# 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 us
ed 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

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)

```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_with_path(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

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

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
```