ChatGPT

Algorithmic Techniques Study Guide

Sliding Window

Sliding Window is an algorithmic technique for efficiently processing subarrays or substrings by maintaining a "window" of elements and sliding it over the input. Instead of using nested loops to check every possible subarray, the window is moved one step at a time, reusing computations from the previous position. It's useful when we need to find a subarray/substring that meets certain conditions (fixed length or variable length constraints) in O(n) time.

- **Use Cases:** Finding a contiguous subarray with a target sum or maximum sum (e.g., max sum of size k subarray), longest substring with unique or at most k distinct characters, minimum length subarray that meets a condition.
- **Time Complexity:** Typically O(n) because each element enters and exits the window at most once.
- **Space Complexity:** O(1) auxiliary (some problems may require extra O(n) for tracking counts, but usually constant overhead).

```
# Example: Find a subarray with target sum using sliding window (assume non-
negative numbers)
window_sum = 0
left = 0
for right in range(len(arr)):
    window_sum += arr[right]
    # Shrink the window from the left if sum exceeds target
    while window_sum > target:
        window_sum -= arr[left]
        left += 1
    if window_sum == target:
        print("Target sum found in window:", left, "to", right)
        break
```

Two Pointers

The Two Pointers technique uses two indices to traverse a data structure (usually an array or list) in a coordinated way. The pointers can move towards each other (for example, one from the start and one from the end) or in the same direction with different speeds. This method eliminates the need for some nested loops by systematically exploring pairs of elements or partitions in a single pass.

- **Use Cases:** Finding pairs or triplets in a sorted array that meet a condition (like two-sum problems), partitioning arrays (as in quicksort or Dutch national flag problem), comparing two sequences, fast/slow pointers for cycle detection in linked lists.
- **Time Complexity:** Often O(n) for linear scans (or O(n log n) if sorting is required beforehand).

- **Space Complexity:** O(1) auxiliary.

```
# Example: Two-pointer approach for two-sum in a sorted array
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, right  # found a pair
        elif s < target:
            left += 1          # move right to increase sum
        else:
            right -= 1         # move left to decrease sum
    return None
```

Binary Search

Binary Search is a divide-and-conquer algorithm that finds an element's position (or checks for existence) in a sorted array in O(log n) time. It repeatedly divides the search interval in half by comparing the target value to the middle of the interval. There are two common patterns: the standard binary search for exact values and the "binary search on answer" technique for finding an optimal value by checking a monotonic condition.

- **Use Cases:** Locating an element in a sorted array or monotonic function, finding boundaries (e.g., first occurrence, lower/upper bound), or solving problems by binary searching the answer (like minimum feasible value in scheduling, allocation, or other optimization problems).
- **Time Complexity:** O(log n) per search on a sorted array. "Binary search the answer" has O(log R) iterations, where R is the range of possible answers, combined with the complexity of the check per iteration.
- **Space Complexity:** O(1) auxiliary.

Standard Binary Search (on sorted array):

```
def binary_search(arr, target):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == target:
            return mid       # found target index
        elif arr[mid] < target:
            lo = mid + 1     # target in right half
        else:
            hi = mid - 1     # target in left half
    return -1                # target not found
```

Answer-Based Binary Search (binary search on solution space):

```
# Example: find smallest x in [lo, hi] such that condition(x) is True
lo, hi = 1, 1000  # example range
result = hi
while lo <= hi:
    mid = (lo + hi) // 2
    if condition(mid):        # if mid satisfies the problem's condition
        result = mid
        hi = mid - 1          # try to find an even smaller solution
    else:
        lo = mid + 1          # mid is too low, go higher
print("Optimal value:", result)
```

Depth-First Search (DFS)

Depth-First Search (DFS) is a graph/tree traversal algorithm that explores as far down one path as possible before backtracking. It can be implemented recursively or iteratively using a stack. DFS is useful for exploring connectivity, generating combinations (backtracking), and performing tasks like detecting cycles or topological sorting in graphs.

- **Use Cases:** Traversing all nodes in a graph or tree, solving maze and puzzle paths, topological sort (via DFS-based algorithm), checking connectivity or cycles in graphs, backtracking algorithms.
- **Time Complexity:** O(V + E) for graphs (visiting each vertex and edge once), or O(n) for a tree with n nodes.
- **Space Complexity:** O(V) in worst case for recursion stack or explicit stack (proportional to depth of graph or tree).

Recursive DFS (graph traversal):

```
def dfs(node, visited):
    if node in visited:
        return
    visited.add(node)
    # process node here (e.g., print or collect value)
    for neighbor in graph[node]:
        dfs(neighbor, visited)
```

Iterative DFS (using a stack):

```
def dfs_iterative(start):
    stack = [start]
    visited = set()
```

```
    while stack:
        node = stack.pop()
        if node in visited:
            continue
        visited.add(node)
        # process node here
        for neighbor in graph[node]:
            stack.append(neighbor)
```

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph/tree traversal that explores neighbors level by level. Starting from a source node, it visits all nodes at distance 1 before moving to distance 2, and so on. BFS uses a queue to maintain the frontier of exploration. It's particularly useful for finding the shortest path in an unweighted graph and for level-order traversal of trees.

- **Use Cases:** Shortest path in unweighted graphs, level-order traversal of binary trees, finding connected components by repeated BFS, solving puzzles where the shortest sequence of moves is needed.
- **Time Complexity:** O(V + E) for graphs, or O(n) for traversing n nodes in a tree.
- **Space Complexity:** O(V) in worst case (the queue can hold up to O(V) nodes in the worst scenario).

```
from collections import deque

def bfs(start):
    queue = deque([start])
    visited = {start}
    while queue:
        node = queue.popleft()
        # process node (e.g., print or collect)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Topological Sort (Kahn's Algorithm)

Topological Sort orders the vertices of a directed acyclic graph (DAG) such that every directed edge (u -> v) goes from earlier to later in the order. Kahn's algorithm is a BFS-based approach that repeatedly removes nodes with no incoming edges (indegree 0) and appends them to the order, updating indegrees of neighbors. This produces a valid ordering for tasks or prerequisites when a solution exists.

- **Use Cases:** Scheduling tasks with dependencies, ordering courses given prerequisites, any scenario that requires a linear ordering of a DAG.
- **Time Complexity:** O(V + E) as each node and edge is processed once.

- **Space Complexity:** O(V) for storing indegrees and using a queue.

```
from collections import deque

def topo_sort_kahn(graph):
    indegree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    queue = deque([u for u in graph if indegree[u] == 0])
    topo_order = []
    while queue:
        u = queue.popleft()
        topo_order.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    return topo_order
```

Union-Find (Disjoint Set Union)

Union-Find is a data structure that keeps track of a set of elements partitioned into disjoint subsets and supports efficient union and find operations. It is often used to solve connectivity problems by uniting sets and checking if two elements are in the same set. Path compression and union by rank (or size) optimize the operations to almost constant time.

- **Use Cases:** Checking connectivity in networks, detecting cycles in undirected graphs, Kruskal's algorithm for Minimum Spanning Tree, dynamic connectivity problems.
- **Time Complexity:** Nearly O(1) (amortized) for union and find operations with optimizations.
- **Space Complexity:** O(n) to store parent and rank arrays.

```
# Union-Find with path compression and union by rank
parent = list(range(n))
rank = [0] * n

def find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])  # path compression
    return parent[x]

def union(x, y):
    rootX = find(x)
    rootY = find(y)
    if rootX == rootY:
        return
```

```
    # union by rank
    if rank[rootX] < rank[rootY]:
        parent[rootX] = rootY
    elif rank[rootX] > rank[rootY]:
        parent[rootY] = rootX
    else:
        parent[rootY] = rootX
        rank[rootX] += 1
```

Dynamic Programming (DP)

Dynamic Programming is an optimization technique that transforms expensive brute-force recursion into a more efficient form by storing intermediate results. It applies to problems with overlapping subproblems and optimal substructure. There are two approaches: **top-down** (memoization) which caches results of recursive calls, and **bottom-up** which builds solutions iteratively from base cases.

- **Use Cases:** Fibonacci and other sequence computations, knapSack and coin change problems, shortest paths in DAG, edit distance, DP on grids (paths counting, etc.), many combinatorial optimization problems.
- **Time Complexity:** Typically reduces exponential brute-force solutions to polynomial time (exact complexity depends on number of subproblems * cost per subproblem).
- **Space Complexity:** O(number of states) for the memo or DP table (plus recursion stack for top-down).

Top-Down DP (with memoization) example – Fibonacci:

```
from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Bottom-Up DP example – Fibonacci:

```
def fib_bottom_up(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]
```

Backtracking

Backtracking is a general algorithmic technique that builds solutions incrementally and abandons a path (backtracks) as soon as it determines the path cannot lead to a valid solution. It is effectively a DFS on the solution space, and is used to generate all solutions or find a solution under constraints by exploring possibilities and pruning those that fail constraints.

- **Use Cases:** Generating all permutations or combinations of a set, solving puzzles like Sudoku or N-Queens, subset sum and partition problems, backtracking in mazes or game move sequences.
- **Time Complexity:** Generally exponential in the worst case (since it may explore all possible configurations).
- **Space Complexity:** O(d) for recursion depth d (plus space for storing the current solution state).

```
# Example: Generate all subsets of list `nums` using backtracking
result = []
def backtrack(start, current):
    result.append(current.copy())            # record current subset
    for i in range(start, len(nums)):
        current.append(nums[i])             # choose element at index i
        backtrack(i + 1, current)           # explore further with this element
        current.pop()                       # backtrack (remove last element)
backtrack(0, [])
print(result)  # contains all subsets of nums
```

Trie (Prefix Tree)

A Trie is a tree-like data structure for storing strings where each node represents a prefix of some keys. Each edge corresponds to a character. This allows efficient insertion and lookup of words by character prefix, making tries useful for dictionary and autocomplete features. Searching for a full word or prefix takes time proportional to the length of the word or prefix.

- **Use Cases:** Autocomplete and spell-check systems, prefix-based searches (like finding all words with a given prefix), word games (e.g., Boggle), storing a large list of words for quick prefix queries.
- **Time Complexity:** O(m) for inserting or searching a string of length m (each character processed sequentially).
- **Space Complexity:** O(total characters stored) which can be high; tries trade space for speed.

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

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

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

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

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

Heap (Min-Heap and Max-Heap)

A Heap is a complete binary tree data structure that maintains a partial order such that the root is always the minimum (min-heap) or maximum (max-heap). Heaps allow efficient retrieval and removal of the min or max element, making them ideal for priority queues. In Python, the `heapq` module provides a min-heap implementation.

- **Use Cases:** Priority queue for task scheduling, Dijkstra's algorithm for shortest paths, finding the k-th smallest or largest element, any scenario requiring repeated retrieval of min/max.
- **Time Complexity:** Insertion and removal are O(log n); peeking at min/max is O(1); building a heap from n items is O(n).
- **Space Complexity:** O(n) to store the heap elements.

```
import heapq

# Min-heap example:
min_heap = []
heapq.heappush(min_heap, 5)
heapq.heappush(min_heap, 2)
heapq.heappush(min_heap, 8)
smallest = heapq.heappop(min_heap)   # gets the smallest element
```

```
# Max-heap example (using negation trick):
max_heap = []
heapq.heappush(max_heap, -5)
heapq.heappush(max_heap, -2)
heapq.heappush(max_heap, -8)
largest = -heapq.heappop(max_heap)   # gets the largest element
```

Monotonic Stack

A Monotonic Stack is a stack data structure where the elements are maintained in sorted order (either non-decreasing or non-increasing). As new elements come in, the stack pops out elements that violate the monotonic order. This structure is useful for problems where we need to find the "next greater" or "next smaller" element efficiently by eliminating unnecessary comparisons.

- **Use Cases:** Next Greater Element, Next Smaller Element, stock span problem, computing largest rectangle in histogram, "daily temperatures" problem.
- **Time Complexity:** O(n) for processing n elements (each element is pushed and popped at most once).
- **Space Complexity:** O(n) in the worst case (stack can hold all elements if input is monotonic).

```
# Example: Next Greater Element using a monotonic decreasing stack
stack = []
result = [-1] * len(arr)
for i, value in enumerate(arr):
    # Maintain stack such that top of stack is greater than current value
    while stack and arr[stack[-1]] < value:
        idx = stack.pop()
        result[idx] = value
    stack.append(i)
print(result)  # Next greater element for each index (-1 if none)
```

Monotonic Queue

A Monotonic Queue is a double-ended queue (deque) that maintains its elements in sorted order (either increasing or decreasing). It allows push and pop from both ends while enforcing that the sequence of elements is monotonic. This is particularly useful for efficiently finding the minimum or maximum in any sliding window of a list.

- **Use Cases:** Sliding window maximum/minimum problems, optimizing certain DP algorithms that rely on a window of candidate values, maintaining a window of values with quick min/max queries.
- **Time Complexity:** O(n) for processing n elements (each element is enqueued and dequeued at most once).
- **Space Complexity:** O(n) in worst case (deque can contain all elements in a pathologically ordered input).

```python
from collections import deque

# Example: Sliding window maximum for window size k
dq = deque()
max_in_window = []
for i, value in enumerate(arr):
    # Remove indices out of the current window from the front
    if dq and dq[0] < i - k + 1:
        dq.popleft()
    # Remove smaller values from the back to maintain decreasing order
    while dq and arr[dq[-1]] < value:
        dq.pop()
    dq.append(i)
    if i >= k - 1:  # window is at full size
        max_in_window.append(arr[dq[0]])
print(max_in_window)
```

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path distances from a single source to all other vertices in a weighted graph with non-negative edge weights. It uses a greedy approach: at each step, it picks the unvisited node with the smallest tentative distance and relaxes its outgoing edges. A priority queue (min-heap) is typically used to efficiently select the next vertex with the smallest distance.

- **Use Cases:** Shortest path in road networks or maps, network routing protocols, any scenario requiring the minimum cost path in a graph without negative weights.
- **Time Complexity:** O((V + E) * log V) using a min-heap (where V is the number of vertices and E is the number of edges).
- **Space Complexity:** O(V + E) for storing the graph, and O(V) for the distance array and priority queue.

```python
import heapq

def dijkstra(graph, source):
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    pq = [(0, source)]  # (distance, node)
    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]:
            continue  # stale distance entry
        for v, w in graph[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return dist
```

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm computes shortest paths between all pairs of vertices in a weighted graph (all-pairs shortest paths). It incrementally improves an initial distance matrix by considering each vertex as an intermediate point in potential paths. After processing all vertices, the distance matrix gives the shortest distance between any two vertices (or reveals a negative cycle if any distance on the diagonal becomes negative).

- **Use Cases:** Computing distances between every pair of cities in a map, finding transitive closure (by treating reachability as weights 0/1), any scenario where all-pairs path information is needed.
- **Time Complexity:** O(n^3) for a graph with n vertices (due to three nested loops).
- **Space Complexity:** O(n^2) for the distance matrix.

```
# Assume dist is an n x n matrix initialized with direct distances or inf (and 0
on the diagonal).
n = len(dist)
for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
# dist[i][j] now holds the shortest path distance from i to j
```

Bellman-Ford Algorithm

The Bellman-Ford algorithm computes the shortest paths from a single source to all other vertices in a graph that may have negative weight edges (but no negative cycles reachable from the source). It works by relaxing all edges repeatedly (up to V-1 times for V vertices). If an additional pass can still improve a distance, it indicates a negative weight cycle. Bellman-Ford can both find shortest paths and detect negative cycles.

- **Use Cases:** Shortest paths in graphs with negative weights (e.g., detecting arbitrage opportunities in currency exchange), verifying feasibility of constraints, detecting negative cycles in a graph.
- **Time Complexity:** O(V * E) where V is the number of vertices and E is the number of edges.
- **Space Complexity:** O(V) for distance storage (and O(E) for storing edges).

```
def bellman_ford(edges, V, source):
    dist = [float('inf')] * V
    dist[source] = 0
    # Relax all edges V-1 times
    for _ in range(V - 1):
        for u, v, w in edges:  # edges is list of (u, v, weight)
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    # Check for negative weight cycle
```

```
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            print("Negative weight cycle detected")
            break
    return dist
```

Bitmasking (Subset Iteration)

Bitmasking uses binary representations of integers to represent subsets or states, where each bit corresponds to the inclusion or exclusion of an element. By iterating over the range [0, 2^n), we can enumerate all subsets of an n-element set. Bit operations are also used in dynamic programming to compactly represent states (for example, using a bitmask to represent a subset of visited nodes in the Traveling Salesman Problem).

- **Use Cases:** Generating all subsets or combinations, bit DP problems (like TSP or subset sum with bit states), low-level optimizations using bit operations, toggling feature flags.
- **Time Complexity:** Iterating all subsets of n items is O(2^n * n) (2^n subsets and up to n bit checks per subset).
- **Space Complexity:** O(1) auxiliary (subset generation uses minimal extra space aside from output).

```
# Iterate over all subsets of array `arr` using bitmasks
n = len(arr)
for mask in range(1 << n):  # from 0 to 2^n - 1
    subset = []
    for j in range(n):
        if mask & (1 << j):   # if j-th bit is set, include arr[j]
            subset.append(arr[j])
    print(subset)
```

Prefix Sum (1D & 2D)

Prefix Sum is a technique of precomputing cumulative sums to answer range sum queries efficiently. In a 1D prefix sum array, each element at index i represents the sum of all elements up to i (usually exclusive) in the original array. This allows calculation of any subarray sum in O(1) by subtracting two prefix values. Similarly, a 2D prefix sum (cumulative sum matrix) stores sums of all elements in a submatrix from the origin, enabling O(1) retrieval of any submatrix sum via inclusion-exclusion.

- **Use Cases:** Quickly computing sum of any subarray or submatrix, answering range sum queries in static arrays (as in range sum query problems), cumulative frequency or histogram computations.
- **Time Complexity:** O(n) to build a 1D prefix array for an array of length n (O(n*m) for a matrix of size n x m). Querying a range sum is O(1).
- **Space Complexity:** O(n) for a 1D prefix array (O(n*m) for a 2D prefix matrix).

```
# 1D prefix sum for array `arr`
n = len(arr)
```

```
prefix = [0] * (n + 1)
for i in range(1, n+1):
    prefix[i] = prefix[i-1] + arr[i-1]

# Sum of subarray from index L to R (inclusive):
subarray_sum = prefix[R+1] - prefix[L]

# 2D prefix sum for matrix `mat` of size rows x cols
rows, cols = len(mat), len(mat[0])
prefix2D = [[0] * (cols + 1) for _ in range(rows + 1)]
for i in range(1, rows+1):
    for j in range(1, cols+1):
        prefix2D[i][j] = mat[i-1][j-1] + prefix2D[i-1][j] + prefix2D[i][j-1] -
prefix2D[i-1][j-1]

# Sum of submatrix from (r1, c1) to (r2, c2) inclusive:
submatrix_sum = (prefix2D[r2+1][c2+1] - prefix2D[r1][c2+1]
                 - prefix2D[r2+1][c1] + prefix2D[r1][c1])
```

Segment Tree

A Segment Tree is a binary tree data structure that stores information about intervals or segments of an array. It allows efficient range queries and updates (for example, range sum or minimum queries). Each node of the tree represents a segment (interval) of the array, and the tree is built such that queries can be answered by combining results from a small number of segments (logarithmic factor).

- **Use Cases:** Range sum or minimum queries on arrays, range count queries, scenarios where array values change (updates) and queries must be answered quickly, implementing operations like point updates or range updates efficiently.
- **Time Complexity:** Building the tree takes O(n). Query and update operations take O(log n) time each.
- **Space Complexity:** O(n) for the tree storage (typically about 4*n for a segment tree array).

```
# Segment Tree for range sum queries
arr = [/* initial array values */]
n = len(arr)
size = 1
while size < n:
    size *= 2
seg = [0] * (2 * size)
# Build: initialize leaves and then internal nodes
for i in range(n):
    seg[size + i] = arr[i]
for i in range(size - 1, 0, -1):
    seg[i] = seg[2*i] + seg[2*i + 1]
```

```
def query(L, R):
    # Query sum in interval [L, R) (R is exclusive)
    res = 0
    L += size; R += size
    while L < R:
        if L & 1:
            res += seg[L]
            L += 1
        if R & 1:
            R -= 1
            res += seg[R]
        L //= 2; R //= 2
    return res

def update(index, value):
    # Update arr[index] = value
    idx = index + size
    seg[idx] = value
    idx //= 2
    while idx:
        seg[idx] = seg[2*idx] + seg[2*idx + 1]
        idx //= 2
```

Binary Indexed Tree (Fenwick Tree)

A Binary Indexed Tree, or Fenwick Tree, is a data structure that provides efficient prefix sum and update operations on an array. It uses an implicit tree structure where each index has a parent determined by bit manipulation. Fenwick Trees are often easier to implement than segment trees and use less memory, at the cost of not supporting as many types of range queries without modification.

- **Use Cases:** Prefix sum or frequency queries with updates (e.g., cumulative frequency tables, computing inversions in a permutation, supporting queries like "how many elements are <= X" dynamically).
- **Time Complexity:** O(log n) for both updates and prefix sum queries. Building a BIT can be done in O(n) time.
- **Space Complexity:** O(n) for the BIT array.

```
# Fenwick Tree for prefix sums
n = len(arr)
BIT = [0] * (n + 1)
# Build BIT (1-indexed array for convenience)
for i, val in enumerate(arr, start=1):
    BIT[i] += val
    j = i + (i & -i)
    if j <= n:
        BIT[j] += BIT[i]
```

```
def prefix_sum(i):
    # Return sum of arr[0..i-1] (i is 1-indexed for BIT)
    s = 0
    while i > 0:
        s += BIT[i]
        i -= i & -i
    return s

def update(i, delta):
    # Add `delta` to arr[i-1] (i is 1-indexed for BIT)
    while i <= n:
        BIT[i] += delta
        i += i & -i
```

Tree Techniques

Depth-First Traversals (Preorder, Inorder, Postorder): These are DFS-based traversals on a binary tree. Preorder (Node-Left-Right) visits the root first, then left subtree, then right. Inorder (Left-Node-Right) visits left subtree, then root, then right (yielding sorted order for BSTs). Postorder (Left-Right-Node) visits children before the root. These traversals each run in O(n) time for n nodes (visiting each node once) and use O(h) space for recursion (h = height of tree). They're useful for various tree algorithms (preorder for copying a tree, postorder for evaluating expressions, etc.). (Assume each node is a TreeNode with `val`, `left`, and `right` attributes.)

```
def preorder(root):
    if root:
        print(root.val)
        preorder(root.left)
        preorder(root.right)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val)
```

Breadth-First Traversal (Level Order): Level order traversal visits a tree level by level from the root. It uses a queue (BFS) and runs in O(n) time and O(n) space (worst-case) for n nodes. Level order is useful for problems where nodes are processed by depth (e.g., printing level by level, populating next pointers).

```
from collections import deque

def level_order(root):
    result = []
    if not root:
        return result
    q = deque([root])
    while q:
        node = q.popleft()
        result.append(node.val)
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return result
```

Tree Height / Max Depth: The maximum depth (height) of a binary tree is the longest root-to-leaf path. This can be found via a simple DFS. It runs in O(n) time and uses O(h) space (recursion depth).

```
def max_depth(root):
    if not root:
        return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

Lowest Common Ancestor (LCA): The LCA of two nodes in a binary tree is the deepest node that has both as descendants. A recursive solution checks left and right subtrees for the targets. If one target is found in each side, the current root is the LCA. This runs in O(n) time in the worst case.

```
def lowest_common_ancestor(root, p, q):
    if not root or root == p or root == 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 if left else right
```

Binary Search Tree (BST) Insert and Validate: In a BST, insertion places a new value in sorted order (left child if smaller, right child if larger than the current node). Validating a BST involves checking that every node's value is between the valid range limits passed down from its ancestors.

- **Time Complexity:** BST insertion is O(h) (height of tree) per insert. Validating a BST is O(n) for n nodes (must check every node).
- **Space Complexity:** O(h) for recursion in both operations.

```
def insert_into_bst(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert_into_bst(root.left, val)
    else:
        root.right = insert_into_bst(root.right, val)
    return root

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