# Master LeetCode Templates

## Comprehensive Playbook for Classic Patterns

Updated November 5, 2025

Generated with ❤️ using LaTeX and Pygments

# Contents

# 1 Trees

These templates cover essential tree traversals and constructions. Mix and match traversal patterns with memoization or BFS when shapes change.

## 1.1 DFS Traversals (Recursive & Iterative)

Recursion keeps the intent clear; fall back to iterative helpers when call stacks are shallow or you need explicit control. **Complexity:** Time $O(n)$, Space $O(h)$ for recursion and $O(n)$ worst-case for the explicit stack.

```python
def preorder_recursive(node):
    order = []

    def dfs(cur):
        if not cur:
            return
        order.append(cur.val)
        dfs(cur.left)
        dfs(cur.right)

    dfs(node)
    return order


def inorder_recursive(node):
    order = []

    def dfs(cur):
        if not cur:
            return
        dfs(cur.left)
        order.append(cur.val)
        dfs(cur.right)

    dfs(node)
    return order


def postorder_recursive(node):
    order = []

    def dfs(cur):
        if not cur:
            return
        dfs(cur.left)
        dfs(cur.right)
        order.append(cur.val)
```

```
    dfs(node)
    return order
```

```
def preorder_iterative(root):
    if not root:
        return []
    stack, output = [root], []
    while stack:
        node = stack.pop()
        output.append(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return output


def inorder_iterative(root):
    stack, output = [], []
    node = root
    while stack or node:
        while node:
            stack.append(node)
            node = node.left
        node = stack.pop()
        output.append(node.val)
        node = node.right
    return output


def postorder_iterative(root):
    if not root:
        return []
    stack, output = [root], []
    while stack:
        node = stack.pop()
        output.append(node.val)
        if node.left:
            stack.append(node.left)
        if node.right:
            stack.append(node.right)
    return output[::-1]
```

Morris traversal temporarily threads the tree to achieve inorder traversal in O(1) extra space.

```
def inorder_morris(root):
    order = []
    cur = root
```

```
    while cur:
        if not cur.left:
            order.append(cur.val)
            cur = cur.right
        else:
            pred = cur.left
            while pred.right and pred.right is not cur:
                pred = pred.right
            if not pred.right:
                pred.right = cur
                cur = cur.left
            else:
                pred.right = None
                order.append(cur.val)
                cur = cur.right
    return order
```

## 1.2 Path Sum with Backtracking

Use DFS with cumulative sums and path tracking. Backtrack siblings cleanly. **Complexity:** Time $O(n)$ visiting each node once, Space $O(h)$ for the recursion stack.

```
def path_sum(root, target):
    paths, cur = [], []

    def dfs(node, running):
        if not node:
            return
        cur.append(node.val)
        running += node.val
        if not node.left and not node.right and running == target:
            paths.append(cur[:])
        dfs(node.left, running)
        dfs(node.right, running)
        cur.pop()

    dfs(root, 0)
    return paths
```

## 1.3 Collect All Root-to-Leaf Paths

Capture every path and return as lists or strings. **Complexity:** Time $O(n)$ across nodes, Space $O(h)$ recursion plus output size.

```python
def binary_tree_paths(root):
    if not root:
        return []
    paths = []

    def dfs(node, trail):
        trail.append(str(node.val))
        if not node.left and not node.right:
            paths.append("->".join(trail))
        else:
            if node.left:
                dfs(node.left, trail)
            if node.right:
                dfs(node.right, trail)
        trail.pop()

    dfs(root, [])
    return paths
```

## 1.4 Lowest Common Ancestor (Binary Tree)

Return the first node that contains both targets in separate subtrees. **Complexity:** Time $O(n)$ scanning each node, Space $O(h)$ recursion depth.

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

## 1.5 Serialize and Deserialize (Binary Tree)

Use preorder traversal with null markers for compact storage. **Complexity:** Time $O(n)$ and Space $O(n)$ for recording null markers.

```python
class Codec:
    def serialize(self, root):
        values = []

        def dfs(node):
            if not node:
                values.append("#")
                return
            values.append(str(node.val))
            dfs(node.left)
            dfs(node.right)

        dfs(root)
        return " ".join(values)

    def deserialize(self, data):
        values = iter(data.split())

        def dfs():
            val = next(values)
            if val == "#":
                return None
            node = TreeNode(int(val))
            node.left = dfs()
            node.right = dfs()
            return node

        return dfs()
```

## 1.6 Level Order Traversal (BFS)

Level-order reveals breadth snapshots for BFS-based tree problems. **Complexity:** Time $O(n)$ and Space $O(n)$ for the queue in the widest level.

```python
from collections import deque


def level_order(root):
    if not root:
        return []
    queue = deque([root])
    levels = []
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        levels.append(level)
    return levels
```

# 2 🔗 Graphs

Blend DFS, BFS, and Union-Find patterns to answer reachability, component counts, and shortest paths in graph problems.

## 2.1 Graph DFS (Recursive and Iterative)

Flexible template accepts either traversal style. **Complexity:** Time $O(V + E)$, Space $O(V)$ for the recursion stack or explicit stack.

```python
def dfs_recursive(graph, start):
    seen = set()

    def dfs(node):
        if node in seen:
            return
        seen.add(node)
        for nei in graph[node]:
            dfs(nei)

    dfs(start)
    return seen


def dfs_iterative(graph, start):
    stack, seen = [start], set()
    while stack:
        node = stack.pop()
        if node in seen:
            continue
        seen.add(node)
        for nei in graph[node]:
            if nei not in seen:
                stack.append(nei)
    return seen
```

## 2.2 BFS Shortest Path

Classic breadth-first template for unweighted graphs. **Complexity:** Time $O(V + E)$, Space $O(V)$ for the queue and seen set.

```
from collections import deque


def bfs_shortest_path(graph, start, target):
    queue = deque([(start, 0)])
    seen = {start}
    while queue:
        node, dist = queue.popleft()
        if node == target:
            return dist
        for nei in graph[node]:
            if nei not in seen:
                seen.add(nei)
                queue.append((nei, dist + 1))
    return -1
```

## 2.3 Connected Components

Traverse all nodes, counting component sizes. **Complexity:** Time $O(V + E)$, Space $O(V)$ for visited bookkeeping.

```
def count_components(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    seen, components = set(), 0

    def dfs(node):
        stack = [node]
        while stack:
            cur = stack.pop()
            if cur in seen:
                continue
            seen.add(cur)
            for nei in graph[cur]:
                if nei not in seen:
                    stack.append(nei)

    for node in range(n):
        if node not in seen:
            components += 1
            dfs(node)
```

```
    return components
```

## 2.4 Path Existence (DFS/BFS)

Return a boolean by cutting search early once target appears. **Complexity:** Time $O(V+E)$, Space $O(V)$ at worst for the stack/set.

```
def exists_path(graph, start, target):
    stack, seen = [start], {start}
    while stack:
        node = stack.pop()
        if node == target:
            return True
        for nei in graph[node]:
            if nei not in seen:
                seen.add(nei)
                stack.append(nei)
    return False
```

## 2.5 Cycle Detection (Directed)

Track recursion stack to catch back edges. **Complexity:** Time $O(V+E)$, Space $O(V)$ for recursion and the path set.

```
def has_cycle_directed(graph):
    seen, path = set(), set()

    def dfs(node):
        if node in path:
            return True
        if node in seen:
            return False
        seen.add(node)
        path.add(node)
        for nei in graph[node]:
            if dfs(nei):
                return True
        path.remove(node)
        return False

    return any(dfs(node) for node in graph)
```

## 2.6 Topological Sort (Kahn's Algorithm)

BFS with indegree tracking orders DAG vertices. **Complexity:** Time $O(V+E)$, Space $O(V+E)$ for indegree bookkeeping.

```python
from collections import deque, defaultdict


def topo_sort(n, edges):
    indegree = [0] * n
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    queue = deque([node for node in range(n) if indegree[node] == 0])
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for nei in graph[node]:
            indegree[nei] -= 1
            if indegree[nei] == 0:
                queue.append(nei)
    return order if len(order) == n else []
```

## 2.7 Union-Find (Disjoint Set Union)

Union-Find with path compression and union-by-size keeps connectivity checks nearly constant. **Complexity:** Each `find`/`union` is amortized $\alpha(n)$ time, Space $O(n)$.

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n
        self.count = n

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def union(self, x, y):
        root_x, root_y = self.find(x), self.find(y)
        if root_x == root_y:
            return False
        if self.size[root_x] < self.size[root_y]:
            root_x, root_y = root_y, root_x
```

```
        self.parent[root_y] = root_x
        self.size[root_x] += self.size[root_y]
        self.count -= 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

    def component_size(self, x):
        return self.size[self.find(x)]

    def components(self):
        return self.count
```

## 2.8 Union-Find with Rollback

Keeps history on a stack so unions can be undone during divide-and-conquer offline queries. **Complexity:** Union/Find $O(\log^* n)$, Rollback $O(1)$, Space $O(n)$ plus stack.

```
class UnionFindRollback:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n
        self.changes = []

    def find(self, x):
        while self.parent[x] != x:
            x = self.parent[x]
        return x

    def union(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            self.changes.append((-1, -1, -1))
            return False
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.changes.append((y, self.parent[y], self.size[x]))
        self.parent[y] = x
        self.size[x] += self.size[y]
        return True

    def snapshot(self):
        return len(self.changes)

    def rollback(self, snap):
        while len(self.changes) > snap:
            node, parent, size_before = self.changes.pop()
            if node == -1:
```

```
                continue
            root = self.parent[node]
            self.parent[node] = parent
            self.size[root] = size_before
```

## 2.9 Dijkstra's Shortest Path

Greedy expansion with priority queue handles weighted graphs. **Complexity:** Time $O((V + E) \log V)$ with a binary heap, Space $O(V + E)$ for the graph and distance arrays.

```
import heapq
from collections import defaultdict


def dijkstra(n, edges, start):
    graph = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((v, w))

    dist = [float('inf')] * n
    dist[start] = 0
    heap = [(0, start)]
    while heap:
        cur_dist, node = heapq.heappop(heap)
        if cur_dist > dist[node]:
            continue
        for nei, weight in graph[node]:
            new_dist = cur_dist + weight
            if new_dist < dist[nei]:
                dist[nei] = new_dist
                heapq.heappush(heap, (new_dist, nei))
    return dist
```

# 3 Grids

Grid problems transform rows and columns into graph traversal with extra structure such as bounds checks.

## 3.1 DFS for Islands

Mark visited land to avoid recounting. **Complexity:** Time $O(RC)$ touching each cell once, Space $O(RC)$ in worst case for the stack or visited set.

```
def num_islands(grid):
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    seen = set()

    def dfs(r, c):
        stack = [(r, c)]
        while stack:
            x, y = stack.pop()
            if (
                x < 0 or x >= rows or
                y < 0 or y >= cols or
                grid[x][y] == '0' or
                (x, y) in seen
            ):
                continue
            seen.add((x, y))
            for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                stack.append((x + dx, y + dy))

    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1' and (r, c) not in seen:
                count += 1
                dfs(r, c)
    return count
```

## 3.2 BFS Flood Fill

Replace region colors using BFS/DFS flood. **Complexity:** Time $O(RC)$ and Space $O(RC)$ for the queue in the worst region.

```
from collections import deque


def flood_fill(image, sr, sc, color):
    rows, cols = len(image), len(image[0])
    target = image[sr][sc]
    if target == color:
        return image

    queue = deque([(sr, sc)])
    image[sr][sc] = color
    while queue:
        r, c = queue.popleft()
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and image[nr][nc] == target:
                image[nr][nc] = color
                queue.append((nr, nc))
    return image
```

## 3.3 Word Search DFS

Perform backtracking with visited set or board mutation. **Complexity:** Time $O(RC \cdot 4^L)$ in worst case, Space $O(L)$ for recursion depth.

```
def exist(board, word):
    rows, cols = len(board), len(board[0])

    def dfs(r, c, idx):
        if idx == len(word):
            return True
        if (
            r < 0 or r >= rows or
            c < 0 or c >= cols or
            board[r][c] != word[idx]
        ):
            return False
        tmp, board[r][c] = board[r][c], '#'
        found = any(
            dfs(r + dr, c + dc, idx + 1)
            for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1))
        )
        board[r][c] = tmp
        return found
```

```
    for r in range(rows):
        for c in range(cols):
            if dfs(r, c, 0):
                return True
    return False
```

## 3.4 Shortest Path in Grid

Use BFS with state storing distance. **Complexity:** Time $O(n^2)$ and Space $O(n^2)$ for the queue/visited set.

```
def shortest_path_binary_matrix(grid):
    n = len(grid)
    if grid[0][0] or grid[n - 1][n - 1]:
        return -1
    directions = [
        (1, 0), (-1, 0), (0, 1), (0, -1),
        (1, 1), (1, -1), (-1, 1), (-1, -1)
    ]
    queue = deque([(0, 0, 1)])
    seen = {(0, 0)}
    while queue:
        r, c, dist = queue.popleft()
        if (r, c) == (n - 1, n - 1):
            return dist
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < n and 0 <= nc < n and not grid[nr][nc] and (nr, nc)
            ↪  not in seen:
                seen.add((nr, nc))
                queue.append((nr, nc, dist + 1))
    return -1
```

## 3.5 Multi-Source BFS

Push all starting sources initially to expand simultaneously. **Complexity:** Time $O(RC)$, Space $O(RC)$ for the queue.

```
def oranges_rotting(grid):
    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                queue.append((r, c, 0))
            elif grid[r][c] == 1:
                fresh += 1
    minutes = 0
    while queue:
        r, c, minutes = queue.popleft()
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                grid[nr][nc] = 2
                fresh -= 1
                queue.append((nr, nc, minutes + 1))
    return minutes if fresh == 0 else -1
```

## 3.6 Surrounded Regions

Flip interior regions by capturing escape nodes first. **Complexity:** Time $O(RC)$, Space up to $O(RC)$ from the stack/queue.

```
def solve(board):
    if not board:
        return
    rows, cols = len(board), len(board[0])

    def dfs(r, c):
        stack = [(r, c)]
        while stack:
            x, y = stack.pop()
            if 0 <= x < rows and 0 <= y < cols and board[x][y] == 'O':
                board[x][y] = 'A'
                for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                    stack.append((x + dx, y + dy))

    for r in range(rows):
        if board[r][0] == 'O':
            dfs(r, 0)
        if board[r][cols - 1] == 'O':
```

```
            dfs(r, cols - 1)
    for c in range(cols):
        if board[0][c] == 'O':
            dfs(0, c)
        if board[rows - 1][c] == 'O':
            dfs(rows - 1, c)

    for r in range(rows):
        for c in range(cols):
            if board[r][c] == 'O':
                board[r][c] = 'X'
            elif board[r][c] == 'A':
                board[r][c] = 'O'
```

# 4 Backtracking

Backtracking explores decision trees while pruning invalid branches for exponential searches.

## 4.1 Subsets (Power Set)

Use DFS to include or exclude each element. **Complexity:** Time $O(2^n)$ generating all subsets, Space $O(n)$ recursion plus output size.

```
def subsets(nums):
    result, cur = [], []

    def dfs(idx):
        if idx == len(nums):
            result.append(cur[:])
            return
        cur.append(nums[idx])
        dfs(idx + 1)
        cur.pop()
        dfs(idx + 1)

    dfs(0)
    return result
```

## 4.2 Permutations

Swap in-place to generate permutations with O(n) extra space. **Complexity:** Time $O(n \cdot n!)$, Space $O(n)$ for recursion frames plus output.

```
def permute(nums):
    result = []

    def backtrack(first):
        if first == len(nums):
            result.append(nums[:])
            return
        for i in range(first, len(nums)):
            nums[first], nums[i] = nums[i], nums[first]
            backtrack(first + 1)
            nums[first], nums[i] = nums[i], nums[first]

    backtrack(0)
    return result
```

## 4.3 Combination Sum

Choose numbers allowing reuse by skipping backwards. **Complexity:** Time up to $O(2^n)$ in the branching search, Space $O(n)$ for recursion depth.

```python
def combination_sum(candidates, target):
    candidates.sort()
    combos, path = [], []

    def dfs(idx, remaining):
        if remaining == 0:
            combos.append(path[:])
            return
        for i in range(idx, len(candidates)):
            val = candidates[i]
            if val > remaining:
                break
            path.append(val)
            dfs(i, remaining - val)
            path.pop()

    dfs(0, target)
    return combos
```

## 4.4 N-Queens

Track threatened columns and diagonals with sets. **Complexity:** Time $O(n!)$ in the worst case, Space $O(n)$ for recursion and board state.

```python
def solve_n_queens(n):
    cols, diag1, diag2 = set(), set(), set()
    board = ["." * n for _ in range(n)]
    result = []

    def place(row, layout):
        if row == n:
            result.append(layout[:])
            return
        for col in range(n):
            if col in cols or (row + col) in diag1 or (row - col) in diag2:
                continue
            cols.add(col)
            diag1.add(row + col)
            diag2.add(row - col)
            new_row = board[row][:col] + "Q" + board[row][col + 1:]
            place(row + 1, layout + [new_row])
            cols.remove(col)
            diag1.remove(row + col)
            diag2.remove(row - col)
```

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

## 4.5 Sudoku Solver

Fill blanks by choosing the lowest option cell first for pruning. **Complexity:** Time is exponential in the number of blanks (worst-case near $O(9^m)$), Space $O(m)$ for recursion plus board state.

```
def solve_sudoku(board):
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]
    empties = []

    for r in range(9):
        for c in range(9):
            val = board[r][c]
            if val == '.':
                empties.append((r, c))
            else:
                rows[r].add(val)
                cols[c].add(val)
                boxes[(r // 3) * 3 + c // 3].add(val)

    digits = set(map(str, range(1, 10)))

    def backtrack(idx):
        if idx == len(empties):
            return True
        r, c = sorted(
            empties[idx:],
            key=lambda cell: len(digits - rows[cell[0]] - cols[cell[1]] -
            ↪  boxes[(cell[0] // 3) * 3 + cell[1] // 3])
        )[0]
        empties[idx], empties[empties.index((r, c))] =
        ↪  empties[empties.index((r, c))], empties[idx]
        candidates = digits - rows[r] - cols[c] - boxes[(r // 3) * 3 + c // 3]
        for val in candidates:
            board[r][c] = val
            rows[r].add(val)
            cols[c].add(val)
            boxes[(r // 3) * 3 + c // 3].add(val)
            if backtrack(idx + 1):
                return True
            board[r][c] = '.'
            rows[r].remove(val)
            cols[c].remove(val)
            boxes[(r // 3) * 3 + c // 3].remove(val)
```

```
        return False

    backtrack(0)
```

# 5 Dynamic Programming

DP converts exponential recursion into polynomial time by caching overlapping subproblems.

## 5.1 DFS with Memoization

Decorator caches states to avoid recomputation. **Complexity:** Time proportional to the number of reachable states times their outgoing transitions, Space $O(|S|)$ for memo storage.

```python
from functools import lru_cache


def dfs_with_memo(state, transitions):
    @lru_cache(maxsize=None)
    def dfs(node):
        if node.is_terminal():
            return node.value
        return max(dfs(next_state) for next_state in transitions(node))

    return dfs(state)
```

## 5.2 Unique Paths in Grid

Bottom-up DP for movement constrained to right and down. **Complexity:** Time $O(mn)$, Space $O(mn)$ (or $O(n)$ with a single rolling row).

```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for r in range(1, m):
        for c in range(1, n):
            dp[r][c] = dp[r - 1][c] + dp[r][c - 1]
    return dp[-1][-1]
```

## 5.3 0/1 Knapsack

Pick items maximizing value under capacity limit. **Complexity:** Time $O(nC)$ with $n$ items and capacity $C$, Space $O(C)$ for the 1-D DP.

```python
def knapsack(weights, values, capacity):
    dp = [0] * (capacity + 1)
    for weight, value in zip(weights, values):
        for cap in range(capacity, weight - 1, -1):
            dp[cap] = max(dp[cap], dp[cap - weight] + value)
    return dp[capacity]
```

## 5.4 Word Break

DP with prefix scanning reduces to substring membership checks. **Complexity:** Time $O(n^2)$ over substring endpoints, Space $O(n)$ for the DP array.

```python
def word_break(s, word_dict):
    words = set(word_dict)
    dp = [False] * (len(s) + 1)
    dp[0] = True
    for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in words:
                dp[i] = True
                break
    return dp[-1]
```

## 5.5 Longest Increasing Subsequence

Binary search compresses DP to O(n log n). **Complexity:** Time $O(n \log n)$ using binary search, Space $O(n)$ for the tails array.

```python
import bisect


def length_of_lis(nums):
    tails = []
    for num in nums:
        idx = bisect.bisect_left(tails, num)
        if idx == len(tails):
            tails.append(num)
        else:
            tails[idx] = num
    return len(tails)
```

## 5.6 Edit Distance (Levenshtein)

Iterative DP with substitution, insertion, deletion choices. **Complexity:** Time $O(mn)$ and Space $O(mn)$ for the DP table (reducible to $O(n)$ with rolling rows).

```python
def edit_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],
                    dp[i][j - 1],
                    dp[i - 1][j - 1]
                )
    return dp[m][n]
```

## 5.7 Partition DP (Palindrome Partitioning)

Memoize minimal cuts or partitions. **Complexity:** Time $O(n^2)$ and Space $O(n^2)$ for palindrome caching.

```python
def min_cut(s):
    n = len(s)
    dp = [0] * n
    pal = [[False] * n for _ in range(n)]
    for end in range(n):
        min_cuts = end
        for start in range(end + 1):
            if s[start] == s[end] and (end - start <= 2 or pal[start + 1][end
            ↪  - 1]):
                pal[start][end] = True
                min_cuts = 0 if start == 0 else min(min_cuts, dp[start - 1] +
                ↪  1)
        dp[end] = min_cuts
    return dp[-1]
```

## 5.8 Traveling Salesman Bitmask DP

Memoize '(mask, last)' state for minimal Hamiltonian tour cost. **Complexity:** Time $O(n^2 2^n)$, Space $O(n2^n)$.

```
from functools import lru_cache


def tsp(dist):
    n = len(dist)

    @lru_cache(maxsize=None)
    def dp(mask, last):
        if mask == (1 << n) - 1:
            return dist[last][0]
        best = float('inf')
        for nxt in range(n):
            if not mask & (1 << nxt):
                cand = dist[last][nxt] + dp(mask | (1 << nxt), nxt)
                best = min(best, cand)
        return best

    return dp(1, 0)
```

## 5.9 DP on Subsets (Mask Enumeration)

Iterate masks and submasks to build answers over all subsets. **Complexity:** Typically $O(3^n)$ for all mask/submask loops.

```
def max_team_score(skills):
    n = len(skills)
    scores = [0] * (1 << n)
    for mask in range(1, 1 << n):
        lsb = mask & -mask
        idx = (lsb.bit_length() - 1)
        prev = mask ^ lsb
        scores[mask] = scores[prev] + skills[idx]

    dp = [0] * (1 << n)
    for mask in range(1, 1 << n):
        sub = mask
        while sub:
            dp[mask] = max(dp[mask], dp[mask ^ sub] + scores[sub] ** 2)
            sub = (sub - 1) & mask
    return dp[(1 << n) - 1]
```

# 6 🌀 BFS Variants

Specialized BFS setups reduce runtime for layered expansions and symmetric searches.

## 6.1 Multi-Source BFS Template

Reuse for contagion or simultaneous diffusion problems. **Complexity:** Time $O(V + E)$, Space $O(V)$ for queue and visited set.

```python
def multi_source_bfs(starts, neighbors):
    queue = deque([(start, 0) for start in starts])
    seen = set(starts)
    depth = 0
    while queue:
        node, depth = queue.popleft()
        for nei in neighbors(node):
            if nei not in seen:
                seen.add(nei)
                queue.append((nei, depth + 1))
    return depth
```

## 6.2 Bidirectional BFS

Search from both ends to shrink branching factor. **Complexity:** Time $O(V + E)$ in worst case but typically faster due to halved depth; Space $O(V)$ for frontier sets.

```python
def bidirectional_bfs(start, target, neighbors):
    if start == target:
        return 0
    front, back = {start}, {target}
    seen = {start: 0, target: 0}
    steps = 0
    while front and back:
        steps += 1
        if len(front) > len(back):
            front, back = back, front
        next_front = set()
        for node in front:
            for nei in neighbors(node):
                if nei in back:
                    return steps
                if nei not in seen:
                    seen[nei] = steps
                    next_front.add(nei)
        front = next_front
    return -1
```

## 6.3 Level Order Traversal Skeleton

Give each level custom processing. **Complexity:** Time $O(V + E)$, Space $O(V)$ for queue and seen set.

```
def level_order_bfs(starts, neighbors):
    queue = deque(starts)
    levels = []
    seen = set(starts)
    while queue:
        level_size = len(queue)
        level_nodes = []
        for _ in range(level_size):
            node = queue.popleft()
            level_nodes.append(node)
            for nei in neighbors(node):
                if nei not in seen:
                    seen.add(nei)
                    queue.append(nei)
        levels.append(level_nodes)
    return levels
```

## 6.4 Unweighted Graph Shortest Path

Track parent pointers to reconstruct path. **Complexity:** Time $O(V+E)$, Space $O(V)$ for the queue and predecessor map.

```
def shortest_path_unweighted(graph, start, target):
    queue = deque([start])
    prev = {start: None}
    while queue:
        node = queue.popleft()
        if node == target:
            break
        for nei in graph[node]:
            if nei not in prev:
                prev[nei] = node
                queue.append(nei)
    else:
        return []

    path = []
    cur = target
    while cur is not None:
        path.append(cur)
        cur = prev[cur]
    return path[::-1]
```

# 7 🔪 Two Pointers

Two pointers shrink ranges or sweep arrays in linear time without extra space.

## 7.1 Two Sum II (Sorted)

Move pointers towards target sum. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        s = nums[left] + nums[right]
        if s == target:
            return left + 1, right + 1
        if s < target:
            left += 1
        else:
            right -= 1
    return -1, -1
```

## 7.2 Remove Duplicates In-Place

Keep slow pointer for placement of unique values. **Complexity:** Time $O(n)$, Space $O(1)$ modifying in place.

```python
def remove_duplicates(nums):
    if not nums:
        return 0
    slow = 1
    for fast in range(1, len(nums)):
        if nums[fast] != nums[fast - 1]:
            nums[slow] = nums[fast]
            slow += 1
    return slow
```

## 7.3 Partition Array by Pivot

Lomuto or Hoare style partitioning organizes elements. **Complexity:** Time $O(n)$ to scan the array once, Space $O(1)$ in-place.

```
def partition(nums, pivot):
    left, right = 0, len(nums) - 1
    while left <= right:
        while left <= right and nums[left] < pivot:
            left += 1
        while left <= right and nums[right] >= pivot:
            right -= 1
        if left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1
    return left
```

## 7.4 Trapping Rain Water

Two pointers monitor max height from both sides. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def trap(height):
    left, right = 0, len(height) - 1
    left_max = right_max = 0
    water = 0
    while left < right:
        if height[left] < height[right]:
            if height[left] >= left_max:
                left_max = height[left]
            else:
                water += left_max - height[left]
            left += 1
        else:
            if height[right] >= right_max:
                right_max = height[right]
            else:
                water += right_max - height[right]
            right -= 1
    return water
```

# 8 📐 Sliding Window

Sliding window techniques maintain dynamic ranges with O(1) updates per step.

## 8.1 Fixed Window Average

Update running sum as window slides. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def max_average_subarray(nums, k):
    window_sum = sum(nums[:k])
    max_sum = window_sum
    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)
    return max_sum / k
```

## 8.2 Variable Window (At Most K)

Expand right, contract left while condition violated. **Complexity:** Time $O(n)$, Space $O(k)$ for the frequency map.

```python
def longest_subarray_at_most_k(nums, k):
    freq = {}
    left = best = 0
    for right, num in enumerate(nums):
        freq[num] = freq.get(num, 0) + 1
        while len(freq) > k:
            freq[nums[left]] -= 1
            if freq[nums[left]] == 0:
                freq.pop(nums[left])
            left += 1
        best = max(best, right - left + 1)
    return best
```

## 8.3 Longest Substring Without Repeating Characters

Map characters to indices, shift left pointer ahead of duplicates. **Complexity:** Time $O(n)$, Space $O(\min(n, |\Sigma|))$ for the index map.

```python
def length_of_longest_substring(s):
    seen = {}
    left = best = 0
    for right, ch in enumerate(s):
        if ch in seen and seen[ch] >= left:
            left = seen[ch] + 1
        seen[ch] = right
        best = max(best, right - left + 1)
    return best
```

## 8.4 Minimum Window Substring

Maintain counts until window covers need, then shrink. **Complexity:** Time $O(n)$, Space $O(|\Sigma|)$ for frequency maps.

```python
from collections import Counter


def min_window(s, t):
    need = Counter(t)
    missing = len(t)
    left = start = end = 0
    for right, ch in enumerate(s, 1):
        if need[ch] > 0:
            missing -= 1
        need[ch] -= 1
        if missing == 0:
            while left < right and need[s[left]] < 0:
                need[s[left]] += 1
                left += 1
            if end == 0 or right - left < end - start:
                start, end = left, right
            need[s[left]] += 1
            missing += 1
            left += 1
    return s[start:end]
```

## 8.5 Sliding Window Maximum (Deque)

Keep deque decreasing to track max quickly. **Complexity:** Time $O(n)$, Space $O(k)$ for the deque.

```
from collections import deque


def max_sliding_window(nums, k):
    dq, result = deque(), []
    for i, num in enumerate(nums):
        while dq and dq[0] <= i - k:
            dq.popleft()
        while dq and nums[dq[-1]] <= num:
            dq.pop()
        dq.append(i)
        if i >= k - 1:
            result.append(nums[dq[0]])
    return result
```

# 9 Arrays

Array patterns rely on prefix sums, interval merges, and rotation tricks.

## 9.1 Prefix Sum

Prefix sums answer range queries quickly for arrays and grids once the cumulative table is built. **Complexity:** Build in $O(n)$ (1D) or $O(mn)$ (2D); each query is $O(1)$.

```python
class PrefixSum:
    def __init__(self, nums):
        self.prefix = [0]
        for num in nums:
            self.prefix.append(self.prefix[-1] + num)

    def range_sum(self, left, right):
        return self.prefix[right + 1] - self.prefix[left]

    def append(self, val):
        self.prefix.append(self.prefix[-1] + val)


class PrefixSum2D:
    def __init__(self, grid):
        if not grid or not grid[0]:
            raise ValueError("grid must be non-empty")
        rows, cols = len(grid), len(grid[0])
        self.prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
        for r in range(rows):
            for c in range(cols):
                self.prefix[r + 1][c + 1] = (
                    grid[r][c]
                    + self.prefix[r][c + 1]
                    + self.prefix[r + 1][c]
                    - self.prefix[r][c]
                )

    def query(self, r1, c1, r2, c2):
        r1, c1, r2, c2 = r1 + 1, c1 + 1, r2 + 1, c2 + 1
        return (
            self.prefix[r2][c2]
            - self.prefix[r1 - 1][c2]
            - self.prefix[r2][c1 - 1]
            + self.prefix[r1 - 1][c1 - 1]
        )
```

## 9.2 Kadane's Maximum Subarray

Track best subarray ending at each position. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def max_sub_array(nums):
    best = cur = nums[0]
    for num in nums[1:]:
        cur = max(num, cur + num)
        best = max(best, cur)
    return best
```

## 9.3 Merge Intervals

Sort, then sweep merging overlapping ranges. **Complexity:** Time $O(n \log n)$ from sorting, Space $O(1)$ additional (excluding output).

```python
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []
    for start, end in intervals:
        if not merged or merged[-1][1] < start:
            merged.append([start, end])
        else:
            merged[-1][1] = max(merged[-1][1], end)
    return merged
```

## 9.4 Rotate Array

Rotate via reverse trick, juggling, or extra array. **Complexity:** Time $O(n)$, Space $O(1)$ in-place.

```python
def rotate(nums, k):
    k %= len(nums)
    if k == 0:
        return

    def reverse(left, right):
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

    reverse(0, len(nums) - 1)
    reverse(0, k - 1)
    reverse(k, len(nums) - 1)
```

## 9.5 Quickselect

Select the k-th element in expected linear time via randomized partition. **Complexity:** Average Time $O(n)$, worst-case $O(n^2)$; Space $O(1)$.

```
import random


def quickselect(nums, k):
    k_index = k

    def partition(left, right, pivot_index):
        pivot_value = nums[pivot_index]
        nums[pivot_index], nums[right] = nums[right], nums[pivot_index]
        store = left
        for i in range(left, right):
            if nums[i] < pivot_value:
                nums[store], nums[i] = nums[i], nums[store]
                store += 1
        nums[right], nums[store] = nums[store], nums[right]
        return store

    left, right = 0, len(nums) - 1
    while True:
        pivot_index = random.randint(left, right)
        pivot_index = partition(left, right, pivot_index)
        if pivot_index == k_index:
            return nums[pivot_index]
        if k_index < pivot_index:
            right = pivot_index - 1
        else:
            left = pivot_index + 1
```

# 10 Intervals & Scheduling

Intervals combine greedy sweeps, heaps, and merge patterns for calendar-style problems.

## 10.1 Meeting Rooms (Overlap Check)

Sort by start times and ensure no interval starts before the previous one ends. **Complexity:** Time $O(n \log n)$ for sorting, Space $O(1)$ beyond input.

```python
def can_attend_meetings(intervals):
    if not intervals:
        return True
    intervals.sort(key=lambda x: x[0])
    for i in range(1, len(intervals)):
        if intervals[i][0] < intervals[i - 1][1]:
            return False
    return True
```

## 10.2 Meeting Rooms II (Minimum Rooms)

Two-pointer sweep counts overlap depth to find the maximum concurrent meetings. **Complexity:** Time $O(n \log n)$ for sorting, Space $O(n)$ for start/end arrays.

```python
def min_meeting_rooms(intervals):
    if not intervals:
        return 0
    starts = sorted(start for start, _ in intervals)
    ends = sorted(end for _, end in intervals)
    rooms = end_ptr = 0
    for start in starts:
        if start < ends[end_ptr]:
            rooms += 1
        else:
            end_ptr += 1
    return rooms
```

## 10.3 Insert Interval

Merge overlapping segments while splicing the new interval into place. **Complexity:** Time $O(n)$, Space $O(n)$ for the result list.

```
def insert_interval(intervals, new_interval):
    merged = []
    start, end = new_interval
    i = 0
    n = len(intervals)
    while i < n and intervals[i][1] < start:
        merged.append(intervals[i])
        i += 1
    while i < n and intervals[i][0] <= end:
        start = min(start, intervals[i][0])
        end = max(end, intervals[i][1])
        i += 1
    merged.append([start, end])
    merged.extend(intervals[i:])
    return merged
```

## 10.4 Interval Intersection

Advance the pointer with the earlier finishing interval to gather overlaps. **Complexity:** Time $O(m + n)$, Space $O(1)$ aside from output.

```
def interval_intersection(a, b):
    i = j = 0
    result = []
    while i < len(a) and j < len(b):
        start = max(a[i][0], b[j][0])
        end = min(a[i][1], b[j][1])
        if start <= end:
            result.append([start, end])
        if a[i][1] < b[j][1]:
            i += 1
        else:
            j += 1
    return result
```

## 10.5 Sweep Line Maximum Overlap

Process event endpoints to track concurrent intervals or resource usage. **Complexity:** Time $O(n \log n)$ for sorting events, Space $O(n)$ for event list.

```
def max_overlap(intervals):
    events = []
    for start, end in intervals:
        events.append((start, 1))
        events.append((end, -1))
    events.sort(key=lambda x: (x[0], x[1]))
    active = best = 0
    for _, delta in events:
        active += delta
        best = max(best, active)
    return best
```

# 11 Tries & Prefix Structures

Prefix trees store shared prefixes efficiently for dictionary and autocomplete problems.

## 11.1 Trie Insert and Search

Store lowercase words and support prefix queries. **Complexity:** Insert/Search Time $O(L)$, Space $O(26 \cdot N)$ where $L$ is word length.

```
class TrieNode:
    __slots__ = ("children", "word")

    def __init__(self):
        self.children = {}
        self.word = False


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

    def insert(self, word):
        node = self.root
        for ch in word:
            node = node.children.setdefault(ch, TrieNode())
        node.word = True

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.word

    def starts_with(self, prefix):
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return True
```

## 11.2 Word Dictionary with Wildcard

DFS on trie nodes handles '.' wildcard matches. **Complexity:** Average Time $O(L)$, worst-case $O(26^L)$ for many wildcards; Space $O(26 \cdot N)$.

```python
class WordDictionary:
    def __init__(self):
        self.root = TrieNode()

    def add_word(self, word):
        node = self.root
        for ch in word:
            node = node.children.setdefault(ch, TrieNode())
        node.word = True

    def search(self, word):
        def dfs(node, idx):
            if idx == len(word):
                return node.word
            ch = word[idx]
            if ch == '.':
                return any(dfs(child, idx + 1) for child in
                ↪  node.children.values())
            if ch not in node.children:
                return False
            return dfs(node.children[ch], idx + 1)

        return dfs(self.root, 0)
```

## 11.3 Replace Words

Traverse the trie to substitute words with their shortest root prefix. **Complexity:** Building trie $O(N \cdot L)$, sentence replacement $O(M \cdot L)$.

```
def replace_words(dictionary, sentence):
    trie = Trie()
    for root in dictionary:
        trie.insert(root)

    def replace(word):
        node = trie.root
        path = []
        for ch in word:
            if ch not in node.children:
                return word
            node = node.children[ch]
            path.append(ch)
            if node.word:
                return ''.join(path)
        return word

    return ' '.join(replace(word) for word in sentence.split())
```

# 12 Range Query Trees

Fenwick and segment trees enable fast range updates and queries beyond prefix sums.

## 12.1 Fenwick Tree (Binary Indexed Tree)

Supports prefix sums and point updates in logarithmic time. **Complexity:** Update/Query $O(\log n)$, Space $O(n)$.

```
class FenwickTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.bit = [0] * (self.n + 1)
        for idx, val in enumerate(nums, 1):
            self._add(idx, val)

    def _add(self, index, delta):
        while index <= self.n:
            self.bit[index] += delta
            index += index & -index

    def update(self, index, delta):
        self._add(index + 1, delta)

    def prefix_sum(self, index):
        index += 1
        total = 0
        while index > 0:
            total += self.bit[index]
            index -= index & -index
        return total

    def range_sum(self, left, right):
        return self.prefix_sum(right) - (self.prefix_sum(left - 1) if left
        ↪  else 0)
```

## 12.2 Segment Tree (Range Sum)

Iterative tree handles range queries and point updates generically. **Complexity:** Update/Query $O(\log n)$, Space $O(n)$.

```
class SegmentTree:
    def __init__(self, nums):
        self.n = len(nums)
        self.size = 1
        while self.size < self.n:
            self.size <<= 1
        self.tree = [0] * (2 * self.size)
        for i, val in enumerate(nums):
            self.tree[self.size + i] = val
        for i in range(self.size - 1, 0, -1):
            self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    def update(self, index, value):
        pos = self.size + index
        self.tree[pos] = value
        pos //= 2
        while pos:
            self.tree[pos] = self.tree[2 * pos] + self.tree[2 * pos + 1]
            pos //= 2

    def query(self, left, right):
        left += self.size
        right += self.size
        res = 0
        while left <= right:
            if left % 2 == 1:
                res += self.tree[left]
                left += 1
            if right % 2 == 0:
                res += self.tree[right]
                right -= 1
            left //= 2
            right //= 2
        return res
```

## 12.3 Difference Array

Range increment updates convert to prefix sums when materializing results. **Complexity:** Update $O(1)$, Build output $O(n)$.

```
class DifferenceArray:
    def __init__(self, nums):
        self.diff = [0] * (len(nums) + 1)
        prev = 0
        for i, num in enumerate(nums):
            self.diff[i] = num - prev
            prev = num

    def range_add(self, left, right, delta):
        self.diff[left] += delta
        if right + 1 < len(self.diff):
            self.diff[right + 1] -= delta

    def materialize(self):
        result = []
        running = 0
        for delta in self.diff[:-1]:
            running += delta
            result.append(running)
        return result
```

# 13 String Algorithms

Classic linear-time string algorithms speed up pattern search and palindrome problems.

## 13.1 KMP Prefix Function

Build longest proper prefix/suffix table to skip mismatches. **Complexity:** Time $O(n + m)$, Space $O(m)$ for the failure table.

```
def kmp_search(text, pattern):
    if not pattern:
        return 0
    lps = [0] * len(pattern)
    length = 0
    for i in range(1, len(pattern)):
        while length and pattern[i] != pattern[length]:
            length = lps[length - 1]
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length

    j = 0
    for i, ch in enumerate(text):
        while j and ch != pattern[j]:
            j = lps[j - 1]
        if ch == pattern[j]:
            j += 1
            if j == len(pattern):
                return i - j + 1
    return -1
```

## 13.2 Z Algorithm

Compute longest prefix match at each position for multiple pattern uses. **Complexity:** Time $O(n)$, Space $O(n)$ for the Z array.

```
def z_function(s):
    n = len(s)
    z = [0] * n
    l = r = 0
    for i in range(1, n):
        if i <= r:
            z[i] = min(r - i + 1, z[i - l])
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
        if i + z[i] - 1 > r:
            l, r = i, i + z[i] - 1
    return z
```

## 13.3 Rabin-Karp Rolling Hash

Polynomial rolling hash finds substring matches with expected linear time. **Complexity:** Average Time $O(n + m)$, Space $O(1)$.

```
def rabin_karp(text, pattern, base=257, mod=10**9 + 7):
    m = len(pattern)
    if m == 0:
        return 0
    high = pow(base, m - 1, mod)
    hash_pat = 0
    hash_win = 0
    for ch_p, ch_t in zip(pattern, text):
        hash_pat = (hash_pat * base + ord(ch_p)) % mod
        hash_win = (hash_win * base + ord(ch_t)) % mod
    for i in range(len(text) - m + 1):
        if hash_pat == hash_win and text[i:i + m] == pattern:
            return i
        if i + m < len(text):
            hash_win = (
                (hash_win - ord(text[i]) * high) * base + ord(text[i + m])
            ) % mod
    return -1
```

## 13.4 Manacher's Algorithm

Transforms string to compute all odd/even palindromic radii in linear time. **Complexity:** Time $O(n)$, Space $O(n)$.

```
def longest_palindrome(s):
    transformed = "#" + "#".join(s) + "#"
    n = len(transformed)
    radius = [0] * n
    center = right = best_len = best_center = 0
    for i in range(n):
        mirror = 2 * center - i
        if i < right:
            radius[i] = min(right - i, radius[mirror])
        while (
            i + radius[i] + 1 < n
            and i - radius[i] - 1 >= 0
            and transformed[i + radius[i] + 1] == transformed[i - radius[i] -
            ↪  1]
        ):
            radius[i] += 1
        if i + radius[i] > right:
            center, right = i, i + radius[i]
        if radius[i] > best_len:
            best_len, best_center = radius[i], i
    start = (best_center - best_len) // 2
    return s[start:start + best_len]
```

# 14 🪝 Linked List

Linked list templates revolve around pointer juggling while keeping dummy heads handy.

## 14.1 Reverse Linked List (Iterative)

Reverse pointers in-place. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def reverse_list(head):
    prev = None
    cur = head
    while cur:
        nxt = cur.next
        cur.next = prev
        prev = cur
        cur = nxt
    return prev
```

## 14.2 Reverse Linked List (Recursive)

Unwind recursion to rewire nodes. **Complexity:** Time $O(n)$, Space $O(n)$ from recursion depth.

```
def reverse_list_recursive(head):
    if not head or not head.next:
        return head
    new_head = reverse_list_recursive(head.next)
    head.next.next = head
    head.next = None
    return new_head
```

## 14.3 Merge Two Sorted Lists

Use dummy node and tail pointer. **Complexity:** Time $O(m+n)$, Space $O(1)$ beyond reusing list nodes.

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

## 14.4 Detect Cycle (Floyd)

Fast and slow pointers detect loop and meeting point. **Complexity:** Time $O(n)$, Space $O(1)$.

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

## 14.5 Find Middle Node

Advance fast twice speed of slow. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

## 14.6 Remove Nth from End

Use two-pointer gap to delete target. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    slow = fast = dummy
    for _ in range(n):
        fast = fast.next
    while fast.next:
        slow = slow.next
        fast = fast.next
    slow.next = slow.next.next
    return dummy.next
```

## 14.7 Add Two Numbers

Simulate long addition while traversing. **Complexity:** Time $O(m + n)$, Space $O(1)$ beyond the output list.

```python
def add_two_numbers(l1, l2):
    dummy = tail = ListNode()
    carry = 0
    while l1 or l2 or carry:
        val = carry
        if l1:
            val += l1.val
            l1 = l1.next
        if l2:
            val += l2.val
            l2 = l2.next
        carry, digit = divmod(val, 10)
        tail.next = ListNode(digit)
        tail = tail.next
    return dummy.next
```

# 15 Stacks & Queues

Stack and queue templates cover monotonic structures and bracket validation.

## 15.1 Monotonic Stack

Maintain decreasing stack for next greater element. **Complexity:** Time $O(n)$, Space $O(n)$ for the stack.

```python
def next_greater(nums):
    stack, result = [], [-1] * len(nums)
    for i, num in enumerate(nums):
        while stack and nums[stack[-1]] < num:
            idx = stack.pop()
            result[idx] = num
        stack.append(i)
    return result
```

## 15.2 Min Stack

Track running minimum alongside values. **Complexity:** Each operation is $O(1)$ time and space.

```python
class MinStack:
    def __init__(self):
        self.stack = []

    def push(self, x):
        cur_min = x if not self.stack else min(x, self.stack[-1][1])
        self.stack.append((x, cur_min))

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

    def top(self):
        return self.stack[-1][0]

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

## 15.3 Valid Parentheses

Push matching braces and check final stack. **Complexity:** Time $O(n)$, Space $O(n)$ in worst case.

```
def is_valid(s):
    pairs = {')': '(', ']': '[', '}': '{'}
    stack = []
    for ch in s:
        if ch in pairs:
            if not stack or stack.pop() != pairs[ch]:
                return False
        else:
            stack.append(ch)
    return not stack
```

## 15.4 Deque Tricks

Use deque for push-pop from both ends. **Complexity:** Time $O(n)$, Space $O(k)$ for the sliding window.

```
from collections import deque


def moving_average(nums, k):
    dq = deque()
    window_sum, result = 0, []
    for i, num in enumerate(nums):
        dq.append(num)
        window_sum += num
        if len(dq) > k:
            window_sum -= dq.popleft()
        if len(dq) == k:
            result.append(window_sum / k)
    return result
```

# 16 Heaps

Heaps surface extreme values efficiently for priority scheduling problems.

## 16.1 Kth Largest Element

Maintain min-heap of size k. **Complexity:** Time $O(n \log k)$, Space $O(k)$.

```
def kth_largest(nums, k):
    heap = nums[:k]
    heapq.heapify(heap)
    for num in nums[k:]:
        if num > heap[0]:
            heapq.heapreplace(heap, num)
    return heap[0]
```

## 16.2 Merge K Sorted Lists

Push heads of lists into min-heap by value. **Complexity:** Time $O(N \log k)$ for $N$ total nodes, Space $O(k)$ for the heap plus output list.

```
def merge_k_lists(lists):
    heap = []
    for idx, node in enumerate(lists):
        if node:
            heapq.heappush(heap, (node.val, idx, node))
    dummy = tail = ListNode()
    while heap:
        val, idx, node = heapq.heappop(heap)
        tail.next = node
        tail = node
        if node.next:
            heapq.heappush(heap, (node.next.val, idx, node.next))
    return dummy.next
```

## 16.3 Top-K Frequent Elements

Count with hash map then heapify. **Complexity:** Time $O(n + k \log n)$ with $n$ unique items, Space $O(n)$ for the frequency map and heap.

```python
from collections import Counter


def top_k_frequent(nums, k):
    freq = Counter(nums)
    heap = [(-count, num) for num, count in freq.items()]
    heapq.heapify(heap)
    return [heapq.heappop(heap)[1] for _ in range(k)]
```

## 16.4 Median of Data Stream

Use two heaps to maintain balance. **Complexity:** Each insertion runs in $O(\log n)$, retrieving the median is $O(1)$; Space $O(n)$.

```python
class MedianFinder:
    def __init__(self):
        self.small = []  # max-heap via negatives
        self.large = []  # min-heap

    def add_num(self, num):
        if not self.small or num <= -self.small[0]:
            heapq.heappush(self.small, -num)
        else:
            heapq.heappush(self.large, num)
        if len(self.small) > len(self.large) + 1:
            heapq.heappush(self.large, -heapq.heappop(self.small))
        elif len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def find_median(self):
        if len(self.small) > len(self.large):
            return float(-self.small[0])
        return (-self.small[0] + self.large[0]) / 2
```

# 17 🔍 Binary Search

Binary search splits space decisively across sorted arrays or monotonic answers.

## 17.1 Standard Binary Search

Return index or -1 when target missing. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def binary_search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            return mid
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

## 17.2 Lower and Upper Bound

Find first >= target and first > target. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def lower_bound(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left


def upper_bound(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] <= target:
            left = mid + 1
        else:
            right = mid
    return left
```

## 17.3 Search Insert Position

Return insertion index for target. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def search_insert(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left
```

## 17.4 Search Rotated Sorted Array

Determine which side is sorted before recursing. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

## 17.5 Peak Element

Binary search on gradients. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def find_peak(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid
    return left
```

## 17.6 Binary Search on Answer

Search minimal feasible value via feasibility checker. **Complexity:** Time $O(n \log R)$ where $R$ is the search range, Space $O(1)$.

```python
def min_capacity(weights, days):
    left, right = max(weights), sum(weights)

    def can_ship(capacity):
        used_days = 1
        cur = 0
        for weight in weights:
            if cur + weight > capacity:
                used_days += 1
                cur = 0
            cur += weight
        return used_days <= days

    while left < right:
        mid = (left + right) // 2
        if can_ship(mid):
            right = mid
        else:
            left = mid + 1
    return left
```

## 17.7 Matrix Search

Treat matrix as flattened sorted array. **Complexity:** Time $O(\log(mn))$, Space $O(1)$.

```python
def search_matrix(matrix, target):
    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1
    while left <= right:
        mid = (left + right) // 2
        r, c = divmod(mid, cols)
        if matrix[r][c] == target:
            return True
        if matrix[r][c] < target:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

# 18 Bit Tricks & Prefix XOR

Bitwise helpers optimize parity checks, prefix XOR queries, and bit DP transitions.

## 18.1 Prefix XOR

Maintain running XOR to answer range XOR queries in $O(1)$. **Complexity:** Preprocess $O(n)$, Query $O(1)$.

```
class PrefixXOR:
    def __init__(self, nums):
        self.prefix = [0]
        for num in nums:
            self.prefix.append(self.prefix[-1] ^ num)

    def range_xor(self, left, right):
        return self.prefix[right + 1] ^ self.prefix[left]
```

## 18.2 Count Bits DP

Use last set bit and offset to compute population counts. **Complexity:** Time $O(n)$, Space $O(n)$.

```
def count_bits(n):
    bits = [0] * (n + 1)
    for i in range(1, n + 1):
        bits[i] = bits[i >> 1] + (i & 1)
    return bits
```

## 18.3 Single Number Pair

Bit partition finds two unique numbers among duplicates. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def single_number_pair(nums):
    xor_all = 0
    for num in nums:
        xor_all ^= num
    diff = xor_all & -xor_all
    a = b = 0
    for num in nums:
        if num & diff:
            a ^= num
        else:
            b ^= num
    return a, b
```

# 19 Miscellaneous

Grab bag of handy utilities for hash maps, greedy, and randomized tasks.

## 19.1 HashMap / Counter Patterns

Use Counter to frequency map quickly. **Complexity:** Typical counting runs in $O(n)$ time with $O(n)$ space for the map (plus per-item key work).

```
from collections import Counter, defaultdict


def group_anagrams(words):
    buckets = defaultdict(list)
    for word in words:
        signature = tuple(sorted(word))
        buckets[signature].append(word)
    return list(buckets.values())


def find_duplicates(nums):
    counts = Counter(nums)
    return [num for num, freq in counts.items() if freq > 1]
```

## 19.2 Custom Sort

Sort with comparator by key tuple or decorated values. **Complexity:** Time $O(n \log n)$ comparisons (each comparator may cost key-length work), Space $O(1)$ beyond output.

```
def smallest_number(nums):
    from functools import cmp_to_key

    def compare(a, b):
        if a + b < b + a:
            return -1
        if a + b > b + a:
            return 1
        return 0

    arr = sorted(map(str, nums), key=cmp_to_key(compare))
    return str(int("".join(arr)))
```

## 19.3 Greedy Interval Scheduling

Select tasks by earliest finishing time. **Complexity:** Time $O(n \log n)$ for sorting, Space $O(1)$ additional.

```
def erase_overlap(intervals):
    intervals.sort(key=lambda x: x[1])
    count = 0
    end = float('-inf')
    for start, finish in intervals:
        if start >= end:
            end = finish
        else:
            count += 1
    return count
```

## 19.4 Union-Find for Kruskal

Collect minimum spanning tree edges by sorting weights. **Complexity:** Time $O(E \log E)$ from sorting plus near-constant Union-Find operations, Space $O(V)$.

```
def kruskal(n, edges):
    uf = UnionFind(n)
    mst = []
    cost = 0
    for weight, u, v in sorted(edges):
        if uf.union(u, v):
            mst.append((u, v, weight))
            cost += weight
    return cost, mst
```

## 19.5 Reservoir Sampling

Uniform random selection from stream. **Complexity:** Time $O(n)$ for $n$ stream elements, Space $O(1)$.

```
import random


def reservoir_sample(stream):
    sample = None
    for i, value in enumerate(stream, 1):
        if random.randrange(i) == 0:
            sample = value
    return sample
```