Advanced Algorithmic Problem Solving Solutions (Continued)

38. Generate All Permutations of a Given Array (continued)

python

```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]

    backtrack(0)
    return result

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

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

39. Difference Between Subsets and Permutations

Subsets and permutations are fundamentally different combinatorial concepts:

- **Subsets** are selections of elements where order doesn't matter. A set with n elements has 2^n possible subsets. Example: For set {1, 2, 3}, subsets include {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}.
- **Permutations** are arrangements of elements where order matters. A set with n elements has n! possible permutations. Example: For set {1, 2, 3}, permutations include [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1].

40. Element with Maximum Frequency in an Array

python

```python
from collections import Counter

def max_frequency_element(arr):
    count = Counter(arr)
    max_freq = max(count.values())

    for num, freq in count.items():
        if freq == max_freq:
            return num

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

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

41. Maximum Subarray Sum using Kadane's Algorithm

```python
def max_subarray_sum(nums):
    max_so_far = nums[0]
    current_max = nums[0]

    for i in range(1, len(nums)):
        current_max = max(nums[i], current_max + nums[i])
        max_so_far = max(max_so_far, current_max)

    return max_so_far

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(nums))
```

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

42. Dynamic Programming and Maximum Subarray Problem

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It solves each subproblem only once and stores the results to avoid redundant calculations.

For the maximum subarray problem:

1. We define a state: `dp[i]` represents the maximum sum ending at index i
2. Transition function: `dp[i] = max(nums[i], dp[i-1] + nums[i])`
3. Base case: `dp[0] = nums[0]`
4. Final answer: maximum value in the dp array

The Kadane's algorithm is an efficient implementation of this DP approach that uses constant space by maintaining only the current maximum and global maximum.

43. Top K Frequent Elements

```python
import heapq
from collections import Counter

def top_k_frequent(nums, k):
    count = Counter(nums)
    return [num for num, _ in count.most_common(k)]

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

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

44. Two Sum Problem Using Hashing

```python
def two_sum(nums, target):
    num_dict = {}

    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_dict:
            return [num_dict[complement], i]
        num_dict[num] = i

    return []

nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))
```

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

45. Priority Queues and Their Applications

A priority queue is an abstract data type where each element has a priority associated with it. Elements with higher priorities are served before elements with lower priorities.

Applications in algorithm design:

1. Dijkstra's shortest path algorithm
2. Prim's algorithm for minimum spanning tree
3. Huffman coding for data compression
4. Task scheduling in operating systems
5. Event-driven simulation
6. Finding k largest/smallest elements
7. Median maintenance
8. Implementation of heapsort

46. Longest Palindromic Substring

```python
def longest_palindrome(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_palindrome(s))
```

Time Complexity: O(n^2)
Space Complexity: O(1)

47. Histogram Problems and Their Applications

Histogram problems deal with analyzing distributions of data represented as bars with different heights. These problems often involve finding patterns, areas, or specific features within the histogram.

Applications in algorithm design:

1. Image processing and computer vision
2. Data compression
3. Statistical analysis
4. Finding maximum rectangular area in binary matrices
5. Stock market analysis for finding maximum profit
6. Storage allocation problems
7. Water trapping problems
8. Building skyline problems

48. Next Permutation

```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)

49. 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)

50. Equilibrium Index and Its Applications

An equilibrium index of an array is an index such that the sum of elements at lower indices is equal to the sum of elements at higher indices.

Applications in array problems:

1. Finding pivot points in data sets
2. Load balancing algorithms
3. Data distribution problems
4. Center of mass calculations
5. Split array problems where equal weight distributions are needed
6. Financial analysis for portfolio balancing

```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)