Back

PART-2: CODING PATTERN & TEMPLATE for LINEAR DS (Personal Notes)

adityakrverma ★ 738 Last Edit: November 23, 2024 9:33 AM 332 VIEWS

5

SECTION-A : **CODING PATTERNS** : Updated 10/2/24 | **CODING PATTERNS -1** | **CODING PATTERNS -2** | **PART-1 Graph Template**

```
ARRAYS &  STRINGS: ( Below are the algorithm and techniques )
================
1. Prefix Sum [ 560, 525] [303, 304]
2. Kadane's Algorithm [53, 152]
3. Sliding Window
4. Two Pointer. -> Reduces time complexity from O(n^2) to O(n)
5. Overlapping / Merge Intervals
6. Modified Binary Search [33,153]
7. Cyclic Sort [41]
8. Recusrion, Backtracking
9. Greedy Algorithm [ 55, 45, 53 ] [ Dijkstra - 743, 787]
10. Dynamic Programming [ 53, 152, 91 ..]


BACKTRACKING:
=============
1. Subset
2. Combinations
3. Permutation


DYNAMIC PROGRAMMING:
====================
1. 0/1 Knapsack
2. Unbounded Knapsack
3. Fibonacci Numbers [ 70, 746, 198, 91 ]
4. Kadane's Algorithm [ 53, 152, 42, 303 ]
5. Longest Increasing Subsequence (LIS)
6. Longest Common Subsequence (LCS) - 2D DP
7. DP on Matrix/Grid [62,63,64]     - 2D DP
8. Matrix Chain Multiplication


STACK:
======
1. Monotonic Stack [ 739, 496, 84] Find next greater/smaller. Reduces time complexity from O(N^2) to O(N)
2. Max Stack
3. Min Stack


LINKED LIST:
============
1. Fast and Slow Pointer ( Find mid-point. Cyclic Detection) - 141, 287
2. LinkedList In-place Reversal (206, 92, 24, 25) - Using prev, head, tail pointers.
( Reverse a linked-list without using extra memory)


HEAP:
=====
1. Top K Elements
2. Two Heaps - Median of Stream
3. K-way Merge (21, 23)


TREES:
======
1. Tree DFS Traversal (InOrder, PreOrder, PostOrder)
2. Tree BFS ( Iterative using Queue)
3. Tries
4. N-ary (Generic Tree)
5. QuadTrees
6. Segment Trees


GRAPHS:
=======
1. Matrix DFS
2. Matrix BFS - Specially for shortest path problems.
3. Graph DFS ( Adjacency List / Adjacency Matrix, Edges/Vertex)
4. Graph BFS ( Adjacency List. and all - Shortest Distance )
5. Union Find ( Disjoint Sets) - Cyclic detection for unidirectional graphs, Find Disjoint-Sets
6. Dijkstra Algorithm - Shortest Path for Weighted graphs
7. Topological Sort - Cyclic detection for Directed Graphs (DAG). Ordering
8. Kruskal Algorithm
9. Prims Algorithm
```

Graph Summary from Ashish's Article

- **CODING PATTERNS ARTICLE-1**
- **CODING PATTERNS ARTICLE-2** | PYTHON COURSE
- **NEETCODE IS BIBLE**

SECTION-B : **TEMPLATE FOR BELOW TOPICS ARE COVERED:** Practice **Neetcode-150** & **MY-LIST**

```
# 01. BINARY SEARCH TEMPLATE
# 02. SLIDING WINDOW TEMPLATE
# 03. TWO POINTER TECHNIQUE ( Reduces Time Complexity from O(N^2) to O(N))
# 04. PREFIX SUM TEMPLATE/CONCEPTS
# 05. RECUSION CONCEPTS
# 06. BACKTRACKING CONCEPTS
# 07. DYNAMIC PROGRAMMING CONCEPTS
# 08. HEAP EXAMPLES
# 09. HASH TABLE CONCEPTS
# 10. LINKED LIST CONCEPTS
# 11. QUEUE DESIGN CONCEPTS
# 12. MONOTONIC STACK CONCEPTS
# 13. BIT MANIPULATION TIPS/TRICKS
# 14. CYCLIC SORT
# 15. GREEDY APPROACH
```

1. Binary Search : O (log N) : Updated 10/2/24:

When to Use Binary Search ?
Use Binary Search when:

1. Data is sorted: The most straightforward use case. LC-981, 278, 34
2. Searching for a condition: Problems where you need to find the smallest/largest value satisfying a condition. LC-162
3. Optimization problems: Finding an optimal value (e.g., minimum capacity, maximum profit). LC-875
4. Searching in special data structures: Such as rotated arrays or matrices. LC-153, 33, 74

```
Key Points :

1. Middle Calculation: mid = left + (right - left) // 2 prevents potential overflow compared to (left + right) // 2.
2. Loop Condition: left <= right ( for standard template) ensures all elements are checked.
3. Adjusting Pointers: Depending on comparison, adjust left or right to narrow the search. i.e right=mid-1,
   left=mid+1 (for standard BS template), since we already checked target==mid condition.
```

Standard Binary Search Algorithm Template :

```
class Solution(object):

    def binary_search(self, nums, target):

        left = 0
        right = len(nums) - 1

        while left <= right:                                    # NOTE-1: left <= right  [ If we take "while left < right" then "right = mid"

            mid = left + (right - left) // 2                    # NOTE-2: Prevent potential overflow.

            if nums[mid] == target:                             # NOTE-3: Mostly for search-based problems LC-981, 33, 34.
                return mid

            elif nums[mid] < target:
                left = mid + 1                                  # NOTE-4: mid+1
            else:
                right = mid - 1                                 # NOTE-5: mid-1 ( we already considered mid in note-3)

        return -1
```

Modified Binary Search Algorithm Template (for finding minimum): Note LC-153, 162 use this template, but logic slight different.

```
class Solution(object):

    def findMin(self, nums):

        low = 0
        high = len(nums)-1
        ans = 0

        while low < high:                           # NOTE-1: Its NOT low <= high unlike Standard BS
```

```
            mid = low +(high-low)//2                # NOTE-2: Prevent potential overflow.

                                                    # NOTE-3: MISSING comparision with mid value as no target, unlike Standard BS

            if nums[mid] > nums[high]:
                low = mid+1                         # NOTE-4: mid +1
            else:
                high = mid                          # NOTE-5: consider mid as well instead of (mid-1), as missed comparing mid in note-3, unlik
        return nums[low]
```

Common Binary Search Variations :

```
Here are some popular LeetCode problems where Binary Search is the key to an efficient solution:

704. Binary Search
Description: Classic binary search implementation on a sorted array.
Solution: Apply the basic iterative or recursive Binary Search.

34. Find First and Last Position of Element in Sorted Array
Description: Find the starting and ending position of a given target.
Solution: Use Binary Search to find the first and last occurrences separately.

33. Search in Rotated Sorted Array
Description: Search for a target in a rotated sorted array.
Solution: Identify the sorted half in each iteration and decide where to search next.

162. Find Peak Element
Description: Find a peak element where its greater than its neighbors.
Solution: Use a modified Binary Search to move towards the peak.

410. Split Array Largest Sum
Description: Split the array into m subarrays to minimize the largest sum.
Solution: Apply Binary Search on the possible sum range and validate splits.

875. Koko Eating Bananas
Description: Determine the minimum eating speed to finish bananas within H hours.
Solution: Binary Search on the possible eating speeds.

4. Median of Two Sorted Arrays
Description: Find the median of two sorted arrays.
Solution: Apply Binary Search on the partitioning of the arrays.

1351. Count Negative Numbers in a Sorted Matrix
Description: Count negative numbers in a matrix where each row and column is sorted.
Solution: Apply Binary Search on each row to find the first negative number.
```

Facebook Onsite 2019: Given array of int numbers nums = [1,2,3,6,8,9,12,4]. Find closest value to 3.2 . Write binary seach code | Search TARGET given , use Standard BS Template

```
def find_closest(arr, target):
    # Sort the array to apply binary search
    arr.sort()

    left, right = 0, len(arr) - 1
    closest = arr[0]  # Initialize closest to the first element

    while left <= right:                    # NOTE
        mid = left + (right - left) // 2

        # Update closest if the current mid is closer to the target
        if abs(arr[mid] - target) < abs(closest - target):
            closest = arr[mid]

        # Binary search logic
        if arr[mid] < target:
            left = mid + 1
        elif arr[mid] > target:
            right = mid - 1
        else:
            return arr[mid]  # Exact match

    return closest

# Example usage
nums = [1, 2, 3, 6, 8, 9, 12, 4]
target = 3.2

result = find_closest(nums, target)
print(f"The closest value to {target} is {result}")
```

LC:981: Time-based Key-Value Store: Search TARGET given , use Standard BS Template

```
class TimeMap(object):

    def __init__(self):
        self.h = {}


    def set(self, key, value, timestamp):

        #self.h[key].append((value,timestamp))
        self.h[key] = self.h.get(key, [])   +    [(value, timestamp)]


    def get(self, key, timestamp):

        if key not in self.h:
            return ""
        return self.binarySearch(key,timestamp)


    def binarySearch(self, key, timestamp):

        values = self.h[key]
        low = 0
        high = len(values) - 1
        ans = ""

        while low <= high:           # NOTE

            mid = low + (high - low)//2

            if values[mid][1] == timestamp:
                return values[mid][0]

            elif values[mid][1] < timestamp:
                ans = values[mid][0]          # IMPORTANT : Since the list is sorted, this could be the latest value less than or equal to the t
                low = mid + 1

            elif values[mid][1] > timestamp:
                high = mid - 1

        return ans
```

LC-33: Search in rotated sorted array : Search TARGET given , use Standard BS template

```
class Solution(object):
    def search(self, nums, target):

        start = 0
        end = len(nums)-1

        while start <= end:               # NOTE
            mid = (start + end)//2

            if nums[mid] == target:
                return mid

            if nums[start] <= nums[mid]: # array is sorted for first half. Not rotated
                if nums[start] <= target < nums[mid]:
                    end = mid-1
                else:
                    start = mid+1

            else:         # It means left subarray was rotated, and right subarray is sorted.
                          # Therefore, we can determine whether to proceed with the right subarray
                          # by comparing the target with its boundary elements:
                if nums[mid] < target <= nums[end]:
                    start = mid+1
                else:
                    end = mid-1

        return -1
```

LC-34: Find first and last occurance of element in Sorted array | Search TARGET given , use Standard BS Template

```
class Solution(object):
    def searchRange(self, nums, target):

        # 10/2/24:
        # -------
        first = self.find_first(nums, target)
        last  = self.find_last(nums, target)
        return [first, last]


    def find_first(self, nums, target):
```

```
    def find_first(self, nums, target):

        left, right = 0, len(nums) - 1
        first_occurrence = -1

        while left <= right:                    # NOTE
            mid = left + (right - left) // 2

            if nums[mid] == target:             # Mostly for search problem.
                first_occurrence = mid
                right = mid - 1  # Continue searching in the left half

            elif nums[mid] < target:
                left = mid + 1                   # NOTE
            else:
                right = mid - 1                  # NOTE

        return first_occurrence

    def find_last(self, nums, target):

        left, right = 0, len(nums) - 1
        last_occurrence = -1

        while left <= right:                       # NOTE
            mid = left + (right - left) // 2

            if nums[mid] == target:                # Mostly for search problem.
                last_occurrence = mid
                left = mid + 1  # Continue searching in the right half

            elif nums[mid] < target:
                left = mid + 1                     # NOTE
            else:
                right = mid - 1                    # NOTE

        return last_occurrence
```

LC-35. Search Insert Position | Search TARGET given , use Standard BS Template

```
class Solution(object):
    def searchInsert(self, nums, target):

        # Yay ! Solved myself :)

        start = 0
        end = len(nums)-1

        while start <= end:

            mid = start + (end-start)//2

            if nums[mid] == target:
                return mid

            if nums[mid] > target:
                end = mid-1
            else:
                start = mid+1

        return start
```

LC-278: First Bad Version | Search TARGET given , use Standard BS Template

```
class Solution(object):
    def firstBadVersion(self, n):

        # Yay ! Solved myself :)

        start = 1
        end = n

        while start <= end:

            mid = start + (end-start)//2

            if isBadVersion(mid) == False:
                start = mid +1
            else:
                end = mid-1

        return start
```

LC-74: Binary Search on 2D Matrix: | Search TARGET given , use Standard BS Template

```python
class Solution(object):
    def searchMatrix(self, matrix, target):

        if not matrix or target is None:
            return False

        # TIP: Don't treat it as a 2D matrix, just treat it as a sorted list
        left = 0
        right = len(matrix) * len(matrix[0]) - 1

        while left <= right:                  # NOTE

            mid = left + (right-left)//2

            row, col = mid//len(matrix[0]), mid%(len(matrix[0]))
            #row, col = divmod(mid, len(matrix[0])). # Both same
            num = matrix[row][col]

            if num == target:                 # NOTE
                return True

            elif num < target:                # NOTE
                left = mid + 1

            else:
                right = mid - 1

        return False
```

IMPORTANT: **When to Use Each Condition** :

- Use **while left < right** When:
  Objective: Narrowing down to a single candidate (e.g., **finding a peak, searching for a minimum/maximum**) . Also use (right = mid) instead of (right= mid-1) to make sure we don't miss potential peak/minimun based on comparison. Also note if we took classic BS, then we do compare mid with target already, that why we consider mid+1 or mid-1, however with peak/minimum BS problems we're not checking mid, so we need to include mid to make sure we don't miss on potential candidate.
  Example: LC-162- Find Peak element, LC-152: Find minimum in Rotated Sorted array.
  Guarantee: The algorithm can guarantee that a solution exists within the narrowed search space.
  Simpler Logic: You prefer a clear termination when left equals right.
- Use **while left <= right** When:
  Objective: Finding a specific target (e.g., **classic binary search for exact match**).
  Guarantee: You need to consider every element, including when left equals right.
  Flexibility: The search might need to return a condition based on different criteria (e.g., closest match, first occurrence).

Key Takeaway :

- Use while left < right when:
  You **aim to narrow down the search** to a single candidate.
  The boundary adjustments include mid as a potential candidate (e.g., right = mid).
- Use while left <= right when:
  You **search for a specific target** and need to consider every element.
  Boundary adjustments exclude mid (e.g., right = mid - 1).

Understanding the Binary Search Approach in LC-162

Standard Binary Search vs. Modified Binary Search

1. **Standard Binary Search**:
   - **Objective**: Find the index of a specific target value in a **sorted** array.
   - **Loop Condition**: `while left <= right`
   - **Boundary Adjustments**:
     - If `nums[mid] < target`: `left = mid + 1`
     - If `nums[mid] > target`: `right = mid - 1`
   - **Termination**: When `left > right`, target not found.
2. **Modified Binary Search for Find Peak Element**:
   - **Objective**: Find **any peak** in an **unsorted** array.
   - **Loop Condition**: `while left < right`
   - **Boundary Adjustments**:
     - If `nums[mid] < nums[mid + 1]`: `left = mid + 1`
     - Else: `right = mid`
   - **Termination**: When `left == right`, peak found at index `left`.

Key Differences

Aspect	Standard Binary Search	Find Peak Element (LC-162)
Objective	Find exact target index in sorted array	Find any peak element in unsorted array
Loop Condition	`while left <= right`	`while left < right`
Boundary Adjustment (Right)	`right = mid - 1`	`right = mid`
Boundary Adjustment (Left)	`left = mid + 1`	`left = mid + 1`
Termination Condition	`left > right`	`left == right`
Reason for Boundary Adjustment	Eliminates `mid` as a possible candidate after comparison	Keeps `mid` as a potential peak based on comparison

---

LC 153 : Find minimum in Rotated Sorted Array : No Taregt to compare and we need to find dup/peak/min, so use Modified BS Template.

NOTE: Here LC-153: To find the minimum, you only need to determine which side of the array the minimum lies based on a simple comparison, without needing to know the exact sorted order of both halves. which is more simplified. Unlike LC-33: To efficiently find the target, you need to know which half of the array is sorted. This knowledge helps you decide whether the target lies within the sorted half or the unsorted (rotated) half.

```
class Solution(object):

    def findMin(self, nums):

        # Bruteforce : Iterate through list and keep min variable - O(n)
        # Optimal : Apply BS and find the min.

        # Given array is sorted in ascending order is rotated, so its easier to check if last element is less than mid, meaning lowest might be

        low = 0
        high = len(nums)-1
        ans = 0

        while low < high:
            mid = low +(high-low)//2
            if nums[mid] > nums[high]:
                low = mid+1
            else:
                high = mid
        return nums[low]
```

LC-162: Find Peak Element | No Taregt to compare and we need to find dup/peak/min, so use Modified BS Template.

```
class Solution(object):
    def findPeakElement(self, nums):

        left = 0
        right = len(nums) - 1

        while left < right:
            mid = left + (right - left) // 2  # Prevents potential overflow

            # Compare mid element with its right neighbor
            if nums[mid] < nums[mid + 1]:
                # Peak must be in the right half
                left = mid + 1
            else:
                # Peak is in the left half (including mid)
                right = mid

        # When left == right, we've found a peak
        return left
```

LC-287: Find the duplicate number | No Target to compare ; instead we need to find dup/peak/min, so use Modified BS Template.

```
class Solution(object):
    def findDuplicate(self, nums):

        # Time: O(N*log N)
        # Initialize the binary search range
        left, right = 1, len(nums) - 1  # Search space is 1 to n

        while left < right:
            mid = left + (right - left) // 2  # Calculate mid-point

            # Count how many numbers are <= mid
            count = sum(1 for num in nums if num <= mid)

            if count > mid:
```

```
            if count > mid:
                # If count is greater than mid, the duplicate is in the left half
                right = mid
            else:
                # Otherwise, the duplicate is in the right half
                left = mid + 1

        # When left == right, we found the duplicate number
        return left
```

LC-875 Koko eating Bananas: | No Target to compare ; instead we need to find dup/peak/min, so use Modified BS Template.

```
class Solution(object):
    def minEatingSpeed(self, piles, h):

        left = 1
        right = max(piles)
        res = right

        while left < right:
            k = left + (right-left) // 2

            totalTime = 0
            for p in piles:
                totalTime += math.ceil(float(p) / k)

            if totalTime <= h:
                res = k
                right = k
            else:
                left = k + 1

        return res
```

My Calender : 10/22/24:

```
class MyCalendar:

    def __init__(self):
        self.calender = []

    #  The binary search goes through the calender to find where a new interval (based on the start time) could be inserted.
    # If the exact start time is not found, it returns the appropriate index (s) for inserting the interval.

    # 10/21/24: Standard Binary Search.
    def binary_search(self, start_point):

        start = 0
        end = len(self.calender) - 1

        while start <= end:
            mid = start + (end - start) // 2
            if self.calender[mid][0] < start_point:
                start = mid + 1
            elif self.calender[mid][0] > start_point:
                end = mid - 1
            else:
                return mid
        return start


    def book(self, start, end):

        # After the binary search, i gives the position where the new
        # interval [start, end) could potentially be inserted.
        i = self.binary_search(start)

        # Checking Overlapping condition :

        # Check the NEXT event
        # If there is an event at i and its start time of NEXT event is earlier
        # than end time of the new event (self.calender[i][0] < end), it means
        # there's an overlap.
        if i < len(self.calender) and self.calender[i][0] < end: # for NEXT event
            return False

        # Check the PREVIOUS event:
        # If there's an event before i and its end time is of PREVIOUS event is
        # later than new event's start time, it also indicates an overlap.
        if i - 1 >= 0 and self.calender[i - 1][1] > start:  # for PREVIOUS event
            return False

        # If the new event starts exactly when the previous one ends
        # (calender[i - 1][1] == start), they are merged by updating the
        # end time of the previous event.
```

```
        if i - 1 >= 0 and self.calender[i - 1][1] == start:
            self.calender[i - 1][1] = end

        self.calender.insert(i, [start, end]) # Insert the new interval at ith position.

        return True
```

LC-658: Find K closest Element:

Approach:

1. Use binary search to find the starting index of the k closest elements.
2. The search space will be from left = 0 to right = len(arr) - k because we want a window of size k to be contained within the array.
3. For each middle point mid, we compare the distances between x and the elements at mid and mid + k. This helps in determining whether to move the left or right pointer.

```
class Solution(object):

    # Time complexity: O(log(N-k)+k).
    def findClosestElements(self, arr, k, x):

        # Initialize binary search bounds
        left = 0
        right = len(arr) - k

        # Binary search against the criteria described
        while left < right:
            mid = (left + right) // 2

            if x - arr[mid] > arr[mid + k] - x:
                left = mid + 1
            else:
                right = mid

        return arr[left:left + k]
```

Summary of Loop Conditions Across Different Binary Search Problems

Aspect	Find Duplicate Number (LC-287)	Find Minimum in Rotated Sorted Array (LC-153)	Find Peak Element (LC-162)	Standard Binary Search (Find Target)
Loop Condition	`while left < right`	`while left < right`	`while left < right`	`while left <= right`
When `left < right`	Ensures search space converges to a single number	Ensures search space converges to the minimum	Ensures search space converges to a single peak	Continues until the target is found or space exhausted
Boundary Adjustment (Right)	`right = mid` if count > mid	`right = mid` if `nums[mid] <= nums[right]`	`right = mid` if `nums[mid] > nums[mid + 1]`	`right = mid - 1` if `nums[mid] > target`
Boundary Adjustment (Left)	`left = mid + 1` if count <= mid	`left = mid + 1` if `nums[mid] > nums[right]`	`left = mid + 1` if `nums[mid] < nums[mid + 1]`	`left = mid + 1` if `nums[mid] < target`
Termination	`left == right` (duplicate found)	`left == right` (minimum found)	`left == right` (peak found)	`left > right` (target not found) or target found
Purpose of Using `<` vs `<=`	To preserve potential duplicate and converge precisely	To preserve potential minimum and converge precisely	To preserve potential peak and converge precisely	To ensure all elements are considered

Key Takeaways

1. **Understand the Objective**:
   - **LC-287**: Find any duplicate number.
   - **LC-153**: Find the minimum number in a rotated sorted array.
   - **LC-162**: Find any peak element.
   - **tandard Binary Search**: Find the exact target number.
2. **Choose the Appropriate Loop Condition**:
   - `while left < right` is ideal when you want to **converge** to a single element that satisfies a condition (like duplicate, minimum, or peak).

- `while left <= right` is suited for **exact searches** where every element needs to be considered, typically when searching for a specific target.

3. **Boundary Adjustments Must Align with Loop Conditions**:
   - When using `while left < right`, ensure that you **retain potential candidates** within the search space by setting `right = mid` or `left = mid + 1`.
   - When using `while left <= right`, adjust boundaries in a way that **excludes** elements already considered, such as setting `right = mid - 1`.
4. **Preservation of Potential Candidates**:
   - In problems like **LC-287**, **LC-153**, and **LC-162**, it's crucial to **retain potential candidates** within the search space to ensure the correct element is found upon termination.
5. **Termination Guarantees**:
   - `while left < right` ensures that the search converges to a single element, which is the desired result.
   - `while left <= right` requires careful handling to avoid infinite loops and ensure that the termination condition correctly identifies the result.

Binary search based problems from Neetcode-150:

Binary Search (0 / 7)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Binary Search	Easy	■	Python
☐	☆	Search a 2D Matrix	Medium	■	Python
☐	☆	Koko Eating Bananas	Medium	■	Python
☐	☆	Find Minimum In Rotated Sorted Array	Medium	■	Python
☐	☆	Search In Rotated Sorted Array	Medium	■	Python
☐	☆	Time Based Key Value Store	Medium	■	Python
☐	☆	Median of Two Sorted Arrays	Hard	■	Python

---

2. SLIDING WINDOW TEMPLATE/CONCEPTS: | Updated 10/1/24

In any sliding window based problem we have two pointers. One right pointer whose job is to expand the current window and then we have the left pointer whose job is to contract a given window. At any point in time only one of these pointers move and the other one remains fixed.

Below is the main thought-process for solving sliding window problems :

1. We start with two pointers, left and right initially pointing to the first element of the string.
2. We use the right pointer to expand the window until we get a desirable window i.e. a window that contains all of the characters of T.
3. Once we have a window with all the characters, we can move the left pointer ahead one by one. If the window is still a desirable one we keep on updating the minimum window size (LC-76). Updateing result.
4. Repeat step-2 in loop until we reach to end of input array/string.

Below are the 4- STEPS to solve any sliding window problems :

STEP-1: Initialization - left, right (optional), result, hash/counter
STEP-2: Start iterating with right pointer for desired window using loop(for/while).
STEP-3: As we iterate, check if we're violating CONDITION for desired window. If yes, shrink window by moving left pointer using loop.
STEP-4: Keep capturing optimal result (max/min) using left/right pointers. Example: (right-left+1).

Lets see few sliding window problems. FOCUS on **TEMPLATE** [4-STEPS]

340: Longest substring with almost K distinct Characters

```
class Solution:
    def lengthOfLongestSubstringKDistinct(self, s, k):

        # 10/1/24: Below is very standard 4-step process for sliding window problems.
        # Step-1: Initialize. Three things, res, left and counter to maintain freq
        res = 0
        left = 0 # left pointer
        count = collections.Counter()

        # Step-2: Start iterating with right pointer for desired window using loop( for/while)
        for right in range(len(s)): # Right pointer
            count[s[right]] += 1

            # Step-3: As we iterate, check if we're violating CONDITION for desired window.
            # ( Its important to come up with logic for condition - when we need to start shrinking )
            # If yes, shrink window by moving left pointer using loop ( aka while replacements are > k)
            while len(count) > k:  # Because only two unique chars are allowed.
                count[s[left]] -= 1
```

```
                count[s[left]] -= 1
                if count[s[left]] == 0:
                    del count[s[left]]
                left += 1

            # Step-4: Keep capturing optimal solution (max/min) using left/right pointers.
            res = max(res, right - left + 1)

        return res
```

159: Longest substring with almost 2 distinct Characters : Same as above

```
class Solution(object):
    def lengthOfLongestSubstringTwoDistinct(self, s):

        # 10/1/24: Below is very standard 4 step process for sliding window problems.
        # Step-1: Initialize. Three things, res, left and counter to maintain freq
        res = 0
        left = 0 # left pointer
        count = collections.Counter()

        #  # Step-2: Start iterating with right pointer for desired window using loop( for/while)
        for right in range(len(s)): # Right pointer
            count[s[right]] += 1

            # Step-3: As we iterate, check if we're violating CONDITION for desired window.
            # ( Its important to come up with logic for condition - when we need to start shrinking )
            # If yes, shrink window by moving left pointer using loop ( aka while replacements are > k)
            while len(count) > 2:  # Because only two unique chars are allowed.
                count[s[left]] -= 1
                if count[s[left]] == 0:
                    del count[s[left]]
                left += 1

            # Step-4: Keep capturing optimal solution (max/min) using left/right pointers.
            res = max(res, right - left + 1)

        return res
```

424: Longest Repeating Character Replacement

```
class Solution(object):
    def characterReplacement(self, s, k):

        # 10/1/24: Below is very standard 4 step process for sliding window problems.
        # Step-1: Initialize. Three things, res, left and counter to maintain freq
        left = 0      # left pointer
        res = 0
        count = collections.Counter() # To maintain the frequency of each element as we go.

        # Step-2: Start iterating with right pointer for desired window using loop( for/while)
        for right in range(len(s)):
            count[s[right]] += 1 # Move right pointer.

            # Step-3: As we iterate, check if we're violating CONDITION for desired window.
            # ( Its important to come up with logic for condition - when we need to start shrinking )
            # If yes, shrink window by moving left pointer using loop ( aka while replacements are > k)
            while (right - left + 1) - max(count.values()) > k: # we could use if or while both here.
                count[s[left]] -= 1
                left += 1

            # Step-4: Keep capturing optimal solution (max/min) using left/right pointers.
            res = max(res, right-left+1)

        return res
```

713. Subarray Product Less than K

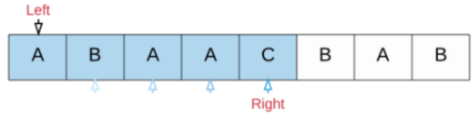
```
class Solution(object):
    def numSubarrayProductLessThanK(self, nums, k):

        # Conner case
        if k <= 1: return 0

        # 10/1/24: Below is very standard 4 step process for sliding window problems.
        # Step-1: Initialize. Three things, res, left and counter/prod to maintain freq/target-result
        left = 0
        res = 0
        prod = 1

        # Step-2: Start iterating with right pointer for desired window using loop( for/while)
        for right in range(len(nums)):
            prod *= nums[right]

            # Step-3: As we iterate, check if we're violating CONDITION for desired window.
            # If yes, shrink window by moving left pointer using loop ( aka while replacements are > k)
            while prod >= k:
                prod /= nums[left]
                left += 1

            # Step-4: Keep capturing optimal solution (max/min) using left/right pointers.
```

```
            # Step-4: Keep capturing optimal solution (max/min) using left/right pointers.
            # Basically right-left+1 will give number - product of all number in lesser subsets
            res += right-left +1
            print (left, right, res)
            #res = max(res, right-left+1)

        return res

#  Two pointer : O(n) time O(1) space:
```

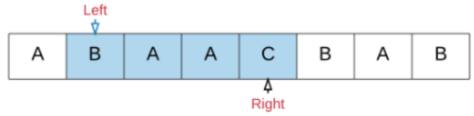
Minimum Window Substring : LC-76: Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window.

1. Initial State: Left and Right pointers are at index 0.

2. Moving the right pointer until the window has all the elements from string T. Record this desirable window.



3. Now move the left pointer. Notice the window is still desirable and smaller than the previous window.




```
class Solution(object):
    def minWindow(self, s, t):

        left = right = 0
        res = "#"

        src_sofar = collections.defaultdict(int)    # Will maintain count of target-element's presence in SOURCE.
        targert_freq = collections.Counter(t)       # Hash table to store char frequency from TARGET
        target_len = len(set(t))                    # Total number of unique chars we care from TARGET

        while right < len(s):

            # BLOCK-I : We find that window with all elements from target in given source.
            if s[right] in targert_freq :   # Block-1 ( expanding from right)
                src_sofar[s[right]] += 1    # s[j] is part of target, so src_sofar[s[j]] +=1
                if src_sofar[s[right]] == targert_freq[s[right]]:
                    target_len -= 1
            right += 1                  # Keep expanding Right-side until we don't find all target elements in source)


            # BLOCK-II: So we found desirable window, now reduce the size of window by moving left pointer
            while left <= right and target_len == 0:   # we need to keep shrinking from left up-till current right.
                                                       # "target_len == 0" means, we found all target elements already.

                if res == "#" or len(res) > len(s[left:right]):
                    res = s[left:right]

                if s[left] in targert_freq:                # Block-2 ( almost same as Block-1, just reducing window from left side)
                    src_sofar[s[left]] -= 1
                    if src_sofar[s[left]] == targert_freq[s[left]]-1:
                        target_len += 1
                left += 1

        return "" if res == "#" else res
```

LIttle Harder problems ..

Sliding Window Maximum: LC-239: You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
```

```
1  3  -1  -3  5 [3  6  7]      7
```

```
    def get_next_max(self, heap, left):
        while True:
            x,idx = heapq.heappop(heap)
            if idx >= left:
                heapq.heappush(heap, (x, idx)) #  add the max back to the heap - it could be relevant to future ranges.
                return -x, idx


    def maxSlidingWindow(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """

        if k == 0:
            return []

        heap = []
        for i in range(k):                      # Make first Window, and then keep sliding - left + 1, right + 1
            heapq.heappush(heap, (-nums[i], i))   # Pushing both value and index to MAX HEAP.

        result = []
        left, right = 0, k-1

        while right < len(nums):
            # Get. current max from heap and insert back max if idx was > left.
            x, idx = self.get_next_max(heap, left)
            result.append(x)

            # Increment/Move window and general push to element to maxheap.
            left, right = left + 1, right + 1
            if right < len(nums):
                heapq.heappush(heap, (-nums[right], right))   # General Push of next element to heap

        return result
```

Sliding Window (0 / 6)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Best Time to Buy And Sell Stock	Easy	■	Python
☐	☆	Longest Substring Without Repeating Characters	Medium	■	Python
☐	☆	Longest Repeating Character Replacement	Medium	■	Python
☐	☆	Permutation In String	Medium	■	Python
☐	☆	Minimum Window Substring	Hard	■	Python
☐	☆	Sliding Window Maximum	Hard	■	Python

3. Two Pointer Technique :

Its efficient algorithmic approach often used in problems involving arrays, strings, or linked lists. It involves using two pointers to traverse the data structure in a way that optimizes the solution, often reducing time complexity. Advantage : **Reduces Time Complexity**: By moving two pointers simultaneously, the technique often reduces the time complexity from **O(N^2) to O(N)**

This technique is versatile and can be used in a wide range of problems, including:

1. Finding pairs or triplets in an array that satisfy certain conditions (e.g., sum equals a target).
2. Merging two sorted arrays or lists.
3. Removing duplicates from a sorted array.
4. Checking palindromes in a string or linked list.
5. Partitioning arrays around a pivot.

LC-15: Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.
Notice that the solution set must not contain duplicate triplets.

Example 1:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& num) {

    vector<vector<int> > res;
    std::sort(num.begin(), num.end());

    for (int i = 0; i < num.size(); i++) {
```

```
        int current = num[i];
        int front = i + 1;
        int back = num.size() - 1;

        // Two pointer techinique ( instead of using two for-loops)
        while (front < back) {

            int sum = num[front] + num[back] + current;

            // Finding answer which start from number num[i]
            if (sum < 0)
                front++;

            else if (sum > 0)
                back--;

            else {
                vector<int> triplet = {num[i], num[front], num[back]};
                res.push_back(triplet);

                // Processing duplicates of Number 2
                // Rolling the front pointer to the next different number forwards
                while (front < back && num[front] == triplet[1])
                    front++;

                // Processing duplicates of Number 3
                // Rolling the back pointer to the next different number backwards
                while (front < back && num[back] == triplet[2])
                    back--;
            }

        }

        // Processing duplicates of Number 1
        while (i + 1 < num.size() && num[i] == num[i+1])
            i++;

    }

    return res;
    }
};
```

Two-pointer problems from Neetcode-150 list:

Two Pointers (0 / 5)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Valid Palindrome	Easy	■◀	Python
☐	☆	Two Sum II Input Array Is Sorted	Medium	■◀	Python
☐	☆	3Sum	Medium	■◀	Python
☐	☆	Container With Most Water	Medium	■◀	Python
☐	☆	Trapping Rain Water	Hard	■◀	Python

4. Prefix Sum :
++++++++++++

Applications of Prefix Sums

- Range Sum Queries: Quickly compute the sum of elements in a specific range. LC-303, 304, 307
- Counting Problems: Count the number of elements satisfying certain conditions within a range. - LC-560, 930, 325 , 974, 525
- String Problems: Compute cumulative character counts or other properties. 1170
- Dynamic Programming: Optimize state computations by reusing cumulative results. LC-53, 152, 1143

Tips for Leveraging Prefix Sums in LeetCode Problems

1. Understand the Problem Constraints:
   Determine if the problem involves multiple queries or requires frequent range calculations, which are ideal scenarios for prefix sums.
2. Preprocess Cumulative Information:
   Compute the prefix sum array or other cumulative structures during the preprocessing phase to enable efficient query handling.
3. Use Hash Maps for Enhanced Counting:
   Especially in **counting** problems, combining prefix sums with hash maps can optimize the counting of specific sum conditions.
4. Adapt to Multidimensional Data:
   For grid-based problems, consider 2D prefix sums to handle submatrix queries efficiently.
5. Combine with Other Techniques:
   Prefix sums can be integrated with dynamic programming, sliding windows, and other algorithmic strategies to solve complex problems.

Let's see the problems below :

238. Product of Array except self :

Approach:

- First, traverse the array from left to right, storing the prefix product up to each element.
- Then, traverse the array from right to left, multiplying each element of the result by the suffix product and updating the suffix product.
- This approach avoids division, ensuring the result is O(n) with O(1) extra space (excluding the output array).

```
def productExceptSelf(nums):

    n = len(nums)
    result = [1] * n  # Initialize the result array with 1s

    # Step 1: Calculate prefix products
    prefix_product = 1
    for i in range(n):
        result[i] = prefix_product
        prefix_product *= nums[i]

    # Step 2: Calculate suffix products and multiply to result array
    suffix_product = 1
    for i in range(n - 1, -1, -1):
        result[i] *= suffix_product
        suffix_product *= nums[i]

    return result
```

303. Range Sum Query - Immutable

How Prefix-Sum Applies: By precomputing the prefix sum array during initialization, each sumRange query can be answered in constant time O(1) by subtracting the prefix sums.

```
class NumArray:
    def __init__(self, nums: List[int]):
        self.prefix_sum = []
        current_sum = 0
        for num in nums:
            current_sum += num
            self.prefix_sum.append(current_sum)
        # Debug: Print the prefix sum array
        print(f"Prefix Sum Array: {self.prefix_sum}")

    def sumRange(self, left: int, right: int) -> int:
        if left == 0:
            range_sum = self.prefix_sum[right]
        else:
            range_sum = self.prefix_sum[right] - self.prefix_sum[left - 1]
        # Debug: Print the computation details
        print(f"Sum of elements from index {left} to {right}: {range_sum}")
        return range_sum
```

560. Subarray Sum Equals K

How Prefix-Sum Applies: Utilize a hash map to store the frequency of prefix sums. For each element, check if there's a prefix sum that, when subtracted from the current sum, equals k. This allows counting subarrays in linear time.

```
def subarray_sum(nums: List[int], k: int) -> int:
    count = 0
    current_sum = 0
    prefix_sums = {0: 1}  # Initialize with sum 0 occurring once

    for num in nums:
        current_sum += num
        if (current_sum - k) in prefix_sums:
            count += prefix_sums[current_sum - k]
        prefix_sums[current_sum] = prefix_sums.get(current_sum, 0) + 1
        # Debug: Print current sum and prefix_sums
        print(f"Current Sum: {current_sum}, Prefix Sums: {prefix_sums}, Count: {count}")

    return count
```

325. Maximum Size Subarray Sum Equals k

How Prefix-Sum Applies
Similar to problem 560, use a hash map to store the earliest occurrence of each prefix sum. For each index, check if there exists a prefix sum that allows a subarray summing to k.

```
def max_subarray_len(nums: List[int], k: int) -> int:
    prefix_sums = {0: -1}  # Initialize with sum 0 at index -1
    current_sum = 0
    max_len = 0

    for i, num in enumerate(nums):
```

```
        current_sum += num
        if (current_sum - k) in prefix_sums:
            max_len = max(max_len, i - prefix_sums[current_sum - k])
        # Only store the first occurrence to maximize subarray length
        if current_sum not in prefix_sums:
            prefix_sums[current_sum] = i
        # Debug: Print current sum, prefix_sums, and max_len
        print(f"Index: {i}, Num: {num}, Current Sum: {current_sum}, Prefix Sums: {prefix_sums}, Max Length: {max_len}")

    return max_len
```

974. Subarray Sums Divisible by K

How Prefix-Sum Applies
Similar to problem 560, use a hash map to store the frequency of prefix sums modulo k. If two prefix sums have the same modulo, the subarray between them is divisible by k.

```
def subarrays_div_by_k(nums: List[int], k: int) -> int:
    from collections import defaultdict
    prefix_mod = defaultdict(int)
    prefix_mod[0] = 1
    current_sum = 0
    count = 0

    for i, num in enumerate(nums):
        current_sum += num
        mod = current_sum % k
        # Handle negative numbers
        if mod < 0:
            mod += k
        if mod in prefix_mod:
            count += prefix_mod[mod]
        prefix_mod[mod] += 1
        # Debug: Print current sum, mod, prefix_mod, and count
        print(f"Index: {i}, Num: {num}, Current Sum: {current_sum}, Mod: {mod}, Prefix Mod: {dict(prefix_mod)}, Count: {count}")

    return count
```

930: Binary Subarray with Sum :

How Prefix-Sum Applies. (Looks similar to 560)
Use a hash map to track the frequency of prefix sums. For each index, determine if there's a prefix sum that allows a subarray sum equal to goal.

```
def num_subarrays_with_sum(nums: List[int], goal: int) -> int:
    from collections import defaultdict
    prefix_sum_counts = defaultdict(int)
    prefix_sum_counts[0] = 1
    current_sum = 0
    count = 0

    for i, num in enumerate(nums):
        current_sum += num
        if (current_sum - goal) in prefix_sum_counts:
            count += prefix_sum_counts[current_sum - goal]
        prefix_sum_counts[current_sum] += 1
        # Debug: Print current_sum, prefix_sum_counts, and count
        print(f"Index: {i}, Num: {num}, Current Sum: {current_sum}, Prefix Sum Counts: {dict(prefix_sum_counts)}, Count: {count}")

    return count
```

523. Continuous Subarray Sum

How Prefix-Sum Applies
Use prefix sums modulo k to identify if the same remainder has been seen before, indicating a subarray sum that is a multiple of k.

```
def check_subarray_sum(nums: List[int], k: int) -> bool:
    prefix_mod = {0: -1}  # Initialize with modulo 0 at index -1
    current_sum = 0

    for i, num in enumerate(nums):
        current_sum += num
        if k != 0:
            current_mod = current_sum % k
        else:
            current_mod = current_sum
        if current_mod in prefix_mod:
            if i - prefix_mod[current_mod] >= 2:
                # Debug: Print the valid subarray
                print(f"Valid subarray found from index {prefix_mod[current_mod]+1} to {i}")
                return True
        else:
            prefix_mod[current_mod] = i
        # Debug: Print current sum, current_mod, and prefix_mod
```

```
            print(f"Index: {i}, Num: {num}, Current Sum: {current_sum}, Current Mod: {current_mod}, Prefix Mod: {prefix_mod}")

    return False
```

---

525. Contiguous Array

Approach:

- Convert all 0s in the array to -1s so that we are looking for subarrays with a sum of 0. This means the number of -1s (original 0s) equals the number of 1s.
- As we iterate through the array, we maintain a running prefix sum. Whenever we encounter the same prefix sum at two different indices, it means that the subarray between these two indices has a sum of 0 and contains equal numbers of 0s and 1s.
- Use a hashmap to store the first occurrence of each prefix sum. If a prefix sum is seen again, calculate the length of the subarray and update the maximum length.

```
def findMaxLength(nums):
    # Replace 0s with -1s to use prefix sum technique
    prefix_sum_map = {0: -1}  # To handle the case when the subarray starts from index 0
    prefix_sum = 0
    max_len = 0

    for i, num in enumerate(nums):
        # Update prefix sum
        prefix_sum += 1 if num == 1 else -1

        # Check if the current prefix sum has been seen before
        if prefix_sum in prefix_sum_map:
            # If yes, calculate the length of the subarray
            max_len = max(max_len, i - prefix_sum_map[prefix_sum])
        else:
            # If not, store the first occurrence of this prefix sum
            prefix_sum_map[prefix_sum] = i

    return max_len
```

Explanation:

- Prefix Sum: We iterate over the list, adjusting the prefix_sum by +1 for 1s and -1 for 0s.
- HashMap (prefix_sum_map): It stores the first occurrence of each prefix_sum. If the same prefix sum appears again, it means there's a subarray with equal 0s and 1s between the two indices.
- Result Calculation: Each time we find a matching prefix sum, we update max_len with the length of the subarray.

Time Complexity:

- Time: O(n) — We traverse the array once.
- Space: O(n) — In the worst case, we store every prefix sum in the hashmap.

---

5. RECURSION CONCEPTS
++++++++++++++++++++++

- Recursion is a powerful programming technique where a function calls itself to solve smaller instances of the same problem.
- Recursion can simplify complex problems by breaking them down into more manageable sub-problems. However, it's essential to define a base case to prevent infinite recursion and ensure that each recursive call progresses towards this base case.

Key Components of Recursion :

1. Base Case: The condition under which the recursive function stops calling itself. Without a base case, recursion would continue indefinitely, leading to a stack overflow.
2. Recursive Case: The part of the function where it calls itself with a modified argument, moving towards the base case.

Best Practices When Using Recursion :

1. Always Define Base Cases: Ensure that every recursive function has one or more base cases to terminate recursion.
2. Progress Towards the Base Case: Modify the input in each recursive call so that it eventually reaches the base case.
3. Be Mindful of Stack Depth: Python has a recursion limit (typically 1000). Deep recursion can lead to a RecursionError. For problems requiring deep recursion, consider iterative solutions or techniques like tail recursion optimization (though Python doesn't support tail call optimization natively).
4. Use Memoization for Optimization: For problems with overlapping subproblems (like Fibonacci), memoization can cache results of recursive calls to improve efficiency.

Time/Space Complexity :
In general recusion has exponential time complexity. BUT not always. For large n, consider using iterative methods or memoization to optimize performance. We can improve that time complexity by Memoization/ DP

Lets START with simple examples:

Factorial Calculation : [Recursion w/ Base case]
Calculate the factorial of a non-negative integer n

```
def factorial(n):
    if n == 0 or n == 1:  # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive case
```

print(factorial(5)) # Output: 120

Time Complexity: O(n) (Linear time due to n recursive calls).

Space Complexity: O(n) (Space used by the recursion stack).

Fibonacci Sequence: [Recursion w/ Base case]
Find the n-th Fibonacci number, where each number is the sum of the two preceding ones, starting from 0 and 1.

```
def fibonacci(n):
    if n == 0:  # Base case 1
        return 0
    elif n == 1:  # Base case 2
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case
```

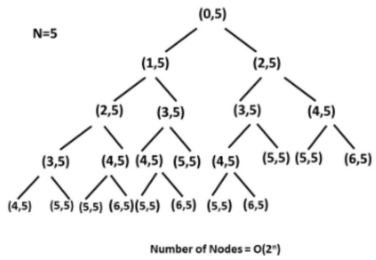
Time Complexity=O(2^n) - (Exponential time due to repeated recursive calls).
Space Complexity=O(n) - (Space used by the recursion stack).

Climbing Stairs: [Based on fibonacci]

```
class Solution:

    def climbStairs(self, n: int) -> int:
        return self.climb_Stairs(0, n)

    def climb_Stairs(self, i: int, n: int) -> int:
        if i > n:
            return 0
        if i == n:
            return 1
        return self.climb_Stairs(i + 1, n) + self.climb_Stairs(i + 2, n)
```

Time complexity : O(2 ^n). Size of recursion tree will be 2^n

Recursion tree for n=5 would be like this:



Optimised via DP :

```
class Solution(object):
    def climbStairs(self, n):

        # Since we can take one step or two step behind  to reach top, so we need to look back two places behind.
        # So lets initialize dp[0] = 1, dp[1] = 1

        if n < 1 :
            return 0

        dp = [0] *(n+1)
        dp[0] = 1
        dp[1] = 1

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

Time complexity : O(n). Size of recursion tree can go up to n.
Space complexity : O(n). The depth of recursion tree can go up to n.

Unique Path Counts (LC-62) :

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:

        if m == 1 or n == 1:
            return 1

        return self.uniquePaths(m - 1, n) + self.uniquePaths(m, n - 1)
```

Bianry Search : [Recursion w/ Base case]

```
def binary_search(arr, target, low, high):
```

```
def binary_search(arr, target, low, high):

    if low > high:  # Base case: not found
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid + 1, high)  # Search right half
    else:
        return binary_search(arr, target, low, mid - 1)  # Search left half
```

sorted_list = [1, 3, 5, 7, 9, 11]
print(binary_search(sorted_list, 7, 0, len(sorted_list) - 1)) # Output: 3

Tower of Hanoi [Recursion w/ Base case]
Moving n disks from the source rod to the destination rod using an auxiliary rod.

```
def tower_of_hanoi(n, source, auxiliary, destination):

    if n == 1:  # Base case
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n - 1, source, destination, auxiliary)  # Move n-1 disks to auxiliary
    print(f"Move disk {n} from {source} to {destination}")  # Move nth disk to destination
    tower_of_hanoi(n - 1, auxiliary, source, destination)  # Move n-1 disks from auxiliary to destination
```

Explanation

- Base Case: Move a single disk directly from the source to the destination.
- Recursive Case:
    - Move n-1 disks from the source to the auxiliary rod.
    - Move the n-th disk from the source to the destination.
    - Move the n-1 disks from the auxiliary rod to the destination.

tower_of_hanoi(3, 'A', 'B', 'C')

Output:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Permutations of a String (TODO):
NOTE: I have done using backtracking, but this is purely Recusion (little different).

```
def permutations(s):

    if len(s) == 0:  # Base case: empty string
        return ['']
    else:
        result = []
        for i in range(len(s)):
            char = s[i]
            remaining = s[:i] + s[i+1:]
            for p in permutations(remaining):  # Recursive call
                result.append(char + p)
        return result
```

Explanation

- Base Case: The permutation of an empty string is a list containing an empty string.
- Recursive Case:
    - Iterate through each character in the string.
    - Recursively find permutations of the remaining string after removing the current character.
    - Prepend the current character to each permutation of the remaining string.

print(permutations("abc"))
Output: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

Generate Parentheses LC-22 (TODO) : Nested Recursive Calls with Multiple Returns
Note: Solving using Recursion purely (NOT using backtracking technique)

```
def generate_parentheses(n):

    if n == 0:
        return ['']
```

```
    result = []
    for c in range(n):
        for left in generate_parentheses(c):
            for right in generate_parentheses(n - 1 - c):
                result.append('(' + left + ')' + right)
    return result
```

Explanation

- Base Case: If n is 0, return a list with an empty string.
- Recursive Case:
  - Iterate through all possible splits of n-1 into c and n-1-c.
  - For each split, generate all combinations of left and right parentheses.
  - Combine them with a pair of parentheses around the left part.

print(generate_parentheses(3))
Output: ['((()))', '(()())', '(())()', '()(())', '()()()']

Files in given directory : TESLA - [Recursion w/ different return]
Note directory many have sub-directories (which may have files) and files.

```
class Files:

    def __init__(self):
        self.files_list = []  # List to store all files

    def get_file_list(self, path):
        """
        Recursively gets a list of all files within the specified directory.
        Args: path (str): The path to the directory.
        Returns:  list: A list of file paths.
        """

        files = os.listdir(path)

        for f in files:
            full_path = os.path.join(path, f)

            if os.path.isfile(full_path):
                self.files_list.append(full_path)
            else:
                self.get_file_list(full_path)  # Recursively traverse subdirectories

        # return list to recusive calls. This is returning condition. No base-condition here.
        return self.files_list
```

Time Complexity:
$O(N)$ (where N is the total number of files and directories).

Space Complexity:
O(D+F) (where D is the depth of the directory tree, and F is the total number of files).

Other OS functions : FYI

- os.path.getmtime(path) // Return the time of last modification of path.
- os.path.getsize(path) // Return the size, in bytes, of path
- os.path.isfile(path)
- os.path.isdir(path)
- os.path.join(path, *paths)

Subset (used recusion with backtracking technique)

Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

```
class Solution(object):
    def subsets(self, nums):

        res = []
        if len(nums) < 1:
            return 0
        self.dfs(nums,0, [], res)
        return res

    def dfs(self, nums, start, cur, res):
        res.append(cur[:])
        #return

        for i in range(start,len(nums)):
            #print "# Still looping # for i = ", i
            cur.append(nums[i])
            self.dfs(nums, i+1, cur, res)
            cur.pop()
```

NOTE: Something different. Even though recusive calls, but no base condition/ returning condition as its termainates naturally when loop completes.

- Base Condition (Implicit): the recursion stops when start reaches or exceeds the length of the input list nums. This is indirectly enforced by the for loop in the dfs method:

for i in range(start, len(nums)).

- Return Condition for Recursion:
  There is no explicit return condition (return) inside the dfs method, as the recursion terminates when the loop completes. The method continues to explore subsets by recursively adding elements to cur and removing the last element (cur.pop()) as part of the backtracking process.

Visualization of the Recursion Tree

```
Start at 0: []
├── Include 1: [1]
│   ├── Include 2: [1, 2]
│   │   ├── Include 3: [1, 2, 3]
│   │   └── Exclude 3: [1, 2]
│   └── Exclude 2: [1]
│       ├── Include 3: [1, 3]
│       └── Exclude 3: [1]
├── Exclude 1: []
│   ├── Include 2: [2]
│   │   ├── Include 3: [2, 3]
│   │   └── Exclude 3: [2]
│   └── Exclude 2: []
│       ├── Include 3: [3]
│       └── Exclude 3: []
```

Analysing Time-complexity :

Number of Subsets:: For a list of length n, there are 2^n (2 to the power of n) possible subsets. This is because each element has two choices: either include it in a subset or exclude it.
Time to Generate Each Subset: Each time a subset is appended to res, a copy of the current subset cur is made.

Time Complexity = O(nx 2^n)

Also Note :

- Permutations:
  Number of Results: n!
  Time Complexity: O(n × n!)
- Subsets (Power Set):
  Number of Results: 2^n
  Time Complexity: O(n × 2^n)

Max Depth of Tree: [Recursion w/ Base case]
[PostOrder Traversal] Left-child, Right-child, Root/Node -
RETURN manipuated node/root

```
class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        # 10/8/24 : Same style as LC-543.

        if root is None:      # BASE CASE
            return 0

        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        return max(left_depth, right_depth) + 1
```

Time Complexity=O(N) - Since each of the N nodes in the tree is visited once.

Space Complexity
O(logN) # if the tree is balanced
O(N) # if the tree is skewed

Path Sum : [Recursion w/ Base case]
[PreOrder Traversal] Root/node, Left-child, Right-child.
RETURN left/right recursive calls and just make those calls.

[TODO] : Why not just make recursive calls to left-child/right-child and why we have to return as below ?

```
class Solution(object):

    def hasPathSum(self, node, sum):

        if node == None:    # BASE CASE
            return

        if node.left == None and node.right == None:
            return node.val == sum

        return self.hasPathSum(node.left, sum - node.val) or self.hasPathSum(node.right, sum - node.val)
        # self.hasPathSum(node.left, sum - node.val)
        # self.hasPathSum(node.right, sum - node.val)
```

Time Complexity=O(N) - Since each of the N nodes in the tree is visited once.

Space Complexity
O(logN) # if the tree is balanced
O(N) # if the tree is skewed

Similarly : Same Tree

```
class Solution(object):

    def isSameTree(self, t1, t2):

        if t1 is None and t2 is None:    # BASE CASE
            return True

        elif t1 is None or t2 is None:     # BASE CASE
            return False

        elif t1.val != t2.val:                  # Returning condition
            return False

        return self.isSameTree(t1.left, t2.left) and self.isSameTree(t1.right, t2.right)
```

Validate Binary Search Tree : [Recursion w/ Base case]
[InOrder Traversal]: Simple recusrive calls with Base case. NO need to **RETURN** recusive calls unlike before.

```
class Solution(object):
    def isValidBST(self, root):

        self.prev = -float('inf')   # define -float('inf')
        self.flag = True
        self.inOrder(root)
        return self.flag

    def inOrder(self, root):

        if root is None:        # BASE CASE
            return

        self.inOrder(root.left)
        if root.val <= self.prev:
            self.flag = False
        self.prev = root.val
        self.inOrder(root.right)
```

Similarly : Kth smallest elemnt in BST
NO need to **RETURN** Recusive calls unlike before.

```
class Solution(object):

    def kthSmallest(self, root, k):

        self.res = float('-inf')
        self.k = k
        self.helper(root)
        return self.res

    def helper(self, root):
        if root is None:      # BASE CASE
            return

        self.helper(root.left)
        self.k -= 1
        if self.k == 0:
            self.res = root.val
            return
        self.helper(root.right)
```

---

6. BACKTRACKING: (Recursion with limitations)

Forward and backward Recursion strategies (examples) :

```
result = 0
def max_depth_forward(root,curr_depth):
    if root.left == None and root.right == None:
        result = max(result,curr_depth)

    max_depth(root.left,depth+1)
    max_depth(root.right,depth+1)

def max_depth_backward(root)--> int:
    if root.left == None and root.right == None:
            return 1

    value_left = max_depth_backward(root.left)
    value_right = max_depth_backward(root.right)
```

```
60
61      return max(value_left,value_right)
62
63
```

NOTE-1: Recusion is father of Dynamic Programming, where we save states in array/matrix to avoid future recusrive calls & hence provide optimised solutions. This is Memoization or Top-Down Approach.

-Recursion is used by multiple algorithm:

1. Backtracking - Find all possible solutions BUT it aborts if sub-solution is invalid and backtracks.
2. DFS (Trees & Graphs) - Most tree problems uses DFS. Graph (2D grid) - No. of Islands, Flood Fills, Clone graphs.
   Recursion implementation must use Base case for exit. Recusrion internally uses stack.
   [Just FYI: Iterative Apprach used deque like BFS - finding shortest path.]

NOTE-2: DP is great for optimization, however it might not be useful to optimise Backtracking problem in below scenarios:

- No Overlapping Subproblems: Problems such as Palindrome Partitioning (LeetCode 131) and Subsets (LeetCode 78) involve unique recursive paths without revisiting the same subproblems, negating the benefits of DP.
- Unique Paths: Problems like Permutations (LeetCode 46) and Generate Parentheses (LeetCode 22) involve unique, non-overlapping subproblems, making DP unnecessary.
- Complex State Representation: Challenges like Sudoku Solver (LeetCode 37) and N-Queens (LeetCode 51) have intricate state spaces that are difficult to represent for DP, favoring backtracking instead.

Backtracking Though process:

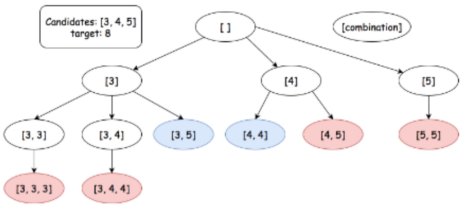
1. Need to choose (& later unchoose aka Backtrack)
2. DFS call - How to choose next options/positions. Based on that loop gets decised & hence next DFs calls.
3. Unchoose (if that path doesn't work).
   Note: When condition is met -append current ans to the result and return.

Lets see few classic questions to understand backtracking concepts:

LC-39: Combination Sum :

Return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order. The same number may be chosen from candidates an unlimited number of times. Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]

The idea is that it incrementally builds candidates to the solutions, and once we find the current combination is not valid, we backtrack and try another option.



An important detail on choosing the next number for the combination is that we select the candidates in order, where the total candidates are treated as a list.
Once a candidate is added into the current combination, we will NOT look back to all the previous candidates in the next explorations. This helps to avoid duplication combinations.

```
class Solution(object):
    def combinationSum(self, nums, target):

        res = []
        self.backtrack(nums, target, 0, [], res)
        return res

    def backtrack(self, nums, target, start, cur, res):

        if target == 0:
            res.append(cur[:])
            return     # return and then pop ( which is backtracking)

        if target < 0:  # Important
            return  # Prune the search if the target goes negative

        for i in range(start, len(nums)):
            cur.append(nums[i])
            self.backtrack(nums, target - nums[i], i, cur,  res)  # We don't increment i since we can reuse unlike LC-40
            cur.pop()
```

LC-40. Combination Sum II
Same logic and decision-tree diagram for explanation as above, just need to avoid duplicates AND same numbers can't be used twice.

```
class Solution(object):
    def combinationSum2(self, candidates, target):

        candidates.sort()  # Sort to handle duplicates    <===
        res = []
        self.dfs(candidates, target, 0, [], res)
        return res
```

```
    def dfs(self, candidates, target, start, curr, res):

        if target == 0:
            res.append(curr[:])
            return

        if target < 0:  # Important
            return  # Prune the search if the target goes negative

        for i in range(start, len(candidates)):
            # Skip duplicates  <===
            if i > start and candidates[i] == candidates[i - 1]:
                continue

            curr.append(candidates[i])
            self.dfs(candidates, target - candidates[i], i + 1, curr, res)  # i+1 because we can't reuse same number like LC-39
            curr.pop()
```

Combination Sum III

Again same decision-tree diagram and logic for explanation.

```
class Solution(object):
    def combinationSum3(self, k, n):

        # Yay ! did myself :)

        nums = [1,2,3,4,5,6,7,8,9]
        res = []
        self.dfs(nums,k,n,0,[], res)
        return res

    def dfs(self, nums, k, target, start, curr, res):

        if len(curr) == k and target == 0:
            res.append(curr[:])    # curr[:] creates a shallow copy via reference to those objects.
            return
        if target < 0:
            return

        for i in range(start, len(nums)):
            curr.append(nums[i])
            self.dfs(nums, k, target-nums[i], i+1, curr, res )
            curr.pop()
```

LC-78: Subset

Same logic as combination sum, we need to build decision tree and choose. since we always move forward in terms of our choices we don't collect duplicates.

```
   nums = [ 1,2,3]
   Output: [ [],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3] ]

                     []

            [1]          [2]       [3]

         [1,2] [1,3]    [2,3]

       [1,2,3]
```

```
class Solution(object):
    def subsets(self, nums):

        if len(nums) < 1:
            return 0

        res = []
        self.dfs(nums,0, [], res)   # Must 4 parameters ( Generic Case):  num, start, curr, res
        return res

    def dfs(self, nums, start, cur, res):
        res.append(cur[:])   No if condition, since we need all possibilities unlike LC-46. Also no return
        #return

        for i in range(start,len(nums)):
            cur.append(nums[i])
            self.dfs(nums, i+1, cur, res)
            cur.pop()   # backtrack
```

LC-90: Subset-II

Same as above, Just with duplicate inputs. So we need to sort and skip them. However we keep moving ahead with next numbers so i+1 (don't need to resue same number).

```
class Solution(object):
    def subsetsWithDup(self, nums):

        # To handle duplicates correctly, we need to sort the input
        nums.sort() # <===
        res = []

        # Instead of returning 0, return [[]] for an empty input
        if len(nums) < 1:
            return [[]]

        self.backtrack(nums, 0, [], res)
        return res

    def backtrack(self, nums, start, cur, res):
        res.append(cur[:])

        for i in range(start, len(nums)):
            if i > start and nums[i] == nums[i - 1]: # Skip duplicates  <===
                continue

            cur.append(nums[i])
            self.backtrack(nums, i + 1, cur, res)
            cur.pop()
```

IMPORTANT: Few thoughts :

Firstly, In problems like Subset or Combinations (37), how come we are not getting duplicates in output even without using set(). Its the nature of algorithm which making recursive calls in forward direction inside for loop..

```
How Duplicates are Avoided: ( Problems Subset LC-78, and Combinations LC-77 )
Ordered Traversal: The DFS starts from a specific start index and moves forward without revisiting previous indices.
When iterating through numbers, we always move forward (i + 1) in the recursive call. This ordered traversal ensures that each
combination is generated in lexicographical order and no number is reused in the current combination.

Strict Length Check: The recursion stops when the length of the current combination (curr) reaches the desired length k. At this point,
the current combination is added to the results. Since we only add combinations of the correct length, no partial or incorrect duplicates
are introduced.

Backtracking: After each recursive call, the last element added to curr is removed (curr.pop()). This backtracking step ensures that
all combinations that include the last element are explored before removing it and moving on to the next possible element.
```

Secondly its thoughful - These are recusrive calls still base case are very different then regular recusrive calls. Here is base case is hit when we are out of choices.

Next try solving few problems : LC 46, 47, 78, 39, 40

Permutation

```
        Input: nums = [1,2,3]
        Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]


             [1]                   [2]                  [3]

          [1,2], [1,3]          [2,1], [2,3]        [3,1]     [3,2]

        [1,2,3]  [1,3,2]   [2,1,3]   [2,3,1]    [3,1,2]       [3,2,1]
```

```
class Solution(object):

    def permute(self, nums):
        res = []
        if len(nums) == 0:
            return res  # Empty list returns an empty result

        self.dfs(nums, [], res)
        return res

    def dfs(self, nums, curr, res):
        if len(curr) == len(nums):
            res.append(curr[:])
            return

        for i in range(len(nums)):  # No need for a `start` index here

            if nums[i] in curr:     # <===
                continue  # Skip if the number is already in the current permutation

            curr.append(nums[i])
            self.dfs(nums, curr, res)
            curr.pop()
```

NOTE: Why `start` Isn't Needed unlike subset/combination-sum problems: Becoz order matters, every element needs to be considered for every position, regardless of their

position in the original list.

Also Handling Used Elements: Instead of using a start index, you track which elements have been used in the current permutation to avoid repeats.

However In Combinations and Subsets
Purpose of start: Ensures that each element is considered only once and in a specific order to avoid duplicate selections.

How It Works: By starting the loop from the start index, you prevent reusing elements that have already been considered in previous recursive calls.

EASIER Way to solve using SWAP & TRACK method - Both LC-46,47. Just use set() for 47

```
class Solution(object):

    def permute(self, nums):
        res = []
        self.dfs(nums, 0, res)
        return res

    def dfs(self, nums, start, res):
        if start == len(nums):
            res.append(nums[:])

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]     # choose i for start position
            self.backtrack(nums, start+1, res)              # Incrementing start means: Considering options of i for next position
            nums[start], nums[i] = nums[i], nums[start]     # unchoose after return - backtracking


    # Idea is - Filling in available ( or remaining) options for different positions ( as we move from left to right)
```

Generate Parenthesis:

```
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        # 10/24/23: Solved this completely by myself and more simplified approach.
        res = []
        self.bt( n, [], res)
        return res

    def bt(self, n, curr, res):
        choices = ['(', ')']

        if len(curr) == 2*n:
            if self.valid(curr):
                res.append(''.join(curr[:]))
            return

        for c in choices:
            curr.append(c)
            self.bt(n, curr, res)
            curr.pop()  # backtrack

    def valid(self, string):  # Exactly same as LC-20

        stack = []
        map = {'(':')'}
        for s in string:
            if s == '(':
                stack.append(s)
            elif not stack or s != map[stack.pop()]:
                return False
        return not stack
```

Backtracking problems from Neetcode-150 list:

Backtracking (0 / 9)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Subsets	Medium	■	Python
☐	☆	Combination Sum	Medium	■	Python
☐	☆	Permutations	Medium	■	Python
☐	☆	Subsets II	Medium	■	Python
☐	☆	Combination Sum II	Medium	■	Python
☐	☆	Word Search	Medium	■	Python
☐	☆	Palindrome Partitioning	Medium	■	Python
☐	☆	Letter Combinations of a Phone Number	Medium	■	Python

☐	☆	N Queens	Hard	■	Python

7. DYNAMIC PROGRAMMING: Updated 9/16/24:

Method-1: Bottom-up (Tabulation) :
Also Called as `Dynamic Programming` which has initialization of dp array and then compute tabulation (aka dp) array/matrix within for loop- See LCS, LIS, LPS problem where we build matrix. Also 62, 63, 64, 53, 91, 70. Solved via Iterative way using loop

NOTE: Mostly we use DP[Bottom-Up Approach] by initializating DP array &
not Top-Down (Recursion w/ Memoization) approach.

- Decode Ways - https://leetcode.com/problems/decode-ways/ [Fibonacci Pattern]
- Climbing Stairs - https://leetcode.com/problems/climbing-stairs/ [Fibonacci Pattern]
- House Robber - https://leetcode.com/problems/house-robber/ [Fibonacci Pattern]
- Unique Paths - https://leetcode.com/problems/unique-paths/ [LC-62]
- Unique Path II - https://leetcode.com/problems/unique-paths-ii/ [LC-63]
- Unique Path III - https://leetcode.com/problems/unique-paths-iii/description/ [why this is DFS and not DP ? Think]
- Minimum Path Sum - https://leetcode.com/problems/minimum-path-sum/description/ [LC-64]
- Jump Game - https://leetcode.com/problems/jump-game/ [Use Greedy, than DP. See Neetcode on Youtube]
- Coin Change - https://leetcode.com/problems/coin-change/
- LIS - Longest Increasing Subsequence - https://leetcode.com/problems/longest-increasing-subsequence/ [DP, Neetcode/Tushar Roy]
- LCS - Longest Common Subsequence - https://leetcode.com/problems/longest-common-subsequence/description/ [**2D-DP array**, Neetcode/Tushar, READ Editorial]
- Longest Palindrome Subsequence https://leetcode.com/problems/longest-palindromic-subsequence/description/ [Same as LCS, **2D-DP array**]
- Edit distance https://leetcode.com/problems/edit-distance/description/ [**2D-DP array**]
- Word Break Problem - https://leetcode.com/problems/word-break/
- Combination Sum - https://leetcode.com/problems/combination-sum-iv/
- Maxmium Subarray - https://leetcode.com/problems/maximum-subarray/description/
- Maximum Product Subarray - https://leetcode.com/problems/maximum-product-subarray/description/

Method-2: Top-down (Memoization) :
Its `Recursion with Memoization` . To avoid re-calculating solutions to solved-subProblems. Mostly recusion based solutions with base case.

FYI: **Converting TopDown to BottomUp**:

1. Convert base constion of recursive calls to initialization of dp array.
2. Recursive calls changes to tabulation(aka dp) array/metrix within for loop.

Unique Path : LC-62

Classic Recursion :

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        if m == 1 or n == 1:
            return 1

        return self.uniquePaths(m - 1, n) + self.uniquePaths(m, n - 1)
```

NOTE: Problem sounds like exploring all paths and getting count, so we can use backtacking aka modified recursion, but recusion will be inefficient for large inputs due to its exponential time complexity and redundant calculations. Implementing Memoization significantly improves performance by caching intermediate results, effectively transforming the solution into a dynamic programming approach. Since this problem can be broken down into multiple sub-solutions so we can solve in O(m+n) using DP, rather than O(2 ^ (m+n)) which is exponential time.

Bottom-up Apprach:

```
class Solution(object):

    def uniquePaths(self, m, n):

        # 10/24/23:
        # First we can initialise first row and col - it will bbe all one
        # because no of ways to reach any col on first row is 1.
        # Similarly all rows of first column will bbe 1 for same reason.

        dp = [[1 for _ in range(n)] for _ in range(m)]
        print dp

        for i in range(1,m):
            for j in range(1, n):

                dp[i][j] = dp[i][j-1]+dp[i-1][j]

        return dp[-1][-1]
```

Unique Path-II : LC-63

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m,n=len(obstacleGrid),len(obstacleGrid[0])

        def countPath(i,j):
            if i>=m or j>=n or obstacleGrid[i][j]==1 :
                return 0
            if i==m-1 and j==n-1:
```

```
                if obstacleGrid[i][j]==1:
                    return 0
                return 1
            return countPath(i+1,j) + countPath(i,j+1)

        return countPath(0,0)
```

Bottom-up :

```
class Solution(object):
    def uniquePathsWithObstacles(self, obstacleGrid):

        N, M = len(obstacleGrid), len(obstacleGrid[0])
        dp = [[1 for x in range(M)] for y in range(N)]

        if obstacleGrid[0][0] == 1:
            return 0 # Because we cannot start since starting point itself is obstacle.

        for i in range(N):
            for j in range(M):

                if obstacleGrid[i][j] == 1:  # If we find obstacle
                    dp[i][j] = 0 # Mark that cell as 0 in dp matrix. It reduces path count for next futher cells - that path was eleminated

                else:
                    if i == 0:   # Basically for first row, if any col is blocked then successive columns in that first row also get blocked
                        dp[i][j] = dp[i][j - 1]

                    elif j == 0: # Similarly for first col, if above row was blocking, then all below are blocked too.
                        dp[i][j] = dp[i - 1][j]

                    else:
                        dp[i][j] = dp[i][j - 1] + dp[i - 1][j]

        return dp[-1][-1]

    # Note in these problems first row and first col are special.
    # First row gets all values from its immediate left cell.
    # First Col gets all values from its immediate upper cell.
```

Unique Paths-III : LC-980

Here we use recursion but we can't optimise using DP, because subsolution will not be re-used in future computation as problem says - walk over every non-obstacle square exactly once. Its straight forward recusion / dfs problem looking into all 4 directions.

```
IMPORTANT NOTE: 10/5/24
---------------

LC-63 focuses on counting all possible paths from start to end while avoiding obstacles, with movement restricted to right and down.
Dynamic Programming is the optimal approach due to its efficiency in handling overlapping subproblems.

LC-980 requires finding paths that cover every non-obstacle square exactly ONCE, with movement allowed in all four directions.
Requires Backtracking or Depth-First Search (DFS) with pruning because it involves exploring all possible paths that satisfy the unique visit
condition.

Understanding the problem requirements and constraints is crucial in determining the most suitable algorithmic approach. While both
problems deal with unique paths, their differing conditions necessitate distinct strategies to solve them effectively.
```

Min Path Sum: LC-64

```
class Solution(object):

    def minPathSum(self, grid):

        if not grid:
            return

        r, c = len(grid), len(grid[0])
        dp = [[0 for _ in xrange(c)] for _ in xrange(r)]

        dp[0][0] = grid[0][0]

        for i in xrange(1, r):  # For 0th Col, update row values. Basically update leftmost row.
            dp[i][0] = dp[i-1][0] + grid[i][0]

        for i in xrange(1, c):  # For 0th row, update all col values. Basically update top row
            dp[0][i] = dp[0][i-1] + grid[0][i]

        for i in xrange(1, len(grid)):
            for j in xrange(1, len(grid[0])):

                dp[i][j] = min( dp[i-1][j], dp[i][j-1] )  +   grid[i][j]   # Update all cells except top row and left most column.

        return dp[-1][-1]
```

```
        # Above one is O(m*n) space
```

Word Break :

NOTE: `Word break-II`, is great problem which shows how to use sub-solution to solve overall problem via DP. Next how we can use Backtracking to get all possible words. This problem uses concept of both DP and later seprately Backtracking to find all possible words.

```
class Solution(object):
    def wordBreak(self, s, wordDict):

        # https://www.youtube.com/watch?v=Sx9NNgInc3A

        dp = [False] * (len(s) + 1)
        dp[len(s)] = True

        for i in range(len(s) - 1, -1, -1):
            for w in wordDict:
                if (i + len(w)) <= len(s) and s[i : i + len(w)] == w:
                    dp[i] = dp[i + len(w)]
                if dp[i]:
                    break

        return dp[0]
```

Decode Ways: | Similar to fibonacii series (but not 100%)

```
class Solution(object):
    def numDecodings(self, s):
        """
        :type s: str
        :rtype: int
        """

        # We need to build the sub-solutions for various combinations and try it out.
        # since we have two digits possibility - which can be translated to single letter ( Z - 26),
        # so we need to look two places behind for sub-soluions.
        # So we take array dp for storing results, then we need to intialise dp[0], dp[1] as 1.

        # IMP : This problem is based on concept of Fibonacci Numbers. See DP hand-written notes.
        # Same concept can be applied to:
        # 1. Stairs Climbing: LC-70, 746
        # 2. House thief: LC-198
        # 3. Decode Ways: LC-91

        dp = [0] * (len(s)+1)
        dp[0] = 1
        dp[1] = 1

        if int(s[0]) == 0 or len(s) < 1:
            return 0

        for i in range(2, len(s)+1):

            if int(s[i-1]) > 0:
                dp[i] = dp[i-1]
            if 9 < int(s[i-2:i]) <= 26:
                dp[i] += dp[i-2]
        print (dp)
        return dp[-1]
```

Climbing Stairs | Pure Fibonacci Series

```
class Solution(object):
    def climbStairs(self, n):

        # Build sub solution array - dp
        if n < 1 :
            return 0

        dp = [0] *(n+1)
        dp[0] = 1
        dp[1] = 1

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

Min Cost Climbing Stairs: | Pure Fibonacii Series.

```
class Solution(object):
    def minCostClimbingStairs(self, cost):

        dp = [0]*(len(cost))

        dp[0] = cost[0]
        dp[1] = cost[1]

        for i in range(2,len(cost)):
            dp[i] = min(dp[i-2]+cost[i], dp[i-1]+cost[i])

        return min(dp[-2], dp[-1])  # OR simply return dp[-1]
```

Coin change

```
class Solution(object):
    def coinChange(self, coins, amount):
        """
        :type coins: List[int]
        :type amount: int
        :rtype: int
        """
        # 9/17/24: https://www.youtube.com/watch?v=H9bfqozjoqs
        # -------
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for a in range(1, amount + 1):
            for c in coins:
                if a - c >= 0:
                    dp[a] = min(dp[a], 1 + dp[a - c])
        return dp[amount] if dp[amount] != amount + 1 else -1
```

Jump Game :

```
class Solution(object):
    def canJump(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        # 9/19/24:
        # https://www.youtube.com/watch?v=Yan0cv2cLy8

        goal = len(nums) - 1

        for i in range(len(nums) - 2, -1, -1):
            if i + nums[i] >= goal:
                goal = i
        return goal == 0
```

LIS Longest Increasing Subsequence : 1D- DP

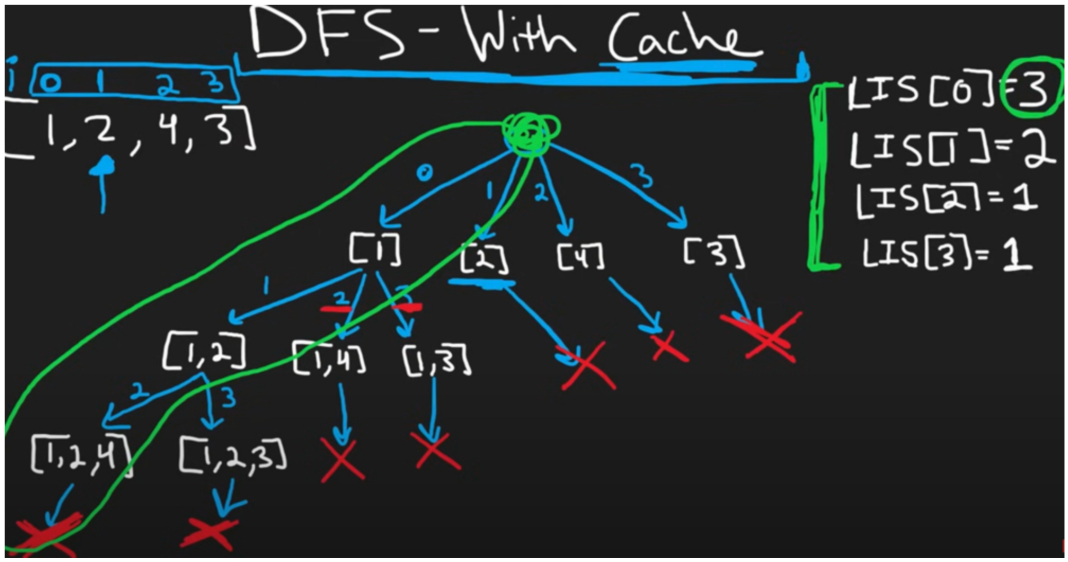
Important: First explain the brute force way, where each element has two choices either we include it or don't include it to build Longest increasing subsequence. But this ends up in 2^N time complexity, so we apprach with recursion w/ mem (top-down/Memoization) or DP(bottom-up / tabulation) approach. See the Needcode video how he approached systemically and don't jump to DP directly.

Example : nums = [1,2,4,3]
Index -> 0,1,2,3
Output = 3

Initaly dp = [1,1,1,1]
Starting from reverse -

dp[3] = 1 LIS at index 3 is 1 because there are no higher number than element at index 3. Its last element.
dp[2] = 1 LIS at index 2 is also 1 because next element at index 3 is 3 which is lower than 4.

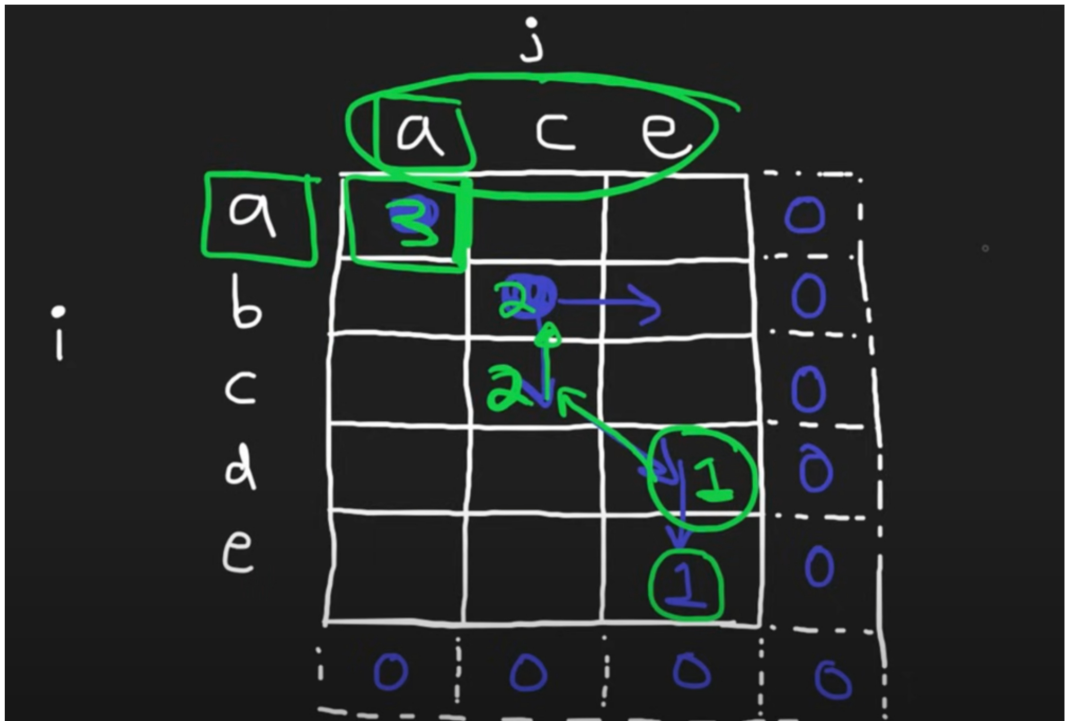
dp[1] = 2 LIS at index 1 is 2 because we have two elements greater, but we can choose either 3 or 4, so its = 1 + dp[2]
dp[0] = 3 LIS at index 0 is 3 as we can see already. which is = 1 + dp[1] = 1+2 =3




```
class Solution:
     def lengthOfLIS(self, nums: List[int]) -> int:

        #  https://www.youtube.com/watch?v=cjWnW0hdF1Y
         LIS = [1] * len(nums)

         for i in range(len(nums) - 1, -1, -1):
             for j in range(i + 1, len(nums)):

                 if nums[i] < nums[j]:
                     LIS[i] = max(LIS[i], 1 + LIS[j])

         return max(LIS)
```

LCS Longest Common Subsequence : Strictly follow Needcode video only. No Tushar. It will be simple.

Important: First explain the brute-force solution by simple comparing two strings, but that takes 2^L time, so we try Greedy but that's not optimal either. See editorial solution so we can apply DP (bottom-up) by showing on examples. See video for explanations. Don't directly jump to DP.



```python
class Solution(object):
    def longestCommonSubsequence(self, text1, text2):

        # 10/05/24: Also, Use same solution to solve LC-516
        # Let's stick to Needcode solution please.

        dp = [[0 for j in range(len(text2) + 1)] for i in range(len(text1) + 1)]

        for i in range(len(text1) - 1, -1, -1):
            for j in range(len(text2) - 1, -1, -1):

                if text1[i] == text2[j]:
                    dp[i][j] = 1 + dp[i + 1][j + 1]
                else:
                    dp[i][j] = max(dp[i][j + 1], dp[i + 1][j])

        return dp[0][0]
```

Longest Palindromic Subsequence : Strictly follow Needcode video only. No Tushar. It will be simple.

```python
class Solution(object):
    def longestPalindromeSubseq(self, s):

        return self.longestCommonSubsequence(s, s[::-1])


    # Reusing exactly same solution from LCS. Pure copy-paste. LC-1143
    # Let't stck to Neetcode's 1143 and NOT tushar's code
    # Just reverse the text2 as reverse of string as we need to check the palindrome.

    def longestCommonSubsequence(self, text1, text2):

        m, n = len(text1), len(text2)

        # Create 2-dimension array for storing the result
        dp = [[0 for j in range(len(text2) + 1)] for i in range(len(text1) + 1)]

        for i in range(len(text1) - 1, -1, -1):
            for j in range(len(text2) - 1, -1, -1):

                if text1[i] == text2[j]:
                    dp[i][j] = 1 + dp[i + 1][j + 1]
                else:
                    dp[i][j] = max(dp[i][j + 1], dp[i + 1][j])

        return dp[0][0]
```

Edit Distance : Follow Tushar for 2D DP or Needcode for consistency.

```python
class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """

        # 9/18/24: This looks very similar to LCS LC-1143 and LC-516. See that first.
        # I prefered Tushar Roy over Neetcode
        if not word1:
            return len(word2)
        if not word2:
            return len(word1)

        m = len(word1)
        n = len(word2)
        dp = [[0 for i in range(n+1)] for j in range(m+1)]

        for i in range(m+1):
            for j in range(n+1):

                # build base case: what happens if one of prefixes is zero
                # (1) word1 prefix is empty, then keep adding
                dp[0][j] = j # meaning word1 is length 0 and word2 is length j, then takes j insert/deletion.

                # (2) word2 prefix is empty, then keep deleting
                dp[i][0] = i # meaning word1 is length i and word2 is length 0, then takes i insert/deletion.

                if word1[i-1] == word2[j-1]: # If letter is same, then no change, so copy previous result
                    dp[i][j] = dp[i-1][j-1]

                else: # Else we need some operation, so we take min of surrounding + 1

                    # dp[i-1][j-1] -> replace
                    # dp[i][j-1] -> add
                    # dp[i-1][j] -> delete
```

```
                    dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])

        return dp[m][n] # or dp[-1][-1]
```

Below is the list from Neetcode-150:

1-D Dynamic Programming (0 / 12)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Climbing Stairs	Easy		Python
☐	☆	Min Cost Climbing Stairs	Easy		Python
☐	☆	House Robber	Medium		Python
☐	☆	House Robber II	Medium		Python
☐	☆	Longest Palindromic Substring	Medium		Python
☐	☆	Palindromic Substrings	Medium		Python
☐	☆	Decode Ways	Medium		Python
☐	☆	Coin Change	Medium		Python
☐	☆	Maximum Product Subarray	Medium		Python
☐	☆	Word Break	Medium		Python
☐	☆	Longest Increasing Subsequence	Medium		Python
☐	☆	Partition Equal Subset Sum	Medium		Python

2-D Dynamic Programming (0 / 11)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Unique Paths	Medium		Python
☐	☆	Longest Common Subsequence	Medium		Python
☐	☆	Best Time to Buy And Sell Stock With Cooldown	Medium		Python
☐	☆	Coin Change II	Medium		Python
☐	☆	Target Sum	Medium		Python
☐	☆	Interleaving String	Medium		Python
☐	☆	Longest Increasing Path In a Matrix	Hard		Python
☐	☆	Distinct Subsequences	Hard		Python
☐	☆	Edit Distance	Medium		Python
☐	☆	Burst Balloons	Hard		Python
☐	☆	Regular Expression Matching	Hard		Python

```
- Unique Path : LC 62 63 980
- Subarray : LC 53, 152
- Decode ways: LC 91 [Same as Climing stairs]
- Word Break: 139 140
- Subsequence LIS : LC 300
* Longest Common substring: 718
* Palindrome Based: LC 516, 5 647;
* Palindrome Partitioning: LC 132, 133

* Jump Game : 55 45 ( Greedy )
* Stairs: 70 746 ( 70 Same to LC-91)
* Coin Change: LC 322, 518, 441
* Stocks: LC 121 122 309 714
* House Robber: LC 213, 198
* Paint House : LC 256 265
* Wildcard: LC 44 10
* Max Sq & Rect: 85 221
* Super Egg: LC 887
* Math: LC 279 343 204
```

8. HEAP : (Its non-linear DS)

Find median price of stocks. Gievn stock price changes everytime. Find median price at any given time. [ASKED at Oracle 10/27/23]

```
class MedianFinder(object):

    def __init__(self):
        #keep smaller half (size >= 1)
        self.maxHeap = []
        self.minHeap = []
```

```
    def addNum(self, num):
        heappush(self.maxHeap, -num)                    # Add to maxheap
        heappush(self.minHeap, -heappop(self.maxHeap))  # Balancing step - Add to minheap (by popping from maxheap)

        # The max-heap is allowed to store, at worst, one more element more than the min-heap
        if len(self.minHeap) > len(self.maxHeap):       # Maintain size property
            heappush(self.maxHeap, -heappop(self.minHeap))


    def findMedian(self):
        if len(self.maxHeap) > len(self.minHeap):
            return float(-self.maxHeap[0])                          # Don't pop, just verify/peek top
        return ((-self.maxHeap[0] + self.minHeap[0] + 0.00 )/2)     # Don't pop, just verify/peek top
```

Meeting Room [ASKED in Oracle 10/9/23]

```
    def minMeetingRooms(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: int
        """

        # Checkout testcase - [[0,7],[5,10],[8,20]]. So we need to maintain minHeap of endtime.

        # Idea: Build the min-heap using end-time, so we know if some room was avaiable earlier than other-room.
        # Always check minheap top and if start > minheap top we can re-use same room. Else allocate new room
        # by pushing to heap. In the end, Size of heap is the number of rooms needed.

        intervals.sort(key=lambda x:x[0]) # Sort the intervals with respect to "START" time
        heap = []                         # IMP: Build heap of "END" time of intervals -

        for i in intervals:
            if heap and i[0] >= heap[0]: # MEANS -If start time is greater than top of Min heap (endtime), just assign
                                         # another meeting to same room aka replace endtime, since we can reuse same room
                heapq.heapreplace(heap, i[1]) # OR heapq.heappushpop(heap, i[1])
            else:
                # a new room is allocated
                heapq.heappush(heap, i[1])  # Build heap with end time.

        return len(heap)
```

Heap based problems from Neetcode-150 list:

Heap / Priority Queue (0 / 7)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Kth Largest Element In a Stream	Easy	■	Python
☐	☆	Last Stone Weight	Easy	■	Python
☐	☆	K Closest Points to Origin	Medium	■	Python
☐	☆	Kth Largest Element In An Array	Medium	■	Python
☐	☆	Task Scheduler	Medium	■	Python
☐	☆	Design Twitter	Medium	■	Python
☐	☆	Find Median From Data Stream	Hard	■	Python

Input/Output in Python : MixPanel : Implement Unix command `tail -10` in python

```
import os

def tail(file_path=None, num_lines=10):
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            file.seek(0, os.SEEK_END)
            file_size = file.tell()
            lines = []
            newline_chars = ['\n']

            for i in range(file_size - 1, -1, -1):
                file.seek(i)
                current_char = file.read(1)
                lines.append(current_char)

                if current_char in newline_chars:
                    num_lines -= 1

                if num_lines == 0:
                    break

            lines.reverse()
```

```
            print(''.join(lines).strip())

    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    # Example usage:
    # tail.py filename.txt
    file_path = input("Enter file path: ")  # or provide a file path directly: file_path = 'yourfile.txt'
    tail(file_path)
```

---

9. HASH TABLE:

Time Based Key-Value Store Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

```
    def __init__(self):
         self.h = {}

    def set(self, key, value, timestamp):
        """
        :type key: str
        :type value: str
        :type timestamp: int
        :rtype: None
        """
        self.h[key] = self.h.get(key, [])   +    [(value, timestamp)]

    def get(self, key, timestamp):
        if key not in self.h:
            return ""
        return self.binarySearch(key,timestamp)

    def binarySearch(self, key, timestamp):

        values = self.h[key]
        low = 0
        high = len(values) - 1
        ans = ""

        while low <= high:

            mid = low + (high - low)//2   # It can't be just (high-low)//2 because imagine low=10, high=100, then mid is 55 and not 45.

            if values[mid][1] == timestamp:
                return values[mid][0]

            elif values[mid][1] < timestamp:
                ans = values[mid][0]
                low = mid + 1

            elif values[mid][1] > timestamp:
                high = mid - 1

        return ans
```

---

380. Insert Delete getRandom in O(1) - Nvidia

Here insert can be done in O(1) using a dynamic array (ArrayList in Java or list in Python). However, the issue we run into is how to go about an O(1) remove. Generally we learn that removing an element from an array takes a place in O(N), unless it is the last element in which case it is O(1).

The key here is that we don't care about order. For the purposes of this problem, if we want to remove the element at the ith index, we can simply swap the ith element and the last element, and perform an O(1) pop . Later adjust the last's element's index in hash and delete the value to be removed.

So, We need to find the index of the element we have to remove. All we have to do is have an accompanying data structure that maps the element values to their index. So along with list, we should also use hash to store the index of elements. That's why we have list and dict both.

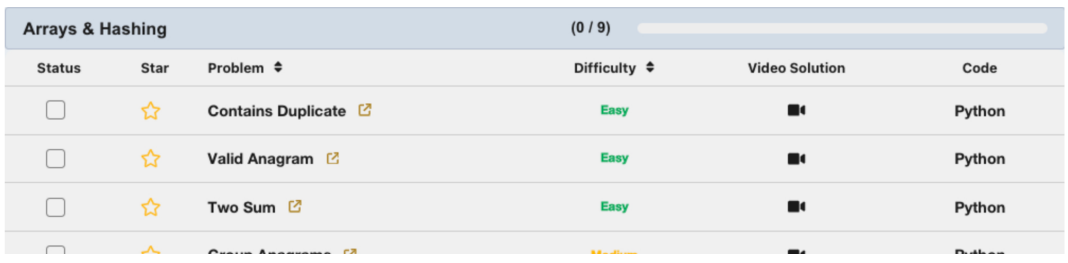
```
class RandomizedSet(object):

    def __init__(self):
        self.list = []
        self.dic =  {}

    def insert(self, val):

        if val not in self.dic:
            self.list.append(val)               # Step-01: First append it to list
            self.dic[val] = len(self.list) - 1  # Step-02: Add "val" as key & (len(list)-1) as value
            return True

        return False

     # Example:  array = [ 6,7,3,2,5]
     #             Index   0,1,2,3,4
     # Insert them one by one in O(1)
```

```
    # Insert them one by one in O(1)
    # Remove 3, which is at index 2.

    # 10/21/24:
    def remove(self, val):   # Same three steps as LC-381

        if val in self.dic:

            # STEP-1: Get the index of element from hash to remove. Also find last elem in list.
            last = self.list[-1]  # 5
            out  = self.dic[val]  # 2 ( We 're removing 3 )

            # STEP-2: Copy last element to the 'out' position. Update hash also.
            self.list[out] = last
            self.dic[last] = out    # Update that element's index in hash

            # STEP-3: Remove from list and hash.
            self.list.pop();
            del self.dic[val]       # self.dic.pop(val)

            return True

        return False

    def getRandom(self):
        return random.choice(self.list)
```

381. Insert Delete GaetRandom O(1) - Dups allowed

Same as 380. But duplicate allowed.

```
import collections
import random

class RandomizedCollection(object):
    def __init__(self):
        self.list = []
        self.dic = collections.defaultdict(set) # Unlike 380 where we don't need set.

    def insert(self, val):
        self.list.append(val)
        self.dic[val].add(len(self.list) - 1)
        return len(self.dic[val]) == 1

     # Example:  array = [ 6,7,3,2,5]
     #             Index   0,1,2,3,4
     # Insert them one by one in O(1)
     # Remove 3, which is at index 2.

    # 10/21/24:
    def remove(self, val):  # Same three steps
        if self.dic[val]:

            # STEP-1: Get the index of element from hash to remove. Also find last elem in array.
            out = self.dic[val].pop()  # Get index of the element to remove
            last = self.list[-1]       # Get the last element

            # STEP-2: Copy last element to the 'out' position. Update hash also
            self.list[out] = last

            # Update the dictionary for the last element
            if self.dic[last]:
                self.dic[last].add(out)                    # Add the new index for last element
                self.dic[last].discard(len(self.list) - 1)  # Remove the old index

            # STEP-3: Remove from list and hash.
            self.list.pop()        # Remove the last element
            if not self.dic[val]:  # Clean up if the set is empty
                del self.dic[val]
            return True

        return False

    def getRandom(self):
        return random.choice(self.list)
```



Arrays & Hashing (0 / 9)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Contains Duplicate	Easy	■	Python
☐	☆	Valid Anagram	Easy	■	Python
☐	☆	Two Sum	Easy	■	Python
☐	☆	Group Anagrams	Medium	■	Python

☐	☆	Group Anagrams	Medium	■	Python
☐	☆	Top K Frequent Elements	Medium	■	Python
☐	☆	Encode and Decode Strings	Medium	■	Python
☐	☆	Product of Array Except Self	Medium	■	Python
☐	☆	Valid Sudoku	Medium	■	Python
☐	☆	Longest Consecutive Sequence	Medium	■	Python

---

10. LINKEDLIST CONCEPTS/TEMPLATES

Possible testcases for interviews :

TC-01: Empty List head= None
TC-02: Single Node
TC-03: Two Nodes
TC-04: Three or more Nodes
TC-05: Odd and Even length Nodes
TC-06: List with All same values
TC-07: List with duplicates
TC-08: List with -ve values.
TC-09: List with loop/cycle
TC-10: List with large number of Nodes (10^5)

1. Find the midpoint of Linked List. (LC-143, 234)

```
def midpoint(self, head):
        slow = fast = head
        while fast and fast.next and fast.next.next:      # Takes care of empty node also
            slow = slow.next
            fast = fast.next.next

        mid = slow.next
        slow.next = None # optional
```

2. Reverse a Linked List. (LC-143, 234)

```
    def reverse(self, head):
        prev = None
        while head:
            tail = head.next
            head.next = prev

            prev = head    # Imp - first assign to prev and
            head = tail    # later to assign to head
        return prev
```

3. Find if Linked List got cycle. (LC-141)

```
   def hasCycle(self, head):

        # Floyd's Tortoise and Hare Algorithm
        slow = fast = head

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

            if slow == fast:
                return True
        return False
```

4. Merge two list alternatively. (LC-143)

```
 def mergeAlternate(self, l1, l2 ):
       curr = dummy = ListNode(0)
       while (l1 or l2):
           curr.next = l1
           l1 = l1.next         # Move
           curr = curr.next     # Move

           l1, l2 = l2, l1      # Swap for alternating
       return dummy
```

5. Check two linkedlist are palindrome (LC-234)

```
def isPalindrome(self, l1, l2):
       while l1 and l2:
           if l1.val != l2.val:
               return False
           l1 = l1.next           # Move
           l2 = l2.next           # Move
       return True
```

6. Reverse K group of LinkedList (LC-25)

```
    def reverseK(self, head, k): # We gets whole remaining LL from curr.next onwards BUT only reverse K elements.
        prev = None
        curr = head  # Note: Before this function returns head is updated for start of next block.

        while k > 0: # Same as 'while head' logic, but since we are dealing with blocks of K instead of going till end.
            tail = curr.next
            curr.next = prev # Flip pointer

            prev = curr
            curr = tail
            k -= 1

        # EXPLANATION FOR (head.next = tail): Let's say we have input 1->2->3->4->5->6->7->8.
        # Note after reverse k group it looks like prev<-curr<-tail (aka 1<-2<-3),
        # so head.next  = tail and we return prev. Basically we say dummy->3->2->1. So we return prev (aka 1).

        head.next = tail   # head is now the tail of the reversed group
        return prev
```

Intersection of Two Linked Lists : ASKED in AirBnb, US Bank

```
def getIntersectionNode(self, headA, headB):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """

    # Concatenate list A and list B, if there's an intersection, there's loop,
    # so we need to find the start of the loop if there's any
    # Checkout LC-142 To see how we find start of cycle.

    # 10/27/23:
    # Example: listA = [2->3->4->9->1]
    #          listB = [5->6->4->9->1]

    # Concatelate: [2-> 3-> 4-> 9-> 1-> 5-> 6 ]
    #               |       ^      S=F      |
    #               |       |_______________|
    #               A
    #               S (slow)
    #               F (fast)

    def getIntersectionNode(self, A, B):
        """
        :type head1, head1: ListNode
        :rtype: ListNode
        """
        if not A or not B:
            return None

        # Concatenate A and B
        head = A
        while head.next:
            head = head.next
        head.next = B

        # Find the start of the loop
        fast = slow = A
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
            if slow == fast:

                fast = A
                while fast != slow:
                    slow, fast = slow.next, fast.next
                head.next = None    # Because - Your code should preferably run in O(n) time and use only O(1) memory.
                return slow         # else you get - ERROR: linked structure was modified.

        # No loop found
        head.next = None    # Because - Your code should preferably run in O(n) time and use only O(1) memory.
        return None
```

Copy List with Random Pointers :

Concept: We can iternate through given LL from head to tail and as we iternate:
We can create newnode for new copy-of-list. If node already exists in map, reuse/return from map, else create newNode

```
class Solution(object):

    def __init__(self):
        self.map = {} # Reason we have, since random pointers can form cycles and if node was already processed, we can loopup from map

    def copyRandomList(self, head):
        """
        :type head: Node
        :rtype: Node
```

```
        """

        if head is None:
            return None

        if head in self.map:
            return self.map[head]

        clone_node = Node(head.val, head.next, head.random)
        self.map[head] = clone_node

        clone_node.next = self.copyRandomList(head.next)
        clone_node.random = self.copyRandomList(head.random)

        return clone_node
```

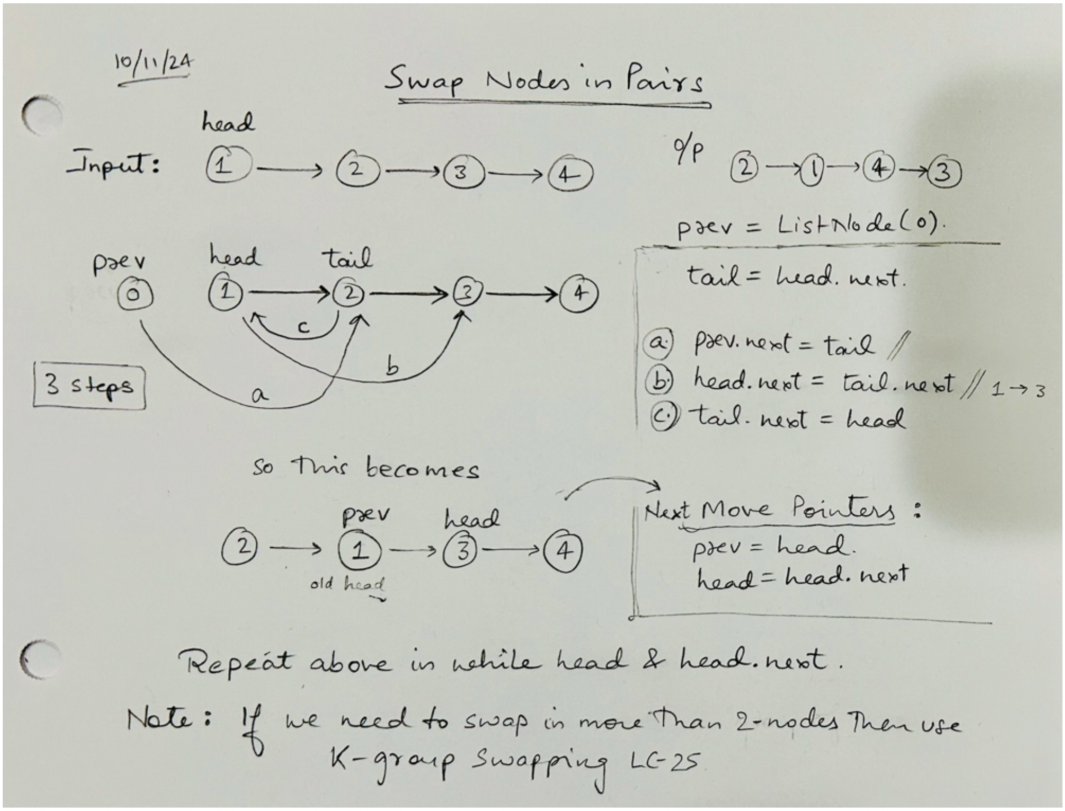
Test Case Categories

- Empty List
- Single Node
- Multiple Nodes with random Pointers as None
- Multiple Nodes with random Pointers Pointing to Various Nodes
- Lists with Cycles via random Pointers

LC-24: Swap Nodes in Pair

Input: head = [1,2,3,4]

Output: [2,1,4,3]

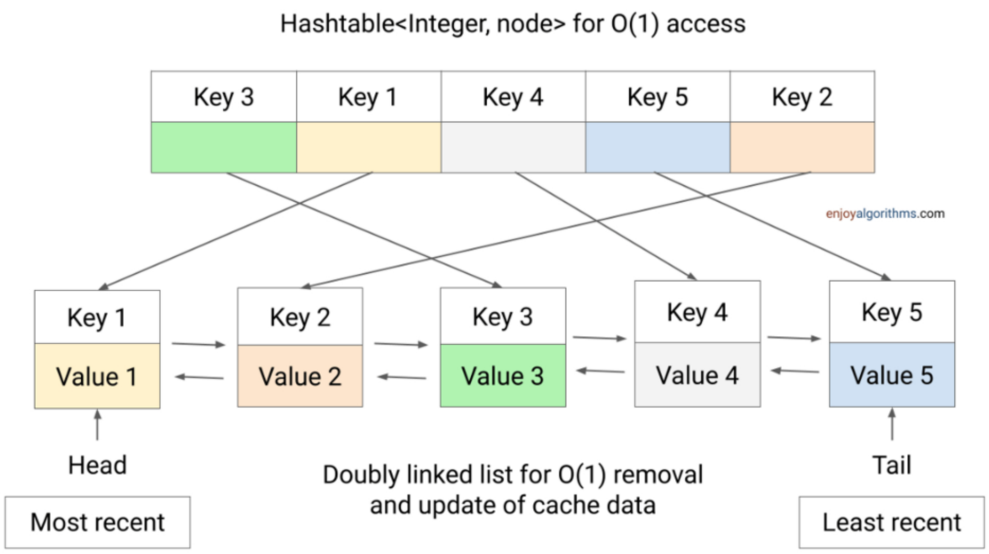



```
class Solution(object):
    def swapPairs(self, head):

        dummy = prev = ListNode(0)
        dummy.next = head

        while head and head.next:    # Checkout the diagram.
            tail = head.next

            prev.next = tail         # step a
            head.next = tail.next    # step b
            tail.next = head         # step c

            prev = head              # Moving pointers. Same as "Reverse Linked List"
            head = head.next

        return dummy.next
```

LRU Implementation :

Visual of LRU implementation : https://www.enjoyalgorithms.com/blog/implement-least-recently-used-cache

Visual of LRU Implementation : https://www.enjoyalgorithms.com/blog/implement-least-recently-used-cache



Another Visual of LRU showing Nodes :

```
Map:                          Doubly Linked List:
mapKey (int) ------> mapValue (Node [mapKey, value]) <-- Front (Head) of List (Least Recent in Cache)
                              ^ |     (If we are at capacity and evict this, we instantly know which key to remove from the map)
                              | |
                              | v
mapKey (int) ------> mapValue (Node [mapKey, value])
                              ^ |
                              | |
                              | v
mapKey (int) ------> mapValue (Node [mapKey, value]) <-- Back (Tail) of List (Most Recent in Cache)
```

[FYI: LFU : https://www.enjoyalgorithms.com/blog/least-frequently-used-cache]

Note: We use hash for fast lookup and Doubly LinkedList for fast insertion and deletion.
Time Complexity: O(1), Space Complexity: O(capacity)

Note : In below design, we have considered - head is least-recent and tail is most-recent.

```
class LRUnode(object):
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.next = None
        self.prev = None
```

```
class LRUCache(object):
    def __init__(self,c):
        self.capacity = c
        self.dict = {}

        self.dummyhead = LRUnode(0,0)
        self.dummytail = LRUnode(0,0)

        self.dummytail.prev = self.dummyhead
        self.dummyhead.next = self.dummytail

    def _append(self, node): # Append at end
        tail = self.dummytail.prev

        tail.next = node
        node.prev = tail

        self.dummytail.prev = node
        node.next = self.dummytail

    def _remove(self, node):   # Just Remove
        prev = node.prev
        next = node.next

        prev.next = next
        next.prev = prev

    def get(self, key):

        if key not in self.dict:
            return -1
        node = self.dict[key]
        self._remove(node)
```

```
        self._append(node)  # Append at end. Note end is most-recent and head is least-recent
        return node.val

    def put(self, key,val):
        if key in self.dict:  # First just remove if it exist. since we need to update its position anyways
            self._remove(self.dict[key])

        newNode = LRUnode(key, val) # IMP Note: key is added as part of dict as well node both.
        self.dict[key] = newNode
        self._append(newNode)

        #Remove from head, if exceeding the capacity, as its LRU
        if len(self.dict) > self.capacity:
            head = self.dummyhead.next
            self._remove(head)              # Remove node
            del self.dict[head.key]         # Delete it from hash also using its key.
```

Practice below from Neetcode-150 list :

Linked List (0 / 11)

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Reverse Linked List	Easy	■	Python
☐	☆	Merge Two Sorted Lists	Easy	■	Python
☐	☆	Reorder List	Medium	■	Python
☐	☆	Remove Nth Node From End of List	Medium	■	Python
☐	☆	Copy List With Random Pointer	Medium	■	Python
☐	☆	Add Two Numbers	Medium	■	Python
☐	☆	Linked List Cycle	Easy	■	Python
☐	☆	Find The Duplicate Number	Medium	■	Python
☐	☆	LRU Cache	Medium	■	Python
☐	☆	Merge K Sorted Lists	Hard	■	Python
☐	☆	Reverse Nodes In K Group	Hard	■	Python

11. QUEUE CONCEPTS

LC-622 Design Circular Queue: (Without using built-in queue data structure)

```
class MyCircularQueue:

    def __init__(self, k):
       # Initialize your data structure here. Set the size of the queue to be k.

        self.queue = [0]*k
        self.headIndex = 0
        self.count = 0
        self.capacity = k

    def enQueue(self, value):
        # Insert an element into the circular queue. Return true if the operation is successful. Insert at end which will be Rear.

        if self.count == self.capacity:
            return False

        self.queue[(self.headIndex + self.count) % self.capacity] = value
        self.count += 1
        return True

    def deQueue(self):
        # Delete an element from the circular queue. Return true if the operation is successful.

        if self.count == 0:
            return False

        self.headIndex = (self.headIndex + 1) % self.capacity
        self.count -= 1
        return True

    def Front(self):
        # Get the front item from the queue.

        if self.count == 0:
            return -1
```

```
        return self.queue[self.headIndex]

    def Rear(self):
        # Get the last item from the queue.

        if self.count == 0:     # empty queue
            return -1

        return self.queue[(self.headIndex + self.count - 1) % self.capacity]

    def isEmpty(self):
        # Checks whether the circular queue is empty or not.

        return self.count == 0

    def isFull(self) :
        # Checks whether the circular queue is full or not.

        return self.count == self.capacity

    # How it works !!
    # If capacity = 3. Enqueue first three elements
    # [1,2,3] headIndex = 0 , front=1, rear=3, count =3,
    # deQueue (delete from Front) so now [_,2,3],
    # enque 4 ( at Rear), so now [2,3,4] since its CIRCULAR.
```

---

12. MONOTONIC STACK CONCEPT

Per ChatGPT:
A monotonic stack is a stack data structure that maintains its elements in a specific monotonic order—either increasing or decreasing. This property allows for efficient querying of elements relative to others in the sequence, such as finding the next greater or smaller element.

- Monotonic Increasing Stack: Each element is greater than or equal to the element below it.
- Monotonic Decreasing Stack: Each element is less than or equal to the element below it.

Monotonic stacks are particularly useful for problems where you need to:

1. Find the Next Greater Element (NGE) or Next Smaller Element (NSE) for each element in an array.
2. Calculate spans (e.g., Stock Span Problem), processing temperatures
3. Solve range-based problems (e.g., Largest Rectangle in Histogram).

Key Takeaways :

- Monotonic Decreasing Stack: Ideal for finding next greater elements in a sequence.
- Monotonic Increasing Stack: Suited for finding next smaller elements.

Algorithm :

Below is a generic template for solving problems using a monotonic stack. This template is adaptable to both increasing and decreasing monotonic stacks.

```
Template Steps:
---------------
1. Initialize:
* Create an empty stack to store indices or elements.
* Prepare a result array to store answers (e.g., next greater elements).

2. Iterate Through the Array:
* For each element in the array:
    * While the stack is not empty and the current element satisfies the monotonic condition with the element at the top of the stack:
        * Pop the element from the stack.
        * Update the result for the popped element.
    * Push the current elements index onto the stack.

3. Finalize:
After the iteration, handle any remaining elements in the stack if necessary (e.g., assign default values like -1).
```

Higly Generic Template code :

```
def monotonic_stack_template(arr, find_next=True, is_increasing=True):
    """
    Generic monotonic stack template.

    Parameters:
    - arr: List of elements to process.
    - find_next: If True, find next greater/smaller. If False, find previous.
    - is_increasing: If True, use increasing stack. If False, use decreasing stack.

    Returns:
    - result: List containing the next/previous greater/smaller elements.
    """
    n = len(arr)
    stack = []
    result = [-1] * n  # Initialize result array with default values

    # Define comparison based on stack type
    if is_increasing:
        compare = lambda x, y: x < y  # For Next Greater or Previous Greater
    else:
        compare = lambda x, y: x > y  # For Next Smaller or Previous Smaller
```

```
    # Define the range of iteration based on find_next
    iterable = range(n) if find_next else range(n-1, -1, -1)

    for i in iterable:
        while stack and compare(arr[i], arr[stack[-1]]):
            index = stack.pop()
            result[index] = arr[i]
        stack.append(i)

    return result
```

Example:

```
def next_greater_element(arr):
    return monotonic_stack_template(arr, find_next=True, is_increasing=True)

# Example usage:
arr = [4, 5, 2, 25]
print(next_greater_element(arr))  # Output: [5, 25, 25, -1]
```

Tips and Best Practices

1. Use Indices Instead of Values:
   Storing indices allows you to calculate widths or distances between elements easily.
2. Sentinel Values:
   Adding a sentinel value (like 0 or -1) at the end of the array can simplify the logic by ensuring the stack is emptied, handling remaining elements. Example Largest Rectangle in Histogram
3. Choose the Right Monotonic Order:
   Determine whether you need an increasing or decreasing stack based on the problem requirements (e.g., NGE vs. NSE).
4. Understand the Problem Requirements:
   Carefully read the problem to decide whether you need the next, previous, or both greater/smaller elements.

NOTE: Most of the questions I have solved below needs Monotonic Descresing Stack.

Daily Temperature : Monotonic Decreasing Stack. | Checkout Neetcode YT video for explanation. Same follows for all question.

Bruteforce way it can solved by comparing each element which would be O(N^2), but we can solve this in O(N) by using monotonic stack.

However , This problem looks like finding next greater (aka higher temperature). We know - Monotonic Decreasing Stack are Ideal for finding next greater elements in a sequence. Below is the TEMPLATE for MDS.

```
class Solution(object):
    def dailyTemperatures(self, temperatures):

        # Step-1: Initialization
        res = [0] * len(temperatures)
        stack = []  # pair: [temp, index]

        # Step-2: Iterate Through the Array
        for i, t in enumerate(temperatures):

            # 2.1: While stack not empty and current element satisfies
            # monotonic condition (higher) w/ elem on top of stack.

            # IMP: Pop elements from the stack while the current price is higher or equal
            # determining how far you can go to find a day with a higher temperature.
            # If not, then add lower stock temperature to stack, resulting in
            # Monotonically Decreasing Stack (w.r.t temperature, not Index)
            while stack and t > stack[-1][0]:

                 # 2.1.1: Pop the element from the stack.
                stackT, stackInd = stack.pop()

                # 2.1.2: Update the result for the popped element.
                res[stackInd] = i - stackInd

            # 2.2: Push the current element's index onto the stack.
            stack.append((t, i))
            #print(stack)            # Monotonically Decreasing Stack (w.r.t temperature, not Index)

        # Step-3: No need to finalize since remaining indices in the stack have result[i] = 0
        return res
```

Online Stock Spanner : Monotonic decreasing stack.

Returns the span of the stock's price given that today's price. The span of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

Note: Online Stock Span is closer to the Next Greater Right (NGR) problem because it deals with finding the number of consecutive days where the current stock price is greater than or equal to the price of previous days.

- Monotonic Stack Type: You should use a Monotonic decreasing stack.
- Why Use a Monotonic Decreasing Stack?
  - Because You are essentially looking for how far back (to the left) you can go where the price was greater than or equal to the current price.

- If you maintain a decreasing stack (highest prices at the top), when a new price arrives that is greater than the prices in the stack, you can pop them out and calculate the span.

```
class StockSpanner(object):

    def __init__(self):

        # Step-1: Initialization
        self.stack = [] # Stack will store pairs of (price, span)


    # Step-2: Iterate Through the Array
    def next(self, price):

        span = 1  # Initialize span for the current day

        # 2.1: While stack not empty and current element satisfies
        # monotonic condition w/ elem on top of stack.

        # IMP: Pop elements from the stack while the current price is higher or equal
        # Determining how far back you can go to find a day with a higher stock price.
        # If not, then add lower stock price to stack, resulting in
        # Monotonically Decreasing Stack (w.r.t prices, not Index)
        while self.stack and price >= self.stack[-1][0]:

            # 2.1.1: Pop the element from the stack.
            popped_price, popped_span = self.stack.pop()

            # 2.1.2: Update the result for the popped element.
            span += popped_span  # Accumulate span

        # 2.2: Push the current price and its span onto the stack
        self.stack.append((price, span)) # Monotonically Decreasing Stack (w.r.t prices, not Index)
        #print(self.stack)

        return span
```

Next greater Element-I : Monotonic decreasing stack.

```
class Solution(object):
    def nextGreaterElement(self, nums1, nums2):

        # Finds the next greater element for each element in nums1 based on nums2.
        # Parameters:
        # - nums1: List[int] - Subset of nums2.
        # - nums2: List[int] - Array to find the next greater elements from.

        # By utilizing a Monotonic Decreasing Stack, we efficiently solve the Next
        # Greater Element I problem with linear time complexity.

        stack = []  # Monotonic stack
        next_greater_map = {}  # Mapping from element to its next greater element

        # Iterate through nums2
        for num in nums2:
            # While current num is greater than the last element in the stack
            while stack and num > stack[-1]:
                popped_num = stack.pop()
                next_greater_map[popped_num] = num
            stack.append(num) # Monotonically Decreasing Stack. Same as Daily temperature , Online stock Span
            # print(stack)

        # For remaining elements in the stack, there is no next greater element
        while stack:
            popped_num = stack.pop()
            next_greater_map[popped_num] = -1

        # Generate the result for nums1 based on the mapping
        return [next_greater_map[num] for num in nums1]
```

503. Next Greater Element II : Monotonic decreasing stack.

```
class Solution:
    def nextGreaterElements(self, nums):

         # Step-1: Initialization
        n = len(nums)
        result = [-1] * n  # Initialize the result array with -1
        stack = []  # Monotonic stack to store indices


        # Step-2: Iterate Through the Array
        # Iterate twice to simulate the circular array behavior
```

```
        for i in range(2 * n):
            current_index = i % n

            # 2.1: While stack not empty and current element satisfies
            # monotonic condition w/ elem on top of stack.

            # IMP: Pop elements from the stack while the current price is higher or equal
            # Determining how far you can go to find higher number.
            # If not, then add lower number's index to stack, resulting in
            # Monotonically Decreasing Stack (w.r.t number, not Index)
            while stack and nums[stack[-1]] < nums[current_index]:

                # 2.1.1: Pop the element from the stack.
                index = stack.pop()

                # 2.1.2: Update the result for the popped element.
                result[index] = nums[current_index]

            # 2.2: Push the current price and its span onto the stack
            if i < n:
                stack.append(current_index)

        return result
```

IMPORTANT : stack stores indices and values at those indices is decreasing (not indicies themselves). Here The stack stores indices, not the actual values. Observing the order of indices might give an impression of an increasing stack, but it's the values (nums[index]) that determine the monotonicity.

Remove Nodes from LinkedList: Monotonic decreasing stack.

The task requires removing nodes from the list if a larger node exists on the right side of that node. This can be done efficiently using a monotonic decreasing stack where we maintain the largest elements from right to left.

Approach using Monotonic Stack:

1. Iterate through the linked list and push nodes onto a monotonic decreasing stack.
2. For each node, if it has a greater node on the right, we pop smaller elements from the stack.
3. Finally, reconstruct the linked list from the stack, as it will now contain the remaining nodes in the correct order.

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

class Solution:
    def removeNodes(self, head: ListNode) -> ListNode:
        # Step 1: Use a list to simulate a stack
        stack = []

        # Step 2: Traverse the linked list from left to right
        current = head
        while current:
            # Step 3: Maintain the monotonic decreasing stack
            while stack and stack[-1].val < current.val:
                stack.pop()
            stack.append(current)
            current = current.next

        # Step 4: Rebuild the resulting linked list from the stack
        # Reverse iterate through the stack and fix the 'next' pointers
        for i in range(len(stack) - 1):
            stack[i].next = stack[i + 1]
        stack[-1].next = None  # The last node's next should be None

        return stack[0] if stack else None
```

Key Takeaways :

- Monotonic Decreasing Stack: Ideal for finding next greater elements in a sequence.
- Monotonic Increasing Stack: Suited for finding next smaller elements.

MAX STACK :

```
class MaxStack(object):

    # 9/20/24: Let's provide naive solution with worst came time-complexity of O(n) for popMax
    # which uses two stacks. why two stacks because we have popMax which needs cleanup and elect new max.
    # If interviewer asks for log(n) complexity then we need to use maxHeap solution discussed later.

    def __init__(self):
        self.stack = []
        self.max_stack = []

    def push(self, x):

        self.stack.append(x)

        # Always push the latest max value on to max stack
```

```
        if self.max_stack:
            new_max_value = max(x, self.max_stack[-1])
        else:
            new_max_value = x

        self.max_stack.append(new_max_value)

    def pop(self):
        self.max_stack.pop()
        return self.stack.pop()

    def top(self):
        return self.stack[-1]

    def peekMax(self):
        return self.max_stack[-1]

    def popMax(self):

        max_value = self.max_stack.pop()

        tmp = []
        while self.stack[-1] != max_value:
            tmp.append(self.pop())
        self.stack.pop()                # Finally remove the max value element from stack

        # Using self.push(), push back those
        while tmp:
            self.push(tmp.pop())

        return max_value
```

Below is self-explanatory example :

```
# Example:

# ["MaxStack","push","push","push","push","push","top","popMax","top","peekMax","pop","top"]
# [[],[6],[7],[2],[1],[5],[],[],[],[],[],[]]

# Stack MaxStack
#   5       7
#   1       7
#   2       7
#   7       7
#   6       6

# During popmax() - We pop all ele from stack ( & max_stack) until we don't pop max i.e 7
# and then push back remaining element back

#   5       6
#   1       6
#   2       6
#   6       6

Overall Time Complexity Summary:
push(x): O(1)
pop(): O(1)
top(): O(1)
peekMax(): O(1)
popMax(): O(n)
```

---

13. BIT MANIPULATION TIPS/TRICKS

Few Bit Manipulation Tips :

1. Left Shift is like multiply by 2. (2 << 1 = 4, 3 << 1 = 8)
2. Right Shift is like divide by 2. (4 >>1 = 2, 5 >> 1 = 2, 6 >>1 = 3)
3. Is n power of 2 ? --> return n & (n-1) == 0
4. Swap two numbers without temporary variable --> a ^= b; b ^= a; a ^= b
5. Check if number id even or odd --> If (x &1) == 0 its even else odd.
6. Set nth bit of number x --> x & (1 << n)
7. Missing number --> a^b^a = a

268. Missing Number

We all know that a^b^b =a, which means two xor operations with the same number will eliminate the number and reveal the original number.

In this solution, I apply XOR operation to both the index and value of the array. Basically In a complete array with no missing numbers, the index and value should be perfectly corresponding(nums[index] = index), so in a missing array, what left finally is the missing number.

```
class Solution(object):
    def missingNumber(self, nums):

        res = len(nums)

        for i in range(len(nums)):
            res = res ^ i ^ nums[i]
```

```
        return res
```

136. Single Number

Finding unique element amount duplicates.
1 XOR 1 = 0. All duplicate numbers will result to 0, So eventualy single number will only be left.

```
class Solution(object):
    def singleNumber(self, nums):

        res = 0
        for num in nums:
            res ^= num
        return res
```

191. Number of 1 Bits

Below reduces '1' from rightmost eveytime &
then we increment count. It does cut the last digit 1 in the binary.

```
class Solution(object):
    def hammingWeight(self, n):

        count = 0
        while n:
            n = n & (n-1)
            count += 1
        return count
```

14. CYCLE SORT

41. First Missing Positive

```
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:

        n = len(nums)

        # Use cycle sort to place positive elements smaller than n
        # at the correct index
        i = 0
        while i < n:
            correct_idx = nums[i] - 1
            if 0 < nums[i] <= n and nums[i] != nums[correct_idx]:
                # swap
                nums[i], nums[correct_idx] = nums[correct_idx], nums[i]
            else:
                i += 1

        # Iterate through nums
        # return smallest missing positive integer
        for i in range(n):
            if nums[i] != i + 1:
                return i + 1

        # If all elements are at the correct index
        # the smallest missing positive number is n + 1
        return n + 1
```

15. GREEDY STRATEGY :

Task Scheduler :

Below approache use a greedy strategy, meaning decisions are made step by step, focusing on what seems best in the moment to reach the overall best solution

```
class Solution(object):
    def leastInterval(self, tasks, n):

        task_counts = Counter()
        count_of_counts = Counter()
        max_task_count = 0

        freq = Counter(tasks)
        max_task_count = max(freq.values())

        for char in freq:
            count_of_counts[freq[char]] += 1  # count_of_counts : Get count of all those task with max counts.

        return max(len(tasks), max_task_count + (max_task_count-1) * n + count_of_counts[max_task_count]-1) # Simple Maths
```

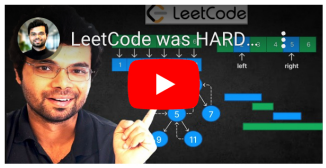
Jump Game

APPENDIX :

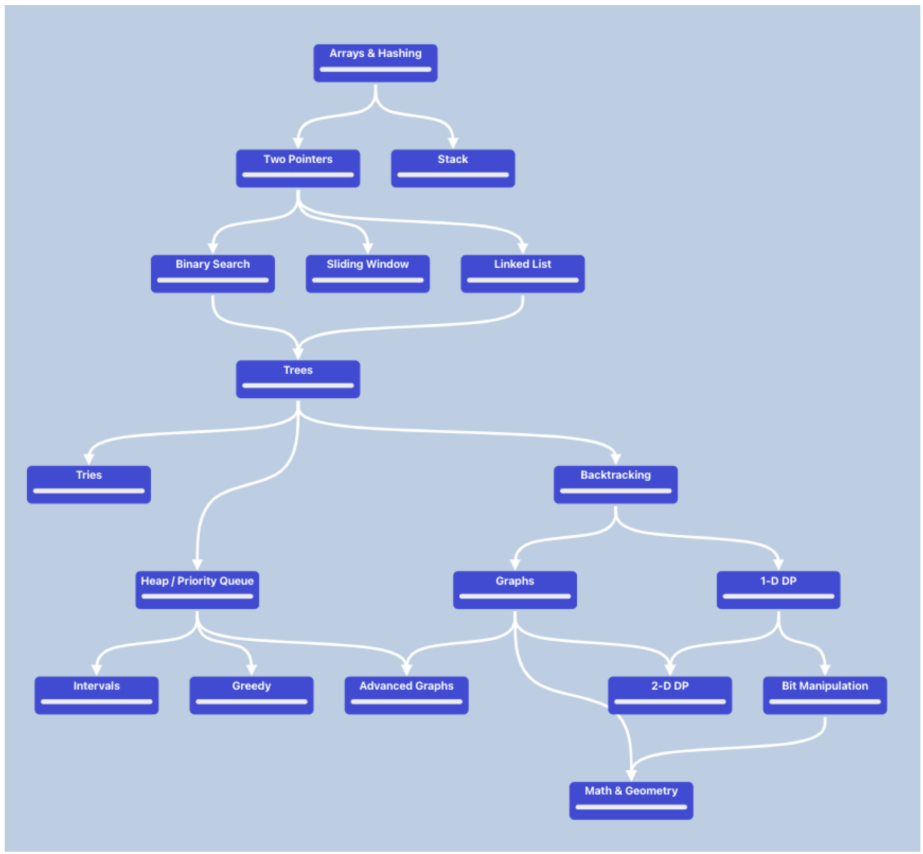
I.) Approach for solving DS problems if new :

Below video tells different pattern and how they are solved Brute-force and how different pattern/technique helps to reduce those bruteforce solution into optimal solution.

Leetcode Patterns : https://www.youtube.com/watch?v=DjYZk8nrXVY&t=611s



Roadmap to follow :



II.) Personal Schedule & Mindset :

```
* Wake-up at 7:00 am
* 8:00 - 12:00 PM: Non-stop, uninterrupted. Force brain to think | Uninterrupted | Time-boxed | Every day.
* 1:00 - 5:00 PM: Mocks, Interview/HR Calls, System Design prep.
* 7:00 - 9:00 PM: Practice/schedule with someone (techbay area). Apply on LinkedIn
```

- Prepare/Practice 4 Leetcode just like normal day stuff, since we just need to keep doing until we hit offer.
  We don't know when Good/Big opportunity may come in. We just need to be Well-prepared.
- Present your SD, because presenting has 2 benefits- you gain confidence as you speak out loud and know
  where you are lagging. Plus you get criticism and feedbacks for self-improvement.
- More you sweat in practice less you bleed in war.

III.)INTERVIEW FEEDBACKS AND REALIZATIONS : 10/5/24 (After Nvidia)

Current Situation :
++++++++++++

1. System Design is not a problem generally. Just need to drive/lead as per hints & expectations from interviewers. Review your SD Designs/Notes & do mock w/ Sagar.

2. Behavioral is not a problem. Give better examples showing your leadership and ownership with deep dive in STAR format.
3. Coding is either a HIT (if you know) or a MISS (if you try to solve new problem). Its just depends on your day. Irrespective need to be FAST.

How / What to improve ?
++++++++++++++++

1. With Coding Interviewer (Senior/Staff Engineers) :
    - Show clarity of thoughts - tell right DS and approach with reasoning - **CLEARLY**, **SMARTLY**, **QUICKLY**.
    - Be a **FAST** coder. Tell brute-force QUICKLY and code optimal QUICKLY.
    - Dress-up smartly and sound very smart and experienced.
    - Be concise and to the point of question asked. Okay to talk in details about what's asked. But don't talk other topics not asked. It's just waste of time for them as they have their own agenda in that 45 mins interview time.
    - Fallbacks: They can easily reject you for multiple reasons - if your are slow OR didn't complete code OR didn't pass all test-cases given by them OR didn't coded optimized solution, OR didn't tell brute-force way initially OR didn't gave them time to ask follow-up questions and time-space complexity OR didn't tell +ve & -ve test-cases. So be careful and Smart.
2. With Behavioral/Cultural Interviewer (Managers/Directors) :
    - Showcase your experience and how it matches with profile. Okay to exaggerate on your background. Tell answers with **SOLID examples** in **STAR format** showing ownership and leadership.
    - Fallbacks : If your examples are not strong and not in STAR format.
3. With System Design (Architect/Staff Engineer) :
    - **Drive/lead discussions**. Go in the format - Functional/Non-functional reqs quickly, Capacity estimates quickly, API design quickly, DB schema quickly, HLD and make sure to complete and design all **core things**.
    - Don't talk about lesser important things like security etc which would waste time as cost of core components designs.
    - Talk about **Trade-off**s on DB and others.
    - Talk about **Scalability** (as per Non-functional reqs) - example introduce Queues for Asycronous (event driven) . Highly availability [Per Sagar's Feedback].
    - Make sure your all functional reqs are covered end-to-end.
    - Make sure all open-questions and concern of Interviewer are addressed before concluding interview.

- Per Amazon Recruiter - Perfect and quick CODING is MUST for Engineering positions. Coding is same for L5/L6, However System Designs and Leadership Principles will decide your level.
- Per Meta Recruiter - Coding Rubrics: Points are given for Speed, Accurate (poper DS and algo choosen) and Clean (no mistake) coding and test-cases/verification/conner-cases + communication-style / energy. 35 mins 2 Medium questions. Again Coding is same for IC4/5. What decides level is Leadership Principles, System Design and Communication Style. Breakdown of the Coding Rubric : Total 40 mins two medium leetcode questions (FB tagged). Each question 15-20 mins time-boxed. [First 1-2 mins understand question and ask clarifying questions, next 3-5 mins come up with bruteforce explanation with time complexity and telling optimal solution. Next 6 -15 mins coding clean optimal solution. 16-18 mins writing testcases/validation and time/space complexity]. Most people fail because they don't time-box well. Good news is that Meta asked its tagged problems mostly from Leetcode. Preparing from FB-tagged problems from last 6 months is everything mostly. SD is also not super hard. People have been asked design web-crawlers, Instagram etc.
- Per Google Employee: Google mostly focuses on Non-linear data-structures - Tree, Tries and Graphs problems. Not much dynamic programming. Questions are mostly logic based and not much from Leetcode (google-tagged). Not sure yet on speed and other rubrics. System Design could include both design & coding. Example Ashish Patil was asked Design Logging & Monitoring system and write code for logging class implementation.

FYI: For Senior System Design interviews - I have seen and being asked design & code Monitoring & Alerting system. Design and code Logging system. Design and code Job scheduler. Design & code multi-threaded pub-sub system (asked @ ID.me) etc

---

Startup Experiences : They usually DON'T ask Leetcode problems. They ask some real-time design/coding.

- Design/Implement Notes Application in Python using Flask - design REST APIs and all its DB schema etc (See Lightening AI Question).
- Design REST APIs for image retriever.
- Design REST API to get country code from given URL and response as +(country_code) phone_number. (TEK System) - Similar to `horus-gw` REST server.
- Design multi-threaded Consumer/Producer (ID.me).
- Design in-memory database asked in Klaviyo assignment. Fix issues in train-signalling system (Klaviyo onsite).
- Develop a session-based authentication system (Kognitos) in python.
- Design/Implement "tail -f 10" command in python (MixPanel)
- Given the adjacency matrix of the connected undirected graph with no loops or multiple edges, find the shortest distance b/w two vertices. (Verkada)
- Design DB schema with many to many relation and so on ...
- Design Notification System (slack, email, sms)
- Implement Consistent Hashing (w/ virtual nodes for load distribution)
- Implement Rate Limiter (token-bucket)

Leetcode based favorite questions for Senior Roles :

- Design Hashmap - https://leetcode.com/problems/design-hashmap/
- Design in-memory file system - https://leetcode.com/problems/design-in-memory-file-system/description/ [Snowflake 10/4/23 - Tries]
- Design circular queue - https://leetcode.com/problems/design-circular-queue/description/ [Just practice, LL, Concurrency]
- LRU Cache - https://leetcode.com/problems/lru-cache/
- Time based key-value store - https://leetcode.com/problems/time-based-key-value-store/
- Implement Trie - https://leetcode.com/problems/implement-trie-prefix-tree/description/
- Min Stack - https://leetcode.com/problems/min-stack/
- Max Stack - https://leetcode.com/problems/max-stack/
- Course Schedule II - https://leetcode.com/problems/course-schedule-ii/description/ [Topological Sort]
- Basic Graph Traversal DFS/BFS- Shortest distance from source/destination [Verkada]
- Minimum Window Substring - https://leetcode.com/problems/minimum-window-substring/
- Meeting Rooms II (Leetcode Premium) - https://leetcode.com/problems/meeting-rooms-ii/
- Find Median from Data Stream - https://leetcode.com/problems/find-median-from-data-stream/ [Oracle 10/27/23]
- Reverse Nodes in K groups - https://leetcode.com/problems/reverse-nodes-in-k-group/ [Meta 2019]
- Find first missing positive - https://leetcode.com/problems/first-missing-positive/ [Netscope]
- Insert, delete, Random in O(1): https://leetcode.com/problems/insert-delete-getrandom-o1/description/ [Nvidia 2024, Meta 2019]
- Word Search II - https://leetcode.com/problems/word-search-ii/description/ [Oracle 10/23] [DFS, Backtracking, Trie]
- Word Break II - https://leetcode.com/problems/word-break-ii/description/
- Questions on Graphs - All Paths in Graph, Shortest distance on Matrix

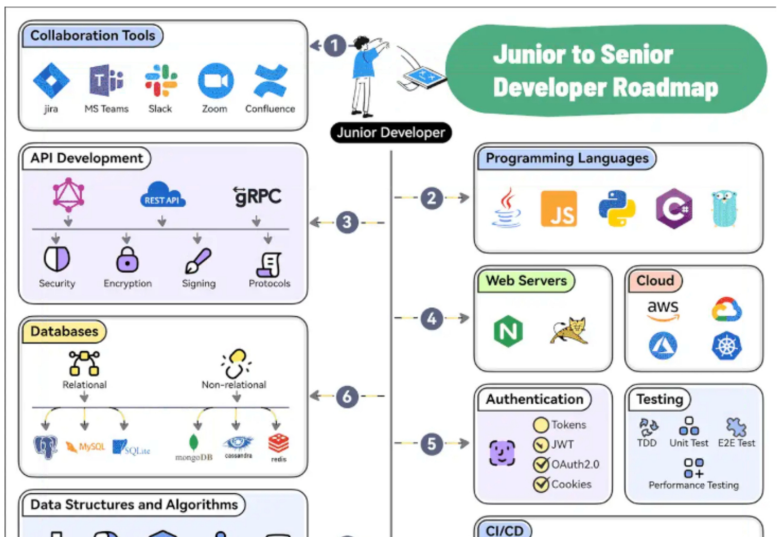
---

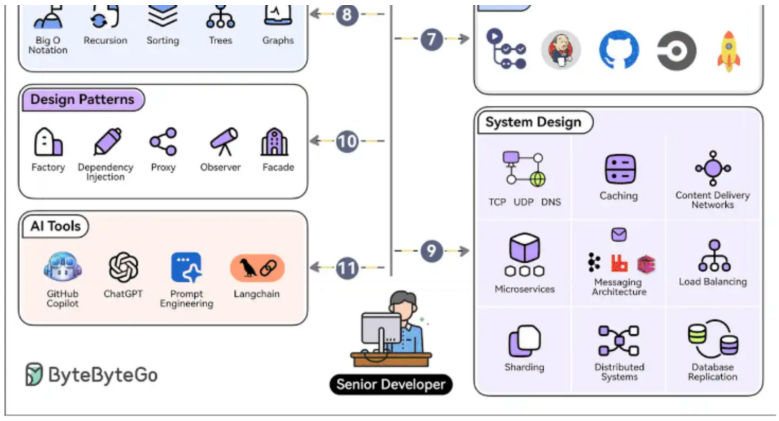
IV. Testcases Example :

IV. Testcases Example :

Implementing testcases for Array based problems : LC128

```
import unittest

def longest_consecutive(nums):
    numSet = set(nums)
    longest = 0

    for n in numSet:
        if (n - 1) not in numSet:
            length = 1
            while (n + length) in numSet:
                length += 1
            longest = max(length, longest)
    return longest

class TestLongestConsecutiveSequence(unittest.TestCase):

    def test_empty_list(self):
        self.assertEqual(longest_consecutive([]), 0)

    def test_single_element(self):
        self.assertEqual(longest_consecutive([1]), 1)

    def test_all_elements_same(self):
        self.assertEqual(longest_consecutive([2, 2, 2, 2]), 1)

    def test_all_consecutive(self):
        self.assertEqual(longest_consecutive([1, 2, 3, 4, 5]), 5)

    def test_non_consecutive(self):
        self.assertEqual(longest_consecutive([10, 5, 12, 3, 55, 30, 4, 11, 2]), 4)

    def test_multiple_sequences(self):
        self.assertEqual(longest_consecutive([100, 4, 200, 1, 3, 2]), 4)

    def test_mixed_positive_negative(self):
        self.assertEqual(longest_consecutive([-1, -2, -3, 0, 1, 2]), 6)

    def test_large_input_with_duplicates(self):
        self.assertEqual(longest_consecutive([1, 2, 2, 3, 4, 4, 5, 6]), 6)

    def test_sequences_with_gaps(self):
        self.assertEqual(longest_consecutive([1, 9, 3, 10, 4, 20, 2]), 4)

    def test_negative_numbers_only(self):
        self.assertEqual(longest_consecutive([-5, -4, -3, -2, -1]), 5)

    def test_unordered_with_duplicates(self):
        self.assertEqual(longest_consecutive([4, 4, 4, 3, 2, 2, 1, 1]), 4)

    def test_long_consecutive_sequence(self):
        self.assertEqual(longest_consecutive(list(range(1, 100001))), 100000)

    def test_consecutive_sequence_wrapped(self):
        self.assertEqual(longest_consecutive([0, -1, 1, 2, -2, 3]), 6)

    # Add more test cases as needed

if __name__ == '__main__':
    unittest.main()
```





V.) LINK FOR GRAPHS & TREES TEMPLATES

Comments: 0

Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post