Algorithmic Patterns Reference Guide Outline

PHASE 1: FOUNDATION

FAST & SLOW POINTER

This technique uses two pointers moving at different speeds to solve problems involving **cycles**, such as finding the middle of a list, detecting loops, or checking for palindromes.[^1][^2] **Complexity:** O(N) Time | O(1) Space.[^1][^3]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Scan for “cycle/middle”; note list length N 10 .
2. Identify	1–2 min	If you see “cycle/middle”, choose Fast & Slow pointers.
3. Plan	4–6 min	Draw the list; plan `slow = 1` step, `fast = 2` steps; confirm O(N) scan.
4. Code	8–10 min	Initialize `slow/fast`; write a loop with null checks; comment each step.
5. Test	3–5 min	Test cycle vs no-cycle; edge cases: empty list, single node; log positions.
6. Review	2–3 min	Note relation to Floyd’s cycle detection; confirm O(1) extra space.

Keywords: cycle, loop, middle, meet **Key phrase:** cycle **30-second detection:** “cycle” or “middle of linked list” → Fast/Slow pointers.[^2][^1]

How to Solve (3 Steps)

1. Initialize `slow` and `fast` pointers at the head node.[^4][^5]
2. Move `slow` by 1 step and `fast` by 2 steps in each iteration.[^1][^2]
3. If `slow` and `fast` meet, a cycle exists; if `fast` or `fast.next` becomes null, there is no cycle.[^3][^1]

Cheat code: use `slow = slow.next` and `fast = fast.next.next`. **Memory hack:** “1 step vs 2 steps; the faster pointer eventually laps the slower one in a cycle (tortoise and hare)”.[^2]

Real-Life Intuition

- Detecting **repetitive sequences** in streams (network packets, sensor readings) that indicate cyclic behavior.[^6]
- Finding the **middle** of a very long sequential log without loading the entire log into memory at once.[^1]

Detect: cycle, loop, middle, tortoise and hare. **Visual:** two runners on a circular track, one twice as fast.[^2]

Practice: Easy & Medium

- 141.Linked List Cycle (Easy).[^7][^4]
- 21.Merge Two Sorted Lists (Easy).
- 142.Linked List Cycle II (Medium).[^8][^1]

Practice: Medium & Hard

- 876.Middle of the Linked List (Medium).
- 287.Find the Duplicate Number (Hard) using cycle modeling.

Code Snippet (Java)

```java
ListNode slow = head, fast = head;

while (fast != null && fast.next != null) {
    slow = slow.next;           // 1 step
    fast = fast.next.next;      // 2 steps
    if (slow == fast) {
        return true;            // cycle detected
    }
}
return false;                   // no cycle
```

References – Fast & Slow Pointer

- How does Floyd's slow and fast pointers approach work? (GeeksforGeeks) [web:87]
- Floyd's Cycle Finding Algorithm (GeeksforGeeks) [web:88]
- Floyd's tortoise and hare explanation (YouCademy) [web:93]
- LeetCode Patterns – Fast & Slow Pointers section (Algomaster blog) [web:92]
- Fast and Slow Pointers in 6 Minutes (YouTube) [web:95]
- Linked List Cycle – Fast and Slow Pointers, LeetCode 141 (YouTube) [web:94]
- Why Floyd's tortoise and hare works for arrays / Find the Duplicate Number (Stack Overflow) [web:96]

TWO POINTERS

This technique uses two indices that move over an array or string (from the same side or opposite ends) to efficiently solve problems like pair sums, partitioning, and palindrome checks without extra space.[^10][^11] **Complexity:** O(N) Time | O(1) Space.[^10]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Scan for “sorted array/string” and operations on pairs or ends.
2. Identify	1–2 min	If you see “two indices”, “pair sum”, or “check from both ends”, choose Two Pointers.
3. Plan	4–6 min	Decide start positions (both ends or same side) and pointer movement rules.
4. Code	8–10 min	Initialize `left/right`; loop while `left   right`; move one or both pointers based on condition.
5. Test	3–5 min	Test typical, no-solution, and edge cases (empty, size 1, already sorted).
6. Review	2–3 min	Confirm linear scan, no extra array; relate to Sliding Window and Fast/Slow.

Keywords: sorted, pair, triplet, left/right, indices **Key phrase:** “two indices on a sorted array” **30-second detection:** “sorted + pair/triplet or palindrome” → Two Pointers.[^11]

How to Solve (3 Steps)

1. Initialize two pointers (e.g., `left = 0`, `right = n - 1` on a sorted array).[^12][^10]
2. Evaluate current configuration (sum/comparison); if too small, move the pointer that should increase the value; if too large, move the other.[^10][^12]
3. Continue moving pointers until they cross or the condition is satisfied.[^11][^10]

Cheat code: sorted array + “find pair/triplet with sum/condition” → `left++` or `right--` instead of nested loops.[^12] **Memory hack:** two people walking from opposite ends of a line of numbers and shaking hands at the answer.[^13]

Real-Life Intuition

- Matching **two sorted logs** from the ends until constraints are met in the middle.[^14]
- Checking if a string is a **palindrome** by comparing characters from both ends.[^15]

Detect: "from both ends", "pair sum in sorted array", "minimal difference between two elements".[^11] **Visual:** two markers sliding toward each other on a ruler.

Practice: Easy & Medium

- 167.Two Sum II – Input array is sorted (Easy).[^16][^12]
- 125.Valid Palindrome (Easy).[^15]
- 15.3Sum (Medium) – sort + outer loop + inner two pointers.[^10]

Practice: Medium & Hard

- 11.Container With Most Water (Medium).[^11]
- 42.Trapping Rain Water (Hard, two-pointer version).[^10]

Code Snippet (Java)

```java
boolean hasPairWithSum(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left < right) {
        int sum = nums[left] + nums[right];

        if (sum == target) {
            return true;
        } else if (sum < target) {
            left++;         // need larger sum
        } else {
            right--;        // need smaller sum
        }
    }
    return false;
}
```

References – Two Pointers

- Two Pointers Technique (GeeksforGeeks) [web:100]
- The two-pointers approach for array and string problems (article) [web:102]
- Two Pointers in 7 Minutes | LeetCode Pattern (YouTube) [web:97]
- Two-Pointer Technique – in-depth guide with visuals and LeetCode questions (Reddit post) [web:98]
- What is the Two-Pointers Technique & How to use it? (YouTube) [web:104]
- Using the Two Pointer Technique (AlgoDaily guide) [web:105]

SLIDING WINDOW

This technique uses a moving "window" over a sequence (array or string) to efficiently handle all contiguous subarrays/substrings that satisfy some condition (sum, length, counts) by updating the window incrementally.[^17][^18] **Complexity:** O(N) Time | O(1)–O(K) Space depending on tracked data (sum, counts, map).[^19][^20]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Scan for "subarray/substring", "contiguous", and maybe window size K.
2. Identify	1–2 min	If you see "contiguous segment" + optimize length/sum/count → Sliding Window.
3. Plan	4–6 min	Decide fixed vs variable window; choose what to track (sum, freq map).
4. Code	8–10 min	Initialize `left/right`; expand right; shrink left while condition breaks; update answer.
5. Test	3–5 min	Test fixed-size, variable-size, and no-solution cases.
6. Review	2–3 min	Relate to Two Pointers; confirm linear scan and incremental updates.

Keywords: subarray, substring, contiguous, window, range **Key phrase:** "contiguous subarray/substring with some condition" **30-second detection:** "contiguous + optimize length/sum/count" → Sliding Window.[^21][^17]

How to Solve (3 Steps)

1. Initialize `left = 0`, a running state (sum/map), and move `right` to expand the window.[^19][^20]
2. While the window violates the condition, move `left` to shrink and update the state.[^22][^23]
3. After each valid window update, update the best answer (max/min length, sum, etc.).[^18][^20]

Cheat code: "add the new right element, remove the old left element; never recompute from scratch".[^19] **Memory hack:** a highlighter strip sliding over the array; only edges change.

Real-Life Intuition

- Rolling **average** of CPU/network usage over last K readings.[^22]

- Longest **streak** of valid days in a log by expanding and shrinking a time window.[^20]

Detect: “every subarray/substring of size K”, “longest/shortest contiguous segment”, “minimum window containing…”.[^17][^18] **Visual:** a rectangle sliding along a line of elements.

Practice: Easy & Medium

- Maximum Sum Subarray of Size K (Easy).
- 3.Longest Substring Without Repeating Characters (Medium).
- 209.Minimum Size Subarray Sum (Medium).

Practice: Medium & Hard

- 340.Longest Substring with At Most K Distinct Characters (Medium).
- 239.Sliding Window Maximum (Hard).

Code Snippet (Java)

```java
int maxSubarraySumOfSizeK(int[] nums, int k) {
    if (nums.length < k) return 0;

    int windowSum = 0;
    for (int i = 0; i < k; i++) {
        windowSum += nums[i];
    }

    int maxSum = windowSum;
    for (int right = k; right < nums.length; right++) {
        windowSum += nums[right];
        windowSum -= nums[right - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

References – Sliding Window

- Sliding Window Technique (GeeksforGeeks) [web:107]
- How to Use the Sliding Window Technique – Algorithm Guide (freeCodeCamp) [web:110]
- Sliding Window Algorithm – Fixed & Variable Size Solutions (Tech With KP) [web:113]
- Sliding Window Algorithm Explained (Built In) [web:114]
- Sliding Window in 7 Minutes | LeetCode Pattern (YouTube) [web:111]

- Sliding Window Technique (YouTube, with static and dynamic window examples) [web:108]
- Sliding Window Technique + 4 Questions (YouTube, multiple difficulty levels) [web:109]
- Effective LeetCode: Understanding the Sliding Window Pattern (blog) [web:115]
- Sliding Window: The Complete Guide – Patterns, Templates, and LeetCode Problems [web:112]

PREFIX SUM

This technique precomputes cumulative sums (or counts/XORs) so range queries can be answered quickly using differences of prefix values.[^26] **Complexity:** O(N) Preprocessing | O(1) per query | O(N) Space.[^27][^26]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for many range queries like "sum from L to R" or repeated subarray queries.
2. Identify	1–2 min	If you see "many range queries" or "subarray sum quickly", think Prefix Sum.
3. Plan	4–6 min	Define `prefix[i]` = combined value from start to i (sum/count/XOR).
4. Code	8–10 min	Build prefix array in one pass; answer each query using differences.
5. Test	3–5 min	Test single query, multiple queries, edges (L = 0, R = n-1).
6. Review	2–3 min	Confirm O(N + Q) overall; relate to Sliding Window where K is fixed.

Keywords: range sum, range count, many queries, cumulative **Key phrase:** "answer many range queries fast" **30-second detection:** "subarray sum between L and R for many queries" → Prefix Sum.[^28][^26]

How to Solve (3 Steps)

1. Build an array `prefix` where `prefix[^0] = nums[^0]` and `prefix[i] = prefix[i-1] + nums[i]` (or analogous for count/XOR).[^27][^26]
2. To answer sum for range `[L, R]`, use `prefix[R] - (L > 0 ? prefix[L-1] : 0)`.
3. Repeat for each query in O(1) using the precomputed prefix array.

Cheat code: "store the work once; each query just does `right - leftMinusOne`." **Memory hack:** think of prefix as checkpoints along a road showing distance from the start.

Real-Life Intuition

- Answering many **sales total** queries between two dates from a daily revenue list.[^26]
- Quickly computing **population** or **traffic** in subranges on a timeline.

Detect: "many queries", "sum/count between indices L and R".[^28] **Visual:** cumulative graph where any segment height difference gives the range result.

Practice: Easy & Medium

- 303.Range Sum Query – Immutable (Easy).
- 560.Subarray Sum Equals K (Medium) – prefix + hashmap of seen sums.[^27]
- 2270.Number of Ways to Split Array (Medium).

Practice: Medium & Hard

- 327.Count of Range Sum (Hard) – prefix + divide-and-conquer or BIT/segment tree.
- 304.2D Prefix Sum problems (matrix sums).

Code Snippet (Java)

```java
int[] buildPrefixSum(int[] nums) {
    int n = nums.length;
    int[] prefix = new int[n];
    if (n == 0) return prefix;
    prefix[^0] = nums[^0];
    for (int i = 1; i < n; i++) {
        prefix[i] = prefix[i - 1] + nums[i];
    }
    return prefix;
}

int rangeSum(int[] prefix, int L, int R) {
    if (L == 0) return prefix[R];
    return prefix[R] - prefix[L - 1];
}
```

References – Prefix Sum

- Prefix Sum Array – Implementation and Applications (GeeksforGeeks) [web:117]

- Introduction to Prefix Sums (USACO Guide) [web:119]
- Prefix Sum of Matrix / 2D Array (GeeksforGeeks) [web:123]
- Understanding LeetCode Prefix Sum Pattern – Engineering With Java [web:125]
- A Visual Guide to Prefix Sums (Reddit post) [web:118]
- Prefix Sum in 4 Minutes | LeetCode Pattern (YouTube) [web:124]
- Prefix Sum Array and Range Sum Queries (YouTube) [web:120]
- The Prefix Sum Array Problem – Java walkthrough (TheServerSide) [web:121]
- More Prefix Sums – max subarray and 2D variants (USACO Guide) [web:126]

CYCLIC SORT

This technique places each number at its correct index (usually when the array contains numbers from 1…N or 0…N) by swapping elements in-place, which naturally exposes missing and duplicate values.[^29] **Complexity:** O(N) Time | O(1) Extra Space.[^29]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for arrays containing values in range 1…N or 0…N, often with missing/duplicate.
2. Identify	1–2 min	If problem asks for missing/duplicate with constant space, think Cyclic Sort.
3. Plan	4–6 min	Plan a loop index `i` and compute the correct index for `nums[i]`.
4. Code	8–10 min	While `nums[i]` is not at its correct index and not equal to target spot, swap.
5. Test	3–5 min	Test arrays with no duplicates, with duplicates, small N, and all-correct cases.
6. Review	2–3 min	Confirm each element moves at most once or twice; verify O(N), O(1) space.

Keywords: 1 to N, 0 to N, missing number, duplicate number, constant space
Key phrase: "numbers are in the range 1…N (or 0…N) with missing/duplicate"
30-second detection: "find missing/duplicate in 1…N with O(1) space" → Cyclic Sort.[^29]

How to Solve (3 Steps)

1. Use index `i = 0`; for each `nums[i]`, compute its correct index (e.g., `correct = nums[i] - 1` for 1…N).

2. If `nums[i]` is in range and `nums[i] != nums[correct]`, swap `nums[i]` and `nums[correct]`; otherwise, increment `i`.[^29]
3. After the pass, any index where `nums[i] != i+1` reveals missing/duplicate depending on the problem.

Cheat code: "put each number at index (value−1); what doesn't fit points to the answer". **Memory hack:** think of a set of labeled lockers 1…N; each item must go to its own locker.

Real-Life Intuition

- Assigning **ID-tagged items** to bins labeled by ID; misplaced items reveal duplicates/missing IDs.
- Auditing **sequential invoice numbers** to find which numbers are missing.

Detect: "array of size N with values 1…N or 0…N", "missing/duplicate with constant space". **Visual:** repeatedly swapping items into their correct labeled slots.

Practice: Easy & Medium

- 268.Missing Number (Easy) – can be done by XOR/sum or by Cyclic Sort style.
- 645.Set Mismatch (Easy/Medium) – find one duplicate and one missing.
- 41.First Missing Positive (Hard-ish classic pattern, often taught with Cyclic Sort).

Practice: Medium & Hard

- 448.Find All Numbers Disappeared in an Array (Easy/Medium).
- 442.Find All Duplicates in an Array (Medium).

Code Snippet (Java)

```java
void cyclicSort(int[] nums) {
    int i = 0;
    while (i < nums.length) {
        int correct = nums[i] - 1;       // for values 1..N
        if (nums[i] >= 1 && nums[i] <= nums.length && nums[i] != nums[correct]) {
            int tmp = nums[i];
            nums[i] = nums[correct];
            nums[correct] = tmp;
        } else {
            i++;
        }
    }
}
```

References – Cyclic Sort

- Cycle Sort / Cyclic Sort – theory and basic implementation (GeeksforGeeks) [web:127]
- What is a cyclic sort algorithm? – overview, intuition, and use-cases (Educative) [web:128]
- Cycle Sort Algorithm – minimal swaps explanation (Baeldung) [web:131]
- Cycling Through Cycle Sort – interview-focused lesson (AlgoDaily) [web:130]
- LeetCode pattern: Cyclic Sort – when and how to use it, with templates [web:132]
- Cyclic Sort: In-Place, Linear Time Sorting – deep dive and applications [web:133]
- Mastering Cyclic Sort – step-by-step algorithm and examples (Algocademy) [web:134]
- Coding Patterns: Cyclic Sort – pattern-based explanation with LeetCode examples [web:135]

PHASE 2: INTERMEDIATE

REVERSE LINKED LIST

This technique reverses the direction of pointers in a linked list so that the head becomes the tail and traversal order is flipped.[^1][^2] **Complexity:** O(N) Time | O(1) Space (iterative).[^1]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Scan for "reverse a linked list" or "reverse sub-list / in groups".
2. Identify	1–2 min	If the task says reverse list (entire, partial, or k-group), choose Reverse Linked List.
3. Plan	4–6 min	Decide iterative vs recursive; track `prev`, `curr`, `next`.
4. Code	8–10 min	Loop while `curr != null`; redirect `curr.next` to `prev`; advance pointers.
5. Test	3–5 min	Test empty, single node, two nodes, and normal list.
6. Review	2–3 min	Confirm links are not lost (always save `next` first); verify O(1) extra memory.

Keywords: reverse, linked list, reverse in groups, reverse sublist **Key phrase:** "reverse the linked list" **30-second detection:** any variant of "reverse list" → Reverse Linked List.[^3][^1]

How to Solve (3 Steps)

1. Initialize `prev = null`, `curr = head`, and `next` as a temp pointer.[^2][^1]
2. For each node, save `next = curr.next`, set `curr.next = prev`, then move `prev = curr`, `curr = next`.
3. When `curr` becomes null, `prev` is the new head; return `prev`.

Cheat code: "prev–curr–next triangle: save next, reverse link, move forward."
Memory hack: imagine turning each arrow in the list around one by one.

Real-Life Intuition

- Reversing the order of **events** in a log so the latest appear first.
- Undoing a **stack of tasks** represented as a linked sequence by flipping the direction.

Detect: "reverse linked list", "reverse in k-group", "reverse between positions m and n". **Visual:** a chain of arrows all flipped to point backwards.

Practice: Easy & Medium

- 206.Reverse Linked List (Easy) – full list reverse.[^1]
- 92.Reverse Linked List II (Medium) – reverse a sub-list.
- 25.Reverse Nodes in k-Group (Hard/Medium pattern).[^3]

Practice: Medium & Hard

- 234.Palindrome Linked List – reverse second half and compare.
- 143.Reorder List – partial reversing plus weaving nodes.

Code Snippet (Java)

```java
ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;
    while (curr != null) {
        ListNode next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

References – Reverse Linked List

- Reverse a Linked List – iterative and pointer-based explanation (GeeksforGeeks) [web:138]
- Reverse a linked list using recursion (GeeksforGeeks) [web:137]

- Reverse a linked list – iterative and recursive approaches (YouCademy) [web:141]
- 206. Reverse Linked List – LeetCode solution and patterns (AlgoMap) [web:142]
- Reverse Linked List – Iterative AND Recursive | LeetCode 206 (NeetCode video) [web:139]
- Reverse Linked List | Iterative and Recursive (takeUforward video) [web:140]
- Reverse Linked List | LeetCode 206 | Iterative vs Recursive Solution (article + code) [web:144]
- 206. Reverse Linked List – stepwise explanation and solutions (PrepInsta) [web:145]
- Reverse Linked List II – sublist reversal variant (LeetCode problem) [web:146]

MODIFIED BINARY SEARCH

This pattern adapts binary search to handle rotated arrays, search ranges, or predicate-based decisions while preserving logarithmic time.[^4][^5] **Complexity:** O(log N) Time | O(1) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “sorted or rotated” array and log-time requirement.
2. Identify	1–2 min	If array is sorted/rotated or question asks “log N”, think Modified Binary Search.
3. Plan	4–6 min	Choose invariant: mid comparison, check which half is sorted or which side is valid.
4. Code	8–10 min	Use `left`, `right`, compute `mid`; adjust boundaries based on rules.
5. Test	3–5 min	Test present/absent target, edge sizes (1,2), rotated and non-rotated.
6. Review	2–3 min	Confirm each loop halves search space; no infinite loops.

Keywords: sorted, rotated, first/last occurrence, log N **Key phrase:** “sorted/rotated + O(log N)” **30-second detection:** “search in rotated sorted array” or “find boundary with log N” → Modified Binary Search.[^5][^4]

How to Solve (3 Steps)

1. Set `left = 0`, `right = n - 1`; compute `mid = (left + right) / 2` each loop.[^5]
2. Determine which half is sorted or which side satisfies the predicate; if the target lies in that region, narrow to it; otherwise, go to the other half.[^4]
3. Repeat until target is found or `left > right`; for bounds (first/last), bias search appropriately.

Cheat code: "identify sorted half, then decide if target is inside that half."[^4]
Memory hack: picture a rotated ruler; one side is still ordered and can be checked for containment.

Real-Life Intuition

- Searching in a **rotated time-series** (e.g., logs starting mid-night) by focusing on the ordered portion.
- Finding first/last time a condition became true using yes/no checks in a timeline.

Detect: "search in rotated sorted array", "find first/last position in sorted array", "minimum in rotated array". **Visual:** a spotlight that always halves the remaining search region.

Practice: Easy & Medium

- 704.Binary Search (Easy) – base template.
- 33.Search in Rotated Sorted Array (Medium).[^5][^4]
- 34.Find First and Last Position of Element in Sorted Array (Medium).

Practice: Medium & Hard

- 153.Find Minimum in Rotated Sorted Array (Medium).
- 852.Peak Index in a Mountain Array / 162.Find Peak Element.

Code Snippet (Java)

```java
int searchRotated(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;

        if (nums[left] <= nums[mid]) { // left half sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else { // right half sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

References – Modified Binary Search

- Coding Patterns: Modified Binary Search – core pattern + LeetCode examples [web:148]
- Pattern 11: Modified Binary Search – Coding Patterns for Interviews (GitHub notes) [web:153]
- Several Coding Patterns – Pattern 11: Modified Binary Search (GitHub) [web:147]
- LeetCode Patterns – Modified Binary Search section (Algomaster blog) [web:92]
- Search in Rotated Sorted Array – in-depth explanation (AlgoMonster) [web:152]
- Binary Search on Answer – technique and identification (GeeksforGeeks) [web:151]
- Binary Search – base algorithm and edge cases (GeeksforGeeks) [web:154]
- Binary search on rotated sorted array – discussion and ideas (Stack Overflow) [web:155]

OVERLAPPING INTERVALS

This pattern sorts intervals and then merges or processes them linearly to handle overlaps, insertions, and resource calculations.[^6][^7] **Complexity:** O(N log N) Time (sorting) | O(N) Space (for result).

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for intervals with start/end: meetings, ranges, bookings, segments.
2. Identify	1–2 min	If asked to merge, insert, or count overlaps, choose Overlapping Intervals.

Step	Time	Description
3. Plan	4–6 min	Sort by start; keep a “current merged interval” to grow or finalize.
4. Code	8–10 min	Loop over sorted intervals; merge if overlap, else push and reset.
5. Test	3–5 min	Test disjoint, fully overlapping, nested intervals, and single interval.
6. Review	2–3 min	Verify stable merging condition and sorted order; complexity O(N log N).

Keywords: intervals, start/end, merge, overlaps, schedule **Key phrase:** “merge overlapping intervals” **30-second detection:** intervals + “merge/insert/overlap” → Overlapping Intervals.[^7][^6]

How to Solve (3 Steps)

1. Sort intervals by start time.[^6]
2. Initialize a current interval; for each interval, if it overlaps with current, extend current’s end to `max(end, newEnd)`; otherwise, append current and start a new one.[^7]
3. After processing all intervals, append the last current interval to the result.

Cheat code: “sort by start; if `start2 <= end1`, merge; else, push and reset.”[^6] **Memory hack:** think of painting segments on a number line and merging overlapping paint strokes.

Real-Life Intuition

- Merging **meeting times** so a calendar shows consolidated busy slots.[^6]
- Consolidating **IP or numeric ranges** for firewall or resource allocation.

Detect: “meeting rooms”, “merge ranges”, “minimum number of rooms”, “can we attend all meetings”. **Visual:** overlapping bars on a timeline collapsing into larger bars.

Practice: Easy & Medium

- 56.Merge Intervals (Medium).[^7][^6]
- 57.Insert Interval (Medium).
- 252.Meeting Rooms / 253.Meeting Rooms II (Medium).

Practice: Medium & Hard

- 759.Employee Free Time (Hard).
- Number of Airplanes in the Sky (line sweep variant).

Code Snippet (Java)

```
int[][] merge(int[][] intervals) {
    if (intervals.length == 0) return intervals;

    Arrays.sort(intervals, (a, b) -> Integer.compare(a[^0], b[^0]));
    List<int[]> res = new ArrayList<>();
    int[] curr = intervals[^0];

    for (int i = 1; i < intervals.length; i++) {
        int[] next = intervals[i];
        if (next[^0] <= curr[^1]) {
            curr[^1] = Math.max(curr[^1], next[^1]); // extend
        } else {
            res.add(curr);
            curr = next;
        }
    }
    res.add(curr);
    return res.toArray(new int[res.size()][]);
}
```

References – Overlapping Intervals

- Merge Overlapping Intervals – explanation and code (GeeksforGeeks) [web:156]
- Merge Overlapping Intervals – step-by-step article with visuals (InterviewBit) [web:159]
- Merge Overlapping Intervals – intuition and in-place merge approach (EnjoyAlgorithms) [web:160]
- Merge Overlapping Intervals – video walkthrough (GeeksforGeeks) [web:158]
- Merge Overlapping Intervals – brute and optimal + TC analysis (takeUforward video) [web:157]
- LeetCode Patterns – Overlapping Intervals section (Algomaster blog) [web:92]
- How to solve interval problems on LeetCode with a simple pattern – sort + greedy/sweep line (LinkedIn post) [web:161]
- Merging Overlapping Intervals – discussion and implementations (Stack Overflow) [web:162]
- Generic interval-processing template with merge logic (CodeInMotion patterns blog) [web:164]

MATRIX MANIPULATION

This pattern handles 2D matrices using controlled traversal, boundary management, and sometimes in-place transformations (rotate, transpose, flip, search).[^8] **Complexity:** Typically O(R · C) Time | O(1)–O(R · C) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for 2D grid/matrix, rows and columns, neighbors, directions.
2. Identify	1–2 min	If operations are per-cell (rotate, search, transform), choose Matrix Manipulation.
3. Plan	4–6 min	Define coordinate system, loops (by row/column/diagonal), and boundary checks.
4. Code	8–10 min	Implement nested loops; update cells or use extra structure as needed.
5. Test	3–5 min	Test square vs rectangular, 1-row/1-column, and edge rotations.
6. Review	2–3 min	Ensure indices and bounds are correct; verify in-place vs extra memory.

Keywords: matrix, grid, rows, columns, rotate, transpose, spiral **Key phrase:** "2D grid/matrix operations" **30-second detection:** "matrix + rotate/transpose/spiral/search" → Matrix Manipulation.

How to Solve (3 Steps)

1. Identify traversal order (row-wise, column-wise, layer by layer, or directional).
2. Implement nested loops or direction vectors to visit required cells while checking bounds.
3. Perform in-place swaps/updates or populate a result matrix as required.

Cheat code: for rotation, often "transpose + reverse rows/columns"; for spiral, think shrinking boundaries. **Memory hack:** picture the matrix as layers of an onion or as rows of shelves you step through.

Real-Life Intuition

- Manipulating **image pixels** stored in a 2D array (rotate, flip, transpose).
- Operating on **game boards** or grids like Sudoku and minesweeper.

Detect: "rotate matrix", "spiral order", "search in 2D matrix", "zero out rows/columns". **Visual:** walking through rows/columns with clear borders.

Practice: Easy & Medium

- 48.Rotate Image (Medium).
- 54.Spiral Matrix (Medium).
- 73.Set Matrix Zeroes (Medium).

Practice: Medium & Hard

- 74.Search a 2D Matrix / 240.Search a 2D Matrix II.
- 79.Word Search (borderline DFS + matrix pattern).

Code Snippet (Java – rotate matrix 90°)

```java
void rotate(int[][] matrix) {
    int n = matrix.length;

    // transpose
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = tmp;
        }
    }

    // reverse each row
    for (int i = 0; i < n; i++) {
        int left = 0, right = n - 1;
        while (left < right) {
            int tmp = matrix[i][left];
            matrix[i][left] = matrix[i][right];
            matrix[i][right] = tmp;
            left++; right--;
        }
    }
}
```

References – Matrix Manipulation

- Top 50 Matrix/Grid Coding Problems for Interviews (GeeksforGeeks) [web:165]
- Top 50 Matrix/Grid Data Structure Problems – curated list (Tutorials-Point) [web:166]
- 10 Matrix Interview Questions – classic matrix tasks list (GitHub) [web:169]
- Rotate Image | LeetCode 48 – transpose + reverse in-place technique [web:170]

- Rotate Image – LeetCode solution and explanation (AlgoMap) [web:173]
- Frequently Asked Matrix Coding Questions – concepts and step-by-step solutions (Educative) [web:172]
- Matrix Interview Questions & Tips – when and how to use matrices in interviews [web:167]
- 20 Essential Coding Patterns – Island / Matrix Traversal pattern section [web:174]

BITWISE XOR

This pattern uses XOR's properties (self-canceling, no carry) to find unique elements, swap without temp, or track parity efficiently.[^9] **Complexity:** O(N) Time | O(1) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for "every element appears twice except…", or parity/bit tricks.
2. Identify	1–2 min	If pairs cancel out and one or two unique values remain, think XOR.
3. Plan	4–6 min	Decide what to XOR (all numbers, or partition into groups by a bit).
4. Code	8–10 min	Accumulate XOR in one pass; possibly split by significant bit.
5. Test	3–5 min	Test small arrays, all-duplicate cases, and multiple unique values.
6. Review	2–3 min	Re-check XOR properties: `a^a=0`, `a^0=a`, commutative & associative.

Keywords: XOR, bitwise, unique element, parity, toggling **Key phrase:** "every element appears twice except…" **30-second detection:** "all elements twice except one/two" → Bitwise XOR.

How to Solve (3 Steps)

1. XOR all elements; pairs cancel leaving the unique element (for single-unique problems).
2. For two unique elements, take XOR of all to get `x = a ^ b`, find a set bit, and partition numbers by that bit.[^9]
3. XOR each group separately to recover each unique value.

Cheat code: treat duplicates as +1 and -1 in XOR; they zero out, leaving the odd one. **Memory hack:** imagine flipping switches; each duplicate flips twice and returns to off.

Real-Life Intuition

- Detecting a **single corrupted record ID** among paired logs.
- Tracking **parity** or changes where only mismatched items matter.

Detect: "appears once while others appear twice/thrice", "no extra memory allowed". **Visual:** bits turning on/off with each XOR.

Practice: Easy & Medium

- 136.Single Number (Easy).
- 137.Single Number II / 260.III (Medium).
- 268.Missing Number (XOR solution).

Practice: Medium & Hard

- Problems combining XOR with subset or DP logic.
- Bitmask-based subset enumeration with XOR.

Code Snippet (Java – single unique)

```java
int singleNumber(int[] nums) {
    int x = 0;
    for (int v : nums) {
        x ^= v;
    }
    return x;
}
```

References – Bitwise XOR

- Bitwise XOR (eXclusive OR) – intuitive explanation and use-cases (Interview Cake) [web:179]
- Understanding Bitwise XOR: How It Works with Clear Examples [web:181]
- Coding Patterns: Bitwise XOR – common LeetCode-style patterns (single number, missing number, etc.) [web:184]
- Pattern 12: Bitwise XOR – Several Coding Patterns for Interviews (GitHub notes) [web:182]
- Bit Manipulation Coding Questions for Interviews – XOR-heavy problem list (GeeksforGeeks) [web:175]
- Top 25 Bitwise Operators Interview Questions and Answers [web:177]
- Bitwise AND, OR, XOR Explained Simply | Master Bit Manipulation (YouTube) [web:178]

- XOR Magic: Missing Number (LeetCode 268) in O(1) Space – explanation video [web:180]
- Missing Number (LeetCode 268) – XOR solution with visuals (YouTube) [web:183]
- What No One Tells You About Bitwise XOR and Interview Performance – article [web:176]

PHASE 3: HEAP & STACK

TOP K ELEMENTS

This pattern uses a heap (usually a min-heap of size K) to efficiently track the K largest/smallest or K most frequent elements from a stream or array.[^1][^2] **Complexity:** O(N log K) Time | O(K) Space.[^1][^3]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for "top K", "K largest/smallest", or "K most frequent elements".
2. Identify	1–2 min	If problem size is large and only K results are needed, think Top K Elements.
3. Plan	4–6 min	Choose heap type (min-heap for K largest, max-heap for K smallest, or frequency+heap).
4. Code	8–10 min	Build heap from first K items; for each new item, compare with heap top and update if better.
5. Test	3–5 min	Test K=1, K=N, duplicates, and negative/large values.
6. Review	2–3 min	Confirm heap never grows beyond K; verify O(N log K).

Keywords: top K, K largest, K smallest, K most frequent, priority queue **Key phrase:** "only need K best elements" **30-second detection:** "top K … in large array/stream" → Top K Elements.[^4][^1]

How to Solve (3 Steps)

1. Initialize a min-heap (for K largest) or max-heap (for K smallest) of capacity K.[^1]
2. Add the first K elements into the heap; for each remaining element, if it is "better" than the root, pop the root and push the new element.[^3][^1]
3. After processing all elements, the heap contains the desired K elements, which can be extracted in any order.

Cheat code: "keep a min-heap of the top K best; drop the worst whenever size exceeds K".[^1] **Memory hack:** imagine a leaderboard with only K slots; any new entry must beat the current worst to stay.

Real-Life Intuition

- Showing **top K trending items** in a feed from millions of candidates.[^4]
- Keeping **K highest scores** in an online game leaderboard.

Detect: "K largest/smallest", "K most frequent", "streaming data but only top K needed". **Visual:** a small bucket of size K where the lowest-ranked item is always at the top.

Practice: Easy & Medium

- 215.Kth Largest Element in an Array (Medium).
- 347.Top K Frequent Elements (Medium).
- 973.K Closest Points to Origin (Medium).

Practice: Medium & Hard

- 295.Find Median from Data Stream (Hard, related to Two Heaps).
- Sliding Window Top K / Kth element variants(239, 480).

Code Snippet (Java – K largest)

```java
int[] kLargest(int[] nums, int k) {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int x : nums) {
        if (minHeap.size() < k) {
            minHeap.offer(x);
        } else if (x > minHeap.peek()) {
            minHeap.poll();
            minHeap.offer(x);
        }
    }
    int[] res = new int[minHeap.size()];
    int i = 0;
    while (!minHeap.isEmpty()) {
        res[i++] = minHeap.poll();
    }
    return res;
}
```

References – Top K Elements (Heap / Priority Queue)

- Pattern 13: Top "K" Elements – Several Coding Patterns for Interviews (GitHub notes) [web:185]

- Coding Patterns: Top K Numbers – when and how to use heaps [web:186]
- Mastering the Top "K" Elements Pattern – with Java code and heap choice table [web:189]
- Top K Elements in 6 Minutes | LeetCode Pattern (YouTube) [web:188]
- 215. Kth Largest Element in an Array – LeetCode heap solution (AlgoMap) [web:190]
- Top K Frequent Elements in an Array – explanation and approaches (GeeksforGeeks) [web:192]
- Find Top K Frequent Elements – 3 approaches with code [web:194]
- Find the Top K elements in O(N log K) time using heaps (Stack Overflow discussion) [web:187]

K-WAY MERGE

This pattern uses a min-heap keyed by the current element of each list to merge K sorted lists/arrays into one sorted sequence.[^6][^7] **Complexity:** O(N log K) Time where N is total elements | O(K) Space.[^6]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for "merge K sorted arrays/lists/streams".
2. Identify	1–2 min	If K sorted sequences must become one sorted output, choose K-Way Merge.
3. Plan	4–6 min	Use a min-heap of size K storing (value, source list index, element index).
4. Code	8–10 min	Push first element of each list to heap; repeatedly pop min and push next from same list.
5. Test	3–5 min	Test empty lists, different lengths, all negatives/duplicates.
6. Review	2–3 min	Confirm each element is pushed/popped once; heap operations are log K.

Keywords: merge K sorted lists, K arrays, external merge **Key phrase:** "K sorted → 1 sorted" **30-second detection:** "merge K sorted lists/arrays" → K-Way Merge.[^7][^6]

How to Solve (3 Steps)

1. Initialize a min-heap and insert the first element of each non-empty list along with identifiers.[^7]

2. While heap not empty, pop the smallest element, append to result, and if its list has more elements, push the next one from that list.[^6]
3. Continue until heap is empty, meaning all elements are merged.

Cheat code: "always take the smallest current head among lists using a min-heap."[^6] **Memory hack:** think of K conveyor belts, each with sorted items, feeding into one output chute controlled by a min-heap.

Real-Life Intuition

- Merging **sorted log files** from multiple servers into a global sorted log.[^6]
- Implementing **external merge sort** where chunks fit in memory but full data does not.

Detect: "K sorted lists", "merge K streams", "multi-way merge". **Visual:** K sorted queues with their heads in a priority queue.

Practice: Easy & Medium

- 23.Merge k Sorted Lists (Hard on LeetCode but core pattern).[^7]
- Small custom K-way merges in interview exercises.

Practice: Medium & Hard

- External sorting / merging large datasets.
- Multi-way merges with additional constraints (e.g., deduplication).

Code Snippet (Java – merge K sorted lists)

```java
ListNode mergeKLists(ListNode[] lists) {
    PriorityQueue<ListNode> pq =
        new PriorityQueue<>((a, b) -> Integer.compare(a.val, b.val));

    for (ListNode node : lists) {
        if (node != null) pq.offer(node);
    }

    ListNode dummy = new ListNode(0), tail = dummy;
    while (!pq.isEmpty()) {
        ListNode node = pq.poll();
        tail.next = node;
        tail = tail.next;
        if (node.next != null) pq.offer(node.next);
    }
    return dummy.next;
}
```

References – K-Way Merge

- Coding Patterns: K-way Merge – core idea with LeetCode-style examples [web:195]
- Pattern 14: K-way Merge – Several Coding Patterns for Interviews (GitHub notes) [web:197]
- K-way Merge Algorithm – theory, heaps, tournament trees, complexity (Wikipedia) [web:196]
- Why K-Way Merge is the Algorithm Pattern You Need to Know – use-cases and intuition [web:198]
- Merge K Sorted Arrays – min-heap solution and complexity (GeeksforGeeks) [web:199]
- Introduction to K-way Merge Pattern – DesignGurus / Grokking [web:202]
- Merge K Sorted Lists – LeetCode 23 video solution (NeetCode) [web:200]
- Merging K Sorted Lists using Priority Queue – explanation and O(n log k) analysis (Stack Overflow) [web:204]
- Merging K Sorted Arrays – Priority Queue vs traditional merge discussion [web:201]

TWO HEAPS

This pattern uses a max-heap for the lower half of numbers and a min-heap for the upper half to maintain balance and quickly query medians or similar statistics.[^8] **Complexity:** O(log N) Time per insertion | O(N) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “running median”, “online median”, or dynamic data stream.
2. Identify	1–2 min	If asked to maintain median or balance low/high halves, choose Two Heaps.
3. Plan	4–6 min	Use max-heap for lower half, min-heap for upper half; keep sizes balanced.
4. Code	8–10 min	Insert into one heap, then move top to the other if needed; rebalance size difference 1.
5. Test	3–5 min	Test with increasing/decreasing sequences, duplicates, odd/even counts.
6. Review	2–3 min	Verify median logic: if sizes equal, average of tops; else top of larger heap.

Keywords: median, online, streaming numbers, two heaps **Key phrase:** “running/online median” **30-second detection:** “data stream + median at each step” → Two Heaps.[^8]

How to Solve (3 Steps)

1. Insert new number into max-heap (lower half) if it is smaller than or equal to current median; otherwise into min-heap (upper half).
2. Rebalance: if one heap has more than one extra element, move the root to the other heap.
3. Read median as either top of larger heap or average of both tops when sizes equal.

Cheat code: “lower half → max-heap, upper half → min-heap; keep sizes within 1.” **Memory hack:** imagine two piles on each side of the median; they stay roughly the same height.

Real-Life Intuition

- Computing **live median latency** or response time from a service’s request stream.
- Maintaining **balanced partitions** of users by some score.

Detect: “stream of numbers + need median quickly after each insertion”. **Visual:** two heaps back-to-back around the median.

Practice: Easy & Medium

- 295.Find Median from Data Stream (Hard core problem but main example).
- 295.Problems that ask for efficient median maintenance.

Practice: Medium & Hard

- 480.Sliding window median (combines heaps + removal support).
- Quantile approximations with heap-based structures.

Code Snippet (Java – core structure)

```java
class MedianFinder {
    private PriorityQueue<Integer> low;  // max-heap
    private PriorityQueue<Integer> high; // min-heap

    public MedianFinder() {
        low = new PriorityQueue<>(Collections.reverseOrder());
        high = new PriorityQueue<>();
    }

    public void addNum(int num) {
        if (low.isEmpty() || num <= low.peek()) {
            low.offer(num);
        } else {
            high.offer(num);
        }

        // rebalance
        if (low.size() > high.size() + 1) {
            high.offer(low.poll());
        } else if (high.size() > low.size()) {
            low.offer(high.poll());
        }
    }

    public double findMedian() {
        if (low.size() == high.size()) {
            return (low.peek() + high.peek()) / 2.0;
        }
        return low.peek();
    }
}
```

References – Two Heaps (Median / Balancing)

- Coding Patterns: Two Heaps – median of a number stream and variations [web:208]
- Find Median from Running Data Stream – two-heap approach (GeeksforGeeks) [web:205]
- 295. Find Median from Data Stream – in-depth heap explanation (AlgoMonster) [web:209]
- Two Heaps (Median of Stream) Visualizer – interactive min/max-heap balancing [web:206]
- Find Median from Data Stream | Two Heap | Coding Interview (YouTube) [web:207]
- Find Median from Data Stream – LeetCode 295 video walkthrough (Geekific) [web:212]
- 295. Find Median from Data Stream – LeetCode Discuss explanation [web:213]
- 480. Sliding Window Median – detailed two-heaps + lazy deletion strategy (DesignGurus) [web:211]
- Two Heap | LeetCode 480 Sliding Window Median – coding pattern video [web:214]

MONOTONIC STACK

This pattern uses a stack that is strictly increasing or decreasing to efficiently find next/previous greater or smaller elements and solve span problems.[^9][^10]
Complexity: O(N) Time | O(N) Space.[^10]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “next greater/smaller element”, “span”, or “first bigger to left/right”.
2. Identify	1–2 min	If scanning linearly while needing next/previous greater/smaller, think Monotonic Stack.
3. Plan	4–6 min	Decide increasing vs decreasing stack; plan which index/value to store.
4. Code	8–10 min	Traverse array; while top of stack breaks monotonic condition, pop; use stack top as answer.
5. Test	3–5 min	Test strictly increasing, strictly decreasing, random, and circular variants.
6. Review	2–3 min	Verify each index is pushed and popped at most once; total O(N).

Keywords: next greater element, previous smaller, span, histogram **Key phrase:** “next greater/smaller to left/right” **30-second detection:** “next/previous greater/smaller element” → Monotonic Stack.[^10][^9]

How to Solve (3 Steps)

1. Initialize an empty stack to store indices (or values) and an answer array.[^10]
2. Traverse the array; while stack is not empty and current value violates the stack’s monotonic property, pop and fill answer for that index.[^9][^10]
3. Push current index onto stack and continue; remaining indices get default values (e.g., -1 for no greater).

Cheat code: “each element kicks out all weaker elements from the stack in one pass.” **Memory hack:** think of a skyline; each taller building hides all shorter ones behind it when viewed from one side.

Real-Life Intuition

- Calculating **stock span** where each day’s price looks left until a higher price appears.[^10]
- Finding the next warmer/cooler day in a **temperature** series.

Detect: "for each element, find next/previous element with a larger/smaller value", "largest rectangle in histogram". **Visual:** a stack of bars where shorter ones are popped once a taller bar arrives.

Practice: Easy & Medium

- 496.Next Greater Element I / 503.II (Medium).[^9][^10]
- 739.Daily Temperatures (Medium).
- 901.Online Stock Span (Medium).

Practice: Medium & Hard

- 84.Largest Rectangle in Histogram (Hard).
- 42.Trapping Rain Water (can also use stack-based solution).

Code Snippet (Java – next greater element)

```java
int[] nextGreaterElements(int[] nums) {
    int n = nums.length;
    int[] res = new int[n];
    Arrays.fill(res, -1);
    Deque<Integer> stack = new ArrayDeque<>(); // stores indices

    for (int i = 0; i < n; i++) {
        while (!stack.isEmpty() && nums[i] > nums[stack.peek()]) {
            int idx = stack.pop();
            res[idx] = nums[i];
        }
        stack.push(i);
    }
    return res;
}
```

References – Monotonic Stack

- Introduction to Monotonic Stack – definition, intuition, and basic templates (GeeksforGeeks) [web:215]
- Introduction to Monotonic Stack – Grokking-style explanation and templates (DesignGurus) [web:218]
- Monotonic Stack / Deque Intro – core ideas and common problems (AlgoMonster) [web:219]
- Monotonic Stack in 6 Minutes | LeetCode Pattern (YouTube) [web:216]
- Monotonic Stack Data Structure Explained – pattern, examples like Next Greater Element & Daily Temperatures (YouTube) [web:217]
- Guide for Solving Problems Based on Monotonic Stack – LeetCode-focused templates (LinkedIn article) [web:220]
- Monotonic Stacks and Queues – with LeetCode examples (blog) [web:223]

- What is Monotonic Stack – a practical guide for 2025 (problems, patterns, and pitfalls) [web:222]
- Next Greater Element I – monotonic stack solution explanation (AlgoMonster) [web:224]
- Next Greater Element II – circular array monotonic stack pattern (AlgoMonster) [web:221]

PHASE 4: TREES & TRAVERSAL

TREES (Level Order Traversal)

This technique uses a queue to traverse a tree level by level, visiting all nodes at depth 0, then depth 1, and so on.[^1][^2] **Complexity:** O(N) Time | O(W) Space, where W is the maximum width of the tree.[^1][^3]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “level by level”, “level order”, or “nodes at each depth”.
2. Identify	1–2 min	If problem groups nodes by level or asks for min/max depth using BFS, choose Level Order.
3. Plan	4–6 min	Use a queue; push root; process nodes in FIFO order, adding children.
4. Code	8–10 min	While queue not empty, pop size times per level; collect node values and push children.
5. Test	3–5 min	Test empty tree, single node, skewed tree, and full tree.
6. Review	2–3 min	Confirm each node enqueued/dequeued once; check O(N) time.

Keywords: level order, breadth-first, level by level, depth **Key phrase:** “print nodes level by level” **30-second detection:** “tree + level order / breadth-first per level” → Level Order Traversal.[^3][^1]

How to Solve (3 Steps)

1. Initialize a queue and push the root if not null.[^1]
2. While the queue is not empty, record the current `levelSize`, then pop that many nodes, processing their values and enqueuing their children.[^3]
3. Store each level’s values (or perform level-based logic) before moving to the next.

Cheat code: "queue + while not empty + levelSize loop for each depth." **Memory hack:** imagine scanning each floor of a building before going to the next.

Real-Life Intuition

- Processing **organizational hierarchies** one level at a time (CEO → managers → employees).
- Broadcasting a **message** outward from a root node layer by layer.

Detect: "each level separately", "minimum depth to a leaf", "right/left side view by level". **Visual:** concentric circles outward from the root.

Practice: Easy & Medium

- 102.Binary Tree Level Order Traversal (Medium).[^2][^1]
- 111.Minimum Depth of Binary Tree (Easy).
- 199.Binary Tree Right Side View (Medium).

Practice: Medium & Hard

- 103.Binary Tree Zigzag Level Order Traversal (Medium).
- 116.Connect Next Right Pointers in Each Node.

Code Snippet (Java)

```java
List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if (root == null) return res;

    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    while (!q.isEmpty()) {
        int size = q.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = q.poll();
            level.add(node.val);
            if (node.left != null) q.offer(node.left);
            if (node.right != null) q.offer(node.right);
        }
        res.add(level);
    }
    return res;
}
```

References – Trees (Level Order Traversal / BFS)

- Level Order Traversal (Breadth First Search) of Binary Tree – queue-based BFS (GeeksforGeeks) [web:225]
- Level Order Traversal (BFS Traversal) of Binary Tree – intuition and step-by-step algorithm (EnjoyAlgorithms) [web:227]
- Level-order Traversal of Binary Tree – iterative and recursive BFS implementations (Baeldung) [web:228]
- Level Order Traversal of Binary Tree – multiple approaches with examples (InterviewBit) [web:232]
- 102. Binary Tree Level Order Traversal – in-depth BFS explanation (AlgoMonster) [web:229]
- LeetCode 102. Binary Tree Level Order Traversal – problem-focused write-up [web:230]
- Binary Tree Level Order Traversal – LeetCode 102 video solution (NeetCode) [web:226]
- 103. Binary Tree Zigzag Level Order Traversal – BFS with direction toggling (AlgoMonster) [web:231]
- Binary Tree Zigzag Level Order Traversal – LeetCode problem [web:234]
- 107. Binary Tree Level Order Traversal II – bottom-up level order (AlgoMonster) [web:233]

BFS (Grid Traversal)

This technique applies breadth-first search on a 2D grid/graph to find shortest paths, reachable regions, or minimum steps in an unweighted graph.[^4][^5] **Complexity:** O(R · C) or O(V + E) Time | O(V) Space.[^5]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for grids/mazes and “minimum steps” or “shortest path” with equal weights.
2. Identify	1–2 min	If graph is unweighted and shortest distance is needed, choose BFS.
3. Plan	4–6 min	Use queue + direction vectors; track visited or distance matrix.
4. Code	8–10 min	Enqueue start cell(s); at each step, expand to valid neighbors and mark visited.
5. Test	3–5 min	Test blocked start, unreachable target, single-cell grid.
6. Review	2–3 min	Confirm each cell processed at most once; distances increase by 1 per layer.

Keywords: grid, maze, shortest path, minimum steps, BFS **Key phrase:** "unweighted grid + shortest path / min steps" **30-second detection:** "grid + BFS + shortest path" → BFS Grid Traversal.[^5][^4]

How to Solve (3 Steps)

1. Initialize a queue with the start cell(s) and set distance/start depth.[^4][^5]
2. While queue not empty, pop a cell, explore its valid (in-bounds, not blocked, not visited) neighbors, mark visited, and push them with distance+1.[^4]
3. Stop when target is reached (for shortest path) or when queue empties (for reachability/region size).

Cheat code: "unweighted → BFS levels are distances; each ring is +1 step from source." **Memory hack:** waves spreading out from a starting point on the grid.

Real-Life Intuition

- Finding the **shortest route** in a maze or tile-based game.
- Calculating how quickly a **signal** or infection spreads over a uniform network.

Detect: "grid/graph + minimum steps from start to goal" with no edge weights. **Visual:** rings expanding from the start until they hit the target.

Practice: Easy & Medium

- 994.Rotting Oranges (Medium).
- 1091.Shortest Path in Binary Matrix (Medium).
- 200.Number of Islands (can be BFS or DFS).

Practice: Medium & Hard

- 286.Walls and Gates (Medium).
- Multi-source BFS problems (multiple starts).

Code Snippet (Java)

```java
int shortestPath(char[][] grid, int sr, int sc, int tr, int tc) {
    int rows = grid.length, cols = grid[^0].length;
    int[][] dist = new int[rows][cols];
    for (int[] d : dist) Arrays.fill(d, -1);

    int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
    Queue<int[]> q = new LinkedList<>();
    q.offer(new int[]{sr, sc});
    dist[sr][sc] = 0;

    while (!q.isEmpty()) {
        int[] cur = q.poll();
        int r = cur[^0], c = cur[^1];
        if (r == tr && c == tc) return dist[r][c];
        for (int[] d : dirs) {
            int nr = r + d[^0], nc = c + d[^1];
            if (nr >= 0 && nr < rows && nc >= 0 && nc < cols &&
                grid[nr][nc] != '#' && dist[nr][nc] == -1) {
                dist[nr][nc] = dist[r][c] + 1;
                q.offer(new int[]{nr, nc});
            }
        }
    }
    return -1; // unreachable
}
```

References – BFS (Grid Traversal)

- Breadth First Traversal (BFS) on a 2D Array – grid BFS template (GeeksforGeeks) [web:236]
- Breadth First Search or BFS for a Graph – general BFS idea (GeeksforGeeks) [web:235]
- Breadth First Search – grid shortest path (YouTube, William Fiset) [web:237]
- BFS Grid Python – basic grid BFS implementation (YouTube) [web:238]
- 1091. Shortest Path in Binary Matrix – BFS with 8-directional moves (AlgoMonster) [web:72]
- Shortest Path in Binary Matrix – BFS with 8-directional vector pattern (YouTube) [web:242]
- Multi-Source BFS for Grids – Distance to Nearest 1 (LinkedIn article) [web:243]
- [Multi-Source BFS on Grid – "As Far from Land as Possible", LeetCode 1162 (YouTube)]

DFS (Recursive Pathfinding)

This technique explores paths deeply using recursion or an explicit stack, useful for exploring all paths, components, or performing tree/graph traversals.[^6][^7]
Complexity: O(V + E) Time | O(H) Space for recursion depth.[^7]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “all paths”, “connected components”, “explore every possibility”.
2. Identify	1–2 min	If depth-first exploration or full traversal is needed, choose DFS.
3. Plan	4–6 min	Decide recursive vs explicit stack; define visited and recursion prototype.
4. Code	8–10 min	From each node/cell, recursively visit unvisited neighbors/children.
5. Test	3–5 min	Test graphs with cycles, trees, and edge cases (no edges, single node).
6. Review	2–3 min	Confirm visited is set correctly; avoid infinite recursion on cycles.

Keywords: depth-first, recursion, all paths, components **Key phrase:** “explore as far as possible before backtracking” **30-second detection:** “need all paths/regions, not just shortest” → DFS.[^7][^6]

How to Solve (3 Steps)

1. From a start node, mark it visited and process it.[^7]
2. For each neighbor/child not yet visited, recursively call DFS on it.
3. Optionally track current path and backtrack when returning from recursion.

Cheat code: “visit node, mark visited, DFS to each unvisited neighbor; this is recursive template.” **Memory hack:** imagine walking a maze always taking the next unexplored corridor until hitting a dead end, then backtracking.

Real-Life Intuition

- Enumerating **all paths** in a small graph or maze (not just the shortest).
- Finding **connected components** or regions in networks and images.

Detect: “all possible solutions”, “count components”, “traverse entire graph/tree”. **Visual:** a branching tree where you go to the bottom of each branch before coming back up.

Practice: Easy & Medium

- 200.Number of Islands (DFS version).
- 695.Max Area of Island.
- 112.Path Sum / 113.Path Sum II in trees.

Practice: Medium & Hard

- 79.Word Search/ 212.Word Search (board DFS).

- Articulation points / bridges in graphs.

Code Snippet (Java – DFS on grid)

```
void dfs(char[][] grid, int r, int c, boolean[][] vis) {
    int rows = grid.length, cols = grid[^0].length;
    if (r < 0 || r >= rows || c < 0 || c >= cols) return;
    if (grid[r][c] == '0' || vis[r][c]) return;

    vis[r][c] = true;
    dfs(grid, r + 1, c, vis);
    dfs(grid, r - 1, c, vis);
    dfs(grid, r, c + 1, vis);
    dfs(grid, r, c - 1, vis);
}
```

References – DFS (Recursive Pathfinding)

- Depth First Search or DFS for a Graph – recursive DFS basics (GeeksforGeeks) [web:245]
- Depth First Search (DFS) – tutorial with recursion template (takeUforward) [web:253]
- Generic Template for Depth First Search (DFS) on a Graph – recursion + visited set (DSAGuide) [web:250]
- Depth First Search in Matrix using Recursion – path existence and number of islands (blog) [web:251]
- Recursive Depth-First Search (DFS) for Graph Exploration – video guide [web:244]
- Creating a Recursive Path Finding Algorithm to Find All Paths Between Two Nodes (Stack Overflow) [web:246]
- DFS Path Finding with Obstacles – grid DFS discussion (Stack Overflow) [web:247]
- DFS + Backtracking vs plain DFS – conceptual differences (Stack Overflow / LeetCode discussion) [web:249]
- DFS + Backtracking clarifications – revisiting states across calls (Reddit) [web:252]

BACKTRACKING

This technique systematically builds candidates and abandons a partial solution as soon as it violates constraints, exploring the solution space via DFS plus undoing choices.[^6][^7] **Complexity:** Exponential in worst case | O(H) Space for recursion/choices.[^7]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “generate all”, “combinations/permutations”, or constraint puzzles.
2. Identify	1–2 min	If problem says “all solutions” with pruning based on constraints, choose Backtracking.
3. Plan	4–6 min	Define state, choices at each step, and constraint checks; design recursion signature.
4. Code	8–10 min	At each step, choose an option, recurse, then undo (backtrack) before next option.
5. Test	3–5 min	Test small N to verify all unique solutions; check pruning logic.
6. Review	2–3 min	Confirm every choice is reverted; ensure no missing or duplicate solutions.

Keywords: generate all, subsets, permutations, combinations, N-queens, Sudoku **Key phrase:** “all valid configurations under constraints” **30-second detection:** “all solutions + constraints + DFS with undo” → Backtracking.[^6][^7]

How to Solve (3 Steps)

1. Define a recursive function that takes current state (e.g., index, partial solution) and explores choices.[^7]
2. For each valid choice: apply it, recurse to next level; after returning, undo the choice (backtrack).
3. When reaching a complete valid state, record the solution.

Cheat code: “choose → recurse → un-choose” pattern in code. **Memory hack:** like filling slots one by one, erasing and re-writing when a path fails.

Real-Life Intuition

- Solving **Sudoku**, placing numbers that obey row/column/box rules and backtracking when stuck.
- Placing **N queens** on a chessboard so none attack each other.

Detect: “all permutations/combinations/subsets”, “puzzle with strict constraints”, “count all ways”. **Visual:** a decision tree where you walk down, and climb back up when a branch fails.

Practice: Easy & Medium

- 78.Subsets / 90.Subsets II.
- 46.Permutations / 47.Permutations II.
- 77.Combinations / 39.Combination Sum.

Practice: Medium & Hard

- 51.N-Queens (Hard).
- 37.Sudoku Solver (Hard).

Code Snippet (Java – subsets)

```java
List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    backtrack(nums, 0, new ArrayList<>(), res);
    return res;
}

void backtrack(int[] nums, int start, List<Integer> cur, List<List<Integer>> res) {
    res.add(new ArrayList<>(cur));
    for (int i = start; i < nums.length; i++) {
        cur.add(nums[i]);              // choose
        backtrack(nums, i + 1, cur, res); // recurse
        cur.remove(cur.size() - 1);    // un-choose
    }
}
```

References – Backtracking

- Backtracking Algorithm – definition, template, and classic problems (GeeksforGeeks) [web:260]
- Top 20 Backtracking Algorithm Interview Questions – curated practice set (GeeksforGeeks) [web:254]
- Backtracking – topic track with problems (InterviewBit) [web:256]
- Backtracking Overview – patterns, templates, and common pitfalls (Hello Interview) [web:257]
- Mastering Backtracking – A Comprehensive Guide for Coding Interviews (Algocademy) [web:258]
- Backtracking Algorithms Interview Questions and Answers – DevInterview notes [web:255]
- Backtracking to Solve Subset / Permutation / Combination (labuladong) [web:259]
- Permutation Backtracking – detailed example in Hello Algo book [web:261]
- Difference Between Backtracking and DFS – conceptual clarification (Stack Overflow) [web:249]
- DFS + Backtracking? – revisiting states in recursion vs simple DFS (Reddit) [web:252]

PHASE 5: GRAPHS

UNION FIND (Disjoint Set)

This technique maintains a collection of disjoint sets and supports efficient operations to find which set an element belongs to and to union two sets, using path compression and union by rank/size.[^1][^2] **Complexity:** Amortized ~O((N)) per operation (almost constant) | O(N) Space.[^1][^3]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for "connected components", "friend groups", "are these connected?", or Kruskal MST.
2. Identify	1–2 min	If operations are repeated `union(x, y)` / `connected(x, y)`, choose Union Find.
3. Plan	4–6 min	Represent each element with a parent array and optional rank/size array.
4. Code	8–10 min	Implement `find(x)` with path compression and `union(x, y)` by rank/size.
5. Test	3–5 min	Test unions forming chains, stars, and repeated finds; check component counts.
6. Review	2–3 min	Confirm trees flatten over time; verify near O(1) behavior in loops.

Keywords: disjoint set, union-find, connected, components, Kruskal **Key phrase:** "dynamic connectivity queries" **30-second detection:** "many union/connected queries on elements" → Union Find.[^2][^1]

How to Solve (3 Steps)

1. Initialize each element as its own parent; optionally set rank/size to 1.[^3][^2]
2. Implement `find(x)` that follows parent pointers to the root and applies path compression by making nodes point directly to the root.[^4][^1]
3. Implement `union(a, b)` by finding roots and attaching the smaller-rank/size tree under the larger to keep trees shallow.[^5][^1]

Cheat code: "parents + path compression + union by rank; treat each component as a tree with a root."[^2] **Memory hack:** think of friend circles gradually merging; each circle has a leader (root).

Real-Life Intuition

- Maintaining **friend groups** as new friendships (edges) appear and asking if two users are in the same group.

- Building **minimum spanning trees** with Kruskal's algorithm while avoiding cycles.[^2]

Detect: "are these nodes in the same group?", "how many components?", "union and find operations". **Visual:** small trees of nodes whose roots get closer as paths are compressed.[^2]

Practice: Easy & Medium

- 323.Number of Connected Components in an Undirected Graph.
- 684.Redundant Connection (detecting an extra edge forming a cycle).
- 721.Accounts Merge (group emails/users).

Practice: Medium & Hard

- Kruskal's MST problems.
- Connectivity with dynamic edge additions.

Code Snippet (Java)

```java
class DSU {
    int[] parent, rank;

    DSU(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]); // path compression
        return parent[x];
    }

    void union(int a, int b) {
        int ra = find(a), rb = find(b);
        if (ra == rb) return;
        if (rank[ra] < rank[rb]) {
            parent[ra] = rb;
        } else if (rank[ra] > rank[rb]) {
            parent[rb] = ra;
        } else {
            parent[rb] = ra;
            rank[ra]++;
        }
    }
}
```

References – Union Find (Disjoint Set)

- Introduction to Disjoint Set (Union-Find Data Structure) – basics, API, and examples (GeeksforGeeks) [web:262]
- Union by Rank and Path Compression in Union-Find – optimization details (GeeksforGeeks) [web:271]
- Disjoint Set Union – theory, proofs, and competitive programming usage (CP-Algorithms) [web:263]
- Disjoint Set (Union Find Algorithm) – tutorial with examples (Scaler) [web:266]
- Disjoint Set Union / Union Find Algorithm Simplified – path compression + rank (Dev.to) [web:269]
- LeetCode Pattern 12 – Union Find, with templates and problems [web:267]
- [Union Find in 5 Minutes – quick visual explanation (YouTube)](https://www.youtube.com/watch?v=ayV

TOPOLOGICAL SORT

This technique orders vertices of a directed acyclic graph (DAG) so that every directed edge goes from an earlier node to a later node, often using Kahn's algorithm (BFS on indegrees) or DFS.[^6][^7] **Complexity:** O(V + E) Time | O(V) Space.[^6][^7]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for "prerequisites", "order of tasks/courses", or DAG requirement.
2. Identify	1–2 min	If the graph is directed and acyclic and you need a valid ordering, choose Topological Sort.
3. Plan	4–6 min	Choose Kahn's (indegree + queue) or DFS post-order; build adjacency list and indegree.
4. Code	8–10 min	Push all 0-indegree nodes into queue; repeatedly pop, append to order, and decrement neighbors' indegrees.
5. Test	3–5 min	Test simple chains, multiple starts, and graphs with cycles (detect failure).
6. Review	2–3 min	Ensure result size is V; if not, a cycle exists.

Keywords: DAG, prerequisites, course schedule, ordering **Key phrase:** "tasks with prerequisites in a DAG" **30-second detection:** "directed + need valid order respecting dependencies" $\rightarrow$ Topological Sort.[^7][^6]

How to Solve (3 Steps)

1. Compute indegree (incoming edge count) for every node.[^6]
2. Push all nodes with indegree 0 into a queue, then repeatedly pop a node, add it to the result, and reduce indegree of its neighbors, pushing those that become 0.[^7]
3. After processing, if the result list has V nodes, it is a valid topological order; otherwise, a cycle exists.

Cheat code: "queue of indegree-0 nodes; pull them out and 'remove' edges until done."[^6] **Memory hack:** think of taking courses where you can only enroll once all prerequisites are cleared.

Real-Life Intuition

- Scheduling **courses** or **tasks** with prerequisites in a legal order.
- Determining **build or deployment order** in software projects.

Detect: "prerequisite graph", "can we complete all tasks?", "valid ordering of tasks". **Visual:** a layered diagram where arrows always point from earlier to later tasks.

Practice: Easy & Medium

- 207.Course Schedule (can detect cycle).
- 210.Course Schedule II (return one valid order).
- 1203.Sort Items by Groups Respecting Dependencies (harder variant).

Practice: Medium & Hard

- 269.Alien Dictionary (derive ordering of characters).
- Build order in complex dependency graphs.

Code Snippet (Java – Kahn's algorithm)

```java
int[] topoSort(int n, List<List<Integer>> adj) {
    int[] indeg = new int[n];
    for (int u = 0; u < n; u++) {
        for (int v : adj.get(u)) indeg[v]++;
    }

    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        if (indeg[i] == 0) q.offer(i);
    }

    int[] order = new int[n];
    int idx = 0;
    while (!q.isEmpty()) {
        int u = q.poll();
        order[idx++] = u;
        for (int v : adj.get(u)) {
            if (--indeg[v] == 0) q.offer(v);
        }
    }
    return idx == n ? order : new int[^0]; // empty if cycle
}
```

References – Topological Sort

- Topological Sorting – definitions, DFS-based and Kahn's algorithm (CP-Algorithms) [web:279]
- Topological Sorting (Wikipedia) – theory, properties, and applications [web:274]
- Kahn's Algorithm – indegree-based topological sorting (GeeksforGeeks) [web:273]
- Kahn's Algorithm vs DFS Approach – comparative analysis and when to use which (GeeksforGeeks) [web:272]
- Topological Sort Kahn's Algorithm – BFS or DFS? conceptual clarification (Stack Overflow) [web:275]
- Topological Sort – Distilled LeetCode pattern, with templates and problems [web:277]
- Course Schedule I and II – prerequisite tasks via topological sort (takeUforward) [web:281]
- Topological Sorting to Determine Course Order – course schedule example with BFS (Dev.to) [web:278]
- Kahn's Algorithm | Topological Sort Algorithm | BFS (YouTube) [web:276]
- [What No One Tells You About Topological Sorting, LeetCode, and Interview Performance (article)](https://www.vervecopilot.com/interview-questions/what-no-one-tells-you-about-topological-sorting-leetcode-and-interview-performance

---

GRAPH ALGORITHMS (Dijkstra's)

This pattern applies Dijkstra's algorithm with a priority queue to find shortest paths from a single source in graphs with non-negative edge weights.[^8][^9]
Complexity: O((V + E) log V) Time with binary heap | O(V + E) Space.[^8][^9]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for weighted graph, non-negative weights, and “shortest path / minimum cost”.
2. Identify	1–2 min	If edge weights are non-negative and single-source shortest paths needed, choose Dijkstra’s.
3. Plan	4–6 min	Use adjacency list, distance array initialized to ∞, and min-priority queue of (dist, node).
4. Code	8–10 min	Push `(0, source)`; while heap not empty, pop smallest dist, relax edges, and push improved neighbors.
5. Test	3–5 min	Test small graphs, unreachable nodes, and varying weights (including 0).
6. Review	2–3 min	Confirm negative edges are not present; verify distances never increase.

Keywords: weighted graph, non-negative weights, shortest path, minimum cost **Key phrase:** “non-negative edges + single-source shortest path” **30-second detection:** “weighted graph + min distance/cost + no negative edges” → Dijkstra.[^9][^8]

How to Solve (3 Steps)

1. Initialize `dist[]` with infinity, except `dist[src] = 0`; push `(0, src)` into a min-priority queue.[^8][^9]
2. While queue not empty, pop `(d, u)`; if `d` is greater than current `dist[u]`, skip; otherwise relax all edges `(u, v, w)` and if `dist[u] + w < dist[v]`, update and push `(dist[v], v)`.
3. After processing, `dist[v]` holds the shortest distance from source to every reachable node.

Cheat code: “greedy frontier: always expand the closest unprocessed node using a min-heap.”[^8] **Memory hack:** think of filling a map with the cheapest known cost, expanding outward from the cheapest frontier.

Real-Life Intuition

- Finding **shortest driving routes** on a road map with travel times or distances.
- Computing **cheapest costs** in a network of flights, cables, or data links.

Detect: “graph + weighted + non-negative + minimize distance/cost from one node”. **Visual:** a growing cloud of finalized shortest paths starting from the source.

Practice: Easy & Medium

- 743.Network Delay Time (Medium).
- 787.Cheapest Flights Within K Stops (modified Dijkstra/BFS).
- 1631.Path With Minimum Effort (grid with weights).

Practice: Medium & Hard

- Shortest Path in a Weighted Grid / Maze.
- Multi-source and constrained variations (like K stops, extra state).

Code Snippet (Java)

```java
int[] dijkstra(int n, List<List<int[]>> adj, int src) {
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;

    PriorityQueue<int[]> pq =
        new PriorityQueue<>((a, b) -> Integer.compare(a[^0], b[^0]));
    pq.offer(new int[]{0, src}); // {dist, node}

    while (!pq.isEmpty()) {
        int[] cur = pq.poll();
        int d = cur[^0], u = cur[^1];
        if (d > dist[u]) continue;

        for (int[] e : adj.get(u)) {
            int v = e[^0], w = e[^1];
            if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.offer(new int[]{dist[v], v});
            }
        }
    }
    return dist;
}
```

References – Graph Algorithms (Dijkstra’s Shortest Path)

- Dijkstra’s Shortest Path Algorithm – explanation, pseudo-code, and complexity (GeeksforGeeks) [web:282]
- Dijkstra’s Shortest Path Algorithm – step-by-step online tutorial (TutorialsPoint) [web:284]
- A Complete Guide to Dijkstra’s Shortest Path Algorithm – with Python code (Codecademy) [web:286]

- Dijkstra's Algorithm – W3Schools DSA: explanation, visuals, and example run-through [web:285]
- Understanding Dijkstra's Algorithm: A Step-by-Step Guide (Dev.to) [web:289]
- Priority Queues in Dijkstra's Shortest Path Algorithm – why we need a min-heap (Rice University) [web:290]
- Priority Queues in Dijkstra – discussion and intuition (Reddit) [web:287]
- Difference Between BFS and Dijkstra's Algorithms – weighted vs unweighted graphs (Techkluster) [web:288]
- Dijkstra in the streets, BFS in the sheets – intuitive proof and BFS connection [web:291]
- Dijkstra's Shortest Path Algorithm Explained with Example (YouTube) [web:283]

PHASE 6: DYNAMIC PROGRAMMING

TAKE / NOT TAKE (0/1 KNAPSACK)

This pattern decides for each item whether to take it once or skip it, using DP over item index and remaining capacity to maximize value.[^1][^2] **Complexity:** O(N · W) Time | O(N · W) or optimized O(W) Space.[^1][^3]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for items with weights/values and a capacity/limit ("each item at most once").
2. Identify	1–2 min	If each item can be picked 0 or 1 time, think 0/1 Knapsack Take/Not Take DP.
3. Plan	4–6 min	Define `dp[i][w]` = best value using first i items and capacity w, or 1D variant.
4. Code	8–10 min	For each item and capacity, set `dp[i][w] = max(skip, take)` if weight fits.
5. Test	3–5 min	Test small N, capacity 0, weight > capacity, and boundary cases.
6. Review	2–3 min	Confirm item is not reused (iterate capacity backward for 1D DP).

Keywords: 0/1, knapsack, capacity, weight, profit/value **Key phrase:** "each item at most once under capacity" **30-second detection:** "capacity + items each usable once" → 0/1 Knapsack.[^4][^1]

How to Solve (3 Steps)

1. Set base: `dp[^0][*] = 0` and `dp[*][^0] = 0` for no items or zero capacity.[^2][^1]
2. For each item i and capacity w, compute: skip = `dp[i-1][w]`; take = `value[i] + dp[i-1][w-weight[i]]` if `weight[i]` `w`; choose max.[^1][^4]
3. Answer is `dp[n][W]` (or `dp[W]` in 1D formulation).

Cheat code: "for each item: value = max(not take, take once if fits)." **Memory hack:** think of building rows of a table where each new row adds the choice of taking a new item.

Real-Life Intuition

- Picking **projects** under a budget where each can be funded at most once.
- Packing a **bag** with limited capacity choosing the most valuable subset of items.

Detect: "max value under capacity", "use item 0 or 1 time", "subset of items". **Visual:** a grid with items on one axis and capacity on the other.

Practice: Easy & Medium

- 0/1 Knapsack basic problem.(416, 494, 1049)
- 416.Subset Sum / Partition Equal Subset Sum.
- 494.Target Sum (variation).

Practice: Medium & Hard

- Problems phrased as "maximize profit under limit, each project once".
- Multi-constraint knapsack variants.

Code Snippet (Java – 1D optimization)

```java
int knapSack(int W, int[] wt, int[] val) {
    int n = wt.length;
    int[] dp = new int[W + 1];

    for (int i = 0; i < n; i++) {
        for (int w = W; w >= wt[i]; w--) {  // backward for 0/1
            dp[w] = Math.max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }
    return dp[W];
}
```

References – Take / Not Take (0/1 Knapsack)

- 0/1 Knapsack Problem – classic DP, recursion, and tabulation (GeeksforGeeks) [web:292]
- The 0/1 Knapsack Problem – explanation, table reconstruction, and included items (W3Schools) [web:295]
- 0/1 Knapsack Pattern – "take or not take" DP on subsequences (blog) [web:300]
- Solution: 0/1 Knapsack – Grokking the DP pattern, include/exclude formulation (DesignGurus) [web:299]
- 0/1 Knapsack – Two Methods: DP Tabulation and Set-based (YouTube) [web:293]
- 0/1 Knapsack | Recursion → Memoization → Tabulation → Space Optimized (takeUforward video) [web:294]
- 0/1 Knapsack | Recursion + Memoisation + Tabulation | DP-11 (DP Is Easy video) [web:296]
- 0/1 Knapsack Problem – LeetCode-style PDF overview (include vs exclude states) [web:298]
- 0/1 Knapsack Problem LeetCode – PDF (problem statement and DP discussion) [web:301]
- 0/1 Knapsack using Least Cost Branch and Bound – advanced variant [web:297]

INFINITE SUPPLY (Unbounded Knapsack)

This pattern is similar to knapsack but each item can be chosen multiple times, using DP where transitions allow staying on the same item after a take.[^5][^6]
Complexity: O(N · W) Time | O(N · W) or O(W) Space.[^5]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for items with weights/values and explicit "unlimited" or "infinite supply".
2. Identify	1–2 min	If each item can be picked any number of times, think Unbounded Knapsack.
3. Plan	4–6 min	Define `dp[i][w]` or 1D `dp[w]` where taking an item allows staying at same i.
4. Code	8–10 min	Loop items; for each capacity, consider `dp[w] = max(dp[w], val[i] + dp[w - wt[i]])`.
5. Test	3–5 min	Test capacity small, single item, and combinations relying on repeats.
6. Review	2–3 min	Confirm capacity loop is forward (allow reuse); verify O(N · W).

Keywords: unbounded, infinite supply, coin change, unlimited uses **Key phrase:** "use items any number of times" **30-second detection:** "coin change / unlimited items under capacity" → Unbounded Knapsack.[^6][^5]

How to Solve (3 Steps)

1. Initialize `dp[^0] = 0` and others to 0 or negative infinity depending on formulation.[^5]
2. For each item i, loop capacity w from `wt[i]` to W (forward), updating `dp[w] = max(dp[w], val[i] + dp[w - wt[i]])` to allow repeated usage.[^6][^5]
3. Answer is `dp[W]` for max value or count, depending on problem definition.

Cheat code: "for unbounded: capacity loop forward so the same item can be reused." **Memory hack:** imagine unlimited coins of each type that you can keep taking.

Real-Life Intuition

- **Coin change** problems, where you can use any coin denomination many times.
- Producing a **product** using raw materials where each resource type is reusable.

Detect: "unlimited coins/items", "ways to make sum", "min coins to reach amount". **Visual:** stacks of coins of each type that never run out.

Practice: Easy & Medium

- 322.Coin Change (ways) and 518.Coin Change II (min coins).
- Unbounded Knapsack standard problem.

Practice: Medium & Hard

- Rod Cutting problem.
- 343.Integer Break (DP formulation).

Code Snippet (Java – unbounded knapsack / coin change max value)

```java
int unboundedKnapsack(int W, int[] wt, int[] val) {
    int n = wt.length;
    int[] dp = new int[W + 1];

    for (int i = 0; i < n; i++) {
        for (int w = wt[i]; w <= W; w++) {  // forward for unbounded
            dp[w] = Math.max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }
    return dp[W];
}
```

References – Infinite Supply (Unbounded Knapsack)

- Unbounded Knapsack – repetition of items allowed (GeeksforGeeks) [web:302]
- Unbounded Knapsack Problem – Hello Algo, with clear DP state and transitions [web:304]
- Unbounded Knapsack Pattern – explanation, transitions, and examples [web:305]
- Knapsack DP – includes unbounded knapsack and coin change section (USACO Guide) [web:309]
- Coin Change 2 – Unbounded Knapsack DP, LeetCode 518 (NeetCode video) [web:303]
- Coin Change Problems – Knapsack Variation, LeetCode 322 & 518 (DP + Infinite Supply) [web:306]
- Unbounded Knapsack / Coin Change with Optimal Solution for Non-standard Coins (Stack Overflow) [web:307]
- Coin Change Program Using DP – Knapsack with Repetitions Allowed (Stack Overflow) [web:310]
- LeetCode 322, 377, 139 – Unbounded Knapsack Problem (blog walk-through) [web:308]
- Unbounded Knapsack Pattern: Maximize Value or Minimize Operations – LinkedIn post with LeetCode mapping [web:311]

LONGEST INCREASING SUBSEQUENCE (LIS)

This pattern finds the longest strictly increasing subsequence in an array using either O(N^2) DP or an O(N log N) binary search approach.[^7][^8] **Complexity:** O(N^2) or O(N log N) Time | O(N) Space.[^7][^8]

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for “longest increasing subsequence/sequence” or chain of increasing values.
2. Identify	1–2 min	If subsequence (not necessarily contiguous) must be strictly increasing, think LIS.
3. Plan	4–6 min	Choose O(N^2) DP or tails array + binary search (patience sorting).
4. Code	8–10 min	For N^2: `dp[i] = 1 + max(dp[j])` for $j < i$ and $a[j] < a[i]$; track max.

Step	Time	Description
5. Test	3–5 min	Test sorted, reverse, duplicates, and random sequences.
6. Review	2–3 min	For N log N approach, verify tails is non-decreasing and updated via lower_bound.

Keywords: LIS, longest increasing subsequence, chain, sequence **Key phrase:** "longest strictly increasing subsequence" **30-second detection:** "increasing subsequence length, not contiguous" → LIS.[^8][^7]

How to Solve (3 Steps, N log N idea)

1. Maintain an array `tails`, where `tails[len]` is the smallest possible tail of an increasing subsequence of length `len+1`.[^7][^8]
2. For each number x, binary search in `tails` for the first element x and replace it with x; if none found, append x.
3. The length of `tails` at the end is the LIS length.

Cheat code: "`tails` holds best tails; replace with smaller tails to keep future options open."[^7] **Memory hack:** think of creating piles in patience sorting where each pile's top is as small as possible.

Real-Life Intuition

- Longest **career progression** or rank improvements over time.
- Longest **increasing trend** in stock prices ignoring non-increasing days.

Detect: "subsequence (skip allowed) + strictly increasing + longest length". **Visual:** stepping up a staircase where you can skip some steps but never go down.

Practice: Easy & Medium

- 300.Longest Increasing Subsequence (Medium).
- 354.Russian Doll Envelopes (Hard, LIS after sorting).
- 646.Longest Chain of Pairs.

Practice: Medium & Hard

- Problems that reduce to LIS after coordinate compression or sorting.

Code Snippet (Java – N log N)

```java
int lengthOfLIS(int[] nums) {
    int[] tails = new int[nums.length];
    int size = 0;
    for (int x : nums) {
        int l = 0, r = size;
        while (l < r) {
            int m = l + (r - l) / 2;
            if (tails[m] < x) l = m + 1;
            else r = m;
        }
        tails[l] = x;
        if (l == size) size++;
    }
    return size;
}
```

References – Longest Increasing Subsequence (LIS)

- Longest Increasing Subsequence – O(N^2) DP and explanation (GeeksforGeeks) [web:313]
- Longest Increasing Subsequence Size – O(N log N) binary search method (GeeksforGeeks) [web:312]
- Longest Increasing Subsequence | Binary Search | DP-43 (takeUforward article) [web:315]
- DP 43. Longest Increasing Subsequence | Binary Search | Intuition (takeUforward video) [web:319]
- Longest Increasing Subsequence N log N | LeetCode 300 (TechDose video) [web:314]
- Patience Sorting – LIS using patience sorting, LeetCode 300 (video) [web:317]
- Longest Increasing Subsequence (LIS) – DP and patience sorting explanation (Logicmojo) [web:318]
- Longest Increasing Subsequence (LIS) – Patience Sorting and Fenwick approaches (CSU083 notes) [web:320]
- Longest Increasing Subsequence O(N log N) – Stack Overflow discussion on correctness [web:316]
- Dynamic Programming – Longest Increasing Subsequence lecture notes (PDF) [web:321]

DP ON GRIDS

This pattern solves path-counting or min-cost problems on a grid using DP over rows and columns.[^9] **Complexity:** O(R · C) Time | O(R · C) or O(C) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for m×n grid with moves (usually right/down) and paths or costs.
2. Identify	1–2 min	If moves are constrained and costs/ways accumulate cell-wise, think DP on Grids.
3. Plan	4–6 min	Define `dp[r][c]` as number of ways or min cost to reach (r,c); set base row/column.
4. Code	8–10 min	Fill DP row by row or column by column using transitions from neighbors.
5. Test	3–5 min	Test 1×n, m×1, obstacles, and boundary conditions.
6. Review	2–3 min	Consider 1D optimization over columns if only previous row is needed.

Keywords: grid, paths, ways, min path sum, obstacles **Key phrase:** "grid + restricted moves + count ways or min cost" **30-second detection:** "m×n grid with right/down only" → DP on Grids.

How to Solve (3 Steps)

1. Initialize base cell (0,0) and first row/column depending on allowed moves and obstacles.
2. For each cell, compute `dp[r][c]` using neighbors (e.g., ways = sum from top and left; cost = value + min(top, left)).
3. Answer is `dp[m-1][n-1]`.

Cheat code: "dp[r][c] depends only on neighbors (top/left) and cell value." **Memory hack:** imagine filling a table such that each cell knows how many ways lead into it.

Real-Life Intuition

- Counting **unique paths** in a grid-like city with blocked streets.
- Finding **minimum-cost** path across a weighted grid.

Detect: "unique paths", "min path sum", "grid with obstacles/right and down moves only". **Visual:** dynamic programming table being filled from top-left to bottom-right.

Practice: Easy & Medium

- 62.Unique Paths / 63.Unique Paths II.
- 64.Minimum Path Sum.
- 174.Dungeon Game (harder variant but same idea + backwards DP).

Practice: Medium & Hard

- Paths with constraints (e.g., maximum moves, health thresholds).

Code Snippet (Java – min path sum)

```java
int minPathSum(int[][] grid) {
    int m = grid.length, n = grid[^0].length;
    int[][] dp = new int[m][n];
    dp[^0][^0] = grid[^0][^0];

    for (int i = 1; i < m; i++) dp[i][^0] = dp[i-1][^0] + grid[i][^0];
    for (int j = 1; j < n; j++) dp[^0][j] = dp[^0][j-1] + grid[^0][j];

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = grid[i][j] + Math.min(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m-1][n-1];
}
```

References – DP on Grids

- Dynamic Programming (DP) on Grids – overview and classic patterns (GeeksforGeeks) [web:322]
- Unique Paths in a Grid – DP example based on LeetCode 62 [web:326]
- Unique Paths – Dynamic Programming, LeetCode 62 (NeetCode video) [web:325]
- DP 8. Grid Unique Paths – "learn everything about DP on grids" (takeUforward video) [web:324]
- 64. Minimum Path Sum – in-depth grid DP explanation (AlgoMonster) [web:328]
- Finding the Minimum Path Sum in a Grid with Dynamic Programming (Dev.to) [web:329]
- Minimum Path Sum in Triangular Grid – variant of grid DP (takeUforward) [web:331]
- DP on Grids – pattern 14 in "20 Patterns to Master Dynamic Programming" (Algomaster) [web:327]
- LeetCode Patterns repo – includes DP and grid-based categories (GitHub) [web:330]
- Shortest Path in Grid with Obstacles – DP/BFS discussion (GeeksforGeeks) [web:323]

DP ON STRINGS

This pattern uses DP over indices of one or two strings to solve problems like edit distance, longest common subsequence, subsequences, and palindromes.[^10] **Complexity:** Typically O(N · M) Time | O(N · M) or O(min(N,M)) Space.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for two strings with alignment/transform questions, or subsequence/palindrome queries.
2. Identify	1–2 min	If recurrence depends on prefixes `s1[0..i]`, `s2[0..j]`, think DP on Strings.
3. Plan	4–6 min	Define `dp[i][j]` meaning (e.g., length/cost) for prefixes; derive recurrence by matching/mismatching chars.
4. Code	8–10 min	Fill DP table with base cases for empty prefixes, then iterate i,j.
5. Test	3–5 min	Test empty strings, equal strings, and small examples.
6. Review	2–3 min	Optimize to 1D if recurrence uses only previous row/column.

Keywords: edit distance, LCS, subsequence, palindrome, string transform **Key phrase:** "cost/length based on prefixes of two strings" **30-second detection:** "2D DP over string indices" → DP on Strings.

How to Solve (3 Steps)

1. Choose meaning of `dp[i][j]` (e.g., min edits to convert first i chars of A to first j of B).[^10]
2. Set base rows/columns for empty prefixes; fill by comparing characters and combining neighboring states (top/left/diagonal).
3. Read answer from `dp[n][m]` or appropriate cell.

Cheat code: "grid where rows = string1 prefix, columns = string2 prefix; recurrence from neighbors." **Memory hack:** think of aligning two strings on a grid and walking from top-left to bottom-right.

Real-Life Intuition

- **Spell checking** and auto-correct via edit distance.
- Comparing **DNA sequences** (LCS, alignment).

Detect: "min edits", "longest common subsequence/substring", "is A a subsequence of B". **Visual:** two strings forming axes of a table.

Practice: Easy & Medium

- 72.Edit Distance (Hard in rating, classic example).
- 1143.Longest Common Subsequence.
- 516.Longest Palindromic Subsequence.

Practice: Medium & Hard

- 115.Distinct Subsequences.
- 10.Regular Expression Matching / 44.Wildcard Matching.

Code Snippet (Java – edit distance)

```java
int minDistance(String a, String b) {
    int n = a.length(), m = b.length();
    int[][] dp = new int[n + 1][m + 1];

    for (int i = 0; i <= n; i++) dp[i][^0] = i;
    for (int j = 0; j <= m; j++) dp[^0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a.charAt(i - 1) == b.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = 1 + Math.min(dp[i - 1][j - 1],
                               Math.min(dp[i - 1][j], dp[i][j - 1]));
            }
        }
    }
    return dp[n][m];
}
```

References – DP on Strings

- String-based Dynamic Programming Problems – categorized patterns (The Algorist) [web:332]
- DP on Strings – LCS, edit distance, palindrome, wildcard matching (theroyakash) [web:335]
- Pattern 8: String DP (LCS / Edit Distance alignment pattern) [web:340]
- Dynamic Programming – Edit Distance and LCS (GeeksforGeeks) [web:337]
- String DP – palindromes, subsequences, and partitions (Bea.AI blog) [web:338]
- Lecture: Memoization / DP on Strings (Harvard CS125 notes, PDF) [web:336]
- DP On Strings – DP Pattern 4 (YouTube lecture) [web:339]

- Interleaving Strings – Dynamic Programming, LeetCode 97 (video) [web:334]
- Dynamic Programming Overview – includes string DP categories (GeeksforGeeks) [web:333]
- DP Practice List – includes many string DP problems (AlgoMonster) [web:341]

DP ON STOCKS

This pattern models stock trading with states (day, holding/not holding, transaction count, cooldown, fee) and uses DP to maximize profit under constraints.[^9] **Complexity:** O(N · K) or O(N) Time | O(K) or O(1) Space depending on constraints.

Study Workflow

Step	Time	Description
1. Read	3–5 min	Look for stock prices over days with rules: max transactions, cooldown, or fee.
2. Identify	1–2 min	If decisions per day are "buy/sell/hold" with constraints, think DP on Stocks.
3. Plan	4–6 min	Define states (e.g., `hold`, `notHold`, maybe `cooldown` or `k` transactions).
4. Code	8–10 min	Write transitions for each day: update states using max of buy/sell/hold actions.
5. Test	3–5 min	Test small arrays, monotonic prices, and edge cases (length 0/1).
6. Review	2–3 min	Check state transitions carefully; ensure constraints (cooldown, fee, K) are enforced.

Keywords: buy/sell, max profit, transactions, cooldown, fee, at most K **Key phrase:** "stock prices over days + constraints on buy/sell" **30-second detection:** "prices array + choose buy/sell days under rules" → DP on Stocks.

How to Solve (3 Steps – simplest version, unlimited transactions)

1. Track two states per day: `hold` (max profit while holding stock) and `notHold` (max profit while not holding).
2. For each price p:
   - `hold = max(hold, notHold - p)` (buy or keep holding)
   - `notHold = max(notHold, hold + p)` (sell or keep not holding)
3. Final answer is `notHold` after last day.

Cheat code: "two running variables: best profit when holding vs not holding; update each day." **Memory hack:** treat it like toggling between two bank accounts: one when holding stock, one when in cash.

Real-Life Intuition

- Choosing **when to trade** to maximize profit given transaction costs or cooldown rules.
- Budgeting **energy or resources** over days where "buy" and "sell" are generic invest/divest decisions.

Detect: "prices[i] each day + maximize profit with rules like cooldown/fee/at most K transactions". **Visual:** time series where at each day you can move between "holding" and "not holding" states.

Practice: Easy & Medium

- 121.Best Time to Buy and Sell Stock (I).
- 122.Best Time to Buy and Sell Stock II (unlimited transactions).
- 714.Best Time to Buy and Sell Stock with Transaction Fee / with Cooldown.

Practice: Medium & Hard

- 123.Best Time to Buy and Sell Stock III (at most 2 transactions).
- 188.Best Time to Buy and Sell Stock IV (at most K transactions).

Code Snippet (Java – unlimited transactions)

```java
int maxProfit(int[] prices) {
    if (prices.length == 0) return 0;
    int hold = -prices[^0];
    int notHold = 0;

    for (int i = 1; i < prices.length; i++) {
        hold = Math.max(hold, notHold - prices[i]);
        notHold = Math.max(notHold, hold + prices[i]);
    }
    return notHold;
}
```

References – DP on Stocks (Buy / Sell)

- 121. Best Time to Buy and Sell Stock – introductory one-transaction problem (AlgoMonster) [web:347]
- Stock Buy and Sell – Multiple Transactions Allowed (Greedy baseline) (GeeksforGeeks) [web:350]

- Best Time to Buy and Sell Stock II – three approaches and DP formulation [web:348]
- Best Time to Buy and Sell Stock – DP on Stocks | DP-35 (takeUforward video) [web:351]
- Best Time to Buy and Sell Stock with Cooldown – LeetCode problem [web:342]
- Stock Buy and Sell with Cooldown – DP states and transitions (GeeksforGeeks) [web:343]
- Best Time to Buy and Sell Stock with Cooldown – detailed write-up [web:345]
- Best Time to Buy and Sell Stock with Cooldown – explanation video [web:344]
- Buy and Sell Stocks with Cooldown – Infinite Transactions Allowed (Pepcoding video) [web:346]
- DP 4. Coin Change / Stock Variants – "DP on Stocks" family overview (video, includes fee and k-transactions links) [web:306]

Key LeetCode variants you can map under this pattern (with separate problem lookup):

- Best Time to Buy and Sell Stock I (single transaction, LC 121)
- Best Time to Buy and Sell Stock II (infinite transactions, LC 122)
- Best Time to Buy and Sell Stock III (at most two transactions, LC 123)
- Best Time to Buy and Sell Stock IV (at most k transactions, LC 188)
- Best Time to Buy and Sell Stock with Cooldown (LC 309)
- Best Time to Buy and Sell Stock with Transaction Fee (LC 714) [web:348]