DSA Patterns — Usable Notes (summary + examples + Python snippets)

Based on pattern■based learning advice (like the LinkedIn post you shared) and public domain examples. This condensed notes PDF highlights common patterns, how to identify them, approach, complexity, and one or two exemplar problems with short Python solutions.

1. Sliding Window

What / When: Use when you need to find subarray/subsequence with a condition (sum/length/max/min) and all elements are non-negative or window can expand/contract. Maintain a window with two indices and adjust based on condition.

Approach: Expand right pointer, update state; while condition violated, shrink left pointer and update state.

Time Complexity: O(n) typical

Example:

```
def max_sum_k(nums, k):
    if k > len(nums): return None
    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
```

2. Two Pointers

What / When: Use two indices moving from ends or one slow/one fast depending on ordering. Common for sorted arrays, pair-sum, removing duplicates in-place.

Approach: Sort if needed. Move pointers based on comparison; when condition met, move pointers appropriately.

Time Complexity: O(n) after sort (O(n log n)) if sorting required

Example:

```
def two_sum_sorted(nums, target):
    i, j = 0, len(nums)-1
    while i < j:
        s = nums[i] + nums[j]
        if s == target:
            return (i, j)
        if s < target:
            i += 1
        else:
            j -= 1
    return None
```

3. Fast and Slow Pointers (Floyd / Cycle Detection)

What / When: Use when detecting cycles in linked lists or when you need to find middle of list. One pointer advances faster than the other.

Approach: Advance slow by 1, fast by 2. If they meet, cycle exists. To find start of cycle, reset one pointer to head and move both by 1.

Time Complexity: O(n)

Example:

```
# Conceptual (uses node.next)
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
```

4. Prefix Sum (Cumulative Sum)

What / When: Precompute cumulative sums to answer range-sum queries or to convert problems into constant-time checks for sums.

Approach: Build prefix array where pref[i] = sum(nums[:i]). Range sum i..j = pref[j+1]-pref[i]. Useful with hashmaps to find subarray with target sum (including negatives).

Time Complexity: O(n)

Example:

```
from collections import defaultdict
def count_subarrays_with_sum(nums, target):
    pref = 0
    cnt = 0
    freq = defaultdict(int)
    freq[0] = 1
    for x in nums:
        pref += x
        cnt += freq[pref - target]
        freq[pref] += 1
    return cnt
```

5. Hashing / Frequency Map

What / When: Use hashmap to store frequencies, indices, or existence checks. Great for anagrams, two-sum (unsorted), longest substring without repeating chars, etc.

Approach: Map keys to required information (count, last index). Combine with sliding window or prefix sums when needed.

Time Complexity: O(n)

Example:

```
def length_of_longest_substring(s):
    last = {}
    start = 0
    max_len = 0
    for i, ch in enumerate(s):
        if ch in last and last[ch] >= start:
            start = last[ch] + 1
        last[ch] = i
        max_len = max(max_len, i - start + 1)
    return max_len
```

6. DFS / Backtracking

What / When: Use when exploring all combinations/paths (permutations, subsets, combinations, n-queens). DFS with recursion and undo (backtrack) changes.

Approach: Choose a candidate, recurse, then undo (pop) to explore other choices.

Time Complexity: Exponential in worst-case (e.g., O(2^n) for subsets)

Example:

```
def subsets(nums):
    res = []
    def dfs(i, path):
        if i == len(nums):
            res.append(path[:])
            return
        # include
        path.append(nums[i])
        dfs(i+1, path)
        path.pop()
        # exclude
        dfs(i+1, path)
    dfs(0, [])
    return res
```

7. Graph Traversals (BFS / DFS)

What / When: Use BFS for shortest path in unweighted graphs and layering; DFS for connectivity, topological sort (with modifications).

Approach: Represent graph with adjacency list. Use queue for BFS, stack/recursion for DFS. Track visited set.

Time Complexity: O(V+E)

Example:

```
from collections import deque
def bfs_shortest_paths(adj, src):
    dist = {src: 0}
    q = deque([src])
    while q:
        u = q.popleft()
        for v in adj.get(u, []):
            if v not in dist:
                dist[v] = dist[u] + 1
                q.append(v)
    return dist
```

8. Topological Sort (Directed Acyclic Graphs)

What / When: Use to order tasks given dependencies (build order). Kahn's algorithm (BFS with indegree) or DFS post-order.

Approach: Kahn: compute indegree, push zeros to queue, pop and decrement neighbors' indegree.

Time Complexity: O(V+E)

Example:

```
from collections import deque
```

```
def topo_sort(adj):
    indeg = {u:0 for u in adj}
    for u in adj:
        for v in adj[u]:
            indeg[v] = indeg.get(v,0) + 1
    q = deque([u for u in indeg if indeg[u]==0])
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in adj.get(u,[]):
            indeg[v] -= 1
            if indeg[v]==0:
                q.append(v)
    if len(order) == len(indeg):
        return order
    return None  # cycle found
```

9. Dynamic Programming (Top-Down & Bottom-Up)

What / When: Use when optimal substructure and overlapping subproblems exist. Identify state and transition. Start with a recursive relation, memoize (top-down) or tabulate (bottom-up).

Approach: Define dp state (e.g., dp[i] or dp[i][j]), base cases, and transitions. Test small inputs and build up.

Time Complexity: Depends on state size (often polynomial vs exponential brute-force)

Example:

```
def fib(n, memo={}):
    if n < 2: return n
    if n in memo: return memo[n]
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

10. Greedy

What / When: Use when local optimal choices lead to global optimum (can be proved by exchange argument or matroid theory). Examples: interval scheduling, coin change (for canonical coin systems), activity selection.

Approach: Sort by key (finish time, cost/benefit ratio) and pick iteratively if compatible.

Time Complexity: O(n log n) typically due to sorting

Example:

```
def max_activities(intervals):
    intervals.sort(key=lambda x: x[1])  # sort by finish time
    res = []
    last_end = -10**18
    for s, e in intervals:
        if s >= last_end:
            res.append((s,e))
            last_end = e
    return res
```

Practice Plan (Suggested)

1. Fundamentals (1 week): arrays, strings, hashmaps, stacks, queues, basic sorting.
2. Pattern buckets (4-6 weeks): pick 1 pattern per 2-3 days and solve 15-25 problems deeply; review mistakes.
3. Timed mocks (2 weeks): 1-2 timed sessions per week, simulate interviews/contests.
4. Revision (ongoing): revisit patterns, redo problems you previously failed, explain solutions aloud.

Tips:
- Focus on spotting patterns first; write down which pattern(s) apply before coding.
- When stuck, try to reduce constraints (small inputs) and brute-force to find structure.
- Analyze complexity and edge cases; add tests for empty inputs, single-element, duplicates.
- Maintain a personal notebook of 3-5 solved problems per pattern with short notes.