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)
        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/	

```
def is_valid(inputs):                                                    python
    #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.

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)]
```

```python
# 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 prop
ogated)
        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
```

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.

```
visited = set([start])  # Track visited nodes, starting with the source node     python
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 (cyc
le 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

    # 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 the 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)

    # 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]

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
```

```python
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
            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 cool
down
        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)
```