TakeUForward DSA Quick Revision - Complete Solutions

Day 1

---

1. 3 Sum

Problem: Find all unique triplets in the array that sum to zero.

Approach (Step-by-Step):

Brute Force:

1. Use 3 nested loops with indices i, j, k where i < j < k
2. For each triplet, check if `nums[i] + nums[j] + nums[k] == 0`
3. To avoid duplicates, sort each valid triplet and store in a set
4. Convert set to list at the end

Optimal (Two Pointer):

1. Sort the array first
2. Fix first element using loop `i` from 0 to n-2
3. Skip duplicates: if `nums[i] == nums[i-1]` and `i > 0`, continue
4. For remaining two elements, use two pointers: `left = i+1`, `right = n-1`
5. Calculate `sum = nums[i] + nums[left] + nums[right]`
6. If sum == 0: add triplet, skip duplicates by moving left while `nums[left] == nums[left+1]`, same for right, then move both pointers
7. If sum < 0: move left pointer right (need larger value)
8. If sum > 0: move right pointer left (need smaller value)

```python
def threeSum_brute(nums):
    """
    Time: O(n³) - Three nested loops
    Space: O(1) - excluding result storage
    """
    n = len(nums)
    result = set()

    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if nums[i] + nums[j] + nums[k] == 0:
                    triplet = tuple(sorted([nums[i], nums[j], nums[k]]))
                    result.add(triplet)

    return [list(t) for t in result]

def threeSum_optimal(nums):
    """
    Time: O(n²) - Sort + Two pointer for each element
    Space: O(1) - excluding result storage
    """
    nums.sort()
    result = []
    n = len(nums)

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

        left, right = i + 1, n - 1

        while left < right:
            total = nums[i] + nums[left] + nums[right]

            if total == 0:
                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
            elif total < 0:
```

```
                left += 1
            else:
                right -= 1

    return result
```

---

2. Sort an Array of 0's, 1's and 2's (Dutch National Flag)

Problem: Sort array containing only 0, 1, 2 without using sorting algorithm.

Approach (Step-by-Step):

Brute Force (Counting):

1. Count occurrences of 0, 1, 2 in first pass
2. In second pass, overwrite array: first count0 zeros, then count1 ones, then count2 twos

Optimal (Dutch National Flag - Single Pass):

1. Use 3 pointers: `low = 0`, `mid = 0`, `high = n-1`
2. Invariant: [0, low-1] = 0s, [low, mid-1] = 1s, [high+1, n-1] = 2s
3. While `mid <= high`:
   - If `arr[mid] == 0`: swap arr[low] and arr[mid], increment both low and mid
   - If `arr[mid] == 1`: just increment mid (1 is in correct region)
   - If `arr[mid] == 2`: swap arr[mid] and arr[high], decrement high only (don't increment mid, need to check swapped element)

```python
def sortColors_counting(nums):
    """
    Time: O(2n) - Two passes
    Space: O(1)
    """
    count0 = count1 = count2 = 0
    for num in nums:
        if num == 0: count0 += 1
        elif num == 1: count1 += 1
        else: count2 += 1

    idx = 0
    for _ in range(count0): nums[idx] = 0; idx += 1
    for _ in range(count1): nums[idx] = 1; idx += 1
    for _ in range(count2): nums[idx] = 2; idx += 1
    return nums

def sortColors_optimal(nums):
    """
    Time: O(n) - Single pass
    Space: O(1)
    """
    low, mid, high = 0, 0, len(nums) - 1

    while mid <= high:
        if nums[mid] == 0:
            nums[low], nums[mid] = nums[mid], nums[low]
            low += 1
            mid += 1
        elif nums[mid] == 1:
            mid += 1
        else:  # nums[mid] == 2
            nums[mid], nums[high] = nums[high], nums[mid]
            high -= 1
            # Don't increment mid - need to check swapped element

    return nums
```

---

3. Kadane's Algorithm (Maximum Subarray Sum)

Problem: Find the contiguous subarray with the largest sum.

Approach (Step-by-Step):

Brute Force:

1. For each starting index i from 0 to n-1
2. For each ending index j from i to n-1
3. Calculate sum from i to j, update max_sum if larger

Optimal (Kadane's):

1. Initialize `max_sum = -infinity`, `current_sum = 0`
2. For each element in array:
   - Add element to current_sum: `current_sum += num`
   - Update max_sum: `max_sum = max(max_sum, current_sum)`
   - If current_sum becomes negative, reset to 0: `if current_sum < 0: current_sum = 0`
   - (Negative sum won't help any future subarray, so discard it)
3. Return max_sum

To track indices:

1. Maintain `start`, `end`, `temp_start`
2. When resetting current_sum, set `temp_start = i + 1`
3. When updating max_sum, set `start = temp_start`, `end = i`

```python
def maxSubArray_brute(nums):
    """
    Time: O(n²) - Check all subarrays
    Space: O(1)
    """
    max_sum = float('-inf')
    n = len(nums)

    for i in range(n):
        current_sum = 0
        for j in range(i, n):
            current_sum += nums[j]
            max_sum = max(max_sum, current_sum)

    return max_sum

def maxSubArray_kadane(nums):
    """
    Time: O(n) - Single pass
    Space: O(1)
    """
    max_sum = float('-inf')
    current_sum = 0

    for num in nums:
        current_sum += num
        max_sum = max(max_sum, current_sum)
        if current_sum < 0:
            current_sum = 0

    return max_sum

def maxSubArray_with_indices(nums):
    """Returns max_sum and the subarray indices [start, end]"""
    max_sum = float('-inf')
    current_sum = 0
    start = end = temp_start = 0

    for i, num in enumerate(nums):
        current_sum += num

        if current_sum > max_sum:
            max_sum = current_sum
            start = temp_start
            end = i
```

```
        if current_sum < 0:
            current_sum = 0
            temp_start = i + 1

    return max_sum, start, end
```

---

4. Majority Element II (Elements appearing more than n/3 times)

Problem: Find all elements appearing more than ⌊n/3⌋ times.

Approach (Step-by-Step):

Key Insight: At most 2 elements can appear more than n/3 times (since 3 such elements would need > n elements total)

Brute Force:

1. For each unique element, count its occurrences
2. If count > n/3, add to result

Optimal (Boyer-Moore Extended):

1. **Phase 1 - Find candidates:** Maintain two candidates with their counts
2. Initialize: `candidate1 = None, candidate2 = None, count1 = 0, count2 = 0`
3. For each element:
   - If element == candidate1: increment count1
   - Else if element == candidate2: increment count2
   - Else if count1 == 0: set candidate1 = element, count1 = 1
   - Else if count2 == 0: set candidate2 = element, count2 = 1
   - Else: decrement both count1 and count2
4. **Phase 2 - Verify candidates:** Count actual occurrences of candidate1 and candidate2
5. Add to result only if actual count > n/3

```python
def majorityElement_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(nums)
    result = []

    for num in nums:
        if num not in result and nums.count(num) > n // 3:
            result.append(num)

    return result

def majorityElement_optimal(nums):
    """
    Time: O(n)
    Space: O(1)
    """
    n = len(nums)
    candidate1, candidate2 = None, None
    count1, count2 = 0, 0

    # Phase 1: Find potential candidates
    for num in nums:
        if candidate1 == num:
            count1 += 1
        elif candidate2 == num:
            count2 += 1
        elif count1 == 0:
            candidate1 = num
            count1 = 1
        elif count2 == 0:
            candidate2 = num
            count2 = 1
        else:
            count1 -= 1
            count2 -= 1

    # Phase 2: Verify candidates
    result = []
    count1, count2 = 0, 0
    for num in nums:
        if num == candidate1:
            count1 += 1
        elif num == candidate2:
```

```
            count2 += 1

    if count1 > n // 3:
        result.append(candidate1)
    if count2 > n // 3:
        result.append(candidate2)

    return result
```

---

5. Find the Repeating and Missing Number

Problem: Given array of n numbers from 1 to n, one is missing and one is repeated. Find both.

Approach (Step-by-Step):

Brute Force:

1. For each number i from 1 to n, count occurrences in array
2. If count == 2, it's repeating; if count == 0, it's missing

Optimal (Math):

1. Let X = repeating, Y = missing
2. Calculate: `S = sum(array)`, `Sn = n*(n+1)/2` (expected sum)
3. Calculate: `S2 = sum of squares of array`, `S2n = n*(n+1)*(2n+1)/6` (expected sum of squares)
4. From equations:
   - `S - Sn = X - Y` (call this diff)
   - `S2 - S2n = X² - Y² = (X-Y)(X+Y)`
   - So `X + Y = (S2 - S2n) / diff`
5. Solve: `X = (diff + sum_xy) / 2`, `Y = X - diff`

Optimal (XOR):

1. XOR all array elements with 1 to n: result = X ^ Y
2. Find rightmost set bit: `rightmost = xor_result & (-xor_result)`
3. Divide all numbers (array + 1 to n) into two buckets based on this bit
4. XOR within each bucket gives X and Y separately
5. Verify which is repeating by checking in original array

```python
def findRepeatingMissing_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(nums)
    repeating = missing = -1

    for i in range(1, n + 1):
        count = nums.count(i)
        if count == 2:
            repeating = i
        elif count == 0:
            missing = i

    return [repeating, missing]

def findRepeatingMissing_math(nums):
    """
    Time: O(n)
    Space: O(1)
    """
    n = len(nums)

    # Expected sums
    Sn = n * (n + 1) // 2
    S2n = n * (n + 1) * (2 * n + 1) // 6

    # Actual sums
    S = sum(nums)
    S2 = sum(x * x for x in nums)

    # X - Y
    diff = S - Sn
    # X + Y = (X² - Y²) / (X - Y)
    sum_xy = (S2 - S2n) // diff

    X = (diff + sum_xy) // 2  # Repeating
    Y = sum_xy - X            # Missing

    return [X, Y]

def findRepeatingMissing_xor(nums):
    """
    Time: O(n)
    Space: O(1)
```

```
    """
    n = len(nums)
    xor_all = 0

    # XOR of all array elements and 1 to n
    for i, num in enumerate(nums):
        xor_all ^= num
        xor_all ^= (i + 1)

    # xor_all = X ^ Y
    # Find rightmost set bit (differentiating bit)
    rightmost_bit = xor_all & (-xor_all)

    bucket1 = bucket2 = 0

    # Divide array elements into buckets
    for num in nums:
        if num & rightmost_bit:
            bucket1 ^= num
        else:
            bucket2 ^= num

    # Divide 1 to n into buckets
    for i in range(1, n + 1):
        if i & rightmost_bit:
            bucket1 ^= i
        else:
            bucket2 ^= i

    # Verify which is repeating
    for num in nums:
        if num == bucket1:
            return [bucket1, bucket2]  # bucket1 is repeating

    return [bucket2, bucket1]  # bucket2 is repeating
```

---

6. Maximum Product Subarray

Problem: Find contiguous subarray with maximum product.

Approach (Step-by-Step):

Key Insight: Negative × Negative = Positive, so track both max and min at each position

Brute Force:

1. For each starting index i, compute product from i to j for all j >= i
2. Update max_product if current product is larger

Optimal:

1. Initialize: `max_product = -inf`, `current_max = 1`, `current_min = 1`
2. For each element:
   - If element is negative, swap current_max and current_min (because multiplying by negative flips signs)
   - `current_max = max(element, current_max * element)`
   - `current_min = min(element, current_min * element)`
   - `max_product = max(max_product, current_max)`

Alternative (Prefix-Suffix):

1. Compute prefix products from left, suffix products from right
2. Reset to 1 when encountering 0
3. Maximum of all prefix and suffix products is the answer
4. Why it works: If there's even number of negatives, full array product is max. If odd, either prefix (excluding last negative) or suffix (excluding first negative) is max.

```python
def maxProduct_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    max_product = float('-inf')
    n = len(nums)

    for i in range(n):
        product = 1
        for j in range(i, n):
            product *= nums[j]
            max_product = max(max_product, product)

    return max_product

def maxProduct_optimal(nums):
    """
    Time: O(n)
    Space: O(1)
    """
    max_product = float('-inf')
    current_max = current_min = 1

    for num in nums:
        # If negative, swap max and min
        if num < 0:
            current_max, current_min = current_min, current_max

        # Either start fresh with num or extend previous
        current_max = max(num, current_max * num)
        current_min = min(num, current_min * num)

        max_product = max(max_product, current_max)

    return max_product

def maxProduct_prefix_suffix(nums):
    """
    Time: O(n)
    Space: O(1)
    """
    n = len(nums)
    max_product = float('-inf')
    prefix = suffix = 1
```

```
    for i in range(n):
        prefix *= nums[i]
        suffix *= nums[n - 1 - i]

        max_product = max(max_product, prefix, suffix)

        # Reset if zero encountered
        if prefix == 0:
            prefix = 1
        if suffix == 0:
            suffix = 1

    return max_product
```

---

7. Reverse Pairs (Count pairs where i < j and nums[i] > 2*nums[j])

Problem: Count reverse pairs in array.

Approach (Step-by-Step):

Brute Force:

1. For each pair (i, j) where i < j
2. Check if nums[i] > 2 * nums[j]
3. Increment count if true

Optimal (Modified Merge Sort):

1. Use merge sort, but before merging, count reverse pairs
2. In merge step, both halves are sorted
3. For counting: use two pointers - for each element in left half, find how many elements in right half satisfy the condition
4. Since right half is sorted, once `left[i] > 2*right[j]`, all elements after j also satisfy for this i
5. Count pairs: `count += (right_end - j)`
6. Then do normal merge to maintain sorted order

Counting Logic in Detail:

1. Left half: [left, mid], Right half: [mid+1, right]
2. For each `i` in left half, find first `j` in right where `arr[i] <= 2*arr[j]`
3. All indices before `j` form valid pairs with `i`
4. Since arrays are sorted, `j` never decreases as `i` increases

```python
def reversePairs_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    count = 0
    n = len(nums)

    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] > 2 * nums[j]:
                count += 1

    return count

def reversePairs_optimal(nums):
    """
    Time: O(n log n)
    Space: O(n)
    """
    def merge_count(arr, left, mid, right):
        count = 0
        j = mid + 1

        # Count pairs: for each i in left half, count valid j in right half
        for i in range(left, mid + 1):
            # Find first j where arr[i] <= 2 * arr[j]
            while j <= right and arr[i] > 2 * arr[j]:
                j += 1
            # All elements from mid+1 to j-1 form valid pairs with arr[i]
            count += (j - (mid + 1))

        # Standard merge of two sorted halves
        temp = []
        i, j = left, mid + 1
        while i <= mid and j <= right:
            if arr[i] <= arr[j]:
                temp.append(arr[i])
                i += 1
            else:
                temp.append(arr[j])
                j += 1

        while i <= mid:
            temp.append(arr[i])
            i += 1
```

```
        while j <= right:
            temp.append(arr[j])
            j += 1

        # Copy back to original array
        for i, val in enumerate(temp):
            arr[left + i] = val

        return count

    def merge_sort(arr, left, right):
        if left >= right:
            return 0

        mid = (left + right) // 2
        count = merge_sort(arr, left, mid)      # Count in left half
        count += merge_sort(arr, mid + 1, right) # Count in right half
        count += merge_count(arr, left, mid, right)  # Count cross pairs

        return count

    return merge_sort(nums, 0, len(nums) - 1)
```

Day 2

---

8. Search in Rotated Sorted Array II (with duplicates)

Problem: Search target in rotated sorted array that may contain duplicates.

Approach (Step-by-Step):

Key Insight: In rotated sorted array, at least one half is always sorted

Optimal (Modified Binary Search):

1. Initialize: `left = 0`, `right = n-1`
2. While `left <= right`:
   - `mid = (left + right) // 2`
   - If `arr[mid] == target`: return True

- **Handle duplicates:** If `arr[left] == arr[mid] == arr[right]`, we can't determine which half is sorted. Shrink both ends: `left++`, `right--`, continue
- **Left half is sorted** (arr[left] <= arr[mid]):
  - If target is in range [arr[left], arr[mid]): search left, `right = mid - 1`
  - Else: search right, `left = mid + 1`
- **Right half is sorted** (arr[mid] < arr[right]):
  - If target is in range (arr[mid], arr[right]]: search right, `left = mid + 1`
  - Else: search left, `right = mid - 1`

3. Return False

```python
def search_brute(nums, target):
    """
    Time: O(n)
    Space: O(1)
    """
    return target in nums

def search_optimal(nums, target):
    """
    Time: O(log n) average, O(n) worst case (all duplicates)
    Space: O(1)
    """
    left, right = 0, len(nums) - 1

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

        if nums[mid] == target:
            return True

        # Handle duplicates - can't determine sorted half
        if nums[left] == nums[mid] == nums[right]:
            left += 1
            right -= 1
            continue

        # Left half is sorted
        if nums[left] <= nums[mid]:
            # Target in left sorted half?
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Right half is sorted
        else:
            # Target in right sorted half?
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return False
```

9. Koko Eating Bananas

Problem: Find minimum eating speed K to eat all bananas within H hours.

Approach (Step-by-Step):

Key Insight: Binary search on answer. If speed K works, any speed > K also works.

Optimal:

1. Search space: `[1, max(piles)]` - minimum 1 banana/hr, maximum is largest pile
2. Binary search for minimum valid speed
3. For each `mid` speed, calculate hours needed: `sum(ceil(pile/mid))` for all piles
4. If hours <= H: this speed works, try smaller: `right = mid - 1`, store as potential answer
5. If hours > H: too slow, need faster: `left = mid + 1`
6. Return the stored answer

Helper function:

- `can_finish(speed)`: returns True if total hours needed <= H
- Hours for one pile = ceil(pile / speed) = (pile + speed - 1) // speed

```python
import math

def minEatingSpeed_brute(piles, h):
    """
    Time: O(max(piles) * n)
    Space: O(1)
    """
    def can_finish(speed):
        hours = sum(math.ceil(pile / speed) for pile in piles)
        return hours <= h

    for k in range(1, max(piles) + 1):
        if can_finish(k):
            return k

    return max(piles)

def minEatingSpeed_optimal(piles, h):
    """
    Time: O(n * log(max(piles)))
    Space: O(1)
    """
    def can_finish(speed):
        hours = sum((pile + speed - 1) // speed for pile in piles)  # Ceiling division
        return hours <= h

    left, right = 1, max(piles)
    result = right

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

        if can_finish(mid):
            result = mid      # Valid speed, try smaller
            right = mid - 1
        else:
            left = mid + 1    # Too slow, try faster

    return result
```

---

10. Aggressive Cows

Problem: Place C cows in N stalls to maximize minimum distance between any two cows.

Approach (Step-by-Step):

Key Insight: Binary search on answer. If we can place with distance D, we can place with any distance < D.

Optimal:

1. Sort stalls array
2. Search space: `[1, stalls[n-1] - stalls[0]]`
3. Binary search for maximum valid minimum distance
4. For each `mid` distance, check if we can place all cows:
   - Place first cow at stalls[0]
   - For each subsequent stall, place cow if distance from last placed cow >= mid
   - If we placed >= C cows, return True
5. If can_place(mid): this distance works, try larger: `left = mid + 1`, store answer
6. If not can_place(mid): distance too large: `right = mid - 1`
7. Return stored answer

```python
def aggressiveCows_brute(stalls, cows):
    """
    Time: O(n² * max_dist)
    Space: O(1)
    """
    stalls.sort()
    n = len(stalls)

    def can_place(min_dist):
        count = 1
        last_pos = stalls[0]

        for i in range(1, n):
            if stalls[i] - last_pos >= min_dist:
                count += 1
                last_pos = stalls[i]
                if count == cows:
                    return True
        return False

    max_dist = stalls[-1] - stalls[0]

    for dist in range(max_dist, 0, -1):
        if can_place(dist):
            return dist

    return 1

def aggressiveCows_optimal(stalls, cows):
    """
    Time: O(n log n + n log(max_dist))
    Space: O(1)
    """
    stalls.sort()
    n = len(stalls)

    def can_place(min_dist):
        count = 1                # Place first cow at first stall
        last_pos = stalls[0]

        for i in range(1, n):
            if stalls[i] - last_pos >= min_dist:
                count += 1
                last_pos = stalls[i]
                if count == cows:
                    return True
```

```
            return False

        left = 1
        right = stalls[-1] - stalls[0]
        result = 1

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

            if can_place(mid):
                result = mid       # Valid, try larger distance
                left = mid + 1
            else:
                right = mid - 1    # Invalid, try smaller distance

        return result
```

---

11. Median of Two Sorted Arrays

Problem: Find median of two sorted arrays in O(log(min(m,n))) time.

Approach (Step-by-Step):

Key Insight: Binary search on partition of smaller array

Optimal:

1. Ensure `nums1` is smaller array (swap if needed)
2. Binary search on nums1 for partition position
3. Total left half should have `(m + n + 1) // 2` elements
4. If `partition1` elements from nums1, then `partition2 = half_len - partition1` from nums2
5. Get max of left partitions and min of right partitions:
   - `max_left1 = nums1[partition1-1]` (or -inf if partition1 = 0)
   - `min_right1 = nums1[partition1]` (or +inf if partition1 = m)
   - Similarly for nums2
6. Valid partition: `max_left1 <= min_right2` AND `max_left2 <= min_right1`
7. If valid:
   - Odd total: return `max(max_left1, max_left2)`
   - Even total: return `(max(max_left1, max_left2) + min(min_right1, min_right2)) / 2`
8. If `max_left1 > min_right2`: move partition1 left, `right = partition1 - 1`
9. Else: move partition1 right, `left = partition1 + 1`

```python
def findMedianSortedArrays_brute(nums1, nums2):
    """
    Time: O(m + n)
    Space: O(m + n)
    """
    merged = sorted(nums1 + nums2)
    n = len(merged)

    if n % 2 == 1:
        return merged[n // 2]
    else:
        return (merged[n // 2 - 1] + merged[n // 2]) / 2

def findMedianSortedArrays_optimal(nums1, nums2):
    """
    Time: O(log(min(m, n)))
    Space: O(1)
    """
    # Ensure nums1 is smaller
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    left, right = 0, m
    half_len = (m + n + 1) // 2

    while left <= right:
        partition1 = (left + right) // 2
        partition2 = half_len - partition1

        # Get boundary elements (use -inf/+inf for edge cases)
        max_left1 = float('-inf') if partition1 == 0 else nums1[partition1 - 1]
        min_right1 = float('inf') if partition1 == m else nums1[partition1]

        max_left2 = float('-inf') if partition2 == 0 else nums2[partition2 - 1]
        min_right2 = float('inf') if partition2 == n else nums2[partition2]

        # Check if valid partition
        if max_left1 <= min_right2 and max_left2 <= min_right1:
            # Found correct partition
            if (m + n) % 2 == 1:
                return max(max_left1, max_left2)
            else:
                return (max(max_left1, max_left2) + min(min_right1, min_right2)) / 2
        elif max_left1 > min_right2:
            right = partition1 - 1   # Move partition1 left
```

```
        else:
            left = partition1 + 1    # Move partition1 right

    return 0.0
```

---

12. Count Subarrays with Given XOR K

Problem: Count subarrays with XOR equal to K.

Approach (Step-by-Step):

Key Insight: If `prefix_xor[j] ^ prefix_xor[i-1] = K`, then subarray [i, j] has XOR = K

Brute Force:

1. For each starting index i, compute XOR from i to j for all j >= i
2. Count subarrays where XOR equals K

Optimal (Prefix XOR + HashMap):

1. Initialize: `count = 0`, `prefix_xor = 0`, `hashmap = {0: 1}` (empty prefix has XOR 0)
2. For each element:
   - Update prefix_xor: `prefix_xor ^= element`
   - Calculate `target = prefix_xor ^ K`
   - If target exists in hashmap, those many subarrays end here with XOR = K
   - Add hashmap[target] to count
   - Add/increment current prefix_xor in hashmap
3. Return count

Why it works:

- If `prefix_xor ^ target = K`, then `target = prefix_xor ^ K`
- We need to find previous prefix XORs that equal `target`
- XOR of subarray [i, j] = prefix_xor[j] ^ prefix_xor[i-1]

```python
def countSubarraysXOR_brute(arr, k):
    """
    Time: O(n²)
    Space: O(1)
    """
    count = 0
    n = len(arr)

    for i in range(n):
        xor_val = 0
        for j in range(i, n):
            xor_val ^= arr[j]
            if xor_val == k:
                count += 1

    return count

def countSubarraysXOR_optimal(arr, k):
    """
    Time: O(n)
    Space: O(n)
    """
    from collections import defaultdict

    count = 0
    prefix_xor = 0
    xor_count = defaultdict(int)
    xor_count[0] = 1  # Empty prefix has XOR 0

    for num in arr:
        prefix_xor ^= num

        # If (prefix_xor ^ target = k), then target = prefix_xor ^ k
        target = prefix_xor ^ k
        count += xor_count[target]

        # Add current prefix_xor to map
        xor_count[prefix_xor] += 1

    return count
```

---

13. Combination Sum II

Problem: Find all unique combinations where candidate numbers sum to target (each number used once).

Approach (Step-by-Step):

Backtracking:

1. Sort array (crucial for handling duplicates)
2. Use recursive backtracking with parameters: `start_index`, `remaining_target`, `current_path`
3. Base case: if remaining_target == 0, add current_path to result
4. For each index i from start to end:
   - **Skip duplicates at same level:** if `i > start` and `arr[i] == arr[i-1]`, skip
   - **Pruning:** if `arr[i] > remaining_target`, break (since array is sorted)
   - Add arr[i] to path
   - Recurse with `i + 1` (can't reuse same element)
   - Remove arr[i] from path (backtrack)

```python
def combinationSum2(candidates, target):
    """
    Time: O(2^n) - subset generation
    Space: O(n) - recursion depth
    """
    candidates.sort()
    result = []

    def backtrack(start, remaining, path):
        if remaining == 0:
            result.append(path[:])
            return

        for i in range(start, len(candidates)):
            # Skip duplicates at same recursion level
            if i > start and candidates[i] == candidates[i - 1]:
                continue

            # Pruning: if current number exceeds remaining, no point continuing
            if candidates[i] > remaining:
                break

            path.append(candidates[i])
            backtrack(i + 1, remaining - candidates[i], path)
            path.pop()  # Backtrack

    backtrack(0, target, [])
    return result
```

---

14. N-Queens

Problem: Place N queens on N×N chessboard such that no two queens attack each other.

Approach (Step-by-Step):

Key Insight: Place one queen per row, track attacked columns and diagonals

Backtracking with Optimized Checking:

1. Use sets to track: `cols` (attacked columns), `pos_diag` (attacked r+c diagonals), `neg_diag` (attacked r-c diagonals)
2. Process row by row (one queen per row guaranteed)
3. For each row, try placing queen in each column:

- If column is attacked OR positive diagonal is attacked OR negative diagonal is attacked: skip
- Place queen: add to sets, mark on board
- Recurse to next row
- Backtrack: remove from sets, unmark on board

4. Base case: if row == n, all queens placed, add board to result

Why r+c and r-c for diagonals:

- Positive diagonal (↗): all cells on same diagonal have same (row + col)
- Negative diagonal (↖): all cells on same diagonal have same (row - col)

```
def solveNQueens_brute(n):
    """
    Time: O(n! * n²) - n² for validation at each step
    Space: O(n²)
    """
    def is_safe(board, row, col):
        # Check column above
        for i in range(row):
            if board[i][col] == 'Q':
                return False

        # Check upper-left diagonal
        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if board[i][j] == 'Q':
                return False
            i -= 1
            j -= 1

        # Check upper-right diagonal
        i, j = row - 1, col + 1
        while i >= 0 and j < n:
            if board[i][j] == 'Q':
                return False
            i -= 1
            j += 1

        return True

    def solve(board, row):
        if row == n:
            result.append([''.join(row) for row in board])
            return

        for col in range(n):
            if is_safe(board, row, col):
                board[row][col] = 'Q'
                solve(board, row + 1)
                board[row][col] = '.'

    result = []
    board = [['.' for _ in range(n)] for _ in range(n)]
    solve(board, 0)
    return result

def solveNQueens_optimal(n):
```

```
    """
    Time: O(n!)
    Space: O(n)
    """
    result = []
    cols = set()        # Attacked columns
    pos_diag = set()    # Attacked positive diagonals (row + col)
    neg_diag = set()    # Attacked negative diagonals (row - col)
    board = [['.' for _ in range(n)] for _ in range(n)]

    def backtrack(row):
        if row == n:
            result.append([''.join(row) for row in board])
            return

        for col in range(n):
            # Check if position is under attack
            if col in cols or (row + col) in pos_diag or (row - col) in neg_diag:
                continue

            # Place queen
            cols.add(col)
            pos_diag.add(row + col)
            neg_diag.add(row - col)
            board[row][col] = 'Q'

            backtrack(row + 1)

            # Backtrack
            cols.remove(col)
            pos_diag.remove(row + col)
            neg_diag.remove(row - col)
            board[row][col] = '.'

    backtrack(0)
    return result
```

Day 3

15. Single Number III

Problem: Given array where every element appears twice except two, find those two.

Approach (Step-by-Step):

Key Insight: XOR of all numbers = XOR of two unique numbers. Use a differentiating bit to separate them.

Optimal (Bit Manipulation):

1. XOR all numbers: result = num1 ^ num2 (since pairs cancel out)
2. Find rightmost set bit: `diff_bit = xor_result & (-xor_result)`
    - This bit is set in one number and not in other
3. Divide all numbers into two groups based on this bit
4. XOR within each group:
    - Group 1 (bit set): pairs cancel, leaves one unique number
    - Group 2 (bit not set): pairs cancel, leaves other unique number
5. Return both numbers

```python
def singleNumber_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    result = []
    for num in nums:
        if nums.count(num) == 1:
            result.append(num)
    return result

def singleNumber_hashmap(nums):
    """
    Time: O(n)
    Space: O(n)
    """
    from collections import Counter
    count = Counter(nums)
    return [num for num, freq in count.items() if freq == 1]

def singleNumber_optimal(nums):
    """
    Time: O(n)
    Space: O(1)
    """
    # XOR all numbers -> result = num1 ^ num2
    xor_all = 0
    for num in nums:
        xor_all ^= num

    # Find rightmost set bit (this bit differs between num1 and num2)
    diff_bit = xor_all & (-xor_all)

    # Divide numbers into two groups and XOR separately
    num1 = num2 = 0
    for num in nums:
        if num & diff_bit:
            num1 ^= num
        else:
            num2 ^= num

    return [num1, num2]
```

16. Minimum Number of Platforms Required

Problem: Find minimum platforms needed for trains (given arrival and departure times).

Approach (Step-by-Step):

Key Insight: Sort events and count overlapping intervals

Optimal (Two Pointers):

1. Sort arrival times and departure times separately
2. Use two pointers: `i` for arrivals, `j` for departures
3. Initialize: `platforms = 0`, `max_platforms = 0`
4. While `i < n`:
   - If `arr[i] <= dep[j]`: train arrives before/when another departs
     - Need new platform: `platforms++`
     - Update max_platforms
     - Move to next arrival: `i++`
   - Else: train departs before next arrival
     - Free a platform: `platforms--`
     - Move to next departure: `j++`
5. Return max_platforms

```python
def findPlatform_brute(arr, dep):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(arr)
    max_platforms = 1

    for i in range(n):
        count = 1
        for j in range(n):
            if i != j:
                # Check if train j overlaps with train i
                if arr[j] <= dep[i] and dep[j] >= arr[i]:
                    count += 1
        max_platforms = max(max_platforms, count)

    return max_platforms

def findPlatform_optimal(arr, dep):
    """
    Time: O(n log n)
    Space: O(1)
    """
    arr.sort()
    dep.sort()

    n = len(arr)
    platforms = 0
    max_platforms = 0
    i = j = 0

    while i < n:
        if arr[i] <= dep[j]:
            # Train arrives
            platforms += 1
            max_platforms = max(max_platforms, platforms)
            i += 1
        else:
            # Train departs
            platforms -= 1
            j += 1

    return max_platforms
```

17. N Meetings in One Room

Problem: Find maximum meetings that can be held in one room (greedy).

Approach (Step-by-Step):

Greedy Strategy: Always pick meeting that ends earliest (leaves max room for others)

1. Create list of meetings: (start, end, original_index)
2. Sort by end time
3. Select first meeting (ends earliest)
4. For each subsequent meeting:
   - If start time > last selected meeting's end time: select it
   - Update last_end to current meeting's end time
5. Return count and selected meeting indices

```
def maxMeetings(start, end):
    """
    Time: O(n log n)
    Space: O(n)
    """
    # Create meetings with index
    meetings = [(start[i], end[i], i + 1) for i in range(len(start))]

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

    result = [meetings[0][2]]  # Include first meeting
    last_end = meetings[0][1]

    for i in range(1, len(meetings)):
        if meetings[i][0] > last_end:  # Start after last meeting ends
            result.append(meetings[i][2])
            last_end = meetings[i][1]

    return len(result), result
```

---

18. Job Sequencing Problem

Problem: Maximize profit by scheduling jobs within their deadlines.

Approach (Step-by-Step):

Greedy Strategy: Process jobs by profit (descending), assign to latest available slot

1. Sort jobs by profit in descending order
2. Find max deadline to create slot array of that size
3. For each job (in profit order):
    - Find latest available slot before its deadline
    - If found: assign job to that slot, add profit
    - Mark slot as occupied
4. Return (count of jobs, total profit)

Optimal (Union-Find):

- Instead of linear search for available slot, use Union-Find
- `find(deadline)` returns latest available slot <= deadline
- After using slot, union it with slot-1

```python
def jobSequencing_greedy(jobs):
    """
    jobs: list of (id, deadline, profit)
    Time: O(n² * d)
    Space: O(d)
    """
    # Sort by profit descending
    jobs.sort(key=lambda x: x[2], reverse=True)

    max_deadline = max(job[1] for job in jobs)
    slots = [-1] * (max_deadline + 1)  # 1-indexed

    count = 0
    total_profit = 0

    for job_id, deadline, profit in jobs:
        # Find latest available slot before deadline
        for slot in range(deadline, 0, -1):
            if slots[slot] == -1:
                slots[slot] = job_id
                count += 1
                total_profit += profit
                break

    return count, total_profit

def jobSequencing_optimal(jobs):
    """
    Time: O(n log n + n * α(n)) ≈ O(n log n)
    Space: O(d)
    """
    jobs.sort(key=lambda x: x[2], reverse=True)
    max_deadline = max(job[1] for job in jobs)

    # Union-Find parent array
    parent = list(range(max_deadline + 2))

    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])  # Path compression
        return parent[x]

    count = 0
    total_profit = 0

    for job_id, deadline, profit in jobs:
```

```
        available_slot = find(deadline)

        if available_slot > 0:
            count += 1
            total_profit += profit
            # Union with previous slot
            parent[available_slot] = find(available_slot - 1)

    return count, total_profit
```

---

19. Longest Substring Without Repeating Characters

Problem: Find length of longest substring without repeating characters.

Approach (Step-by-Step):

Sliding Window:

1. Use hashmap to store last index of each character
2. Maintain window `[left, right]` with `left` being start of current valid substring
3. For each character at `right`:
    - If character exists in map AND its index >= left (in current window):
        - Move left to `map[char] + 1` (skip past the duplicate)
    - Update map[char] = right
    - Update max_length = max(max_length, right - left + 1)
4. Return max_length

```python
def lengthOfLongestSubstring_brute(s):
    """
    Time: O(n³)
    Space: O(min(n, charset_size))
    """
    def has_unique(s, start, end):
        chars = set()
        for i in range(start, end + 1):
            if s[i] in chars:
                return False
            chars.add(s[i])
        return True

    n = len(s)
    max_len = 0

    for i in range(n):
        for j in range(i, n):
            if has_unique(s, i, j):
                max_len = max(max_len, j - i + 1)

    return max_len

def lengthOfLongestSubstring_optimal(s):
    """
    Time: O(n)
    Space: O(min(n, charset_size))
    """
    char_index = {}  # Character -> last seen index
    max_len = 0
    left = 0

    for right, char in enumerate(s):
        # If char seen before and is in current window
        if char in char_index and char_index[char] >= left:
            left = char_index[char] + 1  # Move left past duplicate

        char_index[char] = right
        max_len = max(max_len, right - left + 1)

    return max_len
```

20. Longest Substring With At Most K Distinct Characters

Problem: Find longest substring with at most K distinct characters.

Approach (Step-by-Step):

Sliding Window:

1. Use hashmap to count frequency of each character in current window
2. Expand window by moving right pointer
3. If distinct count exceeds K:
   - Shrink window by moving left pointer
   - Decrement count of left character
   - Remove from map if count becomes 0
4. Update max_length after each valid window
5. Return max_length

```
def lengthOfLongestSubstringKDistinct(s, k):
    """
    Time: O(n)
    Space: O(k)
    """
    from collections import defaultdict

    if k == 0:
        return 0

    char_count = defaultdict(int)
    max_len = 0
    left = 0

    for right, char in enumerate(s):
        char_count[char] += 1

        # Shrink window if too many distinct characters
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

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

    return max_len
```

21. Count Number of Nice Subarrays

Problem: Count subarrays with exactly K odd numbers.

Approach (Step-by-Step):

Key Insight: exactly(K) = atMost(K) - atMost(K-1)

Method 1 (Prefix Count):

1. Treat odd as 1, even as 0
2. Use prefix sum of odd counts
3. Use hashmap: `{prefix_odd_count: frequency}`
4. For each position, if prefix count is X, we need prefix count (X-K) before to get exactly K odds
5. Add hashmap[X-K] to result

Method 2 (At Most):

1. Write helper `atMost(k)` that counts subarrays with at most k odd numbers
2. Use sliding window: expand right, shrink left when odd_count > k
3. For valid window, add `(right - left + 1)` subarrays
4. Answer = atMost(k) - atMost(k-1)

```python
def numberOfSubarrays_brute(nums, k):
    """
    Time: O(n²)
    Space: O(1)
    """
    count = 0
    n = len(nums)

    for i in range(n):
        odd_count = 0
        for j in range(i, n):
            if nums[j] % 2 == 1:
                odd_count += 1
            if odd_count == k:
                count += 1

    return count

def numberOfSubarrays_prefix(nums, k):
    """
    Time: O(n)
    Space: O(n)
    """
    from collections import defaultdict

    prefix_count = defaultdict(int)
    prefix_count[0] = 1  # Empty prefix has 0 odd numbers

    count = 0
    odd_count = 0

    for num in nums:
        if num % 2 == 1:
            odd_count += 1

        # Need (odd_count - k) odds before current position
        count += prefix_count[odd_count - k]
        prefix_count[odd_count] += 1

    return count

def numberOfSubarrays_atmost(nums, k):
    """
    Time: O(n)
    Space: O(1)
    """
```

```
    def at_most(k):
        if k < 0:
            return 0
        count = 0
        odd_count = 0
        left = 0

        for right in range(len(nums)):
            if nums[right] % 2 == 1:
                odd_count += 1

            while odd_count > k:
                if nums[left] % 2 == 1:
                    odd_count -= 1
                left += 1

            count += right - left + 1  # All subarrays ending at right

        return count

    return at_most(k) - at_most(k - 1)
```

Day 4 - Linked List

---

22. Sort a Linked List of 0's, 1's and 2's

Problem: Sort linked list containing only 0, 1, 2.

Approach (Step-by-Step):

Method 1 (Data Replacement):

1. Traverse and count 0s, 1s, 2s
2. Traverse again, overwrite data with count0 zeros, then count1 ones, then count2 twos

Method 2 (Link Change - Better):

1. Create three dummy nodes for 0-list, 1-list, 2-list
2. Traverse original list, append each node to appropriate list
3. Connect: 0-list -> 1-list (or 2-list if 1-list empty) -> 2-list -> None
4. Return 0-list's head

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

def sortList_dataReplace(head):
    """
    Time: O(2n)
    Space: O(1)
    """
    count = [0, 0, 0]
    temp = head

    # Count 0s, 1s, 2s
    while temp:
        count[temp.val] += 1
        temp = temp.next

    # Replace data
    temp = head
    idx = 0
    while temp:
        while count[idx] == 0:
            idx += 1
        temp.val = idx
        count[idx] -= 1
        temp = temp.next

    return head

def sortList_linkChange(head):
    """
    Time: O(n)
    Space: O(1)
    """
    if not head or not head.next:
        return head

    # Create dummy heads
    zero_head = ListNode(-1)
    one_head = ListNode(-1)
    two_head = ListNode(-1)

    zero = zero_head
    one = one_head
    two = two_head
```

```
    curr = head
    while curr:
        if curr.val == 0:
            zero.next = curr
            zero = zero.next
        elif curr.val == 1:
            one.next = curr
            one = one.next
        else:
            two.next = curr
            two = two.next
        curr = curr.next

    # Connect lists
    zero.next = one_head.next if one_head.next else two_head.next
    one.next = two_head.next
    two.next = None

    return zero_head.next
```

---

23. Check if Linked List is Palindrome

Problem: Check if linked list is palindrome.

Approach (Step-by-Step):

Optimal (O(1) space):

1. Find middle using slow-fast pointers (slow stops at mid)
2. Reverse second half (from slow.next)
3. Compare first half with reversed second half
4. (Optional) Restore list by reversing second half again

Finding middle: When fast reaches end, slow is at middle

```python
def isPalindrome_brute(head):
    """
    Time: O(n)
    Space: O(n)
    """
    values = []
    curr = head
    while curr:
        values.append(curr.val)
        curr = curr.next

    return values == values[::-1]

def isPalindrome_optimal(head):
    """
    Time: O(n)
    Space: O(1)
    """
    if not head or not head.next:
        return True

    # Step 1: Find middle (slow stops at middle)
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse second half
    def reverse(node):
        prev = None
        while node:
            next_temp = node.next
            node.next = prev
            prev = node
            node = next_temp
        return prev

    second_half = reverse(slow.next)
    slow.next = None  # Disconnect

    # Step 3: Compare
    first = head
    second = second_half
    result = True

    while second:
```

```
        if first.val != second.val:
            result = False
            break
        first = first.next
        second = second.next

    # Step 4: Restore (optional)
    slow.next = reverse(second_half)

    return result
```

---

24. Find the Starting Point of Loop in Linked List

Problem: Detect cycle and find starting node.

Approach (Step-by-Step):

Floyd's Cycle Detection:

1. Use slow and fast pointers (slow moves 1, fast moves 2)
2. If they meet, cycle exists
3. **Finding start:** Move one pointer to head, keep other at meeting point
4. Move both one step at a time - they meet at cycle start

Why it works:

- Let distance to cycle start = L, cycle length = C
- When slow enters cycle, fast has traveled 2L distance
- They meet when slow has traveled L + K steps (K < C)
- Distance from meeting point to start = C - K
- Distance from head to start = L
- Since L = C - K (can be proved mathematically), both reach start together

```python
def detectCycle_brute(head):
    """
    Time: O(n)
    Space: O(n)
    """
    visited = set()
    curr = head

    while curr:
        if curr in visited:
            return curr
        visited.add(curr)
        curr = curr.next

    return None

def detectCycle_optimal(head):
    """
    Time: O(n)
    Space: O(1)
    """
    if not head or not head.next:
        return None

    slow = fast = head

    # Step 1: Detect cycle
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            # Step 2: Find cycle start
            slow = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow

    return None
```

25. Reverse Linked List in Groups of K

Problem: Reverse every K nodes in linked list.

Approach (Step-by-Step):

1. First check if K nodes exist from current position (don't reverse if less than K)
2. Reverse K nodes:
   - Track `prev_group_end` (end of previously reversed group)
   - Track `group_start` (will become end after reversal)
   - Reverse K nodes using standard reversal
3. Connect:
   - `prev_group_end.next = new_group_start` (head of reversed group)
   - `group_start.next = next_unprocessed_node`
4. Move `prev_group_end = group_start`
5. Repeat until less than K nodes remain

```python
def reverseKGroup(head, k):
    """
    Time: O(n)
    Space: O(1)
    """
    if not head or k == 1:
        return head

    def has_k_nodes(node, k):
        count = 0
        while node and count < k:
            node = node.next
            count += 1
        return count == k

    dummy = ListNode(0)
    dummy.next = head
    prev_group_end = dummy

    while has_k_nodes(prev_group_end.next, k):
        # Reverse k nodes
        group_start = prev_group_end.next  # Will become group end
        curr = group_start
        prev = None

        for _ in range(k):
            next_temp = curr.next
            curr.next = prev
            prev = curr
            curr = next_temp

        # Connect with previous group
        prev_group_end.next = prev  # prev is new group head
        group_start.next = curr     # Connect to remaining list
        prev_group_end = group_start

    return dummy.next
```

---

26. Flattening of Linked List

Problem: Flatten a sorted linked list where each node has bottom pointer.

Approach (Step-by-Step):

Structure: Each node has `next` (horizontal) and `bottom` (vertical) pointers. Each vertical list is sorted.

Merge approach:

1. Recursively flatten from right side first
2. Merge current list with flattened next list (using bottom pointers)
3. Use merge sort's merge logic for two sorted lists

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None
        self.bottom = None

def flatten(root):
    """
    Time: O(n * m) where n is total nodes
    Space: O(1) iterative merge, O(n) for recursive stack
    """
    if not root or not root.next:
        return root

    def merge(a, b):
        """Merge two sorted lists using bottom pointer"""
        dummy = Node(0)
        tail = dummy

        while a and b:
            if a.data < b.data:
                tail.bottom = a
                a = a.bottom
            else:
                tail.bottom = b
                b = b.bottom
            tail = tail.bottom

        tail.bottom = a if a else b
        return dummy.bottom

    # Recursively flatten from right
    root.next = flatten(root.next)

    # Merge current list with flattened next list
    root = merge(root, root.next)

    return root
```

---

27. Find Intersection Point of Y Linked List

Problem: Find node where two linked lists intersect.

Approach (Step-by-Step):

Optimal (Two Pointers):

1. Start pointers at heads of both lists
2. When pointer reaches end of its list, redirect to head of OTHER list
3. Both pointers will travel same total distance (lenA + lenB)
4. They'll meet at intersection or both reach null together

Why it works:

- If intersection exists at distance `a` from headA and `b` from headB
- Pointer from A travels: a + (common) + b
- Pointer from B travels: b + (common) + a
- Both travel same distance, meet at intersection

```python
def getIntersectionNode_brute(headA, headB):
    """
    Time: O(m * n)
    Space: O(1)
    """
    currA = headA
    while currA:
        currB = headB
        while currB:
            if currA == currB:
                return currA
            currB = currB.next
        currA = currA.next
    return None

def getIntersectionNode_hashset(headA, headB):
    """
    Time: O(m + n)
    Space: O(m)
    """
    visited = set()
    curr = headA
    while curr:
        visited.add(curr)
        curr = curr.next

    curr = headB
    while curr:
        if curr in visited:
            return curr
        curr = curr.next

    return None

def getIntersectionNode_optimal(headA, headB):
    """
    Time: O(m + n)
    Space: O(1)
    """
    if not headA or not headB:
        return None

    a, b = headA, headB

    while a != b:
        # When reaching end, switch to other list's head
```

```
        a = a.next if a else headB
        b = b.next if b else headA

    return a  # Either intersection or None
```

---

28. Clone a Linked List with Random Pointer

Problem: Deep copy linked list with random pointers.

Approach (Step-by-Step):

Optimal (Interleaving):

1. **Step 1 - Interleave:** Insert clone after each original node
   - A -> A' -> B -> B' -> C -> C'
2. **Step 2 - Set random pointers:**
   - `clone.random = original.random.next` (clone of original's random)
3. **Step 3 - Separate lists:**
   - Restore original: A -> B -> C
   - Extract clone: A' -> B' -> C'

```python
class Node:
    def __init__(self, val, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random

def copyRandomList_hashmap(head):
    """
    Time: O(n)
    Space: O(n)
    """
    if not head:
        return None

    old_to_new = {}

    # First pass: create all nodes
    curr = head
    while curr:
        old_to_new[curr] = Node(curr.val)
        curr = curr.next

    # Second pass: set pointers
    curr = head
    while curr:
        old_to_new[curr].next = old_to_new.get(curr.next)
        old_to_new[curr].random = old_to_new.get(curr.random)
        curr = curr.next

    return old_to_new[head]

def copyRandomList_optimal(head):
    """
    Time: O(n)
    Space: O(1) excluding output
    """
    if not head:
        return None

    # Step 1: Interleave cloned nodes
    curr = head
    while curr:
        new_node = Node(curr.val)
        new_node.next = curr.next
        curr.next = new_node
        curr = new_node.next
```

```
    # Step 2: Set random pointers for clones
    curr = head
    while curr:
        if curr.random:
            curr.next.random = curr.random.next
        curr = curr.next.next

    # Step 3: Separate lists
    dummy = Node(0)
    copy_curr = dummy
    curr = head

    while curr:
        copy_curr.next = curr.next       # Extract clone
        copy_curr = copy_curr.next
        curr.next = curr.next.next       # Restore original
        curr = curr.next

    return dummy.next
```

Day 5 - Stack & Queue

---

29. Next Greater Element

Problem: Find next greater element for each array element.

Approach (Step-by-Step):

Monotonic Stack (Decreasing):

1. Process array from right to left
2. Maintain stack of elements (in decreasing order from bottom)
3. For each element:
   - Pop all smaller or equal elements (they can't be NGE for anyone)
   - If stack not empty, top is NGE; else NGE is -1
   - Push current element to stack
4. Why it works: Stack always has potential NGEs in decreasing order

For circular array: Process array twice (indices 0 to 2n-1, use i % n)

```python
def nextGreaterElement_brute(nums):
    """
    Time: O(n²)
    Space: O(1) excluding output
    """
    n = len(nums)
    result = [-1] * n

    for i in range(n):
        for j in range(i + 1, n):
            if nums[j] > nums[i]:
                result[i] = nums[j]
                break

    return result

def nextGreaterElement_optimal(nums):
    """
    Time: O(n)
    Space: O(n)
    """
    n = len(nums)
    result = [-1] * n
    stack = []  # Monotonic decreasing stack

    # Process from right to left
    for i in range(n - 1, -1, -1):
        # Pop smaller elements
        while stack and stack[-1] <= nums[i]:
            stack.pop()

        # Top of stack is NGE
        if stack:
            result[i] = stack[-1]

        stack.append(nums[i])

    return result

def nextGreaterElements_circular(nums):
    """For circular array"""
    n = len(nums)
    result = [-1] * n
    stack = []

    # Process array twice for circular
```

```
    for i in range(2 * n - 1, -1, -1):
        while stack and stack[-1] <= nums[i % n]:
            stack.pop()

        if i < n and stack:
            result[i] = stack[-1]

        stack.append(nums[i % n])

    return result
```

---

30. Asteroid Collision

Problem: Simulate asteroid collisions. Positive = right, Negative = left.

Approach (Step-by-Step):

Key Insight: Collision happens only when stack top is positive (moving right) and current is negative (moving left)

1. Use stack to track surviving asteroids
2. For each asteroid:
   - Set `alive = True`
   - While stack not empty AND top is positive AND current is negative:
     - Compare sizes: `|stack[-1]|` vs `|asteroid|`
     - If stack top smaller: pop it, continue checking
     - If equal: pop stack top, current dies (`alive = False`), break
     - If stack top larger: current dies (`alive = False`), break
   - If still alive, push to stack
3. Return stack

```python
def asteroidCollision(asteroids):
    """
    Time: O(n)
    Space: O(n)
    """
    stack = []

    for asteroid in asteroids:
        alive = True

        # Collision: stack top moving right (+), current moving left (-)
        while stack and asteroid < 0 < stack[-1]:
            if stack[-1] < abs(asteroid):
                # Stack top explodes, continue checking
                stack.pop()
                continue
            elif stack[-1] == abs(asteroid):
                # Both explode
                stack.pop()
            # Current asteroid explodes (either equal or smaller)
            alive = False
            break

        if alive:
            stack.append(asteroid)

    return stack
```

---

31. Sum of Subarray Ranges

Problem: Sum of (max - min) for all subarrays.

Approach (Step-by-Step):

Key Insight: Answer = (Sum of subarray maximums) - (Sum of subarray minimums)

Contribution Technique:

- For each element, count how many subarrays it's the min/max of
- Element at index i is min for subarrays [L, R] where:
  - L is first smaller element to left (or -1)
  - R is first smaller element to right (or n)
  - Count = (i - L) * (R - i)

Using Monotonic Stack:

1. For sum of minimums: use increasing stack
2. For sum of maximums: use decreasing stack
3. When popping element, calculate its contribution

```python
def subArrayRanges_brute(nums):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(nums)
    total = 0

    for i in range(n):
        min_val = max_val = nums[i]
        for j in range(i, n):
            min_val = min(min_val, nums[j])
            max_val = max(max_val, nums[j])
            total += max_val - min_val

    return total

def subArrayRanges_optimal(nums):
    """
    Time: O(n)
    Space: O(n)
    """
    n = len(nums)

    def sum_of_contributions(is_min=True):
        """Calculate sum where each element contributes as min or max"""
        result = 0
        stack = []  # (index, value)

        for i in range(n + 1):
            # Use boundary value at the end
            curr = nums[i] if i < n else (float('-inf') if is_min else float('inf'))

            while stack and (
                (is_min and stack[-1][1] > curr) or
                (not is_min and stack[-1][1] < curr)
            ):
                idx, val = stack.pop()
                left = stack[-1][0] if stack else -1
                right = i

                # Number of subarrays where nums[idx] is min/max
                count = (idx - left) * (right - idx)
                result += val * count

            stack.append((i, curr))
```

```
        return result

    sum_max = sum_of_contributions(is_min=False)
    sum_min = sum_of_contributions(is_min=True)

    return sum_max - sum_min
```

---

32. Trapping Rainwater

Problem: Calculate water trapped after rain.

Approach (Step-by-Step):

Key Insight: Water at position i = min(max_left, max_right) - height[i]

Two Pointers (Optimal):

1. Initialize: `left = 0`, `right = n-1`, `left_max = 0`, `right_max = 0`
2. While `left < right`:
   - If `height[left] < height[right]`:
     - We know right side has at least height[right], so left_max determines water
     - If `height[left] >= left_max`: update left_max
     - Else: add `left_max - height[left]` to water
     - Move `left++`
   - Else (height[right] <= height[left]):
     - We know left side has at least height[left], so right_max determines water
     - If `height[right] >= right_max`: update right_max
     - Else: add `right_max - height[right]` to water
     - Move `right--`

```python
def trap_brute(height):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(height)
    water = 0

    for i in range(n):
        left_max = max(height[:i+1]) if i >= 0 else 0
        right_max = max(height[i:]) if i < n else 0
        water += min(left_max, right_max) - height[i]

    return water

def trap_prefix(height):
    """
    Time: O(n)
    Space: O(n)
    """
    n = len(height)
    if n == 0:
        return 0

    # Precompute left_max and right_max arrays
    left_max = [0] * n
    right_max = [0] * n

    left_max[0] = height[0]
    for i in range(1, n):
        left_max[i] = max(left_max[i-1], height[i])

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

    water = 0
    for i in range(n):
        water += min(left_max[i], right_max[i]) - height[i]

    return water

def trap_optimal(height):
    """
    Time: O(n)
    Space: O(1)
```

```
    """
    n = len(height)
    if n == 0:
        return 0

    left, right = 0, n - 1
    left_max, right_max = 0, 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            if height[left] >= left_max:
                left_max = height[left]
            else:
                water += left_max - height[left]
            left += 1
        else:
            if height[right] >= right_max:
                right_max = height[right]
            else:
                water += right_max - height[right]
            right -= 1

    return water
```

---

33. Largest Rectangle in Histogram

Problem: Find largest rectangle area in histogram.

Approach (Step-by-Step):

Key Insight: For each bar, find how far it can extend left and right (bounded by smaller bars)

Monotonic Stack:

1. Use stack to store indices of bars in increasing height order
2. For each bar (including virtual bar of height 0 at end):
   - While stack not empty AND current bar height < stack top's height:
     - Pop top bar - this bar's right boundary is found
     - Calculate width: from (stack top after pop + 1) to (current index - 1)
     - Area = popped_height * width
     - Update max_area
   - Push current index to stack

3. Return max_area

Why it works: When we pop a bar, we've found both its left boundary (previous stack element) and right boundary (current element).

```python
def largestRectangleArea_brute(heights):
    """
    Time: O(n²)
    Space: O(1)
    """
    n = len(heights)
    max_area = 0

    for i in range(n):
        min_height = heights[i]
        for j in range(i, n):
            min_height = min(min_height, heights[j])
            area = min_height * (j - i + 1)
            max_area = max(max_area, area)

    return max_area

def largestRectangleArea_optimal(heights):
    """
    Time: O(n)
    Space: O(n)
    """
    n = len(heights)
    stack = []  # Store indices
    max_area = 0

    for i in range(n + 1):
        curr_height = heights[i] if i < n else 0

        while stack and heights[stack[-1]] > curr_height:
            height = heights[stack.pop()]
            # Width: from (stack[-1] + 1) to (i - 1)
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)

        stack.append(i)

    return max_area
```

34. LRU Cache

Problem: Design a Least Recently Used cache.

Approach (Step-by-Step):

Data Structures:

- Doubly Linked List: for O(1) removal and insertion at ends
- HashMap: key -> node pointer for O(1) lookup

Operations:

1. **get(key):**
   - If key not in map: return -1
   - Move node to front (most recently used)
   - Return value
2. **put(key, value):**
   - If key exists: update value, move to front
   - If at capacity: remove from tail (LRU), delete from map
   - Create new node, add to front, add to map

List structure: HEAD <-> node1 <-> node2 <-> ... <-> TAIL

- Front (after HEAD) = most recently used
- Back (before TAIL) = least recently used

```python
class LRUCache:
    """
    Time: O(1) for get and put
    Space: O(capacity)
    """
    class Node:
        def __init__(self, key=0, val=0):
            self.key = key
            self.val = val
            self.prev = None
            self.next = None

    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}  # key -> node

        # Dummy head and tail
        self.head = self.Node()
        self.tail = self.Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_to_front(self, node):
        """Add node right after head"""
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        """Remove node from list"""
        node.prev.next = node.next
        node.next.prev = node.prev

    def _move_to_front(self, node):
        """Move existing node to front"""
        self._remove_node(node)
        self._add_to_front(node)

    def get(self, key):
        if key not in self.cache:
            return -1

        node = self.cache[key]
        self._move_to_front(node)
        return node.val
```

```
    def put(self, key, value):
        if key in self.cache:
            node = self.cache[key]
            node.val = value
            self._move_to_front(node)
        else:
            if len(self.cache) >= self.capacity:
                # Remove LRU (node before tail)
                lru = self.tail.prev
                self._remove_node(lru)
                del self.cache[lru.key]

            new_node = self.Node(key, value)
            self.cache[key] = new_node
            self._add_to_front(new_node)
```

---

35. Remove K Digits

Problem: Remove K digits to make smallest number.

Approach (Step-by-Step):

Greedy with Monotonic Stack:

1. We want smaller digits at the front
2. Use stack to build result, keeping digits in increasing order when possible
3. For each digit:
    - While stack not empty AND k > 0 AND stack top > current digit:
        - Pop stack (remove larger digit)
        - Decrement k
    - Push current digit
4. If k > 0 after processing: remove k digits from end
5. Remove leading zeros
6. Return "0" if empty

```python
def removeKdigits(num, k):
    """
    Time: O(n)
    Space: O(n)
    """
    if k >= len(num):
        return "0"

    stack = []

    for digit in num:
        # Remove larger digits from stack if we still have removals left
        while stack and k > 0 and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)

    # If k removals remaining, remove from end
    while k > 0:
        stack.pop()
        k -= 1

    # Remove leading zeros and convert to string
    result = ''.join(stack).lstrip('0')

    return result if result else "0"
```

Day 6 - Trees

---

36. Lowest Common Ancestor in Binary Tree

Problem: Find LCA of two nodes in binary tree.

Approach (Step-by-Step):

Recursive:

1. Base case: if root is None or root is p or root is q, return root
2. Recursively search left subtree for p, q
3. Recursively search right subtree for p, q

4. If both return non-null: current node is LCA (p and q on different sides)
5. If only one returns non-null: that's the LCA (both nodes in one subtree, or one is ancestor of other)

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

def lowestCommonAncestor(root, p, q):
    """
    Time: O(n)
    Space: O(h) - height of tree
    """
    # Base case
    if not root or root == p or root == q:
        return root

    # Search in subtrees
    left = lowestCommonAncestor(root.left, p, q)
    right = lowestCommonAncestor(root.right, p, q)

    # If both subtrees return non-null, root is LCA
    if left and right:
        return root

    # Return whichever is non-null
    return left if left else right
```

---

37. Bottom View of Binary Tree

Problem: Print bottom view of binary tree.

Approach (Step-by-Step):

Using Level Order + Horizontal Distance:

1. Assign horizontal distance (HD) to each node:
   - Root HD = 0
   - Left child HD = parent HD - 1
   - Right child HD = parent HD + 1
2. Use level order traversal (BFS)

3. Maintain map: HD -> node value
4. For each HD, the last node visited (deepest) is in bottom view
5. Sort HDs and return corresponding values

```
from collections import deque

def bottomView(root):
    """
    Time: O(n)
    Space: O(n)
    """
    if not root:
        return []

    hd_map = {}  # horizontal distance -> node value
    queue = deque([(root, 0)])  # (node, horizontal_distance)

    while queue:
        node, hd = queue.popleft()

        # Always update - last node at this HD is in bottom view
        hd_map[hd] = node.val

        if node.left:
            queue.append((node.left, hd - 1))
        if node.right:
            queue.append((node.right, hd + 1))

    # Sort by horizontal distance
    return [hd_map[hd] for hd in sorted(hd_map.keys())]
```

---

38. Construct Binary Tree from Preorder and Inorder

Problem: Build binary tree from preorder and inorder traversals.

Approach (Step-by-Step):

Key Insights:

- Preorder: [root, left_subtree, right_subtree]
- Inorder: [left_subtree, root, right_subtree]
- First element of preorder is always root
- Find root in inorder to determine left/right subtree sizes

Algorithm:

1. Create hashmap: value -> index in inorder (for O(1) lookup)
2. First element of preorder is root
3. Find root index in inorder
4. Elements left of root in inorder = left subtree
5. Elements right of root in inorder = right subtree
6. Recursively build left and right subtrees with appropriate preorder and inorder ranges

```
def buildTree(preorder, inorder):
    """
    Time: O(n)
    Space: O(n)
    """
    if not preorder or not inorder:
        return None

    # Map for O(1) lookup in inorder
    inorder_map = {val: idx for idx, val in enumerate(inorder)}

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

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

        # Find root position in inorder
        root_idx = inorder_map[root_val]
        left_size = root_idx - in_start

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

        return root

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

39. Minimum Time to Burn Binary Tree

Problem: Find minimum time to burn entire tree from a given node.

Approach (Step-by-Step):

Key Insight: Convert tree to graph (add parent pointers), then BFS from target

1. **Build parent map:** Use BFS to create parent pointer for each node
2. **Find target node** during the traversal
3. **Multi-directional BFS from target:**
   - Start from target node
   - At each step, spread to left child, right child, and parent
   - Use visited set to avoid revisiting
   - Count levels = time to burn

```python
from collections import deque

def minTimeToBurn(root, target):
    """
    Time: O(n)
    Space: O(n)
    """
    # Step 1: Build parent map and find target node
    parent = {}
    target_node = None

    queue = deque([root])
    while queue:
        node = queue.popleft()

        if node.val == target:
            target_node = node

        if node.left:
            parent[node.left] = node
            queue.append(node.left)
        if node.right:
            parent[node.right] = node
            queue.append(node.right)

    # Step 2: BFS from target in all directions
    visited = {target_node}
    queue = deque([target_node])
    time = 0

    while queue:
        size = len(queue)
        burned_any = False

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

            # Spread to left child
            if node.left and node.left not in visited:
                visited.add(node.left)
                queue.append(node.left)
                burned_any = True

            # Spread to right child
            if node.right and node.right not in visited:
                visited.add(node.right)
```

```
                queue.append(node.right)
                burned_any = True

            # Spread to parent
            if node in parent and parent[node] not in visited:
                visited.add(parent[node])
                queue.append(parent[node])
                burned_any = True

        if burned_any:
            time += 1

    return time
```

---

40. Morris Inorder Traversal

Problem: Inorder traversal without recursion or stack (O(1) space).

Approach (Step-by-Step):

Key Insight: Use threaded binary tree - temporarily link inorder predecessor to current node

Algorithm:

1. Initialize `curr = root`
2. While `curr` is not None:
   - **If no left child:** Visit curr, move to right (`curr = curr.right`)
   - **If left child exists:**
     - Find inorder predecessor (rightmost node in left subtree)
     - **If predecessor.right is None:** Create thread to curr, move left
     - **If predecessor.right is curr:** Remove thread, visit curr, move right
3. Thread creation allows return to current node after left subtree

```python
def morrisInorder(root):
    """
    Time: O(n)
    Space: O(1)
    """
    result = []
    curr = root

    while curr:
        if not curr.left:
            # No left child - visit and go right
            result.append(curr.val)
            curr = curr.right
        else:
            # Find inorder predecessor (rightmost in left subtree)
            pred = curr.left
            while pred.right and pred.right != curr:
                pred = pred.right

            if not pred.right:
                # Create thread: predecessor -> current
                pred.right = curr
                curr = curr.left
            else:
                # Thread exists - remove it, visit current
                pred.right = None
                result.append(curr.val)
                curr = curr.right

    return result
```

---

41. LCA in BST

Problem: Find LCA in Binary Search Tree.

Approach (Step-by-Step):

Key Insight: Use BST property - no need to search both subtrees

1. If both p and q are smaller than root: LCA is in left subtree
2. If both p and q are larger than root: LCA is in right subtree
3. Otherwise (one on each side, or one equals root): root is LCA

```python
def lowestCommonAncestorBST(root, p, q):
    """
    Time: O(h)
    Space: O(1) iterative
    """
    while root:
        if p.val < root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root

    return None
```

---

42. Inorder Successor and Predecessor in BST

Problem: Find inorder successor and predecessor of a given key.

Approach (Step-by-Step):

Successor (smallest element greater than key):

1. Start from root, initialize successor = None
2. If key < node.val: this could be successor, go left to find smaller valid
3. If key >= node.val: go right (need larger value)

Predecessor (largest element smaller than key):

1. Start from root, initialize predecessor = None
2. If key > node.val: this could be predecessor, go right to find larger valid
3. If key <= node.val: go left (need smaller value)

```python
def findPredecessorSuccessor(root, key):
    """
    Time: O(h)
    Space: O(1)
    """
    predecessor = successor = None

    # Find successor
    curr = root
    while curr:
        if key < curr.val:
            successor = curr  # Potential successor
            curr = curr.left
        else:
            curr = curr.right

    # Find predecessor
    curr = root
    while curr:
        if key > curr.val:
            predecessor = curr  # Potential predecessor
            curr = curr.right
        else:
            curr = curr.left

    return predecessor, successor
```

---

43. Correct BST with Two Nodes Swapped

Problem: Two nodes in BST are swapped. Fix the BST.

Approach (Step-by-Step):

Key Insight: Inorder of BST should be sorted. Find violations.

1. Do inorder traversal, track previous node
2. Find violations where `prev.val > curr.val`
3. Two cases:
   - **Adjacent swap:** One violation. Swap `prev` and `curr`
   - **Non-adjacent swap:** Two violations. First violation's `prev` and second violation's `curr` are swapped
4. Swap the values of identified nodes

```
def recoverTree(root):
    """
    Time: O(n)
    Space: O(h) for recursion, O(1) with Morris
    """
    first = second = prev = None

    def inorder(node):
        nonlocal first, second, prev

        if not node:
            return

        inorder(node.left)

        # Check for violation
        if prev and prev.val > node.val:
            if not first:
                first = prev  # First violation
            second = node     # Always update second

        prev = node

        inorder(node.right)

    inorder(root)

    # Swap values of the two nodes
    first.val, second.val = second.val, first.val
```

---

44. Largest BST in Binary Tree

Problem: Find size of largest BST subtree.

Approach (Step-by-Step):

Post-order traversal returning (is_bst, size, min, max):

1. For each node, get info from left and right subtrees
2. Current subtree is BST if:
   - Left subtree is BST
   - Right subtree is BST
   - Left max < current value < Right min

3. If BST: size = left_size + right_size + 1
4. If not BST: propagate max size found
5. Return tuple: (is_bst, size, min_val, max_val)

```
def largestBSTSubtree(root):
    """
    Time: O(n)
    Space: O(h)
    """
    def helper(node):
        # Returns (is_bst, size, min_val, max_val)
        if not node:
            return True, 0, float('inf'), float('-inf')

        left_bst, left_size, left_min, left_max = helper(node.left)
        right_bst, right_size, right_min, right_max = helper(node.right)

        # Check if current subtree is BST
        if left_bst and right_bst and left_max < node.val < right_min:
            size = left_size + right_size + 1
            return True, size, min(left_min, node.val), max(right_max, node.val)

        # Not a BST - return max size found in subtrees
        return False, max(left_size, right_size), 0, 0

    return helper(root)[1]
```

---

45. Two Sum in BST

Problem: Find if two nodes in BST sum to target.

Approach (Step-by-Step):

Method 1 (HashSet):

1. DFS/BFS through tree
2. For each node, check if (target - node.val) exists in set
3. Add current value to set
4. Return True if found

Method 2 (Two BST Iterators - Optimal):

1. Create forward iterator (inorder - smallest to largest)

2. Create backward iterator (reverse inorder - largest to smallest)
3. Two pointer technique:
    - Get smallest (left) and largest (right) values
    - If sum == target: return True
    - If sum < target: move left iterator forward
    - If sum > target: move right iterator backward
    - Stop when iterators cross

```python
def findTarget_hashset(root, k):
    """
    Time: O(n)
    Space: O(n)
    """
    seen = set()

    def dfs(node):
        if not node:
            return False

        if k - node.val in seen:
            return True

        seen.add(node.val)
        return dfs(node.left) or dfs(node.right)

    return dfs(root)

def findTarget_twopointer(root, k):
    """
    Time: O(n)
    Space: O(h)
    """
    class BSTIterator:
        def __init__(self, root, forward=True):
            self.stack = []
            self.forward = forward
            self._push_all(root)

        def _push_all(self, node):
            while node:
                self.stack.append(node)
                node = node.left if self.forward else node.right

        def next(self):
            node = self.stack.pop()
            if self.forward:
                self._push_all(node.right)
            else:
                self._push_all(node.left)
            return node.val

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

```
    left_iter = BSTIterator(root, forward=True)   # Smallest first
    right_iter = BSTIterator(root, forward=False)  # Largest first

    left_val = left_iter.next()
    right_val = right_iter.next()

    while left_val < right_val:
        total = left_val + right_val

        if total == k:
            return True
        elif total < k:
            left_val = left_iter.next()
        else:
            right_val = right_iter.next()

    return False
```

Summary of Time Complexities

Problem	Brute Force	Optimal
3 Sum	$O(n^3)$	$O(n^2)$
Dutch National Flag	O(n log n)	O(n)
Kadane's Algorithm	$O(n^2)$	O(n)
Majority Element II	$O(n^2)$	O(n)
Repeating & Missing	$O(n^2)$	O(n)
Maximum Product Subarray	$O(n^2)$	O(n)
Reverse Pairs	$O(n^2)$	O(n log n)
Search in Rotated Array II	O(n)	O(log n) avg
Koko Eating Bananas	O(max*n)	O(n log max)
Aggressive Cows	$O(n^3)$	O(n log n)
Median of Two Arrays	O(m+n)	O(log min(m,n))
Count XOR Subarrays	$O(n^2)$	O(n)

Problem	Brute Force	Optimal
LRU Cache	-	O(1)
Largest Rectangle Histogram	$O(n^2)$	O(n)
Trapping Rainwater	$O(n^2)$	O(n)

Key Patterns to Remember

1. **Two Pointers**: 3Sum, Trapping Rain Water, Intersection of LL
2. **Sliding Window**: Longest Substring problems, Nice Subarrays
3. **Binary Search on Answer**: Koko Bananas, Aggressive Cows
4. **Monotonic Stack**: NGE, Histogram, Sum of Subarray Ranges
5. **Modified Merge Sort**: Reverse Pairs, Count Inversions
6. **Floyd's Algorithm**: Cycle Detection, Finding Duplicate
7. **Morris Traversal**: O(1) space tree traversal
8. **Boyer-Moore Voting**: Majority Element problems
9. **XOR Properties**: Single Number, Repeating-Missing
10. **Prefix Sum/XOR with HashMap**: Subarray sum/XOR problems