Basic

1. Two Sum

Problem Explanation

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`. You may assume that each input would have exactly one solution, and you may not use the same element twice.

Input and Output Example

```
# Input:
nums = [2, 7, 11, 15]
target = 9

# Output:
[0, 1]
```

Core Logic

The key idea is to use a hash map (dictionary) to store the complement of each number (i.e., `target - nums[i]`) as we iterate through the array. If at any point, the complement is already in the hash map, we've found our solution.

Optimized Code in Python

```python
def two_sum(nums, target):
    complement_map = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in complement_map:
            return [complement_map[complement], i]
        complement_map[num] = i

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

---

2. Reverse Linked List

Problem Explanation

Given the head of a singly linked list, reverse the list, and return the reversed list.

Input and Output Example

```
# Input:
# Linked list: 1 -> 2 -> 3 -> 4 -> 5

# Output:
# Reversed Linked list: 5 -> 4 -> 3 -> 2 -> 1
```

Core Logic

The solution involves iterating through the linked list and reversing the `next` pointer of each node. We maintain three pointers: `prev`, `curr`, and `next`.

Optimized Code in Python

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

def reverse_linked_list(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev

# Example usage:
# Assume ListNode is already defined, and a linked list is created
# head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
# reversed_list = reverse_linked_list(head)
```

---

3. Merge Two Sorted Lists

Problem Explanation

Merge two sorted linked lists and return it as a new sorted list. The new list should be made by splicing together the nodes of the first two lists.

Input and Output Example

```
# Input:
# List 1: 1 -> 2 -> 4
# List 2: 1 -> 3 -> 4

# Output:
# Merged List: 1 -> 1 -> 2 -> 3 -> 4 -> 4
```

Core Logic

The solution involves iterating through both linked lists and merging them by comparing nodes. We create a dummy node to simplify edge cases and build the result list off this node.

Optimized Code in Python

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

def merge_two_lists(l1, l2):
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val < l2.val:
            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

# Example usage:
# merged_list = merge_two_lists(ListNode(1, ListNode(2, ListNode(4))), ListNode(1, ListNode(3,
ListNode(4))))
```

4. Palindrome Linked List

Problem Explanation

Given the head of a singly linked list, determine if it is a palindrome.

Input and Output Example

```
# Input:
# Linked list: 1 -> 2 -> 2 -> 1

# Output:
# True
```

Core Logic

To solve this problem efficiently, we can reverse the second half of the list and compare it with the first half. We use two pointers, slow and fast, to find the midpoint.

Optimized Code in Python

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

def is_palindrome(head):
    # Find the end of the first half and reverse the second half
    def end_of_first_half(node):
        fast = node
        slow = node
        while fast.next and fast.next.next:
            fast = fast.next.next
            slow = slow.next
        return slow

    def reverse_list(head):
        prev = None
        curr = head
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node
        return prev

    if not head:
        return True

    first_half_end = end_of_first_half(head)
    second_half_start = reverse_list(first_half_end.next)

    # Compare both halves
    p1 = head
    p2 = second_half_start
    result = True
    while result and p2:
        if p1.val != p2.val:
            result = False
        p1 = p1.next
        p2 = p2.next

    # Restore the list
    first_half_end.next = reverse_list(second_half_start)
    return result

# Example usage:
```

```
# palindrome_check = is_palindrome(ListNode(1, ListNode(2, ListNode(2, ListNode(1)))))
# print(palindrome_check)  # Output: True
```

---

5. Binary Tree Inorder Traversal

Problem Explanation

Given the root of a binary tree, return its inorder traversal (left, root, right).

Input and Output Example

```
# Input:
# Binary tree: [1, null, 2, 3]

# Output:
# Inorder traversal: [1, 3, 2]
```

Core Logic

The core logic involves recursively visiting the left subtree, then the root node, and finally the right subtree.

Optimized Code in Python

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

def inorder_traversal(root):
    res = []
    def inorder(node):
        if node:
            inorder(node.left)
            res.append(node.val)
            inorder(node.right)
    inorder(root)
    return res

# Example usage:
# root = TreeNode(1, None, TreeNode(2, TreeNode(3), None))
# print(inorder_traversal(root))  # Output: [1, 3, 2]
```

---

6. Binary Tree Level Order Traversal

Problem Explanation

Given the root of a binary tree, return the level order traversal of its nodes' values (i.e., from left to right, level by level).

Input and Output Example

```
# Input:
# Binary tree: [3, 9, 20, null, null, 15, 7]

# Output:
# Level order traversal: [[3], [9, 20], [15, 7]]
```

Core Logic

The solution uses a queue to process nodes level by level, appending the values to a result list as they are dequeued.

Optimized Code in Python

```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(root):
    if not root:
        return []

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

    while queue:
        level_size = len(queue)
        level = []
        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)

    return result

# Example usage:
# root = TreeNode(3, TreeNode(9), TreeNode(20, TreeNode(15), TreeNode(7)))
# print(level_order(root))  # Output: [[3], [9, 20], [15, 7]]
```

---

7. Validate Binary Search Tree

Problem Explanation

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

Input and Output Example

```
# Input:
# Binary tree: [2, 1, 3]

# Output:
# True
```

Core Logic

A valid BST has the property that all nodes in the left subtree are less than the root and all nodes in the right subtree are greater than the root. We can validate this by performing an inorder traversal and checking if the values are in ascending order.

Optimized Code in Python

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

def is_valid_bst(root, low=float('-inf'), high=float('inf')):
    if not root:
        return True

    if not (low < root.val < high):
        return False

    return (is_valid_bst(root.left, low, root.val) and
            is_valid_bst(root.right, root.val, high))

# Example usage:
# root = TreeNode(2, TreeNode(1), TreeNode(3))
# print(is_valid_bst(root))  # Output: True
```

8. Symmetric Tree

Problem Explanation

Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Input and Output Example

```
# Input:
# Binary tree: [1, 2, 2, 3, 4, 4, 3]

# Output:
# True
```

Core Logic

To check for symmetry, the tree's left and right subtrees must be mirror images of each other. This can be verified by recursively comparing nodes on opposite sides.

Optimized Code in Python

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

def is_symmetric(root):
    def is_mirror(t1, t2):
        if not t1 and not t2:
            return True
        if not t1 or not t2:
            return False
        return (t1.val == t2.val and
                is_mirror(t1.right, t2.left) and
                is_mirror(t1.left, t2.right))

    return is_mirror(root, root)

# Example usage:
# root = TreeNode(1, TreeNode(2, TreeNode(3), TreeNode(4)), TreeNode(2, TreeNode(4),
TreeNode(3)))
# print(is_symmetric(root))  # Output: True
```

9. Maximum Depth of Binary Tree

Problem Explanation

Given the root of a binary tree, return its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Input and Output Example

```
# Input:
# Binary tree: [3, 9, 20, null, null, 15, 7]

# Output:
# Maximum depth: 3
```

Core Logic

The maximum depth can be found by recursively calculating the depth of the left and right subtrees and taking the maximum of the two, then adding one for the current node.

Optimized Code in Python

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

def max_depth(root):
    if not root:
        return 0
    left_depth = max_depth(root.left)
    right_depth = max_depth(root.right)
    return max(left_depth, right_depth) + 1

# Example usage:
# root = TreeNode(3, TreeNode(9), TreeNode(20, TreeNode(15), TreeNode(7)))
# print(max_depth(root))  # Output: 3
```

---

10. Minimum Depth of Binary Tree

Problem Explanation

Given the root of a binary tree, return its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Input and Output Example

```
# Input:
# Binary tree: [3, 9, 20, null, null, 15, 7]

# Output:
# Minimum depth: 2
```

Core Logic

The minimum depth can be calculated similarly to the maximum depth, but with special consideration for cases where one subtree is missing. If a node has no left or right subtree, the depth should be calculated based on the subtree that exists.

Optimized Code in Python

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

def min_depth(root):
    if not root:
        return 0
    if not root.left:
        return min_depth(root.right) + 1
    if not root.right:
        return min_depth(root.left) + 1
    return min(min_depth(root.left), min_depth(root.right)) + 1

# Example usage:
# root = TreeNode(3, TreeNode(9), TreeNode(20, TreeNode(15), TreeNode(7)))
# print(min_depth(root))  # Output: 2
```

Intermediate

1. Kth Largest Element in an Array

Problem Explanation

Given an integer array `nums` and an integer `k`, return the `k`th largest element in the array. Note that it is the `k`th largest element in the sorted order, not the `k`th distinct element.

Input and Output Example

```
# Input:
nums = [3, 2, 1, 5, 6, 4]
k = 2

# Output:
5
```

Core Logic

A common approach is to use a min-heap to keep track of the `k` largest elements seen so far. By maintaining a heap of size `k`, we can efficiently find the `k`th largest element.

Optimized Code in Python

```python
import heapq

def find_kth_largest(nums, k):
    return heapq.nlargest(k, nums)[-1]

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

---

2. Find Peak Element

Problem Explanation

A peak element is an element that is strictly greater than its neighbors. Given an integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index of any of the peaks.

Input and Output Example

```
# Input:
nums = [1, 2, 3, 1]

# Output:
2
```

Core Logic

The problem can be solved using a binary search approach. By comparing the middle element with its neighbors, we can determine which half of the array contains a peak element.

Optimized Code in Python

```python
def find_peak_element(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1
    return left

# Example usage:
print(find_peak_element([1, 2, 3, 1]))  # Output: 2
```

---

3. Longest Substring Without Repeating Characters

Problem Explanation

Given a string `s`, find the length of the longest substring without repeating characters.

Input and Output Example

```
# Input:
s = "abcabcbb"

# Output:
3  # "abc"
```

Core Logic

The solution involves using a sliding window approach with two pointers and a set to keep track of characters in the current window. As we slide the window, we adjust the pointers to ensure no duplicate characters are in the substring.

Optimized Code in Python

```python
def length_of_longest_substring(s):
    char_set = set()
    left = 0
    max_length = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
print(length_of_longest_substring("abcabcbb"))  # Output: 3
```

---

4. Valid Parentheses

Problem Explanation

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`, determine if the input string is valid. An input string is valid if open brackets are closed by the same type of brackets and in the correct order.

Input and Output Example

```python
# Input:
s = "()[]{}"

# Output:
True
```

Core Logic

The solution involves using a stack to keep track of the opening brackets. When a closing bracket is encountered, we check if it matches the most recent opening bracket.

Optimized Code in Python

```python
def is_valid(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    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:
print(is_valid("()[]{}"))  # Output: True
```

---

5. Generate Parentheses

Problem Explanation

Given `n` pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Input and Output Example

```python
# Input:
n = 3

# Output:
["((()))", "(()())", "(())()", "()(())", "()()()"]
```

Core Logic

This problem can be solved using backtracking. We keep track of the number of open and close parentheses used and add a valid combination when the count of both reaches `n`.

Optimized Code in Python

```python
def generate_parenthesis(n):
    result = []

    def backtrack(s='', left=0, right=0):
        if len(s) == 2 * n:
            result.append(s)
            return
        if left < n:
            backtrack(s + '(', left + 1, right)
        if right < left:
            backtrack(s + ')', left, right + 1)

    backtrack()
    return result

# Example usage:
print(generate_parenthesis(3))  # Output: ["((()))", "(()())", "(())()", "()(())", "()()()"]
```

---

6. Implement Stack using Queues

Problem Explanation

Implement a last-in-first-out (LIFO) stack using only two queues. The stack should support `push`, `pop`, `top`, and `empty` operations.

Input and Output Example

```python
# Example usage:
stack = MyStack()
stack.push(1)
stack.push(2)
print(stack.top())   # Output: 2
print(stack.pop())   # Output: 2
print(stack.empty()) # Output: False
```

Core Logic

The idea is to maintain the stack order using two queues. For each push operation, we move all elements from the first queue to the second queue, insert the new element into the first queue, and then move the elements back.

Optimized Code in Python

```python
from collections import deque

class MyStack:

    def __init__(self):
        self.queue1 = deque()
        self.queue2 = deque()

    def push(self, x: int) -> None:
        self.queue2.append(x)
        while self.queue1:
            self.queue2.append(self.queue1.popleft())
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self) -> int:
        return self.queue1.popleft()

    def top(self) -> int:
        return self.queue1[0]

    def empty(self) -> bool:
        return not self.queue1

# Example usage:
stack = MyStack()
stack.push(1)
stack.push(2)
print(stack.top())   # Output: 2
print(stack.pop())   # Output: 2
print(stack.empty()) # Output: False
```

7. Implement Queue using Stacks

Problem Explanation

Implement a first-in-first-out (FIFO) queue using only two stacks. The queue should support `push`, `pop`, `peek`, and `empty` operations.

Input and Output Example

```python
# Example usage:
queue = MyQueue()
queue.push(1)
queue.push(2)
print(queue.peek())  # Output: 1
print(queue.pop())   # Output: 1
print(queue.empty()) # Output: False
```

Core Logic

The idea is to use two stacks, one for pushing elements (`in_stack`) and the other for popping elements (`out_stack`). The elements are moved from `in_stack` to `out_stack` when needed, ensuring the FIFO order.

Optimized Code in Python

```python
class MyQueue:

    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def push(self, x: int) -> None:
        self.in_stack.append(x)

    def pop(self) -> int:
        self.peek()
        return self.out_stack.pop()

    def peek(self) -> int:
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())
        return self.out_stack[-1]

    def empty(self) -> bool:
        return not self.in_stack and not self.out_stack

# Example usage:
queue = MyQueue()
queue.push(1)
queue.push(2)
print(queue.peek())  # Output: 1
print(queue.pop())   # Output: 1
print(queue.empty()) # Output: False
```

---

8. Design Circular Queue

Problem Explanation

Design your implementation of a circular queue. The circular queue is a linear data structure that follows the First In First Out (FIFO) principle but wraps around upon reaching the end.

Input and Output Example

```
# Example usage:
cq = MyCircularQueue(3)
print(cq.enQueue(1))  # Output: True
print(cq.enQueue(2))  # Output: True
print(cq.enQueue(3))  # Output: True
print(cq.enQueue(4))  # Output

: False
print(cq.Rear())      # Output: 3
print(cq.isFull())    # Output: True
print(cq.deQueue())   # Output: True
print(cq.enQueue(4))  # Output: True
print(cq.Rear())      # Output: 4
```

Core Logic

To implement a circular queue, we use a fixed-size list and two pointers (head and tail) to manage the front and rear of the queue. The pointers wrap around when they reach the end of the list.

Optimized Code in Python

```python
class MyCircularQueue:

    def __init__(self, k: int):
        self.queue = [0] * k
        self.head = self.tail = -1
        self.max_size = k

    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False
        if self.isEmpty():
            self.head = 0
        self.tail = (self.tail + 1) % self.max_size
        self.queue[self.tail] = value
        return True

    def deQueue(self) -> bool:
        if self.isEmpty():
            return False
        if self.head == self.tail:
            self.head = self.tail = -1
        else:
            self.head = (self.head + 1) % self.max_size
        return True

    def Front(self) -> int:
        if self.isEmpty():
            return -1
        return self.queue[self.head]

    def Rear(self) -> int:
        if self.isEmpty():
            return -1
        return self.queue[self.tail]

    def isEmpty(self) -> bool:
        return self.head == -1

    def isFull(self) -> bool:
        return (self.tail + 1) % self.max_size == self.head

# Example usage:
cq = MyCircularQueue(3)
print(cq.enQueue(1))  # Output: True
print(cq.enQueue(2))  # Output: True
print(cq.enQueue(3))  # Output: True
print(cq.enQueue(4))  # Output: False
print(cq.Rear())      # Output: 3
print(cq.isFull())    # Output: True
print(cq.deQueue())   # Output: True
print(cq.enQueue(4))  # Output: True
print(cq.Rear())      # Output: 4
```

---

9. Design Circular Deque

Problem Explanation

Design your implementation of the circular double-ended queue (deque). The circular deque is a linear data structure that allows elements to be added or removed from both ends.

Input and Output Example

```
# Example usage:
deque = MyCircularDeque(3)
print(deque.insertLast(1))  # Output: True
print(deque.insertLast(2))  # Output: True
print(deque.insertFront(3)) # Output: True
print(deque.insertFront(4)) # Output: False
print(deque.getRear())      # Output: 2
print(deque.isFull())       # Output: True
print(deque.deleteLast())   # Output: True
print(deque.insertFront(4)) # Output: True
print(deque.getFront())     # Output: 4
```

Core Logic

Similar to a circular queue, we use a fixed-size list with two pointers (`head` and `tail`). The difference is that we allow operations at both ends of the deque.

Optimized Code in Python

```python
class MyCircularDeque:

    def __init__(self, k: int):
        self.deque = [0] * k
        self.head = self.tail = -1
        self.max_size = k

    def insertFront(self, value: int) -> bool:
        if self.isFull():
            return False
        if self.isEmpty():
            self.head = self.tail = 0
        else:
            self.head = (self.head - 1 + self.max_size) % self.max_size
        self.deque[self.head] = value
        return True

    def insertLast(self, value: int) -> bool:
        if self.isFull():
            return False
        if self.isEmpty():
            self.head = self.tail = 0
        else:
            self.tail = (self.tail + 1) % self.max_size
        self.deque[self.tail] = value
        return True

    def deleteFront(self) -> bool:
        if self.isEmpty():
            return False
        if self.head == self.tail:
            self.head = self.tail = -1
        else:
            self.head = (self.head + 1) % self.max_size
        return True

    def deleteLast(self) -> bool:
        if self.isEmpty():
            return False
        if self.head == self.tail:
            self.head = self.tail = -1
        else:
            self.tail = (self.tail - 1 + self.max_size) % self.max_size
        return True

    def getFront(self) -> int:
        if self.isEmpty():
            return -1
        return self.deque[self.head]

    def getRear(self) -> int:
        if self.isEmpty():
            return -1
        return self.deque[self.tail]

    def isEmpty(self) -> bool:
        return self.head == -1

    def isFull(self) -> bool:
        return (self.tail + 1) % self.max_size == self.head

# Example usage:
deque = MyCircularDeque(3)
print(deque.insertLast(1))  # Output: True
print(deque.insertLast(2))  # Output: True
print(deque.insertFront(3)) # Output: True
print(deque.insertFront(4)) # Output: False
print(deque.getRear())      # Output: 2
print(deque.isFull())       # Output: True
print(deque.deleteLast())   # Output: True
print(deque.insertFront(4)) # Output: True
print(deque.getFront())     # Output: 4
```

---

10. Median of Two Sorted Arrays

Problem Explanation

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return the median of the two sorted arrays. The overall run time complexity should be `O(log(min(m,n)))`.

Input and Output Example

```python
# Input:
nums1 = [1, 3]
nums2 = [2]

# Output:
2.0
```

Core Logic

The solution involves using binary search. The idea is to partition both arrays such that the elements on the left side are less than or equal to those on the right side. The median is then calculated based on the values around the partition.

Optimized Code in Python

```python
def find_median_sorted_arrays(nums1, nums2):
    A, B = nums1, nums2
    m, n = len(A), len(B)
    if m > n:
        A, B, m, n = B, A, n, m
    imin, imax, half_len = 0, m, (m + n + 1) // 2

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

            if (m + n) % 2 == 1:
                return max_of_left

            if i == m: min_of_right = B[j]
            elif j == n: min_of_right = A[i]
            else: min_of_right = min(A[i], B[j])

            return (max_of_left + min_of_right) / 2.0

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

Advanced

1. Trapping Rain Water

Problem Explanation

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Input and Output Example

```
# Input:
height = [0,1,0,2,1,0,1,3,2,1,2,1]

# Output:
6
```

Core Logic

The solution involves using two pointers (`left` and `right`) to traverse the array from both ends. We calculate the trapped water by comparing the heights at these pointers and moving the pointers inward based on the smaller height.

Optimized Code in Python

```python
def trap(height):
    if not height:
        return 0

    left, right = 0, len(height) - 1
    left_max, right_max = height[left], height[right]
    water_trapped = 0

    while left < right:
        if height[left] < height[right]:
            left += 1
            left_max = max(left_max, height[left])
            water_trapped += left_max - height[left]
        else:
            right -= 1
            right_max = max(right_max, height[right])
            water_trapped += right_max - height[right]

    return water_trapped

# Example usage:
print(trap([0,1,0,2,1,0,1,3,2,1,2,1]))  # Output: 6
```

2. Word Ladder

Problem Explanation

Given two words, `beginWord` and `endWord`, and a dictionary of words, return the length of the shortest transformation sequence from `beginWord` to `endWord`, such that only one letter can be changed at a time, and each transformed word must exist in the word list.

Input and Output Example

```
# Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log","cog"]

# Output:
5
```

Core Logic

This problem can be solved using Breadth-First Search (BFS). We treat each word as a node and transformations as edges. BFS will help us find the shortest path from `beginWord` to `endWord`.

Optimized Code in Python

```python
from collections import deque, defaultdict

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

    L = len(beginWord)

    all_combo_dict = defaultdict(list)
    for word in wordList:
        for i in range(L):
            all_combo_dict[word[:i] + "*" + word[i+1:]].append(word)

    queue = deque([(beginWord, 1)])
    visited = {beginWord: True}

    while queue:
        current_word, level = queue.popleft()
        for i in range(L):
            intermediate_word = current_word[:i] + "*" + current_word[i+1:]
            for word in all_combo_dict[intermediate_word]:
                if word == endWord:
                    return level + 1
                if word not in visited:
                    visited[word] = True
                    queue.append((word, level + 1))
            all_combo_dict[intermediate_word] = []
    return 0

# Example usage:
print(ladder_length("hit", "cog", ["hot","dot","dog","lot","log","cog"]))  # Output: 5
```

3. Word Search

Problem Explanation

Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cells, where "adjacent" cells are horizontally or vertically neighboring.

Input and Output Example

```
# Input:
board = [
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
word = "ABCCED"

# Output:
True
```

Core Logic

We use Depth-First Search (DFS) to explore all possible paths on the board, starting from each cell that matches the first letter of the word.

Optimized Code in Python

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

    def dfs(board, word, i, j, count):
        if count == len(word):
            return True
        if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or board[i][j] !=
word[count]:
            return False

        temp = board[i][j]
        board[i][j] = ' '
        found = dfs(board, word, i + 1, j, count + 1) or dfs(board, word, i - 1, j, count + 1)
or \
                dfs(board, word, i, j + 1, count + 1) or dfs(board, word, i, j - 1, count + 1)
        board[i][j] = temp
        return found

    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == word[0] and dfs(board, word, 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(exist(board, word))  # Output: True
```

4. Clone Graph

Problem Explanation

Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node in the graph contains a value and a list of its neighbors.

Input and Output Example

```
# Input:
# Adjacency list representation of graph:
# {
#    "1": ["2","4"],
#    "2": ["1","3"],
#    "3": ["2","4"],
#    "4": ["1","3"]
# }

# Output:
# A cloned graph with the same adjacency list representation
```

Core Logic

We can use Depth-First Search (DFS) or Breadth-First Search (BFS) to traverse the graph and create a clone. A dictionary is used to store the mapping between the original nodes and their clones.

Optimized Code in Python

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

def cloneGraph(node: 'Node') -> 'Node':
    if not node:
        return node

    old_to_new = {}

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

        copy = Node(node.val)
        old_to_new[node] = copy
        for neighbor in node.neighbors:
            copy.neighbors.append(dfs(neighbor))

        return copy

    return dfs(node)

# Example usage:
# Assuming graph is built and node references are available
# clone = cloneGraph(node)
```

---

5. Course Schedule

Problem Explanation

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-1`. Some courses may have prerequisites. Given the total number of courses and a list of prerequisite pairs, determine if it is possible to finish all courses.

Input and Output Example

```
# Input:
numCourses = 2
prerequisites = [[1, 0]]

# Output:
True
```

Core Logic

This problem can be modeled as a Directed Acyclic Graph (DAG) problem. We need to detect cycles in the graph. If there's a cycle, it's impossible to complete all courses.

Optimized Code in Python

```python
from collections import defaultdict, deque

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

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

    queue = deque([i for i in range(numCourses) if indegree[i] == 0])
    visited = 0

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

    return visited == numCourses

# Example usage:
print(can_finish(2, [[1, 0]]))  # Output: True
```

---

6. Course Schedule II

Problem Explanation

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-1`. Some courses may have prerequisites. Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

Input and Output Example

```
# Input:
numCourses = 4
prerequisites = [[1, 0], [2, 0], [3, 1], [3, 2]]

# Output:
[0, 1, 2, 3]
```

Core Logic

This is a topological sorting problem in a Directed Acyclic Graph (DAG). We can use BFS (Kahn’s algorithm) to get the order of courses.

Optimized Code in Python

```python
from collections import defaultdict, deque

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

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

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

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

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

# Example usage:
print(find_order(4, [[1, 0], [2, 0], [3, 1], [3, 2]]))  # Output: [0, 1, 2, 3]
```

---

7. Top K Frequent Elements

Problem Explanation

Given a non-empty array of integers, return the `k` most frequent elements.

Input and Output Example

```python
# Input:
nums = [1,1,1,2,2,3]
k = 2

# Output:
[1, 2]
```

Core Logic

We can use a hash map to count the frequency of each element and then use a heap to keep track of the top `k` frequent elements.

Optimized Code in Python

```python
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    count = Counter(nums)
    return heapq.nlargest(k, count.keys(), key=count.get)

# Example usage:
print(top_k_frequent([1,1,1,2,2,3], 2))  # Output: [1, 2]
```

---

8. Find Median from Data Stream

Problem Explanation

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values. Design a data structure that supports the following two operations:

- `addNum(int num)`: Add a number to the data stream.
- `findMedian()`: Return the median of all elements so far.

Input and Output Example

```python
# 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
```

Core Logic

To maintain the median efficiently, we can use two heaps: a max-heap for the lower half of numbers and a min-heap for the upper half.

Optimized Code in Python

```python
import heapq

class MedianFinder:

    def __init__(self):
        self.small = []  # max-heap
        self.large = []  # min-heap

    def addNum(self, num: int) -> None:
        heapq.heappush(self.small, -heapq.heappushpop(self.large, num))
        if len(self.small) > len(self.large):
            heapq.heappush(self.large, -heapq.heappop(self.small))

    def findMedian(self) -> float:
        if len(self.large) > len(self.small):
            return float(self.large[0])
        return (self.large[0] - self.small[0]) / 2.0

# 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
```

---

9. Sliding Window Maximum

Problem Explanation

Given an array `nums`, there is a sliding window of size `k` that moves from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position, return the maximum number within the window.

Input and Output Example

```
# Input:
nums = [1,3,-1,-3,5,3,6,7]
k = 3

# Output:
[3, 3, 5, 5, 6, 7]
```

Core Logic

We can use a deque to keep track of the indices of elements in the window. The front of the deque will always have the index of the maximum element for the current window.

Optimized Code in Python

```python
from collections import deque

def max_sliding_window(nums, k):
    deq = deque()
    result = []

    for i in range(len(nums)):
        if deq and deq[0] == i - k:
            deq.popleft()

        while deq and nums[deq[-1]] < nums[i]:
            deq.pop()

        deq.append(i)

        if i >= k - 1:
            result.append(nums[deq[0]])

    return result

# Example usage:
print(max_sliding_window([1,3,-1,-3,5,3,6,7], 3))  # Output: [3, 3, 5, 5, 6, 7]
```

10. Regular Expression Matching

Problem Explanation

Given an input string `s` and a pattern `p`, implement regular expression matching with support for `.` and `*` where:

- `.` matches any single character.
- `*` matches zero or more of the preceding element.

Input and Output Example

```
# Input:
s = "aa"
p = "a*"

# Output:
True
```

Core Logic

The solution involves dynamic programming. We build a 2D DP table where `dp[i][j]` is `True` if the first `i` characters of `s` match the first `j` characters of `p`.

Optimized Code in Python

```python
def is_match(s, p):
    dp = [[False] * (len(p) + 1) for _ in range(len(s) + 1)]
    dp[0][0] = True

    for j in range(1, len(p) + 1):
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 2]

    for i in range(1, len(s) + 1):
        for j in range(1, len(p) + 1):
            if p[j - 1] == s[i - 1] or p[j - 1] == '.':
                dp[i][j] = dp[i - 1][j - 1]
            elif p[j - 1] == '*':
                dp[i][j] = dp[i][j - 2]
                if p[j - 2] == s[i - 1] or p[j - 2] == '.':
                    dp[i][j] = dp[i][j] or dp[i - 1][j]

    return dp[len(s)][len(p)]

# Example usage:
print(is_match("aa", "a*"))  # Output: True
```

Expert

1. Minimum Window Substring

Problem Explanation

Given two strings `s` and `t`, return the minimum window in `s` that will contain all the characters in `t` in any order. If there is no such window, return the empty string `""`.

Input and Output Example

```
# Input:
s = "ADOBECODEBANC"
t = "ABC"

# Output:
"BANC"
```

Core Logic

The problem can be solved using the sliding window approach. We maintain a window using two pointers and expand/shrink the window until it contains all the characters of `t`. We track the minimum window size that meets the requirement.

Optimized Code in Python

```python
from collections import Counter

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

    dict_t = Counter(t)
    required = len(dict_t)
    l, r = 0, 0
    formed = 0
    window_counts = {}
    ans = float("inf"), None, None

    while r < len(s):
        character = s[r]
        window_counts[character] = window_counts.get(character, 0) + 1

        if character in dict_t and window_counts[character] == dict_t[character]:
            formed += 1

        while l <= r and formed == required:
            character = s[l]
            if r - l + 1 < ans[0]:
                ans = (r - l + 1, l, r)

            window_counts[character] -= 1
            if character in dict_t and window_counts[character] < dict_t[character]:
                formed -= 1

            l += 1

        r += 1

    return "" if ans[0] == float("inf") else s[ans[1]: ans[2] + 1]

# Example usage:
print(min_window("ADOBECODEBANC", "ABC"))  # Output: "BANC"
```

2. Word Break

Problem Explanation

Given a string `s` and a dictionary of strings `wordDict`, return `True` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Input and Output Example

```python
# Input:
s = "leetcode"
wordDict = ["leet", "code"]

# Output:
True
```

Core Logic

The problem can be solved using dynamic programming. We use a boolean array `dp` where `dp[i]` is `True` if the substring `s[0:i]` can be segmented into words from the dictionary.

Optimized Code in Python

```python
def word_break(s, wordDict):
    dp = [False] * (len(s) + 1)
    dp[0] = True

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

    return dp[-1]

# Example usage:
print(word_break("leetcode", ["leet", "code"]))  # Output: True
```

---

3. Word Break II

Problem Explanation

Given a string `s` and a dictionary of strings `wordDict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

Input and Output Example

```
# Input:
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]

# Output:
["cats and dog", "cat sand dog"]
```

Core Logic

The problem can be solved using recursion with memoization. We generate all possible sentences by breaking down the string and checking if each segment is in the dictionary.

Optimized Code in Python

```python
def word_break(s, wordDict):
    memo = {}

    def backtrack(start):
        if start in memo:
            return memo[start]

        result = []
        if start == len(s):
            result.append("")

        for end in range(start + 1, len(s) + 1):
            word = s[start:end]
            if word in wordDict:
                for sub in backtrack(end):
                    result.append(word + ("" if sub == "" else " ") + sub)

        memo[start] = result
        return result

    return backtrack(0)

# Example usage:
print(word_break("catsanddog", ["cat", "cats", "and", "sand", "dog"]))
# Output: ["cats and dog", "cat sand dog"]
```

---

4. Maximal Rectangle

Problem Explanation

Given a binary matrix filled with `0`s and `1`s, find the largest rectangle containing only `1`s and return its area.

Input and Output Example

```
# Input:
matrix = [
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]

# Output:
6
```

Core Logic

This problem can be reduced to multiple "Largest Rectangle in Histogram" problems. For each row, we consider it as the base and calculate the largest rectangle area for the histogram formed by the height of `1`s above it.

Optimized Code in Python

```
def maximal_rectangle(matrix):
    if not matrix:
        return 0

    n = len(matrix[0])
    height = [0] * (n + 1)
    max_area = 0

    for row in matrix:
        for i in range(n):
            height[i] = height[i] + 1 if row[i] == '1' else 0

        stack = [-1]
        for i in range(n + 1):
            while height[i] < height[stack[-1]]:
                h = height[stack.pop()]
                w = i - stack[-1] - 1
                max_area = max(max_area, h * w)
            stack.append(i)

    return max_area

# Example usage:
matrix = [
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
print(maximal_rectangle(matrix))  # Output: 6
```

5. Largest Rectangle in Histogram

Problem Explanation

Given an array of integers representing the heights of a histogram's bars, return the area of the largest rectangle in the histogram.

Input and Output Example

```
# Input:
heights = [2,1,5,6,2,3]

# Output:
10
```

Core Logic

We can solve this problem using a stack to maintain the indices of the bars. By iterating through the heights and using the stack, we can calculate the maximum area for each bar as the smallest bar in the rectangle.

Optimized Code in Python

```python
def largest_rectangle_area(heights):
    stack = []
    max_area = 0
    heights.append(0)

    for i in range(len(heights)):
        while stack and heights[i] < heights[stack[-1]]:
            h = heights[stack.pop()]
            w = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, h * w)
        stack.append(i)

    heights.pop()
    return max_area

# Example usage:
print(largest_rectangle_area([2,1,5,6,2,3]))  # Output: 10
```

---

6. Serialize and Deserialize Binary Tree

Problem Explanation

Design an algorithm to serialize and deserialize a binary tree. Serialization is the process of converting a tree to a string representation, and deserialization is converting the string back to the original tree structure.

Input and Output Example

```
# Input:
# A binary tree:  [1,2,3,null,null,4,5]

# Output:
# Serialized string: "1,2,null,null,3,4,null,null,5,null,null"
# Deserialized tree should match the original tree structure
```

Core Logic

We use preorder traversal for serialization and deserialization. For null nodes, we use "null" as a placeholder.

Optimized Code in Python

```python
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):
        def helper(node):
            if not node:
                return "null,"
            return str(node.val) + "," + helper(node.left) + helper(node.right)

        return helper(root)

    def deserialize(self, data):
        def helper(nodes):
            val = nodes.pop(0)
            if val == "null":
                return None
            node = TreeNode(int(val))
            node.left = helper(nodes)
            node.right = helper(nodes)
            return node

        node_list = data.split(',')
        root = helper(node_list)
        return root

# Example usage:
codec = Codec()
root = TreeNode(1, TreeNode(2), TreeNode(3, TreeNode(4), TreeNode(5)))
serialized = codec.serialize(root)
print(serialized)  # Output:

 "1,2,null,null,3,4,null,null,5,null,null"
deserialized = codec.deserialize(serialized)
```

---

7. Alien Dictionary

Problem Explanation

Given a list of words in an alien language sorted lexicographically by the rules of that language, derive the order of characters in the alien alphabet.

Input and Output Example

```python
# Input:
words = ["wrt", "wrf", "er", "ett", "rftt"]

# Output:
"wertf"
```

Core Logic

This problem is a topological sorting problem in a directed graph, where each node represents a character, and edges represent the lexicographical order between characters.

Optimized Code in Python

```python
from collections import defaultdict, deque

def alien_order(words):
    graph = defaultdict(set)
    indegree = {char: 0 for word in words for char in word}

    for i in range(len(words) - 1):
        first, second = words[i], words[i + 1]
        for c1, c2 in zip(first, second):
            if c1 != c2:
                if c2 not in graph[c1]:
                    graph[c1].add(c2)
                    indegree[c2] += 1
                break
        else:
            if len(first) > len(second):
                return ""

    queue = deque([char for char in indegree if indegree[char] == 0])
    result = []

    while queue:
        char = queue.popleft()
        result.append(char)
        for neighbor in graph[char]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    if len(result) != len(indegree):
        return ""

    return "".join(result)

# Example usage:
print(alien_order(["wrt", "wrf", "er", "ett", "rftt"]))  # Output: "wertf"
```

---

8. Longest Increasing Subsequence

Problem Explanation

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

Input and Output Example

```
# Input:
nums = [10,9,2,5,3,7,101,18]

# Output:
4  # The LIS is [2,3,7,101]
```

Core Logic

This problem can be solved using dynamic programming or binary search with patience sorting. We maintain an array `dp` where `dp[i]` holds the length of the longest increasing subsequence that ends with `nums[i]`.

Optimized Code in Python

```python
import bisect

def length_of_lis(nums):
    sub = []
    for x in nums:
        if len(sub) == 0 or x > sub[-1]:
            sub.append(x)
        else:
            idx = bisect.bisect_left(sub, x)
            sub[idx] = x
    return len(sub)

# Example usage:
print(length_of_lis([10,9,2,5,3,7,101,18]))  # Output: 4
```

---

9. Edit Distance

Problem Explanation

Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`. You have the following operations:

- Insert a character
- Delete a character
- Replace a character

Input and Output Example

```python
# Input:
word1 = "horse"
word2 = "ros"

# Output:
3  # Explanation: horse -> rorse -> rose -> ros
```

Core Logic

This problem is a classic dynamic programming problem. We build a 2D DP table where `dp[i][j]` represents the minimum number of operations required to convert `word1[0:i]` to `word2[0:j]`.

Optimized Code in Python

```python
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1

    return dp[m][n]

# Example usage:
print(min_distance("horse", "ros"))  # Output: 3
```

---

10. Interleaving String

Problem Explanation

Given strings `s1`, `s2`, and `s3`, find if `s3` is formed by an interleaving of `s1` and `s2`. An interleaving of two strings `s1` and `s2` is a combination where their characters are mixed without changing the relative order.

Input and Output Example

```
# Input:
s1 = "aabcc"
s2 = "dbbca"
s3 = "aadbbcbcac"

# Output:
True
```

Core Logic

We can use dynamic programming to solve this problem. We maintain a 2D DP table where `dp[i][j]` is `True` if `s3[0:i+j]` can be formed by interleaving `s1[0:i]` and `s2[0:j]`.

Optimized Code in Python

```python
def is_interleave(s1, s2, s3):
    if len(s1) + len(s2) != len(s3):
        return False

    dp = [[False] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    dp[0][0] = True

    for i in range(1, len(s1) + 1):
        dp[i][0] = dp[i - 1][0] and s1[i - 1] == s3[i - 1]

    for j in range(1, len(s2) + 1):
        dp[0][j] = dp[0][j - 1] and s2[j - 1] == s3[j - 1]

    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
            dp[i][j] = (dp[i - 1][j] and s1[i - 1] == s3[i + j - 1]) or \
                       (dp[i][j - 1] and s2[j - 1] == s3[i + j - 1])

    return dp[-1][-1]

# Example usage:
print(is_interleave("aabcc", "dbbca", "aadbbcbcac"))  # Output: True
```

Master

1. Scramble String

Problem Explanation

Given two strings `s1` and `s2`, determine if `s2` is a scrambled string of `s1`. A string `s2` is a scrambled string of `s1` if it can be obtained from `s1` by recursively swapping non-empty substrings.

Input and Output Example

```
# Input:
s1 = "great"
s2 = "rgeat"

# Output:
True
```

Core Logic

We can use recursion with memoization to check if one string can be transformed into the other by scrambles. We compare the sorted characters of both strings to ensure they are the same.

Optimized Code in Python

```python
def is_scramble(s1, s2):
    if len(s1) != len(s2):
        return False
    if s1 == s2:
        return True
    if sorted(s1) != sorted(s2):
        return False

    n = len(s1)
    for i in range(1, n):
        if (is_scramble(s1[:i], s2[:i]) and is_scramble(s1[i:], s2[i:])) or \
           (is_scramble(s1[:i], s2[-i:]) and is_scramble(s1[i:], s2[:-i])):
            return True
    return False

# Example usage:
print(is_scramble("great", "rgeat"))  # Output: True
```

---

2. Best Time to Buy and Sell Stock IV

Problem Explanation

You are given an integer `k` and an array of prices where `prices[i]` is the price of a given stock on the `i-th` day. Your task is to maximize your profit by choosing at most `k` transactions.

Input and Output Example

```
# Input:
k = 2
prices = [2,4,1,7,3,8,4]

# Output:
7  # Buy on day 1 (price = 2) and sell on day 4 (price = 7). Buy on day 5 (price = 3) and sell
on day 6 (price = 8).
```

Core Logic

We can use dynamic programming to solve this problem, where `dp[i][j]` represents the maximum profit using `j` transactions up to day `i`.

Optimized Code in Python

```python
def max_profit(k, prices):
    if not prices:
        return 0
    n = len(prices)
    if k >= n // 2:
        return sum(max(prices[i + 1] - prices[i], 0) for i in range(n - 1))

    dp = [[0] * (k + 1) for _ in range(n)]

    for j in range(1, k + 1):
        max_diff = -prices[0]
        for i in range(1, n):
            dp[i][j] = max(dp[i - 1][j], prices[i] + max_diff)
            max_diff = max(max_diff, dp[i][j - 1] - prices[i])

    return dp[-1][-1]

# Example usage:
print(max_profit(2, [2,4,1,7,3,8,4]))  # Output: 7
```

---

3. Burst Balloons

Problem Explanation

You are given `n` balloons, indexed from `0` to `n - 1`. Each balloon has a number associated with it. If you burst balloon `i`, you earn `nums[i - 1] * nums[i] * nums[i + 1]` coins if `i` is not the first or last balloon. Your task is to maximize the coins you can collect by bursting the balloons wisely.

Input and Output Example

```python
# Input:
nums = [3, 1, 5, 8]

# Output:
167  # Burst balloons in the order: 1, 0, 3, 2
```

Core Logic

This problem can be solved using dynamic programming. We use a 2D DP table where `dp[i][j]` represents the maximum coins that can be collected by bursting all the balloons between `i` and `j`.

Optimized Code in Python

```python
def max_coins(nums):
    nums = [1] + nums + [1]
    n = len(nums)
    dp = [[0] * n for _ in range(n)]

    for length in range(1, n - 1):
        for left in range(1, n - length):
            right = left + length - 1
            for i in range(left, right + 1):
                dp[left][right] = max(dp[left][right], dp[left][i - 1] + dp[i + 1][right] +
nums[left - 1] * nums[i] * nums[right + 1])

    return dp[1][n - 2]

# Example usage:
print(max_coins([3, 1, 5, 8]))  # Output: 167
```

---

4. Longest Valid Parentheses

Problem Explanation

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Input and Output Example

```python
# Input:
s = "(()"

# Output:
2  # The longest valid parentheses substring is "()"
```

Core Logic

We can use a stack to keep track of the indices of the characters. Whenever we encounter a closing parenthesis, we check if it forms a valid pair with the last opened parenthesis.

Optimized Code in Python

```python
def longest_valid_parentheses(s):
    max_length = 0
    stack = [-1]

    for i in range(len(s)):
        if s[i] == '(':
            stack.append(i)
        else:
            stack.pop()
            if not stack:
                stack.append(i)
            else:
                max_length = max(max_length, i - stack[-1])

    return max_length

# Example usage:
print(longest_valid_parentheses("(()"))  # Output: 2
```

---

5. Dungeon Game

Problem Explanation

The prince is trapped in a dungeon and has to defeat all the monsters to save the princess. Each monster has a certain number of health points, and the prince has a certain amount of health points as well. Your task is to determine the minimum initial health the prince needs to survive.

Input and Output Example

```python
# Input:
dungeon = [
    [-2, -3, 3],
    [-5, -10, 1],
    [10, 30, -5]
]

# Output:
7  # The prince needs at least 7 health points to survive.
```

Core Logic

We can use dynamic programming to solve this problem by starting from the bottom-right corner of the dungeon and working our way to the top-left corner.

Optimized Code in Python

```python
def calculate_minimum_hp(dungeon):
    m, n = len(dungeon), len(dungeon[0])
    dp = [[0] * n for _ in range(m)]

    dp[m - 1][n - 1] = max(1, 1 - dungeon[m - 1][n - 1])

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

    for j in range(n - 2, -1, -1):
        dp[m - 1][j] = max(1, dp[m - 1][j + 1] - dungeon[m - 1][j])

    for i in range(m - 2, -1, -1):
        for j in range(n - 2, -1, -1):
            min_health_on_exit = min(dp[i + 1][j], dp[i][j + 1])
            dp[i][j] = max(1, min_health_on_exit - dungeon[i][j])

    return dp[0][0]

# Example usage:
dungeon = [[-2, -3, 3], [-5, -10, 1], [10, 30, -5]]
print(calculate_minimum_hp(dungeon))  # Output: 7
```

---

6. Palindrome Partitioning II

Problem Explanation

Given a string `s`, partition `s` such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of `s`.

Input and Output Example

```python
# Input:
s = "aab"

# Output:
1  # The palindrome partitioning ["aa", "b"] could be produced using 1 cut.
```

Core Logic

We can use dynamic programming to solve this problem. We maintain a DP array where `dp[i]` represents the minimum cuts needed for a palindrome partitioning of the substring `s[0:i + 1]`.

Optimized Code in Python

```python
def min_cut(s):
    n = len(s)
    if n <= 1:
        return 0

    is_palindrome = [[False] * n for _ in range(n)]
    for i in range(n):


 is_palindrome[i][i] = True
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                if length == 2:
                    is_palindrome[i][j] = True
                else:
                    is_palindrome[i][j] = is_palindrome[i + 1][j - 1]

    dp = [0] * n
    for i in range(n):
        if is_palindrome[0][i]:
            dp[i] = 0
        else:
            dp[i] = float('inf')
            for j in range(i):
                if is_palindrome[j + 1][i]:
                    dp[i] = min(dp[i], dp[j] + 1)

    return dp[-1]

# Example usage:
print(min_cut("aab"))  # Output: 1
```

---

7. Recover Binary Search Tree

Problem Explanation

You are given the root of a binary search tree (BST) where two nodes have been swapped by mistake. Your task is to recover the tree so that it becomes a valid BST.

Input and Output Example

```
# Input:
#     2
#    / \
#   3   1

# Output:
#     2
#    / \
#   1   3
```

Core Logic

We can perform an in-order traversal of the tree and look for the two nodes that are out of order. We then swap those two nodes to restore the BST.

Optimized Code in Python

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

def recover_tree(root):
    first = second = prev = None

    def inorder(node):
        nonlocal first, second, prev
        if not node:
            return
        inorder(node.left)
        if prev and prev.val > node.val:
            if not first:
                first = prev
            second = node
        prev = node
        inorder(node.right)

    inorder(root)
    first.val, second.val = second.val, first.val

# Example usage:
# root = TreeNode(2, TreeNode(3), TreeNode(1))
# recover_tree(root)
```

---

8. Count of Smaller Numbers After Self

Problem Explanation

You are given an integer array `nums` and for each element in `nums`, you need to count how many numbers are smaller than it to its right.

Input and Output Example

```
# Input:
nums = [5, 2, 6, 1]

# Output:
[2, 1, 1, 0]  # Explanation:
# There are 2 smaller numbers than 5 (2, 1),
# 1 smaller number than 2 (1),
# 1 smaller number than 6 (1),
# 0 smaller numbers than 1.
```

Core Logic

We can use a modified merge sort algorithm to count the smaller numbers while sorting the array.

Optimized Code in Python

```python
def count_smaller(nums):
    count = [0] * len(nums)
    indices = list(range(len(nums)))

    def merge_sort(start, end):
        if end - start <= 1:
            return
        mid = (start + end) // 2
        merge_sort(start, mid)
        merge_sort(mid, end)

        j = mid
        for i in range(start, mid):
            while j < end and nums[indices[i]] > nums[indices[j]]:
                j += 1
            count[indices[i]] += j - mid

        indices[start:end] = sorted(indices[start:end], key=lambda x: nums[x])

    merge_sort(0, len(nums))
    return count

# Example usage:
print(count_smaller([5, 2, 6, 1]))  # Output: [2, 1, 1, 0]
```

---

9. Palindrome Pairs

Problem Explanation

Given a list of unique words, find all pairs of distinct indices `(i, j)` such that `words[i] + words[j]` is a palindrome.

Input and Output Example

```
# Input:
words = ["bat", "tab", "cat"]

# Output:
[[0, 1], [1, 0]]  # "battab" and "tabbat" are palindromes.
```

Core Logic

We can use a hash map to store words and their indices. For each word, we check if any of its prefixes or suffixes can form a palindrome when combined with other words.

Optimized Code in Python

```python
def is_palindrome(s):
    return s == s[::-1]

def palindrome_pairs(words):
    word_map = {word: i for i, word in enumerate(words)}
    result = []

    for i, word in enumerate(words):
        for j in range(len(word) + 1):
            prefix, suffix = word[:j], word[j:]
            if is_palindrome(suffix) and prefix[::-1] in word_map and word_map[prefix[::-1]] !=
i:
                result.append([i, word_map[prefix[::-1]]])
            if j != len(word) and is_palindrome(prefix) and suffix[::-1] in word_map and
word_map[suffix[::-1]] != i:
                result.append([word_map[suffix[::-1]], i])

    return result

# Example usage:
print(palindrome_pairs(["bat", "tab", "cat"]))  # Output: [[0, 1], [1, 0]]
```

10. Shortest Palindrome

Problem Explanation

Given a string `s`, you need to find the shortest palindrome you can create by adding characters in front of it.

Input and Output Example

```
# Input:
s = "aacecaaa"

# Output:
"aaacecaaa"  # Adding "aa" in front makes it a palindrome.
```

Core Logic

We can use KMP (Knuth-Morris-Pratt) algorithm to find the longest prefix which is also a palindrome and then construct the shortest palindrome.

Optimized Code in Python

```python
def shortest_palindrome(s):
    rev_s = s[::-1]
    combined = s + "#" + rev_s
    lps = [0] * len(combined)

    for i in range(1, len(combined)):
        j = lps[i - 1]
        while j > 0 and combined[i] != combined[j]:
            j = lps[j - 1]
        if combined[i] == combined[j]:
            j += 1
        lps[i] = j

    return rev_s[:len(s) - lps[-1]] + s

# Example usage:
print(shortest_palindrome("aacecaaa"))  # Output: "aaacecaaa"
```