())

      # Two-pointer approach to check palindrome
      left, right = 0, len(s) - 1
      while left < right:
          if s[left] != s[right]:
              return False
          left += 1
          right -= 1

      return True

  # Example usage:
  test_string = "A man, a plan, a canal: Panama"
  print(f"Is '{test_string}' a palindrome? {is_palindrome(test_string)}")

  test_string = "race a car"
  print(f"Is '{test_string}' a palindrome? {is_palindrome(test_string)}")
```

Longest Palindromic Substring[Do it without DP]

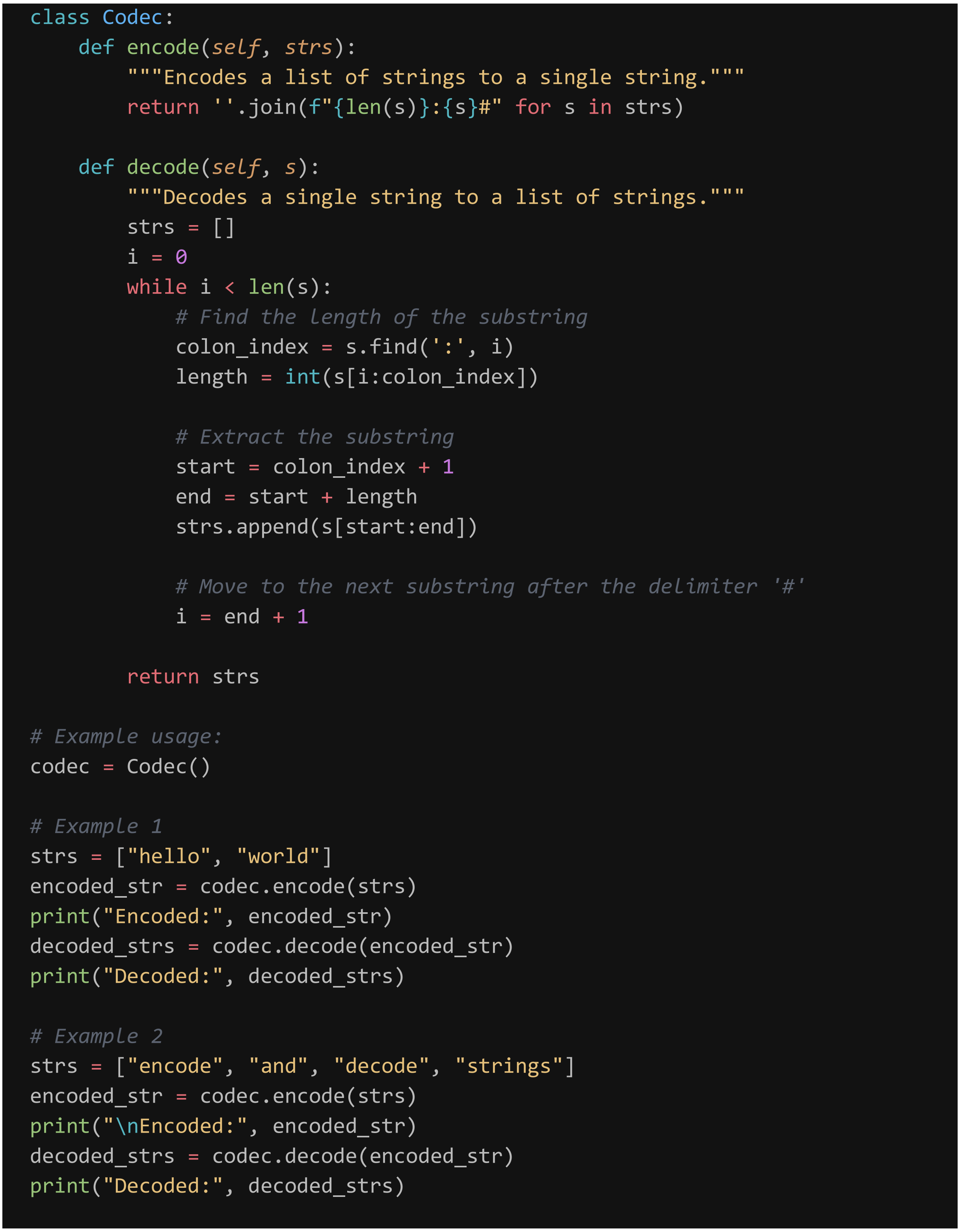
```
def longest_palindromic_substring(s):
    if not s:
        return ""

    start, end = 0, 0

    def expand_around_center(left, right):
        nonlocal start, end
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        # After exiting the loop, s[left+1:right] is the palindrome
substring
        if right - left - 1 > end - start:
            start = left + 1
            end = right

    for i in range(len(s)):
        # Check odd-length palindromes centered at s[i]
        expand_around_center(i, i)
        # Check even-length palindromes centered between s[i] and s[i+1]
        if i + 1 < len(s):
            expand_around_center(i, i + 1)

    return s[start:end]

# Example usage:
input_string = "babad"
print("Longest palindromic substring in '{}' is: '{}'".format(input_string,
longest_palindromic_substring(input_string)))

input_string = "cbbd"
```

```
  print("Longest palindromic substring in '{}' is: '{}'".format(input_string,
longest_palindromic_substring(input_string)))

  input_string = "racecar"
  print("Longest palindromic substring in '{}' is: '{}'".format(input_string,
longest_palindromic_substring(input_string)))
```

Palindromic Substrings

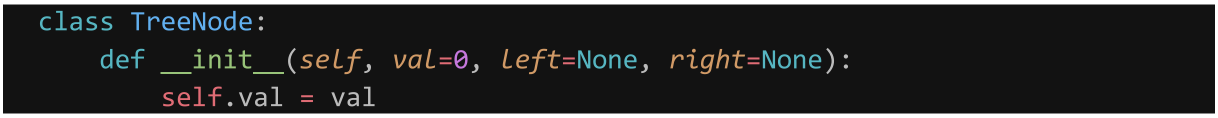
```
  def palindromic_substrings(s):
      if not s:
          return []

      result = []

      def expand_around_center(left, right):
          nonlocal result
          while left >= 0 and right < len(s) and s[left] == s[right]:
              result.append(s[left:right+1])
              left -= 1
              right += 1

      for i in range(len(s)):
          # Check odd-length palindromes centered at s[i]
          expand_around_center(i, i)
          # Check even-length palindromes centered between s[i] and s[i+1]
          if i + 1 < len(s):
              expand_around_center(i, i + 1)

      return result

  # Example usage:
  input_string = "babad"
  print("Palindromic substrings in '{}':".format(input_string))
  print(palindromic_substrings(input_string))

  input_string = "cbbd"
  print("Palindromic substrings in '{}':".format(input_string))
  print(palindromic_substrings(input_string))

  input_string = "racecar"
  print("Palindromic substrings in '{}':".format(input_string))
  print(palindromic_substrings(input_string))
```

Encode and Decode Strings (Leetcode Premium)

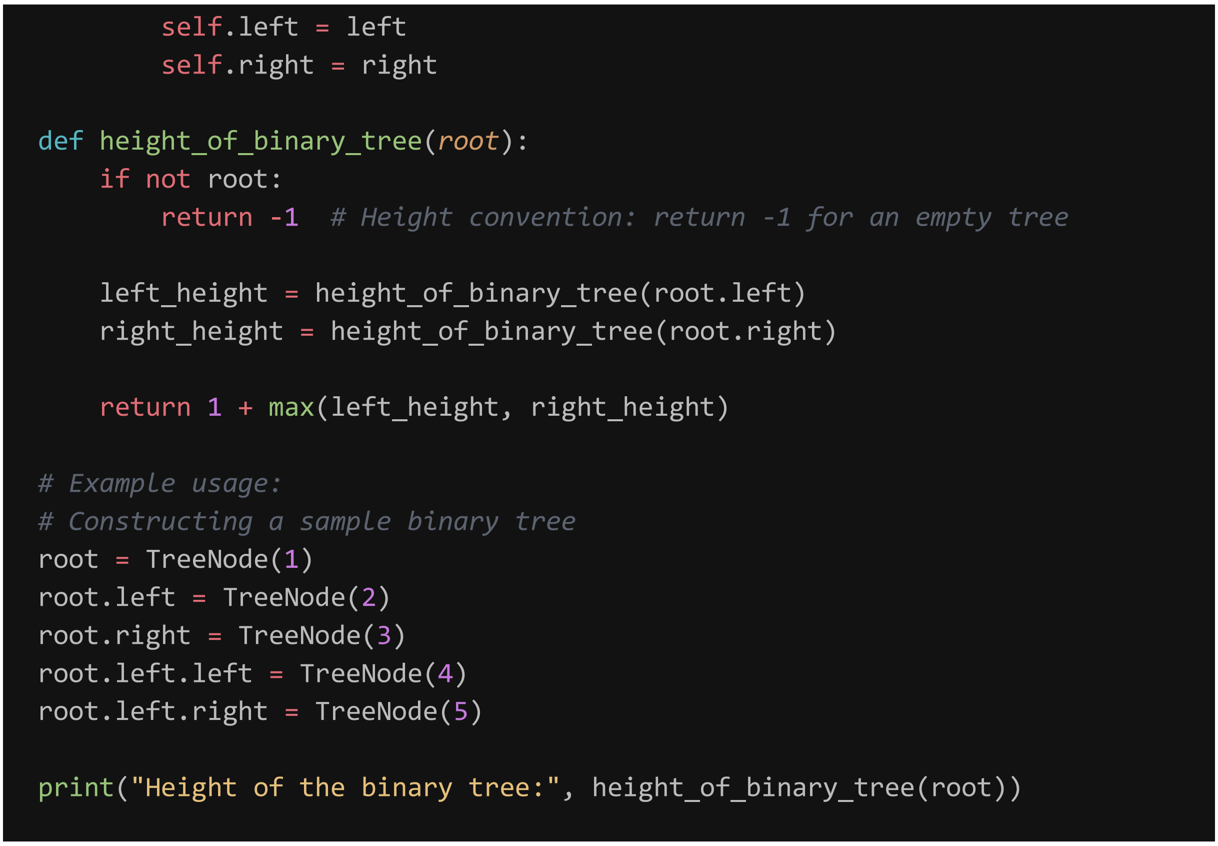


```
class Codec:
    def encode(self, strs):
        """Encodes a list of strings to a single string."""
        return ''.join(f"{len(s)}:{s}#" for s in strs)

    def decode(self, s):
        """Decodes a single string to a list of strings."""
        strs = []
        i = 0
        while i < len(s):
            # Find the length of the substring
            colon_index = s.find(':', i)
            length = int(s[i:colon_index])

            # Extract the substring
            start = colon_index + 1
            end = start + length
            strs.append(s[start:end])

            # Move to the next substring after the delimiter '#'
            i = end + 1

        return strs

# Example usage:
codec = Codec()

# Example 1
strs = ["hello", "world"]
encoded_str = codec.encode(strs)
print("Encoded:", encoded_str)
decoded_strs = codec.decode(encoded_str)
print("Decoded:", decoded_strs)

# Example 2
strs = ["encode", "and", "decode", "strings"]
encoded_str = codec.encode(strs)
print("\nEncoded:", encoded_str)
decoded_strs = codec.decode(encoded_str)
print("Decoded:", decoded_strs)
```

Height of a Binary Tree

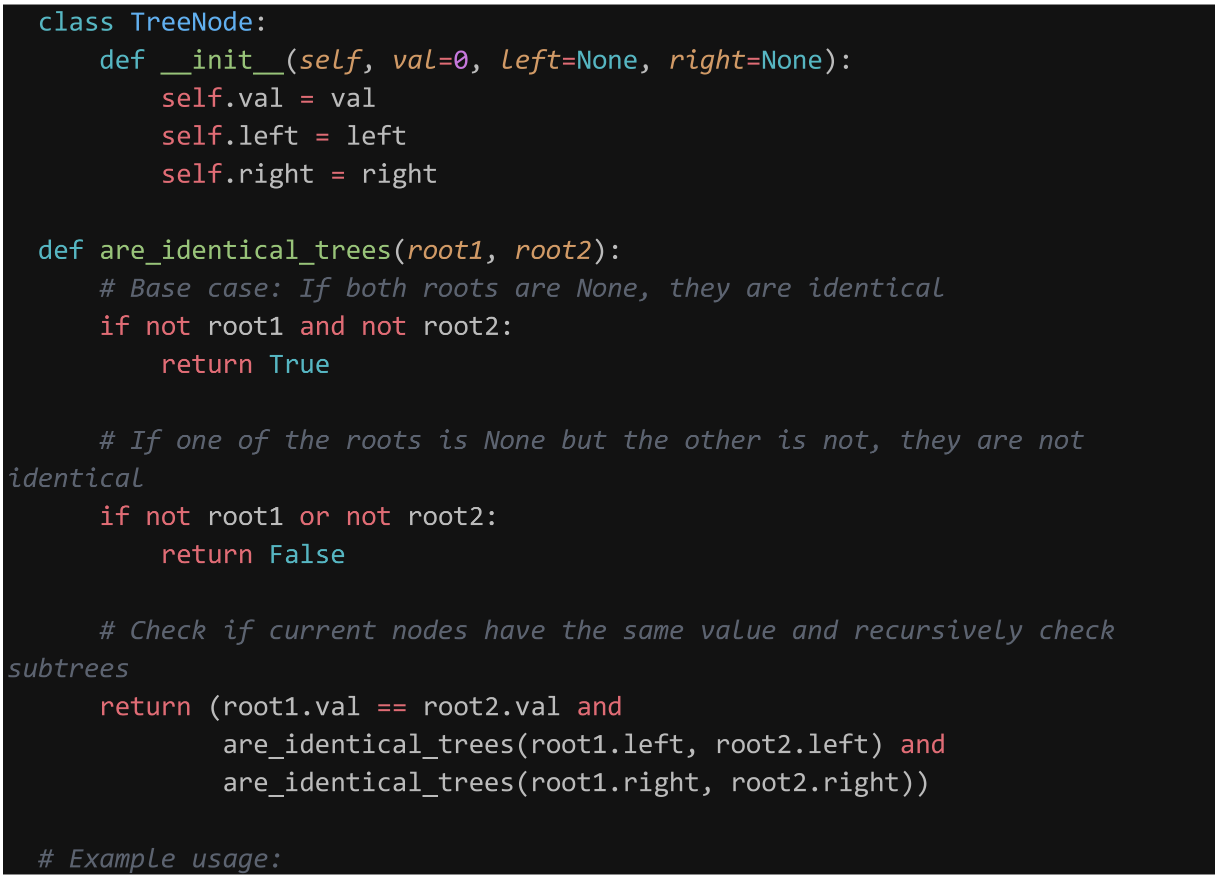


```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
```



```
        self.left = left
        self.right = right

def height_of_binary_tree(root):
    if not root:
        return -1  # Height convention: return -1 for an empty tree

    left_height = height_of_binary_tree(root.left)
    right_height = height_of_binary_tree(root.right)

    return 1 + max(left_height, right_height)

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print("Height of the binary tree:", height_of_binary_tree(root))
```

Check if two trees are identical or not



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

def are_identical_trees(root1, root2):
    # Base case: If both roots are None, they are identical
    if not root1 and not root2:
        return True

    # If one of the roots is None but the other is not, they are not
identical
    if not root1 or not root2:
        return False

    # Check if current nodes have the same value and recursively check
subtrees
    return (root1.val == root2.val and
            are_identical_trees(root1.left, root2.left) and
            are_identical_trees(root1.right, root2.right))

# Example usage:
```

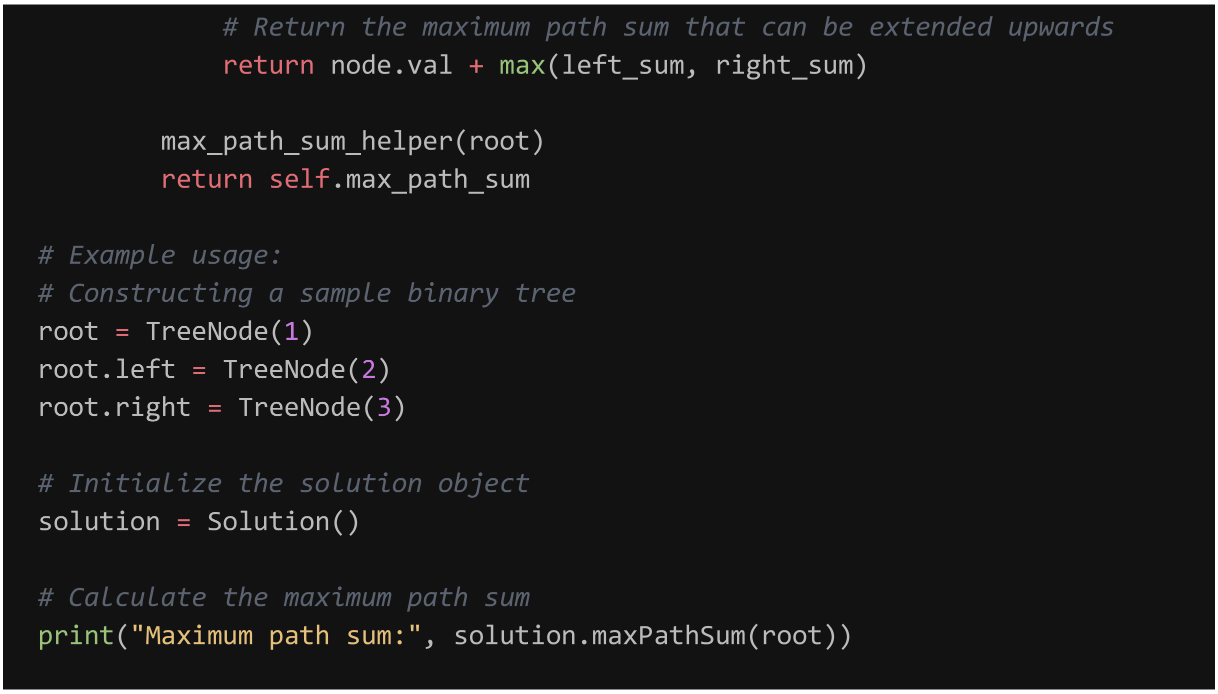
```
# Constructing two identical binary trees
root1 = TreeNode(1)
root1.left = TreeNode(2)
root1.right = TreeNode(3)
root1.left.left = TreeNode(4)
root1.left.right = TreeNode(5)

root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.right = TreeNode(3)
root2.left.left = TreeNode(4)
root2.left.right = TreeNode(5)

print("Are the two trees identical?", are_identical_trees(root1, root2))

# Constructing two non-identical binary trees
root3 = TreeNode(1)
root3.left = TreeNode(2)
root3.right = TreeNode(3)

root4 = TreeNode(1)
root4.left = TreeNode(2)
root4.right = TreeNode(4)

print("Are the two trees identical?", are_identical_trees(root3, root4))
```

Invert/Flip Binary Tree (Create)

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

def invert_tree(root):
    if not root:
        return None

    # Swap left and right subtrees recursively
    root.left, root.right = invert_tree(root.right), invert_tree(root.left)

    return root

def print_tree_inorder(root):
    if root:
        print_tree_inorder(root.left)
        print(root.val, end=" ")
        print_tree_inorder(root.right)

# Example usage:
# Constructing a sample binary tree
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

print("Original tree (inorder traversal):")
print_tree_inorder(root)
print("\n")

# Invert the binary tree
inverted_root = invert_tree(root)

print("Inverted tree (inorder traversal):")
print_tree_inorder(inverted_root)
```

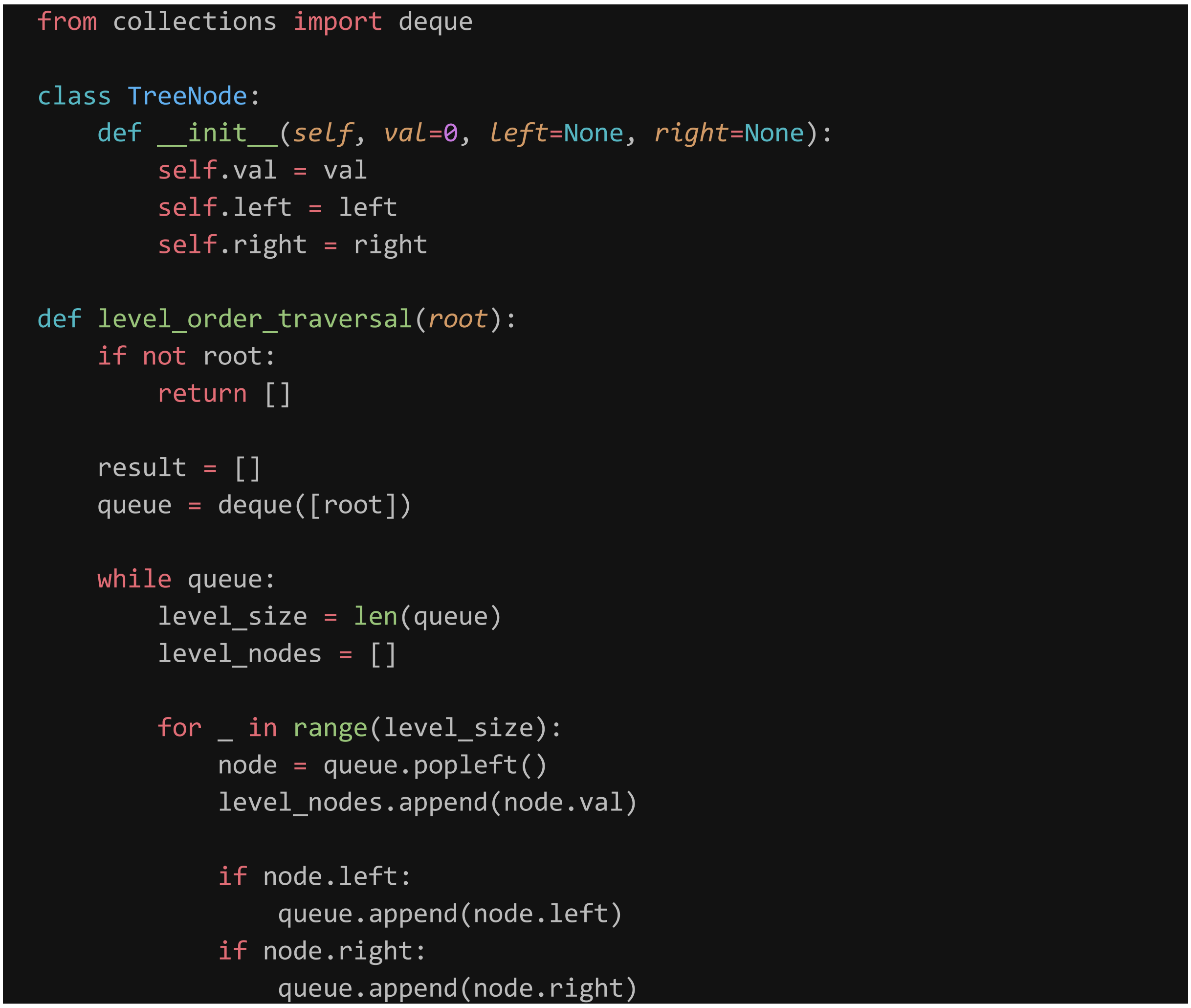
Maximum path sum

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

class Solution:
    def __init__(self):
        self.max_path_sum = float('-inf')

    def maxPathSum(self, root: TreeNode) -> int:
        def max_path_sum_helper(node):
            if not node:
                return 0

            # Calculate the maximum path sum for the left and right subtrees
            left_sum = max(0, max_path_sum_helper(node.left))
            right_sum = max(0, max_path_sum_helper(node.right))

            # Update the maximum path sum encountered so far
            self.max_path_sum = max(self.max_path_sum, node.val + left_sum +
right_sum)
```

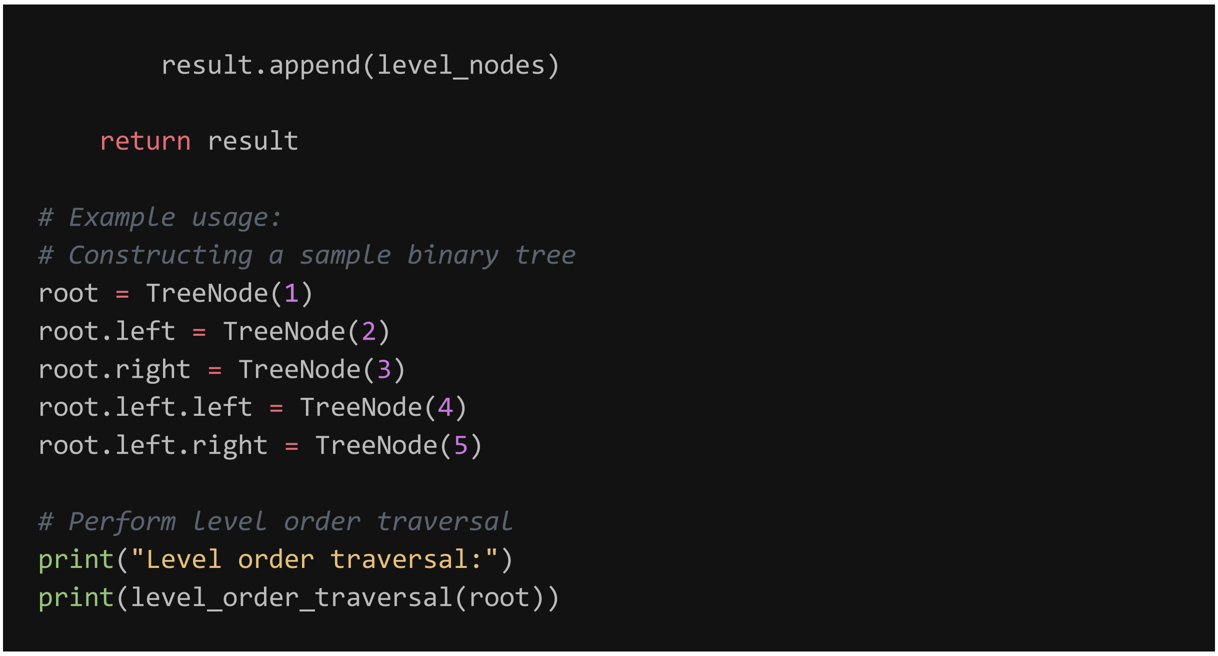


```
            # Return the maximum path sum that can be extended upwards
            return node.val + max(left_sum, right_sum)

        max_path_sum_helper(root)
        return self.max_path_sum

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)

# Initialize the solution object
solution = Solution()

# Calculate the maximum path sum
print("Maximum path sum:", solution.maxPathSum(root))
```

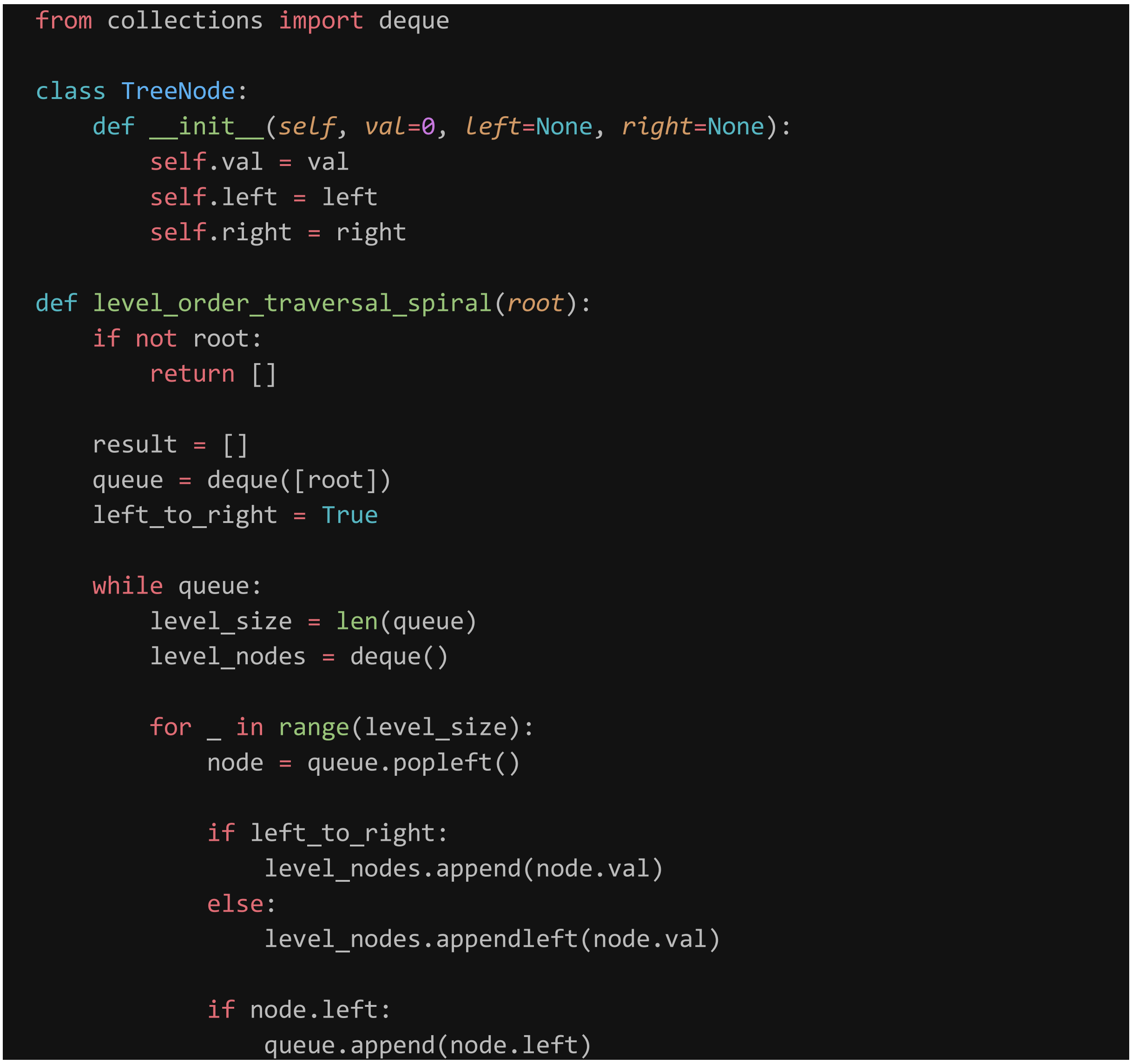
Level order Traversal / Level order traversal in spiral form



```
from collections import deque

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

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

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

    while queue:
        level_size = len(queue)
        level_nodes = []

        for _ in range(level_size):
            node = queue.popleft()
            level_nodes.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
```



```python
        result.append(level_nodes)

    return result

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform level order traversal
print("Level order traversal:")
print(level_order_traversal(root))
```



```python
from collections import deque

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

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

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

    while queue:
        level_size = len(queue)
        level_nodes = deque()

        for _ in range(level_size):
            node = queue.popleft()

            if left_to_right:
                level_nodes.append(node.val)
            else:
                level_nodes.appendleft(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(list(level_nodes))
        left_to_right = not left_to_right

    return result

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform level order traversal in spiral form
print("Level order traversal in spiral form:")
print(level_order_traversal_spiral(root))
```

Serialize and deserialize Binary Tree

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

class Codec:
    def serialize(self, root):
        """Encodes a binary tree to a single string."""
        def serialize_helper(node):
            if not node:
                return '#'
            return str(node.val) + ',' + serialize_helper(node.left) + ',' + serialize_helper(node.right)

        return serialize_helper(root)

    def deserialize(self, data):
        """Decodes your encoded data to tree."""
        def deserialize_helper(values):
            val = next(values)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = deserialize_helper(values)
            node.right = deserialize_helper(values)
            return node

        values = iter(data.split(','))
        return deserialize_helper(values)

# Example usage:
# Constructing a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.right.left = TreeNode(4)
root.right.right = TreeNode(5)

# Initialize the codec
codec = Codec()

# Serialize the binary tree
serialized_tree = codec.serialize(root)
print("Serialized tree:", serialized_tree)

# Deserialize the serialized string back into a binary tree
deserialized_tree = codec.deserialize(serialized_tree)

# Function to print inorder traversal of the binary tree
def print_inorder(root):
    if root:
        print_inorder(root.left)
        print(root.val, end=" ")
        print_inorder(root.right)

# Print inorder traversal of the deserialized tree to verify correctness
print("Inorder traversal of deserialized tree:")
print_inorder(deserialized_tree)
```

Subtree of Another Tree

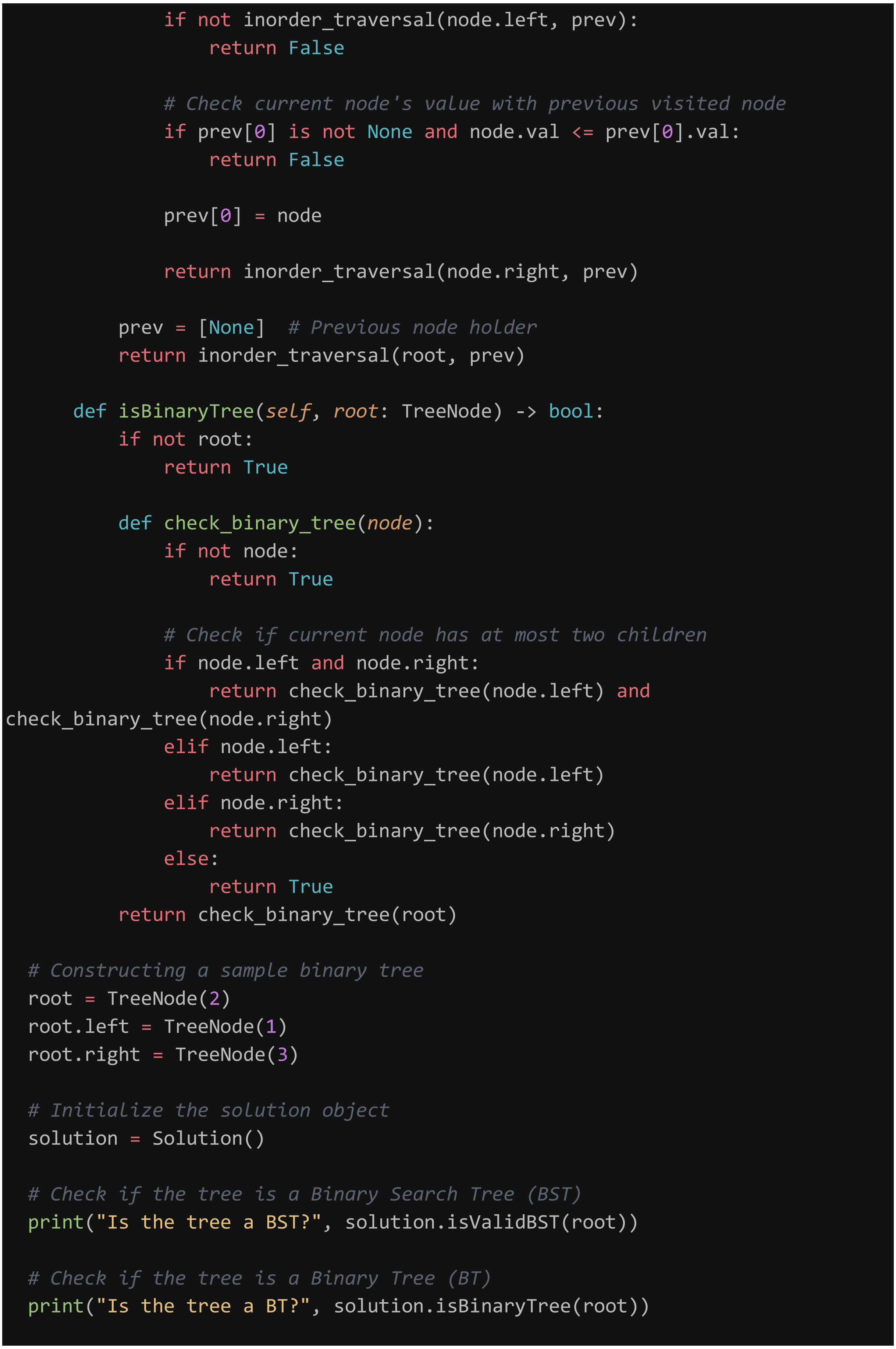
```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isSubtree(self, tree1: TreeNode, tree2: TreeNode) -> bool:
        def is_identical(s, t):
            if not s and not t:
                return True
            if not s or not t:
                return False
            return s.val == t.val and is_identical(s.left, t.left) and is_identical(s.right, t.right)

        def is_subtree_helper(s, t):
            if not s:
                return False
            return is_identical(s, t) or is_subtree_helper(s.left, t) or is_subtree_helper(s.right, t)

        return is_subtree_helper(tree1, tree2)

# Example usage:
# Constructing tree1 and tree2
tree1 = TreeNode(3)
tree1.left = TreeNode(4)
tree1.right = TreeNode(5)
tree1.left.left = TreeNode(1)
tree1.left.right = TreeNode(2)

tree2 = TreeNode(4)
tree2.left = TreeNode(1)
tree2.right = TreeNode(2)

# Initialize the solution object
solution = Solution()

# Check if tree2 is a subtree of tree1
print("Is tree2 a subtree of tree1?", solution.isSubtree(tree1, tree2))
```

Construct the Binary Tree from Postorder and Inorder Traversal

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

class Solution:
    def buildTree(self, inorder, postorder):
        if not inorder or not postorder:
            return None

        root_val = postorder.pop()
        root = TreeNode(root_val)
        inorder_index = inorder.index(root_val)

        # Build right subtree first, then left subtree (reverse due to postorder)
        root.right = self.buildTree(inorder[inorder_index + 1:], postorder)
        root.left = self.buildTree(inorder[:inorder_index], postorder)

        return root

# Example usage:
# Constructing inorder and postorder traversal sequences
inorder = [9, 3, 15, 20, 7]
postorder = [9, 15, 7, 20, 3]

# Initialize the solution object
solution = Solution()

# Build the binary tree from inorder and postorder traversal sequences
root = solution.buildTree(inorder, postorder)

# Function to perform inorder traversal of the binary tree
def print_inorder(root):
    if root:
        print_inorder(root.left)
        print(root.val, end=" ")
        print_inorder(root.right)

# Print inorder traversal of the constructed binary tree to verify correctness
print("Inorder traversal of the constructed binary tree:")
print_inorder(root)
```

Check if a tree is a BST or BT

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

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        def inorder_traversal(node, prev):
            if not node:
                return True
```



```
              if not inorder_traversal(node.left, prev):
                  return False

              # Check current node's value with previous visited node
              if prev[0] is not None and node.val <= prev[0].val:
                  return False

              prev[0] = node

              return inorder_traversal(node.right, prev)

          prev = [None]  # Previous node holder
          return inorder_traversal(root, prev)

      def isBinaryTree(self, root: TreeNode) -> bool:
          if not root:
              return True

          def check_binary_tree(node):
              if not node:
                  return True

              # Check if current node has at most two children
              if node.left and node.right:
                  return check_binary_tree(node.left) and
check_binary_tree(node.right)
              elif node.left:
                  return check_binary_tree(node.left)
              elif node.right:
                  return check_binary_tree(node.right)
              else:
                  return True
          return check_binary_tree(root)

  # Constructing a sample binary tree
  root = TreeNode(2)
  root.left = TreeNode(1)
  root.right = TreeNode(3)

  # Initialize the solution object
  solution = Solution()

  # Check if the tree is a Binary Search Tree (BST)
  print("Is the tree a BST?", solution.isValidBST(root))

  # Check if the tree is a Binary Tree (BT)
  print("Is the tree a BT?", solution.isBinaryTree(root))
```

Find K-th smallest element in BST

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

  class Solution:
      def kthSmallest(self, root: TreeNode, k: int) -> int:
          # Helper function to perform inorder traversal
          def inorder_traversal(node):
              if not node:
                  return []
              return inorder_traversal(node.left) + [node.val] +
inorder_traversal(node.right)

          # Get the inorder traversal of the BST
          inorder_elements = inorder_traversal(root)

          # Return the k-th smallest element
          return inorder_elements[k - 1]

  # Example usage:
  # Constructing a sample BST
  root = TreeNode(5)
  root.left = TreeNode(3)
  root.right = TreeNode(6)
  root.left.left = TreeNode(2)
  root.left.right = TreeNode(4)
  root.left.left.left = TreeNode(1)

  # Initialize the solution object
  solution = Solution()

  # Find the 3rd smallest element in the BST
  k = 3
  print(f"The {k}th smallest element in the BST is:",
solution.kthSmallest(root, k))
```

Find LCA of two nodes in BST

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

```python
class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
        # Ensure p.val < q.val
        if p.val > q.val:
            p, q = q, p

        # Traverse the BST to find the LCA
        current = root
        while current:
            if current.val < p.val:
                current = current.right
            elif current.val > q.val:
                current = current.left
            else:
                # Found the LCA
                return current

# Example usage:
# Constructing a sample BST
root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode(8)
root.left.left = TreeNode(0)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)
root.left.right.left = TreeNode(3)
root.left.right.right = TreeNode(5)

# Initialize the solution object
solution = Solution()

# Nodes p and q to find LCA for
p = root.left
q = root.left.right.right

# Find the LCA of nodes p and q
lca_node = solution.lowestCommonAncestor(root, p, q)
print(f"The LCA of nodes {p.val} and {q.val} is: {lca_node.val}")
```

Implement Trie (Prefix Tree)

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
```

```
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def startsWith(self, prefix: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    def delete(self, word: str) -> None:
        def _delete(node, word, index):
            if index == len(word):
                node.is_end_of_word = False
                return len(node.children) == 0

            char = word[index]
            if char in node.children:
                should_delete = _delete(node.children[char], word, index +
1)
                if should_delete:
                    del node.children[char]
                    return len(node.children) == 0
            return False

        _delete(self.root, word, 0)

# Initialize the Trie
trie = Trie()
```

```
# Insert words into the Trie
trie.insert("apple")
trie.insert("banana")
trie.insert("app")
trie.insert("ban")

# Search for words in the Trie
print(trie.search("apple"))    # Output: True
print(trie.search("app"))      # Output: True
print(trie.search("orange"))   # Output: False

# Check if prefixes exist in the Trie
print(trie.startsWith("app"))  # Output: True
print(trie.startsWith("ora"))  # Output: False

# Delete a word from the Trie
trie.delete("apple")
print(trie.search("apple"))    # Output: False
```

Implement Trie - 2 (Prefix Tree)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
    def startsWith(self, prefix: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    def delete(self, word: str) -> None:
        def _delete(node, word, index):
            if index == len(word):
                node.is_end_of_word = False
                return len(node.children) == 0

            char = word[index]
            if char in node.children:
                should_delete = _delete(node.children[char], word, index +
1)

                if should_delete:
                    del node.children[char]
                    return len(node.children) == 0
            return False

        _delete(self.root, word, 0)

# Initialize the Trie
trie = Trie()

# Insert words into the Trie
trie.insert("apple")
trie.insert("banana")
trie.insert("app")
trie.insert("ban")

# Search for words in the Trie
print(trie.search("apple"))    # Output: True
print(trie.search("app"))      # Output: True
print(trie.search("orange"))   # Output: False

# Check if prefixes exist in the Trie
print(trie.startsWith("app"))  # Output: True
print(trie.startsWith("ora"))  # Output: False

# Delete a word from the Trie
trie.delete("apple")
print(trie.search("apple"))    # Output: False
```

K most frequent elements

```
import heapq
from collections import Counter

def topKFrequent(nums, k):
    # Step 1: Build frequency counter
    freq = Counter(nums)

    # Step 2: Use a min-heap to keep track of top k frequent elements
    min_heap = []
    for num, count in freq.items():
        heapq.heappush(min_heap, (count, num))
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    # Step 3: Extract k most frequent elements from heap
    result = []
    while min_heap:
        result.append(heapq.heappop(min_heap)[1])

    return result

# Example usage:
nums = [1, 1, 1, 2, 2, 3]
k = 2
print(topKFrequent(nums, k))  # Output: [1, 2]
```

Find Median from Data Stream

```
import heapq

class MedianFinder:

    def __init__(self):
        self.left = []  # Max-heap (store negative values to simulate max-
heap)
        self.right = []  # Min-heap

    def addNum(self, num: int) -> None:
        if len(self.left) == 0 or num <= -self.left[0]:
            heapq.heappush(self.left, -num)
        else:
            heapq.heappush(self.right, num)

        # Balance the heaps
        if len(self.left) > len(self.right) + 1:
            heapq.heappush(self.right, -heapq.heappop(self.left))
```

```python
        elif len(self.right) > len(self.left):
            heapq.heappush(self.left, -heapq.heappop(self.right))

    def findMedian(self) -> float:
        if len(self.left) == len(self.right):
            return (-self.left[0] + self.right[0]) / 2.0
        else:
            return -self.left[0] / 1.0

# Example usage:
medianFinder = MedianFinder()
medianFinder.addNum(1)
medianFinder.addNum(2)
print(medianFinder.findMedian())  # Output: 1.5
medianFinder.addNum(3)
print(medianFinder.findMedian())  # Output: 2.0
```