Advanced Algorithmic Problem Solving Solutions

1. Concept of Prefix Sum Array and Its Applications

A prefix sum array is a data structure that stores the cumulative sum of elements of an array up to a given index. For an array `A`, a prefix sum array `P` is defined such that `P[i]` is the sum of all elements from `A[0]` to `A[i]`.

Applications:

- Range sum queries in O(1) time
- Finding subarrays with a given sum
- Equilibrium index problems
- Sliding window problems

2. Sum of Elements in Range [L, R] using Prefix Sum Array

```python
def build_prefix_sum(arr):
    n = len(arr)
    prefix_sum = [0] * n
    prefix_sum[0] = arr[0]
    for i in range(1, n):
        prefix_sum[i] = prefix_sum[i-1] + arr[i]
    return prefix_sum

def range_sum(prefix_sum, l, r):
    if l == 0:
        return prefix_sum[r]
    return prefix_sum[r] - prefix_sum[l-1]

arr = [1, 2, 3, 4, 5]
prefix_sum = build_prefix_sum(arr)
print(range_sum(prefix_sum, 1, 3))
```

Time Complexity:

- Building prefix sum array: O(n)
- Range sum query: O(1)

Space Complexity: O(n) for storing the prefix sum array

3. Finding Equilibrium Index in an Array

```python
def equilibrium_index(arr):
    total_sum = sum(arr)
    left_sum = 0

    for i in range(len(arr)):
        total_sum -= arr[i]

        if left_sum == total_sum:
            return i

        left_sum += arr[i]

    return -1

arr = [-7, 1, 5, 2, -4, 3, 0]
print(equilibrium_index(arr))
```

Time Complexity: O(n)
Space Complexity: O(1)

4. Split Array into Two Parts with Equal Sum

```python
def can_split_array(arr):
    total_sum = sum(arr)
    if total_sum % 2 != 0:
        return False

    prefix_sum = 0
    for i in range(len(arr) - 1):
        prefix_sum += arr[i]
        if prefix_sum == total_sum - prefix_sum:
            return True

    return False

arr = [1, 2, 3, 4, 4]
print(can_split_array(arr))
```

Time Complexity: O(n)

Space Complexity: O(1)

5. Maximum Sum of Subarray of Size K

```python
def max_subarray_sum(arr, k):
    n = len(arr)
    if n < k:
        return -1

    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, n):
        window_sum = window_sum + arr[i] - arr[i-k]
        max_sum = max(max_sum, window_sum)

    return max_sum

arr = [1, 4, 2, 10, 2, 3, 1, 0, 20]
k = 4
print(max_subarray_sum(arr, k))
```

Time Complexity: O(n)

Space Complexity: O(1)

6. Length of Longest Substring Without Repeating Characters

```python
def length_of_longest_substring(s):
    n = len(s)
    char_index = {}
    max_length = 0
    start = 0

    for end in range(n):
        if s[end] in char_index and char_index[s[end]] >= start:
            start = char_index[s[end]] + 1
        else:
            max_length = max(max_length, end - start + 1)

        char_index[s[end]] = end

    return max_length

s = "abcabcbb"
print(length_of_longest_substring(s))
```

Time Complexity: O(n)
Space Complexity: O(min(m, n)) where m is the size of the character set

7. Sliding Window Technique and Its Use in String Problems

The sliding window technique is an algorithm that uses two pointers to create a window that slides through an array or string to process contiguous subarrays or substrings. This technique is particularly efficient for problems that require finding subarrays or substrings that satisfy certain conditions.

In string problems, the sliding window technique is used for:

- Finding the longest substring with distinct characters
- Finding anagrams in a string
- Finding minimum window substring containing all characters of another string
- Finding smallest substring with all occurrences of most frequent character

8. Longest Palindromic Substring

python

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

    n = len(s)
    start = 0
    max_length = 1

    def expand_around_center(left, right):
        while left >= 0 and right < n and s[left] == s[right]:
            left -= 1
            right += 1
        return right - left - 1

    for i in range(n):
        len1 = expand_around_center(i, i)
        len2 = expand_around_center(i, i + 1)

        length = max(len1, len2)
        if length > max_length:
            max_length = length
            start = i - (length - 1) // 2

    return s[start:start + max_length]

s = "babad"
print(longest_palindromic_substring(s))
```

Time Complexity: O(n²)
Space Complexity: O(1)

9. Longest Common Prefix Among List of Strings

python

```python
def longest_common_prefix(strs):
    if not strs:
        return ""

    min_len = min(len(s) for s in strs)

    low, high = 0, min_len

    while low < high:
        mid = (low + high + 1) // 2
        if is_common_prefix(strs, mid):
            low = mid
        else:
            high = mid - 1

    return strs[0][:low]

def is_common_prefix(strs, length):
    prefix = strs[0][:length]
    return all(s.startswith(prefix) for s in strs)

strs = ["flower", "flow", "flight"]
print(longest_common_prefix(strs))
```

Time Complexity: O(S log m) where S is the sum of all characters in all strings, m is the minimum length string
Space Complexity: O(1)

10. Generate All Permutations of a Given String

```python
def generate_permutations(s):
    result = []
    s = list(s)

    def backtrack(start):
        if start == len(s) - 1:
            result.append(''.join(s[:]))
            return

        for i in range(start, len(s)):
            s[start], s[i] = s[i], s[start]
            backtrack(start + 1)
            s[start], s[i] = s[i], s[start]

    backtrack(0)
    return result

s = "abc"
print(generate_permutations(s))
```

Time Complexity: O(n * n!)
Space Complexity: O(n * n!)

11. Two Numbers in a Sorted Array that Add Up to a Target

```python
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1

    while left < right:
        current_sum = arr[left] + arr[right]

        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1

    return [-1, -1]

arr = [2, 7, 11, 15]
target = 9
print(two_sum_sorted(arr, target))
```

Time Complexity: O(n)
Space Complexity: O(1)

12. Next Greater Permutation

python

```python
def next_permutation(nums):
    n = len(nums)

    i = n - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1

    if i >= 0:
        j = n - 1
        while nums[j] <= nums[i]:
            j -= 1
        nums[i], nums[j] = nums[j], nums[i]

    left, right = i + 1, n - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1

    return nums

nums = [1, 2, 3]
print(next_permutation(nums))
```

Time Complexity: O(n)

Space Complexity: O(1)

13. Merge Two Sorted Linked Lists

```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)
    tail = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next

    tail.next = l1 if l1 else l2

    return dummy.next
```

```python
def create_linked_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for i in range(1, len(arr)):
        current.next = ListNode(arr[i])
        current = current.next
    return head

def print_linked_list(head):
    values = []
    while head:
        values.append(str(head.val))
        head = head.next
    return "->".join(values)

l1 = create_linked_list([1, 2, 4])
l2 = create_linked_list([1, 3, 4])
merged = merge_two_lists(l1, l2)
print(print_linked_list(merged))
```

Time Complexity: O(n + m) where n and m are the lengths of the two lists

Space Complexity: O(1)

14. Median of Two Sorted Arrays

python

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

    x, y = len(nums1), len(nums2)
    low, high = 0, x

    while low <= high:
        partitionX = (low + high) // 2
        partitionY = (x + y + 1) // 2 - partitionX

        maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        minX = float('inf') if partitionX == x else nums1[partitionX]

        maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
        minY = float('inf') if partitionY == y else nums2[partitionY]

        if maxX <= minY and maxY <= minX:
            if (x + y) % 2 == 0:
                return (max(maxX, maxY) + min(minX, minY)) / 2
            else:
                return max(maxX, maxY)
        elif maxX > minY:
            high = partitionX - 1
        else:
            low = partitionX + 1

nums1 = [1, 3]
nums2 = [2]
print(find_median_sorted_arrays(nums1, nums2))
```

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

15. Kth Smallest Element in a Sorted Matrix

python

```python
import heapq

def kth_smallest(matrix, k):
    n = len(matrix)
    min_heap = []

    for r in range(min(n, k)):
        heapq.heappush(min_heap, (matrix[r][0], r, 0))

    for i in range(k - 1):
        val, r, c = heapq.heappop(min_heap)
        if c + 1 < n:
            heapq.heappush(min_heap, (matrix[r][c + 1], r, c + 1))

    return heapq.heappop(min_heap)[0]

matrix = [
    [1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
]
k = 8
print(kth_smallest(matrix, k))
```

Time Complexity: O(k log min(n, k))
Space Complexity: O(min(n, k))

16. Majority Element in an Array

python

```python
def majority_element(nums):
    count = 0
    candidate = None

    for num in nums:
        if count == 0:
            candidate = num

        count += 1 if num == candidate else -1

    return candidate

nums = [2, 2, 1, 1, 1, 2, 2]
print(majority_element(nums))
```

Time Complexity: O(n)

Space Complexity: O(1)

17. Trapping Rain Water

python

```python
def trap(height):
    n = len(height)
    if n <= 2:
        return 0

    left, right = 0, n - 1
    left_max, right_max = height[left], height[right]
    water = 0

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

    return water

height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
print(trap(height))
```

Time Complexity: O(n)

Space Complexity: O(1)

18. Maximum XOR of Two Numbers in an Array

```python
def find_maximum_xor(nums):
    max_xor = 0
    mask = 0

    for i in range(31, -1, -1):
        mask |= (1 << i)
        prefixes = set()

        for num in nums:
            prefixes.add(num & mask)

        potential_max = max_xor | (1 << i)

        for prefix in prefixes:
            if (prefix ^ potential_max) in prefixes:
                max_xor = potential_max
                break

    return max_xor

nums = [3, 10, 5, 25, 2, 8]
print(find_maximum_xor(nums))
```

Time Complexity: O(n)
Space Complexity: O(n)

19. Maximum Product Subarray

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

    max_so_far = min_so_far = result = nums[0]

    for i in range(1, len(nums)):
        temp = max_so_far
        max_so_far = max(nums[i], max(max_so_far * nums[i], min_so_far * nums[i]))
        min_so_far = min(nums[i], min(temp * nums[i], min_so_far * nums[i]))
        result = max(result, max_so_far)

    return result

nums = [2, 3, -2, 4]
print(max_product(nums))
```

Time Complexity: O(n)
Space Complexity: O(1)

20. Count Numbers with Unique Digits

```python
def count_numbers_with_unique_digits(n):
    if n == 0:
        return 1

    count = 10
    unique_digits = 9
    available_digits = 9

    for i in range(2, min(n + 1, 11)):
        unique_digits *= available_digits
        count += unique_digits
        available_digits -= 1

    return count

n = 2
print(count_numbers_with_unique_digits(n))
```

Time Complexity: O(min(n, 10))
Space Complexity: O(1)

21. Count 1s in Binary Representation from 0 to n

```python
def count_bits(n):
    result = [0] * (n + 1)

    for i in range(1, n + 1):
        result[i] = result[i & (i - 1)] + 1

    return result

n = 5
print(count_bits(n))
```

Time Complexity: O(n)
Space Complexity: O(n)

22. Check if a Number is a Power of Two

```python
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

n = 16
print(is_power_of_two(n))
```

Time Complexity: O(1)
Space Complexity: O(1)

23. Maximum XOR of Two Numbers in an Array

```python
def find_maximum_xor(nums):
    max_xor = 0
    mask = 0

    for i in range(31, -1, -1):
        mask |= (1 << i)
        prefixes = set()

        for num in nums:
            prefixes.add(num & mask)

        potential_max = max_xor | (1 << i)

        for prefix in prefixes:
            if (prefix ^ potential_max) in prefixes:
                max_xor = potential_max
                break

    return max_xor

nums = [3, 10, 5, 25, 2, 8]
print(find_maximum_xor(nums))
```

Time Complexity: O(n)
Space Complexity: O(n)

24. Bit Manipulation and Its Advantages

Bit manipulation involves directly manipulating individual bits in a binary representation of data.

Advantages in algorithm design:

1. Efficiency: Bit operations are typically faster than arithmetic operations
2. Memory usage: Representing multiple boolean values as bits in an integer saves space
3. Simplicity: Complex algorithms can sometimes be simplified using bit manipulations
4. Unique solutions: Some problems are naturally suited for bit manipulation

Common bit manipulation operations include:

- Setting bits: OR operation (|)
- Clearing bits: AND operation with complement (&)
- Toggling bits: XOR operation (^)
- Checking bits: AND operation (&)
- Power of two: x & (x-1) == 0
- Isolating rightmost bit: x & -x

25. Next Greater Element in an Array

```python
def next_greater_element(nums):
    n = len(nums)
    result = [-1] * n
    stack = []

    for i in range(n):
        while stack and nums[stack[-1]] < nums[i]:
            result[stack.pop()] = nums[i]
        stack.append(i)

    return result

nums = [4, 5, 2, 25]
print(next_greater_element(nums))
```

Time Complexity: O(n)
Space Complexity: O(n)

26. Remove Nth Node From End of Linked List

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

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

    first = dummy
    second = dummy

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

    while first:
        first = first.next
        second = second.next

    second.next = second.next.next

    return dummy.next

def create_linked_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for i in range(1, len(arr)):
        current.next = ListNode(arr[i])
        current = current.next
    return head

def print_linked_list(head):
    values = []
    while head:
        values.append(str(head.val))
        head = head.next
    return "->".join(values)

head = create_linked_list([1, 2, 3, 4, 5])
n = 2
result = remove_nth_from_end(head, n)
print(print_linked_list(result))
```

Time Complexity: O(n)
Space Complexity: O(1)

27. Find Intersection of Two Linked Lists

python

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

def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None

    ptrA, ptrB = headA, headB

    while ptrA != ptrB:
        ptrA = headB if ptrA is None else ptrA.next
        ptrB = headA if ptrB is None else ptrB.next

    return ptrA

def create_linked_list(arr):
    if not arr:
        return None
    head = ListNode(arr[0])
    current = head
    for i in range(1, len(arr)):
        current.next = ListNode(arr[i])
        current = current.next
    return head

intersect = create_linked_list([8, 4, 5])
headA = create_linked_list([4, 1])
headB = create_linked_list([5, 6, 1])

tempA, tempB = headA, headB
while tempA.next:
    tempA = tempA.next
tempA.next = intersect
while tempB.next:
    tempB = tempB.next
tempB.next = intersect

result = get_intersection_node(headA, headB)
print(result.val if result else "No intersection")
```

Time Complexity: O(m + n)
Space Complexity: O(1)

28. Implementing Two Stacks in a Single Array

python

```python
class TwoStacks:
    def __init__(self, n):
        self.size = n
        self.arr = [0] * n
        self.top1 = -1
        self.top2 = n

    def push1(self, x):
        if self.top1 < self.top2 - 1:
            self.top1 += 1
            self.arr[self.top1] = x
            return True
        return False

    def push2(self, x):
        if self.top1 < self.top2 - 1:
            self.top2 -= 1
            self.arr[self.top2] = x
            return True
        return False

    def pop1(self):
        if self.top1 >= 0:
            x = self.arr[self.top1]
            self.top1 -= 1
            return x
        return -1

    def pop2(self):
        if self.top2 < self.size:
            x = self.arr[self.top2]
            self.top2 += 1
            return x
        return -1

ts = TwoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)
print(ts.pop1())
print(ts.pop2())
```

Time Complexity: O(1) for all operations

Space Complexity: O(n)

29. Check if Integer is a Palindrome Without Converting to String

python

```python
def is_palindrome(x):
    if x < 0:
        return False

    original = x
    reversed_num = 0

    while x > 0:
        reversed_num = reversed_num * 10 + x % 10
        x //= 10

    return original == reversed_num

x = 121
print(is_palindrome(x))
```

Time Complexity: O(log n) where n is the input number
Space Complexity: O(1)

30. Concept of Linked Lists and Their Applications

Linked lists are linear data structures consisting of nodes, where each node contains data and a reference to the next node.

Applications of linked lists in algorithm design:

1. Implementing dynamic data structures like stacks, queues, and hash tables
2. Memory management and garbage collection
3. Browser history implementation
4. Undo functionality in applications
5. Circular buffers and round-robin scheduling
6. Implementation of graphs for adjacency lists
7. Polynomial manipulation
8. Implementing LRU caches

31. Maximum in Every Sliding Window of Size K Using Deque

python

```python
from collections import deque

def max_sliding_window(nums, k):
    n = len(nums)
    if n == 0 or k == 0:
        return []
    if k == 1:
        return nums

    result = []
    dq = deque()

    for i in range(n):
        while dq and dq[0] < i - k + 1:
            dq.popleft()

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

        dq.append(i)

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

    return result

nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
print(max_sliding_window(nums, k))
```

Time Complexity: O(n)
Space Complexity: O(k)

32. Largest Rectangle in Histogram

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

    while i < len(heights):
        if not stack or heights[stack[-1]] <= heights[i]:
            stack.append(i)
            i += 1
        else:
            top = stack.pop()
            area = heights[top] * (i - stack[-1] - 1 if stack else i)
            max_area = max(max_area, area)

    while stack:
        top = stack.pop()
        area = heights[top] * (i - stack[-1] - 1 if stack else i)
        max_area = max(max_area, area)

    return max_area

heights = [2, 1, 5, 6, 2, 3]
print(largest_rectangle_area(heights))
```

Time Complexity: O(n)
Space Complexity: O(n)

33. Sliding Window Technique and Its Applications in Array Problems

The sliding window technique uses two pointers to create a window that slides through an array to process contiguous subarrays. This technique is efficient for array problems that require finding subarrays satisfying certain conditions.

Applications in array problems:

- Finding maximum/minimum sum subarray of a given size
- Finding smallest subarray with sum greater than a given value
- Finding longest subarray with K distinct elements
- Finding the maximum number of fruits of 2 types
- Finding the longest subarray with ones after replacement

34. Subarray Sum Equal to K Using Hashing

```python
def subarray_sum(nums, k):
    count = 0
    sum_so_far = 0
    prefix_sums = {0: 1}

    for num in nums:
        sum_so_far += num
        if sum_so_far - k in prefix_sums:
            count += prefix_sums[sum_so_far - k]

        prefix_sums[sum_so_far] = prefix_sums.get(sum_so_far, 0) + 1

    return count

nums = [1, 1, 1]
k = 2
print(subarray_sum(nums, k))
```

Time Complexity: O(n)
Space Complexity: O(n)

35. K Most Frequent Elements Using Priority Queue

```python
import heapq
from collections import Counter

def top_k_frequent(nums, k):
    counts = Counter(nums)
    heap = []

    for num, count in counts.items():
        heapq.heappush(heap, (count, num))
        if len(heap) > k:
            heapq.heappop(heap)

    return [num for count, num in sorted(heap, reverse=True)]

nums = [1, 1, 1, 2, 2, 3]
k = 2
print(top_k_frequent(nums, k))
```

Time Complexity: O(n log k)
Space Complexity: O(n + k)

36. Generate All Subsets of a Given Array

```python
def subsets(nums):
    result = []

    def backtrack(start, current):
        result.append(current[:])

        for i in range(start, len(nums)):
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop()

    backtrack(0, [])
    return result

nums = [1, 2, 3]
print(subsets(nums))
```

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

37. All Unique Combinations That Sum to Target

```python
def combination_sum(candidates, target):
    result = []

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

        for i in range(start, len(candidates)):
            if candidates[i] > remaining:
                continue

            current.append(candidates[i])
            backtrack(i, current, remaining - candidates[i])
            current.pop()

    candidates.sort()
    backtrack(0, [], target)
    return result

candidates = [2, 3, 6, 7]
target = 7
print(combination_sum(candidates, target))
```

Time Complexity: O(2^n * k) where k is the average length of each combination
Space Complexity: O(target)

38. Generate All Permutations of a Given Array

```python
def permute(nums):
    result = []

    def backtrack(start):
        if start == len(nums):
            result.append(nums[:])
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start]
```