The Importance of a Methodical Approach in Coding Interviews

The primary goal of a coding interview is not just to test your algorithm knowledge. While understanding algorithms is important, it's not the main focus. The primary objective is to assess whether you have a methodical approach to solving problems. This is crucial because software engineering relies on engineers being pragmatic and consistent.

Companies want to see if you can apply a systematic approach when faced with challenging situations. Even if this isn't explicitly stated, it's a trait that companies should be looking for in potential hires.

Instead of relying on a "light bulb moment" to magically find a solution, I suggest you adopt a structured problem-solving template. Here's why having a template is beneficial:

1. **Consistency**: A template ensures that you approach every problem with the same systematic method, reducing the chances of missing crucial steps.
2. **Clarity**: It helps you clearly outline your thought process, making it easier for the interviewer to follow your reasoning and understand your approach.
3. **Efficiency**: By following a template, you can quickly identify the key aspects of the problem and focus on finding a solution without getting sidetracked.
4. **Confidence**: Knowing you have a reliable approach can boost your confidence, allowing you to tackle even the most daunting problems with a clear head.

A Problem-Solving Template for Coding Interviews

1. **Clarify Requirements**:

- **Understand the Problem Prompt**: Carefully read the problem statement to grasp the core issue.

- **Clarify Functional Requirements**: Determine what you're trying to fix or achieve.

- **Verify Technical Requirements**: Understand the input and output data types, constraints, and invalid cases.

- **Data Type Specifications**: If dealing with collections, identify the type of each element.

2. **Come Up with Test Cases**:

- **Write Your Own Test Cases**: Create comprehensive and challenging test cases similar to those found on platforms like LeetCode.

- **Avoid Surprises**: By rigorously testing your code, you can ensure robustness and impress interviewers.

3. **Write Down Observables**:

- **Detective Work**: Note down every observable detail and clue.

- **Intuitive Insights**: Record potential solutions and why certain approaches (e.g., BFS or DFS) may or may not work.

4. **Come Up with Strategies**:

- **Develop Multiple Strategies**: Brainstorm various approaches to solve the problem.

- **Iterate and Update**: Continuously refine your strategies based on new insights from your observables.

- **List Format**: Use a numbered list of concise sentences to outline your strategies, as pseudocode can sometimes be repetitive.

5. **Implement**:

- **Translate to Code**: Convert your strategies and observations into your preferred programming language.

6. **Verify**:

- **Run Against Test Cases**: Test your implementation against the comprehensive test cases you created to ensure accuracy and efficiency.

By following this template, you'll demonstrate to interviewers that you have a structured approach to problem-solving, which is a highly valued skill in software engineering. This methodical process not only helps you tackle complex problems more effectively but also showcases your ability to think critically and logically.

Two Sum Problem Solution

1. Clarify Requirements

- **Problem Statement:** Find two distinct indices in the array whose elements sum up to a given target.
- **Function Output:** Return the indices of the two numbers.

- **Input Data Type:** `nums` (List[int]), `target` (int)
- **Output Data Type:** List[int] (indices)

2. Come Up with Test Cases

- **Test Case 1:**
  - Input: `nums = [2, 7, 11, 15]`, `target = 9`
  - Output: `[0, 1]`
- **Test Case 2:**
  - Input: `nums = [3, 2, 4]`, `target = 6`
  - Output: `[1, 2]`
- **Test Case 3:**
  - Input: `nums = [3, 3]`, `target = 6`
  - Output: `[0, 1]`
- **Test Case 4:**
  - Input: `nums = [2, 5, 5, 11]`, `target = 10`
  - Output: `[1, 2]`

3. Write Down Observables

- **Observations:**
  - Using a nested for loop is not ideal due to high time complexity (O(n^2)).
  - With a single pass, we can keep track of elements using a hash map (dictionary in Python).
  - The hash map will store elements we have seen so far and their indices.
  - For each element, we check if `target - currentElement` exists in the map.
  - If it does, we found the pair. If not, we add the current element to the map.

4. Come Up with Strategy

1. Initialize a hash map (dictionary) to store elements as keys and their indices as values.
2. Iterate through the input array:
   - For each element, check if `target - currentElement` exists in the map.
   - If it exists, return `[map[target - currentElement], currentIndex]`.
   - Otherwise, add `currentElement` and its index to the map.
3. If no such indices exist, return an empty list or handle accordingly.

5. Implement

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

# Test the function with the provided test cases
print(two_sum([2, 7, 11, 15], 9))  # Output: [0, 1]
print(two_sum([3, 2, 4], 6))       # Output: [1, 2]
print(two_sum([3, 3], 6))          # Output: [0, 1]
print(two_sum([2, 5, 5, 11], 10))  # Output: [1, 2]
```

6. Verify

- **Test Case 1:**
  - Input: `nums = [2, 7, 11, 15]`, `target = 9`
  - Expected Output: `[0, 1]`
  - Actual Output: `[0, 1]`
- **Test Case 2:**
  - Input: `nums = [3, 2, 4]`, `target = 6`
  - Expected Output: `[1, 2]`
  - Actual Output: `[1, 2]`
- **Test Case 3:**
  - Input: `nums = [3, 3]`, `target = 6`
  - Expected Output: `[0, 1]`
  - Actual Output: `[0, 1]`
- **Test Case 4:**
  - Input: `nums = [2, 5, 5, 11]`, `target = 10`
  - Expected Output: `[1, 2]`
  - Actual Output: `[1, 2]`

The function works as expected for all the provided test cases.

Algorithm Templates

Benefits of having Algorithm templates.

- **Efficiency and Speed:** Memorized templates save time and reduce errors, allowing you to quickly implement algorithms and focus on problem-specific logic during interviews.
- **Confidence and Consistency:** Templates boost confidence by providing a reliable structure, ensuring all necessary steps are included and demonstrating thorough preparation.
- **Demonstrates Proficiency:** Using templates shows mastery of foundational concepts, adaptability in problem-solving, and a methodical approach, which are highly valued traits in candidates.

1. Breadth-First Search (BFS) Template

```python
from collections import deque

def bfs(start_node):
    queue = deque([start_node])
    visited = set([start_node])

    while queue:
        current_node = queue.popleft()
        # Process current node
        for neighbor in get_neighbors(current_node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

- **Key Points:**
  - Use a queue to explore nodes level by level.
  - Keep track of visited nodes to avoid cycles.

2. Depth-First Search (DFS) Template

Iterative DFS

```python
def dfs_iterative(start_node):
    stack = [start_node]
    visited = set([start_node])

    while stack:
        current_node = stack.pop()
        # Process current node
        for neighbor in get_neighbors(current_node):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(neighbor)
```

- **Key Points:**
  - Use a stack to explore nodes depth-first.
  - Keep track of visited nodes to avoid cycles.

Recursive DFS

```
def dfs_recursive(node, visited=set()):
    if node in visited:
        return
    visited.add(node)
    # Process current node
    for neighbor in get_neighbors(node):
        if neighbor not in visited:
            dfs_recursive(neighbor, visited)
```

- **Key Points:**
  - Use recursion to explore nodes depth-first.
  - Keep track of visited nodes to avoid cycles.

3. Dijkstra's Algorithm Template

```
import heapq

def dijkstra(graph, start):
    pq = [(0, start)]
    distances = {start: 0}
    visited = set()

    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if current_node in visited:
            continue
        visited.add(current_node)

        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight
            if neighbor not in distances or distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances
```

- **Key Points:**
  - Use a priority queue to always expand the shortest known distance.
  - Keep track of distances and visited nodes.

4. Union-Find (Disjoint Set) Template

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [1] * size

    def find(self, node):
        if self.parent[node] != node:
            self.parent[node] = self.find(self.parent[node])  # Path compression
        return self.parent[node]

    def union(self, node1, node2):
        root1 = self.find(node1)
        root2 = self.find(node2)

        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1
```

- **Key Points:**
  - Use path compression for efficient `find` operations.
  - Use union by rank for efficient `union` operations.

How to Study

1. **Depth-First Search (DFS) Approach:**

- **In-Depth Exploration:** Like DFS explores all nodes deeply before backtracking, I dive deep into each topic, thoroughly understanding it before moving on to the next. This ensures a comprehensive grasp of individual subjects.

2. **Sliding Window Technique:**
   - **Progressive Difficulty:** I apply the sliding window technique to manage and adjust the difficulty of topics I study. By gradually increasing the complexity, I build my skills progressively, ensuring a smooth learning curve without overwhelming myself.
3. Do not mistake as leetcode == interview. In fact you should practice interviewing separately.