# Contents



# Backtracking

1. Always sketch out the recursion tree in backtracking problems, solving becomes easy from there

## String Partitioning

1. Partition a string into all possible substrings
2. Uses recursive calls to partition and check whatever valid condition needs to be checked (eg. is a palindrome, is present in a dictionary etc.)
3. Example Problems

| Problem | Link | Notes |
|---|---|---|
| Palindrome Partitioning | https://leetcode.com/problems/palindrome-partitioning/description/ | condition is palindrome |
| Word Break II | https://leetcode.com/problems/word-break-ii/description/ | condition is present in dictionary |

```python
result = []
n = len(s)

def backtrack(curr_partition, start):
    if start == n:
        result.append(curr_partition.copy())
        return

    for end in range(start + 1, n + 1):
        substring = s[start:end]
        if is_valid_substring(substring): #is_valid_substring changes with the type of problem at hand
            curr_partition.append(substring)
            backtrack(curr_partition, end) #Can also use backtrack(curr_partition + [substring], end)
            curr_partition.pop()

backtrack([], 0)
return result
```

`s`: The input string to be partitioned.

`is_valid_substring`: A function that checks if a given substring is valid (e.g., checks if it is a palindrome or if it is in a dictionary).

`result`: A list to store all the valid partitions.

`backtrack` **Function**: A helper function that performs the actual backtracking.

1. `curr_partition`: The current partition being constructed.
2. `start`: The starting index for partitioning the string.

**Base Case**: If `start` reaches the end of the string (`n`), we add the current partition to the result.

**Recursive Case**: We iterate through possible end indices (`end`), generate substrings from `start` to `end`, and check if they are valid using the `is_valid_substring` function. If valid, we recursively call `backtrack` with the new substring added to the current partition. After the recursive call, we backtrack by removing the last added substring.

## Subsets

1. Recursively generate all subsets from a given set
2. Can use 2 principles, 1st is for loop (so subsets starting with element) and 2nd is inclusion exclusion (Add curr to result only at the end)
    1. It is better to use inclusion/exclusion when you want all subsets without any restrictions (combination sum, original subsets problem etc.), otherwise when you have restrictions (length, no duplicates etc.), it is better to use for loop strategy
3. Can slightly modify the logic on what subsets to include based on any conditions (duplicates etc.)
4. Example problems

| Problem | Link | Notes |
|---|---|---|
| Subsets | https://leetcode.com/problems/subsets/description/ | |
| Subsets II | https://leetcode.com/problems/subsets-ii/description/ | No Duplicate Subsets |
| Combinations | https://leetcode.com/problems/combinations/description/ | Subsets of length k |
| Combination Sum | https://leetcode.com/problems/combination-sum/description/ | Repetition Allowed |
| Combination Sum II | https://leetcode.com/problems/combination-sum-ii/description/ | No Repetition |

```python
result = []
n = len(nums)

def backtrack(curr_subset, start):
    result.append(list(curr_subset))

    for i in range(start, n):
        curr_subset.append(nums[i]) # Add the current subset to the result
        backtrack(curr_subset, i + 1) # Move to the next element
        curr_subset.pop() # Exclude the current element (backtrack)

backtrack([], 0)
return result
```

```python
result = []
n= = len(nums)
```

```
def backtrack(curr, i):
    if i == n: # Base case: if we've considered all elements
        result.append(curr)
        return

    dfs(curr + [nums[i]], i + 1) # Inclusion: include nums[i] in the subset
    dfs(curr, i + 1) # Exclusion: exclude nums[i] from the subset

backtrack([], 0)
return result
```

`nums`: The input list of numbers from which to generate subsets.

`result`: A list to store all the subsets.

`backtrack` **Function**: A helper function that performs the actual backtracking.

1. `curr_subset`: The current subset being constructed.
2. `start`: The starting index for the next element to consider.

**Base Case**: Every time we call `backtrack`, we add the current subset (`curr_subset`) to the result.

**Recursive Case**: We iterate through the elements starting from `start` to `n`, include the current element in the subset, and recursively call `backtrack` with the next starting index. After the recursive call, we backtrack by removing the last added element to explore other subsets.

## Code to identify non repeating elements in an array

```python
prev = None

for i in range(n):
    if arr[i] != prev:
        # Do something with the unique element 'arr[i]'
        print(arr[i])  # Replace this with your desired operation
    prev = arr[i]

# Or easier, just use continue statement, if arr[i]==arr[i-1]: continue
```

## Permutations

1. Just a slight variation of the combinations problem. Permutations generate all possible orders of elements in a given set.
2. The idea is to explore every possible order by fixing one element at a time and recursively permuting the remaining elements.
3. Example problems:

| Problem | Link | Notes |
|---|---|---|
| Permutations | https://leetcode.com/problems/permutations/description/ | |
| Permutations II | https://leetcode.com/problems/permutations-ii/ | No Duplicates |

```python
result = []
n = len(nums)

def backtrack(curr_permutation, used):
    if len(curr_permutation) == n: # Base case: if the current permutation is of length n, add it to result
        result.append(list(curr_permutation))
        return

    for i in range(n):
        if used[i]: # Skip already used elements
            continue

        used[i] = True # Mark the current element as used
        curr_permutation.append(nums[i])
        backtrack(curr_permutation, used) # Recurse with the updated permutation and used status
        used[i] = False # Backtrack: unmark the element and remove it from the current permutation
        curr_permutation.pop()

backtrack([], [False] * n)
return result

#You can maintain used, or just directly check if nums[i] is present in curr (this is O(n)), and if yes, just skip it.
```

`nums`: The input list of numbers for which we want to generate permutations.

`result`: A list to store all the permutations.

`backtrack` **Function**: A helper function to perform the actual backtracking.

- `curr_permutation`: The current permutation being constructed.
- `used`: A boolean list to keep track of which elements are used in the current permutation.

**Base Case**: If the length of `curr_permutation` is equal to `n`, the current permutation is added to the result.

**Recursive Case**: Iterate through the elements, check if the current element is used, and recursively generate permutations with the rest of the elements. After the recursive call, backtrack by marking the element as unused and removing it from the permutation.

## Constructing Valid Configurations

1. Construct valid solutions that adhere to specific constraints (like placing elements on a grid)

2. Example Problems:

| Problem | Link | Notes |
|---|---|---|
| N-Queens | https://leetcode.com/problems/n-queens/description/ | |
| Sudoku Solver | https://leetcode.com/problems/sudoku-solver/description/ | |

```python
def is_valid(inputs):
    #Check the validity (n-queens is valid, sudoku board is valid etc.)
def backtrack(inputs):

    #the outer loops can changem this template is not 100% fitting, just for idea
    #for loop (whatever you want to loop on)
        if is_valid(inputs):
            board[row][col] = 'Q'  # Place queen (In case of sudoku, its number)
            #backtrack
            board[row][col] = '.'  # Backtrack (remove queen/number)

backtrack(inputs)
return result
```

Note: Have to write code templates for dynamic programming + backtracking, DFS/BFS + backtracking. Mostly will be covered in DP and graph subsections.

Note: There's also problems like valid parentheses, that don't fall into any of the patterns listed above, but it's just general backtracking and knowing when to stop (opening brackets >= closing brackets)

# Graphs

1. Most leetcode problems in graphs are either adjacency lists, or grid based problems. Occasionally, you might encounter graphs represented using Nodes, and graphs represented using adjacency matrix.

2. Try to pass both the graph, visited set as arguments to the function, as in Python, only references to mutable objects are passed, so it is the same object, not a copy.

## Converting edges to adjacency list

```python
def edge_list_to_adj_list(edges: list, n: int):
    # Create an empty adjacency list with default as an empty list
    adj_list = defaultdict(list)

    # Iterate over the edge list to populate the adjacency list
    for u, v in edges:
        adj_list[u].append(v)
        adj_list[v].append(u)  # If the graph is undirected, add both ways

    return adj_list
```

## DFS vs BFS

**DFS:**

1. **Mark node as visited**: When **popped** from the stack (ready to process).

2. **Why?**: Ensures full exploration of neighbors before marking as visited.

3. **Additional visited check**: Needed before **pushing neighbors onto the stack** to avoid pushing the same node multiple times (because DFS might revisit nodes from different branches). (Only in case of iterative)

4. **Recursive DFS**:

- No need for an additional visited check because the recursion stack inherently manages depth-first traversal, and nodes are marked as visited immediately when the function is called.
- The call stack prevents revisiting nodes by the nature of recursion.
- In **recursive DFS**, you process the node first, recursively call neighbors, and only after all recursive calls are done does the node "pop" from the stack (when the function returns).

5. **Iterative DFS**:
   - Requires two visited checks:
     1. **Before pushing neighbors** onto the stack, to avoid pushing already visited nodes.
     2. **Before processing the node** after popping from the stack, to ensure the node is only processed once, even if it's added to the stack multiple times from different paths.
   - In **iterative DFS**, you pop the node first, process it, then push neighbors to the stack.
6. The difference arises from how the call stack in recursion automatically manages depth-first exploration compared to manual stack management in iterative DFS, so don't worry too much, just memorize.

**BFS:**

1. **Mark node as visited**: When **enqueued** (immediately after adding to the queue).
2. **Why?**: BFS processes nodes level by level, so marking when enqueuing prevents revisiting and ensures the shortest path is maintained.
3. **No additional visited check**: Since nodes are marked visited when enqueued, they won’t be added to the queue again, making an additional check after dequeuing unnecessary.
4. By the way, normally for BFS, the main space complexity lies in the process rather than the initialization. For instance, for a BFS traversal in a tree, at any given moment, the queue would hold no more than 2 levels of tree nodes. Therefore, the space complexity of BFS traversal in a tree would depend on the***width***of the input tree.

**Key Points:**

1. **DFS** explores deeply; multiple paths might push the same node, so check before pushing.
2. **BFS** explores level by level; mark when enqueuing to ensure each node is processed once, in the correct order.
3. **Efficiency**: BFS marking on enqueue is optimal for reducing redundant checks.

# DFS

1. **DFS magic spell: 1]push to stack, 2] pop top , 3] retrieve unvisited neighbours of top, push them to stack 4] repeat 1,2,3 while stack not empty. Now form a rap !**

## Recursive DFS (adjacency list)

```python
def dfs(node: int, visited: set, graph: dict):
    # Mark the current node as visited
    visited.add(node)

    # Process the current node (e.g., print, collect data, etc.)
    print(f"Visiting node {node}")

    # Recursively visit all unvisited neighbors
```

```
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(neighbor, visited, graph)
```

### Recursive DFS (grid)

```python
def dfs(x: int, y: int, visited: set, grid: list):
    # Mark the current cell as visited
    visited.add((x, y))

    # Process the current cell (e.g., print, collect data, etc.)
    print(f"Visiting cell ({x}, {y})")

    # Define the directions for neighbors: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Get the grid dimensions
    rows, cols = len(grid), len(grid[0])

    # Recursively visit all unvisited neighbors within bounds
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited: #Here you can add additional conditions, like edge exists only if it is a 1 etc.
            dfs(nx, ny, visited, grid)
```

### Iterative DFS (adjacency list)

1. **When you push a node onto the stack**, you're only checking if it's **not visited yet at that moment**. However, **the same node might get added to the stack multiple times** through different paths before it is actually processed. Thats the reason why we check visited both at the beginning and before adding neighbor. Think 1 - 2 - 3 - 4 loop.

```python
visited = set()  # To track visited nodes
stack = [start]  # Initialize the stack with the starting node

while stack:
    node = stack.pop()  # Pop the last node added (LIFO order)

    if node not in visited:
        # Mark the node as visited
        visited.add(node)
        print(f"Visiting node {node}")

        # Push all unvisited neighbors onto the stack
        for neighbor in graph[node]:
            if neighbor not in visited:
                stack.append(neighbor)
```

### Iterative DFS (grid)

```python
visited = set()  # To track visited cells
stack = [(start_x, start_y)]  # Initialize the stack with the starting cell

# Define the directions for neighbors: up, down, left, right
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Get the grid dimensions
rows, cols = len(grid), len(grid[0])

while stack:
    x, y = stack.pop()  # Pop the last cell added

    if (x, y) not in visited:
        # Mark the current cell as visited
        visited.add((x, y))
        print(f"Visiting cell ({x}, {y})")

        # Push all unvisited neighbors onto the stack (within bounds)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited:
                stack.append((nx, ny))
```

## Recursive DFS to keep track of Path (adjacency list)

1. This works both in cyclic and acyclic graphs, as in the **backtracking step**, we are removing the node from the visited set once we finish exploring its neighbors. This prevents the algorithm from getting stuck in an infinite loop caused by cycles while still allowing revisits to nodes in different paths.

```python
def dfs(node, target, graph, visited, path, all_paths):
    visited.add(node)
    path.append(node)

    if node == target:
        # If we've reached the target, store the current path
        all_paths.append(list(path))
    else:
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor, target, graph, visited, path, all_paths)

    path.pop()  # Backtrack
    visited.remove(node)
```

## Recursive DFS to keep track of Path (grid)

```python
def dfs(x, y, target_x, target_y, grid, visited, path, all_paths):
    # Add current cell to the path and mark it as visited
    path.append((x, y))
    visited.add((x, y))
```

```
    # If we reach the target cell, store the current path
    if (x, y) == (target_x, target_y):
        all_paths.append(list(path))  # Store a copy of the path
    else:
        # Explore the 4 possible directions: up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        rows, cols = len(grid), len(grid[0])

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            # Check if the next cell is within bounds and not visited
            if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited and grid[n
x][ny] == 1:
                dfs(nx, ny, target_x, target_y, grid, visited, path, all_paths)

    # Backtrack: remove the current cell from the path and unmark it as visited
    path.pop()
    visited.remove((x, y))
```

## Recursive DFS for topological sort (Directed Graph)

1. Key point is, Once all neighbors of the current node have been processed, the current node is added to the stack.
2. After performing DFS on all unvisited nodes, the stack will contain the nodes in reverse topological order (because nodes are pushed to the stack after their dependencies have been processed).
3. **Result**: The topological order is obtained by reversing the stack.
4. Some important points: This code will only work if there is no cycle, i.e , incase of a DAG. If you want it to work even when cycles are there, and return empty array if cycles are there, you need to add cycle detection logic.

```python
def dfs_topological(node, graph, visited, stack):
    visited.add(node)  # Mark the current node as visited

    # Recursively visit all unvisited neighbors
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_topological(neighbor, graph, visited, stack)

    # After all neighbors are processed, add the current node to the stack
    stack.append(node)

visited = set()  # Set to keep track of visited nodes
stack = []  # Stack to store the topological order

# Perform DFS from every node to ensure all nodes are visited
for node in range(n):
    if node not in visited:
        dfs_topological(node, graph, visited, stack)

# The topological order is the reverse of the DFS post-order traversal
return stack[::-1]
```

### Recursive DFS for Cycle Detection (Directed Graph)

1. Cycle detection is based on the Key point: In the current path, if there is back edge, i.e, node connecting to any previous nodes only in the current path, there is a cycle.
2. You cannot use visited to keep track of cycles, i.e claim that if we revisit the node there is a cycle, as a node maybe visited multiple times in DFS.
3. You also can't check something like if node in recursion_stack at the very beginning, because we will never visit the same node again due to us keeping track of visited. So that statement would never be True. So we always have to keep the main logic as detecting back edge.

```python
def dfs_cycle(node, graph, visited, recursion_stack):
    visited.add(node)  # Mark the node as visited
    recursion_stack.add(node)  # Add the node to the current recursion stack

    # Explore the neighbors
    for neighbor in graph[node]:
        if neighbor not in visited:
            if dfs_cycle(neighbor, graph, visited, recursion_stack):
                return True  # Cycle detected (If you don't do this, True won't be propogated)
        elif neighbor in recursion_stack:
            return True  # Cycle detected (back edge found)

    # Backtrack: remove the node from the recursion stack
    recursion_stack.remove(node)
    return False

def dfs_cycle(node, graph, visited, recursion_stack):
    if node in recursion_stack:
        return True
    if node in visited:
        return False
    visited.add(node)  # Mark the node as visited
    recursion_stack.add(node)  # Add the node to the current recursion stack

    # Explore the neighbors
    for neighbor in graph[node]:
        if dfs_cycle(neighbor, graph, visited, recursion_stack):
                return True  # Cycle detected (If you don't do this, True won't be propogated)

    # Backtrack: remove the node from the recursion stack
    recursion_stack.remove(node)
    return False
```

## BFS

### BFS (adjacency list)

```python
visited = set()  # To track visited nodes
queue = deque([start])  # Initialize the queue with the starting node
```

```
visited.add(start)  # Mark the start node as visited when enqueuing

while queue:
    node = queue.popleft()  # Dequeue the first node in the queue
    print(f"Visiting node {node}")  # Process the node (e.g., print or collect data)

    # Enqueue all unvisited neighbors and mark them as visited when enqueuing
    for neighbor in graph[node]:
        if neighbor not in visited:
            queue.append(neighbor)
            visited.add(neighbor)  # Mark as visited when enqueuing
```

### BFS (grid)

```python
visited = set()  # To track visited cells
queue = deque([(start_x, start_y)])  # Initialize the queue with the starting cell
visited.add((start_x, start_y))  # Mark the start cell as visited when enqueuing

# Define the directions for neighbors: up, down, left, right
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
rows, cols = len(grid), len(grid[0])

while queue:
    x, y = queue.popleft()  # Dequeue the first cell
    print(f"Visiting cell ({x}, {y})")  # Process the current cell

    # Enqueue all unvisited valid neighbors
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in visited:
            queue.append((nx, ny))
            visited.add((nx, ny))  # Mark as visited when enqueuing
```

### Multisource BFS

1. The multi-source BFS pattern is useful when you need to start BFS from multiple starting points simultaneously. This pattern ensures that all sources are explored in parallel, and it's commonly used in problems like finding the shortest distance from multiple sources to a destination.

2. The only change is from normal BFS code is that you add all the source nodes in the queue and call BFS

### Maintaining Level information in BFS

1. Simple way is just to maintain (node, level) instead of just node. Each time you are enqueuing new nodes, increment the level by 1. This way, you have level information for all the nodes. In this method, level information will be lost at the end, as the queue will become empty. It can still be used if you only need the end result, but if you need information like no. of nodes in each level etc., It is better to use level processing approach.

2. Other way is using array for levels, like so. Idea is to process nodes level by level, tracking the current level by processing all nodes at the same depth in one batch, and incrementing the level after processing each layer. Useful in tree problems (level order traversal) too.

```python
visited = set([start])  # Track visited nodes, starting with the source node
queue = deque([start])  # Queue to store nodes to be processed
level = 0  # Start from level 0 (the level of the start node)

while queue:
    # Get the number of nodes at the current level
    level_size = len(queue)

    # Process all nodes at the current level
    for _ in range(level_size):
        node = queue.popleft()  # Pop a node from the queue
        print(f"Node: {node}, Level: {level}")

        # Add unvisited neighbors to the queue
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    # After processing all nodes at the current level, increment the level
    level += 1
```

### BFS for topological sort (Cycle detection built in) (Kahn's algorithm)

```python
graph = defaultdict(list)  # Adjacency list representation of the graph
in_degree = [0] * n  # In-degree of each node

# Build the graph and calculate in-degrees
for start, end in edges:
    graph[start].append(end)
    in_degree[end] += 1

# Initialize the queue with all nodes that have in-degree of 0
queue = deque([i for i in range(n) if in_degree[i] == 0])
topo_order = []

while queue:
    node = queue.popleft()  # Get the node with zero in-degree
    topo_order.append(node)  # Add it to the topological order

    # Reduce in-degree of all its neighbors
    for neighbor in graph[node]:
        in_degree[neighbor] -= 1
        # If a neighbor now has in-degree of 0, add it to the queue
        if in_degree[neighbor] == 0:
            queue.append(neighbor)

# If all nodes are processed, return the topological order, otherwise return empty (cycle detected)
if len(topo_order) == n:
    return topo_order
```

```
else:
    return []  # Cycle detected
```

## Eulerian Path/Cycle

1. **Eulerian Path**: If there is exactly one vertex with `out-degree` greater by 1 and one with `in-degree` greater by 1.
2. **Eulerian Cycle**: If all vertices have equal `in-degree` and `out-degree`
3. Basically Eulerian Path/Cycle means we cover all edges of a graph exactly once
4. The algorithm is simple, we recursively keep removing edges one by one, so no need of visited set as we can revisit a node multiple times, but cannot revisit an edge. More elegant implementations also exist, but this should suffice for this rare problem.

```python
def dfs_eularian(node, graph, stack):
    while graph[node]:
        next_node = graph[node].pop(0) # Only if lexcial order pop(0), else its fine to pop any neighbor
        dfs_eularian(next_node)
    stack.append(node)

# Start DFS from the determined starting airport
dfs_eularian(start) # For determining start node, follow the instructions in the notes above.
return stack[::-1]  # Reverse the itinerary to get the correct order
```

## Disjoint Set Union / Union Find

1. **Purpose**: DSU is used to manage and merge disjoint sets, mainly in graph problems for tracking connected components and detecting cycles.
2. **Key Operations**:
   - **Find** with Path Compression: Reduces the time complexity by flattening the tree, so future operations are faster.
   - **Union by Rank/Size**: Keeps the tree balanced by attaching the smaller tree under the root of the larger one.
3. **Time Complexity**: Both `find` and `union` have nearly constant time complexity, due to path compression and union by rank
4. **Common Use Cases**:
   - **Cycle Detection** in undirected graphs.
   - **Kruskal's MST Algorithm** to avoid cycles when adding edges.
   - **Connected Components** to check if two nodes are in the same component.
5. **Initialization**: Use two arrays—`parent` (each node points to itself initially) and `rank` (initially 0 for all nodes).

```python
class DSU:
    def __init__(self, n):
        # Initialize parent and rank arrays
```

```
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, x):
        '''
        # Intialize parent and rank as dicts {} if no. of nodes is not known, and just
add these lines of code
        # Initialize parent and rank if node is encountered for the first time
        if x not in self.parent:
            self.parent[x] = x
            self.rank[x] = 0
        '''
        # Find the root of the set containing x with path compression
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Union by rank
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            # Attach smaller rank tree under root of the higher rank tree
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1

    def connected(self, x, y):
        # Check if two elements are in the same set
        return self.find(x) == self.find(y)
```

## Minimum Spanning Trees (MST)

1. An **MST** connects all nodes in an undirected, weighted graph with the minimum possible total edge weight, ensuring there are no cycles and the graph remains fully connected.

## MST Kruskal's

1. **Approach**: Edge-based, Greedy

2. **Process**:

   - Sort all edges in non-decreasing order by weight.
   - Initialize an empty MST and start adding edges from the sorted list.
   - For each edge, check if it forms a cycle using DSU. If not, add it to the MST.
   - Repeat until the MST has V-1 exactly edges (where V is the number of vertices).

3. **Best for**: Sparse graphs where sorting edges is manageable.

```python
def kruskal_mst_fixed(edges):
    # Initialize DSU and collect unique nodes to determine number of nodes
    dsu = DynamicDSU()
    unique_nodes = set(u for u, v, _ in edges).union(set(v for u, v, _ in edges))
    num_nodes = len(unique_nodes)

    # Sort edges by weight (ascending order)
    edges.sort(key=lambda x: x[2])

    mst = []  # To store edges in MST
    total_cost = 0

    # Iterate through sorted edges
    for u, v, weight in edges:
        # Only add edge if it doesn't form a cycle
        if not dsu.connected(u, v):
            dsu.union(u, v)  # Union the two vertices
            mst.append((u, v, weight))  # Add edge to MST
            total_cost += weight  # Add edge weight to total cost

            # Stop if MST has enough edges (n - 1 edges for n nodes)
            if len(mst) == num_nodes - 1:
                break

    return mst, total_cost
```

## MST Prim's

1. **Approach**: Vertex-based, Greedy
2. **Process**:
    - Start from any initial node.
    - Use a min-heap (priority queue) to track edges that extend from the MST.
    - Repeatedly pop the minimum edge from the heap:
        - If it connects to an unvisited node, add it to the MST and add its neighbors to the heap.
    - Stop when the MST includes all nodes.
3. **Best for**: Dense graphs with adjacency lists/matrices.

```python
# Redefining Prim's algorithm for MST with adjacency list input
def prim_mst(graph, start):
    # Initialize structures
    mst = []  # To store the edges in the MST
    total_cost = 0  # To accumulate the total weight of the MST
    visited = set()  # Track nodes already included in the MST
    min_heap = []  # Priority queue (min-heap) for edges
```

```python
    # Function to add edges to the priority queue
    def add_edges(node):
        visited.add(node)
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                heapq.heappush(min_heap, (weight, node, neighbor))

    # Start from the initial node
    add_edges(start)

    # Process until MST includes all nodes or min-heap is empty
    while min_heap and len(visited) < len(graph):
        weight, u, v = heapq.heappop(min_heap)
        if v not in visited:  # Only add edge if it connects to an unvisited node
            mst.append((u, v, weight))
            total_cost += weight
            add_edges(v)  # Add edges from the newly added node

    return mst, total_cost
```

## Shortest Path Dijkstra

1. **Approach**: Single-source shortest path for non-negative weights
2. **Process**:
   - Initialize distances from the start node to all other nodes as infinity (except for the start, set to 0).
   - Use a min-heap to manage nodes by their current shortest distance.
   - Pop the node with the smallest distance:
     - For each neighbor, calculate the potential new distance.
     - If this distance is shorter than the known distance, update it and push the neighbor with the updated distance.
   - Continue until all reachable nodes have the shortest path from the start.
3. **Best for**: Shortest paths in non-negative weighted graphs.

```python
def dijkstra(graph, start):
    # Initialize distance dictionary with infinity for all nodes except the start
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Priority queue (min-heap) initialized with the starting node
    min_heap = [(0, start)]  # (distance, node)

    while min_heap:
        # Pop the node with the smallest distance
        current_distance, u = heapq.heappop(min_heap)

        # Process only if the current distance is the smallest known distance for u
        if current_distance > distances[u]:
```

```
            continue

        # Check each neighbor of the current node
        for v, weight in graph[u]:
            distance = current_distance + weight  # Calculate potential new distance to
neighbor

            # Only consider this path if it's shorter than the known distance
            if distance < distances[v]:
                distances[v] = distance  # Update to the shorter distance
                heapq.heappush(min_heap, (distance, v))  # Push updated distance into t
he heap

    return distances
```

### Tips and Tricks to Solve graph problems

1. To detect length of cycle or elements in cycle, you can keep track of entry times in the recursive_stack. This can also help you in finding the exact cycle.

2. For **undirected graphs**, cycles are found using DSU or DFS with back edges. For **directed graphs**, cycles are detected through **DFS with recursion stack tracking**.

# Dynamic Programming

1. General tip - In bottom up DP, if 2D DP, draw the matrix and visualize the dependencies, becomes easier.

2. Sometime memoization is more intuitive, sometime DP is more intuitive. DP you can usually perform space optimization.

### 0/1 Knapsack Pattern -

```python
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Fill the DP table
    for i in range(1, n + 1):  # For each item
        for w in range(1, capacity + 1):  # For each capacity
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i -
1]]) #i-1 is the actual item
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]
```

### Unbounded Knapsack Patten - 322, 343, 279

```python
def unbounded_knapsack(values, weights, capacity):
    n = len(values)
    dp = [0] * (capacity + 1)
```

```python
    # Fill the DP array
    for i in range(n):  # For each item
        for w in range(weights[i], capacity + 1):  # For each capacity
            dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

    return dp[capacity]
```

1. Coin Change II - Since ordering doesnt matter, it is a 2 state problem instead of 1 state. little tricky to catch. You use inclusion/exclusion decision tree, memoization solution is simple to implement.

2. If ordering does matter, then it is a simple 1 state solution like Coin Change I

## Fibonacci

1. The Fibonacci pattern shows up in many dynamic programming problems where each state depends on a fixed number of previous states.

```python
def fibonacci_dp(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

## Longest Palindromic Substring

1. If you know that a substring `s[l+1:r-1]` is a palindrome, then `s[l:r]` is also a palindrome if `s[l] == s[r]`.

2. In problems involving **palindromic substrings or subsequences**, the goal is often to:

   - **Identify the longest palindromic substring** (continuous sequence).

   - **Count the number of palindromic substrings**.

   - **Find the longest palindromic subsequence** (which doesn't need to be contiguous).

3. Important: Diagonal Filling of the matrix

```python
def longestPalindrome(self, s: str) -> str:
n= len(s)
dp = [[False]*n for _ in range(n)]

for i in range(n):
    dp[i][i] = True
ans = [0,0]

for i in range(n - 1):
    if s[i] == s[i + 1]:
        dp[i][i + 1] = True
        ans = [i, i + 1]
```

```python
for i in range(n-1, -1, -1):
    for j in range(n-1, -1, -1):
        if j<=i+1:
            continue
        if s[i]==s[j] and dp[i+1][j-1]:
            dp[i][j] = True
            if j-i>ans[1]-ans[0]:
                ans = [i,j]
return s[ans[0]:ans[1]+1]
```

## Maximum Sum/Product Subarrays

```python
def max_subarray_sum(nums):
    max_sum = nums[0]
    current_sum = nums[0]

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

    return max_sum

def maxProduct(nums):
        prev_max, prev_min = nums[0], nums[0]
        ans=prev_max
        for i in range(1,len(nums)):
            curr_max = max(nums[i], nums[i]*prev_max, nums[i]*prev_min)
            curr_min = min(nums[i], nums[i]*prev_max, nums[i]*prev_min)
            prev_max, prev_min = curr_max, curr_min
            ans = max(ans, prev_max)
        return ans
```

## Word Break

```python
#O(n^2), dp[i] represents if word till i can be broken into parts. Important problem as
you need to check all previous indices.
```

## Longest Increasing Subsequence

1. The **Longest Increasing Subsequence (LIS) Pattern** is a common dynamic programming pattern used to find subsequences within a sequence that meet certain increasing criteria. This pattern typically involves identifying or counting **subsequences** (not necessarily contiguous) that satisfy conditions related to increasing order, longest length, or specific values.

2. **Define the DP Array**: Use an array `dp` where `dp[i]` represents the length of the longest increasing subsequence ending at index `i`.

3. **Transition**: For each element `i`, check all previous elements `j < i`. If `nums[j] < nums[i]`, update `dp[i] = max(dp[i], dp[j] + 1)` to extend the subsequence ending at `j`.

4. **Important**: O(nlogn) Use `tails` to store the smallest ending of increasing subsequences. For each `num`, use binary search to find its position in `tails` – replace if within bounds, or append if beyond (is the last element)

```python
def length_of_lis(nums):

    dp = [1] * len(nums)  # Each element is at least an increasing subsequence of lengt
h 1

    for i in range(1, len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

## Counting Paths/Combinations Pattern

1. **Goal**: Given a target, find distinct ways to reach it based on given moves/rules.
2. **DP Array/Table**: Use `dp[i]` or `dp[i][j]` to store the count of ways to reach each target or cell.
3. **Common Formula**: For each `i`, update `dp[i]` by summing counts from preceding states based on allowed moves.

**Key Examples**

1. **Climbing Stairs**: `dp[i] = dp[i-1] + dp[i-2]`
2. **Coin Change (Combinations)**: `dp[i] += dp[i - coin]` for each coin
3. **Grid Unique Paths**: `dp[i][j] = dp[i-1][j] + dp[i][j-1]`

## Longest Common Subsequence (LCS) Pattern

1. **Goal**: Compare two sequences and find:
    - Length of Longest Common Subsequence (LCS).
    - Minimum edits to transform one sequence into another (Edit Distance).
    - Longest contiguous substring (Longest Common Substring).
2. **Key Idea**: Use a 2D DP table `dp[i][j]` where:
    - `dp[i][j]` represents the result for substrings `s1[:i]` and `s2[:j]`.
3. **Base Cases**: If `i == 0` or `j == 0`, the result is `0` (empty string).

```python
def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
```

```python
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

### Variants

1. **Edit Distance**: Modify `dp[i][j]` transition to consider insert, delete, substitute. Also modify to correct base case.

```python
dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+(1 if s1[i-1]!=s2[j-1] else 0))
```

2. **Longest Common Substring**: If characters match, add `+1` to previous diagonal:

```python
if text1[i - 1] == text2[j - 1]: dp[i][j] = dp[i - 1][j - 1] + 1 else: dp[i][j] = 0
```

## State based DP Pattern

1. **Goal**: Optimize decisions across states influenced by actions (e.g., buying/selling, resting/working).
2. **Key Idea**: Define **states** to track situations (e.g., holding stock, not holding, cooldown) and **transition equations** between states.
3. **Steps**:
   - Identify **states** based on possible actions.
   - Draw a **state diagram** to visualize transitions.
   - Write **recurrence relations** for each state based on dependencies.
4. **Base Cases**: Define initial conditions (e.g., starting with no stock or zero profit).

```python
def state_based_dp_problem(prices):
    # Base cases (initial states)
    hold = -prices[0]   # Max profit when holding stock on day 0
    not_hold = 0        # Max profit when not holding stock on day 0
    cooldown = 0        # Max profit in cooldown on day 0

    for i in range(1, len(prices)):
        prev_hold = hold
        # State transitions
        hold = max(hold, not_hold - prices[i])        # Continue holding or buy today
        not_hold = max(not_hold, cooldown)           # Continue not holding or end cooldown
        cooldown = prev_hold + prices[i]             # Sell today and enter cooldown

    # Final result: max profit can be in not_hold or cooldown
    return max(not_hold, cooldown)
```

## Todo

1. Negative marking
2. Prefix Sum + Hashmap pattern (also modulo if involved)
3. Sliding window + Hashmap (also using matched variable to avoid hashmap)
4. Monotonic Stack
5. QuickSelect Algorithm
6. Storing index as key in hashmap (Problem #3 leetcode)
7. Sliding window - complement of sliding window pattern **2516. Take K of Each Character From Left and Right**

# Arrays + Hashmaps

## QuickSelect

1. **Use Cases**:
   - **Kth largest/smallest element** or **Top K elements** in O(n) average time.
   - Avoids full sorting for subset problems (better than nlogn).
2. **Key Insight**:
   - Partition around a **pivot** to partially sort: left satisfies the comparator, pivot lands in its correct position.
3. **How to Use**:
   - Set `k = k - 1` for 0-based indexing.
   - Use `x >= y` for **descending order** (Kth largest, Top K).
   - Use `x <= y` for **ascending order** (Kth smallest, Bottom K).

```python
def quickselect(arr, left, right, k, comparator):
    def partition(arr, left, right):
        pivot = arr[right]  # Choose the last element as pivot
        p = left  # Pointer for elements satisfying comparator
        for i in range(left, right):
            if comparator(arr[i], arr[right]):  # Compare with pivot
                arr[i], arr[p] = arr[p], arr[i]
                p += 1
        arr[p], arr[right] = arr[right], arr[p]  # Place pivot in position
        return p

    if left <= right:
        pivot_index = partition(arr, left, right)
        if pivot_index == k:  # Found the Kth element
            return arr[pivot_index]
        elif pivot_index < k:  # Look for Kth in the right part
            return quickselect(arr, pivot_index + 1, right, k, comparator)
        else:  # Look for Kth in the left part
            return quickselect(arr, left, pivot_index - 1, k, comparator)

# Comparator for kth largest and top k elements
```

```
comparator = lambda x, y: x >= y

# Comparator for kth smallest and bottom k elements
comparator = lambda x, y: x <= y
```

## Prefix Sum (Prefix Sum, Prefix Sum+Binary Search, Prefix Sum+Hashmaps)

1. **Key Idea**: Combine prefix sums with a hashmap to store cumulative sums and solve subarray problems dynamically.
2. `prefix[i] - prefix[j] = k => prefix[j] = prefix[i] - k`
3. **Use Cases**:
   - Count subarrays with specific conditions (e.g., sum equals k).
   - Modular conditions (e.g., divisible by k).
4. Important points: **Particularly useful when sliding window approaches fail (because of presence of negative numbers), If only +ve numbers are present, prefix_sum array is sorted, can think if binary search is needed.**

```python
def prefix_sum_with_hashmap(arr, k):
    prefix = 0
    count = 0
    hashmap = {0: 1}  # Initialize with prefix 0 to handle exact matches

    for num in arr:
        prefix += num
        # Check if prefix - k exists in the hashmap
        if (prefix - k) in hashmap:
            count += hashmap[prefix - k]
        # Update the hashmap with the current prefix
        hashmap[prefix] = hashmap.get(prefix, 0) + 1

    return count
```

## Hashsets (Building consecutive sequences pattern)

**Key Idea:**

1. Use a `HashSet` for:
   - Quick lookups for presence/absence.
   - Ensuring uniqueness of elements.
   - Problems involving element relationships (e.g., consecutive sequences).

```python
def longest_consecutive(nums):

    num_set = set(nums)
    longest = 0

    for num in num_set:
```

```
        # Check if it's the start of a sequence
        if num - 1 not in num_set:
            length = 0
            current = num
            while current in num_set:
                length += 1
                current += 1
            longest = max(longest, length)

    return longest
```

## Sorting

```python
# 1. Sort in Ascending Order
arr.sort()  # [1, 3, 5, 8]

# 2. Sort in Descending Order
arr.sort(reverse=True)  # [8, 5, 3, 1]

# 3. Sort by the Second Element in a Tuple
arr.sort(key=lambda x: x[1])  # [(3, 1), (1, 2), (2, 3)]

# 4. Sort by Length of Strings
arr.sort(key=lambda x: len(x))  # ['kiwi', 'apple', 'banana']

# 5. Sort by Multiple Keys (Primary Ascending, Secondary Descending)
arr.sort(key=lambda x: (x[0], -x[1]))  # [(1, 3), (1, 2), (2, 3), (2, 2)]

# 6. Sort by the second element in a tuple using sorted()
sorted_arr = sorted(arr, key=lambda x: x[1])  # [(3, 1), (1, 2), (2, 3)]
```

## Negative Marking

1. The problem involves integers bounded by the size of the array (e.g., values from 1 to n).

2. You're asked to find duplicates, missing elements, or cycles in O(n) time and O(1) space.

3. The input array can be modified in-place.

```python
def find_duplicates(nums): #nums has only values 1 to n
    res = []
    for num in nums:
        index = abs(num) - 1  # Map value to index
        if nums[index] < 0:
            res.append(abs(num))  # Already marked negative -> duplicate
        else:
            nums[index] = -nums[index]  # Mark as visited
    return res
```

## Majority Element (Boyer-Moore Voting Algorithm + Hashmap Alternative)

1. Candidate votes for itself, all other candidates vote against it

```python
def majority_element(nums):

    # Step 1: Find the candidate
    candidate, count = None, 0
    for num in nums:
        if count == 0:
            candidate = num
        count += 1 if num == candidate else -1
```

# 2 pointers

## Opposite Direction Two Pointers

Sorting may be useful if not explicitly prohibited

In the **Opposite Direction Two Pointers** pattern, two pointers are initialized at opposite ends of a list or array, and they are moved toward each other based on conditions to solve a problem. This approach is widely applicable to problems involving sorted arrays, searching for optimal solutions, or evaluating complex conditions involving multiple indices.

Use `l < r`: For problems comparing pairs of elements where overlapping isn't meaningful (most problems, 2 sum sorted etc). Use `l <= r`: For problems where overlapping or single element checks are necessary (palindrome check).

```python
def opposite_direction_two_pointers(arr, condition):
    left, right = 0, len(arr) - 1

    while left < right:
        # Perform actions based on condition
        if condition(arr[left], arr[right]):
            # Example: process a valid pair
            process(arr[left], arr[right])

        # Update pointers based on the problem logic
        if move_left_condition:  # Replace with actual condition
            left += 1
        elif move_right_condition:  # Replace with actual condition
            right -= 1
        else:
            # Break if no further action is possible
            break
```

**General Problem Categories**

1. **Simple Conditions**:
   - Problems with straightforward conditions for moving pointers (e.g., sums, comparisons, or matching characters).
   - **Examples**:
     - Two-Sum in a sorted array.

- Checking if a string is a palindrome.

2. **Complex Conditions**:
   - Problems where the condition to move pointers involves more elaborate calculations or logic.
   - **Examples**:
     - Maximizing or minimizing values (e.g., Container with Most Water).
     - Aggregating values across the pointers (e.g., Trapping Rain Water).
3. **Advanced Applications**:
   - Problems that incorporate sorting or nested loops (e.g., counting triplets or evaluating multiple conditions)
   - **Examples:**
     - Valid triangle number, 3-sum
     - Note: Was not able to solve 3-sum (**including all duplicates**) using this method

```python
def opposite_direction_with_sorting(arr):
    # Step 1: Sort the array if needed
    arr.sort()

    # Step 2: Iterate through the array with one fixed pointer
    for i in range(len(arr)):
        left, right = i + 1, len(arr) - 1 #Can be adjusted, you can start at end of array too, like Valid triangle

        # Step 3: Use two pointers to evaluate the condition
        while left < right:
            if condition(arr[i], arr[left], arr[right]):
                process(arr[i], arr[left], arr[right])
                # Adjust pointers based on requirements
                left += 1
                right -= 1
            elif adjust_left_condition:  # Example condition to move left
                left += 1
            else:  # Adjust right
                right -= 1
```

## Same Direction Pointers (Not Sliding Window)

**Note: In cases where you can modify input array, you can use 2 pointers, and overwrite the original array if your final answer is always smaller (length wise) than original array. We dont care about elements already processes, e.g. Encode String (aabbbb to a2b4)**

This pattern involves two pointers (`left` and `right`) that move in the same direction. The **right pointer** moves first until a condition is met or unmet. Once the condition changes, the **left pointer** is adjusted to the position of the right pointer. This is **not a sliding window** since the left pointer does not increment gradually but instead jumps to match the right pointer.

**Key Insights**

1. **Right Pointer Moves First**:

- Start with `left` and `right` at the same position.
- Increment `right` while evaluating a condition.

2. **Adjust Left Pointer**:

- When the condition is violated or met, bring the `left` pointer to match the `right` pointer.

3. **Non-Overlapping Subarrays**:

- This pattern ensures that the ranges between `left` and `right` pointers are disjoint or non-overlapping.

```python
def same_direction_pointers(arr):
    left = 0
    right = 0
    while right < len(arr):
        # Expand the right pointer
        while right < len(arr) and condition(arr[right]):
            #Can have complex logic here instead of having condition above
            right += 1

        # Process the range [left, right)
        process(arr[left:right])

        # Increment right pointer by 1 to start new window
        right+=1
        # Move the left pointer to match the right
        left = right
```

**Common Applications**

1. **Splitting Strings or Arrays**:

- Breaking a sequence into segments based on a condition.

2. **Processing Subarrays**:

- Analyze non-overlapping subarrays meeting specific criteria.

3. **Counting or Extracting Ranges**:

- Count segments, extract substrings, or find subarrays.

4. **Skipping Invalid Values**:

- Handle sequences with gaps or delimiters by skipping invalid parts.

## Three pointers

Basically partition array into three groups by some conditions.

```python
def sort_colors(nums):
    p1, p2, p3 = 0, 0, len(nums) - 1

    while p2 <= p3:
        if nums[p2] == 0:  # Move 0s to the left
            nums[p1], nums[p2] = nums[p2], nums[p1]
```

```
            p1 += 1
            p2 += 1
        elif nums[p2] == 1:  # Keep 1s in the middle
            p2 += 1
        else:  # Move 2s to the right
            nums[p2], nums[p3] = nums[p3], nums[p2]
            p3 -= 1
```

### Pattern: Two Pointers on Two Different Arrays/Strings

This pattern involves using two pointers, each operating on a separate array or string. The pointers traverse independently or interact based on specific conditions to solve a problem efficiently.

```python
def two_pointers_on_two_arrays(arr1, arr2):
    # Initialize two pointers
    i, j = 0, 0
    result = []

    while i < len(arr1) and j < len(arr2):
        if condition(arr1[i], arr2[j]):
            process(arr1[i], arr2[j], result)
            i += 1
            j += 1
        elif adjust_pointer_1_condition:
            i += 1
        else:
            j += 1

    # Process remaining elements if required
    while i < len(arr1):
        process(arr1[i], None, result)
        i += 1
    while j < len(arr2):
        process(None, arr2[j], result)
        j += 1

    return result
```

### Pattern: Fast and Slow Pointers on Arrays (Tortoise and Hare)

Just writing it down here, not very important, revisit if you have time. 2 problems - LeetCode 457: Circular Array Loop, LeetCode 202: Happy Number

# Sliding Window

### Fixed Size

```python
def fixed_size_sliding_window(arr, k):
    n = len(arr)
    window_sum = 0
    result = []
```

```
    # Initialize the first window
    for i in range(k):
        window_sum += arr[i]

    # Append the result of the first window
    result.append(window_sum)

    # Slide the window across the array
    for i in range(k, n):
        window_sum += arr[i] - arr[i - k]  # Add the next element, remove the first ele
ment of the previous window
        result.append(window_sum)

    return result
```

## Dynamic Size

```python
l = 0
for r in range(len(nums)):
    # Expand window by adding nums[r]
    update_window(nums[r])

    # Shrink window if condition is violated
    while condition_not_met():
        # Update result if required inside the loop (e.g., for min problems)
        update_window_on_shrink(nums[l])
        l += 1

    # Update result if required outside the loop (e.g., for max problems)
    update_result(l, r)
```

### Dynamic Sliding Window Notes

1. **General Rules**:
   - Use two pointers (`l`, `r`): expand with `r`, shrink with `l`.
   - **Update result inside** `while` if intermediate windows matter (e.g., **min problems**).
   - **Update result after** `while` if only the final window matters (e.g., **max problems**).
2. **Sliding Window + HashMap/Set**:
   - Use a hashmap/set to track element frequencies or uniqueness.
   - Shrink when the hashmap/set exceeds constraints (e.g., distinct elements > `k`).
3. **Index Trick (Last Occurrence)**:
   - Use a hashmap to store the last occurrence of an element.
   - Update `l` to `max(l, last_occurrence + 1)` to skip invalid windows.
4. **Matches Trick (Two HashMaps)**:

- Use two hashmaps and a `matches` variable to track when the window satisfies the target hashmap.
- Increment `matches` when counts match; decrement on invalid shrink.

5. **Sliding Window + Prefix Sum**:
   - Use prefix sums to compute subarray sums efficiently.
   - Track prefix sums in a hashmap for difference-based lookups.
6. **Sliding Window + Deque**:
   - Use a deque to maintain a monotonic order of indices/values in the window.
   - Useful for problems like finding max/min in a sliding window. (Covered in detail in queues section)
7. **Complement of Sliding Window**
   - The answer lies outside the window (example select minimum/maximum something from left and right) problem [2516. Take K of Each Character From Left and Right]
8. **Quirks and Edge Cases**:
   - For substring problems, slicing (`s[l:r+1]`) is useful.
   - Sliding window can combine with binary search for length checks.
   - Two-pass sliding window works when expansion and shrinking need separate logic.

# Stacks

## Stack Simulation Pattern

### 1. Direct Stack Usage

- **Use Case**: Maintain elements or operations in a stack to process them in LIFO order.
- **Key Steps**:
  1. Push elements onto the stack as needed.
  2. Pop elements off the stack to resolve conditions or complete operations.
- Example problems - Valid Paranthesis,

```python
stack = []
for char in input_sequence:
    if condition_to_push(char):
        stack.append(char)
    elif condition_to_pop(stack, char):
        stack.pop()
return result_based_on_stack(stack)
```

### 2. Auxiliary Stack for Tracking

- **Use Case**: Use a secondary stack to track auxiliary information (e.g., **min/max**, **wildcards**).
- **Key Steps**:

3. Use the main stack for primary operations.

4. Use the auxiliary stack to maintain additional data in sync.

5. Synchronize both stacks during push and pop operations.

```python
stack, aux_stack = [], []
for char in input_sequence:
    if condition_to_push(char):
        stack.append(char)
        aux_stack.append(update_aux(char, aux_stack))
    elif condition_to_pop(stack, char):
        stack.pop()
        aux_stack.pop()
return result_based_on_aux(aux_stack)
```

## Monotonic Stack

**Overview:**

- **Definition**: A stack that maintains elements in a strictly increasing or decreasing order (monotonic).
- **Use Case**: Solve problems involving nearest greater/smaller elements, range computations, and areas/volumes efficiently.
- **Key Ideas**:
  - Push elements while maintaining the order.
  - Pop elements when the order is violated, typically while processing conditions.

**Two Types of Monotonic Stacks:**

**1. Monotonically Increasing Stack**

- **Maintains elements in increasing order**.
- **Usage**: "Next Smaller Element" or "Nearest Smaller Element" problems. (Largest Rectangle in Histogram)

```python
def monotonically_increasing_stack(array):
    stack = []
    result = [-1] * len(array)  # Result to store required output (e.g., next smaller/greater)

    for i, val in enumerate(array):
        while stack and array[stack[-1]] > val:
            index = stack.pop()
            result[index] = val  # Process the popped element (e.g., update result)
        stack.append(i)  # Push the current index onto the stack

    return result
```

**2. Monotonically Decreasing Stack**

- **Maintains elements in decreasing order**.
- **Usage**: "Next Greater Element" or "Nearest Greater Element" problems. (Trapping Rain Water)

```python
def monotonically_decreasing_stack(array):
    stack = []
    result = [-1] * len(array)  # Result to store required output (e.g., next smaller/greater)

    for i, val in enumerate(array):
        while stack and array[stack[-1]] < val:
            index = stack.pop()
            result[index] = val  # Process the popped element (e.g., update result)
        stack.append(i)  # Push the current index onto the stack

    return result
```

# Queue

## Monotonic Queue (decreasing example, for sliding window maximum)

**Key Idea**:

- Remove elements from the front of the queue if they are not part of the current window
- Remove elements from the back of the queue that are smaller than the current element to maintain the order.

```python
from collections import deque

def monotonic_decreasing_queue(nums, k):
    queue = deque()
    result = []

    for i, num in enumerate(nums):
        # Remove indices out of the current sliding window
        if queue and queue[0] < i - k + 1:
            queue.popleft()

        # Remove elements smaller than the current element from the back
        while queue and nums[queue[-1]] < num:
            queue.pop()

        queue.append(i)  # Add current index to the queue

        # Add the maximum for the current window to the result
        if i >= k - 1:
            result.append(nums[queue[0]])

    return result
```

# Binary Search

```python
def binary_search_variant(arr, target):
    left, right = 0, len(arr) - 1
```

```
    result = -1  # Initialize result if needed
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] ? target:  # Replace '?' with the appropriate operator
            result = mid  # Update result if needed
            # Decide whether to move left or right based on the pattern
            right = mid - 1 or left = mid + 1
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result
```

## Binary Search on Answer: Concise Notes

**1. How to Identify**

1. **Optimization Problem**:
    - Find the **minimum** or **maximum** value satisfying a condition (e.g., minimize largest, maximize smallest).
2. **Monotonic Search Space**:
    - If a value satisfies the condition, all larger (or smaller) values also satisfy it.
3. **Helper Function**:
    - A function `can_satisfy(value)` exists to verify if a value meets the condition (usually O(n)).
4. **Common Problems**
    - **Split Array Largest Sum** (Minimize largest subarray sum).
    - **Koko Eating Bananas** (Minimize eating speed).
    - **Aggressive Cows** (Maximize minimum distance).
    - **Allocate Minimum Pages** (Minimize pages per student).
5. **Key Idea**: Binary search efficiently narrows the range of possible answers; the helper function validates feasibility at each step.

```python
def binary_search_on_answer(arr, condition_fn, low, high):
    while low <= high:
        mid = low + (high - low) // 2
        if condition_fn(mid, arr):
            high = mid - 1  # Try for a smaller/larger valid value
        else:
            low = mid + 1  # Discard invalid values
    return low
```

## Binary Search on Unsorted Arrays: Concise Notes

**1. When to Apply**

- **Input is not fully sorted**, but the problem has a **monotonic property** or a specific structure:

1. Rotated Sorted Arrays.
2. Peak Element or Mountain Arrays.
3. Virtual/Conceptual Search Spaces (e.g., infinite arrays).

- **Key Requirement**:
    - The **search space** or **conditions** allow the array to be divided into parts where binary search can narrow down the range.

**2. Key Problems**

1. **Search in Rotated Sorted Array**: Narrow down the range using rotation logic.
2. **Find Peak Element**: Use binary search to locate a peak.
3. **Find Minimum in Rotated Array**: Locate the point of rotation.

**3. Template**

- **Core Idea**: Identify the property to decide which half of the array to discard.

**4. Examples:**

**Problem Find Peak Element**: Find an element that is greater than its neighbors in an unsorted array.

```
def find_peak_element(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < nums[mid + 1]:  # Move toward the peak
            left = mid + 1
        else:  # Discard the right part
            right = mid
    return left
```

**Searching in a rotated sorted array**:

```python
def search_in_rotated_array(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = left + (right - left) // 2

        # If target is found
        if nums[mid] == target:
            return mid

        # Determine which side is sorted
        if nums[left] <= nums[mid]:  # Left half is sorted
            if nums[left] <= target < nums[mid]:
                right = mid - 1  # Target is in the left half
            else:
                left = mid + 1  # Target is in the right half
        else:  # Right half is sorted
            if nums[mid] < target <= nums[right]:
```

```
                left = mid + 1  # Target is in the right half
            else:
                right = mid - 1  # Target is in the left half

    return -1  # Target not found
```

**5. Explanation**

- **Monotonic Property**:
  - If `nums[mid] < nums[mid + 1]`, the peak must lie in the right half.
  - Else, it lies in the left half.
- **Termination**: `left == right`, pointing to the peak element.

This pattern efficiently handles problems where the **array isn't fully sorted**, but **binary search works due to conceptual or monotonic properties**.

# Heaps

## 1. Kth Largest/Smallest Element

**Pattern Overview**

- Use a **min-heap** for finding the kth largest element.
- Use a **max-heap** for finding the kth smallest element.
- The idea is to maintain a heap of size `k` that tracks the desired elements efficiently.

**Common Problems**

1. Kth Largest Element in an Array.
2. Kth Smallest Element in a Sorted Matrix.

**Key Points**

- Min-heap is used for the kth largest because the smallest element in the heap is replaced once the size exceeds `k`.
- Max-heap is used for the kth smallest by negating the values (since Python's `heapq` is a min-heap by default).

**Example Code**

```python
import heapq

# Kth Largest Element in an Array
def findKthLargest(nums, k):
    min_heap = []
    for num in nums:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)
    return min_heap[0]
```

## 2. Top K Elements

### Pattern Overview

- Use a **heap** to efficiently retrieve the top `k` elements based on custom criteria (e.g., frequency, value).
- Combine **heap operations** with frequency maps or sorted structures.

### Common Problems

1. Top K Frequent Elements.
2. Top K Frequent Words.

### Key Points

- Use a **max-heap** if k elements need to be sorted in descending order.
- Use a **min-heap** to maintain a heap of size `k`.

### Example Code

```python
# Top K Frequent Elements
from collections import Counter
import heapq

def topKFrequent(nums, k):
    freq_map = Counter(nums)
    min_heap = []

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

    return [num for freq, num in min_heap]
```

## 3. Merge K Sorted Lists/Arrays

### Pattern Overview

- Use a **min-heap** to efficiently merge k sorted arrays or lists.
- The heap is used to track the smallest element from each array, and the result is built incrementally.

### Common Problems

1. Merge K Sorted Lists.
2. Smallest Range Covering Elements from K Lists.

### Key Points

- Push the first element of each list/array into the heap with an identifier (e.g., index).
- Extract the smallest element, add it to the result, and push the next element from the same list into the heap.

### Example Code

```python
import heapq

# Merge K Sorted Lists
def mergeKLists(lists):
    min_heap = []

    # Push initial elements of each list into the heap
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(min_heap, (lst[0], i, 0))  # (value, list index, element index)

    result = []
    while min_heap:
        val, list_idx, elem_idx = heapq.heappop(min_heap)
        result.append(val)
        if elem_idx + 1 < len(lists[list_idx]):
            heapq.heappush(min_heap, (lists[list_idx][elem_idx + 1], list_idx, elem_idx + 1))

    return result
```

## 4. Two Heaps Pattern

#### Pattern Overview

- Use **two heaps** (max-heap and min-heap) to efficiently manage data in two halves.
- Useful for problems requiring **median calculation** or **balancing data partitions**.

#### Common Problems

1. Find Median from Data Stream.
2. Sliding Window Median.

#### Key Points

- Use a **max-heap** for the left half of the data and a **min-heap** for the right half.
- Maintain the size property:
    - Max-heap can have at most one more element than the min-heap.

#### Example Code

```python
import heapq

# Find Median from Data Stream
class MedianFinder:
    def __init__(self):
        self.small = []  # Max-heap for the smaller half (invert values for max-heap)
        self.large = []  # Min-heap for the larger half

    def addNum(self, num):
```

```
        heapq.heappush(self.small, -num)
        heapq.heappush(self.large, -heapq.heappop(self.small))

        if len(self.small) < len(self.large):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2.0
```

# Linked Lists

### 1. Reversal-Based Pattern

**Template Code: Reverse a Linked List (Iterative)**

```python
def reverseList(head):
    prev, curr = None, head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev, curr = curr, next_node
    return prev
```

**Common Problems**:

- Reverse a Linked List.
- Reverse Nodes in K-Groups.
- Reverse Linked List II (Partial reversal).

### 2. Delete/Add a Node (Dummy Node Simplification)

**Template Code: Delete/Add a Node in a Linked List**
*Delete/add a node when the head or a dummy node simplifies pointer handling.*

```python
def add_node(prev, val):
    new_node = ListNode(val, prev.next)
    prev.next = new_node

def delete_node(prev):
    if not prev or not prev.next:
        return None  # Nothing to delete
    to_delete = prev.next
    prev.next = to_delete.next
    to_delete.next = None
    return to_delete  # Return the deleted node
```

**Common Problems**:

- Delete Node in a Linked List (given node reference).

- Remove Nth Node from the End of List (use dummy node + 2 pointers).

### 3. Fast and Slow Pointer Pattern

**Template Code: Detect a Cycle in a Linked List**

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

**Common Problems**:

- Linked List Cycle Detection.
- Find the Middle of the Linked List.
- Detect Cycle and Return Starting Node.
- Intersection of Two Linked Lists (length adjustment with pointers).

### 4. Dummy Node for Simplification

**Template Code: Merge Two Sorted Lists**

```python
def mergeTwoLists(l1, l2):
    dummy = ListNode(0)
    curr = dummy
    while l1 and l2:
        if l1.val < l2.val:
            curr.next, l1 = l1, l1.next
        else:
            curr.next, l2 = l2, l2.next
        curr = curr.next
    curr.next = l1 or l2
    return dummy.next
```

**Common Problems**:

- Merge Two Sorted Lists.
- Add Two Numbers.
- Partition List.
- Merge K Sorted Lists (heap approach uses dummy node for simplicity).

### 5. Linked List ListNode

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

```
        self.val = val
        self.next = next
```

# Trees

**1. DFS Traversals (Recursive & Iterative)**

**Recursive DFS Template**:

```python
def dfs(node):
    if not node:
        return
    # Preorder logic (process node)
    dfs(node.left)
    # Inorder logic (process node)
    dfs(node.right)
    # Postorder logic (process node)
```

**Iterative DFS (Inorder Example)**:

```python
def dfs_iterative(root):
    stack, result = [], []
    while stack or root:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        result.append(root.val)  # Process node
        root = root.right
    return result
```

**2. BFS (Level Order Traversal)**

**Note: In BFS, you may also want vertical levels. You can maintain index structure (2*level, 2*level+1) (max width of binary tree) or level+1, level-1 (vertical order traversal) depending on type of problem**

**You can also simulate BFS using DFS. Just append a new [] to levels when you encounter a new level, i.e level == len(levels). This eliminates the need of using a queue. This trick can be used in some problems.**

**Template**:

```python
from collections import deque
def bfs(root):
    if not root:
        return []
    queue, result = deque([root]), []
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)  # Process node
            if node.left: queue.append(node.left)
```

```
            if node.right: queue.append(node.right)
        result.append(level)
    return result
```

3. **Tree Construction (Preorder + Inorder Example)**

**Template**:

```python
def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_val = preorder.pop(0)
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left = build_tree(preorder, inorder[:idx])
    root.right = build_tree(preorder, inorder[idx+1:])
    return root
```

### Binary Trees (Additional Patterns)

1. **Symmetric Tree**:

**Template**:

```python
def is_symmetric(root):
    def check(left, right):
        if not left and not right:
            return True
        if not left or not right or left.val != right.val:
            return False
        return check(left.left, right.right) and check(left.right, right.left)
    return check(root.left, root.right) if root else True
```

2. **Flatten Binary Tree to Linked List**:

**Template**:

```python
def flatten(root):
    def dfs(node):
        if not node:
            return None
        left_tail = dfs(node.left)
        right_tail = dfs(node.right)
        if left_tail:
            left_tail.right = node.right
            node.right = node.left
            node.left = None
        return right_tail or left_tail or node
    dfs(root)
```

3. **Lowest Common Ancestor (LCA)**:

**Template**:

```python
def lca(root, p, q):
    if not root or root == p or root == q:
        return root
    left = lca(root.left, p, q)
    right = lca(root.right, p, q)
    return root if left and right else left or right
```

### Binary Search Trees (BST Patterns)

1. **Validate BST**:

**Template**:

```python
def is_valid_bst(root):
    def validate(node, low, high):
        if not node:
            return True
        if not (low < node.val < high):
            return False
        return validate(node.left, low, node.val) and validate(node.right, node.val, high)
    return validate(root, float('-inf'), float('inf'))
```

2. **Search in BST**:

**Template**:

```python
def search_bst(root, val):
    if not root or root.val == val:
        return root
    return search_bst(root.left, val) if val < root.val else search_bst(root.right, val)
```

3. **Kth Smallest Element in BST**:

**Template**:

```python
def kth_smallest(root, k):
    stack = []
    while True:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        k -= 1
        if k == 0:
            return root.val
        root = root.right
```

4. **Convert Sorted Array to BST**:

**Template**:

```python
def sorted_array_to_bst(nums):
    if not nums:
        return None
    mid = len(nums) // 2
    root = TreeNode(nums[mid])
    root.left = sorted_array_to_bst(nums[:mid])
    root.right = sorted_array_to_bst(nums[mid+1:])
    return root
```

### Key Notes:

1. **DFS/BFS**: DFS is used for depth-based operations (e.g., paths, height), while BFS is ideal for level-based operations.
2. **Tree Construction**: Always rely on unique traversal pairs (e.g., Preorder + Inorder).
3. **Binary Trees**: Focus on symmetry, serialization, and manipulation (invert/flatten).
4. **BST**: Leverage sorted property for efficient search, validation, and construction.

### Concise Notes on Propagating Results in Recursion

1. **Global Best Answer**: Use a global variable to track the best value across all nodes (e.g., max path sum, tree diameter). Subtree results help compute the local value, and the global variable holds the final answer.
2. **Aggregate Subtree Results**: Combine results from left and right subtrees to compute the parent's value, which may also be the final answer (e.g., count nodes, check balance).
3. **Intermediate Results**: Propagate meaningful results upward to aid parent computations (e.g., LCA, path sums). Use special values (e.g., `-1`) to signal specific states.
4. **Global State Updates**: Use external variables to track cumulative or specific results during recursion (e.g., count of good subtrees).
5. **Key Idea**: Identify whether the task requires combining subtree results, tracking a global best, or propagating intermediate states. Optimize by only returning necessary information.

# Trie

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_word = False

class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        curr = self.root
        for ch in word:
            if ch not in curr.children:
                curr.children[ch] = TrieNode()
            curr = curr.children[ch]
```

```python
        curr.end_of_word = True

    def search(self, word: str) -> bool:
        curr = self.root
        for ch in word:
            if ch not in curr.children:
                return False
            curr = curr.children[ch]
        return curr.end_of_word == True

    def startsWith(self, prefix: str) -> bool:
        curr = self.root
        for ch in prefix:
            if ch not in curr.children:
                return False
            curr = curr.children[ch]
        return True

    def dfs(self, root, word):
            curr = root
            for i in range(len(word)):
                if word[i] != '.' and word[i] not in curr.children:
                    return False
                elif word[i]!= '.' and word[i] in curr.children:
                    curr = curr.children[word[i]]
                else:
                    for child in curr.children:
                        if self.dfs(curr.children[child], word[i+1:]):
                            return True
                    return False
            return curr.end_of_word
```