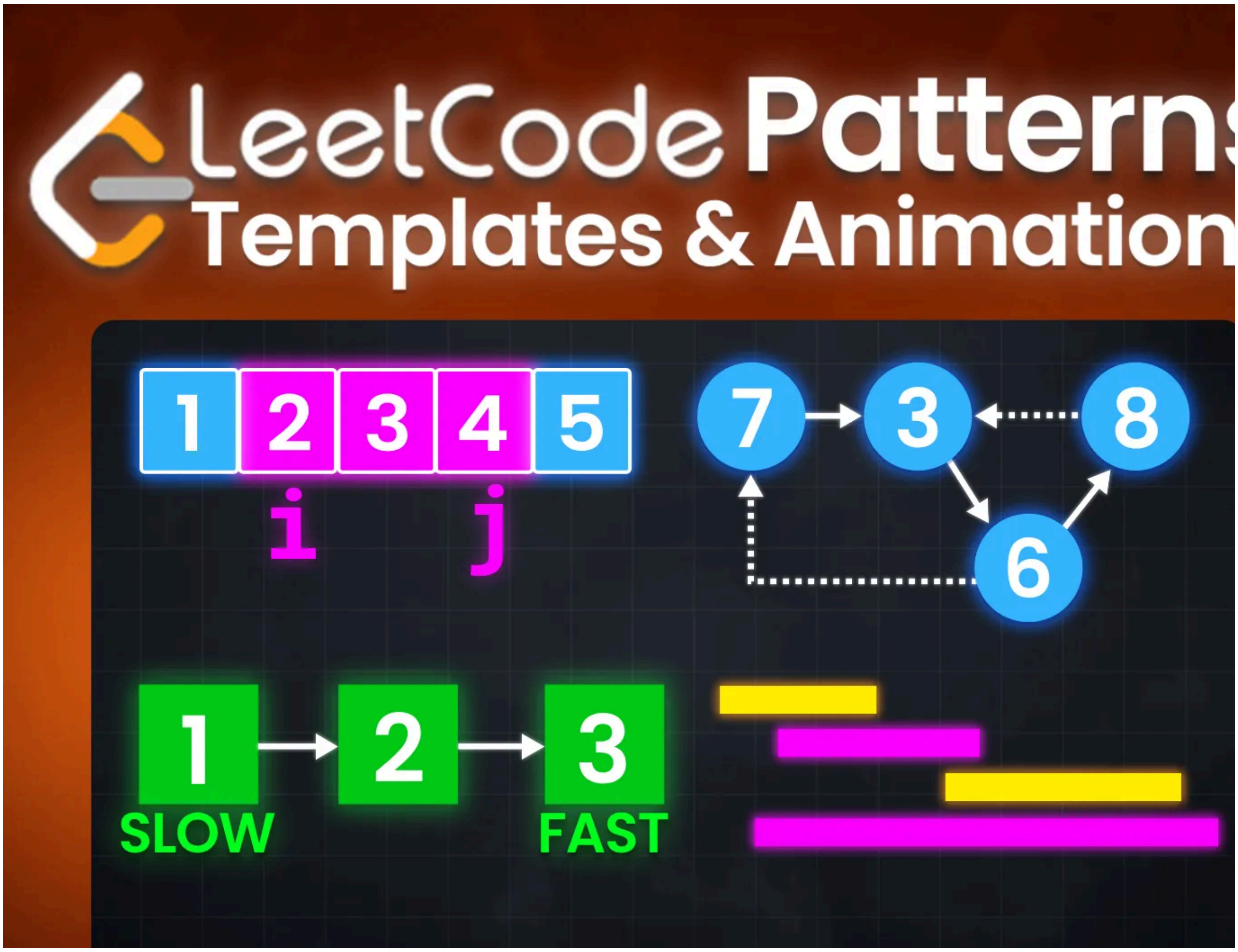
14 LeetCode Patterns to Solve Any Question

The only 14 patterns you'll ever need to master LeetCode Interviews!

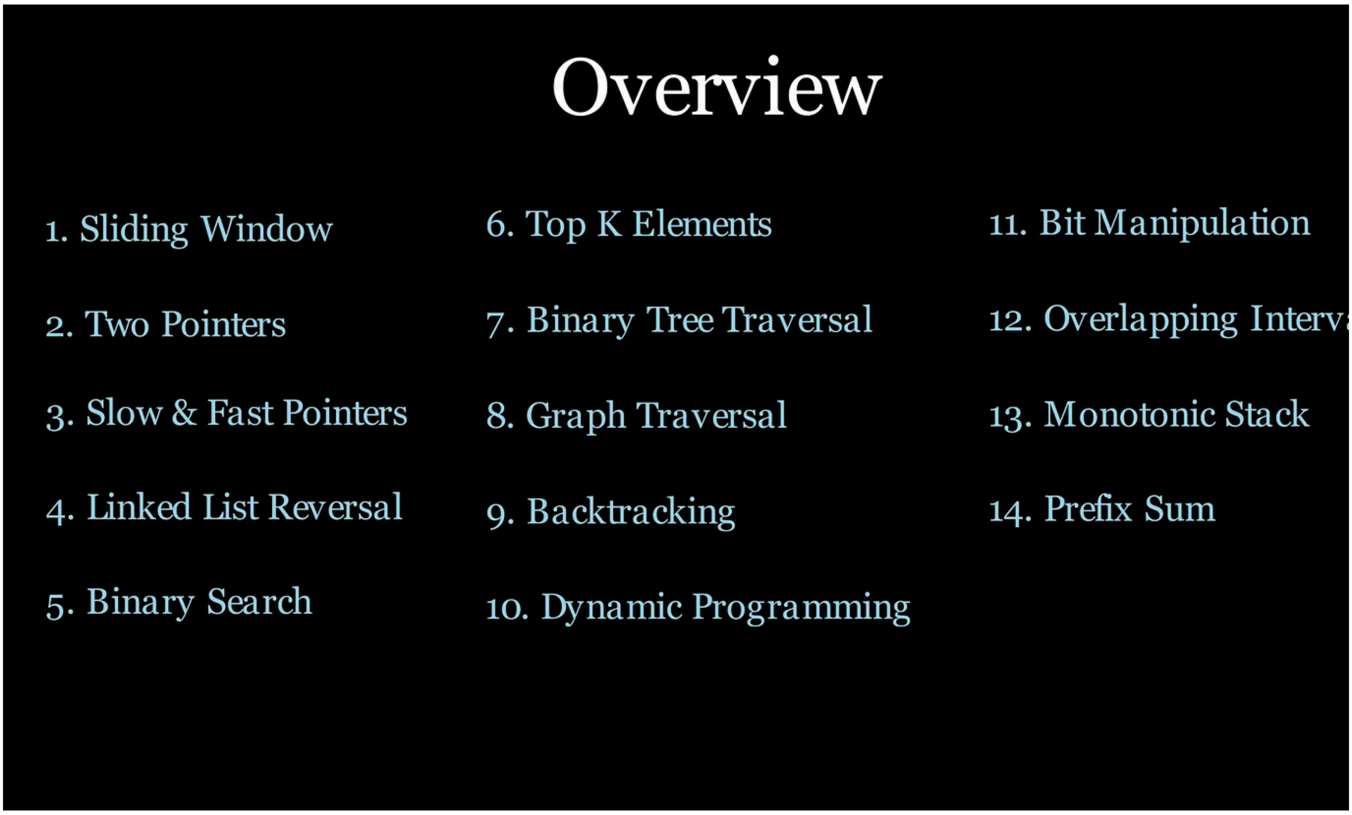
CODEINMOTION
NOV 19, 2024



If you prefer to watch this in video format with animations, watch it below!

> Instead of memorizing hundreds of LeetCode questions and answers, **learn generalized patterns to solve any question**.

Today, I will teach you all the coding patterns you need to know, when to use then what the benefit of the pattern is, a visual example, a code template to follow, and LeetCode questions you should solve to master the pattern.



1. Sliding Window

When to use it?

1. Linear data structures (arrays, lists, strings)
2. Must scan through a subarray or substring
3. When the subarray must satisfy some condition (shortest/longest/min/max)
4. Improve time complexity from O(N^2) to O(N)

Technique

In the sliding window, you have **2 pointers**, **i and j**. Move j as far as you can until y condition is no longer valid, then move the i pointer closer to j until the condition valid again to shrink the window. At every iteration, keep track of the min/max ler of the subarray for the result. Without the sliding window technique, we would ne use a double for loop resulting in O(N^2) time. The sliding window is O(N) time complexity.

Dynamic Sliding Window

In the dynamic sliding window, the size of the window (subarray between i and j) changes throughout the algorithm. In this example, we scan the subarray “bacb” a

find that we have a duplicate “b”, so we will move the i pointer to shrink the windo
and move on to letter “a”, resulting in “acb”, then we start moving j again.

Fixed Sliding Window

In the fixed sliding window, the size of the window is the same length throughout
algorithm. In this case, we need scan subarrays of length 3 for the final result, so v
initialize i and j to indices 0 and 2 and at every iteration we increment i and j by 1.

Coding Templates

```
"""
A generic template for dynamic sliding window finding min window length
"""
def shortest_window(nums, condition):
    i = 0
    min_length = float('inf')
    result = None

    for j in range(len(nums)):
        # Expand the window
        # Add nums[j] to the current window logic
```

```python
        # Shrink window as long as the condition is met
        while condition():
            # Update the result if the current window is smaller
            if j - i + 1 < min_length:
                min_length = j - i + 1
                # Add business logic to update result

            # Shrink the window from the left
            # Remove nums[i] from the current window logic
            i += 1

    return result

"""
A generic template for dynamic sliding window finding max window length
"""
def longest_window(nums, condition):
    i = 0
    max_length = 0
    result = None

    for j in range(len(nums)):
        # Expand the window
        # Add nums[j] to the current window logic

        # Shrink the window if the condition is violated
        while not condition():
            # Shrink the window from the left
            # Remove nums[i] from the current window logic
            i += 1

        # Update the result if the current window is larger
        if j - i + 1 > max_length:
            max_length = j - i + 1
            # Add business logic to update result

    return result

"""
A generic template for sliding window of fixed size
"""
def window_fixed_size(nums, k):
    i = 0
    result = None
```

```python
    for j in range(len(nums)):
        # Expand the window
        # Add nums[j] to the current window logic

        # Ensure window has size of K
        if (j - i + 1) < k:
            continue

        # Update Result
        # Remove nums[i] from window
        # increment i to maintain fixed window size of length k
        i += 1

    return result
```

LeetCode Questions

- 3. Longest Substring Without Repeating Characters
- 424. Longest Repeating Character Replacement
- 1876. Substrings of Size Three with Distinct Characters
- 76. Minimum Window Substring

2. Two Pointers

When to use it?

1. Linear data structures (arrays, lists, strings)
2. When you need to scan the start and end of a list
3. When you have a sorted list and need to find pairs
4. Removing duplicates or filtering

Technique

Instead of scanning all possible subarrays or substrings, use two pointers i and j at ends of a string or sorted array to be clever how you increment i or decrement j as scan the input. This will lower your time complexity from O(N^2) to O(N). In the example above, to detect if a string is a palindrome we scan the ends of the string character at a time. If the characters are equal, move i and j closer together. If they not equal, the string is not a palindrome.

Coding Templates

```
def two_pointer_template(input):
    # Initialize pointers
```

```
i = 0
j = len(input) - 1
result = None

# Iterate while pointers do not cross
while i < j:
    # Process the elements at both pointers

    # Adjust the pointers based on specific conditions
    # i += 1 or j -= 1

    # Break or continue based on a condition if required

# Return the final result or process output
return result
```

LeetCode Questions

- 125. Valid Palindrome
- 15. 3Sum
- 11. Container With Most Water

3. Slow and Fast Pointers

When to use it?

1. Linear data structures (arrays, lists, strings)
2. Detect cycle in linked list
3. Find middle of linked list
4. Perform in one pass with O(1) space

Technique

Use two pointers, a slow and fast pointer. Slow moves once and fast moves twice a every iteration. Instead of using a data structure to store previous nodes to detect cycle which requires O(N) space, using the two pointer technique will find a cycle O(1) space if fast loops around the cycle and will eventually meet slow. You can als use this technique to find the middle of a linked list in O(1) space and 1 pass.

Coding Templates

```
def slow_fast_pointers(head):
    # Initialize pointers
    slow = head
    fast = head
    result = None

    # move slow once, move fast twice
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        # update result based on custom logic
        # Example: if fast == slow then cycle is detected

    return result
```

LeetCode Questions

- 141. Linked List Cycle
- 142. Linked List Cycle II

- 19. Remove Nth Node From End of List

4. In Place Linked List Reversal

When to use it?

1. Reverse a linked list in 1 pass and O(1) space
2. Reverse a specific portion of a linked list
3. Reverse nodes in groups of k

Technique

Use two pointers, prev and ptr which point to the previous and current nodes. To reverse a linked list, ptr.next = prev. Then, move prev to ptr and move ptr to the ne node. At the end of the algorithm, prev will point to the head of the reversed list.

Coding Templates

```python
def reverse_linked_list(head):
    prev = None
    ptr = head

    while ptr:
        # Save the next node
        next_node = ptr.next

        # Reverse the current node's pointer
        ptr.next = prev

        # Move the pointers one step forward
        prev = ptr
        ptr = next_node

    # prev is the new head after the loop ends
    return prev
```

LeetCode Questions

- 206. Reverse Linked List
- 143. Reorder List
- 25. Reverse Nodes in k-Group

5. Binary Search

When to use it?

1. Input is sorted and you need to find a number
2. Finding the position of insertion in a sorted list
3. Handling duplicates in sorted arrays
4. Searching in rotated sorted arrays

Technique

Start left and right pointers at indices 0 and n-1, then find the mid point and see i is equal to, less than, or greater than your target. If nums[mid] > target, go left by moving the right pointer to mid-1. If nums[mid] < target, go right by moving left t mid+1. Binary Search reduces search time complexity from O(N) to O(NLogN)

Modified Binary Search

You may be required to find the leftmost or rightmost target value in an array containing duplicates.

You may be required to find a target in a rotated sorted array.

Coding Templates

```python
"""
Classic binary search algorithm that finds a target value
"""
def classic_binary_search(array, target):
    left, right = 0, len(array)-1
    while left <= right:
        mid = left + (right - left) // 2
        if array[mid] == target:
            return mid
        elif array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

"""
A generic template for binary search such that the returned value
is the minimum index where condition(k) is true

Example 1:
array = [1,2,2,2,3]
target = 2
binary_search(array, lambda mid: array[mid] >= target) --> 1

Example 2:
array = [1,2,2,2,3]
target = 2
binary_search(array, lambda mid: array[mid] > target) --> 4
"""
def binary_search(array, condition):
    left, right = 0, len(array)
    while left < right:
        mid = left + (right - left) // 2
        if condition(mid):
            right = mid
        else:
            left = mid + 1
    return left

"""
Binary search algorithm that can search a rotated array
by selected the appropriate half to scan at each iteration
"""
```

```python
def binary_search_rotated_array(array, target):
    left, right = 0, len(array)-1

    while left <= right:
        mid = (left + right) // 2
        if array[mid] == target:
            return mid

        # left side sorted
        if array[left] <= array[mid]:
            # if target is contained in left sorted side, go left
            if array[left] <= target <= array[mid]:
                right = mid - 1
            else:
                left = mid + 1

        # right side sorted
        else:
            # if target is contained in right sorted side, go right
            if array[mid] <= target <= array[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1
```

LeetCode Questions

- 34. Find First and Last Position of Element in Sorted Array
- 153. Find Minimum in Rotated Sorted Array
- 33. Search in Rotated Sorted Array

6. Top K Elements

When to use it?

1. Find the top k smallest or largest elements
2. Find the kth smallest or largest element
3. Find the k most frequent elements

Technique

You can always sort an array and then take the first or last k elements, however th time complexity would be O(NLogN). A heap can pop and push elements in O(Log where K is the size of the heap. Therefore, instead of sorting, we can use a heap to the smallest or largest K values, and for every element in the array check whether pop/push to the heap, resulting in a time complexity of O(NLogK).

Coding Templates

```
"""
A generic template for the Top K Smallest elements.
"""
import heapq
def top_k_smallest_elements(arr, k):
```

```python
    if k <= 0 or not arr:
        return []

    # Use a max heap to maintain the k smallest elements
    max_heap = []

    for num in arr:
        # Python does not have a maxHeap, only min Heap
        # Therefore, negate the num to simulate a max heap
        heapq.heappush(max_heap, -num)
        if len(max_heap) > k:
            heapq.heappop(max_heap)

    # Convert back to positive values and return
    return [-x for x in max_heap]
```

```python
"""
A generic template for the Top K Largest elements.
"""
import heapq
def top_k_largest_elements(arr, k):
    if k <= 0 or not arr:
        return []

    # Use a min heap to maintain the k largest elements
    min_heap = []

    for num in arr:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    return min_heap
```

LeetCode Questions

- 215. Kth Largest Element in an Array
- 347. Top K Frequent Elements
- 23. Merge k Sorted Lists

7. Binary Tree Traversal

When to use it?

1. Preorder: Serialize or deserialize a tree
2. Inorder: Retrieve elements in sorted order (BSTs)
3. Postorder: Process children before parent (bottom-up)
4. BFS: Level by level scanning

Technique

For the preorder, inorder, and postorder traversals use recursion (DFS). For the lev level scan use BFS iteratively with a queue.

Coding Templates

```
"""
Preorder traversal: visit node, then left subtree, then right subtree.
"""
def preorder_traversal(node):
    if not node:
        return
    # visit node
    preorder_traversal(node.left)
    preorder_traversal(node.right)

"""
Inorder traversal: visit left subtree, then node, then right subtree.
"""
def inorder_traversal(node):
    if not node:
        return
    inorder_traversal(node.left)
    # visit node
    inorder_traversal(node.right)

"""
Postorder traversal: visit left subtree, then right subtree, then node
"""
def postorder_traversal(node):
    if not node:
        return
    postorder_traversal(node.left)
    postorder_traversal(node.right)
    # visit node
```

```python
"""
BFS traversal: Visit all nodes level by level using a queue
"""
from collections import deque
def bfs_traversal(root):
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node:
            # visit node
            queue.append(node.left)
            queue.append(node.right)
```

LeetCode Questions

- 104. Maximum Depth of Binary Tree
- 102. Binary Tree Level Order Traversal
- 105. Construct Binary Tree from Preorder and Inorder Traversal
- 124. Binary Tree Maximum Path Sum

8. Graph and Matrices

When to use it?

1. Search graphs or matrices
2. DFS: Explore all possible paths (e.g., maze)

3. BFS: Find the shortest path
4. Topological Sort: Order tasks based on dependencies

Technique

DFS (**Depth-First Search**) traverses as deep as possible along each branch before backtracking, prioritizing visiting nodes or cells in a recursive or stack-based man **BFS** (**Breadth-First Search**) explores all neighbors of a node or cell before moving deeper, traversing level by level using a queue. For **DFS** use recursion with a visite to keep track of visited nodes. For **BFS** use iteration with a queue and a visited set keep track of visited nodes. In a graph, neighbors are found in the adjacency list. I matrix, neighbors are up/down/left/right cells, with some examples including diag too.

Coding Templates

```
"""
DFS for a graph represented as an adjacency list
"""
def dfs(graph):
    visited = set()
    result = []

    def explore(node):
        visited.add(node)
        result.append(node)  # process node
        for neighbor in graph[node]:
            if node not in visited:
                explore(neighbor)

    def dfs_driver(graph):
        for node in graph:
            if node not in visited:
                explore(node)

    dfs_driver()
    return result

"""
BFS for a graph represented as an adjacency list
"""
```

```python
from collections import deque
def bfs(graph, start):
    visited = set()
    result = []

    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)  # process node
            for neighbor in graph[node]:
                queue.append(neighbor)

    return result

"""
Topological Sort only works on DAG graphs with no cycles
"""
def topological_sort(graph):
    visited = set()
    topo_order = []

    def hasCycle(node, curpath):
        visited.add(node)
        curpath.add(node)

        for neighbor in graph[node]:
            if neighbor in curpath: # cycle detected, no topo sort
                return True
            if neighbor in visited:
                continue
            if hasCycle(neighbor, curpath):
                return True

        curpath.remove(node)
        topo_order.append(node) # process node
        return False

    for node in graph:
        if node not in visited:
            if hasCycle(node, set()):
                return None # cycle detected, no topo sort

    # reverse to get the correct topological order
    return topo_order[::-1]
```

```
"""
DFS for a matrix, visiting all connected cells.
"""
def dfs_matrix(matrix):
    m, n = len(matrix), len(matrix[0])
    visited = set()
    result = []

    def explore(i, j):
        if not (0 <= i < m and 0 <= j < n):
            return
        if ((i,j)) in visited:
            return
        visited.add((i,j))
        result.append(matrix[i][j])  # process the cell

        # Explore neighbors (up, down, left, right)
        for deltaI, deltaJ in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            explore(i + deltaI, j + deltaJ)

    def dfs_driver():
        for i in range(m):
            for j in range(n):
                if (i, j) not in visited:
                    explore(i, j)

    dfs_driver()
    return result

"""
BFS for a matrix, visiting all connected cells.
"""
from collections import deque
def bfs_matrix(matrix, startI, startJ):
    m, n = len(matrix), len(matrix[0])
    visited = set()
    result = []

    queue = deque([(startI,startJ)])
    while queue:
        i, j = queue.popleft()
        if not (0 <= i < m and 0 <= j < n):
            continue
        if ((i,j)) in visited:
            continue
```

```
        visited.add((i,j))
        result.append(matrix[i][j])  # process the cell

        # Enqueue neighbors (up, down, left, right)
        for deltaI, deltaJ in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            queue.append((i + deltaI, j + deltaJ))

    return result
```

LeetCode Questions

- 79. Word Search
- 207. Course Schedule
- 994. Rotting Oranges
- 417. Pacific Atlantic Water Flow
- 127. Word Ladder

9. Backtracking

When to use it?

1. Combinatorial problems (combinations, permutations, subsets)
2. Constraint satisfaction (Sudoku, N-Queens)
3. Prune paths using constraints to reduce search space

Technique

Backtracking is closely related to DFS, but with a focus on finding solutions while validating their correctness. If a solution doesn't work, you backtrack by returning the previous recursive state and trying a different option. Additionally, backtracki uses constraints to eliminate branches that cannot lead to a valid solution, making search more efficient.

Coding Templates

```
"""
Generic backtracking template.
"""
def backtrack(candidates, curPath):
    # Base case: Check if the solution meets the problem's criteria
```

```python
    if is_solution(curPath):
        process_solution(curPath)
        return

    for candidate in candidates:
        if is_valid(candidate, curPath):
            # Take the current candidate
            curPath.append(candidate)

            # Recurse to explore further solutions
            backtrack(candidates, curPath)

            # Undo the choice (backtrack)
            curPath.pop()
```

LeetCode Questions

- 78. Subsets
- 46. Permutations
- 39. Combination Sum
- 37. Sudoku Solver
- 51. N-Queens

10. Dynamic Programming

When to use it?

1. Overlapping subproblems and optimal substructure
2. Optimization problems (min/max distance, profit, etc.)
3. Sequence problems (longest increasing subsequence)
4. Combinatorial problems (number of ways to do something)
5. Reduce time complexity from exponential to polynomial

Technique

Dynamic Programming is used when you need to solve a problem that depends o
previous results from subproblems. You can effectively "cache" these previous res
values when you calculate them for the first time to be re-used later. Dynamic Programming has 2 main techniques:

- **Top Down** - Recursion (DFS) with Memoization. Memoization is a fancy word
  a hashmap that can cache the values previously calculated. In the top down approach you start with the global problem and the recursively split it into subproblems to then solve the global problem.

- **Bottom Up -** Iteratively performed by using an array/matrix to store previous values. In the bottom up approach we start with base cases and then build up the global solution iteratively.
- Many times bottom up is preferred since you can reduce the space complexity you don't need access to all subproblems and can store the last couple of subproblem results using variables.

Coding Templates

```
"""
Top-down recursive Fibonacci without memoization.
Time: O(2^N) | Space: O(N)
"""
def fib_top_down(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib_top_down(n-1) + fib_top_down(n-2)

"""
Top-down recursive Fibonacci with memoization.
Time: O(N) | Space: O(N)
"""
def fib_top_down_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n == 0:
        return 0
    if n == 1:
        return 1
    memo[n] = fib_top_down_memo(n-1, memo) + fib_top_down_memo(n-2, memo)
    return memo[n]

"""
Bottom-up Fibonacci using an array.
Time: O(N) | Space: O(N)
"""
def fib_bottom_up_array(n):
    dp = [0] * (n + 1)
    dp[1] = 1
```

```
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

"""
Bottom-up Fibonacci using 2 variables.
Time: O(N) | Space: O(1)
"""
def fib_bottom_up(n):
    prev2, prev1 = 0, 1

    for i in range(2, n + 1):
        fib = prev2 + prev1
        prev2 = prev1
        prev1 = fib

    return prev1
```

LeetCode Questions

- 70. Climbing Stairs
- 322. Coin Change
- 1143. Longest Common Subsequence
- 300. Longest Increasing Subsequence
- 72. Edit Distance

11. Bit Manipulation

When to use it?

1. Count number of 0 or 1 bits in a number
2. Add numbers without using addition or substraction
3. Find a missing number in a list

Technique

For bit manipulation, ensure you understand the basic bitwise operators: AND, O NOT, XOR and bitwise shifts. Specifically, XOR has interesting properties that all you to find a missing number in a list or add up 2 numbers without using the addi or subtraction operation. You want to be comfortable with some of the basic bitwi operators, but don't go too in depth here as these questions can be quite rare.

Coding Templates

```
"""
Useful bitwise operators for LeetCode
"""
def binary_operators():
    return {
```

```
        "AND": a & b,
        "OR": a | b,
        "XOR": a ^ b,
        "NOT": ~a, # ~a = -a-1 in python
        "Left Shift (a << b)": a << b,  # left shift 'a' by 'b' bits
        "Right Shift (a >> b)": a >> b,  # right shift 'a' by 'b' bits
        "Mask": a & 1 # gives you the least significant bit of a
    }
```

LeetCode Questions

- 191. Number of 1 Bits
- 190. Reverse Bits
- 268. Missing Number
- 371. Sum of Two Integers
- 338. Counting Bits

12. Overlapping Intervals

When to use it?

1. Merge or consolidate ranges
2. Schedule or find conflicts (e.g. meeting rooms)
3. Find gaps or missing intervals

Technique

Knowing how to merge overlapping intervals is crucial for these problems.

$$\text{If } a_{\text{end}} \geq b_{\text{start}} : \text{mergedInterval} = [\min(a_{\text{start}}, b_{\text{start}}), \max(a_{\text{end}}, b_{\text{end}}))$$

Usually, you will want to sort the input by the start times so you can guarantee orc and always have “a” appear before “b”, making it easier to compare time ranges in chronological order. In the example above, we insert [4,8] into the array, but that eventually gets merged with [3,5], [6,7], and [8,10], into [3,10].

Coding Templates

```
"""
Generic template for interval problems
```

```python
"""
def process_intervals(intervals):
    # Sort intervals by start time (common preprocessing step)
    intervals.sort(key=lambda x: x[0])

    # Example: Merged intervals (modify as needed for your problem)
    result = []
    for interval in intervals:
        # If result is empty or no overlap with the last interval in result
        if not result or result[-1][1] < interval[0]:
            result.append(interval)  # Add the interval as is
        else:
            # Merge overlapping intervals
            result[-1][1] = max(result[-1][1], interval[1])

    return result
```

LeetCode Questions

- 57. Insert Interval
- 56. Merge Intervals
- 435. Non-overlapping Intervals
- 1834. Single-Threaded CPU

13. Monotonic Stack

When to use it?

1. Find Next Greater or Smaller Element
2. Find left/right boundary points in histograms or rectangles

3. Maintain elements in order to optimize operations

Technique

If you want to find the next greater/smaller element for all elements in an array th brute force approach will take O(N^2). However, with the use of a monotonic stack (either increasing or decreasing order, depending on the problem), we can achieve O(N) time by storing and keeping track of the greatest/smallest elements up until t current iteration. In the example above, notice that we popped 57 from the stack because 69 is greater than 57. However, 76 is greater than 69 so it is a valid solutio and we update the output.

Coding Templates

```
"""
Monotonic increasing stack template.
"""
def monotonic_increasing_stack(arr):
    stack = []
    for i, num in enumerate(arr):
        # Modify condition based on the problem
        while stack and stack[-1][0] > num:
            stack.pop()

        if stack:
            pass # process result from top of stack

        # Append current value and index
        stack.append((num,i))

"""
Monotonic decreasing stack template.
"""
def monotonic_decreasing_stack(arr):
    stack = []
    for i, num in enumerate(arr):
        # Modify condition based on the problem
        while stack and stack[-1][0] < num:
            stack.pop()

        if stack:
            pass # process result from top of stack
```

```
        # Append current value and index
        stack.append((num,i))
```

LeetCode Questions

- 496. Next Greater Element I
- 503. Next Greater Element II
- 739. Daily Temperatures
- 84. Largest Rectangle in Histogram

14. Prefix Sum

When to use it?

1. Cumulative sums are needed from index 0 to any element
2. Querying subarray sums frequently across multiple ranges
3. Partial sums can be reused efficiently

Technique

To sum a subarray would take O(N). To sum Q subarrays would take O(N*Q). Can w perform a more efficient algorithm to answer queries? Yes, we can calculate a pref sum array where

$$\text{prefix}[i] = \text{prefix}[i-1] + \text{input}[i]$$

And then we can find the sum of any subarray in O(1) time using the formula

$$\text{sum}[i:j] = \text{prefix}[j] - \text{prefix}[i-1]$$

Therefore, we can answer Q queries in O(N) time complexity with a prefix sum.

Coding Templates

```
"""
Builds the prefix sum array
"""
def build_prefix_sum(arr):
    # Initialize prefix sum array
    n = len(arr)
    prefix = [0] * n

    # First element is the same as the original array
    prefix[0] = arr[0]

    # Build the prefix sum array
    for i in range(1, n):
        prefix[i] = prefix[i - 1] + arr[i]

    return prefix

"""
Queries the sum of elements in a subarray [left, right] using prefix sum.
"""
def query_subarray_sum(prefix, i, j):
    if i == 0:
        return prefix[j]
    return prefix[j] - prefix[i - 1]
```

LeetCode Questions

- 303. Range Sum Query - Immutable
- 523. Continuous Subarray Sum
- 560. Subarray Sum Equals K

101 Likes · 13 Restacks

Discussion about this post

Comments Restacks

Write a comment...

Lavesh choudhary 16 авг.

I liked the order in which you have given the problems to practice. I just want to tell you about correction needed. It's the time complexity of binary search which is mentioned O(NlogN).

LIKE REPLY

javinpaul 27 мар.

Nice article loved it.

LIKE REPLY

1 more comment...