Solutions to DSA Questions 26-50 (Strings and Linked Lists) For 1-2 Years

Experience Roles at EPAM Compiled on September 26, 2025

Introduction

This document provides detailed solutions for 25 Data Structures and Algorithms (DSA) problems (questions 26 to 50) from the Strings and Linked Lists categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity and ease of understanding. These problems cover fundamental to intermediate concepts frequently tested in technical interviews.

Contents

1 Check if a String is a Palindrome **4**
- 1.1 Problem Statement . . . 4
- 1.2 Dry Run on Test Cases . . . 4
- 1.3 Algorithm . . . 4
- 1.4 Python Solution . . . 5

2 Reverse Words in a String **5**
- 2.1 Problem Statement . . . 5
- 2.2 Dry Run on Test Cases . . . 5
- 2.3 Algorithm . . . 5
- 2.4 Python Solution . . . 5

3 Longest Substring Without Repeating Characters **6**
- 3.1 Problem Statement . . . 6
- 3.2 Dry Run on Test Cases . . . 6
- 3.3 Algorithm . . . 6
- 3.4 Python Solution . . . 6

4 Valid Parentheses **7**
- 4.1 Problem Statement . . . 7
- 4.2 Dry Run on Test Cases . . . 7
- 4.3 Algorithm . . . 7
- 4.4 Python Solution . . . 7

5 Longest Palindromic Substring **7**
- 5.1 Problem Statement . . . 8
- 5.2 Dry Run on Test Cases . . . 8
- 5.3 Algorithm . . . 8
- 5.4 Python Solution . . . 8

6 Generate All Permutations of a String 8
6.1 Problem Statement . . . 9
6.2 Dry Run on Test Cases . . . 9
6.3 Algorithm . . . 9
6.4 Python Solution . . . 9

7 Check if Strings are Rotations of Each Other 9
7.1 Problem Statement . . . 9
7.2 Dry Run on Test Cases . . . 10
7.3 Algorithm . . . 10
7.4 Python Solution . . . 10

8 Find First Non-Repeating Character 10
8.1 Problem Statement . . . 10
8.2 Dry Run on Test Cases . . . 10
8.3 Algorithm . . . 10
8.4 Python Solution . . . 11

9 String to Integer (atoi) 11
9.1 Problem Statement . . . 11
9.2 Dry Run on Test Cases . . . 11
9.3 Algorithm . . . 11
9.4 Python Solution . . . 12

10 Longest Common Prefix 12
10.1 Problem Statement . . . 12
10.2 Dry Run on Test Cases . . . 12
10.3 Algorithm . . . 12
10.4 Python Solution . . . 13

11 Group Anagrams 13
11.1 Problem Statement . . . 13
11.2 Dry Run on Test Cases . . . 13
11.3 Algorithm . . . 13
11.4 Python Solution . . . 13

12 Valid IP Address 14
12.1 Problem Statement . . . 14
12.2 Dry Run on Test Cases . . . 14
12.3 Algorithm . . . 14
12.4 Python Solution . . . 14

13 Edit Distance 15
13.1 Problem Statement . . . 15
13.2 Dry Run on Test Cases . . . 15
13.3 Algorithm (Memoization) . . . 15
13.4 Python Solution (Memoization) . . . 15
13.5 Python Solution (Tabulation) . . . 16

14 Smallest Window Containing All Characters 16

14.1 Problem Statement 16
14.2 Dry Run on Test Cases 16
14.3 Algorithm 17
14.4 Python Solution 17

15 Longest Increasing Subsequence in String 18
15.1 Problem Statement 18
15.2 Dry Run on Test Cases 18
15.3 Algorithm (Memoization) 18
15.4 Python Solution (Memoization) 18
15.5 Python Solution (Tabulation) 19

16 Check for Valid Shuffle of Two Strings 19
16.1 Problem Statement 19
16.2 Dry Run on Test Cases 19
16.3 Algorithm 19
16.4 Python Solution 20

17 Remove Duplicate Letters 20
17.1 Problem Statement 20
17.2 Dry Run on Test Cases 20
17.3 Algorithm 20
17.4 Python Solution 20

18 Find All Palindromic Substrings 21
18.1 Problem Statement 21
18.2 Dry Run on Test Cases 21
18.3 Algorithm 21
18.4 Python Solution 21

19 Rabin-Karp String Matching 22
19.1 Problem Statement 22
19.2 Dry Run on Test Cases 22
19.3 Algorithm 22
19.4 Python Solution 22

20 KMP Algorithm for Pattern Searching 23
20.1 Problem Statement 23
20.2 Dry Run on Test Cases 23
20.3 Algorithm 23
20.4 Python Solution 23

21 Reverse a Linked List 24
21.1 Problem Statement 24
21.2 Dry Run on Test Cases 25
21.3 Algorithm 25
21.4 Python Solution 25

22 Detect Cycle in a Linked List 25
22.1 Problem Statement 25

22.2 Dry Run on Test Cases 25
22.3 Algorithm 26
22.4 Python Solution 26

23 Merge Two Sorted Linked Lists **26**
23.1 Problem Statement 26
23.2 Dry Run on Test Cases 26
23.3 Algorithm 27
23.4 Python Solution 27

24 Remove Nth Node from End **27**
24.1 Problem Statement 27
24.2 Dry Run on Test Cases 27
24.3 Algorithm 28
24.4 Python Solution 28

25 Find Middle of Linked List **28**
25.1 Problem Statement 28
25.2 Dry Run on Test Cases 28
25.3 Algorithm 28
25.4 Python Solution 29

1 Check if a String is a Palindrome

1.1 Problem Statement

Given a string, determine if it is a palindrome (reads the same forward and backward), considering only alphanumeric characters and ignoring cases.

1.2 Dry Run on Test Cases

- **Test Case 1**: Input = "A man, a plan, a canal: Panama" → Output: True
- **Test Case 2**: Input = "race a car" → Output: False
- **Test Case 3**: Input = "" → Output: True
- **Test Case 4**: Input = "0P" → Output: False

1.3 Algorithm

1. Convert string to lowercase and filter alphanumeric characters.
2. Use two pointers: left from start, right from end.
3. Compare characters; if mismatch, return False.
4. If pointers meet, return True.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ or $O(n)$ if new string created

1.4 Python Solution

```python
def is_palindrome(s):
    # Filter alphanumeric and convert to lowercase
    filtered = ''.join(c.lower() for c in s if c.isalnum())
    left, right = 0, len(filtered) - 1

    while left < right:
        if filtered[left] != filtered[right]:
            return False
        left += 1
        right -= 1
    return True

# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))  # Output:
    True
```

2 Reverse Words in a String

2.1 Problem Statement

Given a string, reverse the order of words, removing extra spaces.

2.2 Dry Run on Test Cases

- **Test Case 1**: Input = "the sky is blue" → Output: "blue is sky the"
- **Test Case 2**: Input = " hello world " → Output: "world hello"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

2.3 Algorithm

1. Split string into words, filter out empty strings.
2. Reverse the list of words.
3. Join words with single space.

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

2.4 Python Solution

```python
def reverse_words(s):
    words = [word for word in s.split() if word]
```

```python
    words.reverse()
    return ' '.join(words)

# Example usage
print(reverse_words("the sky is blue"))  # Output: "blue is sky
    the"
```

3 Longest Substring Without Repeating Characters

3.1 Problem Statement

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

3.2 Dry Run on Test Cases

- **Test Case 1**: Input = "abcabcbb" $\rightarrow$ Output: 3 ("abc")
- **Test Case 2**: Input = "bbbbb" $\rightarrow$ Output: 1 ("b")
- **Test Case 3**: Input = "pwwkew" $\rightarrow$ Output: 3 ("wke")
- **Test Case 4**: Input = "" $\rightarrow$ Output: 0

3.3 Algorithm

1. Use sliding window with hashmap to store last seen index of characters.
2. Move right pointer, update max length.
3. If character repeats, move left pointer to last seen + 1.

Time Complexity: $O(n)$ **Space Complexity**: $O(\min(m, n))$

3.4 Python Solution

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

    for right, char in enumerate(s):
        if char in char_index and char_index[char] >= left:
            left = char_index[char] + 1
        else:
            max_length = max(max_length, right - left + 1)
        char_index[char] = right
    return max_length

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

4 Valid Parentheses

4.1 Problem Statement

Given a string containing only '(', ')', '', '', '[', ']', determine if it is valid (matching pairs).

4.2 Dry Run on Test Cases

- **Test Case 1**: Input = "()" $\rightarrow$ Output: True
- **Test Case 2**: Input = "()[]" $\rightarrow$ Output: True
- **Test Case 3**: Input = "(]" $\rightarrow$ Output: False
- **Test Case 4**: Input = "([)]" $\rightarrow$ Output: False

4.3 Algorithm

1. Use a stack to track opening brackets.
2. For each character:
   - If opening, push to stack.
   - If closing, check if matches stack top; pop if match, else return False.
3. Return True if stack empty.

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

4.4 Python Solution

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

    for char in s:
        if char in brackets.values():
            stack.append(char)
        elif char in brackets:
            if not stack or stack.pop() != brackets[char]:
                return False
    return len(stack) == 0

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

5 Longest Palindromic Substring

5.1 Problem Statement

Given a string, find the longest substring that is a palindrome.

5.2 Dry Run on Test Cases

- **Test Case 1**: Input = "babad" → Output: "bab" or "aba"
- **Test Case 2**: Input = "cbbd" → Output: "bb"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

5.3 Algorithm

1. For each index, expand around center for odd and even length palindromes.
2. Track max length and substring.
3. Return longest palindrome found.

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

5.4 Python Solution

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

    start, end = 0, 0
    for i in range(len(s)):
        left1, right1 = expand_around_center(i, i)  # Odd length
        left2, right2 = expand_around_center(i, i + 1)  # Even
           length
        if right1 - left1 > end - start:
            start, end = left1, right1
        if right2 - left2 > end - start:
            start, end = left2, right2
    return s[start:end + 1]

# Example usage
print(longest_palindrome("babad"))  # Output: "bab" or "aba"
```

6 Generate All Permutations of a String

6.1 Problem Statement

Given a string, return all possible permutations.

6.2 Dry Run on Test Cases

- **Test Case 1**: Input = "abc" → Output: ["abc", "acb", "bac", "bca", "cab", "cba"]
- **Test Case 2**: Input = "a" → Output: ["a"]
- **Test Case 3**: Input = "" → Output: []
- **Test Case 4**: Input = "aa" → Output: ["aa"]

6.3 Algorithm

1. Use backtracking: swap characters at each position.
2. Recurse to generate permutations for remaining characters.
3. Collect all permutations in result.

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

6.4 Python Solution

```python
def permute(s):
    def backtrack(arr, start, result):
        if start == len(arr):
            result.append(''.join(arr))
        for i in range(start, len(arr)):
            arr[start], arr[i] = arr[i], arr[start]
            backtrack(arr, start + 1, result)
            arr[start], arr[i] = arr[i], arr[start]

    result = []
    backtrack(list(s), 0, result)
    return result

# Example usage
print(permute("abc"))  # Output: ["abc", "acb", "bac", "bca", "
    cab", "cba"]
```

7 Check if Strings are Rotations of Each Other

7.1 Problem Statement

Given two strings, check if one is a rotation of the other.

7.2 Dry Run on Test Cases

- **Test Case 1**: s1 = "abcde", s2 = "cdeab" → Output: True
- **Test Case 2**: s1 = "abcde", s2 = "abced" → Output: False
- **Test Case 3**: s1 = "", s2 = "" → Output: True
- **Test Case 4**: s1 = "a", s2 = "a" → Output: True

7.3 Algorithm

1. Check if lengths are equal; if not, return False.
2. Concatenate s1 with itself.
3. Check if s2 is a substring of s1 + s1.

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

7.4 Python Solution

```python
def are_rotations(s1, s2):
    if len(s1) != len(s2):
        return False
    if not s1 and not s2:
        return True
    return s2 in (s1 + s1)

# Example usage
print(are_rotations("abcde", "cdeab"))  # Output: True
```

8 Find First Non-Repeating Character

8.1 Problem Statement

Given a string, find the index of the first non-repeating character.

8.2 Dry Run on Test Cases

- **Test Case 1**: Input = "leetcode" → Output: 0 ('l')
- **Test Case 2**: Input = "loveleetcode" → Output: 2 ('v')
- **Test Case 3**: Input = "aabb" → Output: -1
- **Test Case 4**: Input = "" → Output: -1

8.3 Algorithm

1. Use hashmap to count character frequencies.

2. Iterate string again to find first character with count 1.
3. Return its index or -1 if none.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ (26 chars max)

8.4 Python Solution

```python
def first_non_repeating(s):
    count = {}
    for char in s:
        count[char] = count.get(char, 0) + 1

    for i, char in enumerate(s):
        if count[char] == 1:
            return i
    return -1

# Example usage
print(first_non_repeating("leetcode"))  # Output: 0
```

9 String to Integer (atoi)

9.1 Problem Statement

Convert a string to a 32-bit signed integer, handling whitespace, signs, and overflow.

9.2 Dry Run on Test Cases

- **Test Case 1**: Input = "42" → Output: 42
- **Test Case 2**: Input = " -42" → Output: -42
- **Test Case 3**: Input = "4193 with words" → Output: 4193
- **Test Case 4**: Input = "2147483648" → Output: 2147483647

9.3 Algorithm

1. Strip leading whitespace.
2. Check sign (+ or -).
3. Build number digit by digit, check for overflow.
4. Return number or clamp to 32-bit range.

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

9.4 Python Solution

```python
def atoi(s):
    s = s.strip()
    if not s:
        return 0

    sign = 1
    i = 0
    if s[0] in ['+', '-']:
        sign = -1 if s[0] == '-' else 1
        i += 1

    result = 0
    while i < len(s) and s[i].isdigit():
        result = result * 10 + int(s[i])
        if result * sign > 2**31 - 1:
            return 2**31 - 1
        if result * sign < -2**31:
            return -2**31
        i += 1
    return result * sign

# Example usage
print(atoi("   -42"))  # Output: -42
```

10 Longest Common Prefix

10.1 Problem Statement

Given an array of strings, find the longest common prefix among them.

10.2 Dry Run on Test Cases

- **Test Case 1**: Input = ["flower", "flow", "flight"] → Output: "fl"
- **Test Case 2**: Input = ["dog", "racecar", "car"] → Output: ""
- **Test Case 3**: Input = ["interspecies", "interstellar"] → Output: "inter"
- **Test Case 4**: Input = ["a"] → Output: "a"

10.3 Algorithm

1. If empty array, return "".
2. Take first string as prefix.
3. For each string, reduce prefix while it doesn't match.

Time Complexity: $O(S)$ (S = total characters) **Space Complexity**: $O(1)$

10.4 Python Solution

```python
def longest_common_prefix(strs):
    if not strs:
        return ""
    prefix = strs[0]

    for s in strs[1:]:
        while s[:len(prefix)] != prefix:
            prefix = prefix[:-1]
            if not prefix:
                return ""
    return prefix

# Example usage
print(longest_common_prefix(["flower", "flow", "flight"]))  # Output: "fl"
```

11 Group Anagrams

11.1 Problem Statement

Given an array of strings, group anagrams together.

11.2 Dry Run on Test Cases

- **Test Case 1**: Input = ["eat", "tea", "tan", "ate", "nat", "bat"] → Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
- **Test Case 2**: Input = [""] → Output: [[""]]
- **Test Case 3**: Input = ["a"] → Output: [["a"]]
- **Test Case 4**: Input = [] → Output: []

11.3 Algorithm

1. Use hashmap with sorted string as key, list of strings as value.
2. For each string, sort and add to map.
3. Return map values.

Time Complexity: $O(n \cdot k \log k)$ (k = max string length) **Space Complexity**: $O(n \cdot k)$

11.4 Python Solution

```python
def group_anagrams(strs):
    anagrams = {}
    for s in strs:
```

```
        key = ''.join(sorted(s))
        anagrams[key] = anagrams.get(key, []) + [s]
    return list(anagrams.values())

# Example usage
print(group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"]))
```

12 Valid IP Address

12.1 Problem Statement

Given a string, determine if it is a valid IPv4 address.

12.2 Dry Run on Test Cases

- **Test Case 1**: Input = "192.168.1.1" → Output: True
- **Test Case 2**: Input = "192.168.01.1" → Output: False
- **Test Case 3**: Input = "256.1.2.3" → Output: False
- **Test Case 4**: Input = "1.2.3" → Output: False

12.3 Algorithm

1. Split string by '.' and check for 4 parts.
2. For each part:
   - Check length, leading zeros, and range (0-255).
   - Ensure only digits.
3. Return True if all valid.

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

12.4 Python Solution

```
def valid_ip_address(s):
    parts = s.split('.')
    if len(parts) != 4:
        return False

    for part in parts:
        if not part or (part[0] == '0' and len(part) > 1) or not part.isdigit():
            return False
        num = int(part)
        if num < 0 or num > 255:
```

```python
            return False
    return True

# Example usage
print(valid_ip_address("192.168.1.1"))  # Output: True
```

13 Edit Distance

13.1 Problem Statement

Given two strings, find minimum operations (insert, delete, replace) to convert one to another.

13.2 Dry Run on Test Cases

- **Test Case 1**: word1 = "horse", word2 = "ros" → Output: 3
- **Test Case 2**: word1 = "intention", word2 = "execution" → Output: 5
- **Test Case 3**: word1 = "", word2 = "abc" → Output: 3
- **Test Case 4**: word1 = "a", word2 = "a" → Output: 0

13.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. If strings empty, return length of other.
3. If characters match, recurse on rest.
4. Else, take min of insert, delete, replace.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

13.4 Python Solution (Memoization)

```python
def edit_distance(word1, word2):
    memo = {}

    def dp(i, j):
        if i == 0:
            return j
        if j == 0:
            return i
        if (i, j) in memo:
            return memo[(i, j)]

        if word1[i-1] == word2[j-1]:
            memo[(i, j)] = dp(i-1, j-1)
```

```python
        else:
            memo[(i, j)] = min(
                dp(i-1, j) + 1,   # Delete
                dp(i, j-1) + 1,   # Insert
                dp(i-1, j-1) + 1  # Replace
            )
        return memo[(i, j)]

    return dp(len(word1), len(word2))

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

13.5 Python Solution (Tabulation)

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

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

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

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

14 Smallest Window Containing All Characters

14.1 Problem Statement

Given two strings s and t, find the smallest window in s containing all characters of t.

14.2 Dry Run on Test Cases

- **Test Case 1**: s = "ADOBECODEBANC", t = "ABC" → Output: "BANC"

- **Test Case 2**: s = "a", t = "a" → Output: "a"
- **Test Case 3**: s = "a", t = "aa" → Output: ""
- **Test Case 4**: s = "abc", t = "d" → Output: ""

14.3 Algorithm

1. Use sliding window with two hashmaps.
2. Move right pointer until window contains all t characters.
3. Shrink left pointer to minimize window.
4. Track smallest window.

Time Complexity: $O(n)$ **Space Complexity**: $O(k)$ (k = charset size)

14.4 Python Solution

```python
from collections import Counter

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

    t_count = Counter(t)
    required = len(t_count)
    formed = 0
    window_counts = {}

    left = right = 0
    min_len = float('inf')
    min_window_substr = ""

    while right < len(s):
        window_counts[s[right]] = window_counts.get(s[right], 0)
            + 1
        if s[right] in t_count and window_counts[s[right]] ==
            t_count[s[right]]:
             formed += 1

        while left <= right and formed == required:
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_window_substr = s[left:right + 1]

            window_counts[s[left]] -= 1
            if s[left] in t_count and window_counts[s[left]] <
                t_count[s[left]]:
                 formed -= 1
            left += 1
```

```
        right += 1
    return min_window_substr

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

15 Longest Increasing Subsequence in String

15.1 Problem Statement

Given a string, find the length of the longest increasing subsequence of characters.

15.2 Dry Run on Test Cases

- **Test Case 1**: Input = "aebbcg" → Output: 3 ("abc")
- **Test Case 2**: Input = "abcde" → Output: 5
- **Test Case 3**: Input = "a" → Output: 1
- **Test Case 4**: Input = "" → Output: 0

15.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. For each index, consider including character if greater than previous.
3. Return max length.

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

15.4 Python Solution (Memoization)

```
def longest_increasing_subsequence(s):
    memo = {}

    def lis(index, prev_char):
        if index == len(s):
            return 0
        if (index, prev_char) in memo:
            return memo[(index, prev_char)]

        not_take = lis(index + 1, prev_char)
        take = 0
        if prev_char < s[index]:
            take = 1 + lis(index + 1, s[index])

        memo[(index, prev_char)] = max(take, not_take)
        return memo[(index, prev_char)]
```

```python

    return lis(0, chr(0))

# Example usage
print(longest_increasing_subsequence("aebbcg"))  # Output: 3
```

15.5 Python Solution (Tabulation)

```python
def longest_increasing_subsequence_tab(s):
    if not s:
        return 0

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

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

# Example usage
print(longest_increasing_subsequence_tab("aebbcg"))  # Output: 3
```

16 Check for Valid Shuffle of Two Strings

16.1 Problem Statement

Given strings s1, s2, and result, check if result is a valid shuffle of s1 and s2.

16.2 Dry Run on Test Cases

- **Test Case 1**: s1 = "abc", s2 = "def", result = "adbcef" → Output: True
- **Test Case 2**: s1 = "abc", s2 = "def", result = "abcdefg" → Output: False
- **Test Case 3**: s1 = "", s2 = "", result = "" → Output: True
- **Test Case 4**: s1 = "a", s2 = "b", result = "ba" → Output: True

16.3 Algorithm

1. Check if len(result) = len(s1) + len(s2).
2. Use two pointers for s1 and s2, one for result.
3. Match characters; if no match, return False.

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

16.4 Python Solution

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

    i = j = k = 0
    while k < len(result):
        if i < len(s1) and s1[i] == result[k]:
            i += 1
        elif j < len(s2) and s2[j] == result[k]:
            j += 1
        else:
            return False
        k += 1
    return i == len(s1) and j == len(s2)

# Example usage
print(is_valid_shuffle("abc", "def", "adbcef"))  # Output: True
```

17 Remove Duplicate Letters

17.1 Problem Statement

Given a string, remove duplicate letters so each letter appears once, in smallest lexicographical order.

17.2 Dry Run on Test Cases

- **Test Case 1**: Input = "bcabc" → Output: "abc"
- **Test Case 2**: Input = "cbacdcbc" → Output: "acdb"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

17.3 Algorithm

1. Track last occurrence of each character.
2. Use stack to build result, ensuring lexicographical order.
3. Pop from stack if current char is smaller and later occurrences exist.

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

17.4 Python Solution

```python
def remove_duplicate_letters(s):
    last_occurrence = {}
```

```python
    for i, char in enumerate(s):
        last_occurrence[char] = i

    stack = []
    seen = set()

    for i, char in enumerate(s):
        if char not in seen:
            while stack and char < stack[-1] and i <
               last_occurrence[stack[-1]]:
                seen.remove(stack.pop())
            stack.append(char)
            seen.add(char)
    return ''.join(stack)

# Example usage
print(remove_duplicate_letters("cbacdcbc"))  # Output: "acdb"
```

18 Find All Palindromic Substrings

18.1 Problem Statement

Given a string, find the count of all palindromic substrings.

18.2 Dry Run on Test Cases

- **Test Case 1**: Input = "aaa" → Output: 6 ("a", "a", "a", "aa", "aa", "aaa")
- **Test Case 2**: Input = "abc" → Output: 3 ("a", "b", "c")
- **Test Case 3**: Input = "" → Output: 0
- **Test Case 4**: Input = "aba" → Output: 4 ("a", "b", "a", "aba")

18.3 Algorithm

1. For each index, expand around center for odd and even palindromes.
2. Count all valid palindromes.

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

18.4 Python Solution

```python
def count_palindromic_substrings(s):
    def expand_around_center(left, right):
        count = 0
        while left >= 0 and right < len(s) and s[left] == s[right
           ]:
            count += 1
```

```python
            left -= 1
            right += 1
        return count

    total = 0
    for i in range(len(s)):
        total += expand_around_center(i, i)  # Odd length
        total += expand_around_center(i, i + 1)  # Even length
    return total

# Example usage
print(count_palindromic_substrings("aaa"))  # Output: 6
```

19 Rabin-Karp String Matching

19.1 Problem Statement

Given a text and pattern, find all occurrences of pattern in text using Rabin-Karp.

19.2 Dry Run on Test Cases

- **Test Case 1**: text = "AABAACAADA", pattern = "AA" → Output: [0, 3, 6]
- **Test Case 2**: text = "abcd", pattern = "xyz" → Output: []
- **Test Case 3**: text = "", pattern = "a" → Output: []
- **Test Case 4**: text = "aaa", pattern = "aaa" → Output: [0]

19.3 Algorithm

1. Compute hash of pattern and first window of text.
2. Slide window, update hash, compare if equal.
3. Verify matches to avoid hash collisions.

Time Complexity: $O(n + m)$ average **Space Complexity**: $O(1)$

19.4 Python Solution

```python
def rabin_karp(text, pattern):
    if not pattern or not text:
        return []

    d = 256  # Number of characters
    q = 101  # Prime number
    m, n = len(pattern), len(text)
    result = []

```

```
    h = pow(d, m-1) % q
    p = t = 0

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for i in range(n - m + 1):
        if p == t:
            if text[i:i+m] == pattern:
                result.append(i)
        if i < n - m:
            t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) %
                q
            if t < 0:
                t += q
    return result

# Example usage
print(rabin_karp("AABAACAADA", "AA"))  # Output: [0, 3, 6]
```

20 KMP Algorithm for Pattern Searching

20.1 Problem Statement

Given a text and pattern, find all occurrences of pattern in text using KMP algorithm.

20.2 Dry Run on Test Cases

- **Test Case 1**: text = "AABAACAADA", pattern = "AA" → Output: [0, 3, 6]
- **Test Case 2**: text = "abcd", pattern = "xyz" → Output: []
- **Test Case 3**: text = "", pattern = "a" → Output: []
- **Test Case 4**: text = "aaa", pattern = "aaa" → Output: [0]

20.3 Algorithm

1. Compute LPS (longest prefix suffix) array for pattern.
2. Use LPS to skip redundant comparisons while matching.
3. Collect all match indices.

Time Complexity: $O(n + m)$ **Space Complexity**: $O(m)$

20.4 Python Solution

```python
def kmp_search(text, pattern):
    def compute_lps(pattern):
        m = len(pattern)
        lps = [0] * m
        length = 0
        i = 1
        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

    result = []
    if not pattern or not text:
        return result

    m, n = len(pattern), len(text)
    lps = compute_lps(pattern)
    i = j = 0

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            result.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return result

# Example usage
print(kmp_search("AABAACAADA", "AA"))  # Output: [0, 3, 6]
```

21 Reverse a Linked List

21.1 Problem Statement

Given a singly linked list, reverse it and return the new head.

21.2 Dry Run on Test Cases

- **Test Case 1**: Input = 1->2->3->4->5 → Output: 5->4->3->2->1
- **Test Case 2**: Input = 1 → Output: 1
- **Test Case 3**: Input = None → Output: None
- **Test Case 4**: Input = 1->2 → Output: 2->1

21.3 Algorithm

1. Initialize prev = None, curr = head.
2. While curr, save next, set curr.next = prev, move prev and curr.
3. Return prev as new head.

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

21.4 Python Solution

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

def reverse_list(head):
    prev = None
    curr = head

    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev

# Example usage (simplified)
# head = ListNode(1, ListNode(2, ListNode(3)))
# reversed_head = reverse_list(head)
```

22 Detect Cycle in a Linked List

22.1 Problem Statement

Given a linked list, determine if it has a cycle.

22.2 Dry Run on Test Cases

- **Test Case 1**: 1->2->3->4->2(cycle) → Output: True

- **Test Case 2**: 1->2->3 → Output: False
- **Test Case 3**: None → Output: False
- **Test Case 4**: 1 → Output: False

22.3 Algorithm

1. Use two pointers: slow (1 step), fast (2 steps).
2. If they meet, cycle exists.
3. If fast reaches end, no cycle.

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

22.4 Python Solution

```python
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3)))
# head.next.next.next = head.next  # Creates cycle
# print(has_cycle(head))  # Output: True
```

23 Merge Two Sorted Linked Lists

23.1 Problem Statement

Given two sorted linked lists, merge them into one sorted list.

23.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 1->2->4, l2 = 1->3->4 → Output: 1->1->2->3->4->4
- **Test Case 2**: l1 = None, l2 = None → Output: None
- **Test Case 3**: l1 = 1, l2 = None → Output: 1
- **Test Case 4**: l1 = 2, l2 = 1 → Output: 1->2

23.3 Algorithm

1. Use dummy node to simplify merging.
2. Compare heads of l1 and l2, append smaller to result.
3. Move to next node of chosen list.
4. Append remaining nodes.

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

23.4 Python Solution

```python
def merge_two_lists(l1, l2):
    dummy = ListNode(0)
    curr = dummy

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

    curr.next = l1 if l1 else l2
    return dummy.next

# Example usage
# l1 = ListNode(1, ListNode(2, ListNode(4)))
# l2 = ListNode(1, ListNode(3, ListNode(4)))
# merged = merge_two_lists(l1, l2)
```

24 Remove Nth Node from End

24.1 Problem Statement

Given a linked list and integer n, remove the nth node from the end.

24.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, n = 2 → Output: 1->2->3->5
- **Test Case 2**: head = 1, n = 1 → Output: None
- **Test Case 3**: head = 1->2, n = 2 → Output: 2
- **Test Case 4**: head = None, n = 1 → Output: None

24.3 Algorithm

1. Use two pointers: fast moves n steps ahead.
2. Move slow and fast until fast reaches end.
3. Slow points to node before nth from end; remove it.

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

24.4 Python Solution

```python
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    slow = fast = dummy

    for _ in range(n):
        fast = fast.next

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

    slow.next = slow.next.next
    return dummy.next

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode
    (5)))))
# new_head = remove_nth_from_end(head, 2)
```

25 Find Middle of Linked List

25.1 Problem Statement

Given a linked list, return the middle node (if even, second middle node).

25.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5 → Output: 3
- **Test Case 2**: head = 1->2->3->4 → Output: 3
- **Test Case 3**: head = 1 → Output: 1
- **Test Case 4**: head = None → Output: None

25.3 Algorithm

1. Use two pointers: slow (1 step), fast (2 steps).

2. When fast reaches end, slow is at middle.

3. Return slow.

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

25.4 Python Solution

```python
def middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode
    (5)))))
# middle = middle_node(head)
```